TU Delft

Delft University of Technology

Improving Change Prediction Models with Code Smell-Related Information

Catolino, Gemma; Palomba, Fabio; Arcelli Fontana, Francesca; De Lucia, Andrea; Zaidman, Andy; Ferrucci, Filomena

DOI
10.1007/s10664-019-09739-0

Publication date
2019

Document Version
Accepted author manuscript

Published in
Empirical Software Engineering

Citation (APA)
Catolino, G., Palomba, F., Arcelli Fontana, F., De Lucia, A., Zaidman, A., & Ferrucci, F. (2019). Improving Change Prediction Models with Code Smell-Related Information. *Empirical Software Engineering*, *25 (2020)*, 49–95. https://doi.org/10.1007/s10664-019-09739-0

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Noname manuscript No.
(will be inserted by the editor)

Improving Change Prediction Models with Code Smell-Related Information

Gemma Catolino · Fabio Palomba · Francesca Arcelli Fontana · Andrea De Lucia · Andy Zaidman · Filomena Ferrucci

Received: date / Accepted: date

Abstract Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate

Gemma Catolino, Andrea De Lucia, Filomena Ferrucci
University of Salerno, Italy
E-mail: gcatolino@unisa.it, adelucia@unisa.it, fferrucci@unisa.it

Fabio Palomba
University of Zurich, Switzerland
E-mail: palomba@ifi.uzh.ch

Francesca Arcelli Fontana
University of Milano-Bicocca, Italy
E-mail: arcelli@disco.unimib.it

Andy Zaidman
Delft University of Technology, The Netherlands
E-mail: a.e.zaidman@tudelft.nl

a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change Prediction · Code Smells · Empirical Study

1 Introduction

During maintenance and evolution, software systems are continuously modified in order to adapt them to changing needs (e.g., new platforms), improve their performance, or rid them from potential bugs [70]. As a consequence, they become more complex, possibly eroding the original design with a subsequent reduction in their overall maintainability [102]. In this context, predicting the low-quality source code components having a higher likelihood to change in the future represents an important activity to enable developers to plan preventive maintenance operations such as refactoring [44, 3, 145] or peer-code reviews [9, 103, 15, 104]. For this reason, the research community proposed several approaches in order to enable developers to control these changes [28, 36, 37, 47, 66, 71, 72, 110, 130, 151]. Such approaches are based on machine learning models which exploit several predictors capturing product, process, and developer-related features of classes.

Despite the good performance that these existing models have shown, recent studies [63, 96] have explored new factors contributing to the change-proneness of classes, finding that it is strongly influenced by the presence of so-called bad code smells [44], i.e., sub-optimal design and/or implementation choices adopted by practitioners during software development and maintenance. Specifically, such studies have shown that smelly classes are significantly more likely to be the subject of changes than classes not affected by any design problem. Nevertheless, most of the changes done on these smelly classes do not pertain to code smell refactoring [13, 99, 140].

Based on these findings, we empirically investigate the extent to which smell-related information can actually be useful when considered in the context of the prediction of change-prone classes, i.e., the prediction of a binary value indicating whether a class is likely to be frequently changed in the future: our conjecture is that *the addition of a measure of code smell severity can improve the performance of existing change prediction models, as it may help in the correct assessment of the change-proneness of smelly classes*. For severity, we mean a metric able to quantify how much a certain code smell instance is harmful for the design of a source code class. To test our conjecture, we (i) add the *intensity index* defined by Arcelli Fontana et al. [41] in three state of the art change prediction models based on product [151], process [36], and developer-related metrics [28] and (ii) evaluate how much the new models, including the intensity index, overcome the prediction capabilities of the baseline models on 43 releases of 14 large systems. We also evaluate whether the new change prediction models including the intensity index have better change prediction than models built with alternative smell-related information such as

the antipattern metrics defined by Taba et al. [128]. These metrics are able to capture historical information on code smell instances (e.g., the recurrence of a certain instance over time). The results show that the addition of the intensity index significantly improves the performance of the baseline change prediction models. Moreover, models including the intensity index have a higher accuracy than the models including the alternative antipattern metrics.

We also observe that intensity and antipattern metrics are orthogonal, i.e., the predictions made by the two models correctly identify the change-proneness of different smelly classes. Given such an orthogonality, we then further explore the possibility to improve change prediction models by devising a smell-aware combined approach that mixes together the features of the models used, i.e., structural, process, developer-, and smell-related information, with the aim of boosting the change-proneness prediction abilities. As a result, we discovered that such a combined model overcomes the performance of the other experimented models. To sum up, the contributions of this paper are the following:

1. A large-scale empirical assessment of the role of the intensity index [41] when predicting change-prone classes;
2. An empirical comparison between the capabilities of the intensity index and the antipattern metrics defined by Taba et al. [128] in the context of change prediction;
3. A novel smell-aware combined change prediction model, which has better prediction performance than the other experimented models;
4. A replication package that includes all the raw data and working data sets of our study [27].

Structure of the paper. Section 2 discusses the related literature on change prediction models and code smells. Section 3 describes the design of the case study aimed at evaluating the performance of the models, while Section 4 reports the results achieved. Section 5 discusses the threats to the validity of our empirical study. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Related Work

Change-prone classes represent source code components that, for different reasons, tend to change more often than others. This phenomenon has been widely investigated by the research community [63, 64, 34, 19, 124, 96, 123] with the aim of studying the factors contributing to the change-proneness of classes. Among all these studies, Khomh et al. [63] showed that the presence of code smells makes the affected classes more change-prone than non-smelly classes. The results were later confirmed by several studies in the field [30, 86, 96, 125], that pointed out how code smells represent a factor making classes more change-prone. For instance, Palomba et al. [96] showed that classes affected by code smells have a statistically significant higher change-proneness with

respect to classes not affected by code smells. Our work is clearly based on these findings, and aims at providing additional evidence of how code smells can be adopted to classify change-prone classes.

Motoring the classes that are more prone to change might help developers to be aware of the presence of something wrong that might require some preventive maintenance operations (e.g., code review [9] or refactoring [44]) aimed at improving the quality of the source code. In this regard, previous researchers have intensely investigated the feasibility of machine learning techniques for the identification of change-prone classes [28, 36, 37, 47, 66, 71, 72, 110, 151]. In the following, we discuss the advances achieved in the context of change prediction models. At the same time, as this paper reports on the role of *code smell intensity*, we also summarize the literature related to the detection and prioritization of code smells.

2.1 Change Prediction Approaches

The most relevant body of knowledge related to change prediction techniques is represented by the use of product and process metrics as independent variables able to characterize the change-proneness of software artifacts [2, 7, 23, 71, 72, 73, 137, 151]. Specifically, Romano et al. [110] relied on code metrics for predicting change-prone so-called fat interfaces (i.e., poorly-cohesive Java interfaces), while Eski et al. [37] proposed a model based on both CK and QMOOD metrics [11] to estimate change-prone classes and to determine parts of the source code that should be tested first and more deeply.

Conversely, Elish et al. [36] reported the potential usefulness of process metrics for change prediction. In particular, they defined a set of *evolution* metrics that describe the historical characteristics of software classes: for instance, they defined metrics like the birth date of a class or the total amount of changes applied in the past. As a result, their findings showed that a prediction model based on those evolution metrics can overcome the performance of structural-based techniques. These results were partially confirmed by Girba et al. [47], who defined a tool that suggests change-prone code elements by summarizing previous changes. In a small-scale empirical study involving two systems, they observed that previous changes can effectively predict future modifications.

More recently, Catolino et al. [28] have empirically assessed the role of developer-related factors in change prediction. To this aim, they have studied the performance of three developer-based prediction models relying on (i) entropy of development process [53], (ii) number of developers working on a certain class [14], and (iii) structural and semantic scattering of changes [32], showing that they can be more accurate than models based on product or process metrics. Furthermore, they have also defined a combined model which considers a mixture of metrics and that has shown to be up to 22% more accurate than the previously defined ones.

Our work builds upon the findings reported above. In particular, we study to what extent the addition of information related to the presence and severity of code smells can contribute to the performance of change prediction models based on product, process, and developer-based metrics.

Another consistent part of the state of the art concerns with the use of alternative methodologies to predict change-prone classes. For instance, the combination of (i) dependencies mined from UML diagrams and code metrics [51, 52, 112, 117, 118], and (ii) genetic and learning algorithms [74, 77, 105] have been proposed. Finally, some studies focus on the adoption of ensemble techniques for change prediction [25, 58, 67, 75]. In particular, Malhotra and Khanna [75] have proposed a search-based solution to the problem, adopting a Particle Swarm Optimization (PSO)-based classifier [58] for predicting the change-proneness of classes. Malhotra and Khanna have conducted their study on five Android application packages and the results encouraged the use of the adopted solution for developing change prediction models. Kumar et al. [67] have studied the correlation between 62 software metrics and the likelihood of a class to change in the future. Afterwards, they build a change prediction model considering eight different machine learning algorithms and two ensemble techniques. The results have shown that with the application of feature selection techniques, the change prediction models relying on ensemble classifiers can obtain better results. These results were partially contradicted by Catolino and Ferrucci [25, 26], who empirically compared the performance of three ensemble techniques (i.e., Boosting, Random Forest, and Bagging) with the one of standard machine learning classifiers (e.g., , Logistic Regression) on eight open source systems. The key results of the study showed how ensemble techniques in some cases perform better than standard machine learning approaches, however the differences among them is generally small.

2.2 Code Smell Detection and Prioritization

Fowler defined “bad code smells” (abbreviated as, “code smells” or simply “smells”) as *“symptoms of the presence of poor design or implementation choices applied during the development of a software system”* [44]. Starting from there, several researchers heavily investigated (i) how code smells evolve over time [99, 98, 107, 138, 139, 140], (ii) the way developers perceive them [92, 129, 148], and (iii) what is their impact on non-functional attributes of source code [1, 46, 61, 63, 89, 96, 122, 147, 90]. All these studies came up with a shared conclusion: code smells negatively impact program comprehension, maintainability of source code, and development costs. In the scope of this paper, the most relevant empirical studies are those reported by Khomh et al. [63] and Palomba et al. [96], who explicitly investigated the impact of code smells on software change proneness. Both the studies have reported that classes affected by design flaws tend to change more frequently than classes that are not affected by any code smell. Moreover, refactoring practices notably help in keeping under control the change-proneness of classes. These

studies motivate our work: indeed, following the findings on the influence of code smells, we believe that the addition of information coming from the analysis of the severity of code smells can positively improves the performance of change prediction models. As explained later in Section 5, we measure the intensity rather than the simple presence/absence of smells because a severity metric can provide us with a more fine-grained information on how much a design problem is "dangerous" for a class.

Starting from the findings on the negative impact of code smells on source code maintainability, the research community has heavily focused on devising techniques able to automatically detect code smells. Most of these approaches rely on a two-steps approach [62, 68, 78, 82, 84, 87, 136]: in the first one, a set of structural code metrics are computed and compared against predefined thresholds; in the second one, these metrics are combined using operators in order to define detection rules. If the logical relationships expressed in such detection rules are violated, a code smell is identified. While these approaches already have good performance, Arcelli Fontana et al. [43] and Aniche et al. [4] proposed methods to further improve it by discarding false positive code smell instances or tailoring the thresholds of code metrics, respectively.

Besides structural analysis, the use of alternative sources of information for smell detection has been proposed. Ratiu et al. [109] and Palomba et al. [91, 93] showed how historical information can be exploited for detecting code smells. These approaches are particularly useful when dealing with design issues arising because of evolution problems (e.g., how a hierarchy evolves over time). On the other hand, Palomba et al. [95, 101] have adopted Information Retrieval (IR) methods [10] to identify code smells characterized by promiscuous responsibilities (*Blob* classes).

Furthermore, Arcelli Fontana et al. [6, 40] and Kessentini et al. [21, 59, 60, 113] have used machine learning and search-based 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. Nevertheless, recent findings [33, 8] have shown that the performance of such techniques may vary depending on the exploited dataset.

Finally, Morales et al. [83] proposed a developer-based approach that leverages contextual information on the task a developer is currently working on to recommend what are the smells that can be removed in the portion of source code referring to the performed task, while Palomba et al. [100] have proposed community smells, i.e., symptoms of the presence of social debt, as additional predictors of the code smell severity.

In parallel with the definition of code smell detectors, several researchers faced the problem of prioritizing code smell instances based on their harmfulness for the overall maintainability of a software project. Vidal et al. [144] developed a semi-automated approach that recommends a ranking of code smells based on (i) past component modifications (e.g., number of changes during the system history), (ii) important modifiability scenarios, and (iii) relevance of the kind of smell as assigned by developers. In a follow-up work, the same

authors introduced a new criteria for prioritizing groups of code anomalies as indicators of architectural problems in evolving systems [143].

