ImprovMLCQ: A Feature-Enriched Dataset for Advancing Code Smell Detection

Joanne Carneiro
PUC-Rio
Rio de Janeiro, Brazil
joannecarneiro@aise.inf.puc-rio.br

Jessica Ribas
PUC-Rio
Rio de Janeiro, Brazil
jribas@aise.inf.puc-rio.br

Amanda Santana
UFMG
Belo Horizonte, Brazil
amandads@dcc.ufmg.br

Eduardo Figueiredo
UFMG
Belo Horizonte, Brazil
figueiredo@dcc.ufmg.br

Juliana Alves Pereira
PUC-Rio
Rio de Janeiro, Brazil
juliana@inf.puc-rio.br

ABSTRACT

Code smells are indicators of poor design choices in source code which negatively impact software quality. While manual detection of code smells is time-consuming, their automated detection requires high-quality datasets. This work evaluates an improved version of the Madeyski Lewowski Code Quest (MLCQ), called ImprovMLCQ, which incorporates an extensive list of features extracted with four tools: CK, PMD, Organic, and Designite; along with several project characteristics. Our goal is to leverage these features to gain deeper insights into the detection or four code smells (Blob, Data Class, Feature Envy, and Long Method), assessing the effectiveness of different Machine Learning and Deep Learning models, and exploring the impact of feature selection on predictive performance. We evaluate fifteen Machine Learning-based algorithms and four Deep Learning algorithms using ImprovMLCQ, leveraging various feature engineering and selection mechanisms to optimize predictive performance. Our results show that the Extra Trees and Random Forest classifiers achieved the highest F1-Score among the tested algorithms, showing the importance of additional features to improve prediction performance.

KEYWORDS

code smell, software quality, maintainability, machine learning.

1 Introduction

Code smells are symptoms of poor design and implementation choices [10]. Several studies confirmed their negative consequences on the maintenance and evolution of software systems [27, 28, 40]. Long methods and large classes are examples of smell's symptoms, indicating the need for refactoring. *Refactoring* consists of improving the code quality without changing its external behavior [10], *e.g.,* by splitting methods or simplifying the code.

Although code smells can be detected manually [37], it is a time-consuming task in large systems. Thus, several automated detection tools have been proposed to help developers improve code quality [7, 25, 29–31, 35], exploiting different sources of information. For instance, DECOR [25] uses lexical and structural properties to identify smells. Palomba et al. [29] proposed a detection technique based on mining system history, and later [30] proposed a technique based on textual information, such as comments. In addition, several authors [7, 31, 35] proposed detection techniques based on combinations of software metrics. However, according to a systematic literature review in the area [42], their use has limitations: *(i)* difficulties in specifying metrics thresholds; *(ii)* code smells on specific metrics thresholds are often sensitive to the developer's perception; and *(iii)* a poor agreement in the tool's output.

Motivated by these limitations, our goal is to provide evidence of the usefulness of Machine Learning (ML) and Deep Learning (DL) models in detecting code smells, since both approaches uncover patterns in the data, *i.e.,* they can identify which features are aligned to the developer's perception of the presence of smells. In analyzing different models' performances, we can identify which model performs the best in different contexts. As a consequence, developers can use the model that is best suited to their needs.

For our purpose, we used the dataset *Madeyski Lewowski Code Quest*[1] (MLCQ) [21], composed of 14 features and 15,000 code fragments extracted from 523 industry-relevant Java projects, manually labeled by experienced software developers. The experts labeled the presence and severity of four code smells: *Blob*, *Data Class*, *Feature Envy*, and *Long Method*. However, due to the dataset focusing on the expert feedback (classification and severity), we extended the MLCQ dataset with new features that may help in the model learning process. We call our dataset *ImprovMLCQ*. Our new features are based on the output of (i) four state-of-the-art tools (CK [3, 7], Organic [7], PMD [31], and Designite [35]), selected based on their popularity [42]; (ii) features created using feature engineering techniques, such as clustering results, to enhance the predictive capabilities of the dataset; (iii) project characteristics, such as number of commits, code churn, file age, and change frequency.

To motivate the use of learning techniques in the field, we first presented an analysis of the agreement rate between three detection tools (Organic, PMD and Designite) and the experts' manual classification on the MLCQ original dataset. We show the poor capability of these tools in capturing the nuances of code smells as perceived by experts, obtaining many False Negatives, for example, Organic obtained 830 False Negatives, a number greater than the True Positives in 120 code fragments. Designite is the tool that detects the most code smell clusters. We also observed that the largest number of clusters appeared in code snippets manually labeled with class-level smells (Data Class and Blob).

[1] https://doi.org/10.5281/zenodo.3666840

Second, we trained 15 ML-based and 4 DL-based models on three distinct datasets based on feature selection (FS): (FS1) software metrics extracted by CK and Organic; (FS2) all features from FS1 plus 30 new measures obtained from project characteristics and feature engineering; and (FS3) top-5 features from the top-3 models obtained with FS2 learning. Our results demonstrate that enriching the MLCQ dataset significantly improves the performance of code smell detection models. ML algorithms such as Extra Trees, Random Forest, and Decision Tree achieved an F1-score of 100% for method-level smells (Long Method and Feature Envy) when trained on the feature-enriched dataset, the *ImprovMLCQ*. Similarly, DL models (e.g., GRU and AutoKeras) also achieved outstanding results, particularly for class-level smells (Data Class and Blob), surpassing traditional baselines. These findings validate the effectiveness of integrating project-level characteristics and advanced feature engineering techniques into code smell detection pipelines.

Our contributions are as follows:

- We analyzed the agreement on the classification and severity of code smells between automatic tools and experts. We have found that existing tools fail to capture the nuances of how developers perceive and assess code smells. This underscores the limitations of relying solely on predefined metric thresholds for automatic labeling, reinforcing the need for more sophisticated techniques that incorporate contextual and qualitative factors to improve detection accuracy.
- We analyzed 15 ML-based and 4 DL-based algorithms to assess their performance, and to identify which features contribute to the model's learning. We rank these algorithms using accuracy, recall, precision, and F1 metrics. For ML-based algorithms, *Extra Trees* (ET), *Decision Tree* (DT) and *Random Forest* (RF) performed the best. For DL-based algorithms, the best performing models were *Multi-Layer Perceptron* (MLP), *Gated Recurrent Unit* (GRU) and *Autokeras*. We also found that the enriched dataset significantly boosts the performance of ML and DL models. However, DL-based algorithms tend to underperform when detecting complex class-level smells.
- We identify the key features that had the greatest impact on the model's classification performance. For instance, *Number of Commits*, *Lines of Code* (LOC), and *Number of Agglomerations*, emerged as the most influential metrics, appearing among the top-ranked features in 5 out of 15 models, highlighting a clear distinction from the features traditionally used in state-of-the-art metric-based tools.
- Our pre-trained model serves as a valuable resource for deployment and customization in diverse software development contexts. The *ImprovMLCQ* dataset and all artifacts related to this research are publicly available in [6].

