Understanding and Detecting Harmful Code

Rodrigo Lima
UFPE, Brazil
rsl@cin.ufpe.br

Jairo Souza
UFPE, Brazil
jrmcs@cin.ufpe.br

Baldoino Fonseca
UFAL, Brazil
baldoino@ic.ufal.br

Leopoldo Teixeira
UFPE, Brazil
lmt@cin.ufpe.br

Rohit Gheyi
UFCG, Brazil
rohit@dsc.ufcg.edu.br

Márcio Ribeiro
UFAL, Brazil
marcio@ic.ufal.br

Alessandro Garcia
PUC-Rio, Brazil
afgarcia@inf.puc-rio.br

Rafael de Mello
CEFET/RJ, Brazil
rafael.mello@cefet-rj.br

ABSTRACT

Code smells typically indicate poor design implementation and choices that may degrade software quality. Hence, they need to be carefully detected to avoid such poor design. In this context, some studies try to understand the impact of code smells on the software quality, while others propose rules or machine learning-based techniques to detect code smells. However, none of those studies or techniques focus on analyzing code snippets that are really harmful to software quality. This paper presents a study to understand and classify code harmfulness. We analyze harmfulness in terms of Clean, Smelly, Buggy, and Harmful code. By Harmful Code, we define a Smelly code element having one or more bugs reported. These bugs may have been fixed or not. Thus, the incidence of Harmful Code may represent a increased risk of introducing new defects and/or design problems during its fixing. We perform our study with 22 smell types, 803 versions of 13 open-source projects, 40,340 bugs and 132,219 code smells. The results show that even though we have a high number of code smells, only 0.07% of those smells are harmful. The *Abstract Function Call From Constructor* is the smell type more related to Harmful Code. To cross-validate our results, we also perform a survey with 60 developers. Most of them (98%) consider code smells harmful to the software, and 85% of those developers believe that code smells detection tools are important. But, those developers are not concerned about selecting tools that are able to detect Harmful Code. We also evaluate machine learning techniques to classify code harmfulness: they reach the effectiveness of at least 97% to classify Harmful Code. While the **Random Forest** is effective in classifying both Smelly and Harmful Code, the **Gaussian Naive Bayes** is the less effective technique. Our results also suggest that both software and developers' metrics are important to classify Harmful Code.

CCS CONCEPTS

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

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422420

KEYWORDS

Code Smells, Software Quality, Machine Learning

ACM Reference Format:
Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rohit Gheyi, Márcio Ribeiro, Alessandro Garcia, and Rafael de Mello. 2020. Understanding and Detecting Harmful Code. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422420

1 INTRODUCTION

During software development, developers perform changes to implement new requirements, to improve the code or to fix bugs. While those changes contribute to software evolution, they may introduce code smells, smells represent symptoms of poor design and implementation choices [13, 29]. The growing incidence of code smells is also an indicator of poor design [29], change and fault-proneness [33] as well as it may hinder comprehensibility [1]. Thus, code smell detection is an elementary technique to improve the software longevity.

Several approaches have been proposed to identify *code smells* in an automatic way. For instance, some studies [26, 31] propose techniques that rely on smell detection rules defined by developers or reused from other projects or tools. Other studies [3, 5, 21, 23, 30] indicate that Machine Learning techniques (ML-techniques) such as *Decision Trees* [3, 5], *Support Vector Machines* [30], *Genetic Programming* [21], and *Bayesian Belief Networks* [23], are a promising way to automate part of the detection process, without asking the developers to define their own smell detection rules. Even though studies [1, 29, 33] have analyzed the impact of code smells and there are techniques to detect those smells [3, 5, 21, 23, 26, 30, 31], none of those works focus on analyzing code snippets that are really harmful to software quality.

This paper presents a study to understand and classify code harmfulness. In our study, we consider bugs as the main harmfulness aspect as they induce software failures. This way, we assess the harmfulness of code snippets in terms of four categories: Clean: there is no smell or bug historically associated with the code; Smelly: code contains smells, but no bug is associated with it; Buggy: code element that has one or more bugs reported. These bugs may have been fixed or not, but no smell has been detected into the code history; Harmful: a Smelly code element that has one or more bugs reported, fixed, or not on its history. First, we analyze the occurrence of each category in open-source projects and which smell types are more related to Harmful Code. We also investigate the developers' perceptions regarding the harmfulness

of code smells. Moreover, we evaluate the effectiveness of machine learning techniques to classify the harmfulness of code snippets, focusing on Smelly and Harmful Code. Finally, we investigate which metrics are most influential in Harmful Code detection.

To perform our study, we define an experiment with 22 smell types, 803 versions of 13 open-source projects, 40,340 bugs mined from GitHub issues and 132,219 code smells collected by the tools *Designite*, *PMD*, and *Intellij*. The results show that even though we have a high number of code smells, only 0.07% of those smells are harmful. The *Abstract Function Call From Constructor* is the smell type more related to Harmful Code. To investigate the developers' perceptions, we perform a survey with 60 developers. Most of them (98%) consider code smells as harmful. Moreover, 85% of those developers believe that code smells detection tools are important, but these developers do not concern about selecting tools able to detect Harmful Code. Our results also indicate that machine learning techniques reach the effectiveness of at least 97% to detect Harmful Code. While the **Random Forest** [18] is effective on detecting both Smelly and Harmful Code, the **Gaussian Naive Bayes** [17] is the less effective technique. Finally, our results suggest software metrics (such as *CBO (Couple Between Objects)* and *WMC (Weight Method Class)*) are important to customize machine learning techniques to detect Harmful Code.

2 STUDY DESIGN

Several studies have investigated the impact of code smells [1, 29, 33] as well as tools and techniques to detect smells [3, 5, 21, 23, 26, 30, 31]. However, none of those works focus on analyzing code snippets that are really harmful to software quality. In this context, we try to answer four main research questions:

RQ$_1$. *Which is the incidence of Harmful Code in software projects?*

This research question assesses the presence of Clean, Smelly, Buggy, and Harmful code. As a result, we expect to understand the frequency of each category in open-source projects. This way, we can analyze whether smells might indicate bugs in such projects, as well as finding out the frequency of smells that are considered harmful. A high frequency of Harmful code indicates a close relation between smells and bugs. On the other hand, a low frequency of Harmful Code suggests that smells may not be a good proxy for bugs and, consequently, developers should focus their efforts on refactoring Harmful Code, instead of refactoring a high number of smells. After assessing Harmful code, we investigate which smell types are more harmful. The recognition of such smell types may help developers to be more cautious during software development to avoid these smell types.

RQ$_2$. *In which extent developers prioritize refactoring Harmful Code?*

In this research question, we analyze to which extent developers prioritize Harmful Code in refactoring tasks. This analysis may help us to understand if developers prioritize Harmful Code and if they believe that code smells are harmful to the software quality. Also, we investigate how developers handle code smells as harmful. In particular, we analyze the developer's perceptions regarding the use of appropriate tools to not only detect Smelly code but also Harmful Code.

