



# Developers' perception matters: machine learning to detect developer-sensitive smells

Daniel Oliveira<sup>1</sup> · Wesley K. G. Assunção<sup>1</sup> · Alessandro Garcia<sup>1</sup> · Balduino Fonseca<sup>2</sup> · Márcio Ribeiro<sup>2</sup>

Accepted: 29 August 2022 / Published online: 12 October 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

## Abstract

Code smells are symptoms of poor design that hamper software evolution and maintenance. Hence, code smells should be detected as early as possible to avoid software quality degradation. However, the notion of whether a design and/or implementation choice is smelly is subjective, varying for different projects and developers. In practice, developers may have different perceptions about the presence (or not) of a smell, which we call *developer-sensitive smell detection*. Although Machine Learning (ML) techniques are promising to detect smells, there is little knowledge regarding the accuracy of these techniques to detect developer-sensitive smells. Besides, companies may change developers frequently, and the models should adapt quickly to the preferences of new developers, i.e., using few training instances. Based on that, we present an investigation of the behavior of ML techniques in detecting developer-sensitive smells. We evaluated seven popular ML techniques based on their accuracy and efficiency for identifying 10 smell types according to individual perceptions of 63 developers, with some divergent agreement on the presence of smells. The results showed that five out of seven techniques had statistically similar behavior, being able to properly detect smells. However, the accuracy of all ML techniques was affected by developers' opinion agreement and smell types. We also observed that the detection rules generated for developers individually have more metrics than in related studies. We can conclude that code smells detection tools should consider the individual perception of each developer to reach higher accuracy. However, untrained developers or developers with high disagreement can introduce bias in the smell detection, which can be risky for overall software quality. Moreover, our findings shed light on improving the state of the art and practice for the detection of code smells, contributing to multiple stakeholders.

**Keywords** Software quality · Machine learning · Code smell detection · Code smell

---

Communicated by: Foutse Khomh, Gemma Catolino, Pasquale Salza

This article belongs to the Topical Collection: *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQUE)*

---

✉ Daniel Oliveira  
doliveira@inf.puc-rio.br

Extended author information available on the last page of the article.

## 1 Introduction

Code smells are considered symptoms of poor design and/or implementation choices that make software systems hard to evolve and maintain (Fowler 1999). Due to their harmfulness to software quality (Abbes et al. 2011; Khomh et al. 2011a; Yamashita and Moonen 2013), code smells should be detected and removed as early as possible. Unfortunately, several reasons make the code smell detection a challenging task. For instance, code smell detection usually relies on the expertise and preferences of developers (Mäntylä and Lassenius 2006; Mäntylä 2005; de Mello et al. 2017). In this practical context, the code smell detection has a subjective nature. A recent study indicates a high divergence among developers about the existence of a same code smell into a code fragment (Hozano et al. 2018). Hence, detecting code smells in practice is much harder than related studies usually discuss (Marinescu 2004; Palomba et al. 2013; Moha et al. 2009, 2010; Gopalan 2012; Bertran et al. 2013; Bertran 2011; Oizumi et al. 2016; Sousa et al. 2018).

Detection strategies based only on metrics or general-purpose tools are not totally effective and overload developers with many potential false positives (Arcoverde et al. 2013; Bertran et al. 2012a, b; Silva et al. 2013; Fernandes et al. 2017; Sousa et al. 2020; Oizumi et al. 2019). To make matters worse, definitions for some code smells are informal, ambiguous, or insufficient to describe them precisely. For instance, let us consider the definition of *Long Method* (LM) code smell, which is *a method that is too long and tries to do too much* (Fowler 1999). Although such definition states what a *Long Method* is, it does not describe what should be considered as “too long” neither what “to do too much” is. As a consequence, when developers are focused on detecting *Long Methods*, they rely on their perception to reason about the following questions: How to define whether a method is long? How to define whether a method is doing too much? Is it possible to identify a *Long Method* solely based on the lines of code of a method? How many lines of code are required to characterize a method as a *Long Method*? Will methods with this number of lines of code be difficult to maintain or evolve?

Different developers working on the same code base may have different answers to these questions. As a consequence, while a developer may confirm a code fragment as the host of a *Long Method*, other developers may not necessarily agree. Indeed, the literature discusses this divergence among developers (Hozano et al. 2017b, 2018). In this context, we can say that smell detection is a *developer-sensitive* task (Hozano et al. 2017b), i.e., detecting code-smells relies on developer’s perception for each smell type. In this way, the same code fragment can be classified as host of a smell or not by different developers. However, detecting smells taking into account the individual perception of each developer remains a prevailing challenge (Hozano et al. 2017b).

To extract knowledge from smells observed in real-world projects, a recent systematic mapping study (Azeem et al. 2019) reveals there are many studies that analyzed the use of Machine Learning (ML) techniques to identify smells (Hozano et al. 2017a, b; Khomh et al. 2009, 2011b; Maneerat and Muenchaisri 2011; Maiga et al. 2012; Fontana et al. 2013; Amorim et al. 2015; Arcelli Fontana et al. 2016). In a nutshell, the ML techniques require a training dataset containing code instances annotated by developers as smelly or non-smelly. From these training instances, the ML techniques generate smell detection models. Even though such studies indicate that ML techniques are a promising way to detect smells (Azeem et al. 2019), there is little knowledge of how accurate these generated smell detection models are when they are used for detecting developer-sensitive smells (Hozano et al. 2017a). Smell detection based on the perception of a developer enables the automatic

prioritization of smell instances, allowing developers to focus on more relevant tasks. This prioritization also allows the reduction of false positives within the context of a specific team, project or system's module, avoiding unnecessary warnings that deviate developer's attention.

In this context, this paper reports a comprehensive study about the accuracy and efficiency of ML techniques on detecting smells based on developers' perceptions. Accuracy refers to the number of instances correctly labeled as smelly. For this study, we measure accuracy through F-measure, which considers both the recall and precision to compute a score. Efficiency is related to how new validations from different developers with divergent perceptions affect the accuracy of ML techniques. For instance, high values of efficiency indicate the ML techniques need a low number of training instances to reach high F-measure (accuracy).

In order to evaluate the accuracy and efficiency of ML techniques, we assess seven ML techniques considering their capability of detecting 10 different smell types in accordance with the individual perception of 63 developers. We conducted our study in four main steps, one step to create the dataset and three steps to evaluate the results. In the first step, (i) *Dataset composition*, for each smell type, 63 developers evaluated the presence (or not) of a code smell. Altogether, we collected a dataset with 1,800 classifications, which were used to evaluate the sensitivity of the ML techniques. To ensure that the chosen developers have different perceptions, which is required for our evaluation, we computed the agreement between them using Fleiss' Kappa measure (Fleiss 1971). This measure confirmed a slight or fair agreement (for different smells), indicating the reliability of the dataset for the goal of our study.

The composed dataset was used as input for the seven ML techniques, and the results were evaluated as follows: (ii) *Accuracy evaluation*, we initially evaluated the global accuracy of the ML techniques on detecting each one of the 10 smell types for all developers together based on F-measure values. Then, we evaluated the global accuracy regardless of the smell type. Finally, we applied statistical tests to observe if there is a statistical difference across the techniques' results. (iii) *Dispersion evaluation*, we investigated the dispersion of the accuracy for the ML techniques when detecting developer-sensitive smells. The dispersion is the F-measure variation of the ML technique per developer, i.e., when the techniques detect smells taking into account the individual perception of each developer. For (iv) *Efficiency evaluation*, we assessed the efficiency of the ML techniques by evaluating the accuracy of each technique on detecting developer-sensitive smells, in which we gradually increase the number of instances provided by different developers used to perform its training.

The analysis of the results of our study led to the following main findings:

- Five out of seven studied ML techniques (*J48*, *Naive Bayes*, *OneRule*, *Random Forest*, and *Sequential Minimal Optimization*) have a statistically similar behavior, obtaining high F-measure values when detecting developer-sensitive smells. Only two techniques (*JRip* and *Support Vector Machine*) had lower results in comparison to the other techniques, in contrast to previous studies that do not consider developers' perceptions (Arcelli Fontana et al. 2016; Azeem et al. 2019).
- The F-measure of all the analyzed ML techniques was influenced by the developers' perceptions. As a consequence, all the techniques had high dispersion, reaching a maximum and minimum F-measure of 1 and 0, respectively, for different developers in the same smell type.

- *Support Vector Machine* is the technique that obtained the most dispersed results, whereas *Naive Bayes* reached the least dispersed results. Long method is the code smell type in which the values of F-measure were the least dispersed.
- When the agreement among the developers decreases, the dispersion of the ML techniques' accuracy increases.
- For the majority of techniques, the number of instances used as training does not impact directly the accuracy of the ML techniques.

These findings indicate that, when detecting developer-sensitive smells, most of the ML techniques reached high F-measure. However, we could see that no technique obtained the best results for all the analyzed smells. Therefore, developers need to take into consideration which smells are most relevant to them when choosing the technique to be used. Finally, we observed that the techniques needed a low number of instances to reach their maximum accuracy. This indicates the use of these techniques is a good choice for companies integrated by developers with different experiences.

We complemented the analysis of the accuracy and efficiency of the ML techniques to detect developer-sensitive smells with additional observations. First, we also compared the detection rules generated in our study to the rules reported in related studies. More specifically, we compare the detection rules generated by J48, JRip, and OR, for individual developers as well as for all the developers together, to the four most closely related studies by Arcelli Fontana et al. (2016), Rasool and Arshad (2015), Pecorelli et al. (2020), and Bigonha et al. (2019). We observed that the detection rules for both individual developers and all developers together frequently include software metrics proposed by the literature for specific smell types. However, differently from related studies, we observe that for individual developers the rules have a higher amount of metrics, including different types of metrics not identified or proposed in previous research. Another observation is that ML techniques commonly create diverse detection rules for individual developers, even in the cases that developers highly agree about the existence of code smells, e.g., the code smell Data Class.

Despite the findings and observations of our study, we also highlight the risks of relying exclusively on the preferences of individual developers. When developers write code with low quality, they might also lead to the derivation of inappropriate detection rules, severely harming the overall internal software quality. Furthermore, in real-world projects, developers often work independently on different parts of the software. In these projects, developer-sensitive smell detection can lead to nonuniform levels of quality in every part of the system, which may consequently impact the project quality as a whole. We also discuss ways to circumvent these potential issues.

Based on the methodology, findings and insights of our study, the contributions and implications are many-fold and target different stakeholders. For *researchers*, we make available a package with supplementary material for replications and new studies. Additionally, our results reveal the need for approaches that automatically decide which ML techniques should be applied taking into the agreement of developers and the target smell type. Another direction is to monitor the perception of developers during the project development, identifying (dis)agreements on the fly and using this information to better guide developers. For *practitioners*, our results show that ML techniques can quickly adapt to newcomers entering the project. We also recommend the use of models trained by senior developers to support junior developers on learning the quality standards of the project and/or company. For *tool builders*, there is the opportunity to develop tooling support that at least enables individual developers to include their preferences in the detection of smells.

For *educators*, our study can be used as a source of information to initiate the discussion of trade-offs and divergences that exist when developers discuss internal software quality.

The remaining of this document is structured as follows. Section 2 describes the design of our study and the research questions. In Section 3 we present the results, answer the research questions, and discuss the contributions and implications of the study. Section 4 details the threats of the study. Next, Section 5 presents the related work. Finally, Section 6 presents the conclusions observed in our study.

## 2 Study Design

Previous studies suggest that ML techniques are a promising way to identify code smells (Hozano et al. 2017a, b). However, in practice, the code smell detection involves the perception of different developers, usually influenced by their expertise and preferences, which might lead to divergences on defining whether a code fragment is a smell (Mäntylä and Lassenius 2006; Mäntylä 2005; Hozano et al. 2018; Schumacher et al. 2010; Santos et al. 2013). This divergence among developers may considerably influence the set of code fragments perceived as smelly and, consequently, being identified. In particular, such divergence may impact the accuracy of code smell detection techniques based on ML, which rely on code smell instances previously annotated by developers as training.

To collect evidence of this phenomenon and advance the knowledge of developer-sensitive smell detection, we conducted an empirical study that is described in this section. We follow the Goal/Question/Metric (GQM) approach to design our study (van Solingen et al. 2002). The goal of our study is *to understand the behavior of ML techniques to detect code smells based on different developers' perceptions*. From this goal, we posed three research questions (RQs) that guided the evaluation, as follows:

### **RQ1: How accurate are the ML techniques on detecting developer-sensitive smells?**

RQ1 aims at investigating the *global* accuracy of seven ML techniques, by analyzing F-measure, when detecting 10 smell types. Even though a recent systematic literature review study (Azeem et al. 2019) reported that several studies have investigated the accuracy of ML techniques to detect code smells, they do not discuss the characteristics of their dataset regarding the divergences among developers. Thus, differently from previous studies, we analyzed each one of the 10 smell types performing the training of the ML techniques on a small number of different instances. These instances were classified multiple times by developers with different perceptions (see Section 2.3). This question focuses on understanding how the ML techniques are affected by different opinions regarding the existence or not of a smell.

### **RQ2: How dispersed is the accuracy of ML technique on detecting developer-sensitive smells?**

Different from RQ1 that focus on the *global* accuracy of each ML technique, RQ2 investigates the dispersion, i.e., accuracy variation, of ML techniques on detecting smells in accordance with the *individual* perceptions of each developer. To answer this question, we evaluated the accuracy of each technique for each developer separately. The main motivation for this research question is the fact that developers have different backgrounds, experiences, and skills. These and other factors naturally lead developers to have different perceptions about the occurrence of smells in the same pieces of code. As a consequence of this divergence among

developers, ML techniques may present a dispersion in their accuracy on detecting developer-sensitive smells (Hozano et al. 2018).

