# Tuning Code Smell Prediction Models: A Replication Study

Henrique Gomes Nunes
henrique.mg.bh@gmail.com
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Amanda Santana
amandads@dcc.ufmg.br
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Eduardo Figueiredo
figueiredo@dcc.ufmg.br
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Heitor Costa
heitor@ufla.br
Federal University of Lavras
Lavras, Minas Gerais, Brazil

## ABSTRACT

Identifying code smells in projects is a non-trivial task, and it is often a subjective activity since developers have different understandings about them. The use of machine learning techniques to predict code smells is gaining attention. In this replication study, our goals are: (i) verify if previous model's performance maintain when we extract data from updated systems; and (ii) explore and provide evidences of how the use of different feature engineering and resampling techniques can enhance code smell prediction model's performance. For these purposes, we evaluate four smells: God Class, Refused Bequest, Feature Envy and Long Method. We first replicate a previous study that focus on the algorithm's performance to identify the best models for each smell using a different dataset composed of 30 Java systems. This first experiment provides us a baseline model that is used in the second experiment. In the second experiment, we compare the performance of the baseline model with other models tuned with polynomial features and resample techniques. Our main results are: for datasets with imbalances lower than a ratio of 1:100, such as God Class and Long Method, the use of oversample techniques yielded better results. For datasets with more severe imbalance, like Refused Bequest and Feature Envy, the undersample techniques performed better. The feature selection technique, despite a minor impact on the results, provided insights. For instance, we need new features to represent some code smells, such as Long Method and Feature Envy.

## KEYWORDS

Code Smells, Machine Learning, Replication Study

**ACM Reference Format:**
Henrique Gomes Nunes, Amanda Santana, Eduardo Figueiredo, and Heitor Costa. . Tuning Code Smell Prediction Models: A Replication Study. In . ACM, New York, NY, USA, 11 pages.

## 1 INTRODUCTION

Code smells are code structures that can compromise the internal quality of systems [17]. Between 1990 and 2017, at least 351 studies were conducted on code smells, with detecting their structures being the most prevalent focus, representing 30% of the total [45]. Those studies presented approximately 80 tools and techniques to identify code smells [14, 45]. Several techniques to detect smells in source code have been proposed, most based on software metrics and thresholds. However, to use thresholds, developers must adjust them according to their context and system size. One way to overcome the threshold limitation is to use machine learning techniques to detect code smells.

,
.

To use machine learning, most models need a ground truth to know beforehand which instances contain code smells. However, building datasets is a non-trivial task and requires substantial effort [43]. Another problem raised by machine learning modeling is that code smell datasets are highly imbalanced [1, 10, 40], *i.e.,* they have more negative instances than positive ones. Several studies used machine learning techniques to deal with imbalanced datasets [18, 22]. To our knowledge, those studies did not try to understand how different techniques and their parametrizations may impact the model performance in predicting code smells. Furthermore, most of the studies in the literature only use linear features [10]; thus, exploring polynomial features can lead to relevant research studies.

This paper replicates a base study [10] to obtain new insights into predicting code smells using machine learning on systems from GitHub[1]. However, we further expand this study by evaluating the impact of feature engineering and resample data techniques on the machine learning model performance. For that task, we collected 30 Java software systems from Github and evaluated them using four code smells (God Class, Refused Bequest, Feature Envy, and Long Method). We have used five code smell detection tools to build our ground truth.

The results highlight that the models achieved better prediction capacity by reducing imbalance. In our study, models generated by Decision Tree, Random Forest and Gradient Boosting Machine achieved, in general, the best performance. The performance in predicting the God Class and Long Method code smells was superior, and the performance was worse in predicting the Refused Bequest and Feature Envy code smells. The Undersample technique was better for cases with higher imbalance, while oversample techniques performed better for cases with less data imbalance. The Refused Bequest and Long Method did not improve their performance when adding new features at training. In God Class and Feature Envy, feature selection indicated valuable insights, suggesting the need for new ways of representing them.

The main contributions of this study were:

[1]https://github.com/

- Provide a public dataset with 50k classes and 295k method instances, with ground truth for four types of code smell;
- Replicate a study using a dataset with more modern systems;
- Insights about feature selection, polynomial features, and resample data.

The rest of this paper is structured as follows. Section 2 presents a background for this study. Section 3 discusses the base study that we replicated. Section 4 shows how we conducted the experiments. Section 5 presents the results. Section 6 discusses the findings of this study and how they can be useful for different actors. Section 7 shows threats to validity. Section 8 discusses related works. Section 9 includes the conclusion and ideas for future studies.

## 2 BACKGROUND

### 2.1 Code Smells

Code Smell is an indication that usually corresponds to a deeper problem in the design or code structure of systems [17]. They contribute directly to technical debt if overlooked and left unaddressed; thus, successful long-term projects must avoid them. Code smells are often detected using hard-coded rules in detection tools [14, 21]. Thus, code smells are code snippets (or several snippets) that normally correspond to a more extensive problem in the system [30]. Once such code smells are detected, increased focus must be placed on them, as they will likely require extra effort during maintenance and implementation of new features. Sometimes, such detection can lead to reworking the module as refactoring [30]. Code smells have been widely studied and associated with faults and bad-quality code [7, 11, 19, 44, 51, 52]. We analyzed four code smells, where two are related to classes (God Class - GC and Refused Bequest - RB), and two are related to methods (Feature Envy - FE and Long Method - LM). The God Class code smell represents a class with excessive responsibilities, strongly indicating design flaws [37]. The Refused Bequest code smell means a child class does not fully support all the methods or data it inherits [17]. The Feature Envy code smell indicates, for instance, that a method is more interested in a class other than the one it is in [17]. One method with the Long Method code smell is complex, including many data and responsibilities [17].

### 2.2 Machine Learning

Several techniques have been proposed to identify code smells, such as a combination of software metrics [27, 36, 48], the use of historical data [39], refactoring opportunities [15], and machine learning models [2, 10, 13, 41]. Machine learning approaches eliminate constraints relying on metrics thresholds. Our study aims to replicate previous study and evaluate the performance of seven machine learning algorithms, considering the impact of using different resampling techniques, non-polynomial features, and hyperparameters tuning. Exploring and presenting the results of different techniques can bring insight into which one should be further evaluated and adopted by the community.

Identifying code smells is a classification problem in which each class/method of a project is analyzed and classified according to the smells it contains [3]. Consequently, the models receive as input a vector of features, in our case, software metrics. There are two ways to learn: a supervised and unsupervised approach. Supervised learning indicates that the model is trained based on the actual classifications of the classes/methods, *i.e.*, we need a ground truth containing if the class/method is smelly or not to adjust the model. Meanwhile, an unsupervised model separates the data according to the available data, classifying the instances according to their similarities. In this study, we evaluated seven supervised models: Multi-Layer Perceptron (MLP) [20], Naïve Bayes (NB) [28], Logistic Regression (LR) [23], Decision Trees (DT) [6], Random Forest (RF) [5], Gradient Boosting Machine (GBM) [8], and K-Nearest Neighbors (KNN) [9]. The selected models use different strategies, for instance, conditional probability (NB), function modeling (LR), tree-based algorithms (DT, RF, GBM), clustering (KNN), and neural network (MLP). We further separated our data into training and unseen data. We used the training data to train and adjust the models, enhancing their performance, and we used unseen data to evaluate the performance of the models after training. Consequently, we did not consider the instances on this part of the dataset in the training since it avoids model overfitting.

