Developer-Driven Code Smell Prioritization

Fabiano Pecorelli
SeSa Lab - University of Salerno
Fisciano (Salerno), Italy
fpecorelli@unisa.it

Fabio Palomba
SeSa Lab - University of Salerno
Fisciano (Salerno), Italy
fpalomba@unisa.it

Foutse Khomh
École Polytechnique de Montréal
Montréal (Québec), Canada
foutse.khomh@polymtl.ca

Andrea De Lucia
SeSa Lab - University of Salerno
Fisciano (Salerno), Italy
adelucia@unisa.it

ABSTRACT

Code smells are symptoms of poor implementation choices applied during software evolution. While previous research has devoted effort in the definition of automated solutions to detect them, still little is known on how to support developers when prioritizing them. Some works attempted to deliver solutions that can rank smell instances based on their severity, computed on the basis of software metrics. However, this may not be enough since it has been shown that the recommendations provided by current approaches do not take the developer's perception of design issues into account. In this paper, we perform a first step toward the concept of *developer-driven* code smell prioritization and propose an approach based on machine learning able to rank code smells according to the perceived criticality that developers assign to them. We evaluate our technique in an empirical study to investigate its accuracy and the features that are more relevant for classifying the developer's perception. Finally, we compare our approach with a state-of-the-art technique. Key findings show that the our solution has an F-Measure up to 85% and outperforms the baseline approach.

KEYWORDS

Code smells; Machine Learning for Software Engineering; Empirical Software Engineering.

ACM Reference Format:
Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. 2018. Developer-Driven Code Smell Prioritization. In *MSR 2020: IEEE/ACM International Conference on Mining Software Repositories, May 25–26, 2020 - Seoul, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As a software system evolves, continuous modifications are required to adapt it to new requirements and/or changing environments or even fix defects that can preclude its correct functioning [70]. These activities are usually performed between strict deadlines and constraints to meet [10]. As a side effect, developers risk to introduce *technical debts* [10, 78], *i.e.*, sub-optimal implementation decisions that provide short-term benefits but cause a decrease of software quality. One of the main indicators of technical debt is represented by the presence of *code smells* [27], which are poor design/implementation choices applied by developers during maintenance and evolution, *e.g.*, the introduction of complex and/or long classes, excessive coupling between objects, and so on.

Code smells have been often associated to a decrease of program comprehensibility [2], maintainability [39, 60], testability [29] as well as an increase of maintenance effort and costs [80]. These findings have motivated researchers to propose automated mechanisms to support developers in both the identification [5, 19, 22] and removal [52] of code smells, obtaining promising results.

Despite this effort, the adoption of code smell detectors in practice is still limited [58]. Among the others, two notable reasons precluding the applicability of automated detectors in practice may be: (1) the large amount of code smell instances detected by these tools combined to the little knowledge on how to prioritize their refactoring [26, 50]; (2) the empirical evidence showing how most of the available detectors identify code smells that developers do not perceive or do not consider critical [24, 66].

While some researchers provided initial attempts toward the prioritization of code smells using measures of severity derived from software metrics [4, 26, 50, 93], the available solutions either rely on predefined heuristics that have not been empirically assessed or do not address the problem of providing developers with recommendations aligning with their perception of design issues, thus possibly being still ineffective in practice.

In this paper, we build on this line of research and propose the first step toward the concept of *developer-driven code smell prioritization* as an alternative and more pragmatic solution to the problem: Rather than ranking code smell instances based on their severity computed using software metrics, we propose to prioritize them according to the criticality perceived by developers.

In particular, we first perform surveys to collect a dataset composed of developers' perception of the severity of 1,332 code smell instances—pertaining to four different types of design problems—for which original developers rated their actual criticality and then propose a novel supervised approach that learns from such labeled data to rank unseen code smell instances. Afterwards, we conduct an empirical study to (1) verify the performance of our prioritization

MSR 2020, May 25–26, 2020, Seoul, South Korea

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

approach, (2) understand what are the features that contribute most to model the developer's perceived criticality of code smells, and (3) compare our approach with the state-of-the-art baseline proposed by Arcelli Fontana and Zanoni [26]. The key differences between our approach and the baseline considered for the comparison are (i) the usage of different kinds of predictors (*e.g.*, process metrics) rather than considering only structural ones, and (ii) the definition of a dependent variable based on the developers' perception.

The main result of the study highlights that the devised approach can classify the developer's perceived criticality of code smells with an F-Measure ranging between 72% and 85%. Moreover, we discovered that, depending on the code smell type, specific features are more relevant to classify its criticality. Finally, our approach performs better than the experimented baseline when classifying all considered code smells.

To sum up, this paper provides the following contributions:

(1) A new dataset reporting the criticality perceived by original developers with respect to 1,332 code smell instances of four different types, which can be further used by the community to build upon our research;

(2) The first machine learning-based approach to prioritize code smells according to the real developers' perceived criticality;

(3) An empirical study that showcases the performance of our approach, the importance of the employed features, and the reasons why our technique goes beyond the state-of-the-art;

(4) An online appendix with the datasets used in the study, that can be exploited to replicate and extend our work.

Structure of the paper. In Section 2 we summarized the related literature. Section 3 describes the methods employed to construct the dataset, while Section 4 reports on the definition of our approach and its empirical evaluation. In Section 5 we discuss the results of the study, while Section 6 overviews the threats to the validity of the study and how we mitigated them. Finally, Section 7 concludes the paper and outlines our future research agenda.

2 RELATED WORK

The research community actively studied code smells in the past [5, 19, 22]. On the one hand, researchers focused on understanding the characteristics of code smells and, in particular, their origin [48, 61, 91], evolution [57], diffuseness [60], relevance for developers [62, 66, 85, 94, 95], impact on software comprehensibility [2, 67] and maintainability [7, 39, 80]. On the other hand, a number of automatic code smell detection approaches and tools have been developed and validated [21, 25, 38, 40, 53, 63, 65, 69, 72, 89].

When it comes to code smell prioritization, however, the research contribution so far is notably less prominent and much more focused on the idea of ranking refactoring recommendations. For instance, Tsantalis and Chatzigeorgiou [90] proposed to order the refactoring suggestions given by JDeodorant using historical information and, particularly, modeling the probability that a certain refactorable class will change in the near future. Along these lines, Girba *et al.* [28] employed the number of changes as metric to decide on whether a class should be refactored in the current release of a software system. With respect to these papers, there are two main aspects that make our paper different. First, there may be refactoring opportunities that do not target classes affected by code smells, *i.e.*, many refactoring actions are driven by other considerations, *e.g.*, the introduction of design patterns [7, 79]. As such, the goal of our paper is diametrically different, since we aimed at ranking code smells based on their perceived criticality. In the second place, the papers above are based on the underlying concept that the recent history of class is the main driver for its refactorability, while we showed that multiple metrics may possibly contribute to it.

