Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells

Fabio Palomba*, Marco Zanoni†, Francesca Arcelli Fontana†, Andrea De Lucia*, Rocco Oliveto‡
*University of Salerno, Italy, †University of Milano-Bicocca, Italy, ‡University of Molise, Italy
fpalomba@unisa.it, 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 and comparing the results of the new model against the baseline model. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also evaluate the actual gain provided by the intensity index with respect to the other metrics in the model, including the ones used to compute the code smell intensity. We observe that the intensity index is much more important as compared to other metrics used for predicting the buggyness of smelly classes.**

I. Introduction

In the last decade, the research community has spent a lot of effort in investigating bad code smells (shortly "code smells" or simply "smells"), *i.e.*, symptoms of poor design and implementation choices applied by programmers during the development of a software project [1]. Besides approaches for the automatic identification of code smells in source code [2]–[7], empirical studies have been conducted to understand when and why code smells appear [8], the relevance they have for developers [9], [10], their evolution and longevity in software projects [11]–[14], as well as the negative effects of code smells on software understandability [15], and maintainability [16]–[19]. Recently, Khomh *et al.* [20] have also empirically demonstrated that classes affected by design problems ("antipatterns") 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 not been captured in bug prediction models yet. Indeed, while previous work has proposed the use of predictors based on product metrics (e.g., see [21]–[23]), as well as the analysis of change-proneness [24]–[26], the entropy of changes [27], or human-related factors [28]–[30] to build accurate bug prediction models, none of them takes into account a measure able to quantify the presence and the severity of design problems affecting code components.

In this paper, we aim at making a further step ahead by studying the role played by bad code smells in bug prediction. Our hypothesis is 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 buggyness of such a component*. To verify this conjecture, we use the intensity index (*i.e.*, a metric able to estimate the severity of a code smell) defined by Arcelli Fontana *et al.* [31] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluate the predictive power of the intensity index by adding it in a bug prediction model based on structural quality metrics [32], 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. Finally, we report further analyses aimed at understanding (i) the accuracy of a model where a simple truth value reporting the presence/absence of code smells rather than the intensity index is added to the baseline model, (ii) the impact of false positive smell instances identified by the code smell detector, and (iii) the contribution of the intensity index in bug prediction models based on process metrics.

The results of our study indicate that:

- The addition of the intensity index as predictor of buggy components positively impact the accuracy of a bug prediction model based on structural quality metrics. We observed an improvement of the accuracy of the classification up to 25% as compared to the accuracy achieved by the baseline model.
- The intensity index is more important than other quality metrics for the prediction of the bug-proneness of smelly classes.
- 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 specialized bug prediction model.
- The intensity index positively impacts the performance of bug prediction models based on process metrics, increasing the accuracy of the classification up to 47%.

Structure of the paper. Section II discusses the related literature on bug prediction models, while Section III presents the specialized bug prediction model for *smelly* classes. Section IV describes the design and the results of the case study aimed at evaluating the accuracy of the proposed model. Section V discusses the results of the additional analyses we conducted.

Finally, Section VI concludes the paper and outlines directions for future work.

II. Related Work

The research community spent a lot of effort in the definition of techniques aimed at predicting bug-prone code components, mainly proposing the use of *product metrics* and *process metrics* as indicators of the bug-proneness of a code component.

A. Bug Prediction using Structural-based Predictors

Basili *et al.* [21] proposed the use of the Object-Oriented metric suite (i.e., CK metrics) [33] 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.* [34] and by Subramanyam *et al.* [22] corroborate the results previously observed in [21]. On the same line, Gyimothy *et al.* [23] 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 [33] is the best metric among the CK ones in predicting defects. Ohisson *et al.* [35] 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 [36] exploited the use of static code analysis tools to predict the bug density of for 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.* [37] 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.* [38], which reports a positive correlation between code complexity and bugs. Finally, Nikora *et al.* [39] 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.

B. Bug Prediction using Process-based Predictors

Khoshgoftaar *et al.* [40] 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.* [41] 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.* [42] 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 [43] introduced the concept of entropy of changes as a measure of the complexity of the development process. Moser *et al.* [24] 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.* [25] 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.* [26] confirm that the change-proneness is the best bug predictor. Hassan [27] exploit 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 reported case study indicate that the proposed techniques have higher prediction accuracy than models purely based on code components changes. All of 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.* [28], [29] propose the use of the *number of developers who modified a code component in a give time period* as a bug-proneness predictor, demonstrating that products and process metrics is poorly (positively) impacted by also considering the developers' information. Finally, Di Nucci *et al.* [30] exploited the role of two metrics, i.e., structural and semantic scattering of changes performed by a developer in bug prediction. Their findings demonstrate on the one hand the superiority of the bug prediction model built using scattering metrics with respect other state-of-the-art models. Moreover, they also show that the proposed metrics are orthogonal with respect to other predictors.

