On the Performance of Method-Level Bug Prediction: A Negative Result

Luca Pascarella,[1] Fabio Palomba,[2] Alberto Bacchelli[2]

[1] *Delft University of Technology, The Netherlands* — [2] *University of Zurich, Switzerland*
l.pascarella@tudelft.nl, palomba@ifi.uzh.ch, bacchelli@ifi.uzh.ch

Abstract

Bug prediction is aimed at identifying software artifacts that are more likely to be defective in the future. Most approaches defined so far target the prediction of bugs at class/file level. Nevertheless, past research has provided evidence that this granularity is too coarse-grained for its use in practice. As a consequence, researchers have started proposing defect prediction models targeting a finer granularity (particularly method-level granularity), providing promising evidence that it is possible to operate at this level. Particularly, models mixing product and process metrics provided the best results.

We present a study in which we first replicate previous research on method-level bug-prediction, by using different systems and timespans. Afterwards, based on the limitations of existing research, we (1) re-evaluate method-level bug prediction models more realistically and (2) analyze whether alternative features based on textual aspects, code smells, and developer-related factors can be exploited to improve method-level bug prediction abilities. Key results of our study include that (1) the performance of the previously proposed models, tested using the same strategy but on different systems/timespans, is confirmed; but, (2) when evaluated with a more practical strategy, all the models show a dramatic drop in performance, with results close to that of a random classifier. Finally, we find that (3) the contribution of alternative features within such models is limited and unable to improve the prediction capabilities significantly. As a consequence, our replication and negative results indicate that method-level bug prediction is still an open challenge.

Keywords: Defect Prediction, Empirical Software Engineering, Mining Software Repositories

1. Introduction

The necessary evolution of software systems often leads to the introduction of defects, which possibly preclude the correct functioning of a piece of software and reduce its overall reliability [67]. To tackle this problem, researchers have been developing several techniques to support developers (*e.g.,* verification and testing [17]): one of the most investigated areas is *bug-prediction* [50], which consists in detecting the areas of a software more likely to contain bugs in the future. Researchers have proposed and evaluated a variety of bug prediction models based on product [5, 74, 73], process [104, 93, 79], socio-technical [102, 13], and developer-related [37, 24] metrics. These models have been evaluated both in within-project scenarios and in cross-project ones [105, 122, 127], with several approaches achieving remarkable prediction performance [33]. Nevertheless, the practical relevance of bug prediction research has been put into question by studies that suggest that bug prediction does not address any real need of developers [109, 68, 66]. One of the main criticisms regards the *granularity* at which bugs are found [109]. In fact, most of the presented models predict bugs in modules or files – a granularity that is deemed not informative enough for practitioners, because files and modules can be arbitrarily large and inspecting them can require too much work [46]. In addition, considering that larger classes tend to be more bug-prone [62, 87], the effort required to identify the defective part in these classes is even more pronounced [5, 48, 84, 99].

To tackle this limitation, Menzies *et al.* [74] and Tosun *et al.* [120] conducted the first investigations on a finer granularity, *i.e.,* function-level. Successively, Hata *et al.* [54] applied this idea to the context of object-oriented systems, proposing a method-level prediction model built using a set of historical metrics and that reported promising performance. Giger *et al.* [46] went even further and investigated the value of both product and process metrics for method-level prediction model. Specifically, Giger *et al.* devised three prediction models based on the combination of the two sets of features and evaluated how well they could classify which methods would have at least a bug (binary classification) within a specified time frame. They considered single snapshots of 21 open source software (OSS) projects in Java and reported promising results: 84% precision and 88% recall.

In this paper, we present a work that continues on this line of research.[1] First, we replicate the investigation conducted by Giger *et al.* [46] on bug prediction at

[1]The work presented here is an extension of the conference paper 'Re-evaluating Method-Level Bug Prediction' [97], appeared in the proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. pp. 592-601.

method-level. We use the same features and classifiers as the reference work, but on a different dataset to test the generalizability of their findings. While our results show similar performance as the reference work, we observed two key limitations that may possibly bias the interpretation of the achieved findings. On the one hand, Giger *et al.* [46] took the change history and predicted bugs from the same time frame (which could lead to incorrect results) and used cross-validation (which have been reported as at the risk of producing biased estimates in certain circumstances [118]). On the other hand, they did not consider a number of alternative features that have been proved to impact the performance of class-level bug prediction models, namely textual, code smell-related, and developer-based metrics [10, 58, 93]. We tackle these limitations by (1) estimating the models' performance using data from subsequent releases (as done by more recent studies, which did it at a coarser granularity [104]), and by (2) adding a set of new alternative features to the considered method-level bug prediction model.

Our results show that—when evaluated on a release-by-release strategy—all the existing method-level bug prediction models present lower performance, close to that of a random classifier. As a consequence, even though we could replicate the reference work, a more realistic evaluation lead to negative results. Furthermore, all the alternative features we experiment with only marginally improve the performance, suggesting that method-level bug-prediction is still not a solved problem.

2. Background and Related Work

Research in the field of bug prediction is highly active [50, 56] and can be roughly divided in two sets: On the one hand, researchers focused on the characteristics relating to source code being more defect prone [102, 62, 87, 2, 3, 6, 23, 86, 89, 112]; on the other hand, researchers defined bug prediction techniques based on unsupervised [35, 80, 126] and supervised [18, 38, 59, 92, 128] approaches. More recently, the concept of *just-in-time* bug-prediction has been introduced—techniques with the purpose of recommending defective files as developers commit them [60, 110, 125, 97, 20, 21, 55].

The current paper presents a work that focuses on investigating how well supervised approaches can identify bug-prone methods. In this section we discuss the literature related to class-/method-level bug prediction and describe the role of textual information for software quality.

2.1. Class-level Bug-Prediction

The approaches in this category differ from each other mainly for the underlying prediction algorithm, *e.g., Logistic Regression* vs *Random Forest*, and for the considered features, *e.g., product* (*e.g.,* lines of code) or *process metrics* (*e.g.,* number of changes performed to a class).

Product metrics. Basili *et al.* [5] found that five CK metrics [27] can help one to determine defective classes and that Coupling Between Objects (CBO) is the most related to bugs. These results were re-confirmed in further studies [48, 61, 113]. Ohisson *et al.* [83] focused on design metrics (*e.g.,* 'number of nodes') to identify bug-prone modules, revealing the applicability of such metrics for the identification of buggy modules. Nagappan and Ball exploited two static analysis tools to predict the pre-release bug density for Windows Server [77]. Nagappan *et al.* [78] experimented with code metrics for predicting buggy components across five Microsoft projects, finding that no single metric is the best across all projects. Zimmerman *et al.* [128] investigated complexity metrics for bug-prediction and reported a positive correlation between code complexity and bugs. Nikora *et al.* [81] showed that measurements of a system's structural evolution (*e.g.,* 'number of executable statements') can serve as bug-predictors. More recently, Dam *et al.* [31] reported an experience report of using product metrics and abstract representation of source code in practice, showing that it is possible to achieve good prediction accuracy when employing them within deep learning models.

Process metrics. Graves *et al.* [119] experimented with both product and process metrics for bug-prediction, finding that product metrics are poorer predictors for bugs. Moser *et al.* [76, 106] performed two comparative studies, which provided additional corroborating evidence on the superiority of process metrics in predicting buggy code components. Later on, D'Ambros *et al.* [34] performed an extensive comparison of bug-prediction approaches relying on both product and process metrics, finding that no technique works better in all contexts.

Despite the aforementioned promising results, a study by Shihab *et al.* [109] reported that developers consider class or module level bug-prediction too coarse-grained for being useful in practice. Also, a study by Lewis *et al.* [68] reported similar issues when trying defect prediction in practice at Google. This situation calls for the creation of methods able to provide a more fine-grained prediction (*e.g.,* at *method-level*), re-evaluating and adapting what has been learned in the preceding work.

Alternative metrics. Despite product and process features have been the most widely used for bug prediction purposes, researchers have been also investigating the value of alternative metrics. In this category, several researchers exploited developer-related factors. For example, Hassan investigated a technique based on the entropy of developers' code changes [53], finding that it has better performance than models based on changes to code components. Ostrand *et al.* [10, 85] proposed the use of the number of developers who modified a code component as a bug-proneness predictor: however, the performance of the resulting model was poorly improved with respect to existing models. Later on, Di Nucci *et al.* [37] defined a bug-prediction model based on a mixture of code, process, and developer-based metrics outperforming the performance of existing models. As part of their experimentation, Di Nucci *et al.* also re-assessed the contribution given by the metric proposed by Ostrand *et al.* [10, 85], showing that in certain circumstances the number of developers may represent a relevant factor to characterize the bug-proneness of classes. Bird *et al.* [14] found that the addition of an information related to the code ownership of classes can make bug prediction models more accurate; these findings were later confirmed by other researchers [47, 103]. On a similar line of research, socio-technical factors have been also exploited for bug prediction: for instance, Posnett *et al.* [102] proposed two novel metrics based on the developer's social-network that may be used as predictors of faults in production, while Bird *et al.* [13] studied how developers contribute to source code and found that lack of collaboration and coordination are associated with an increase of the number of bugs.

Other researchers focused on improving bug prediction capabilities using the information coming from code smells [43], *i.e.,* sub-optimal design implementations applied by developers. Khomh *et al.* [62] and Palomba *et al.* [87] indeed reported that such smells have a strong, negative impact on the bug proneness of source code. Following these findings, Taba *et al.* [114] studied how the addition of variables characterizing the presence of 13 different types of code smells can improve the performance of bug prediction models built using standard product metrics: they reported an improvement close to 13% in terms of F-Measure. Later on, Palomba *et al.* [93] showed that the intensity of code smells, namely a measure of their severity, can further improve bug prediction capabilities with respect to the work of Taba *et al.* [114].