Other researchers took a closer look to the problem of prioritizing code smells. Marinescu [50] defined a method to rank smell instances based on their severity that can be applied on top of heuristic-based code smell detectors: given a set of metrics that characterize a class and given the corresponding thresholds that discriminate high/low values for these metrics, the method computes the average distance between the actual code metric values and the fixed thresholds. That is, smell instances having higher average distance from the thresholds are ranked at the top. Later on, Arcoverde *et al.* [4] proposed heuristics to rank code anomalies based on their impact on the overall architecture of the system, computed on the basis of number of changes and faults. These two works are essentially based on predefined heuristics, hence implementing a different approach with respect to the one proposed in our paper. Also, these approaches have not been tested in practice and, for this reason, little is known about their actual capabilities. Nevertheless, in our study we considered these papers when defining the set of independent variables of our model: we considered both number of changes and faults coming from Arcoverde *et al.* [4] as well as the severity computed as suggested by Marinescu [50].

The two closest works come from Vidal *et al.* [93] and Arcelli Fontana and Zanoni [26]. The first approach takes into account three main factors to prioritize code smells, *i.e.*, stability, relevance, and modifiability scenarios; however, it is semi-automated and requires the input of experts to be actually used. On the contrary, with this paper we aimed at providing a fully automated solution. As for Arcelli Fontana and Zanoni [26], the authors defined a machine learner based on a large variety of size, cohesion, coupling, and complexity metrics, to predict the severity of the presence of a certain code smell in a class/method (from *low* to *high*). In their empirical study, the approach reached up to 95% in terms of F-Measure. Our work built on top of the work by Arcelli Fontana and Zanoni [26] and proposed a further step ahead: first, we aimed at classifying the perceived criticality of code smells in order to define methods that are closer to what developers consider real design issues; second, we do that by exploiting metrics of different nature and able to characterize the perceived quality of classes under different perspectives.

3 DATASET CONSTRUCTION

To perform our empirical study, we needed to collect a dataset reporting the perceived criticality of a set of code smells large enough to train a machine learning model. To this aim, we first defined the *objects* of the study, namely (1) a set of software projects and (2) the code smell types we were interested in with their corresponding detectors; Then, we inquired the *subjects* of our study, namely the original developers of the considered projects, in order to collect

Table 1: Software Projects in Our Dataset.

Project	#Commits	#Devs	#Classes	KLOCs
Apache Mahout	3,054	55	813	204
Apache Cassandra	2,026	128	586	111
Apache Lucene	3,784	62	5,506	142
Apache Cayenne	3,472	21	2,854	542
Apache Pig	2,432	24	826	372
Apache Jackrabbit	2,924	22	872	527
Apache Jena	1,489	38	663	231
Eclipse CDT	5,961	31	1,415	249
Eclipse CFX	2,276	21	655	106
Overall	**32,889**	**436**	**5,506**	**542**

their perceived criticality of the code smell instances detected on their codebase. The next subsections describe the various steps we followed to build our dataset.

3.1 Selecting projects

The *context* of the study consisted of nine open-source projects belonging to two major ecosystems such as Apache[1] and Eclipse.[2] Basic information and statistics about the selected projects are summarized in Table 1. Specifically, for each considered project, we report (i) the total number of commits available in its change history, (ii) the total number of contributors, and (iii) the size as the number of classes and KLOCs. The selection of these projects was driven by a number of factors. In the first place, we only focused on open-source projects since we needed to access source code to detect the considered design flaws. Similarly, we limited ourselves to Java systems as most of the smells, as well as code smell detectors, have been only defined for this programming language [5, 19, 64]. Furthermore, we aimed at analyzing projects having different (a) codebase size, (b) domain, (c) longevity, (d) activity, and (e) population. As such, starting from the set of 2,576 open-source systems written in Java and belonging to the two considered ecosystems available at the time of the analysis on Github,[3] we only took into account those having a number of classes higher than 500, with a change history at least 5 years long, having at least 1,000 commits, and with a number of contributors higher than 20. This filter gave us a total of 682 systems: of these, we randomly selected 9 of them.

3.2 Selecting code smells

We focused on four class-level types of code smells, namely:

Blob (or God Class). This code smell type affects classes that do not follow the single-responsibility principle [88], *i.e.*, they implement more than one responsibility, thus being poorly cohesive and hard to understand/maintain [27]. Previous studies demonstrated that classes affected by this smell are connected to higher change- and defect-proneness [39, 60] as well as maintenance and evolution costs [2, 80, 81]. According to recent findings [62, 85], this smell is among the most critical ones for practitioners.

Complex Class. Instances of this smell affect classes that have high cyclomatic complexity [11] and that, therefore, may primarily make the testing of those classes harder [29, 51], but also lower the ability of developers to evolve them [60]. Existing empirical evidence on the developer's perception about this smell indicated that practitioners are generally able to recognize this smell and assess its criticality [62, 95].

Spaghetti Code. This smell indicates the presence of classes that do not present a well-defined structure, and that usually declare a number of long methods [27]. Similarly to the other considered code smells, also *Spaghetti Code* has been widely investigated in the past by researchers, who discovered that it hinders the ability of developers to comprehend source code [2], increases maintenance effort [39, 80], and can be accurately assessed by developers with respect to its criticality [62, 85].

Shotgun Surgery. When a change to a class (*e.g.*, to one of its fields/methods) triggers many little changes to other classes of the system, such class is affected by this smell [27]. Instances of this smell are associated with a higher defect-proneness of the involved class [18]. Previous studies on the perception of this smell revealed that practitioners perceive its presence depending on the intensity of the flaw [62], *i.e.*, depending on the number of changes triggered on other classes of the project.

Three specific factors drove the selection of these four code smell types. First, they have been shown to be highly diffused in real software systems [60], thus allowing us to target code smells that are relevant in practice. Second, they are reported to negatively impact maintainability, comprehensibility, and/or testability of software systems [29, 39, 60]: as such, we could investigate design flaws that practitioners may be more able to analyze and assess. Finally, previous findings [62, 85, 95] showed not only that they are actual problems from the developer's perspective, but also that their criticality can be accurately assessed by practitioners, thus mitigating potential problems due to the presence of the so-called conceptual false positives [24], *i.e.*, code smell instances detected as such by automated tools but not representing issues for developers.

3.3 Selecting code smell detectors

Once we had selected the specific code smells object of our investigation, we then proceeded with the choice of automated code smell detectors that could identify them. Among all the available solutions proposed so far by researchers [22], we opted for Decor [53] and Hist [63]. The first was selected to identify instances of *Blob*, *Complex Class*, and *Spaghetti Code*, while the latter for the detection of *Shotgun Surgery*.

More specifically, Decor is an automated solution which adopts a set of "rule cards",[4] namely rules able to describe the intrinsic characteristics that a class must have to be affected by a certain code smell type. In the case of *Blob*, the approach identifies it when a class has a Lack of Cohesion of Method (LCOM5) [35] higher than α, a total number of methods and attributes higher than β, it is associated to many data classes (*i.e.*, classes having just `get` and `set` methods), and has a name having a suffix in the set {*Process,*

[1] https://www.apache.org
[2] https://www.eclipse.org/org/
[3] https://github.com
[4] http://www.ptidej.net/research/designsmells/

Control, Command, Manage, Drive, System}, where α and β are relative threshold values. When detecting *Complex Class* instances, Decor computes the Weighted Methods per Class metric (WMC), *i.e.*, the sum of the cyclomatic complexity of all methods of the class [51], and marks a class as smelly if the WMC is higher than a defined threshold. Finally, *Spaghetti Code* instances are represented by classes presenting (i) at least one method without parameters and having a number of lines of code higher than a defined threshold, (ii) no inheritance, as indicated by the Depth of Inheritance Tree metric (DIT) [16] which must be equal to 1, and (iii) a name suggesting procedural programming, thus having as prefix/suffix a word in the set {*Make, Create, Exec*}. There are two key reasons leading us to rely on Decor for the detection of these three smells. In the first place, this detector has been employed in several previous studies on code smells [37, 40, 47, 66, 67], showing good results when considering both precision and recall. At the same time, it implements a lightweight mechanism with respect to other existing approaches (*e.g.*, textual-based techniques relying on information retrieval [65, 68]): the scalability of Decor allows us to perform an efficient detection on the large systems considered in the study.