Lanza and Marinescu [68] have proposed a metric-based rules approach in order to detect code smells, or identify code problems called disharmonies. The classes (or methods) that contain a high number of disharmonies are considered more critical. Marinescu [79] has also presented the *Flaw Impact Score*, i.e., a measure of criticality of code smells that considers (i) negative influence of a code smell on coupling, cohesion, complexity, and encapsulation; (ii) granularity, namely the type of component (method or a class) that a smell affects; and (iii) severity, measured by one or more metrics analyzing the critical symptoms of the smell.

Murphy-Hill and Black [85] have introduced an interactive visualization environment aimed at helping developers when assessing the harmfulness of code smell instances. The idea behind the tool is to visualize classes like petals, and a higher code smell severity is represented by a larger petal size. Other studies have exploited developers' knowledge in order to assign a level of severity with the aim to suggest relevant refactoring solutions [81], while Zhao and Hayes [150] have proposed a hierarchical approach to identify and prioritize refactoring operations based on predicted improvement to the maintainability of the software.

Besides the prioritization approaches mentioned above, more recently Arcelli Fontana and Zanoni [39] have proposed the use of machine learning techniques to predict code smell severity, reporting promising results. The same authors have also proposed JCodeOdor [41], a code smell detector that is able to assign a level of severity by computing the so-called *intensity index*, i.e., the extent to which a set of structural metrics computed on smelly classes exceed the predefined thresholds: the higher the distance between the actual and the threshold values the higher the severity of a code smell instance. As explained later (Section 3), in the context of our study we adopt JCodeOdor since it has previously been evaluated on the dataset we exploited, reporting a high accuracy [41]. This is therefore the best option we have to conduct our work.

3 Research Methodology

In this section, we present the empirical study definition and design that we follow to assess the addition of the code smell intensity index to existing change prediction models.

3.1 Research Questions

The *goal* of the empirical study is to evaluate the contribution of the *intensity* index in prediction models aimed at discovering change-prone classes, with the *purpose* of understanding how much the allocation of resources in preventive

Table 1: Characteristics of the Software Projects in Our Dataset.

System	Releases	Classes	KLOCs	% Change Cl.	% Smelly Cl.
Apache Ant	5	83-813	20-204	24	11-16
Apache Camel	4	315-571	70-108	25	9-14
Apache Forrest	3	112-628	18-193	64	11-13
Apache Ivy	1	349	58	65	12
Apache Log4j	3	205-281	38-51	26	15-19
Apache Lucene	3	338-2,246	103-466	26	10-22
Apache Pbeans	2	121-509	13-55	37	21-25
Apache POI	4	129-278	68-124	22	15-19
Apache Synapse	3	249-317	117-136	26	13-17
Apache Tomcat	1	858	301	76	4
Apache Velocity	3	229-341	57-73	26	7-13
Apache Xalan	4	909	428	25	12-22
Apache Xerces	3	162-736	62-201	24	5-9
JEdit	5	228-520	39-166	23	14-22
Overall	**43**	**24,630**	**84,612**	**26**	**15**

maintenance tasks such as code inspection [9] or refactoring [44] might be improved in a real use case. The *quality focus* is on the prediction performance of models that include code smell-related information when compared to state of the art, while the *perspective* is that of researchers, who want to evaluate the effectiveness of using information about code smells when identifying change-prone components. More specifically, the empirical investigation aims at answering the following research questions:

- $\mathbf{RQ}_1$. *To what extent does the addition of the intensity index as additional predictor improve the performance of existing change prediction models?*
- $\mathbf{RQ}_2$. *How does the model including the intensity index as predictor compare to a model built using antipattern metrics?*
- $\mathbf{RQ}_3$. *What is the gain provided by the intensity index to change prediction models when compared to other predictors?*
- $\mathbf{RQ}_4$. *What is the performance of a combined change prediction model that includes smell-related information?*

As detailed in the next sections, the first research question ($\mathbf{RQ}_1$) is aimed at investigating the contribution given by the intensity index within change prediction models built using different types of predictors, i.e., product, process, and developer-related metrics. In $\mathbf{RQ}_2$ we empirically compare models relying on two different types of smell-related information, i.e., the intensity index [41] and the *antipattern metrics* proposed by Taba et al. [128]. $\mathbf{RQ}_3$ is concerned with a fine-grained analysis aimed at measuring the actual gain provided by the addition of the intensity metric within different change prediction models. Finally, $\mathbf{RQ}_4$ has the goal to assess the performance of a change prediction model built using a combination between smell-related information and other product, process, and developer-related features.

3.2 Context Selection

The *context* of the study is represented by the set of systems reported in Table 1. Specifically, we report (i) the name of the considered projects, (ii) the number of releases 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 change-prone classes (identified as explained later), and (iv) the percentage (min-max) of classes affected by design problems (detected as explained later). Overall, the dataset contains 43 releases of 14 projects, accounting for a total of 24,630 source code files.

We have built this in three steps. First, we have exploited the dataset made available by Jureczko and Madeyski [56], which contains (i) meta-information (e.g., links to GitHub repositories) and (ii) the values of 20 code metrics (some of those were later used to build the structural baseline model - see Section 3.3.2). The selection of this dataset is driven by its availability and by the fact that some metrics are already available; moreover, the dataset contains information related to several releases as well as meta-information that has eased the mining process aimed at enriching the dataset with further metrics and change-proneness information.

On top of the dataset by Jureczko and Madeyski [56], for each release we have then computed other metrics required to answer our research questions, as described in Section 3.3.2, i.e., (i) product metrics of the structural model that are not available in the original dataset, (ii) process metrics of the model by Elish et al. [36], and (iii) scattering metrics of the model by Di Nucci et al. [32]. Furthermore, we run the identification of the change-prone classes, as indicated by Romano et al. [110] (see Section 3.3.3).

When computing the additional metrics we used the meta-information contained in the dataset by Jureczko and Madeysky [56] to make sure to use exactly the same set of classes. This meta-information contains the full qualifiers of all classes of the considered systems (for all the releases). Using such meta-information, we could precisely retrieve the classes on which to apply our measurements. It is important to note that the starting dataset of Jureczko and Madeyski [56] was originally hosted on the Promise repository [80][1]. As reported by Shepperd et al. [119], such dataset may contain noise and/or erroneous entries that possibly negatively influence the results. To account for this aspect, before running our experiments we have performed a data cleaning on the basis of the algorithm proposed by Shepperd et al. [119], which consists of 13 corrections able to remove identical features, features with conflicting or missing values, etc. During this step, we have removed 58 entries from the original dataset. To enable the replication of our work, we have made available the entire enlarged dataset that we have built in the context of our study [27].

As for the code smells, our investigation copes with six types of design problems, namely:

[1] Up to date, the dataset is not available anymore on the repository.

- *God Class* (a.k.a., *Blob*): A poorly cohesive class that implements different responsibilities;
- *Data Class*: A class whose only purpose is holding data;
- *Brain Method*: A large method implementing more than one function, being therefore poorly cohesive;
- *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 of the project;
- *Message Chains*: A method containing a long chain of method calls.

The choice of focusing on these specific smells is driven by two main aspects: (i) on the one hand, we have taken into account code smells characterizing different design problems (e.g., excessive coupling vs poorly cohesive classes/methods) and having different granularities; (ii) on the other hand, as explained in the next section, we can rely on a reliable tool to properly identify and compute their intensity in the classes of the exploited dataset.

3.3 $\mathbf{RQ}_1$ - The contribution of the Intensity Index

The goal of the first research question is aimed at investigating the contribution given by the intensity index within change prediction models built using different types of predictors, i.e., product, process, and developer-related metrics. To answer our first research question, we need to (i) identify code smells in the subject projects and compute their intensity, and (ii) select a set of existing change prediction models to which to add the information on the intensity of code smells. Furthermore, we proceed with the training and testing of the built change prediction models. The following subsections detail the process that we have conducted to perform such steps.

3.3.1 Code Smell Intensity Computation

To compute the severity of code smells in the context of our work we have used JCodeOdor [41], a publicly available tool that is able to both identify code smells and assign them a degree of severity by computing the so-called *intensity index*. Such an index is represented by a real number contained in the range [0, 10]. Our choice to use this tool was driven by the following observations. JCodeOdor works with all the code smells that we consider in our work. Moreover, it is fully automated, meaning that it does not require any human intervention while computing the intensity of code smells. Finally, it is highly accurate: in a previous work by Palomba et al. [97] the tool has been empirically assessed on the same dataset adopted in our context, showing an F-Measure of 80%. For these reasons, we believe that JCodeOdor was the best option we had to conduct our study.

Table 2: Code Smell Rules Strategies (the complete names of the metrics are given in Table 3 and the explanation of the rules in Table 4)

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))

From a technical point of view, given the set of classes composing a certain software system the tool performs two basic steps to compute the intensity of code smells:

1. *Detection Phase.* Given a software system as input, the tool starts by detecting code smells relying on the detection rules reported in Table 2. Basically, each rule is represented by a logical composition of predicates, and each predicate is based on an operator that compares a metric with a threshold [68, 94]. Such detection rules are similar to those defined by Lanza and Marinescu [68], who have used the set of code metrics described in Table 3 to identify the six code smell types in our study. To ease the comprehension of the detection approach, Table 4 describes the rationale behind these detection rules.
   A class/method of a project is marked as *smelly* if one of the logical propositions shown in Table 2 is true, i.e., if the actual metrics computed on the class/method exceed the threshold values defined in the detection strategy. It is worth pointing out that the thresholds used by JCodeOdor have been empirically calibrated on 74 systems belonging to the Qualitas Corpus dataset [134] and are derived from the statistical distribution of the metrics contained in the dataset [41, 42].
2. *Intensity Computation.* If a class/method is identified by the tool as smelly, 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 [0,10] using a min-max normalization process [135]: specifically, this is a feature scaling technique where the values of a numeric range are reduced to a scale between 0 and 10. To compute z, i.e., the normalized value, the following formula is applied:

Table 3: 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 measured 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.
*NOMNAMM	Number of Not Accessor or Mutator Methods	The number of methods defined locally in a class, counting public as well as private methods, excluding 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.

Table 4: 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 excludes them from the computation.
	NOMNAMM $\geq$ HIGH	*Implements a high number of functions.* We exclude getter and setter because we consider only the methods that effectively implement functionality of the class.
	TCC $\leq$ LOW	*Functions 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.

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

where min and max are the minimum and maximum values observed in the distribution. This step enables 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.

3.3.2 Selection of Basic Prediction Models

Our conjecture is concerned with the gain given by the addition of information on the intensity of code smells within existing change prediction models. To test such a conjecture, we need to identify the state of the art techniques to which to add the intensity index: we have selected three models based on product, process, and developer-related metrics that have been shown to be accurate in the context of change prediction [28, 32, 36, 151].

Product Metrics-based Model. The first baseline is represented by the change prediction model devised by Zhou et al. [151]. It is composed of a set of metrics computed on the basis of the structural properties of source code: these are cohesion (i.e., the Lack of Cohesion of Method — LCOM), coupling (i.e., the Coupling Between Objects — CBO — and the Response for a Class — RFC), and inheritance metrics (i.e., the Depth of Inheritance Tree — DIT). To actually compute these metrics, we rely on a publicly available and widely used tool originally developed by Spinellis [126]. In the following, we refer to this model as SM, i.e., *Structural Model*.

Process Metrics-based Model. In their study, Elish et al. [36] have reported that process metrics can be exploited as better predictors of change-proneness with respect to structural metrics. For this reason, our second baseline is the Evolution Model (EM) proposed by Elish et al. [36]. More specifically, this model relies on the metrics shown in Table 5, which capture different aspects of the evolution of classes, e.g., the weighted frequency of changes or the first time changes introduced. To compute these metrics, we have adopted the publicly available tool that was previously developed by Catolino et al. [28]. In the following, we refer to this model as PM, i.e., *Process Model*.

Developer-Related Model. In our previous work [28], we have demonstrated how developer-related factors can be exploited within change prediction models since they provide orthogonal information with respect to product and process metrics that takes into account how developers perform modifications and how complex the development process is. Among the developer-based models available in the literature [14, 32, 53], in this paper we rely on the Developer Changes Based Model (DCBM) devised by Di Nucci et al. [32], as it has been shown to be the most effective one in the context of change prediction. Such a model uses as predictors the so-called structural and semantic

Table 5: Independent variables considered by Elish et al.

