Toward a Smell-aware Bug Prediction Model

Fabio Palomba[1], Marco Zanoni[2], Francesca Arcelli Fontana[2], Andrea De Lucia[3], Rocco Oliveto[4]
[1]Delft University of Technology, The Netherlands, [2]University of Milano-Bicocca, Italy
[3]University of Salerno, Italy, [4]University of Molise, Italy
f.palomba@tudelft.nl, marco.zanoni@disco.unimib.it, arcelli@disco.unimib.it
adelucia@unisa.it, rocco.oliveto@unimol.it

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper, we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (*i.e.,* code smell intensity) by adding it to existing bug prediction models based on both product and process metrics, and comparing the results of the new model against the baseline models. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also compare the results achieved by the proposed model with the ones of an alternative technique which considers metrics about the history of code smells in files, finding that our model works generally better. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models. By evaluating the actual information gain provided by the intensity index with respect to the other metrics in the model, we found that the intensity index is a relevant feature for both product and process metrics-based models. At the same time, the metric counting the average number of code smells in previous versions of a class considered by the alternative model is also able to reduce the entropy of the model. On the basis of this result, we devise and evaluate a *smell-aware* combined bug prediction model that included product, process, and smell-related features. We demonstrate how such model classifies bug-prone code components with an F-Measure at least 13% higher than the existing state-of-the-art models.

Index Terms—Code Smells, Bug Prediction, Empirical Study, Mining Software Repositories

1 Introduction

In the real-world scenario, software systems change every day to be adapted to new requirements or to be fixed with regard to discovered bugs [1]. The need of meeting strict deadlines does not always allow developers to manage the complexity of such changes in an effective way. Indeed, often the development activities are performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing *technical debts* [2]. This phenomenon is widely known as *software aging* [3]. Some researchers measured the phenomenon in terms of entropy [4], [5], while others defined the so-called *bad code smells* (shortly "code smells" or simply "smells"), *i.e.,* recurring cases of poor design choices occurring as a consequence of aging, or when the software is not properly designed from the beginning [6]. Long or complex classes (*e.g., Blob*), poorly structured code (*e.g., Spaghetti Code*), or long *Message Chains* used to develop a certain feature are only few examples of code smells that can possibly affect a software system [6].

Besides approaches for the automatic identification of code smells in source code [7], [8], [9], [10], [11], [12], the research community devoted a lot of effort in studying the code smell lifecycle as well as in providing evidence of the negative effects of the presence of design flaws on non-functional attributes of the source code.

On the one hand, empirical studies have been conducted to understand when and why code smells appear [13], what is their evolution and longevity in software projects [14], [15], [16], [17], and to what extent they are relevant for developers [18], [19]. On the other hand, several studies showed the negative effects of code smells on software understandability [20] and maintainability [21], [22], [23], [24].

Recently, Khomh *et al.* [25] and Palomba *et al.* [26] have also empirically demonstrated that classes affected by design problems are more prone to contain bugs in the future. Although this study showed the potential importance of code smells in the context of bug prediction, these observations have been only partially explored by the research community. A prior work by Taba *et al.* [27] defined the first bug prediction model that includes code smell information. In particular, they defined three metrics, coined as *antipattern metrics*, based on the history of code smells in files and able to quantify the average number of antipatterns, the complexity of changes involving antipatterns and their recurrence length. Then, a bug prediction model exploiting antipattern measures besides structural metrics was devised and evaluated, showing that the performances of bug prediction models can increase up to 12.5% when considering design flaws.

In our preliminary study [28], we conjectured that *taking into account the severity of a design problem affecting a source code element in a bug prediction model can*

contribute to the correct classification of the bugginess of such a component. To verify this conjecture, we exploited the intensity index defined by Arcelli Fontana *et al.* [29] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluated the predictive power of the intensity index by adding it in a bug prediction model based on structural quality metrics [30], and comparing its accuracy against the one achieved by the baseline model on six large Java open source systems. We also quantified the gain provided by the addition of the intensity index with respect to the other structural metrics in the model, including the ones used to compute the intensity. The results indicated that the addition of the intensity index as predictor of buggy components has a positive impact on the accuracy of a bug prediction model based on structural quality metrics. Moreover, the results show that the intensity index is more important than other quality metrics for the prediction of the bug-proneness of smelly classes.

On the basis of the positive results achieved so far, in this paper we extend our previous analyses [28] to further investigate the properties of the intensity index in the context of bug prediction. In particular:

1) We extend the empirical validation of the *smell intensity-including* (from now on, simply *intensity-including*) bug prediction model by considering a set of 45 releases of 14 software projects. This allows to substantially increase the generalizability of the achieved results.
2) Besides evaluating the contribution of the intensity index in the context of a structural-based bug prediction model [30], we extend our analysis to consider three more baseline models, all of them relying on process metrics. Specifically, we tested the contribution of the intensity index in the Basic Code Change Model devised by Hassan [5], the Developer-based Model proposed by Ostrand *et al.* [31], and the Developer Changes Based Model defined by Di Nucci *et al.* [32], [33].
3) We perform an empirical comparison of the performances achieved by our model and by the model suggested by Taba *et al.* [27].
4) We devise and discuss the results of a *smell-aware* bug prediction model, built by combining product, process, and smell-related information.
5) We provide a comprehensive replication package [34] including all the raw data and working data sets of our study.

The results confirm that the addition of the intensity index as predictor of buggy components generally increases the performance of structural-based baseline bug prediction models, but also highlight the importance of considering the severity of code smells in process metrics-based prediction models, where we observed improvements up to 47% in terms of F-Measure. Moreover, the models exploiting the intensity index obtain performances up to 16% higher than models built with the addition of antipattern metrics [27]. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models that pushed us to investigate the possibility of a combined model including product, process, and smell-related metrics. As a result, the *smell-aware* combined model is able to provide a consistent boost in terms of prediction accuracy (*i.e.,* F-Measure) of +13% with respect to the best performing model.

Structure of the paper. Section 2 discusses the related literature on bug prediction models. Section 3 presents the specialized bug prediction model for *smelly* classes. In Section 4 we describe the design of the case study aimed at evaluating the accuracy of the proposed model, while Section 5 reports the results achieved. Section 6 discusses the threats to the validity of our empirical study. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Related Work

Although the main contribution of this paper spans in the field of bug prediction, the work is built upon previous knowledge in the field of bad code smell detection and management. For this reason, in this Section we provide an overview of the related literature in the context of both bug prediction and code smells.

2.1 Related Literature on Bug Prediction

Allocating resources for testing all the parts of a large software system is a cost-prohibitive task [35]. To alleviate this issue, the research community spent a lot of effort in the definition of approaches for making testing easier. Most of them try to identify the software code components having a higher probability to be faulty through the definition of prediction models, to allow developers to focus on that components when testing the system. Roughly speaking, a bug prediction model is a supervised method where a set of independent variables (the predictors) are used to predict the value of a dependent variable (the bug-proneness of a class) using a machine learning technique (*e.g.,* Linear Regression [36]). 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.

Several factors can influence the performances of bug prediction models. For instance, Ghotra *et al.* [37] found that the accuracy of a bug prediction model can increase or decrease up to 30% depending on the type of classification applied, while Turhan *et al.* [38] showed that the performances of cross-project models can be hindered by data heterogeneity, paving the way to new *local learning* prediction models [39]. However, the key factor to achieve good performances is represented by the choice of the predictors used in the process of bug prediction

[40]. In this sense, the existing literature mainly propose a distinction between the use of *product* metrics and *process* metrics as indicators of the bug-proneness of a code component.

The value of *product* metrics in bug prediction have been widely explored, especially in the earlier papers in the field. Basili *et al.* [41] proposed the use of the Object-Oriented metric suite (*i.e.,* CK metrics) [42] as indicators of the presence of buggy components. They demonstrated that 5 of them are actually useful in the context of bug prediction. El Emam *et al.* [43] and Subramanyam *et al.* [44] corroborate the results previously observed in [41]. On the same line, Gyimothy *et al.* [45] reported a more detailed analysis among the relationships between code metrics and the bug-proneness of code components. Their findings highlight that the Coupling Between Object metric [42] is the best metric among the CK ones in predicting defects. Ohisson *et al.* [46] conducted an empirical study aimed at evaluating to what extent code metrics are able to identify bug-prone modules. Their model has been experimented on a system developed at Ericsson and the results indicate the ability of code metrics in detecting buggy modules. Nagappan and Ball [47] exploited the use of static code analysis tools to predict the bug density of Windows Server, showing that it is possible to perform a coarse grained classification between high and low quality components with an accuracy of 83%. Nagappan *et al.* [48] also investigated the use of metrics in the prediction of buggy components across 5 Microsoft projects. Their main finding highlights that while it is possible to successfully exploit complexity metrics in bug prediction, there is no single metric that could act as a universally best bug predictor (i.e., the best predictor is project-dependent). Complexity metrics in the context of bug prediction is also the focus of the work by Zimmerman *et al.* [49], which reports a positive correlation between code complexity and bugs. Finally, Nikora *et al.* [50] showed that measurements of a system's structural evolution (e.g., number of executable statements) can serve as predictors of the number of bugs inserted into a system during its development.

On the other hand, *process* metrics have been considered in several modern approaches for defect prediction. Khoshgoftaar *et al.* [51] assessed the role played by debug churns (i.e., the number of lines of code changed to fix bugs) in the identification of bug-prone modules, while Graves *et al.* [52] experimented both product and process metrics for bug prediction. Their findings contradict in part what observed by other authors, showing that product metrics are poor predictors of bugs. D'Ambros *et al.* [53] performed an extensive comparison of bug prediction approaches relying on process and product metrics, showing that there is not a technique that works better in all contexts. Hassan and Holt [54] introduced the concept of entropy of changes as a measure of the complexity of the development process. Moser *et al.* [40] performed a comparative study between the predictive power of product and process metrics. Their study, performed on Eclipse, highlights the superiority of process metrics in predicting buggy code components. Moser *et al.* [55] also performed a deeper study on the bug prediction accuracy of process metrics, reporting that the *past number of bug-fixes performed on a file* (i.e., bug-proneness), and the *number of changes involving a file in a given period* (i.e., change-proneness) are the best predictors of buggy components. Bell *et al.* [56] confirm that the change-proneness is the best bug predictor. Hassan [5] exploits the entropy of changes to build two bug prediction models which mainly differ for the choice of the temporal interval where the bug proneness of components is studied. The results of a case study indicate that the proposed techniques have higher prediction accuracy than models purely based on the number of changes to code components. All the predictors above do not consider how many developers apply changes to a component, neither how many components they changed at the same time. Ostrand *et al.* [31], [57] propose the use of the *number of developers who modified a code component in a given time period* as a bug-proneness predictor, demonstrating that products and process metrics is poorly (positively) impacted by also considering the developers' information. Di Nucci *et al.* [32], [33] exploited the role of structural and semantic scattering of changes performed by a developer in bug prediction. Their findings demonstrate the superiority of the bug prediction model built using scattering metrics with respect to other baseline models. Moreover, they also show that the proposed metrics are orthogonal with respect to other predictors.

Finally, there are two papers closely related to the one proposed here. The first one is the study conducted by Hall *et al.* [58], which found that some code smells are correlated with the presence of faults only in some circumstances, however the effect that these smells have on faults is small. The second one is the paper by Taba *et al.* [27], that reports the use of historical metrics computed on classes affected by design flaws (called *antipattern* metrics) as additional source of information for predicting bugs. They found that such metrics can increase the performances of bug prediction models up to 12.5%. This is clearly the closest work to the one presented in this paper. Unlike that work, we propose the use of a measure of *intensity* of code smells rather than the computation of historical metrics on classes/methods affected by smells. Section 4 reports a detailed comparison between our technique and the one proposed by Taba *et al.* [27].

2.2 Related Literature on Code Smells

Bad code smells have been introduced by Fowler to define symptoms of the presence of poor design or implementation choices applied during the development of a software system [6]. They have been several times the object of empirical studies aimed at investigating their evolution and their effect on source code compre-

hension and maintenance, as well as their impact on non-functional attributes of source code such as change- and fault-proneness.

Tufano *et al.* [13], [59] conducted a large scale empirical study aimed at investigating *when* and *why* code smells are introduced. Their findings show that code smells are generally introduced during the first commit of the artifact affected by the smell, even though several instances are introduced after several maintenance operations performed on an artifact over history. Moreover, code smells are introduced because of operations aimed at implementing new features or enhancing existing ones, even if in some cases also refactoring can be the cause of smell introduction.

Other studies on the evolution of design flaws demonstrated that in most cases the number of smells in software projects tends to increase over time, and that very few code smells are removed from a project [15]. Moreover, most of the times developers are aware of the presence of code smells, but they deliberately postpone their removal [60] to avoid APIs modifications [14] or simply because developers do not perceive them as actual problems [18], [21]. Finally, a recent study [61] found significant differences in the way code smells detected using different sources of information evolve over time: specifically, developers tend to maintain and refactor more code smells identified using textual information, while design problems affected by structural issues (*e.g.*, too many dependencies between classes) are more difficult to understand and, therefore, more difficult to manage [61].

At the same time, several empirical studies showed the negative effect of code smells on program comprehension [20], as well as the impact of the interaction of more code smells on the reduction of the developers' performance during maintenance tasks [22], [23].

More important in the context of this paper is the work made by the research community to investigate the influence of code smells on change- and fault-proneness. Khomh *et al.* [24], [25] found that classes affected by code smells tend to be significantly more change- [24] and fault-prone [25] than classes not affected by design problems. Palomba *et al.* [26] confirmed such findings on a larger set of 13 code smell types, and also reported that the removal of code smells might be not always beneficial for improving source code maintainability. Also Gatrell and Counsell [62] and by Li and Shatnawi [63] confirmed the negative impact of code smells on fault-proneness; in addition, they suggested that refactoring a class, besides improving the architectural quality, reduces the probability of the class having errors in the future [62], [63].

All the reasons mentioned above pushed researchers in devising techniques for the detection of code smells in the source code. Most of these approaches aim at identifying the key *symptoms* characterizing the presence of specific code smells by using a set of thresholds based on the measurement of structural metrics (*e.g.*, if the Lines of Code $\geqslant \alpha$), and then conflating them in order to lead to the final rule for detecting a smell [9], [12], [64], [65], [66], [67], [68]. These detection techniques mainly differ in the set of used structural metrics, which depends on the type of code smells to detect, and how the identified key *symptoms* are combined. Arcelli Fontana *et al.* [69] and Aniche *et al.* [70] defined methods for discarding false positive code smell instances or tailoring the thresholds of code metrics, respectively.