III. 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, other than 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.* [31]. Specifically, the index is computed by *JCodeOdor*, a code smell detector which relies on detection strategies applied on

TABLE I
CODE SMELL DETECTION STRATEGIES (THE COMPLETE NAMES OF THE METRICS ARE GIVEN IN TABLE II)

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))

metrics. The tool is able to detect, filter [44] and prioritize [31] instances of six kinds of code smells [1], [45]:

- 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 in the range [1,10]. In particular, given a code smell instance, its intensity is computed by relying on different kinds of information, i.e., (i) the code smell detection strategy, (ii) the metric thresholds used in the detection strategy, (iii) the statistical distribution of the metric values computed on a large dataset represented as a quantile function, and (iv) the actual values of the metrics used in the detection strategies.

Above all, the detection strategies used are the ones proposed in *JCodeOdor*, reported in Table I. Detection strategies have often been used in the literature [45], [46] for code smell detection. They rely on the evaluation of a set of metric values against defined thresholds, composed in a logical proposition. A code component is detected as *smelly* if one of the logical propositions shown in Table I is true, namely if the actual metrics of the code component exceed the threshold values composing a detection strategy. The list of the metrics applied in the detection rules is reported in Table II (see [31] for reference). The thresholds are represented as logical values associated to an actual value, and they are derived from the statistical distribution [47] of metrics in 74 systems of the Qualitas Corpus [48]. Table III reports all the threshold values associated to each of the detected code smells. Specifically, for each metric used in a detection strategy, *JCodeOdor* extracts five meaningful values to be used as thresholds: VERY-LOW, LOW, MEAN, HIGH, VERY-HIGH. For metrics representing ratios defined in the range [0,1] (e.g, the Tight Class Cohesion), these values are fixed to 0.25, 0.33, 0.5, 0.66 and 0.75, respectively. For all other metrics, they are associated to percentile values on the metric distribution [47]. 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. Finally, the value is normalized in the range [1,10]. The intensity index of the code smell is given by the mean of the exceeding amounts of the metrics used for the detection. The higher the intensity index, the higher the severity of the code smell under analysis. More details on the computation of the intensity index can be found in [31].

TABLE II
METRICS USED FOR CODE SMELLS DETECTION

Short Name	Long Name
ATFD	Access To Foreign Data
*ATLD	Access To Local Data
CC	Changing Classes
CDISP	Coupling Dispersion
CINT	Coupling Intensity
CM	Changing Methods
CYCLO	McCabe Cyclomatic Complexity
FANOUT	Number of Called Classes
LOC	Lines Of Code
*LOCNAMM	Lines of Code Without Accessor or Mutator Methods
*MaMCL	Maximum Message Chain Length
MAXNESTING	Maximum Nesting Level
*MeMCL	Mean Message Chain Length
*NMCS	Number of Message Chain Statements
NOAM	Number Of Accessor Methods
NOLV	Number Of Local Variables
*NOMNAMM	Number of Not Accessor or Mutator Methods
NOPA	Number Of Public Attributes
TCC	Tight Class Cohesion
*WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods
WOC	Weight Of Class

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 additional predictor of a structural metrics-based bug prediction model. Indeed, we cannot use the intensity index as single predictor, since in this case we could not predict the bug-proneness

TABLE III
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

of classes not affected by any design problem (the intensity index for *non-smelly* classes is equal to 0). Thus, 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. Finally, we add the information about the intensity to a set of structural metrics in order to apply the predictions.

IV. EVALUATION OF THE PROPOSED MODEL

The *goal* of the empirical study is to evaluate the contribution of the *intensity* index in a prediction model 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 accuracy and completeness as compared to 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.

The *context* of the study consists of six software systems having different size and scope, namely `Apache Xerces`[1], `Apache Xalan`[2], `Apache Velocity`[3], `Apache Tomcat`[4], `Apache Lucene`[5], and `Apache Log4j`[6]. Table IV reports the characteristics of the analyzed

[1]http://xerces.apache.org
[2]http://xalan.apache.org
[3]http://velocity.apache.org
[4]http://tomcat.apache.org
[5]http://lucene.apache.org
[6]http://logging.apache.org/log4j/2.x/

TABLE IV
SOFTWARE PROJECTS IN OUR DATASET

System	Classes	KLOCs	% Buggy Cl.	% Smelly Cl.
Apache Xerces 1.4.4	588	141	74	5
Apache Xalan 2.7	909	428	86	12
Apache Velocity 1.6.1	229	57	15	7
Apache Tomcat 6.0	858	301	6	4
Apache Lucene 2.4	338	103	59	10
Apache Log4j 1.2	205	38	87	15

software systems in terms of (i) system's size considering number of classes and KLOC, (ii) the percentage of buggy files (identified as explained later), and (iii) the percentage of classes affected by design problems (detected as explained later). All the data used in the study are publicly available in our online appendix [49].

