An Extensive Analysis of Efficient Bug Prediction Configurations

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

Mircea Lungu
University of Groningen
The Netherlands
m.f.lungu@rug.nl

ABSTRACT

Background: Bug prediction helps developers steer maintenance activities towards the buggy parts of a software. There are many design aspects to a bug predictor, each of which has several options, *i.e.*, software metrics, machine learning model, and response variable.
Aims: These design decisions should be judiciously made because an improper choice in any of them might lead to wrong, misleading, or even useless results. We argue that bug prediction *configurations* are intertwined and thus need to be evaluated in their entirety, in contrast to the common practice in the field where each aspect is investigated in isolation.
Method: We use a cost-aware evaluation scheme to evaluate 60 different bug prediction configuration combinations on five open source Java projects.
Results: We find out that the best choices for building a cost-effective bug predictor are change metrics mixed with source code metrics as independent variables, Random Forest as the machine learning model, and the number of bugs as the response variable. Combining these configuration options results in the most efficient bug predictor across all subject systems.
Conclusions: We demonstrate a strong evidence for the interplay among bug prediction configurations and provide concrete guidelines for researchers and practitioners on how to build and evaluate efficient bug predictors.

KEYWORDS

Bug Prediction, Effort-Aware Evaluation,

ACM Reference format:
Haidar Osman, Mohammad Ghafari, Oscar Nierstrasz and Mircea Lungu. 2017. An Extensive Analysis of Efficient Bug Prediction Configurations. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127017

PROMISE , November 8, 2017, Toronto, Canada

ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127017

1 INTRODUCTION

The main promise of bug prediction is to help software engineers focus their testing and reviewing efforts on those software parts that most likely contain bugs. Under this promise, for a bug predictor to be useful in practice, it must be *efficient*, that is, it must be optimized to locate the maximum number of bugs in the minimum amount of code [31][32][2].[1] Optimizing a bug predictor requires making the right decisions for (i) the independent variables, (ii) the machine learning model, and (iii) the response variable.[2] We call this triple, *bug prediction configurations*.

These configurations are interconnected. The entire configuration should be evaluated in order to provide individual answers for each aspect reliably. However, the advice found in the literature focuses on each aspect of bug prediction in isolation and it is unclear how previous findings hold in a holistic setup. In this paper, we adopt the *Cost-Effectiveness* measure (*CE*), introduced by Arisholm *et al.* [2], to empirically evaluate the different options of each of the bug prediction configurations all at once, shedding light on the interplay among them. Consequently, we pose and answer the following research questions:

RQ1: What type of software metrics are cost-effective? We find that using a mix of source code metrics and change metrics yields the most cost-effective predictors for all subject systems in the studied dataset. We observe that change metrics alone can be a good option, but we advise against using source code metrics alone. These findings contradict the advice found in the literature that object-oriented metrics hinders the cost-effectiveness of models built using change metrics [2]. In fact although source code metrics are the worst metrics set, it can still be used when necessary, but with the right configuration combination.

RQ2: What prediction model is cost-effective? In this study we compare five machine learning models: Multilayer Perceptron, Support Vector Machines, Linear Regression, Random Forest, and K-Nearest Neighbour. Our results show that Random Forest stands out as the most cost-effective one. Support Vector Machines come a close second. While some previous studies suggest that Random Forest performs generally better than other machine learning models [18], other studies note that Random Forest does not perform as well [20]. Our findings suggest that Random Forest performs the best with respect to cost-effectiveness.

[1] Efficient bug prediction as we define it, is sometimes referred to as effort-aware bug prediction in the literature
[2] Also known as the dependent variable or the output variable

RQ3: What is the most cost-effective response variable to predict? We establish that predicting the number of bugs in a software entity is the most cost-effective approach and predicting bug proneness is the least cost-effective one. To our knowledge, this research question has not been investigated before in the literature.

RQ4: Is there a configuration combination that consistently produces highly cost-effective bug predictors? Here we evaluate all configurations at once to provide more reliable guidelines for building cost-effective bug predictors. We conclude that *both source code and change metrics as independent variables mixed, Random Forest as the prediction model, and bug count as the response variable*, form the configuration combination of the most cost-effective bug predictor across all subject systems in the studied dataset.

The rest of the paper is organized as follows: We explain and motivate the experimental setup of our empirical study in section 2. We demonstrate the results and answer the posed research questions in section 3, then we discuss the threats to validity in section 4. We survey the related work and compare our findings with the literature in section 5. Finally, in section 6, we conclude this study with specific guidelines on building cost-effective bug predictors.

2 EMPIRICAL STUDY SETUP

2.1 Evaluation Scheme

There is a strong relationship between what is expected from a model and how the model is evaluated. In the field of bug prediction, the desired value expected from a bug predictor is to enhance the efficiency of the quality assurance procedure by directing it to the buggy parts of a software system. This is possible only when the bug predictor can find most of the bugs in the least amount of code. Intuitively, the efficiency of a predictor increases inverse-proportionally with the number of lines of code in which it suspects a bug might appear because writing unit tests for large software entities or inspecting them requires more effort.

