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 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 currently maintained 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 obtained 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 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. 2024. Tuning Code Smell Prediction Models: A Replication Study. In *Proceedings of 46th International Conference on Program Comprehension (ICPC 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICPC 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code smells are code structures that can compromise the internal quality of a system [17]. Between 1990 and 2017, at least 351 studies were conducted on *code smells*, with the detection of their structures being the most prevalent focus, representing 30% of the total [45]. These studies presented approximately 80 different tools and techniques to identify *code smells* [45]. That is, many ways to detect smells in source code have been proposed, most of them based on software metrics and thresholds. However, to use thresholds, developers have to adjust them according to their context and system size. One way to get around this limitation is the use of machine learning techniques to detect code smells.

However, to use machine learning, most models need a ground truth. That is, to know beforehand which instances contain code smells. Consequently, building datasets is not a trivial task, and require substantial effort [43]. Another problem raised by the use of machine learning modeling is that, usually, code smell datasets are very imbalanced [1, 10, 40], it has more true negatives than positives. Several studies used machine learning techniques to deal with imbalanced datasets [18, 22], but to our knowledge, none of these studies tried to understand how different techniques with different parametrizations may impact the model performance of code smells predictions. Furthermore, most of the studies in the literature only use linear features [10] and consequently, polynomial features should be further explored.

This paper replicates a basis study [10] with the aim of obtaining new insights into predicting code smells using machine learning. However, we further expand the analysis by evaluating the impact of feature engineering and resample data techniques on the machine learning model performance. For this task, a dataset of 30 Java software systems was collected from Github. Four code smells were evaluated: God Class, Refused Bequest, Feature Envy and Long Method. In order to build our ground truth, we have used five code smell detection tools.

The results highlight that by reducing unbalance rates, the models achieve better prediction capacity. 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, while Refused Bequest and Feature Envy were worse. Undersample technique was better for cases with higher imbalance, while oversample techniques performed better for cases with less data imbalance. Some cases, such as Refused Bequest and Long Method, did not experience any performance gains when adding more features during training. In these cases feature selection indicated valuable insights, suggesting the need for new ways of representing certain code smells.

The main contributions of this study were:

- Provide a public dataset with 50k classes and 295k methods instances, with ground truth for four types of bad smell.
- A protocol for tuning machine learning models.
- Insights about feature selection, polynomial features and resample data.

The rest of this paper is structured as follows. Section 2 presents important concepts for understanding the study. Section 3 discusses the basis study that we replicated. Section 4 shows how the experiments were performed. Section 5 presents the results. Section 6 discusses the findings of the study and how they can be useful for different actions. Section 7 shows threats to validity. Section 8 discusses related works. Finally, Section 9 includes the conclusion and ideas for future studies.

2 BACKGROUND

2.1 Code Smells

Code Smell is an indication that usually, but not always, corresponds to a deeper problem in the system's design or code structure [17]. They contribute directly to technical debt if they are overlooked and left unaddressed. Successful long-term project must avoid them. Code smells are often detected using rules hard-coded in detection tools [21]. Thus, code smells are code snippet (or several snippets) that normally corresponds 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 a refactoring [30]. Code smells have been widely studied and associated with the presence of faults and software quality issues [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. A bad case of this smell exists when the class refuses to implement an interface [10]. The Feature Envy code smell indicates, for instances, 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 includes a lot of data and it is complex[17].

2.2 Machine Learning

Several techniques have been proposed to overcome the limitations to evaluating the presence of code smells, such as the combination of software metrics [27, 36, 48], historical data [39], refactoring opportunities [15], and machine learning models [2, 10, 13, 41]. Machine learning approaches eliminate constraints on relying solely on deterministic and heuristic solutions with explicitly implemented algorithms and thresholds. In our study, we aim to explore and evaluate the performance of seven machine learning algorithms, considering different resampling techniques, the use of 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 smell 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 model the learning: supervised and unsupervised. Supervised learning indicates that the model is trained based on the classifications of the classes/methods, *i.e.*, we need a ground truth containing if the class/method presents the smell under consideration to make model adjustments. 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 algorithms also 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 separate our data into training data 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 their training. Consequently, we did not consider the instances on this part of the dataset in the training, and it avoids model overfitting.

