Re-evaluating Method-Level Bug Prediction

Luca Pascarella
Delft University of Technology
The Netherlands
L.Pascarella@tudelft.nl

Fabio Palomba
University of Zurich
Switzerland
palomba@ifi.uzh.ch

Alberto Bacchelli
University of Zurich
Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Bug prediction is aimed at supporting developers in the identification of code artifacts more likely to be defective. Most approaches defined so far target the prediction of bugs at class-level, thus pinpointing the presence of a bug in an entire source file. Nevertheless, past research has provided evidence that this granularity might be too coarse-grained, thus reducing the usability of bug prediction in practice. As a consequence, researchers have started proposing method-level bug prediction models, showing promising evidence that it is possible to operate at this level of granularity.**

In this study, we first replicate previous research on method-level bug prediction on different systems/timespans. Afterwards, we reflect on the evaluation strategy and propose a more realistic one. Key results of our study show that the performance of the method-level bug prediction model is similar to what previously reported also for different systems/timespans, when evaluated with the same strategy. However—when evaluated with a more realistic strategy—all the models show a dramatic drop in performance showing results close to that of a random classifiers. Our replication and negative results indicate that method-level bug prediction is still an open challenge.

***Index Terms*—empirical software engineering; bug prediction; replication; negative results**

I. Introduction

The last decade has seen a remarkable involvement of software artifacts in our daily life [1]. Reacting to the frenzied demands of the market, most software systems nowadays grow fast introducing new and complex functionalities [38]. While having more capabilities in a software system can bring important benefits, there is the risk that this fast-paced evolution leads to a degradation in the maintainability of the system [67], with potentially dangerous consequences [6].

Maintaining an evolving software structure becomes more complex over time [47]. Since time and manpower are typically limited, software projects must strategically manage their resources to deal with this increasing complexity. To assist this problem, researchers have been conducting several studies on how to advise and optimize the limited project resources. One broadly investigated idea, known as *bug prediction* [32], consists in determining non-trivial areas of systems subjected to a higher quantity of bugs, to assign them more resources.

Researchers have introduced and evaluated a variety of bug prediction models based on the evolution [34] (*e.g.*, number of changes), the anatomy [7] (*e.g.*, lines of code, complexity), and the socio-technicals aspects (*e.g.*, contribution organization) of software projects and artifacts [21]. These models have been evaluated individually or heterogeneously combining different projects [71], [84], [88].

Even though several proposed approaches achieved remarkable prediction performance [55], the practical relevance of bug prediction research has been largely criticized as not capable of addressing a real developer's need [75], [48], [46]. One of the criticisms regards the *granularity* at which bugs are found; in fact, most of the presented models predict bugs at a coarse-grained level, such as modules of files. This granularity is deemed not informative enough for practitioners, because files and modules can be arbitrarily large, thus requiring a significant amount of files to be examined [29]. In addition, considering that large classes tend to be more bug-prone [42], [63], the effort required to identify the defective part is even more substantial [7], [30], [60].

Giger *et al.* [29] and Hata *et al.* [35] presented the first work acting at a finer granularity: method-level. Giger *et al.* found that product and process metrics contribute to the identification of buggy methods andtheir combination achieves promising performance [29]. Hata *et al.* found that method-level bug prediction saves more effort than both file-level and package-level prediction [35].

In this paper, we replicate their investigation on bug prediction at method-level, focusing on the study by Giger *et al.* [29]. We use the same features and classifiers as the reference work, but on a different dataset to test the generalizability of their findings. Then we reflect on the evaluation strategy and propose a more realistic one. That is, instead of taking change history and predicted bugs from the same time frame and of using cross-validation, we estimate the performance using data from subsequent releases (as done by the most recent studies, but at a coarser granularity [70]).