A. Empirical Study Definition and Design

In the context of this empirical investigation, we formulated the following research questions:

> **RQ_1:** *To what extent the intensity index contributes to the prediction of bug-prone code components?*

> **RQ_2:** *What is the gain provided by the intensity index to the bug prediction model when compared to the other predictors?*

To answer RQ_1, we firstly need an oracle reporting the presence of bugs in the source code of the analyzed software projects. Fortunately, all the systems are hosted on the PROMISE repository [50], which collects a large dataset of bugs and provides oracles for all the projects in this study. Secondly, we need to instantiate the prediction model presented in Section III to define (i) the basic predictors, (ii) the code smell detection process, and (iii) the machine learning technique to use for classifying buggy instances. 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 better understand the predictive power of the intensity index, we decide to test its contribution in a bug prediction model composed by structural predictors, and in particular the 20 quality metrics exploited by Jureczko *et al.* [32]. Our choice is 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. Thus, to measure the extent to which the contribution of the *intensity* index is useful for predicting bugs, we experimented the following bug prediction models:

- *Basic Model*: The model based on the 20 software metrics defined by Jureczko *et al.* [32];

- *Basic Model + Intensity*: The model above based on the 20 software metrics plus the intensity index. It is worth remembering 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. Regarding the code smell detection process, our study is focused on the analysis of the code smells for which an intensity index has been defined (see Section III). To this aim, we rely on the detection performed by *JCodeOdor* [31], because on the one hand it has been empirically validated demonstrating good performances in detecting code smells, and on the other hand it detects all the code smells considered in the empirical study. Finally, it computes the value of Intensity on the detected code smells. To build a bug prediction model that discriminates actual *smelly* and *non-smelly* classes, we decide to discard the false positive instances from the set of candidate code smells given by the detection tool (in other words, we set the intensity of false positives to 0). To this aim, we manually discard such instances by comparing the results of the tool against an annotated set of code smell instances publicly available [51]. 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* 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 0, and still have no effect on the prediction model. The final step is the definition of the machine learning classifier to use. We experimented several classifiers, namely Multilayer Perceptron [52], ADTree [53], Naive Bayes [54], Logistic Regression [55], Decision Table Majority [56], and Simple Logistic [57]. 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 [49]. Over all the systems, the best results on the baseline model were obtained using the Simple Logistic, confirming previous findings in the field [42], [58]. Thus, in this paper we report the results of the models built with this classifier. This classifier 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 [57].

Once the model has been instantiated, to assess its performance we adopted the 10-fold cross-validation strategy [59]. This strategy randomly partitions the original set of data into 10 equal sized subset. 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 [59]. 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 [28], [60]–[62]). Finally, we answer $\mathbf{RQ}_1$ by reporting three widely-adopted metrics, namely accuracy, precision and recall [63]. In addition, we also report 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. Besides the analysis of the performance of the specialized bug prediction model and its comparison with the baseline model, we also investigate the behavior of the experimented models in the classification of *smelly* and *non-smelly* instances. Specifically, we compute 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.

As for $\mathbf{RQ}_2$, we conduct a *fine-grained* investigation aimed at measuring how important is the intensity index with respect to the other features (i.e., metrics) composing the model. In particular, we use an *information gain* algorithm [64] to quantify the gain provided by adding the intensity index in the 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 [64] 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:

$$InfoGain(M, p_i) = H(M) - H(M|p_i) \tag{1}$$

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:

$$H(M) = -\sum_{i=1}^{n} prob(p_i) \log_2 prob(p_i) \tag{2}$$

In other words, the algorithm quantifies how much uncertainty in M was reduced after splitting M on attribute p_1. In the context of our work, we apply the *Gain Ratio Feature Evaluation* algorithm [64], 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 at the top. Using this procedure, we evaluate the relevance of the predictors in the prediction model, possibly 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)

TABLE V
ACCURACY, PRECISION, RECALL, F-MEASURE, AUC-ROC, AND PERCENTAGE OF BUGGY CLASSES AFFECTED (AND NOT) BY A SMELL CORRECTLY CLASSIFIED BY THE EXPERIMENTED PREDICTION MODELS

Project	Model	Accuracy	Precision	Recall	F-Measure	AUC-ROC	% Cor. S-Cl.	Class. NS-Cl.
Apache Xerces	Basic	94	93	94	93	95	72	92
	Basic + Int.	95	96	95	95	95	97	92
Apache Xalan	Basic	99	99	100	99	99	94	100
	Basic + Int.	100	100	100	100	100	99	100
Apache Velocity	Basic	67	18	28	22	89	56	32
	Basic + Int.	92	31	46	37	90	100	35
Apache Tomcat	Basic	56	9	16	12	81	34	17
	Basic + Int.	79	20	31	24	94	83	20
Apache Lucene	Basic	78	75	76	75	81	50	78
	Basic + Int.	80	80	80	80	83	82	77
Apache Log4j	Basic	94	99	96	97	89	55	93
	Basic + Int.	95	100	98	99	93	74	93

or with respect the other structural metrics contained in the model.