3 STUDY REPLICATION

In this study, we tried to closely replicate one previous study on machine learning to predicting code smells [10] but using a different and a more current dataset extracted from GitHub. However, we further expanded our analysis to include a study on the variation of feature engineering and resampling techniques and how they impact the model's ability to identify the code smells correctly. For the rest of this paper, we will address Cruz et al.'s manuscript [10] as a basis study. Table 1 summarizes the study design differences between the basis study and ours. We highlighted different aspects of both studies in italics.

Both studies proposed to evaluate seven machine learning algorithms to predict four code smells (Feature Envy, God Class, Long Method, and Refused Bequest). Both use the same machine learning algorithms, described in Section 2.2. The basis study used data from 20 Java software available in Qualita Corpus [46, 47]. Meanwhile, our replication evaluated 30 top-stared Java projects from GitHub, currently maintained by the community. We extracted 12 class and 10 method metrics using the CK Metrics[1] tool for the base study. These software metrics comprised the features of the dataset used in this study. In addition, we used five code smell detection tools to obtain information on which class/method contained the code smells analyzed. However, since Decor [34] does not support current Java versions, we did not use it to detect the Refused Bequest and Long Method code smells, as the basis study does. However, we used the results provided by the Designite [42]. For the dataset construction, we used the same votation method as the basis study: class/method has the X code smell whether at least two of the three

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

Table 1: Experiments Comparison.

Category	Basis 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*

detection tools identified that code smell in the class/method. For each smell, both studies had five stages:

- **Data Separation**: we split the total dataset into two parts. The first one the size of 80% of the total for training the data, and the second one of the remaining 20% for testing models generated from training data;
- **Data Analysis**: we evaluated training results and features to identify and mitigate possible causes of overfitting;
- **Models Parametrization**: the Random Search model selection technique was used during training to generated the models;
- **Models Comparison**: F1 metric was used in both studies to determine the models with the best results. Our study also considered the AUC metric since it is less restrictive 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 highlight 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 (technique that creates a feature matrix with polynomial combinations up to a specified degree) and different resampling techniques, such as oversample and undersample. We presented more details in Section 4.

4 STUDY DESIGN

4.1 Research Questions

This study has two main goals: (i) to complement and check if the basis study results apply to a newer dataset from popular GitHub projects, that reflect current trends in the collaborative developer community and (ii) to explore how resample data, feature selection, and polynomial features can impact on code smell predictions in this dataset. The following research questions guide our study:

- **RQ1.** Which machine learning algorithms perform best 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?

The mentioned work in Section 3 inspired RQ1, which aimed to understand the best algorithms for predicting code smells and their efficiency. Unlike the basis study, we carried out the same task with a different dataset and used the mentioned machine learning techniques in the current study.

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

4.2 GitHub Dataset

We first built a dataset from Java software projects to achieve our goals, 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 the models with non-trivial software systems that uses newer technologies, such as Continuous Integration/Continuous Delivery (CI/CD) and gatekeepers.

Table 2 presents the complete list of selected software projects (total = 30 systems). The first column presents the software systems 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 code smells related to classes (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 code smells related to methods (Feature Envy -FE and Long Method - LM, respectively). The last line of the Table 2 presents the total values from the second to seventh columns. Still in the last line, in columns three, four, six and seven, the values in parentheses represent the proportion of smells in relation to all elements. This is an important detail, to understand that the dataset has a large imbalance, something common even in other code smell prediction studies [1, 10, 40].

