Applying Machine Learning to Customized Smell Detection: A Multi-Project Study

Daniel Oliveira
doliveira@inf.puc-rio.br
Pontifical Catholic University
Rio de Janeiro, RJ

Wesley K. G. Assunção
wesleyk@utfpr.edu.br
Federal University of Technology
Toledo, PR

Leonardo Souza
leo.sousa@sv.cmu.edu
Carnegie Mellon University
Silicon Valley, CA

Willian Oizumi
woizumi@inf.puc-rio.br
Pontifical Catholic University
Rio de Janeiro, RJ

Alessandro Garcia
afgarcia@inf.puc-rio.br
Pontifical Catholic University
Rio de Janeiro, RJ

Baldoino Fonseca
baldoino@ic.ufal.br
Federal University of Alagoas
Maceió, AL

ABSTRACT

Code smells are considered symptoms of poor implementation choices, which may hamper the software maintainability. Hence, code smells should be detected as early as possible to avoid software quality degradation. Unfortunately, detecting code smells is not a trivial task. Some preliminary studies investigated and concluded that machine learning (ML) techniques are a promising way to better support smell detection. However, these techniques are hard to be customized to promote an early and accurate detection of specific smell types. Yet, ML techniques usually require numerous code examples to be trained (composing a relevant dataset) in order to achieve satisfactory accuracy. Unfortunately, such a dependency on a large validated dataset is impractical and leads to late detection of code smells. Thus, a prevailing challenge is the early customized detection of code smells taking into account the typical limited training data. In this direction, this paper reports a study in which we collected code smells, from ten active projects, that were actually refactored by developers, differently from studies that rely on code smells inferred by researchers. These smells were used for evaluating the accuracy regarding early detection of code smells by using seven ML techniques. Once we take into account such smells that were considered as important by developers, the ML techniques are able to customize the detection in order to focus on smells observed as relevant in the investigated systems. The results showed that all the analyzed techniques are sensitive to the type of smell and obtained good results for the majority of them, especially JRip and Random Forest. We also observe that the ML techniques did not need a high number of examples to reach their best accuracy results. This finding implies that ML techniques can be successfully used for early detection of smells without depending on the curation of a large dataset.

SBES '20, October 21–23, 2020, Natal, Brazil
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8753-8/20/09...$15.00
https://doi.org/10.1145/3422392.3422427

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**.

KEYWORDS

code smell, code smell detection, software quality

ACM Reference Format:
Daniel Oliveira, Wesley K. G. Assunção, Leonardo Souza, Willian Oizumi, Alessandro Garcia, and Baldoino Fonseca. 2020. Applying Machine Learning to Customized Smell Detection: A Multi-Project Study. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422427

1 INTRODUCTION

Code smells are considered symptoms of poor implementation choices, which make software systems hard to maintain [20]. Due to their harmfulness to software quality [1, 28, 50], code smells should be detected and removed as early as possible along the software lifecycle. Although removing code smells is of paramount importance to keep the internal quality of a software, their detection is not an easy task. Their detection is mostly subject to developer judgment. In fact, identifying code smells is a subjective activity [24]. Developers have the knowledge to confirm the harmfulness of a smelly structure [13]. This knowledge varies according to the developer's experience, skills, and mastery of the source code being analyzed.

Existing literature suggests different strategies to detect code smells. The vast majority of these detection strategies are based on metrics and their respective thresholds [3, 32, 34]. These strategies tend to analyze each code fragment and employ some previously defined thresholds to classify the fragment as host (or not) of a specific smell. This difficulty stems from the fact that the operationalization (*i.e.*, the threshold definition) of the strategy for detecting each smell type requires proper reasoning. Such an operationalization cannot be solely based on finding metrics and thresholds in compliance with the conceptual definition of a smell type. The operationalization also needs to be customized by considering various contextual information of the projects, to which only the developer has access.

Preliminary studies have analyzed the use of machine learning (ML) techniques as a promising way to detect code smells [2, 17, 18, 24, 25, 38]. These previous studies evaluate the use of training datasets containing numerous code examples annotated as smelly or non-smelly code by developers. From these training datasets, the ML techniques generate detection models that can successfully detect similar smells to the ones used for the training. However, these techniques are hard to be customized, for example, the detection of only certain types of code smell that are relevant for developers of specific projects. Yet, ML techniques usually require numerous code examples to be trained. Obtaining such many instances with desired properties to compose a relevant dataset can be time-consuming, leading to late detection of code smells. Thus, a prevailing challenge is the early customized detection of code smells taking into account the typical limited training data.

To overcome the described limitations, this paper reports a multi-project study in which we collected code smells that were actually refactored by developers, differently from studies that infer code smells based on metrics and thresholds. This study relies on ten active projects, with different sizes and belonging to distinct domains, and six types of code smells. We chose smell types that cover different system scopes, such as classes, methods, fields, and parameters. The composed dataset was used in the training and evaluation of seven ML techniques in terms of the accuracy regarding early detection of code smells. The accuracy was computed based on the traditional F-Measure [27]. In addition to existing strategies, since we take into account such smells that were considered as important by developers, the ML techniques are able to customize the detection in order to focus on smells observed as relevant in the investigated systems.

The results pointed out that, when considering only relevant smells, the ML techniques have similar behavior for the same smell type. Besides that, they also have good support for detecting code smells in projects with different sizes, once they do not need a high number of examples to reach high results. By the results obtained and analysis performed, our study led to the following findings:

- *Smell types vs. ML techniques.* When detecting harmful smells, the smell types have more influence on accuracy than the ML techniques themselves.
- *Effect of the metrics on the customized smell detection.* Particularities of each code smell type affect the accuracy of the ML techniques. For example, the metrics (features) that best represent the smell.
- *The subjectivity on detecting customized smells affect the accuracy of the ML techniques.* Smells with more subjective definitions, i.e more complex, tend to obtain less accuracy since the training set will have a higher variation in the values of the features.
- *The ML techniques using a small set of curated examples (based on previous refactorings of developers) can successfully support early customized detection of code smells.* When using on datasets that well-represent what developers consider as relevant in practice, ML techniques are able to, significantly, early detect code smells.