In recent years, alternative sources of information have been considered for code smell detection. Ratiu *et al.* [17] propose to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Palomba *et al.* [11], [71] showed how historical data can be successfully exploited to identify smells that are intrinsically characterized by their evolution across the program history – such as *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery* – but also smells such as *Blob* and *Feature Envy* [11].

The use of Information Retrieval (IR) techniques [72] has been also exploited in order to detect code smells characterized by promiscuous responsibilities at different levels of granularity, such as *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package*, and *Misplaced Class* [10].

Arcelli Fontana *et al.* [7], [73] suggested the use of learning algorithms to discover code smells, pointing out that a training set composed of one hundred instances is sufficient to reach very high values of accuracy.

Kessentini *et al.* [74] formulated code smell detection as an optimization problem, leading to the usage of search algorithms to solve it [74], [75], [76], [77].

Finally, Morales *et al.* [78] proposed the use of developers' context as a way for improving the practical usefulness of code smell detectors, devising an approach for automatic refactoring code smells.

3 A Specialized Bug Prediction Model for Smelly Classes

Previous work has proposed the use of structural quality metrics to predict the bug-proneness of code components. The underlying idea behind these prediction models is that the presence of bugs can be predicted by analyzing the quality of source code. However, none of them take into account the presence and the severity of well-known indicators of design flaws, i.e., *code smells*, affecting the source code. In this paper, we explicitly consider this information. Indeed, we believe that a more clear description and characterization of the severity of design problems affecting a source code instance can help a machine learner in distinguishing those components having higher probability to be subject of bugs in the future. To this aim, once the set of code components affected by code smells have been detected, we build a prediction model that, in addition to relying on structural metrics, also includes the information about the severity of design problems computed using the intensity index defined by Arcelli Fontana *et al.* [29].

TABLE 1: Code Smell Detection Strategies (the complete names of the metrics are given in Table 2)

Code Smells	Detection Strategies: LABEL(n) $\rightarrow$ LABEL has value n for that smell
God Class	LOCNAMM $\geq$ HIGH(176) $\wedge$ WMCNAMM $\geq$ MEAN(22) $\wedge$ NOMNAMM $\geq$ HIGH(18) $\wedge$ TCC $\leq$ LOW(0.33) $\wedge$ ATFD $\geq$ MEAN(6)
Data Class	WMCNAMM $\leq$ LOW(14) $\wedge$ WOC $\leq$ LOW(0.33) $\wedge$ NOAM $\geq$ MEAN(4) $\wedge$ NOPA $\geq$ MEAN(3)
Brain Method	(LOC $\geq$ HIGH(33) $\wedge$ CYCLO $\geq$ HIGH(7) $\wedge$ MAXNESTING $\geq$ HIGH(6)) $\vee$ (NOLV $\geq$ MEAN(6) $\wedge$ ATLD $\geq$ MEAN(5))
Shotgun Surgery	CC $\geq$ HIGH(5) $\wedge$ CM $\geq$ HIGH(6) $\wedge$ FANOUT $\geq$ LOW(3)
Dispersed Coupling	CINT $\geq$ HIGH(8) $\wedge$ CDISP $\geq$ HIGH(0.66)
Message Chains	MaMCL $\geq$ MEAN(3) $\vee$ (NMCS $\geq$ MEAN(3) $\wedge$ MeMCL $\geq$ LOW(2))

TABLE 2: Metrics used for Code Smells Detection

Short Name	Long Name	Definition
ATFD	Access To Foreign Data	The number of attributes from unrelated classes belonging to the system, accessed directly or by invoking accessor methods.
ATLD	Access To Local Data	The number of attributes declared by the current classes accessed by the measured method directly or by invoking accessor methods.
CC	Changing Classes	The number of classes in which the methods that call the measured method are defined in.
CDISP	Coupling Dispersion	The number of classes in which the operations called from the measured operation are defined, divided by CINT.
CINT	Coupling Intensity	The number of distinct operations called by the measured operation.
CM	Changing Methods	The number of distinct methods that call the meas-ured method.
CYCLO	McCabe Cyclomatic Complexity	The maximum number of linearly independent paths in a method. A path is linear if there is no branch in the execution flow of the corresponding code.
FANOUT		Number of called classes.
LOC	Lines Of Code	The number of lines of code of an operation or of a class, including blank lines and comments.
LOCNAMM	Lines of Code Without Accessor or Mutator Methods	The number of lines of code of a class, including blank lines and comments and excluding accessor and mutator methods and corresponding comments.
MaMCL	Maximum Message Chain Length	The maximum length of chained calls in a method.
MAXNESTING	Maximum Nesting Level	The maximum nesting level of control structures within an operation.
MeMCL	Mean Message Chain Length	The average length of chained calls in a method.
NMCS	Number of Message Chain Statements	The number of different chained calls in a method.
NOAM	Number Of Accessor Methods	The number of accessor (getter and setter) methods of a class.
NOLV	Number Of Local Variables	Number of local variables declared in a method. The method's parameters are considered local variables.
NOMNAMM	Number of Not Accessor or Mutator Methods	The number of methods defined locally in a class, counting public as well as private methods, exclud-ing accessor or mutator methods.
NOPA	Number Of Public Attributes	The number of public attributes of a class.
TCC	Tight Class Cohesion	The normalized ratio between the number of methods directly connected with other methods through an instance variable and the total number of possible connections between methods. A direct connection between two methods exists if both access the same instance variable directly or indirectly through a method call. TCC takes its value in the range [0,1].
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods	The sum of complexity of the methods that are defined in the class, and are not accessor or mutator methods. We compute the complexity with the Cyclomatic Complexity metric (CYCLO).
WOC	Weight Of Class	The number of "functional" (*i.e.,* non-abstract, non-accessor, non-mutator) public methods divided by the total number of public members.

Specifically, the index is computed by *JCodeOdor*[1], a code smell detector which relies on detection strategies applied on metrics. The tool is able to detect, filter [79] and prioritize [29] instances of six types of code smells [6], [66]:

- God Class: A large class implementing different responsibilities;
- Data Class: A class whose only purpose is holding data;
- Brain Method: A large method that implements more than one function;
- Shotgun Surgery: A class where every change triggers many little changes to several other classes;
- Dispersed Coupling: A class having too many relationships with other classes;
- Message Chains: A method containing a long chain of method calls.

The intensity index is an estimation of the severity of a code smell, and its value is defined as a real number in the range [1,10]. In particular, given the set of classes composing the software system that a developer wants to evaluate, *JCodeOdor* adopts the following two steps to compute the intensity of code smells:

1) In the first step the tool aims at detecting code

1. tool available at http://essere.disco.unimib.it/wiki/jcodeodor

smells in the system given as input, relying on the detection strategies reported in Table 1. Each detection strategy is a logical composition of predicates, and each predicate is based on an operator that compares a metric with a threshold [66], [80]. Our detection strategies are similar to those defined by Lanza and Marinescu [66], which adopted the metrics reported in Table 2 to detect the six code smells described above. More specifically, Lanza and Marinescu [66] observed that code smells often exhibit (i) low cohesion and high coupling, (ii) high complexity, and (iii) extensive access to the data of foreign classes: for this reason, our approach considers (i) cohesion (*i.e.,* TCC) and coupling (*i.e.,* CC, CDISP, CINT, CM, FANOUT), (ii) complexity (*i.e.,* CYCLO, MaMCL, MAXNESTING, MeMCL, NMCS, WMCNAMM, WOC), and (iii) data access (*i.e.,* ATFD and ATLD) metrics. Furthermore, the approach also computes size-related metrics such as LOC, LOCNAMM, NOAM, NOLV, NOMNAMM, and NOPA. To ease the comprehension of the detection approach, Table 2 reports the full metric names and definitions, while Table 3 describes the rationale behind the use of each predicate of the detection strategies. Moreover, in Table 4 we provide data on the distribution of the metrics used for code smell detection on the dataset exploited in this paper (more details on the systems and their selection are provided in Section 4).

Following the detection rules, a code component is detected as *smelly* if one of the logical propositions shown in Table 1 is true, namely if the actual metrics of the code component exceed the threshold values composing a detection strategy. It is important to note that the thresholds used by the tool have been empirically calibrated on 74 systems of the Qualitas Corpus dataset [81] and are derived from the statistical distribution of the metrics contained in the dataset [82]. For metrics representing ratios defined in the range [0,1] (*e.g.,* the Tight Class Cohesion), threshold values are fixed to 0.25, 0.33, 0.5, 0.66 and 0.75. For all other metrics, they are associated to percentile values on the metric distribution [82]. For sake of completeness, we report in Table 5 all the threshold values associated to each of the detected code smells. The thresholds are also mapped by the tool onto a nominal value, *i.e.,* VERY-LOW, LOW, MEAN, HIGH, VERY-HIGH, to ease their interpretation.

2) If a code component is detected as a code smell, the actual value of a given metric used for the detection will exceed the threshold value, and it will correspond to a percentile value on the metric distribution placed between the threshold and the maximum observed value of the metric in the system under analysis. The placement of the actual metric value in that range represents the "exceeding amount" of a metric with respect to the defined threshold. Such "exceeding amounts" are then normalized in the range [1,10] using a min-max normalization process [83]: specifically, this is a feature scaling technique where the values of a numeric range are reduced to a scale between 1 and 10. To compute z, *i.e.,* the normalized value, the following formula is applied:

$$z = [\frac{x - min(x)}{max(x) - min(x)}] \cdot 10 \quad (1)$$

where min and max are the minimum and maximum values observed in the distribution. This step allows to have the "exceeding amount" of each metric in the same scale. To have a unique value representing the intensity of the code smell affecting the class, the mean of the normalized "exceeding amounts" is computed.

When considered as bug predictor, the intensity has two relevant properties: (i) its value is derived from a set of other metric values, and (ii) since it relies on the statistical distribution of metrics, it can be seen as a non-linear combination of their values.

We include the intensity index as an additional predictor of a structural and process metrics-based bug prediction model. It is important to note that we cannot use the intensity index as single predictor, since this choice might lead to two important limitations. On the one hand, we would take into account only the information about the smelliness of classes, missing other pieces of information useful when classifying buggy classes: indeed, other metrics used by structural and process metrics-based bug prediction models might provide important contributions in the classification [84]. Thus, in case of a prediction model only based on the intensity index, the additional information would be lost. On the other hand, a model exploiting the information given by the intensity index in isolation would have been not enough accurate when classifying non-smelly classes. In these cases, the intensity index is equal to zero, thus not providing any detailed information that the prediction model may use to learn the characteristics of non-smelly classes. These observations are also supported by the results achieved when testing the performances of the prediction model built only using the intensity index on the dataset used in the study, where we observed low performances. Detailed results are reported in our online appendix [34]. As a consequence, to build the proposed bug prediction model we firstly split the training set by considering *smelly* (as identified by the code smell detector) and *non-smelly* classes. We then assign to *smelly* classes an intensity index according to the evaluation performed by *JCodeOdor*, while we set the intensity of *non-smelly* classes to 0. In case a certain class is affected by more than one smell, we assign to it the maximum intensity computed by the tool: the rationale behind this choice is that the most severe code smell affecting the class is the one that impacts more the maintainability

TABLE 3: Code Smell Detection Rationale and Details

	Clause	Rationale
God Class	LOCNAMM $\geq$ HIGH	*Too much code.* We use LOCNAMM instead of LOC, because getter and setter methods are often generated by the IDE. A class that has getter and setter methods, and a class that has not getter and setter methods, must have the same "probability" to be detected as God Class.
	WMCNAMM $\geq$ MEAN	*Too much work and complex.* Each method has a minimum cyclomatic complexity of one, hence also getter and setter add cyclomatic complexity to the class. We decide to use a complexity metric that exclude them from the computation.
	NOMNAMM $\geq$ HIGH	*Implements a high number of functionalities.* We exclude getter and setter because we consider only the methods that effectively implement functionality of the class.
	TCC $\leq$ LOW	*Functionalities accomplish different tasks.*
	ATFD $\geq$ MEAN	*Uses many data from other classes.*
Data Class	WMCNAMM $\leq$ LOW	*Methods are not complex.* Each method has a minimum cyclomatic complexity of one, hence also getter and setter add cyclomatic complexity to the class. We decide to use a complexity metric that exclude them from the computation.
	WOC $\leq$ LOW	*The class offers few functionalities.* This metrics is computed as the number of functional (non-accessor) public methods, divided by the total number of public methods. A low value for the WOC metric means that the class offers few functionalities.
	NOAM $\geq$ MEAN	*The class has many accessor methods.*
	NOPA $\geq$ MEAN	*The class has many public attributes.*
Brain Method	LOC $\geq$ HIGH	*Too much code.*
	CYCLO $\geq$ HIGH	*High functional complexity*
	MAXNESTING $\geq$ HIGH	*High functional complexity. Difficult to understand.*
	NOLV $\geq$ MEAN	*Difficult to understand.* More the number of local variable, more the method is difficult to understand.
	ATLD $\geq$ MEAN	*Uses many of the data of the class.* More the number of attributes of the class the method uses, more the method is difficult to understand.
Shot. Surg.	CC $\geq$ HIGH	*Many classes call the method.*
	CM $\geq$ HIGH	*Many methods to change.*
	FANOUT $\geq$ LOW	*The method is subject to being changed.* If a method interacts with other classes, it is not a trivial one. We use the FANOUT metric to refer Shotgun Surgery only to those methods that are more subject to be changed. We exclude for example most of the getter and setter methods.
Dis. Coup.	CINT $\geq$ HIGH	*The method calls too many other methods.* With CINT metric, we measure the number of distinct methods called from the measured method.
	CDISP $\geq$ HIGH	*Calls are dispersed in many classes.* With CDISP metric, we measure the dispersion of called methods: the number of classes in which the methods called from the measured method are defined in, divided by CINT.
Mess. Chain	MaMCL $\geq$ MEAN	*Maximum Message Chain Length.* A Message Chains has a minimum length of two chained calls, because a single call is trivial. We use the MaMCL metric to find out the methods that have at least one chained call with a length greater than the mean.
	NMCS $\geq$ MEAN	*Number of Message Chain Statements.* There can be more Message Chain Statement: different chains of call. More the number of Message Chain Statements, more the method is interesting respect to Message Chains code smell.
	MeMCL $\geq$ LOW	*Mean of Message Chain Length.* We would find out non-trivial Message Chains, so we need always to check against the Message Chain Statement length.

of the class [85]. Finally, we build a prediction model using as predictors the intensity index and a set of other product/process metrics.