Our results—computed on different systems/timeframes than the reference work—corroborate the generalizability of the performance of the proposed method-level models, when estimated using the previous evaluation strategy. However, when evaluated with a release-by-release strategy, all the estimated models present lower performance, close to that of a random classifier. As a consequence, even though we could replicate the reference work, we found that its realistic evaluation leads to negative results. This suggests that method-level bug prediction is still not a solved problem and the research community has the chance to devote more effort in devising more effective models that better assist software engineers in practice.

II. Background and Related Work

Bug prediction has been extensively studied by our research community in the last decade [32]. Researchers have investigated what makes source code more bug-prone (*e.g.*, [3], [4], [18], [8], [12], [42], [62], [63], [64], [73], [69]), and have proposed several unsupervised (*e.g.*, [20], [56], [87]) as well as supervised (*e.g.*, [11], [22], [39], [65], [89]) bug prediction techniques. More recently, researchers have started investigating the concept of *just-in-time* bug prediction, which has been proposed with the aim of providing developers with recommendations at commit-level (*e.g.*, [43], [40], [76], [27], [86], [51], [37]).

Our current paper focuses on investigating how well supervised approaches can identify bug-prone methods. For this reason, we first describe related work on predicting bug-prone classes, then we detail the earlier work on predicting bug-prone methods and how our work investigates its limitations and re-evaluates it.

A. Class-level Bug Prediction

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

Product metrics. Basili *et al.* [7] found that five of the CK metrics [15] can help determining buggy classes and that Coupling Between Objects (CBO) is that mostly related to bugs. These results were later re-confirmed [30], [41], [78].

Ohisson *et al.* [59] 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 [53] exploited two static analysis tools to predict the pre-release bug density for Windows Server, showing good performance. Nagappan *et al.* [54] experimented with code metrics for predicting buggy components across five Microsoft projects, finding that there is no single universally best metric. Zimmerman *et al.* [89] investigated complexity metrics for bug prediction reporting a positive correlation between code complexity and bugs. Finally, Nikora *et al.* [57] showed that measurements of a system's structural evolution (*e.g.*, 'number of executable statements') can serve as bug predictors.

Process metrics. Graves *et al.* [83] experimented both product and process metrics for bug prediction, finding that product metrics are poor predictors of bugs.

To further investigate the role played by product and process metrics, Moser *et al.* [52], [72] performed two comparative studies, which highlighted the superiority of process metrics in predicting buggy code components. Later on, D'Ambros *et al.* [19] performed an extensive comparison of bug prediction approaches relying on both the sources of information, finding that no technique works better in all contexts. A complementary approach is the use of developer-related factors for bug prediction. For example, Hassan investigated a technique based on the entropy of code changes by developers [34], reporting that it has better performance than models based on code components changes. Ostrand *et al.* [9], [61] 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. Finally, Di Nucci *et al.* [21] defined a bug prediction model based on a mixture of code, process, and developer-based metrics outperforming the performance of existing models.

Despite the aforementioned promising results, developers consider class/module level bug prediction too coarse-grained for practical usage [75]. Hence, the need for a more fine-grained prediction, such as *method-level*. This target adjustment does not negate the value of the preceding work but calls for a re-evaluation of the effectiveness of the proposed methods and, possibly, a work of adaptation.

B. Method-level Bug Prediction

So far, only Giger *et al.* [29] and Hata *et al.* [35] independently and almost contemporaneously targeted the prediction of bugs at method-level. Overall they defined a set of metrics (Hata *et al.* mostly process metrics, while Giger *et al.* also considered product metrics) and evaluated their bug prediction capabilities. Giger *et al.* found that both product and process metrics contribute to the identification of buggy methods and their combination achieves promising performance (*i.e.*, F-Measure=86%) [29]. Hata *et al.* found that using method-level bug prediction one saves more effort (measured in number of LOC to be analyzed) than both file-level and package-level prediction [35]. The data collection approach used by both sets of researchers is very similar, here we detail that used by Giger *et al.* [29], as an exemplification.