B. Analysis of the Results

In the following we will discuss the achieved results aiming at providing an answer to our research questions.

To what extent the intensity index contributes to the prediction of bug-prone code components? Table V reports for each considered software project, the results achieved when considering (i) the baseline prediction model built using the structural metrics in [32] and (ii) the model built by adding to the baseline model the intensity index of *smelly* classes. In addition, the column *% Cor. Class. S-Cl.* of Table V reports the percentages of *smelly* classes correctly classified (with respect to buggyness) by each of the analyzed models, while *% Cor. Class. NS-Cl.* reports the percentage of correctly classified *non-smelly* instances.

Looking at Table V, the first thing that leaps to the eye is that, overall, the basic prediction model is able to achieve high accuracy. For instance, on `Apache Xalan` the basic model obtains 100% of recall, 99% of precision and accuracy. From a practical point of view, this result means that the model misclassifies a small subset of instances. On the other hand, taking into account the intensity index of the *smelly* classes results in 100% for all the considered metrics, correctly classifying the instances missed by the basic model. It is worth noting that obtaining an increment of the performance in situations when the *Basic* model works well is quite hard. Still, in this situations the intensity index is able to contribute by refining the predictions of the *Basic* model, and increasing model's accuracy. Analyzing the percentage of *smelly* and *non-smelly* classes correctly classified by the specialized bug prediction model, we can understand that the increment of the performance is due to a better classification of instances composing the set of classes having design flaws (+5%), while the *non-smelly* classes are treated generally in the same way by both the models. An interesting example is represented by the class `ElemTextLiteral` contained in the `org.apache.xalan.templates` package. This class contains a *Brain Method* code smell having an intensity index of 5.5. The basic model classifies this class as buggy, since its structural metrics are considered by the *Basic* model as indicators of the presence of a bug. Conversely, the low level of intensity allows the *intensity-including* 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 `XSLTProcessorApplet` class from the `org.apache.xalan.client` 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 `ElemTextLiteral` and `XSLTProcessorApplet` classes have similar metrics (as shown in Table VI), and the only predictor able to distinguish them is the intensity index.

Looking at the other software systems analyzed, we can observe for `Apache Xerces` and `Apache Log4j` a behavior of the intensity index similar to the one achieved on `Apache Xalan`. In these cases, the performances of the *Basic* model are always slightly improved by the addition of the intensity index. Also, the intensity index enables the correct classification of *smelly* instances, while the predictions made on classes not affected by design problems remain exactly the same.

A different analysis can be done for the other systems. Indeed, when the performances of the *Basic* model are quite low, the intensity index is able to give a strong contribution in the classification of buggy components. For instance, the accuracy of the *intensity-including* model in `Apache Velocity` is 25% higher than the one achieved by the basic one. Here we can note that the proposed bug prediction model is not only able to correctly classify all the instances affected by smells, but also gives a slight contribution (+3%) to the classification of the *non-smelly* instances. This result clearly highlights the importance of considering the design quality characteristics of a code component when predicting bugs.

> **Summary for RQ$_1$.** The addition of the intensity index as predictor of buggy components generally increases the performance of the baseline bug prediction model over all the analyzed projects. We observed cases in which the prediction accuracy increases up to 25% with respect to the performance achieved not considering the intensity metric.

What is the gain provided by the intensity index to the bug prediction model when compared to the other predictors? Table VII shows the results achieved when applying the *Gain Ratio Feature Evaluation* algorithm [64] on the set of predictors composing the *intensity-including* bug prediction model. Specifically, for each software system, we report the ranking of the predictors based on their importance for the model, together with a value representing the expected reduction in entropy caused by partitioning the prediction model according to a given predictor (i.e., column *Gain*). The results show that the Coupling Between Objects (CBO) metric is highly

TABLE VI
COMPARISON BETWEEN ELEMTEXTLITERAL AND XSLTPROCESSORAPPLET (APACHE XALAN) IN TERMS OF STRUCTURAL METRICS

Class	WMC	DIT	NOC	CBO	RFC	LCOM	CA	CE	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC	MAX(CC)	AVG(CC)	Intensity
org.apache.xalan.templates.ElemTextLiteral	11	3	0	9	22	29	6	4	11	0.88	127	0.8	0	0.95	0.34	2	5	10	2	1	5.5
org.apache.xalan.client.XSLTProcessorApplet	12	5	0	7	23	31	5	2	11	0.89	125	0.9	1	0.94	0.28	3	5	11	2	1	9.1

TABLE VII
GAIN PROVIDED BY EACH METRIC TO THE PREDICTION MODEL.