### RQ3: How efficient are the ML techniques for detecting developer-sensitive smells?

Although ML techniques have been considered a promising way to detect code smells (Arcelli Fontana et al. 2016; Hozano et al. 2017b; Azeem et al. 2019), these techniques usually require several code smell instances annotated by developers to perform their training. However, the annotation of a large number of instances may introduce an unfeasible additional effort to the developers. In this way, RQ3 aims at analyzing the efficiency of the ML techniques on detecting developer-sensitive smells, i.e., how the accuracy of the ML techniques are affected while we gradually increase the number of classified instances used to perform its training. An efficient technique requires a low number of instances for its training, consequently enabling its use in a wide number of scenarios.

To achieve the goal of our study and answer the RQs, we evaluated the accuracy and efficiency of each technique in terms of three aspects: (i) the *global* accuracy of the ML techniques, (ii) the *individual* accuracy of the ML techniques based on the perceptions of developers about the presence of code smells, and (iii) the *number of instances* used to perform the training of the ML techniques. The metrics used for the evaluation are presented in the next section.

## 2.1 Metrics and Statistical Tests

To assess the accuracy of the ML techniques, we used *F-measure* that considers both *recall* and *precision* to compute a score. For our study, the *true positive (TP)* elements represent the code fragment classified by the ML techniques as a code smell that are, actually, a real code smell, as well as the *false positive (FP)* elements refer to the code fragments wrongly classified as code smell. Similarly, the *true negative (TN)* represents the code fragments correctly classified as not-smell and the *false negative (FN)* represents the wrong ones. In this context, we define the *recall*, *precision*, and *F-Measure* as:

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

$$R = \frac{TP}{TP + FN} \quad (1)$$

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

$$P = \frac{TP}{TP + FP} \quad (2)$$

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

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \quad (3)$$

To ensure that developers that participate in our study have different perceptions regarding the existence or not of smells in the same code fragment, we computed the Fleiss' Kappa measure (Fleiss 1971). This is a statistical measure for assessing the reliability of agreement between a fixed number of raters. In this way, we could observe the agreement among the classifications, i.e., perceptions of different developers. The values for this measure can range from negative numbers to a max of 1, where less than 0 indicates poor

**Table 1** Fleiss' Kappa interpretation

Fleiss' Kappa	Interpretation
< 0	Poor agreement
0.01–0.20	Slight agreement
0.21–0.40	Fair agreement
0.41–0.60	Moderate agreement
0.61–0.80	Substantial agreement
0.81–1.00	Almost perfect agreement

agreement and 1 indicates almost perfect agreement. Table 1 presents the interpretation of values for this measure.

To statistically compare the ML algorithms, firstly, we used the Shapiro-Wilk statistical test (Surhone et al. 2010) to investigate the sample distributions. As there is no normal distribution of the results, Friedman non-parametric test (Friedman 1937) with Nemenyi's test as post hoc multiple pairwise comparisons was adopted for the analysis of the results. We adopt a confidence level of 95% ( $p\text{-value} \leq 0.05$ ). To further analysis, we also computed the effect size with the Vargha-Delaney's  $\hat{A}_{12}$  measure (Vargha and Delaney 2000). All statistical tests were applied using R (Lantz 2019). Finally, to reason about the correlation between developers' agreement, namely Fleiss' Kappa values and the dispersion of the ML techniques, which was evaluated based on standard deviation, we applied the Spearman test (Spearman 1904). Spearman's rank correlation coefficient is a non-parametric measure of rank correlation, i.e., statistical dependence between the rankings of two variables.

## 2.2 Projects, Smells and Subjects

We analyzed the code smells on five open-source Java projects, namely GanttProject<sup>1</sup> (2.0.10), Apache Xerces<sup>2</sup> (2.11.0), ArgoUML<sup>3</sup> (0.34), jEdit<sup>4</sup> (4.5.1) and Eclipse<sup>5</sup> (3.6.1). We selected such projects because they have been evaluated by existing smell detection techniques (Lanza et al. 2005; Munro 2005; Khomh et al. 2011b; Marinescu 2004; Moha et al. 2010; Fontana et al. 2011; Palomba et al. 2014a) and their source code contains a variety of suspicious code smells (Hozano et al. 2017b, 2018) that provides a rich dataset for our study.

The next step was to select the smell types to be analyzed. Table 2 presents the smell types selected for our study. We have chosen these smell types because they affect different scopes of a program, i.e., classes, methods or parameters. Additionally, they are smell types recurrently investigated in literature (Khomh et al. 2009; Maiga et al. 2012; Arcelli Fontana et al. 2016; Khomh et al. 2011b; Amorim et al. 2015).

Finally, we selected developers to collect their different perceptions of code smells. To recruit developers, we sent invitations for several contacts from companies and academic institutions. Altogether, 63 developers from different companies and institutions accepted the invitation. Then, they rated their own experience in development. All of them mentioned

<sup>1</sup><http://www.ganttproject.biz>

<sup>2</sup><http://xerces.apache.org>

<sup>3</sup><http://argouml.tigris.org>

<sup>4</sup><http://www.jedit.org>

<sup>5</sup><http://eclipse.org>

**Table 2** Types of code smells investigated in this study

Name	Description
God Class (GC)	Classes that tend to centralize the intelligence of the system.
Data Class (DCL)	Classes that have fields, getting and setting methods for the fields, and nothing else.
Long Method (LM)	Methods that are too long and try to do too much.
Feature Envy (FE)	Methods that use more attributes from other classes than from its own class, and use many attributes from few classes.
Message Chains (MC)	An object that calls another object, that requests yet another one, and so on.
Inappropriate Intimacy (II)	Classes that use internal fields and methods that do not belong to them.
Middle Main (MM)	Classes that delegate too much work to other classes and do nothing by herself.
Primitive Obsession (PO)	Using a lot of primitives as a substitute for small objects.
Refused Bequest (RB)	Classes inherit from a superclass and do not use any of the inherited functionality.
Speculative Generality (SG)	Unused classes, methods, fields or parameters created to future features that never get implemented.

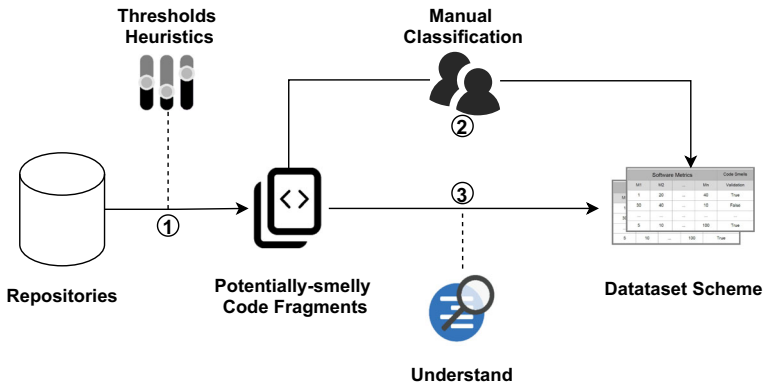
having at least three years of experience in Java for software development. Besides, these developers also have previous experience in code smell detection in software projects. The composition of the dataset based on the perceptions of these developers is presented next.

### 2.3 Data Collection

Figure 1 illustrates the steps of the data collection. In the step one, we extracted 15 potentially-smelly code fragments for each type of code smell studied from the projects analyzed. For this extraction, we used heuristics provided by previous studies to identify potentially-smelly code fragments (Palomba et al. 2014b; Lanza et al. 2005). A potentially-smelly code fragment is a set of statements where its behavior indicates a possible existence of a code smell. The used heuristics rely on software metrics and thresholds to define code elements as a possible candidate of host code smell. Once the metrics of a code element overpass predefined thresholds, this element is a potentially-smelly code.

After the extraction was finished, in the second step of the collection, we instructed the 63 participating developers to classify the extracted fragments as a smelly or not. At the beginning, the developers were informed that they can interrupt and resume the classification at any time, to make sure they were able to classify when they felt comfortable. Then, to turn this classification feasible and to avoid developers' fatigue during classification, we grouped the 63 developers into 10 groups. Each group, composed of 12 developers (six developers from academy and six from industry), was responsible to classify the same 15 code fragments concerning the existence (or not) of only a specific smell type. This repetitive classification for the same code fragment enabled us to reason about the same code fragment classified by different developers with different perspectives. We took into account that each developer should be allocated at most to three groups to avoid fatigue. Also, a description of the evaluated smell based on the definition presented by Fowler's catalog (Fowler 1999) were presented to the developer. Finally, we ensure that each developer belonging to more than one group only classify one smell type per time.





**Fig. 1** Data collection steps


The classification process concerns the analysis of a potentially-smelly code fragment by looking for the specific smell type reported. The code fragments comprehend the scope of the smell type analyzed, i.e., method, class, package, or project. Developers had to indicate whether a code fragment contains a smell type or not by providing the following answers: **YES**, if the developer agrees that a given code fragment presents the reported smell type; or **NO**, otherwise.

Following the classification of code fragments, in the third step illustrated in Fig. 1, we used Understand<sup>6</sup> to extract software metrics. These metrics were used to characterize each code fragment in terms of features to be used during the training process of the ML techniques. We also collected popular metrics from literature (Arcelli Fontana et al. 2016; Palomba et al. 2014b; Lanza et al. 2005; Munro 2005; Gopalan 2012). The complete list of metrics used is presented in Appendix A. Figure 2 illustrates a table of the dataset's schema containing the features, i.e., metrics (M1...Mn), and classifications (True or False) associated with the code fragments. Altogether, the dataset is composed of 120 tables, one per each developer in each group. Each table has 15 lines, comprising the 1800 assessments mentioned above. The complete dataset, including the values for the metrics, labels of the manual validation, and detection rules is available in the supplementary material (Oliveira et al. 2022).

## 2.4 Machine Learning Techniques

To investigate the behavior of different ML techniques, we chose seven techniques to be evaluated, which are described next. We chose these techniques because of their comprehensiveness, different data analysis approaches, i.e., decision trees, regression analysis, and rule-based analysis that are responsible to create the classifier models. This diversity of the approach allows us to compare the accuracy and efficiency of them on detecting each studied smell, this comparison led us to understand the scenarios that each approach can be better applied. Another reason is regarding that they also are widely evaluated in previous studies related to code smell detection (Hozano et al. 2017a, b), which is also confirmed by a recent systematic literature review (Azeem et al. 2019).

<sup>6</sup><https://scitools.com/features/>



Code Snippets	Software Metrics				Code Smells
	M1	M2	...	Mn	Validation
1	20	...	40	True	
30	40	...	10	False	
...	...	...	...	...	
5	10	...	100	True	

**Fig. 2** Schema of the dataset

**Naive Bayes (NB)** Probabilistic classifier based on the application of Bayes' theorem (Mitchell 1997). NB uses this theorem together with a strong assumption that the attributes are conditionally independent given the class. This technique employs a mechanism for using the information in sample data to estimate the posterior probability  $P(y|x)$  of each class  $y$  given an object  $x$ . Then, it uses such estimates for classification. In summary, the main idea is to describe the probability of an event based on prior knowledge of conditions that might be related to this event. NB is highly scalable and completely disregards the correlation between the variables in the training set.

**Support Vector Machine (SVM)** Implementation of integrated software for the classification of support vectors (Steinwart and Christmann 2008) that analyzes the data used for classification and regression analysis. SVM assigns new instances to one of the two categories introduced in the training set, making it a non-probabilistic binary linear classifier. In order to make this classification, SVM creates classification models that are a representation of instances as points in space. These points are mapped in such a way that the instances 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 (Platt 1998). Training a SVM requires the solution of a very large quadratic programming (QP) optimization problem. To deal with this situation, SMO breaks this large QP problem into small QP problems. Then, each of these small QP problem are solved analytically, without using a time-consuming numerical QP optimization as an inner loop. In other words, SMO is a technique for optimizing the SVM training to turn it faster and less complex than the previous methods.

**OneRule (OneR)** Classification technique that generates a rule for each predictor in the data, then selects the rule with the lowest total error as its "single rule" (Holte 1993). In order to create this rule, this technique analyzes the training set and associates a single data to a specific category based on its frequency. For instance, if a specific data is usually classified as *category A*, then a rule is created linking them. After creating the rules, 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 (Ho 1995). In training, the RF algorithm creates multiple trees (Breiman et al. 1984), each trained on a sample of the original training data, and searches only across a randomly selected subset of the input variables to determine a split. For classification, each tree in the Random Forest casts a unit vote for the most popular class of the input pattern. The output of the classifier is determined by a majority vote of the trees. The RF technique adds extra randomness to the model during the creation of the trees. 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 that 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 (Cohen 1995). This technique is based on association rules with reduced error pruning, a very common and effective technique found in decision tree algorithms. Differently from the other algorithms, JRip splits its training stage into two steps, namely 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 allows the pruning of any final sequences of the antecedents.

**J48** A Java implementation of the C4.5 decision tree technique (Quinlan 1993). J48 builds decision trees from a training dataset by, at each node of the tree, choosing 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.

