Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study

Daniel Cruz
danielvsc@dcc.ufmg.br
Federal University of Minas Gerais
Belo Horizonte, Brazil

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

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

ABSTRACT

Bad smells are symptoms of bad design choices implemented on the source code. They are one of the key indicators of technical debts, specifically, design debt. To manage this kind of debt, it is important to be aware of bad smells and refactor them whenever possible. Therefore, several bad smell detection tools and techniques have been proposed over the years. These tools and techniques present different strategies to perform detections. More recently, machine learning algorithms have also been proposed to support bad smell detection. However, we lack empirical evidence on the accuracy and efficiency of these machine learning based techniques. In this paper, we present an evaluation of seven different machine learning algorithms on the task of detecting four types of bad smells. We also provide an analysis of the impact of software metrics for bad smell detection using a unified approach for interpreting the models' decisions. We found that with the right optimization, machine learning algorithms can achieve good performance (F1 score) for two bad smells: *God Class (0.86)* and *Refused Parent Bequest (0.67)*. We also uncovered which metrics play fundamental roles for detecting each bad smell.

KEYWORDS

bad smells detection, machine learning, software quality, software measurement, empirical software engineering

ACM Reference Format:
Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. 2020. Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. In *International Conference on Technical Debt (TechDebt '20), October 8–9, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387906.3388618

1 INTRODUCTION

Bad smells are symptoms of bad design choices implemented on source code [22]. They negatively affect software quality attributes, such as software comprehension [2] and robustness [26]. Fundamental processes of software development, such as software evolution and maintenance, are directly impacted by the presence of smells [55]. Furthermore, they are one of the key indicators of technical debts, specifically design debts [52], resulting in extra costs in the software development lifecycle, such as rework [48]. Even though a bad smell should not always be removed [18], it is important to be aware of their existence to be able to manage this design debt [52]. Thus, to detect bad smells, several techniques and tools have been proposed [17, 27, 37–39, 43, 54]. Moreover, they present different types of detection strategies, such as software metrics, textual analysis, and AST analysis [16]. For the software metrics based detection strategies, thresholds must be defined in order to identify bad smells, becoming a major drawback due to its innate complexity, being adressed by several studies that propose methods for threshold derivation [19, 23].

Recently, tools with different detection strategies have been proposed, for instance, using machine learning algorithms [3, 21, 27, 28, 35, 36]. These approaches remove the constraints on the use of purely deterministic and heuristic solutions, in which algorithms and thresholds are explicitly implemented and defined. Instead, machine learning provides the capability of understanding the properties present on the target systems, and based on them, it performs the bad smell detection.

To assess the performance of these detection approaches, several studies conducted empirical evaluations on different datasets and experimental setups, including manual analysis with professionals [27, 43, 44]. The efforts applied to manual identification of bad smells aim at providing ground truth for the evaluation of the proposed detection strategy. However, we lack empirical knowledge on the performance of machine learning algorithms in a large unified dataset.

This work aims at evaluating and comparing seven machine learning algorithms on the detection of four bad smells: *God Class (GC)*, *Long Method (LM)*, *Feature Envy (FE)*, and *Refused Parent Bequest (RPB)*. For this task, we generated an unified dataset of 20 systems written in Java [53] by using five bad smell detection tools, and their set of software metrics. We then performed a statistic evaluation of a representative sample of the ground truth in order to verify if the strategies of the tools comply with the human perception of bad smells.

As a result, we evaluate more than three thousand models, with different parametrizations, comparing seven machine learning algorithms. Since they may use a very complex set of decision boundaries and, consequently, being hard to understand, we provided an explanation of the software metric contributions to the bad smell detection. Besides providing greater confidence on the results found, this approach may provide visual feedback, helping practitioners in the refactoring of smelly instances.

We found that *Random Forest* and *Gradient Boosting Machines* perform better than other classifiers. Their performance may be

TechDebt '20, October 8–9, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7960-1/20/05…$15.00
https://doi.org/10.1145/3387906.3388618

due to their capabilities in dealing with unbalanced data, since they are based on ensemble methods. Previous work [42] confirms that the proportion of bad smells in software systems is usually low, resulting in a unbalanced dataset. These classifiers achieved good performance for two out of four analyzed bad smells: *GC* and *RPB*. Moreover, we performed an analysis on the contribution of each metric for the bad smell detection. We rely on a SHAP-based[34] visualization technique to interpret the detection results. For instance, we discovered that *Depth of Inheritance Tree (DIT)* is the metric that mostly contributes for detecting *Refused Parent Bequest*. This observation differs from known detection strategies, such as the one proposed by Marinescu [31], which use different sets of metrics, like *WMC* and *NOM*, but do not use DIT.

The main contributions of this paper are summarized as follows.

- A public dataset with about 232K methods and 36K classes with ground truth for four types of bad smell [1].
- An empirical evaluation of seven machine learning algorithms on the detection of bad smells.
- Software metrics contribution evaluation for each bad smell.
- A method to explain why a instance is labeled as smelly or non-smelly, based on the software metrics used.

The rest of this paper is structured as follows. Section 2 describes how the dataset was built. Section 3 presents our goal, research questions, and the design of this study. Section 4 discusses the machine learning model results and the software metric contributions. Section 5 discusses some threats to the study validity. Section 6 presents related work, focusing on the detection of bad smells using machine learning. Finally, Section 7 presents the conclusive remarks and outlines future works.

2 DATASETS

In this work, we evaluate machine learning detection of four bad smells [22]: *God Class (GC)*, *Long Method (LM)*, *Feature Envy (FE)*, and *Refused Parent Bequest (RPB)*. The *GC* and *RPB* smells are related to a class, i.e , they are at class level. The *LM* and *FE* are related to a method, i. e., they are at method level.

In this study, we used a dataset with 20 open source systems written in Java that contain instances of the four bad smells. The systems selected were extracted from the *Qualita Corpus* [53], with most of them being used in other studies on machine learning usage [15, 20]. The system size ranges from small ones, like *SquirrelSql* (6.9K LOC), to larger systems, such as *Hibernate* (621.5K LOC). For each system, the set of classes, methods, and software metrics were extracted. The list of systems can be found on our complementary webpage [1]. We opted to create a new dataset in order to vary the system domain and its innate characteristics, such as metric values, to assure the quality of the built ground truth, and to perform the comparison on a large and unbalanced dataset.