Turning our attention to the detection of *Shotgun Surgery*, the discussion is different. Approaches based on source code analysis have been shown to be poorly effective for the detection of this smell [63]; for instance, the approach proposed by Rao and Reddy [75]—which computes coupling metrics to build a change probability matrix that is then filtered to detect the smell—was not able to identify any *Shotgun Surgery* instance when applied in large-scale studies [63]. For this reason, we relied on Hist [63], a historical-based technique that (1) computes association rules [3] to identify methods of different classes that often change together and (2) identifies instances of the smell if a class contains at least one method that frequently changes with methods present in more than 3 different classes. Such a historical-based approach has shown an F-Measure close to 92% [63], thus representing a suitable solution for conducting our study.

On a technical note, we relied on the original implementations of Decor and Hist, hence avoiding potential threats to construct validity due to re-implementations.

3.4 Collecting the criticality of code smells

The last step required to build our dataset was the actual detection of code smells in the considered systems and the subsequent inquiry on their criticality. We followed a similar strategy as Silva *et al.* [79]: in short, in a time period of 6 months, from January 1st to June 30th, 2018, we monitored the activities performed on the selected repositories and, as soon as a developer committed changes to classes affected by any of the considered smells, we sent an e-mail to that developer to ask (i) whether s/he actually perceived/recognized the presence of a code smell and (ii) if so, rate its criticality using a Likert scale from 1 (very low) to 5 (very high) [44].

In particular, we built an automated mechanism that fetched—using the `git-fetch` command—commits from the repositories to a local copy on a daily basis. This gave us the possibility to generate the list of classes modified during the workday. At this point, we performed the actual smell detection. In the case of Decor, we parsed each modified class and run the detection rules described in Section 3.3 to identify instances of *Blob*, *Complex Class*, and *Spaghetti Code*. As for Hist, it requires information about the change history of the involved classes: for this reason, before running the *Shotgun Surgery* detection algorithm, we mined the log file of the projects to retrieve the set of changes applied on the classes modified during the workday. Through the procedure described so far, we obtained a list of smelly classes, and, for each of them, we stored the e-mail address of the developer who committed changes on it. Afterwards, we manually double-checked the smelly classes given by the automated tools with the aim of discarding possible false positives, thus avoiding asking developers useless information. Overall, the code smell detection phase resulted in a total of 2,675 candidate code smells. Of these, we discarded 455 (≈17%) false positives.

Finally, we sent e-mails to the original developers. In the text, we first presented ourselves and then explained that our analysis tool suggested that the developer likely worked on a class affected by a design issue—without revealing the exact code smell to avoid confirmation bias [56]. Then we asked three specific questions:

(1) *Were you aware of the presence of a design flaw?*
(2) *If yes to question (1), may you please briefly describe the type of design flaw affecting the class?*
(3) *If yes to question (1), may you please rate the criticality of the design flaw from (1) (very low) to (5) (very high)?*

We asked the first question to make sure that the contacted developers perceived classes as affected by code smells. If not, they could not obviously provide meaningful information on their criticality, and the answers were discarded. Otherwise, we further asked to explain the design problem perceived, so that we could understand if developers were actually aware of the specific smell. When receiving the answers, we checked if the explanation given by developers was in line with the definition of the smell: for instance, one developer was contacted to rate the criticality of a *Blob* class and explained that *"[the class] is a well-known problem, it is huge in size and has high coupling"*, thus indicating that s/he correctly recognized the smell we were proposing to him/her. In these cases, we considered the answer to the third question valid, otherwise we discarded it. Note that if a code smell was detected in the same class more than once, we did not send any e-mail to not bother the developers multiple times for the same class.

As an outcome, we sent a total of 1,733 e-mails to 372 distinct developers, *i.e.*, an average of 0.77 e-mails per month per developer, while 487 code smells affected the same classes multiple times and, therefore, we avoided sending e-mails for them. Moreover, we could not assess the criticality of 310 code smells because the 139 developers responsible for them did not reply to our e-mails. Also, we had to discard 91 answers received since the contacted developers did not perceive the presence of code smells, *i.e.*, they answered *'no'* to question (1).

Hence, we finally gathered 1,332 valid answers coming from 233 developers: the high response rate (62%) is in line with previous works that implemented a similar recruitment strategy [67, 79] and indicates that contacting developers immediately after their activities with short surveys not only increases the chances of receiving accurate answers [79], but also helps increasing their overall responsiveness. As a final note, the 1,332 code smell instances evaluated were almost equally distributed among the four considered types of design flaw: indeed, we had answers for 341 *Blob*, 349 *Complex*

Class, 313 *Spaghetti Code*, and 329 *Shotgun Surgery* instances. Also, the criticality values assigned by developers to each smell type were almost uniformly distributed over the possible ratings (① to ⑤)—more details are available in our online appendix [1].

4 A NOVEL CODE SMELLS PRIORITIZATION APPROACH AND ITS EVALUATION

The *goal* of our study is to define and assess the feasibility of using a machine learning-based solution to prioritize code smells according to their perceived criticality, with the *purpose* of providing developers with recommendations that are more aligned to the way they actually refactor source code. The *perspective* is of both practitioners and researchers: the former are interested in adopting more practical solutions to prioritize refactoring activities, while the latter are interested in assessing how well machine learning can be employed to model developer's criticality of code smells.

4.1 Research Questions

The empirical study revolves around three main research questions (**RQ**s). We started with the definition of a machine learning-based approach to model the developer's perceived criticality of code smells. Starting from the dataset built following the strategy reported in Section 3, we defined dependent and independent variables of the model as well as the appropriate machine learning algorithms to deal with the problem. These steps led to the definition of our first research question:

> **RQ$_1$.** *Can we predict developer's perception of the criticality of a code smell?*

Besides assessing the model as a whole, we then took a closer look at the contributions given by the independent variables exploited, namely what are the features that help the most when classifying the perceived criticality of code smells. This step allowed us to verify some of the conjectures made when defining the set of independent variables to be used. Hence, we asked:

> **RQ$_2$.** *What are the features of the proposed approach that contribute most to its predictive performance?*

As a final step of our investigation, we considered the literature in the field to identify existing code smell prioritization approaches that can be used as baselines, thus allowing us to assess how useful our technique can be when compared to existing approaches. This led to our final research question:

> **RQ$_3$.** *How does our approach perform when compared with existing code smell prioritization techniques?*

In the next sections, we describe the methodological details of the evaluation of the proposed approach.

4.2 RQ$_1$. Defining and assessing the performance of the prioritization approach

To address **RQ**$_1$, we defined a novel code smell prioritization approach that aims at classifying smell instances based on the developer's perceived criticality using machine learning algorithms. This implied the definition of an appropriate set of independent variables able to predict the dependent variable, *i.e.*, the developer's perception, as well as the proper algorithms and their configuration.