## 3 STUDY REPLICATION

In this study, we replicated a previous study [10] on machine learning to detect code smells using different and currently maintained systems from GitHub. Although modern systems deal better with code smells, they are still very common, especially in collaborative environments like Github. We expanded the replication to include a study on the impact of varying the features and resampling techniques used on the performance of models for detecting code smells. For the rest of this paper, we addressed Cruz et al.'s manuscript [10] as *base study*. Table 1 summarizes the study design of both experiments. We highlighted different aspects of both studies in italics.

**Table 1: Experiments Comparison.**

| Category | Base study | Current |
|---|---|---|
| Projects | *20*, Qualita Corpus[46] | *30*, GitHub |
| Datasets | *35,600 (classes)* *263,211 (methods)* | *50,765 (classes)* *295,832 (methods)* |
| Detection Tools | JDeodorant, JSpirit, Organic, PMD, *DECOR* | JDeodorant, JSpirit, Organic, PMD, *Designite* |
| Algorithms | DT, RF, NB, LR, KNN, MLP, GBM | |
| Features Selection | *30 manual* | *22 manual, 5 auto* |
| Resample Data | *No* | *Yes* |
| Measures | F1 | F1, *ROC-AUC* |
| Models Selection | Randomized Search | |
| Feature Engineering | None | *Polynomial Features* |

Both studies evaluated the same seven machine learning algorithms (Section 2.2) to predict four code smells (Section 2.1), chosen based on their coverage in the literature [45]. The base study used data from 20 Java software projects from Qualitas Corpus [46, 47]. Meanwhile, our replication evaluated 30 top-stared Java projects from GitHub. We extracted 12 class and 10 method metrics using the CK Metrics tool[2]; those metrics were our features. In addition, we used five code smell detection tools to obtain information on which class/method contained the code smells analyzed. However, instead of the Decor tool [34] to detect the Refused Bequest and Long Method code smells, we used the Designite tool [42]. For the dataset construction, we used the same vote method as the base study: class/method has the $X$ code smell, whether at least two out of the three detection tools identified that code smell in the class/method. For the machine learning modeling, both studies followed five steps:

- **Data Separation**: we split the dataset into two parts. The first one compromises 80% of the whole data to use in the training of the models, while the second one uses the remaining 20% of data for testing the models generated from training data;
- **Data Analysis**: we ensure that only pertinent features are maintained, through correlation analysis, mitigating overfitting;
- **Models Parametrization**: the Random Search model selection technique was used during training to improve the models;
- **Models Comparison**: the F1 metric was used in both studies to determine the models with the best performance. Additionally, our study presented the AUC metric, a less restrictive metric than the F1 metric;
- **Test on Unseen Data**: the test data was given as input to the best models of each machine learning algorithm to evaluate the effectiveness in predicting each smell.

We also highlighted that both studies conducted feature selection and feature engineering, in which we excluded highly correlated features from the set of features since they can cause overfitting, removed missing data from our dataset, and normalized the metric values. However, we took a step further: we explored the impact of using polynomial features (for creating a feature matrix with polynomial combinations up to a specified degree) and different resampling techniques, such as oversample and undersample. We present more details in Section 4.

## 4 STUDY DESIGN

### 4.1 Research Questions

This study has two main goals. The first one is to complement and check if the base study results apply to other projects from GitHub, reflecting current trends in the collaborative developer community. The second one is to explore how the resampling, the feature selection, and the polynomial features impact on the performance of code smell identification. The following research questions guide our study:

[2]https://github.com/mauricioaniche/ck

- **RQ1.** Which machine learning algorithm performs better using resample data, feature selection, and polynomial features techniques for popular GitHub projects?
- **RQ2.** To what extent can we use the mentioned techniques to improve machine learning model training for popular GitHub projects?

RQ1 aims to understand what algorithms for predicting code smells perform best. Unlike the base study, we did the same task with a different dataset. However, we tried to replicate the base study design, but the authors did not make all design decisions clear on the base study and its supplementary material. Consequently, we filled these gaps in our study design based on our knowledge.

Several techniques and parametrizations can directly impact the machine learning algorithm's performance when identifying code smells. Thus, to answer RQ2, we evaluated each of those techniques individually to understand how their variation can positively/negatively impact the performance of code smell predictions.

### 4.2 GitHub Dataset

First, we have built a dataset from Java software projects, selecting those currently maintained by the open-source GitHub community. The criteria for choosing the software systems were: i) at least 80% of its code is in Java; ii) the last update was in 2021 or later; iii) broad recognition by the programming community (we selected software projects with at least 1,000 stars in GitHub) [4]. Thus, we trained our models with non-trivial software systems that uses newer technologies, such as Continuous Integration/Continuous Delivery (CI/CD) and gatekeepers.

Table 2 presents the information about the 30 selected software projects. The first column presents the software system name and the evaluated version. The second column shows the number of classes of the software projects. The third and fourth columns exhibit the number of two class code smells (God Class - GC and Refused Bequest - RB, respectively). The fifth column shows the number of methods of the software projects. The sixth and seventh columns present the number of two method code smells (Feature Envy -FE and Long Method - LM, respectively). Details about our ground truth creation are present in Section 4.4. The last line of Table 2 presents the total values from the second to seventh columns, and values inside parentheses represent the proportion of smells in relation to all elements. It is important to notice that the dataset has a large imbalance, something common in other code smell prediction studies [1, 10, 40].

### 4.3 Model Features

We used software metrics for this study as our data input, *i.e.*, *features* for the machine models. We selected them considering aspects of software quality, such as coupling, cohesion, complexity, and data abstraction [32]. We obtained those features using the CK Metrics tool, like the base study. The base study authors discussed which features they used in the study. In our study, we did the same: we decided together what features could represent the aspects for evaluation. First, we discussed which features can represent software quality aspects related to the chosen code smells. Next, we performed a correlation analysis of the features using Spearman's