The contributions of this work are as follows. First, different from previous studies, we focused on assessing the use of ML techniques only for detecting smells that ended up being refactored by a developer. The refactoring of the smelly code indicates that the developer, either consciously or not, confirmed the relevance of a smell. Those smells can be considered relevant to the program as their removal helped the developer to achieve his maintenance goal. Second, we assessed the accuracy of six well-known ML techniques on detecting code smells using our customized dataset, considering different samples of training, whereas we gradually increase the number of examples used to perform ML technique training. Third, we make publicly available[1] our dataset for future studies and replication.

The remaining of this document is structured as follows. Section 2 describes the background related to code smells and ML techniques. Related work is described in Section 3. Section 4 details the design of our study. Section 5 presents all the results and analysis as well as the answers to our research questions. Section 6 describes the threats and limitations of the study. Finally, Section 7 presents the concluding remarks and discusses future work.

2 BACKGROUND

Our study encompasses two research topics, namely, code smells and ML techniques. The next sections describe the six code smells we considered and the seven ML techniques, respectively.

2.1 Code Smells

Poor code structures are often represented by the so-called code smells [19]. A code smell is considered a symptom of a poor implementation choice. As a consequence, software maintenance requires additional cost and effort on understanding and re-structuring the smelly code [8, 9, 30]. Due to their harmfulness to software maintainability [1, 28, 50], code smells should be detected and removed as early as possible along the software lifecycle.

There are several types of code smells described in the literature [19]. Each one characterizes a recurring poor code structure and affects a specific scope of program elements. For our study, we selected the six smell types listed in Table 1. These smell types are related to bad design decisions or reflect important maintainability aspects [49]. The first column represents the smell type name. The second column presents a brief description of the type. We have chosen these smell types due to the different scopes of a program affected by them, namely classes, methods, fields, and parameters.

Table 1: Types of Code Smells Investigated in this Study

Name	Description
Complex Class (CC)	Classes that involve a lot of different but related parts.
Class Data Should be Private (CDSBP)	Classes that expose its attributes unnecessarily.
God Class (GC)	Classes that tend to centralize the intelligence of the system.
Lazy Class (LC)	Classes that do not do enough.
Spaghetti Code (SC)	Code that has a complex and tangled structure.
Speculative Generality (SG)	Unused classes, methods, fields or parameters created to future features that never get implemented.

[1]https://smelldetection.github.io/

Previous studies proposed strategies to detect code smells [8, 15, 16, 33, 37, 39], including the use of ML techniques [17, 24, 29, 35]. However, these strategies are mostly not representative of how developers perform smell detection, once developers have particular ways to derive a detection strategy. In this sense, some studies suggest that developers customize their detection strategies according to their own previous experience in recognizing which smells are harmful and should be refactored [25, 26]. As aforementioned, smell detection takes into account the particular ways to derive a detection strategy. As a consequence, particular customizations largely differ from others and it is rare or even impossible to derive a single detection strategy that is acceptable in every software project.

In fact, empowering smell detection strategies with customization can help developers and companies to consider code smells that are indeed harmful according to their quality standards. Therefore, customized strategies can identify and report to developers only smells in which they are interested. Also, constant warnings of non-customized detection strategies can cause a waste of time on the inspection of irrelevant smells. Strategies without customization also hinder the developer concentration on harmful smells, or camouflage smells that are considered more harmful according to the developers' perception.

2.2 ML Techniques

To investigate the customization of smell detection, we analyzed seven ML techniques frequently used in literature [24, 25]. These techniques involve different data analysis approaches, such as decision trees, regression analysis and based-rule analysis that are responsible to create the classifier models. The seven ML techniques are listed below:

Naive Bayes: A probabilistic classifier based on the application of Bayes' theorem [36]. This technique is highly scalable and completely disregards the correlation between the variables in the training set. This classifier describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

Support Vector Machine (SVM): An implementation of integrated software for the classification of support vectors [47] that analyzes the data used for classification and regression analysis. SVM assigns new examples to one of the two categories introduced in the training set, making it a non-probabilistic binary linear classifier. To make this classification, SVM creates classification models that are a representation of examples as points in space. These points are mapped in such a way that the examples in each category are divided by a clear space that is as broad as possible. Each new instance is mapped in the same space and predicted as belonging to a category based on which side of space they are placed.

Sequential Minimal Optimization (SMO): An implementation of John Platt's minimal sequential optimization algorithm to train a support vector classifier [43]. In other words, SMO is a technique for optimizing the SVM training expediting the training process and making it less complex. For that, SMO breaks the problem to be solved into a series of smallest possible sub-problems, which are solved analytically.

OneRule (OneR): A classification technique that generates a rule for each predictor in the data. Then, it selects the rule with the lowest total error as its "single rule" [23]. In order to create this rule, this technical analysis of the training set associating a single data to a specific category based on its frequency; in other words, if a specific *data* is usually classified as *category A*, then a rule is created linking them. After the rules creation, the technique chooses the one with the lowest total error.

Random Forest (RF): A classifier responsible for building numerous classification trees representing a forest with random decision trees [22]. The RF technique adds extra randomness to the model when the tree's creation. Instead of looking for the best feature when partitioning nodes, it looks for the best feature in a random subset of features. This process creates a great diversity, which generally leads to the generation of better models, besides that this diversity also reduces the overfitting effect.

JRip: An implementation of an apprentice of propositional rules [10]. It is based in association rules with reduced error pruning, a very common and accurate technique found in decision tree algorithms. Different from the other algorithms, JRip splits its training stage into two steps, a growing phase, and a pruning phase. The first phase grows a rule by greedily adding antecedents (or conditions) to the rule until the rule is perfect, (*i.e.*, 100% of accuracy). The second phase incrementally prunes each rule and allow the pruning of any final sequences of the antecedents.