## 2.5 Experimental Configurations

Using the datasets containing the different perceptions of developers and the software metrics for each analyzed code fragment, we performed two different experiments. Each configuration focusing a specific RQ, as follows:

**Assessment of ML techniques accuracy:** To answer **RQ1**, we used the dataset to analyze the global accuracy of the ML techniques on detecting a specific smell type by relying on *F-measure* as the metric. For each smell type, we calculated the *global* accuracy of each technique by applying a 10-fold cross validation procedure on each table of the dataset associated with that smell type. Then, in order to answer **RQ2**, we repeated this experiment, but in this case, analyzing the *individual* accuracy variation obtained by each technique on detecting smells to the different developers.

**Assessment of ML techniques efficiency:** Aiming to answer **RQ3**, we evaluate the efficiency of the ML techniques. For that, we evaluated the accuracy by considering the 15 classifications performed by each developer. However, we used sets with different number of instances for training. These sets ranged from three (20% of the instances) to 12 (80% of the instances), with the goal of guarantying that both, the training and test sets, were composed of fragments classified as *smelly* and *non-smelly* by each developer.

**Table 3** Fleiss' Kappa per smell type

Smell	Fleiss' Kappa	Agreement
GC	0.308	Fair
DCL	0.321	Fair
LM	0.310	Fair
FE	0.246	Fair
MC	0.083	Slight
II	0.088	Slight
MM	0.128	Slight
PO	0.096	Slight
RB	0.078	Slight
SG	0.065	Slight

## 2.6 Implementation Aspects

The ML techniques presented in the previous section were implemented on top of Weka<sup>7</sup> and R Project.<sup>8</sup> Weka is an open source ML software, based Java programming language, containing a plethora of tools and algorithms (Hall et al. 2009). R is a free software environment for statistical computing that is widely used for data mining, data analysis, and to implement ML techniques (Lantz 2019). Finally, we applied these algorithms by following the same configuration adopted by Arcelli Fontana et al. (2016).

## 3 Results and Discussion

This section presents and discusses the results of our study. The discussions are organized in terms of the experimental configurations in order to answer the three research questions presented in Section 2. As part of the discussions, we also present details of the detection rules generated by the ML techniques, comparing them with related studies. Furthermore, we describe potential risks that should be taken into account when considering developer-sensitive smells. Lastly, the contributions and implications of our work are presented.

As a basis for our analysis, we firstly analyzed the characteristics of the dataset used in our study (see Section 2.2). For that, we computed the Fleiss' Kappa measure; its values are presented in Table 3. In our dataset, the developers had a "Fair agreement" for four different smell types (i.e., GC, DCL, LM, and FE). For the remaining smell types (i.e., MC, II, MM, PO, RB, and SG), the developers had a "Slight agreement" on the perception of the existence (or not) of code smells. That is, the developers often disagree with each other, which indicates different perceptions regarding the existence of a smell type in the same code fragment. These characteristics are relevant for our study, as our goal is to understand how ML techniques behave when learning from a dataset that represents different developers' perceptions.

The next sections present the results and discussions of global accuracy for separated and grouped smell types (RQ1—Section 3.1), the dispersion in the accuracy of ML techniques when considering the individual opinion of each developer (RQ2—Section 3.2), and the

<sup>7</sup><https://www.cs.waikato.ac.nz/ml/weka/>

<sup>8</sup><https://www.r-project.org>

efficiency of the ML techniques when different number of instances are available for their training (RQ3—Section 3.3).

### 3.1 Assessment of Global Accuracy

Figure 3 presents boxplots of the global accuracy, i.e., F-measure, of the ML techniques on detecting the 10 smell types analyzed. Each boxplots represents 12 values of F-measure.

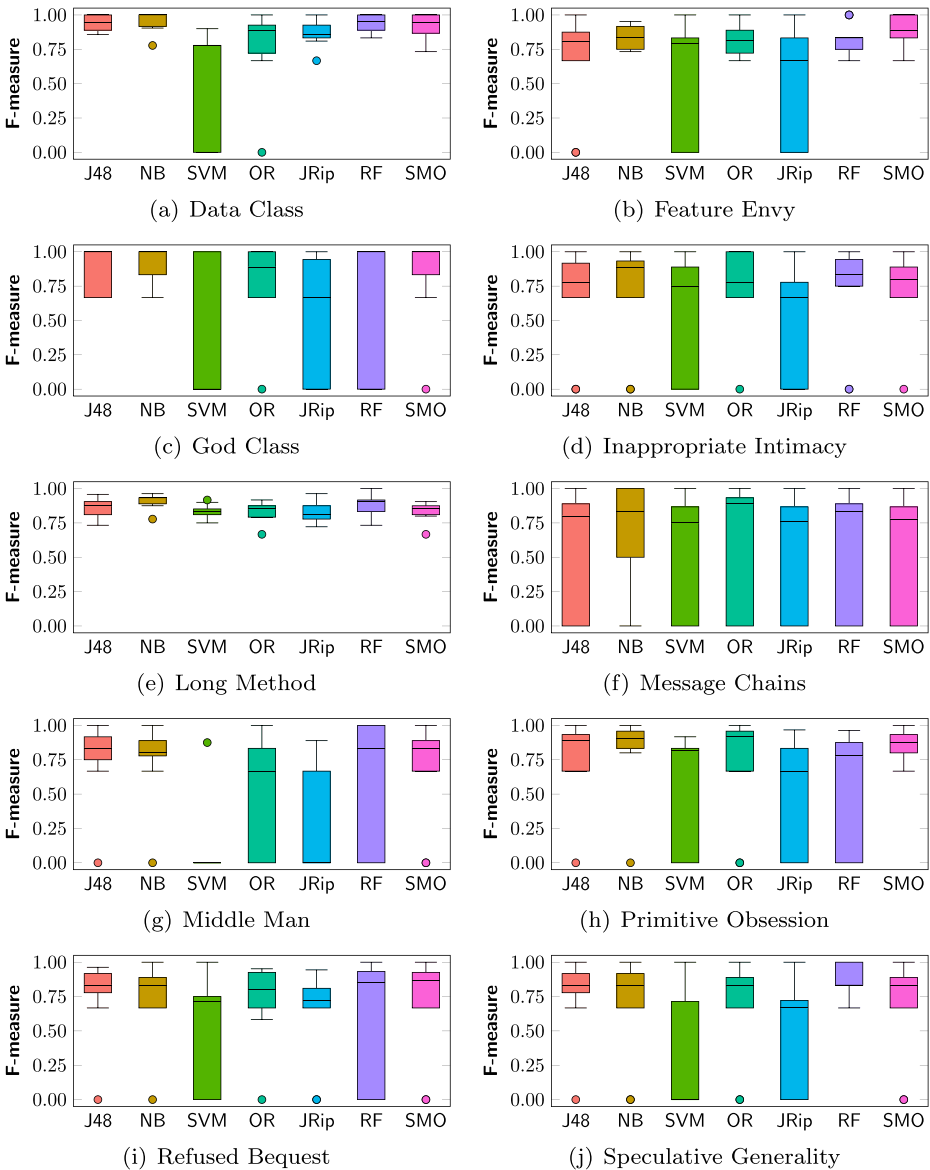


Fig. 3 Global accuracy reached by the ML techniques on detecting smells

**Table 4** ML techniques with the best value of F-measure per code smell type

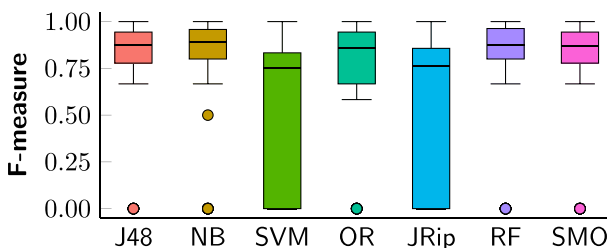
Smell	DCL	FE	GC	II	LM	MC	MM	PO	RB	SG
ML technique	NB	SMO	J48, NB RF, SMO	NB	NB	OR	RF	OR	SMO	RF

**Behavior of ML Techniques Per Smell Type** At first, we can observe that the range of values presented by the boxplots varies for different smell types. This situation is evident when comparing Long Method (Fig. 3e), for which F-measure values are higher than 0.72, and Message Chains (Fig. 3f), for which the values range from 0 to 1. Also, the behavior of the different ML techniques commonly varies for the same smell types. In almost every boxplot there are some ML techniques with F-measure values higher than 0.6 and other techniques varying from 0 to 1.

Despite these variations in the range of values, the median values are higher than 0.69 in all cases, except for SVM in three smell types, namely God Class (Fig. 3c), Middle Man (Fig. 3g), and Speculative Generality (Fig. 3j). Table 4 presents the best ML techniques for each smell, according to the highest median values of F-measure. God Class (Fig. 3c) was the only smell type that four ML techniques reached the best value, which is equal to 1. We can also see that there is no predominant technique. From a software engineering perspective, these results mean that if engineers want to prioritize the detection of specific smells, some ML techniques may be more indicated than others.

**Global Behavior of ML Techniques** Table 4 also enables us to observe which are the best ML techniques across different smell types. NB reached the best median of F-measure for four smell types, namely Data Class, God Class, Inappropriate Intimacy, and Long Method. RF was the best for three smell types: God Class, Middle Man, and Speculative Generality. SMO also was the best for the same number of smell types: God Class, Refused Bequest, and Feature Envy. Finally, OR reached the best F-measure values for Message Chains, and Primitive Obsession; and J48 only for God Class. SVM and JRip did not reach the best values in any case.

To analyze the behavior of the ML techniques independently of code smells, we group all the values of F-measure for each ML technique. For example, each group is composed of 120 values of F-measure, i.e., 12 developers times 10 code smell types. Figure 4 presents



**Fig. 4** Boxplot with the global accuracy by grouping results of all code smells for each ML technique

**Table 5** P-value of the pairwise comparisons grouping results of all code smells for each ML technique. This comparison uses Nemenyi multiple comparison test with q approximation for unrepliated blocked data

	J48	NB	SVM	OR	JRip	RF
NB	0.90178	–	–	–	–	–
SVM	<b>1.3e-09</b>	<b>2.8e-13</b>	–	–	–	–
OR	0.21554	<b>0.00730</b>	<b>0.00057</b>	–	–	–
JRip	<b>5.5e-09</b>	<b>1.4e-12</b>	0.99999	<b>0.00146</b>	–	–
RF	0.99994	0.75746	<b>8.9e-09</b>	0.37571	<b>3.6e-08</b>	–
SMO	0.99161	0.48195	<b>1.2e-07</b>	0.65392	<b>4.4e-07</b>	0.99959

the boxplot for these groups. In this figure, we can see that SVM and JRip have different results in comparison to the other techniques. To a in-depth analysis, we performed statistical comparisons (see Section 2.1). Firstly, by using the Shapiro-Wilk statistical test, we observed that all groups had a non-normal distribution of F-measure. Then, we applied the Friedman non-parametric test, which resulted in p-value < 2.2e-16, which confirms the statistical difference among the ML techniques. As the Friedman test has rejected the null hypothesis, we used the Nemenyi post-hoc test for multiple pairwise comparisons, in order to see which pairs of techniques are statistically different.

Table 5 presents the p-values of the pairwise comparisons taking into account the groups per ML technique. P-values lower than 0.05, highlighted in boldface, indicate a difference with 95% of confidence. At first, we can confirm that J48, NB, OR, RF, and SMO are statistically better than SVM and JRip, and these latter are similar. Additionally, there is a difference between NB and OR. To confirm global similarities and differences between the ML techniques, Table 6 presents the effect size of pair comparisons computed with the Vargha-Delaney’s  $\hat{A}_{12}$  measure. We removed SVM and JRip from this comparison, as they were already confirmed as worse than the others. As the results confirm, we can only observe a difference of small magnitude between NB and OR, in which the former is better. There is no other pairwise comparison that shows differences between the ML techniques; all other comparisons have a negligible effect size. Such results reinforce the findings of previous studies (Arcelli Fontana et al. 2016; Azeem et al. 2019) that indicate high accuracy of the RF, NB, and J48, and low accuracy of the SVM. Interestingly, our results also diverge from these previous studies, which indicate JRip as one of the best techniques. In our case, JRip had the worst accuracy when compared to the other ones.