Finally, a less explored yet worth to discuss bug prediction angle concerns the usage of textual metrics. In the first place, the use of textual information has represented a promising and orthogonal dimension to improve software quality assessment [52, 71, 123]. For instance, textual information has already been used in several software engineering tasks or activities such as information retrieval [70, 71, 94], code smell detection [88, 91, 89], refactoring [7, 8, 9], and meaning extraction [96]. Perhaps more importantly, the addition of textual-related information has been proved to enhance the performance of bug prediction models. Marcus *et al.* [71] defined the Conceptual Cohesion of Classes (C3) and added it within a bug prediction model based on product metrics, finding that textual information can provide a boost of ≈23% in terms of F-Measure. Walid *et al.* [58] provided initial compelling evidence that a lack of coherence between code comments and corresponding source code (due to the missing update of code documentation) impacts the bug-proneness of code elements. Aman *et al.* [1] further explored the problem and found that certain types of code comments (e.g., those explaining functionalities implemented in a class) are associated with a higher bug-proneness of the code. Later on, Buse and Weimer [16] found that poor readability of source code contributes to the identification of buggy classes. These findings were also confirmed by Binkley *et al.* [12], who showed that a lower source code readability is often associated to an increase of the production code fault-proneness; When employed within predictive models, readability metrics provide an additional contribution that allow these models to perform ≈10% better than models built without using them.

Inspired by the results of these previous studies, in this work we aim at evaluating the effect of characterizing bug-prone methods considering alternative features, thus providing a wider overview of the performance achievable with method-level bug prediction.[2]

2.2. Method-level Bug-Prediction

While the seminal idea of lowering the granularity of bug prediction is to attribute to Menzies *et al.* [74] and Tosun *et al.* [120], the work by Giger *et al.* [46] was the first explicitly aimed at predicting bugs at method-level in object-oriented software systems. Giger *et al.* defined a set of product and process metrics to characterize a method and evaluated these metrics in three method-level bug prediction models, respectively based on: (i) product metrics, (ii) process metrics, and (iii) their combination. Giger *et al.* [46] found that both product and process metrics contribute to the identification of buggy methods and their combination achieves the best performance (*i.e.,* F-Measure=86%). To produce the dataset used in their evaluation, Giger *et al.* took the following steps [46]: they (1) considered a large time frame in the history of 21 Java OSS systems, (2) focused on the methods present at the end of the time frame, (3) computed product metrics for each method at the end of the time frame, (4) computed process metrics (*e.g.,* number of changes) for each method throughout the time frame, and (5) counted the number of bugs for each method throughout the time frame, relying on bug fixing commits. Finally, they used 10-fold cross-validation [64] to evaluate the three aforementioned models, considering the presence/absence of bug(s) in a method as the dependent (binary) variable. Similarly to the paper discussed above, Hata *et al.* [54] proposed a fine-grained prediction model in which they computed a number of historical metrics to predict the bug-proneness of Java methods. Their results reported that method-level predictions are more effective than file- and package-level ones when considering the effort required by developers to locate and debug a potential defect in source code.

In this work, we re-evaluate the paper by Giger *et al.* [46] using data from subsequent releases (*i.e.,* a release-by-release validation), which better models a real-case scenario where a prediction model is updated as soon as new information is available.[3] The choice of focusing on the

[2]This part is a novel contribution of this paper.

[3]This part was previously presented at an academic software engineering conference [97].

work of Giger *et al.* [46] rather than the one of Hata *et al.* [54] is motivated by the fact that Giger *et al.* experimented with both code and process metrics (as opposed to Hata *et al.* who only considered process metrics), thus giving us the opportunity of providing a wider overview of the performance of method-level defect prediction.

3. Research Goals and Context

In this section, we define both the research questions guiding our study and the context of our investigation.

3.1. Research Questions

The *goal* of the empirical study is to re-evaluate how bug prediction can be applied at method-level, with the *purpose* of understanding the performance of models built using different sets of features. We start our investigation by replicating the study conducted by Giger *et al.* [46] on a partially overlapping set of software systems (but considering different moments in time) to evaluate the generalizability of their findings. Thus, we ask:

RQ$_1$. *How effective are existing method-level bug prediction approaches when tested on new systems/timespans?*

While replicating the methodology proposed by Giger *et al.* [46], we detected some limitations concerning the validation approach: (1) it uses 10-fold cross-validation, which is at the risk of producing biased estimates in certain circumstances [118], (2) product metrics are considered only at the end of the time frame (while bugs are found *within* the time frame), and (3) the number of changes and the number of bugs were both considered in the same time frame (this *time-insensitive* validation strategy may have led to biased results). Thus, in the second part of our study we try to overcome the aforementioned limitations by re-evaluating the performance using data from subsequent releases. A *release-by-release* validation better models a real-case scenario where a prediction model is updated as soon as new information is available. Our expectation is that the performance is going to be weaker in this setting. This leads to our second research question:

RQ$_2$. *How effective are existing method-level bug prediction models when validated with a release-by-release validation strategy?*

A second limitation we identify is related to the independent variables exploited by Giger *et al.* [46]. While they performed an extensive analysis of product and process metrics, the role of other types of information—that have been shown to boost the performance of bug prediction models in the past [12, 58, 71]—was not assessed. In the context of our study, we assess the impact of three families of metrics such as: (1) textual features, whose aim is to capture the readability of the considered code as well as its alignment with code comments and their types; (2) code smells [43], which describe potential design flaws in source code; and (3) developer-related factors, that analyze properties related to the developers working on a system. Hence, we ask:

RQ$_3$. *How effective are method-level bug prediction models built using alternative features?*

Table 1: Overview of the projects used in this study.

Projects	LOC	Developers	Releases	Methods	Buggy Methods
Ant	213k	15	4	42k	2.3k
Checkstyle	235k	76	6	31k	4.1k
Cloudstack	1.16M	90	2	85k	13.4k
Eclipse JDT	1.55M	22	33	810k	3.3k
Eclipse Platform	229k	19	3	7k	2.7k
Emf Compare	3.71M	14	2	9k	0.7k
Gradle	803k	106	4	73k	4.6k
Guava	489k	104	17	262k	1.2k
Guice	19k	32	4	9k	0.5k
Hadoop	2.46M	93	5	179k	5.8k
Lucene-solr	586k	59	7	213k	8.7k
Vaadin	7.06M	133	2	43k	11.3k
Wicket	328k	19	2	30k	4.9k
Overall	**19M**	**782**	**91**	**1.8M**	**63.5k**

3.2. Subject systems

The *context* of our work consists of 13 software systems whose characteristics are reported in Table 1. For each system, the table reports its size (in terms of LOCs) and how many developers contributed over the entire history, as well as the number of releases, methods, and buggy methods. In particular, we focus on systems implemented in Java (*i.e.,* one of the most popular programming languages [39]), since both the metrics previously defined by Giger *et al.* [46] and the alternative features proposed in this study mainly target this programming language. In addition, we select projects whose source code and change history are publicly available (*i.e.,* open-source software projects using a version control system) to enable the extraction of product, process, and alternative metrics.

Starting from the $81,327,803$ open-source systems written in Java available at the time of the analysis on GitHub,[4] we first filter out those having less that $1,000$ commits and more than $5,000$ methods: this filter gives us a total of $6,753,654$ systems. Finally, we randomly select 13 of them. Compared to Giger *et al.* [46], we consider fewer, but larger systems, which are composed of a much larger number of methods (1.8M vs 112,058) and bugs (63,400 vs 23,762). This choice allows us to test the effectiveness of method-level bug prediction on software systems of a different scale.

[4]https://github.com

Table 2: List of releases with the related information on buggy methods used in the first research question.

Projects	Release Tag	Files	Buggy Files	Methods	Buggy Methods
Ant	ANT_190	1,574	459	11,326	3,201
Checkstyle	checkstyle-8.0	2,362	1,864	8,626	6,523
Cloudstack	4.10	62,627	16,275	47,430	12,311
Eclipse JDT	v_710	618	235	34,438	12,063
Eclipse Platform	R3_0	6,513	4,489	4,408	2,983
Emf Compare	2.0.0	6,613	1,356	4,547	1,021
Gradle	REL_2.2	3,603	308	20,414	1,722
Guava	v23.0	883	449	23,439	10,221
Guice	4.0	1,638	1,000	3,485	2,001
Hadoop	branch-3.0	56,338	4,622	76,281	5,247
Lucene-solr	7.0.0	61,259	5,412	50,033	3,420
Vaadin	8.0.0	57,319	22,290	28,647	11,028
Wicket	wicket-7.0.0	963	131	15,729	2,174
Overall		**262k**	**59k**	**329k**	**79k**

4. RQ$_1$. Replicating Method-Level Bug Prediction

Our first research question aims at replicating the study conducted by Giger *et al.* [46] on a different set of systems and time spans.

4.1. ***RQ***$_1$ *- Research Method*

To answer our first research question, we (i) build a method-level bug prediction model using the same features as Giger *et al.* [46] and (ii) evaluate its performance using the original evaluation strategy and applying the model to our projects. To this aim, we follow a set of methodological steps such as (i) the creation of an oracle reporting buggy methods in each of the projects considered, *i.e.,* the dependent variable to predict, (ii) the definition of the independent variables, *i.e.,* the metrics on which the model relies on, (iii) the assessment of the performance of different machine learning algorithms, and (iv) the definition of the validation methodology to test the performance of the models devised in the reference paper.