2.1 Metrics

Table 1 presents in its first and second columns the metrics that were calculated at the class and method levels, respectively. For every class and method in our dataset, we extracted 17 metrics for the class level (Table 1, first column) and 12 metrics for the method level (Table 1, second column). The class level metrics were extracted using the tool VizzMaintenance [6]. This tool calculates

Table 1: Software Metrics

Class Level	Method Level
Couple Between Objects (CBO)	Couple Between Objects (CBO)
McCabe Cyclomatic Complexity (CYC)	Highest Number of Nested Blocks (maxNestedBlocks)
Data Abstraction Coupling (DAC)	Lines of Code (LOC)
Depth of Inheritance Tree (DIT)	Number of Unique Words (uniqueWordsQty)
Improved Lack of Cohesion in Methods (ILCOM)	Quantity of Assignments (assignmentsQty)
Lack of Cohesion in Methods (LCOM)	Quantity of Comparisons (comparisonsQty)
Locality of Data (LD)	Quantity of Loops (loopQty)
Length of Class Names (LEN)	Quantity of Parenthesized Expressions (parenExpsQty)
Lines of Code (LOC)	Quantity of Variables (variables)
Lack of Documentation (LOD)	Response For a Class (RFC)
Message Passing Coupling (MPC)	Size of Parameter List (parameters)
Number of Attributes and Methods (NAM)	Method Start Line(startLine)
Number of Children (NOC)	Weighted Method Count (WMC)
Number of Local Methods (NOM)	
Response For a Class (RFC)	
Tight Class Cohesion (TCC)	
Weighted Method Count (WMC)	

known metric suites, like *Chidamber & Kemerer* [13], *Li & Henry* [33], *Bieman & Kang* [8], and *Hitz & Montazeri* [24]. The description for each metric can be found at the tool webpage[1]. The method level metrics were extracted by the CKMetrics tool [4]. The tool computes known metrics, such as *Weighted Method Count* and more experimental ones, like *Quantity of Assignments*. The description of the metrics can be found at the tool webpage[2].

We can observe in Table 1 that some metrics originally defined at class level (*WMC*, *CBO* and *RFC*) have alternative versions at method level. For *WMC*, it is the cyclomatic complexity of the method. For *CBO* and *RFC*, they measure the class coupling realized by the method.

2.2 Ground Truth Creation

To create the ground truth of bad smells, we combined the result of five automatic detection tools: PMD[3], JDeodorant [17], JSpirit [54], an implementation[4] of DECOR [39], and Organic [40]. For each bad smell, we applied three detection tools and computed an agreement voting between their results. That is, an instance (class or method) is considered smelly if two or more tools detect it. Each tool is able to detect only a small subset of bad smells, so we identified the intersection between them to assure that each smell could be detected by three tools. Table 2 presents these intersections. For instance, we can observe in Table 2 that the ground truth for the *LM* smell was obtained by combining the results of JDeodorant, Organic, and DECOR.

It is worth noticing that the number of smell instances is small for all types of bad smells, as expected [42]. For *GC*, about 4.77%

[1] arisa.se/compendium/
[2] github.com/mauricioaniche/ck
[3] pmd.github.io
[4] ptidej.net/publications/Keyword/CODE-AND-DESIGN-SMELLS.php

Table 2: Tools Used for the Ground Truth Creation

	PMD	JDeodorant	JSpirit	DECOR	Organic
GC	x	x	x		
LM		x		x	x
FE		x	x		x
RPB			x	x	x

of classes were detected as smelly in our dataset (i.e., 1,689 smelly instances out of 35.6K classes). For *LM*, this proportion decreases even more, from the 232K methods extracted, only 2,023 instances were labeled on the ground truth as positive, i. e., 0.87% of all methods. For *FE*, considering the 232K methods, 3.46% instances were considered as smelly (8,016 instances). Finally, for *RPB*, 8.96% of classes were labeled as positive for this bad smell (3,190 instances of 35.6K classes).

Since most of the selected tools are metrics based, and their thresholds remained fixed to make the process of detection homogeneous, we rely on an evaluation of the agreement between the tools results and a manual validation. This comparison allow us to understand if the tools comply with the human perception of what a bad smell is. For this purpose, we statiscally sampled our dataset considering each bad smell with a confidence level of 90% and error of 10%. The obtained subset was manually validated by six practitioners using a similar strategy of Schumacher et al. [47]. It consists of formulating questions to evaluate if a instance is smelly or not. This technique allowed each instance in our sample to be analysed in a uniform way. The obtained agreement was reasonable. The questions and details on the procedures for manual validation of the ground truth are available at our webpage [1].

3 STUDY DESIGN

In this section, we describe the machine learning process (Section 3.1), the experimental steps (Section 3.2) and the components of the binary classification task: Features (Section 3.3), Classifiers (Section 3.4), and Evaluation (Section 3.5).

3.1 Machine Learning Detection of Bad Smells

The detection of bad smells can be performed by the task of supervised learning named *Binary Classification*. This task consists of classifying the instances of a given set in two groups: smelly and non-smelly. For instance, given a set of features of a class, predicts whether it is a *God Class* or not. In general, we can apply this classification for every bad smell. This supervised machine learning process consists on two main phases:

(1) Training: train a model using a machine learning algorithm (*classifier*) in a dataset composed by labeled instances. Each instance has the form of a tuple <*X,y*> where *X* is a vector of features (x_1, x_2, ..., x_n) and *y* is the label to be predicted (0 or 1). In the training, the algorithm learns from the data.
(2) Validation: after the training, the learned model is used to perform predictions on an unseen dataset, outputing 0 or 1 for each instance. Later, the model is evaluated through performance metrics, in which a comparison between the already know values for *y* on the ground truth and the predictions made by the classifier model is made.

The goal of this study is to evaluate and compare seven machine learning algorithms for the detection of bad smells by: (i) comparing different classification methods with optimized parameters; (ii) using a realistic dataset composed of 20 systems; and, (iii) explaining the decisions performed by the classifiers, i.e, evaluating which metrics contributes the most in the classification problem. To achieve this goal, we define three research questions (RQ) and design an empirical study to answer them:

- *RQ1: How accurate is the detection of bad smells using static software metrics and a machine learning algorithm?*
- *RQ2: How does the accuracy of the detection vary across the use of different machine learning algorithms?*
- *RQ3: Which are the best software metrics to detect each bad smell?*

3.2 Experimental Setup

For each smell, we performed an experiment with the following steps.

Data Separation. Each dataset was divided into two parts: 80% for Training/Validation and 20% for Test. The first part was employed on the training and validation of the models. The second one was used to perform the tests on unseen data.

Data Analysis. The results and the feature set were analyzed in order to avoid possible causes of overfitting, described further in Section 4.2.

Models Parametrization. For each classifier, we have generated multiple models by using *Random Search*, an automated technique that optimizes the hyperparameters of the classifiers [7]. It resulted in more than one thousand models, with a great range of variation on the hyperparameters.

Models Comparison. The models were compared by the metrics (Section 3.5) obtained on the cross-validation [30]. We compared both the performance of each algorithm alone (regarding its parameterization), and between different classifiers. For the former, the best parametrization was used. The best algorithms are compared using multiple performance metrics presented in Section 3.5.