4.3 Model Features

We used software metrics for this study as our input, *i.e. features* for the machine models. We selected features considering aspects of software quality, such as coupling, cohesion, complexity, and data abstraction [32]. We obtained our features using the CK Metrics tool, which calculates 49 class and 29 method metrics. We first discussed which features are able to represent software quality aspects related

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 2287	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
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.

to the chosen smells. Next, we performed a correlation analysis of the features using Spearman's correlation rank [26] and, at the end, evaluated all correlations higher than 0.8. This step is necessary to avoid collinearities and avoid overfitting. After discussions and removing highly correlated features, we obtained a subset of 12 class and 10 method features. The selected features and brief descriptions of the class and method metrics are presented in Tables 3 and 4, respectively.

4.4 Ground Truth Creation

After extracting and selecting features, we identified which classes and methods the evaluated code smells affect. In the literature, different methods exist to detect those code smells; the most common are using object-oriented software metrics and thresholds [43]. However, there neither exists a consensus on which metrics are best for each code smell nor what thresholds should be used [25][43]. Our study used a compromise 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 one limited set of code smells. We guaranteed that for each evaluated code smell, three tools could detect it. If two tools (or more) 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 we concluded that this voting strategy is reasonable for our purposes.

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

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 (a)

Table 5: Detection Tools for Our Voting Strategy

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

smelly classes/methods not identified by the models and (b) not smelly classes/methods identified as smelly.

Meanwhile, 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 highlight we have also calculated the accuracy, recall, and precision metrics (available in study documentation[2]).

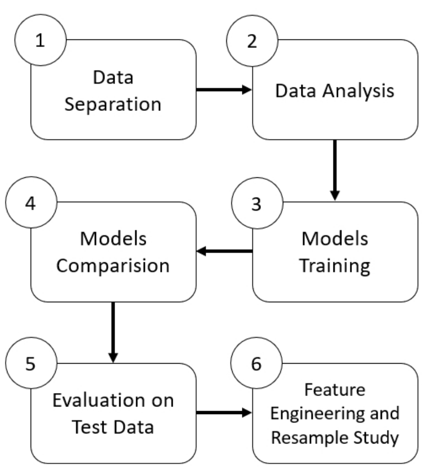

Figure 1: Experiment Execution

Figure 1 presents the study execution steps: (1) Data Separation, (2) Data Analysis, (3) Models Training, (4) Models Comparision, (5) Evaluation on Test Data, (6) Feature Engineering and Resample Study. Our study and the basis study share steps 1 to 5. Step 6 was added 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 big part to train the machine learning algorithms, while we used the small one to evaluate the effectiveness of the trained model on unseen data. The *Data Analysis* step tried to maintain only the relevant features. Highly correlated data can bias machine learning training [13], and because of that, we used Spearman's correlation algorithm to eliminate similar features. We removed instances with at least one blank value from our dataset. Finally, for more efficient training, we normalized all values.

The training result of one machine learning algorithm is a predictive model. This model performance can vary depending on the dataset used and the values of the algorithm parameters, known as hyperparameters. We can test the multiple combinations of hyperparameters to obtain better results (the *Model Selection* step), but manually doing so would be costly. For this purpose, we used the RandomizedSearch algorithm in the Scikit-Learn[3] library in Python. That algorithm tries to optimize permutations of hyperparameters by choosing samples randomly. Several models are generated for each permutation of hyperparameters, and their performance is listed, allowing us to identify the best model for each situation.

Another important detail to highlight is all training used the techniques of resampling data, feature selection, and polynomial features. In the *Models Training step* (Section 5.1), feature engineering and resample data techniques were applied with fixed values, based on trial and error by one of the authors (upper and lower extremes were tested using decision tree). Feature selection was done using Scikit-Learn's SelectKBest[4] class, which selects features according to the k highest scores. The fixed k parameter was 5. For the polynomial features technique, the PolynomialFeatures [5] class was used with a polynomial of degree of 2. The use of the SelectKBest class occurred independently and in conjunction with the PolynomialFeatures class, and in the second case, the polynomial features were created after the automatic selection of the features. The used classes for resample data were SMOTE[6] and RandomUnderSampler[7]. Both 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"*. For both cases the value for *strategy* parameter was 0.2. At this point in the experiment, the objective was not to choose the correct parameters, but rather to identify algorithms that were good candidates for generating prediction models for each of the smells, so that it would later be possible to investigate the impact of different feature engineering and resampling techniques on code smell predictions.