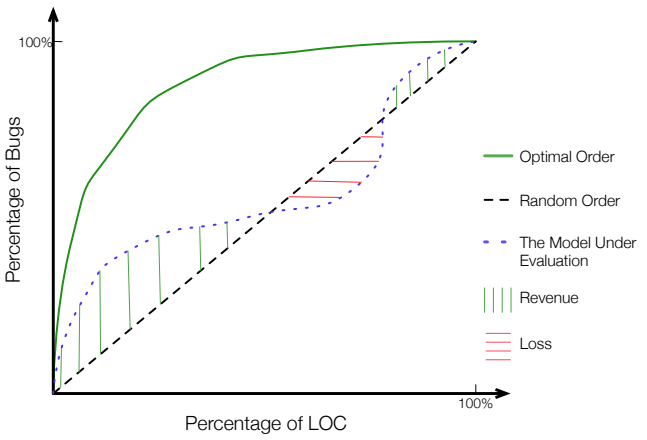


Figure 1: An overview of the *CE* measure as defined by Arisholm *et al.* [2].

Arisholm *et al.* state that *"... the regular confusion matrix criteria, although popular, are not clearly related to the problem at hand, namely the cost-effectiveness of using fault-proneness prediction models to focus verification efforts to deliver software with less faults at less cost"*[2]. Consequently, they proposed a cost-aware evaluation scheme called *Cost-Effectiveness* (*CE*) [2]. *CE* measures the benefit of using a certain bug prediction model. It summarizes the accuracy measures and the usefulness of a model by measuring how close the prediction model is to the optimal model, taking the random order as the baseline. This scheme assumes the ability of the prediction model to rank software entities in an ordered list. To demonstrate *CE*, we show in Figure 1 an example cumulative lift curves (Alberg diagrams [37]) of three orderings of software entities:

(1) **Optimal Order:** The green curve represents the ordering of the software entities with respect to the bug density from the highest to the lowest.
(2) **Random Order:** The dashed diagonal line is achieved when the percentage of bugs is equal to the percentage of lines of code. This is what one gets, on average, with randomly ordering the software entities.
(3) **Predicted Order:** The blue curve represents the ordering of the software entities based on the predicted dependent variable.

The area under each of these diagrams is called the Cost-Effectiveness (*CE*) area. The larger the *CE* area, the more cost-effective the model. However, two things need to be taken into account in this scenario. First, optimal models are different for different datasets. Second, the prediction model should perform better than the random ordering model to be considered valuable. That's why Arisholm *et al.* [2] took the optimal ordering and the random ordering into consideration in the Cost-Effectiveness measure as:

$$CE(model) = \frac{AUC(model) - AUC(random)}{AUC(optimal) - AUC(random)}$$

where $AUC(x)$ is the area under the curve x.

CE assesses how good the prediction model is in producing a total order of entities. The value of *CE* ranges from -1 to +1. The larger the *CE* measure is, the more cost-effective the model is. There are three cases:

(1) When *CE* is close to 0, it means that there is no gain in using the prediction model.
(2) When *CE* is close to 1, it means that the prediction model is close to optimal.
(3) When *CE* is between 0 and -1, it means that the cost of using the model is more than the gain, making the use of the model actually harmful.

In our experiments, we use *CE* to compare prediction results and draw conclusions.

2.2 Dataset

In our study, we use the "Bug Prediction Dataset" provided by D'Ambros *et al.* [11] in the form of publicly available software system metrics at the class level. The purpose of of this dataset is to provide a benchmark for researchers to run bug prediction experiments on. It contains source code metrics and change metrics of five popular Java systems (Table 1) and has been used previously

in many bug prediction studies. The dependent variable is the number of bugs on the Java class level.

Table 1: Details about the systems in the studied dataset, as reported by D'Ambros *et al.* [11]

System	KLOC	#Classes	% Buggy classes	% Buggy classes with more than one bug
Eclipse JDT Core	≈ 224	997	≈ 20%	≈ 33%
Eclipse PDE UI	≈ 40	1,497	≈ 14%	≈ 33%
Equinox	≈ 39	324	≈ 40%	≈ 38%
Mylyn	≈ 156	1,862	≈ 13%	≈ 28%
Lucene	≈ 146	691	≈ 9%	≈ 29%

2.3 Independent Variables

Source code metrics[3] are the metrics extracted from the source code itself. The most often-used metrics are the Chidamber and Kemerer (CK) metrics suite [10]. These more complex metrics are usually used in conjunction with simpler counting metrics like the number of lines of code (LOC), number of methods (NOM), or number of attributes (NOA). Source code metrics try to capture the quality (*e.g.*, LCOM, CBO) and complexity (*e.g.*, WMC, DIT) of the source code itself. The rationale behind using the source code metrics as bug predictors is that there should be a strong relation between source code features (quality and complexity) and software defects [47]. In other words, the more complex a software entity is, the more likely it contains bugs. Also the poorer the software design is, the more bug-prone it is.