Apache Xerces		Apache Xalan		Apache Velocity		Apache Tomcat		Apache Lucene		Apache Log4j	
Metric	Gain	Metric	Gain	Metric	Gain	Metric	Gain	Metric	Gain	Metric	Gain
CBO	0.47	RFC	0.38	CAM	0.66	CBO	0.41	**Intensity**	**0.47**	CBO	0.49
CE	0.39	NPM	0.37	NPM	0.58	RFC	0.39	RFC	0.45	**Intensity**	**0.49**
CA	0.22	LCOM	0.22	RFC	0.56	MOA	0.38	CBO	0.45	CA	0.47
AMC	0.17	WMC	0.22	**Intensity**	**0.53**	LOC	0.37	NPM	0.43	NPM	0.38
AVG(CC)	0.17	**Intensity**	**0.21**	WMC	0.53	**Intensity**	**0.32**	CE	0.42	LCOM	0.33
MFA	0.15	CE	0.20	CE	0.48	MAX(CC)	0.30	WMC	0.41	CE	0.31
LOC	0.12	CBO	0.19	LCOM	0.46	WMC	0.22	LCOM	0.29	LCOM3	0.31
DIT	0.11	LOC	0.14	DIT	0.35	AMC	0.22	DAM	0.26	NOC	0.29
CBM	0.11	CAM	0.13	LCOM3	0.35	CAM	0.21	CAM	0.24	DIT	0.29
IC	0.11	DIT	0.11	CA	0.35	DAM	0.19	LOC	0.22	RFC	0.29
Intensity	**0.10**	IC	0.09	CBO	0.29	AVG(CC)	0.19	AMC	0.16	LOC	0.26
LCOM3	0.07	CBM	0.09	NOC	0.28	LCOM	0.17	IC	0.11	AVG(CC)	0.25
MAX(CC)	0.07	MFA	0.08	AVG(CC)	0.28	NPM	0.16	CA	0.11	AMC	0.19
CAM	0.06	MAX(CC)	0.08	CBM	0.25	NOC	0.11	MAX(CC)	0.10	MAX(CC)	0.15
DAM	0.05	NOC	0.07	MAX(CC)	0.23	DIT	0.08	MFA	0.09	DAM	0.14
RFC	0.05	AMC	0.06	AMC	0.21	IC	0.05	CBM	0.07	CBM	0.08
MOA	0.04	CA	0.03	IC	0.19	CA	0.05	NOC	0.07	IC	0.06
WMC	0.02	MOA	0.03	DAM	0.08	MFA	0.05	DIT	0.07	CAM	0.04
NOC	0.01	DAM	0.01	MFA	0.06	CBM	0.03	AVG(CC)	0.06	MFA	0.04
LCOM	0.01	AVG(CC)	0.01	MOA	0.03	LCOM3	0.02	LCOM3	0.05	MOA	0.02
NPM	0.01	LCOM3	0.01	LOC	0.02	CE	0.02	MOA	0.02	WMC	0.01

TABLE VIII
PERFORMANCE OF JCODEODOR ON THE SOFTWARE PROJECTS OBJECT OF THE EMPIRICAL STUDY

Code Smell	Precision	Recall	F-Measure	# TP	# FP	# FN
Blob	78%	81%	79%	25	7	6
Data Class	89%	100%	94%	8	1	8
Brain Method	87%	92%	89%	124	19	11
Shotgun Surgery	60%	64%	62%	9	6	5
Dispersed Coupling	76%	81%	78%	25	8	6
Message Chains	65%	79%	71%	15	8	4
Overall	81%	87%	84%	206	49	32

relevant for the predictions made on 3 of the analyzed projects (i.e., `Apache Xerces`, `Apache Tomcat`, and `Apache Log4j`), confirming the findings by Gyimóthy *et al.* [23] on the predictive power of the metric. On the other systems, different complexity metrics (e.g., RFC and CAM) appear to the top of the ranked list. As for the intensity index, we observe that the contribution given by the metric is valuable on all the object projects (minimum gain=0.10, maximum gain=0.53), since it is generally included at the first places of the ranked list. This is a quite surprising result, since the goal of the addition of the intensity index is not to provide the most relevant predictor, but to complement the information used by a prediction model with a metric able to quantify in a single value the severity of design problems affecting a class. For instance, it is interesting to discuss the result achieved on the `Apache Lucene` 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 can see 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., LCOM3 is only partially relevant and it provides a small gain of 0.05). 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, 18% more than LCOM 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` increases of 22% (see Table V). Another interesting observation can be made by looking at the results of `Apache Velocity`. 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=0.29), while the intensity measure is instead considered as a very useful predictor (gain=0.53). 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. In the worst case, the intensity index is ranked as the eleventh more useful predictor on `Apache Xerces` with a gain equals to 0.10. However it is important to highlight, as shown in Table V, that on this project the performances of the baseline model are high, and the intensity index contributes to the increment of 1% in terms of accuracy.

Summary for RQ$_2$. 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.