Acronym	Metric
BOC	Birth of a Class
FCH	First Time Changes Introduced to a Class
FRCH	Frequency of Changes
LCH	Last Time Changes Introduced to a Class
WCD	Weighted Change Density
WFR	Weighted Frequency of Changes
TACH	Total Amount of Changes
ATAF	Aggregated Change Size Normalized by Frequency of Change
CHD	Change Density
LCA	Last Change Amount
LCD	Last Change Density
CSB	Changes since the Birth
CSBS	Changes since the Birth Normalized by Size
ACDF	Aggregated Change Density Normalized by Frequency of Change
CHO	Change Occurred

scattering of the developers that worked on a code component in a given time period α. Specifically, for each class c, the two metrics are computed as follows:

$$\text{StrScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} \text{StrScat}_{d,\alpha} \tag{2}$$

$$\text{SemScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} \text{SemScat}_{d,\alpha} \tag{3}$$

where $developers_{c,\alpha}$ represents the set of developers that worked on the class c during a certain period α, and the functions $StrScat_{d,\alpha}$ and $SemScat_{d,\alpha}$ return the structural and semantic scattering, respectively, of a developer d in the time window α. Given the set $CH_{d,\alpha}$ of classes changed by a developer d during a certain period α, the formula of structural scattering of a developer is:

$$\text{StrScat}_{d,\alpha} = |CH_{d,\alpha}| \times \underset{\forall c_i, c_j \in CH_{d,\alpha}}{average} [dist(c_i, c_j)] \tag{4}$$

where $dist$ is the distance in number of packages from class c_i to class c_j. The structural scattering is computed by applying the shortest path algorithm on the graph representing the system's package structure. The higher the measure, the higher the estimated developer's scattering, this means that if a developer applies several changes in different packages in a certain time period, the value of structural scattering will be high.

Regarding the semantic scattering of a developer, it is based on the textual similarity of the classes changed by a developer in a certain period α and it is

computed as:

$$\text{SemScat}_{d,\alpha} = |CH_{d,\alpha}| \times \frac{1}{\underset{\forall c_i, c_j \in CH_{d,\alpha}}{average} [sim(c_i, c_j)]} \quad (5)$$

where the *sim* function returns the textual similarity between classes c_i and c_j according to the measurement performed using the *Vector Space Model* (VSM) [10]. The metric ranges between zero (no textual similarity) and one (if the representation of two classes using VSM is equal). Specifically, a developer can apply several changes within the same package (obtaining a value of structural scattering equal to 0), but could have a high value of semantic scattering since there is a low textual similarity between the pairs of classes contained in this package (where the developer has applied changes).

In our study, we set the parameter α of the approach as the time window between two releases $R-1$ and R, as done in previous work [97]. To compute the metrics, we rely on the implementation provided by Di Nucci et al. [32].

It is important to note that all the baseline models, including the model to which we have added the intensity index, might be affected by multi-collinearity [88], which occurs when two or more independent variables are highly correlated and can be predicted one from the other, thus possibly leading to a decrease of the prediction capabilities of the resulting model [120, 132]. For this reason, we decided to use the *Vif* (Variance inflation factors) function [88] implemented in R[2] to discard redundant variables. *Vif* is based on the square of the multiple correlation coefficient resulting from regressing a predictor variable against all other predictor variables. If a variable has a strong linear relationship with at least one other variable, the correlation coefficient would be close to 10, and VIF for that variable would be large. As indicated by previous work [35, 149], a VIF greater than 10 is a signal that the model has a collinearity problem. 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 enabling the identification of the metric that is better to drop from the model [88].

3.3.3 Dependent Variable

Our dependent variable is represented by the actual change-proneness of the classes in our dataset. As done in most of the previous work in literature [28, 36, 151], we have adopted a within-project strategy, meaning that we compute the change-proneness of classes for each project independently. To compute it, we have followed the guidelines provided by Romano et al. [110], who consider a class as change-prone if, in a given time period TW, it underwent a number of changes higher than the median of the distribution of the

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

Table 6: Changes extracted by ChangeDistiller while computing the change-proneness. '✓' symbols indicate the types we considered in our study.

ChangeDistiller	Our Study
Statement-level changes	
Statement Ordering Change	✓
Statement Parent Change	✓
Statement Insert	✓
Statement Delete	✓
Statement Update	✓
Class-body changes	
Insert attribute	✓
Delete attribute	✓
Declaration-part changes	
Access modifier update	✓
Final modifier update	✓
Declaration-part changes	
Increasing accessibility change	✓
Decreasing accessibility change	✓
Final Modified Insert	✓
Final Modified Delete	✓
Attribute declaration changes	
Attribute type change	✓
Attribute renaming change	✓
Method declaration changes	
Return type insert	✓
Return type delete	✓
Return type update	✓
Method renaming	✓
Parameter insert	✓
Parameter delete	✓
Parameter ordering change	✓
Parameter renaming	✓
Class declaration changes	
Class renaming	✓
Parent class insert	✓
Parent class delete	✓
Parent class update	✓

number of changes experienced by all the classes of the system. In particular, for each pair of commits (c_i, c_{i+1}) of TW we run ChangeDistiller [38], a tree differencing algorithm able to extract the fine-grained code changes between c_i and c_{i+1}. Table 6 reports the entire list of change types identified by the tool. As it is possible to observe, we have considered all of them while computing the number of changes. It is worth mentioning that the tool ignores white space-related differences and documentation-related updates: in this way, it only considers the changes actually applied on the source code. More importantly, ChangeDistiller is able to identify rename refactoring operations: this means that we could handle cases where a class was modified during the change history, thus not biasing the correct counting of the number of changes. In our study, the time window TW represents the time between two subsequent releases. While for most of the projects in our dataset we have more than one release, for Apache Ivy and Apache Tomcat we only have one release. In these cases, we computed the dependent variable looking at the time window between the versions used in the study and the corresponding subsequent versions of the systems, so that we could treat these systems exactly in the same way as the others contained in the dataset and compute

the change-proneness values. Note that, as for the independent variables, we compute them considering the release *before* the one where the dependent variable was computed, i.e., we computed the independent variables between the releases R_{i-1} and R_i, while the change-proneness was computed between R_i and R_{i+1}: in this way, we avoid biases due to the computation of the change-proneness in the same periods as the independent ones. The dataset with the oracle is available in the online appendix [27].

3.3.4 Experimented Machine Learning Models

To answer $\mathbf{RQ}_1$, we built two prediction models for each baseline: the first *does not include* the intensity index as predictor, thus relying on the original features only; the second model *includes* the intensity index as an additional predictor. Using this procedure, we experiment with 6 different models, and we can control the actual amount of improvement given by the intensity index with respect to the baselines (if any). It is worth remarking that, for *non-smelly* classes, the intensity value is set to 0.

3.3.5 Classifier Selection

Different machine learning classifiers have been proposed in the literature to distinguish change-prone and non-change-prone classes (e.g., Romano and Pinzger [110] have adopted Support Vector Machines [20], while Tsantalis et al. [137] have relied on Logistic Regression [69]): based on the results of previous studies, there seems not to be a classifier that provides the best overall solution for all situations. For this reason, in our work we experimented the different change prediction models with different classifiers, i.e., ADTree [146], Decision Table Majority [65], Logistic Regression [29], Multilayer Perceptron [111], Naive Bayes [55], and Simple Logistic Regression[106]. Overall, for all considered models the best results in terms of F-measure (see Section 3.3.7 for the evaluation metrics) are obtained using the Simple Logistic Regression. In the remaining of the paper, we only report the results that we have obtained when using this classifier, while a complete report of the performance of other classifiers is available online [27].

3.3.6 Validation Strategy

As for validation strategy we adopt *10-Fold Cross Validation* [127]. Using it, each experimented prediction model has been trained and evaluated as follows. For each project considered in our study, the validation methodology randomly partitions the available set of data into 10 folds of equal size, applying a stratified sampling, i.e., all the folds have a similar proportion of change- and non-change-prone classes. Then, a single fold is used as test set, while the remaining ones are used to train the change prediction model under examination. The process is repeated 10 times, using each time a different fold as

test set: therefore, at the end of the process the experimented model outputs a prediction for each class of each project. Finally, the performance of the model is reported using the mean achieved over the ten runs. It is important to note that we have repeated the 10-fold validation 100 times (each time with a different seed) to cope with the randomness arising from using different data splits [50].

3.3.7 Evaluation Metrics

To measure and compare the performance of the models, we compute two well-known metrics such as *precision* and *recall* [10], which are defined as follow:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + TN} \tag{6}$$

where TP is the number of true positives, TN the number of true negatives, and FP the number of false positives. To have a unique value representing the goodness of the model, we compute the F-Measure, i.e., the harmonic mean of precision and recall:

$$F\text{-}Measure = 2 * \frac{precision * recall}{precision + recall} \tag{7}$$

Moreover, we consider another indicator: the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) metric [22]. A ROC curve is a graph showing the performance of a classification model at all classification thresholds. The AUC measures the entire two-dimensional area underneath the entire ROC curve. This metrics quantifies the overall ability of a change prediction model to discriminate between change-prone and non-change-prone classes: the closer the AUC-ROC to 1 the higher the ability of the classifier, while the closer the AUC-ROC to 0.5 the lower its accuracy. In other words, this metric can quantify how robust the model is when discriminating the two binary classes.

In addition, we compare the performance achieved by the experimented prediction models from a statistical point of view. As we need to perform multiple comparisons among the performance of the considered models over multiple datasets, exploiting well-known and widely-adopted tests like, for instance, the Mann-Whitney test [76] is not recommended because of two main reasons: (i) the performance of a machine learner can vary between one dataset and another [131, 133]; (ii) the interpretation of the results might be biased because of overlapping problems, in other words the possibility for one or more treatments to be classified in more than one group: this aspect might cause serious issues for the experimenter to really distinguish the real groups to which the means should belong [115]. To overcome these problems, we exploit the Scott-Knott Effect Size Difference test [133].

This is an effect-size aware version of the original Scott-Knott test [115]. More specifically, this algorithm implements a two-step analysis: first, it hierarchically clusters treatments into distinct groups, meaning that there is no

possibility of one or more treatments to be classified in more than one group; secondly, the means of the clusters are compared to understand whether they are statistically different. The major benefits provided by the application of the effect-size aware test designed by Tantithamthavorn et al. [133] are that the algorithm (i) hierarchically clusters the set of treatment means into statistically distinct groups, (ii) corrects the non-normal distribution of a dataset if needed, and (iii) merges two statistically distinct groups in case their effect size—measured using Cliff's Delta (or d) [48]—is negligible, so that the creation of trivial groups is avoided. To perform the test, we rely on the implementation[3] provided by Tantithamthavorn et al. [133]. It is important to note that recently Herbold [54] has discussed the implications as well as the impact of the normality correction of Scott-Knott ESD test, concluding that this correction does not necessarily lead to the fulfilment of the assumptions of the original Scott-Knott test and may cause problems with the statistical analysis. The author has also proposed a modification to the original implementation of the test that can overcome the identified problems. In response to these comments, Tantithamthavorn et al. have followed the recommendations provided and modified the implementation of the Scott-Knott ESD test: we made sure to use the latest version of the algorithm, available in the version `v1.2.2` of the `R` package.

3.4 $\mathbf{RQ_2}$ - Comparison between Intensity Index and Antipattern Metrics

In $\mathbf{RQ_2}$ our goal is to compare the performance of change prediction models relying on the intensity index against the one achieved by models exploiting other existing smell-related metrics. In particular, the comparison is done considering the so-called *antipattern* metrics, which have been defined by Taba et al. [128]: these metrics aimed at capturing different aspects related to the maintainability of classes affected by code smells. More specifically:

- the Average Number of Antipatterns (ANA) computes how many code smells were in the previous releases of a class over the total number of releases. This metric is based on the assumption that classes that have been more prone to be smelly in the past are somehow more prone to be smelly in the future. More formally, ANA is computed as follows. For each file $f \in S$ (System):

$$ANA(f) = \frac{1}{n} * \sum_{i=1}^{n} N_{AP}(f^i), \tag{8}$$

  where $N_{AP}(f^i)$ represents the total number of antipatterns in past change prone version f^i ($i \in \{1...n\}$), and n is the total number of versions in the history of f and $f = f^n$.

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

- the Antipattern Complexity Metric (ACM) computes the entropy of changes involving smelly classes. Such entropy refers to the one originally defined by Hassan [53] in the context of defect prediction. The conjecture behind its use relates to the fact that a more complex development process might lead to the introduction of code smells. The specific formula leading to its computation is:

$$ACM(f) = \sum_{i=1}^{n} p(f^i) * H^i, \quad (9)$$

where H^i represents the Shannon entropy as computed by Hassan [53], p is the probability of having antipatterns in file f, and n is the total number of versions in the history of f and $f = f^n$.
- the Antipattern Recurrence Length (ARL) measures the total number of subsequent releases in which a class has been affected by a smell. This metric relies on the same underlying conjecture as ANA, i.e., the more a class has been smelly in the past the more it will be smelly in the future. Formally:

$$ARL(f) = rle(f) * e^{\frac{1}{n}*(c(f)+b(f))}, \quad (10)$$

where n is the total number of versions in the history of f, $c(f)$ is the number of *"changy"* versions in the history of file f in which f has at least one antipattern, $b(f) < n$ is the ending index of the longest consecutive stream of antipatterns in *"changy"* versions of f, and $rle(f)$ is the maximum length of the longest consecutive stream of antipatterns in the history of f.

