Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Pecorelli[1], Fabio Palomba[2], Dario Di Nucci[3], Andrea De Lucia[1]

[1]University of Salerno, Italy, [2]University of Zurich, Switzerland, [3]Vrije Universiteit Brussel, Belgium

fpecorelli@unisa.it, palomba@ifi.uzh.ch, dario.di.nucci@vub.be, adelucia@unisa.it

Abstract—**Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules; while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artefacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DECOR, a state-of-the-art heuristic-based approach. Key findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection: while DECOR generally achieves better performance than a machine learning baseline, its precision is still too low to make it usable in practice.**

Index Terms—**Code Smells Detection; Heuristics; Machine Learning; Empirical Study**

I. INTRODUCTION

Software maintenance and evolution is a complex activity that enforces developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to set aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect systems maintainability in the future. One of the foremost indications of the presence of technical debt is represented by *code smells* [6], *i.e.,* sub-optimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells has been investigated under different perspectives [8], [9]. Their introduction [10], [11] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19], as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbes *et al.* [23] and Yamashita and Moonen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artefacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha *et al.* [27] devised DECOR, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DECOR, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, *e.g.,* the F-Measure of DECOR when detecting *Blob* instances is ≈80%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adopted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (*e.g.,* Logistic Regression [33]) exploits a set of independent variables (a.k.a., predictors) to calculate the value of a dependent variable (*i.e.,* the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

releases, 13 software systems, and 5 code smell types—in which we compare the performance of machine learning techniques and heuristic approaches for code smell detection. We experiment with five code smell prediction models built using different algorithms and compare their performance with DECOR [27], a state-of-the-art heuristic-based approach that is the most adopted one in literature [8], [25].

Our findings report that DECOR has slightly better performance than machine learning approaches, thus indicating that heuristic techniques for code smell detection still perform better. However, its precision is extremely low and, as such, its application in practice would be still limited. As a consequence, we argue that the research on both machine learning and heuristic approaches for code smell detection requires substantial advances to make such techniques effective.

Structure of the paper. Section II discusses the literature related to code smell detection. Section III describes the design of the empirical study, while Section IV analyses the achieved results. Section VI further discusses the results of the study and overviews the possible threats affecting our findings. Finally, Section VII concludes the paper.

II. RELATED WORK

Over the last decades, several researchers studied the nature and the impact of code smells [8]. The research on the topic can be broadly categorised in two sets. On the one hand, a number of empirical studies have been conducted with the aim of understanding (i) the evolution of code smells [10]–[13], [15], [16], [35], (ii) their perception [20], [21], [36]–[38], and (iii) their impact on non-functional attributes of source code [7], [17]–[19], [23], [24], [39]–[41]. On the other hand, detection approaches exploiting structural source-code-related information [27], [42]–[44], alternative sources (*i.e.,* historical [45] or textual [46] properties of source code), and search-based software engineering methods [47]–[50] have been proposed and assessed.

In this paper, we focus on the comparison between traditional heuristic approaches and supervised methods for metric-based code smell detection. In this section we briefly overview the literature related to both the aspects; a comprehensive review of the state of the art is available in the surveys recently conducted by de Paulo Sobrinho *et al.* [8] and Azeem *et al.* [9].

A. Heuristic Detection of Code Smells

Heuristic approaches identify code smells by means of detection rules based on software metrics. The general process followed by such approaches consists of two steps: (i) the identification of the key *symptoms* characterising a code smell that can be mapped to a set of thresholds based on structural metrics (*e.g.,* if Lines Of Code $\geqslant k$); (ii) the combination of these symptoms, which leads to the final rule for detecting the smell [27], [42]–[44], [51]. The detection techniques falling into this category mainly differ in the set of the structural metrics exploited, which depends on the type of code smells to detect and how the identified key symptoms are combined. Most of the proposed approaches obtained such a combination

by employing AND/OR operators [42], [44], [51], while more recent ones adopted clustering methods [43].

In this context, Moha *et al.* [27] introduced DECOR, a method to specify and detect code and design smells using a Domain-Specific Language (DSL). Following the general process described above, DECOR uses a set of rules, called "rule card"[1], that describe the intrinsic characteristics of a class affected by a smell. For instance, a *Blob* is detected when a class has an LCOM5 (Lack of Cohesion Of Methods) [52] higher than 20, a number of methods and attributes higher than 20, a name that contains a suffix in the set {*Process*, *Control*, *Command*, *Manage*, *Drive*, *System*}, and it has a one-to-many association with data classes. The authors showed that DECOR can identify smells with an average F-Measure of $\approx 80\%$.

Tsantalis *et al.* [43] presented JDEODORANT, a tool whose first version was able to detect *Feature Envy* bad smells and suggest move method refactoring opportunities. Afterwards, other code smells have been supported (*i.e., State Checking, Long Method,* and *Blob*) [53]–[55]. The detection strategies for these smells are based on code metrics that are then connected to each other using supervised clustering algorithms and thresholds to cut the resulting dendrograms. The empirical assessment of the performance of JDEODORANT showed its high accuracy (on average, $\approx 75\%$).

Despite the good performance achievable with the discussed techniques, previous work [25], [28] pointed out three important limitations that might preclude their use in practice: (i) subjectiveness of developers with respect to code smells detected by these tools, (ii) scarce agreement between different detectors, and (iii) difficulties in finding good thresholds to be used for detection. The adoption of machine learning techniques may potentially mitigate these problems, however there is limited evidence of whether and how much machine learning actually improves the performance of traditional approaches: thus, our paper aims at extending previous work by comparing the performance obtainable by heuristic and machine learning approaches for code smell detection.

B. Machine Learning Techniques for Detecting Code Smells

As opposed to heuristic approaches, the techniques in this category exploit a supervised method: more specifically, a set of independent variables (a.k.a., *predictors*) is used to predict the value of a dependent variable (*i.e.,* the smelliness of a class) using a machine learning classifier (*e.g.,* Logistic Regression [33]). The model can be trained using a sufficiently large amount of data available from the project under analysis, *i.e., within-project* strategy, or using data coming from other (similar) software projects, *i.e., cross-project* strategy. These approaches clearly differ from the heuristics-based ones, as they rely on classifiers to discriminate the smelliness of classes rather than on predefined thresholds upon computed metrics.