To produce the dataset used in their evaluation, Giger *et al.* conducted the following steps [29]: they (1) took a large time frame in the history of 22 Java OSS systems, (2) considered 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 [44] to evaluate three models (only process metrics, only product metrics, and both combined), considering the presence/absence of bug(s) in a method as the dependent binary variable.

In the work presented in this paper, we replicate the same methodology of Giger *et al.* and Hata *et al.* on an overlapping sets of projects to see whether we are able to reach similar results for other contexts. For simplicity and because the methodological details are more extensive, we follow more closely the case of Giger *et al.* [29].

III. Research Goal and Subjects

This section defines the goal of our empirical study in terms of research questions and the subject systems we consider.

A. Research Questions

The first *goal* in our study is to replicate bug-prediction work at method level, by using the research method employed by Giger *et al.* [29] on a partially overlapping set of software systems in different moments in time, with the *purpose* of understanding the extent to which their results are actually generalizable. This leads to our first research question:

> **RQ1.** *How effective are existing method-level bug prediction approaches when tested on new systems/timespans?*

While replicating the methodology proposed by Giger *et al.* [29], we found some limitations with the validation approach that they followed to assess the effectiveness of the prediction methods. In fact, although reasonable for an initial validation, the type of validation followed by Giger *et al.* has the following limitations: (1) it uses 10-fold cross-validation, which is at the risk of producing biased estimates in certain circumstances [82], (2) product metrics are considered only at the end of the time frame (while bugs are found *within* the time frame), (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).

In the second part of our study we try to overcome the aforementioned limitations by re-evaluating the performance using data from subsequent releases (*i.e.*, a release-by-release validation). 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, but we hope still promising. This leads to our second research question:

> **RQ2.** *How effective are existing method-level bug prediction models when validated with a release-by-release validation strategy?*

B. Subject systems

The *context* of our work consists of the 13 software systems whose characteristics are reported in Table I. For each system, the table reports its size (in KLOCs), number of contributors, releases, methods, and number of buggy methods over the entire change history, and number of buggy methods contained in its last release. In particular, we focus on systems implemented in Java (*i.e.*, one of the most popular programming languages [23]), since the metrics previously used/defined by both Giger *et al.* [29] and Hata *et al.* [35] mainly target this programming language. In addition, we choose projects whose source code is publicly available (*i.e.*, open-source software projects) and are developed using GIT as version control system, in order to enable the extraction of product and process metrics. Hence, starting from the list of open-source projects available on GITHUB,[1] we randomly selected 13 systems that have a change history composed of at least $1,000$ commits and more than $5,000$ methods. Our dataset is numerically smaller than the one by Giger *et al.*, but comprises larger systems composed of a much larger number of both methods (1.8M vs 112,058) and bugs (63,400 vs 23,762); this allows us to test the effectiveness of method-level bug prediction on software systems of a different kind of size.

IV. RQ1 - Replicating method-level bug prediction

Our RQ1 aims at replicating the study conducted by Giger *et al.* [29] on a different set of systems/timespans.

A. RQ1 - 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.* [29] and (ii) evaluate its performance applying it to our projects. To this aim, we follow a set of methodological steps such as (i) creation of an oracle reporting buggy methods in each of the projects considered, *i.e.*, the dependent variable to predict (ii) definition of the independent variables, *i.e.*, the metrics on which the model relies on, (iii) testing of the performance of different machine learning algorithms, and (iv) definition of the validation methodology to test the performance of the model.

Extraction of Buggy 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.* [29]. Given the issues available in the issue tracking systems (*i.e.*, BUGZILLA or JIRA) of the subject systems, we firstly use RELINK [85] to identify links between issues and commits. RELINK considers several constraints, *i.e.*, (i) a match exists 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 note and the last comment referred above, computed using the Vector Space Model (VSM) [5], is greater than 0.7. Afterwards, we consider as buggy all the methods actually 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 filtered out test cases, which might be modified with the production code, but might not directly be implicated in a bug.