Extraction of Bug Data. For each system we need to detect the buggy methods contained at the end of the time frame, *i.e.,* in the *last* release R_{last}, to do so we use a methodology in line with that followed by Giger *et al.* [46]. Given the tagged issues available in the issue tracking systems (*i.e.,* Bugzilla or Jira) of the subject systems, we use ReLink [124] to identify links between issues and commits.[5] Afterwards, we consider as buggy all the methods changed in the buggy commits detected by ReLink and referring to the time period between the R_{last-1} and R_{last} (*i.e.,* the ones introduced during the final time frame). We discarded test cases because tests may be modified with the production code without being implicated in a bug. Table 2 reports the release tags together with the number of files, methods, and defective methods used to answer RQ$_1$.

[5]ReLink considers several constraints: (i) a match must exist between the committer and the contributor who created the issue in the issue tracking system, (ii) the time interval between the commit and the last comment posted by the same contributor in the issue tracker is less than seven days, and (iii) the cosine similarity between the commit message and the last comment referred above, computed using the Vector Space Model (VSM) [4], is greater than 0.7.

Table 3: List of method-level product metrics used in this study

Metric name	Description (applies to method-level)
FanIN	# of methods that reference a given method
FanOUT	# of methods referenced by a given method
LocalVar	# of local variables in the body of a method
Parameters	# of parameters in the declaration
CommentToCodeRatio	Ratio of comments to source code (line based)
CountPath	# of possible paths in the body of a method
Complexity	McCabe Cyclomatic complexity of a method
execStmt	# of executable source code statements
maxNesting	Maximum nested depth of all control structures

Independent variables. To characterize source code methods, we compute the set of 9 product and 15 process features previously defined by Giger *et al.* [46].

Table 4: List of method-level process metrics used in this study

Metric name	Description (applies to method level)
MethodHistories	# of times a method was changed
Authors	# of distinct authors that changed a method
StmtAdded	Sum of all source code statements added
MaxStmtAdded	Maximum StmtAdded
AvgStmtAdded	Average of AvgStmtAdded
StmtDeleted	Sum of all source code statements deleted
MaxStmtDeleted	Maximum of StmtDeleted
AvgStmtDeleted	Average of StmtDeleted
Churn	Sum of stmtAdded - stmtDeleted
MaxChurn	Maximum churn for all method histories
AvgChurn	Average churn per method history
Decl	# of method declaration changes
Cond	# of condition changes over all revisions
ElseAdded	# of added else-parts over all revisions
ElseDeleted	# of deleted else-parts over all revisions

- *Product Metrics:* Existing literature demonstrated how effective product metrics are in characterizing the extent to which a source code method is difficult to maintain, possibly indicating the presence of defects [5, 27, 83, 34]. Giger *et al.* [46] proposed the use of the metrics reported in Table 3. The features regard different method characteristics, *e.g.,* number of parameters or McCabe's cyclomatic complexity [72]. Although this may introduce a threat to the validity of our results, we had to re-implement all of the metrics due to the lack of available tools.

- *Process Metrics:* According to previous literature [106, 115], process features can complement the capabilities of product predictors for bug prediction. For this reason, Giger *et al.* [46] relied on the change-based metrics described in Table 4. These metrics characterize the life of source code methods (*e.g.,* by considering how many statements were added over time or the number of developers that touched the method). As we did for product metrics, we had to re-implement the proposed process metrics defined at method-level by Giger *et al.* [46].

In line with Giger *et al.* [46], in the context of $\mathbf{RQ}_1$, we build three different method-level bug prediction models relying on (i) *only* product metrics, (ii) *only* process metrics, and (iii) *both* product and process metrics.

Preprocessing The Training Data. Once obtained the subject dataset, we tackle two common problems that may affect machine learning algorithms: (i) data unbalance [26][6] and (ii) multi-collinearity [82].[7] We address the former problem by applying the Random Over-Sampling algorithm [25] implemented as a supervised filter in the Weka toolkit.[8] The filter re-weights the instances in the dataset to give them the same total weight for each class maintaining unchanged the total sum of weights across all instances. We address the second problem by filtering out the unwanted features. Specifically, we apply the *Correlation-based Feature Selection* [49] algorithm implemented as a filter in the Weka toolkit. It computes the correlation between each pair of features and, if this is higher than 0.7, removes one of them by considering their individual predictive power.

Machine Learner. Once the training data is preprocessed, we need to select a classifier that best leverages the independent variables to predict buggy methods [44]. To this aim, we investigate the four classifiers used by Giger *et al.* [46]: *Random Forest*, *Support Vector Machine*, *Bayesian Network*, and *J48*. Afterwards, we compare the different classification algorithms using the validation strategy and metrics described later.

Evaluation Strategy. The final step to answer $\mathbf{RQ}_1$ is the validation of the prediction models. As done in the reference work, we adopt the 10-fold cross-validation strategy [64, 116]. This strategy randomly partitions the original set of data into 10 equal sized subset. Of the 10 subsets, one is retained as test set, while the remaining nine are used as training set. This validation is then repeated 10 times, switching on which subset the model is tested.

Evaluation Metrics. Once we had run the experimented models over the considered systems, we measure their performance using the same metrics proposed by Giger *et al.* [46] to allow for comparison: *precision* and *recall* [4]. Precision is defined as

$$precision = \frac{|TP|}{|TP \bigcup FP|} \tag{1}$$

where TP (True Positives) are methods that are correctly classified as buggy by the prediction model and FP (False Positives) are methods that are wrongly classified as buggy. Recall is defined as

$$recall = \frac{|TP|}{|TP \bigcup FN|} \tag{2}$$

where FN (False Negatives) are buggy methods misclassified as non-buggy by the model. We also compute F-Measure [4], which combines precision and recall:

$$F - Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

In addition to the aforementioned metrics, we also compute the *Area Under the Receiver Operation Characteristic* curve (AUC-ROC) [51]. In fact, the classification chosen by the machine learning algorithms is based on a threshold (*e.g.,* all the method whose predicted value is above the threshold 0.5 are classified as buggy), which can greatly affect the overall results [118]; precision and recall alone are not able to capture this aspect. ROC plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1; the diagonal represents the expected performance of a random classifier. AUC computes the area below the ROC and allows us to have a comprehensive measure for comparing different ROCs: An area of 1 represents a perfect classifier (all the defective methods are recognized without any error), whereas for a random classifier an area close 0.5 is expected (since the ROC for a random classifier tends to the diagonal).

4.2. $\boldsymbol{RQ}_1$ *- Results*

Table 5 reports the median values for precision, recall, F-measure, and AUC-ROC achieved by models based on (i) only product, (ii) only process, and (iii) both product and process features when using different classifiers.[9] To ease the comparison between our replication (*'O'* in Table 5) and the work of Giger *et al.* [46], in the table we also report the results achieved by the original work (*'o'* in Table 5). Overall, from this first analysis we can claim that the obtained results are in line with those by Giger *et al.* [46], yet 10 percentage points lower on average.

The model based on product metrics achieves the lowest results. For instance, the overall precision is 0.71, meaning that a software engineer using this model has to needlessly analyze almost 29% of the recommendations it outputs. This result is in line with the findings provided by Giger *et al.* , who showed that the model only trained on product metrics offers generally lower performance.

Secondly, our results confirm that process metrics are stronger indicator of bug-proneness of source code methods (overall F-Measure=0.80). This finding is in line with the previous results achieved by the research community

[6]This frequent issue in bug prediction occurs when the number of instances that refer to buggy resources (in our case, source code methods) is drastically smaller than the number of non-buggy ones.

[7]Independent variables highly correlated cause collinearity that negatively impacts the reliability of the prediction models [40].

[8]https://www.cs.waikato.ac.nz/ml/weka/

[9]Our appendix [98] provides a detailed report of the performance achieved by the single classifiers over all the considered systems.

Table 5: Median classification results of method-level bug prediction models when validated using 10-fold cross validation. To ease the comparison between Giger *et al.* [46] and ours, the table reports the results of both studies: *'G'* stands for Giger *et al.* , *'O'* for ours; we also used a white background for the results of the previous study, a grey one for the ours.

π = Product	Study	Precision			Recall			F-measure			AUC-ROC		
Π = Process		π	Π	π&Π	S	Π	π&Π	π	Π	π&Π	π	Π	π&Π
Random Forest	O	0.72	0.85	**0.86**	0.64	**0.86**	0.86	0.68	0.85	**0.86**	0.66	0.84	**0.86**
	G	0.84	0.50	0.85	0.88	0.64	0.95	0.86	0.56	0.90	0.95	0.72	0.95
Support Vector Machines	O	0.66	0.74	0.74	0.09	0.80	0.79	0.16	0.77	0.76	0.50	0.51	0.51
	G	0.83	0.48	0.80	0.86	0.63	0.96	0.84	0.56	0.87	0.96	0.70	0.95
Bayesian Network	O	0.71	0.77	0.77	0.46	0.68	0.70	0.56	0.72	0.72	0.60	0.72	0.72
	G	0.82	0.46	0.81	0.86	0.73	0.96	0.84	0.58	0.87	0.96	0.73	0.96
J48	O	0.73	0.82	0.84	0.60	0.84	0.83	0.65	0.83	0.83	0.60	0.79	0.80
	G	0.84	0.56	0.83	0.82	0.58	0.89	0.83	0.57	0.86	0.95	0.69	0.91
Overall	**O**	**0.71**	**0.80**	**0.80**	**0.44**	**0.80**	**0.80**	**0.51**	**0.80**	**0.80**	**0.59**	**0.72**	**0.73**
	G	**0.82**	**0.72**	**0.75**	**0.85**	**0.64**	**0.91**	**0.83**	**0.57**	**0.86**	**0.95**	**0.71**	**0.88**

that report the superiority of process metrics with respect to product ones [104, 106]. Our results also confirm another finding by Giger *et al.* [46]: The combination of product and process metrics does not improve dramatically the prediction capabilities: Results are—at most—two points percentage higher than the model with process metrics only. This is surprising, because we expected that the use of these orthogonal predictors would improve the overall performance.

