Hyperparameter Optimization to Improve Bug Prediction Accuracy

Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{*osman, ghafari, oscar*}@*inf.unibe.ch*

***Abstract*—Bug prediction is a technique that strives to identify where defects will appear in a software system. Bug prediction employs machine learning to predict defects in software entities based on software metrics. These machine learning models usually have adjustable parameters, called hyperparameters, that need to be tuned for the prediction problem at hand. However, most studies in the literature keep the model hyperparameters set to the default values provided by the used machine learning frameworks.**

In this paper we investigate whether optimizing the hyperparameters of a machine learning model improves its prediction power. We study two machine learning algorithms: k-nearest neighbours (IBK) and support vector machines (SVM). We carry out experiments on five open source Java systems. Our results show that (i) models differ in their sensitivity to their hyperparameters, (ii) tuning hyperparameters gives at least as accurate models for SVM and significantly more accurate models for IBK, and (iii) most of the default values are changed during the tuning phase. Based on these findings we recommend tuning hyperparameters as a necessary step before using a machine learning model in bug prediction.

I. Introduction

Bug prediction is a technique that helps software teams focus their quality assurance efforts on the parts that are more likely to contain bugs. Technically, a bug predictor is a machine learning model trained on software metrics to predict bugs in software entities. Building a bug predictor involves deciding which software metrics to use, what to predict, and what machine learning model to train.

Hyperparameters are the parameters that are set for a machine learning model and affect its learning, construction, and evaluation. These parameters need to be set before training the model. Example hyperparameters are the *complexity* parameter in support vector machines and the number of neurons in the hidden layer in a feed-forward neural network. Different machine learning problems have different characteristics and the hyperparameters need to be optimized accordingly.

In the past three decades, researchers have analyzed the performance of various machine learning models in bug prediction. However, more often than not, model hyperparameters were set to the default values of machine learning frameworks, which are not necessarily the optimal ones.

The 1st international Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017), Klagenfurt, Austria, co-located with SANER 2017

TABLE I
THE BUG PREDICTION DATASET DETAILS, AS REPORTED BY D'AMBROS *et al.* [1]

System	Release	#Classes	% Buggy
Eclipse JDT Core	3.4	997	$\approx 20\%$
Eclipse PDE UI	3.4.1	1,497	$\approx 14\%$
Equinox	3.4	324	$\approx 40\%$
Mylyn	3.41	1,862	$\approx 13\%$
Lucene	2.4.0	691	$\approx 9\%$

In this paper, we investigate the effect of the hyperparameter optimization[1] of a model on its prediction accuracy. We study two machine learning models: support vector machines (SVM), and an implementation of the k-nearest neighbours algorithm called IBK. Using a grid search algorithm, the search space of the hyperparameter values of each model is traversed and the optimal values are reported. We evaluate the prediction accuracy of each model before and after hyperparameter optimization on five open source Java systems, listed in Table I.

Our results reveal that tuning model hyperparameters has a statistically significant positive effect on the prediction accuracy of the models. The prediction accuracy is improved by up to 20% in IBK and by up to 10% in SVM. However, we notice that IBK is more sensitive to its hyperparameter values than SVM in our experiments. We also observe that most of the hyperparameter values are changed during the tuning phase, indicating that default values are suboptimal. Our findings suggest that researchers in bug prediction need to take hyperparameter optimization into account in their bug prediction pipelines, as it potentially improves the prediction accuracy of the machine learning models.

II. Empirical Study

A. The Dataset

D'Ambros *et al.* [1] provided the "bug prediction dataset"[2] for benchmarking bug predictors. It has been used by many bug prediction studies [2][3][4][5]. Actually, it is one of the

[1]Hyperparameter optimization is also known as model selection and hyperparameter tuning.
[2]http://bug.inf.usi.ch/

TABLE II
THE CK METRICS SUITE [6] AND OTHER OBJECT-ORIENTED METRICS INCLUDED AS THE SOURCE CODE METRICS IN THE BUG PREDICTION DATASET [1]

Metric Name	Description
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
FanIn	Number of classes that reference the class
FanOut	Number of classes referenced by the class
LCOM	Lack of Cohesion in Methods
NOC	Number Of Children
NOA	Number Of Attributes in the class
NOIA	Number Of Inherited Attributes in the class
LOC	Number of lines of code
NOM	Number Of Methods
NOIM	Number of Inherited Methods
NOPRA	Number Of PRivate Atributes
NOPRM	Number Of PRivate Methods
NOPA	Number Of Public Atributes
NOPM	Number Of Public Methods
RFC	Response For Class
WMC	Weighted Method Count