Kreimer [56] proposed a prediction model that, on the basis of code metrics used as independent variables, can lead to high values of accuracy. It adopts DECISION TREES to detect two

---

[1] http://www.ptidej.net/research/designsmells/

Table I
PROJECTS CONSIDERED IN THE STUDY

Project	Description	# Releases	# Classes	# Methods
Ant	Build System	10	1,002-1,218	9,333-11,919
ArgoUML	UML Modeling Tool	13	889-2,221	7,252-17,309
Cassandra	Database Management System	8	470-727	4,422-7,901
Derby	Relational Database Management System	9	1,733-2,920	23,107-421,183
Eclipse	Integrated Development Environment	21	812-5,736	10,819-51,008
Elastic Search	RESTful Search and Analytics Engine	8	1,785-2,466	12,393-18,225
Hadoop	Tool for Distributed Computing	9	148-344	1,224-3,080
HSQLDB	HyperSQL Database Engine	10	556-601	10,075-11,016
Incubating	Codebase	6	497-787	4,210-5,767
Nutch	Web-search Software	7	304-453	1,846-2,761
Qpid	Messaging Tool	5	1,547-2,118	14,858-20,402
Wicket	Java Application Framework	9	2,133-2,212	12,370-12,824
Xerces	XML Parser	10	483-542	5,280-6,126

Table II
DESCRIPTIVE STATISTICS FOR SMELLS DISTRIBUTION

Code Smell	min	mean	median	max	total
God Class	0	5.5	4	24	509
Spaghetti Code	0	12.7	11	31	1443
Class Data Should Be Private	0	11.4	11	37	1150
Complex Class	0	6.4	4	20	669
Long Method	3	48.3	26	147	4763

code smells (*i.e., Blob* and *Long Method*). Later on, Amorim *et al.* [57] confirmed the previous findings by evaluating the performance of DECISION TREES on four medium-scale open-source projects. Vaucher *et al.* [58] studied *Blob*'s evolution relying on a NAIVE BAYES classifier, whereas Maiga *et al.* [59], [60] proposed the use of SUPPORT VECTOR MACHINE (SVM) and showed that such a model can reach an F-Measure of ≈80%. The use of BAYESIAN BELIEF NETWORKS to detect *Blob*, *Functional Decomposition*, and *Spaghetti Code* instances on open-source programs, proposed by Khomh *et al.* [61], [62] lead to an overall F-Measure close to 60%. Similarly, Hassaine *et al.* [63] defined an immune-inspired approach for the detection of *Blob* smells, while Oliveto *et al.* [64] used a B-Splines to detect them. More recently, some authors investigated the feasibility of machine-learning to detect code clones [65]–[67]. Arcelli Fontana *et al.* made the most relevant progress in this field [32], [68], [69]. In their work, they (i) theorised that ML might lead to a more objective evaluation of code smells hazardousness [68], (ii) provided a ML method to assess code smell intensity [69], and (iii) compared 16 ML techniques for the detection of four code smell types [32] showing that ML can lead to F-Measure values close to 100%. Nevertheless, recently Di Nucci *et al.* [34] demonstrated that, in a real use-case scenario, the results achieved by Arcelli Fontana *et al.* [32] cannot be generalised, thus contrasting the real effectiveness of machine learning for code smell detection.

Our work builds on top of the papers discussed above and aims at comparing machine learning approaches with heuristic metric-based ones to assess the real capabilities of ML in the context of code smell detection and provide a more solid ground for future research in the field.

III. EMPIRICAL STUDY DEFINITION AND DESIGN

The *goal* of the study is to compare heuristic with machine learning approaches for code smell detection, with the *purpose* of assessing the extent to which code smell prediction models can be effectively used in practice. The *perspective* is of both researchers and practitioners: both of them are interested in evaluating the performance of machine learning for code smell detection, while the former are interested also in understanding possible limitations of the current approaches. More specifically, we aim at addressing the following research question:

RQ. *How do machine learning-based techniques for code smell detection perform when compared to a baseline heuristic-based approach?*

With this research question we aim at providing a deeper knowledge on the capabilities of machine learning for code smell detection. The following subsections report the methodological steps that we conducted to address **RQ**.

A. Context of the Study

The *context* of the study was represented by software systems and code smells. As for the former, we exploited a publicly available dataset composed of 125 releases of 13 open source software systems [70], whose description is reported in Table I. The projects of the dataset are heterogeneous, have different size, lifetime, and belong to various application domains: as such, we could reduce threats to the generalizability of the empirical study. The dataset contains a set of $8,534$ *manually validated* code smell instances of five different types: thus, we could perform our study on a dataset of real code smells, in contrast with the artificially created ones that were used in previous research on code smell prediction [9].

Table II reports the descriptive statistics related to the distribution of code smells. As it is possible to observe, the median number of code smells in each considered release is pretty low (it ranges from 4 to 26). The absolute numbers correspond to extremely low relative percentages: as an example, we noticed that the maximum number of God Class instances (24), in the project APACHE DERBY 10.3.3.0, only represents the 1% of the total number of classes belonging to this system (2220). On the one hand, the observed distribution

confirms previous findings in the field [70]. On the other hand, the extremely low number of code smells clearly evidences that the problem in question is highly unbalanced.

Table III
DETECTION RULES USED BY THE HEURISTIC-BASED APPROACH.

Code Smell	Detection Rule
God Class	ELOC > 500 ∧ (NOM+NOA > 20 ∨ LCOM > 20)
Spaghetti Code	ELOC > 600 ∧ NMNOPARAM > 0
Class Data Should Be Private	NOPA > 10
Complex Class	WMC > 50
Long Method	LOC_METHOD > 100 ∧ NP > 1

With respect to code smells, we considered five different types defined in the catalog by Fowler [6]:

God Class. This smell characterizes classes having a large size, poor cohesion, and several dependencies with other data classes of the system [6]. Previous work showed that this smell has a negative impact on both program comprehension and software maintainability [19], [23], [70].