Change metrics[4] are extracted from the software versioning systems like CVS, Subversion, and Git. They capture how and when software entities (binaries, modules, classes, methods) change and evolve over time. Change metrics describe software entities with respect to their age [44][5], past faults [27][55][41], past modifications and fixes [17][41][22][21][26][36], and developer information [53][54][43][33]. Using software history metrics as bug predictors is motivated by the following heuristics:

(1) Entities that change more frequently tend to have more bugs.
(2) Entities with a larger number of bugs in the past tend to have more bugs in the future.
(3) Entities that have been changed by new developers tend to have more bugs.
(4) Entities that have been changed by many developers tend to have more bugs.
(5) Bug-fixing activities tend to introduce new bugs.
(6) The older an entity, the more stable it is.

Many researchers argue that source code metrics are good predictors for future defects [23][42][3][8][48][7], but others show that change metrics are better than source code metrics at bug prediction [36][25] [17][26][54][41][54][16]. Moreover, Arisholm *et al.* states that models based on object-oriented metrics are no better than a model based on random class selection [2].

In this study, to take part in this debate, we consider source code metrics alone, change metrics alone, or both combined, and evaluate the cost-effectiveness of the resulted bug predictors.

[3]Source code metrics are also known as product metrics.
[4]Change metrics are also known as process metrics or history metrics

2.4 Response Variable

The chosen dependent variable is an important design decision because it sometimes determines the difference between a usable and an unusable model. We have to keep in mind that the final goal of the prediction is to prioritize the software entities into an ordered list to be able to apply the Cost-Effectiveness (*CE*) measure. There are multiple possible schemes to do it:

(1) predict the number of bugs as the response variable then order the software entities based on the *calculated* bug density.
(2) predict the bug density directly then order the software entities based on this *predicted* bug density.
(3) predict the bug proneness and order the software entities based on it. Small classes come before large ones in case of ties.
(4) classify software entities into buggy and bug-free, then order them as follows: buggy entities come before bug-free ones and small classes come before large ones (with respect to LOC).

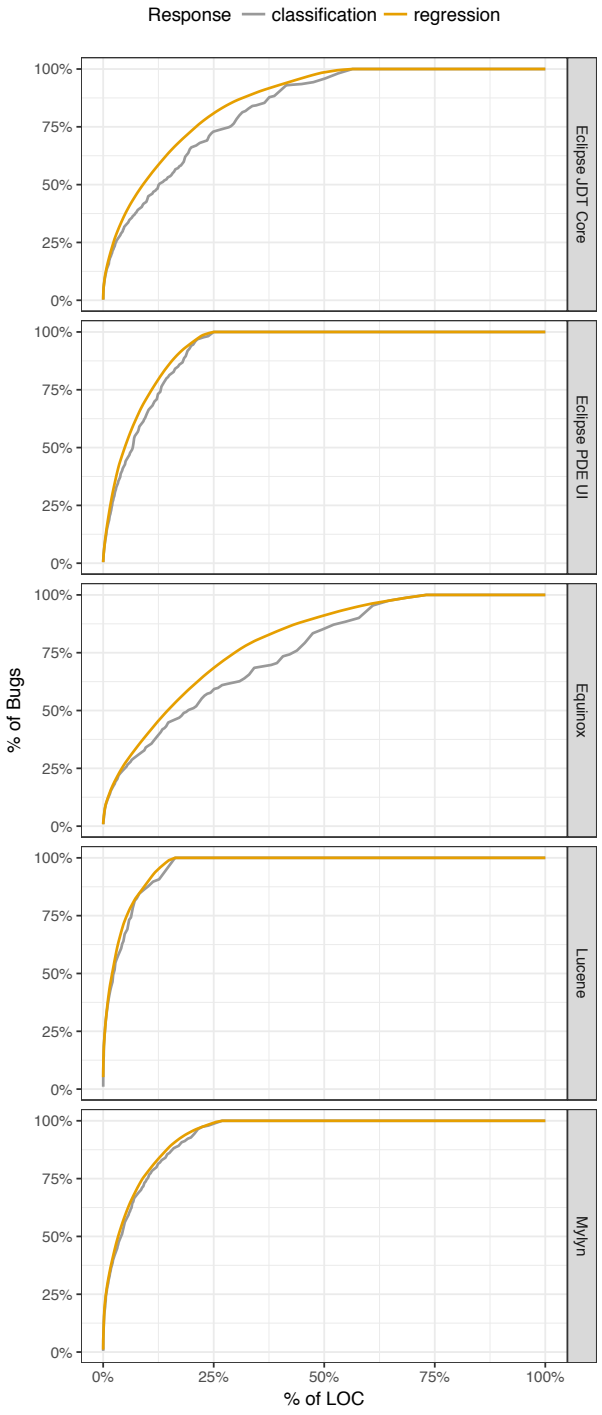
The optimal prediction of number of bugs (first scheme) is equivalent to the optimal prediction of bug density (second scheme), since bug density is calculated from the number of bugs as $bugdensity = (\#bugs/LOC)$. Also the optimal prediction of bug proneness (third scheme) is an optimal classification (fourth scheme). Obviously the optimal regression in the first or second schemes is more cost-effective than the optimal classification in the third or fourth scheme because it reflects exactly the optimal solution in the cost-effectiveness (*CE*) evaluation method. However, we need to verify whether classification is a valid approach in bug prediction and whether we should include the third and fourth schemes in the empirical study. In Figure 2, we evaluate the cost-effectiveness of the optimal classifier following the fourth scheme, for all five systems in our corpus. The cost-effectiveness of the fourth scheme is excellent for Equinox and JDT, and almost optimal for Mylyn, Lucene, and PDE. As shown in Table 1, the percentage of buggy classes with more than one bug ranges from 28% to 38%. This leads to the conclusion that in the set of buggy classes, the number of bugs is proportional to the number of lines of code. This is particularly interesting because it makes classification as good as predicting the number of bugs, in the ideal case. We empirically verify which response variable is better when we train our prediction models.