Audience. Researchers and practitioners in the field of software engineering may find our study's insights highly valuable for improving code smell detection tools.

2 Background and Related Work

In this section, we provide an overview of code smells, and we introduce the MLCQ dataset that served as a base for our work. Furthermore, we explore related works, highlighting existing approaches that rely on metric-based tools, and ML and DL models.

2.1 Code Smells, Severity and Agglomerations

To deliver high-quality software systems, developers should routinely practice good design principles. Deadline pressure or inexperience can lead to poor design and implementation choices, referred to as *code smells* [11]. Code smells are indicators that refactoring (modifying existing code without changing its behavior [11]) may be beneficial, resulting in code that is easier to understand and maintain. Each code smell is related to a code element type, *e.g.* classes or methods. We decided to investigate the four smells in the MLCQ's dataset, due to their reliability, since code smell fragments were manually validated by experts [10, 21]. Table 1 briefly describes the definitions of the four code smells investigated in this study according to [11].

Table 1: Code Smells Definitions

Code Smell	Definition
Long Method	Method with many lines of code and responsibilities.
Feature Envy	Method that calls more methods from another class than those where it belongs.
Blob	Class that centralizes most of the data processing, often handling excessive responsibilities, while other classes primarily encapsulate data and serve as its input.
Data Class	Class that does not implement enough functionality to justify its existence.

An important aspect of code smells is their severity, which indicate the impact they can have on the evolution of a software system. According to Madeyski al. [21], code smell severity can be classified as follows. *Critical*: the code smell must be reviewed immediately. *Major*: a code smell with a high impact on the system. *Minor*: a code smell that can slightly impact the system. *None*: no code smell was detected in the analyzed code fragment. The higher the severity, the more critical the issue. Consequently, developers may use severity to prioritize refactoring. In this study, one of our goals is to understand the correlation between the severity and agglomerations detected by automatic tools. A code smell agglomeration occurs when two or more smells co-occur in the same code fragment [33], for instance, on a class. Specifically, we aim at analyzing whether the presence of agglomerations (broader perspective) impacts severity classification, since the classification of severity is focused on one smell at a time (focused perspective). Moreover, we also investigate how agglomerations may contribute to ML and DL model performance.

2.2 MLCQ dataset

MLCQ is composed of 792 industry-relevant repositories from 37 different GitHub organizations, such as Google and Microsoft. MLCQ has 14 features and 15,000 code fragments manually labeled by 26 software development professionals [21]. Experts reviewed the presence of four code smells and manually assigned their severity. To add instances to the dataset, they used a voting method, in which each instance (a class or method) received votes from the experts

on the fragment smelliness. Additionally, the expert's background data was collected, such as level of education and professional experience in specific programming languages. Both the smell labeling dataset and the experts' background dataset are available at https://zenodo.org/records/3666840.

2.3 Related Work

Travassos et al. [37] introduced manual inspections and reading techniques to detect code smells. They have found an important limitation: *syntactic reading is tedious and should be automated*. Developers have to review the entire codebase of a project to identify code smells, and different developers have different perspectives on what a code smell is [28]. To address this limitation, two main types of automatic detection tools have been proposed in the literature: *metric-based* and *learning-based* tools. Traditional metric-based tools [7, 31, 35] are usually based on combinations of metrics and thresholds, requiring developers to manually adjust the rules. On the other hand, learning-based solutions automatically learn patterns from the data, reducing the need for human intervention [1].

2.4 Metric-based tools

Several tools focused on computing software metrics, applying thresholds to them, and combining the results (heuristic) to identify smelly code [7, 30, 31, 35]. These tools vary significantly in the types of code smells they detect, the target programming languages, the algorithms they employ, and the underlying metrics they leverage. Most of the tools are based on structural metrics, such as coupling, cohesion, complexity, and size. However, defining metric thresholds remains challenging, as they are often context-sensitive. System domain, project size, and coding standards are examples of factors that can influence the metric threshold definition, and their non-consideration may result in misclassifications. In this work, we add new features to the MLCQ dataset by employing a feature engineering framework that dynamically adapts to project-specific characteristics. Thus, reducing the reliance on fixed thresholds, enabling more accurate and context-aware smell detection across diverse software environments (see Section 3.4).

2.5 ML-based and DL-based tools

ML-based tools address metric-based tools' limitations by learning the features and context. Several ML solutions have been used to detect code smells [2, 18, 22, 23, 36]. Khomh et al. [18] extended Moha et al. [25] work using Bayesian Belief Networks to identify the Blob smell. Maiga et al. [23] used Support Vector Machine to identify four code smells (Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife) in three different systems. Amorim et al. [2] used Decision Trees to identify four code smells (Feature Envy, Long Method, Large Class and Misplaced Class) in one system. Contrary to our study, these studies are based on small and automatically labeled datasets.

Additionally, other studies are using DL-based tools [19, 26]. Nanadani et al. [26] investigate how manual labeling can be used to customize DL algorithms and improve their ability to detect code smells. The study analyzes 3 types of smells (Complex Method, Long Parameter List and Multifaceted Abstraction), and uses three DL algorithms (Autoencoder with MLP classifier, Autoencoder with LSTM classifier and Variational Autoencoder). Liu et al. [19] also used DL-based tools to detect smells (Feature Envy, Long Method, Large Class, and Misplaced Class). However, they do not use a manually labeled dataset as ground truth. These studies motivated us to further explore the performance of ML and DL-based tools when using the ImprovMLCQ, a manually labeled and larger dataset.

Unlike previous studies that also used the MLCQ [1, 22, 24], our work enhances the dataset by incorporating features extracted from four tools, and new features generated through feature engineering techniques. We investigate the feasibility of using automatic classification by metric-based tools as a ground truth benchmark, assessing its reliability and limitations. Additionally, we explore feature selection strategies to identify the most relevant features for code smell prediction. It is important to note that the original MLCQ dataset contained only information on the code smell type and its severity, whereas *ImprovMLCQ* significantly expands the available feature set, enabling deeper insights into code quality assessment.

3 Study Design

This section outlines our research questions, the methodological steps we followed, and the design of our model training and evaluation process.

3.1 Research Questions

This study aims to assess ML and DL-based solutions' performance in a feature-enriched version of the MLCQ, the *ImprovMLCQ*, in identifying and uncovering patterns in experts' perception of what is considered a code smell. We aim to answer the following research questions (RQ_s):

> **RQ_1**: How consistent are code smell labels between manual expert annotations and automated detection tools?