Spaghetti Code. Classes affected by this smell exhibit a functional-style programming structure, declaring a number of long methods without parameters [6]. Also in this case, the negative impact on comprehensibility and maintainability has been previously shown [19], [23], [70].

Class Data Should be Private. This smell appears in cases where a class exposes its attributes, thus violating the information hiding principle [6]; in this case, previous work showed that developers often do not recognize the presence of this smell and consider it as less harmful than others for maintainability [20], [22].

Complex Class. Classes presenting a overly high cyclomatic complexity [71] are affected by this design flaw. As shown in the literature [19], [20], [70], it can worsen software maintainability and reduce the ability of developers to properly enhance the corresponding source code.

Long Method. Methods implementing more than one functionality are affected by this smell [6]. It can lower program understanding and make the source code more change- and fault-prone [19], [20], [70].

In the remaining of this section, we explain the machine learning-based and heuristic-based detection solutions exploited in our study to identify instances of these code smells.

Table IV
FULL NAMES OF THE CONSIDERED METRICS.

Acronym	Full Name
ELOC	Effective Lines Of Code
LCOM	Lack of COhesion in Methods
LOC_METHOD	Lines Of Code of METHOD
NOA	Number Of Attributes
NOM	Number Of Methods
NOPA	Number Of Public Attributes
NP	Number of Parameters
NMNOPARAM	Number of Methods with NO PARAMeters
WMC	Weighted Methods Count

B. Heuristic-Based Detection of Code Smells

Among all the techniques and tools available for code smell detection [8], [25], we relied on DECOR [27] as a metric-based heuristic baseline for several reasons. First, this technique has been extensively used and showed good detection performance (*e.g.,* [45], [46], [61], [72]), thus representing a valid candidate to be a baseline in our study. Furthermore, it is based on detection rules that can be computed directly looking at the source code of a class/method, without the need of computationally-expensive operations (*e.g.,* the construction of the Abstract-Syntax Tree and the subsequent clustering mechanism applied by JDEODORANT [43]) that would have made the detection phase unfeasible because of the amount of data we had to analyze. Last but not least, DECOR is publicly available. It provided out-of-the-box the detection rules able to identify two code smells considered in the study (i.e., *God Class* and *Spaghetti Code*), while for the remaining ones we relied on the definitions provided by Tufano et al. [10]

God Class. A smelly instance is detected when a class has a size higher than 500 lines of code and either an LCOM5 (Lack of Cohesion Of Methods) [52] higher than 20 or a number of methods and attributes higher than 20.

Spaghetti Code. DECOR identifies this smell in cases where a class has a size higher than 600 lines of code and a number of long methods (identified as explained later) without parameters higher than 0.

Class Data Should Be Private. In this case, DECOR computes the Number Of Public Attributes (NOPA) of a class and, if this is higher than 10, then a smell is identified.

Complex Class. The detection rule for this smell considers the Weighted Methods per Class (WMC) metric, namely the sum of the McCabe's cyclomatic complexity [71] of the methods of a class. If WMC is higher than 50, a class is detected as affected by the smell.

Long Method. To detect this smell, the lines of code of the method (LOC_METHOD) and the number of parameters of the method (NP) are used. DECOR indicates the presence of the smell is a method has more than 100 lines of code and at least one parameter.

Table III reports the summary of all the detection rules. The full name of the metrics is reported in Table IV. We ran DECOR over all the 125 releases and, on the basis of the output recommendations, we re-constructed the confusion matrices. These were analyzed and compared with those obtained with the machine learning models in terms of the evaluation metrics described in Section III-D.

C. Machine Learning-Based Detection of Code Smells

Once we had defined the heuristic-based baseline, we configured the machine learning-based classifiers to detect the considered smells. This required several steps, ranging from the definition of the dependent and the independent variables to the pre-processing actions needed to avoid common problems such as multi-collinearity and biased interpretation [73].

Dependent variable. Since in our work we were interested in detecting code smells, we set the presence/absence of a certain code smell as dependent variable of the machine learning model. This information was already available in the considered dataset.

Independent variables. As the overall goal of the study was the comparison between heuristic and machine learning-based detectors, we wanted to avoid that such a comparison could have been biased by other co-factors. For this reason, the independent variables of the model were exactly the same used by the heuristic approach (see Table III): in this way, we avoid the possibility that different performance might be due to the selected metrics rather than the technique exploited.

Selection of the classifier. While several classifiers have been previously used for code smell detection, the related literature showed that it is unclear which of them represents the best solution [9]. For this reason, in this work we compared the five most commonly used ML algorithms such as J48 [74], RANDOM FOREST [75], NAIVE BAYES [76], SUPPORT VECTOR MACHINES [77], and JRIP [78]. To perform a fair comparison, we applied the same configuration, preprocessing, and training strategies to all the classifiers, as described in the following. For the sake of space limitations, in Section IV we only report the results achieved by the best classifier, *i.e.,* NAIVE BAYES, while a comprehensive report of the accuracy of the other classifiers is available in our online appendix [79].

Configuration and preprocessing steps. Before assessing the accuracy of the machine learning-based models, we took into account two aspects, *i.e.,* classifier configuration and feature selection, that might possibly bias their performance [80]–[82]. We configured the hyper-parameters of the considered classifiers by exploiting the GRID SEARCH algorithm [83], that implements an exhaustive searching approach of the hyper-parameter space. Secondly, we removed highly correlated independent variables through the CORRELATION-BASED FEATURE SELECTION (CFS) approach [84]: this method uses correlation measures and a heuristic search strategy to identify a subset of actually relevant features for a model. We applied the feature selection algorithm on each release independently, so that the model took into account only the features that are relevant for a specific release of the considered systems. It is important to note that we consciously avoid the application of balancing algorithms [85], *i.e.,* techniques that ensure a similar proportion of smelly and non-smelly classes/methods in the training set. This decision was taken as a result of experimental tests, where we observed that such algorithms can bias the interpretation of the results in the context of code smell prediction. Detailed motivations and analyses of this aspect are reported in Section V.