2.5 Machine Learning Models

We investigate the following machine learning models: Random Forest (RF), Support Vector Machine (SVM), Multilayer Perceptron (MLP), an implementation of the K-nearest neighbours algorithm called IBK, and Linear Regression (LinR) / Logistic Regression (LogR)[5]. We choose these machine learning models for two reasons: First, they are extensively used in the bug prediction literature [29]. Second, each one of them can be used as a regressor and as a classifier, making comparisons across different configurations possible. Classifiers are used to predict the bug proneness or the class (buggy, bug-free) and regressors are used to predict the bug count and

[5]Linear Regression and Logistic Regression are equivalent, but with different types of response variables. Linear Regression is a regressor and Logistic Regression is a classifier.

Figure 2: Commutative lift curves (Alberg diagrams [37]) comparing the optimal regressor (bug density predictor) and the optimal classifier with ranking based on LOC (smallest to largest). These diagrams show that optimal classification performs almost as well as optimal regression.



bug density. We use the Weka data mining tool [19] to build these prediction models.

2.6 Hyperparameter Optimization

Machine learning models may have configurable parameters that should be set before starting the training phase. This process is called hyperparameter optimization or model tuning, and can have a positive effect on the prediction accuracy of the models. However, different models have different sensitivities to this process. While model tuning improves IBK and MLP substantially, it has a negligible effect on SVM and RF [49][39]. In this study, we follow the same procedure proposed by Osman *et al.* for hyperparameter optimization [39]. The used model parameters are detailed in Table 2.

Table 2: The tuning results for the hyperparameters

RF	Number of Trees= 100
SVM	Kernel= RBF {Gamma=0.1} Complexity=10
MLP	Learning Rate=0.6 Momentum=0.6 Hidden Layer Size= 32
IBK	#Neighbours=5 Search Algorithm= Linear Search Evaluation Criterion=Mean Squared Error
LinR/LogR	No parameters to tune

2.7 Feature Selection

The prediction accuracy of machine learning models is highly affected by the quality of the features used for prediction. Irrelevant and correlated features can increase prediction error, increase model complexity, and decrease model stability. Feature selection is a method that identifies the relevant features to feed into machine learning models. We apply wrapper feature selection for SVM, MLP, and IBK as it has been shown that it leads to higher prediction accuracy [40]. We do not apply feature selection for RF because it performs feature selection internally. Following the guidelines by Osman *et al.* [38], we apply $l2$ regularization (Ridge) on LinR/LogR as the feature selection method.

2.8 Data Pre-Processing

Bug datasets are inherently imbalanced where most software entities are bug-free. This is called the class-imbalance problem and can negatively affect the performance of machine learning models [52][1]. To cope with this problem, we divide the data set for each project into two sets: test set (25%) and training set (75%). The samples in each set are taken at random but maintain the distribution of buggy classes similar to the one in the full data set. We then balance the training set by oversampling. This is important for training and for evaluating the prediction models. The machine learning models are then trained using the balanced training set and evaluated on the unseen test set.

3 RESULTS

In this study, we consider the following configurations:

(1) Independent Variables:
  (a) source code metrics (`src`)
  (b) change metrics (`chg`)
  (c) both of them combined (`both`).
(2) Machine Learning Model:
  (a) Support Vector Machines (`SVM`)
  (b) Random Forest (`RF`)
  (c) Multilayer Perceptron (`MLP`)
  (d) K-Nearest Neighbours (`IBK`)
  (e) Linear Regression (`LinR`) or Logistic Regression (`LogR`)
(3) Response Variable:
  (a) bug count (`cnt`)
  (b) bug density (`dns`)
  (c) bug proneness (`prs`)
  (d) classification (`cls`)

There are 60 possible configuration combinations. For each one, we pre-process the data, train the model on the training set, perform the predictions on the test data, and calculate the Cost Effectiveness measure (*CE*). We repeat this process 50 times to mitigate the threat of having outliers because of the random division of the dataset into a training set and a test set. We do not perform k-fold cross-validation method because calculating *CE* over a small set of classes can be misleading. Instead, we perform the repeated hold-out validation because it is known to have lower variance than k-fold cross validation making it more suitable for small datasets [4].