We carried out each training using 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 hyperparameterization performance in the *Models Comparison* step, we selected the parameters that lead to the best F1 measure. After we obtained obtained the optimal models for the seven evaluated algorithms, we tested the optimal models by providing the test dataset as input (equivalent to 20% of the total data) in the *Evaluation on*

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

[3]https://scikit-learn.org/stable/

[4]https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

[5]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

[6]https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

[7]https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html

Test Data step. The F1 and AUC metrics were the output of that step. This step predicts how the models behave in a real situation.

The output of *Evaluation on Test Data* is input to *Feature Engineering and Resample Study*. In this step, machine learning algorithms with the best performance were chosen for code smell prediction. Our goal was to explore how different variations of feature quantity, polynomial features and resample data can impact the models performance. The automatic selection of features occurred using Scikit-Learn's SelectKBest class. In feature engineering experiment, the k parameter ranged from 1 to the total number of features for each dataset, 12 in the case of class code smells and 10 in the case of methods. The feature selection experiment was carried out in two ways, the first in which only the k features with the highest scores were selected, per interaction, and the second in which the same selection was made, but the polynomial features function was subsequently applied. We chose to do it this way, considering that in a real situation, the features are first collected, and then new features are created using machine learning techniques. Both resample techniques, oversample and undersample, were used individually to evaluate their impact on the imbalanced datasets. The *strategy* parameter values range from 0, where the majority data remains untouched during undersampling or synthetic minority data are not generated during oversampling, to 1, where the majority data are entirely removed in undersampling or the minority data are equalized to the majority in oversampling.

5 RESULTS

This section reports the performance of the experiments in classifying the unseen instances on our test data. Section 5.1 presents the replication of the basis study [10]. We call our replication as baseline model experiment. Sections 5.2 and 5.3 report the experiments to evaluate the variation of feature engineering and resample data techniques. We choose for these experiments the best models found in Section 5.1. As in the baseline model experiment, the classes used in the study were SelectKBest, SelectPolynomialFeatures, SMOTE and RandomUnderSampler. However, in these experiments the parameters of these classes varied, with the aim of evaluating their impact on predicting code smells.

The results for each of the experiments will be discussed. Due to space limitations, only the data considered most important to be discussed in this paper were presented. The source codes used in the experiments and the tabulated results are available in the experiment documentation[8].

5.1 Baseline Model Experiment

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

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

Table 6 presents the results of the 3 best algorithms for each of the code smells for the F1 and AUC metrics. The first column lists the code smells. The second column lists the algorithms that generated the models with the best results for F1 and the third column for AUC. The fourth column presents the algorithm chosen to carry out the experiments with feature engineering and resample data. The complete results can be accessed in the experiment repository.

Table 6: Base Line Experiment - Best Results

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

God Class had the best results in this experiment. In terms of F1, the best models were generated by RF and GBM, both with performance of 0.72, and DT with 0.71. The best performance in terms of AUC was for the DT with 0.96, while RF and GBM models obtained values of 0.95. NV and LR obtained good results, above 0.8. The best model generated by the MLP algorithm was not able to predict classes with God Class due to the unbalanced data in the dataset (2.31% according to Table 2).