Validation strategy. To assess the capabilities of the machine learning models, we adopted *10-Fold Cross Validation* [86]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (*e.g.,* each fold has the same proportion of code smell instances). A single fold is used as test set, while the remaining ones are used as training set. The process was repeated 10 times, using each time a different fold as test set.

The result of the process described above consisted of a confusion matrix for each code smell type, for each of the 125 releases of the considered projects and for each experimented classifier. These matrices have been later analyzed to measure the evaluation metrics described in the following section.

D. Data Analysis and Metrics

To assess the performance of the experimented detection techniques we computed three well-known metrics [87], namely, *precision*, *recall* and *F-measure*. In addition, we also computed the *Matthews Correlation Coefficient (MCC)* [88]. Since we considered several releases of several systems, we needed to aggregate the results achieved for each release to have a clearer overview of the performance [89]. Therefore, we aggregated the obtained confusion matrices before computing precision, recall, F-Measure, and MCC.

Formally, let *TP* (True Positives) be the actual smelly instances that have been correctly identified as smelly by an approach, *FP* (False Positives) the non-smelly instances that have been erroneously identified as smelly, *TN* (True Negatives) the non-smelly instances that have been correctly identified as non-smelly, and *FN* (False Negatives) the smelly instances that have been erroneously identified as non-smelly, we computed:

- **Precision.** It represents the fraction of instances predicted as smelly that are actually smelly, namely:

$$Precision = \frac{\sum_i \#TP_i}{\sum_i \#(TP_i + FP_i)}\% \qquad (1)$$

- **Recall.** represents the fraction of actually smelly instances that have been correctly predicted as smelly:

$$Recall = \frac{\sum_i \#TP_i}{\sum_i \#(TP_i + FN_i)}\% \qquad (2)$$

- **F-measure.** is defined as the weighted harmonic mean of the precision and recall, and it is computed as:

$$F - Measure = 2 * \frac{precision * recall}{precision + recall} \qquad (3)$$

- **MCC** is a correlation coefficient between the observed and predicted binary classifications. It has values in the range [-1,+1] where a coefficient of +1 represents a perfect prediction and 1 indicates total disagreement between prediction and observation:

$$MCC = \frac{\sum_i \#(TP_i * TN_i - FP_i * FN_i)}{\sum_i \#\sqrt{(TP_i + FP_i)(TP_i + FN_i)(TN_i + FP_i)(TN_i + FN_i)}} \qquad (4)$$

where *i* ranges over the entire dataset, including the releases with no smelly instances. Aggregate metrics are more robust than the mean, which is biased by the fact that datasets are unbalanced for different smell types in terms of smelly and

Table V
AGGREGATE RESULTS FOR PRECISION, RECALL, F-MEASURE, AND MCC

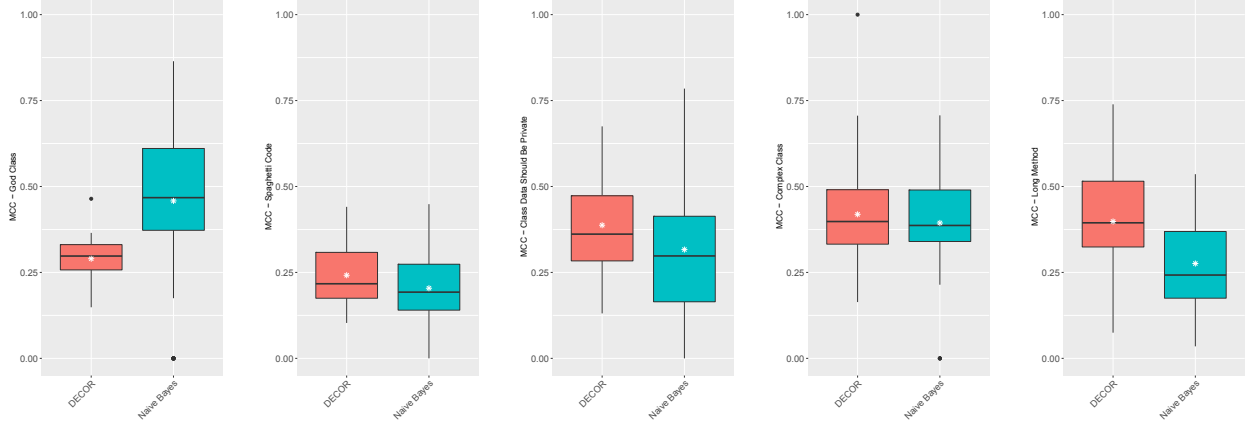|	Precision		Recall		F-Measure		MCC	
	NB	DECOR	NB	DECOR	NB	DECOR	NB	DECOR
God Class	**0.27**	0.08	0.85	**1**	**0.41**	0.16	**0.47**	0.28
Spaghetti Code	**0.15**	0.11	0.30	**0.47**	**0.20**	0.18	0.20	**0.22**
Class Data Should Be Private	**0.29**	0.23	0.34	**0.42**	0.29	**0.30**	0.29	**0.31**
Complex Class	0.23	0.23	0.57	**0.72**	0.33	**0.35**	0.36	**0.37**
Long Method	0.15	**0.57**	**0.56**	0.37	0.23	**0.44**	0.30	**0.42**

Figure 1. Boxplots representing the MCC values obtained by DECOR and NAIVE BAYESIAN (NB) for all the considered code smells

non smelly instances (in some cases the datasets do not contain any smelly instance).

As a final step, we statistically verified the differences between the performance obtained by the experimented approaches. To this aim, we exploited the Wilcoxon test [90] computed on the distributions of MCC values of machine learning-based and heuristic-based techniques over the different releases and the different smell types. The results are intended as statistically significant at α=0.05. Furthermore, we estimated the magnitude of the measured differences by using Cliff's Delta (or d), a non-parametric effect size measure [91] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [91].