RQ$_3$. *How effective are Machine Learning techniques to detect Harmful Code?*

After analyzing the existence of Harmful Code in open-source projects and how harmful are code smells for developers, we evaluate the effectiveness of ML-techniques to detect Harmful code in open-source projects. Several studies have indicated ML-techniques as an effective way to detect smells [3, 5, 6], but there is no knowledge if they are as effective in detecting Harmful code. As a result, we expect to shed light toward the use of ML-techniques to detect smells that are really harmful to the software. This way, developers could use existing tools and techniques not only to detect Smelly but also Harmful Code.

RQ$_4$. *Which metrics are most influential on detecting Harmful Code?*

Once we recognize the more effective Machine Learning techniques, we also analyze the metrics that have more influence to identify Harmful code. Identifying these influential metrics may help developers to avoid common code structures that are more likely to produce Harmful Code.

2.1 Projects Selection

We manually selected 13 Java projects according to the following criteria: (i) they must be open-source and hosted at GitHub; (ii) they must use the GitHub issues bug-tracking tool, to standardize our process of collecting bug reports (Section 2.4); and (iii) they must have had at least 3,000 smells and 20 bugs along their development.

Table 1 summarizes the characteristics of the selected projects. For each project, we analyze its complete history, from the first commit until the last one at the moment we collect the project information. The number of code smells ranges from 1,523 (*Apollo*) up to 121,165 (*Dubbo*), and the number of bugs varies from 22 (Apollo) up to 10,085 (Okhttp). The high numbers are due to analyzing the whole history of the projects.

Table 1: Analyzed Projects.

Project Name	Domain	# Versions	# Smells	# Bugs
Acra	Application	52	3,470	163
ActiveAndroid	Library	1	141	754
Aeron	Application	76	10,914	1,634
Aerosolve	Library	100	1,863	443
Apollo	Library	32	1,524	27
Butterknife	Library	45	1,636	861
Dubbo	Framework	62	27,843	2,216
Ethereumj	Cryptocurrency	50	9,169	2,763
Ganttproject	Application	31	8,275	4,365
Joda-time	Library	64	7,421	684
Lottie-android	Library	75	4,143	58
Okhttp	Library	80	9,412	10,111
Spring-boot	Framework	136	46,551	40

2.2 Code Smells

In our study, we analyze 22 smell types, as reported in Table 2. We select these smell types because: (i) they affect different scopes, i. e., methods, and classes; (ii) they are also investigated in previous works on code smell detection [2, 3, 5, 12, 23, 24, 30]; (iii) they have detection rules defined in the literature or implemented in available

Table 2: Selected Code Smells and Tools

Code Smell	Affected Code	Tool
Broken Hierarchy Cyclic-Dependent Modularization Deficient Encapsulation Imperative Abstraction Insufficient Modularization Multifaceted Abstraction Rebellious Hierarchy Unnecessary Abstraction Unutilized Abstraction	Class	Designite
Long Method Long Parameter List Long Statement Switch Statements Simplified Ternary Abst Func Call From Const Magic Number Missing default Missing Hierarchy Unexploited Encapsulation Empty catch clause	Method	PMD / Designite / Intellij Designite
Long Identifier	Class / Method	PMD / Designite / Intellij

tools[1][2]; and (iv) studies [16, 28, 33] indicate a negative impact of those smell types on software quality.

To collect code smells, we use three smell detection tools: *Designite*, *PMD*, and *Intellij*. These tools have already been used in previous studies [2, 12]. We execute these tools for each version of the software, in the entire history of the repository. For each class and method, we register the smells associated with it along with the software versions. One might argue that smell may be registered several times to the same class or method along with the versions. To mitigate this issue, we only count the smell if a class or method has changed when compared to the previous version. Furthermore, we conduct a careful manual validation on a sample of smells detected. Two pairs of researchers (familiar with code smells detection) from our research lab validated this sample. Each pair was responsible for a fraction of the sample, and each individual validated the same candidate smells. In Table 2, we present names, scope (i.e., method or class), and the tool used to collect each smell. More details about the instrumentation used by each tool to detect the analyzed smell types are available on our website[3].

2.3 Metrics

For each version of the analyzed projects, we compute metrics related to the source code and developers involved.

Source Code Metrics. We collect 52 metrics at class and method level, covering six quality aspects of object-oriented software: complexity, cohesion, size, coupling, encapsulation, and inheritance. We chose the CK (Chidamber and Kemerer) metrics [11], which are well-known and have been used by previous studies to detect code smells [3, 19, 24]. More details about the metrics, as well as the tools used to collect them, are described at our website[3].

Developer Metrics. Open-source environments, such as GitHub, made it easier for developers with different technical capabilities and social interactions to actively and simultaneously contribute to the same project. In such environments, developers can perform a variety of activities, such as committing code, opening and closing pull requests and issues, and discussing contributions. Even though developers can collaborate on different projects, their technical capabilities and social interactions can be determinant factors to the software quality. For example, a novice developer can introduce bugs when performing some change. Developer metrics may be a promising way to recognize which developers' actions may lead to bugs [4]. To perform our study, we collect 14 previously described developer metrics [4], related to three main factors: experience, contribution, and social aspects (i.e., number of followers). We collect metrics using *PyDriller* [36], a tool to support Git repository analysis. More details are described at our website[3].

2.4 Locating Bug Fixes

GitHub issues are useful to keep track of tasks, enhancements, and bugs related to a project. Furthermore, developers can label issues to characterize them. For example, an issue opened to fix a bug would typically be associated with the bug label. After fixing the bug, the issue can be closed. To collect the reports of fixed bugs in the selected projects, we mined the closed issues related to bugs in each project. To identify these issues, we verified whether they contained the bug or defect labels. As a result of this process, we collected 40,340 bug reports from the 13 analyzed projects. A similar approach was made in recent studies [4, 24, 33].

2.5 Locating the Buggy Code

GitHub provides the functionality to close an issue using commit messages. For example, prefacing a commit message with keywords "Fixes", "Fixed", "Fix", "Closes", "Closed" or "Close", followed by an issue number, such as, "Fixes #12345", will automatically close the issue when the commit is merged into the master branch. This way, when this strategy is used to close a bug issue, we assume the commit that closed the issue as being the bug fixing commit.

Associating a bug (issue) with a commit fixing allows us to identify the methods and classes that were modified to fix the bug. This way, we conservatively establish that in the immediate previous commit to the fix, these methods and classes are considered Buggy. We perform a similar approach by previous studies [4, 22, 33], which consists of assuming that the snippet is directly or indirectly linked to the bug if they were modified in the commit fix.