To compute these metrics, we have employed the tool developed and made available by Palomba et al. [97]. Then, as done in the context of $\mathbf{RQ}_1$, we plugged the *antipattern* metrics into the experimented baselines, applying the *Variance inflation factor (Vif)* and assessing the performance of the resulting change prediction models using the same set of evaluation metrics described in Section 3.3.7, i.e., F-Measure and AUC-ROC. Finally, we statistically compare the performance with the one obtained by the models including the intensity index as predictor.

Besides the comparison in terms of evaluation metrics, we also analyze the extent to which the two types of models are orthogonal with respect to the classification of change-prone classes. This was done with the aim of assessing whether the two models, relying on different smell-related information, can correctly identify the change-proneness of different classes. More formally, 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 compute the following overlap metrics on the set of *smelly and change-prone* instances of each system:

$$TP_{m_{int} \cap m_{ant}} = \frac{|TP_{m_{int}} \cap TP_{m_{ant}}|}{|TP_{m_{int}} \cup TP_{m_{ant}}|}\% \quad (11)$$

$$TP_{m_{int} \setminus m_{ant}} = \frac{|TP_{m_{int}} \setminus TP_{m_{ant}}|}{|TP_{m_{int}} \cup TP_{m_{ant}}|}\% \qquad (12)$$

$$TP_{m_{ant} \setminus m_{int}} = \frac{|TP_{m_{ant}} \setminus TP_{m_{int}}|}{|TP_{m_{ant}} \cup TP_{m_{int}}|}\% \qquad (13)$$

where $TP_{m_{int}}$ represents the set of change-prone classes correctly classified by the prediction model m_{int}, while $TP_{m_{ant}}$ is the set of change-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 change-prone classes correctly classified by m_{int} only and missed by m_{ant}, and $TP_{m_{ant} \setminus m_{int}}$ measures the percentage of change-prone classes correctly classified by m_{ant} only and missed by m_{int}.

3.5 **RQ$_3$** - Gain Provided by the Intensity Index

The goal of this question is to analyze the actual gain provided by the addition of the intensity metric within different change prediction models. To this aim, we conduct a *fine-grained* investigation aimed at measuring how important the intensity index is with respect to other features (i.e., product, process, developer-related, and *antipattern* metrics) composing the considered models. We use an *information gain* algorithm [108] to quantify the gain provided by adding the intensity index in each prediction model. In our context, this algorithm ranks the features of the models according to their ability to predict the change-proneness of classes. More specifically, let M be a change prediction model, let $P = \{p_1, \ldots, p_n\}$ be the set of predictors composing M, an *information gain* algorithm [108] 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_i:

$$InfoGain(M, p_i) = H(M) - H(M|p_i) \qquad (14)$$

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:

$$H(M) = -\sum_{i=1}^{n} prob(p_i) \log_2 prob(p_i) \qquad (15)$$

From a more practical perspective, the algorithm quantifies how much uncertainty in M is reduced after splitting M on predictor p_i. In our work, we employ the *Gain Ratio Feature Evaluation* algorithm [108] implemented in the Weka toolkit [49], 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 evaluate the relevance of the predictors in the change prediction models experimented, 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) or with respect the other metrics contained in the models.

3.6 $\mathbf{RQ}_4$ - Combining All Predictors and Smell-Related Information

As a final step of our study, this research question has the goal to assess the performance of a change prediction model built using a combination between smell-related information and other product, process, and developer-related features. To this aim, we firstly put all the independent variables considered in the study in a single dataset. Then, we apply the variable removal procedure based on the *vif* function (see Section 3.3.2 for details on this technique): in this way, we were able to remove the independent variables that do not significantly influence the performance of the combined model. Finally, we test the ability of the newly devised model using the same procedures and metrics used in the context of $\mathbf{RQ}_1$, i.e., F-measure and AUC-ROC, and statistically comparing the performance of the considered models by means of the Scott-Knott ESD test.

4 Analysis of the Results

In this section we report and sum up the results of the presented research questions, discussing the main findings of our study.

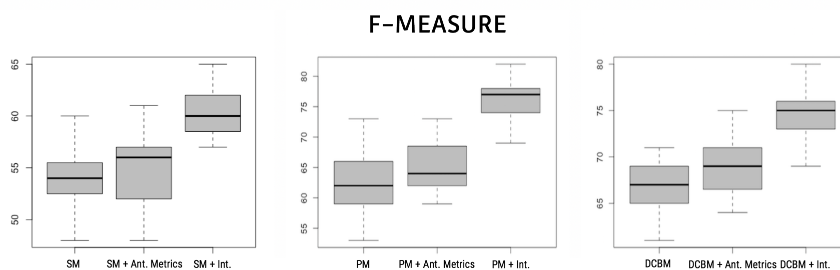

Fig. 1: Overview of the value of F-measure among the models. Note that the starting value of the y-axis is different based on the performance of the models.

Table 7: Average number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) output by the experimented change prediction models. Standard deviation is reported in parenthesis.

SM				SM+Ant.				SM+Int.			
TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
1305 (35)	369 (21)	616 (24)	172 (28)	1404 (54)	320 (18)	567 (21)	172 (25)	1530 (16)	129 (10)	688 (14)	115 (11)
PM				PM+Ant.				PM+Int.			
TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
1527 (33)	246 (31)	369 (28)	320 (26)	1576 (31)	271 (27)	320 (23)	296 (22)	1872 (24)	172 (22)	246 (21)	172 (13)
DCBM				DCBM+Ant.				DCBM+Int.			
TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
1650 (27)	271 (20)	320 (12)	222 (15)	1700 (42)	246 (33)	246 (16)	271 (13)	1823 (16)	148 (7)	296 (21)	197 (9)

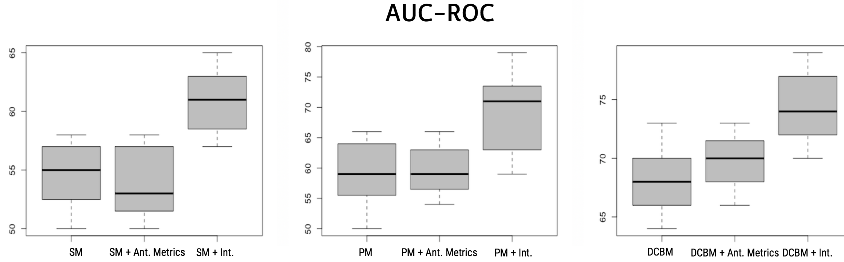

Fig. 2: Overview of the value of AUC-ROC among the models. Note that the starting value of the y-axis is different based on the performance of the models.

4.1 $\mathbf{RQ}_1$: *To what extent does the addition of the intensity index as additional predictor improve the performance of existing change prediction models?*

Before describing the results related to the contribution of the intensity index in the three prediction models considered, we report the results of the feature selection process aimed at avoiding multi-collinearity. According to the results achieved using the *vif* function [88], we remove FCH, LCH, WFR, ATAF, CHD, LCD, CSBS, and ACDF from the process-based model [36], while we do not remove any variables from the other baselines.

Figures 1 and 2 show the box plots reporting the distributions of F-Measure and AUC-ROC achieved by the (i) basic models that do not include any smell-related information - SM, PM, and DCBM, respectively; (ii) models including the antipattern metrics - those having "+ Ant. Metrics" as suffix; and (iii) models including the intensity index - those reporting "+ Int." as suffix. To address $\mathbf{RQ}_1$, in this section we discuss the performance of the intensity-including models, while the comparison with the antipattern metrics-including models is reported and discussed in Section 4.2. Note that for the sake of readability, we only report the distribution of F-Measure rather than the distributions of precision and recall; moreover, in Table 7 we report the confusion matrices for the models built using structural, process, and developer-oriented metrics as basic predictors: since our validation strategy, i.e., 100 times 10-fold validation, required the generation of 1,000 confusion matrices for each release and for each experimented model, we report average and standard deviation

of the overall number of true positives, true negatives, false positives, and false negatives for each model. Note that the fine-grained results of our analyses are available in our online appendix [27].

Looking at Figure 1, we can observe that the basic model based on scattering metrics (i.e., DCBM) tends to perform better than models built using structural and process metrics. Indeed, DCBM [32] has a median F-Measure 5% and 13% higher than structural (67% vs 54%) and process (67% vs 62%) models, respectively. This result confirms our previous findings on the power of the developer-related factors in change prediction [28] as well as the results achieved by Di Nucci et al. [32] on the value of the scattering metrics for the prediction of problematic classes. As for the role of the intensity index, we notice that with the respect to the SM, PM and DCBM model, the intensity of code smells provides an additional useful information able to increase the ability of the model in discovering change-prone code components. This is observable by looking at the performance in Figures 1 and 2. In the following, we further discuss our findings by reporting our results for each prediction model experimented.

Contribution in Structural-based Models. The addition of the intensity index within the SM model enables the model to reach a median F-Measure of 60% and an AUC-ROC of 61%, respectively. Looking more in depth into the results, we observe that the shapes of the box plots for the intensity-including model appear less dispersed than the basic one, i.e., the smaller the shape, the smaller the variability of the results. This means that the addition of the intensity index makes the performance of the model better and more stable. This is also visible by looking at the confusion matrices of the two models (Table 7): the one referring to the intensity-including model (column "SM+Int.") has an high average number of true positives and a lower standard deviation with respect to the baseline model, indicating that the addition of the smell-related information can lead to better change prediction performance. The same observation is true when considering true negatives, false positive, and false negatives. An an example, let us consider the `Apache-ant-1.3` project, where the basic structural model reaches 50% precision and 56% recall (F-Measure=53%), while the model that includes the intensity index has a precision of 61% and a recall of 66% (F-Measure=63%), thus obtaining an improvement of 10%. The same happens in all the considered systems: based on our findings, we can claim that the performance of change prediction models improves when considering the intensity of code smells as additional independent variable. The observations made above were also confirmed from a statistical point of view. Indeed, the intensity-including prediction model consistently appeared in the top Scott-Knott ESD rank in terms of AUC-ROC: this indicates that its performance was *statistically higher* than the baselines in most of the cases (40 projects out of 43).

Contribution in Process-based Models. Also in this case the addition of the intensity index in the model defined by Elish et al. [36] improved its performance with respect to the basic model (PM). The overall median value

of F-Measure increased of 15%, i.e., F-Measure of *PM + Int.* is 77% while that of *PM* is 62%. In this case, the intensity-including model can increase both precision and recall with respect to the basic model. This is, for instance, the case of `Apache Ivy 2`, where PM reaches 61% of precision and 49% of recall; by adding the intensity index, the prediction model increases its performances to 76% (+15%) in terms of precision and 77% (+28%) of recall, demonstrating that a better characterization of classes having design problems can help in obtaining more accurate predictions. The higher performance of the intensity-including models is mainly due to the better characterization of true positive instances (see Table 7). The statistical analyses confirm the findings: the likelihood to be ranked at the top by the Scott-Knott ESD test is always higher for the model including the intensity index, thus we can claim that the performance of the intensity-including model is statistically higher than the basic model over all the considered systems.

Contribution in Developer-Related Model. Finally, the results for this type of model are similar to those discussed above. Indeed, the addition of the intensity in DCBM [32] enables the model to reach a median F-Measure of 75% and an AUC-ROC of 74%, respectively. Compare to the standard model, "DCBM + Int." performs better (i.e., +7% in terms of median F-Measure and +6% in terms of median AUC-ROC). For instance, in the `Apache Synapse 1.2` project the *"DCBM + Int."* obtains values for F-Measure and AUC-ROC 12% and 13% higher than *DCBM*, respectively. The result holds for all the systems in our dataset, meaning that the addition of the intensity always provides improvements with respect to the baseline. The Scott-Knott ESD test confirms our observations from a statistical point of view. The likelihood of the intensity-including model to be ranked at the top is always higher than the other models. Thus, the intensity-including model is statistically superior with respect to the basic model over all the considered projects. From the analysis of Table 7 we can confirm the improved stability of the intensity-including model, which presents a lower standard deviation as well as a higher number of true positive instances identified.

> **$\mathbf{RQ_1}$ - To what extent does the intensity index improve the performance of existing change prediction models?** The addition of the intensity index [97] as a predictor of change-prone components increases the performance of the baseline change prediction models in terms of both F-Measure and AUC-ROC by up to 10%, i.e., up to 87 actual change-prone classes with respect to the baseline models. This is confirmed statistically.

4.2 $\mathbf{RQ_2}$: *How does the model including the intensity index as predictor compare to a model built using antipattern metrics?*

From $\mathbf{RQ_1}$, we observe that the addition of the intensity index within state of the art techniques can improve their performance. Nevertheless, the second

research questions aimed at evaluating whether the smell intensity index is a better predictor of change proneness than other smell related information, in particular, the antipattern metrics defined by Taba et al. [128]. For this reason, we compared the models including the intensity index with those including the antipattern metrics. The boxplots which summarize this comparison are in Figures 1 and 2, while average and standard deviation related to the confusion matrices are available in Table 7.

Table 8: Overlap analysis between the model including the intensity index and the model including the antipattern metrics.