Only 0.41% of classes in the dataset are affected by Refused Bequest (Table 2), indicating the need to explore further techniques that deal with unbalanced data. The models generated from DT and RF obtained the best performances for F1 and AUC. The values of the F1 and AUC metrics obtained from the GBM model were 0.09, for F1, and 0.70 for AUC. NB model obtained 0.64 for AUC score. The worst case occurred with MLP, in which the data imbalance did not allow the creation a model able of identifying the classes with Refused Bequest, such that in this case the F1 and AUC values were respectively 0.0 and 0.5

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

The models generated by machine learning algorithms to predict Long Method obtained low values for the F1 metric, but for AUC the majority exceeded the score of 0.8. The highest score for the F1 metric was obtained from the model generated by KNN, at 0.41, followed by RF and LR, both with 0.33. The highest AUC scores were obtained from the models generated by GBM, DT, NB and LR,

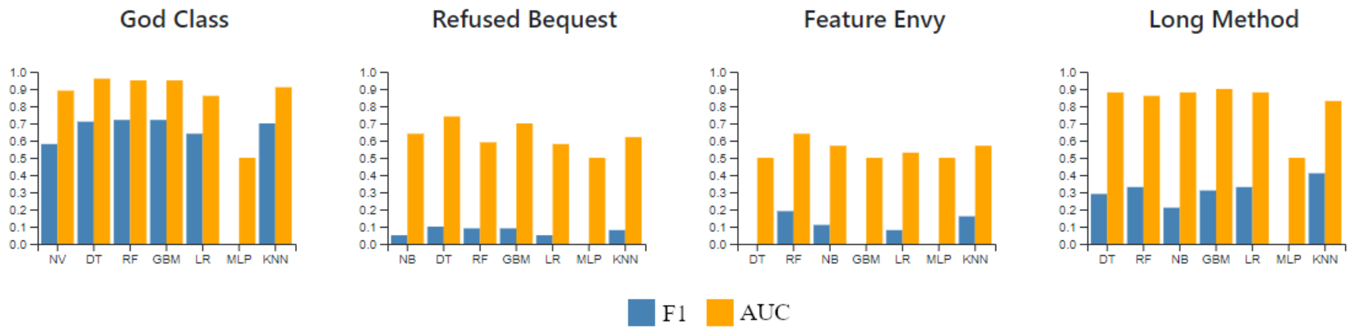

Figure 2: Baseline Experiment - Results

the first with 0.9 and the others with 0.88. However, in the case of AUC, the RF and KNN models were above 0.8, reaching 0.86 and 0.83, respectively. As in other cases, the data imbalance did not allow the MLP model to be able to predict smelly elements.

An important aspect to note in the dataset of the four code smells is that they are all imbalanced. However, in cases such as RB and FE, this imbalance is below the proportion of 1:100, being considered a more severe imbalance [24]. If compared with GC and LM, which have an imbalance above 1:100, we notice that the performance of the RB and FE models are 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 better performance of the DT, RF and GBM algorithms, which are able to capture more complex patterns, especially in the last two cases that make use of ensemble techniques. Given the explained results, we summarize the answers to the first research questions as follows.

RQ1: For class 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 smell, which obtained better results than RF for the AUC metric. For GC smell, the DT algorithm, with 0.71 for F1 and 0.96 for AUC, was chosen for experiments with feature engineering and resample techniques. For RB, DT was chosen too, with 0.10 for F1 and 0.74 for AUC. For the FE code smell, the algorithms with the best performance, for F1 and AUC, were RF, KNN and NB. In this case, RF obtained the best performance for both metrics, 0.19 for F1 and 0.64 for AUC, therefore it was chosen for experiments with feature engineering and resample techniques. For the LM code smell, the algorithms with the best performance for F1 were KNN, RF and LR and for AUC they were GBM, DT and NB. As for these smells the AUC results were high and the F1 results were not, we decided to choose the algorithm with the best F1 result, KNN (F1 = 0.41), to see the impact of the feature engineering and resample techniques on it.