**Table 6**  $\hat{A}_{12}$  measure computed by grouping results of all code smells for each ML technique. Symbols “ $\approx$ ” and “ $\pm$ ” indicate effect size magnitude *negligible* and *small*, respectively

	J48	NB	OR	RF
NB	0.5517014 $\approx$	–	–	–
OR	0.4691667 $\approx$	<b>0.4212153 <math>\pm</math></b>	–	–
RF	0.5232639 $\approx$	0.4739236 $\approx$	0.5498264 $\approx$	–
SMO	0.5074306 $\approx$	0.4571875 $\approx$	0.5378472 $\approx$	0.4885069 $\approx$

*Answering RQ1: How accurate are the ML techniques on detecting developer-sensitive smells?*

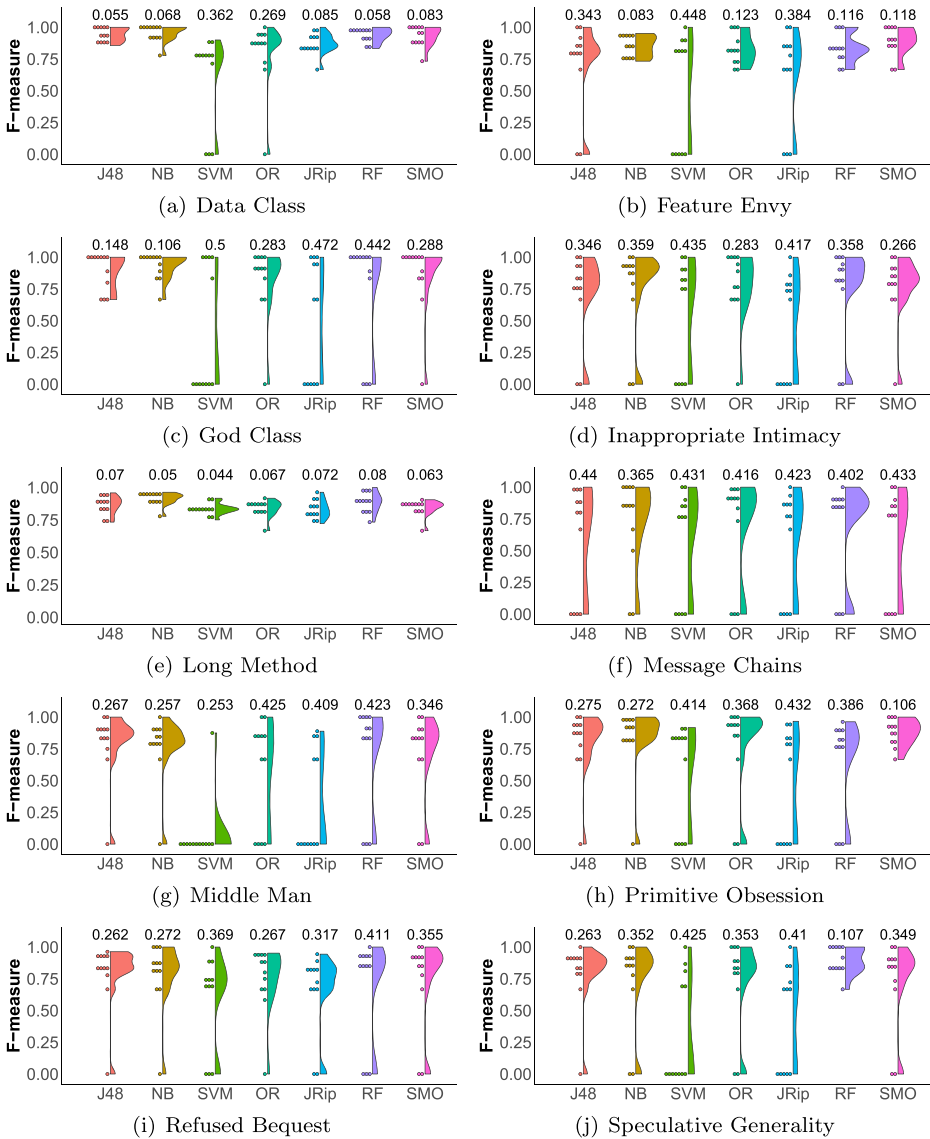
In an analysis of the global accuracy of the ML techniques, we could observe that J48, NB, OR, RF, and SMO have similar behavior. These techniques obtained good F-measure values, even with only fair and slight agreement among developers. This means that having few developers with the same perception of a code fragment as smelly, is enough for training an ML technique. Nonetheless, we could see that no technique obtained the best results for all the analyzed smells, therefore, developers need to take into consideration which smells are most relevant to them when choosing the technique to be used. In summary, we can conclude that in general, ML techniques are accurate for detecting developer-sensitive smells.

### 3.2 Assessment of ML Techniques Dispersion

As discussed in the previous section, the ML techniques could not reach a global accuracy above 0.8 in the vast majority of the cases analyzed. Our initial hypothesis was that the techniques had a high dispersion on detecting code smells for each developer. As a consequence, the techniques obtained a low global accuracy. Hence, we investigate the **RQ2** aiming at analyzing the accuracy dispersion of each ML technique on detecting developer-sensitive smells. Figure 5 presents violin plots that support the discussions about this RQ. In each plot, the *x-axis* describes the ML technique evaluated, together with points that indicate the F-measure values obtained by the technique on detecting smells for each developer. In our study, we use the *standard deviation* (SD) to quantify the dispersion of the accuracy values obtained by each technique on detecting smells for the developers. We attach the SD to the top of the bars associated with each ML technique.

All the ML techniques present some dispersion in their accuracy, similarly to what we discussed in RQ1. Additionally, by analyzing the SD obtained by each technique, we can observe that the SVM technique presented the highest dispersion in five out of 10 smell types analyzed. In the previous section, we also observed that the SVM presented the lowest global accuracy. JRip had the highest value of SD in only one case. RF had the highest dispersion in two cases, while J48 and OR in one case. Long Method was the code smell type in which the values of F-measure were the least dispersed. On the other hand, Message Chain was the one with the highest dispersion. Taking into account the Fleiss' Kappa values computed among developers (see Table 3), the smell types with the least dispersion were the ones with a fair agreement, namely Long Method, Data Class, Feature Envy, and God Class. The average dispersion per ML technique ranges between 0.2185, for NB, to 0.3681, for SVM. Across different smell types, the ML techniques have similar behavior in regard to dispersion, except for smell types Data Class and Long Methods. Finally, for an objective comparison, we compute the Spearman correlation between the values of agreement and the average standard deviation per smell type. The p-value of the correlation is equal to 0.02419, confirming the alternative hypothesis that there is a correlation. The value of the correlation is -0.721, interpreted as *High negative correlation*. As expected, this result means that when the agreement decreases, the dispersion of accuracy increases.





**Fig. 5** Accuracy Density Reached by the ML techniques on detecting the smells according to the individual perception of each developer

Although the ML techniques have presented a dispersion of their accuracy on detecting code smells for different developers, these techniques were able to reach high accuracy for specific developers. Hence, we analyze the most accurate techniques to detect smells for each developer. Figure 6 presents the main results that support the discussions about this analysis. The *x-axis* describes the *id* of a developer that evaluated code fragments related to a smell type, and the ML techniques that reached the highest accuracy, in terms of F-measure, on detecting smells for the developer. The *y-axis* presents the highest accuracy reached by the ML techniques for the corresponding developer.



Fig. 6 Accuracy reached by the ML techniques on detecting smells according to the individual perception of each developer

We observe that different techniques could obtain the highest accuracy in detecting developer-sensitive smells. For instance, every technique could reach the highest accuracy for at least one developer for Data Class. Indeed, the results indicated that more than half of the techniques obtained F-measure equals 1 for at least one developer in each type of smell,

**Table 7** Techniques that reached F-measure equals to 1.0 for at least one developer of the respective smell

	Data class	Feature envy	God class	Inappropriate intimacy	Long method	Message chain	Middle man	Primitive obsession	Refuse bequest	Speculative generality	Total
NB	✓		✓	✓		✓	✓	✓	✓	✓	8
SMO	✓	✓	✓	✓		✓	✓	✓	✓	✓	9
Jrip	✓	✓	✓	✓		✓				✓	6
RF	✓	✓	✓	✓	✓	✓	✓		✓	✓	9
SVM			✓	✓		✓			✓	✓	5
J48	✓	✓	✓	✓		✓	✓	✓		✓	8
OR	✓	✓	✓	✓		✓	✓	✓		✓	8

except for Long Method. Also, these results suggest that there is no predominant technique for all cases.

Table 7 highlights which techniques obtained F-measure equals 1 for at least one developer of the respective smell type. In this table, we can see that for the God Class, Inappropriate Intimacy, and Speculative Generality, all techniques obtained F-measure equal to 1 at least once. However, for the Long Method, only RF obtained the highest F-measure. Also, for Long Method (see Fig. 6e), there is only one best technique for each developer, with RF and NB being the best techniques for nine out of 12 developers associated with this smell.

Finally, we also observed that the techniques reached an F-measure above 0.8 for at least nine out of 12 developers for all smell types. Even in the cases in which the techniques could not obtain that accuracy, they obtained values of F-measure that varied from 0.67 to 0.79, except for the cases of four different developers with F-measure equals to 0.

A previous study (Hozano et al. 2018) indicates a statistically significant divergence among the developers' perceptions about the existence of the same code smell. In our study, we could observe that this divergence has an impact on the ML techniques. Thus, our results reinforce such findings, since we observe that ML techniques are accurate in detecting smells for each developer. Note that the techniques reached an accuracy above 0.8 in the vast majority of the cases analyzed. Indeed, the techniques obtained an accuracy of 1 in a high number of the cases analyzed.

***Answering RQ2: How disperse is the accuracy of ML technique on detecting developer-sensitive smells?***

Our results indicate that the accuracy of all machine learning techniques has a high dispersion when considering different developers individually. That is, we could observe techniques simultaneously having maximum and minimum F-measure of 1 and 0, respectively, for different developers, but the same smell type. This might be due to the different agreements among developers. In summary, ML techniques present a dispersed result of F-measure when considering individual perceptions of the developer.

### 3.3 Assessment of ML Techniques Efficiency

According to the results of the previous RQs, we observed that ML techniques were able to reach high accuracy in detecting developer-sensitive smells. In **RQ3**, we investigate the impact of the percentage of instances used for training on the values of F-measure. Figure 7 presents the results that support the discussions regarding this research question. The sub-figures represent the efficiency reached by the ML techniques on detecting the smell types

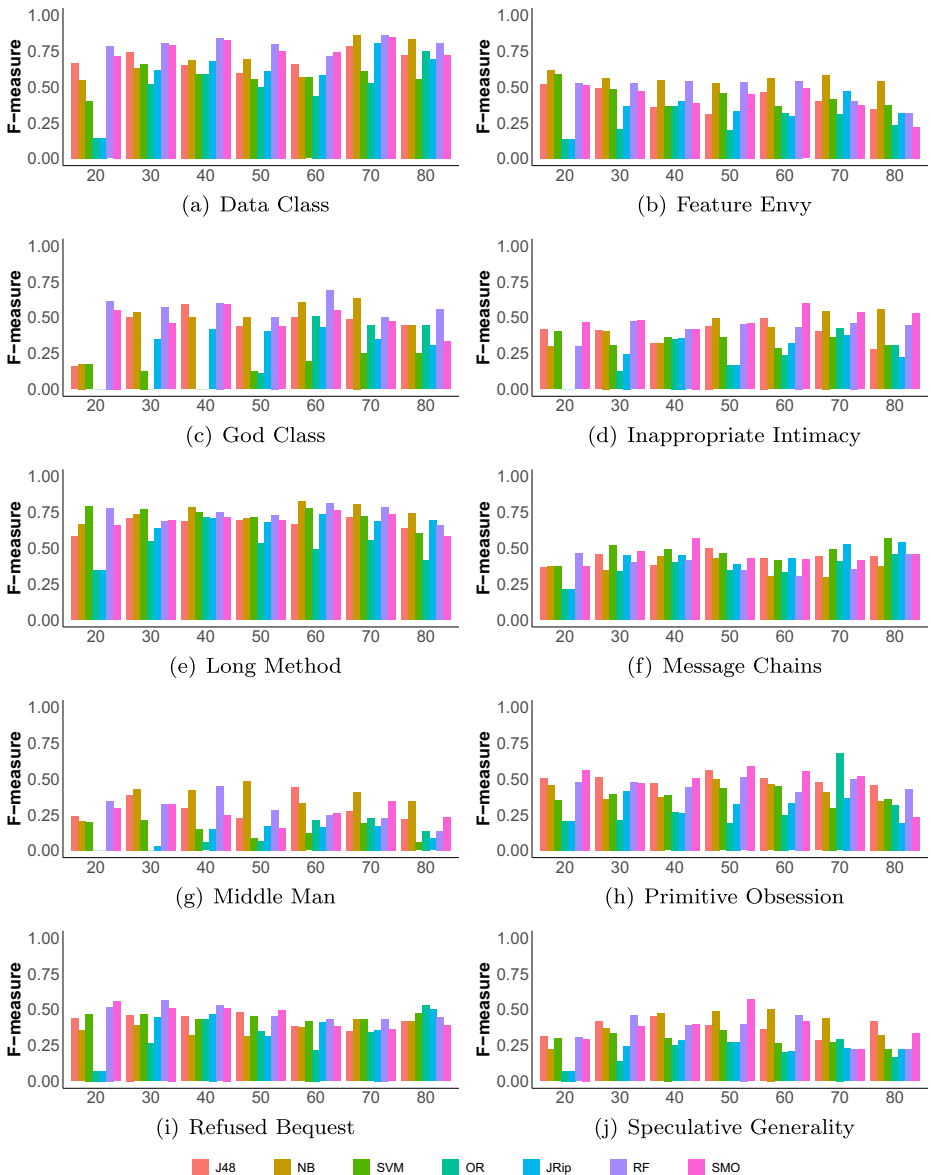


Fig. 7 Efficiency of the ML techniques on detecting the smells

analyzed using different sizes of datasets for training. The *x-axis* describes the percentage of the instances used in the training phase of the techniques, while the *y-axis* represents the *median* of the accuracy values obtained by each ML technique on detecting smells for different developers.

In an overall analysis of Fig. 7, we can observe that using different percentages of the dataset for the training does not impact the accuracy of the ML techniques. However, the same behavior of difference across different smell types is still present here. However, in all cases, the techniques JRip and OR had low values of F-measure with a lower number of instances but reached results similar to the other ones after around 40% of the instance.