In this experiment, statistically speaking, the treatment is the configuration combination and the outcome is the *CE* score. Hence, we have 60 different treatments and one outcome measure. To answer the posed research questions, we need to compare the *CE* of different configuration combinations. Since there is a large number of treatments, traditional parametric tests (*e.g.*, ANOVA) or non-parametric tests (*e.g.*, Friedman) have the overlapping problem, *i.e.*, the clusters of treatment overlap. Therefore, we use the Scott-Knott (SK) cluster analysis for grouping of means [45], which is a clustering algorithm used as a multiple comparison method for the analysis of variance. SK clusters the treatments into statistically distinct non-overlapping groups (*i.e.*, ranks), which makes it suitable for this study. We apply SK with 95% confidence interval to cluster the configuration combinations for each project in the dataset.

Figure 3 shows box plots of the *CE* outcomes for each configuration combination. Each box plot represents the population of the 100 runs of the corresponding configuration. The box plots are sorted in an increasing order of the means of *CE*, represented by the red points. Alternating background colors indicate the Scott-Knott statistically distinct groups (*i.e.*, clusters or ranks).

The results in Figure 3 clearly demonstrate the interplay between the design choices in bug prediction. Changing one value in the configuration can transform a bug predictor from being highly cost-effective, to being actually harmful. For instance, while `both-RF-cnt` is the most cost-effective configuration in Figure 3(a), `both-RF-dns` is in the least cost-effective cluster. This means that although `RF` is the best machine learning model and `both` is the best choice of metrics, using them with the wrong response variable renders a bug predictor useless. There are many examples where changing one configuration parameter brings the bug predictor from one cluster to another. Examples for each configuration variable include:

(1) In Figure 3(a), `both-RF-cnt` is ranked 1^{st} whereas `both-RF-dns` is ranked 7^{th}.
(2) In Figure 3(b), `both-SVM-cnt` is ranked 2^{nd} whereas `both-MLP-cnt` is ranked 6^{th}.
(3) In Figure 3(e), `chg-SVM-cls` is ranked 2^{nd} where as `src-SVM-cls` is ranked 6^{th}.

These examples constitute a compelling evidence that bug prediction configurations are interconnected.

To answer the first research question (*RQ1*) regarding the choice of independent variables, we analyze the top cluster of configurations in Figure 3. We observe that for Eclipse PDE UI, Equinox, and Mylyn, the software metrics value in the top rank is either `both` or `chg`. In Eclipse JDT Core, `src` appears in one configuration out of three in the top rank. In Lucene, *src* appears in one configuration out of 27 in the top rank. These results suggest that the use of both source code and change metrics together is the most cost-effective option for the independent variables. Using only change metrics is also a good choice, but using only source code metrics rarely is. It was shown in the literature that adding source code metrics to change metrics hinders the cost-effectiveness and using source code alone is not better than random guessing [2]. Our results show that although less cost-effective, source code metrics can be used alone when necessary (*e.g.*, change metrics cannot be computed). There is always a cost-effective configuration combination with the source code metrics as the independent variables (*e.g.*, `src-RF-cnt`).

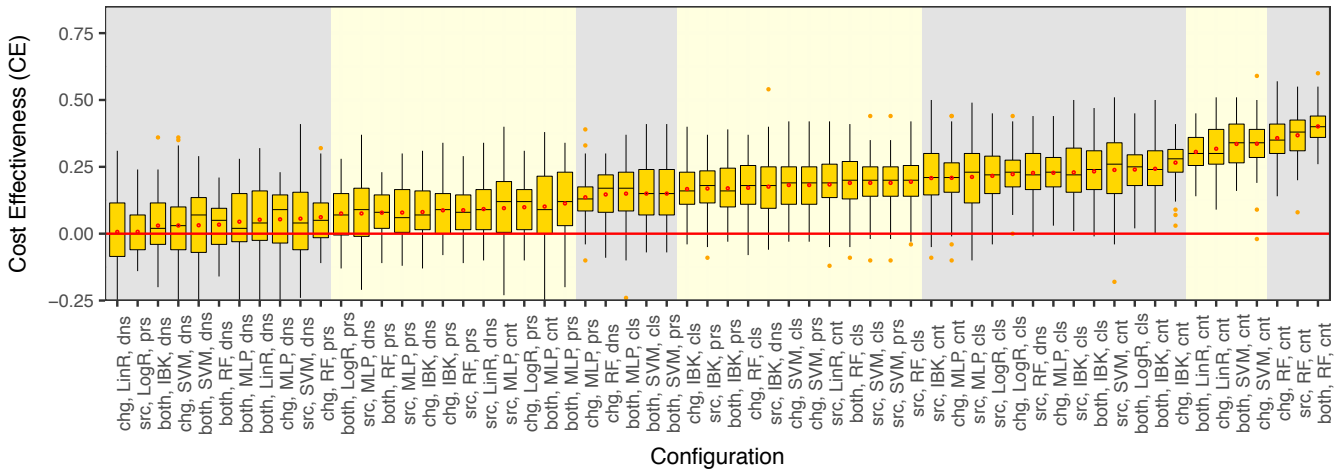
For the second research question (*RQ2*) regarding the choice of the machine learning model, we observe that `RF` is *the only* option in the top rank in Eclipse JDT Core, Eclipse PDE UI, and Equinox, and it is in the top rank of Lucene and Mylyn. `SVM` also performs well. It appears in the top rank in Lucene and Mylyn, and in the second rank in the rest of the projects. These results indicate the superiority of Random Forest and Support Vector Machines in producing cost-effective bug predictors. On the other hand, `MLP` and `IBK` made it to the top two clusters only in Lucene, suggesting that Multilayer Perceptron and K-Nearest Neighbour do not fit the bug prediction problem well.