C. Threats to Validity

Threats to *construct validity* are related to the relationship between theory and observation. Above all, we relied on *JCodeOdor* [31] for detecting code smells. We have validated the code smell detector performance on the software projects

analyzed in this paper. Table VIII 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 [49]. We can observe that the performance of the detector ranges between 62% and 94% in terms of F-Measure. Despite the quite high precision (i.e., 81% on overall), the tool still identifies 49 false positives which we discarded to make the set of code smells as close as possible to the *golden set*. As removing false positives is not always feasible, in Section V we evaluate the effect of including false positives in the construction of the bug prediction model. On the other hand, the tool achieves an overall recall of 87%. In the empirical study, we were not able to include false negative smell instances, because the tool assigns an intensity index equal to zero to such instances. Even if this could have influenced our results, it is worth noting that only 32 of such instances (out of the total 238) are missed from the analysis. Future work will be devoted to improve the detection performance of the tool by including rules used by other smell detectors (e.g., DECOR [4]). 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 [50] which have been widely used in previous research [32], [65]–[68]. For code smells, we rely on the oracles publicly available in [51], previously used in [5], [6], [8], [9], [69]. 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. The metrics used to evaluate the bug prediction models, i.e., accuracy, precision, recall, F-Measure, and AUC-ROC are widely used in the evaluation of the performances of bug prediction techniques [42]. 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.

Finally, threats to *external validity* concern the generalization of results. We analyzed six different software projects from different application domains and with different characteristics (size, number of classes, etc). However, our future agenda includes the analysis of other systems aimed at corroborating our findings. Another threat in this category regards the choice of the baseline model. We selected the model by Jureczko *et al.* [32] since it is more interesting and challenging of the predictive power of the intensity index when it is added to a model characterized by other structural metrics, including the ones used for the computation of the intensity index. Moreover, the selected model contains a comprehensive set of quality metrics, which allowed a more detailed analysis of the gain provided by the intensity index in the context of the structural-based bug prediction model. However, as pointed out by Moser *et al.* [24], predictors based on process metrics can achieve better performances in predicting bugs. To deal with this threat, in Section V we discuss the results achieved when considering the intensity index as additional predictor of models including process metrics.

V. Discussion and Further Analysis

The results of the empirical study reveal the usefulness of considering the intensity of code smells as additional predictor in order to classify instances affected by design problems. From a practical perspective, results indicate that smells having low severity are less prone to be affected by a bug with respect to smells with high severity. In this sense, the use of an indicator of intensity is beneficial to correctly discriminate the bug-proneness of *smelly* classes. Moreover, the results also reveal that the structural metrics (including the ones used for detecting smells) are not effective when applied to predict the buggyness of classes affected by design flaws. Even if this can appear as a quite surprising result, we observe that when evaluating the bug-proneness of *smelly* classes, the prediction model is not able to correctly deal with the whole set of software metrics. In other words, several quality indicators considered in isolation work worse than a single aggregative metric reporting the degree of severity of the design flaw affecting a class.

In the context of our work, we exploited the use of the intensity index, rather than using a simple truth value providing information about the presence of a design problem. Indeed, the latter solution could also provide a complementary information with respect to the structural metrics, leading to improvements similar to the ones achieved by considering the intensity index. This issue is analyzed in Section V-A. Another discussion point is related to the threats to validity pointed out in Section IV-C. On the one hand, in the empirical study we discarded the false positive instances given by the smell detection tool to consider in the prediction model a set of code smells as close as possible to the *golden set*. However, removing such instances could not be practically applicable for several reasons (e.g., effort needed to validate smells). In order to evaluate the impact that false positives have on the performance of the prediction model, in Section V-B we evaluate the performance of the model obtained without removing the false positive code smell instances detected by the tool. Finally, a threat to the generalizability of the results regards the choice of the baseline model. To evaluate the contribution of the intensity index in different contexts, Section V-C reports the results achieved when adding the intensity of code smells in a prediction model based on process metrics, as well as the analysis of the contribution of the intensity index in a model composed by both structural and process metrics. Note that for all the additional analyses we follow the same experimental design described in Section IV.

A. Comparing the presence/absence of smells rather than the intensity index

We have added to the baseline bug prediction model defined in [32] the boolean information about the presence of code smells in a class (note that we used the *golden set* of code smells in this case, to avoid bias deriving from the use of a particular tool). The results are reported in Table IX. However,

TABLE IX
ACCURACY METRICS FOR THE MODEL BUILT BY ADDING TO THE BASELINE MODEL A TRUTH VALUE INDICATING THE PRESENCE OF A SMELL.