Dependent Variable. As a first step, we defined the developer's perceived criticality of code smells as a variable that the model has to estimate. The dataset described in Section 3 reports, for each code smell instance, a value ranging from 1 to 5 describing the perceived criticality of that instance. Thus, we mapped the problem as a classification one [17], namely we took into account the case in which the learner has to classify the criticality of code smells in multiple categorical classes [17]. In this case, we converted the integers of our dataset in nominal values in the set {*low*, *medium*, *high*}: if a code smell instance was associated to 1 or 2 in the original dataset, then we considered its perceived criticality as *low*; if it was equal to 3, we converted its value in *medium*; otherwise, we considered its criticality as *high*. With this mapping, we merged the values assigned by developers in order to build three main classes. This was a conscious decision given by experimental tests: indeed, when experimenting with a 5-point classification problem, we observed that several misclassifications were due to the approach not able to correctly distinguish (i) *very-low* from *low* and (ii) *high* from *very-high*. Thus, we opted for a 3-point classification.

Independent Variables. To predict the perceived criticality of code smells, we considered a set of features able to capture the characteristics of classes under different angles. Table 2 summarizes the families of metrics considered, the rationale behind their use, and the specific indicators measured.

Previous research has not only shown that product metrics can indicate actual design problems in source code [14, 54], but also lead developers to recognize the presence of sub-optimal implementation solutions that would deserve some refactoring [62, 84, 85]. For these reasons, we considered four types of product metrics, such as (i) size, (ii) cohesion, (iii) coupling, and (iv) complexity metrics. For each of these types, we selected indicators having different nature (*e.g.*, structural aspects of source code rather than textual components) and able to capture in different ways the considered phenomena (*e.g.*, we computed source code complexity using both the McCabe metric and readability, which targets a more cognitive dimension of complexity).

While product metrics can provide indications about the structure of source code, we complemented them with orthogonal metrics that capture the way the code has been modified as well as who was responsible for that, *i.e.*, process and developer-related metrics. Indeed, the developer's perception of criticality may be not always due to the complex structure of source code, but rather to the problems it causes during the evolution process [66, 74]; similarly, the criticality of code smells may be perceived differently depending on whether the maintainer is an expert of the class or not [9, 15]. Hence, we selected a number of metrics related to those aspects: for instance, we computed the average number of co-changing classes for the smelly class (AVG_CS) to assess whether smells that often change with several classes are perceived as more critical by developers or the number of previous bug fixing involving the smelly class to observe if classes that are more fault-prone are actually those perceived more critical. We

Table 2: Software metrics used as independent variables split by categories - The motivations for their use are also reported.

Metric	Acronym	Description
Product metrics. Cohesion, coupling, and complexity may lower code quality and affect the perceived criticality of code smells [62, 84, 85].		
Lines of Code	LOC	Amount of lines of code of a class excluding white spaces and comments.
Lack of Cohesion of Methods [16]	LCOM5	Number of method pairs in a class having no common attribute references.
Conceptual Cohesion of Classes [49]	C3	Average cosine similarity [6] computed among all method pairs of a class.
Coupling between Object Classes [16]	CBO	Number of classes in the system that call methods or access attributes of a class.
Message Passing Coupling [16]	MPC	Number of method calls made by a class to external classes of the system.
Response for a Class [16]	RFC	Sum of the methods of a class, *i.e.*, number of methods that can potentially be executed in response to a message received by an object of a class.
Weighed Methods per Class [16]	WMC	Sum of the McCabe cyclomatic complexity [51] computed on all methods of a class.
Readability [12]	Read.	Measure of source code readability based on 25 features, *e.g.*, number of parentheses per lines of code.
Process metrics. The amount of activities made on smelly classes may affect the developer's perceived criticality [43, 46, 74].		
Average Commit Size	AVG_CS	Average number of classes that co-changed in commits involving a class.
Number of Changes	NC	Number of commits in the change history of the system involving a class.
Number of Bug Fixes	NF	Number of bug fixing activities performed on a class in the change history of the system.
Number of Committers	NCOM	Number of distinct developers who performed commits on a class in the change history of the system.
Developer-related metrics. Experience and workload of developers may affect the perceived criticality of code smells [9, 13, 15, 91].		
Developer's Experience [9]	EXP	Average number of commits of the committers of a class.
Developer's Scattering Changes [20]	DSC	Average number of distinct subsystems in which the committers of a class made changes.
Development Change Entropy [34]	CE	Shannon's entropy [77] computed on the number of changes of a class in the change history of the system.
Code Ownership [9]	OWN	Number of commits of the major contributor of a class over the total number of commits for that class.
Code smell-related metrics. Persistence of code smells and availability of refactoring opportunities may affect the developer's perception [59, 69].		
Persistence	Pers.	Number of subsequent major/minor releases in which a certain smell affects a class.
Intensity [50]	Sev.	Average distance between the actual metric values used for the detection of code smells and the corresponding thresholds considered by the detectors to distinguish smelly and non-smelly elements.
Refactorable	Ref.	Existence of refactoring opportunities for a class, as detected by automated tools.
Number of Refactoring Actions	NR	Number of previous refactoring operations made by developers on a class.

also computed measures of experience and ownership of developers working on the smelly class to test whether these factors influence the developer's ability to work on it.

Finally, we took into account some specific metrics related to code smells: the idea here is that a number of aspects connected to the smell itself may be relevant for developers when assessing its criticality. In particular, the continuous presence of smell over the history of the project (*i.e.*, Pers.) may influence the ability of developers to recognize its harmfulness better. Much in the same way, the presence of refactoring opportunities (Ref.) or even the number of previous refactoring actions done on the smelly class (NR) may affect the perception of developers. Finally, we also considered the code smell intensity, which is a measurable amount of intensity of a certain code smell instance [50]: we included this metric to understand whether there is a match between an "objective" measurement of code smell severity and its real perceived criticality.

From a technical perspective, we employed the tool by Spinellis [83] to compute product metrics, the one made available by Buse and Weimer [12] for the computation of the readability index, and PyDriller [82] to compute process and developer-related metrics. As for the smell-related indicators, we developed our own tool to compute Pers. and NR. Starting from the release R_i of the projects taken into account, the former metric counts in how many consecutive previous major and minor releases—identified using the corresponding Git tags—the considered smell was present, according to the employed detectors. The latter metric, instead, was computed by mining the messages of commits involving the smelly classes and looking for the presence of keywords recommended in [92], *e.g.*, *'refactor'* or *'restructure'*. The intensity of code smells has been assessed using the tool by Marinescu [50], which computes the average distance between the actual code metric values of the smell instance and the thresholds fixed by the detection rules. Finally, the Ref. metric was computed by (i) running JDeodorant [23], an existing refactoring recommender that covers all refactoring actions associated to the considered code smells, and (ii) putting the metric to 1 if the tool retrieved at least one recommendation, 0 otherwise.

Machine Learning Algorithms. Once we had computed dependent and independent variables, we proceeded with the definition of the machine learning algorithms to be used.