J48: A Java implementation of the C4.5 decision tree technique [44]. J48 builds decision trees from a training data set. At each node of the tree, this technique chooses the data attribute that most effectively partitions its set of samples into subsets tending to one category or another. The partitioning criterion is the information gain. The attribute with the highest gain of information is chosen to make the decision. This process is repeated on the smaller partitions.

3 RELATED WORK

Several machine learning techniques have been applied to derive automated smell detection strategies (*e.g.*, [17, 24, 25]). In addition to the detection of code smells, such techniques have also been applied in close activities such as prioritizing code smells [42] and detecting architectural smells [11].

To design and evaluate ML-based smell detection strategies, researchers often rely on large datasets of manually validated code smells. These datasets are large to facilitate the learning process, thereby leading to more accurate results. The study described in [35], for example, assessed the accuracy of *Support Vector Machine* (SVM) in the detection of four types of code smell: *Blob*, *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*. The SVM obtained an accuracy of up to 0.74.

In [2], the authors proposed the use of *Decision Tree* technique to detect code smells. The authors used a single dataset containing a huge number of examples validated by a few developers. The results indicate that the *Decision Tree* is able to reach an accuracy up to 0.78. Fontana *et.al.* [17] presented a large study that compares and experiments different configurations of machine learning techniques to detect four code smell types (Data Class, Large Class, Feature Envy, Long Method). To perform the training of these techniques, the authors used a dataset containing several examples of code smells manually validated by a few developers. The *J48* and *Random Forest* obtained the highest accuracy, reaching values up to 0.95. However, a recent study [14] indicates that the dataset used by Fontana [17] had a high influence on the accuracy obtained by the techniques.

Hozano *et al.* [24] analyzed the accuracy and efficiency of ML techniques in the detection of four distinct code smell types (God Class, Data Class, Feature Envy, Long Method). The results indicate that the *Random Forest* is able to reach high accuracy and efficiency when detecting these smell types. Despite presenting important advances regarding the detection of code smells, this study and other ones reported in the literature are far from capturing how developers work on smell detection. The reason is that existing studies rely on the premise that either (i) it is possible to derive a universal strategy based on a large training dataset or (ii) each company will customize strategies for the context of each software project. Nevertheless, both premises are false.

There is evidence that the detection of code smells is highly sensitive to contextual factors, such as the software developer [12, 25]. This happens because each developer may have a particular way of deriving detection strategies for code smell based on previous experiences. Thus, a universal strategy would hardly present satisfactory results in any project. In addition, it is difficult to imagine that developers would spend time validating datasets of code smells for each software project, which makes it difficult to adopt existing techniques.

Another limitation of existing ML-based smell detection strategies is regarding data balancing [14]. This happens because the proportion of samples without code smells is usually much higher than the proportion of samples affected by validated smells. Some researchers [40, 41] investigated whether data balancing techniques are able to improve the accuracy of ML-based smell detection strategies. However, their results indicate that the existing techniques for data balancing are not capable of significantly improving accuracy. Therefore, there is still a need for other ways to improve existing ML-based smell detection strategies.

Given the aforementioned limitations, in this work, we apply and evaluate a new way for the automated customization of early code smell detection strategies. For training the ML algorithms, we take a dataset of code smells that were refactored in practice. We conjecture that such refactorings are strong indicators of relevance as they indicate that the developers actually spent effort on removing the smells. This prevents developers from receiving late warnings about smells that they may consider not harmful. Therefore, following this customization approach, the application of ML-based smell detection strategies can become viable in any project or company that has a history of refactorings carried out by the development team in the past. Even when the number of smell instances is rare.

4 STUDY DESIGN

The goal of our study is to investigate the early customized detection of code smells taking into account the typical limited training data. Based on this goal, we derived two research questions, as follows.

RQ1. How accurate are the ML techniques on customizing the detection of smells? This RQ aims at investigating the accuracy of the seven ML techniques in customizing the detection of six smell types. A customized detection focuses only on smell instances in which developers are interested. These smells are considered more harmful based on developers' perceptions. In this way, the detection will allow the proper removal of these smells. Thus, it is important to investigate techniques that are able to detect harmful smells with high accuracy. Once ML techniques have been considered a promising way to detect code smells [4], these techniques are a strong candidate for customizing smell detection properly.

RQ2. How efficient are the ML techniques for early customized detecting of smells? Our second RQ aims at analyzing the efficiency of the ML techniques in detecting smells, *i.e.*, how accurate a ML technique detects smells whereas we gradually increase the number of examples used to perform its training. Although ML techniques have been considered a promising way to detect code smells, these techniques require code smell examples annotated to perform their training. However, the annotation of a large number of examples may introduce an unfeasible additional time and effort. Hence, it is important to analyze the accuracy of these techniques with a low number of examples used in the training set.

To answer the posed RQs we rely on two aspects: (i) the type of smell analyzed; and (ii) the number of instances used to perform the training of the ML techniques, i.e. the training dataset. The next sections present the subject projects and how we collected and analyze data regarding these two aspects.

4.1 Subject Projects

The instances of code smells considered in the scope of our study (see Table 1) were detected from ten open source Java projects: Apache Ant[2], Apache Derby[3], Apache Tomcat[4], Elastic Search[5], Argouml[6], Apache Xerces[7], Google j2objc[8], Presto db[9], SpringFramework[10] and Achilles[11]. We selected such projects because they have different sizes and are from distinct domains. A mix of domains is an interesting benchmark for companies that have a few projects in their portfolio but operates in multiple domains. Also, they were evaluated by existing smell detection techniques and, which found that their source code contains a variety of suspicious code smells that enable the execution of our study [8].