For the third research question (*RQ3*) regarding the most cost-effective response variable, we observe that `cnt` is in the top rank in Lucene and is *the only* response variable in the top rank in the other projects. It is clear that predicting the bug count results in the most cost-effective bug predictors. Another observation is that the response variable configuration in the bottom two clusters is almost conclusively either `dns` or `prs`. This means that predicting bug density or bug proneness actually hinders the cost-effectiveness of the bug prediction.

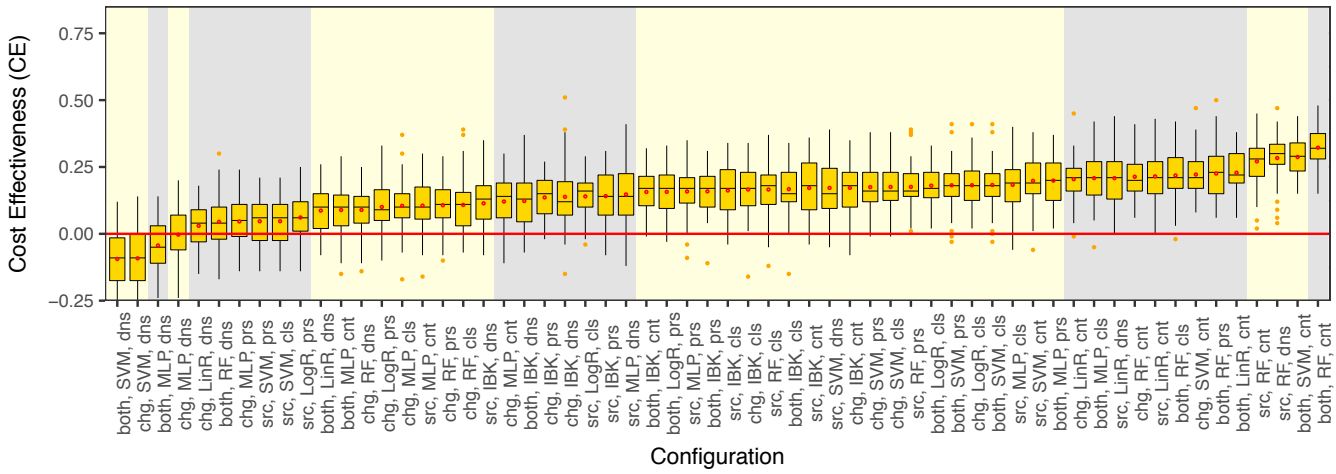
Overall, one result that stands out is that the configuration `both-RF-cnt` is in the top cluster across projects (*RQ4*). In fact, it is *the most* cost-effective configuration in Eclipse JDT Core, Eclipse PDE UI, and Equinox and it is in the top cluster in Lucene and Mylyn. This finding suggests that this configuration seems to be the best from the cost-effectiveness point of view.

Software projects differ in their domains, development methods, used frameworks, and developer experiences. Consequently,