4 Empirical Study Definition and Design

The *goal* of the empirical study is to evaluate the contribution of the *intensity* index in prediction models aimed at discovering bug-prone code components, with the *purpose* of improving the allocation of resources in the verification & validation activities focusing on components having a higher bug-proneness. The *quality focus* is on the prediction performances as compared to the state-of-the-art approaches, while the *perspective* is of researchers, who want to evaluate the effectiveness of using information about code smells when identifying bug-prone components. More specifically, the empirical investigation aims at answering the following research questions:

RQ1: *To what extent does the intensity index contribute to the prediction of bug-prone code components?*

RQ2: *How the proposed specialized model works when compared to a state-of-the-art model built using antipattern metrics?*

RQ3: *What is the gain provided by the intensity index to the bug prediction model when compared to the other predictors?*

RQ4: *What are the performances of a combined bug prediction model that includes smell-related information?*

The first research question (**RQ1**) aims at providing information about the actual contribution given by the intensity index within bug prediction models built using different types of information, *i.e.,* exploiting product and process metrics. With **RQ2** we are interested in comparing our solution with the one proposed by Taba *et al.* [27], who defined the so-called *antipattern metrics*

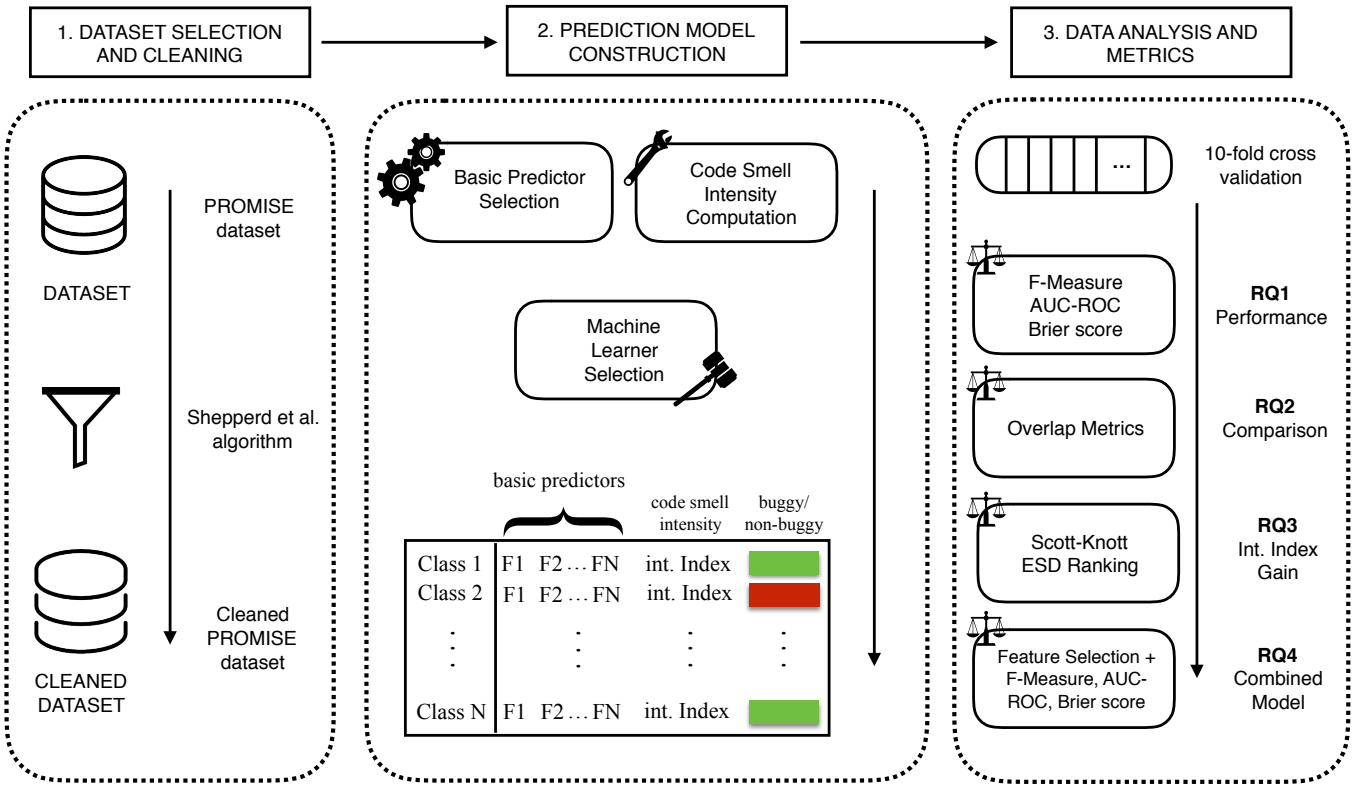


Fig. 1: Overview of the Empirical Study Design.

TABLE 4: Distribution of Metrics used for Code Smells Detection

Metric	Min	1st quart.	Median	Mean	3rd quart.	Max
ATFD	0	0	0	4	0	234
*ATLD	0	0	1	12	1	275
CC	0	0	0	0	8	293
CDISP	0	0	0	0.32	0.67	1
CINT	0	0	0	15	1	280
CM	0	0	0	5	1	497
CYCLO	0	1	1	1	2	415
FANOUT	0	0	0	5	1	280
LOC	0	15	44	51	113	6769
*LOCNAMM	0	3	5	41	15	6989
*MaMCL	0	0	0	3	0	5
MAXNESTING	0	0	0	2	3	9
*MeMCL	0	0	0	3	0	7
*NMCS	0	0	0	33	1	501
NOAM	0	0	1	9	2	79
NOLV	0	1	3	5	6	411
*NOMNAMM	0	0	0	11	0	274
NOPA	0	0	0	3	5	34
TCC	0	0	0.43	0.47	1	1
*WMCNAMM	0	2	6	8	14	3457
WOC	0	0.36	0.88	0.82	1	1

TABLE 5: Default thresholds for all smells

	Metric	VERY-LOW	LOW	MEAN	HIGH	VERY-HIGH
God Class	LOCNAMM	26	38	78	176	393
	WMCNAMM	11	14	22	41	81
	NOMNAMM	7	9	13	21	30
	TCC	0.25	0.33	0.5	0.66	0.75
	ATFD	3	4	6	11	21
Data Class	WMCNAMM	11	14	21	40	81
	WOC	0.25	0.33	0.5	0.66	0.75
	NOPA	1	2	3	5	12
	NOAM	2	3	4	7	13
Brain Method	LOC	11	13	19	33	59
	CYCLO	3	4	5	7	13
	MAXNESTING	3	4	5	6	7
	NOLV	4	5	6	8	12
	ATLD	3	4	5	6	11
Shotgun Surgery	CC	2	3	4	5	10
	CM	2	3	4	6	13
	FANOUT	2	3	4	5	6
Disp. Coup.	CINT	3	4	5	8	12
	CDISP	0.25	0.33	0.5	0.66	0.75
Message Chains	MaMCL	2	3	3	4	7
	MeMCL	2	2	3	4	5
	NMCS	1	2	3	4	5

to improve the performances of bug prediction models. **RQ3** is concerned with a fine-grained analysis aimed at measuring the actual gain provided by the addition of the intensity metric within different bug prediction models. Finally, **RQ4** has the goal to assess the performances of a smell-aware combined bug prediction model built mixing together the features exploited by the experimented models. Figure 1 overviews the main steps performed to conduct the empirical study, *i.e.*, (i) dataset selection and cleaning, (ii) prediction model building, and (iii) data analysis to answer our four research questions. The following subsections detail each

TABLE 6: Software Projects in Our Dataset

System	Releases	Classes	KLOCs	% Buggy Cl.	% Smelly Cl.	EPV
Apache Ant	5	83-813	20-204	68-72	11-16	12-15
Apache Camel	4	315-571	70-108	30-38	9-14	16-21
Apache Forrest	3	112-628	18-193	37-39	11-13	14-17
Apache Ivy	1	349	58	29	12	14
JEdit	5	228-520	39-166	36-43	14-22	11-12
Apache Velocity	3	229-341	57-73	15-23	7-13	16-18
Apache Tomcat	1	858	301	6	4	15
Apache Lucene	3	338-2,246	103-466	59-63	10-22	11-18
Apache Pbeans	2	121-509	13-55	29-33	21-25	14-16
Apache POI	4	129-278	68-124	62-68	15-19	15-22
Apache Synapse	3	249-317	117-136	17-26	13-17	14-19

of these three steps.

4.1 Dataset Selection and Cleaning

The *context* of the study consists of 34 releases of 11 open source software systems. Table 6 reports (i) the analyzed software systems, (ii) the number of releases considered for each of them, (iii) their size (min-max) in terms of minimum and maximum number of classes and KLOCs across the considered releases, (iv) the percentage (min-max) of buggy files (identified as explained later), and (iv) the percentage (min-max) of classes affected by design problems (detected as explained later). In addition, we also report the number of events per variables (EPV), *i.e.,* the ratio between the number of occurrences of the least frequently occurring class of the dependent variable and the number of independent variables used to train the model. The selection of the dataset was driven by three main factors, as suggested by previous work [86]: in the first place, we only focused on publicly available datasets reporting a large set of bugs and providing oracles for all the projects in the study. To this aim, we started from the dataset by Jureczko *et al.* [30] contained in the PROMISE repository [87] because it provides a rich collection of 44 releases of 14 projects for which 20 code metrics as well as bugs occurring in each release are available. Furthermore, it is important to note that the dataset contains systems having different size and scope, allowing us to increase the validity of our study [37], [88]. In the second place, we took into account the findings by Mende *et al.* [89], who reported that models trained using small datasets may produce unstable performance estimates. To estimate how good a bug prediction dataset is, Tantithamthavorn *et al.* [90] suggested the use of the number of EPV. In particular, datasets having EPVs lower than 10 are particularly susceptible to unstable results. Thus, from the initial dataset we removed an entire system (*i.e.,* APACHE XERCES) composed of 3 releases. Finally, to ensure data robustness [91] we discarded 7 releases of two systems (*i.e.,* APACHE XALAN and APACHE LOG4J) having a rate of buggy classes higher than 75%, leading to the final dataset composed of 34 releases of 11 systems.

Once defined the context of our study, we performed a data preprocessing phase. Specifically, as reported in previous research [90], [92] bug prediction datasets may contain noise and/or erroneous entries that possibly negatively influence the results. To ensure the quality of the data, we performed a data cleaning following the algorithm proposed by Shepperd *et al.* [92]: it includes a list of 13 corrections aimed at removing identical features, features with conflicting values or missing values *etc.*. This step lead to the definition of a cleaned dataset where a total of 58 entries were removed from the original one. It is worth remarking that the data and scripts used in the study are publicly available in our online appendix [34].

4.2 Prediction Model Construction

To answer our research questions, we needed to instantiate the prediction model presented in Section 3 to define (i) the basic predictors, (ii) the code smell detection process, and (iii) the machine learning technique to use for classifying buggy instances.

4.2.1 Basic Predictors

As for the software metrics to use as basic predictors in the model, the related literature proposes several alternatives, with a main distinction between *product* metrics (*e.g.,* lines of code, code complexity, etc) and *process* metrics (e.g., past changes and bug fixes performed on a code component). To have a detailed overview of the predictive power of the intensity index when applied in different contexts, we decided to test its contribution in prediction models using both product and process metrics as basic predictors.

To this aim, we firstly set up a bug prediction model composed of structural predictors, and in particular the 20 quality metrics exploited by Jureczko *et al.* [30]. The model is characterized by a mix of size metrics (*e.g.,* Lines of Code), coupling metrics (*e.g.,* Coupling Between Object Classes [42]), cohesion metrics (*e.g.,* Lack of Cohesion of Methods [42]), and complexity metrics (*e.g.,* McCabe Complexity [93]). In this case, the choice of the baseline was guided by the will to investigate whether the use of a single additional structural metric representing the intensity of code smells is able to add useful information in a prediction model already characterized by structural predictors, as well as by the set of code metrics used for the computation of the intensity index. It is important to note that this model might be affected by multi-collinearity [94], which occurs when two or more independent variables are highly correlated and can be predicted one from the other. Recent work [95], [96] showed that highly-correlated variables represent a problem for bug prediction models since they can create interferences to the analysis of the importance of variables, thus possibly leading to a decreasing of the prediction capabilities of the resulting model. We assessed the model for the presence of multi-collinearity in two different ways:

- Given the metrics composing each of the analyzed systems, we computed the Spearman's rank correlation [97] between all possible pairs of metrics,

to determine whether there are pairs of strongly correlated metrics (*i.e.,* with a Spearman's $\rho > 0.8$). In particular, this is a non-parametric measure of the statistical dependence between the ranking of two variables. If two independent variables are highly correlated, one of them should be removed from the model;
- We used a stepwise variable removal procedure based on the Companion Applied Regression (*car*) R package[2], and in particular based on the *vif* (variance inflation factors) function [94]. Basically, this function provides an index for each independent variable which measures how much the variance of an estimated regression coefficient is increased because of collinearity. The square root of the variance inflation factor indicates how much larger the standard error is, compared with what it would be if that variable were uncorrelated with the other predictor variables in the model. Based on this information, we could understand which metric produced the largest standard error, thus allowing the identification of the metric that was better to drop from the model.

We also set up three baseline prediction models based on process metrics. We used (i) the Basic Code Change Model (BCCM) proposed by Hassan [5] which uses the entropy of changes of a given time period to predict defects, (ii) the model based on the number of developers that worked on a code component (DM) in a specific time period [31], and (iii) the Developer Changes Based Model (DCBM) [33] which considers how scattered are the changes applied by developers that worked on a code component in a given time window. While a number of other process metrics as well as prediction approaches relying on such metrics have been defined in the literature [98], the selected models have different characteristics that allowed us to evaluate the contribution of the intensity index from several perspectives, *i.e.,* when the entropy of the development process is considered or in case of different developer-related measurements.