Table 2: Open-source Java projects statistics

| *Projects* | *Classes* | *GC* | *RB* | *Methods* | *FE* | *LM* |
|---|---|---|---|---|---|---|
| Checkstyle 9.2 | 1,295 | 14 | 0 | 4,797 | 80 | 2 |
| CoreNLP 4.3.2 | 4,295 | 310 | 22 | 33,718 | 673 | 202 |
| Dbeaver 21.0.2 | 6,511 | 101 | 15 | 37,108 | 223 | 440 |
| ElasticSearch Analysis 7.14.0 | 28 | 2 | 0 | 204 | 5 | 2 |
| FastJson 1.2.76 | 6,328 | 31 | 2 | 21,491 | 247 | 90 |
| Gson (no version) | 816 | 9 | 0 | 2,553 | 67 | 11 |
| Guava 30.1.1 | 6,477 | 146 | 38 | 29,480 | 69 | 18 |
| Hikari 4.0.3 | 159 | 3 | 0 | 1,307 | 49 | 0 |
| Java Faker 1.0.2 | 224 | 1 | 1 | 1,383 | 0 | 0 |
| Jedis 3.7.0 | 903 | 10 | 3 | 7,291 | 128 | 9 |
| Jenkins 2.287 | 3,898 | 55 | 19 | 17.887 | 125 | 27 |
| Jitwatch 1.3.0 | 592 | 23 | 1 | 7,728 | 76 | 55 |
| Jsoup 1.14.2 | 326 | 15 | 1 | 2,611 | 89 | 0 |
| Junit 4.13.2 | 1,474 | 7 | 2 | 4,307 | 28 | 0 |
| Libgdx 1.9.14 | 1,776 | 136 | 81 | 31,613 | 294 | 94 |
| Mapstruct 1.4.2 | 3,857 | 21 | 1 | 13,798 | 134 | 1 |
| Mockserver 5.11.2 | 995 | 25 | 4 | 7,227 | 342 | 13 |
| Mybatis 3.5.6 | 1,540 | 18 | 2 | 7,533 | 138 | 8 |
| NanoHttpd 2.3.1 | 145 | 6 | 0 | 715 | 8 | 1 |
| Netty 1.7.18 | 157 | 3 | 0 | 758 | 10 | 0 |
| Redisson 3.15.3 | 2,179 | 64 | 0 | 17,202 | 295 | 36 |
| Retrofit 1.6.0 | 290 | 1 | 0 | 955 | 6 | 10 |
| Shenyu 2.4.1 | 1,234 | 2 | 0 | 6,927 | 219 | 0 |
| Shopizer 2.17.0 | 1,325 | 23 | 7 | 8,062 | 47 | 45 |
| Spark 2.9.3 | 222 | 2 | 0 | 1,369 | 15 | 2 |
| Vert.x 4.1.2 | 1,248 | 72 | 6 | 13,593 | 447 | 24 |
| Webmagic 0.7.3 | 307 | 3 | 1 | 1,137 | 48 | 4 |
| XDM 7.2.11 | 306 | 23 | 1 | 1,961 | 51 | 30 |
| YsoSerial 0.0.5 | 121 | 0 | 0 | 366 | 4 | 0 |
| Zookeeper 3.6.3 | 1,737 | 49 | 2 | 10,751 | 223 | 50 |
| | | | | | | |
| All Projects | Classes | GC | RB | Methods | LM | FE |
| **Total** | **50,765** | **1175** (2.31%) | **209** (0.41%) | **295,832** | **4,140** (1.39%) | **1,174** (0.39%) |

Table 3: Classes Software Metrics

| *Metrics* | *Description* |
|---|---|
| DIT | Depth Inheritance Tree counts the number of *fathers* a class has. |
| FANIN | Counts the number of input dependencies a class has. |
| FANOUT | Counts the number of output dependencies a class has. |
| LCC | Loose Class Cohesion includes the number of indirect connections between visible classes for the cohesion calculation. Low values of cohesion are bad. |
| ILCOM | Improved Lack of Cohesion in Methods measures the number of connected components in a class. |
| LOC | Lines of Code counts the number of lines in the class. |
| NOC | Number of Children counts how many classes directly inherits from this class. |
| RFC | Response for a Class counts the total number of methods and the number of methods which can be invoked by them. |
| ICQ | Count the Inner Classes Quantity in a class. |
| TFQ | Count the Total Fields Quantity in a class. |
| TMQ | Count the Total Methods Quantity in a class. |
| WMC | Weighted Method Count computes the weighted sum of the methods implemented in a class, considering the weight as the cyclomatic complexity [33] of the method. |

Table 4: Methods Software Metrics

| *Metrics* | *Description* |
|---|---|
| FANIN | Counts the number of input dependencies a method has. |
| FANOUT | Counts the number of output dependencies a method has. |
| WMC | Weighted Method Count computed in this case for only the own method, it counts the cyclomatic complexity of itself. |
| LOC | Lines of Code counts the number of lines in a method. |
| RQ | Quantity of returns in a method. |
| VQ | Quantity of Variables counts how many variables are declared in a method. |
| PQ | Size of Parameter List computes the number of parameters in a method. |
| LQ | Quantity of Loops counts how many loops a method implements. |
| CQ | Quantity of Anonymous Classes in a method. |
| ICQ | Count the Inner Classes Quantity in a method. |

correlation rank [26], like the base study, and, in the end, evaluated all correlations higher than 0.8. Feature correlation analysis helps prevent overfitting by identifying and removing redundant or highly correlated variables, reducing model complexity. In the end, we considered 49 class and 29 method metrics. After discussions and removing highly correlated features, we obtained a subset of 12 class and 10 method features. The selected features and brief class and method metrics descriptions are presented in Tables 3 and 4, respectively.

## 4.4 Ground Truth Creation

After extracting and selecting features, we identified which classes and methods present the analyzed code smells. In the literature, different methods exist to detect those code smells. However, no consensus exists on which metrics are best for each code smell, nor what thresholds should be used [25, 43]. Our study used a voting strategy to classify each code smell considered the following five distinct tools to detect code smells: Designite [42], JDeodorant [15], JSpirit [48], PMD [38], and Organic [36]. Our voting strategy applied three different detection techniques for each smell for each class and method.

Code smell detection tools have limitations, as each detects a limited set of smells. We guaranteed that for each evaluated code smell, three tools could detect it. If two (or more) tools detected the code smell in the class/method, then we added that class/method as a true positive in our dataset. We manually validated a sample of the automatically classified smells and verified a precision rate above 80%. Therefore, this voting strategy is reasonable for our purposes.

Table 5 shows the relationships between the tool and which code smells they detect. Each column is associated with one analyzed code smells, and each line represents one of the five tools used in the experiments. We use the "✓" symbol to represent that the tool was used to detect the code smells.

## 4.5 Machine Learning Study Design

We focused our analysis on two performance metrics: F1 and AUC. The Precision metric penalizes models with a large number of false positives. The Recall metric penalizes false negatives. The F1 metric is a harmonic mean of the Precision and Recall metrics and is a way to penalize false positives and false negatives equally. That is, the code smells prediction problem, the F1 metric equally penalizes

**Table 5: Detection Tools for Our Voting Strategy**

| *Tools* | *GC* | *RF* | *FE* | *LM* |
|---|---|---|---|---|
| PMD | ✓ | | | |
| JDeodorant | ✓ | ✓ | ✓ | |
| JSpirit | ✓ | | ✓ | ✓ |
| Organic | | ✓ | ✓ | ✓ |
| Designite | | ✓ | | ✓ |

(a) smelly classes/methods not identified by the models and (b) not smelly classes/methods identified as smelly. The AUC metric plots a curve on a two-dimensional graph, where the X-axis represents the proportion of false positive results, and the Y-axis plots the proportion of true positives. The larger the area under this curve, the greater the model can correctly distinguish true positives and false positives. In the case of code smell predictions, the AUC metric measures the ability of the models to correctly distinguish smelly classes/methods from those incorrectly identified as smelly ones. Based on the results of those metrics, we can rank which algorithms achieved the best performance. However, we highlighted we have also calculated the Accuracy, Recall, and Precision metrics (available in the study documentation[3]).