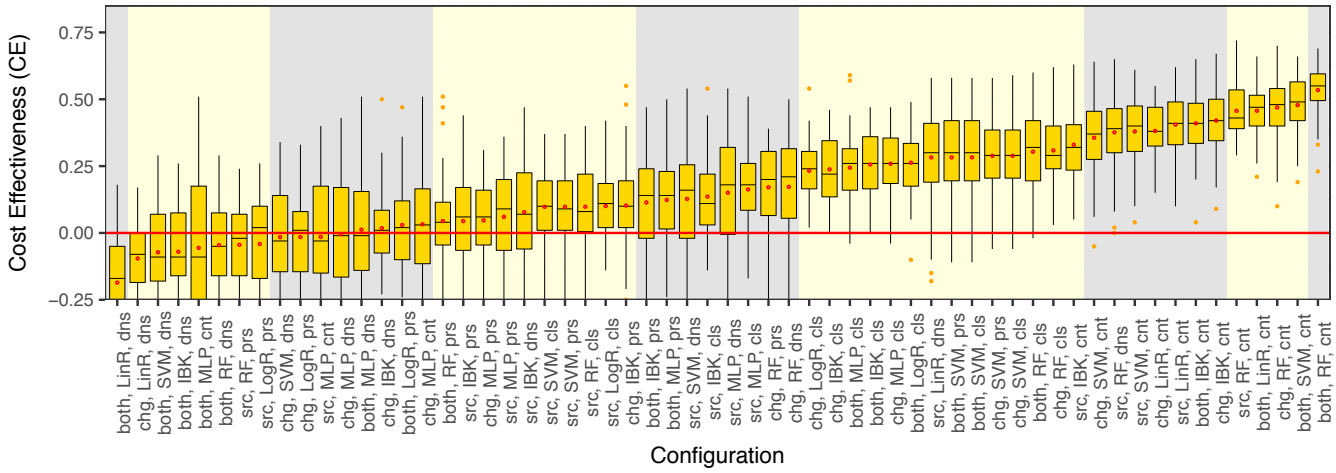
Figure 3: Boxplots of the *CE* outcome. Each box plot represents the *CE* obtained by 100 runs of the corresponding configuration combination on the x-axis. Different background colors indicate the statistically distinct groups obtained by applying the Scott-Knott clustering method with 95% confidence interval. The configurations on the x-axis are of the form `Metrics-Model-Response`.