Test on Unseen Data. Finally, we applied the seven models learned on the second part of the dataset. This data was never seen by the models on the previous phase (training/validation). Our goal was to create a hypothetical, but practical situation, where a new system or a set of systems is available, and the machine learning approach must be applied to predict the presence of bad smells in the new (set of) system(s).

3.3 Features

The feature vector of each instance is composed of the set of software metrics. Nucci et. al [15] raised in their work that a machine learning model created using independent variables that are highly correlated can lead to overfitting and, consequently, a biased evaluation [41]. For this reason, we opted to evalute the correlation between features using the *Spearman correlation coefficient* before applying the classifiers. We have found a strong positive correlation between two pairs of metrics: 1) *DAC* and *CBO*, with a ρ of 0.98956, and 2) *NOM* and *NAM*, with a ρ of 0.93208.

The feature *DAC* is a software metric that measures coupling between abstract data types and, as expected, has a high correlation with the feature *CBO*, which measures coupling to other classes. The difference between them is that *DAC* is limited to type references. Then, the feature *DAC* was discarded. The features *NOM* and *NAM* are also correlated by their own definitions. The metric *NOM* measure the number of methods present in a class, while the metric *NAM* measures the number of methods and attributes (or fields) in a class. It is impossible that the *NOM* metric presents values greater than *NAM*. Thus, the metric *NOM* was discarded.

3.4 Classifiers

Several machine learning algorithms can be used for the task of detecting bad smells. We selected the following subset: Naive Bayes (Gaussian likelihood) [32], Logistic Regression [29], Decision Tree [10], Multilayer Perceptron (Artificial Neural Network) [25], K-Nearest Neighbors (KNN) [14], Random Forests [9], and Gradient Boosting Machine (GBM) using XGBoost Trees as classifiers [12].

We selected these algorithms due to their differences in nature and their wide use in the literature. For instance previous study [20] experimented implementations and variations of several algorithms, such as Decision Tree (J48 Pruned, J48 Unpruned), Random Forest, Naive Bayes, and SVM (Linear Kernel, Polynomial Kernel). To extend and compare the results with previous works, we included most of the algorithms used in other studies, and added others, such as Logistic Regression, Multilayer Perceptron, KNN, and Gradient Boosting Machine. Different from previous work, which applied Adaboost for all classifiers, we considered an algorithm with boosting separately.

3.5 Evaluation

To assess the performance of the models and to train the classifiers, we applied a method called K-Fold (with K=10) [49] on the 80% part of the dataset used as Training/Validation, as described in Section 3.2. In a nutshell, the K-Fold splits the dataset into *K* partitions, where *K-1* partitions are used as training and the last one is used as validation. These partitions are permutated on each iteration, each one of them being used only one time for validation. The K-Fold provides a way to compare different models and best evaluate them, since it uses ten different validations, instead of only one. Furthermore, it helps to avoid overfitting, as the training set varies on each interaction [11, 30].

As the number of smelly instances (1,698) is much smaller than the number of clear instances (33,902), the models can lead to bias, where they prefer to always predict the instances as non-smelly, benefiting their accuracy. For instance, consider a *dummy* classifier which always predicts as non-smelly. If this classifier is used to perform predictions on this dataset, it will have an accuracy of 95.23%. To avoid this kind of bias, we performed the evaluation of the models using 3 additional micro-weighted metrics (Precision, Recall, and F-Measure) for the positive class. Be *TP* the *True Positives*; *FP* the *False Positives*; *TN* the *True Negatives*; the *FN* the *False Negatives*. The performance metrics are defined as follows.

Accuracy. The accuracy is calculated as:

$$Acc = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision. Measures how much the predictions of smelly instances are correct. The precision is calculated as:

$$Precision = \frac{TP}{TP + FP}$$

Recall. Measures how much of the existent smelly instances were detected by the model. The recall is calculated as:

$$Recall = \frac{TP}{TP + FN}$$

F1. F1 or F-Measure is the harmonic mean of precision and recall:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

4 RESULTS AND DISCUSSION

This section presents the results of the models generated with each classifier on the detection of four bad smells. In total, 3,460 models were generated. For each classifier, we present statistics of the models performance: the columns *Min*, *Median*, *Mean*, *Standard Deviation (SD)*, and *Max* in Tables 3 to 9. Our analysis is organized for each bad smell: *God Class (GC)*, *Long Method (LM)*, *Feature Envy (FE)*, and *Refused Parent Bequest (RPB)*. Each model was evaluated by the four metrics defined in Section 3.5, and are presented inside each bad smell group as rows: *Accuracy*, *Precision*, *Recall*, and *F1*.

4.1 Naive Bayes

Naive Bayes classifier (NB) is very simple, so the search for the best hyperparameters is critical. In Table 3, we can observe this by the *SD* values. The metrics concerning the positive classes (*precision*, *recall*, and *F1*) vary largely. Also, this classifier reached significative results only for the *GC* bad smell, presenting an *F1* of 0.655. It may indicate that the detection of *GC* is easier compared to other smells.

Table 3: Performance of models using NB

NB	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.951	0.960	0.958	0.005	0.964
	Precision	0	0.618	0.571	**0.246**	0.814
	Recall	0	0.445	0.398	**0.325**	0.756
	F1	0	0.547	0.397	**0.283**	**0.655**
LM	Accuracy	0.954	0.986	0.977	0.017	0.991
	Precision	0	0.116	0.084	**0.081**	0.243
	Recall	0	0.231	0.29	**0.301**	0.635
	F1	0	0.196	0.118	**0.114**	0.256
FE	Accuracy	0.916	0.963	0.947	0.022	0.966
	Precision	0	0.038	0.045	**0.047**	0.098
	Recall	0	0.003	0.062	**0.078**	0.175
	F1	0	0.005	0.049	**0.058**	0.126
RPB	Accuracy	0.892	0.903	0.902	0.008	0.911
	Precision	0	0.335	0.303	0.124	0.508
	Recall	0	0.093	0.110	**0.099**	0.242
	F1	0	0.145	0.143	**0.116**	0.284

4.2 Logistic Regression

Table 4 presents the general performance of all models using the Logistic Regression classifier (LR). In general, all models performed better on the perspective of *recall*. However, there is a huge tradeoff with the *precision*, which means that these models only achieve high *recall* by guessing that an instance is always smelly. Thus, a large number of bad smells were detected, yet with a high number

Table 4: Performance of models using LR

LR	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.220	0.949	0.811	0.293	0.972
	Precision	0.058	0.440	0.470	0.247	0.799
	Recall	0.138	0.768	0.678	0.326	**0.978**
	F1	0.109	0.580	0.449	0.243	0.722
LM	Accuracy	0.517	0.945	0.876	0.165	0.991
	Precision	0	0.066	0.105	**0.113**	0.293
	Recall	0	0.448	**0.449**	**0.421**	**0.917**
	F1	0	0.092	**0.071**	**0.048**	0.127
FE	Accuracy	0.034	0.867	0.730	0.336	0.966
	Precision	0	0.043	0.050	0.035	0.106
	Recall	0	0.311	0.386	0.408	**1**
	F1	0	0.039	**0.061**	0.067	0.156
RPB	Accuracy	0.270	0.845	0.740	0.246	0.922
	Precision	0.099	0.216	0.312	**0.220**	0.711
	Recall	0.003	0.433	0.432	**0.350**	**0.896**
	F1	0.006	0.198	**0.215**	0.126	0.352

of false positives. In Table 4, we can observe this behavior on the *Mean value* reported by the *F1 score*, which is very low for three bad smells: *LM*, *FE*, and *RPB*.