IV. ANALYSIS OF THE RESULTS

Table V shows the aggregate results for precision, recall, F-measure, and MCC achieved by the machine learning model ("NB" in the table) and DECOR. The overall results immediately highlight that the performance of both the approaches is generally low: indeed, the maximum F-measure is 41% and 44% for the NB and DECOR, respectively. This is especially due to the extremely low precision achieved over the entire dataset. More in detail, the high number of false positives is likely influenced by the fact that the dataset contains instances of different code smell types that sometimes have characteristics that may bias the detection approaches. As an example, let consider the cases of *God Class* and *Complex*

Class. While the detection rules for these smells are different, it is reasonable to believe that some code metrics tend to have a similar distributions in the classes affected by those smells; for instance, being a *God Class* poorly cohesive and with a large number of methods, it is likely that also the complexity of the class tends to be high. This is the case of `taskdefs.optional.net.FTP` of the APACHE ANT 1.6.0 system, that is a *God Class* but has WMC=39. Such a value is not that high to make the class affected by *Complex Class* too, but it is enough to confound the machine learning technique, which wrongly signals the class as complex, thus giving a false positive. This result seems to suggest that an improved characterization of the symptoms behind specific code smell types (*e.g.,* by means of textual or historical analyses [45], [46], [92]) may make code smell detection more effective.

Table VI
COMPARISON BETWEEN NB AND DECOR IN TERMS OF WILCOXON AND CLIFF'S DELTA EFFECT SIZES. SIGNIFICANT P-VALUES ARE REPORTED IN BOLD FACE.

	p-value	d	Meaning
God Class	**<0.01**	0.13	Negligible
Spaghetti Code	**0.01**	-0.29	Small
Class Data Should Be Private	**<0.01**	-0.43	Medium
Complex Class	**0.01**	-0.41	Medium
Long Method	**<0.01**	-0.33	Small

It is also worth to discuss the values achieved when considering the recall. In this case, we observed that DECOR is superior to NB in most cases: this confirms the experimental

Table VII
OVERLAP BETWEEN ML AND DECOR IN ABSOLUTE TERMS AND PERCENTAGES

	NB \ DECOR		NB ∩ DECOR		DECOR \ NB		NB ∪ DECOR		¬NB ∩ ¬DECOR	
	#	%	#	%	#	%	#	%	#	%
God Class	0	0	435	85	74	15	509	100	0	0
Spaghetti Code	14	1	419	29	266	18	699	48	744	52
Class Data Should Be Private	91	8	298	26	186	16	575	50	575	50
Complex Class	50	8	329	49	155	23	534	80	135	20
Long Method	1577	33	1076	23	650	14	3303	70	1460	30

results obtained by the original authors [27] on the high recall of the approach. Finally, the low MCC values of both the approaches (see Figure 1) confirm that code metrics are not enough when it turns to code smell detection [93], [94].

From a statistical point of view, Table VI reports the results of the Wilcoxon and Cliff's delta tests computed on the MCC values of the experimented approaches. As shown, the difference between the techniques are statistically significant in all cases ($p-$values lower than 0.05). With the exception of *God Class*—where the machine learning model achieves higher MCC—all the other cases show that DECOR is statistically better than the baseline. Nevertheless, these differences are mostly negligible or have at most a medium effect.

In the following subsections, we discuss the findings achieved by considering each code smell independently and reporting qualitative examples aimed at further analyzing the performance of the detectors. Also, we discuss the complementarity between the sets of code smells correctly identified by the detectors (see VII), namely the extent to which NB and DECOR are able to identify the same instances.

A. Results for God Class

Looking at the results in both Table V and Figure 1, we can say that this smell is the easiest to detect and, in fact, all the instances affected by this smell have been correctly classified as smelly by at least one of the experimented techniques. In particular, we note that DECOR reaches a recall of 100%, which means that it is able to detect all *God Class* instances. Nevertheless, the high recall has a cost in terms of precision, that is just 8%. On the one hand, our findings are in line with those reported in previous studies [27], [45]. On the other hand, they confirm the need for further methodologies able to improve metric-based code smell detection.

When considering the complementarity of the approaches (Table VII), our results indicate a high overlap (85%): this means that in the vast majority of cases NB and DECOR can identify the same instances. However, a total of 74 actual *God Class* cases, corresponding to 15%, were only correctly identified by DECOR and missed by NB. To better understand the reasons that enable the heuristic approach to discover different instances than the machine learning model, we went manually over the outputs of the techniques to conduct a manual analysis. As a result, we found that in most cases the machine learning model is biased because it considers all the metrics together to make a prediction, while the heuristic approach can logically combine them obtaining better results. As an example,

let consider the class CBZip2OutputStream belonging to the project APACHE ANT 1.6.3: despite it is characterized by an LCOM < 20 (*i.e.,* 19), it respects the rule reported in Table III, as it has an ELOC = 2346 and NOM+NOA = 161. As such, the heuristic approach can still correctly identify its smelliness. On the contrary, the machine learning model classifies the instance as non-smelly.

B. Results for Spaghetti Code

As opposed to the previous case, this smell does not seem to be easily detectable automatically, as demonstrated by the results in Table VII, where we can see that more than half of the instances affected by this smell are not detected as smelly by any of the approaches. Overall, the performance achieved by the machine learning technique is extremely low, both in terms of precision and recall (15% and 30%, respectively). At the same time, DECOR has a higher recall (+17% with respect to other technique), but a lower precision (-4%), which had the effect to make the overall results of the two approaches comparable (F-Measure for DECOR is just 2% lower than NB, while MCC is 2% higher). Thus, we can claim that the metric-based detection is not able to provide good results, independently from the underlying technique adopted. Once again, this may indicate the need for further work aimed at improving the characterization of this smell type. Looking at the complementarity, also in this case the overlap is higher than the number of instances correctly detected by only one of the two. However, in the remaining cases DECOR is able to identify 266 code smell instances (18%) that are not correctly detected by the machine learning model. For example, the class MeridioAuthority of the APACHE INCUBATING 0.3 project is correctly detected only by DECOR. Basically, the reason is exactly the same observed before: the value of the ELOC metric of this class is 661, which is very close to the threshold (*i.e.,* 600). While the heuristic technique discriminates based on thresholds, thus identifying the smelly class, the ML approach might be confounded by borderline metric values.