In order to perform the classification, we investigated the use of multiple algorithms [17], *i.e.*, Random Forest, Logistic Regression, Vector Space Machine, Naive-Bayes, and Multilayer Perceptron, in order to assess what is the one giving the best performance. Note that, before running the algorithms, we first applied a forward selection of independent variables using the Correlation-based feature selection (CFS) approach [45], which uses correlation measures and a heuristic search strategy to identify a subset of actually relevant features for a model, thus mitigating possible problems due to multi-collinearity of features [55]. Then, we configured classifiers' hyper-parameters by exploiting the Grid Search [8] algorithm, that runs an exhaustive search of the hyper-parameter space.

Training/Testing the Model. We built different models for each code smell considered in the study, so the training data is represented by the set of observations available for a certain smell in the collected dataset—the distribution of the criticality values assigned by developers to the code smell instances evaluated is available in our appendix [1]: the distribution is almost uniform for all the criticality values. This aspect affected our decision to not apply any balancing algorithm. On the one hand, there are no classes requiring to be balanced with respect to the others. On the other hand, previous findings [71, 72] have shown that balancing

code smell-related datasets can even damage the performance of the resulting models.

To train the model, we employed a 10-fold cross-validation strategy [41]: it randomly partitions the available dataset into 10 folds of equal size, applying a stratified sampling—meaning that each fold has the same proportion of the various criticality classes. A single fold is then used as test set, while the remaining ones are employed for training the model. The process is repeated ten times so that each fold will be the test set exactly once.

Performance Assessment. We evaluated the performance of the experimented model by analyzing confusion matrices, obtained from the testing strategy described above, reporting the number of true and false positives as well as the number of true and false negatives. We analyzed these matrices by first computing precision, recall, and F-Measure [6]. Then, we computed the Matthew's Correlation Coefficient (MCC), a correlation index between the observed and predicted binary classifications, and the Area Under the ROC Curve (AUC-ROC), which quantifies the ability of the model to discriminate between the different classes.

4.3 RQ_2. Explaining the Proposed Approach

In the context of $\mathbf{RQ}_2$, we took a deeper look into the performance of the best model coming from the previous research question. We aimed at understanding the value of the individual metrics selected as independent variables; this step could possibly help us explaining why the proposed approach works (or not) when predicting the criticality of code smells. To this aim, we employed an *information gain* algorithm [73], a category of methods that can be used to quantify the gain provided by each feature to the correct prediction of a dependent variable. From a formal point of view, let M be a supervised learning model, let $F = \{f_1, \ldots, f_n\}$ be the set of features composing M, *i.e.*, the metrics reported in Table 2, an information gain algorithm measures the difference in entropy from before to after the set M is split on a feature f_i. Roughly speaking, the algorithm gives a measure of how much uncertainty in the model M is reduced after splitting M on feature f_i. In our work, we implement this analysis using the *Gain Ratio Feature Evaluation* algorithm [73], which is directly made available by the Weka toolkit [32]. It ranks the features $f_1, \ldots, f_n$ in descending order based on the entropy reduction provided by f_i to the decisions made by M, *i.e.*, it gives as output a ranked list where the more relevant features are reported at the top. We ran the algorithm considering each code smell independently.

To analyze the resulting rank and have statistically significant conclusions, we finally exploited the Scott-Knott Effect Size Difference (ESD) test [87]. This is an effect-size aware variation to the original Scott-Knott test [76] that has been recommended for software engineering research in previous studies [36, 42, 86] as it (i) uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups according to their influence, (ii) corrects the non-normal distribution of an input dataset, and (iii) merges any two statistically distinct groups that have a negligible effect size into one group to avoid the generation of trivial groups. As effect size measure, the test relies on Cliff's Delta (or d) [30]. To compute the test, we used the publicly available implementation[5] provided by Tantithamthavorn *et al.* [87].

4.4 RQ_3. Comparison with the state of the art

Finally, we investigated whether and to what extent the proposed code smell prioritization approach overcomes the performance of existing techniques. This step is paramount to understand the novelty of our solution and how it may support developers better than the baseline approaches.

The two closest techniques with respect to the one proposed herein are those by Vidal *et al.* [93] and Arcelli Fontana and Zanoni [26]: we set them as initial baselines for the comparison. In the former, the authors proposed SpIRIT, a semi-automated technique that relies on three main criteria, namely (1) *stability*, *i.e.*, number of previous changes applied on a smelly class over the number of total changes applied on the system, (2) *relevance*, *i.e.*, the relative importance of the class within the system according to the feedback given by a developer, and (3) *modifiability scenarios*, *i.e.*, the number of possible use cases of the application that risk to be impacted by the presence of the smell according to the opinion of an expert. The three criteria are then combined through a weighted average, where the weights are assigned by the user of the tool. As the reader might have noticed, SpIRIT explicitly requires the intervention of an expert to be employed in practice: indeed, the technique has been tested in an industrial case study involving a Java project affected by a total of 47 code smells and requiring the interaction of core developer of the subject application. For this reason, we could not use it as a baseline for a *in-vitro* assessment of our proposed approach and we plan to perform a comparison with SpIRIT in our future research agenda, as further explained in Section 7.

As for the technique proposed by Arcelli Fontana and Zanoni [26], this is a machine learning-based solution that relies on a total of 61 product metrics to predict how critical a certain code smell instance is. This technique can be fully automated and, therefore, we could use it as the baseline for our study. Given the absence of a publicly available implementation, we developed our own version of the technique and run it using the same dependent variable, training, validation strategy, and dataset employed to validate our approach. Once obtained the output from the baseline, we compared it with ours by means of the same set of metrics used in $\mathbf{RQ}_1$, *i.e.*, precision, recall, F-Measure, MCC, and AUC-ROC.

5 RESULTS AND DISCUSSION

This section reports the results of our study, presenting each research question independently.

5.1 RQ_1. The Performance of our Model

In the context of $\mathbf{RQ}_1$, we aimed at assessing how well can we predict the perceived criticality of code smells. Table 3 reports the confusion matrices obtained when running the proposed approach against our dataset of four code smell types, while Table 4 presents the weighted average performance for each code smell. For the sake of space limitations, we only report the results achieved with the best classifier, *i.e.*, Random Forest. A summary of the performance of the other classifiers is available in our replication package [1].

[5]Link: https://github.com/klainfo/ScottKnottESD

Table 3: RQ_1 - RQ_3. Confusion matrices obtained when running the proposed model against our dataset.

Model	Class/Smell	Blob			Complex Class			Spaghetti Code			Shotgun Surgery		
		Non-severe	Medium	Severe	Non-severe	Medium	Severe	Non-severe	Medium	Severe	Non-severe	Medium	Severe
Our approach	Non-severe	54	10	6	46	17	27	57	11	2	46	37	13
	Medium	6	171	18	14	176	0	5	151	6	8	134	10
	Severe	1	10	65	6	1	62	0	7	74	5	16	60
Baseline	Non-severe	16	40	14	45	43	2	37	19	21	11	83	6
	Medium	12	174	9	21	161	8	0	172	0	18	122	12
	Severe	8	23	45	7	48	14	13	0	79	6	79	4

Table 4: RQ_1 - RQ_3. Weighted Average of the performance achieved by the experimented models against our dataset.