5.2 Feature Engineering

Figure 3 shows the results of the Feature Engineering experiments. The first chart presents the F1 results for the variations of the *K* feature selection technique. The second chart presents the F1 results for variations of the *K* feature selection technique, followed by the application of 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 4 symbols, which represent the smells: the circle is GC, the triangle is RB, the square is FE and the cross is LM. 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 values 11 and 12, the square and cross symbols do not appear, this is because the methods dataset only has 10 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 4 code smells, the second column is the F1 results using only the feature selection technique and the third uses the feature selection technique, followed by polynomial features. The fourth and fifth columns are equivalent to the second and third columns, but for AUC metrics. 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. Metrics that remained unchanged in results 0.0 for F1 or 0.50 for AUC are represented as N/A. Detailed results for each of the parameters can be found in the experiment documentation.

For the God Class code smell, 5 features the test results stabilized both for the F1 metric and for AUC, which reached values of 0.73 and 0.86, respectively, at this point. The 5 features were

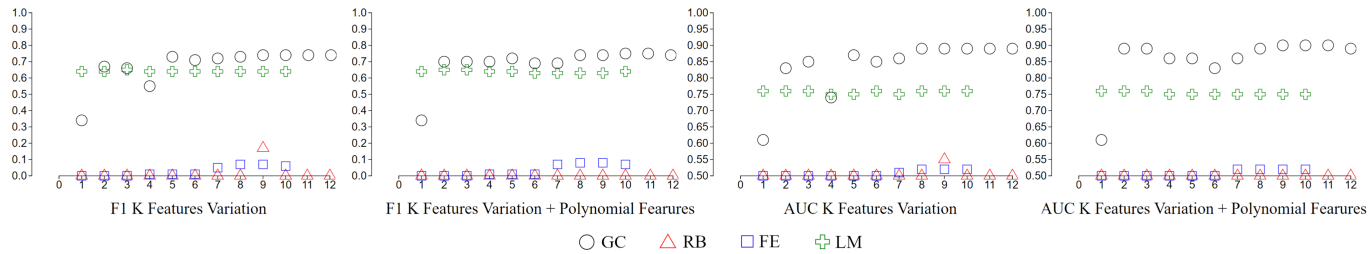

Figure 3: Feature Engineering Results

FANOUT, LCOM*, RFC, TMQ, WMC (more details can be found in the experiment documentation). In the case where 2 features were selected (RFC and WMC) and then applied the polynomial features technique, the F1 value was 0.69 and AUC was 0.89, showing that in this specific case, the use of the polynomial features technique allows us to get closer to the same result in which it is absent with 3 fewer features.

The Feature Envy code smell achieved a significant improvement in F1 results from the selection of 8 features (FANIN, FANOUT, WMC, LOC, RQ, VQ, PQ, LQ, CQ), from 0.0 (only one feature) to 0.07. With the same 8 features, but using polynomial features, there is also a slight improvement going 0.8. This same improvement was not observed in the case of the AUC metric, when all features were used, only 1 percentage point increased (0.50 to 0.51). Although the Feature Envy results are still very low, it is noted that both the increase in the number of features and the application of polynomial features had positive impacts on the predictive models, indicating that it might be necessary to collect more features that are able of representing the smell and continue using polynomial features to deal with the data variance problem. Finally, the use of these techniques in the Refused Bequest and Feature Envy code smells did not have a significant impact on improving performance on 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. In the cases of GC and FE, increasing the models dimensionality was beneficial to a certain extent. From this point onwards, the increase in features seemed to no longer have any effect. As for the RB and LM smells, increasing the number of features did not seem to have positive or negative impacts, indicating that only one feature is decisive in predicting these smells. In this case, it might be more interesting to explore new features to predict these smells. Therefore, increasing features is something that can be explored in some cases to improve models, but it is not a guarantee that it will be useful for all types of datasets. Its use may even indicate the need to explore new features not yet included in the generation of models. Regarding the use of polynomial features, we notice that in some cases, such as GC and FE, it provided an improvement, but in both cases, very slight (of just one percentage point). Therefore, its use, although positive, may not be so relevant.