TABLE III
THE CHANGE METRICS PROPOSED BY MOSER *et al.* [7] INCLUDED IN THE BUG PREDICTION DATASET [1]

Metric Name	Description
REVISIONS	Number of reversions
BUGFIXES	Number of bug fixes
REFACTORINGS	Number Of Refactorings
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines ofcode deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code - deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVE_CODECHURN	Average CODECHURN for all revisions
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	Sum over age of a file in weeks times number of lines added during that week normalized by the total number of lines added to that file

few datasets that has the number of bugs as the response variable. This is the main reason we choose it for the empirical study, as our aim is to study the accuracy of predicting the number of bugs, rather than just the likely presence of bugs.

The "bug prediction dataset" has source code metrics (Table II) and change metrics (Table III) for the classes in five open-source Java systems (Table I). More details can be found in the original paper [1]. All 32 metrics are used as the features (*i.e.*, independent variables) and the number of bugs is used as the response, or dependent, variable.

B. *The Machine Learning Algorithms*

For this experiment, we pick two machine learning algorithms: k-nearest neighbours (IBK) and support vector machines (SVM). SVM is geometrically-inspired machine learning technique that can, in the feature space, separate data points by choosing the best separating hyperplane (*i.e.*, with highest separation margin that is). IBK is an implementation of the k-nearest neighbours algorithm where the value of the dependent variable of a new data point is calculated based on the average values of the dependent variable of the k-nearest data points in the feature space.

We choose these two machine learning algorithms because they operate differently, have many hyperparameters, and have two different track records in the field of bug prediction. While SVM has been extensively used and shown to be one of the best performing models [8][9][10][11][12], IBK has not been reported to excel in the bug prediction literature [13]. Studying these two models reveals whether the tuning process affects how they compare to each other. We use the WEKA[3] data mining framework [14] to train and test the models.

C. *Parameter Tuning*

For IBK, we tune three hyperparameters: the number of neighbours, the evaluation criterion, and the neighbour search algorithm, as detailed in Table IV. For SVM, we tune the complexity parameter and the used kernel and its parameters, as detailed in Table V and Table VI.

The search space for the optimal hyperparameter values is large and it is impractical to try every possible combination of values. We use *Multisearch-weka*[4] to search for the optimal hyperparameter values. It implements a hill-climbing grid search algorithm. An initial point in the search space is considered the center, then the algorithm performs 10-fold cross validation on the adjacent parameter values. The best one is considered as the new center. This process is repeated until no better values are found or the navigation hits the search space borders.

D. *Procedure*

For each project, we split the data into two sets using stratified sampling: tuning set (10%) and experimentation set (90%). The hyperparameters are tuned using the tuning set only. Then, for each machine learning model, we compare the

[3]http://www.cs.waikato.ac.nz/~ml/weka/

[4]https://github.com/fracpete/multisearch-weka-package

TABLE IV
IBK HYPERPARAMETERS THAT ARE TUNED.

Parameter	Default Value	Search Range
Number of neighbours	1	1 to 5 by step of $+1$
Evaluation Criterion	Mean Absolute Error	{Mean Absolute Error, Mean Squared Error}
Neighbour Search Algorithm	Linear Search	{Linear, BallTree, CoverTree, KDTree, Filtered Neighbour Search}

TABLE V
SVM HYPERPARAMETERS THAT ARE TUNED.

Parameter	Default Value	Search Range
Complexity	1	10^{-4} to 10^{2} by step of $\times 10$
Kernel	Polynomial	{Polynomial, RBF, PUK, Normalized Polynomial}

prediction error between the model with the default hyperparameter values and the model with the tuned values. For this comparison we use stratified 10-fold cross-validation on the experimentation set and the root mean squared error (RMSE) is calculated for each fold. This 10-fold cross-validation is repeated 30 times.

Up to this point, for each project/model, we have 300 RMSEs for the model with default hyperparameter values and 300 RMSEs for it with the optimal ones. We use paired student's *t*-test with 95% confidence interval to compare the two populations and determine whether the tuning process improves the prediction accuracy of the model.