With respect to the different classifiers we experimented, *Support Vector Machines* gives the worst results; likely, this is due to the extreme sensitivity of the classifier to the configuration [29].[10] Future studies could be setup and conducted to investigate the impact of the configuration on SVM for method-level bug prediction. Other classifiers provide more stable results. *Random Forest* and *J48* obtain the best prediction accuracy considering all the evaluation metrics. The differences are particularly evident when considering the AUC-ROC values, which are 36% and 29% higher than VSM, respectively. Our results confirm what was reported by Giger *et al.* on the capabilities of *Random Forest* and, more in general, on the performance of this classifier for bug prediction [38, 69].

We compared the AUC-ROC values of the experimented models using the Scott-Knott Effect Size Difference (ESD) test [117].[11] As a result, process-based models built using *Random Forest* and *J48* are statistically better than product-based ones, while they work similarly to the combined ones.[12]

[10]Previous research [29, 57] has shown that the use of the default configuration might lead to significantly worsen the overall performance of the machine learner.

[11]ESD is an effect-size aware variant of the Scott-Knott test [107] that improves the original test in three ways: (i) it relies on hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups, (ii) it corrects non-normal distributions of a dataset, and (iii) it merges any two statistically distinct groups that have a negligible effect size into one group to avoid the generation of trivial groups. To execute the test, we employ the `ScottKnottESD` implementation: `https://github.com/klainfo/ScottKnottESD` provided by Tantithamthavorn *et al.* [117].

[12]Detailed statistical results are in our appendix [98].

Result 1: Our results, evaluated with the same strategy but on a different set of systems/timespans, confirm the findings by Giger *et al.* [46]: Method-level bug prediction models based on process metrics outperform those based on product metrics. Our results are 10 percentage points lower than those of Giger *et al.*, yet far better than random. Combining predictors with different nature improves the prediction only marginally.

5. Limitations Of The Existing Approach

By replicating the work by Giger *et al.* [46], we could identify two possible major points for improvement: the evaluation strategy and the set of features employed.

Reflecting on the evaluation strategy. Figure 1 shows an exemplification of the history of a system and how the training and testing are done in the approach by Giger *et al.* (named '10-fold overall evaluation' in the figure and depicted using red lines and text) and in the one we propose in this work (named 'release-by-release' and depicted in blue).

The system in Figure 1 has four methods (*i.e.,* M_a, M_b, M_c, M_d) that were changed several times throughout the history of the system. The changes sometimes were related to a bug (*i.e.,* the method was involved in a bug fix; purple dot), sometimes not (*i.e.,* green dot). For example method M_a was changed four times, two of which involving a bug fix. This system had at least three releases (*i.e.,* R_x throughout R_{x+1}).

The approach applied by Giger *et al.* collects all the available information until the 'data collection point', then marks a method as 'buggy' whenever the method was involved in a bug fix (hence it was buggy before the bugfix) in the entire history of the system. Then, each method would be considered as an instance to classify, where the independent variable is whether the method was marked as 'buggy' or not. In this case, the validation would be done "vertically": 10-fold cross validation ensures that the classifier is trained on a subset of methods (*e.g.,* M_a, M_b, M_c

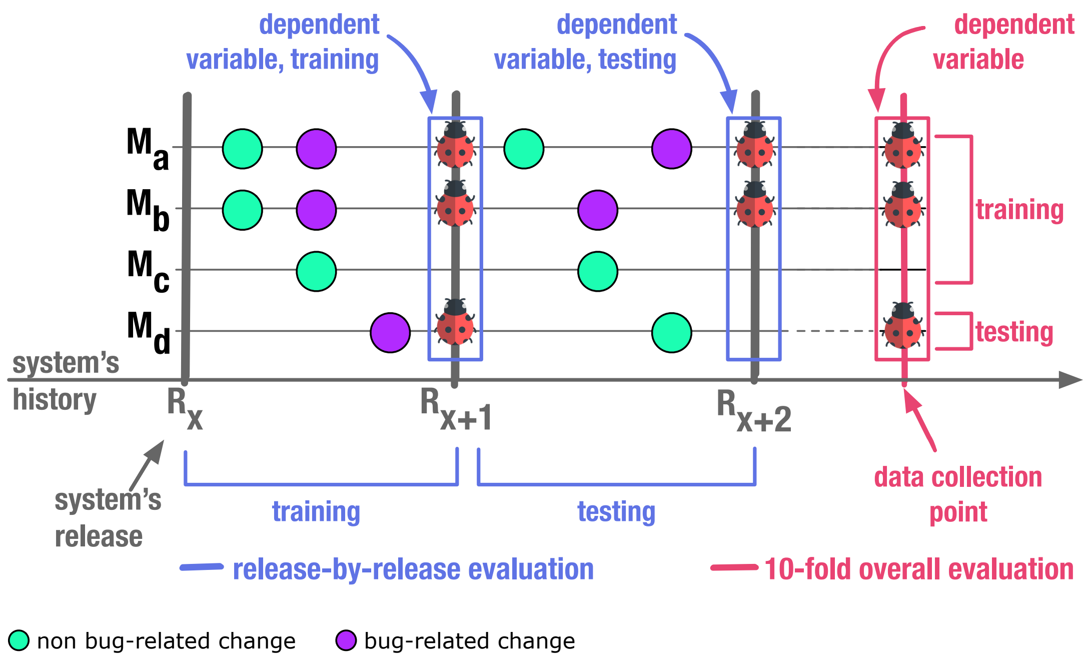

Figure 1: Training and testing strategies for method-level bug prediction.

in Figure 1) that is different from that used for the testing (*e.g.,* M_d). The limitation of this approach is that it uses dependent variables (such as most of the process metrics, including 'number of changes') (1) whose value could not be known at prediction time in a real-world scenario (*i.e.,* one would try to predict bugs that still have to occur, not that have already happened) and that (2) seem to be highly correlated to the independent variable (for each bug fix there has to be at least one change). Moreover, there are moments in which the methods were not buggy, but if they have been buggy at least once in the lifetime of the system, they are considered as buggy. Although reasonable for an initial validation, the approach followed by Giger *et al.* may lead to unreliable results.

To try to avoid unreliable results, we propose a release-by-release strategy, similar to one adopted by Kpodjedo *et al.* [65]. We train and test "horizontally" instead of "vertically": We assume the stakeholder interested in the prediction to be in the moment of a release (*e.g.,* R_{x+1} in Figure 1) and we train on all the information available from the previous release to this moment (*e.g.,* from R_x); in this case the dependent variable is whether a method has been buggy during the considered release. Then, we consider the next release (*e.g.,* R_{x+2}) and try to predict which methods will be buggy in the course of the development of this release. We do not consider any information available from the current release to the next, because this would not be available in real life. With this strategy we are going to answer $\mathbf{RQ}_2$.

An addition to the release-by-release strategy would be to consider the SZZ algorithm [111] and consider as buggy only the methods in which a bug was introduced before the release (regardless of when the fix happened). We decided *not* to follow this path for three reasons: (1) SZZ could give information that is not available at prediction time (*e.g.,* when the bug fix happens after the considered release, but the bug inducing commit happens before the release), (2) SZZ has been proven to be not reliable [30], and (3) we want to reduce at a minimum the differences from the work of Giger *et al.* we are replicating, so that the obtained results are not due to unconsidered causes.

Reflecting on the independent variables. As shown in previous work [104, 63], the choice of the features to employ when classifying defective components is key for the performance of bug prediction. In the work by Giger *et al.* [46], the authors investigated a variety of product and process metrics: while the experimental setting already dealt with the differences among these two sets of features, it is worthwhile to consider extensions to it

Looking at the existing literature, textual features have been shown to be valuable for both software quality assessment in general and class-level bug prediction in particular. Thus, we propose to investigate the extent to which features regarding code readability and alignment of comments can improve existing bug prediction models at method level. In $\mathbf{RQ}_3$, therefore, we define a set of features based on previous literature and test whether (i) a model solely based on those features perform better than the existing ones and (ii) their addition to a combined model (which also features process and product metrics) can give lead to improved performance.

Table 6: Distribution of defective methods over the releases.

		Buggy Methods Stats				
Projects	Releases	Min	Max	Average	Median	StdDev
Ant	4	627	3,302	2,061	2,158	1,350
Checkstyle	6	20	6,809	3,083	1,710	3,305
Cloudstack	2	7,323	12,213	9,768	9,768	5,069
Eclipse JDT	33	151	15,551	4,299	3,390	3,684
Eclipse Platform	3	1,194	2,702	1,984	1,984	1,006
Emf Compare	2	932	1,594	1,263	1,263	468
Gradle	4	1,321	1,666	1,467	1,416	178
Guava	17	35	3,709	1,472	1,310	1,163
Guice	4	63	2,127	1,176	1,258	867
Hadoop	5	1,666	3,321	2,467	2,462	828
Lucene-solr	7	661	6,766	3,991	4,698	2,436
Vaadin	2	6,738	11,139	8,938	8,938	3,111
Wicket	2	2,051	3,114	2,434	2,139	589
Overall	**91**	**20**	**15,551**	**3,427**	**2,139**	**3,231**

6. $\mathbf{RQ_2}$. Re-evaluating Method-Level Prediction

We seek to evaluate the performance of method-level bug prediction models in a more realistic setting.

6.1. $\boldsymbol{RQ}_2$ - Methodology

To answer $\mathbf{RQ_2}$, we need to (i) extract all the releases of the considered projects, (ii) identify the buggy methods in each of the releases, and (iii) build the three bug prediction models used for $\mathbf{RQ}_1$.