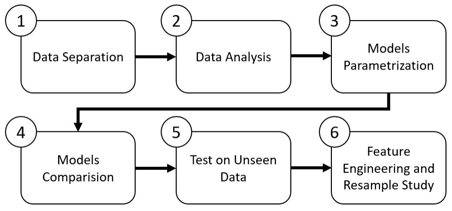

**Figure 1: Experiment Execution**

Figure 1 presents the study execution steps: (1) Data Separation, (2) Data Analysis, (3) Models Parametrization, (4) Models Comparision, (5) Test on Unseen Data, and (6) Feature Engineering and Resample Study. Our study and the base study share Steps 1 to 5, and we added Step 6 to our study to evaluate the impact of Feature Engineering and Resample techniques.

In the *Data Separation* step, we split the dataset into two parts (80% and 20% of the data, respectively). We used the bigger part to train the machine learning algorithms and the smaller to evaluate the effectiveness of the model trained on unseen data. The *Data Analysis* step assures that only the relevant feature remains: highly correlated data can bias the training [13]. We used Spearman's correlation algorithm to eliminate similar features because it is a nonparametric option. After, we removed instances that had missing/invalid values. Finally, for more efficient training, we normalized all feature values.

The model performance can vary depending on the dataset used and the values of the algorithm parameters, known as hyperparameters. We can test multiple combinations of hyperparameters to obtain the best results (the *Models Parametrization* step). However, manually testing all combinations is costly; thus, we used the RandomizedSearch algorithm (similar to the base study) from the Scikit-Learn[4] library in Python. The algorithm tries to optimize hyperparameters permutations by choosing samples randomly, generating several models for each permutation of hyperparameters, and listing their performance, which allows us to identify the best model for each situation. Since the base study authors did not clarify the values used for the search, we empirically tested several parameters for RandomizedSearch.

Another detail to highlight is all trained models used the techniques of resampling data, feature selection, and polynomial features. In the *Models Parametrization step* (Section 5.1), one of the authors applied feature engineering and resample data techniques with fixed values based on trial and error (we tested upper and lower extremes using a decision tree). We selected the feature using the SelectKBest[5] class from Scikit-Learn, which selects features according to the $k$ highest scores (we fixed the $k$ parameter as 5). We used the PolynomialFeatures[6] class with a polynomial of degree 2 for the polynomial feature techniques. We used the SelectKBest class independently and in conjunction with the PolynomialFeatures class. In the second case, we created the polynomial features after automatically selecting the features. We used the SMOTE[7] and RandomUnderSampler[8] classes for the resampling technique because they have a strategy parameter that defines a *"desired ratio of the number of samples in the minority class over the number of samples in the majority class after resampling"*. We defined the value for the *strategy* parameter for both cases as 0.2. At this stage of the experiment, the goal was not only to choose the correct parameters but also to identify good candidate algorithms for correctly detecting code smells so that we could later investigate the impact of different feature engineering and resampling techniques on the best models; consequently, we tried to enhance the performance of the best models.

To train our models, we used the cross-validation technique, which splits the training data into parts, and we performed the algorithm training according to the number of pre-defined cuts. In our case, we split into ten parts ($k$ = 10). To evaluate the hyperparametrization performance in the *Models Comparison* step, we selected the parameters that lead to the best F1 metric values. After obtaining the optimal models for the seven evaluated algorithms, we tested the optimal models against the Unseen Data. The F1 and AUC metrics were the output of that step. This step predicts how the models behave when we provide new instances to classify the model.

The *Evaluation on Test Data* step output is input to the *Feature Engineering and Resample Study* step. In that step, we chose machine learning algorithms with the best performance for code smell prediction. We aimed to explore how different variations of feature quantities, polynomial features, and resample data can impact the performance of the model. We used the SelectKBest class from

[3]https://github.com/labsoft-ufmg/ml_predictions_replication_2024

[4]https://scikit-learn.org/stable/
[5]https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html
[6]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html
[7]https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html
[8]https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html

Scikit-Learn for the automatic selection of features. For the feature engineering experiment, our $k$ parameter ranged from 1 to the total number of features for each dataset, i.e., 12 for the class and 10 for the method. We carried out the feature selection experiment in two ways. First, we selected only the $k$ features with the highest scores per interaction. Second, we made the same selection of the first step and next applied the polynomial features function. We chose to do it this way, considering that the features are first collected realistically, and then new features are created based on existing data. We used both resample techniques (oversample and undersample) to evaluate their impact on balancing the dataset. The *strategy* value ranges from 0, where the majority data remains untouched during undersampling or synthetic minority data is not generated during oversampling, to 1, where the undersampling entirely removes majority data, or the oversampling equalizes minority data to the majority.

## 5 RESULTS

This section reports the performance of the models in classifying the unseen instances on our test data. Section 5.1 presents the replication of the base study [10]. We called our replication of the baseline model experiment since we compared them with further model enhancements. Sections 5.2 and 5.3 report the experiments to evaluate the variation of feature engineering and resample techniques. We chose the best models found in Section 5.1 for those experiments, where the parameters of those classes varied to evaluate their impact on predicting code smells. Due to space limitations, we only discussed the relevant results in this paper. The source codes used in the experiments and the tabulated results are available in the experiment documentation[9].

### 5.1 Baseline Model Experiment

The baseline model experiment used feature engineering and resample data techniques with fixed parameter values to identify algorithms with better code smell prediction performance. Figure 2 presents the F1 and AUC metrics performance for the best models obtained for each of the seven algorithms for each code smell evaluated. The X-axis is the algorithm initials, and the Y-axis represents the values obtained for each performance metric: the blue bar represents the F1 metric, and the orange bar represents the AUC metric. In both cases, the closer to 1.0, the better the prediction capacity. However, it is important to note that the minimum value (the worst case) starts at 0.5 for the AUC metric, while the minimum value of the F1 metric is 0.0.

Table 6 presents the results of the three best algorithms for each code smell for the F1 and AUC metrics. The first column lists the code smells. The second and third columns list the algorithms that generated the models with the best F1 and AUC metrics results, respectively. The fourth column presents the algorithm chosen for experiments with feature engineering and resample data, in the next step. The complete results are accessible in the experiment documentation.

The God Class code smell had the best results in this experiment. Regarding the F1 metric, the RF and GBM algorithms generated the best models with a performance of 0.72 and the DT algorithm

[9]https://github.com/labsoft-ufmg/ml_predictions_replication_2024

**Table 6: Base Line Experiment - Best Results**

| *Code Smells* | *Highest F1* | *Highest AUC* | *Chosen Algorithm* |
|---|---|---|---|
| GC | RF (.72), GBM (.72), DT (.71) | DT (.96), RF (.95), GBM (.95) | DT |
| RB | DT (.10), RF (.09), GBM (.09) | DT (.74), GBM (.70), NB (.64) | DT |
| FE | RF (.19), KNN (.16), NB (.11) | RF (.64), KNN (.57), NB (.57) | RF |
| LM | KNN (.41), RF (.33), LR (.33) | GBM (.90), DT (.88), NB (.88), LR (.88) | KNN |

with a performance of 0.71. Regarding the AUC metric, the DT algorithm obtained the best performance with 0.96, while the RF and GBM algorithms obtained a performance of 0.95. The NV and LR algorithms obtained good results for the AUC metric, all above 0.8. The best model generated by the MLP algorithm could not predict classes with the God Class code smell due to the imbalanced data in the dataset (2.31% according to Table 2).