Note that all the process metrics-based prediction models described above refer to a specific time period in which these metrics have to be computed. In our case, the information about entropy of changes, number of developers, and scattering metrics that are related to the specific release R of a software in our dataset refer to the time window between the previous release $R-1$ and R.

To measure the extent to which the contribution of the *intensity* index is useful for predicting bugs, we experimented (i) the baseline models described above and (ii) the same baseline models where the intensity index is plugged as additional metric. For instance, we test the model based on the 16 software metrics and the model composed of the 16 software metrics plus the intensity index. It is worth remarking that, for *non-smelly* classes, the intensity value is set to 0. Applying this procedure, we were able to *control* the effective contribution of the index during the prediction of bugs.

2. http://cran.r-project.org/web/packages/car/index.html

4.2.2 Code Smell Detection

Regarding the code smell detection process, our study was focused on the analysis of the code smells for which an intensity index has been defined (see Section 3). The related literature offers a large amount of code smell detectors [99], however all such tools classify classes strictly as being or not affected by a code smell, thus not computing a degree of intensity of code smells. At the same time, some other code smell prioritization approaches have been proposed [67], [68], but unfortunately they cannot handle all the code smells considered in this study. As an example, the Bayesian technique proposed by Khomh *et al.* [67] assigns a probability that a certain class is affected by the God Class code smell, while it has not been defined for other smell types. For this reason, we relied on the detection performed by *JCodeOdor* [29], because (i) it has been empirically validated demonstrating good performances in detecting code smells (see Section 6), and (ii) it detects all the code smells considered in the empirical study. Finally, it computes the value of the intensity index on the detected code smells.

To build a bug prediction model that discriminates actual *smelly* and *non-smelly* classes, we decided to discard the false positive instances from the set of candidate code smells given by the detection tool. To discard such instances we compared the results of the tool against an annotated set of code smell instances publicly available [100]. Specifically, we set the intensity of a class equals to 0 in case such a class represents a false positive with respect to all the considered code smells. Once concluded this process, we trained the prediction model taking into account the actually smelly classes only. Note that to ensure a fair comparison, we discarded false positive classes from all the other experimented baselines, so that all of them worked on the same dataset.

It is worth observing that the best solution would be that of considering all the actual smell instances in a software project (*i.e.,* the *golden set*). However, the smell instances which are not detected by *JCodeOdor* (*i.e.,* false negatives) do not exceed the structural metric thresholds that allow the tool to detect and assign them an intensity value. As a consequence, the intensity index assigned to these instances would be equal to zero, and still have no effect on the prediction model. While we can fix the results of *JCodeOdor* in the case of false positives (by setting the intensity index to zero), we cannot assign an intensity index to false negatives. For this reason, we discarded them from the training of the model. The effect of including false positive and false negative instances in the training of the model is discussed in Section 6.

4.2.3 Machine Learning Technique

The final step was the definition of the machine learning classifier to use. We experimented several classifiers, namely Multilayer Perceptron [101], ADTree [102], Naive Bayes [103], Logistic Regression [104], Decision Table Majority [105], and Simple Logistic [36]. It is worth noting that most of the classifiers do not output a boolean value indicating the presence/absence of a bug, but rather a probability that a certain class is buggy or not. While we are aware of the possible impact of the cut-off selection on the performance of the classifier, finding the ideal settings in the parameter space of a single classification technique would be prohibitively expensive [106]. For this reason, we decided to test the different classification techniques using the default setting, *i.e.,* a class is buggy if the probability found by the classifier is higher than 0.5, non-buggy otherwise.

We empirically compared the results achieved by the prediction model on the software systems used in our study (more details on the adopted procedure later in this section). A complete comparison among the experimented classifiers can be found in our online appendix [34]. Over all the systems, the best results on the baseline model were obtained using the Simple Logistic, confirming previous findings in the field [37], [53]. Thus, in this paper we report the results of the models built with this classifier. Simple Logistic uses a statistical technique based on a probability model. Indeed, instead of simple classification, the probability model gives the probability of an instance belonging to each individual class (*i.e.,* buggy or not), describing the relationship between a categorical outcome (*i.e.,* buggy or not) and one or more predictors [36].

4.3 Data Analysis and Metrics

Once the model has been instantiated, we validated its performance and answered our research questions as explained in the following.

4.3.1 Validation Methodology

To assess the performance of the model we adopted the 10-fold cross-validation strategy [107]. This strategy randomly partitions the original set of data into 10 equal sized subsets. Of the 10 subsets, one is retained as *test* set, while the remaining 9 are used as *training* set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the *test* set exactly once [107]. We used this test strategy since it allows all observations to be used for both training and test purpose, but also because it has been widely-used in the context of bug prediction (*e.g.,* see [31], [108], [109], [110]).

4.3.2 RQ1 - The contribution of the Intensity Index

To answer **RQ1** we computed two widely-adopted metrics in bug prediction, namely precision and recall [72]:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + TN} \tag{2}$$

where TP is the number of classes containing bugs that are correctly classified as bug-prone; TN denotes the number of bug-free classes classified as non bug-prone classes; and FP measures the number of classes for which a prediction model fails to identify bug-prone classes by declaring bug-free classes as bug-prone. Along with precision and recall we computed the F-Measure, defined as the harmonic mean of precision and recall:

$$F\text{-}Measure = 2 * \frac{precision * recall}{precision + recall} \tag{3}$$

Furthermore, we reported the Area Under the Curve (AUC) obtained by the prediction model. The AUC quantifies the overall ability of a prediction model to discriminate between buggy and non-buggy classes. The closer the AUC to 1, the higher the ability of the classifier to discriminate classes affected and not by a bug. On the other hand, the closer the AUC to 0.5, the lower the accuracy of the classifier. In addition, we also computed the Brier score [111], [112], a metric previously employed to interpret the results of machine learning models in software engineering [90], [113] that measures the distance between the probabilities predicted by a model and the actual outcome. Formally, the Brier score is computed as follow:

$$Brier\text{-}score = \frac{1}{N}\sum_{i=1}^{N}(p_c - o_c) \tag{4}$$

where p_c is the probability predicted by the model on a class c, o_c is the actual outcome for class c, *i.e.,* 0 if the class is bug-free and 1 if it is buggy, and N is the total number of classes in a dataset. Low Brier scores indicate good classifier performances, while high scores indicate low performances.

Besides the analysis of the performance of the specialized bug prediction model and its comparison with the baseline model, we also investigated the behavior of the experimented models in the classification of *smelly* and *non-smelly* instances. Specifically, we computed the percentage of *smelly* and *non-smelly* classes correctly classified by each of the prediction models, to evaluate whether the *intensity-including* model is actually able to give a contribution in the classification of classes affected by a code smell, or whether the addition of the intensity index also affects the classification of smell-free classes.

In addition, we statistically compared the performances achieved by the experimented prediction models. While the use of the Mann-Whitney test [114] is widely spread in the literature [84], it is not recommended when comparing the prediction capabilities of more models over multiple datasets since the performances of a machine learner can variate between a dataset and another [37], [115]. For this reason, we compared the AUC performance distribution of the models using the Scott-Knott Effect Size Difference (ESD) test [90], [91]. It represents an effect-size aware variant of the original Scott-Knott test [116] that (i) uses hierarchical

cluster analysis to partition the set of treatment means into statistically distinct groups, (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. To measure the effect size, the tests uses the Cliff's Delta (or d) [117]. In the context of our study, we employed the `ScottKnottESD` implementation[3] provided by Tantithamthavorn *et al.* [90]. The rationale behind the usage of this test was that the Scott-Knott ESD can be adopted to control dataset-specific performances: indeed, it evaluates the performances of the different prediction models on each dataset in isolation, thus ranking the top models based on their performances on each project. For this reason, we had 34 different Scott-Knott ranks that we analyzed by measuring the likelihood of a model to be in the top Scott-Knott ESD rank, as done in previous work [90], [118], [119].

4.3.3 RQ2 - Comparison between Intensity Index and Antipattern Metrics

The goal of **RQ2** is to compare the performances of prediction models based on the intensity index with the performances that is possible to achieve using other existing metrics which take into account smell-related information. To perform this comparison, we used the *antipattern* metrics defined by Taba *et al.* [27], *i.e.,* the Average Number of Antipattern (ANA) in previous buggy versions of a class, the Antipattern Complexity Metric (ACM) computed using the entropy of changes involving smelly classes, and the Antipattern Recurrence Length (ARL) that measures the total number of releases in which a class has been affected by a smell. To compute the metrics, we first manually detected the public releases of the software projects considered in the study. Note that this step was required because our dataset does not include all the releases of the software systems.

Secondly, we used our `ChangeHistoryMiner` tool [13] to (i) download the source code of each release R of a software project p_i, (ii) detect code smell instances present in R, and (iii) compute the *antipattern* metrics. As done for the evaluation of the contribution of the intensity index, also in this case we plugged the *antipattern* metrics into the product- and process-based baseline models.

We compared the performances of the resulting models with the ones achieved by the model built using the intensity index using the same set of accuracy metrics (*i.e.,* precision, recall, F-Measure, AUC-ROC, and Brier score). At the same time, we also statistically compared the models using the Scott-Knott ESD test. Finally, we also analyzed to what extent the two models are complementary in the classification of the bugginess of classes affected by code smells. Specifically, let m_{int} be the model built plugging in the intensity index; let m_{ant} be the model built by considering the antipattern metrics, we computed the following overlap metrics on the set of *buggy and smelly* instances of each system:

$$TP_{m_{int} \cap m_{ant}} = \frac{|TP_{m_{int}} \cap TP_{m_{ant}}|}{|TP_{m_{int}} \cup TP_{m_{ant}}|}\% \quad (5)$$

$$TP_{m_{int} \setminus m_{ant}} = \frac{|TP_{m_{int}} \setminus TP_{m_{ant}}|}{|TP_{m_{int}} \cup TP_{m_{ant}}|}\% \quad (6)$$

$$TP_{m_{ant} \setminus m_{int}} = \frac{|TP_{m_{ant}} \setminus TP_{m_{int}}|}{|TP_{m_{ant}} \cup TP_{m_{int}}|}\% \quad (7)$$

where $TP_{m_{int}}$ represents the set of bug-prone classes correctly classified by the prediction model m_{int}, while $TP_{m_{ant}}$ is the set of bug-prone classes correctly classified by the prediction model m_{ant}. The $TP_{m_{int} \cap m_{ant}}$ metric measures the overlap between the sets of true positives correctly identified by both models m_{int} and m_{ant}, $TP_{m_{int} \setminus m_{ant}}$ measures the percentage of bug-prone classes correctly classified by m_{int} only and missed by m_{ant}, and $TP_{m_{ant} \setminus m_{int}}$ measures the percentage of bug-prone classes correctly classified by m_{ant} only and missed by m_{int}.

4.3.4 RQ3 - Gain Provided by the Intensity Index

As for **RQ3**, we conducted a *fine-grained* investigation aimed at measuring how important is the intensity index with respect to the other features (*i.e.,* product, process, and *antipattern* metrics) composing the experimented models. In particular, we used an *information gain* algorithm [120] to quantify the gain provided by adding the intensity index in each prediction model. Formally, let M be a bug prediction model, let $P = \{p_1, \ldots, p_n\}$ be the set of predictors composing M, an *information gain* algorithm [120] applies the following formula to compute a measure which defines the difference in entropy from before to after the set M is split on an attribute p_1:

$$InfoGain(M, p_i) = H(M) - H(M|p_i) \quad (8)$$

where the function $H(M)$ indicates the entropy of the model that includes the predictor p_i, while the function $H(M|p_i)$ measures the entropy of the model that does not include p_i. Entropy is computed as follow:

$$H(M) = -\sum_{i=1}^{n} prob(p_i) \log_2 prob(p_i) \quad (9)$$

In other words, the algorithm quantifies how much uncertainty in M was reduced after splitting M on predictor p_1. In the context of our work, we applied the *Gain Ratio Feature Evaluation* algorithm [120] implemented in the WEKA toolkit [121], which ranks $p_1, \ldots, p_n$ in descending order based on the contribution provided by p_i to the decisions made by M. In particular, the output of the algorithm is a ranked list in which the predictors having the higher expected reduction in entropy are placed on the top. Using this procedure, we evaluated the relevance of the predictors in the prediction model, possibly

3. https://github.com/klainfo/ScottKnottESD

understanding whether the addition of the intensity index gives a higher contribution with respect to the structural metrics from which it is derived (*i.e.,* metrics used for the detection of the smells) or with respect the other metrics contained in the models. During this step, we also verified—through the `evaluateAttribute` function of the WEKA implementation of the algorithm—whether a certain predictor mainly contributes to the identification of buggy or non-buggy classes, *i.e.,* if there exists a positive or negative relationship between the predictor and the bug-proneness of classes. The algorithm was applied on each system of the dataset and for each set of predictors considered (*i.e.,* based on structural metrics [30], entropy of changes [5], number of developer [56], scattering metrics [32], [33], and antipattern metrics [27]), and for this reason we had to analyze 34 ranks for each basic model. Therefore, as suggested by previous work [122], [123], [124] we adopted again the Scott-Knott ESD test [90], which in this case had the goal to find statistically significant relevant features composing the models.

4.3.5 RQ4 - Combining Basic Predictors and Smell-related Metrics

Until now, we assessed the contribution of the intensity index in prediction models based on different sets of metrics without considering a combination of product and process predictors. In **RQ4** our goal is to find a combined set of metrics that uses smell-related information together with product and process metrics to achieve better performances. To build the smell-aware combined model, we defined the following process:

- The metrics belonging to the previously tested models, *i.e.,* the 16 structural metrics [30], the entropy of changes [5], the number of developers [31], the scattering metrics [33], the *antipattern* metrics [27], and the intensity index [29] are put all together in a single dataset.
- To select the variables actually relevant to predict bugs and avoid multicollinearity [94], we applied the variable removal procedure based on the *vif* function described for **RQ1**.
- Finally, we tested the performances of the combined model using the same procedures and metrics used in the context of **RQ1**, *i.e.,* precision, recall, F-measure, AUC-ROC, and Brier score. At the same time, we statistically compared the performances of the experimented models by means of Scott-Knott ESD test.