C. Results for Class Data Should Be Private

As for this smell, half of the smelly instances are not identified by any of the two techniques: this seems to be a clear indication of the need for more effective detection strategies. Between the two experimented techniques, DECOR is the one with the highest recall and this allows it to be slightly better than the machine learning model, overall. This is likely due to the very simple, yet clearer, detection rule applied by DECOR to identify instances of this smell. Conversely, the machine

learning model seems to be less stable both in terms of precision and recall because it may be confounded by borderline values. The results shown in Table VII confirm that the prediction model can only identify a limited number of instances that are not identified by DECOR (91), while in most cases there is an overlap (298) or the heuristic approach works better (186).

D. Results for Complex Class

The detection rule for this smell is only based on WMC, so the only factor that can determine different predictions is the value of this metric. First, we can confirm the results discussed so far, with DECOR having a high recall but a low precision. Moreover, the MCC of both the approaches is slightly in favor of DECOR (0.37 vs 0.36), indicating that (i) there is not a clear winner between the two and (ii) more sophisticated techniques for the detection of this smell would be worthwhile. The discussion of the overlap metrics is also very similar to the other smells discussed above. In general, we observed that the values that bring to an erroneous detection are the ones close to the threshold boundaries. The impact of boundary values can be also analyzed by another point of view. Let consider the class `ServerSession` in the APACHE CASSANDRA 0.8.0 project, which shows a WMC = 47 that can be considered high but does not exceed the threshold of 50. In this case, the ML technique correctly detects the instance as smelly, while DECOR cannot.

E. Results for Long Method

As for *Long Method*, this was the only case in which the machine learning technique had higher recall than DECOR (*i.e.,* +19%). Also for this code smell, the differences between the two approaches mainly concern instances with metric values close to the thresholds used by DECOR. An example is provided by the method `doSnapshot` belonging to the class `BlobStoreIndexShardGateway` of the *Elasticsearch* 0.17.0 project. The method under considerations has 79 lines of code and takes only 1 parameter. It is correctly detected by the prediction model as smelly but not by DECOR because it requires 100 or more lines of code and more than one parameter to identify the smell.

V. DISCUSSION AND IMPLICATIONS

The results of of our study provided a number of insights that deserve some further considerations.

A. On the Role of Data Balancing

As we discussed in Section III, we consciously chose to not use balancing techniques when building our code smell prediction model. This choice was due to experimental tests where we compared the balanced version of the model with the non-balanced one. As balancing algorithm, we applied the SYNTHETIC MINORITY OVER-SAMPLING TECHNIQUE, (SMOTE) [85]: to ensure a similar proportion of elements belonging to the two classes (*i.e.,* smelly/non-smelly), it takes samples of the feature space for each target class and its nearest neighbours, and generates new examples that combine features of the target case with features of its neighbours.

The results of this analysis are shown in Figure 2. At a first sight, it may seem that the balanced model achieves better performance for most of the considered smells (except the one created for *Long Method*). However, by further investigating this result we realized that such performance is biased by the high number of cases in which the data balancing algorithm failed, not producing any valid predictions. In particular, the problem of code smell detection is strongly unbalanced and, in many cases, SMOTE does not have a feature space large enough to perform a balancing, thus producing a failure. Such failures do not contribute to the computation of the evaluation metrics used for performance assessment. As such, the interpretation of the results would have been biased by the absence of many predictions.

To further investigate this aspect, we tried to reduce the failure rate by considering the default CLASS BALANCER provided by WEKA: differently from SMOTE, it performs a very simple balancing that has the goal of re-weighting the instances in each class to obtain the same total class weight. Thus, it theoretically reduces the number of failures as it does not require a large feature space. As expected, the failure rate was actually reduced but, however, the results (shown in Figure 2) showed lower performance than the ones produced by the non-balanced model in most cases.

Thus, our findings have two clear implications for the research community. First, the problem of code smell detection is naturally unbalanced and, for this reason, *difficult to treat with machine learning techniques*: as such, researchers interested in this finding are called to devise novel effective strategies to make machine learning really suitable for code smell detection. Perhaps more importantly, the problem of code smell prediction lends itself to possible *interpretation bias*: thus, we advice researchers in the field to carefully analyze the internal mechanisms of prediction models before interpreting their goodness.

B. On the Performance of Heuristic Approaches

Likely, the most surprising result of our study concerns the fairly low performance achieved by DECOR over the considered dataset. Specifically, while the recall of the approach was in line with the one stated in literature [27], we found its precision to be extremely low. We see two main motivations behind this result. First, in our study we employed DECOR with a larger variety of code smells with respect to previous work [45], [46], [62]: therefore, we tested its performance in the wild, showing some limitations of the technique when employed for the detection of certain code smell types. Secondly, we relied on a manually-validated dataset containing real instances: as shown by previous work [34], the composition of the dataset might influence the performance of a technique; this is especially true in the case of code smell detection, where a detector should recognize code smells over datasets that are both unbalanced (*i.e.,* limited number of actual instances) and noisy (*i.e.,* the presence of several smell types might interfere and make the detection rules less effective).
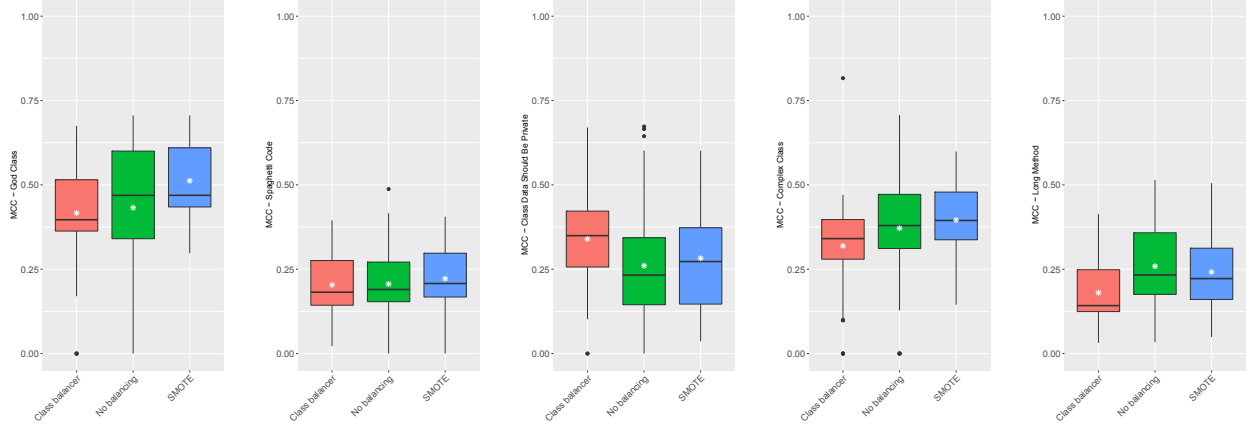
Figure 2. Boxplots representing the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for all the considered code smells