Only 0.41% of classes in the dataset are affected by the Refused Bequest code smell (Table 2), indicating the need to explore further techniques that deal with imbalanced data. The models generated from the DT, GBM algorithms obtained one of the best performances for the F1 and AUC metrics. The RF algorithm obtained one of best performance for F1 metric (0.09) and NB (0.64) for the AUC metric. The worst case occurred with the MLP algorithm, in which the data imbalance did not allow the creation of a model to identify the classes with RB smell. Consequently, the F1 and AUC metrics values were, respectively, 0.0 and 0.5

The Feature Envy code smell affects only 0.39% of the methods (Table 2). The low number of smelly elements makes it difficult for the machine learning algorithms to learn about them. For this reason, the models generated by the DT, GBM, and MLP algorithms could not predict when a method has that code smell. In models generated by other algorithms, the results are still low. For the F1 metric, the best performance results were the RF (0.19), KNN (0.16), and NB (0.11) algorithms. For the AUC metric, the RF algorithm obtained a performance of 0.64, and the KNN and NB algorithms obtained a performance of 0.57.

The models generated by machine learning algorithms to predict the Long Method code smell obtained low values for the F1 metric. Still, for the AUC metric, the majority exceeded the score of 0.8. The KNN (0.41) algorithm obtained the highest performance for the F1 metric, followed by the RF and LR algorithms with a performance of 0.33. The GBM, DT, NB, and LR algorithms obtained the highest for the AUC metric, with the first at 0.9 and the others at 0.88. In addition to that, in the case of the AUC metric, other algorithms also obtained a good performance: RF (0.86) and KNN (0.83). Like the other code smells, the data imbalance did not allow the MLP algorithm to predict smelly elements.

As highlighted before, our dataset of the four code smells is imbalanced. However, in cases such as the RB and FE code smells, this imbalance is below the proportion of 1:100 and is considered a severe imbalance [24]. Compared with the GC and LM code smells,

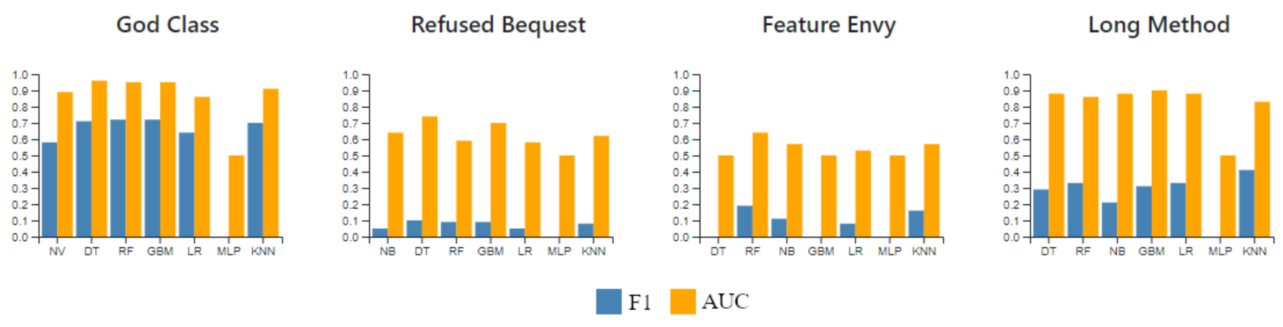

**Figure 2: Baseline Experiment - Results**

which have an imbalance above 1:100, we noticed that the performance for the RB and FE code smells is much lower than those of the other two smells. Another important point to highlight is that all algorithms linearly divide the data. The inability of the MLP algorithm to predict code smell indicates a possible variance problem in the datasets. Variance defines how spread out the data are [35]. The more spread out the data are in dimensions, the more difficult it is to draw lines that make up their divisions. This hypothesis can be reinforced by the performance of the DT, RF, and GBM algorithms, which can capture more complex patterns, especially in the last two algorithms that use the ensemble technique. Given the explained results, we summarize the answers to the first research questions.

**Answer of RQ1:** For classes with code smells, the algorithms with the best performance for the F1 and AUC metrics were DT, RF, and GBM. The exception in this case was the NB algorithm for the RB code smell, which obtained better results than the RF algorithm for the AUC metric. For GC smell, we chose the DT algorithm, with 0.71 for the F1 metric and 0.96 for the AUC metric, for experiments with feature engineering and resample techniques. We chose the DT algorithm for the RB code smell since its scores were 0.10 for the F1 metric and 0.74 for the AUC metric. For the FE code smell, the algorithms with the best performance for the F1 and AUC metrics were the RF, KNN, and NB algorithms. In this case, the RF algorithm obtained the best performance for both metrics (0.19 for the F1 metric and 0.64 for the AUC metric). Therefore, we chose it for the next experiment. For the LM code smell, the best-performing algorithms for the F1 metric were the KNN, RF, and LR. For the AUC metric, the best-performing algorithms were GBM, DT, and NB. For that smell, the AUC metric results were higher, and the F1 metric results were lower, so we chose the algorithm with the best F1 metric result, KNN (0.41).

## 5.2 Feature Engineering

Figure 3 shows the results of the Feature Engineering experiment. The first chart presents the F1 metric results for the $K$ feature selection technique variations. The second chart presents the F1 metric results for variations of the $K$ feature selection technique, followed by applying the polynomial features technique. The third and fourth charts are the same as previous charts one and two, respectively, but for the AUC metric. Each chart presents four symbols representing the code smells: the circle is the GC code smell, the triangle is the RB code smell, the square is the FE code smell, and the cross is the LM code smell. The Y-axis is the values for the F1 and AUC metrics. The X-axis is the number of selected features. Note that on the X-axis, for $k$ equal to 11 and 12, the square and cross symbols do not appear since our method dataset only has ten features.

**Table 7: Feature Engineering Experiment - Variations Results**

| *Code Smells* | *F1: FS* | *F1: FS + PF* | *AUC: FS* | *AUC: FS + PF* |
|---|---|---|---|---|
| GC | ↓ .34 ↑ .73 | ↓ .34 ↑ .74 | ↓ .61 ↑ .89 | ↓ .61 ↑ .90 |
| RB | ↓ .00 ↑ .17 | ↓ .50 ↑ .54 | N/A | N/A |
| FE | ↓ .00 ↑ .07 | ↓ .00 ↑ .08 | ↓ .50 ↑ .51 | ↓ .50 ↑ .52 |
| LM | ↓ .63 ↑ .64 | ↓ .62 ↑ .64 | ↓ .75 ↑ .76 | ↓ .74 ↑ .76 |

Table 7 presents the best and worst results of the feature engineering experiment. The first column lists the four code smells, the second column is the F1 metric results using only the feature selection technique, and the third column is the results using the feature selection technique, followed by polynomial features. The fourth and fifth columns are equivalent to the second and third columns but for the AUC metric. Arrows pointing down indicate the worst results, and those pointing up represent the best results. We represented the metrics that remained unchanged in results (0.0 for the F1 metric or 0.50 for the AUC metric) as "N/A". The complete results are accessible in the experiment documentation.