This question examines the level of agreement between automated code smell metric-based tools and the classification provided by experts. This analysis provides an assessment of the reliability of the tools, identifying gaps in current detection techniques and motivating researchers to seek other solutions to achieve a more accurate and context-aware classification.

> **RQ_2**: Can automatically labeled code smell agglomeration explain the severity of code smells?

This question aims at investigating if agglomerations can be used as a feature to enhance model performance, through a quantitative analysis of (i) agglomerations that were found in the same code fragment that the expert evaluated; and (ii) agreement between the presence of agglomeration on the code fragment and the severity assigned by experts. By answering this question, we can understand if the presence of agglomerations aligns with experts' perceptions of a smelly code, thereby improving the ability to prioritize the more severe code smells.

RQ$_3$: What is the *ImprovMLCQ* learning performance of state-of-the-art multi-label ML-based and DL-based models?

Our goal is to assess if ML and DL models can accurately identify code smells, evaluating 15 different ML-based and 4 DL-based models. Through the assessment of the model's performance, we can identify feature combinations that can lead to more reliable and effective code smell detection. This insight can guide future automated detection, focusing on high-impact features that support different coding standards and project contexts.

3.2 Study Steps

Figure 1 shows an overview of the study steps taken to build the *ImprovMLCQ* dataset and to answer our RQ$_s$: (1) MLCQ dataset (Step 1); (2) detection tools (Step 2) – Organic, PMD and Designite; (3) metrics tools (Step 3) – Organic and CK; (4) projects' repository characteristics (Step 4); and (5) feature engineering and preprocessing strategies (Step 5).

In Step 1, we pre-process the MLCQ to identify which information to collect, such as project characteristics, and to prepare it to include our new features. In Step 2, we used three detection tools (Organic [7], Designite [35], and PMD [31]) to identify the same code smells from the MLCQ. We selected these tools due to their community acceptance [42]. More details can be found at Section 3.3.

In Step 3, we used Organic [7] and CK [3] to compute software metrics at class and method level without the need to compile the projects. These metrics are used as features for the model's learning. Since the projects from the MLCQ dataset use GitHub, we have used the GitHub API (Step 4) to mine project features, such as the number of commits and collaborators.

In Step 5, we apply feature engineering and preprocessing techniques. For instance, we add a feature that indicates the presence of agglomerations; features that indicate the smells detected by automatic tools in that fragment, beyond the four smells labeled in MLCQ. For instance, Divergent Change and Shotgun Surgery. Finally, we also created new features using the output of two cluster techniques: Gaussian mixture models [14] and K-means clustering [20]. More details are presented in Section 3.4.

To answer RQ$_1$ and RQ$_2$, we use the results from steps 1 and 2. For RQ$_1$ we analyzed through a confusion matrix and the MCC measure the agreement rates between the manual classification made by the experts and the automatic ones performed by the tools. For RQ$_2$, we analyzed whether the presence of agglomerations detected by these tools influences the code smell severity labeled in that code fragment. To answer RQ$_3$, we have used the resulting *ImprovMLCQ* dataset (Step 5) to apply three Feature Selection (FS) techniques: (FS1, FS2 and FS3 – Section 3.4). Then, we used several ML-based and DL-based models and compared their performance across these different techniques. In the next sections, we will better detail each step.

3.3 Code Smell Detection and Metrics Tools

In this work, we used three different automatic detection tools in order to, beyond understanding if their output are good representatives of developer's perception on code smell presence, if the tool's output can be used as a feature in the ML models, *i.e.,* if different code smells identified by the tools can be useful in predicting the four code smells that we analyze (Long Method (LM), Feature Envy (FE), Data Class (DC), and Blob). For instance, the presence of a Large Class can help identify a Blob class. The selected tools are: Organic, PMD, and Designite. Organic and Designite are capable of identifying the four smells. Meanwhile, PMD can identify *Long Method*, *Blob*, and *Data Class*. They also allow developers to select which detection strategies to use: developers can configure threshold metric values to adjust for their context, or use its default configuration. For our purpose, we used the default configuration of the tools to avoid overfitting the tools to the analyzed projects.

Additionally, we use two state-of-the-art tools, Organic [7] and CK [3], to compute 20 additional software metrics. These tools calculate class-level and method-level code metrics using static analysis (*i.e.,* no need for executing the code). Currently, it calculates a large set of known metrics, including the Number of Methods, LOC (Lines of code), and Cyclomatic Complexity, and others.

A step to match the data from the automated tools (*e.g.*, CK and Organic) with the *ImprovMLCQ* was also needed. To match the original data from the MLCQ with the tools' output, we considered for each code fragment on MLCQ the commit hash, path, start, and end line. As each fragment are reviewed by several experts, we recorded for each of them the list of experts who performed the review and their corresponding classification and severity. And then, having matched a method/class on the MLCQ with the tools output, we added to the code fragment data the smells detected by each detection tool and its respective software metrics.

3.4 Feature Engineering

The MLCQ only has basic features about manually labeled code fragments, which are: start and end lines, commit hash, path, code smells manually labeled by experts, and the severity of these labeled code smells. With this data alone, it is not possible to use ML and DL techniques, so we expanded this dataset. So for our study, we created new features based on the results of the three code smell detection tools, two metric tools, and the projects' characteristics mined from GitHub. Table 2 illustrates some different types of features that are part of *ImprovMLCQ*. A few examples of features are: (i) code granularity, whether the code fragment is part of a method or a class; (ii) the reviewer's skill; (iii) its smell type, whether the code fragment is Long Method, Feature Envy, Data Class, or Blob; (iv) severity; (v) agglomeration, indicating the number and types of code smell agglomerations in a code fragment; (vi) number of Lines of Code for a code fragment; (vii) number of commits for a code fragment; and (viii) the relevance of the project according to GitHub classification. For example, the smell_toolname_num_aglomeration feature contains the number of agglomerations found by the tool in that code fragment. The smell_tool_name_aglomeration feature is binary and indicates whether one of the tools found an agglomeration in that code fragment. In total, *ImprovMLCQ* is composed of 191 features, see all the features in the supplementary material [6].

