Against the Mainstream in Bug Prediction

Haidar Osman

Software Composition Group
University of Bern, Switzerland
osman@inf.unibe.ch

Abstract

Bug prediction is a technique used to estimate the most bug-prone entities in software systems. Bug prediction approaches vary in many design options, such as dependent variables, independent variables, and machine learning models. Choosing the right combination of design options to build an effective bug predictor is hard. Previous studies do not consider this complexity and draw conclusions based on fewer-than-necessary experiments.

We argue that each software project is unique from the perspective of its development process. Consequently, metrics and AI models perform differently on different projects, in the context of bug prediction.

We confirm our hypothesis empirically by running different bug predictors on different systems. We show that no single bug prediction configuration works globally on all projects and, thus, previous bug prediction findings cannot generalize.

1 Introduction

A bug predictor is an intelligent system (model) trained on data derived from software (metrics) to make a prediction (number of bugs, bug proneness, *etc.*) about software entities (packages, classes, files, methods, *etc.*). Over the last two decades, bug prediction has been a hot topic for research in software engineering and many approaches have been devised to build effective bug predictors. Among the scientific findings, two are agreed upon the most: (i) different machine learning models do not differ in predicting bugs [6][11][4][13][5], and (ii) change metrics are better than source code metrics at predicting bugs [15][9][14][17][1][8][5].

However, these studies do not consider the complexity of building a bug predictor, a process that has many design options to choose from:

1. The prediction model (neural network, statistical regression, *etc.*).
2. The independent variables (the metrics used to train the model like source code metrics, change metrics, *etc.*).
3. The dependent variable or the model output (bug proneness, number of bugs, bug density, *etc.*).
4. The granularity of prediction (package, class, binary, *etc.*).
5. The evaluation method (accuracy measures, percentage of bugs in percentage of software entities, *etc.*).

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-Jul-2016, published at http://ceur-ws.org

Most previous approaches vary one design option, which is the studied one, and fix all others. This affects the generalizability of the findings because every option affects the others and, consequently, the overall outcome, as shown in Figure 1.

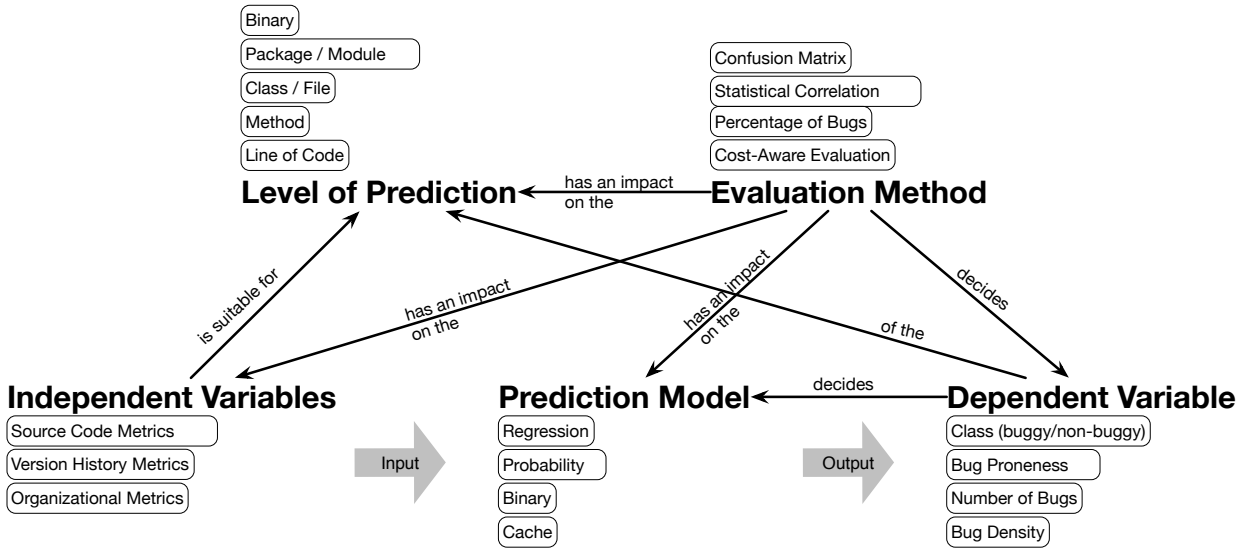

Figure 1: The design aspects of Bug prediction. The diagram shows the effects aspects have on each other.

We hypothesise that bug prediction findings are inherently non-generalizable. A bug prediction configuration that works with one system may not work with another because software systems have different teams, development methods, frameworks, and architectures. All these factors affect the correlation between different metrics and software defects.

To confirm our hypothesis, we run an extended empirical study where we try different bug prediction configurations on different systems. We show that no single configuration generalizes to all our subject systems and every system has its own "best" bug prediction configuration.