*Answering RQ3: How efficient are the ML techniques on detecting developer-sensitive smells?*

As observed in previous RQs, it would not be possible to use only one technique for all smell types, being necessary to use the most appropriate technique for each smell type. In this RQ, the results indicate that for the majority of techniques, the number of instances used as training does not impact directly the accuracy of the ML techniques, except for JRip and OR. In summary, we noticed that increasing the number of training instances does not guarantee an improvement in accuracy.

### 3.4 Discussion About the Detection Rules

The answers to the RQs allowed us to understand the behavior of the ML techniques to detect code smells based on different developers' perceptions. We observed that: (RQ1) having few developers with the same perception of code smells is enough for training an ML technique, however, no technique obtained the best results for all the analyzed smells; (RQ2) ML techniques present disperse results of F-measure when considering individual perceptions of the developer; and (RQ3) increasing the number of training instances does not guarantee an improvement in the accuracy.

From these results, we can conclude that it would not be possible to use only one technique for all smell types. The accuracy of techniques varies per developer and smell. Thus, for further discussion, we complement our results by investigating the detection rules created by some ML techniques and providing insights on their behavior. Additionally, we compare the detection rules generated in our study with the rules present in the literature. More specifically, we compare the metrics present in the detection rules provided by related studies and the detection rules generated in our study.

**Comparison of our Detection Rules to Related Studies** In the previous sections, we analyzed the results of seven ML techniques. However, some techniques do not provide an understandable representation of the detection rules, considering the size and structure of the rules. For instance, RF provides hundreds of random trees used for the classification, which is not suitable for this discussion. On the other hand, the techniques J48, JRip, and OR express rules as a set of metrics and thresholds. J48 provides a pruned tree where the nodes are the conjunctions of metrics and the thresholds define the path until the leaves represent the decision label, which is used to classify an instance as (not) a smell. JRip and OR provide a logical combination of metrics and conditions based on thresholds. If the condition is satisfied, then, the instance receives the decision label. We adjusted the rules to present only the conditions that label a code fragment as smelly. In addition to the analysis

presented below, we provided more detailed rules for five different techniques for all smell types on our supplementary material (Oliveira et al. 2022).

For the analysis of the detection rules created by the ML techniques in our study, in comparison to what is reported in the literature, firstly we select the related studies. We choose related studies that (i) use rules based on metrics and thresholds, obtained both by ML techniques or other approaches; (ii) evaluate metrics and smell types that are also considered in our study, enabling comparison; (iii) describe rules obtained in a primary study or rules reported from other studies, i.e., secondary studies; and (iv) reports at least one detection rule for each smell type. Based on that, we compare the detection rules of our study to the rules of other four related studies, as follows: Arcelli Fontana et al. (2016) investigated the use of ML techniques in detecting code smells. Rasool and Arshad (2015) and Pecorelli et al. (2020) provide a list of metrics from different studies for detecting smell types. Lastly, Bigonha et al. (2019) use a catalog of metrics and thresholds to detect five types of code smells.

Table 8 presents the metrics observed in the detection rules that are reported in the four related studies and the metrics observed in the detection rules created in our study, for J48, JRip, and OR. The first column of the table shows the smell types, followed by the metrics reported to detect each respective smell in the related studies. The last two columns are the metrics used by the ML techniques from our study for *individual* (i.e., grouping the metrics of the models generated for each developer) and *global* (i.e., the model that considers all developers together) rules, respectively. For individual rules, we present only the metrics that compose rules for more than one developer independently of technique, to avoid the noise of very specific metrics. Also, the metrics at the beginning of the list are the most frequent ones, being present in detection rules for a higher number of developers. For global rules, we trained the ML techniques using the dataset with all developers. We could observe that the studies use different names for similar metrics, thus, we adjusted the name of metrics to ease the comparison. Also, there are pair of metrics that are not exactly the same metric but are considerably similar based on their description. We identified these cases with an asterisk (\*). For instance, for the smell Refused Bequest (RB), both Pecorelli et al and Bigonha et al. use a metric similar to BOVR, thus we presented them as BOVR\*.

In Table 8, we notice that the detection rules for the individual developers differ from the related studies, especially for the number of used metrics. The individual detection rules include metrics presented in the literature and also several other specific metrics. For instance, for the smell Speculative Generality (SG), on related study reports a detection rule using only the metric, namely NOC (number of children). This detection rule is similar to the one provided by J48 and JRip in the global rules: ( $NOC \geq 1$ ). However, in the individual detection rules, despite the NOC metric being one of the most frequently observed metrics (second position in the list), there are other 22 metrics that compose the detection rules. For instance, J48 produced the following detection rules for the developers 49 and 53, respectively: ( $NOC > 0$  and  $CountClassBase \leq 1$  and  $LOCactual \leq 29$ ) and ( $NOC > 0$  and  $NBD > 1$ ).

This behavior of having a larger set of metrics in the individual detection rules is similar for most of the smells, except for II, in which we do not have the metrics FANIN and FANOUT in any individual or global rules. For Middle Man (MM) and Primitive Obsession (PO) we have not observed the exact metrics, however, we observed semantically similar metrics. For instance, for MM the studies report the metric NOM, which counts the number of methods without inherited ones, and the individual rules listed the metric RFC, which counts the number of methods including the inherited ones. The same is observed for PO, in which the related studies use the metric NOV (number of variables) and our individual rules include GroupedVariables and Constants metrics that are also related to variables.

**Table 8** Comparison of most frequent software metrics present in the detection rules

Smell	Fontana et al.	Rasool et al.	Pecorelli et al.	Bigonha et al.	Our study (individual)	Our study (global)
DC	WOC, NOAM, RFC, AMW, CFNAMM, NOPVA, NIM, WMCNAMM	NOAM, NOV, LOCPROB, LOCRACTUAL, WOC, NOPA, WMC	-	NOC, DIT, NOA	LOCactual, RFC, AMW, WOC, WMCNAMM, CFNAMM, NOAM, IntelligentMethods, AvgLineBlank, ELOC, MaxCyclomatic, MaxCyclomaticStrict, NOC, AvgLine	WOC, IntelligentMethods
FE	ATFD, LAA, NOA, FDP, NMO	CBO, LCOM, ATFD, LAA, FDP	MC, ATFD	LCOM	LAA, ATFD, FANIN, CountLineComment, CyclomaticStrict, CountSemicolon, CountLineBlank, NMO, NOA, CountStmtDecl, FDP, Cyclo, CountStmtExe, CountPath, LOCactual, CountStmt, LOC	ATFD, LAA, FDP, NMO Essential, CountLineComment, CountStmtExe
GC	WMCNAMM	-	ELOC, WMC, NOA, LCOM	LCOM, WMC, NOA, NOM	WMC, MaxEssential, CountDeclMethodPrivate, MaxCyclomatic, ATFD, WMCNAMM, NOV, CBO, AvgCyclomatic, InnerClass, AvgLine, NProtM*, CountLineComment, DIT, CountClassBase	WMC, ATFD, MaxEssential
II	-	-	FANIN, FANOUT	-	AvgLineBlank, CountDeclInstanceVariable, DIT, AvgEssential, NProtM*, AvgCyclomatic, CountClassBase, CountDeclMethodDefault, AvgLineComment, CountLineBlank, RFC, NBD, MaxCyclomaticStrict, LOC	NOV, AvgCyclomatic, avgLineComment
LM	LOC, CYCLO	-	LOC, NP	MLOC, CYCLO, NBD	FANIN, CountStmtExe, MLOC, LOC, ELOC, CountLineComment, FANOUT, Cyclo, Essential, CountLineBlank, LOCprob*, CyclomaticModified	MLOC, ELOC, CountStmtExe

**Table 8** (continued)

Smell	Fontana et al.	Rasool et al.	Pecorelli et al.	Bigonha et al.	Our study (individual)	Our study (global)
MC	-	LMC, MC	-	-	Essential, LOCactual, FANIN, LMC, FANOUT, RatioCommentToCode, CountLineComment, LOCprob*, CountStmtDecl, NBD, ELOC	Essential, LMC, FANIN
MM	-	PDM, NOM	PDM	-	CountLineBlank, AvgCyclomatic, LOCactual, NIM, AvgLine, AvgLineComment, NOC, CountDeclMethodPrivate, RFC, LCOM, NBD, RatioCommentToCode, CBO, CountDeclInstanceVariable, CountDeclMethodDefault, CountLineComment	LOCactual, CountStmt
PO	-	NOV, VAVG	-	-	GroupedVariables, Constants, AvgCyclomatic, Primitives, RatioCommentToCode, NOC, CountDeclClassMethod, CountDeclMethodDefault, DIT, NOM, NBD, NProtM*, WMC, SumCyclomaticModified, MaxEssential	GroupedVariables, AvgLine, LOCprob
RB	-	BUR, NOA, NProtM, BOVR, AMW, WMC, NOM	BOVR*	BOVR*	AvgEssential, NOC, CountDeclMethodDefault, CountDeclClassMethod, RFC, MaxEssential, SumEssential, NIM, CountClassBase, MaxCyclomaticModified, NProtM*, CountLineComment, RatioCommentToCode, LOCactual, AvgCyclomatic, LCOM, SumCyclomaticModified, CountDeclMethodPublic, MaxCyclomatic, AvgLineBlank	NProtM*, CountClassBase CBO, AvgCyclomatic, RFC



**Table 8** (continued)

Smell	Fontana et al.	Rasool et al.	Pecorelli et al.	Bigonha et al.	Our study (individual)	Our study (global)
SG	-	-	NOC	-	AvgLineBlank, NOC, LOCactual, AvgLine, AvgLineComment, AvgEssential, CBO, CountLineBlank, AvgCyclomatic, CountStmtDecl, MaxCyclomatic, AvgCyclomaticStrict, AvgLineCode, CountClassBase, LOCprob*, DJT, CountDeclInstanceVariable, CountStmt, LOC, CountLineComment, NBD, MaxEssential,NIM	NOC, AvgLineCode

**Insight:** Individual and global detection rules proposed by J48, JRip and OR frequently include the software metrics proposed by the literature. Additionally, individual rules per developer tend to frequently combine popular metrics on literature with other metrics.

**Detection Rules for Individual Developers** Table 9 presents the detection rules created by the ML techniques J48, JRip, and OR for each developer individually for the smell Data Class. Despite being a smell with a fair agreement among developers, according to the Fleiss' Kappa (Table 3), it is possible to observe that each developer has a unique set of rules. The detection rules vary among developers and ML techniques. We can observe that several rules (for at least 5 out of 12 developers) contain popular metrics present in the literature, such as RFC, WOC, AMW, CFNAMM and WMCNAMM (Table 8). These metrics are seen especially in the study of Arcelli Fontana et al. (2016), which also evaluated the same three ML techniques in their study. However, some detection rules include only metrics not related to the code smell definitions found in the literature. For instance, the detection rules provided by J48 for developers #2 and #18 are composed only of metrics related to blank lines, namely AvgLineBlank. These unconventional metrics are also seen in the detection rules of other smells (Table 8). Fortunately, this is not recurrent when considering all rules from different techniques for the same developer. When considering the rules from all techniques for a developer, we obtain high F-measures values, as seen in Fig. 6a.

**Insight:** The detection rules created by different ML techniques for each developer are diverse. This confirms that ML techniques are developer-sensitive, even for cases with a fair agreement.

**Global Detection Rules** Table 10 presents the detection rules when considering the dataset with all developers together. In this case, we also observed the same behavior as in the detection rules for individual developers. The three ML techniques generated detection rules completely different for the same smell type for the majority of the cases. One exception is the case of Speculative Generality (SG) for J48 and JRip, already discussed above. Due to the different developers' perceptions of the same code snippet, some ML techniques are more affected than others based on the nature of the technique's approach. For instance, OR tends to create a unique and less error-prone detection rule. However, based on this behavior, the technique OR creates specific rules for the analyzed dataset, since it focused only on one metric for each type of smell. This impacts the use of the rules generated by OR in the scenario with a divergence of perceptions among developers.

**Insight:** The results reinforce the tendency of different ML techniques to generate diverse detection rules for the same smell type, as already discussed in the literature [3]. Here, the same situation is observed also in the scenario with low agreement among the developers. Furthermore, in this scenario, the technique OR generates classification rules strongly linked to the dataset, making it difficult or even impossible to reuse these generated rules in other projects.