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 results using the oversample technique, the second chart the F1 results using the undersample technique, and the third and fourth charts present the oversample and undersample results, respectively, for the AUC metric. Each chart presents 4 symbols, which represent the smells: the circle is GC, the triangle is RB, the square is FE and the cross is LM. The Y axis is the values for the F1 and AUC metrics. The X axis is the value of *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 4 code smells, the second and third columns are the F1 and AUC results for the oversample techniques, respectively. The fourth and fifth columns are the F1 and AUC 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. Detailed results for each of the parameters can be found in the experiment documentation.

The use of the SMOTE oversample technique did have the lowest impact on the F1 metric for the code smells with the lowest imbalance, GC (2.31%) and LM (1.39%). On the contrary, the code smells with the greatest imbalance, RB (0.41%) and FE (0.39%), had significant improvements, but they were not enough to achieve good results for the F1 metric. For the undersample technique, performance equivalent to oversample was noted for the RB and FE smells, however in the cases of GC and LM, when the *strategy* parameter is equal to 0.1, there is a slight improvement in both. It is also possible to note that when the *strategy* parameter has higher values, the removal of majority data can harm the model's ability to make predictions. It is possible to evaluate that the prediction performance from the point of view of the AUC metric is good in the cases of GC and LM, and very good in the cases of RB and FE.

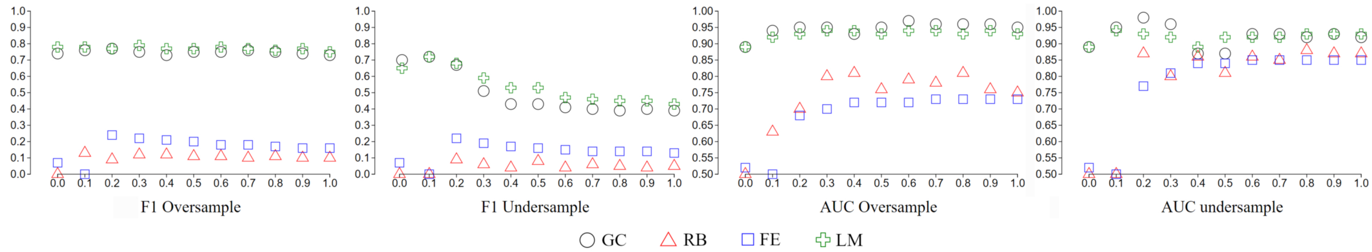

Figure 4: Resample Data Results

As the results show, the use of resample techniques can be a good solution to deal with the imbalance of code smells datasets, but they must be used with care. For datasets with imbalance greater than 1:100, RB and FE, the undersample technique performed better than oversample in the case of the AUC metric and obtained a similar result for the F1 metric. For smells with lower imbalance, GC and LM, oversample may be a better option, but this should be further explored with new experiments.

After the conclusions of Sections 5.2 and 5.3, we can answer the second research question as follows.

RQ2: The use of feature selection, polynomial features and resampling data techniques can help in refining predictive models and enable the identification of possible changes in the dataset. In the case of selecting the number of features, it was possible to obtain improvements in cases such as GC and FE, but it also allowed us to evaluate that only one feature was relevant in the prediction of RB and LM, indicating the need to explore new features if we want to improve the models. The use of polynomial features, despite having proven positive in the cases of GC and FE, but did not justify its use, as it improved in both cases by a percentage of just 1 point. 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, in cases where it was below 1:100, undersample seemed to be a better option.

6 DISCUSSION

In this section, we compare the results of this experiment with the basis 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 basis study: GC: 4.77%, RB: 8.96%, FE: 3.46% and LM: 0.87%. While in the current study, this 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 basis study, they were GC and RB, in the current study they were GC and LM. We also highlight that 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 values were higher in the basis study, and; (ii) the best prediction performances in the basis study were for the smells GC and RB, while in the current study they were GC and LM. The first difference is justified by the fact that the imbalance in the basis study dataset is smaller. The dataset for the basis study is from 2010, while the current study is from 2021. Older Java systems were expected to have more code smells, as new development pratices and tools, such as Linters, mitigate them. Regarding the second difference between the studies, this was already justified when we concluded that the smallest imbalances provided the best results, regardless of which smell it was.