Taking into account the problem of imbalanced data, this classifier addresses this problem by using a weighted penalty parameter for the wrong predictions on the training. The model with the best performance, among the others, presents this hyperparameter with the *"balanced"* value, which means that the penalty error is higher for wrong predictions on the positive class, that has fewer instances. However, this mechanism is sufficient to overcome the problem of imbalanced data, as it increases a bias that leads to many *false positives*, i.e, poor *precision*.

4.3 Multilayer Perceptron

Table 5 presents the general performance of all models using a Multilayer Perceptron classifier (MLP). The models also presented a high variation on their performance, as we observe by the *SD* of *precision*, *recall* and *F1*, all above 70% of the *Mean* value. Like *Naive Bayes* and *Logistic Regression* models, the *MLP* models present very different results with the variation of hyperparameters.

Table 5: Performance of models using MLP

MLP	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.590	0.957	0.950	0.043	0.979
	Precision	0	0.596	0.465	**0.305**	0.842
	Recall	0	0.488	0.396	**0.281**	0.797
	F1	0	0.528	0.398	**0.303**	0.777
LM	Accuracy	0.794	0.991	0.980	0.033	0.991
	Precision	0	0.050	0.129	**0.161**	0.525
	Recall	0	0.006	0.034	**0.056**	0.253
	F1	0	0.007	0.028	**0.045**	0.213
FE	Accuracy	0.593	0.966	0.959	0.032	0.966
	Precision	0	**0**	0.051	**0.119**	0.575
	Recall	0	**0**	0.009	**0.038**	0.400
	F1	0	**0**	0.003	**0.007**	0.034
RPB	Accuracy	0.299	0.911	0.887	0.079	0.937
	Precision	0	0.417	0.337	**0.260**	0.746
	Recall	0	0.144	0.149	**0.134**	0.761
	F1	0	0.139	0.165	**0.146**	**0.567**

As the features were not normalized to avoid biasing the feature importance analysis, the *MLP* models were very impacted by feature scaling. Some software metrics can have very large values, such as *LCOM*, while other metrics like *DIT* have, in general, lower values. For example, in our dataset, the highest value for *DIT* is only 11, with a *mean* of 0.98, while for *LCOM* is 234K, with a *mean* of 150.

We can also observe in Table 5 for the smell *FE*, most models performed very badly, except for *accuracy*, which takes into account the negative prediction. All other metrics reported a *median* of 0. For this smell, the best *F1* value achieved was 0.034, the lowest in this study. However, considering the first three classifiers, it was the only one to achieve an *F1* score above 0.5 for the *RPB* bad smell.

4.4 Decision Tree

Table 6 presents the general performance of all models using a Decision Tree classifier (DT). Decision Trees often performs well on imbalanced datasets, as the construction of the trees is performed by an algorithm that splits the nodes based on information gain (algorithms like C4.5) or Gini impurity (CART algorithm) [45]. Thus, decision boundaries for DTs are less influenced by the proportion of the target class, and are more prone to linearly separate them.

This behavior can be observed in Table 6, where almost all models achieved a good or average performance for the bad smells *GC* and *RPB*. For the bad smells *LM* and *FE*, despite the low *F1* scores, these models behaved like the *Logistic Regression* ones, focusing on trying to guess more positive predictions, as can be observed by the maximum *recalls* of 0.801 and 0.810, respectively.

Table 6: Performance of models using DT

DT	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.946	0.977	0.975	0.005	0.982
	Precision	0.482	**0.767**	**0.751**	0.056	**0.803**
	Recall	0.718	**0.754**	**0.765**	0.045	**0.929**
	F1	0.620	**0.763**	**0.754**	0.032	**0.816**
LM	Accuracy	0.893	0.987	0.979	0.023	0.991
	Precision	0.059	0.247	0.240	0.072	0.466
	Recall	0.035	0.253	0.287	0.147	**0.801**
	F1	0.065	0.248	0.230	0.050	0.292
FE	Accuracy	0.721	0.951	0.932	0.058	0.966
	Precision	0.082	0.296	0.275	0.076	0.508
	Recall	0.001	0.309	0.324	0.151	**0.810**
	F1	0.002	0.296	0.263	0.078	0.319
RPB	Accuracy	0.791	0.919	0.912	0.027	0.936
	Precision	0.240	**0.542**	**0.531**	0.080	**0.720**
	Recall	0.225	**0.541**	**0.546**	0.078	**0.792**
	F1	0.336	**0.541**	**0.527**	0.049	**0.577**

4.5 K-Nearest Neighbors

Table 7 presents the general performance of all models using the K-Nearest Neighbors classifier (KNN). We observe that, in general, these models performed much better than the other classifiers. For all bad smells, the models achieved a better *precision* than *recall*. For instance, for the bad smell *FE*, the *mean precision* was 0.440, against a *mean recall* of 0.022.

The intuition underneath KNN is that an instance belongs to a class if most number of the K instance neighbors belong to that class, which is called majority voting. Since the dataset is very imbalanced, as the K increases, the probability of the majority vote

Table 7: Performance of models using KNN

KNN	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.970	0.975	0.974	0.002	0.975
	Precision	0.761	**0.773**	**0.775**	0.015	**0.794**
	Recall	0.517	**0.684**	**0.652**	0.082	**0.724**
	F1	0.625	0.725	0.704	0.047	0.742
LM	Accuracy	0.991	0.991	0.991	0	0.991
	Precision	0.347	**0.402**	**0.430**	0.088	**0.579**
	Recall	0.027	**0.048**	**0.054**	0.028	**0.098**
	F1	0.043	0.084	0.091	0.042	0.154
FE	Accuracy	0.964	0.965	0.965	0	0.966
	Precision	0.303	**0.406**	**0.440**	0.135	**0.628**
	Recall	0.002	**0.024**	**0.022**	0.016	**0.039**
	F1	0.004	0.044	0.040	0.028	0.069
RPB	Accuracy	0.912	0.915	0.915	0.002	0.916
	Precision	0.508	**0.554**	**0.547**	0.025	**0.575**
	Recall	0.205	**0.221**	0.244	0.050	0.329
	F1	0.301	0.315	0.332	0.039	0.399

is biased to negative class. However, if this hyperparameter is low, the model will lose its generalization capacity. Thus, the models with the best performance used a K value of 2, which provides a good *precision*, but for the price of predicting only a few instances as smelly, resulting in a bad *recall*.