Independent variables. As for the metrics to characterize source code methods, we compute the set of 9 product and 15 process features defined by Giger *et al.* [29].

- *Product Metrics:* Existing literature demonstrated how such set of features might be effective to characterize the extent to which a source code method is difficult to

[1] https://github.com

Table I
OVERVIEW OF THE SUBJECT PROJECTS INVESTIGATED IN THIS STUDY

Projects	LOC	Developers	Releases	Methods	All Buggy Methods	Last Buggy Methods
Ant	213k	15	4	42k	2.3k	567
Checkstyle	235k	76	6	31k	4.1k	670
Cloudstack	1.16M	90	2	85k	13.4k	6.8K
Eclipse JDT	1.55M	22	33	810k	3.3k	96
Eclipse Platform	229k	19	3	7k	2.7k	932
Emf Compare	3.71M	14	2	9k	0.7k	444
Gradle	803k	106	4	73k	4.6k	1.1k
Guava	489k	104	17	262k	1.2k	71
Guice	19k	32	4	9k	0.5k	145
Hadoop	2.46M	93	5	179k	5.8k	1.3k
Lucene-solr	586k	59	7	213k	8.7k	962
Vaadin	7.06M	133	2	43k	11.3k	7.7K
Wicket	328k	19	2	30k	4.9k	2.2K
Overall	**19M**	**782**	**91**	**1.8M**	**63.4k**	**22.9k**

Table II
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

Table III
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

maintain, possibly indicating the presence of defects [7], [15], [19], [59]. Giger *et al.* [29] proposed the use of the metrics reported in Table II. The features cover different method characteristics, *e.g.*, number of parameters or McCabe's cyclomatic complexity [50]. We re-implement all of the metrics due to the lack of available tools.

- *Process Metrics:* According to previous literature [72], [79], process features effectively complement the capabilities of product predictors for bug prediction. For this reason, Giger *et al.* [29] relied on the change-based metrics described in Table III and that widely 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. Also in this case, we re-implement the proposed process metrics defined at method-level by Giger *et al.* [29].

Similarly to Giger *et al.* [29], in the context of RQ1, we build three different method-level bug prediction models relying on (i) *only* product metrics, (ii) *only* process metrics, and (ii) *both* product and process metrics.

Training Data Preprocessing. Once we have the dataset containing (i) product and process metrics (*i.e.*, the independent variables) and (ii) buggy methods (*i.e.*, the dependent variable), we start the method-level bug prediction process. As first step, we take into account two common problems that may affect machine learning algorithms, namely (i) data unbalance [14] and (ii) multi-collinearity [58].
The former represents a frequent issue in bug prediction occurring 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 instances. We address this problem by applying the RANDOM OVER-SAMPLING algorithm [13] implemented as a supervised filter in the WEKA toolkit.[2] 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.
The second problem comes from the use of multiple metrics. These independent variables may have a high correlation causing collinearity that negatively impacts the performance of bug prediction models [24]. To cope with this problem, we preprocess our dataset filtering out the unwanted features. Specifically, we apply the *Correlation-based Feature Selection* [31] algorithm implemented as a filter in the WEKA toolkit: It evaluates the worth of a subset of attributes by considering the individual predictive ability of each feature along with their degree of redundancy.

Machine Learner. Once preprocessed the training data, we need to select a classifier that leverages the independent

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

variables to predict buggy methods [26]. To this aim, we exploit the four classifiers used by Giger *et al.*, which are all available in WEKA toolkit: *Random Forest*, *Support Vector Machine*, *Bayesian Network*, and *J48*. Afterwards, we compare the different classification algorithms using validation strategy and metrics we describe later.

Evaluation Strategy. The final step to answer RQ1 consists of the validation of the prediction models. As done in the reference work, we adopt the 10-fold cross-validation strategy [44], [80]. This strategy randomly partitions the original set of data into 10 equal sized subset. Of the 10 subsets, one is retained as test set, while the remaining 9 are used as training set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the test set exactly once.