E. Results

Figure 1 shows boxplots of the RMSEs of each model before and after tuning. Red frames indicate statistically significant results (the null hypothesis of student's *t*-test is rejected).

For IBK, tuning hyperparameters improves prediction accuracy significantly for all projects. The RMSE is reduced between 13% (in Equinox) and 22% (in Lucene). The results for SVM are not as significant as in IBK. Only in Equinox does hyperparameter tuning reduce RMSE significantly by 10%, while prediction accuracy of the tuned and untuned models are similar in the other four projects. This means that IBK is more sensitive to its hyperparameter settings than SVM in bug prediction.

We also compare IBK and SVM. A paired student's *t*-test with 95% significance interval shows that the RMSE of SVM is always statistically lower than the RMSE of IBK. However, after tuning, the two models are actually statistically equivalent for four projects and SVM is more accurate than IBK only in Equinox. This result shows that tuning not only potentially improves the prediction accuracy of a model, but also changes how different machine learning models compare to each other.

Finally, our results show that the values of model hyperparameters often change from the default after tuning. Table VII shows the parameter values after tuning. The tuning phase always changes the value of at least one hyperparameter from the default. Some hyperparameter values actually always change, such as the number of neighbours and the evaluation criterion in IBK. This leads to the conclusion that default values in machine learning frameworks are suboptimal for bug prediction.

In summary, we conclude the following:

1) Tuning hyperparameters significantly improves the prediction accuracy of IBK. Previously, IBK has not been in the top performing models for bug prediction [13], but with tuning it can be as performant as SVM.
2) IBK is more sensitive to hyperparameter tuning than SVM. We recommend that researchers experiment with the hyperparameters of the machine learning models they use before using them, as some require tuning while others are less susceptible to the hyperparameter values.
3) Many studies have been conducted to compare different machine learning models in the context of bug prediction. Some studies conclude that classifiers perform similarly in bug prediction [15][16][17][18][19] while others suggest that the choice of the classification model has a significant impact on the performance of bug prediction [20][8][21]. We show that hyperparameter tuning can have an effect on how machine learning models compare to each other. This raises doubts in the outcome of studies where no tuning of or at least experimentation with the hyperparameter values has been conducted.

F. Threats to Validity

The main threats to the validity of our study are threats to generalizability. First, the dataset we use contains only open-source Java projects. Replicating the same experiments on industrial projects or projects written in other languages may give different results.

We only experiment on two machine learning models only. Hyperparameter optimization might have different effects on other machine learning models. Also we carry out the tuning process on only on 10% of the dataset with bounded value ranges for the hyperparameters. A larger tuning set and wider value ranges might provide better hyperparameter values.

TABLE VI
SVM KERNEL HYPERPARAMETERS THAT ARE TUNED.

Kernel	Parameter	Default Value	Search Range
Polynomial	Exponent	1	1 to 10 by step of $+0.5$
Normalized Polynomial	Exponent	2	1 to 10 by step of $+0.5$
RBF	Gamma	0.01	$10^{-}4$ to 10 by step of $\times 10$
PUK	Omega	1	1 to 10 by step of $+0.5$
	Sigma	1	1 to 10 by step of $+0.5$