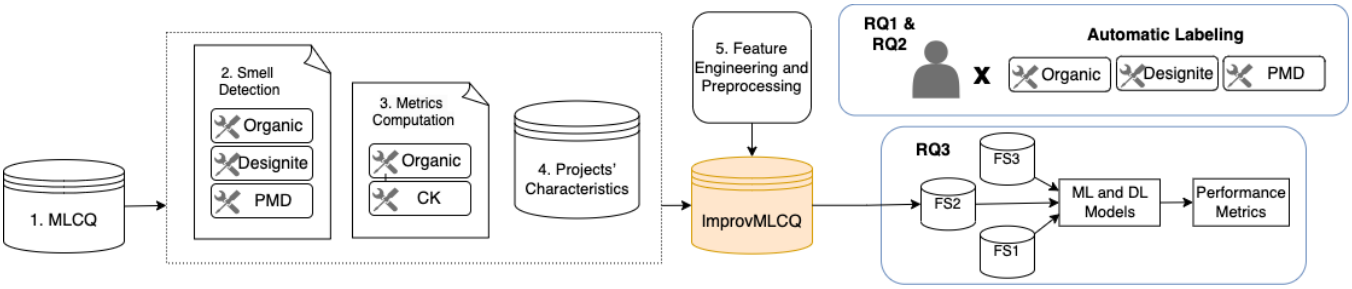

Figure 1: Overview of the Study Design

Table 2: Examples of Features from *ImprovMLCQ*

Type of Features	Description
MLCQ Dataset (FS1)	Classification of code fragments regard four code smell types and four severity categories
Metrics Computation Tools (FS1)	Use of CK and Organic to quantitative measures of the code fragments
Smell Detection Tools (FS2)	Use of PMD, Designite, and Organic to identify and label code smells automatically within a code fragment
Project Characteristics (FS2)	Broader characteristics related to the software project rather than individual code fragments
Feature Engineering (FS2)	Generation of new features based on existing data to reveal hidden patterns
top-5 features (FS3)	Features with the best FS2 training performance

Moreover, before using our data in the models' learning, we first had to pre-process it. For this purpose, we performed an Exploratory Data Analysis, utilizing the *ProfileReport* function from the YData Profiling library[2]. We selected this function due to its capability to generate a comprehensive and detailed report on the dataset. The initial report identified several issues, including duplicate rows, columns with null values (NaN), the presence of outliers, and variance problems. Additionally, it enabled the identification of columns requiring data transformation and standardization, particularly numerical variables stored as lists.

We established a discard criterion for all columns where more than 50% of the values were NaN, as the application of standard imputation methods in these cases could compromise data integrity, introduce noise, and reduce overall reliability. The columns affected by outliers and variance problems were already discarded due to these missing values, making additional preprocessing unnecessary. Subsequently, we reviewed missing records and replaced empty fields with 0, as many of the dataset's variables are binary, where 0 represents the default "False" state. We identified and removed duplicated lines. It happened because CK and Organic computed the same metrics. During the data standardization phase, all numerical columns prefixed with "CK" underwent a transformation, as these fields contained lists of numerical values, we replace each field with the average of the respective list elements. Furthermore, we excluded all textual variables that did not conform to categorical data (such as URLs, file names, and project addresses), as they do not contribute meaningful information to the model learning.

[2]https://docs.profiling.ydata.ai/latest/

The system retained all synthetically generated variables after data cleaning. Upon completion of the preprocessing steps, the dataset consists of 71 features, categorized into two types: 42 numerical and 29 categorical, and 13,489 records. Then, we proceed with considering three Feature Selection (FS) datasets: FS1, FS2, and FS3. Number of Methods and Exception Type Count are examples of software metrics extracted from the CK and Organic tool that were added to our first dataset (FS1). The (i) output of the three smell detection tools, (ii) the agreement between the tools when at least two tools detect code smells in the same piece of code, (iii) the project characteristics, such as team size and number of commits, and (iv) application of feature engineering, such as clustering of the systems and the presence of agglomeration, were added to our second dataset (FS2). Finally, our third dataset (FS3) is composed of the top-5 features discovered by learning from FS2 (Table 2). See our supplementary material for a detailed data dictionary of all considered features [6].

3.5 Model Training and Evaluation

We selected fifteen ML algorithms[3] using feature engineering and feature selection to compare their predictive performance (see Section 4.3). The algorithms are: *k-Nearest Neighbor* (kNN) [17], *Linear Regression* (LR) [17], *Support Vector Machine* (SVM) [17], *Decision Tree* (DT) [4], *Random Forest* (RF) [17], *Extreme Gradient Boosting package* (XGB) [17], *Ridge Classifier* (RC) [16], *Light Gradient Boosting* (LGB) [9], *Gradient Boosting* (GB) [13], *Ada Boost* (ADA) [12], *Extra Trees* (ET) [15], *Naive Bayes* (NB) [17], *Linear Discriminant Analysis* (LDA) [5], *Quadratic Discriminant Analysis* (QDA) [32] e *Dummy Classifier* (DC) [39]. These algorithms were chosen based on a list of the most frequently used ML algorithms in the areas of defect and code smell classification [34]. In addition, we selected four DL algorithms: *Multi Layer Perceptron* (MLP) [26], *Long Short-Term Memory* (LSTM) [26], *Gated Recurrent Unit* (GRU) [41] and *Autokeras*[4] algorithms. They were selected based on their performance on previous studies [17, 26, 41].

We trained the models using multi-label targets and calculated their performance based on the F1-score, a measure derived from precision and recall metrics that is more suitable for unbalanced datasets, such as *ImprovMLCQ*. We split our data in 80% for training and 20% for testing to avoid overfitting. Additionally, various

[3]We used the PyCaret library https://pycaret.readthedocs.io/en/latest to build and train our models.
[4]https://autokeras.com/tutorial/multi/

resampling methods were considered, including cross-validation and stratified sampling, ensuring a balanced representation of different classes. We focus our report on the average values across these splits. To fine-tune each model, we used the grid search technique. We highlight that we train the models over three variances of feature selection (FS) (*i.e.*, three datasets):

- (FS1) Uses automatically computed metrics from the Organic and CK tools. This set of 34 features provides a broad foundation of both structural metrics and labeling outputs to support predictive analysis.
- (FS2) Builds on FS1 by applying feature engineering techniques to create new features based on automatic code smell classifications and project characteristics. This step expands the feature set with engineered features, and we aim to understand if they can enhance the models' ability to capture patterns in code smells. FS2 has a total of 64 features.
- (FS3) Uses the top-5 most relevant features[5] selected from the ML-based model trained on FS2. We trained all top-3 models from FS2 to get the top-5 features, and we show the results for the model with the highest F1-score. For an exhaustive analysis of all our results and findings, please see our supplementary material [6].