5 ANALYSIS OF THE RESULTS

In the following we discuss the results, aiming at providing an answer to our research questions. To avoid redundancies, we discuss the first two research questions together.

5.1 The performances of the proposed model and its comparison with the state-of-the-art

Before describing the results related to the addition of the intensity index in the different prediction models considered, it is worth reporting the output of the feature selection process aimed at avoiding multi-collinearity by removing irrelevant features from the structural model. In particular, for each considered project we discovered a recurrent pattern in the pairs of metrics highly correlated:

1) Weighted Method per Class (WMC) and Response for a Class (RFC);
2) Coupling Between Objects (CBO) and Afferent Couplings (CA);
3) Lack of Cohesion of Methods (LCOM) and Lack of Cohesion of Methods 3 (LCOM3);
4) Maximum Cyclomatic Complexity (MAX(CC)) and Average Cyclomatic Complexity (AVG(CC));

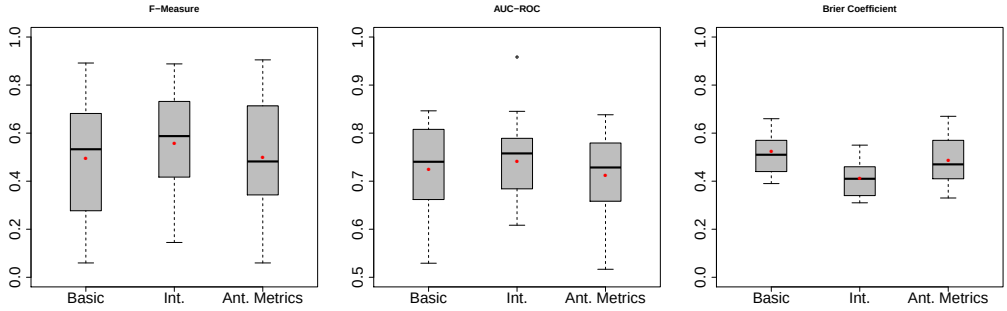
According to the results achieved using the *vif* function [94], we removed the RFC, CA, LCOM, and MAX(CC) metrics. Therefore, the resulting structural model is composed of 16 metrics.

Figure 2 depicts the box plots reporting the distributions of F-Measure, AUC-ROC, and Brier score achieved by the experimented models on the systems in our datasets. For sake of readability, we decided to report only the distribution of the harmonic mean of precision and recall (*i.e.,* F-Measure) rather than also reporting the distributions of precision and recall. Detailed results for these two metrics can be found in our online appendix [34]. Figure 2 reports the performances of the (i) basic prediction model (label "Basic"), (ii) intensity-including prediction models (label "Int."), and (iii) antipattern metrics-including prediction models (label "Ant. Metrics") built using different basic predictors, *i.e.,* structural metrics in Figure 2a, entropy of changes in Figure 2b, number of developers in Figure 2c, and scattering metrics in Figure 2d. Furthermore, Table 7 reports the average percentages of *smelly* and *non-smelly* classes correctly classified (with respect to bugginess) by each of the analyzed models.

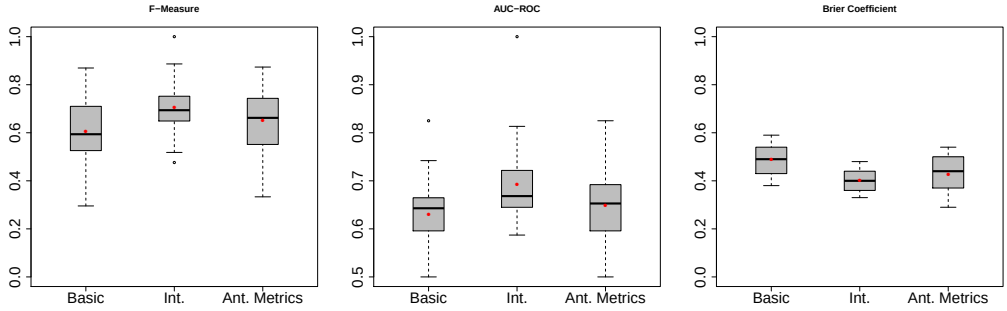
Looking at Figure 2, models based on entropy of changes and scattering metrics tend to perform generally better than models built using structural metrics. For instance, DCBM (the model using the scattering metrics as predictors [33]) has a median F-Measure 12% higher than the structural model (69% vs 57%), while the improvement considering each dataset independently varies between 1% and 32%. This result confirms previous findings on the superiority of process metrics in the prediction of bugs [40], [55]. The only exception regards the Developer Model (DM) which uses the number of developers as predictor of the bugginess of a code component. However, also in this case the result confirms previous analyses conducted by Ostrand *et al.* [31] and by Di Nucci *et al.* [33] about the limited usefulness of this metric in bug prediction.

Fig. 2: Performances achieved by the experimented prediction models. Red dots indicate the mean of the distributions

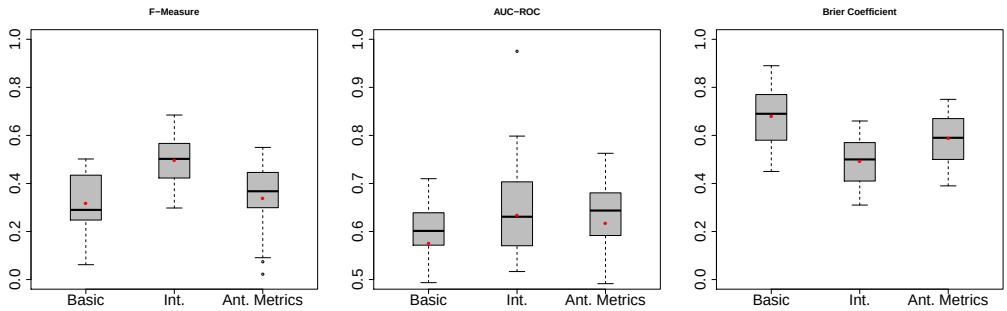
(a) Performances of Structural Metric-based Models. *Basic* refers to the model built only using structural metrics, *Int.* refers to the model in which the intensity index is an additional predictor, *Ant. Metrics* refers to the model where the antipattern metrics are included as additional predictors.



(b) Performances of Entropy-based Models. *Basic* refers to the model built only using the entropy of changes, *Int.* refers to the model in which the intensity index is an additional predictor, *Ant. Metrics* refers to the model where the antipattern metrics are included as additional predictors.



(c) Performances of DM-based Models. *Basic* refers to the model built only using the number of developers, *Int.* refers to the model in which the intensity index is an additional predictor, *Ant. Metrics* refers to the model where the antipattern metrics are included as additional predictors.



(d) Performances of DCBM-based Models. *Basic* refers to the model built only using scattering metrics, *Int.* refers to the model in which the intensity index is an additional predictor, *Ant. Metrics* refers to the model where the antipattern metrics are included as additional predictors.

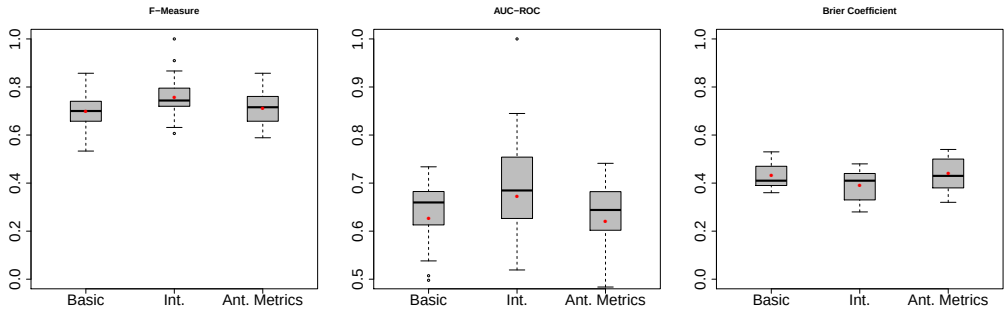


TABLE 7: Percentage of Smelly and Non-Smelly Classes Correctly Classified by the Experimented Models

Basic Model	Configuration	% Cor. Class. Smelly Instances	Cor. Class. Non-Smelly Instances
Structural Metrics [30]	Basic	45	66
	Basic + Intensity	**69**	**68**
	Basic + Ant. Metrics	48	66
BCCM [5]	Basic	42	55
	Basic + Intensity	**55**	**68**
	Basic + Ant. Metrics	49	56
DM [56]	Basic	31	41
	Basic + Intensity	**63**	**46**
	Basic + Ant. Metrics	42	43
DCBM [33]	Basic	53	87
	Basic + Intensity	**77**	**88**
	Basic + Ant. Metrics	64	85

TABLE 8: Comparison between ColorOptionPane and JEditMetalTheme (JEdit) in terms of Structural Metrics

Class	WMC	DIT	NOC	CBO	CE	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC	AVG(CC)	Intensity
org.gjt.sp.jedit.options.ColorOptionPane	7	2	0	6	5	8	0.68	277	0.74	0	0.87	0.24	4	3	9	1	4.5
org.gjt.sp.jedit.gui.JEditMetalTheme	8	4	0	7	4	7	0.59	265	0.78	1	0.88	0.29	6	2	7	1	8.4

TABLE 9: The likelihood of each model appearing in the top Scott-Knott ESD rank. A likelihood of 80% indicates that a classification technique appears at the top-rank for 80% of the studied datasets.

Basic Model	Configuration	SK-ESD Likelihood
Structural Metrics [30]	Basic	29
	Basic + Intensity	**85**
	Basic + Ant. Metrics	34
BCCM [5]	Basic	44
	Basic + Intensity	**69**
	Basic + Ant. Metrics	55
DM [56]	Basic	9
	Basic + Intensity	**52**
	Basic + Ant. Metrics	37
DCBM [33]	Basic	57
	Basic + Intensity	**86**
	Basic + Ant. Metrics	69

Turning to the role of the intensity index, we observed that regardless of the set of basic predictors used by a prediction model, the intensity of code smells provides an additional useful information able to increase the ability of the model in discovering bug-prone code components. This is observable by looking at all the accuracy indicators reported in Figure 2.

Contribution in Structural-based Models. The addition of the intensity allows the model to reach a median F-Measure of 59%, AUC-ROC of 76%, and a Brier Score of 0.4, *i.e.,* +4%, +3%, and -0.10 with respect to the basic model, respectively. When compared to the antipattern metrics-including model, the intensity-including one still performs better (*i.e.,* +8% in terms of median F-Measure, +3% in terms of median AUC-ROC, and -0.06 in terms of Brier score). When considering the improvement on each dataset independently, the model that includes the intensity index has an improvement ranging between 5% and 17% in terms of F-Measure than the basic one.

Looking deeply into the results, we observed that the box plots for the intensity-including model appear less disperse than the basic one, meaning that the addition of the intensity index makes the performances of the model more stable. Furthermore, the entire distributions report improved values with respect to the model that does not include the intensity index. For instance, an interesting example regards the `JEdit 4.0` project, where the basic model reached 81% of precision and 67% of recall (F-Measure=73%). By adding the intensity index, the model obtained 83% of precision and 75% of recall (F-Measure=79%). Investigating more in depth the causes behind the increment of the performances, we found that the model including the intensity index was able to correctly classify all the smell instances in the system (*i.e.,* 16% of the total classes). These results highlight how the addition of a measure reporting the severity of a smell is actually useful when predicting the bugginess of a code smell.

It is also important to highlight that the structural based intensity-including model also performs better than the basic one when considering the Brier score: from a practical point of view, this result means that the predictions provided by the model relying on the intensity index are closer to the actual outcome, thus being more accurate when defining the bugginess of classes. Analyzing the percentage of *smelly* and *non-smelly* classes correctly classified by the specialized bug prediction

model instantiated on the basis of the structural model, we can understand that the increment of the performances is mainly due to a better classification of instances composing the set of classes having design flaws (+24%), while the *non-smelly* classes are treated generally in the same way by both the models, even if also in this case a slight increment is visible (+2%). An interesting example regarding the previously mentioned `JEdit` project is represented by the class `ColorOptionPane` contained in the `org.gjt.sp.jedit.options` package. This class contains a *Dispersed Coupling* code smell having an intensity index of 4.5. The basic model classifies this class as buggy, since its structural metrics are considered by the model as indicators of the presence of a bug. Conversely, the low level of intensity allows the *Basic + Intensity* model to correctly mark this class as non-buggy. On the other hand, an example of code component correctly classified as buggy thank to the use of the intensity index computed for the *Message Chains* smell is the `JEditMetalTheme` class from the `org.gjt.sp.jedit.gui` package. In this case, the *Basic* model misclassifies this class as non-buggy, while the specialized model correctly classifies it as buggy. It is important to note that the `ColorOptionPane` and `JEditMetalTheme` classes have similar metrics (as shown in Table 8), and the only predictor able to distinguish them is the intensity index.

When studying the *Structural Model* which includes the antipattern metrics defined by Taba *et al.* [27], we observed that it achieves performances similar to the basic model when considering the F-Measure. At the same time, the addition of the antipattern metrics provide benefits with respect to the Brier score, with an improvement of 0.04. We also found cases where the antipattern metrics allow the prediction model to be up to 10% more precise in the predictions. Thus, we can confirm what has been found in the previous empirical validation conducted by Taba *et al.* [27]. For example, the precision achieved by the *Basic + Ant. Metrics* model on the `Apache Camel 1.4` project is 54%, while its recall reached 13% (F-Measure=20%). In this case, the value of precision is exactly 10% higher than the one of the basic model. We manually analyzed this specific case in order to understand the reasons behind this result. Surprisingly, we observed that in this project the code smell instances tend to frequently co-occur with buggy classes. As a result, several buggy classes contain more code smells. By definition, the antipattern metrics have the goal to measure the quantity, the complexity or the recurrence length of code smells. Since in `Apache Camel 1.4` the quantity of code smells contained by a class is a particularly relevant feature, this aspect gave the possibility to the antipattern metrics to better characterize the bug-prone components. Note that in this project the *Basic + Intensity* model still outperforms the other baseline models (F-Measure=42%). However, we observed that in this particular situation the main contribution of the intensity index was given to the classification of *non-smelly* classes (the percentage of correctly classified *non-smelly* instances is 16% higher than the basic model), while the *smelly* classes have been classified better by the *Basic + Ant. Metrics* model (the percentage of correctly classified *smelly* instances is 4% higher than our model).