2 Experimental Setup

Dataset

We run the experiments on the "bug prediction data set" provided by D'Ambros *et al.* [3] to serve as a benchmark for bug prediction studies. This data set contains software metrics on the class level for five software systems (Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Lucene, and Mylyn). Using this data set constrains the level of prediction to be on the *class level*. We compare source code metrics and version history metrics (change metrics) as the independent variables.

Dependent Variable

All bug-prediction approaches predict one of the following: (1) the classification of the software entity (buggy or bug-free), (2) the number of bugs in the software entity, (3) the probability of a software entity to contain bugs (bug proneness), (4) the bug-density of a software entity (bugs per LOC), or (5) the set of software entities that will contain bugs in the near future (*e.g.*, within a month). In this study, we consider three dependent variables: *number of bugs*, *bug proneness*, and *classification*.

Evaluation Method

Recently, researchers have drawn the attention to the principle of cost or effort of using a bug prediction model [12][1][8][10] [7][16][2]. The cost-aware evaluation schemes measure the maximum percentage of faults found in the top k% of lines of code (instead of entities) of a system, taking the lines of code (LOC) as a proxy for the effort of unit testing and code reviewing. In this study, we use an evaluation scheme called *cost-effectiveness (CE)*, proposed by Arisholm *et al.* [1]. *CE* ranges between -1 and $+1$. The closer *CE* gets to $+1$, the more

cost-effective the bug predictor is. A value of *CE* around zero indicates that there is no gain in using the bug predictor. Once *CE* goes below zero, it means that using the bug predictor costs more than not using it.

Machine Learning Model

For classification, we use *RandomForest (RF)*, *K-Nearest Neighbour (KNN)*, *Support Vector Machine (SVM)*, and *Neural Networks (NN)*. To predict bug proneness, we use *RF*, *KNN*, and *J48*. To predict the number of bugs, we use *linear regression (LR)*, *SVM*, and *NN*.

Procedure

For every configuration, we randomly split the data set into a training set (75%) and a test set (25%) in a way that retains the ratio between buggy and non-buggy entities. Then we train the prediction model on the training set and run it on the test set and calculate the *CE* of the bug predictor. For each configuration, we repeat this process 10 times and take the mean *CE*.

3 Results

We compare the mean *CE* of the different configuration of the different bug predictors. A preliminary analysis of the results shows that there is no global configuration of settings that suits all projects. In Table 1, we report the highest mean *CE* and the configuration of the bug predictor behind. We can make the following observations:

1. Every system has a unique configuration of its most cost-effective bug predictor.

2. There is no dominant configuration value for metrics, model, or output variable.

3. The cost effectiveness of bug prediction is different from one system to another.

We acknowledge the fact that a more rigorous analysis of the results is needed to have better confidence in the generalizability of the findings. However, the results in Table 1 are enough to show that (i) different machine learning models actually perform differently in predicting bugs, and (ii) there is no general rule about which metrics are better at predicting bugs.

Table 1: The most cost-effective bug prediction configuration for each system and the corresponding mean *CE*.

Subject System	Independent Variables (Metrics)	Prediction Model	Dependent Variable (Output)	Mean CE
Eclipse JDT Core	Change Metrics	Support Vector Machine	Number of Bugs	0.351
Eclipse PDE UI	Source Code Metrics	Neural Network	Classification	0.237
Equinox	Source Code Metrics	Support Vector Machine	Number of Bugs	0.498
Lucene	Change Metrics	Random Forest	Bug Proneness	0.6
Mylyn	Change Metrics	Linear Regression	Number of Bugs	0.434

4 Conclusions

Building a software bug predictor is a complex process with many interleaving design choices. In the bug prediction literature, researchers have overlooked this complexity, suggesting generalizability where none is warranted. We argue that bug prediction studies cannot be generalized because software systems are different. Among the five subject systems we have, no two have the same configuration for building a cost-effective bug predictor. This indicates a need for more research to revisit literature findings while taking bug prediction complexity into account.

References

[1] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, Jan. 2010.

[2] 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, Mar. 2013.

[3] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31–40. IEEE CS Press, 2010.

[4] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[5] 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.

[6] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417–428. IEEE, 2004.

[7] 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*, ICSE '12, pages 200–210, Piscataway, NJ, USA, 2012. IEEE Press.

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

[9] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[10] K. Kobayashi, A. Matsuo, K. Inoue, Y. Hayase, M. Kamimura, and T. Yoshino. ImpactScale: Quantifying change impact to predict faults in large software systems. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[11] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[12] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[13] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, Dec. 2010.

[14] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

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

[16] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *In the 20th ACM SIGSOFT FSE*. ACM, 2012.

[17] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Engg.*, 13(5):539–559, Oct. 2008.