3.6 Evaluation Metrics

We use the confusion matrix (Table 3) aiming to investigate the agreements' rate between the manual and automatic labeling, and the ML-based model performance. Each row of the matrix represents the automatic tool labeling (*i.e.*, by Organic, PMD, Designite, and ML-based models), while each column represents the MLCQ experts' labeling (a.k.a., ground truth). In this matrix, TP (True Positive) is when both the tool and MLCQ label the code sample with the smell. TN (True Negative) is when both the tool and MLCQ label the code sample as not having the smell. FP (False Positive) is when the tool labels the code sample with a smell, but MLCQ does not label it. FN (False Negative) is when the tool does not label the code sample with the smell, but in MLCQ the expert labels it with the smell. Several important metrics are derived from the confusion matrix (see Table 4):

- *Precision*: It is the proportion of correctly predicted positive instances out of all instances predicted as positive.
- *Recall*: It is the proportion of actual positive instances that were correctly identified by the model.
- *F1-Score*: It is the harmonic mean of precision and recall, balancing the trade-off between these metrics.

Table 3: Confusion matrix. TP = True positive; FP = False positive; TN = True negative; FN = False negative

		MLCQ labeling	
		Smelly	N-Smelly
Tools' labeling	Smelly	TP	FP
	N-Smelly	FN	TN

[5]We used PyCaret's function plot_model(model, plot='feature').

Table 4: Evaluation Metrics Used in this Study

Metric	Precision	Recall	F1-Score
Formula	$\frac{TP}{TP+FP}$	$\frac{TP}{TP+FN}$	$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

We also use Matthews Correlation Coefficient (MCC). MCC is used for evaluating the agreement between automatic and manual labeling. It measures the quality of binary classifications and uses all values of the confusion matrix (see Equation 1).

$$MCC = \frac{(TP * TN - FP * FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (1)$$

The MCC value ranges from -1 to +1. A coefficient +1 indicates perfect agreement, 0 indicates no better than random labeling, and -1 indicates total disagreement.

3.7 Experiment Setup

The machine specifications for running Organic, PMD, Designite, and CK were 64 GB of RAM memory, 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz, and no dedicated GPU. The Operating System was Ubuntu 22.10. All learning experiments have been performed on Google Colaboratory[6] using Phyton as a programming language, and the scikit-learn[7] and Pycaret[8] libraries for predictive data analysis. All used scripts and data can be found in our complementary material (see Section 5).

4 Results and Discussion

This section discusses the results of the research questions (see Section 3.1). We focus on the most interesting insights and present the complete report in our supplementary material [6].

4.1 Manual vs Automatic Labeling (RQ_1)

Manually labeling a large dataset of code smells is both costly and subjective. Developers' perceptions of smelly and non-smelly (N-smelly) code are influenced by various factors, such as their background and project characteristics. To assess the reliability of automatic detection tools, we first analyzed the agreement between manual and automated code smell classification. Table 5 presents the confusion matrix for the detection strategies, comparing automatic labeling with expert-labeled MLCQ data. Rows shows the results for the detection tools, while the columns presents the results for the manual classification. For instance, in the case of the Long Method (LM) smell, we observed for Organic – True Positives (TP): 120 instances correctly labeled as smelly by both Organic and MLCQ experts. For Organic – True Negatives (TN), 217 instances were correctly labeled as non-smelly (N-Smelly) by both sources. Organic – False Positives (FP): 830 instances incorrectly identified as smelly by Organic but labeled as non-smelly by MLCQ experts. Organic – False Negatives (FN): 3 instances labeled as non-smelly by Organic but identified as smelly by experts.

[6]https://colab.research.google.com/
[7]https://scikit-learn.org/
[8]https://pycaret.readthedocs.io/en/latest

Table 5: Confusion Matrix for Code Smells and Tools. Columns labeled *Smelly* and *N-Smelly* correspond to the experts' labeling.

Tool	Classification	Long Method (LM)		Feature Envy (FE)		Data Class (DC)		Blob	
		Smelly	N-Smelly	Smelly	N-Smelly	Smelly	N-Smelly	Smelly	N-Smelly
Organic	Smelly	***120***	830	***29***	474	70	1,236	104	1,203
	N-Smelly	3	217	***36***	592	10	706	36	784
PMD	Smelly	43	***907***	-	-	***191***	1,115	***270***	1,037
	N-Smelly	2	***218***	-	-	***36***	680	***80***	740
Designite	Smelly	99	851	0	***503***	0	***1,306***	0	***1,307***
	N-Smelly	***15***	205	0	***628***	0	***716***	0	***820***

For Long Method, Organic exhibited the highest capability in correctly identifying the smell, with a low False Negatives (FN) rate (3 instances). However, its reliability is compromised by an excessive number of False Positives (FP) cases (830), suggesting an overly aggressive classification approach. Similarly, PMD and Designite also show high FP rates, with 907 and 851 instances misclassified as Long Method, respectively.

For Feature Envy, all tools performed poorly in detecting smelly instances, despite showing some agreement on N-smelly ones (True Negatives (TN): 592 for Organic and 628 for Designite). For Data Class and Blob smells, Organic and Designite exhibit high False Positives (FP) rates (1,236 and 1,203 for Organic; 1,115 and 1,037 for PMD; 1,306 and 1,037 for Designite). This suggests significant disagreement between automatic tools and manual classification, highlighting a lack of precision in identifying these code smells. Consequently, using these tools in isolation may bias the findings with FP. This motivates us to further explore if they can be used as a feature for more sophisticated detection tools.

To quantify agreement, we compute the Matthews Correlation Coefficient (MCC) in Table 6. High MCC values (close to 1) indicate strong agreement, while low values (close to -1) signify poor correlation. Values close to 0 indicates weak or almost no agreement. The results show consistently low MCC values, confirming weak alignment between automatic tools and expert labeling. Improving code smell detection requires balancing sensitivity (detecting TP) and specificity (reducing FP). This highlights the need for enhanced feature engineering, threshold adjustments, and ML-based and DL-based solutions.

Table 6: MCC for Code Smells and Tools

Tool	LM	FE	DC	Blob
Organic	***0.1435***	***0.0007***	0.0972	0.0700
PMD	0.0735	-	***0.1454***	***0.1431***
Designite	0.0475	-	-	-

Finding 1: All three automated tools show limited ability to align with MLCQ data labeled by experts in identifying instances of code smells, as indicated by the high rate of False Positives (FP). Therefore, the current tools may not adequately capture the nuances of code smells as perceived by experts. This suggests a systemic issue with previous studies that use these tools for identifying code smells, *e.g.*, to train models. The poor agreement emphasizes the importance of being cautious when using these tools as ground truth.