4.6 Random Forest

Table 8 presents the general performance of all models using a Random Forest classifier (RF). This classifier is a tree-based ensemble method, so-called *bagging*, that builds a set of Decision Trees and uses their average to perform the prediction. It helps to prevent overfitting, given that the used Decision Trees are often very simple and with low depth. Thus, this technique achieved good results for the bad smells *GC* and *RPB*, as presented in Table 8.

Furthermore, these models presented a behavior similar to the *KNN* models. We observe this result by the *precision* reported by them. In fact, for the bad smell *FE*, some models even reached a *precision* of 100%. As the *maximum F1* was 0.351, this model certainly reported a bad *recall*.

Table 8: Performance of models using RF

RF	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.980	0.985	0.985	0	0.986
	Precision	0.745	0.863	0.858	0.022	0.875
	Recall	0.773	0.837	0.836	0.028	0.927
	F1	0.814	0.847	0.846	0.010	**0.860**
LM	Accuracy	0.948	0.992	0.989	0.009	0.992
	Precision	0.117	**0.647**	**0.585**	0.148	0.777
	Recall	0.031	**0.142**	**0.194**	0.160	0.770
	F1	0.060	0.225	0.233	0.052	0.381
FE	Accuracy	0.770	0.969	0.957	0.046	0.971
	Precision	0.110	**0.702**	**0.677**	0.189	**1**
	Recall	0.013	**0.188**	**0.224**	0.154	0.814
	F1	0.025	0.300	0.281	0.058	0.351
RPB	Accuracy	0.909	0.949	0.947	0.009	0.951
	Precision	0.492	**0.835**	**0.813**	0.078	0.880
	Recall	0.412	**0.531**	**0.539**	0.060	0.753
	F1	0.559	0.650	0.641	0.024	**0.671**

4.7 Gradient Boosting Machine

Table 9 presents the general performance of all models using a Gradient Boosting Machine classifier (GBM). Boosting is an ensemble technique for both regression and classification problems, which makes use of a set of simple classifiers and generate complementary learning from each one. The goal is to prevent the bias on generalization error, by applying weight on the wrong predictions by the base classifiers (Decision Trees in our case). It provides, like in *Random Forest*, a good way to deal with imbalanced data. As shown in Table 9, the *RF* and *GBM* performances were very similar.

Table 9: Performance of models using GBM

GBM	Metrics	Min	Median	Mean	SD	Max
GC	Accuracy	0.976	0.985	0.984	0.002	0.986
	Precision	0.743	0.839	0.835	0.021	0.861
	Recall	0.756	0.850	0.844	0.023	0.872
	F1	0.755	0.843	0.839	0.020	**0.861**
LM	Accuracy	0.983	0.991	0.991	0.001	0.992
	Precision	**0**	0.533	0.485	0.174	0.843
	Recall	**0**	0.164	0.148	0.075	0.275
	F1	**0**	0.242	0.211	0.092	0.320
FE	Accuracy	0.937	0.966	0.965	0.006	0.970
	Precision	**0**	0.528	0.512	0.271	0.969
	Recall	**0**	0.074	0.101	0.090	0.251
	F1	**0**	0.132	0.146	0.119	0.332
RPB	Accuracy	0.904	0.944	0.941	0.009	0.953
	Precision	0.284	0.790	**0.769**	0.083	0.893
	Recall	0.012	0.546	**0.500**	0.113	0.600
	F1	0.023	0.629	0.593	0.102	**0.688**

Both classifiers presented the *precision* values greater than the *recall* ones. Both have good *F1* values for *GC* and *RPB*. However, for some of the tested hyperparameters, the models using *GBM* presented a lower performance than the *RF* models. For the bad smells *FE* and *LM*, the *minimum* value reported was 0 for three out of the four performance metrics.

4.8 Comparison

Regarding the first two research questions, we compared the best representative model for each classifier. That is, the model with the best result by the criteria of the *F1* score. Tables 10 to 13 present the results for the selected models. On the left side of these tables, the results are related to the evaluation using the Cross-validation Technique, as described in Section 3.5. On the right side, the results are related to the evaluation using 20% of the dataset that has never been used as training (Test Data). The first column (C.) identifies the classifiers, followed by the respective evaluation metrics: *Accuracy (Acc)*, *Precision (P)*, *Recall (R)*, and *F1*.

Note that the evaluation of imbalanced datasets must be focused on the class with fewer instances (smelly ones), otherwise it can lead us to misinterpretation. All classifiers performed well by the perspective of the *accuracy* metric. However, we need to take a deep look on the other metrics. Observing the metrics that consider the positive class, i.e, how many of the bad smells detected are true (*precision*), and how many of the total of existing bad smells were detected (*recall*), we can properly evaluate them. *F1* provides a single balanced value.

Focusing on comparing the cross-validation and test results, we observe in Table 10 that the best classification methods for *GC* detection are *Gradient Boosting Machine (GBM)* and *Random Forest (RF)*, with *F1* of 0.859, 0.861, respectively. *GBM* presented a better *precision* (0.860), while *RF* a better *recall* (0.910). Besides the best classifiers, *Logistic Regression (LR)* presented the best *recall* among all classifiers (0.976), but simultanetly the worst *precision* (0.574). The *LR* classifier was able to detect almost all instances of *GC*, but the number of false positives was very high.

Characteristics from the results on cross-validation were preserved on the test results. *LR* kept the best *recall* and the worst *precision*, while *RF* and *GBM* were the best methods, with a little difference in the *F1* values, with 0.851 and 0.868, respectively.

Table 10: Comparison of God Class Detection

C.	God Class							
	Cross-Validation				**Test Data**			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
NB	0.964	0.618	0.697	0.655	0.966	0.640	0.716	0.676
LR	0.963	0.574	**0.976**	0.723	0.962	0.566	**0.982**	0.718
MLP	0.978	0.762	0.797	0.777	0.978	0.794	0.730	0.760
DT	0.982	0.799	0.836	0.816	0.981	0.805	0.807	0.806
KNN	0.975	0.762	0.724	0.742	0.976	0.763	0.730	0.746
RF	0.985	0.815	**0.910**	**0.859**	0.984	0.798	**0.911**	**0.851**
GBM	0.986	**0.860**	0.862	**0.861**	0.987	**0.879**	0.856	**0.868**

For *LM* detection, the performance is depicted in Table 11. The best classifier was *RF* with *F1* of 0.381. *GBM* presented a better *precision* (0.472), while *LR* presented a better *recall* (0.848). However, both failed to obtain a good performance for the other metrics. That is, *LR* performed poorly in terms of *precision* (0.069), and *GBM* in terms of *recall* (0.243). Finally, even the best classifier, *Random Forest (RF)*, performed badly on the detection of this bad smell.