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.* [29] to allow for comparison: *precision* and *recall* [5]. Precision is defined as $precision = \frac{|TP|}{|TP+FP|}$ where TP (True Positives) are methods that are correctly retrieved by a prediction model and FP (False Positives) are methods that are wrongly classified by a prediction model. Recall is defined as $recall = \frac{|TP|}{|TP+FN|}$, where FN (False Negatives) are methods that are not retrieved by a prediction model (*i.e.*, buggy methods misclassified as non-buggy by a model). We also compute F-Measure [5], which combines precision and recall in a single metric: $F-Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$.
In addition to the aforementioned metrics, we also compute the *Area Under the Receiver Operation Characteristic* curve (AUC-ROC) [33]. 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 [82]; 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).

B. RQ1 - Results

Table IV 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 features when using different classifiers. A detailed report of the performance achieved by the single classifiers over all the considered systems is available in our online appendix [68]. Overall, the obtained results are in line with those by Giger *et al.*, yet we achieve values that are 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 39% of the recommendations it outputs. This result is in line with the findings provided by Giger *et al.*, who already showed that the model only trained on product metrics offers performance generally lower than all the other experimented models.

Secondly, our results confirm that process metrics are stronger indicator of bug-proneness of source code methods (overall F-Measure=0.80). Also in this case, this finding is in line with the previous results achieved by the research community that report the superiority of process metrics with respect to product ones [70], [72]. Our results also confirm another finding by Giger *et al.*: 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. We find this surprising, since both set of metrics have values in the prediction and we expected that the use of these orthogonal predictors would improve the overall performance of the approach.

As for the different classifiers experimented, *Support Vector Machines* gives the worst results; likely, this is due to the extreme sensibility of the classifier to the configuration [16]. In fact, as shown in previous research [16], [36], the use of the default configuration might lead to significantly worsen the overall performance of the machine learner. 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 in the context of bug prediction [22], [49].

To test the statistical significance of the results discussed so far, we compared the AUC-ROC values of the experimented models over the different systems using the Scott-Knott Effect Size Difference (ESD) test [81], which is effect-size aware variant of the Scott-Knott test [74] that is recommended in case of comparisons of multiple models over multiple datasets [81]. As a result, process-based models built using *Random Forest* and *J48* are considered statistically better than product-based ones, while they work similarly to the combined ones. Detailed statistical results are in our online appendix [68].

Table IV
MEDIAN CLASSIFICATION RESULTS OF METHOD-LEVEL BUG PREDICTION MODELS WHEN VALIDATED USING 10-FOLD CROSS VALIDATION.

π = Product	Precision			Recall			F-measure			AUC-ROC		
Π = Process	π	Π	π&Π	S	Π	π&Π	π	Π	π&Π	π	Π	π&Π
Bayesian Network	0.71	0.77	0.77	0.46	0.68	0.70	0.56	0.72	0.72	0.60	0.72	0.72
J48	0.73	0.82	0.84	0.60	0.84	0.83	0.65	0.83	0.83	0.60	0.79	0.80
Random Forest	0.72	0.85	**0.86**	0.64	**0.86**	0.86	0.68	0.85	**0.86**	0.66	0.84	**0.86**
Support Vector Machines	0.66	0.74	0.74	0.09	0.80	0.79	0.16	0.77	0.76	0.50	0.51	0.51
Overall	0.71	0.80	0.80	0.44	0.80	0.80	0.51	0.80	0.80	0.59	0.72	0.73

Result 1: Our results, computed with the same evaluation strategy but on a different set of systems/timespans, confirm the findings by Giger *et al.*: Method-level bug prediction models based on process metrics perform better than those based on product metrics. Our results are 10 percentage points lower than those of Giger *et al.*, yet far better than random. The combination of predictors of different nature does not dramatically improve the prediction capabilities.