4.2 Agglomeration Analysis (RQ_2)

Our goal in answering this question is two-fold. First, we want to understand if agglomerations detected by automated tools agree with the experts' manual classification. We highlight that we consider an agglomeration as two or more occurrences of smells on a code fragment, and they may not necessarily contain the four analyzed smells. For instance, we analyzed whether an agglomeration composed of Divergent Change and Long Parameter List may indicates the Long Method as classified by the experts. Second, we want to understand if agglomerations detected by automated tools reflect in the severity assigned by the experts.

Table 7 shows a deeper view of the agreement rates of agglomerations identified by detection tools with expert classification of smell types (left side) and severity (right side). The last row (#Instances by Experts) provides a total count of instances labeled by experts in each category. These totals help in understanding the distribution of instances across different smell types and severity categories, providing insight into how prevalent each type of code smell is within the dataset.

Table 7 (left side) presents the agreement rates between the manual labeling and the agglomeration presence. Rows presents the tools that detected the agglomerations, while the columns presents, for each of the four smells classified by the experts, the respective agreement. For instance, Organic identified that 6% (74 instances) of the Long Method (LM) instances identified by the experts have an agglomeration. The results show that agglomerations identified by Designite (3rd row) are good indicators of the presence of the smells classified by the experts, especially for smells at class (Data Class and Blob). Meanwhile, agglomerations identified by Organic did not agree meaningfully with the experts (2-6%).

Table 7: Agreement of Agglomerations Identified by Detection Tools with Expert Classification of Smell Type and Severity

Tool	Smell Type				Severity			
	LM	FE	DC	Blob	Critical	Major	Minor	None
Organic	74 (6%)	53 (5%)	50 (2%)	86 (4%)	37 (8%)	85 (6%)	73 (3%)	270 (2%)
PMD	246 (21%)	127 (11%)	606 (30%)	1,003 (47%)	205 (42%)	437 (32%)	661 (30%)	1,967 (14%)
Designite	***569 (49%)***	***506 (45%)***	***1,288 (64%)***	***1,344 (63%)***	***317 (65%)***	***820 (60%)***	***1,225 (55%)***	***7,184 (52%)***
#Instances by Experts	1,170	1,131	2,022	2,127	487	1,359	2,220	13,767

Overall, we can observe that Designite's agglomerations can have the potential to uncover patterns in experts' classifications.

Finding 2: The highest agreement is between Designite's agglomeration and expert classification, mainly for Data Class (DC) and Blob. This underscores the importance of incorporating new feature engineering techniques, (*e.g.*, based on Designite's patterns) to automatically detect code smells and assist software quality and maintainability. This finding also indicates that studies considering code smell co-occurrence as a feature, investigating how different combinations of them can improve the code smell detection's performance in both Machine Learning and Deep Learning, should be further explored.

Table 7 (right side) is organized in the same fashion as the Smell Type analysis, but it is focused on presenting the results by severity degrees while considering all code smell types. The rows present the agreement rates for each tool, and the columns represent the agreement rates by severity degree. The table shows that Designite achieves the highest agreement across different severity categories (see 3rd row). For instance, 65% of instances labeled as *Critical* by experts were also classified as agglomerations by Designite. Notice that Designite classifies a significant number of instances (52%) as having agglomeration, while the severity classification was *None* by experts.

The significant presence of agglomerations in non-smelly instances highlights the tool's tendency to over-identify agglomerations, which might contribute to False Positives (FP). This suggests that while Designite is adept at identifying multiple issues, it might also incorrectly flag clean code fragments as problematic. Moreover, the lower percentages of single-smell instances indicate that Designite may have a higher tendency to identify code fragments with multiple intertwined issues rather than isolated ones. Consequently, further investigation may be warranted to understand factors contributing to Designite's high agreement rates, addressing any discrepancies of FP identified in its classification results.

Finding 3: We show Designite's discrepancies related to *None* instances. The significant agreement for these instances raises questions about the correlation between severity and agglomeration.

4.3 ImprovMLCQ Learning Performance (RQ_3)

Table 8 presents the results obtained for the top-3 ML-based models. The predominant occurrence in the top-3 models includes algorithms that work with a high degree of randomness during the learning process, such as ET and RF, as well as those that perform well with imbalanced datasets, like LGB and GBC.

It becomes evident that FS1 underperforms when compared to FS2 across all four code smells, regardless of the selected model (see F1-score values). This performance gap can be attributed to the limited scope of FS1, which includes only metrics derived from CK and Organic tools. In contrast, FS2 leverages a broader and more diverse set of features, integrating structural metrics with repository-based metadata and aggregated characteristics derived from preprocessing. This richer representation enabled the models to learn more nuanced patterns, contributing significantly to the improved classification performance.

Interestingly, FS3 (composed of the top-5 features selected via feature importance rankings) fails to outperform FS2. This outcome suggests that, in this domain, feature selection alone may not be sufficient to maximize performance. Although feature selection techniques aim to retain the most informative variables and discard noisy or redundant ones, they inherently assume that a subset of features can capture the relevant patterns for all classes and models equally well. This assumption may not hold in complex tasks, such as code smell classification, where different smells may depend on distinct and possibly complementary feature sets. Therefore, the reduction in feature diversity in FS3 likely led to the loss of useful contextual information, limiting the model's ability to distinguish between smelly and non-smelly code. Thus, broader and well-engineered feature sets (as in FS2) may yield better results than narrowly focused or overly reduced subsets.

The highest F1-score for code smells defined at the method level was 1 (for FS2), exhibiting a higher F1-score than code smells from the class level. For class-level code smells, the highest F1-score value was 0.58 (also for FS2). This could be because the complexity involved in defining method code smells is lower than that for class code smells, regardless of whether the evaluation is conducted by a human or a tool. This complexity is reflected in the data. This result aligns with expectations, leading us to validate the features used in the model composition as satisfactory. See the comparison of FS to ML in the table 9.

Table 8: Top-3 ML Models per Feature Selection Strategy - Long Method, Feature Envy, Data Class and Blob