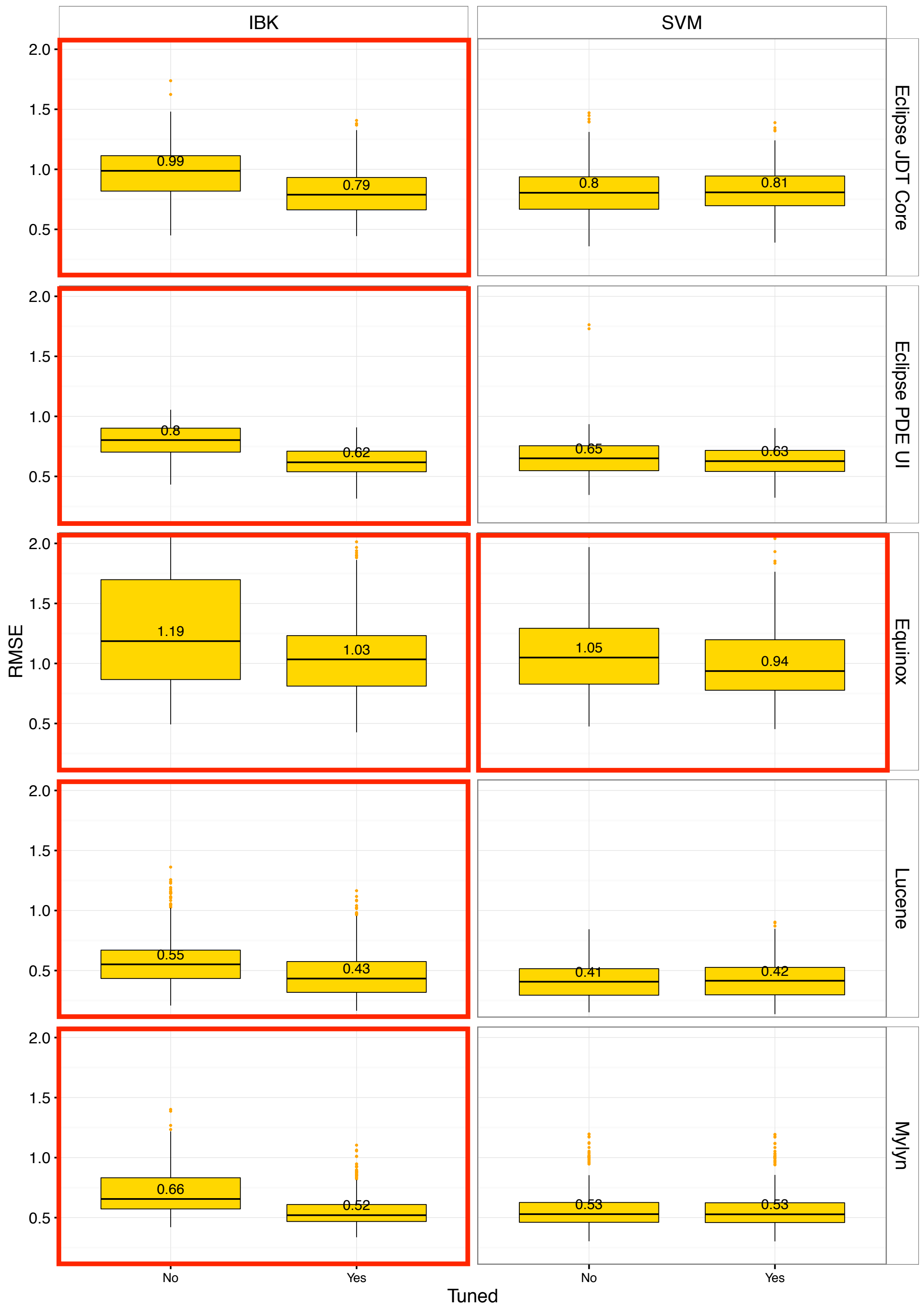

Fig. 1. Boxplots of all the experiments in our empirical study. The y-axis is the root mean squared error (RMSE). For each project, we compare each model before and after tuning. We carried out the student's *t*-test at 95% confidence interval. Red bold frames indicate the statistically significant results, where the tuning significantly reduced the RMSE of the model.

TABLE VII
THE TUNING RESULTS FOR THE HYPERPARAMETERS FOR ALL PROJECTS.

	Eclipse JDT Core	Eclipse PDE UI	Equinox	Lucene	Mylyn
IBK					
#Neighbours	5	5	2	5	5
Evaluation Criterion	Mean Squared Error	Mean Squared Error	Mean Squared Error	Mean Squared Error	Mean Squared Error
Search Algorithm	CoverTree	Linear Search	Linear Search	Linear Search	Linear Search
SVM					
Complexity	10	1	10	10	10
Kernel	PUK{Omega=1, Sigma=4.1}	Normalized Polynomial {Exponent=5}	RBF {Gamma=0.01}	RBF {Gamma=0.1}	Polynomial {Exponent=1}

III. Related Work

Model hyperparameters are often left set to their default values in the bug prediction literature. Very few studies inspect the effect of hyperparameter settings on the bug prediction accuracy.

Martino *et al.* [12][11] use a genetic search algorithm to optimize the hyperparameter settings for SVM and compare it with six machine learning models. Their experiments are carried out on jEdit data from the PROMISE repository [22]. They report that the genetically optimized SVM gives the highest *F-measure*. In their study, Martino *et al.* do not optimize the other models making the comparisons unfair.

The response variable of probabilistic models, such as Naiv̈e Beyes, is the probability that a software entity is defective. When these models are used as classifiers, they take the threshold that separates the classes as a hyperparameter. It is usually 0.5 by default. Tosun and Bener [23] report that optimizing this threshold decreases false alarms by up to 11%.