Code smell	Model	Prec.	Rec.	F-Meas.	MCC	AUC-ROC
Blob	Our approach	86%	85%	85%	75%	89%
	Baseline	66%	69%	66%	44%	78%
Complex Class	Our approach	79%	81%	80%	71%	89%
	Baseline	62%	63%	63%	33%	76%
Spaghetti Code	Our approach	90%	88%	89%	83%	92%
	Baseline	83%	85%	84%	77%	89%
Shotgun Surgery	Our approach	74%	71%	72%	61%	78%
	Baseline	33%	40%	35%	32%	61%

In the first place, it is worth noting that the performance values of our model are rather high and, indeed, it has an F-Measure that ranges between 72% and 85%. This indicates that, in most of the cases, our model can accurately classify the severity perceived by developers. The worst case is represented by *Shotgun Surgery*, where the model has an F-Measure of 72% and an AUC-ROC of 61%. On the one hand, the former metric still indicates that the model is able to correctly classify most of the instances of our dataset. On the other hand, the latter suggests that the ability of separating criticality classes may be further improved; this is also visible when considering the confusion matrix for this smell (Table 3), where we noticed that in 52% of cases the model classified non-severe code smells as medium or severe cases. An example is represented by the class `security.JackrabbitAccessControlManager` of the Jackrabbit project. This class has 164 lines of code and has been detected as smelly because every time it is changed an average of other 11 classes are also modified. Nevertheless, it has been subject to a relatively low number of changes (19) and defects (1), likely being less harmful than other instances of the smell. Analyzing the other misclassified cases, we noticed a similar trend: the model tends to misclassify instances because it is not always able to learn how to balance the information coming from the number of classes to be modified with the smelly one and the actual number of changes that involve the smelly instance. As such, we can claim that possible improvements to the classification model may concern the addition of combined metrics, *e.g.*, the ratio between number of co-changing classes and number of previous changes of the smelly class.

As for the other code smells considered, the performance values are higher and all above 80% and 70% in terms of F-Measure and AUC-ROC. Hence, we can claim that the proposed model can be effectively adopted by developers to prioritize code smell instances. The best result is the one of *Spaghetti Code* (F-Measure=89%, AUC-ROC=92%): in this case, the model misclassifies only 31 cases (10% of the instances). By looking deeper at those cases, we could not find any evident property of the source code leading to those false positives. A similar discussion can be drawn when considering the *Blob* and *Complex Class* code smells. Part of our future research agenda includes the adoption of mechanisms able to better describe the functioning of the learners used for the classification, *e.g.*, explainable AI algorithms [31].

> ***Finding 1***. *The proposed model has an F-Measure ranging between 72% and 85%, hence being accurate in the classification of the perceived criticality of code smells. The worst case relates to Shotgun Surgery, where the model misclassifies non-severe instances because of its partial inability to take into account other process-related information like number of changes involving the smelly classes.*

5.2 RQ_2. Features Contributing to the Model

Table 5 reports the list of features contributing the most to the performance of the proposed model. As shown, each code smell has its own peculiarities. To classify *Blob* instances, the model mostly relies on structural metrics that capture complexity (RFC, WMC), cohesion (LCOM5, C3), and coupling (CBO) of the source code: basically, it means that the criticality of this code smell is given by a mix of various structural factors and cannot be described by just looking at them independently. At the same time, the number of previous defects affecting those instances (NF) as well as the workload of the committers (DSC) have a non-negligible effect. As such, on the one hand we can confirm previous findings that showed historical and socio-technical factors as relevant to manage code smells [63, 67]. On the other hand, our findings suggest that these metrics may possibly be useful for detecting code smells in the first place or even filtering the results of currently available detectors, so that they may give recommendations that are closer to the developer's perceived criticality. Finally, the lines of code also contributes to the model, being however not the strongest factor—confirming again previous findings in the field [63, 65].

When considering *Complex Class*, a similar discussion can be done. While the most impactful metrics concern with the structure of the code (CBO, WMC, LCOM5), other metrics seem to have a relevant effect on the classification model. In particular, readability is the strongest factor after code metrics, indicating that developers consider comprehensibility important when prioritizing this code smell. Other relevant factors are the number of previous changes of classes (NC) and socio-technical aspects like experience and workload of the committers (EXP, DSC): again, this result confirms that the management of code smells may require additional information than structural aspects of source code [91].

Table 5: RQ_2. Information Gain of the independent variables of our approach. For space limits, only metrics providing significant contributions are reported.

Code smell	Metric	Mean	SK-ESD
Blob	RFC	0.65	68
	LCOM5	0.57	66
	NF	0.56	66
	DSC	0.55	64
	CBO	0.45	64
	WMC	0.42	64
	C3	0.35	45
	LOC	0.34	41
Complex Class	CBO	0.59	71
	WMC	0.54	69
	LCOM5	0.54	69
	Read.	0.54	69
	NC	0.50	54
	DSC.	0.49	51
	EXP	0.27	33
	RFC.	0.25	31
Spaghetti Code	Read.	0.65	53
	NF	0.57	46
	C3	0.38	41
Shotgun Surgery	NC	0.32	44
	LCOM5	0.31	39
	AVG_CS	0.24	33
	Pers.	0.17	21

Surprisingly, when considering *Spaghetti Code* instances we noticed that no structural factors strongly influence the classification. Readability is indeed the key factor leading developers to prioritize instances of this smell, followed by the number of previous defects affecting those classes (NF) and by the conceptual cohesion of classes (C3). Hence, it seems that developers prioritize instances of this smell that are semantically incoherent or that suffered from defects in the past. Our findings could again be used by code smell detection and filtering approaches to tune the list of recommendations to provide to developers.

Finally, the prioritization of the *Shotgun Surgery* smell is mainly driven by process-related factors. Not only the number of changes (NC) is the most powerful metric, but also the number of co-changing classes (AVG_CS) turned out to be relevant. Also, this is the only case in which the persistence of the smell (Pers.) appeared to impact the classification. These result seem to confirm that developers assess the severity of this code smell based on the intensity of the problem [62, 85], *i.e.*, when number of changes or co-changing classes is high or when the problem is constantly affecting the codebase. Furthermore, the cohesion of the class (LCOM5) affects the classification, even though at a lower extent if compared to the contribution given for other code smell types.

> ***Finding 2.*** *The developer's perceived criticality of code smells represents a multi-faceted problem that can be tackled considering a mix of metrics having different nature (*e.g.*, structural or historical) and working at various levels of granularity (*e.g.*, process or socio-technical aspects).*

5.3 RQ_3. Comparison with the state of the art

We compared the proposed model with a baseline. The results are reported in Tables 3 and 4, where we show confusion matrices and weighted performance values obtained when running the baseline against our dataset, respectively. Also in this case, we report the results obtained with the best classifier, that in this case was Logistic Regression—confirming the findings of the original authors [26].

In the first place, we can notice that the baseline is decently accurate and, indeed, its F-Measure values on *Blob*, *Complex Class*, and *Spaghetti Code* range between 63% and 84%. The exception is *Shotgun Surgery* (F-Measure=35%), where the baseline fails the classification in most of the cases. Despite its performance, however, the baseline never outperforms our technique. While this is especially true when considering *Shotgun Surgery* (-37% of F-Measure, -17% of AUC-ROC), also for the other code smells the difference is non-negligible: the F-Measure is 19%, 17%, and 5% lower than our model for *Blob*, *Complex Class*, and *Spaghetti Code*, respectively.