V. REFLECTING ON THE EVALUATION STRATEGY

By replicating the work by Giger *et al.*, we had the chance to reflect 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).

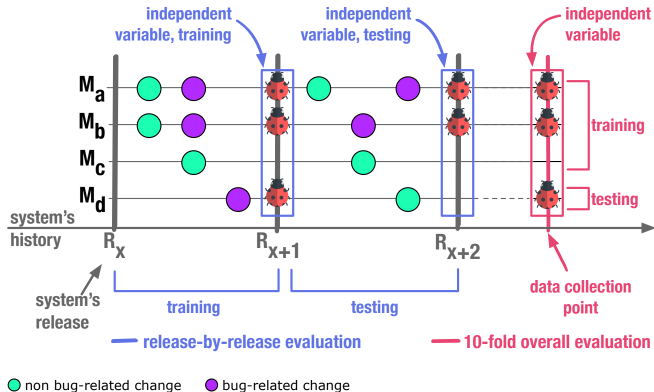

Figure 1. Training and testing strategies for method-level bug prediction.

The system in Figure 1 has four methods (*i.e.*, M_a, M_b, M_c, M_d) that where 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 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 already happened) and that (2) seem to be highly correlated to the independent variable (for each bug fix there has to be a 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 unrealistic results. For this reason, we propose a release-by-release strategy. We train and test "horizontally" instead of "vertically": We assume 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 independent variable is whether or not 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; yet, 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 answer RQ2.

An addition to the release-by-release strategy would be to consider the SZZ algorithm [77] 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 [17], 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.

VI. RQ2 - RE-EVALUATING METHOD-LEVEL BUG PREDICTION

Our RQ2 seeks to evaluate the performance of method-level bug prediction models in a more realistic setting.

A. RQ2 - Methodology

To answer RQ2, we need to (i) extract all the releases of the considered projects, (ii) identify the buggy methods occurring in each of them, and (iii) build the three bug prediction models considered in the context of RQ1.

Extraction of The Major Releases. The first step to test the performance of method-level bug prediction models consists in the identification of the major releases of the considered systems. To this purpose, we automatically extract them from the list of releases declared on the GITHUB repository of the subject systems. To discriminate major releases 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 a major release is identified. We manually verified the performance of this heuristic on one of the systems considered in the study: We verified that all the major releases of LUCENE-SOLR were correctly caught, thus quantifying the actual performance of this approach.

Extraction of Buggy Data. Differently from what we have done in RQ1, in this research question we need to extract the buggy data for all the considered releases. For each release pair $ri-1$ and r_i, 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 $ri-1$ and r_i. Also in this case, we filtered out test cases.

Bug Prediction Models: Setup. As done for RQ1, 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*. Also in this case, the training data is preprocessed to avoid (i) data unbalance and (ii) multicollinearity by using the same set of techniques previously exploited, *i.e.*, *Random Over-Sampling* algorithm [13] and *Correlation-based Feature Selection* [31], respectively.

Bug Prediction Models: Validation. As a final step to answer the second research question, 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_{i-1} and tested it on r_i. This technique implies that the first release of each system could not be used as testing set as well as the last release could not be used as training. To measure the accuracy of such models, we computed the same set of metrics previously exploited, *i.e.*, precision, recall, F-Measure, and AUC-ROC.

B. RQ2 - Results

Table IV 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. For sake of space limitation, we report the results aggregated using the median operator, however, detailed reports are available in our appendix [68].

The performance achieved by all the prediction models experimented is significantly lower than those found in the replication presented in RQ1. We observe a limited decrease between the highest/lowest values and overall in each of the subject systems in our dataset.