2.6 Locating Harmful Code

We consider a code snippet that already had bugs associated with it and still contains smells as Harmful. To identify Harmful Code, we first collect the code smells detected along with the software history (from the first software version until the current one). Then, we verify which code smells already had bugs associated to it. As a result, we obtain the code snippets that contain smells and already had bugs associated to it, i.e., the Harmful Code snippets.

2.7 Dataset Structure

After collecting metrics and classifying code snippets, we build our datasetEach instance in the dataset represents a code snippet extracted from the analyzed projects. Associated with each code snippet, we have the software and developer metrics as well as its harmfulness (Clean, Smelly, Buggy, or Harmful). While the

[1]http://tusharma.in/smells/
[2]https://pmd.github.io/latest/pmd_rules_java_design.html
[3]https://harmfulcode.github.io/

metrics represent characteristics (e. g., number of lines of code), the harmfulness (Clean, Smelly, Buggy and Harmful) indicates the category that the code snippet belongs. Our dataset is composed of 28,371,822 instances of code snippets, containing 28,216,238 Clean, 132,219 Smelly, 40,340 Buggy and 1,048 Harmful.

2.8 Calculating Effectiveness

We evaluate the effectiveness of seven ML-techniques to classify Smelly and Harmful Code as follows. First, we split the dataset into 11 smaller datasets, one for each smell type. They are extremely imbalanced, i.e., the percentage of Clean code is higher compared to the other categories (Smelly, Harmful, Buggy). Class imbalance can result in a serious bias towards the majority class, thus reducing the evaluation or classifications tasks [14]. To avoid this, we apply an under-sampling technique that consists on randomly and uniformly under-sampling the majority across other classes, similar to a recent study [6]. We chose under-sampling instead of over-sampling to avoid the generation of artificial instances of Smelly, Harmful, and Buggy, as performed by over-sampling techniques.

Next, we preprocess the dataset to avoid inconsistencies and to remove irrelevant and redundant features [3, 5, 6]. In particular, we normalize metrics with the default range between 0 and 1. Moreover, if two metrics present a Pearson correlation higher than 0.8, we randomly remove one of them.

After performing the preprocessing, we split each dataset into two sets: training and testing set containing $\frac{2}{3}$ and $\frac{1}{3}$ of the entire data respectively. We use the training data to learn the hyper-parameter configurations of the ML-techniques by applying a stratified 5-fold cross-validation procedure to ensure that each fold has the same proportion of observations with a given class outcome value, as previous studies do [5, 6]. Searching for the best hyper-parameter configuration for each technique is a complex, time-consuming, and challenging task. Each technique has a set of hyper-parameters that can be of different types, i.e., continuous, ordinal, and categorical, making it more difficult to calibrate them, due to the large space of hyper-parameters. The Grid Search algorithm [8] is the most common way to explore different configurations for each ML-technique. However, it would be very time consuming to explore the entire configuration search space for each technique. To avoid this, we tune the hyper-parameters using Hyperopt-Sklearn [25],[4] which consists of an automatic hyper-parameter configuration for the Scikit-Learn Machine Learning library. Hyperopt-Sklearn uses Hyperopt [9] to describe a search space over possible parameters configurations for different ML-techniques using Bayesian optimization. The use of Bayesian optimization algorithms is very effective and less time-consuming than the traditional approaches [25].

Once we learn the hyper-parameters configuration of the techniques, we evaluate the effectiveness of these techniques on unseen data — testing data. Previous studies [3] have used this procedure to avoid overfitting of the techniques, i. e., to avoid that, a technique becomes too specific to the training data, without it being generalized to unseen data. We measure effectiveness in terms of f-measure, which is typically used for evaluating the performance of classification tasks, providing a more realistic performance measure of a test, since it is computed by the harmonic mean of the precision and recall.

[4]https://github.com/hyperopt/hyperopt-sklearn

2.9 Machine Learning Techniques

In this study, we apply a variety of techniques based on popular algorithms that had a good performance in previous studies [7] on classifying code smells. We use the scikit-learn [34] python library in the following ML-techniques: K-nearest Neighbors (KNN); Decision Tree; Random Forest; Gradient Boosting; Gaussian Naive Bayes; Support Vector Machines (SVM) and Ada Boost [15].

2.10 Evaluating the Importance of Features

To evaluate the importance of each metric (feature) in the effectiveness of the ML-techniques analyzed, we use the *Shapley Additive Explanations* (SHAP) framework. It consists of a unified approach to explain the output of any ML model [27]. The SHAP framework uses game theory with local explanations to give an importance value to each feature used by the ML-technique that had the highest score to detect Harmful code. This way, we can find out which metrics are more important in the detection of Harmful code.

2.11 Conducting the Survey

We conducted an online survey with developers about their perceptions of code smells harmfulness as complementary to the quantitative results. Thus, we are providing a mixed-method study aiming for qualitative and quantitative evidence.

Survey Design. Our survey consisted of three main parts. The first part comprises four questions about developer demographics and experience, related to code smells. Then, three questions related to different aspects of code smells harmfulness, followed by possible additional comments justifying the responses. Finally, the last part contains one open-ended question to allow developers to comment. The main goal of conducting the survey was to further understand the perceptions of professionals regarding the harmfulness of code smells. The complete list of questions and more details about the survey are described at our website[3].

Participants. We sent the survey to a list of developers, including contributors of the 13 projects described in Section 2.1. Moreover, we included the survey in developers discussion forums such as Reddit[5], Hackernews[6] and Facebook[7] Groups, aiming to reach as many developers from diverse backgrounds. The survey ran for one week. As a result, we received the response of 60. Analyzing the background responses, we found that we had successfully targeted developers with high programming experience: about 59% have more than seven years of software development experience, while another 28% have between 4 and 6 years of experience. All the respondents have some knowledge about code smells.

2.12 Data Analysis

To answer $\mathbf{RQ}_1$, we analyze the frequency of Clean, Smelly, Buggy, and Harmful code in the projects. Moreover, we assess the proportion of Harmful code when compared to the number of smells.

[5]https://www.reddit.com/
[6]https://news.ycombinator.com/
[7]https://facebook.com/

Such analysis helps us to understand how close is the relationship between Smelly and Buggy code. We further investigate the Harmful code by analyzing which smell types are more harmful. In $\mathbf{RQ}_2$, we use an online survey to analyze in which extent developers prioritize refactoring Harmful Code. Also, we investigate the developers' perceptions on the harmfulness of Code Smells. In $\mathbf{RQ}_3$, we use the *f1-score* metric to analyze the effectiveness of ML-techniques to recognize Smelly and Harmful Code by following the procedures described in Section 2.8. Finally, to answer $\mathbf{RQ}_4$, we analyze which metrics are more important to the prediction model produced from the most effective ML-technique to detect Harmful code. From this analysis, we can identify code implementations that are more related to Harmful code and, consequently, help developers to avoid those implementations.