The main reason for these differences is likely imputable to the metrics employed. As shown in $\mathbf{RQ}_2$, structural aspects of source code can only partially contribute to the classification of the developer's perceived criticality of code smells and, as such, the inclusion of factors covering other dimensions better fits the problem.

Of particular interest is the analysis of the results for the *Spaghetti Code* smell, where the baseline has the highest performance despite the fact that our findings in $\mathbf{RQ}_2$ reported structural aspects to be negligible. The baseline employs a variety of metrics that can capture different aspects of source code (*e.g.*, coupling or cohesion) under different angles (*e.g.*, by considering the lines of code with and without access methods). Some of the complexity metrics are highly correlated to readability of source code and its fault-proneness and, as such, they have the effect of "simulating" the presence of metrics like the one found to be relevant in $\mathbf{RQ}_2$. This claim is supported by an additional analysis in which we compute the correlation (using the Spearman's test) between the metrics used by the baseline and those which turned out to be relevant in our previous analysis (Read., NF, and C3): we discovered that five of them (*i.e.*, WMC-NAMM_type, NOMNAMM_type, AMW_type, CFNAMM_type, and num_final_static_attributes) are highly correlated, *i.e.*, ρ>0.7, to at least one of the variables found in $\mathbf{RQ}_2$.

In conclusion, based on our findings we can claim that an approach solely based on structural metrics cannot be as accurate in the classification of the perceived criticality of code smells as a technique that includes information coming from other sources, confirming again that the problem of code smell management should be tackled in a more comprehensive manner.

> ***Finding 3.*** *The proposed model is, on average, 20% more accurate than the baseline when classifying the perceived criticality of code smells. Only in the case of Spaghetti Code the usage of multiple structural metrics can lead to results similar to those of our model.*

6 THREATS TO VALIDITY

Some threats may have influenced our empirical study. This section discusses and overviews how we addressed them.

Threats to construct validity. Potential issues related to the relationship between theory and observation firstly refer to the dataset exploited in the study. Our goal was to define a developer-driven prioritization approach, and, as such, we needed to collect the developer's perceived criticality of a set of code smells. To this aim, we followed a similar strategy as previous work [67, 79]: we monitored nine large open-source systems for a period of 6 months and inquired the original developers as soon as they modified smelly classes in order to let them rank how harmful the involved code smells actually were. In so doing, we adopted some precautions: (1) First, we detected code smells using state-of-the-art tools [53, 63] that showed high accuracy, yet checking their output to remove false positives; (2) Second, we asked preliminary questions on whether they perceived the presence of a design issue in the proposed class and recognized the same problem they were contacted for. These questions aimed at ensuring that developers were really aware of the code smells they were assessing and, thus, could provide us with reliable feedback. Of course, we are aware that some of the contacted developers might be peripheral contributors without the experience required to assess the harmfulness of code smells. To account for this aspect, we conducted a follow-up verification of the role of the subject developers within their corresponding projects: to this aim, we computed the number of commits they performed (i) over the entire change history of their projects and (ii) on the specific classes they were contacted for. As a result, we discovered that all our respondents have contributions that exceed the median number of commits made by all project's developers both in terms of changes done over the history and on the smelly classes objects of our inquiry. As a conclusion, we can argue that the dataset collection method is sound and allows a reliable analysis of the perceived criticality of code smells.

Another threat in this category may be related to the granularity of the considered code smells, *i.e.*, they are all computed at class-level. We focused on this family of smell since we aimed at characterizing the quality of classes under different angles using various metrics, with the final aim of assessing how well these metrics could be adopted to predict the perceived criticality of code smells. The selection of method-level design issues may have deserved a further, different investigation into the metrics that could capture the quality of methods: we plan to perform such an investigation as part of our future research agenda.

Still related to the dataset, the perceived criticality assigned by developers when building the dataset might have been influenced by the co-occurrence of multiple code smells [2]. We mitigated this problem by presenting to developers classes affected by single code smell types among those considered in this paper, *e.g.*, we only presented cases where a *Blob* did not occur with any of the other smells considered in the study. Nevertheless, we cannot exclude the presence of further design issues among those that we did not consider in the paper. As such, a larger experimentation would be desirable to corroborate our observations.

Finally, it is worth remarking that most of the independent metrics computed as well as the algorithms exploited (*e.g.*, the machine learners, the information gain method) were computed by relying on well-tested, publicly available tools. This allowed us to reduce biases due to re-implementation. Their selection was based on convenience, and particularly on the skills that the authors have with them. There are a few exceptions, like the tool employed to assess the persistence and refactorability of code smells as well as the baseline prioritization technique [26]: in those cases, we either relied on established guidelines (*e.g.*, the refactorability metric has been implemented by taking into account previous research [92]) or followed the exact description provided in the corresponding papers (*e.g.*, the baseline was implemented following step-by-step the indications provided by Arcelli Fontana and Zanoni [26]). To verify the correct (re-)implementation, we also manually checked the output of our tools: specifically, we randomly verified a sample of 30 cases to assess whether (1) the number of refactoring actions identified by the tool matched the refactorings that we could identify by accessing the history of the classes, (2) the persistence of the smell was actually in line with the result given by the tool, and (3) the metrics computed when running the baseline were correct with respect to what we could measure manually. This additional analysis gives us confidence of the reliability of our tooling.

Threats to conclusion validity. As for the relation between treatment and outcome, we relied on a set of widely-used metrics to evaluate the experimented techniques (*i.e.*, SA, precision, recall, F-measure, MCC, and AUC-ROC). Secondly, we exploited appropriate statistical tests to support our findings. As for the machine learning models experimented, the reported results may have been biased by the selected 10-fold cross-validation strategy. Previous research [87] has criticized it because the randomness with which it creates training and test data may lead to an under- or over-estimation of the real performance of a machine learning model—this is especially true in the case of classification algorithms [87]. To verify this possible bias, we conducted an additional analysis following the recommendation by Hall *et al.* [33]: we ran the experimented models multiple times and assessed the stability of the predictions performed. In particular, we ran a 10 times 10-fold cross-validation and measured how many times the classification for a certain smell instance changed in those runs. We observed that in 93% of the cases, such predictions remained stable, thus allowing us to claim that the validation strategy did not bias the core findings of the study. As a final note, when building the machine learners, we also took into account common confounding effects like multi-collinearity and lack of hyper-parameter configurations.

Threats to external validity. As for the generalizability of the results, there are two main considerations. In the first place, we took into account four code smell types. On the one hand, we were somehow required to reduce the scope of the problem, given the amount of effort/time that we required to the involved developers. On the other hand, an extension targeting more code smells is still desirable and part of our future research agenda.

The second point is related to the number of systems considered. We limited ourselves to nine open-source projects from two ecosystems. Yet, these systems are highly active and have been widely studied in the past by the research community, especially because they have various characteristics. Nevertheless, we are aware that our findings might not hold on to other (eco-)systems

or in an industrial setting. It is our goal to replicate the paper in other contexts and corroborate the findings reported so far.

7 CONCLUSION

This paper presented a novel code smell prioritization approach based on the developers' perceived criticality of code smells. We exploited a number of independent variables (a.k.a. predictors), describing several aspects related to code quality to predict the criticality of code smells, computed by collecting feedback from original developers about their perception of 1,332 code smell instances. Then, we applied several machine learning techniques to perform the classification of a three-level variable describing the code smell criticality, and compared their results with a state-of-the-art tool. The results reported Random Forest to be the best machine-learning algorithm with an F-measure ranging between 72% and 85%. Moreover, we found that our approach is, on average, 20% more accurate than the considered baseline when classifying the perceived criticality of code smells.