Table 11: Comparison of Long Method Detection

C.	Long Method							
	Cross-Validation				**Test Data**			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
NB	0.976	0.176	0.472	0.256	0.977	0.176	0.464	0.255
LR	0.900	0.069	**0.834**	0.128	0.901	0.068	**0.848**	0.126
MLP	0.990	0.415	0.155	0.213	0.991	0.394	0.067	0.115
DT	0.977	0.199	0.549	0.292	0.978	0.204	0.580	0.302
KNN	0.991	0.372	0.098	0.154	0.991	0.440	0.124	0.193
RF	0.989	0.383	0.381	**0.381**	0.989	0.372	0.407	**0.389**
GBM	0.991	**0.472**	0.243	0.320	0.991	**0.444**	0.237	0.309

Again, the characteristics from the results on cross-validation were preserved on the test results. *LR* kept the best *recall* and the worst *precision*, and *GBM* was the best method in terms of *precision* (0.444), being closer to the *KNN* values than in cross-validation, with a *precision* of 0.440. Finally, the best *F1* was achieved by *RF*. *FE* detection was also poor, as we see in Table 12. The best classifiers were *RF* and *GBM*, with *F1* values of 0.351 and 0.332, respectively. On the other hand, different from the *LM* detection, both performed better in terms of *precision* (0.694 and 0.462) for *GBM* and *RF*, respectively. In the case of *GBM*, it performed even better on the Test Data, with a *precision* of 0.738.

Table 12 shows that *MLP* on Test Data presented 0 performance in terms of *precision*, *recall*, and *F1*. This occurred because the model predicts all test instances as non-smelly and, consequently, in terms of detecting the bad smell, the performance was null.

Table 12: Comparison of Feature Envy Detection

C.	Feature Envy							
	Cross-Validation				**Test Data**			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
NB	0.916	0.098	0.175	0.126	0.915	0.092	0.159	0.117
LR	0.759	0.088	**0.647**	0.156	0.762	0.093	**0.659**	0.163
MLP	0.923	0.019	0.145	0.034	0.965	0*	0*	0*
DT	0.950	0.299	0.343	**0.319**	0.950	0.314	0.354	**0.333**
KNN	0.964	0.321	0.039	0.069	0.964	0.375	0.044	0.079
RF	0.964	0.457	0.285	**0.351**	0.963	0.462	0.317	**0.376**
GBM	0.970	**0.694**	0.219	**0.332**	0.970	**0.738**	0.244	**0.367**

Table 13: Comparison of Refused Parent Bequest Detection

C.	Refused Parent Bequest							
	Cross-Validation				**Test Data**			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
NB	0.892	0.347	0.242	0.284	0.885	0.340	0.246	0.285
LR	0.772	0.236	**0.699**	0.352	0.767	0.245	**0.717**	0.366
MLP	0.937	0.727	0.467	0.567	0.936	0.729	0.498	0.600
DT	0.924	0.570	0.585	0.577	0.921	0.585	0.570	0.577
KNN	0.912	0.508	0.329	0.399	0.903	0.471	0.277	0.349
RF	0.950	**0.825**	0.566	**0.670**	0.945	**0.810**	0.542	**0.649**
GBM	0.953	**0.828**	0.589	**0.688**	0.948	**0.829**	0.562	**0.670**

For the last bad smell, *RPB*, we observe in Table 13 a positive performance. The best classifiers were *XGB* and *RF*, with both reaching almost a 0.7 *F1* score. Note that *precision* was very high for both, with values of 0.828 and 0.825, respectively. This means that, even if they missed some detections, the correct detection was much higher. This is a good result, since it avoid reporting false positives and flooding the interface or log file in which the results are outputted. We can also observe for this smell that *DT* and *MLP* also reported a good result, both with about 60% of the *F1* score. Finally, given the explained results, we summarize the answers for the first two research questions as follows.

RQ1: *In this study, the models achieved a good F1 performance for the bad smells GC and RPB, and bad performance for the bad smells LM and FE.*

RQ2: *The difference between classifiers, reported by F1 measure, can be of more than ten times. For example, for FE, MLP achieved 0.034 and RF 0.351. The bad smell with less divergence on detection performance was GC, with a difference between the maximum and the minimum of about 0.2 (NB: 0.655 / GBM: 0.861). We can observe that XGB and RF performed better on the detection of all smells.*

4.9 Models Interpretation

In this section, we discuss the third research question (RQ3). The best performance was achieved by two models that uses the ensemble technique. Unfortunately, this kind of meta-algorithm increases the complexity of the model, impacting on how much we can interpret them, given the thousands of rules that are usually created. Beyond the performance of the models, recently, many advances have been made towards the understanding on how to explain the models' predictions [5, 46, 50, 51]. In this work, we applied a unified

approach called SHAP [34], which connects game theory and local explanations to explain the output of machine learning models. There are several applications provided by this approach, but we focus on two: (i) to understand how the software metrics contributes to detect a bad smell, including how much they were responsible for the increase or decrease of the probability of an instance being classified as smelly; and, (ii) to provide practitioners with the motivation for detecting bad smells, instead of only indicate if an instance has a bad smell or not, like many tools [17, 39, 54].

Let's consider a single class instance from the dataset. We performed the classification to detect if it is a *GC*. Figure 1 (A) shows the importance of the top five features with negative values, i.e, these features are telling us that the class should be considered non-smelly. Otherwise, Figure 1 (B) shows the importance of the top five features with positive values, i.e, these features are telling us the class should be classified as a *GC*. The horizontal axis on both graphics present the proportion of contribution for each feature. We see in Figure 1 (A) that *TCC* is the metric which contributes more, with about 30%, for a negative classification (non-smelly). While in Figure 1 (B), we see that *WMC* is the metric which contributes the most, with about 80% for a positive classification (smelly). Moreover, note that negative contributions are more distributed than positive contributions, in which *WMC* dominates. To discover which features have more impact, we can analyze both contributions together. As presented in Figure 1 (C), the top five features on this detection were: *WMC*, *TCC*, *LOC*, *LEN*, and *LOD*. The remaining of this section discusses the two software metrics found with more importance for each bad smell detection in general, not only for a single prediction. We performed this analysis by applying SHAP on thousands of predictions for each bad smell. The single contributions were added and grouped by the software metrics. Then, this aggregation was ranked in descending order considering the total contribution.