3 RESULTS AND DISCUSSIONS

In this section, we describe and discuss the main results of the study. We structure the data presentation and discussion in terms of our three research questions.

$\mathbf{RQ}_1$. Which is the incidence of Harmful Code in software projects?

In this research question, we analyze the code smells harmfulness by assessing the proportion of smells that are associated with bugs. Table 3 presents the number and proportion of code snippets considered as Clean, Smelly, Buggy, or Harmful.

Clean. All the analyzed projects present more than 96% of Clean code. This high proportion is expected since we analyze all software versions along with its evolution history. Although we have been careful by avoiding unchanged code snippets between versions, the code snippets are frequently changed over time. This high proportion has pros and cons. **High proportion means that developers could focus their efforts on a small part of the source code to identify smells or bugs.** On the other hand, a lower proportion could mean that the software has many smells or bugs, which can be harmful to quality but can also be useful for researchers to better understand inherent characteristics of this type of code.

Table 3: Code Category

Project	# Clean	# Smelly	# Buggy	# Harmful
Acra	195,874 (98.18%)	3,470 (1.74%)	141 (0.07%)	22 (0.01%)
Aeron	1,861,703 (99.33%)	10,914 (0.58%)	1,489 (0.08%)	145 (0.01%)
Aerosolve	532,715 (99.57%)	1,863 (0.35%)	308 (0.06%)	135 (0.03%)
Apollo	313,379 (99.51%)	1,523 (0.48%)	22 (0.01%)	5 (0%)
Butterknife	74,153 (96.74%)	1,636 (2.13%)	857 (1.12%)	4 (0.01%)
Dubbo	2,854,863 (98.96%)	27,843 (0.97%)	1,939 (0.07%)	277 (0.01%)
Ethereumj	1,690,701 (99.3%)	9,168 (0.54%)	2,638 (0.15%)	125 (0.01%)
Ganttproject	4,263,646 (99.7%)	8,275 (0.19%)	4,077 (0.1%)	288 (0.01%)
Joda-time	3,021,538 (99.73%)	7,420 (0.24%)	676 (0.02%)	8 (0%)
Lottie-android	499,291 (99.17%)	4,143 (0.82%)	52 (0.01%)	6 (0%)
Okhttp	1,422,553 (98.65%)	9,412 (0.65%)	10,085 (0.7%)	26 (0%)
Spring-boot	11,485,822 (99.6%)	46,551 (0.4%)	33 (0%)	7 (0%)

Smelly. Different from Clean code, the projects present a low proportion of Smelly code. The projects have at most 2.13%. Although the proportion of Smelly code is lower than Clean, smells are still present in a large portion of the code snippets. The number of Smelly code snippets in the projects varies from 1,523 (*Apollo*) up to more than 27 thousand (*Dubbo*). Such results indicate that **code smells detection tools still identify a large number of smells, making it difficult for developers to use them in practice**.

Buggy. The proportion of Buggy code is lower than Smelly code. The projects contain a proportion lower than 2%. But, this low proportion does not mean a low number of Buggy code snippets. The projects contain from 22 (*Apollo*) up to more than 10,085 (*Okhttp*) Buggy code snippets. Which means that **developers still have to concern with a considerable number of Buggy code snippets to inspect**.

Harmful. Similarly to Buggy code, the projects also present low proportion of Harmful code. The number of Harmful code snippets is much lower than the number of Buggy ones, varying from only four (*Butterknife*) up to 288 (*Ganttproject*). This suggests that **even though existing tools detect a large number of smells, there is still a high number of bugs that are not related to those smells**. Such results reinforce a previous study [32], indicating that only a few smells types are considered as harmful by developers. Hence, we also analyze the smell types more related to Smelly and Harmful code. Table 4 describes the smell types analyzed in our study, the number of smells, and Harmful Code associated with each type.

Table 4: Harmful Smell Types.

Smell Type	# Code Smells	# Harmful Code
Abstract Function Call From Constructor	118 (92.2%)	10 (7.8%)
Broken Hierarchy	18,152 (99.8%)	30 (0.2%)
Cyclic-Dependent Modularization	20,448 (99.6%)	79 (0.4%)
Deep Hierarchy	88 (98.9%)	1 (1.1%)
Deficient Encapsulation	31,774 (99.8%)	56 (0.2%)
Empty catch clause	8,116 (99.7%)	23 (0.3%)
Imperative Abstraction	1,100 (98%)	23 (2%)
Insufficient Modularization	20,351 (99.6%)	75 (0.4%)
Long Identifier	4,588 (99.7%)	15 (0.3%)
Long Method	11,371 (99.4%)	70 (0.6%)
Long Parameter List	10,839 (99.4%)	61 (0.6%)
Long Statement	52,701 (99.7%)	169 (0.3%)
Magic Number	40,388 (99.6%)	155 (0.4%)
Missing default	3,888 (99.7%)	12 (0.3%)
Missing Hierarchy	908 (99.8%)	2 (0.2%)
Multifaceted Abstraction	885 (99.4%)	5 (0.6%)
Rebellious Hierarchy	671 (99.3%)	5 (0.7%)
SimplifiedTernary	314 (98.4%)	5 (1.6%)
Switch Statements	7,740 (99.3%)	58 (0.7%)
Unexploited Encapsulation	688 (99.7%)	2 (0.3%)
Unnecessary Abstraction	665 (99.7%)	2 (0.3%)
Unutilized Abstraction	128,751 (99.9%)	188 (0.1%)

Smell Types Harmfulness. *Abstract Function Call From Constructor* presents the highest percentage of Harmful code, reaching 7.8% of Harmful smells. Both *Imperative Abstraction* and *SimplifiedTernary* present a slightly lower percentage than *Abstract Function Call From Constructor*, reaching 2% and 1.6% of Harmful Code, respectively. While the *Switch Statements*, *Rebellious Hierarchy*, *Long Method*, *Long Parameter List* and *Switch Statements* present percentage between 0.6% and 0.7%, the *Magic Number*, *Cyclic-dependent Modularization* and *Long Identifier* present percentage between 0.3% and 0.4%. Even tough some smell types present a percentage higher 1%, in half of the smell types analyzed the percentage of Harmful Code is close to zero.

Summary $\mathbf{RQ}_1$: Code smell detection tools identify a large number of smells, but only a few of them are associated with bugs. The *Abstract Function Call From Constructor* presents the highest proportion of Harmful Code.

RQ$_2$. In which extent developers prioritize refactoring Harmful Code?

This research question focuses on analyzing to which extent developers prioritize Harmful Code in refactoring tasks. We also investigate other aspects that can help us to understand better the refactoring prioritization. In particular, we also investigate how harmful are code smells to the software quality and the importance of existing code smells detection tools from the developers' perspective. Figure 1(a)-(c) summarizes the survey results.