4.2 Data Collection

Data to answer RQ1. We extracted 200 (100 smelly and 100 non-smelly) code fragments from the analyzed projects for each smell type. The smell detection process was made using a detection tool [8]. This tool is based on a set of metrics and thresholds and has a high overall recall of 81% [15], i.e., it detects the vast majority of existing smells. These 200 code fragments for each smell allow us to observe the behavior of the techniques in a diversity of code smell instances. In addition to detected code smells, we selected

[2]https://ant.apache.org/
[3]https://db.apache.org/derby/
[4]http://tomcat.apache.org/
[5]https://www.elastic.co/
[6]https://argouml.tigris.org/
[7]http://xerces.apache.org/
[8]https://github.com/google/j2objc
[9]https://prestodb.io/
[10]https://spring.io/
[11]http://www.ganttproject.biz

only those that were directly refactored by developers. That is, the examples of smells used in the training dataset and in the evaluation of the ML techniques were evidently relevant for developers. These smells were considered important since they were identified and removed by the developer, so they were harmful and possibly hindered some development tasks. In this way, the tool's threshold was only a starting point to identify the smells. Then, the project's developers indicated the harmful smells. This makes the learning process be based on more relevant smell instances, i.e., not simply based on initial thresholds. Thus, we can verify if the techniques are able to customize their detection for more relevant smell types.

To identify the refactorings that were related to the detected smells, we used the RefMiner [46, 48] tool. RefMiner is widely used in the literature [7, 8, 46, 48]. From this information, we could filter the analyzed smells among those that directly underwent a refactoring. Also, this filter avoids bias regarding a single set of detection metrics/thresholds and ensure a relevant dataset.

Finally, the application of ML techniques requires collecting features for all smell instances. For this task, we used Understand[12], a tool to extract software features. Altogether, 42 features were considered. The complete list of features is publicly available[1]. These features were used during the training process of the seven ML techniques. Once we are addressing smells considered harmful by the developer, these smells may not be aligned with the features proposed by the literature for each smell type [5, 32], since the literature evaluates all instances of smells [4]. Therefore, a high number of features allow the machine learning techniques to evaluate which ones best characterize a smell as harmful. Also, these features cover different information about classes, methods, fields, and parameters, indicating, *e.g.*, the number of lines of the code fragments, relations of complexity within and between elements and several other counters.

Data to answer RQ2. In order to evaluate the accuracy of the early customized detection of ML techniques, we adopted different sizes of training datasets, that were applied incrementally. The dataset of each subject project was split into six subsets of different sizes: 20, 40, 80, 120, 160 and 200 code smell instances. This division was made such that during the evaluation each increment, i.e. new set of instances, also includes the same instances of the preceding set. For example, the second set with 40 instances is composed of all instances of the first set added of more 20 new ones.

To assess the accuracy of the ML techniques, we computed the f-measure that considers both the *recall* and *precision* [27]. To compute these measures, the *true positive (TP)* elements represent the code fragment classified by the ML techniques as a code smell that is, actually, a real code smell. The *false positive (FP)* elements refer to the code fragments wrongly classified as a code smell. Similarly, the *true negative (TN)* represents the code fragments correctly classified as not-smell. Finally, the *false negative (FN)* represents the wrong ones. Based on that, we can compute recall, precision, and f-measure, as described in the equations below. F-measure is widely used in previous studies [17, 24, 38] that assess the ML techniques on detecting code smells.

[12]https://scitools.com/features/

- **Recall (R)** : The Number of code fragments correctly classified as code smells among the total of code smell instances in the data collection.

$$R = \frac{TP}{TP + FN} \tag{1}$$

- **Precision (P)** : The Number of code fragments correctly classified as code smell among the total of code fragments classified as code smell by the ML technique.

$$P = \frac{TP}{TP + FP} \tag{2}$$

- **F-measure**: Harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{3}$$

4.3 Data Analysis

Using the datasets containing the classified (non-)smelly instances and the software features for each analyzed code fragment, we performed two different analyses. Each analysis aims at answering a research question.

Analysis to answer RQ1. Here we used the datasets to analyze the accuracy (in terms of f-measure) of the ML techniques on detecting a specific smell type. For each smell type, we calculated the overall accuracy of each technique by applying a 5-fold cross-validation procedure on the 200 classified instances.

Analysis to answer RQ2. For this analysis, we evaluated the efficiency of the ML techniques, *i.e.*, the accuracy of each ML technique whereas we increment the number of code smells examples used to perform the training of these techniques. In other words, we repeat the accuracy experiment six times, one for each subset of the respective smell. The repetition aimed to guarantee that both, the training and test sets, were composed of equals number of fragments classified as smell or not.

4.4 Implementation Aspects

The ML techniques presented in Section 2.2 were implemented on top of Weka[13] and R Project[14]. Weka is an open source ML software, based Java programming language, containing a plethora of tools and algorithms [21]. R is a free software environment for statistical computing that is widely used for data mining, data analysis, and to implement ML techniques [31].

5 RESULTS AND DISCUSSION

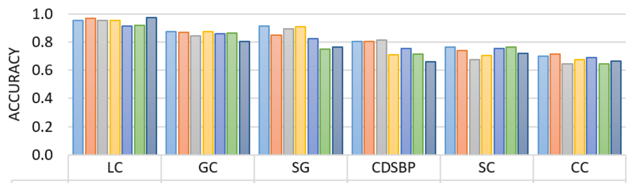
This section presents and discusses the results of our multi-project study. The results are organized in terms of the two research questions presented in the previous section.

5.1 RQ1. How accurate are the ML techniques on customizing the detection of smells?

To evaluate the accuracy of ML techniques in detecting different types of code smells, Figure 1 presents the accuracy when considering the greatest dataset of our study, namely 200 instances of code smells. The use of this dataset allows us to perform an analysis

[13]https://www.cs.waikato.ac.nz/ml/weka/
[14]https://www.r-project.org

of the overall accuracy of the ML techniques in the scenario with the most number of instances for training. In the figure, the *x-axis* is divided per smell and presents sequentially the ML technique used to detect the respective code smell. The *y-axis* describes the values of the accuracy (in terms of *f-measure*) obtained by the ML technique on detecting the respective smell type. To improve readability, we attach the values of the *f-measure* in the table below the bars associated with each smell and highlighted the higher results.