Project	Model	Accuracy	Precision	Recall	F-Measure	AUC-ROC	% Cor. S-Cl.	Class. NS-Cl.
Apache Xerces	Basic + Truth	94	93	94	93	95	72	92
Apache Xalan	Basic + Truth	99	99	100	99	99	94	100
Apache Velocity	Basic + Truth	67	18	28	22	89	56	32
Apache Tomcat	Basic + Truth	56	9	16	12	81	34	17
Apache Lucene	Basic + Truth	78	75	76	75	81	50	78
Apache Log4j	Basic + Truth	94	99	96	97	89	55	93

comparing Table V and IX, this choice would not lead to improvements in the performance of the baseline model (the performance of the two models are exactly the same). Indeed, the addition of such information does not provide any type of complementary information that the model can use to predict the bug-proneness of *smelly* classes. As a consequence, the presence of the additional predictor is totally irrelevant. This is because the simple truth value does not quantify the extent to which the design problem is actually harmful. For instance, let us recall the example shown in Table VI. In this case, both the instances are smelly but they have different intensity. A prediction model based on the simple thruth value would not distinguish their bug-proneness, and it would not be able to correctly classify the buggyness of the `XSLTProcessorApplet` class.

B. Evaluating the Impact of False Positive Smells in the Bug Prediction Model

Table X reports the results achieved when building the proposed bug prediction model without filtering the false positive instances from the set of candidate smells identified by *JCodeOdor*. Comparing these results with the performance of the models shown in Table V, we can provide two main observations. First of all, without filtering false positives, the bug prediction model obtains accuracy values always higher than the baseline model. This means that, even in the presence of false positive instances, the use of the proposed model in a practical case guarantees higher performance with respect to the baseline prediction model. Indeed, it is important to observe that the smelly instances correctly classified by the model ranges between 71% and 99%, clearly indicating its ability to distinguish the bug-proneness of classes affected by design problems. At the same time, the performance of the model in the classification of *non-smelly* classes are in line with the ones of the baseline. On the other hand, discarding false positive code smell instances does not result in significantly better performances with respect to including the false positives detected by the tool (compare Tables V and X). Indeed, in this case the performances of the *non-filtered false positives* model are only slightly lower, indicating that false positive instances do not have a significative impact on the results and do not need to be necessarily validated and filtered out. Summarizing, we can claim that *a fully automatic code smell detection still improves the performance of the baseline bug prediction model*.

TABLE X
ACCURACY METRICS FOR THE MODEL WHERE FALSE POSITIVE SMELLS ARE NOT FILTERED.

Project	Model	Accuracy	Precision	Recall	F-Measure	AUC-ROC	% Cor. S-Cl.	Class. NS-Cl.
Apache Xerces	Basic + Int.	94	94	94	91	95	96	92
Apache Xalan	Basic + Int.	100	100	100	100	100	99	100
Apache Velocity	Basic + Int.	90	28	43	35	89	97	35
Apache Tomcat	Basic + Int.	77	18	29	24	93	80	20
Apache Lucene	Basic + Int.	79	77	78	77	83	80	77
Apache Log4j	Basic + Int.	95	99	97	95	93	71	92

C. Evaluating the Contribution of the Intensity Index in a Process Metrics-based Bug Prediction Model

In order to evaluate the contribution of the intensity index in a process metrics-based bug prediction model, we exploit the model defined by Hassan [27], which is built by considering the entropy of changes as predictor of buggy components. The choice of using this process-based model is not random, but guided to the will to select a bug prediction model having good performance [27] and quite representative of the state-of-the-art [30]. The analysis of the contribution of the intensity index in other process metrics-based bug prediction models (e.g., the ones proposed in [28] and [30]) is part of our future agenda. As we can see from Table XI, the process model relying only on the entropy of changes does not obtain higher performances with respect to the models considering structural properties. At the same time, we can observe that the use of the intensity index as additional feature in the model can increase the number of correctly classified instances, resulting in a higher accuracy. This is a quite expected result, since the addition of the intensity index adds an orthogonal source of information with respect to the process metric. It is worth noting that in the cases when the prediction accuracy of the baseline process-based model is low, the intensity can increase the quality of the predictions up to 47%. This is the case of `Apache Velocity` project, where the baseline 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` 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. As for the other software projects, we can outline a similar trend observed in the case of the structural-based prediction model. Indeed, the intensity index is able to refine the predictions of the baseline model, ensuring slightly higher performances in cases where the performances of the baseline are already high (e.g., see the results achieved on `Apache Xerces` and `Apache Log4j`). In the other cases, we can always observe an improvement of both precision and recall (and, consequently, of the F-measure), but also an improvement of the AUC-ROC metric, which indicates the higher overall ability of the model considering the intensity in

TABLE XI
ACCURACY METRICS FOR THE MODELS BASED ON PROCESS METRICS AND A COMBINATION OF STRUCTURAL AND PROCESS METRICS.