Extracting The Major Releases. The first step to test the performance of method-level bug prediction models is the identification of the major releases in the considered systems. To this purpose, we automatically extract the releases from the list of releases declared on the Github repository of the subject systems. To discriminate a major release from the others, we rely on a heuristic based on naming conventions: If the version name ends with the patterns 0 or 0.0 (*e.g.,* versions 3.0 or 3.0.0), then we define it as a major release.[13]

Extraction of Bug Data. Differently from what we have done in $\mathbf{RQ}_1$, in this research question we need to extract the data about the bugs for *all* the considered releases. For each release pair r_{x-1} and r_x, we (i) run ReLink and (ii) consider as buggy all the methods actually changed in the buggy commits detected by ReLink and referring to the time frame between r_{x-1} and r_x. We removed the test cases, as in $\mathbf{RQ}_1$. Table 6 summarizes the distribution of buggy methods considering every inspected release for each project.

Bug Prediction Models: Setup. As done for $\mathbf{RQ}_1$, we test the performance of three bug prediction models, *i.e.,* the ones relying on (i) product metrics *only*, (ii) process metrics *only*, and (ii) *both* product and process metrics, built using the same set of machine learning approaches (*i.e., Random Forest*, *Support Vector Machine*, *Bayesian Network*, and *J48*). The training data is pre-processed to avoid (i) data unbalance and (ii) multicollinearity, by using the techniques previously exploited (*i.e., Random Over-Sampling* algorithm [25] and *Correlation-based Feature Selection* [49], respectively).

Bug Prediction Models: Validation. We test the performance of the prediction models by applying an *inter-release* validation procedure, *i.e.,* we trained the prediction models using the release r_{x-1} and tested it on r_x. This technique implies that neither the first release of each system can be used as testing set nor the last release can be used as training. To measure the performance, we computed the same set of metrics previously exploited, *i.e.,* precision, recall, F-Measure, and AUC-ROC.

6.2. $\boldsymbol{RQ}_2$ - Results

Table 7 reports the median *precision*, *recall*, *F-measure*, and *AUC-ROC* achieved by models based on (i) only product, (ii) only process, and (iii) both product and process metrics when using different classifiers and the release-by-release strategy.[14]

The performance achieved by all the prediction models experimented is substantially lower than those found for $\mathbf{RQ}_1$. We observe a lower variance in the results, for each of the subject systems in our dataset.

In this evaluation scenario, code metrics achieve better performance than process metrics. This is in contrast with past literature reporting the superiority of process metrics for bug prediction [104, 106]. We hypothesize that this result may be caused both by the different granularity of the experimented models and by the different validation strategy. In particular, while the historical information computed at class-level could better characterize the complexity of the development process followed by developers while implementing changes in an entire class [53], it is reasonable to think that the bugginess of source code methods may be better expressed by the methods' current code quality. An additional possible cause that refutes the observation of previous studies [104, 106] comes from the irregular distribution of the length of the time frames for the considered releases. In our analyzed projects, these intervals stretch from a few months to a couple of years and the distribution of the releases is strictly correlated to the needs and the approach adopted by developers in a given historical moment. The higher prediction capabilities of code metrics are confirmed also when looking at other indicators, *i.e.,* precision, recall, AUC-ROC. Moreover, this result holds for all the classifiers considered.

Finally, the performance of the different classifiers has also lower variance. To some extent, this result confirms previous findings in the field [45, 95] showing that different classifiers achieve similar performance. Alternatively, this similarity in results and the low performance may indicate that no machine learning algorithm detects a valid signal

[13]We manually verified the performance of this heuristic on one of the subject systems: We verified that all the major releases of Lucene-Solr were correctly caught, thus quantifying the actual performance of this approach.

[14]Our appendix provides detailed reports [98].

Table 7: Median classification results of method-level bug prediction models when validated using a release-by-release strategy.

S = Product H = Process	Precision S	Precision H	Precision S&H	Recall S	Recall H	Recall S&H	F-measure S	F-measure H	F-measure S&H	AUC-ROC S	AUC-ROC H	AUC-ROC S&H
Bayesian Network	0.72	0.70	0.70	0.58	0.64	**0.65**	0.59	0.60	0.61	0.53	0.52	0.53
J48	0.71	0.71	0.71	0.59	0.59	0.59	0.62	0.62	0.63	0.51	0.51	0.51
Random Forest	0.72	0.70	0.72	0.63	0.60	0.63	**0.64**	0.61	0.63	0.52	0.51	0.52
Support Vector Machines	0.72	**0.73**	0.72	0.59	0.57	0.60	0.62	0.58	0.62	0.53	0.53	**0.53**
Overall	**0.71**	**0.71**	**0.71**	**0.59**	**0.60**	**0.60**	**0.62**	**0.60**	**0.61**	**0.52**	**0.52**	**0.53**

from the considered metrics. This result potentially highlights the possibility to further study the orthogonality of classifiers for method-level bug prediction with the aim of exploiting ensemble methodologies [38, 69].

Result 2: Method-level bug prediction models resulted in much lower performance (up to 20 points percentage less) in terms of AUC-ROC and similar precision when evaluated with the more practical release-by-release evaluation strategy. The achieved AUC-ROC scores achieved by all the models, regardless of the machine learning approach, are close to the results of a random classification.

7. RQ$_3$. Evaluating Alternative Metrics

Our **RQ$_3$** seeks to evaluate the potential of alternative metrics for method-level bug prediction.

*7.1. **RQ**$_3$ - Methodology*

We define a set of metrics based on textual aspects of source code as well as on code smells, and developer-related factors, which we include in the method-level bug prediction models experimented in our previous research questions. In the following, we first describe the metrics and the rationale for their choice, and then we detail the model definition and evaluation process.

Textual Metrics. As summarized in Section 2, previous work [1, 12, 16, 58] highlighted that textual aspects of source code could impact its bug proneness and be useful when considered within bug prediction models. Thus, we first challenge these findings and explore the role of textual features when applied to method-level bug prediction. Table 8 lists the considered metrics. We include:

Code Readability. Based on the findings by Buse and Weimer [16] and Binkley *et al.* [12], we compute a measure of readability of source code. We directly employ the tool proposed in [16]: This outputs an index, which is a decimal score ranging between 0 and 1, where 0 represents unreadable code and 1 refers to easily readable code. This tool relies on a readability model composed of 19 metrics (including line length, number and length of identifiers, number of a predefined list of characters, branches, loops). To compute it, we rely on the publicly available version of the tool provided by the authors.[15] Code readability is not code complexity, which we already defined in other metrics previously.

Table 8: List of the considered method-level textual metrics.

Metric name	Description (applies to method level)
Readability	Source code readability index [16]
Textual Coherence	Measure of the textual coherence between source and code comments [58]
Purpose	# of code comments used to describe the functionality of linked source code
Notice	# of code comments related to the description of warning, alerts, or messages
Under Development	# of code comments covers the topics related to the ongoing and future development
Style&IDE	# of code comments used to logically separate the code or provide special services
Metadata	# of code comments used to classify comments that define meta-information about the code
Other	# of code comments that do not fit into the previously defined categories

Textual Coherence. Based on the findings reported by Walid *et al.* [58], we measure the textual coherence, *i.e.,* the extent to which comment and source code of a method are aligned. To compute it, we first normalize comments and source code using a standard Information Retrieval (IR) process [4].[16] Then, we apply the Vector Space Model (VSM) [4] and measure the textual similarity between comments and source code (*i.e.,* the vectors of VSM) using the cosine distance.

Comment Classification. Based on the findings by Aman *et al.* [1], who reported that different comments types are associated to different bug proneness, we classify source code comments exploiting the model proposed by Pascarella and Bacchelli [96]: It analyzes the text contained in a comment and classifies its semantic. Specifically, Pascarella and Bacchelli defined a hierarchical taxonomy with two levels: the first coarse-grained contains 6 categories, while the second fine-grained contains 16 sub-categories. In our study, we define 6 new

[15]URL: `http://www.arrestedcomputing.com/readability`.

[16]In detail, we (i) separate composed identifiers, (ii) lower case the extracted words, (iii) remove special characters, programming keywords, and common English stop words, and (iv) stem words to their original roots via Porter's stemmer [101].

method-level textual metrics based on the first level, as described by the last six rows in Table 8.

Code smells. These are poor design or implementation choices introduced by developers when maintaining and/or evolving software systems [43, 121]. The addition of the information on the design quality of classes into existing bug prediction models has been proved to improve bug identification capabilities [93, 114]. The contribution of measures of code smell severity appeared to be particularly useful in class-level bug prediction [93]. Following these findings, in the context of our work, we first identify code smell types that affect methods and then compute their intensity. We focus on:

Long Method. This smell refers to methods implementing more than one functionalities and is generally detected by considering its size [43]. Methods affected by this smell are poorly cohesive and possibly impact their understandability, change- and defect-proneness [87].

Long Parameter List. This smell refers to methods having a long list of parameters. Instances of this smell can lower the maintainability of methods and possibly indicate that the method is poorly cohesive [43].

Message Chains. This smell occurs when a client requests an object; this requires yet another one, and so on, thus creating a long concatenation of method calls [43]. This smell has been associated to a higher defect-proneness of the affected methods [32, 87].

The rationale behind the selection of these code smell types is twofold. In the first place, these have been reported to occur in software projects [87]. Perhaps more importantly, they influence the bug-proneness of the affected methods [87, 32], thus perfectly fitting the goal of our paper.

To detect them we rely on DECOR [75], a method to define code smell detection rules using a Domain-Specific Language. The approach uses a set of rules, called "rule cards",[17] which describe the intrinsic characteristics that a method should have to be affected by a certain code smell type. In the case of *Long Method*, DECOR identifies it by considering the number of lines of code of a method: If this is higher than 80, then a code smell is detected. As for *Long Parameter List*, it considers a method affected by this smell if it has more than three parameters. Finally, *Message Chains* instances are detected if a method contains a statement in which more than three method calls are performed.