God Class. *WMC* is the metric which more contributes to the detection of this bad smell, followed by *LOC* and *LCOM*. It means that complexity metrics is better to detect a *GC*, instead of only size ones, e.g., *LOC*, although *LOC* is still a very important metric.

Refused Parent Bequest. *DIT* is the metric which contributes the most to the detection of this smell, followed by *CYC*. As *DIT* calculates the depth of inheritance tree, we expected that the deeper a class is in the inheritance, the more likely it will start ignores the behavior of its predecessors.

Long Method. *LOC* is the metric which more contributes to the detection of this bad smell, followed by *startLine*. Indeed, the size of the method is very important to detect a *LM*, as its name suggests. The metric *startLine* is a surprise, and the analysis provided us with a information that, the more the method is at the end of the file, it contributes most in the detection of this smell. That is, if the *startLine* metric has a low value, the feature does not provide much contribution. Otherwise, if it is high, the contribution is high.

Feature Envy. *RFC* is the metric which more contributes to the detection of this bad smell, followed by *startLine*. This indicates that, if a method has high interaction with others, probably from other classes, this method may be envying some of these classes. Same as with *LM*, the *startLine* reported a good contribution, primarily when a method is positioned at the file bottom.

Through this analysis, we can answer the last research question as follows.

> ***RQ3**: For each bad smell, we found different ranks of metrics. This was expected due to the different nature of the bad smells evaluated on this study. For the smells at method level, the startLine metric appeared on the top two in terms of contribution.*

5 THREATS TO VALIDITY

In this section, we discuss the threats that may have affected this study and what we have performed to try to mitigate them.

Conclusion Validity. With respect to the conclusion validity, the performance metrics used may have led us to false conclusions. However, we applied well-known metrics from machine learning evaluation and information retrieval. We also followed commonly used procedures, like random state, cross-validation, and two-phase evaluation, with a part of the data being never seen in the training.

Internal Validity. Regarding the internal validity, the partition of datasets to execute the experimental phases may have lead us to a selection bias, by providing for the best classifiers found the instances that are easier to classify. To mitigate this threat, we performed these divisions, including in the cross-validation, by defining the same state, which reproduces for all classifiers, the same instances to be evaluated.

Construct Validity. To compare the classification methods, we used only one implementation for each classifier, which can lead us to a mono-method bias. However, these implementations were all provided by two heavily used libraries for machine learning: scikit-learn[5] and XGBoost[6]. Thus, we relied on the best known or optimal methods for the models who performed the detections.

For the ground truth creation, we relied on the combination of detection results from five tools. As many tools are based on software metrics, this could bias our feature contribution analysis. However, we try to mitigate it by applying a different way to calculate feature importance, using the *SHAP* technique. Different from other methods, *SHAP* is more robust as its internal method of evaluating features considers all possible combinations to build the aproximation models.

External Validity. Regarding the generalization issues, the 20 systems used to build the datasets may not represent the entire space of applications. To improve the generalization power, we used systems with different domains and sizes, including large frameworks from industry. Besides, these systems belong to a set of well know collection applied on many empirical studies [53]. We also performed the detection only for Java systems. The reason is mainly because most detection tools are developed for this programming language. With respect to metrics, they can be applied to other object-oriented programming languages.

6 RELATED WORK

Several detection techniques and tools have been proposed in the literature. Recently, the adoption of machine learning techniques to detect bad smells became a trend [20]. Khomh et al. proposed a

[5] scikit-learn.org
[6] xgboost.readthedocs.io

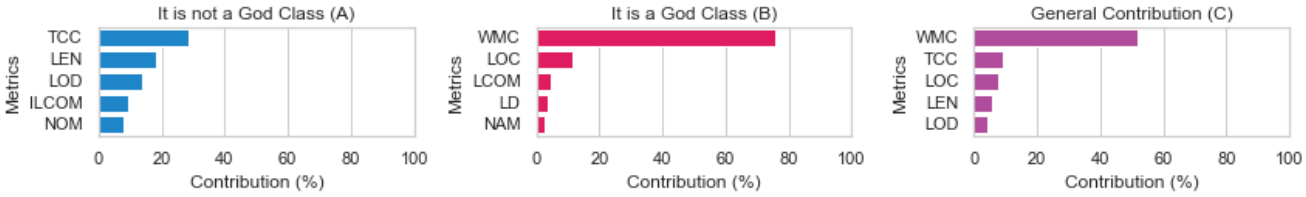

Figure 1: Features Contribution

Bayesian approach which initially converts existing detection rules to a probabilistic model to perform the predictions [27]. Khom et al. [28] extend [27] by the introduction of Bayesian Belief Networks, improving the accuracy of the detection. Maiga et al. proposed an SVM-based approach that uses the feedback information provided by practitioners [35, 36]. Amorim et al. [3] presented an experience report on the effectiveness of Decision Trees for detecting bad smells. They choose these classifiers due to their interpretability [3]. Thus, most of the proposed works focus on only one classifier. They were also trained in a dataset composed of few systems and, consequently, the results may be positive towards their aproach due to overfitting.

Fontana et al. evaluated different machine learning algorithms on a set of different systems [21]. Their work was later extended and refined, providing a larger comparison of classifiers [20]. The notorious impact of this work was the incredible performance reported. Even naive algorithms were able of achieving great results using a small training dataset. This draws attention to possible drawbacks and limitations of their work, which was later reported by Di Nucci et al. [15]. They replicated the study and verified that the reported performance was highly biased by the dataset and the procedures adopted, such as unrealistic balanced dataset, in which one third of the instances were smelly.

Table 14 presents a comparison summary between similar previous work [15, 20] and our work. The columns are related to each work, respectively. The rows present characteristics of the studies, such as which bad smells was studied. In our work, we aim to contribute with empirical evidences on the use of machine learning algorithms for bad smells detection, by overcoming the known limitations [15].

The following considerations can also be made. (i) Our dataset, as described in Section 2, is very close to the real world, being extremely imbalanced, in which less than 5% of the instances are considered as smelly. We observe in the last column of Table 14 that besides the lower number of systems, the number of instances on our dataset is much bigger than on other studies. (ii) Our features were analyzed, and only two strong correlations have been found and discarded. As shown in Table 14, on the *Features* row, we use less metrics. However, the feature selection of a replication study [15] detected that most of the 143 features were high correlated and were discarded correctly. (iii) We found that for the smells at method level, it is harder to obtain good performance.