TABLE 10: Overlap analysis between the model including the intensity index and the model including the antipattern metrics.

Basic Model	Int. $\cap$ Ant.%	Int. $\setminus$ Ant.%	Ant. $\setminus$ Int.%
Structural Metrics [30]	39	34	27
BCCM [5]	45	38	17
DM [56]	54	38	8
DCBM [33]	42	37	21

This result is particularly interesting because, while the *Basic + Ant. Metrics* model always achieved lower performance than the *Basic + Intensity* model (considering the median of the distributions, -8% of F-Measure, -3% of AUC-ROC, and +0.06 of Brier score), it is remarkable that in some cases the antipattern metrics can provide useful and complementary information with respect to the intensity index, paving the way to the possibility to obtain still better performances by considering both the intensity and the antipattern metrics. The claim is supported by the analysis of the overlap metrics computed on the set of *buggy* and *smelly* classes correctly classified by the two models, shown in Table 10. While 39% of the instances are correctly classified by both the models, a consistent portion of instances are classified only by our model (34%) or by the model using the antipattern metrics (27%). From a practical perspective, this means that the smell-related information taken into account by the *Basic + Intensity* and *Basic + Ant. Metrics* models are orthogonal and complement each other.

The observations made above were also confirmed from a statistical point of view. Indeed, as shown in Table 9 the intensity-including prediction model consistently appeared in the top Scott-Knott ESD rank in terms of AUC-ROC, meaning that its performance was statistically higher than the baselines in most of the cases (29 projects out of 34).

Contribution in Process-based Models. When including the intensity index, all the experimented prediction models improved their performance with respect to the basic models. Moreover, we found interesting complementarities between the intensity-including and antipattern metrics-including models even though our solution tends to perform better than the one by Taba *et al.* [27].

Further analyzing the results, we observed that the use of the intensity index as additional feature can increase the number of correctly classified instances, resulting in a lower Brier score than the basic model. This is a quite expected result, since the addition of the intensity index

adds an orthogonal source of information with respect to the process metrics. It is particularly evident in the case of the DM model, where the median F-Measure increased of 22% when the intensity is plugged-in. In the other cases, the addition of the intensity results in a median F-Measure 10% higher in the BCCM model and 6% higher in the case of the DCBM model, with an individual dataset improvement ranging between 2% and 17%, and 1% to 9%, respectively. Moreover, it is important to note that the models including the smell intensity are able to assist in both the prediction of *smelly* and *non-smelly* classes. For instance, the percentage of *non-smelly* instances correctly classified by the *BCCM + Intensity* model is 13% higher with respect to the baseline. This means that the information about the quality of the source code is effectively used by the prediction model to better discriminate bug-prone code components.

An aspect to highlight is that in the cases when the prediction accuracy of the baseline process-based models are low, the intensity can increase the quality of the predictions up to 47%. This is the case of `Apache Velocity 1.4` project, where the BCCM model reaches 33% of accuracy in the predictions. By adding the intensity index, the prediction model increases its performances to 80% (+47%), demonstrating that a better characterization of the classes having design problems can help in obtaining more accurate predictions. It is also interesting to analyze the results on the percentage of *smelly* classes correctly classified. On the `Apache Velocity 1.4` project, the baseline model correctly classifies half of the *smelly* classes, while the model considering the intensity is able to capture 100% of the *buggy and smelly* classes.

Another interesting observation can be made analyzing the results obtained on the DCBM model. Although the performances of such model are quite high (median F-Measure=70%, AUC-ROC=67%, Brier score=0.44), also in this case the addition of the intensity index is able to refine the predictions of the baseline model, ensuring slightly higher performances. This is mainly due to the better results obtained in the classification of *smelly* classes (overall, +24% of correctly classified smelly instances). Thus, we can claim that even in cases where the performances of the baseline models are high, the intensity index still helps in improving them.

Finally, when considering the model including the antipattern metrics we learnt that it has lower performances than the model including the intensity index. However, it is important to point out that the *Basic + Ant. Metrics* model still produces more accurate predictions than the basic models. More importantly, as in the case of the structural model we observed a strong complementarity between the set of *buggy and smelly* classes correctly classified by our model and by the *Basic + Ant. Metrics* (see Table 10). This aspect confirms that also in the prediction models based on process metrics the addition of smell-related metrics is always convenient, and that better performances might be achieved by considering a combination of the intensity and the antipattern metrics.

TABLE 11: Gain Provided by Each Metric To The Prediction Model based on Structural Metrics.

Metric	Mean	St. Dev.	Class	SK-ESD Likelihood
CBO	0.41	0.23	buggy	67
LCOM3	0.34	0.25	non-buggy	74
WMC	0.33	0.05	buggy	61
Intensity	**0.32**	**0.16**	**non-buggy**	**53**
DAM	0.25	0.06	buggy	48
DIT	0.22	0.05	non-buggy	45
Average Number of Antipatterns	0.21	0.09	buggy	44
AMC	0.15	0.11	non-buggy	31
LOC	0.15	0.07	buggy	25
MFA	0.14	0.02	buggy	23
IC	0.11	0.03	non-buggy	17
CBM	0.09	0.04	non-buggy	14
Antipattern Complexity Metric	0.07	0.02	buggy	9
AVG(CC)	0.05	0.05	buggy	5
CE	0.04	0.02	buggy	5
CAM	0.03	0.02	non-buggy	3
Antipattern Recurrence Length	0.03	0.02	buggy	2
MOA	0.02	0.01	non-buggy	2
NOC	0.01	0.02	non-buggy	2
NPM	0.01	0.01	buggy	2

The statistical analyses confirmed the findings discussed above (see Table 9). Indeed, the likelihood to be ranked at the top by the Scott-Knott ESD test is always higher for the models including the intensity index. At the same time, the antipattern metrics-including models were confirmed to provide statistically better performances than the basic ones in most cases.

Summary for RQ1. The addition of the intensity index as predictor of buggy components generally increases the performance of the baseline bug prediction models over all the analyzed projects in terms of F-Measure, AUC-ROC, and Brier score. We also observed cases in which the prediction accuracy increases up to 47% with respect to the performance achieved by models not considering the intensity metric. Our findings are also statistically significant.

Summary for RQ2. Even if the prediction models including the antipattern metrics slightly outperform the basic models, we experienced that they have lower performances than the proposed *intensity-including* models. However, we observed an interesting complementarity between the set of *buggy and smelly* classes correctly classified by the *intensity-including* and by the *antipattern metrics-including* models, which highlights the possibility to obtain still higher performances through a combination of smell-related information.

5.2 The gain provided by the intensity index and by the other predictors

While in the previous analyses we investigated the performances of bug prediction models with and without the addition of smell-related information, in this stage

TABLE 12: Gain Provided by Each Metric To The BCCM Prediction Model.

Metric	Mean	St. Dev.	Class	SK-ESD Likelihood
Entropy of Changes	0.84	0.08	non-buggy	92
Intensity	**0.44**	**0.11**	**buggy**	**77**
Average Number of Antipatterns	0.29	0.13	buggy	56
Antipattern Complexity Metric	0.07	0.03	non-buggy	18
Antipattern Recurrence Length	0.03	0.06	non-buggy	11

TABLE 13: Gain Provided by Each Metric To The DM Prediction Model.

Metric	Mean	St. Dev.	Class	SK-ESD Likelihood
# Developers	0.75	0.14	non-buggy	85
Intensity	**0.39**	**0.15**	**buggy**	**61**
Average Number of Antipatterns	0.34	0.13	buggy	56
Antipattern Complexity Metric	0.12	0.36	buggy	14
Antipattern Recurrence Length	0.07	0.03	non-buggy	8

we are interested in understanding how important are the predictors composing the different models analyzed in this study, with the aim to evaluate the predictive power of the intensity index and of the other predictors.

Table 11 shows the results achieved when applying the *Gain Ratio Feature Evaluation* algorithm [120] on the set of predictors investigated in the structural metrics-based bug prediction model, while Tables 12, 13, and 14 report the contribution given by the predictors studied when considering the BCCM, DM, and DCBM models. The results have been aggregated to provide a clearer visualization. Specifically, we report the ranking of the predictors based on their importance for the model, together with the values of the mean and the standard deviation (computed by considering the results obtained on the single systems) of the expected reduction in entropy caused by partitioning the prediction model according to a given predictor. In addition, we also provide (i) the class to which a certain predictor contributes the most (*i.e.,* if a predictor helps more in the prediction of buggy or non-buggy classes) and (ii) the likelihood of the predictor to be in the top-rank by the Scott-Knott ESD test, *i.e.,* the percentage of times a predictor was statistically superior than the others.

Gain Provided to Structural-based Models. The results show that the Coupling Between Objects (CBO) is the metric having the highest predictive power, confirming the findings by Gyimóthy *et al.* [45]. In particular, we found CBO at the top of the ranked list on 28 out of the total 34 systems analyzed, with an average reduction of entropy of 0.41 and a standard deviation of 0.23 (*i.e.,* we found one case where the expected reduction of entropy reaches 0.64, which means it is a very strong predictor). Interestingly, CBO provides a higher predictive power with respect to the correct assessment of buggy classes, meaning that the coupling between classes is an important factor characterizing classes affected by bugs. The Scott-Knott ESD test statistically confirmed the importance of the predictor, since the information gain given by the metric was statistically higher than other metrics in 67% of the cases. While CBO is the most relevant predictor for buggy classes, the Lack of Cohesion of Methods (LCOM3) was the most important with respect to non-buggy classes. According to the Scott-Knott ESD test, the metric appeared statistically more powerful than the other metrics in 74% of the datasets. To quantify the benefit of this predictor to the resulting model, we observed that on average the information gain provided by the metric was 0.34 with a standard deviation of 0.25. The Weighted Method per Class (WMC) also resulted to be a relevant metric for predicting buggy classes, looking both to the information gain and Scott-Knott ESD results. Indeed, it allowed an information gain equals to 0.33 (standard deviation of 0.05) and was at the top of the Scott-Knott ESD rank in 61% of the datasets.

TABLE 14: Gain Provided by Each Metric To The DCBM Prediction Model.

Metric	Mean	St. Dev.	Class	SK-ESD Likelihood
Structural Scattering	0.67	0.22	buggy	93
Semantic Scattering	0.54	0.26	non-buggy	85
Intensity	**0.33**	**0.15**	**buggy**	**48**
Average Number of Antipatterns	0.25	0.11	buggy	41
Antipattern Complexity Metric	0.10	0.04	buggy	22
Antipattern Recurrence Length	0.06	0.09	non-buggy	11

Thus, in general we can observe that metrics related to coupling, cohesion, and complexity are highly important in the prediction of bugs. Just behind these metrics, the intensity index is the feature providing the highest gain in terms of reduction of entropy. We observed that the contribution given by the metric is valuable on all the object projects (minimum gain=0.16, maximum gain=0.48), confirming that its addition can effectively increase the accuracy of a structural prediction model. This is a quite surprising result, since our goal is not the addition of the most relevant predictor, but rather the introduction of a measure able to complement the information used by a prediction model by quantifying in a single value the severity of design problems affecting a class. A possible reason behind the result come from the fact that in the context of a structural-based bug prediction model the intensity works better in the classification of non-buggy classes: likely, this is because the degree of smelliness of a code component allows the model to balance the structural metric values of classes, as in the case shown in Table 8. Looking at the results of the statistical test, we observed that the intensity index is ranked on the top by the Scott-Knott ESD in 53% of the cases, thus confirming the high predictive power of the metric.

It is interesting to discuss the result achieved on the `Apache Lucene 2.4` project, where the intensity metric is evaluated as the most important by the *Gain Ratio Feature Evaluation* algorithm, which quantifies as 0.47 the gain of the metric in reducing the entropy of the prediction model. Looking at the ranking, we observed that the single quality metrics from which the intensity index is computed (*i.e.,* metrics used for the smell detection) are placed by the algorithm to the bottom of the ranked list

(*e.g.,* in this case, LCOM3 is only partially relevant and it provides a small gain of 0.09). In other words, the single metrics do not reduce in the same measure the entropy with respect to the case in which such metrics are condensed in a single value representing the intensity of a code smell. As an example, the intensity index contributes in reducing the entropy of the prediction model 25% more than the LOC metric, and 6% more than WMC metric. It is worth noting that, as a consequence, the ability of the specialized bug prediction model to correctly classify *smelly* instances on `Apache Lucene 2.4` increases of 22%. Another interesting observation can be made by looking more in depth into the results of `Apache Velocity 1.4`. Also in this case, the metrics used for the detection of smells are partially relevant for the prediction model when considered individually (*e.g.,* CBO=0.27), while the intensity measure is instead considered as a very useful predictor (gain=0.46). Here the performance provided by the *intensity-including* bug prediction model are 25% better than the baseline model and this is due to the fact that the specialized model is able to correctly classify all the *smelly* instances in the system.

As for the antipattern metrics, we observed that the ANA (*i.e., Average Number of Antipatterns*) metric has a good ranking (mean gain=0.21) while the other two metrics, *i.e., Antipattern Complexity Metric* (ACM) and *Antipattern Recurrence Length* (ARL), only provide a partial contribution in the reduction of entropy of the model. Also in this case, the Scott-Knott ESD test statistically confirmed the findings: indeed, ANA was a top predictor in 44% of the datasets, as opposed to ACM and ARL metrics which appeared as statistically more powerful than other metrics only in 9% and 2% of the cases, respectively. It is important to note that the information gain analysis highlighted that ANA performed better in the classification of buggy classes, suggesting that it is somehow complementary with respect to the intensity index in the context of product-based bug prediction models. This result still indicates that taking into account the quantity of code smells in a class represents a useful source of information for improving the performances of bug prediction models.

Gain Provided to Process-based Models. Concerning the *process metric-based* models considered in this study, the results are similar. Indeed, Table 12 highlights how the intensity index has a mean information gain of 0.44 and it is ranked, overall, just behind the entropy of changes, that is the *core* metric of the BCCM model, while the ANA metric is much more important than the other antipattern metrics (+0.22 and +0.26 with respect to ACM and ARL, respectively). It is worth noting that in this case both the intensity index and ANA provided contributions in the identification of buggy classes, while the entropy of changes mainly helped in the characterization of non-buggy instances: thus, the results indicate that taking into account other factors such as the quality of source code actually provide an important contributions for prediction. The results discussed so far were confirmed statistically: indeed, the statistical ranking provided by the Scott-Knott ESD test is similar to the one produced by the information gain algorithm.