Table VIII
TYPE I AND TYPE II ERRORS ACHIEVED IN THE OVERALL EVALUATION

	Naive Bayes		Optimistic Constant		Pessimistic Constant		Random	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
God Class	1,263 (0.90%)	65 (0.10%)	144,798 (99.60%)	0 (0.00%)	0 (0.00%)	509 (0.40%)	72,683 (50.00%)	251 (0.20%)
Spaghetti Code	2,269 (1.40%)	1,009 (0.60%)	159,436 (99.10%)	0 (0.00%)	0 (0.00%)	1,443 (0.90%)	79,669 (49.50%)	690 (0.40%)
Class Data Should Be Private	874 (0.60%)	770 (0.50%)	142,558 (99.20%)	0 (0.00%)	0 (0.00%)	1,150 (0.80%)	71,221 (49.50%)	589 (0.40%)
Complex Class	1,303 (1.00%)	282 (0.20%)	127,538 (99.50%)	0 (0.00%)	0 (0.00%)	669 (0.50%)	63,507 (49.50%)	335 (0.30%)
Long Method	15,449 (1.20%)	2,101 (0.20%)	1,283,312 (99.60%)	0 (0.00%)	0 (0.00%)	4,763 (0.40%)	641,914 (49.80%)	2,431 (0.20%)

Thus, while heuristic techniques still slightly outperform machine learning models, *the problem of detecting code smells using heuristics is still far from being solved*. We believe that our findings support the preliminary research efforts conducted to filter code smell candidates output by the detectors to reduce the false positive rate, thus improving their precision [95]. At the same time, we also envision further research on how to limit the interaction of multiple code smells with similar characteristics on the performance of code smell detectors: to this aim, we envision the concept of *local smell detection*, that, similarly to what has been done in defect prediction [96], would have the goal of clustering similar classes first (possibly positively affecting the interaction problem) and then apply the detector that is most suitable for the classes of a cluster.

C. On the Performance of Machine Learning Models

As we have observed in Section IV, the performance of machine learning techniques are not as good as the one of heuristic approaches. To further investigate the potential of these techniques, we performed an additional analysis aimed at comparing the model with three simple baselines such as: (i) the OPTIMISTIC CONSTANT classifier, that always classifies an instance as smelly; (ii) the PESSIMISTIC CONSTANT classifier, that always classifies an instance as non-smelly; and (iii) a RANDOM classifier, which randomly classifies an instance as smelly or non-smelly. Should the performance of the model be lower than any of this baseline, it would indicate a major threat to the usability of the model in practice. As previously done in literature [97], we performed this comparison in terms

of *Type I* and *Type II* errors, *i.e.,* computed as the total number of false positive and false negative errors.

Table VIII reports the results achieved. We can observe that, for each of the classifiers, the total number of errors (*i.e.,* Type I + Type II) is independent from the smell to detect. The total number of errors in percentage is between 1% and 2% for NAIVE BAYES, higher than 99% for OPTIMISTIC CONSTANT, less than 1% for PESSIMISTIC CONSTANT and around 50% for RANDOM CLASSIFIER. This means that the PESSIMISTIC CONSTANT outperforms all the other classifiers producing a lower number of errors.

Of course, this result was due again to the unbalanced nature of the problem. However, this has a key implication for the research community: based on our results, *machine learning seems still unsuitable for code smell detection* which also involves practitioners who currently cannot use this approach in practice. The inclusion of orthogonal metrics as independent variables (*e.g.,* process indicators), the adoption of ensemble techniques [98], the experimentation of different training strategies (*e.g.,* cross-project models) are just some of the research fields that would require further attention in the future.

VI. THREATS TO VALIDITY

In this section we discuss the threats to construct, external and conclusion validity of our empirical evaluation.

Construct Validity. Threats in this category are related to the relation between theory and observation. In our study, a threat might be represented by the dataset used for the empirical study. The choice was made considering several

factors such as heterogeneity or the presence of manually-validated data, however we have to consider that they may contain possible incompleteness or imprecisions. Another threat can be represented by the detection techniques adopted in the study: on the one hand, we used DECOR [27], a tool that is publicly available and that has been exploited several times in the past [8]; on the other hand, the construction of the machine learning model took into account several aspects that might have possibly influenced the study, *i.e.,* which features to consider, how to train the classifier, etc. Thus, we believe that the procedures followed in this respect are precise enough to ensure the validity of the study.

External Validity. With respect to the generalizability of our findings, we considered a large dataset consisting of 131 releases of 13 open source systems belonging to different application domains and having different characteristics. Another threat concerns the choice of the techniques for comparison: we exploited NAIVE BAYES as machine learning classifier after comparing the top five most commonly used classifiers in this field [9]. Finally, the selection of the code smell to analyze could be a threat. We selected five smells that represent a large variety of design issues (*e.g.,* smells related to complexity or excessive coupling between objects). This allowed us to better understand the potential of machine learning techniques for code smell detection as well as their limitations with respect to heuristic-based approaches. Of course, further experiments performed on different datasets and techniques would be desirable and already part of our future research agenda.