For the God Class code smell, five features were sufficient to stabilize performance for the F1 and AUC metrics, which reached values of 0.73 and 0.86, respectively. The five features were FANOUT, LCOM*, RFC, TMQ, and WMC. In the case where we selected two features (RFC and WMC) and then further applied the polynomial features technique, the F1 metric value was 0.69, and the AUC metric value was 0.89, showing that in this specific case, using the polynomial features technique allows us to get closer to the same result of using five features.

The Feature Envy code smell achieved a significant improvement in F1 metric results when eight features were selected (FANIN, FANOUT, WMC, LOC, RQ, VQ, PQ, LQ, CQ), ranging from 0.0 (only one feature) to 0.07. With the eight features above plus the use of polynomial features, there is a slight improvement of 0.08. We did not observe that improvement for the AUC metric: when we

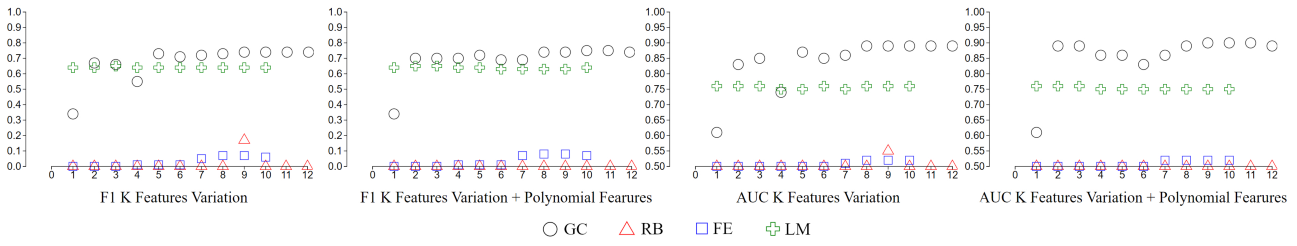

**Figure 3: Feature Engineering Results**

used all features, we obtained an improvement of 0.01. Although the Feature Envy code smell results are still very low, the increase in the number of features and the application of polynomial features had positive impacts on the predictive capacity, indicating the need for (i) collecting more features that can better represent the smell, and (ii) using polynomial features to deal with the data variance problem. Finally, using those techniques in the Refused Bequest and Long Method code smells did not significantly impact on performance of the results.

As evidenced by the results, varying the number of features used to generate predictive models can be used to refine the models in some cases. Regarding the GC and FE code smells, increasing the dimensionality of the models was beneficial to a certain extent; thus, the feature increase no longer has any effect from this point onwards. As for the RB and LM code smells, increasing the number of features did not significantly impact them, indicating that the features used in this study may not be the best to represent these smells; so, exploring new features to predict them can be more interesting.

## 5.3 Resample Data

Figure 4 presents the results of varying the *strategy* parameter for the resample data techniques. The first chart presents the F1 metric results using the oversample technique, and the second chart the F1 metric results using the undersample technique. The third and fourth charts present the oversample and undersample results, respectively, for the AUC metric. Each chart presents four symbols representing the code smells: the circle is the GC code smell, the triangle is the RB code smell, the square is the FE code smell, and the cross is the LM code smell. The Y-axis is the values for the F1 and AUC metrics. The X-axis is the value of the *strategy* parameter.

**Table 8: Resample Data Experiment - Variations Results**

| *Code Smells* | *F1: Oversample* | *AUC: Oversample* | *F1: Undersample* | *AUC: Undersample* |
|---|---|---|---|---|
| GC | ↓ .72 ↑ .76 | ↓ .89 ↑ .96 | ↓ .38 ↑ .71 | ↓ .86 ↑ .97 |
| RB | ↓ .00 ↑ .12 | ↓ .50 ↑ .81 | ↓ .00 ↑ .09 | ↓ .50 ↑ .87 |
| FE | ↓ .00 ↑ .23 | ↓ .50 ↑ .73 | ↓ .00 ↑ .22 | ↓ .50 ↑ .85 |
| LM | ↓ .76 ↑ .78 | ↓ .88 ↑ .94 | ↓ .42 ↑ .72 | ↓ .80 ↑ .93 |

Table 8 presents the best and worst results of the resample data experiment. The first column lists the four code smells, and the second and third columns are the F1 and AUC metrics results for the oversample techniques, respectively. The fourth and fifth columns are the F1 and AUC metrics results for the undersample techniques, respectively. The values between columns two and five are considered only decimal places, with those to the right of the down arrow being the worst results and those to the right of the up arrow being the best results. The complete results are accessible in the experiment documentation.

On the one hand, for the smells with the lowest imbalance (GC - 2.31% and LM - 1.39%), for F1 score, the oversample results (GC: 0.76% and LM: 0.78%) were better than the undersample (GC:0.71% and LM: 0.72%). On the other hand, the code smells with the greatest imbalance (RB - 0.41% and FE - 0.39%) remained with low values for the F1 metric, both using oversample and undersample. It is possible to evaluate that the prediction performance regarding the AUC metric is good (improvement between 6% and 13% for over and undersample cases) in the cases of the GC and LM code smells and very good (improvement higher than 20% for over and undersample cases) in the cases of the RB and FE code smells.

As the results show, resample techniques can be a good solution to deal with the imbalance of code smells datasets, but using them carefully is important. For datasets with an imbalance greater than 1:100, the undersample technique performed better than the oversample in the case of the AUC metric and obtained a similar result for the F1 metric for the RB and FE code smells. The oversample techniques can be a better option for code smells with lower imbalance (GC and LM).

After the analysis of Sections 5.2 and 5.3, we can answer the second research question.

**Answer of RQ2:** Using feature selection, polynomial features, and resampling data techniques can help refine predictive models and identify possible changes in the dataset. In the case of selecting the number of features, we obtained improvements in cases such as the GC and FE code smells, allowing us to evaluate that only one feature was relevant in the prediction of the RB and LM code smells; thus, we need to explore new features if we want to improve the RB and LM models. Despite the positive impact for the GC and FE code smells, using polynomial features is not justified, as it improved in both cases by a percentage of just 0.01. Resample techniques were more promising for refining predictive models. In cases where the dataset imbalance exceeded 1:100, the undersample was a more promising solution (improvement higher than 30% in AUC score); in cases where it was below 1:100, the oversample seemed to be a better option for F1 score and both (over and undersample) are a good option for AUC score.