When considering the DM and DCBM models (see Tables 13 and 14) the results follow the same pattern: the intensity index is placed behind the *core* metrics of the models but before the antipattern metrics. Also in this case, the Scott-Knott ESD test statistically confirmed the findings.

As general observation, it is interesting to notice that the mean information gains of the intensity index and of the metric counting the number of antipatterns are generally higher in the process metrics-based models than in the structural metrics-based model. These metrics are computed by considering static properties of the source code and, therefore, are more relevant in the context of prediction models based on historical metrics because they add an orthogonal source of information.

On the light of these results, we can conclude that the intensity index provides a strong information gain to all the bug prediction models considered in the study. At the same time, we experienced that also the ANA metric has a good predictive power. This confirms somehow what we found when evaluating the overlap between the *intensity-including* and the *antipattern metrics-including* models: indeed, the high predictive power of such predictors suggest that their combination might provide further improvements to the basic bug prediction models.

Summary for RQ3. The intensity index has a higher predictive power with respect to the individual metrics from which it is derived. On all the projects of the study, we found that the intensity metric is one of the most important predictors of the model. As a consequence, the gain provided by the intensity index to the baseline prediction model is highly relevant. Moreover, we found that the metric able to quantify the number of code smells in a class has a good predictive power, confirming that better performances in the prediction of bugs can be achieved by considering both the intensity and the antipattern metrics.

5.3 The performances of the combined smell-aware bug prediction model

The results achieved in the previous research questions highlight the possibility to build a combined bug prediction model that takes into account smell-related information besides the product and process metrics. With this research question (**RQ4**), we aim at evaluating the performances of such prediction model. As explained in Section 4, we performed a feature selection procedure to discard irrelevant features. From the initial set composed

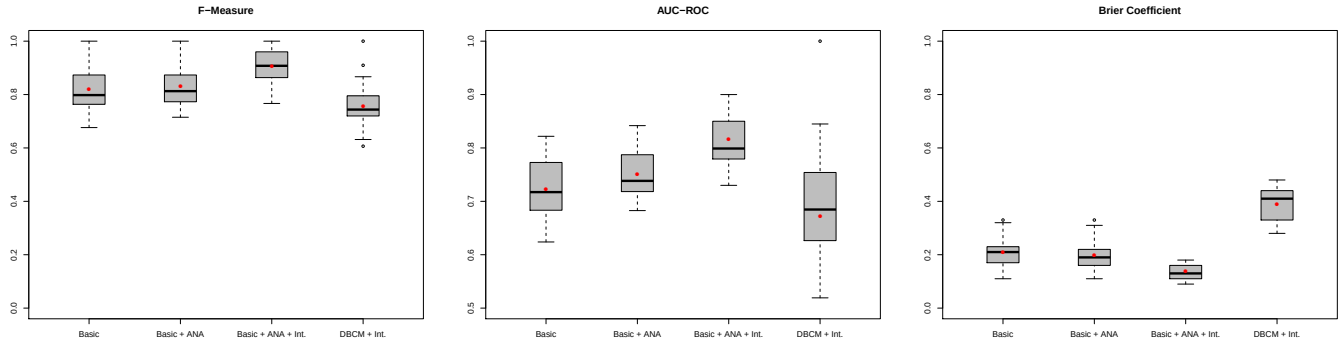


Fig. 3: Performances of the Smell-Aware Bug Prediction Model.

TABLE 15: The likelihood of combined models appearing in the top Scott-Knott ESD rank. A likelihood of 80% indicates that a classification technique appears at the top-rank for 80% of the studied datasets.

Basic Model	Configuration	SK-ESD Likelihood
Combined	Basic	86
Combined	Basic + ANA	86
Combined	**Basic + ANA + Intensity**	**94**
DCBM [33]	Basic + Intensity	47

of 24 product, process, and smell-related metrics, this procedure found ten highly correlated variables:

1) Average Cyclomatic Complexity (AVG(CC)) and Weighted Method per Class (WMC);
2) Number of Children (NOC) and Depth of Inheritance Tree (DIT);
3) Efferent Coupling (CE) and Coupling Between Objects (CBO);
4) Number of Public Methods (NPM) and Measure of Functional Abstraction (MFA);
5) Cohesion Among Methods of Class (CAM) and Lack of Cohesion of Methods 3 (LCOM3);
6) Measure of Aggregation (MOA) and Data Access Metric (DAM);
7) Inheritance Coupling (IC) and Depth of Inheritance Tree (DIT);
8) Number of Developers and Structural Scattering;
9) Antipattern Complexity Metric (ACM) and Entropy of Changes;
10) Antipattern Recurrence Length (ARL) and Entropy of Changes;

According to the results of the *vif* function, we discarded seven structural metrics, *i.e.*, AVG(CC), NOC, CE, NPM, CAM, MOA, IC, one process metric, i.e., Number of Developers, and two smell-related metrics, i.e., ACM and ARL. Therefore, the resulting model contains the following 14 metrics: nine structural metrics, *i.e.*, CBO, WMC, LCOM3, DAM, DIT, AMC, LOC, MFA, CBM, three process metrics, *i.e.*, entropy of changes, structural and semantic scattering, and two smell-related information, *i.e.*, the intensity index and ANA. It is worth noting that the features selected in this stage perfectly correspond to the most powerful predictors identified in **RQ3**.

Figure 3 depicts the box plots reporting the distributions of F-Measure, AUC-ROC, and Brier score related to the smell-aware combined bug prediction model (label "Basic + ANA + Int."). To facilitate the comparison with the models exploited in the context of **RQ1** and **RQ2**, the figure also reports the performances of the best prediction model resulting in the previous analyses, *i.e.*, the *DBCM + Int.* model (label "DBCM + Int." in Figure 3). Moreover, to evaluate to what extent the smell-related information are actually needed in the context of a model mixing product and process metrics, we also report the performances achieved by the combined model built without using the smell-related metrics (label "Basic"). Finally, we also report the performances of the model built including only the ANA metric as additional predictor in the model mixing together product and process metrics (label "Basic + ANA" in Figure 3): this helped us in understanding the contribution given by adding the intensity index in a model already considering a combination of product, process, and smell-related information.

In the first place, the results highlight that the addition of smell-related information always boosts the performances of a combined model built using product and process metrics. Indeed, both the *Basic + ANA* and *Basic + ANA + Int.* provide improved performances than the *Basic* combined model considering all the evaluation metrics. Furthermore, it is important to note that also in this case the intensity index gives an important contribution in the prediction of buggy prone classes, *i.e., Basic + ANA + Int.* is the model achieving the best results: the F-Measure improvement varies between 2% and 14% when compared to the *Basic* one, while it ranges between 2% and 11% when compared with the *Basic + ANA* model. The evaluation of the other metrics, *i.e.*, AUC-ROC and Brier score, confirm the findings indicating that the addition of the intensity index consistently contributes in boosting the performances of the other experimented

TABLE 16: Percentage of Smelly and Non-Smelly Classes Correctly Classified by the Combined Models

Basic Model	Configuration	% Cor. Class. Smelly Instances	Cor. Class. Non-Smelly Instances
Combined	Basic	81	90
Combined	Basic + ANA	84	90
Combined	**Basic + ANA + Intensity**	**93**	**91**
DCBM [33]	Basic + Intensity	77	88

models.

Moreover, it is worth noting that the combined model also achieves higher performances with respect to the *DBCM + Int.* model when considering both the evaluation metrics and the percentage of *smelly* and *non-smelly* instances correctly classified (shown in Table 16). In particular, the model significantly outperforms the other in terms of median F-Measure (+13%), AUC-ROC (+12%), and Brier score (-0.27).

As expected, the results are statistically significant (see Table 15), as the devised *Basic + ANA + Int.* appeared in top Scott-Knott ESD rank in 94% of the cases, overcoming the other models built using a combination of metrics and the one relying on scattering plus the intensity metrics. When considering *buggy and smelly* instances, it is worth observing that the addition of the intensity index allows the combined model to correctly classify a larger number of such instances than (i) the *Basic + ANA* one (+9%), (ii) the *Basic* one (+12%), and (iii) the *DBCM + Int.* one (+16%). At the same time, it slightly improves also the classification of *non-smelly* classes (+1% with respect to the other combined models).

On the one hand, the reason behind the strong improvement obtained by the combined models is that the combination of predictors having different nature provides an improvement of the performances of the models exploiting single types of predictors, as pointed out by D'Ambros *et al.* [53]. On the other hand, it is important to remark that, as demonstrated by the first three research questions, the addition of smell-related information helps in characterizing both the classes affected by code smells and those components not affected by any design flaw.

An interesting example regards the project `Apache POI 2.5.1`. In this case, by mixing product and process metrics only the resulting precision is 76%, while the recall reaches 84% (F-Measure=80%). Even though the performances are higher than the one achieved by the *DBCM + Int.* model (+8% of precision, +5% of recall), at the same time they are strongly lower than the ones obtained by the combined model which takes into account the smell metrics. Indeed, the latter model achieves 100% of precision and recall (+24% of precision and 16% of recall) since it is able to perfectly characterize both *smelly* and *non-smelly* classes.

In conclusion, we can claim that bug prediction models taking into consideration the presence of design problems are able to perform much better than the other models.

TABLE 17: Performance of JCodeOdor on the software projects object of the empirical study

Code Smell	Precision	Recall	F-Measure	# TP	# FP	# FN
God Class	77%	85%	81%	85	25	15
Data Class	79%	86%	83%	83	22	13
Brain Method	73%	77%	75%	79	29	23
Shotgun Surgery	70%	84%	77%	76	32	14
Dispersed Coupling	82%	85%	84%	87	19	15
Message Chains	78%	86%	82%	87	24	14
Overall	76%	84%	80%	496	154	94

Summary for RQ4. As expected, combined models perform better than prediction models relying on single types of predictors. However, the contribution of smell-related information are valuable also when product and process metrics are used together. At the same time, the addition of the intensity index in a combined model built using product, process, and the ANA metric is still valuable and consistently boosts the performances of the prediction model.

6 Threats to Validity

Threats to *construct validity* are related to the relationship between theory and observation. Above all, we relied on *JCodeOdor* [29] for detecting code smells.

The intensity index computed by the tool derives by a set of code metrics characterizing cohesion, coupling, complexity, size, and data access of classes. A first problem threatening our observations might be the redundancy of such metrics [125]. To verify the validity of the intensity computation, we assessed multicollinearity between metrics using the same procedure as the one adopted in the context of **RQ1** and **RQ4**, *i.e.*, we computed the Spearman's rank correlation between all possible pairs of metrics and then we exploited a stepwise variable removal using the *vif* function. As a result, we found all the Spearman's correlation values to be lower than 0.6, thus indicating weak correlations between them. Moreover, the *vif* function did not report the need to remove variables. Thus, we can claim that the intensity computation is not threatened by multicollinearity between the variables it is derived from. Complete results of this analysis are available in our online appendix [34].

Still, our observations might have been threatened by the presence of a high number of code smell co-occurrences in our dataset [23]. In these cases, we built

smell-aware bug prediction models using the maximum intensity computed by *JCodeOdor*: while the maximum value likely highlights the code smell that impact more the maintainability of the class, the co-occurrence of other code smells might hide information that are not captured by our model. To measure the extent to which this phenomenon happened in our study, we computed the percentage of classes in our dataset affected by more smells: as a result, we found that only a small portion of classes, on average 8% of the project classes, contains more than one smell: thus, we can claim that the problem of co-occurrence is quite limited in our case.

We have validated the code smell detector performance on the software projects analyzed in this paper. Table 17 reports precision, recall, and F-Measure values obtained by considering the instances of all the projects as a single dataset (*i.e.,* overall). A detailed analysis of the performance of the detector for each project can be found in our online appendix [34]. We can observe that the performance of the detector ranges between 75% and 84% in terms of F-Measure. Despite the quite high precision and recall (*i.e.,* overall, 76% and 84%, respectively), the tool still identifies 154 false positives and 94 false negatives among the 45 systems considered. To make the set of code smells as close as possible to the *golden set*, in our study we fixed the output of the tool by (i) setting to zero the intensity index of the false positive instances, and (ii) discarding the false negatives, for which we could not assign an intensity value. As such manual adjustments are not always feasible, we evaluated the effect of including false positives and false negatives in the construction of the bug prediction model.

In the first place, we re-run the analyses described in Section 4 and validate the performances of the models including the false positive instances using the same metrics used to assess the performances of the other prediction models (*i.e.,* precision, recall, F-Measure, AUC-ROC, and Brier score). As a result, we observed that these models always perform better than the baselines that do not include any smell-related information, while their performances are slightly lower (2% in terms of median F-Measure) than the ones of the prediction models built discarding the false positive instances.

Secondly, we evaluated what is the impact of including false negative instances. Since their intensity index is equal to zero, they have been considered as *non-smelly* classes. The results showed that the *intensity-including* models still produce more accurate results than the basic models, by boosting their median F-Measure of 5%. At the same time, there is a decrement of 4% in terms of F-Measure with respect to the performances of the prediction models built discarding false negatives.

Finally, we considered the case where both false positives and false negatives are included in the prediction model. We observed that the *Basic + Intensity* models have a median F-Measure 5% lower than the models where the false positive and false negative instances have been filtered out. However, they are still better than basic models (median F-Measure=+4%). Thus, we concluded that *a fully automatic code smell detection process still improves the performance of the baseline bug prediction model*.