	LC	GC	SG	CDSBP	SC	CC
JRip	0.955	**0.877**	**0.913**	0.807	**0.765**	0.702
RF	0.971	0.869	0.849	0.806	0.742	**0.715**
J48	0.956	0.845	0.894	**0.814**	0.675	0.644
OR	0.955	0.874	0.912	0.710	0.706	0.676
SVM	0.913	0.861	0.826	0.756	0.755	0.690
SMO	0.919	0.867	0.749	0.713	0.764	0.645
NB	**0.975**	0.804	0.763	0.662	0.722	0.666

Figure 1: Overall accuracy reached by the ML Techniques on customized smell detection.

5.1.1 *Overall Accuracy.*

By analyzing Figure 1, we can observe that there is no technique with the best accuracy for all code smell types. This is interesting since previous studies observed that Random Forest reached the highest overall accuracy on detecting smells [17, 24, 38]. However, in our study, we noticed that, when detecting harmful smells, Random Forest did not have the best overall accuracy. Although Random Forest obtained only one best result (for CC), it also obtained closer results when compared with the better ones in the other four types of smell, namely CDSBP, GC, LC, and SC. In summary, all the ML techniques have a similar accuracy when observing each smell type individually. The highest divergence (0,164) is observed in the detection of SG, existing between JRip and Sequential Minimal Optimization (SMO). Finally, all techniques achieved accuracy between 0.6 and 0.8 for the detection of CC and SC.

Finding 1*. When detecting harmful smells, the types of smell have more influence on accuracy than the ML techniques themselves.*

5.1.2 *Accuracy per code smell type.*

In the following paragraphs, we discuss the results of each code smell type individually.

Complex Class. Regarding CC, Random Forest reached the best accuracy, equal to 0.715, while J48 and Sequential Minimal Optimization obtained the lowest values with a slight difference between them. Note that none of the ML techniques reached accuracy above 0.8. We believe this happens because the developer's perception of what is a complex class can vary a lot. This subjectivity in the detection of CC could be so divergent, that ML techniques could not find a proper model that best represents them. In other words, since the identification of CC is hard to be found by our customized detection ML techniques, its identification based on metrics and thresholds certainly is even more complex.

Spaghetti Code. Although a slightly better, the results of accuracy obtained by the ML techniques for the customized detection of SC instances is similar to CC. None of the ML techniques was able to reach accuracy better than 0.8. For the SC smell, J48 reached the lowest accuracy (0.675). On the other hand, the highest accuracy (0.765) was obtained by JRip.

Class Data Should Be Private. For CDSBP, the ML techniques J48, JRip, and Random Forest reached results above 0.8, and the remaining techniques obtained accuracy below this value. Naive Bayes obtained only 0.662. Vector Machine, One Rule, and Minimal Optimization obtained values between 0.7 and 0.8.

Speculative Generality. The ML techniques were able to reach values higher than 0,9 for SG. JRip, once again, reached the highest accuracy, equal to 0.913, followed closely by One Rule that also exceeded 0.9. Sequential Minimal Optimization obtained the worst accuracy, equal to 0.749. An important fact to note is that SG should be difficult to detect whether we look at a single instance at a time, as the ML techniques do because this smell occurs when a developer implements an element (*i.e.*, methods, classes, and fields) that never is used. In other words, it should be necessary to look at this element along different versions to decide whether there is the existence of this smell. This contradicts the good result obtained by some algorithms.

God Class. Differently from the smells previously discussed, the result for GC reached high values. All ML techniques could reach an accuracy higher than 0.8. JRip reached the highest accuracy (0.877). Similarly to CDSBP, the worst result was obtained by Naive Bayes. Here we can note that the combination of different features (*i.e.* metrics) also seems to implicate in the accuracy obtained by the techniques, similar to what we discussed for the CC smell. However, for this code smell, some specific metrics such as Lines of Code, can have higher influence [32, 45]. Besides that, GC is usually associated with high values of the metrics.

Lazy Class. The customized detection of LC is by far the one where the ML techniques obtained the best results. All techniques reached accuracy values better than 0.9. Another difference between the smells previously observed is regarding the Naive Bayes technique, which reached the best result, in contrast with its previous results. It is also possible to observe that Random Forest obtained accuracy slightly close to Naive Bayes. This similarity also occurs between Vector Machine and Sequential Minimal Optimization, besides J48 and JRip.

Finding 2*. Particularities of each code smell type, for example the metrics (features) that best represent them, affect the accuracy of the ML techniques.*

5.1.3 *Smells with low accuracy.*

Our Finding 2 can be used to explain some of the lowest results. Although further studies are needed for a more in-depth analysis, a possible explanation for the cases with a low accuracy may be related to the number of features required to train each ML technique. All techniques received as input the same set of 42 features. However, not all these features contribute to equality to identify the smell. For example, the detection of GC using classic detection strategies relies on two features: lines of code and cohesion [32]; whereas, the detection of CC only relies on one metric: cyclomatic complexity [39]. Nevertheless, the techniques received the same features to detect both smells. In this way, the ML techniques might mistakenly consider different features (i.e., the weight assigned to each metric) as relevant. For example, when the techniques were trained, they probably selected more relevant features to detect GC than to detect CC.

Furthermore, developers among distinct projects might not agree with the relevance of CC instances, as they can associate complexity with different features [24, 25]. This disagreement directly affects the accuracy of the customized detection of smells, once we are considering only harmful smells refactored by developers. For example, different developers can classify the smells as relevant considering different metrics, as already identified in existing studies [26]. Therefore, based on this discussion, we have our third finding.

Finding 3. *Smells with more subjective definitions, i.e more complex, tend to obtain less accuracy, since the training set will have a higher variation in the values of the features caused by the divergence of developers' opinion about the relevance of a smell instance.*

This variation in training is even more intense, as we are working with 42 distinct features. Too many features may cause every training instance in the dataset to appear equidistant from all the other ones. Thus, if the distance between the instances appears to be equally alike, the techniques cannot find meaningful clusters to classify a code fragment as smelly or not-smelly. This scenario may have happened to train the detection of some smells (*e.g.*, CC) but not other ones (*e.g.*, GC). This issue is known as the Curse of Dimensionality [6].