Models	Int. ∩ Ant.%	Int. \ Ant.%	Ant. \ Int.%
SM [151]	43	35	22
PM [36]	47	38	15
DCBM [32]	44	31	25

Comparison in Structural-based Models. As reported in the previous section, the intensity-including model has a median F-Measure of 60% and an AUC-ROC of 61%. When compared against the antipattern metrics-including model, the intensity-including one still performs better (i.e., +4% in terms of median F-Measure and +7% in terms of median AUC-ROC). More in detail, we notice that the performance of the antipattern-including model is just slightly better than the basic model in terms of F-Measure (56% vs 54%); more interesting, the AUC-ROC of the "SM + Ant. Metrics" model is lower than the basic one (53% vs 55%). From a practical perspective, these results tell us that the inclusion of the antipattern metrics only provides slight improvement with respect to the number of actual change-prone classes identified, but at the same time, we cannot provide benefits in the robustness of the classifications. Moreover, the confusion matrices of "SM + Ant. Metrics" and "SM" models shown in Table 7 are similar: this confirms that the addition of the antipattern metrics cannot provide major benefits to the baseline.

In the comparison between the "SM + Ant. Metrics" and the "SM + Int." models, we observe that the performance of the former is always lower than the one achieved by the latter (considering the median of the distributions, -4% of F-Measure and -8% of AUC-ROC). This indicates that the intensity index can provide much higher benefits in change prediction than existing metrics that capture other smell-related information. The statistical analysis confirmed the superiority of the intensity-including models, which are always ranked better than the antipattern-including ones. While the models including the antipattern metrics have worse performances than the models including the intensity index in some cases, the antipattern metrics defined by Taba et al. [128] can give orthogonal information with respect to the intensity index, opening the possibility to obtain better performance still by considering both metric types. Our claim is supported by the overlap analysis shown in Table 8

and computed on the set of *change-prone* and *smelly* classes correctly classified by the two models. While 43% of the instances are correctly classified by both the models, a consistent portion of instances are classified only by SM + Int. model (35%) or by the model using the antipattern metrics (22%). Consequently, this means that the smell-related information taken into account by the "SM + Int." and "SM + Ant. Metrics" models are orthogonal.

Comparison in Process-based Models. The median F-Measure of the intensity-including model is 77%, i.e., +15% with respect to the basic model. As for the model that includes the antipattern metrics, we notice that it provides improvements when compared to the basic one. However, such improvements are still minor in terms of F-Measure (64% vs 62%) and thus we can confirm that the addition of the metrics proposed by Taba et al. [128] does not provide a substantial boost in the performance of basic change prediction models. Similarly, the model based on such metrics is never able to outperform the performance of the intensity-including one, being its F-measure 13% lower than the F-measure of the the model including the intensity index. The statistical analyses confirmed these findings. Indeed, the likelihood to be ranked at the top by the Scott-Knott ESD test is always higher for the model including the intensity index. At the same time, the model including the antipattern metrics provides a slight statistical benefits than the basic one (they are ranked in the same cluster in 88% of the cases). Also in this case we found an interesting orthogonality between the set of *change-prone and smelly* classes correctly classified by"PM + Int." and by the "PM + Ant. Metrics" (see Table 8), i.e., the two models correctly capture the change-proneness of different code elements.

Comparison in Developer-Related Model. When comparing the performance of "DCBM + Int." with the model that includes the antipattern metrics [128], we observe that the F-Measure of the former is on average 6% higher than the latter; the better performance of the intensity-including model is also confirmed when considering the AUC-ROC, which is 4% higher. The Scott-Knott ESD test confirms our observations, as (i) the intensity-including model has statistically higher performance than the baseline and (ii) the antipattern metrics-including models are confirmed to provide statistically better performance than the basic models in 67% of the considered systems. Nevertheless, also in this case we found an orthogonality in the correct predictions done by these two models (see Table 8): only 44% of instances are correctly caught by both the models, while 31% of them are only captured by "DCBM + Int." and 25% only by "DCBM + Ant. Metrics".

RQ_2 - How does the model including the intensity index as predictor compare to a model built using antipattern metrics? The prediction models that include the antipattern metrics [128] have lower performance than the intensity-including models, while perform slightly (but statistically) better than the basic models. We observe some orthogonal-

ity between the set of *change-prone and smelly* classes correctly classified by the models that include *intensity index* and the models with *antipattern metrics*, which highlights the possibility to achieve higher performance through a combination of smell-related information.

4.3 **RQ**$_3$: *What is the gain provided by the intensity index to change prediction models when compared to other predictors?*

In this section we analyze the results of *Gain Ratio Feature Evaluation* algorithm [108] in order to understand how important the predictors composing the different models considered in this study are, with the aim to evaluate the predictive power of the intensity index when compared to the other predictors.

Table 9 shows the gain provided by the different predictions employed in the structural metrics-based change prediction model, while Table 10 reports the results for the process-based model and Table 11 those for the DCBM model. In particular, the tables report the ranking of the predictors based on their importance within the individual models. The value of the mean and the standard deviation (computed by considering the results obtained on the single systems) represent the expected reduction in entropy caused by partitioning the prediction model according to a given predictor. In addition, we also provide 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 is statistically better than the others. The following subsections discuss our findings considering each prediction model individually.

Table 9: Gain Provided by Each Metric To The SM Prediction Model.

Metric	Mean	St. Dev.	SK-ESD Likelihood
CBO	0.66	0.09	82
RFC	0.61	0.05	77
Intensity	**0.49**	**0.13**	**75**
LOC	0.44	0.11	55
LCOM 3	0.43	0.12	51
Antipattern Complexity Metric	0.42	0.12	41
Antipattern Recurrence Length	0.31	0.05	32
Average Number of Antipatterns	0.22	0.10	21
DIT	0.13	0.02	3

Gain Provided to Structural-based Models (SM). The results in Table 9 show that Coupling Between Objects (CBO) is the metric having the highest predictive power, with an average reduction of entropy of 0.66 and a standard deviation of 0.09. The Scott-Knott ESD test statistically confirms the importance of the predictor, since the information gain given by the metric is statistically higher than other metrics in 82% of the cases. It is worth noting that this result is in line with previous findings in the field [12, 45, 97, 151]

which showed the relevance of coupling information for the maintainability of software classes. Looking at the ranking, we also notice that Response For a Class (RFC), Lines of Code (LOC), and Lack of Cohesion of Methods (LCOM 3) appear to be relevant. On the one hand, this is still in line with previous findings [12, 97, 151]. On the other hand, it is also important to note that our results indicate that code size is important for change prediction. In particular, unlike the findings by Zhou et al. [151] on the confounding effect of size, we discovered that LOC can be an important predictor to discriminate change-prone classes. This may be due to the large dataset exploited in this study, which allows a higher level of generalizability. The Scott-Knott ESD test confirm that these metrics are among the most powerful ones.

As for the variable of interest, i.e., the intensity index, we observe that it is the feature providing the third highest gain in terms of reduction of entropy, as it has a value of Mean and Standard Deviation of 0.49 and 0.13, respectively. 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 49% of the cases: this indicates that the metric is statistically more relevant than the other predictors in almost half of the projects. These findings lead to two main observations. In the first place, the intensity index has a high predictive power and, for this reason, can provide high benefits for the prediction of change-prone classes (as also observed in $\mathbf{RQ}_1$). Secondly, and perhaps more interesting, the intensity index can be more powerful than other structural metrics from which it is derived: in other words, a metric mixing together different structural aspects to measure how severe a code smell is, seems to be more meaningful than the individual metrics used to derive the index.

As for the antipattern metrics, we observe that all of them appear to be less relevant than the intensity index. This is in line with the results of $\mathbf{RQ}_2$, where we have shown that adding them to change prediction models results in a limited improvement with respect to the baseline. At the same time, it is worth noting that ACM (i.e., *Antipattern Complexity Metric*) may sometimes provide a notable contribution. While the average gain is 0.42, the standard deviation is 0.12: this means that the entropy reduction can be up to 0.54, as in the case of the `Apache-synapse-2.3`. This result suggests that this metric has some potential for effectively predicting the change-proneness of classes. The other two antipattern metrics, i.e., *Average Number of Antipatterns* (ANA) and *Antipattern Recurrence Length* (ARL), instead, provide a mean entropy reduction of 0.31 and 0.22, respectively, with a standard deviation that is never above 0.1. Thus, their contribution is lower than ACM. The Scott-Knott ESD test statistically confirms those findings: indeed, ACM was a top predictor in 41% of the datasets, as opposed to ANA and ARL metrics which appear as statistically more powerful than other metrics in only 32% and 21% of the cases, respectively. We conclude that ACM is statistically more valuable and relevant than the other antipattern metrics. Finally, Depth Inheritance Tree is the least powerful metrics in the ranking, and the Scott-Knott ESD test ranks it at the top in only 3% of the cases.

Table 10: Gain Provided by Each Metric To The PM Prediction Model.

Metric	Mean	St. Dev.	SK-ESD Likelihood
BOC	0.56	0.05	75
FRCH	0.55	0.06	64
Intensity	**0.44**	**0.08**	**61**
WCD	0.42	0.11	55
Antipattern Complexity Metric	0.41	0.04	54
LCA	0.33	0.07	33
CHO	0.28	0.03	31
Antipattern Recurrence Length	0.24	0.05	25
Average Number of Antipatterns	0.09	0.03	2
CSB	0.07	0.01	1
TACH	0.02	0.01	1

Gain Provided to Process-based Models (PM). Regarding the *process metric-based* model considered in this study, the results are similar to the structural model. Indeed, from Table 10 we observe that the intensity index has a mean entropy reduction of 0.44 and it is a top predictor in 61% of the projects. According to the information gain algorithm, it is the third most powerful feature of the model, ranked after the Birth of a Class and Frequency of Changes metrics. On the one hand, this ranking is quite expected, as the top two features are those which fundamentally characterize the notion of process-based change prediction (PM) proposed by Elish et al. [36]. On the other hand, our findings report that the intensity index can be orthogonal with respect to the process metrics present in the model, i.e., a structural-based indicator is orthogonal with respect to the other basic features. As for the antipattern metrics, ACM was within the top predictors in 54% of the projects, while ANA and ARL are top predictors in 25% and 2% of the dataset, respectively: this result confirms that ACM has a statistically higher predictive power than the other antipattern metrics. At the bottom of the ranking there are other basic metrics like Changes since the Birth and Total Amount of Changes: this is in line with previous findings [28] reporting that the overall number of previous changes cannot properly model the change-proneness of classes.

Table 11: Gain Provided by Each Metric To The DCBM Prediction Model.

Metric	Mean	St. Dev.	SK-ESD Likelihood
Semantic Scattering	0.76	0.07	95
Intensity	**0.74**	**0.05**	**94**
Structural Scattering	0.72	0.05	91
Antipattern Complexity Metric	0.66	0.04	78
Antipattern Recurrence Length	0.31	0.02	44
Average Number of Antipatterns	0.11	0.03	21

Gain Provided to Developer-related factors. Looking at the ranking of the features of the DCBM model, we can still confirm the results discussed so far. Indeed, the intensity index is the second most relevant factor, ranked

after the semantic scattering: its mean is 0.74, and the Scott-Knott ESD test indicates the intensity index as top predictor in 94% of the considered projects. It is interesting to note that the intensity index provides a higher contribution than the structural scattering, indicating that the combination from which it is derived can provide a higher entropy reduction with respect to a metric based on the structural distance of the the classes touched by developers in a certain time window. Regarding the antipattern metrics, the results are similar to those of the other models considered; the ACM provided a mean information gain of 0.66, being ranked at top predictors in 78% of the dataset. Instead, the means for ARL and ANA are lower (0.31 and 0.11, respectively): these are the least important features.

As a more general observation, it is worth noting that the values of mean information gain of both the intensity index and ACM are much high ($\approx$0.20 more) for this model when compared to the structural- and process metrics-based models. As such, those metrics can provide a much high information than the other models: this can be due to the limited number of features employed by this model, which makes the additional metrics more useful to predict the change-proneness of classes.

All in all, we confirm that the intensity index is a relevant feature for all the change prediction models considered in the study, together with ACM from the group of antipattern metrics. This possibly highlights how their combination could provide further improvements in the context of change prediction.

RQ_3 - What is the gain provided by the intensity index to change prediction models when compared to other predictors? The intensity index is a relevant predictor for all the considered prediction models (0.49 for SM, 0.44 for PM, and 0.74 DCBM respectively). At the same time, a metric of complexity of the change process involving code smells provides further additional information, highlighting the possibility to obtain even better change prediction performance when mixing different smell-related information.

