PalArch's Journal of Archaeology of Egypt / Egyptology

"RE-ESTIMATING PROCESS-LEVEL MICROBE ESTIMATE"

Busaramoni Jayanth*, Dr.S. Venu Gopal**
PG Scholar*, Associate Professor**
Department Of CSE, Vardhaman College Of Engineering, Hyderabad.

Busaramoni Jayanth*, Dr.S. Venu Gopal, RE-ESTIMATING PROCESS-LEVEL MICROBE ESTIMATE - Palarch's Journal Of Archaeology Of Egypt/Egyptology 18(4). ISSN 1567-214x**

ABSTRACT

Bug prediction is aimed at supporting developers in the identification of code artifacts more likely to be defective. Researchers have proposed prediction models to identify bug prone methods and provided promising evidence that it is possible to operate at this level of granularity. Particularly, models based on a mixture of product and process metrics, used as independent variables, led to the best results. In this study, we first replicate previous research on methodlevel 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 exhibiting results close to that of a random classifier. Our replication and negative results indicate that method-level bug prediction is still an open challenge.

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 [69], with potentially dangerous consequences [6]. Maintaining an evolving software structure becomes more complex over time [48]. 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 [73], [87], [91]. Even though several proposed approaches achieved remarkable prediction performance [57], the practical relevance of bug prediction research has been largely criticized as not capable of addressing a real developer's need [77], [49], [47]. 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 or 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], [65], the effort required to identify the defective part is even more substantial [7], [30], [62]. Menzies et al. [53] and Tosun et al. [86] introduced the first investigations exploring a finer granularity: function-level. Successively, Giger et al. [29] and Hata et al. [35] delved into finer granularity investigating the method-level bug prediction. Giger et al. found that product and process metrics contribute to the identification of buggy methods and their combination achieves promising performance [29]. Hata et al. found that method-level bug prediction saves more effort than both filelevel and package-level prediction [35]. In this paper, we replicate the investigations 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 both 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 [72]). 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 methodlevel 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

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], [64], [65], [66], [75], [71]), and have proposed several unsupervised (e.g., [20], [58], [90]) as well as supervised (e.g., [11], [22], [39], [67], [92]) 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], [78], [27], [89], [52], [37]). Our current paper focuses on investigating how well supervised approaches can identify bug-prone methods. For this reason, we firstly describe related work on predicting bugprone 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], [80]. Ohisson et al. [61] 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 [55] exploited two static analysis tools to predict the pre-release bug density for Windows Server, showing good performance. Nagappan et al. [56] experimented with code metrics for predicting buggy components across five Microsoft projects, finding that there is no single universally best metric. Zimmerman et al. [92] investigated complexity metrics for bug prediction reporting a positive correlation between code complexity and bugs. Finally, Nikora et al. [59] 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. [85] experimented both product and process metrics for bug prediction, finding that product metrics are poor predictors of bugs. Khoshgoftaar et al. [81] assessed the role of debug churns (i.e., the number of lines of code changed to fix bugs) in an empirical study, showing that modules having a large number of debug churns are likely to be defective. To further investigate the role played by product and process metrics, Moser et al. [54], [74] 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], [63] 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 [77]. Hence, the need for a more finegrained 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., FMeasure=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].

BUG DATA

Bug data of software projects is managed and stored in bug tracking systems, such as Bugzilla. Unfortunately, many bug tracking systems are not inherently linked to VCSs. However, developers fixing a bug often manually enter a reference to that particular bug in the commit message of the corresponding revision, e.g.,”fixed bug1234” or ”bug#345”. Researchers developed pattern matching techniques to detect those references accurately [43], and thus to link source code files with bugs. We adapted the pattern matching approach to work at method-level: Whenever we find that a method was changed between two revisions of a file and the commit message contains a bug reference, we consider the method to be affected by the bug. Based on this, we then count the number of bugs per method over the given timeframes in consistently enter and track bugs within the commit messages of the VCS. Furthermore, we rely on the fact that developers commit regularly when carrying out corrective maintenance, i.e., they only change those methods (between two revisions) related to that particular bug report being referenced in the commit message. We discuss issues regarding the data collection, in particular regarding the bug-linking approach, that might threaten the validity

PREDICTION EXPERIMENTS

We conducted a set of prediction experiments using the dataset presented to investigate the feasibility of building prediction models on method-level. We first describe the experimental setup and then report and discuss the results.

Experimental Setup

Prior to model building and classification we labeled each method in our dataset either as bug-prone or not bug-prone as follows:

$$bugClass = \begin{cases} not\ bug-prone & : \ \#bugs = 0 \\ bug-prone & : \ \#bugs >= 1 \end{cases}$$

These two classes represent the binary target classes for training and validating the prediction models. Using 0 (respectively