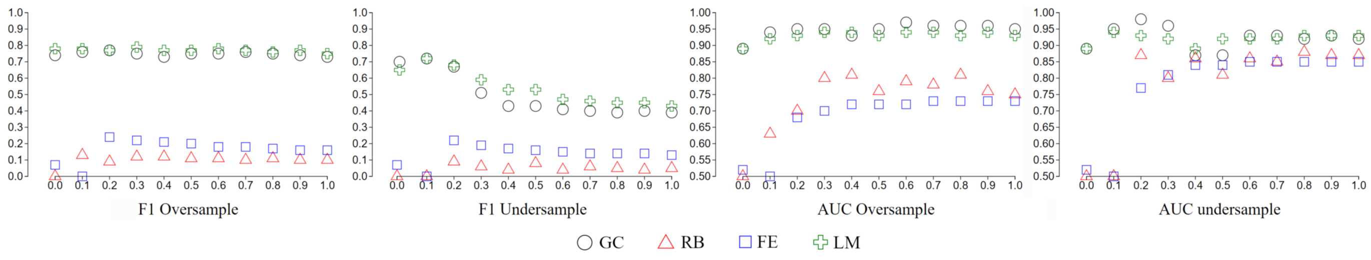

**Figure 4: Resample Data Results**

## 6 DISCUSSION

In this section, we compare the results of this experiment with the base study. An important aspect of both studies is the imbalance of the Qualitas Corpus and GitHub datasets. The code smells had the following proportions in the base study: GC: 4.77%, RB: 8.96%, FE: 3.46%, and LM: 0.87%. In the current study, that proportion is much lower: CG: 2.31%, RB: 0.41%, FE: 1.39%, and LM: 0.39%. We highlight that in both studies, the code smells with less imbalance in the datasets had the best results for prediction. In the case of the base study, they were the GC and RB code smells; in our study, they were the GC and LM code smells. Also, in both studies, the models generated by the RF and GBM algorithms are among the best predictors for most of the smells.

The main differences between the two studies are: (i) the F1 metric values were higher in the base study, and (ii) the best prediction performances in the base study were for the GC and RB code smells, while in our study were GC and LM code smells. The first difference is due to the imbalance in the base study dataset, which is smaller because the dataset for the base study is from 2010 [46], while the current study is from 2021. Older Java systems can have more code smells, as new development practices and tools, such as Linters, mitigate them. Regarding the second difference between the studies, we can justify it when we conclude that the smallest imbalances provided the best results, regardless of which smell it was.

One of the proposals for future work in the base study [10] was to improve the results of models that did not perform well. Our work explored feature engineering and resample data techniques as a way to improve prediction. This improvement was significant in some cases, especially for the AUC metric, like for the RB and FE code smells when they used resample data techniques. In other cases, for the F1 metric, although the techniques did not always achieve satisfactory results, it was still possible to notice improvements, where we checked for the RB and FE code smells when using resample data techniques. In summary, resample data techniques had a greater positive impact than feature engineering techniques. However, undersample techniques performed better for cases with higher imbalance, while oversample techniques were better for cases with less data imbalance. The feature selection technique could still provide valuable insights because, for the RB and LM code smells, only one metric was responsible for the models' performance, indicating the need to seek new ways of representing this smell. Finally, we highlight that very high values for the *strategy* parameter can harm the models.

The results of this study can be useful to different professionals in the following ways:

- **Researcher**: they indicate the need to research other techniques, in addition to undersample and oversample, to deal with data imbalance in code smells;
- **Developer**: they facilitate data-driven source code refactoring;
- **Software Architect**: they allow improving the results of fuzzy predictions to identify points for improvements in the architecture of software systems.

## 7 THREATS TO VALIDITY

Despite the careful design of our empirical study, some limitations may affect the validity of our results. This section discusses threats and our actions to mitigate them, organizing them by construct, internal, external, and conclusion validity [50].

**Construct Validity**. Construct validity concerns the mapping of the results of the study to the concept or theory [50]. For instance, we relied on only two metrics (F1 and AUC) to evaluate the models for code smell detection. That choice threatens construct validity since the metrics can not accurately measure the effectiveness of the machine learning classifiers. However, that threat does not invalidate our main findings since previous related studies use those metrics with similar purposes. Besides, we used only one implementation for each classifier, which may bias our results. However, those implementations were provided by libraries heavily used for machine learning, such as Scikit-Learn. Therefore, we relied on the best-known algorithms to perform the code smell detections. The construction of the ground truth is also a threat since it may include false positives and not identify all true positives. To mitigate it, we relied on the combination of five tools for the ground truth creation.

**Internal Validity**. Threats to internal validity are influences that can affect the independent variable due to causality [50]. In our context, this threat refers to the characteristics of the chosen datasets. For instance, since we selected open-source projects from GitHub, the project domains, involved technologies, and experience of the developers are some confounding factors. Our dataset comprises 30 software systems from different domains to minimize that threat. Although the constraints of the tools used (for instance, Java-based technology) exist, we have tried to analyze and understand the results and mitigate confounding factors.

**External Validity**. Threats to external validity are conditions that limit our ability to generalize the results of our paper [50]. For instance, the 30 systems we evaluated can only represent part of the space of open-source systems since they were all written in Java. Therefore, we cannot generalize our results to other projects, code smells, and technologies. It is important to highlight we only performed the detection in Java systems because not enough tools exist to create the ground truth for other languages. The study design is not limited to the programming language, performing in different datasets. Besides, the selected systems range from different domains and sizes, including large frameworks from industry. Finally, we explored five tools in depth to detect code smells. It would be important to assess how this might impact results.

**Conclusion Validity**. Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion between the treatment and the outcome [50]. Our study performed metrics used can have led us to false conclusions. The main reason for choosing the F1 and AUC metrics is to prioritize something other than precision or recall. Moreover, they are well-known metrics from machine learning evaluation and information retrieval. Finally, since not all information was clear enough to answer the research questions, cross-discussions among the paper authors often occurred to reach a common agreement about our main findings.

## 8 RELATED WORK

Identifying code smells manually in the projects is a time-consuming activity; as a consequence, several detection strategies were proposed in the literature [14], and they use different strategies, such as metric thresholds [38, 49], refactoring opportunities [15], and machine learning algorithms [2, 12, 13, 16, 29–31, 41]. Maiga et al. [31] used the Support Vector Machine model to identify four antipatterns in three Java projects. Later, Amorim et al. [2] investigated the precision of Decision Trees to detect four code smells in four Java projects. Differently from these works, we investigated the performance of seven models for four smells in 30 systems. Fontana et al. [16] used machine learning to identify four code smells on 74 Java systems. In total, they evaluated the performance of 6 algorithms, varying some of its parameters. However, Di Nucci et al. [13] replicated the study, filling some gaps, such as using techniques dealing with data imbalance. Even though our dataset comprises fewer systems, we highlight that the selected systems in our work reflect current practices on open-source development and new technologies. In contrast, previous works analyze the performance of systems collected in 2012 [13, 16]. We also experimented with different techniques that deal with imbalanced data and tuning of hyperparameters, and we have used different polynomial features. We also present our results with an additional measurement, the AUC metric.

More recently, Stefano et al. [12] evaluated if there was an impact on the model performance when analyzing multiple versions of a system versus using only one version. They did not find a statistically significant difference between their results. Lomio et al. [29] evaluated how rules from the SonarQube system can help improve fault prediction algorithms' performance. Santos et al. [41] used an ensemble model to identify how similar the models for defects are in comparison with seven code smells at the class level on a defect dataset with different versions of 14 systems. They have found a high performance for only three smells. Even though the focus of both works differs, we highlight our work to complement them since we also evaluate two smells at the method level. Madeyski et al. [30] evaluated the performance of seven algorithms to detect four code smells (Blob, Data Class, Long Method, and Feature Envy). The authors used a manually validated sample of classes from their industrial partners. Here, we bring the perspective of the performance of models on open-source development, and we evaluate two other code smells, the God Class and Refused Bequest code smells.