	Long Method (LM)				Feature Envy (FE)				Data Class (DC)				Blob			
	Model	Recall	Prec.	F1	Model	Recall	Prec.	F1	Model	Recall	Prec.	F1	Model	Recall	Prec.	F1
FS1	RF	0.64	***0.23***	***0.34***	RF	***0.30***	0.37	***0.33***	RF	***0.41***	***0.61***	***0.49***	ET	0.39	***0.53***	***0.45***
	GBC	***0.65***	***0.23***	***0.34***	ET	0.28	***0.40***	***0.33***	ET	***0.41***	0.60	***0.49***	DT	0.39	***0.53***	***0.45***
	ET	0.63	***0.23***	***0.34***	DT	0.28	***0.40***	***0.33***	DT	***0.41***	0.60	***0.49***	GBC	***0.40***	0.51	***0.45***
FS2	ET	***1***	***1***	***1***	ET	***1***	***1***	***1***	ET	***0.59***	***0.59***	***0.59***	ET	***0.53***	0.66	***0.59***
	RF	***1***	***1***	***1***	RF	***1***	***1***	***1***	RF	0.58	0.58	0.58	RF	0.52	***0.67***	***0.59***
	DT	***1***	***1***	***1***	DT	***1***	***1***	***1***	DT	***0.59***	0.57	0.58	DT	***0.53***	0.66	***0.59***
FS3	LGB	***0.22***	***0.15***	***0.18***	ET	***0.29***	***0.17***	***0.21***	KNN	***0.15***	***0.28***	***0.20***	ET	***0.48***	***0.59***	***0.53***
	ET	***0.22***	***0.15***	***0.18***	DT	***0.29***	***0.17***	***0.21***	LGB	0.13	0.27	0.17	DT	***0.48***	***0.59***	***0.53***
	DT	***0.22***	***0.15***	***0.18***	LGB	***0.29***	***0.17***	***0.21***	RF	0.13	***0.28***	0.17	RF	0.47	***0.59***	***0.53***

Table 9: Comparison for ML Models

Comparison	U Statistic	p-value	Significant (p < =0.05)
FS1 vs FS2	0.00	0.06	No
FS1 vs FS3	9.00	0.06	No
FS2 vs FS3	9.00	0.07	No

Finding 4: Our findings indicate that integrating diverse feature sets (FS2) into *ImprovMLCQ* dataset significantly enhances the learning accuracy of ML-based models for code smell detection. Meanwhile, for the FS3 set, in which the top five features that most contributed to the top-three models on the FS2 set are considered, did not improve the model's capabilities in detecting the four code smells. Even worse, for all smells, except Blob, the model's performance is worse than tossing a coin. This indicate that, even though their contribution is significant in FS2, the models needs more features to learn the patterns.

Table 11 presents the results obtained for the 4 DL-based models. It also indicates that FS1 performed the worst among the three datasets, with an F1-score below 0.4. FS1 for DL showed significantly lower results compared to FS1 for ML. FS2 exhibited the best performance across all code smells for the MLP, GRU, and Autokeras models. The LSTM was unable to learn the pattern from the FS2 dataset, highlighting that the LSTM technique is not suitable for detecting these smells. Comparing the results with the results in Table 8 for class-level code smells, DL-based algorithms performed better. The DL models by code smell type showed superior results to those demonstrated by the ML models for FS3, where method-type code smells showed higher values in all metrics compared to class code smells. An unexpected result was that all four DL-based models achieved an F1-score greater than 0.63 for the FS3 dataset, performing better than ML-based models for class-level smells (Data Class and Blob). This finding suggests that even when considering only the top-5 most relevant features, it is possible to achieve competitive predictive accuracy with DL models. These results highlight the potential of a more refined feature selection approach, where a carefully chosen subset of features can yield effective classification models while reducing complexity.

DL captures complex relationships between attributes and adapts to the contexts of the trained data, which may have influenced its results being better than those of ML. See the comparison of FS to DL in the table 10.

Table 10: Comparison for DL Models

Comparison	U Statistic	p-value	Significant (p < =0.05)
FS1 vs FS2	4.00	0.30	No
FS1 vs FS3	0.00	0.03	Yes
FS2 vs FS3	12.00	0.30	No

Finding 5: Our findings indicate that using the top-5 features in DL-based models yields results that are close to FS2, with a significant performance improvement for class-level smells compared to ML-based models. This allows us to abstract the model, enhancing its efficiency while maintaining good accuracy, ultimately facilitating the classification of code smells in real-world scenarios.

Overall, FS1 performed the worst across all datasets, where F1-scores fell below 0.5 for ML and 0.4 for DL. The superior results of FS2 confirm that *ImprovMLCQ* is a reliable dataset for code smell classification using ML-based and DL-based algorithms.

4.4 Threats to Validity

This section discusses potential threats to the validity of this study according to the four categories proposed by Wholin et al. [38]: *internal*, *external*, *construct*, and *conclusion* validity.

Internal Validity To ensure data integrity and avoid errors during data collection, model training, and analysis, we used well-established Python libraries such as *PyCaret*. Additionally, we manually reviewed all notebooks multiple times to guarantee that the experimental procedures were correctly followed and that the findings are consistent with the data. Still, conducting experiments with

Table 11: Top-4 DL models per feature selection strategy - LM, FE, DC and Blob

		Long Method (LM)			Feature Envy (FE)			Data Class (DC)			Blob		
	Model	Recall	Prec.	F1	Recall	Prec.	F1	Recall	Prec.	F1	Recall	Prec.	F1
FS1	MLP	0.10	0.39	0.16	0	0	0	0.06	***0.58***	0.10	0.05	***0.62***	0.09
	LSTM	0.10	***1***	0.02	0.01	0.29	0.03	0	0	0	0.23	0.55	0.33
	GRU	0.08	0.31	0.12	0.01	***0.50***	0.01	***0.16***	0.46	***0.34***	0.25	0.57	0.34
	AutoKeras	***0.15***	0.35	***0.21***	***0.07***	0.40	***0.11***	0.14	0.43	0.21	***0.28***	0.54	***0.37***
FS2	MLP	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***
	LSTM	0	0	0	0	0	0	0	0	0	0	0	0
	GRU	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***
	AutoKeras	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***1***
FS3	MLP	***1***	***1***	***1***	***1***	***1***	***1***	0.67	0.59	0.63	0.61	0.66	0.63
	LSTM	***1***	***1***	***1***	***1***	***1***	***1***	0.67	***0.61***	***0.64***	***0.64***	***0.69***	***0.66***
	GRU	***1***	***1***	***1***	***1***	***1***	***1***	0.67	0.59	0.63	0.60	0.68	0.64
	AutoKeras	***1***	***1***	***1***	***1***	***1***	***1***	***0.71***	0.60	0.63	0.58	***0.69***	0.63

other algorithms and tuning parameters is an important next step, which is part of our future work.

External Validity The generalizability of our findings is limited in several ways. First, all systems in the dataset are implemented in the Java programming language, which may reduce applicability to systems developed in other languages. Second, although the MLCQ dataset comprises 792 real-world software systems, it may not represent the full diversity of industry projects. Future work will investigate the transferability of our approach to different languages and datasets.

Construct Validity This study focuses on analyzing the severity of four specific code smells: *Long Method*, *Feature Envy*, *Data Class*, and *Blob*. As a result, the findings may not extend to other types of code smells. Moreover, we relied on four state-of-the-art tools (CK, Organic, PMD, and Designite) to extract code metrics and smell labels. While these tools have been widely used in prior studies, their internal heuristics and limitations may introduce bias in how design issues are identified and measured.