One of the proposals for future work in the basis 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. In some cases this improvement was significant, especially for the AUC metric, like for RB and FE 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, something that can be checked for the RB and FE 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. It is important to highlight that very high values for the *strategy* parameter can harm the models. The feature selection technique was still able to provide valuable insights, as for RB and LM, that only one metric was responsible for the models' performance, indicating the need to seek new ways of representing this smell.

We believe that the results of this study can be useful to different professionals in the following ways.

- **Researchers**: indicate the need to research other techniques, in addition to undersample and oversample, to deal with data imbalance in code smells.
- **Developers**: facilitate data-driven source code refactoring.
- **Software Architect**: allow improving the results of fuzzy predictions, in order 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 some threats and our actions to mitigate them, organizing 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 only two metrics, namely F1 and AUC, to evaluate the models for code smell detection. This choice poses a threat to construct validity since the metrics may not accurately measure the effectiveness of the machine learning classifiers. However, we believe this threat does not invalidate our main findings since previous related studies also use these metrics with similar purposes [ref, ref, ref]. Similarly, we also used only one implementation for each classifier which may bias our results. However, these implementations were provided by heavily used libraries 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 lack actual smells. To mitigate this threat, we relied on the combination of five tools for the ground truth creation. We also manually validated a sample of the code smells to cross-validate our voting strategy.

Internal Validity. Threats to internal validity are influences that can affect the independent variable to causality [50]. In our context, this threat refers to 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 (for instance) are some confound factors. To minimize this threat, our dataset is composed of 30 software systems from different domains. Although we are constrained by the tools used (for instance, to Java-based technology), we have done our best effort to analyze and understand the results and mitigate confound 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 may not represent the entire 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 that we only performed the detection in Java systems because we could not find enough tools to create the ground truth for other languages. In fact, the study design is not limited on the programming language and could be performed in different datasets. Furthermore, the selected systems range from different domains and sizes, including large frameworks from industry.

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]. In our study, the performance metrics used may have led us to false conclusions. The main reason for choosing F1 and AUC is that we do not want to prioritize neither precision nor recall. Moreover, they are well-known metrics from machine learning evaluation and information retrieval. Finally, since not all information was clear to answer the research questions, cross-discussions among the paper authors often took place 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 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 Support Vector Machine model to identify four anti-patterns 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, for instance the use of techniques that deals with data imbalance. Even though our dataset is composed of less systems, we highlight that the selected systems in our work reflects current practices on open-source development and new technologies, while previous works analyse the performance on systems collected at 2012 [13, 16]. We also experimented with different techniques that deals with unbalanced data, tuning of hyperparameters, and we have used different polynomial features. We also present our results with an additional measurement, the AUC.

More recently, Stefano et al. [12] evaluated if there was impact in the model performance when analysing multiple versions of a system versus using only one version. They did not found a statistical significant difference between their results. Lomio et al. [29] evaluated how rules from the SonarQube system can help in improving the performance of fault prediction algorithms. Santos et al. [41] used an ensemble model to identify how similar the models for defects are in comparison with seven code smells at 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 are different, we highlight our work complement them, since we also evaluate two smells at 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 samples 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 smells, the God Class and Refused Bequest.

9 CONCLUSION AND FUTURE STUDIES

This study presents a differentiated replication of a basis 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 oversample performed better for datasets with less severe imbalance. The feature selection technique provided insights into the need to collect more information to represent code smells such as RB and LM. The polymial features technique had little impact on the performance of the predictions.

As future works, 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, in addition to the features presented in this study. We intend to expand the dataset, evaluating other systems. Finally, it is also necessary to extend the work to other code smells and languages, in addition to Java.

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