On the Non-Generalizability 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 machine learning 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 there are no universal bug prediction configurations that work on all projects.

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.*).

Bug prediction helps developers focus their quality assurance efforts on the parts of the system that are more likely to contain bugs. Bug prediction takes advantage of the fact that bugs are not evenly distributed across the system but they rather tend to cluster [19]. The distribution of bugs follows the Pareto principle, *i.e.*, 80% of the bugs are located in 20% of the files [17]. An effective bug predictor locates the highest number of bugs in the least amount of code.

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. Researchers have been probing the problem/solution space trying to find universal solutions regarding the software metrics to use [16][10][15][20][1][9][5] and machine learning models to employ [6][12][4][14][5] to predict bugs in software entities.

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

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-13 July 2016, published at http://ceur-ws.org

1. The prediction model (*e.g.*, Neural Networks, Support Vector Machines, Random Forest, Linear Regression).
2. The independent variables (*i.e.*, the metrics used to train the model like source code metrics, change metrics, *etc.*).
3. The dependent variable or the model output (*e.g.*, bug proneness, number of bugs, bug density).
4. The granularity of prediction (*e.g.*, package, class, binary).
5. The evaluation method (*e.g.*, accuracy measures, percentage of bugs in percentage of software entities).

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.

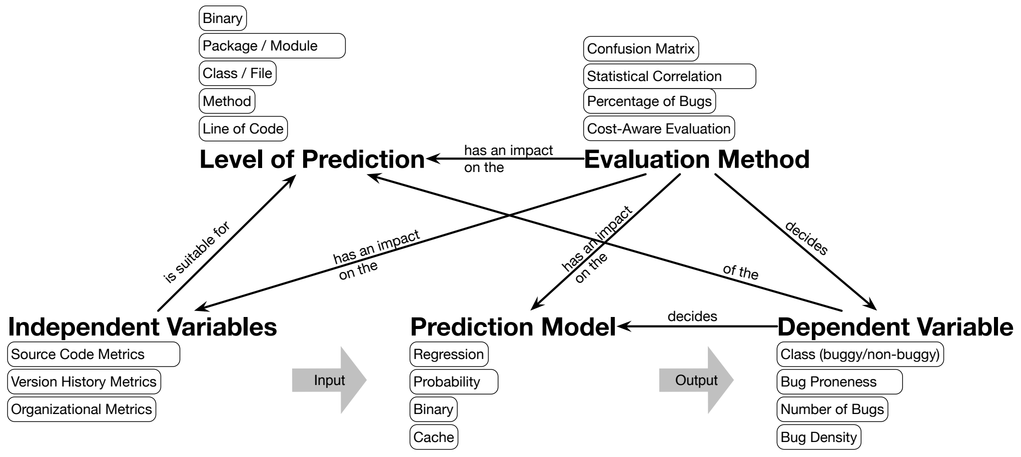

Figure 1: The design aspects of Bug prediction. The diagram shows the effects aspects have on each other.

It has been shown previously that a model trained on data from a specific project does not perform well on another project and the so called cross-project defect prediction rarely works [21]. We take this idea even further and 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 two dependent variables: *number of bugs* and *classification*.

Evaluation Method

An effective bug predictor should locate the highest number of bugs in the least amount of code. Recently, researchers have drawn attention to this principle and proposed evaluation schemes to measure the cost or effort of using a bug prediction model [13][1][9][11][8][18][2]. Cost-aware evaluation schemes rely on the fact that a bug predictor should produce an ordered list of software entities, and measure the maximum percentage of predicted faults in the top k% of lines of code of a system. These schemes take the number of 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 *Random Forest (RF)*, *K-Nearest Neighbour (KNN)*, *Support Vector Machine (SVM)*, and *Neural Networks (NN)*. To predict the number of bugs (regression), we use *linear regression (LR)*, *SVM*, *KNN*, and *NN*. We used the Weka data mining tool [7] to build these prediction models[1].

Procedure

For every configuration, we randomly split the data set into a training set (70%) and a test set (30%) 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 30 times and take the mean *CE*.

3 Results

First, we compare the different machine learning models. To see if machine learning models perform differently, we apply the analysis of variance, ANOVA, and the post-hoc analysis, Tukey's HSD (honest significant difference), to the different models in classification and to different models in regression.

The tests were carried out at 0.95 confidence level. Only when the ANOVA test is statistically significant, do we carry out the post-hoc test. Otherwise, we only report the best performing model. Statistically significant results are reported in boldface in the result Tables 1, 2, and 3. As can be seen from the results in Table 1, It is clear that different machine learning models actually perform differently. Also there is no dominant model that stands out as the best model throughout the experiments.

Second, to compare the two different types of metrics, we compare the best-performing model using source code metrics and the best-performing model using change metrics using the student's t-test at the 95% confidence level. We compare using both classification and regression. Table 2 shows the results of the test, where bold text indicates statistically significant results. It can be deduced from the results that source code metrics are better than change metrics in some projects and worse in others. No type of metrics is constantly the best for all projects in the dataset.