Conclusion Validity. As for concerns with the relationship between treatment and outcome, we exploited a set of widely-used metrics to evaluate the experimented techniques (*i.e.,* precision, recall, F-measure, MCC) and provided qualitative examples aimed at showing the differences between the compared approaches. Furthermore, we used appropriate statistical tests (*i.e.,* Wilcoxon and Cliff's delta) to support our findings. As for the machine learning model, a possible bias related to the interpretation of the results might have been due to the usage of the 10-fold cross validation. This strategy randomly partitions the set of data to create training and test sets: such randomness might have possibly led to the creation of biased training/test sets that have the consequence of under- or over-estimate the model performance. To account for this aspect, we performed an additional analysis: as suggested by Hall *et al.* [99], we ran the experimented model multiple times to assess how stable it is depending on the random splits performed by the validation strategy. Thus, we ran a 10 times 10-fold cross validation and, then, we measured the variability of the predictions performed by the model; as a result, we observed that in 98% of the cases the predictions do not change over different runs. As such, we can conclude that the results achieved by the model are not influenced by the randomness of the validation strategy.

VII. CONCLUSIONS

In this paper we compared a machine learning approach with a heuristic one for code smell detection, considering five different types of code smells over a dataset composed of 125 releases of 13 open source software systems. The main contributions of this paper can be summarized as follow:

- A large-scale empirical comparison of machine learning and heuristic approaches for metric-based code smell detection, which highlighted that heuristic approaches still perform better, yet have low performance.

- The identification of problems, such as data unbalancing or poor precision, that make the application of both machine learning models and heuristic approaches for code smell detection hard;

- A comprehensive online appendix [79] that enables replication and stimulates further research on the topic.

Our future work includes the comparison of machine learning techniques with both other heuristic approaches, available in literature, and other machine learning techniques (*e.g.,* Deep Learning), as well as the definition of novel methods to make machine learning and heuristic approaches more effective.

REFERENCES

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52.

[3] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[4] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, "Technical debt: Showing the way for better transfer of empirical results," in *Perspectives on the Future of Software Engineering*. Springer, 2013, pp. 179–190.

[5] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

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

[7] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[8] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells—5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[9] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, p. in press., 2019.

[10] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, 2017.

[11] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 4–15.

[12] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the 4th Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.

[13] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 106–115.

[14] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.

[15] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. IEEE, 2009, pp. 390–400.

[16] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[17] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–12.

[18] ——, "The smell of fear: On the relation between test smells and flaky tests," *Empirical Software Engineering Journal*, p. in press., 2019.

[19] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[20] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "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, 2014, pp. 101–110.

[21] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 306–315.

[22] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[23] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "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*. IEEE, 2011, pp. 181–190.

[24] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 682–691.

[25] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "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, 2016, p. 18.

[26] F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," in *Advances in Computers*. Elsevier, 2014, vol. 95, pp. 201–238.

[27] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[28] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[29] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 609–613.

[30] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[31] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[32] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[33] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[34] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and*

NEgative Results (RENE) Track. Institute of Electrical and Electronics Engineers (IEEE), 2018.

[35] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 5–14.

[36] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2018.

[37] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 176–185.

[38] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, no. 8, pp. 1144–1156, 2013.

[39] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[40] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, 2017.

[41] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.

[42] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[43] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

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

[45] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[46] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[47] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 6, 2014.

[48] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[49] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[50] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 130–141.

[51] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, ser. METRICS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 15–.

[52] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[53] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 329–331.

[54] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[55] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[56] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

[57] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 261–269.

[58] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 145–154.

[59] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 2012, pp. 466–475.

[60] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 278–281.

[61] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[62] ——, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[63] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, "Ids: An immune-inspired approach for the detection of software design smells," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 343–348.

[64] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Software maintenance and reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 248–251.

[65] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[66] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can i clone this piece of code here?" in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 170–179.

[67] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, 2015.

[68] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 396–399.

[69] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[70] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.

[71] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[72] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559 – 572, 2011, the Ninth International Conference on Quality Software.

[73] R. M. O?brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

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

[75] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[76] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1995, pp. 338–345.

[77] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[78] W. W. Cohen, "Fast effective rule induction," in *Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995, pp. 115–123.

[79] Anonymous. (2019) Comparing machine learning and heuristic approaches for metric-based code smell detection - online appendix https://figshare.com/s/9d086c96eb1e58350f3d.

[80] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2018.

[81] Z. Mahmood, D. Bowes, P. C. Lane, and T. Hall, "What is the impact of imbalance on software defect prediction performance?" in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2015, p. 4.

[82] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 309–320.

[83] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[84] M. A. Hall, "Correlation-based feature selection for machine learning," Tech. Rep., 1998.

[85] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[86] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society. Series B (Methodological)*, pp. 111–147, 1974.

[87] R. Baeza-Yates, B. d. A. N. Ribeiro *et al.*, *Modern information retrieval*. New York: ACM Press; Harlow, England: Addison-Wesley,, 2011.

[88] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[89] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE transactions on software engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[90] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[91] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[92] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 31–34.

[93] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 24, 2016.

[94] C. Simons, J. Singer, and D. R. White, "Search-based refactoring: Metrics are not enough," in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 47–61.

[95] F. A. Fontana, V. Ferme, and M. Zanoni, "Filtering code smells detection results," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 803–804.

[96] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2013.

[97] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus, "Automatic query performance assessment during the retrieval of software artifacts," in *Proceedings of the 27th IEEE/ACM international conference on Automated Software Engineering*. ACM, 2012, pp. 90–99.

[98] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[99] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "Developing fault-prediction models: What the research can show industry," *IEEE Software*, vol. 28, no. 6, pp. 96–99, 2011.