Table 14: Summary Comparison

	[20]	[15]	This work
Systems	74		20
Sample Size	1,680*	3,360**	267,000
Bad Smells	Data Class, God Class, Feature Envy, Long Method		God Class, Feature Envy, Long Method, Refused Parent Bequest
Detection Tools (Ground Truth)	iPlasma, PMD, Fluid Tool, Antipattern Scanner, Marinescu[38]		JDeodorant, JSpirit, PMD, DECOR, Organic
Ground Truth Validation	Manual (Entire sample)		Manual (Statistical sample)
Algorithms (no variations)	Decision Trees Rule Based Random Forest Naive Bayes SVM/SMO		Decision Trees Random Forest Naive Bayes Logistic Regression KNN Multilayer Perceptron
Boosting	Adaboost (for all algorithms)		XGBoost (GBM with decision trees as base classifiers)
Features	143		30
Feature Selection	No	Yes	Yes
Optmization	Grid Search		Random Search

** One dataset for each bad smell, each one with 420 instances*
*** The replication study claims to have duplicated the original dataset. This number was calculated based on this information and is not present in the original work.*

7 CONCLUSION AND FUTURE WORK

This work presented the evaluation of seven different classifiers employed on models with several different parametrizations, to perform detection of four bad smells. The evaluation was performed on a dataset built using twenty different systems, largely used both in industry and in academia, with an imbalanced distribution, as expected in real scenarios where the systems contain a few numbers of smelly instances. Two machine learning approaches, *Random Forest* and *Gradient Boosting Machine* achieved good performance, on both cross-validation and test phases, for two bad smells, the *God Class* and *Refused Parent Bequest*.

Furthermore, we provided an explanation for the detections, interpreting the output predictions based on the feature contributions and on a state-of-the-art approach called *SHAP*. We found which metrics contribute more to each bad smell detection. As further work, we plan to evaluate which features, if discarded, improves the performance of the model without causing overfitting. We also aim to improve the detection of *Feature Envy* and *Long Method*, which were the bad smells whose models did not perform well; evaluate the algorithms with other bad smells; and prototype a detection tool which also provides a visual explanation, inspired by Figure 1.

8 ACKNOWLEDGEMENTS

This research was partially supported by Brazilian funding agencies: CNPq (Grant 424340/2016-0), CAPES, and FAPEMIG (Grant PPM-00651-17).

REFERENCES

[1] 2020. Detecting Bad Smells with Machine Learning Algorithms - Complementary Web Page: dvscross.github.io/BadSmellsDetectionStudy.

[2] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *European Conference on Software Maintenance and Reengineering (CSMR)*. 181–190.

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

[4] M. Aniche. 2015. *Java code metrics calculator (CK)*. Available in github.com/mauricioaniche/ck/.

[5] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and KR. Müller. 2010. How to Explain Individual Classification Decisions. *Journal of Machine Learning Research (JMLR)* 11 (2010), 1803–1831.

[6] H. Barkmann, R. Lincke, and W. Löwe. 2009. Quantitative evaluation of software quality metrics in open-source projects. In *Int'l Conf. on Advanced Information Networking and Applications Workshops (AINA)*. 1067–1072.

[7] J. Bergstra and Y. Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research (JMLR)* 13 (2012), 281–305.

[8] J. M. Bieman and BK. Kang. 1995. Cohesion and reuse in an object-oriented system. *Software Engineering Notes (SEN)* 20 (1995), 259–262.

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

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

[11] G. C. Cawley and N. LC. Talbot. 2010. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research (JMLR)* 11, Jul (2010), 2079–2107.

[12] 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.

[13] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *Transactions on Software Engineering* 20 (1994), 476–493.

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

[15] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In *Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. 612–621.

[16] 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 Int'l Conf. on Evaluation and Assessment in Software Engineering (EASE)*.

[17] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *Int'l Conf. on Software Engineering (ICSE)*. 1037–1039.

[18] F. A. Fontana, V. Ferme, and S. Spinelli. 2012. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the International Workshop on Managing Technical Debt*. 15–22.

[19] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *Proceedings of the International Workshop on Emerging Trends in Software Metrics (WETSoM)*. 44–53.

[20] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[21] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach (ICSM). In *Int'l Conf. on Software Maintenance*. 396–399.

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

[23] S. Herbold, J. Grabowski, and S. Waack. 2011. Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering* 16, 6 (2011), 812–841.

[24] M. Hitz and B. Montazeri. 1995. *Measuring coupling and cohesion in object-oriented systems*.

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

[26] F. Khomh, M. Di Penta, YG. Guéhéneuc, and G. Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[27] F. Khomh, S. Vaucher, YG. Guéhéneuc, and H. Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *Int'l Conf. on Quality Software*. 305–314.

[28] F. Khomh, S. Vaucher, YG. Guéhéneuc, and H. Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

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

[30] R. Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conferences on Artificial Intelligence(IJCAI)*, Vol. 14. 1137–1145.

[31] M. Lanza and R. Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

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

[33] W. Li and S. Henry. 1993. Object-oriented metrics that predict maintainability. *Journal of systems and software* 23, 2 (1993), 111–122.

[34] S. M. Lundberg and SI. Lee. 2017. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems (NIPS)*. 4765–4774.

[35] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, and E. Aimeur. 2012. SMURF: A SVM-based Incremental Anti-pattern Detection Approach. In *Working Conference on Reverse Engineering (WCRE)*. 466–475.

[36] 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.

[37] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. 2005. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*. 77–80.

[38] R. Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *In Proceedings of the Int'l Conf. on Software Maintenance (ICSM)*. 350–359.

[39] N. Moha, YG. Gueheneuc, L. Duchien, and AF. Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *Transactions on Software Engineering* 36, 1 (2009), 20–36.

[40] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society* 24, 1 (2018).

[41] R. M. O'brien. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity* 41, 5 (2007), 673–690.

[42] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[43] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the Int'l Conf. on Automated Software Engineering (ASE)*. 268–278.

[44] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2015. Landfill: An open dataset of code smells with public evaluation. In *Working Conference on Mining Software Repositories (MSR)*. 482–485.

[45] J. R. Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.

[46] M. T. Ribeiro, S. Singh, and C. Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*. 1135–1144.

[47] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. 2010. Building Empirical Support for Automated Code Smell Detection. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

[48] T. Sedano, P. Ralph, and booktitle=Proceedings of the Int'l Conf. on Software Engineering pages=130–140 year=2017 Péraire, C. [n. d.]. Software development waste.

[49] M. Stone. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)* 36, 2 (1974), 111–133.

[50] E. Strumbelj and I. Kononenko. 2010. An Efficient Explanation of Individual Classifications Using Game Theory. *J. Mach. Learn. Res.* 11 (2010), 1–18.

[51] E. Štrumbelj and I. Kononenko. 2014. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems* 41, 3 (2014), 647–665.

[52] G. Suryanarayana, G. Samarthyam, and T. Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

[53] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference (APSEC)*. 336–345.

[54] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *Int'l Conf. of the Chilean Computer Science Society (SCCC)*. 1–6.

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