Finally, the detection of GC is more robust than the detection of CC. Since detecting a GC usually requires analysis of more metrics, the ML technique can be less affected by the curse of dimensionality in detecting GC than detecting CC. However, this same explanation cannot be generalized for LC, which uses a simple detection strategy: fewer lines of code than average lines of code of the system. Consequently, another factor plays an important role in the techniques' accuracy. Some detection strategies use metrics that compute the average value across the system. Detection strategies that use the average measurement tend to create techniques with better accuracy. For example, LC and GC use average measurement have high accuracy, whereas CC and SC do not use these averages [5, 32].

Even though we provided possible explanations for the difference of accuracy, we highlight that understanding the mechanisms of ML techniques is not trivial. Yet, unseen factors can help to contribute to the different accuracy among the different smell types.

RQ1 Answer: *When customizing the detection of code smells, most of the ML techniques detected God Class, Speculative Generality, Large Class, and CDSBP with high accuracy. For Speculative Generality and Complex Class, the ML techniques obtained a lower accuracy, but still reached accuracy above 0.7 for most of the ML techniques.*

5.2 RQ2. How efficient are the ML techniques for early customized detecting of smells?

Differently from the previous section, that discussed overall accuracy, here our focus is the analysis of early customized detection of code smells. In this regard, we verify whether the techniques reached high accuracy with a low number of instances required for training. Figures 2 to 8 present the results that support such analysis. These figures represent the efficiency reached by the ML techniques on detecting each smell type. The *x-axis* describes the number of the examples used (20/40/80/120/160/200) in the training phase of the techniques divided per smell, whereas the *y-axis* represents the accuracy values obtained by each ML technique on detecting smells.

We observed that the ML techniques do not follow a unique behavior when the number of analyzed examples increases. Techniques such as Random Forest and Vector Machine had a significant increase in SG detection during the addition of new (non-)smelly instances in the training dataset. In contrast, for J48 a smaller training dataset resulted in the best results of accuracy. This same behavior can be seen in JRip when detecting CC, and Naive Bayes when detecting CDSBP and GC. In general, the ML techniques reached results near to their best results in this study on detecting the respective smell with a low number of examples. Some cases, such as the detection of CC using Naive Bayes, are exceptions, in these cases, using a dataset containing a low number of instances did not reach high results. We can also note that all algorithms did not need more than 20 instances to reach accuracy above 0.8 for LC and GC smells.

Finding 4. *When using on datasets that well-represent what developers consider as relevant in practice, ML techniques are able to significantly early detect code smells.*

The results and analysis presented in this section allow us provide the answer for the RQ2.

RQ2 Answer: *In most cases of our study, the ML techniques did not need a training dataset with numerous examples to reach their best detection results. In fact, the increase in the number of instances did not appear to have a direct relationship with the increase in accuracy for all ML techniques. Interestingly, our results contradict those results found in previous studies [17, 24, 38], which stated that the techniques needed many examples to get good results.*

6 LIMITATIONS AND THREATS TO VALIDITY

The threats to validity and the limitations of our study, along with the ways we mitigate them, are presented next.

6.1 Threats to Validity

This section discusses the threats to validity.

Construct Validity: The datasets that supported our study were built from code fragments collected using rule-based strategies that have a set of metrics and thresholds. These thresholds are threats, once they can bias the techniques learning because of the analyzed smelly fragments were filtered by these thresholds. To lessen this