Refactoring Prioritization Most of the developers (30%) prioritizes Harmful Code when refactoring, (21.7%) prioritizes Buggy Code, and (23.3%) Code Smells, as shown in Figure 1(a). Looking in another perspective, note that 51.7% and 53.3% of the developers prioritize code associated with bugs or smells, respectively, which suggests that developers care about bugs and smells when refactoring:

> "When refactoring, it is ideal for dealing with all modules of the code, but due to some limitations, such as time, it is most important to look at fragile parts as Harmful Code."

> "Since a smell only indicates there might be a problem and Harmful Code definitely does have a problem, you should look at that first."

Code Smells harmfulness. The vast majority of the developers (98.3%) consider code smells harmful to the software quality, as shown in Figure 1(b). This result suggests that developers believe in the harmfulness of code smells. Some of them left some comments regarding the code smells harmfulness:

> "Code smells often indicate or lead to bigger problems. Those bigger problems can make a code base fragile, difficult to maintain, and prone to errors."

> "Code smells make software readability and comprehensibility worse. That itself already degrades software quality. Moreover, code smells can make it harder to find bugs in the software, since you can only find bugs in code that you can read, and understand."

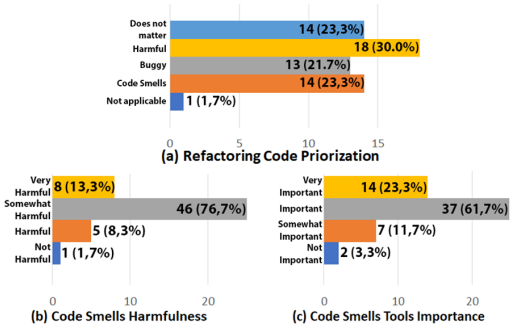

Figure 1: Survey Results.

Importance of Code Smells Tools. Our results indicate that developers can be convinced that code smells are harmful to the software, but they may not be aware that the detection tools are not previously configured to detect smells that are really harmful. In fact, we observe that 85% of the developers believe that code smell detection tools are important, as shown in Figure 1(c). But, when we analyze the comments, there is no concern in selecting tools able to detect code smells that are really harmful to the software quality. Most of the developers mention code smell detection tools that we use in our study.

> **Summary RQ$_2$:** Majority of the developers prioritize code associated with bugs or smells, and most of them prioritizes **Harmful Code** when refactoring code.

RQ$_3$. How effective are Machine Learning techniques to classify Harmful Code?

In RQ1, we observed that even though code smells detection tools identify a large number of smells, only a few of them are really harmful. On the other hand, the results of the RQ2 indicate that developers care about Harmful Code. Hence, it is important to investigate whether ML-techniques classify Harmful Code as effective as they classify code smells.

Although we analyze 22 smell types in our study, we evaluate the effectiveness of the ML techniques in 11 types. The remaining smell types present a low number of bugs associated with them, as described in Table 4. Figures 2 and 3 present the effectiveness of the ML-techniques to recognize *Smelly* and *Harmful* code, respectively. In each figure, the y-axis describes the *f-measure* value reached by each technique on classifying the analyzed smell type. In addition, we attach the exact value to the bar associated with each technique.

Harmful. For the 11 smell types analyzed, the ML techniques could reach high effectiveness on classifying Harmful code, reaching an effectiveness of at least 97% in all the cases analyzed. In particular, **Random Forest reaches an effectiveness equal or greater than the other algorithms in ten of the 11 smell types. Also, *Gradient Boosting* present an effectiveness slightly lower than *Random Forest***, reaching an effectiveness equal or greater than the other algorithms in nine smell types analyzed. Both techniques are more effective in all smells types, except in the *Deficient Encapsulation* smell type, where *Decision Tree* is more effective. On the other hand, ***GaussianNB* reaches the lowest effectiveness in six of the smell types** (*Magic Number*, *Insufficient Modularization*, *Unutilized Abstraction*, *Cyclic-Dependent Modularization*, *Deficient Encapsulation* and *Long Statement*).

Smelly. Differently from Harmful code, the ML techniques could not reach a high effectiveness on classifying some smell types. While the ML techniques reach an effectiveness of at least 97% on classifying Harmful Code, these techniques could not reach an effectiveness greater than 75% in five of the 12 smell types analyzed (*Switch Statements*, *Long Identifier*, *Insufficient Modularization*, *Cyclic-Dependent Modularization* and *Deficient Encapsulation*). **Similarly to Harmful code, *GaussianNB* presents a low effectiveness, reaching the lowest value in six of the smell types analyzed.**. On the other hand, ***Gradient Boosting* is the most effective algorithms** with an effectiveness equal or greater than *Random Forest* and *SVM* in six smell types (*Long Identifier*, *Long*

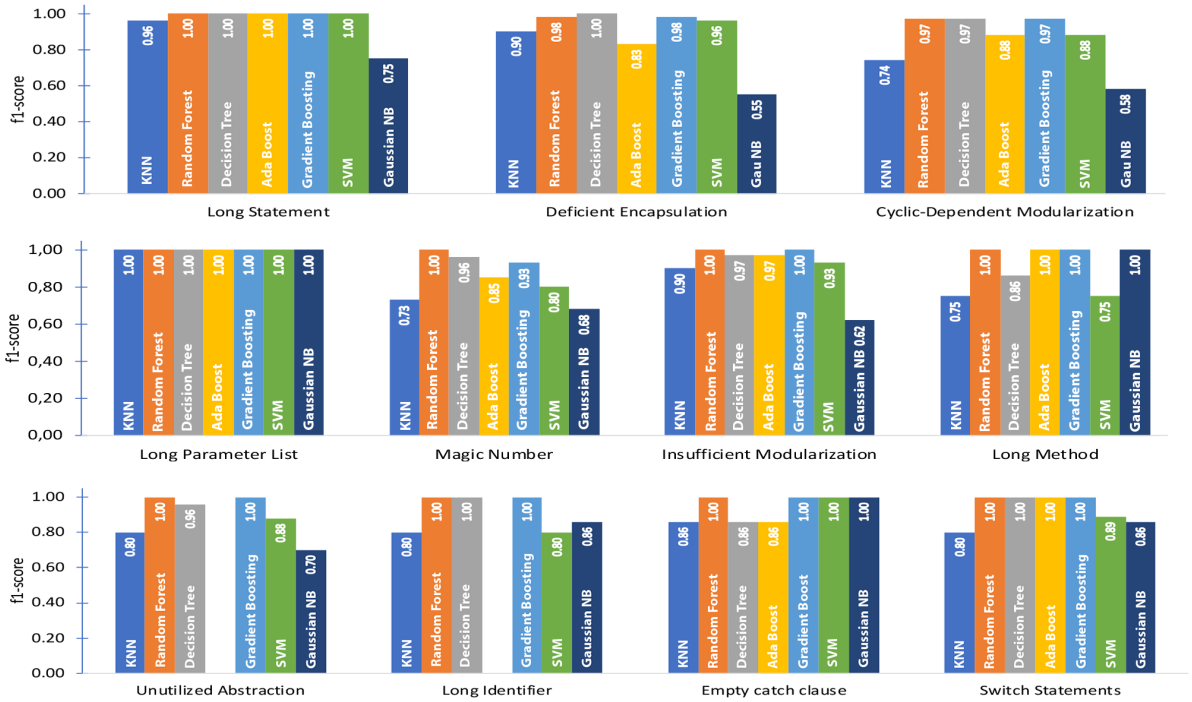

Figure 2: Results for Harmful Code (f1-score).