According to several empirical studies [88, 89, 100], the accuracy of DECOR is relatively high both in terms of precision and recall, with typical values of F-Measure around 75%. This makes the detector more accurate than other available tools [41] and, therefore, suitable for our study.

Once detected code smell instances, we compute their intensity. We follow a similar approach as previous work [89, 90]: as DECOR classifies a method as smelly if a specific condition is satisfied, for instance, if its lines of code > 80, we can say that the higher the distance between the actual code metric value and the fixed threshold, the higher the intensity of the smell. We use this approach to compute the intensity of all the three smells considered.

Developer-related factors. Aspects capturing how developers work on source code and what is the change process they apply when performing software maintenance and evolution activities have been often successfully applied in bug prediction as they showed a great potential for improving predictive models [14, 24, 37, 53, 85].

These previous findings lead us to consider how developer-related factors work when employed in the context of method-level bug prediction. In particular, we focus on three orthogonal aspects such as:

Number of developers. In the first place, for each method of the considered dataset, we compute how many distinct developers worked on it over the history of the project. The contribution of this metric to bug prediction capabilities was firstly assessed by Ostrand *et al.* [10, 85], who reported that individual developer's data provides a limited boost to bug prediction models; Nevertheless, Di Nucci *et al.* [37] performed a larger empirical evaluation of the value of this metric, showing that it can improve the performance of these models by up to 10%. This is the reason why we seek to understand its value at a finer-level. We compute the metric by (i) mining all commits in the change history that changed a method m and (ii) counting the number of developers who made changes to it. To distinguish different developers, we consider the e-mail address they left on GITHUB—we are aware that this computation may be imprecise in cases where a developer uses multiple e-mails when working on the project, however there is no practical way to solve this problem.

Code ownership. According to Bird *et al.* [14], developers having a higher experience on the source code they touch are less prone to introduce bugs. The authors assessed this relation by computing the code ownership of classes and measuring its impact on the performance of bug prediction models, finding that models including this metric have an accuracy that is 24% higher than those not including it as a feature. In our work, we compute code ownership at method-level by following the same approach of Bird *et al.* [14]: given a method m, we compute the ratio of number of commits that a contributor c has made on m with respect to the total number of commits made by c. Once computed the metric for all developers who contributed to m, we assign to the method the maximum ownership computed.

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

Entropy of code changes. Finally, we take into account the way developers make changes in a system and compute the entropy of changes originally defined by Hassan [53] for class-level bug prediction. This metric has been shown to strongly impact on the performance of predictive maintenance models [22, 53]. Following the algorithm of Hassan [53], we first identify all commits which modified a method m and then run a pattern-based technique that can detect the so-called *feature-introduction* modifications, namely changes applied to introduce new or enhancing existing features. Afterwards, the entropy of changes on m is computed exploiting the concept of Shannon entropy [108] as in the following equation:

$$entropy(m, \alpha) = -(p_{m,\alpha} \cdot \log_2 p_{m,\alpha}) \tag{4}$$

where $p_{m,\alpha}$ indicates the probability that m was changed received *feature introduction* modifications over its change history. This probability is computed considering the fraction between the number of *feature introduction* modifications applied on m in the change history over the total number of *feature introduction* modifications applied by developers.

It is important to remark that other alternative metrics have been proposed by researchers in the bug prediction field, like for instance socio-technical [37, 102] or code coverage features [15]. Being aware of those alternative metrics, we decided to exclude them from our study as they cannot be easily computed at method-level. For instance, let consider the case of socio-technical congruence [19]: this measures how much the organizational structure of a development community reflects the actual technical organization among developers. While the metric has been defined in terms of how much the developers' social network matches the modularization of packages, no definition is available with respect to the socio-technical congruence between developers' social network and division of methods among classes. Thus, we preferred to be conservative and not define any novel metric that would have deserved a separate validation before being used in our context. Rather, we relied on metrics that can be directly computed at method-level, *e.g.,* the considered code smells are all directly computable at method-level.

Model Definition and Data Analysis. To test the contribution given by the considered families of features, we build five classes of method-level bug prediction models on the basis of those defined for **RQ**$_1$. Our methodology is inspired by the one of Bird *et al.* [14]: starting from the baseline one, namely the *product + process* one defined by Giger *et al.* [46], we progressively fed up the model with additional features, so that we can measure the contribution given by each family of features to the capabilities of method-level bug prediction (if any). Hence, we build models relying on (i) *product + process + textual* features, (ii) *product + process + textual + code smell-related* features, and (iii) *product + process + textual + code smell-related + developer-related* features. Furthermore, we also build models relying on the various families of features independently so that we can assess the independent value of each of them for method-level bug prediction.

As done in the context of the previous research questions, we apply the RANDOM OVER-SAMPLING and *Correlation-based Feature Selection* [49] algorithms to deal with data balancing and multicollinearity. The performance of all experimented method-level bug prediction models are evaluated using the same validation strategy (*i.e.,* release-by-release) and evaluation metrics (*i.e.,* precision, recall, F-Measure, and AUC-ROC) used in the context of the previous research question. We also conduct a statistical comparison of the performance of the prediction models considered. The Mann-Whitney test [28] is not recommended in the case of comparisons of multiple models over multiple datasets, since the performance of a specific model might vary between two datasets [36]. Thus, we compared the AUC-ROC values of the experimented models over the different systems using the Scott-Knott Effect Size Difference (ESD) test [117].

7.2. ***RQ***$_3$ *- Results*

Figures 2 and 3 show the distribution of F-Measure and AUC-ROC, respectively, achieved by the prediction models built progressively using the various families of features considered in our study. We also report on the performance achieved by the models *only* relying on individual sets of features. Consistently with the methodology adopted for the first research question, we analyze how their performance varies when considering different classifiers (*i.e., Simple Logistic*, *Logistic*, *Multilayer Perceptron*, *Random Forest*, *J48*, *Decision Table*, and *Naive Bayes*). However, we limit the discussion of the results to *Random Forest* because it was the classifier providing slightly better performance (see Table 5).[18]

When looking at the figures, we can see that the models built using individual sets of features have poor performance: for instance, the AUC-ROC of the *only textual* model is close to 53%, which indicates that it is just slightly better than a random classifier. A similar discussion can be drawn for the other individual models, thus confirming that taking those features alone does not give advantages when it comes to the prediction of defective methods. These results confirm what has been previously reported in the literature on the importance of combining multiple features to boost the performance of bug prediction models [5, 37, 104].

Turning the attention to the combined models, we notice that the progressive addition of metrics only gives marginal contribution to the overall classification accuracy. Specifically, the inclusion of textual metrics into a model

[18]A complete overview of the results achieved when using other classifiers is available in our appendix [98].

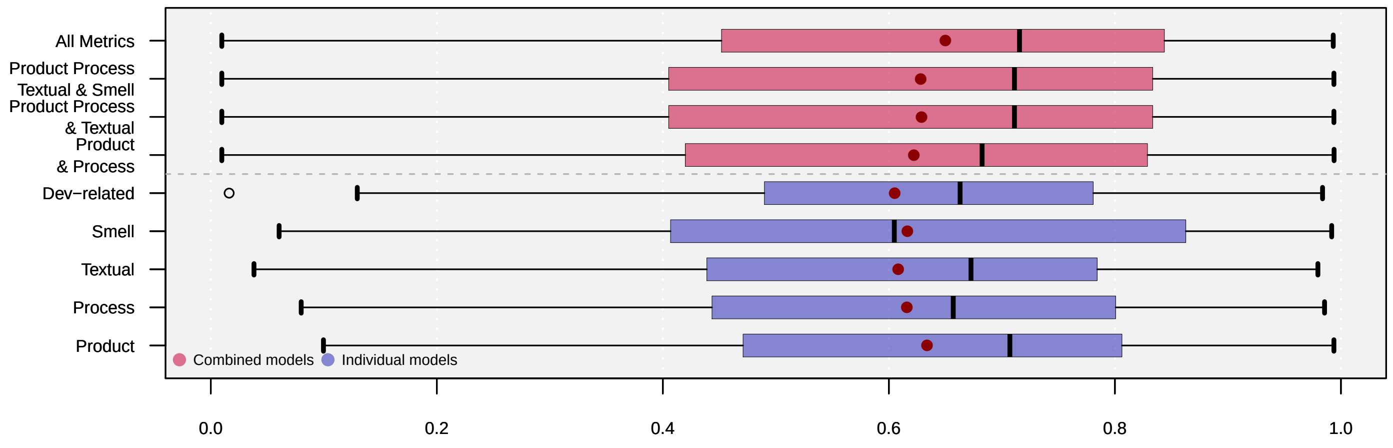

Figure 2: Comparison of the distribution of F-measure values considering the combination of product, process and textual metrics.