Recently, Tantithamthavorn *et al.* [24] carried out a large study on the effect of hyperparameter optimization on the accuracy of 26 classification techniques in bug prediction. They show that tuning model hyperparameters increases the accuracy measures by up to 40%.

All previous studies treat bug prediction as a classification problem where the response variable is the class of a software entity (*i.e.*, buggy or clean). To the best of our knowledge, this is the first study on hyperparameter optimization in bug prediction as a regression problem, where the response variable is the number of bugs.

IV. Conclusions and Future Work

Bug prediction is an active research domain in the field of software engineering. It employs machine learning to identify bugs in software entities and help developers focus their quality assurance efforts on the parts of the system that may contain bugs in the future. These machine learning models often have configurable hyperparameters that control how they behave during the training phase.

In this paper we study the effect of optimizing model hyperparameters to improve the accuracy of predicting the number of bugs. We show that the k-nearest neighbours algorithm (IBK) is always significantly improved and the prediction accuracy of support vector machines (SVM) is either improved or at least retained. We conclude that hyperparameter optimization should be conducted before using a machine learning model and default hyperparameter values are often suboptimal.

In the future, we plan to extend this study to cover more machine learning models in order to have a clearer picture about the sensitivity of each machine learning model to its hyperparameter settings in the context of bug prediction. Another future direction is to study hyperparameter optimization for cross-project defect prediction to identify how the configurations obtained on one project are suitable for other projects.

V. Acknowledgments

The authors gratefully acknowledge the funding of the Swiss National Science Foundations for the project "Agile Software Analysis" (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018).[5]

References

[1] 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)*. IEEE CS Press, 2010, pp. 31–40.

[2] C. Couto, C. Silva, M. Valente, R. Bigonha, and N. Anquetil, "Uncovering causal relationships between software metrics and bugs," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, Mar. 2012, pp. 223–232.

[3] C. Couto, P. Pires, M. T. Valente, R. S. Bigonha, and N. Anquetil, "Predicting software defects with causality tests," *Journal of Systems and Software*, vol. 93, pp. 24–41, 2014.

[4] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 182–191.

[5] K. Muthukumaran, N. Murthy, G. K. Reddy, and M. Aruna, "Comparative study on effectiveness of standard bug prediction approaches," in *Proceedings of the 5th IBM Collaborative Academia Research Exchange Workshop*. ACM, 2013, p. 9.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[5]http://p3.snf.ch/Project-162352

[7] 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*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[8] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[9] Y. Singh, A. Kaur, and R. Malhotra, "Prediction of fault-prone software modules using statistical and machine learning methods," *International Journal of Computer Applications*, vol. 1, no. 22, pp. 8–15, 2010.

[10] R. Malhotra, A. Kaur, and Y. Singh, "Empirical validation of object-oriented metrics for predicting fault proneness at different severity levels using support vector machines," *International Journal of System Assurance Engineering and Management*, vol. 1, no. 3, pp. 269–281, 2010.

[11] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino, "A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction," in *Proceedings of the 27th annual ACM symposium on applied computing*. ACM, 2012, pp. 1215–1220.

[12] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro, "A genetic algorithm to configure support vector machines for predicting fault-prone components," in *International Conference on Product Focused Software Process Improvement*. Springer, 2011, pp. 247–261.

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

[14] 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*, vol. 11, no. 1, pp. 10–18, 2009.

[15] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and software*, vol. 81, no. 5, pp. 823–839, 2008.

[16] 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.*, vol. 17, no. 4, pp. 375–407, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1007/s10515-010-0069-5

[17] 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.*, vol. 34, no. 4, pp. 485–496, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.35

[18] P. Domingos and M. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Mach. Learn.*, vol. 29, no. 2-3, pp. 103–130, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1007413511361

[19] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[20] 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*. IEEE, 2004, pp. 417–428.

[21] 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*. IEEE Press, 2015, pp. 789–800.

[22] J. Sayyad Shirabad and T. Menzies, "The PROMISE repository of software engineering databases," School of Information Technology and Engineering, University of Ottawa, Canada, 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[23] A. Tosun and A. Bener, "Reducing false alarms in software defect prediction by decision threshold optimization," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 477–480.

[24] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 321–332. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884857