Parameter List, *Unutilized Abstraction*, *Cyclic-Dependent Modularization*, *Long Method* and *Long Statement*). Both *Random Forest* and *SVM* reach the highest effectiveness in three smell types.

> **Summary RQ$_3$:** ML-techniques classify Harmful Code as effective as they identify code smells. While *Random Forest* and *Gradient Boosting* are effective on detecting both Smelly and Harmful code, *GaussianNB* is the less effective technique.

RQ$_4$. Which metrics are most influential on classifying Harmful Code?

Figure 4 shows the results of the most important metrics in terms of *SHAP* values for each smell.

Parameters Amount (PA). The *PA* metric has the highest importance on detecting Harmful Code in the *Long Statement* smell type. It reaches an importance of 0.1951. In the case of the *Deficient Encapsulation* and *Long Parameter List*, the *PA* metric has an importance slightly lower than *QUW* and *NQ*, reaching a value of 0.0709 and 0.1176 respectively.

Weight Method Class (WMC). The metric has the highest importance on detecting Harmful Code in the smell types *Insufficient Modularization*, *Empty Catch Clause* and *Deficient Encapsulation*. It reaches an importance of 0.0446 in the *Insufficient Modularization* and 0.1467 in the *Empty Catch Clause*. In the case of the *Deficient Encapsulation*, the *PA* metric has an importance slightly lower than *WMC*, reaching a value of 0.0709.

Math Operations Quantity (MOQ), Numbers Quantity (NQ), and Couple Between Objects (CBO). These metrics reach the highest importance in *Long Identifier*, *Long Parameter List* and *Magic Number*. While *MOQ* reaches an importance of 0.1035, in the *Long Identifier*, *NQ* reaches an importance of 0.1211 in the *Long Parameter List*. In the case of *Magic Number*, only *CBO* could reach an importance above 0.0500.

Returns (RE) and Lines of Code Method (LOCM). The *RE* and *LOCM* metrics present the highest importance in *Cyclic-Dependent Modularization* (0.0699) and *Unutilized Abstraction* (0.0723), respectively. *Assignments Qty* reaches the highest importance in the *Long Parameter List*, (0.1590). Also, these metrics metric have the second highest importance in the smell types *Magic Number* and *Long Statement*, reaching a value of 0.0430 and 0.0321 respectively.

Developer Metrics. Among the developer metrics analyzed in our study, only *Number of Commits (NC)* reaches some importance. In this case, this metric presents the importance of 0.0016 in the *Deficient Encapsulation*. The remaining metrics present importance below 0.0010 in this smell type. One might argue that developer metrics are not important to detect Harmful Code since only one reaches some importance. However, note that we analyze the importance of 52 software metrics and only 11 developers' metrics. Even analyzing a greater number of software metrics, the developer ones could reach some importance in the *Deficient Encapsulation*. These results show that it is worthwhile to investigate developer metrics as other contexts in software engineering (i.e., bug prediction), but it is still early to affirm that they are useful since it is the first study that utilizes developer metrics in the context of code smells.

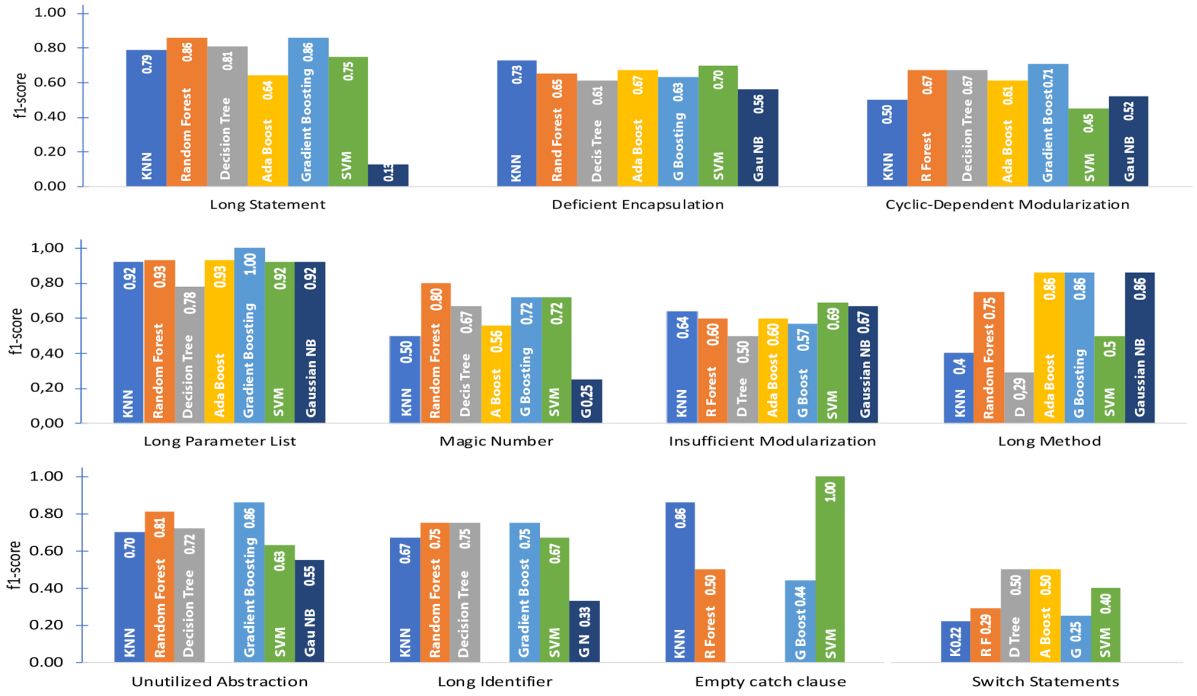

Figure 3: Results for Smelly Code (f1-score).