## 9 CONCLUSION AND FUTURE STUDIES

This study presents a differentiated replication of a base study [10]. We noticed that the more severe the imbalance in the dataset, the worse the model prediction performance. The undersample technique performed better for datasets with more severe imbalance, while both (over and undersample) performed good for datasets with less severe imbalance. The feature selection technique provided insights into collecting more information to represent the RB and LM code smells. The polynomial feature techniques had little impact on the performance of the predictions.

In future work, we intend to research new machine learning techniques that deal with data imbalance, in addition to undersample and oversample. We also want to explore new ways of representing code smells and the features presented in this study and expand the dataset, evaluating other systems. Finally, extending the work to other code smells and languages besides Java is also necessary.

## 10 ACKNOWLEDGEMENTS

This research was partially supported by Brazilian funding agencies: CNPq, CAPES, and FAPEMIG (Grant PPM-00651-17).

## REFERENCES

[1] 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.

[2] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. 2015. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *International Symposium on Software Reliability Engineering (ISSRE)*. 261–269.

[3] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[4] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[5] L. Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] L. Breiman. 2017. *Classification and regression trees*. Routledge.

[7] Aloisio Cairo, Glauco Carneiro, Antonio Resende, and Fernando Brito E Abreu. 2019. The Influence of God Class and Long Method in the Occurrence of Bugs in Two Open Source Software Projects: An Exploratory Study (S). In *International Conferences on Software Engineering and Knowledge Engineering*. KSI Research Inc. and Knowledge Systems Institute Graduate School. https://doi.org/10.18293/seke2019-084

[8] T. Chen and C. Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Int'l Conf. on knowledge discovery and data mining (KDD)*. ACM, 785–794.

[9] T. M. Cover, P. E. Hart, et al. 1967. Nearest neighbor pattern classification. *Transactions on Information Theory* 13, 1 (1967), 21–27.

[10] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. 2020. Detecting bad smells with machine learning algorithms: an empirical study. In *Proceedings of the 3rd International Conference on Technical Debt*. 31–40.

[11] Phongphan Danphitsanuphan and Thanitta Suwantada. 2012. Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. In *2012 Spring Congress on Engineering and Technology*. IEEE. https://doi.org/10.1109/scet.2012.6342082

[12] Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. 2021. Comparing Within- and Cross-Project Machine Learning Algorithms

for Code Smell Detection. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution* (Athens, Greece) *(MaLTESQuE 2021)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3472674.3473978

[13] 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 *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 612–621.

[14] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. 2016. A Review-Based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16)*. Article 18, 12 pages.

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1037–1039.

[16] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. In *Empirical Software Engineering (EMSE)*.

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

[18] Jiri Gesi, Xinyun Shen, Yunfan Geng, Qihong Chen, and Iftekhar Ahmed. 2023. Leveraging Feature Bias for Scalable Misprediction Explanation of Machine Learning Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*.

[19] Mitja Gradisnik, Tina Beranic, Saso Karakatic, and Goran Mausas. 2019. Adapting God Class thresholds for software defect prediction: A case study. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. https://doi.org/10.23919/mipro.2019.8757009

[20] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

[21] Marcel Jerzyk and Lech Madeyski. 2023. Code Smells: A Comprehensive Online Catalog and Taxonomy. In *Studies in Systems, Decision and Control*. Springer Nature Switzerland, 543–576. https://doi.org/10.1007/978-3-031-25695-0_24

[22] Nasraldeen Alnor Adam Khleel and Károly Nehéz. 2023. Detection of code smells using machine learning techniques combined with data-balancing methods. *International Journal of Advances in Intelligent Informatics* 9, 3 (2023), 402–417.

[23] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein. 2002. *Logistic regression*. Springer.

[24] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5, 4 (2016), 221–232.

[25] 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.

[26] David Lane, David Scott, Mikki Hebl, Rudy Guerra, Dan Osherson, and Heidi Zimmer. 2003. *Introduction to statistics*. Citeseer.

[27] M. Lanza, R. Marinescu, and S. Ducasse. 2005. *Object-Oriented Metrics in Practice*. Springer-Verlag.

[28] D. D. Lewis. 1998. Naive (Bayes) at forty: The independence assumption in information retrieval. In *European conference on machine learning (ECML)*. Springer, 4–15.

[29] Francesco Lomio, Sergio Moreschini, and Valentina Lenarduzzi. 2022. A Machine and Deep Learning analysis among SonarQube rules, Product, and Process Metrics for Faults Prediction. *Empirical Software Engineering* 27 (10 2022). https://doi.org/10.1007/s10664-022-10164-z

[30] Lech Madeyski and Tomasz Lewowski. 2023. Detecting code smells using industry-relevant data. *Information and Software Technology* 155 (2023), 107112. https://doi.org/10.1016/j.infsof.2022.107112

[31] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur. 2012. Support vector machines for anti-pattern detection. In *Proceedings of Int'l Conf. on Automated Software Engineering (ASE)*. 278–281.

[32] Radu Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 701–704.

[33] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[34] Naouel Moha and Yann-Gael Guéhéneuc. 2007. Decor: a tool for the detection of design defects. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 527–528.

[35] Glenn J Myatt. 2007. *Making sense of data: a practical guide to exploratory data analysis and data mining*. John Wiley & Sons.

[36] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Alessandro Garcia, Anne Benedicte Agbachi, Roberto Oliveira, and Carlos Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society* 24, 1 (2018), 1–30.

[37] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[38] Thanis Paiva$^1$, Amanda Damasceno$^1$, Juliana Padilha$^1$, Eduardo Figueiredo$^1$, and Claudio Sant'Anna. 2015. Experimental evaluation of code smell detection tools. (2015).

[39] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 268–278.

[40] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2019. On the role of data balancing for machine learning-based code smell detection. In *Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*. 19–24.

[41] 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*, Leen Lambers and Sebastián Uchitel (Eds.). Springer Nature Switzerland, Cham, 282–305.

[42] 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.

[43] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158–173.

[44] Satwinder Singh and K. S. Kahlon. 2012. Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. *ACM SIGSOFT Software Engineering Notes* 37, 2 (apr 2012), 1–11. https://doi.org/10.1145/2108144.2108157

[45] E. Sobrinho, A. De Lucia, and M. Maia. 2021. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Trans. on Software Engineering* 47, 1 (2021), 17–66.

[46] 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.

[47] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S Bigonha. 2013. Qualitas. class Corpus: A compiled version of the Qualitas Corpus. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 1–4.

[48] 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, 1–6.

[49] 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, 1–6.

[50] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[51] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. https://doi.org/10.1145/1985362.1985366

[52] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2013. Comparing four approaches for technical debt identification. *Software Quality Journal* 22, 3 (apr 2013), 403–426. https://doi.org/10.1007/s11219-013-9200-8