**Table 9** Data class' developer-specific detection rules

Developer	148	OR	JRip
2	AvgLineBlank <= 0	(LOCactual < 68.0 or LOCactual >= 152.5) or CFNAMM < 1.5	(NOAM > 1 and CountDeclMethodDefault < 2) or CFNAMM < 2
17	CountStmtExe <= 28	CountSemicolon < 50.5	RFC < 33 and NOC < 1
18	AvgLineBlank <= 0.18	AvgLine < 12.79 or CFNAMM < 3.5 or AMW < 2.14	AvgLine < 11.27 or CFNAMM < 5 or NOAM > 0 or AMW < 2.64
19	ELOC <= 20 or LOCactual <= 55 or IntelligentMethods <= 0	LOCactual < 68.0 or AvgLine < 6.90 or RFC < 19.5 or WOC < 0.1	LOCactual <= 81 or WOC <= 0
21	MaxCyclomaticStrict <= 2 or IntelligentMethods <= 0	LOCactual < 109.5 RFC < 19.5 or WMCNAMM < 5.0 or AMW < 1.35 or WOC < 0.1	AMW < 1.5 or WMCNAMM < 6 or WOC < 0.2
22	(WMCNAMM > 2 and CFNAMM <= 15) or (WMCNAMM > 2 and AMW <= 1.64)	CountDeclMethodPublic < 5.5	(WMCNAMM > 2 and CFNAMM < 24) or (WMCNAMM > 2 and RFC < 34) or (WMCNAMM > 2 and AMW < 2.64)
27	CountLineBlank <= 11 or ELOC <= 17 or WMCNAMM <= 2 or LOCactual <= 53	AccessorsRatio >= 94.4	WMCNAMM <= 2 or LOCactual <= 53

**Table 9** (continued)

Developer	J48	OR	JRip
30	ELOC <= 17 or LOCactual <= 49 or CountLineComment <= 4 (PublicAttributes <= 0 and NOC > 0) or (PublicAttributes > 0)	NOAM >= 5.0	ELOC <= 17 or LOCactual <= 49
77	MaxCyclomaticStrict <= 2 or MaxCyclomatic <= 2 or IntelligentMethods <= 0	NOAM < 7.0 or WOC < 231.0 LOCactual < 109.5 or AMW < 1.05 or RFC < 19.5 or WMCNAMM < 5.0 or AMW < 1.35 or WOC < 0.1	– AMW < 1.5 or WMCNAMM < 6 WOC < 0.2
93	CFNAMM <= 0 or AvgLineBlank <= 0	CFNAMM < 1.5	CFNAMM < 2
104	MaxCyclomaticStrict <= 2 or MaxCyclomatic <= 2) or IntelligentMethods <= 0	LOCactual < 109.5 or AMW < 1.35 or RFC < 19.5 or WMCNAMM < 5.0 or WOC < 0.1	AMW < 1.5 or WMCNAMM < 6 or WOC < 0.2

**Table 10** Global detection rules per smell type

Smell	J48	OR	JRip
DCL	IntelligentMethods <= 0	WOC < 0.1	WOC < 0.2
FE	(FDP > 0 and Essential <= 1 and CountLineComment <= 2 and NMO <= 2 and CountStmtExe <= 9) or (FDP > 0 and Essential <= 1 and CountLineComment <= 2 and 4 NMO <= 2 and CountStmtExe > 9 and ATFD > 9)	(ATFD >= 0.5 and ATFD < 1.5) or (ATFD >= 4.5 and ATFD < 8) or ATFD >= 12.5	LAA < 0.47 or (LAA > 0.56 and LAA < 0.77)
GC	MaxEssential > 3	(ATFD > 42.5 and ATFD < 77.0) or ATFD >= 130.5	wmc >= 101
II	A_NOV > 2 and B_AvgCyclomatic > 2.2	(B_AvgCyclomatic >= 2.23 and B_AvgCyclomatic < 2.76) or (B_AvgCyclomatic >= 3.05 and B_AvgCyclomatic < 3.58) or (B_AvgCyclomatic >= 4.64 and B_AvgCyclomatic < 6.88)	B_AvgLineComment >= 7.46
LM	CountStmtExe > 32 and MLOC <= 168	(MLOC >= 74 and MLOC < 81.5) or (MLOC >= 87.5 and MLOC < 181)	(ELOC > 43 and MLOC < 194)
MC	(Essential <= 1 and LMC <= 4) or Essential > 1	(FANIN >= 15.5 and FANIN < 19.5) or (FANIN >= 1.5 and FANIN < 5.5) or FANIN >= 57.5	Essential >= 3 or LMC <= 4
MM	LOCactual <= 31 and CountStmt > 4	CountStmt >= 4.5 and CountStmt < 8	LOCactual <= 31

**Table 10** (continued)

Smell	J48	OR	JRip
PO	GroupedVariables > 0	(AvgLine >= 4.09 and AvgLine < 16.02) or (AvgLine >= 22.12 and AvgLine < 25.12) or (AvgLine >= 28.67 and AvgLine < 31.75) or (AvgLine >= 34.45 and AvgLine < 35.76) or (AvgLine >= 36.43 and AvgLine < 43.23)	LOCprob <= 148 or GroupedVariables >= 1
RB	(A_NProtM* <= 11 and A_CountClassBase <= 1 and B_CBO <= 2) or (A_NProtM* <= 11 and A_CountClassBase > 1)	A_AvgCyclomatic < 1.74 or (A_AvgCyclomatic >= 2.39 and A_AvgCyclomatic < 2.79) or (A_AvgCyclomatic >= 4.16 and A_AvgCyclomatic < 5.21) or A_AvgCyclomatic >= 15.79	A_RFC <= 39
SG	NOC > 0	(AvgLineCode >= 1.16 and AvgLineCode < 3.93) or (AvgLineCode >= 4.5 and AvgLineCode < 5.25) or (AvgLineCode >= 8.93 and AvgLineCode < 9.93)	NOC >= 1

### 3.5 Risk of the Developer-Sensitive Smell Detection

Creating approaches and tools that learn the preferences of developers can be risky in certain circumstances. For example, let us assume a case in which developers in a project are not aware of good development practices to avoid smells, *e.g.*, writing short methods easy to understand and maintain, or are not able to identify the existence of smells, *e.g.*, Feature Envy. If the training of ML models relies exclusively on the perception of these developers, a severe bias will be introduced in the smell detection, directly harming the internal quality of the software. For instance, based on the examples mentioned above, the source code will have many occurrences of long methods and feature envies without warning the developers. On one hand, our study calls attention to the need of taking into account the perception of developers about the existence of smells. On the other hand, we also stress the risk of how to use their perception as a unique ground truth. Thus, it is recommended that users (companies or developers individually) have strategies to mitigate such risks. For example, users can defined which developers will compose the training step or set minimum quality standard to be used as a sanity check of the performance of the ML models.

In our study, several developers evaluated the same instance of code, which enable us to observe the divergence of their perceptions. However, in a real-world scenario, in which each developer work on their specific pieces of code, each developer will evaluate their own instances of smells. This can lead to significant divergence in the generated detection rules, leading to pieces of code with diverse levels of quality within the same project. However, it is common that certain modules to have similar structural or semantic characteristics within the same system or across different of the same ecosystem, domain or organization. In these cases, different developers working on these modules may either have similar or diverging perceptions about a certain smell type. Thus, a comparison of their generated detection rules should: (i) either confirm that smell detection is likely to be coherent (and, thus, more likely to be accurate), or (ii) warn developers about potential inaccurate detection of smells (even if they are aligned with the individual perception of the developers working on those structurally or semantically).

Finally, the developer-sensitive smell detection requires smell instances of a specific developer to be used as training for the ML techniques. However, a developer might not have enough instances of a certain smell type, in the case of deciding for early detection of code smells. In this scenario, some harmful smell instances may be introduced at the beginning of the software development and remain present while the techniques are not completely trained. The later detection and removal of code smells may require additional efforts as well as a wast of time. Fortunately, the results of RQ3 indicated that the ML techniques do not require several instances to obtain a high F-measure.

### 3.6 Contributions and Implications

The methodology, results, findings, and insights of our study can contribute to different stakeholders, which are described in what follows.

**Researchers:** The first contribution of our study is to make available a package of supplemental material to support new studies or replications (Oliveira et al. 2022). Our supplementary material includes many code snippets, their metrics, and the developers' opinions on whether or not there are occurrences of 10 different smells types. In addition,

we provide the detection rules generated by the ML techniques globally or individually for each developer and for each smell type.

The results of our study also have direct implications for researchers. The result of RQ1 shows that there is no ML technique that is the best for all smell types. Different ML techniques can be used to target different smell types. Based on this, a new study can propose the use of hybrid approaches, i.e., not based on only one ML technique, that decide which technique among several available should be used based on the target smells. When considering the entire set of techniques, the results have been improved considerably for detection per developer (Fig. 6) and per smell type (Table 7).

The results from RQ1 also shows that when the agreement decreases, the dispersion of accuracy increases. This motivates the conduction of a new study with a focus on monitoring individual developers to evaluate when the agreement about smells decreases, and then suggest corrective actions such as creating guidelines or understanding what is the situation for the disagreement. For RQ3, we observed that the techniques do not tend to improve their accuracy as we increase the number of training instances. Our hypothesis is that this is due to the increase in divergence in the dataset that makes it difficult or even impossible to correctly classify all instances in case there are developers who totally disagree with each other. These observed side effects lead to the introduction of a question: Would it be possible to adjust the training to reduce the impact of the developers' perception?

**Practitioners:** The results and insights obtained in our study can support practitioners from several perspectives. An example is the case of developers' turnover. Here, the developer-sensitive smell detection approaches can quickly adapt to represent the perceptions of the new developers. Our results showed that even with a small number of instances, ML techniques can perform well to detect developer-sensitive smells. This implies that companies do not need to wait until having a large dataset to start using ML techniques to detect smells. This can be done since from the early stages of the projects. Additionally, we have the case of newcomer/novice developers. For this case, ML models created with the perception of senior/experienced developers can be used for training. For example, after a novice developer writes a piece of code, an ML model trained with the preferences of the project can check if the quality standards have been followed. Thus, one implication of our work is to use expert developer-sensitive trained models to integrate new developers into ongoing projects.

Another direct implication of our work is to reduce the number of false positives considering the perception of developers within a company or in a specific project. For instance, for complex features practitioners might decide to accept a complex code/design, i.e., with code smells, instead of expending a great number of resources to refactor such features. Hence, a model trained with this perception can avoid continuously indicating the smells in such features, deviating the attention of the practitioners.

The empirical results of our study revealed that in practice developers have different perceptions about smells. This implies that companies may focus on having clear and objective definitions of what are code smells for the projects. This does not mean that developers cannot have a disagreement, but can lead to a reduction in the divergence of what should be considered a code/design problem or not.

**Tool builders:** The first and direct implication of our study to tool builders is that automated support should consider the individual perception of each developer to reach



higher accuracy. This is related to reducing the number of false positives discussed above. We understand that not every tool support must focus on detecting only developer-sensitive smells, but tools may at least offer the option for users to configure/choose between using traditional smell detection approaches or the one based on developers' perception.

Certain ML techniques like J48, JRip and OR generate simpler detection rules that can be easily implemented in plugins for the IDE that already calculates software metrics. That is, ML techniques like these can be more easily used in practice than techniques that have more complex representations, making it difficult to implement and directly be integrated with the IDE.

We discussed the implication of our study to new studies to propose hybrid approaches that consider different ML techniques depending on the targeted (more severe) smell. This can also be leveraged by tool builders. Our results support such a case by reporting which model performs better for which smells. This can serve as a start point for new tool support.

**Educators:** Code smells are commonly the subject of software engineering courses as they provide concrete means for developers to understand recurring opportunities for refactoring. The literature defines what is a smell type, but there is often no consensus in the practice on when a piece of code is (or not) a smell given the abstract, subjective definition of a smell type. Our study highlights this point and provides empirical results that can be used as illustrative examples of the potential divergence among developers. Based on that, educators can make their students aware that the notion of what is a smell is dependent on the context, *e.g.*, domain, module or even certain project or organization practices.

Another contribution of our results is to call the attention to the trade-off between what must be detected as a smell and what does not deserve the attention from developers, such as the case of complex features discussed above for practitioners. Educators can rely on a discussion like this one to highlight the importance of considering diverse factors and decide on which is more important for the situation at hand.

## 4 Threats to Validity

In this section, we discuss the threats to validity in accordance with the criteria defined in Wohlin et al. (2000).

**Construct Validity** The datasets that supported our study were built from code fragments manually evaluated by developers. In this case, the developers evaluated each fragment by reporting the option “YES” or “NO”, referring to the presence or absence of a given code smell into the fragment. Providing only these two options may be a threat, since the developers could not inform the degree of confidence in their answers. However, we adopted such procedure aiming at ensuring that the developers were able to decide about the existence of a code smell and we could obtain a set of instances that enables to perform our study. In addition, code fragments used may contain more than one code smell. Thus, we ask explicitly to the developers about the existence of a specific smell. Besides, the existence of several smells types does not change the fact that the type we want to observe still exists. Finally, the chosen set of metrics used for training the ML techniques are specific for the studied smell, which improves its detection.

The developers who classified the code smells are not the same as the project's developers. However, it is normal that new developers work on legacy projects. Because of this, the ML techniques need to be able to evaluate in that context. Besides, it is impracticable for the project developers to classify all the code fragments since the projects have existed for years. Finally, often developers work in group in a certain code fragment, so we would not know which developer did each code fragment to have that correct evaluation.

In our RQ3, we gradually increase the number of instances to measure the efficiency of the ML techniques. However, once we are not using cross validation, we were susceptible to a bias based on the order of instances that have been used as training instances. To mitigate this threat, we trained the ML techniques using random instances. Also, we ensured that by increasing the dataset, all previous instances would remain present, thus inserting the perspective of new developers into the training set.

**Internal Validity** The use of the Weka package of the R platform to implement the techniques analyzed in our study enabled 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 in 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 Arcelli Fontana et al. (2016). Indeed, Arcelli Fontana et al. (2016) performed a variety of experiments in order to find the best adjust for each technique.

**External Validity** The code fragments evaluated by the developers were extracted from five Java projects. Such projects have been widely used in other studies about code smells (Khomh et al. 2009, 2011b; Moha et al. 2010; Maiga et al. 2012). However, although the implementation of these projects present classes and methods with different characteristics (i.e. size and complexity), our results might not hold to other projects. In the same way, even though we have performed our experiments with 63 different developers, our results might not also hold for other developers since they may have different perceptions about the code smells analyzed in our study (Mäntylä 2005; Mäntylä and Lassenius 2006; Schumacher et al. 2010; Santos et al. 2013).