[1]We use Weka's default configuration values for the models

Table 1: This table shows the results of the analysis of variance, ANOVA, and the post-hoc analysis, Tukey's HSD (honest significant difference). Bold text indicates statistically significant results at 95% confidence level.

Classification

Metrics	JDT	PDE	Equinox	Mylyn	Lucene
Version History	$\boldsymbol{NN > KNN}$	$\boldsymbol{SVM > KNN}$		$\boldsymbol{SVM > KNN}$	$\boldsymbol{SVM > NN}$
	$\boldsymbol{NN > RF}$	$SVM \geq RF$	SVM	$\boldsymbol{SVM > RF}$	$SVM \geq RF$
	$NN \geq SVM$	$SVM \geq NN$		$SVM \geq NN$	$SVM \geq KNN$
Source Code	$\boldsymbol{KNN > SVN}$	$\boldsymbol{KNN > SVN}$		$\boldsymbol{NN > SVM}$	$\boldsymbol{RF > NN}$
	$\boldsymbol{KNN > NN}$	$KNN \geq NN$	KNN	$NN \geq KNN$	$RF \geq SVM$
	$\boldsymbol{KNN > RF}$	$KNN \geq RF$		$NN \geq RF$	$RF \geq KNN$

Regression

Metrics	JDT	PDE	Equinox	Mylyn	Lucene
Version History	$\boldsymbol{SVM > NN}$	$\boldsymbol{LR > NN}$	$\boldsymbol{LR > KNN}$	$\boldsymbol{SVM > KNN}$	
	$\boldsymbol{SVM > KNN}$	$\boldsymbol{LR > KNN}$	$LR \geq SVM$	$\boldsymbol{SVM > NN}$	LR
	$SVM \geq LR$	$LR \geq SVM$	$LR \geq NN$	$SVM \geq LR$	
Source Code	$\boldsymbol{KNN > SVN}$	$\boldsymbol{KNN > SVN}$	$\boldsymbol{SVM > KNN}$	$\boldsymbol{SVM > KNN}$	$\boldsymbol{KNN > NN}$
	$\boldsymbol{KNN > NN}$	$\boldsymbol{KNN > NN}$	$\boldsymbol{SVM > NN}$	$\boldsymbol{SVM > NN}$	$KNN \geq SVN$
	$\boldsymbol{KNN > LR}$	$KNN \geq LR$	$SVM \geq LR$	$SVM \geq LR$	$KNN \geq LR$

Table 2: This tables shows the t-student test between the best performing model trained on source code metrics (SM) and the best performing model trained on change metrics (CM). Bold text indicates the statistically significant results at 95% confidence level.

	JDT	PDE	Equinox	Mylyn	Lucene
Classification	$\boldsymbol{CM < SM}$	$\boldsymbol{CM < SM}$	$\boldsymbol{CM > SM}$	$\boldsymbol{CM > SM}$	$\boldsymbol{CM > SM}$
Regression	$CM \geq SM$	$CM \leq SM$	$CM \leq SM$	$\boldsymbol{CM > SM}$	$\boldsymbol{CM > SM}$

Table 3: This tables shows the student's t-test between the classifier (CLA) and the best regressor (REG). The test is carried out at the 0.95 confidence level. Bold text indicates the statistically significant results at 95% confidence level.

JDT	PDE	Equinox	Mylyn	Lucene
$REG \geq CLA$	$REG \geq CLA$	$\boldsymbol{REG > CLA}$	$REG \geq CLA$	$REG \geq CLA$

Table 4: 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	SVM	Number of Bugs	0.356
Eclipse PDE UI	Source Code Metrics	KNN	Number of Bugs	0.246
Equinox	Source Code Metrics	SVM	Number of Bugs	0.429
Mylyn	Change Metrics	SVM	Number of Bugs	0.484
Lucene	Change Metrics	LR	Number of Bugs	0.588

Third, we compare the the two types of response variables (classification vs regression) by comparing the best performing model from each using also the student's t-test at the 95% confidence level. Table 3 shows that the comparisons are in favour of regression all projects in our dataset but with statistical significance only in case of Equinox. This means that treating bug prediction as a regression problem is more cost effective than classification.

Finally, we compare configurations with the highest *CE* for the five projects in the data set. In Table 4, we report the highest mean *CE* and the configuration of the bug predictor behind. The results show that there is no global configuration of settings that suits all projects.

To summarize the results of the experiments, we make the following observations:

1. Different machine learning models actually perform differently in predicting bugs and there is no dominant model that stands out as the best for all projects.

2. There is no general rule about which metrics are better at predicting bugs.

3. The configurations of the most cost-effective bug predictor vary from one project to another.

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

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. Our results suggest that a universal set of bug prediction configurations is unlikely to exist. Among the five subject systems we have, no type of metrics stands out as the best and no machine learning algorithm prevails when building 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. In the future we plan to explore different ways to automatically find the most effective bug prediction configurations for a specific project. This enables a bug predictor to be adaptive to the different characteristics of different software projects without manual intervention.

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] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

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

[19] S. A. Sherer. Software fault prediction. *Journal of Systems and Software*, 29(2):97–105, 1995.

[20] 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.

[21] 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*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.