Future work includes (1) further improvements of the approach, *e.g.*, by considering social network analysis metrics, (2) an experimentation with a larger number of code smells, and (3) an *in-vivo* assessment of our technique.

REFERENCES

[1] [n. d.]. Developer-Driven Code Smell Prioritization— Online Appendix. https://figshare.com/s/94c699da52bb7b897074.

[2] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. 181–190.

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Acm sigmod record*, Vol. 22. ACM, 207–216.

[4] Roberta Arcoverde, Everton Guimarães, Isela Macía, Alessandro Garcia, and Yuanfang Cai. 2013. Prioritization of code anomalies based on architecture sensitiveness. In *2013 27th Brazilian Symposium on Software Engineering*. IEEE, 69–78.

[5] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* (2019).

[6] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern information retrieval*. Vol. 463. ACM press New York.

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

[8] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 4–14.

[10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 47–52.

[11] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

[12] Raymond PL Buse and Westley R Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.

[13] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. [n. d.]. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software* 143 ([n. d.]), 14–28.

[14] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. 2019. Improving change prediction models with code smell-related information. *Empirical Software Engineering* (02 Aug 2019). https://doi.org/10.1007/s10664-019-09739-0

[15] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. How the Experience of Development Teams Relates to Assertion Density of Test Classes. (2019), to appear.

[16] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[17] William J Clancey. 1984. *Classification problem solving*. Stanford University Stanford, CA.

[18] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software*. IEEE, 23–31.

[19] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Transactions on Software Engineering* (2018).

[20] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2017. A developer centered bug prediction model. *IEEE Transactions on Software Engineering* (2017).

[21] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 612–621.

[22] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 18.

[23] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2012. Identification and Application of Extract Class Refactorings in Object-oriented Systems. *Journal of Systems and Software* 85, 10 (Oct. 2012), 2241–2260. https://doi.org/10.1016/j.jss.2012.04.013

[24] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 609–613.

[25] 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, 3 (2016), 1143–1191.

[26] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58.

[27] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[28] Tudor Girba, Stéphane Ducasse, and Michele Lanza. 2004. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 40–49.

[29] Giovanni Grano, Fabio Palomba, and Harald C Gall. 2019. Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators. *IEEE Transactions on Software Engineering* (2019).

[30] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[31] David Gunning. 2017. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web* 2 (2017).

[32] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[33] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. Developing fault-prediction models: What the research can show industry. *IEEE software* 28, 6 (2011), 96–99.

[34] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 78–88.

[35] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems* 3, 3 (1996), 143–158.

[36] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E Hassan. 2016. Examining the stability of logging statements. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd Int'l Conf. on*, Vol. 1. IEEE, 326–337.

[37] Amandeep Kaur, Sushma Jain, and Shivani Goel. 2017. A support vector machine based approach for code smell detection. In *2017 International Conference on Machine Learning and Data Science (MLDS)*. IEEE, 9–14.

[38] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. 2011. Search-based design defects detection by example. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 401–415.

[39] 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), 243–275.

[40] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*. IEEE, 305–314.

[41] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.

[42] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. 2016. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* (2016), 1–35.

[43] Wei Li and Sallie Henry. 1993. Maintenance metrics for the object oriented paradigm. In *[1993] Proceedings First International Software Metrics Symposium*. IEEE, 52–60.

[44] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[45] Huan Liu and Hiroshi Motoda. 2012. *Feature selection for knowledge discovery and data mining*. Vol. 454. Springer Science & Business Media.

[46] Lech Madeyski and Marian Jureczko. 2015. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal* 23, 3 (2015), 393–422.

[47] Usman Mansoor, Marouane Kessentini, Bruce R Maxim, and Kalyanmoy Deb. 2017. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal* 25, 2 (2017), 529–552.

[48] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 381–384.

[49] Andrian Marcus and Denys Poshyvanyk. 2005. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 133–142.

[50] Radu Marinescu. 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development* 56, 5 (2012), 9–1.

[51] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[52] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[53] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36.

[54] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.

[55] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. 1996. *Applied linear statistical models*. Vol. 4. Irwin Chicago.

[56] Raymond S Nickerson. 1998. Confirmation bias: A ubiquitous phenomenon in many guises. *Review of general psychology* 2, 2 (1998), 175–220.

[57] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. 2009. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 390–400.

[58] Roberto Oliveira, Leonardo Sousa, Rafael de Mello, Natasha Valentim, Adriana Lopes, Tayana Conte, Alessandro Garcia, Edson Oliveira, and Carlos Lucena. 2017. Collaborative identification of code smells: a multi-case study. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 33–42.

[59] Juliana Padilha, Juliana Pereira, Eduardo Figueiredo, Jussara Almeida, Alessandro Garcia, and Cláudio Sant'Anna. 2014. On the effectiveness of concern metrics to detect code smells: An empirical study. In *International Conference on Advanced Information Systems Engineering*. Springer, 656–671.

[60] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* (2017), 1–34.

[61] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology* 99 (2018), 1–10.

[62] 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. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*. IEEE, 101–110.

[63] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2015), 462–489.

[64] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. 2014. Anti-pattern detection: Methods, challenges, and open issues. In *Advances in Computers*. Vol. 95. Elsevier, 201–238.

[65] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[66] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering* 44, 10 (2017), 977–1000.

[67] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering* (2018).

[68] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322.

[69] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* 45, 2 (2017), 194–218.

[70] David Lorge Parnas. 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. IEEE, 279–287.

[71] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2019. On the Role of Data Balancing for Machine Learning-Based Code Smell Detection. (2019).

[72] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 93–104.

[73] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[74] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[75] A Ananda Rao and K Narendar Reddy. 2007. Detecting bad smells in object oriented design using design change propagation probability matrix 1. (2007).

[76] A. J. Scott and M. Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* 30 (1974), 507–512.

[77] Claude E Shannon. 1951. Prediction and entropy of printed English. *Bell system technical journal* 30, 1 (1951), 50–64.

[78] Forrest Shull, Davide Falessi, Carolyn Seaman, Madeline Diep, and Lucas Layman. 2013. Technical debt: Showing the way for better transfer of empirical results. In *Perspectives on the Future of Software Engineering*. Springer, 179–190.

[79] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 858–870.

[80] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2012. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* 39, 8 (2012), 1144–1156.

[81] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2016. Do code smells impact the effort of different maintenance programming activities?. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 393–402.

[82] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 908–911.

[83] Diomidis Spinellis. 2005. Tool writing: a forgotten art?(software tools). *IEEE Software* 22, 4 (2005), 9–11.

[84] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. 2002. Code quality analysis in open source software development. *Information Systems Journal* 12, 1 (2002), 43–60.

[85] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92 (2017), 223–235.

[86] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 812–823.

[87] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2017), 1–18.

[88] Budd Timothy. 2008. *Introduction to object-oriented programming*. Pearson Education India.

[89] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[90] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Ranking refactoring suggestions based on historical volatility. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 25–34.

[91] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[92] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1–15.

[93] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23, 3 (2016), 501–532.

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

[95] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 242–251.