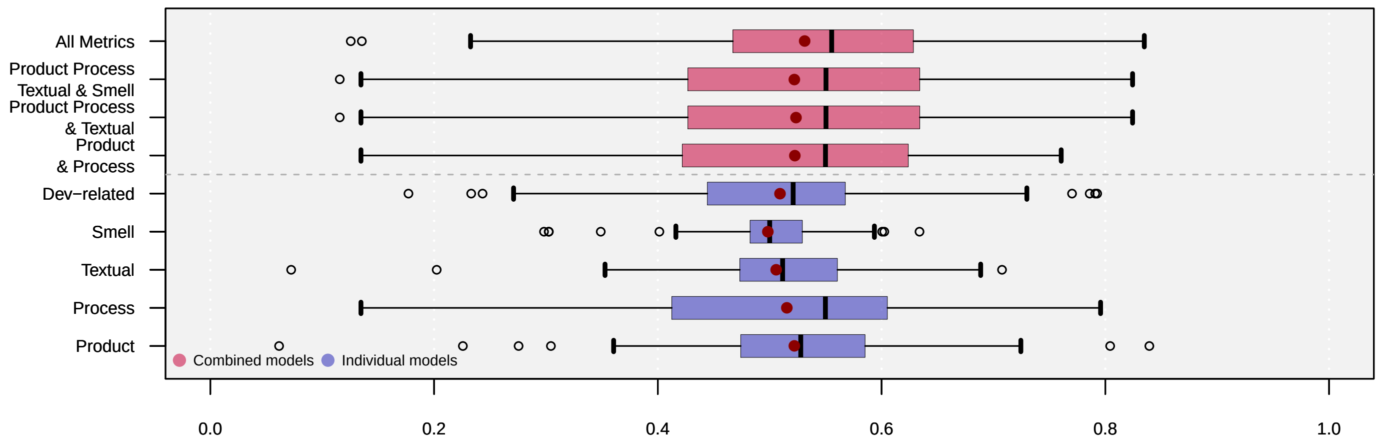

Figure 3: Comparison of the distribution of AUC-ROC values considering the combination of product, process and textual metrics.

containing product and process metrics allows the model to have 4% and 5% more F-Measure and AUC-ROC. In the first place, this indicates that these metrics provide a limited amount of additional information to predict future bugs. At the same time, our findings represent a negative result with respect to the findings reported by all prior studies that we exploited to derive the textual features [1, 12, 16, 58]; indeed, we could not find any textual measure able to significantly increase the performance of the experimented prediction models.

The discussion is similar when considering the addition of code smell-related information. According to previous findings in the field [93, 114], including a measure of code smell severity within models relying on a combination of process and product metrics provides an increase of ≈15% in terms of F-Measure. Unfortunately, this is not the case when lowering the granularity of the predictions. In our case, indeed, the F-Measure of the *product + process + textual + code smell-related* model is even lower than the one not including any smell-related information (-2%). This may indicate that code smells computed at method-level have a limited predictive power when compared to class-level code smells. Thus, our findings are again negative with respect to previous work [93, 114]. Perhaps more importantly, we could not confirm the results of D'Ambros *et al.* [32], in which the authors reported that one of the considered smells, *i.e., Message Chains*, is the one that mostly affects the bug-proneness of source code methods.

As for the developer-related factors (named *All metrics* in Figures 2 and 3), their inclusion provides an increase of 4% with respect to the second best performing model (the *product + process + textual* one). So, also in this case, we claim that the contribution is marginal and that we could not confirm the role of developer-related factors for bug prediction when the granularity is that of methods.

The results discussed so far are all statistically significant. According to the ranking of the performance provided by the Scott-Knott ESD test [117], there is no model performing statistically better than the others, thus indicating that the addition of alternative metrics does not boost the performance of bug prediction.[19]

To conclude the discussion, based on the findings discussed so far we argue that the research on method-level bug prediction still needs notable steps to do for better supporting developers. Our paper provides initial compelling evidence of the need of novel, specific metrics able

[19]Detailed statistical results are reported in our appendix [98].

to capture method-level information that actually relate to the bug-proneness of methods.

Result 3: The addition of alternative features based on textual, code smells, and developer-related factors improves the performance of the existing models only marginally, if at all.

8. Threats to Validity

We describe the factors that can affect the validity of our empirical results.

Threats to Construct Validity. A first factor influencing the relationship between theory and observation is related to the dataset exploited. In our study, we relied on the same methodology previously adopted by Giger *et al.* [46] to build our own repository of buggy methods, *i.e.,* we first retrieved bug-fixing commits using the textual-based technique proposed by Fisher *et al.* [42] and then considered as buggy the methods changed in that commits. While we cannot exclude possible imprecision and/or incompleteness of the data used in this study, we have re-evaluated the performance of the tool by Fisher *et al.* in our context finding that it could detect buggy commits with a precision of 84% correctly. Still in this category, we re-implemented the product and process metrics used to build the experimented models. This was due to the lack of a publicly available tool. When re-implementing such metrics we faithfully followed the descriptions provided by Giger *et al.* [46]. As for the alternative metrics, instead, we used available tools whenever possible (*e.g.,* in the case of the comment classification [96] or when detecting method-level code smells [75]). To enable and stimulate the replicability of our study, we made all tools and scripts exploited publicly available in our online appendix [98]. As for the selection of the classifier to use when building the bug prediction model, we tested the performance of different classifiers, finding RANDOM FOREST to be the one providing the best performance. All the tested classifiers use the default parameters, since finding the best configuration for all of them would have been too expensive [11]. Future work can be devised to investigate the impact of parameters' configuration on our findings.

Threats to Conclusion Validity. A first point of discussion regards the data pre-processing techniques adopted before the construction of the experimented bug prediction models. To ensure that the results would not have been biased by confounding effects such as data unbalance [26] or multi-collinearity [40], we adopted formal procedures aimed at (i) over-sampling the training sets [26] and (ii) removing non-relevant independent variables through feature selection [49]. As for the evaluation of the models, we complemented the results concerning the F-measure by relying on a *threshold-independent* metric such as the AUC-ROC. Furthermore, we supported our findings with an appropriate statistical test like the Scott-Knott ESD one [117].

Threats to External Validity. This category refers to the generalizability of our findings. While in the context of this work we analyzed software projects having different size and scope, we limited our focus to Java systems because some of the tools exploited to compute the considered metrics only work for this programming language (*e.g.,* certain code smells have been only defined for Java [43]). Thus, we cannot claim generalizability concerning systems written in different languages as well as to projects belonging to industrial environments. Similarly, we considered a subset of the available metrics in each of the five families of features considered: in particular, we limited ourselves to the analysis of the metrics which previous works have analyzed, while we cannot exclude that different results could be achieved when considering different metrics (*e.g.,* other method-level code smell types).

9. Conclusion

We investigated (i) the performance of different types of method-level bug prediction models when applied in a real-case scenario and (ii) the contribution given by textual features to existing bug prediction models. The main contributions made by our study are:

1. A re-evaluation on different systems/timespans of previously defined method-level bug prediction models. The results confirm previous findings in the field [46].

2. An empirical analysis of how the performance of existing method-level bug prediction models change when applied to a more realistic, release-by-release scenario. Our results provide evidence that current method-level bug prediction models do not dramatically outperform a random classifier; hence we reveal the need for further research in this area.

3. An empirical analysis of whether the performance of existing method-level bug prediction models can be improved by considering a set of 8 textual features. Our results reveal that the overall prediction capabilities lead to negligible improvements.

4. An online appendix [98] that reports the dataset and all the additional analyses performed in the work described in this paper.

Based on the results achieved so far, our future agenda includes (i) the replication of our study on a broader set of systems also considering ensemble methods [38, 69], (ii) the investigation of novel metrics to properly work for method-level bug prediction, and (iii) an *in-vivo* analysis of the capabilities of method-level bug prediction models, involving practitioners during their daily activities [66].

Acknowledgment

A. Bacchelli and F. Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2_170529 and No. PZ00P2_186090. This project has received funding from the European Union's H2020 programme under the Marie Sklodowska-Curie grant agreement No 642954. 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 our work.

References

[1] Aman, H., Amasaki, S., Sasaki, T., and Kawahara, M. 2014. Empirical analysis of fault-proneness in methods by focusing on their comment lines. In *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 2. IEEE, 51–56.

[2] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 23.

[3] Bacchelli, A., D'Ambros, M., and Lanza, M. 2010. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 59–73.

[4] Baeza-Yates, R., Ribeiro-Neto, B., et al. 1999. *Modern information retrieval*. Vol. 463. ACM press New York.

[5] Basili, V., Briand, L., and Melo, W. 1996. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on 22,* 10, 751–761.

[6] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. 2012. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE, 104–113.

[7] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. 2014. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering 19,* 6, 1617–1664.

[8] Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and Lucia, A. d. 2014. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23,* 1, 4.

[9] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and De Lucia, A. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering 40,* 7, 671–694.

[10] Bell, R., Ostrand, T., and Weyuker, E. 2013. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering 18,* 3, 478–505.

[11] Bergstra, J. and Bengio, Y. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13,* Feb, 281–305.

[12] Binkley, D., Feild, H., Lawrie, D., and Pighin, M. 2009. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software 82,* 11, 1793–1803.

[13] Bird, C., Nagappan, N., Gall, H., Murphy, B., and Devanbu, P. 2009. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*. IEEE, 109–119.

[14] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 4–14.

[15] Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., and Wu, F. 2016. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 330–341.

[16] Buse, R. P. and Weimer, W. R. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering 36,* 4, 546–558.

[17] Buy, U., Orso, A., and Pezze, M. 2000. Automated testing of classes. In *ACM SIGSOFT Software Engineering Notes*. Vol. 25. ACM, 39–48.

[18] Canfora, G., De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., and Panichella, S. 2013. Multi-objective cross-project defect prediction. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 252–261.

[19] Cataldo, M., Herbsleb, J. D., and Carley, K. M. 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2–11.

[20] Catolino, G. 2017. Just-in-time bug prediction in mobile applications: the domain matters! In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, 201–202.

[21] Catolino, G., Di Nucci, D., and Ferrucci, F. 2019. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[22] Catolino, G., Palomba, F., Arcelli Fontana, F., De Lucia, A., Zaidman, A., and Ferrucci, F. 2019. Improving change prediction models with code smell-related information. *Empirical Software Engineering*, to appear.

[23] Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., and Zaidman, A. 2017. Developer-related factors in change prediction: an empirical assessment. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 186–195.