In this evaluation scenario, the use of code metrics as predictors only slightly *improves* the capabilities of method-level bug prediction models. This is in contrast with past literature reporting the superiority of process metrics for bug prediction [70], [72]. We hypothesize that this result may be caused both by the different granularity of the experimented models and by the different validation strategy with respect to the one used in RQ1. 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 [34], 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 confutes the observation of previous studies [70], [72] comes from the irregular distribution of the length of the time frames for the considered releases. In our analyzed projects, these intervals stretch between 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, we observe that the performance of different classifiers is similar and there is no clear winner. To some extent, this result confirms previous findings in the field [28], [66] showing that different classifiers achieve similar performance. 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 [22], [49].

Result 2: All the experimented method-level bug prediction models resulted in dramatically lower performance (up to 20 points percentage less in terms of AUC-ROC) when evaluated with the more realistic release-by-release evaluation strategy, instead of 10-fold cross validation. The achieved AUC-ROC scores achieved by all the models, regardless of the machine learning approach, are close to the results that a random classifier would provide.

VII. THREATS TO VALIDITY

In this section, we describe the factors that might have affected the validity of our empirical study.

Threats to Construct Validity. A first factor influencing the relationship between theory and observation is related to the dataset exploited. In our study, we rely on the same

Table V
MEDIAN CLASSIFICATION RESULTS OF METHOD-LEVEL BUG PREDICTION MODELS WHEN VALIDATED USING A RELEASE-BY-RELEASE STRATEGY.

S = Product H = Process	Precision			Recall			F-measure			AUC-ROC		
	S	H	S&H	S	H	S&H	S	H	S&H	S	H	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

methodology previously adopted by Giger *et al.* [29] to build our own repository of buggy methods, *i.e.*, we first retrieve bug-fixing commits using the textual-based technique proposed by Fisher *et al.* [25] and then consider as buggy the methods changed in that commits. To understand possible imprecisions and/or incompleteness of the data used in this study, we manually validate a statistically significant sample of 275 buggy methods detected on the LUCENE-SOLR system. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 962 total buggy methods detected in the last release of the project. The validation was conducted by the first two authors of this paper. Based on (i) the description of the bug reported on the issue tracker system, (ii) the source code of a method detected as buggy in a commit c_i, and (iii) the list of modifications to the method between c_i and its predecessor c_{i-1} (extracted using the `diff` unix command), each author checked *independently* whether the changes applied between the two revisions might have actually introduced the bug reported on the issue tracker. After the first round, the two inspectors started a discussion on the independent classifications made to reach consensus. The level of agreement between the inspectors is computed using the Krippendor's α [45], finding it to be 0.84, which is higher than the 0.80 used as standard reference score [2]. As for the accuracy of the linking methodology, we found that it correctly captured the bugginess of 85% of methods. Thus, at the end of this process we can claim that the oracle built is accurate enough for our purposes.

A threat to the validity of our replication is that we had to re-implement the product and process metrics used to build the experimented models, due to the lack of a publicly available tool. When re-implementing such metrics we faithfully followed the descriptions by Giger *et al.* [29].

Although we test the performance of the models using the same machine learning classifiers used by Giger *et al.* [29] to closely replicate their study, the use of different classifiers may produce different results. Moreover, all the tested classifiers use the default parameters, since finding the best configurations would have been too expensive [10].

Threats to Conclusion Validity. To ensure that the results would not have been biased by confounding effects such as data unbalance [14] or multi-collinearity [24], we adopt formal procedures aimed at (i) over-sampling the training sets [14] and (ii) removing non-relevant independent variables through feature selection [31].

Threats to External Validity. This category refers to the generalizability of our findings. While in the context of this work we analyze software projects having different size and scope, we limit our focus to Java systems because some of the tools exploited to compute the independent and dependent variables mainly target this programming language. Thus, the generalizability with respect to systems written in different languages as well as to projects belonging to industrial environments is limited.