## 5 Related Work

Several machine learning techniques have been adapted to automate the detection of code smells (Hozano et al. 2017a, b). Although these studies report interesting results concerning the accuracy and efficiency of ML techniques to detect code smells, there is still little knowledge about the sensitivity of ML techniques to detect smells based on different perception of developers.

In Khomh et al. (2009), the authors proposed the *Bayesian Belief Network* (BBN) to detect instances of *God Class*. Altogether, the study is composed of four graduate students that validated several classes. They were instructed to indicate if any of the validated class contains a *God Class* instance. For that, they built a dataset containing 15 smell instances. Then, they applied a 3-fold cross-validation on this dataset in order to evaluate the performance of the BBN. They obtained an accuracy of 0.68 on detecting *God Class*. In Khomh et al. (2011b), the authors extended the study Khomh et al. (2009) by applying the BBN

to detect instances of *Blob*, *Spaghetti Code*, and *Functional Decomposition*. They involved seven students to create datasets and then they evaluated the accuracy of BBN to detect these smell types.

The study described in Maiga et al. (2012) assessed the accuracy of SVM in the detection of four types of code smell: *Blob*, *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*. The SVM obtained an accuracy up to 0.74. In Amorim et al. (2015) the authors reported the accuracy of the DT technique to recognize code smells. They applied the technique in a dataset containing four open source projects. This dataset contains a huge number of instances validated by few developers. Also, the authors compared the results with a manual oracle containing detected smell from other detection approaches and other machine learning techniques. The results indicate that DT is able to reach an accuracy up to 0.78.

Arcelli Fontana et al. (2016) presented a large study involving 16 different ML techniques and 74 software systems. The study focused on four code smells, namely *Data Class*, *Large Class*, *Feature Envy*, and *Long Method*, comparing different configurations of machine learning techniques. They used a dataset containing 1986 manually validated code smell samples. The results indicated that *J48* and *Random Forest* obtained the highest accuracy, reaching values up to 0.95 with at least a hundred training examples. However, a recent study (Di Nucci et al. 2018) indicate that the dataset used by Arcelli Fontana et al. (2016) had a high bias in the accuracy obtained by the techniques.

Finally, Azeem et al. (2019) presented a systematic literature review. They focused on provide an overview and discuss the usage of ML techniques on the detection of code smells. They conclude that *God Class*, *Long Method*, *Functional Decomposition*, and *Spaghetti Code* are constantly investigated in literature. Also, Random Forest and JRip are the most effective classifiers in terms of performance. Finally, they mentioned that there is still room for the improvement of ML techniques in the context of code smell detection.

Other studies (Di Nucci et al. 2018; Pecorelli et al. 2019, 2020) focus on investigating the accuracy of ML techniques on detecting code smells through data balancing (Di Nucci et al. 2018). They investigated whether data balancing is able to improve the accuracy of machine learning techniques. However, their results indicate that the existing techniques for data balancing are not capable of significantly improving accuracy.

When considering the use of ML techniques taking into account developers perception, there are few studies (Hozano et al. 2017a, b, Oliveira et al. 2020). Hozano et al. (2017a) analyzed the accuracy and efficiency of six techniques in the detection of four smell types using a set of 600 examples of (non-) smells manually validated by 40 developers. Their results indicated that RF obtained the best accuracy among the techniques, around 0.63. Also, Hozano et al. (2017b) proposed the *Histrategy*, a guided personalization technique capable of detecting code smells that are sensitive to the perception of each developer. Altogether, four code smells were analyzed and obtained satisfactory results in terms of accuracy and training needs when compared to the other 6 ML techniques used in the study.

Although previous studies have taken into account the developers' perspective to verify the existence of the smells (Hozano et al. 2017a, b). They (i) investigated a smaller dataset, containing a reduced number of smell types and involved developers, (ii) They have a narrow discussion regarding the ML techniques' dispersion and (iii) They use a unique project in their dataset, which limits the external validity of their results. In this way, the observed results are limited to the specific domain of the evaluated project, which may not be truth for other domains. Finally, in a previous study, we investigated seven ML techniques in a dataset containing instances of six different code smells from ten active projects (Oliveira

et al. 2020). Although we considered the perception of the developers, our study was limited to the few instances of actual developers from considered projects. In other words, we studied only the smells that were refactored by these developers. The results showed that all analyzed techniques are sensitive to the type of smell, especially JRip and RF.

In summary, the results of this present paper reinforce and complement the findings of the previous studies (Hozano et al. 2017a, b; Oliveira et al. 2020). First, we could confirm that observed that the RF technique is a promising way to identify code smells even when taking into account developer-sensitive smells. Second, we observed that ML techniques accuracy are directly affected by different perceptions of the same smell and the smell type, even when considering a more robust dataset.

## 6 Conclusion

This paper presented a study to understand the behavior of ML techniques to detect code smells based on developers' perceptions. Firstly, we evaluated the overall accuracy of the ML techniques to recognize smells. Then, we investigated the dispersion (accuracy variation) of ML techniques on detecting smells for different developers. Finally, we analyzed the efficiency of the ML techniques by evaluating their accuracy according to the number of instances used to perform the training process. For a further discussion, we also presented the detection rules for J48, JRip and OR, described potential risks risk of developer-sensitive smell detection, and detail the contributions and implications of our work.

The results indicated that while most of the ML techniques reached similar F-measure on detecting smells, the SVM and JRip obtained the lowest values. We also observed that all the analyzed techniques are sensitive to the developers' perceptions and SVM is the most sensitive one. Besides that, we noticed that the agreement among developers' perceptions is inversely proportional to the ML techniques' accuracy. Finally, we could only observe a relationship between the number of training instances and the accuracy of the techniques for JRip and OR. For the other techniques, increasing the number of instances in the training stage does not indicate an improvement in the technique's accuracy.

In a deeper analysis of the results, we observed that the detection rules generated by the ML techniques differ for each developer. That is, the same smell type has several detection rules that reflect one or more developers' perceptions. Most of these detection rules are composed of metrics also used in the detection rules reported in the literature. However, the generated rules have several additional metrics that are specific to each developer. On the other hand, the detection rules obtained for all developers together (global rules) have similar metrics to previous studies.

Our findings suggest the increasing need for improving smell detection techniques by taking into account the individual perception of each developer. In future work, we intend to investigate a much wider range of smell types. In addition, we also intend to replicate this study in controlled scenarios, considering developers and projects of the same organization. In this way, we expect to identify if developers, who work together, have similar (or widely different) influences on the detection of the same code smells. In this way, we will be able to verify whether the effort to customize the ML techniques will be reduced (or increased) by the similarities (or divergences) among the developers. Finally, we intend to investigate whether experienced developers have a higher agreement than inexperienced developers when identifying different smell types.

## Appendix: List of Metrics

Metric	Description
AMW	Average method weight.
ATFD	Access to foreign data.
BOvR, PRM(similar), SIX(similar)	Base-class overriding ratio;BovR is the ratio of overridden methods to all methods of the given class' parent in the inheritance hierarchy.
BUR	BUR is the ratio of used protected members to all protected members of the given class' parent in the inheritance hierarchy.
CBO, CountClassCoupled	Coupling between objects.
CFNAMM	Called foreign not accessor or mutator methods.
CYCLO, VG, Cyclomatic	McCabe's cyclomatic complexity.
FDP	Foreign data providers.
LAA	Locality of attribute accesses.
LCOM, PercentLackOfCohesion	Lack of cohesion between methods.
LMC, chains	Length of the message chain.
MLOC	Lines of code of a method.
LOC, CountLineCode	Lines of code.
LOActual, CountLine	Total line of code.
LOCprob, CountLineCodeDecl	Number of lines of code for data fields, methods, imported packages, and package declaration.
NIM, CountDeclInstanceMethod	Number of instance methods.
NOA, NOP, NOF, CountDeclProperty	Number of attributes.
NOAM, NACC	Number of accessor methods (getter/setter).
NOC, NSC, CountClassDerived	Number of children.
NOM, CountDeclMethod	Number of methods without inherited ones.
NOMcalls, MC	Number of method calls.
NP	Number of parameters.
NOPA	Number of public attributes.
NOPVA	Number of Private Attributes.
NOV, CountDeclClassVariable	Number of class variables.
NProtM, CountDeclMethodProtected	Number of protected members.
RFC, CountDeclMethodAll	Number of methods, including inherited ones.
WMC, SumCyclomatic	Weighted methods per class.
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods.
WOC	Weight of a class.
NMO	Number of methods overridden.
ELOC, CountLineCodeExe	Effective Lines of Code.

Metric	Description
FANOUT, CountOutput	Max number of references from the subject class to another class in the system
PDM, NFM	Number of forwarding methods.
VAVG	Average count on the number of variables.
DIT, MaxInheritanceTree	Number of classes that are above a certain class in the inheritance hierarchy.
NBD, MaxNesting	Maximum number of nested blocks of a method.
TCC	The cohesion between the public methods of a class.
IntelligentMethods	Number of intelligent methods.
GroupedVariables	Number of grouped variables
Constants	Number of constants
Primitives	Number of primitives
AccessorsRatio	Ratio of accessors methods to other methods
PublicAttributes	Number of public attributes
InnerClass	Number of inner classes
AltAvgLineBlank	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	Average number of lines containing comments for all nested functions or methods, including inactive regions.
AltCountLineBlank	Number of blank lines, including inactive regions.
AltCountLineCode	Number of lines containing source code, including inactive regions.
AltCountLineComment	Number of lines containing comment, including inactive regions.
AvgCyclomatic	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	Average Essential complexity for all nested functions or methods.
AvgEssentialStrictModified	Average strict modified essential complexity for all nested functions or methods.
AvgLine	Average number of lines for all nested functions or methods.

Metric	Description
FANIN, CountInput	Max number of references to the subject class from another class in the system.
AvgLineBlank	Average number of blanks for all nested functions or methods.
AvgLineCode	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average number of lines containing comments for all nested functions or methods.
CountClassBase	Number of immediate base classes. [aka IFANIN]
CountDeclClass	Number of classes.
CountDeclClassMethod	Number of class methods.
CountDeclExecutableUnit	Number of program units with executable code.
CountDeclFile	Number of files.
CountDeclFunction	Number of functions.
CountDeclInstanceVariable	Number of instance variables. [aka NIV]
CountDeclInstanceVariableInternal	Number of internal instance variables.
CountDeclInstanceVariablePrivate	Number of private instance variables.
CountDeclInstanceVariableProtected	Number of protected instance variables.
CountDeclInstanceVariableProtectedInternal	Number of protected internal instance variables.
CountDeclInstanceVariablePublic	Number of public instance variables.
CountDeclMethodConst	Number of local const methods.
CountDeclMethodDefault	Number of local default methods.
CountDeclMethodFriend	Number of local friend methods. [aka NFM]
CountDeclMethodInternal	Number of local internal methods.
CountDeclMethodPrivate	Number of local private methods. [aka NPM]
CountDeclMethodProtectedInternal	Number of local protected internal methods.
CountDeclMethodPublic	Number of local public methods. [aka NPRM]
CountDeclMethodStrictPrivate	Number of local strict private methods.
CountDeclMethodStrictPublished	Number of local strict published methods.
CountDeclModule	Number of modules.
CountDeclProgUnit	Number of non-nested modules, block data units, and subprograms.
CountDeclPropertyAuto	Number of auto-implemented properties.
CountDeclSubprogram	Number of subprograms.
CountLineBlank	Number of blank lines. [aka BLOC]
CountLineComment	Number of lines containing comment. [aka CLOC]
CountLineInactive	Number of inactive lines.
CountLinePreprocessor	Number of preprocessor lines.

Metric	Description
CountPackageCoupled	Number of other packages coupled to.
CountPath	Number of possible paths, not counting abnormal exits or gotos. [aka NPATH]
CountPathLog	Log10, truncated to an integer value, of the metric CountPath
CountSemicolon	Number of semicolons.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtEmpty	Number of empty statements.
CountStmtExe	Number of executable statements.
CyclomaticModified	Modified cyclomatic complexity.
CyclomaticStrict	Strict cyclomatic complexity.
Essential	Essential complexity. [aka Ev(G)]
EssentialStrictModified	Strict Modified Essential complexity.
MaxCyclomatic	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	Maximum essential complexity of all nested functions or methods.
MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed.
MaxEssentialStrictModified	Maximum strict modified essential complexity of all nested functions or methods.
MaxNesting	Maximum nesting level of control constructs.
MinEssentialKnots	Minimum Knots after structured programming constructs have been removed.
PercentLackOfCohesionModified	100% minus the average cohesion for class data members, modified for accessor methods
RatioCommentToCode	Ratio of comment lines to code lines.
SumCyclomaticModified	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum of essential complexity of all nested functions or methods.
SumEssentialStrictModified	Sum of strict modified essential complexity of all nested functions or methods.

**Acknowledgements** This study was partially funded by CNPq grants 434969/2018-4, 312149/2016-6, 309844/2018-5, 421306/2018-1, 427787/2018-1 141276/2020-7 and 408356/2018-9; FAPERJ grants 22520-7/2016, 010002285/2019, 211033/2019, 202621/2019 and PDR-10 Fellowship 202073/2020; FAPPR grant 51435.