[24] Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., and Zaidman, A. 2018. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software 143*, 14–28.

[25] Chawla, N. V. 2009. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*. Springer, 875–886.

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

[27] Chidamber, S. R. and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE) 20,* 6, 476–493.

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

[29] Corazza, A., Di Martino, S., Ferrucci, F., Gravino, C., Sarro, F., and Mendes, E. 2013. Using tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering 18,* 3, 506–546.

[30] da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., and Hassan, A. E. 2017. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering 43,* 7, 641–657.

[31] Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T., and Kim, C.-J. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 46–57.

[32] D'Ambros, M., Bacchelli, A., and Lanza, M. 2010. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 23–31.

[33] D'Ambros, M., Lanza, M., and Robbes, R. 2010. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on.* IEEE, 31–41.

[34] DAmbros, M., Lanza, M., and Robbes, R. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering 17,* 4, 531–577.

[35] Dean, D. J., Nguyen, H., and Gu, X. 2012. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing.* ACM, 191–200.

[36] Demšar, J. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research 7,* Jan, 1–30.

[37] Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., and De Lucia, A. 2017. A developer centered bug prediction model. *IEEE Transactions on Software Engineering.*

[38] Di Nucci, D., Palomba, F., Oliveto, R., and De Lucia, A. 2017. Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence 1,* 3, 202–212.

[39] Diakopoulos, N. and Cass, S. 2016. The top programming languages 2016. *IEEE Spectrum `http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016`.*

[40] Dietterich, T. 1995. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR) 27,* 3, 326–327.

[41] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering.* ACM, 18.

[42] Fischer, M., Pinzger, M., and Gall, H. 2003. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* IEEE, 23–32.

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

[44] Friedman, J., Hastie, T., and Tibshirani, R. 2001. *The elements of statistical learning.* Vol. 1. Springer series in statistics New York.

[45] Ghotra, B., McIntosh, S., and Hassan, A. E. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 789–800.

[46] Giger, E., D'Ambros, M., Pinzger, M., and Gall, H. C. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement.* ACM, 171–180.

[47] Greiler, M., Herzig, K., and Czerwonka, J. 2015. Code ownership and software quality: a replication study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* IEEE, 2–12.

[48] Gyimóthy, T., Ferenc, R., and Siket, I. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE) 31,* 10, 897–910.

[49] Hall, M. A. 1999. Correlation-based feature selection for machine learning.

[50] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering 38,* 6, 1276–1304.

[51] Hanley, J. A. and McNeil, B. J. 1982. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology 143,* 1, 29–36.

[52] Hartzman, C. S. and Austin, C. F. 1993. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1.* IBM Press, 138–170.

[53] Hassan, A. E. 2009. Predicting faults using the complexity of code changes. In *ICSE.* IEEE Press, Vancouver, Canada, 78–88.

[54] Hata, H., Mizuno, O., and Kikuno, T. 2012. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 200–210.

[55] Hoang, T., Dam, H. K., Kamei, Y., Lo, D., and Ubayashi, N. 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories.* IEEE Press, 34–45.

[56] Hosseini, R., Turhan, B., and Gunarathna, D. 2017. A systematic literature review and meta-analysis on cross project defect prediction.

[57] Huang, C., Davis, L., and Townshend, J. 2002. An assessment of support vector machines for land cover classification. *International Journal of remote sensing 23,* 4, 725–749.

[58] Ibrahim, W. M., Bettenburg, N., Adams, B., and Hassan, A. E. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software 85,* 10, 2293 – 2304. Automated Software Evolution.

[59] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., and Hassan, A. E. 2010. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on.* IEEE, 1–10.

[60] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering 39,* 6, 757–773.

[61] Khaled El Emam, W. M. and Machado, J. C. 2001. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software 56,* 1, 63–75.

[62] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., and Antoniol, G. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering 17,* 3, 243–275.

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

[64] Kittler, J. et al. 1982. Pattern recognition. a statistical approach.

[65] Kpodjedo, S., Ricca, F., Galinier, P., Guéhéneuc, Y.-G., and Antoniol, G. 2011. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering 16,* 1, 141–175.

[66] Lanza, M., Mocci, A., and Ponzanelli, L. 2016. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software 33,* 6, 102–105.

[67] Lehman, M. M. and Belady, L. A. 1985. *Program evolution: processes of software change.* Academic Press Professional, Inc.

[68] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Whitehead Jr, E. J. 2013. Does bug prediction support human developers? Findings from a Google case study. In *Proceedings of the 2013 International Conference on Software Engineering.* ICSE 2013. IEEE Press, 372–381.

[69] Malhotra, R. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing 27,* 504–518.

[70] Marcus, A. and Maletic, J. I. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th international conference on software engineering.* IEEE Computer Society, 125–135.

[71] Marcus, A., Poshyvanyk, D., and Ferenc, R. 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering 34,* 2, 287–300.

[72] McCabe, T. J. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4, 308–320.

[73] Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., Turhan, B., and Zimmermann, T. 2013. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering 39,* 6, 822–834.

[74] Menzies, T., Greenwald, J., and Frank, A. 2007. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering 33,* 1, 2–13.

[75] Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering 36,* 1, 20–36.

[76] Moser, R., Pedrycz, W., and Succi, G. 2008. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* ESEM '08. ACM, New York, NY, USA, 309–311.

[77] Nagappan, N. and Ball, T. 2005. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering.* ICSE '05. ACM, New York, NY, USA, 580–586.

[78] Nagappan, N., Ball, T., and Zeller, A. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering.* ICSE '06. ACM, New York, NY, USA, 452–461.

[79] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. 2010. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on.* IEEE, 309–318.

[80] Nguyen, T. T., Nguyen, T. N., and Phuong, T. M. 2011. Topic-based defect prediction (NIER track). In *Proceedings of the 33rd international conference on software engineering.* ACM, 932–935.

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

[82] O'brien, R. M. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity 41,* 5, 673–690.

[83] Ohlsson, N. and Alberg, H. 1996. Predicting fault-prone software modules in telephone switchess. *Software Engineering, IEEE Transactions on 22,* 12, 886–894.

[84] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering 31,* 4, 340–355.

[85] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. 2010. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering.* PROMISE '10. ACM, New York, NY, USA, 19:1–19:10.

[86] Pai, G. J. and Dugan, J. B. 2007. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on software Engineering 33,* 10.

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

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

[89] Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., and De Lucia, A. 2017. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*.

[90] Palomba, F., Tamburri, D. A. A., Fontana, F. A., Oliveto, R., Zaidman, A., and Serebrenik, A. 2018. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering*.

[91] Palomba, F., Zaidman, A., and Lucia, A. 2018. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE.*

[92] Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., and Oliveto, R. 2016. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on.* IEEE, 244–255.

[93] Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., and Oliveto, R. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering 45,* 2, 194–218.

[94] Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 522–531.

[95] Panichella, A., Oliveto, R., and De Lucia, A. 2014. Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on.* IEEE, 164–173.

[96] Pascarella, L. and Bacchelli, A. 2017. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories.* IEEE Press, 227–237.

[97] Pascarella, L., Palomba, F., and Bacchelli, A. 2018a. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 592–601.

[98] Pascarella, L., Palomba, F., and Bacchelli, A. 2018b. Re-evaluating method-level bug prediction - online appendix. *https://doi.org/10.5281/zenodo.3518990*.

[99] Pascarella, L., Palomba, F., and Bacchelli, A. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software 150*, 22–36.

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

[101] Porter, M. F. 1980. An algorithm for suffix stripping. *Program 14,* 3, 130–137.

[102] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. 2013. Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 452–461.

[103] Rahman, F. and Devanbu, P. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 491–500.

[104] Rahman, F. and Devanbu, P. 2013. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 432–441.

[105] Rahman, F., Posnett, D., and Devanbu, P. 2012. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* ACM, 61.

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

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

[108] Shannon, C. E. 1951. Prediction and entropy of printed english. *Bell system technical journal 30,* 1, 50–64.

[109] Shihab, E., Hassan, A. E., Adams, B., and Jiang, Z. M. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* ACM, 62.

[110] Shivaji, S., Whitehead, E. J., Akella, R., and Kim, S. 2013. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering 39,* 4, 552–569.

[111] Śliwerski, J., Zimmermann, T., and Zeller, A. 2005. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes 30,* 4, 1–5.

[112] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. 2018. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE.*

[113] Subramanyam, R. and Krishnan, M. S. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on 29,* 4, 297–310.

[114] Taba, S. E. S., Khomh, F., Zou, Y., Hassan, A. E., and Nagappan, M. 2013. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 270–279.

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

[116] Tan, M., Tan, L., Dara, S., and Mayeux, C. 2015. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2.* IEEE Press, 99–108.

[117] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* IEEE, 321–332.

[118] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. 2017. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering 43,* 1, 1–18.

[119] Todd L. Graves, Alan F. Karr, J. S. M. and Siy, H. P. 2000. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on 26,* 7, 653–661.

[120] Tosun, A., Bener, A., Turhan, B., and Menzies, T. 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology 52,* 11, 1242–1257.

[121] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and 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.

[122] Turhan, B., Menzies, T., Bener, A. B., and Di Stefano, J. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering 14,* 5, 540–578.

[123] Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering.* IEEE Press, 215–223.

[124] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* ACM, 15–25.

[125] Yang, X., Lo, D., Xia, X., Zhang, Y., and Sun, J. 2015. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on.* IEEE, 17–26.

[126] Zhang, F., Zheng, Q., Zou, Y., and Hassan, A. E. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 309–320.

[127] Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 91–100.

[128] Zimmermann, T., Premraj, R., and Zeller, A. 2007. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering.* PROMISE '07. 9–.