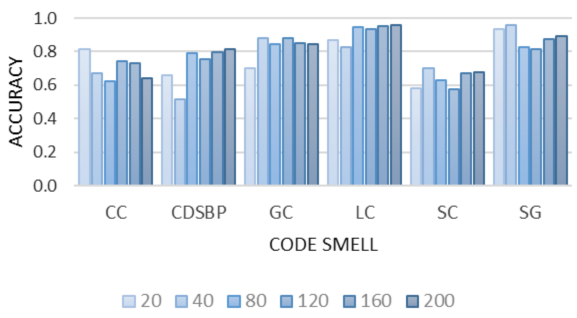


Figure 2: J48 Efficiency

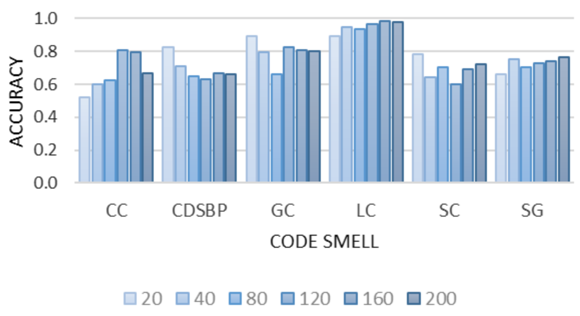


Figure 3: Naive Bayes Efficiency

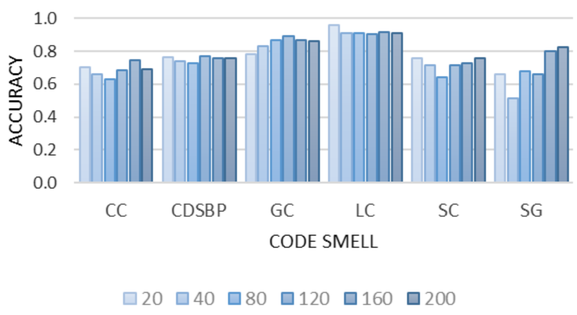


Figure 4: Support Vector Machine Efficiency

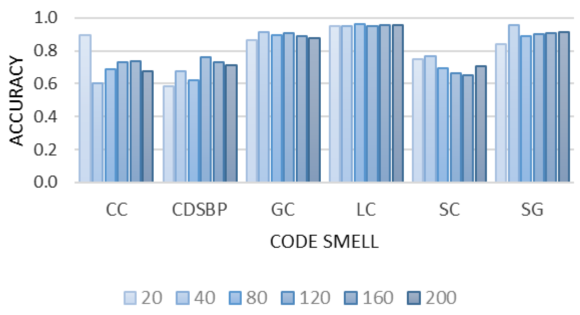


Figure 5: One Rule Efficiency

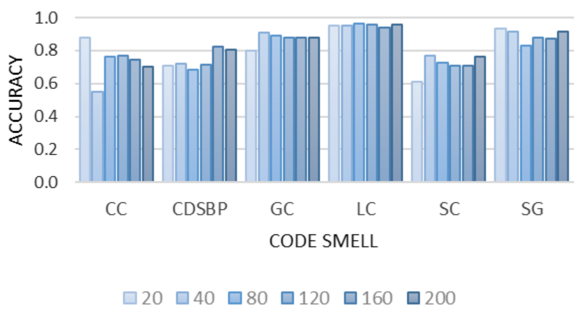


Figure 6: JRip Efficiency

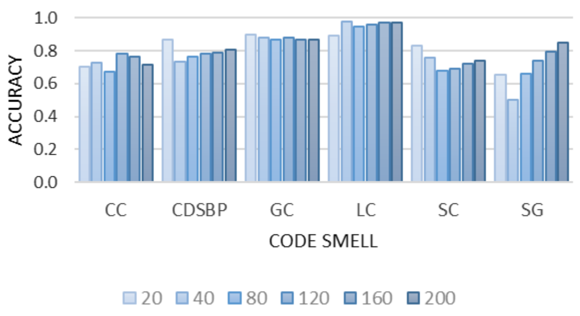


Figure 7: Random Forest Efficiency

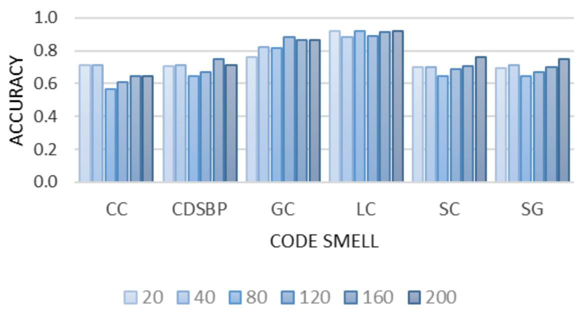


Figure 8: Sequential Minimal Optimization Efficiency

bias, we filter the smells by selecting only those that caught the developer's attention to refactor them.

Internal and External Validity. The use of the Weka package of the R platform to implement the techniques analyzed in our study enabled us to experiment a variety of configurations, which affect the training process of the techniques. In such context, the configurations considered in our experiments may impact the accuracy and efficiency of the techniques. In order to mitigate this threat, we configured all ML techniques according to the better settings defined in [17]. Indeed, [17] performed a variety of experiments in order to find the best adjust for each technique.

As far as external validity is concerned, the code fragments were extracted from ten Java projects. However, although the implementation of these projects presents classes and methods with different characteristics (*i.e.*, size and complexity), our results might not hold to other projects.

6.2 Limitations

This section discusses the limitations found during the study, which will be considered in future studies.

Number of Smells: The catalog of smell types presented in [20] categorizes the smells based on their area of action in the code. Also defining a high number of smell types than those addressed in our empirical study. These additional smells can also harm the quality of the software, making their detection important. However, their detection through machine learning requires the evaluation of code fragments that are suspicious of containing these smells, which leads us to the second limitation.

Evaluated Projects: Ten different projects are currently covered in our dataset. However, all of these projects are open source projects written with the Java programming language. These common characteristics among the chosen projects tend may reduce the variety of particular manifestations of a smell type. A larger dataset, including both closed source and additional open source projects, can expose a wider variety of smell structures.

Classifier Model Customization: We observed that each ML technique did not support general, highly-accurate detection of all smell types. However, the achieved an improvement in their accuracy when are analyzing a subset of specific smell types. This improvement could be related to the classifier model built by the techniques. It is important to note that this model can be improved manually changing the parameters during the technique implementation, or automatically through trial and error. Previous studies [24, 38] suggest that this improvement by customization could also be explored to better detect smells for specifics developers.

Project Sensitive Customization: Better behavior of a ML technique perhaps could also be observed if the training and the detection involves a single software project. Given this narrower scope, we would reduce the number of developers involved in the dataset. Thus, the ML techniques may be able to better adapt themselves during the training process. If we further narrow the scope to the system's modules, we will have code fragments with similar responsibilities and a subset of developers in charge. This change may allow the techniques to customize their detection for the specific concerns being addressed by each module, hopefully further improving their accuracy but in detriment of possibly not having a reasonable number of smelly instances to properly train the model.

7 CONCLUSION AND FUTURE WORK

This study analyzed the accuracy and efficiency of ML techniques for detecting code smells. Firstly, we evaluated the accuracy of the ML techniques for customizing smell detection. Then, we analyzed the efficiency of the ML techniques by evaluating their accuracy according to the number of examples used to perform the training process.

The results indicated that, when detecting harmful smells, the types of smell have more influence on accuracy that the different strategies of the ML techniques. Indeed, particularities of each smell type that represent them affect the accuracy of the ML techniques. That is, ML techniques tend to have low accuracy when detection complex smell types considering only instances relevant for the developers. In this context, JRip and Random Forest reached the highest overall accuracy on detecting smells, meanwhile, Naive Bayes obtained the lowest overall accuracy.

Regarding the techniques' efficiency, we observed a different result from previous studies. In our study, the increase in the number of instances in the training set did not appear to have a direct relationship with the increase in accuracy. Once the ML techniques do not need a high number of examples to reach their best results, the effort to train the techniques is reduced, enabling the use of this technique in projects with different sizes. Also, a reduced number of needed examples allows techniques to early detect smells, enabling the removal of smells at the beginning of the lifecycle of the software.

As future work, we intend to investigate the accuracy of ML techniques on detecting other smell types. In addition, we also intend to replicate this study in controlled scenarios, reducing the analyzed scope per project and, after that, per system's modules. In this way, we expect to identify the behavior of the techniques in more specific contexts.

ACKNOWLEDGMENT

We thank CNPq (grants 427787/2018-1, 434969/2018-4, 312149/2016-6, 141276/2020-7, and 408356/2018-9), CAPES/Procad (grant 175956), CAPES/Proex, FAPPR (grant 51435), and FAPERJ (grant 200773/2019, 010002285/2019).