## Declarations

**Conflict of Interest** The authors declare that they have no conflict of interest.

## References

- Abbes M, Khomh F, Gueheneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 15th European conference on software maintenance and reengineering (CSMR). IEEE, pp 181–190
- Amorim L, Costa E, Antunes N, Fonseca B, Ribeiro M (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), ISSRE '15. IEEE Computer Society, Washington, DC, pp 261–269. <https://doi.org/10.1109/ISSRE.2015.7381819>
- Arcelli Fontana F, Mäntylä MV, Zaroni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng* 21(3):1143–1191
- Arcoverde R, Guimarães ET, Bertran IM, Garcia A, Cai Y (2013) Prioritization of code anomalies based on architecture sensitiveness. In: 27th Brazilian symposium on software engineering, SBES 2013, Brasilia, Brazil, October 1-4, 2013. IEEE Computer Society, pp 69–78. <https://doi.org/10.1109/SBES.2013.14>
- Azeem MI, Palomba F, Shi L, Wang Q (2019) Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. *Inf Softw Technol* 108:115–138. <https://doi.org/10.1016/j.infsof.2018.12.009>
- Bertran IM (2011) Detecting architecturally-relevant code smells in evolving software systems. In: Taylor RN, Gall HC, Medvidovic N (eds) Proceedings of the 33rd international conference on software engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011. ACM, pp 1090–1093. <https://doi.org/10.1145/1985793.1986003>
- Bertran IM, Arcoverde R, Garcia A, Chavez C, von Staa A (2012a) On the relevance of code anomalies for identifying architecture degradation symptoms. In: Mens T, Cleve A, Ferenc R (eds) 16th European conference on software maintenance and reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012. IEEE Computer Society, pp 277–286. <https://doi.org/10.1109/CSMR.2012.35>
- Bertran IM, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A (2012b) Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In: Hirschfeld R, Tanter É, Sullivan KJ, Gabriel RP (eds) Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012. ACM, pp 167–178. <https://doi.org/10.1145/2162049.2162069>
- Bertran IM, Garcia A, Chavez C, von Staa A (2013) Enhancing the detection of code anomalies with architecture-sensitive strategies. In: Cleve A, Ricca F, Cerioli M (eds) 17th European conference on software maintenance and reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013. IEEE Computer Society, pp 177–186. <https://doi.org/10.1109/CSMR.2013.27>
- Bigonha MA, Ferreira K, Souza P, Sousa B, Januário M, Lima D (2019) The usefulness of software metric thresholds for detection of bad smells and fault prediction. *Inf Softw Technol* 115:79–92
- Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees, Wadsworth and Brooks, Monterey
- Cohen WW (1995) Fast effective rule induction. In: Twelfth international conference on machine learning. Morgan Kaufmann, pp 115–123
- de Mello RM, Oliveira RF, Garcia A (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), pp 68–77. <https://doi.org/10.1109/ESEM.2017.13>
- Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A (2018) Detecting code smells using machine learning techniques: Are we there yet? In: IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp 612–621. <https://doi.org/10.1109/SANER.2018.8330266>
- Fernandes E, Vale G, da Silva Sousa L, Figueiredo E, Garcia A, Lee J (2017) No code anomaly is an island - anomaly agglomeration as sign of product line instabilities. In: Botterweck G, Werner CML (eds) Mastering scale and complexity in software reuse - 16th international conference on software reuse, ICSR 2017, Salvador, Brazil, May 29-31, 2017, proceedings. Lecture Notes in Computer Science, vol 10221, pp 48–64. [https://doi.org/10.1007/978-3-319-56856-0\\_4](https://doi.org/10.1007/978-3-319-56856-0_4)
- Fleiss JL (1971) Measuring nominal scale agreement among many raters. *Psychol Bull* 76(5):378

- Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A (2011) An experience report on using code smells detection tools. In: 2011 IEEE Fourth international conference on software testing, verification and validation workshops, pp 450–457. <https://doi.org/10.1109/ICSTW.2011.12>. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5954446>
- Fontana FA, Zanoni M, Marino A, Mäntylä MV (2013) code smell detection: towards a machine learning-based approach. In: 2013 IEEE International conference on software maintenance, pp 396–399. <https://doi.org/10.1109/ICSM.2013.56>. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6676916>
- Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston
- Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *J Am Stat Assoc* 32(200):675–701
- Gopalan R (2012) Automatic detection of code smells in java source code. Ph.D. thesis, Dissertation for Honour Degree The University of Western Australia
- Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. *ACM SIGKDD Expl News* 11(1):10–18
- Ho TK (1995) Random decision forests. In: Proceedings of the third international conference on document analysis and recognition, vol 1. IEEE, pp 278–282
- Holte R (1993) Very simple classification rules perform well on most commonly used datasets. *Mach Learn* 11:63–91
- Hozano M, Antunes N, Fonseca B, Costa E (2017a) 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, pp 474–482
- Hozano M, Garcia A, Antunes N, Fonseca B, Costa E (2017b) Smells are sensitive to developers!: on the efficiency of (un)guided customized detection. In: Proceedings of the 25th international conference on program comprehension, ICPC '17. IEEE Press, Piscataway, pp 110–120. <https://doi.org/10.1109/ICPC.2017.32>
- Hozano M, Garcia A, Fonseca B, Costa E (2018) Are you smelling it? Investigating how similar developers detect code smells. *Inf Softw Technol* 93(C):130–146. <https://doi.org/10.1016/j.infsof.2017.09.002>
- Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: 9th international conference on quality software. QSI'09. IEEE, pp 305–314
- Khomh F, Penta MD, Guéhéneuc YG, Antoniol G (2011a) An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empir Softw Eng* 17(3):243–275. <https://doi.org/10.1007/s10664-011-9171-y>
- Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2011b) Bdtex: a gqm-based bayesian approach for the detection of antipatterns. *J Syst Softw* 84(4):559–572. <https://doi.org/10.1016/j.jss.2010.11.921>
- Lantz B (2019) Machine learning with R: expert techniques for predictive modeling. Packt Publishing Ltd
- Lanza M, Marinescu R, Ducasse S (2005) Object-oriented metrics in practice, Springer, New York
- Maiga A, Ali N, Bhattacharya N, Sabane A, Gueheneuc YG, Aimeur E (2012) SMURF: a SVM-based incremental anti-pattern detection approach. In: 2012 19th Working conference on reverse engineering, pp 466–475. <https://doi.org/10.1109/WCRE.2012.56>
- Maneerat N, Muenchaisri P (2011) Bad-smell prediction from software design model using machine learning techniques. In: 2011 Eighth international joint conference on computer science and software engineering (JCSSE), pp 331–336. <https://doi.org/10.1109/JCSSE.2011.5930143>. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5930143>
- Mäntylä MV (2005) An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: 2005 International symposium on empirical software engineering, p 10. <https://doi.org/10.1109/ISESE.2005.1541837>
- Mäntylä MV, Lassenius C (2006) Subjective evaluation of software evolvability using code smells: an empirical study, vol 11, Springer. <https://doi.org/10.1007/s10664-006-9002-8>
- Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE international conference on software maintenance, ICSM '04. IEEE Computer Society, Washington, DC, pp 350–359. <http://dl.acm.org/citation.cfm?id=1018431.1021443>
- Mitchell TM (1997) Machine learning. McGraw-Hill series in computer science, McGraw-Hill, Boston. <http://opac.inria.fr/record=b1093076>
- Moha N, Guéhéneuc YG, Meur AFL, Duchien L, Tiberghien A (2009) From a domain analysis to the specification and detection of code and design smells. *Form Asp Comput* 22(3):345–361. <https://doi.org/10.1007/s00165-009-0115-x>. <http://link.springer.com/10.1007/s00165-009-0115-x>
- Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2010) DECOR: a method for the specification and detection of code and design smells. *IEEE Trans Softw Eng* 36(1):20–36. <https://doi.org/10.1109/TSE.2009.50>


- Munro M (2005) Product metrics for automatic identification of “Bad smell” design problems in java Source-Code. In: 11th IEEE International software metrics symposium (METRICS’05), pp 15–15. <https://doi.org/10.1109/METRICS.2005.38>
- Oizumi WN, Garcia AF, Sousa LS, Cafeo BBP, Zhao Y (2016) Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016. ACM, pp 440–451. <https://doi.org/10.1145/2884781.2884868>
- Oizumi WN, Sousa LS, Oliveira A, Carvalho L, Garcia A, Colanzi TE, Oliveira RF (2019) On the density and diversity of degradation symptoms in refactored classes: a multi-case study. In: Katinka Wolter, Schieferdecker I, Gallina B, Cukier M, Natella R, Ivaki NR, Laranjeiro N (eds) 30th IEEE International symposium on software reliability engineering, ISSRE 2019, Berlin, Germany, October 28–31, 2019. IEEE, pp 346–357. <https://doi.org/10.1109/ISSRE.2019.00042>
- Oliveira D, Assunção WKG, Souza L, Oizumi W, Garcia A, Fonseca B (2020) Applying machine learning to customized smell detection: a multi-project study. In: 34th Brazilian symposium on software engineering, SBES ’20. Association for computing machinery, New York, pp 233–242. <https://doi.org/10.1145/3422392.3422427>
- Oliveira D, Assunção WKG, Garcia A, Fonseca B, Ribeiro M (2022) Supplementary material—developers’ perception matters: Machine learning to detect developer-sensitive smells. <https://github.com/smellsensitive/smellsensitive.github.io/raw/main/dataset.rar>
- Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: 2013 28th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 268–278. <https://doi.org/10.1109/ASE.2013.669308>. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6693086>
- Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2014a) Mining version histories for detecting code smells. IEEE Trans Softw Eng 5589(c):1–1. <https://doi.org/10.1109/TSE.2014.2372760>
- Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD (2014b) Do they really smell bad? A study on developers’ perception of bad code smells. In: 2014 IEEE International conference on software maintenance and evolution, pp 101–110. <https://doi.org/10.1109/ICSME.2014.32>
- Pecorelli F, Di Nucci D, De Roover C, De Lucia A (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, MaLTeSQuE 2019. Association for Computing Machinery, New York, pp 19–24. <https://doi.org/10.1145/3340482.3342744>
- Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2020) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw 169:110693. <https://doi.org/10.1016/j.jss.2020.110693>. <http://www.sciencedirect.com/science/article/pii/S0164121220301448>
- Platt J (1998) Fast training of support vector machines using sequential minimal optimization. In: Schoelkopf B, Burges C, Smola A (eds) Advances in kernel methods—support vector learning. MIT Press. <http://research.microsoft.com/~jplatt/smo.html>
- Quinlan R (1993) C4.5: programs for machine learning. Morgan Kaufmann Publishers, San Mateo
- Rasool G, Arshad Z (2015) A review of code smell mining techniques. J Softw: Evol Process 27(11):867–895
- Santos JAM, de Mendonça MG, Silva CVA (2013) An exploratory study to investigate the impact of conceptualization in god class detection. In: Proceedings of the 17th international conference on evaluation and assessment in software engineering, EASE ’13. ACM, New York, pp 48–59. <https://doi.org/10.1145/2460999.2461007>. <http://doi.acm.org/10.1145/2460999.2461007>
- Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M (2010) Building empirical support for automated code smell detection. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement—ESEM ’10, p 1. <https://doi.org/10.1145/1852786.1852797>
- Silva AL, Garcia A, Cirilo EJR, de Lucena CJP (2013) Are domain-specific detection strategies for code anomalies reusable? An industry multi-project study. In: 27th Brazilian symposium on software engineering, SBES 2013, Brasilia, Brazil, October 1–4, 2013. IEEE Computer Society, pp 79–88. <https://doi.org/10.1109/SBES.2013.9>
- Sousa LS, Oliveira A, Oizumi WN, Barbosa SDJ, Garcia A, Lee J, Kalinowski M, de Mello RM, Fonseca B, Oliveira RF, Lucena C, de Paes RB (2018) Identifying design problems in the source code: a grounded theory. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp 921–931. <https://doi.org/10.1145/3180155.3180239>
- Sousa LS, Oizumi WN, Garcia A, Oliveira A, Cedrim D, Lucena C (2020) When are smells indicators of architectural refactoring opportunities: a study of 50 software projects. In: ICPC ’20: 28th international

- conference on program comprehension, Seoul, Republic of Korea, July 13-15, 2020. ACM, pp 354–365. <https://doi.org/10.1145/3387904.3389276>
- Spearman C (1904) The proof and measurement of association between two things. *Am J Psychol* 15(1):72–101
- Steinwart I, Christmann A (2008) Support vector machines. Springer Science & Business Media
- Surhone LM, Timpledon MT, Marseken SF (2010) Shapiro-Wilk test. VDM Publishing
- van Solingen R, Basili V, Caldiera G, Rombach HD (2002) Goal question metric (GQM) approach, Wiley, New York
- Vargha A, Delaney HD (2000) A critique and improvement of the cl common language effect size statistics of McGraw and Wong. *J Educ Behav Stat* 25(2):101–132
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell
- Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 682–691. <http://dl.acm.org/citation.cfm?id=2486788.2486878>

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Affiliations

Daniel Oliveira<sup>1</sup>  · Wesley K. G. Assunção<sup>1</sup> · Alessandro Garcia<sup>1</sup> · Balduino Fonseca<sup>2</sup> · Márcio Ribeiro<sup>2</sup>

Wesley K. G. Assunção  
wesleyklewerton@gmail.com

Alessandro Garcia  
afgarcia@inf.puc-rio.br

Balduino Fonseca  
balduino@ic.ufal.br

Márcio Ribeiro  
marcio@ic.ufal.br

<sup>1</sup> Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

<sup>2</sup> Computing Institute, Federal University of Alagoas, Maceió, Brazil