VIII. CONCLUSION

We replicated previous work [29] on method-level bug prediction. We first re-implement the models and evaluate them with the same strategy applied by the reference work, yet on different systems/timespans to test its generalizability, finding aligned results. However, a deep analysis of the evaluation strategy revealed some of its limitations, which we address proposing a to use a release-by-release evaluation strategy. The method-level bug prediction models, when tested with the latter strategy, achieve far lower performance, similar to thatof a random classifier. As such, current strategies for method-level bug prediction do not seem to be sophisticated enough to reach their goal.

The main contributions made by this work are:

1) A re-evaluation on different systems/timespans of method-level bug prediction models. The results confirm previous findings in the field [29].
2) An empirical analysis of how the performance of existing method-level bug prediction models change when applied in a more realistic, release-by-release scenario. Our results provide evidence that current method-level bug prediction models are not able to dramatically outperform a random classifier; hence we reveal the need for further research on this area.
3) An online appendix [68] 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 larger set of systems along with a study aimed to measure the capabilities of ensemble methods [22], [49] and (ii) the investigation of novel features for improving method-level defect prediction.

ACKNOWLEDGMENT

Bacchelli and Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] M. Andreessen. Why software is eating the world. *The Wall Street Journal*, 20(2011):C2, 2011.

[2] J.-Y. Antoine, J. Villaneau, and A. Lefeuvre. Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation. In *EACL 2014*, pages 10–p, 2014.

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

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

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

[6] R. D. Banker, G. B. Davis, and S. A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, 44(4):433–450, 1998.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

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

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

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

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

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

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

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

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

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

[37] Q. Huang, X. Xia, and D. Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 159–170. IEEE, 2017.

[38] Z. Jiang and S. Sarkar. Free software offer and software diffusion: The monopolist case. *ICIS 2003 proceedings*, page 81, 2003.

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

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

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

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

[43] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

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

[45] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage, 2004.

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

[47] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.
[48] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE 2013, pages 372–381. IEEE Press, 2013.
[49] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.
[50] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
[51] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 2017.
[52] R. Moser, W. Pedrycz, and G. Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 309–311, New York, NY, USA, 2008. ACM.
[53] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.
[54] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.
[55] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, 2010.
[56] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction (NIER track). In *Proceedings of the 33rd international conference on software engineering*, pages 932–935. ACM, 2011.
[57] A. P. Nikora and J. C. Munson. Developing fault predictors for evolving software systems. In *Proceedings of the 9th IEEE International Symposium on Software Metrics*, pages 338–349. IEEE CS Press, 2003.
[58] R. M. O'brien. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity*, 41(5):673–690, 2007.
[59] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switchess. *Software Engineering, IEEE Transactions on*, 22(12):886–894, 1996.
[60] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
[61] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 19:1–19:10, New York, NY, USA, 2010. ACM.
[62] G. J. Pai and J. B. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on software Engineering*, 33(10), 2007.
[63] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.
[64] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 2017.
[65] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 244–255. IEEE, 2016.
[66] A. Panichella, R. Oliveto, and A. De Lucia. 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*, pages 164–173. IEEE, 2014.
[67] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.
[68] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating method-level bug prediction - online appendix. http://www.mediafire.com/?u4w771qz2y8be, 2018.
[69] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 452–461. IEEE Press, 2013.
[70] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.
[71] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.
[72] W. P. Raimund Moser and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *International Conference on Software Engineering (ICSE)*, ICSE '08, pages 181–190, 2008.
[73] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.
[74] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30:507–512, 1974.
[75] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.
[76] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.
[77] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.
[78] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, 2003.
[79] A. N. Taghi M. Khoshgoftaar, Nishith Goel and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Software Reliability Engineering*, pages 364–371. IEEE, 1996.
[80] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press, 2015.
[81] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 321–332. IEEE, 2016.
[82] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.
[83] J. S. M. Todd L. Graves, Alan F. Karr and H. P. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.
[84] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.
[85] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. 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*, pages 15–25. ACM, 2011.
[86] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.
[87] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.
[88] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. 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*, pages 91–100. ACM, 2009.
[89] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.