4.4 $\mathbf{RQ}_4$: *What is the performance of a combined change prediction model that includes smell-related information?*

The results achieved in the previous research questions highlight the possibility to build a combined change prediction model that takes into account smell-related information besides the structural, process, and developer-related metrics. For this reason, in the context of $\mathbf{RQ}_4$, we assess the feasibility of a combined solution and evaluate its performance with respect to the results achieved by the models experimented in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. As explained in Section 3, to come up with the combined model we firstly put together all the features of the considered models and then apply a feature selection algorithm to discard irrelevant features. Starting from an initial set of 18 metrics,

this procedure discards DIT, CSB, TACH, and ANA. Thus, the combined model comprises 14 metrics: besides most of the basic structural, process, and developer-related predictors, the model includes three smell-related metrics, namely (i) intensity index, (ii) ACM, and (iii) ARL.

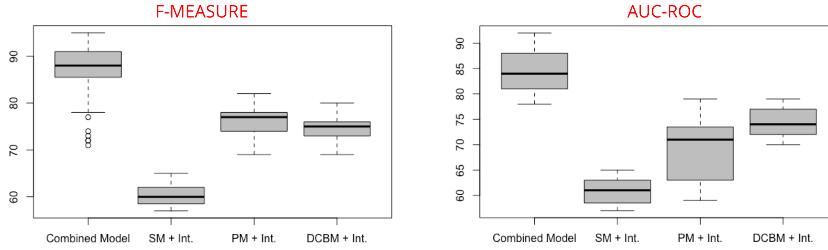

Fig. 3: Overview of the value of F-Measure and AUC-ROC of the Combined Model. Note that the starting value of the y-axis is different based on the performance of the models.

Table 12: Average number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) output by the devised combined model and comparison with the best change prediction models coming from $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. Standard deviation is reported in parenthesis.

Combined				SM+Int.			
TP	FP	TN	FN	TP	FP	TN	FN
2192 (59)	74 (12)	123 (16)	74 (15)	1530 (16)	129 (10)	688 (14)	115 (11)
PM+Int.				**DCBM+Int.**			
TP	FP	TN	FN	TP	FP	TN	FN
1872 (24)	172 (22)	246 (21)	172 (13)	1823 (16)	148 (7)	296 (21)	197 (9)

Figure 3 shows the boxplots reporting the distributions of F-Measure and AUC-ROC related to the smell-aware combined change prediction model. To facilitate the comparison with the models exploited in the context of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we also report boxplots depicting the best models coming from our previous analyses, i.e., *SM + Int.*, *PM + Int.*, and *DCBM + Int.*. Moreover, Table 12 reports average and standard deviation of the overall number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) given as output by the combined model as well as by the baselines.

Looking at the results, the combined model has better performance than all the baselines. The median F-Measure reaches 88%, being 18%, 11%, and 13% more accurate than *SM + Int.*, *PM + Int.*, and *DCBM + Int.*, respectively. This is also true for the AUC-ROC, where the combined model achieves 10% better performances than the basic models that include the intensity. As an example, in the `Apache Xalan 2.5` project the best stand-alone model (the *"SM + Int"* in this case) has an F-Measure close to 73%, while the mixture of features provided by the combined model enables to reach an F-Measure

of 93%. As expected, the results are statistically significant, as the combined smell-aware change prediction model appears in a top Scott-Knott ESD rank in 98% of the cases.

On the one hand, these results confirm previous findings on the importance to combine different predictors of source code maintainability [28, 31]. On the other hand, we can claim that smell-related information [97, 128] improves the capabilities of change prediction models, allowing them to perform better than other existing models.

RQ_4 - What is the performance of a combined change prediction model that includes smell-related information? The devised smell-aware change prediction model performs better than all the baselines considered in the paper, with an F-Measure up to 20% and AUC ROC up to 10%.

5 Discussion and Threats to Validity

The results of our research questions highlight some relevant findings that need to be further discussed, especially with respect to their relevance in a practical use case scenario. Moreover, our findings might be biased by some aspects whose mitigation is subject of discussion in this section.

5.1 Discussion and practical usage of our change prediction model

Our results demonstrate that smell-related information can be used within existing change prediction models to improve the accuracy with which they identify classes more likely to be modified by developers in the future. As a first discussion point, it is worth noting that the performance obtained by the baseline models exploiting individual product, process, or developer-oriented predictors are in line with those reported in recent work [28]: indeed, this aspect indicates that our methodology led to results comparable with those achieved in literature with respect to the performance of different types of change prediction models, thus (i) validating the methodological choices we adopt in this work and (ii) confirming previous findings in the field [28, 36, 151]; moreover, we notice that the improvements obtained by means of the addition of smell-related information have led to the definition of better prediction models, thus suggesting the importance of code smells for the problem of interest and confirming the empirical studies previously conducted on their role on the change-proneness of classes [63, 96].

The findings reported in our work have two main practical implications and applications. First of all, the output of the proposed change prediction model can be adopted by developers to keep track of change-prone classes: this can be relevant to create *awareness* among developers about the classes

that, for some reasons, tend to change frequently and that possibly hide design issues that should be solved. We believe that the output of our technique can provide developers with a tool able to spot problems within source code before their actual occurrence, giving to them early indications on the classes to be maintained and/or improved with respect to their quality.

It is important to note that change-prone classes are those source code elements that, for some reasons, are more likely to be modified in the future maintenance and evolution activities of a software project. As an example, change-prone classes must not be confused with bug-prone code elements. Indeed, these two sets might have some relationships, but they still remain conceptually disjoint. On the one hand, bug-proneness indicates source code that is more likely to have bugs in a close future: as such, the fact that a class has bugs does not imply that it changes more often, moreover, there are some defects that are fixed before they ever lead to a fault. On the other hand, bug-prone classes might also be change-prone (i.e., changes are made to correct faults), however bug fixing activities do not represent the only reason for changes, as classes might change due to different software evolution tasks (e.g., implementation of new change requests), so bugs are only one of the many reasons for change. Thus, change- and bug-proneness of classes might have some relation, but are not the same. To have a comprehensive overview of the causes of change-proneness of a class, in this work we taken into account baseline change prediction models based on different sets of basic predictors, i.e., product, process, and developer-based metrics, that are able to capture different state of the art evolutionary aspects of source code classes.

Besides creating awareness, the devised change prediction model can be directly *integrated* within developers' software analytics dashboards (e.g., Bitergia[4]). This integration may enable a continuous feedback mechanism that allows developers the immediate identification of the source code classes that are more likely to change in the future. Such feedback can be then used by developers as input for performing preventive maintenance activities before putting the code into production: for instance, in a continuous integration (CI) scenario [17], developers might want to refactor the code before the CI pipeline starts to avoid warnings given by static analysis tools [141, 142, 16, 18]. Similarly, our change prediction model might be useful for project managers in order to properly schedule maintenance operations: more specifically, they can exploit the feedback given on the classes that are more likely to change in the future to better plan when, where, and how to perform refactoring, code review, or testing activities and improve the overall quality of the source code. We believe that the proposed change prediction model can be effectively used to continuously provide feedback and recommendations, thus allowing developers and project managers to schedule and apply modifications leading to source code improvement.

[4] https://bitergia.com

5.2 Threats to Validity

In this section we discuss possible threats affecting our results and how we mitigated them.

5.2.1 Threats to Construct Validity

Threats to construct validity are related to the relationship between theory and observation. In our study, a first threat is related to the independent variables used and the dataset exploited: *the selection of the variables to use as basic predictors might have influenced our findings.* To mitigate such a threat, we selected state of the art change prediction models based on a different set of basic features, i.e., structural, process, and developer-related metrics, that capture different characteristics of source code. The selection was mainly driven by recent results [28] that showed that the considered models are (i) accurate in the detection of the change-proneness of classes and (ii) orthogonal to each other, thus correctly identifying different sets of change-prone classes. All in all, this selection process has enabled us to test the contribution of smell-related information in different contexts.

A second threat is related to the dataset exploited, and in particular to *its reliability as source of actual change prediction components.* In this regard, we first relied on a publicly available dataset [56] and, on top of this, we computed product, process, and developer-oriented metrics employing publicly available tools. It is important to note that an important threat would have been related to the alignment between the dataset already available and the classes on which the additional metrics have been computed: to mitigate it, we have excluded inner-classes and only took into account the meta-information available in the original dataset provided by Jureczko and Madeyski [56], which provides the full qualifiers of all classes of the considered systems (for all the releases). As for the dependent variable, we have relied on ChangeDistiller [38] to classify change-prone and non-change-prone classes: despite its accuracy, it is worth noting that this tool does not detect changes in the documentation (i.e., comments): while this could have under-estimated the change-proneness of some classes, changes exclusively targeting the documentation likely do not modify the behavior of a method/class and are not related to maintainability reasons [57]. Thus, we are still confident to have properly estimated the change-proneness of the classes in our dataset.

More in general, the definition of the problem is represented by a binary classification based on ordinal rank. Such a definition assures that both dependent and independent variables are project-dependent. Specifically, the definition of change-proneness is based on the number of changes that the classes of a certain project underwent over their history. As a consequence, also the ranking is deemed to be project-dependent since the change-proneness of the classes depends on their individual number of changes. Similarly, most of the independent variables are locally dependent: product metrics are computed on single classes, while process metrics are computed on the basis of the his-

tory of individual classes. The only exception is related to the developer-based metrics: they are indeed computed on the basis of the activities made by a certain developer on a certain project. We do not consider the activities of a developer made on different projects, i.e., a developer might contribute to different projects, and this might bias the computation of the scattering metrics. However, this likely represents a corner-case rather than a common one.

We adopted JCodeOdor [41] to identify code smells and assign to them a level of intensity: *the choice of the detector could have biased our observations.* To mitigate this possible threat, JCodeOdor was selected based on the results of previous experiments which obtained a high accuracy, i.e., F-measure = 80%, on the same dataset [97]. Nevertheless, the tool still identifies 154 false positives and 94 false negatives among the 43 considered systems: *such imprecisions might have substantially biased the interpretation of the reported results.* To deal with it and make the set of code smells as close as possible to the *ground truth*, in our study we manually analyzed the output of JCodeOdor in order to (i) set to zero the intensity index of the false positive instances, and (ii) discard the false negatives, i.e., the instances for which we could not assign an intensity value. However, since this manual process is not always feasible, we have also evaluated the effect of including false positive and false negative instances in the construction of the change prediction models. More specifically, we re-ran the analyses performed in Section 3 and validated the performance of the experimented models when including the false positive instances using the same metrics used to assess the performance of the other prediction models (i.e., F-Measure and AUC-ROC). Our results report that these models always perform better than other models that do not include any smell-related information, while they are slightly less performing (-3% in terms of median F-Measure) than those built discarding the false positive instances. At the same time, we have evaluated the impact of including false negative instances. Their intensity index is, by definition, equal to zero: as a consequence, they are considered in the same way as *non-smelly* classes. The results of our analyses show that the *intensity-including* models still produce better results than the baselines, as they boosted their median F-Measure of ≈4%. At the same time, we have observed a decrement of 2% in terms of F-Measure with respect to the performance obtained by the prediction models built discarding false negatives. As a final step, we have also considered the case where both false positives and false negatives are incorporated in the experimented models. Our findings report that the *Basic + Intensity* models have a median F-Measure 2% lower than the models where the false positive and false negative instances were filtered out. At the same time, they were still better than the basic models (median F-Measure=+6%). Thus, we can conclude that *a fully automatic code smell detection process still provides better performance than existing change prediction models.* In our opinion, this result is extremely valuable as it indicates that practitioners can adopt automatic code smell detectors without the need of manually investigating the candidates they give as output.

The choice of considering code smell severity rather than the simple presence/absence of smells is driven by the conjecture that the severity can give a more fine-grained information on how much a design problem is harmful for a certain source code class: however, it would still be possible that *a simpler modeling of the problem would have led to better results.* To verify whether our conjecture is actually correct or not, we have conducted a further analysis aimed at establishing the performance of the experimented models where considering a boolean value reporting the presence of code smells rather than their intensity. As expected, our findings are that the models relying on the intensity are more powerful than those based on the boolean indication of the smell presence. This further confirms the idea behind this paper, i.e., code smell intensity can improve change-proneness prediction.

Finally, it is important to remark that our observations might still have been threatened by the presence of code smell co-occurrences [98, 147], which might have biased the intensity level of the smelly classes of our dataset. While on average only 8% of the classes in our dataset contained more code smells, the validity of our conclusions is still limited to the considered code smells, i.e., we cannot exclude that other code smell types not considered in this study co-occurred with those we have investigated. Therefore, further investigations into the role of code smell co-occurrences would be desirable.

5.2.2 Threats to Conclusion Validity

Threats to *conclusion validity* refer to the relation between treatment and outcome. A first threat in this category concerns the evaluation metrics adopted *to interpret the performance of the experimented change prediction models.* To mitigate the interpretation bias, we computed well-established metrics such as F-Measure and AUC-ROC and statistically verified the differences in the performance achieved by the different experimented models using the Scott-Knott ESD statistical test [133]. In this regard, we took into account the problem of data normality: over our dataset, the Shapiro-Wilk test of normality [116] gives an output a ρ-value higher than the threshold of 0.05 for all the considered independent variables, meaning that the data is *not* normally distributed. As a consequence, we relied on the Scott-Knott ESD, which does not assume a normally-distributed data. Moreover, we have also taken the comments made by Herbold [54] into account on the original implementation of the Scott-Knott ESD difference test with respect to the normality correction it applies; in particular, we have exploited version `v1.2.2` of the algorithm, that contains the latest implementation where the normality correction is properly applied. This test allowed us to measure the importance of predictors into the model from a statistical point of view; indeed, the higher the value given by the test to a certain variable (from 1 to 100%), the higher the statistical importance of that variable in correctly predicting the change-proneness of classes [133].