Conclusion Validity Although we employed robust methods and tools throughout our study, threats to conclusion validity still exist. Furthermore, as the results are based on the current version of the dataset and tooling, replication with alternative configurations is essential to confirm the reliability of our findings.

5 Conclusion

Manual labeling of code smells is a complex and time-consuming process. Currently, not all available code smell datasets are reliable reference bases for real-world software development contexts, since they use metric-based tools as a ground truth. This study demonstrates relatively low agreement rates between both labeling processes, highlighting the importance of cautious interpretation of automated metric-based tools' outputs.

We also explore the reliability of a dataset built in an automated manner using other metrics not considered by state-of-the-art tools. Our evaluation of 15 ML-based and 4 DL-based algorithms; and three feature selection strategies (FS1, FS2, and FS3) for predicting four types of code smells (Long Method, Feature Envy, Data Class, and Blob) revealed key insights: (1) The consideration of several features resulting from the employment of feature engineering strategies consistently delivered the best performance across all code smells. (2) ML-based models like RF, ET and DT balanced performance making them suitable for method-level smells (Long Method and Feature Envy), while DL-based models like MLP, GRU and AutoKeras were more suitable for class-level smells (Data Class and Blob). DL-based models presented very good results while considering only the top-5 features.

Based on these findings, future research should focus on refining feature engineering strategies, initiate a review of heuristic-based tools to ensure that they assign greater importance to the variables identified as key by learning models, and experimenting with different sampling and transfer learning approaches. We gathered preliminary insights about the effectiveness of under-sampling (see our supplementary material [6]). Future studies should explore over-sampling strategies, like SMOTE (Synthetic Minority Over-sampling Technique) [8]. We also aim to expand the evaluation to include additional code smells, as more diverse datasets can provide a broader understanding of the models' effectiveness and generalizability.

ARTIFACT AVAILABILITY

All artifacts of this study can be accessed at ImprovMLCQ's GitHub repository in[6].

ACKNOWLEDGMENTS

This research was partially supported by the Brazilian funding agencies CAPES (Grant 88881.879016/2023-01), CNPq (GD), FAPEMIG (Grant APQ-01488-24), and FAPESP (Grant 2023/00811-0). We also acknowledge the Brazilian company Stone[9] for their support.

[9]https://www.stone.com.br/

REFERENCES

[1] 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.

[2] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. 2015. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE, 261–269.

[3] Maurício Aniche. 2015. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/. Accessed March 20, 2025..

[4] Géron Aurélien. 2019. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. *Concepts, tools, and techniques to build intelligent systems, 2nd ednn* (2019).

[5] Suresh Balakrishnama and Aravind Ganapathiraju. 1998. Linear discriminant analysis-a brief tutorial. *Institute for Signal and information Processing* 18, 1998 (1998), 1–8.

[6] Joanne Carneiro, Jessica Barbara da Silva Ribas, Amanda Santana, Juliana Alves Pereira, and Eduardo. Figueiredo. [n. d.]. ImprovMLCQ: Improving the Madeyski Lewowski Code Quest Dataset. https://github.com/aisepucrio/ImprovMLCQ. Accessed March 20, 2025.

[7] Diego Cedrim and Leonardo Sousa. 2017. Organic. Available in https://github.com/diegocedrim/organic. Accessed February 5, 2025..

[8] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[9] Junliang Fan, Xin Ma, Lifeng Wu, Fucang Zhang, Xiang Yu, and Wenzhi Zeng. 2019. Light Gradient Boosting Machine: An efficient soft computing model for estimating daily reference evapotranspiration with local and external meteorological data. *Agricultural water management* 225 (2019), 105758.

[10] Marting Fowler. 1999. Refactoring: Improving the design of existing code addison-wesley professional. *Berkeley, CA, USA* (1999).

[11] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[12] Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[13] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[14] Carl Friedrich Gauss. 1877. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Vol. 7. FA Perthes.

[15] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine learning* 63 (2006), 3–42.

[16] Arthur E Hoerl and Robert W Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (1970), 55–67.

[17] 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 *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 840–849.

[18] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*. IEEE, 305–314.

[19] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.

[20] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[21] 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*. 342–347.

[22] Lech Madeyski and Tomasz Lewowski. 2023. Detecting code smells using industry-relevant data. *Information and Software Technology* 155 (2023), 107112.

[23] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aimeur. 2012. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*. 278–281.

[24] Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, and Juliana Alves Pereira. 2024. Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 302–312.

[25] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.

[26] Himesh Nanadani, Mootez Saad, and Tushar Sharma. 2023. Calibrating deep learning-based code smell detection using human feedback. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 37–48.

[27] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.

[28] 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 *IEEE International Conference on Software Maintenance and Evolution*. 101–110.

[29] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining version histories for detecting code smells. *Transactions on Software Engineering* 41, 5 (2014), 462–489.

[30] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 1–10.

[31] PMD. 2025. *PMD Source Code Analyzer*. Available in https://github.com/pmd/pmd/tree/main. Accessed March 12, 2025..

[32] Yingli Qin. 2018. A review of quadratic discriminant analysis for high-dimensional data. *Wiley Interdisciplinary Reviews: Computational Statistics* 10, 4 (2018), e1434.

[33] Amanda Santana, Eduardo Figueiredo, and Juliana Alves Pereira. 2024. Unraveling the Impact of Code Smell Agglomerations on Code Stability. In *International Conference on Software Maintenance and Evolution (ICSME)*. 461–473.

[34] Geanderson Santos, Amanda Santana, Gustavo Vale, and Eduardo Figueiredo. 2023. Yet Another Model! A Study on Model's Similarities for Defect and Code Smells. In *Fundamental Approaches to Software Engineering*. Springer Nature, 282–305.

[35] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: A software design quality assessment tool. In *Proceedings of the 1st international workshop on bringing architectural design thinking into developers' daily activities*. 1–4.

[36] 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, 336–345.

[37] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R Basili. 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM sigplan notices* 34, 10 (1999), 47–56.

[38] Claes Wholin, Per Runeson, Martin Host, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2000. Experimentation in software engineering: an introduction. 274 pages.

[39] Stewart W Wilson. 2002. Classifiers that approximate functions. *Natural Computing* 1, 2 (2002), 211–234.

[40] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* 86, 10 (2013), 2639–2653.

[41] Dongwen Zhang, Shuai Song, Yang Zhang, Haiyang Liu, and Gaojie Shen. 2023. Code Smell Detection Research Based on Pre-training and Stacking Models. *Latin America Transactions* 22, 1 (2023), 22–30.

[42] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* 23, 3 (2011), 179–202.