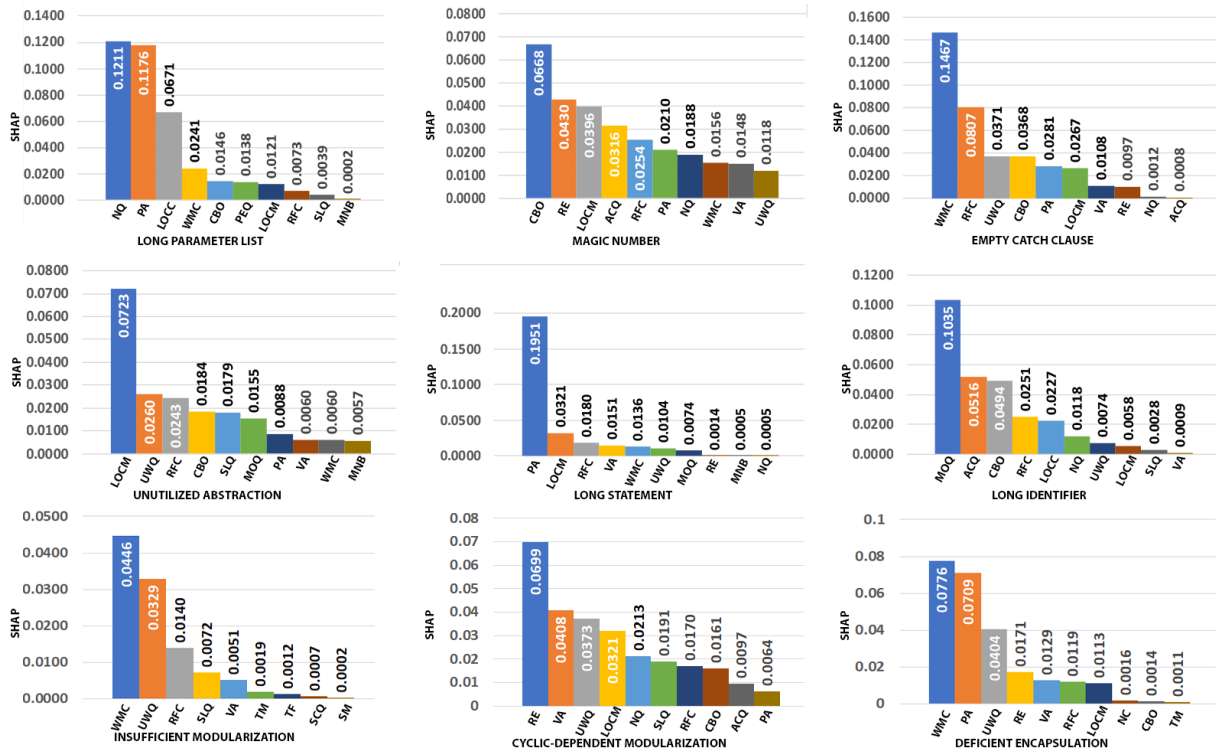

Figure 4: Most Important metrics among code smells (SHAP values).

Summary RQ_4: Software Metrics reach the highest importance. *Weight Method Class (WMC)* present the highest importance in three smells. *RE*, *MOQ*, *NQ*, *PA*, *CBO*, *LOCM* metrics reach the highest importance in the remaining smells. Developers metrics (such as *Number of Commits (NC)*) reach some importance in the smell *Deficient Encapsulation*.

4 LIMITATIONS AND THREATS TO VALIDITY

Construct Validity. In our study, we collected a set of code smells that were not manually validated. To mitigate this threat, we used tools and configurations used in previous studies [3, 5, 6, 19, 20]. Another threat to validity is to identify commits that fixed bugs correctly. GitHub provides the functionality to close issues by commits messages or pull requests comments. We mitigated this threat by identifying as the bug-fix, the commits, or pull requests (the last commit) that close issues labeled as "bug" or "defect" using this functionality. Besides, we identified methods and classes associated with each *bug* and establish the immediate previous commit of these methods and classes as Buggy Code. However, some buggy methods and classes could not have a bug directly associated with him since developers may be working on code improvements or new features during a fix. Also, some types of Harmful Code may not be identified because the tools used in the research not identify all the types of code smells. Other characteristics reported by the developers in the survey of **Harmful Code** like Legibility and Financial Cost could be explored in our dataset, but due to the limitation of resources and time, we opted the only use Bugs and Smells both characteristics were the most reported.

Internal Validity. Another threat is the procedures of the steps adopted in Section 2.8. These steps are related to the type of dataset splitting, selection of hyper-parameters, and construction of the ML-techniques. To mitigate this, we relied on decisions made by previous studies that obtained good results detecting code smells using ML-techniques [7].

External Validity. Regarding the validity of our findings, we selected only projects in which the primary language adopted is *Java*. Although we have selected a large number of projects from six different domains with different sizes and developers, our results might not be generalized to other projects which *Java* is not the primary language as they may have different characteristics.

5 RELATED WORK

Impact. Recent studies [28, 33] investigated the impact of code smells in the change and fault-proneness. Palomba et al. [33] conducted a study on 395 releases of 30 open-source projects and considering 17,350 code smells manually validated of 13 different types. The results show that classes with code smells have a higher change-and-fault-proneness than smell-free classes. Moreover, another study [28] investigated the relationship between smells and fine-grained structural change-proneness. They found that, in most cases, Smelly classes are more likely to undergo structural changes. Tracy et al. [16] investigated the relationship between faults and five smells: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. They collected the fault data from changes and faults in the repositories of the systems. Their findings suggest that some smells do indicate fault-prone in some circumstances, and they have different effects on different systems. Besides, all these studies provide evidence that bad code smells have negative effects on some maintenance properties. Our study complements them on pointing the smells that are really harmful to the software, helping developers prioritize them while refactoring.

Detection Tools and Techniques. Previous studies [3, 5, 23, 24, 30] have indicated the use of machine learning techniques as a promising way to detect code smells. These techniques use examples of code smells previously reported by the developer in order to learn how to detect such smells. In this way, Amorim et al. [3] present a preliminary study on the effectiveness of decision tree techniques to detect code smells by using four open-source projects. The results indicate that this technique can be effective to detect code smells. Other studies [23, 30] also inferred ML techniques to detect code smells by using Bayesian Belief Networks [23] or Support-vector machine [30]. Hozano et al. [19] introduced Histrategy, a guided customization technique to improve the efficiency on smell detection. Histrategy considers a limited set of detection strategies, produced from different detection heuristics, an input of a customization process. The output of the customization process consists of a detection strategy tailored to each developer. The technique was evaluated in an experimental study with 48 developers and four types of code smell. The results show that Histrategy is able to outperform six widely adopted machine learning algorithms – used in unguided approaches – both in effectiveness and efficiency. Fontana et al. [6] extended their previous work [5] by applying several machine learning techniques, varying from multinominal classification to regression. In this study, the authors modeled the code smell harmfulness as an ordinal variable and compared the accuracy of the techniques. Even though the previous studies [3, 5, 23, 24, 30] have contributed to evidence the effectiveness of ML-techniques in the detection of code smells, none of these studies analyze how effective are these techniques to detect Harmful Code. Our study complements all these approaches. Specifically, we focus on the identification of *Harmful Code*, supported by the addition of new features (Developers' Metrics and *Bugs*) approach similar to Catolino et al. work [10].