A second threat is related to the *methodology adopted to measure how much the intensity index has improved the performance of change prediction models.* Similar to previous work [97], we have 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 this way, we could assess the performance of the models with and without the variable of interest, thus assessing the actual gain given by it.

Another threat to the validity that might have possibly affected our conclusions is related to the selection of the threshold used when discarding non-relevant independent variables thought the *Vif* function [88]. Such a selection has been under debate for a long time [114], without a clear and established outcome [88]. In the context of our study, we employed the function implemented in the R toolkit: as explained in the documentation of the package[5], the R implementation of the *Vif* function is based on the square of the multiple correlation coefficient resulting from regressing a predictor variable against all other predictor variables. If a variable has a strong linear relationship with at least one other variable, the correlation coefficient would be close to 10, and *Vif* for that variable would be large. The R toolkit, as well as other previous works [35, 149], recommend to use a threshold higher than 10, as a *Vif* assuming higher values is a signal that the model has a collinearity problem. Thus, while some previous work in software engineering adopted different thresholds [24, 121, 152], e.g., 2.5, we followed the guidelines provided by previous literature and by the specific statistic tool exploited.

Finally, there is a threat related to *the relation between cause and effects*. In particular, while we found certain relationships between independent and dependent variables, we cannot exclude that metrics that we did not take into account might have had an effect on our findings. For example, as reported by Olbrich et al. [86], certain types of smelly classes are not necessarily more change/defect prone after that size is taken into account: to mitigate such an issue, we have taken into account a wide set of independent variables that have been previously studied and assessed as highly effective for change prediction purposes. However, the addition of other variables in the model might still have an impact on our findings and, therefore, we encourage further replications of our study to shed lights on possible additional factors influencing change prediction. Similarly, the application of the proposed change prediction model to other domains having substantially different characteristics with respect to those analyzed herein may lead to different conclusions and, therefore, would deserve further analyses.

5.2.3 Threats to External Validity

Threats in this category mainly concern the generalization of results. A first threat is related to the *heterogeneity of the dataset exploited in the study*. In this regard, we have analyzed a large set of 43 releases of 14 software systems of an ecosystem in particular, i.e., the Apache Software Foundation. We are aware that this can threaten the generalizability of the results; however the considered projects are different in terms of application domain, size, and

[5] https://www.rdocumentation.org/packages/usdm/versions/1.1-18/topics/vif

age. Of course, further analyses targeting different systems and/or ecosystems would be beneficial to corroborate the findings observed on our dataset.

Another threat in this category regards the choice of the baseline models, as *they might not represent the state of the art in change prediction.* To mitigate this potential problem, we have evaluated the contribution of the smell-related information in the context of change prediction models widely used in the past [28, 36, 151] that take into account predictors of different nature, i.e., product, process, and developer-related metrics. However, we are aware that our study is based on systems developed in Java only, and therefore future investigations aimed at corroborating our findings on a different set of systems would be worthwhile. At the same time, we are aware that a possible generalizability issue is related to the transportability of the proposed model to other domains, e.g., High Performance Computing or Embedded systems, which might have different constraints on memory, cache utilization, or parallelization with respect to the systems considered in this study. While a re-calibration of our model in those contexts would surely be a valid option, further methodological decisions might have to be addressed and/or reconsidered to properly adopt the model in other contexts.

6 Conclusion

Based on previous findings [63, 96] that report the impact of code smells on the change-proneness of classes, in this paper investigated the impact of smell-related information for the prediction of change-prone classes. We first conducted a large empirical study on 43 releases of 14 software systems and evaluated the contribution of the intensity index proposed by Arcelli Fontana et al. [5] within existing change prediction models based on product- [151], process-[36], and developer-related [32] metrics. We also compared the gain provided by the intensity index with the one given by the so-called antipattern metrics [128], i.e., metrics capturing historical aspects of code smells. The results indicated that the addition of the intensity index as a predictor of change-prone components increases the performance of baseline change prediction models by an average of 10% in terms of F-Measure (RQ_1). Moreover, the intensity index can boost the performance of such models more than state of the art smell-related metrics such as those defined by Taba et al. [128], even though we have observed an orthogonality between the models exploiting different information on code smells (RQ_2). Based on these results, we built a combined smell-aware change prediction model that takes into account product, process, developer- and smell-related information (RQ_4). The results show that the combined model provides a consistent boost in terms of F-Measure, which improve up to 20%.

Our findings represent the main input for our future research agenda: we first aim at further testing the usefulness of the devised model in an industrial setting. Furthermore, we plan to perform a fine-grained analysis into the role of each smell type independently in the change prediction power. Additionally,

we plan to investigate the feasibility of making change prediction models more useful for developers by defining novel summarization mechanisms able to output a summary reporting the likely reasons behind the decisions taken by the change prediction model, thus explaining why a certain class is more likely to be modified. Finally, we aim at studying the effect of change prediction models in practice with respect to their usefulness in improving the allocation of resources devoted to preventive maintenance operations.

Acknowledgment

The authors would like to thank the anonymous reviewers for the detailed and constructive comments on the preliminary version of this paper, which were instrumental to improving the quality of the work. Fabio Palomba was partially supported by the Swiss National Science Foundation (SNSF) through the Project no. PP00P2_170529.

References

1. Abbes M, Khomh F, Gueheneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Software maintenance and reengineering (CSMR), 2011 15th European conference on, IEEE, pp 181–190
2. Abdi M, Lounis H, Sahraoui H (2006) Analyzing change impact in object-oriented systems. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06), IEEE, pp 310–319
3. Ammerlaan E, Veninga W, Zaidman A (2015) Old habits die hard: Why refactoring for understandability does not give immediate benefits. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 504–507
4. Aniche M, Treude C, Zaidman A, van Deursen A, Gerosa M (2016) Satt: Tailoring code metric thresholds for different software architectures. In: 2016 IEEE 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 41–50
5. Arcelli Fontana F, Ferme V, Zanoni M, Roveda R (2015) Towards a prioritization of code debt: A code smell intensity index. In: Proceedings of the Seventh International Workshop on Managing Technical Debt (MTD 2015), IEEE, Bremen, Germany, pp 16–24, in conjunction with ICSME 2015
6. Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering 21(3):1143–1191, DOI 10.1007/s10664-015-9378-4, URL http://dx.doi.org/10.1007/s10664-015-9378-4

7. Arisholm E, Briand LC, Foyen A (2004) Dynamic coupling measurement for object-oriented software. IEEE Transactions on Software Engineering 30(8):491–506
8. Azeem MI, Palomba F, Shi L, Wang Q (2019) Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology
9. Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 712–721
10. Baeza-Yates R, Ribeiro-Neto B (1999) Modern Information Retrieval. Addison-Wesley
11. Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering 28(1):4–17
12. Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22(10):751–761
13. Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107:1–14
14. Bell RM, Ostrand TJ, Weyuker EJ (2013) The limited impact of individual developer data on software defect prediction. Empirical Software Engineering 18(3):478–505
15. Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), IEEE, pp 202–211
16. Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 470–481
17. Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: An explorative analysis of travis ci with github. In: Proceedings of the International Conference on Mining Software Repositories (MSR), IEEE, pp 356–367
18. Beller M, Gousios G, Zaidman A (2017) Travistorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration
19. Bieman JM, Straw G, Wang H, Munger PW, Alexander RT (2003) Design patterns and change proneness: an examination of five evolving systems. In: Proceedings International Workshop on Enterprise Networking and Computing in Healthcare Industry, pp 40–49, DOI 10.1109/METRIC.2003.1232454
20. Bottou L, Vapnik V (1992) Local learning algorithms. Neural Comput 4(6):888–900
21. Boussaa M, Kessentini W, Kessentini M, Bechikh S, Ben Chikha S (2013) Competitive coevolutionary code-smells detection. In: Search Based Soft-

ware Engineering, Lecture Notes in Computer Science, vol 8084, Springer Berlin Heidelberg, pp 50–65
22. Bradley AP (1997) The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern recognition 30(7):1145–1159
23. Briand LC, Wust J, Lounis H (1999) Using coupling measurement for impact analysis in object-oriented systems. In: Proceedings of International Conference on Software Maintenance (ICSM), IEEE, pp 475–482
24. Cataldo M, Mockus A, Roberts JA, Herbsleb JD (2009) Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering 35(6):864–878
25. Catolino G, Ferrucci F (2018) Ensemble techniques for software change prediction: A preliminary investigation. In: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2018 IEEE Workshop on, IEEE, pp 25–30
26. Catolino G, Ferrucci F (2019) An extensive evaluation of ensemble techniques for software change prediction. Journal of Software Evolution and Process DOI https://doi.org/10.1002/smr.2156
27. Catolino G, Palomba F, Arcelli Fontana F, De Lucia A, Ferrucci F, Zaidman A (2018) Improving change prediction models with code smell-related information - replication package - https://figshare.com/s/f536bb37f3790914a32a
28. Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A (2018) Enhancing change prediction models using developer-related factors. Journal of Systems and Software 143:14–28
29. le Cessie S, van Houwelingen J (1992) Ridge estimators in logistic regression. Applied Statistics 41(1):191–201
30. D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: Quality Software (QSIC), 2010 10th International Conference on, pp 23–31, DOI 10.1109/QSIC.2010.58
31. DAmbros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering 17(4):531–577
32. Di Nucci D, Palomba F, De Rosa G, Bavota G, Oliveto R, De Lucia A (2018) A developer centered bug prediction model. IEEE Transactions on Software Engineering 44(1):5–24
33. Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A (2018) Detecting code smells using machine learning techniques: are we there yet? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 612–621
34. Di Penta M, Cerulo L, Gueheneuc YG, Antoniol G (2008) An empirical study of the relationships between design pattern roles and class change proneness. In: Proceedings International Conference on Software Maintenance (ICSM), IEEE, pp 217–226, DOI 10.1109/ICSM.2008.4658070
35. Eisenlohr PV (2014) Persisting challenges in multiple models: a note on commonly unnoticed issues regarding collinearity and spatial structure of ecological data. Brazilian Journal of Botany 37(3):365–371

36. Elish MO, Al-Rahman Al-Khiaty M (2013) A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. Journal of Software: Evolution and Process 25(5):407–437
37. Eski S, Buzluca F (2011) An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In: Proceedings International Conf Software Testing, Verification and Validation Workshops (ICSTW), IEEE, pp 566–571
38. Fluri B, Wuersch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on software engineering 33(11)
39. Fontana FA, Zanoni M (2017) Code smell severity classification using machine learning techniques. Knowledge-Based Systems 128:43–58
40. Fontana FA, Zanoni M, Marino A, Mantyla MV (2013) Code smell detection: Towards a machine learning-based approach. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, pp 396–399, DOI 10.1109/ICSM.2013.56
41. Fontana FA, Ferme V, Zanoni M, Roveda R (2015) Towards a prioritization of code debt: A code smell intensity index. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), IEEE, pp 16–24
42. Fontana FA, Ferme V, Zanoni M, Yamashita A (2015) Automatic metric thresholds derivation for code smell detection. In: Proceedings of the Sixth international workshop on emerging trends in software metrics, IEEE Press, pp 44–53
43. Fontana FA, Dietrich J, Walter B, Yamashita A, Zanoni M (2016) Antipattern and code smell false positives: Preliminary conceptualization and classification. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 1, pp 609–613, DOI 10.1109/SANER.2016.84
44. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley
45. Fregnan E, Baum T, Palomba F, Bacchelli A (2018) A survey on software coupling relations and tools. Information and Software Technology
46. Gatrell M, Counsell S (2015) The effect of refactoring on change and fault-proneness in commercial c# software. Science of Computer Programming 102(0):44 – 56, DOI http://dx.doi.org/10.1016/j.scico.2014.12.002, URL `http://www.sciencedirect.com/science/article/pii/S0167642314005711`
47. Girba T, Ducasse S, Lanza M (2004) Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE, pp 40–49
48. Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach. Lawrence Erlbaum Associates Publishers

49. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: An update. SIGKDD Explor Newsl 11(1):10–18, DOI 10.1145/1656274.1656278, URL http://doi.acm.org/10.1145/1656274.1656278
50. Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011) Developing fault-prediction models: What the research can show industry. IEEE Software 28(6):96–99
51. Han AR, Jeon SU, Bae DH, Hong JE (2008) Behavioral dependency measurement for change-proneness prediction in uml 2.0 design models. In: 32nd Annual IEEE International Computer Software and Applications Conference, IEEE, pp 76–83
52. Han AR, Jeon SU, Bae DH, Hong JE (2010) Measuring behavioral dependency for improving change-proneness prediction in uml-based design models. Journal of Systems and Software 83(2):222–234
53. Hassan AE (2009) Predicting faults using the complexity of code changes. In: International Conference Software Engineering (ICSE), IEEE, pp 78–88
54. Herbold S (2017) Comments on scottknottesd in response to "an empirical comparison of model validation techniques for defect prediction models". IEEE Transactions on Software Engineering 43(11):1091–1094
55. John GH, Langley P (1995) Estimating continuous distributions in bayesian classifiers. In: Eleventh Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, San Mateo, pp 338–345
56. Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM
57. Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, pp 351–360
58. Kennedy J (2011) Particle swarm optimization. In: Encyclopedia of machine learning, Springer, pp 760–766
59. Kessentini M, Vaucher S, Sahraoui H (2010) 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, ACM, ASE '10, pp 113–122
60. Kessentini W, Kessentini M, Sahraoui H, Bechikh S, Ouni A (2014) A cooperative parallel search-based software engineering approach for code-smells detection. IEEE Transactions on Software Engineering 40(9):841–861, DOI 10.1109/TSE.2014.2331057
61. Khomh F, Di Penta M, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th Working Conference on Reverse Engineering, IEEE, pp 75–84
62. Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: Proceedings of the International Conference on Quality Software (QSIC), IEEE, Hong Kong, China, pp 305–314

63. Khomh F, Di Penta M, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering 17(3):243–275
64. Kim M, Zimmermann T, Nagappan N (2014) An empirical study of refactoringchallenges and benefits at microsoft. IEEE Transactions on Software Engineering 40(7):633–649
65. Kohavi R (1995) The power of decision tables. In: 8th European Conference on Machine Learning, Springer, pp 174–189
66. Kumar L, Behera RK, Rath S, Sureka A (2017) Transfer learning for cross-project change-proneness prediction in object-oriented software systems: A feasibility analysis. ACM SIGSOFT Software Engineering Notes 42(3):1–11
67. Kumar L, Rath SK, Sureka A (2017) Empirical analysis on effectiveness of source code metrics for predicting change-proneness. In: ISEC, pp 4–14
68. Lanza M, Marinescu R (2006) Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer
69. Le Cessie S, Van Houwelingen JC (1992) Ridge estimators in logistic regression. Applied statistics pp 191–201
70. Lehman MM, Belady LA (eds) (1985) Program Evolution: Processes of Software Change. Academic Press Professional, Inc.
71. Lu H, Zhou Y, Xu B, Leung H, Chen L (2012) The ability of object-oriented metrics to predict change-proneness: a meta-analysis. Empirical software engineering 17(3):200–242
72. Malhotra R, Bansal A (2015) Predicting change using software metrics: A review. In: International Conference on Reliability, Infocom Technologies and Optimization (ICRITO), IEEE, pp 1–6
73. Malhotra R, Khanna M (2013) Investigation of relationship between object-oriented metrics and change proneness. International Journal of Machine Learning and Cybernetics 4(4):273–286
74. Malhotra R, Khanna M (2014) A new metric for predicting software change using gene expression programming. In: Proceedings International Workshop on Emerging Trends in Software Metrics, ACM, pp 8–14
75. Malhotra R, Khanna M (2017) Software change prediction using voting particle swarm optimization based ensemble classifier. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, ACM, pp 311–312
76. Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp 50–60
77. Marinescu C (2014) How good is genetic programming at predicting changes and defects? In: International Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, pp 544–548
78. Marinescu R (2004) Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp 350–359

79. Marinescu R (2012) Assessing technical debt by identifying design flaws in software systems. IBM Journal of Research and Development 56(5):9–1
80. Menzies T, Caglayan B, Kocaguneli E, Krall J, Peters F, Turhan B (2012) The PROMISE repository of empirical software engineering data
81. Mkaouer MW, Kessentini M, Bechikh S, Cinnéide MÓ (2014) A robust multi-objective approach for software refactoring under uncertainty. In: International Symposium on Search Based Software Engineering, Springer, pp 168–183
82. Moha N, Guéhéneuc YG, Duchien L, Meur AFL (2010) Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36(1):20–36
83. Morales R, Soh Z, Khomh F, Antoniol G, Chicano F (2016) On the use of developers' context for automatic refactoring of software anti-patterns. Journal of Systems and Software (JSS)
84. Munro MJ (2005) Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proceedings of the International Software Metrics Symposium (METRICS), IEEE, p 15
85. Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on Software visualization, ACM, pp 5–14
86. Olbrich SM, Cruzes DS, Sjøberg DIK (2010) Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: Internationl Conference on Software Maintenance, pp 1–10
87. Oliveto R, Khomh F, Antoniol G, Guéhéneuc YG (2010) Numerical signatures of antipatterns: An approach based on B-splines. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp 248–251
88. O'brien RM (2007) A caution regarding rules of thumb for variance inflation factors. Quality & quantity 41(5):673–690
89. Palomba F, Zaidman A (2017) Does refactoring of test smells induce fixing flaky tests? In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, pp 1–12
90. Palomba F, Zaidman A (2019) The smell of fear: On the relation between test smells and flaky tests. Empirical Software Engineering (EMSE) To Appear
91. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on, IEEE, pp 268–278
92. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? a study on developers' perception of bad code smells. In: Software maintenance and evolution (ICSME), 2014 IEEE international conference on, IEEE, pp 101–110
93. Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015) Mining version histories for detecting code smells. IEEE Trans-

actions on Software Engineering 41(5):462–489, DOI 10.1109/TSE.2014.2372760
94. Palomba F, Lucia AD, Bavota G, Oliveto R (2015) Anti-pattern detection: Methods, challenges, and open issues. Advances in Computers 95:201–238, DOI 10.1016/B978-0-12-800160-8.00004-8
95. Palomba F, Panichella A, De Lucia A, Oliveto R, Zaidman A (2016) A textual-based technique for smell detection. In: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE, pp 1–10
96. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2017) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering pp 1–34
97. Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2017) Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering
98. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology 99:1–10
99. Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2018) The scent of a smell: An extensive comparison between textual and structural smells. Transactions on Software Engineering 44(10):977–1000
100. Palomba F, Tamburri DAA, Fontana FA, Oliveto R, Zaidman A, Serebrenik A (2018) Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Transactions on Software Engineering
101. Palomba F, Zaidman A, De Lucia A (2018) Automatic test smell detection using information retrieval techniques. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 311–322
102. Parnas DL (1994) Software aging. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 279–287
103. Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018) Information needs in contemporary code review. Proceedings of the ACM on Human-Computer Interaction 2(CSCW):135
104. Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. Journal of Systems and Software to appear
105. Peer A, Malhotra R (2013) Application of adaptive neuro-fuzzy inference system for predicting software change proneness. In: Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on, IEEE, pp 2026–2031
106. Peng CYJ, Lee KL, Ingersoll GM (2002) An introduction to logistic regression analysis and reporting. The Journal of Educational Research 96(1):3–14
107. Peters R, Zaidman A (2012) Evaluating the lifespan of code smells using software repository mining. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, pp 411–416

108. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106, DOI 10.1023/A:1022643204877, URL http://dx.doi.org/10.1023/A:1022643204877
109. Ratiu D, Ducasse S, Gîrba T, Marinescu R (2004) Using history information to improve design flaws detection. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp 223–232
110. Romano D, Pinzger M (2011) Using source code metrics to predict change-prone java interfaces. In: Proceedings International Conference Software Maintenance (ICSM), IEEE, pp 303–312
111. Rosenblatt F (1961) Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan Books
112. Rumbaugh J, Jacobson I, Booch G (2004) Unified Modeling Language Reference Manual, The (2Nd Edition). Pearson Higher Education
113. Sahin D, Kessentini M, Bechikh S, Deb K (2014) Code-smell detection as a bilevel problem. ACM Transactions on Software Engineering Methodology 24(1):6:1–6:44, DOI 10.1145/2675067
114. Schwartz J, Landrigan PJ, Feldman RG, Silbergeld EK, Baker EL, von Lindern IH (1988) Threshold effect in lead-induced peripheral neuropathy. The Journal of pediatrics 112(1):12–17
115. Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics pp 507–512
116. Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611
117. Sharafat AR, Tahvildari L (2007) A probabilistic approach to predict changes in object-oriented software systems. In: Proceedings Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp 27–38
118. Sharafat AR, Tahvildari L (2008) Change prediction in object-oriented software systems: A probabilistic approach. Journal of Software 3(5):26–39
119. Shepperd M, Song Q, Sun Z, Mair C (2013) Data quality: Some comments on the nasa software defect datasets. IEEE Transactions on Software Engineering 39(9):1208–1215
120. Shepperd M, Bowes D, Hall T (2014) Researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering 40(6):603–616, DOI 10.1109/TSE.2014.2322358
121. Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE (2010) Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, p 4
122. Sjoberg DI, Yamashita A, Anda BC, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering (8):1144–1156
123. Soetens QD, Pérez J, Demeyer S, Zaidman A (2015) Circumventing refactoring masking using fine-grained change recording. In: Proceedings of the

14th International Workshop on Principles of Software Evolution (IW-PSE), ACM, pp 9–18

124. Soetens QD, Demeyer S, Zaidman A, Pérez J (2016) Change-based test selection: An empirical evaluation. Empirical Software Engineering 21(5):1990–2032
125. Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
126. Spinellis D (2005) Tool writing: A forgotten art? IEEE Software (4):9–11
127. Stone M (1974) Cross-validatory choice and assessment of statistical predictions. Journal of the royal statistical society Series B (Methodological) pp 111–147
128. Taba SES, Khomh F, Zou Y, Hassan AE, Nagappan M (2013) Predicting bugs using antipatterns. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '13, pp 270–279, DOI 10.1109/ICSM.2013.38, URL http://dx.doi.org/10.1109/ICSM.2013.38
129. Taibi D, Janes A, Lenarduzzi V (2017) How developers perceive smells in source code: A replicated study. Information and Software Technology 92:223–235
130. Tamburri DA, Palomba F, Serebrenik A, Zaidman A (2018) Discovering community patterns in open-source: a systematic approach and its evaluation. Empirical Software Engineering pp 1–49
131. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '16, pp 321–332, DOI 10.1145/2884781.2884857, URL http://doi.acm.org/10.1145/2884781.2884857
132. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Comments on researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering 42(11):1092–1094, DOI 10.1109/TSE.2016.2553030
133. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering 43(1):1–18, DOI 10.1109/TSE.2016.2584050, URL https://doi.org/10.1109/TSE.2016.2584050
134. Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: A curated collection of java code for empirical studies. In: Proceedings of 17th Asia Pacific Software Engineering Conference, IEEE, Sydney, Australia, pp 336–345, DOI 10.1109/APSEC.2010.46
135. Theodoridis S, Koutroumbas K (2008) Pattern recognition. IEEE Transactions on Neural Networks 19(2):376–376

136. Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering 35(3):347–367
137. Tsantalis N, Chatzigeorgiou A, Stephanides G (2005) Predicting the probability of change in object-oriented systems. IEEE Transactions on Software Engineering 31(7):601–614
138. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, pp 403–414
139. Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp 4–15
140. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43(11):1063–1088
141. Vassallo C, Palomba F, Gall HC (2018) Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 564–568
142. Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: The developer perspective on the usage of static analysis tools. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 38–49
143. Vidal S, Guimaraes E, Oizumi W, Garcia A, Pace AD, Marcos C (2016) On the criteria for prioritizing code anomalies to identify architectural problems. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, ACM, pp 1812–1814
144. Vidal SA, Marcos C, Díaz-Pace JA (2016) An approach to prioritize code smells for refactoring. Automated Software Engineering 23(3):501–532
145. Vonken F, Zaidman A (2012) Refactoring with unit testing: A match made in heaven? In: Proceeedings of the Working Conference on Reverse Engineering (WCRE), IEEE, pp 29–38
146. Y Freund LM (1999) The alternating decision tree learning algorithm. In: Proceeding of the Sixteenth International Conference on Machine Learning, pp 124–133
147. Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 682–691
148. Yamashita AF, Moonen L (2012) Do code smells reflect important maintainability aspects? In: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE, pp 306–315

149. Yu CH (2000) An overview of remedial tools for collinearity in sas. In: Proceedings of 2000 Western Users of SAS Software Conference, WUSS, vol 1, pp 196–201
150. Zhao L, Hayes JH (2011) Rank-based refactoring decision support: two studies. Innovations in Systems and Software Engineering 7(3):171
151. Zhou Y, Leung H, Xu B (2009) Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Transactions on Software Engineering 35(5):607–623
152. Zogaan W, Sharma P, Mirahkorli M, Arnaoudova V (2017) Datasets from fifteen years of automated requirements traceability research: Current state, characteristics, and quality. In: Requirements Engineering Conference (RE), 2017 IEEE 25th International, IEEE, pp 110–121