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on programcomprehension. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 181–190.

[2] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Marcio Ribeiro. 2015. Experience Report: Evaluating the Effectiveness of Decision Trees for Detecting Code Smells. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE '15)*. IEEE Computer Society, Washington, DC, USA, 261–269. https://doi.org/10.1109/ISSRE.2015.7381819

[3] Roberta Arcoverde, Isela Macia, Alessandro Garcia, and Arndt Von Staa. 2012. Automatically detecting architecturally-relevant code anomalies. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 90–91.

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

[5] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software (JSS)* 107 (2015), 1–14.

[6] Richard Bellman. 1966. Dynamic programming. *Science* 153, 3731 (1966), 34–37.

[7] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[8] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 465–475.

[9] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes? A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES)*. 74–83.

[10] William W. Cohen. 1995. Fast Effective Rule Induction. In *Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 115–123.

[11] Warteruzannan Soyer Cunha and Valter Vieira de Camargo. 2019. Uma Investigação da Aplicação de Aprendizado de Máquina para Detecção de Smells Arquiteturais. In *Anais do VII Workshop on Software Visualization, Evolution and Maintenance (VEM)* (Salvador). SBC, Porto Alegre, RS, Brasil, 78–85. https://doi.org/10.5753/vem.2019.7587

[12] R. M. d. Mello, R. F. Oliveira, and A. F. Garcia. 2017. On the Influence of Human Factors for Identifying Code Smells: A Multi-Trial Empirical Study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 68–77. https://doi.org/10.1109/ESEM.2017.13

[13] Rafael de Mello, Anderson Uchôa, Roberto Oliveira, Willian Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. 2019. Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–6.

[14] 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 *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621.

[15] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo 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)*. 18:1–18:12.

[16] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5–1.

[17] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2015. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* (June 2015). https://doi.org/10.1007/s10664-015-9378-4

[18] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. *2013 IEEE International Conference on Software Maintenance* (sep 2013), 396–399. https://doi.org/10.1109/ICSM.2013.56

[19] Martin Fowler. 1999. *Refactoring* (1 ed.). Addison-Wesley Professional.

[20] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[21] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, and Ian H Reutemann, Peter andWitten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[22] Tin Kam Ho. 1995. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, Vol. 1. IEEE, 278–282.

[23] R.C. Holte. 1993. Very simple classification rules perform well on most commonly used datasets. *Machine Learning* 11 (1993), 63–91.

[24] Mario Hozano, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. 2017. Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers. In *Proceedings of the 19th International Conference on Enterprise Information Systems*. 474–482.

[25] Mario Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. 2017. Smells Are Sensitive to Developers!: On the Efficiency of (Un)Guided Customized Detection. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 110–120. https://doi.org/10.1109/ICPC.2017.32

[26] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. 2018. Are You Smelling It? Investigating How Similar Developers Detect Code Smells. *Information and Software Technology (IST)* 93, C (Jan. 2018), 130–146. https://doi.org/10.1016/j.infsof.2017.09.002

[27] Allen Kent, Madeline M Berry, Fred U Luehrs Jr, and James W Perry. 1955. Machine literature searching VIII. Operational criteria for designing information retrieval systems. *American documentation* 6, 2 (1955), 93–101.

[28] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (Aug. 2011), 243–275. https://doi.org/10.1007/s10664-011-9171-y

[29] F Khomh, S Vaucher, Y G Guéhéneuc, and H Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 305–314.

[30] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring challenges and benefits at Microsoft. *TSE'14* 40, 7 (2014), 633–649.

[31] Brett Lantz. 2019. *Machine learning with R: expert techniques for predictive modeling*. Packt Publishing Ltd.

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

[33] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. 2005. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[34] Isela Macia, Alessandro Garcia, Christina Chavez, and Arndt von Staa. 2013. Enhancing the detection of code anomalies with architecture-sensitive strategies. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 177–186.

[35] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, and Esma Gueheneuc, Yann-Gael andAimeur. 2012. SMURF: A SVM-based Incremental Anti-pattern Detection Approach. *2012 19th Working Conference on Reverse Engineering* (Oct. 2012), 466–475. https://doi.org/10.1109/WCRE.2012.56

[36] Tom M. Mitchell. 1997. *Machine learning*. McGraw-Hill, Boston (Mass.), Burr Ridge (Ill.), Dubuque (Iowa). http://opac.inria.fr/record=b1093076

[37] M.J. Munro. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. *11th IEEE International Software Metrics Symposium (METRICS)* (2005), 15–15. https://doi.org/10.1109/METRICS.2005.38

[38] Daniel Oliveira. 2020. Towards customizing smell detection andrefactorings. (2020). Master dissertation. Pontifical University of Rio de Janeiro.

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

[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* (Tallinn, Estonia) *(MaLTeSQuE 2019)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3340482.3342744

[41] Fabiano Pecorelli, Dario [Di Nucci], Coen [De Roover], and Andrea [De Lucia]. 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software* 169 (2020), 110693. https://doi.org/10.1016/j.jss.2020.110693

[42] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. 2020. Developer-Driven Code Smell Prioritization. In *International Conference on Mining Software Repositories*.

[43] J. Platt. 1998. Fast Training of Support Vector Machines using Sequential Minimal Optimization. In *Advances in Kernel Methods - Support Vector Learning*, B. Schoelkopf, C. Burges, and A. Smola (Eds.). MIT Press. http://research.microsoft.com/~jplatt/smo.html

[44] Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

[45] José Amancio M. Santos, Manoel G. Mendonça, Cleber Pereira dos Santos, and Renato Lima Novais. 2014. The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development* 2 (2014), 1–33.

[46] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor?. In *FSE'16*. 858–870.

[47] Ingo Steinwart and Andreas Christmann. 2008. *Support vector machines*. Springer Science & Business Media.

[48] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *23rd Annual International Conference on Computer Science and Software Engineering*. 132–146.

[49] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 306–315.

[50] Aiko Yamashita and Leon Moonen. 2013. Exploring the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 682–691. http://dl.acm.org/citation.cfm?id=2486788.2486878