Developers' Perceptions. Palomba et al. [32] conducted a survey to investigate developers' perception on bad smells, they showed to developers code entities affected and not by bad smells, and asked them to indicate whether the code contains a potential design problem, nature, and severity. The authors learned the following lessons from the results: I. There are some smells that are generally not perceived by developers as design problems. II. The instance of a bad smell may or may not represent a problem based on the "intensity" of the problem. III. Smells related to complex/long source code are generally perceived as an important threat by developers. IV. Developer's experience and system's knowledge pay an important role in the identification of some smells. Sae-Lim et al. [35] investigated professional developers to determine the factors that they use for selecting and prioritizing code smells. They found that *Task Relevance* and *Smell Severity* were most commonly considered during code smell selection, while *Module Importance* is employed most often for code smell selection. Our study supports those previous studies. Our survey results confirm their results on suggesting that developers consider factors related to readability, maintainability, cost, and effort to fix while detecting smells.

6 CONCLUSIONS

This paper presented a study to understand and classify code harmfulness. First, we analyzed the occurrence of Clean, Smelly, Buggy, and Harmful code in open-source projects as well as which smell types are more related to Harmful Code. Further, we investigated in which extent developers prioritizes refactoring Harmful Code. We also evaluated the effectiveness of machine learning techniques to detect Harmful and Smelly code. Finally, we investigated which metrics are most important in Harmful Code detection.

To perform our study, we defined an experiment with 22 smell types, 803 versions of 13 open-source projects, 40,340 bugs mined from GitHub issues and 132,219 code smells. The results show that even though we have a high number of code smells, only 0.07% of those smells are harmful. The *Abstract Function Call From Constructor* is the smell type more related to Harmful Code. Also, we performed a survey with 60 developers investigated in which extent developers prioritizes refactoring Harmful Code. The mayority (53.8%) of the developers prioritize code associated with bugs, and most of them (30%) prioritizes Harmful Code when refactoring. Also most of them (98%) consider code smells harmful to the software. Regarding the effectiveness of machine learning techniques to detect Harmful Code, our results indicate that they reach effectiveness at least 97%. While the **Random Forest** [18] is effective in detecting both Smelly and Harmful Code, the **Gaussian Naive Bayes** [17] is the less effective technique. Finally, our results suggest software metrics (such as *CBO (Couple Between Objects)* and *WMC (Weight Method Class)*) are important to customize machine learning techniques to detect Harmful Code.

As future work, we intend to extend this investigation by using the theory of Social Representations to further expand the understanding of Harmful Code from the perspective of the developers.

7 ACKNOWLEDGMENTS

This work is supported by CAPES (88887.496429/2020-00), FACEPE (APQ-0570-1.03/14), and CNPq (409335/2016-9, 427787/2018-1), as well as INES 2.0,[8] FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0. The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *CSMR* (2011).

[2] Mamdouh Alenezi and Mohammad Zarour. 2018. An Empirical Study of Bad Smells during Software Evolution Using Designite Tool. *i-Manager's Journal on Software Engineering* 12, 4 (Apr 2018), 12–27.

[3] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. 2016. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. *ISSRE 2015* (2016).

[4] Filipe Falc ao Batista Dos Santos, Caio Barbosa, Baldoino Fonseca dos Santos Neto, Alessandro Garcia, Márcio Ribeiro, and Rohit Gheyi. 2020. On Relating Technical, Social Factors, and the Introduction of Bugs. *SANER 2020*.

[5] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (2016).

[6] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* (2017).

[7] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *IST* 108, 4 (2019), 115–138.

[8] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (Feb. 2012), 281–305.

[9] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, I–115–I–123.

[10] Gemma Catolino, Fabio Palomba, Francesca Arcelli, Fontana Andrea, Andy Zaidman, and Filomena Ferrucci. 2019. Improving change prediction models with code smell-related information. (2019).

[11] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.

[12] Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. 2015. On Experimenting Refactoring Tools to Remove Code Smells. In *XP2015* (Helsinki, Finland). ACM, New York, NY, USA, Article 7, 8 pages.

[13] Martin Fowler. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley.

[14] Yann-Gael Gueheneuc, Houari Sahraoui, and Farouk Zaidi. 2004. Fingerprinting Design Patterns *(WCRE '04)*. IEEE Computer Society, USA, 172–181.

[15] Aurélien Géron. 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly Media, Inc.

[16] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some code smells have a significant but small effect on faults. *TOSEM* 23, 4 (2014).

[17] David J. Hand and Keming Yu. 2001. Idiot's Bayes: Not So Stupid after All? *International Statistical Review / Revue Internationale de Statistique* 69, 3 (2001).

[18] Tin Kam Ho. 1995. Random Decision Forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1 (ICDAR '95)*. IEEE Computer Society, Washington, DC, USA, 278–.

[19] Mario Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. 2017. Smells Are Sensitive to Developers! on the Efficiency of (Un)Guided Customized Detection. *ICPC* (2017).

[20] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. 2018. Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology* 93 (2018).

[21] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. 2011. Search-based Design Defects Detection by Example. In *ETAPS*.

[22] Foutse Khomh, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (2012).

[23] Foutse Khomh, Stéephane Vaucher, Yann Gaël Guéehéeneuc, and Houari Sahraoui. 2009. A bayesian approach for the detection of code and design smells. *QSIC* (2009).

[24] Foutse Khomh, Stephane Vaucher, Yann Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[25] Brent Komer, James Bergstra, and Chris Eliasmith. 2014. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. 32–37.

[26] Michele Lanza and Radu Marinescu. 2010. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (1st ed.). Springer Publishing Company, Incorporated.

[27] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS* (Long Beach, California, USA) *(NIPS'17)*. 10.

[28] Wanwangying Ma, Huihui Liu, Yibiao Yang, Bixin Li, and Ru Jia. 2018. Exploring the Impact of Code Smells on Fine-Grained Change-Proneness. *IJSEKE* (2018).

[29] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt Von Staa. 2012. On the relevance of code anomalies for identifying architecture degradation symptoms. *CSMR* (2012), 277–286.

[30] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur. 2012. Support vector machines for anti-pattern detection. In *ASE*.

[31] Matthew James Munro. 2005. Product metrics for automatic identification of "bad smell" design problems in Java source-code. *Proceedings - International Software Metrics Symposium* 2005, Metrics (2005), 125–133.

[32] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? A study on developers' perception of bad code smells. *ICSME 2014* (2014), 101–110.

[33] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *EMSE* (2018).

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *JMLR* (2011).

[35] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. 2017. How do developers select and prioritize code smells? A preliminary study. *ICSME 2017* (2017).

[36] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. *ESEC/FSE 2018* (2018), 908–911.

[8] http://www.ines.org.br