Project	Model	Process Metrics							Combined Metrics						
		Accuracy	Precision	Recall	F-Measure	AUC-ROC	% Cor. Class. S-Cl.	NS-Cl.	Accuracy	Precision	Recall	F-Measure	AUC-ROC	% Cor. Class. S-Cl.	NS-Cl.
Apache Xerces	Basic	91	91	91	91	71	41	89	94	93	94	93	94	68	91
	Basic + Int.	94	94	94	94	95	86	89	95	96	95	95	95	97	91
Apache Xalan	Basic	99	99	99	99	98	89	99	100	100	100	100	100	92	100
	Basic + Int.	99	99	99	99	97	92	99	100	100	100	100	100	94	100
Apache Velocity	Basic	33	33	33	33	76	50	45	67	18	28	22	79	57	32
	Basic + Int.	80	80	80	80	78	100	46	94	32	47	38	80	100	33
Apache Tomcat	Basic	29	29	29	29	56	50	31	63	9	16	12	82	35	18
	Basic + Int.	67	67	67	67	68	92	30	82	21	33	26	85	83	26
Apache Lucene	Basic	60	60	60	60	55	41	63	79	76	77	76	82	47	60
	Basic + Int.	71	71	71	71	62	79	63	81	79	80	79	83	85	60
Apache Log4j	Basic	92	92	93	92	52	63	88	99	93	96	94	90	52	89
	Basic + Int.	93	93	94	93	60	71	87	97	99	98	98	93	76	90

discriminating between buggy and non-buggy classes.

Finally, we also evaluated the contribution of the intensity index in a bug prediction model composed by both structural and process metrics. Looking at the results reported in Table XI (i.e., see the *Combined* model), we can observe that the addition of the process metric does not have the same impact with respect to the addition of the intensity index in the baseline structural model. Indeed, the *Combined* model never outperforms the performance achieved by the structural model which considers the intensity as additional feature. Thus, we can conclude that the addition of the intensity index is actually needed also in this case to achieve higher performances. Moreover, when adding the intensity index to the *Combined* model, we observe that the contribution of the intensity index is still valuable. For example, let us consider the cases of `Apache Velocity` and `Apache Tomcat`. In the first project, the performance of the *Combined* model are not better with respect to the prediction model purely based on structural code metrics. However, when adding the intensity to the mixed set of metrics characterizing the *Combined* model, the performance are not only better than the baseline structural model (+27% of accuracy), but they are also better than all the other structural and process based prediction models that include the intensity index (i.e., the *Combined + Int.* model has higher performance with respect to the *Basic + Int.* structural models). In the second case, the *Combined* model is 7% more accurate of the baseline structural model. This indicates that the entropy of changes actually complements the structural metrics in the predictions of buggy components. However, also in this case the addition of the intensity index allows the prediction model to obtain a strong higher value of the prediction accuracy (+19%). This results in the achievement of higher values for all the other evaluation metrics: indeed, the precision increases of 12%, the recall of 17% and the AUC-ROC of 3%. A similar discussion can be done for the other software systems analyzed, where the intensity index actually contributes to the improvement of the performance of the *Combined* bug prediction model. Finally, as expected, we can observe that the *buggy and smelly* classes are mainly correctly classified by the model including the intensity index.

VI. CONCLUSION AND FUTURE WORK

In this paper, we investigated the contribution of code smell intensity in the context of bug prediction. Specifically, we evaluate to what extent the addition of the intensity index (i.e., a metric that quantifies the severity of code smells) in an existing structural metrics-based bug prediction model is useful in order to increase the performances of the baseline model. We also quantify the actual gain provided by the intensity index with respect to the other metrics composing the model, including the ones used to compute the code smell intensity. Moreover, we report additional analyses aimed at showing (i) the accuracy of a model where a simple truth value reporting the presence of code smells rather than the intensity index is added to the baseline model, (ii) the impact of false positive smell instances identified by a code smell detector, and (iii) the contribution of the intensity index in bug prediction models based on process metrics.

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 initial results suggest that the intensity of code smells is helpful in all of these cases, and cannot be substituted 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 specialized bug prediction model. The achieved findings highlight—on one hand—the value of code smell detection in the context of bug prediction, and on the other hand the importance of considering the intensity of such design problems as additional indicator in bug prediction models.

As future work, we plan to extend the number of systems analyzed with this method in order to corroborate the results achieved in this paper, and evaluate the contribution of the intensity index into other existing bug-prediction models.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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 International Conference on Software Engineering (ICSE) - Volume 1*. IEEE, 2015, pp. 403–414.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] 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

[27] 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.

[28] 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

[29] 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

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

[31] 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.

[32] 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

[33] 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.

[34] 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.

[35] 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.

[36] 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

[37] 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

[38] 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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells," Tech. Rep., 4 2016. [Online]. Available: http://tinyurl.com/hgorj4z

[50] 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

[51] 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.

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

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

[54] 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.

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

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

[57] 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.

[58] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, May 2015, pp. 789–800.

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

[60] 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

[61] 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.

[62] 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.

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

[64] 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

[65] H. Lu, E. Kocaguneli, and B. Cukic, "Defect prediction between software versions with active learning and dimensionality reduction," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, Nov 2014, pp. 312–322.

[66] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 481–490.

[67] T. Menzies and J. Di Stefano, "How good is your blind spot sampling policy," in *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, March 2004, pp. 129–138.

[68] 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.

[69] 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.