TABLE 18: Code Smells treated by Existing Prioritization Approaches

Tool	God Class	Data Class	Brain Method	Shotgun Surgery	Dispersed Coupling	Message Chains
Arcelli Fontana *et al.* [29]	X	X	X	X	X	X
Arcoverde *et al.* [126]	X					
Khomh *et al.* [67]	X					
Marinescu [127]	X	X	X			
Oliveto *et al.* [68]	X					
Tsantalis *et al.* [128]	X					
Vidal *et al.* [129]	X	X	X	X	X	

It is important to remark that the selection of the code smell detector might have influenced the results. However, among all the detectors proposed in literature [99] *JCodeOdor* is the only one suitable for our purpose since it is the only one able to compute an intensity index for the code smells considered in the study. More specifically, most of the previous approaches do not rank code smells based on their severity but classify code components using a boolean value reporting the presence/absence of a smell [9], [10], [11], [12], [130]; on the other hand, Table 18 reports the code smell prioritization techniques defined in literature and whether they defined an intensity index for the code smells considered in our study. As it is possible to observe, most of them [67], [68], [126], [128] are only able to treat the God Class smell, not defining an intensity index for the other code smells. At the same time, the tool proposed by Vidal et al. [129] is able to deal with five of the considered smells, however it does not treat the Message Chains one. Conversely, the tool by Arcelli Fontana et al. [29] detects and prioritize all the code smells in our study. Moreover, it is also worth to note that the performances of the employed detector are quite high and the presence of false positives does not have an extreme negative impact on the bug prediction capabilities: from a practical point of view, this means that the use of a more accurate code smell detection tool computing an intensity index can only improve the results achieved in this study.

Another threat to construct validity regards the annotated set of bugs and code smells used in the empirical study. As for bugs, we rely on the publicly available oracles in the PROMISE repository [87]: to ensure qual-

ity and robustness of data we performed (i) a careful selection of projects considering only the ones having a percentage of buggy classes lower than 75% and EPV ratios higher than 10, as suggested in previous work [90], [91], and (ii) a preliminary data preprocessing following the guidelines provided by Shepperd *et al.* [92] in order to remove noisy data. For code smells, we rely on the oracles publicly available in [100], previously used in [10], [11], [13], [18], [131]. However, we cannot exclude that the oracle we used misses some bugs or smells, or else include some false positives.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. To evaluate the experimented bug prediction models we employed a number of metrics, *i.e.,* precision, recall, F-Measure, AUC-ROC, and Brier score, able to provide an overview of their performances under different perspectives, thus allowing us to better report pros and cons of using the intensity index as additional predictor. We also assessed the model for the presence of multi-collinearity through the use of proper statistical methods such as the Spearman's rank correlation [97], removing irrelevant features using the *variance inflation factors* function [94]. Moreover, we analyzed to what extent the intensity index is important with respect to the other metrics by analyzing the gain provided by the addition of the severity measure in the model.

In the context of **RQ3**, we measured the contribution of the intensity index in bug prediction model performances by exploiting the *Gain Ratio Feature Evaluation* algorithm [120]. While the usage of other procedures (*e.g.,* the Wrapper approach [132]) might have lead to different results, it is important to note that this algorithm was the most suitable for our study since it allowed us to quantify the exact entropy reduction achieved using the intensity index.

Still in this category there is a possible threat related to the validation methodology exploited. As shown by Tantithamthavorn *et al.* [90], ten-fold cross validation might provide unstable results in cases when the number of events per variables (EPV) is lower than 10. For this reason, we limited our analyses to software projects having a number of EPV higher than 10.

Finally, threats to *external validity* concern the generalization of results. We analyzed a large set of 34 releases of 11 software systems coming from different application domains and with different characteristics (size, number of classes, *etc.*). Note that we had to focus our analyses on the only systems for which an oracle reporting the set of actual bugs was available. Another threat in this category regards the choice of the baseline models. However, we evaluated the contribution of the smell-related information in the context of bug prediction models widely used in the past [33], [53] that take into account predictors of different nature, *i.e.,* product and process metrics.

7 Conclusion and Future Work

Bug prediction models help developers in the analysis and testing of the source code components more likely containing defects. While a lot of work has been done for the definition of product or process metrics having high predictive power, the research community only partially explored the role of code smells in the process of bug prediction.

In this paper, we evaluated to what extent the addition of the intensity index (*i.e.,* a metric that quantifies the severity of code smells) [29] in existing state-of-the-art bug prediction models is useful to increase the performances of the baseline models. Specifically, we firstly set up four baseline prediction models, *i.e.,* a model based on a set of structural metrics [30], the BCCM model proposed by Hassan [5], the DM model defined by Ostrand *et al.* [31], and the DCBM model proposed by Di Nucci *et al.* [33]. Then, we compared the performances of such models *with* and *without* the addition of the intensity index, in order to control the actual contribution of the severity of code smells. Moreover, we also compared the models mentioned above with the same baseline models built by adding the antipattern metrics defined by Taba *et al.* [27] instead of the intensity index.

The results indicated how the addition of the intensity index as predictor of buggy components generally increases the performance of the baseline bug prediction models, reaching an improvement of 7%, 10%, 21%, 8% of the F-Measure in the cases where the basic predictors are the structural metrics, the entropy of changes, the number of developers, and the scattering metrics, respectively. When compared with the models built using the antipattern metrics, we observed that the models including the intensity index obtain an accuracy up to 16% higher. More notably, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two different configuration of models.

In the second step of our analyses, we quantified the actual gain provided by the intensity index with respect to the other metrics composing the models, confirming the high predictive power of the intensity index over all the baseline models. We also noticed how the ANA metric (*i.e., Average Number of Antipatterns*) has a good predictive power if compared to the other antipattern metrics.

Based on these results, we built a smell-aware prediction model which combines product, process, and smell-related information. The performances of this new model outperform the ones of all the other experimented models, confirming once again the usefulness of considering code smells in bug prediction.

According to our experiments, the intensity *always* positively contributes to state-of-the-art prediction models, even when they already have high performances. In particular, the intensity index helps discriminating bug-prone code elements affected by code smells in bug

prediction models based on product metrics, process metrics, and a combination of the two. Our results also suggest that the intensity of code smells is helpful in all these cases, and cannot be replaced by a simple indicator of the presence or absence of a code smell. More importantly, the presence of a limited number of false positive smell instances identified by the code smell detector does not impact the accuracy and the practical applicability of the proposed smell-aware bug prediction model.

As for future work, we firstly plan to further analyze how the intensity index impacts the performances of bug prediction models, by performing a fine-grained analysis into the role of each smell type independently on the prediction power. Furthermore, since our study has focused on *global* bug prediction, future effort will be devoted to the analysis of the contribution of smell-related information in the context of *local-learning* bug prediction models [39]. Finally, our future research agenda includes the definition of new factors influencing the performances of prediction models.

REFERENCES

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

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

[3] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, 1994, pp. 279–287.

[4] W. Harrison, "An entropy-based measure of software complexity," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 1025–1029, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1109/32.177371

[5] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 78–88.

[6] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[7] 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*, Sept 2013, pp. 396–399.

[8] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, July 2014.

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

[10] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "A textual-based technique for smell detection," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016)*. Austin, USA: IEEE, 2016, p. to appear.

[11] 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, May 2015.

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

[13] 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*, p. to appear., 2017.

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

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Int'l Conf. Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

[16] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proceedings of the International workshop on Principles of Software Evolution (IWPSE)*. ACM, 2007, pp. 31–34.

[17] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2004, pp. 223–232.

[18] 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 *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.

[19] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[20] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[21] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[22] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

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

[24] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[25] 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.

[26] 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 study," *Empirical Software Engineering*, p. to appear, 2017.

[27] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 270–279. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.38

[28] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, USA: IEEE, 2016, p. to appear.

[29] F. Arcelli Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *Proceedings of the Seventh International Workshop on Managing Technical Debt (MTD 2015)*. Bremen, Germany: IEEE, Oct. 2015, pp. 16–24, in conjunction with ICSME 2015.

[30] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868342

[31] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9178-4

[32] D. Di Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia, "On the role of developer's scattered changes in bug prediction," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 241–250.

[33] D. D. Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. D. Lucia, "A developer centered bug prediction model," *Transactions on Software Engineering*, p. to appear., 2017.

[34] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Toward a Smell-aware Bug Prediction Model," Tech. Rep., 1 2017. [Online]. Available: http://tinyurl.com/hzusyds

[35] W. Harrison, "Using software metrics to allocate testing resources," *J. Manage. Inf. Syst.*, vol. 4, no. 4, pp. 93–105, Apr. 1988. [Online]. Available: http://dx.doi.org/10.1080/07421222.1988.11517810

[36] C.-Y. J. Peng, K. L. Lee, and G. M. Ingersoll, "An introduction to logistic regression analysis and reporting," *The Journal of Educational Research*, vol. 96, no. 1, pp. 3–14, 2002.

[37] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 789–800.

[38] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[39] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 343–351. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100072

[40] W. P. Raimund Moser and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *International Conference on Software Engineering (ICSE)*, ser. ICSE '08, 2008, pp. 181–190.

[41] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, Oct 1996.

[42] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, Jun 1994.

[43] W. M. Khaled El Emam and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[44] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, 2003.

[45] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 897–910, 2005.

[46] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switchess," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, pp. 886–894, 1996.

[47] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062558

[48] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[49] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[50] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," in *Proceedings of the 9th IEEE International Symposium on Software Metrics*. IEEE CS Press, 2003, pp. 338–349.

[51] A. N. Taghi M. Khoshgoftaar, Nishith Goel and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Software Reliability Engineering*. IEEE, 1996, pp. 364–371.

[52] J. S. M. Todd L. Graves, Alan F. Karr and H. P. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[53] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, 2012.

[54] A. E. Hassan and R. C. Holt, "Studying the chaos of code development," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[55] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 309–311. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414063

[56] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise '11. New York, NY, USA: ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2020390.2020392

[57] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 19:1–19:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868357

[58] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[59] 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," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414.

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

[61] 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," *Transactions on Software Engineering*, 2017.

[62] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Science of Computer Programming*, vol. 102, no. 0, pp. 44 – 56, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314005711

[63] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, pp. 1120–1128, 2007.

[64] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[65] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the International Software Metrics Symposium (METRICS)*. IEEE, September 2005, p. 15.

[66] 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, 2006.

[67] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the International Conference on Quality Software (QSIC)*. Hong Kong, China: IEEE, 2009, pp. 305–314.

[68] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on B-splines," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2010, pp. 248–251.

[69] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *2016 IEEE 23rd Interna-*

tional Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, March 2016, pp. 609–613.

[70] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. Gerosa, "Satt: Tailoring code metric thresholds for different software architectures," in *2016 IEEE 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, p. to appear.

[71] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*. IEEE, 2013, pp. 268–278.

[72] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[73] F. Arcelli 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. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9378-4

[74] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. ACM, 2010, pp. 113–122.

[75] 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, Sept 2014.

[76] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8084, pp. 50–65.

[77] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[78] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of Systems and Software (JSS)*, 2016.

[79] F. Arcelli Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, vol. 2. Florence, Italy: IEEE, May 2015, pp. 803–804.

[80] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[81] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proc. 17th Asia Pacific Software Eng. Conf.* Sydney, Australia: IEEE, December 2010, pp. 336–345.

[82] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics (WETSoM 2015)*. Florence, Italy: IEEE, May 2015, pp. 44–53, co-located with ICSE 2015.

[83] S. Theodoridis and K. Koutroumbas, "Pattern recognition," *IEEE Transactions on Neural Networks*, vol. 19, no. 2, pp. 376–376, 2008.

[84] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov 2012.

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

[86] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[87] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[88] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, May 2011.

[89] T. Mende, "Replication of defect prediction studies: Problems, pitfalls and recommendations," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868336

[90] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2584050

[91] ——, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 321–332. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884857

[92] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *Software Engineering, IEEE Transactions on*, vol. 39, no. 9, pp. 1208–1215, Sept 2013.

[93] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[94] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11135-006-9018-6

[95] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, June 2014.

[96] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Comments on researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1092–1094, Nov 2016.

[97] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 100, no. 3, pp. 441–471, 1987.

[98] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 432–441. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486846

[99] 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*, ser. EASE '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:12. [Online]. Available: http://doi.acm.org/10.1145/2915970.2915984

[100] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Landfill: An open dataset of code smells with public evaluation," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 482–485.

[101] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1961.

[102] L. M. Y. Freund, "The alternating decision tree learning algorithm," in *Proceeding of the Sixteenth International Conference on Machine Learning*, 1999, pp. 124–133.

[103] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Eleventh Conference on Uncertainty in Artificial Intelligence*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345.

[104] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[105] R. Kohavi, "The power of decision tables," in *8th European Conference on Machine Learning*. Springer, 1995, pp. 174–189.

[106] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2188385.2188395

[107] P. A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*, 1982.

[108] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502041

[109] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. IBM, 2008, p. 23.

[110] E. J. W. J. Sunghun Kim and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[111] G. W. Brier, "Verification of Forecasts expressed in terms of probability," *Monthly Weather Review*, vol. 78, no. 1, pp. 1–3, Jan. 1950.

[112] K. Rufibach, "Use of Brier score to assess binary predictions," *Journal of Clinical Epidemiology*, vol. 63, no. 8, pp. 938–939, 2010.

[113] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The impact of switching to a rapid release cycle on the integration delay of addressed issues - an empirical study of the mozilla firefox project," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 374–385.

[114] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[115] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine learning research*, vol. 7, no. Jan, pp. 1–30, 2006.

[116] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[117] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[118] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 135–145.

[119] B. Ghotra, S. Mcintosh, and A. E. Hassan, "A large-scale study of the impact of feature selection techniques on defect classification models," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 146–157. [Online]. Available: https://doi.org/10.1109/MSR.2017.18

[120] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986. [Online]. Available: http://dx.doi.org/10.1023/A:1022643204877

[121] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[122] S. Kabinna, W. Shang, C.-P. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 326–337.

[123] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, pp. 1–35, 2016.

[124] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "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, 2015, pp. 812–823.

[125] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "A study of redundant metrics in defect prediction datasets," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2016, pp. 51–52.

[126] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *Software Engineering (SBES), 2013 27th Brazilian Symposium on*. IEEE, 2013, pp. 69–78.

[127] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

[128] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 25–34.

[129] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, Sep 2016. [Online]. Available: https://doi.org/10.1007/s10515-014-0175-x

[130] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[131] F. Palomba, "Textual analysis for code smell detection," in *Proceedings of the International Conference on Software Engineering (ICSE) - Volume 2*. IEEE, 2015, pp. 769–771.

[132] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artif. Intell.*, vol. 97, no. 1-2, pp. 273–324, Dec. 1997. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(97)00043-X