1) as cut-point is a common approach applied in many studies covering bug prediction models, e.g., [30, 48, 47, 4, 27, 37]. Other cut-points are applied in literature, for instance, a statistical lower confidence bound [33] or the median [16]. Those varying cut-points as well as the diverse
datasets result in different prior probabilities. For instance, in our dataset approximately one third of all methods were labeled as bug-prone; Moser et al. report on prior probabilitiesof 23%–32% with respect to bug-prone files; in [27] 0.4%–49% of all modules contain bugs; and in [48] 50% of all Java packages are bug free. Given this (and the fact that prior probabilities are not consistently reported in literature), the use of precision and recall as classification performance measures across different studies is difficult. Following the advice proposed in [26, 27] we use the area under the receiver operating characteristic curve (AUC) to asses and discuss the performance of our prediction models. AUC is a robust measure since it is independent of prior probabilities [4]. Moreover, AUC has a clear statistical interpretation [26]:
When selecting randomly a bug-prone and a not bug-prone method, AUC represents the probability that a given classifier assigns a higher rank to the bug-prone method. We also report on precision (P) and recall (R) in our experiments to allow for comparison with existing work.
In [26], Lessmann et al. compared the performance of several classification algorithms. They found out that more advanced algorithms, such as Random Forest and Support Vector Machine, perform better. However, the performance differences should not be overestimated, i.e., they are not significant. We observed similar findings in a previous study using fine-grained source code changes to build prediction models on file-level [16]. Menzies et al. successfully used Bayesian classifiers for bug prediction [27]. To contribute to that discussion (on method-level) we chose four different classifiers: Random Forest (RndFor), Bayesian Network (BN), Support Vector Machine (SVM), and the J48 decision tree. The Rapidminer Toolkit [29] was used for running all classification experiments. We built three different models for each classifier: The
first model uses change metrics as predictors, the second uses source code metrics and the third uses both metric sets (CM&SCM) as predictor variables. All our prediction models were trained and validated using 10-fold cross validation (based on stratified sampling ensuring that the class distribution in the subsets is the same as in the whole dataset).

APPLICATION OF RESULTS

The results of our study showed that we can build bug prediction models at the method level with good classification performance by leveraging the change information provided by fine-grained source code changes. In the following we demonstrate the application and benefit of our prediction model to identify the bug-prone methods in a source file compared to a file-level prediction model that performs equally well. For that, we assume a scenario as follows: A software developer of the JDT Core plugin, the largest Eclipse project, and the Derby Engine module, the largest non-Eclipse project in our dataset, receives the task to improve the unit testing in their software application in order to prevent future post-release bugs. For this, she needs to know the most bug-prone methods because they should be tested first and more rigorously than the other methods. For illustration purpose,

we assume the developer has little knowledge about her project (e.g., she is new to the project). To identify the bug-prone methods, she uses two prediction models, one model to predict the bug-prone source files and our Random Forest (RndFor) model to directly predict the bug-prone methods of a given source file. Furthermore, we take as examples release 3.0 of the JDT Core plugin and release 10.2.2.0 of the Derby Engine module. For both releases, she uses the two prediction models trained on the source code metrics and the versioning system history back to the last major release (i.e., 2.1 in case of JDT Core and 10.2.1.6 in case of Derby) for calculating the change metrics. Furthermore, both the models were trained using 1 bug as binning cut-point and 10- fold cross validation and then reapplied to the dataset. To better quantify the advantage of our method level prediction model over the file-level prediction model, we assume that the file-level prediction model performs equally well in terms of AUC, precision, and recall.

CONCLUSIONS AND FUTUREWORK

We empirically investigated if bug prediction models at the method level can be successfully created. We used the source code and change history of 21 Java open-source (sub-)systems. Our experiments showed that:

• Change metrics (extracted from the version control system of a project) can be used to train prediction models with good performance. For example, a Random Forest model achieved an AUC of 0.95, precision of 0.84, and a recall of 0.88 (RQ1).

• Using change metrics as predictor variables produced prediction models with significantly better results compared to source code metrics. However, including both metrics sets did not improve the classification performance of our models (RQ2).

• Different binning values did not affect the AUC values of our models (RQ3). Moreover, with a precision of 0.68 our models identify the ”top 5%” of all bug-prone methods better than chance.

• Conforming prior work, e.g., [26], we could not observe a significant difference among several machine learning techniques with respect to their classification performance. Given their good performance, our method-level prediction models can save manual inspection steps. Currently, we use the entire development history available at the time of data collection to train prediction models. It is part of our future work to measure changes based on different timeframes, e.g., release, quarterly, or yearly based. Furthermore, we plan to investigate a broader feature space, i.e., additional attributes, more advanced attribute selection techniques (rather than ”feeding all data” to the data mining algorithms), e.g., Information Gain [27], for prediction model building.

REFERENCES

[1] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Gu´eh´eneuc. Is it a bug or an enhancement? A text-based approach to classify change requests. In Proc. Conf. of the center for advanced studies on collaborative research: meeting of minds, pages 304–318, 2008.

[2] E. Arisholm and L. Briand. Predicting fault-prone components in a java legacy system. In Proc. Int’l Symp. on Empir. Softw. Eng., pages 8–17, 2006.

[3] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics

as quality indicators. IEEE Trans. Softw. Eng., 22:751–761, October 1996.
[4] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In Proc. Int'l Workshop on Principles of Softw. Evolution, pages 11–18, 2007.
[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In Proc. Joint Eur. Softw. Eng. Conf. and Symp. on the Found. of Softw. Eng., pages 121–130, 2009.
[6] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In Proc. Int'l Conf. on Softw. Eng., pages 518–528, 2009.
[7] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In Proc. Joint Eur Softw. Eng. Conf. and Symp. on the Found. of Softw. Eng., pages 4–14, 2011.
[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Trans. Softw. Eng., 20(6):476–493, June 1994.
[9] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng., pages 1–47, 2011.
[10] G. Denaro and M. Pezz`e. An empirical evaluation of fault-proneness models. In Proc. Int'l Conf. on Softw. Eng., pages 241–251, 2002.
[11] K. E. Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans. on Softw. Eng., 27(7):630–650, July 2001.
[12] B. Fluri, M. W¨ursch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Trans. On Softw. Eng., 33(11):725–743, November 2007.
[13] B. Fluri, J. Zuberbuehler, and H. C. Gall. Recommending method invocation context changes. In Proc. Int'l Workshop on Recomm. Syst. for Softw. Eng., pages 1–5, 2008.
[14] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. IEEE Software, 26(1):26–33, January/February 2009.
[15] G. Ghezzi and H. Gall. Sofas: A lightweight architecture for software analysis as a service. In Proc. Working Conf. on Softw. Architecture, pages 93–102, 2011.