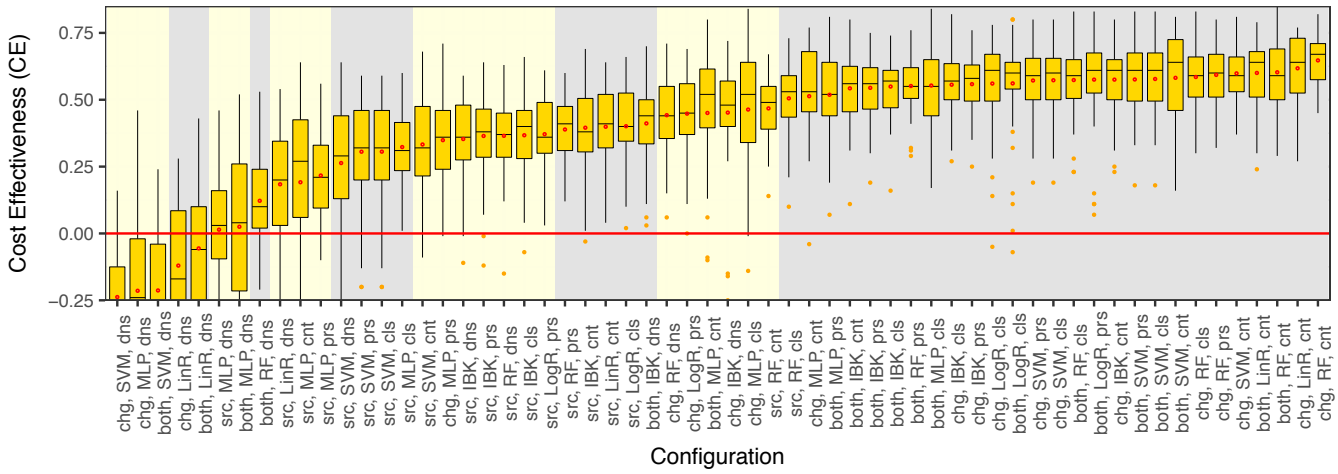
(a) Eclipse JDT Core



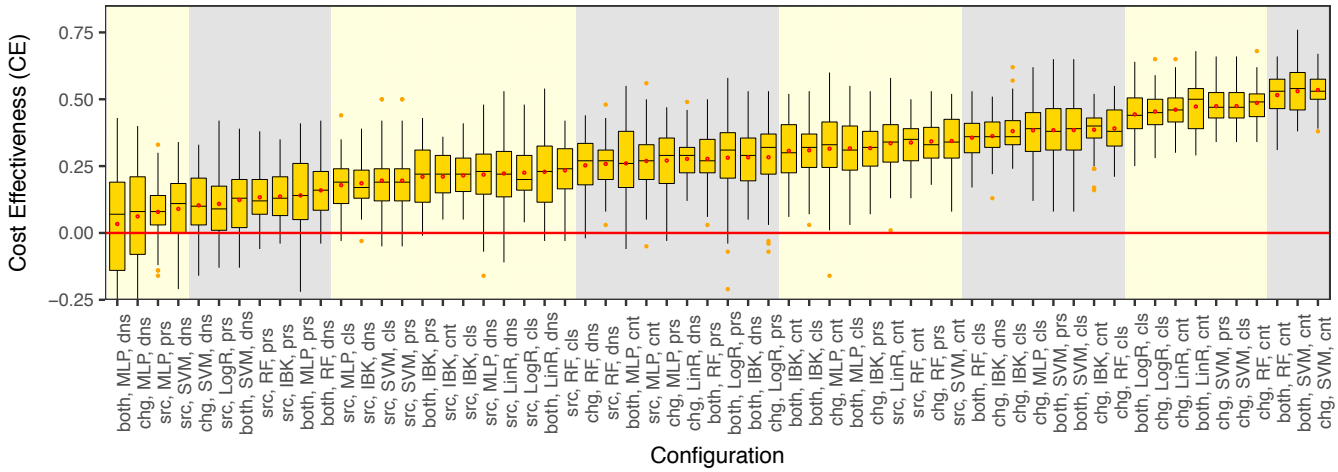
(b) Eclipse PDE UI



(c) Equinox

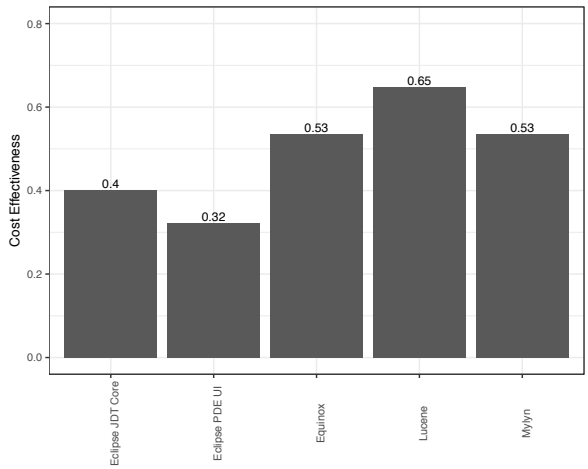


(d) Lucene



(e) Mylyn

Figure 4: The max mean values of *CE* obtained for each project



software metrics differ in the correlation with the number of bugs among projects. This is the reason why using both metrics came out as the best choice of independent variable. However, to deal with the inevitable noise and redundancy in using both metrics, the best configurations includes Random Forest as the machine learning model. Random Forest is an ensemble of decision trees created by using bootstrap samples of the training data and random feature selection in tree induction [6]. This gives RF the ability to work well with high-dimensional data and sift the noise away. This is the reason why feeding `both` types of metrics into Random Forest actually makes sense. Also bug count came out as the best option for response variable because it reflects the "gain" in *CE* better than classification or proneness. Bug density also reflects the "gain" but it is better to calculate it from bug count than to leave it to the prediction model to deduce. Therefore, bug count is a simpler and more appropriate response variable than bug density. All these factors contribute to the fact that `both-RF-cnt` is the most cost-effective configuration for bug prediction.

Finally, the results in Figure 4 also show that the cost-effectiveness of the best bug predictor varies among projects. Although the best bug predictor is never harmful to use (no negative *CE*) in our experiments, it can still be of little value for some projects, *e.g.*, $CE = 0.32$ for Eclipse PDE UI. This means that bug prediction should be evaluated as a technique in the context of a software project before putting it in use in that specific project.

4 THREATS TO VALIDITY

To minimize the threats to validity in our empirical study, we follow the guidelines of Mende [30] by

- using a large dataset to avoid large variance of performance measures,
- maintaining the same fault distribution in the test set and training set as the original set, to minimize bias,
- repeating the experiment 50 times to minimize the effect of outliers,
- and reporting on the dataset, data preprocessing procedure, and model configurations to enable replication.

In our study we use the "Bug Prediction Dataset" provided by D'Ambros *et al.* [11] as a benchmark. Although it is a well-known and studied dataset, the quality of our results is very dependent on the quality of that dataset.

Our dependence on WEKA [19] for building the artificial intelligence models, makes the quality of the models dependent solely on the quality of WEKA itself.

The fact that this dataset contains metrics only from open source software systems makes it hard to generalize to all Java systems. In the future, we plan to apply our study on more datasets and to use other data mining tools.

Another threat to validity comes from the use of LOC as a proxy for cost. As we explained before, reviewing code and writing unit tests take much more effort for large modules than small ones. However, we are aware of the fact that this proxy might introduce some bias. We use it because it has been used in several previous studies as such (*e.g.*, [31][2]) and it is widely accepted in the community as a good measure of effort.

Our study is on the Java-class level. Hence, our findings may not apply on other granularity levels such as method level or commit level.

Finally, in this study, we assume that the purpose of bug prediction is to locate the maximum number of bugs in the minimum amount of code in order to be a useful support to quality assurance activities. However, defect prediction models can be used for other purposes. For example, they can be used as tools for understanding common pitfalls and analyzing factors that affect the quality of software. In these cases, our findings do not necessarily apply. In the future, we plan to extend this study to the broader context of several defect prediction use cases.

5 RELATED WORK

Most studies comparing different machine learning models in bug prediction show no difference in performance [12][28] [51][35]. Menzies *et al.* [35] evaluate many models using the area under the curve of a probability of false alarm versus probability of detection "AUC(pd, pf)". They conclude that better prediction models do not yield better results. Similarly, Vandecruys *et al.* [51] compare the accuracy, specificity (true negative rate), and sensitivity (recall or true positive rate) between seven classifiers. Using the non-parametric Friedman test, as recommended in this type of problem [12], it is shown that there is no statistically significant difference at the 5% significance level. Lessmann *et al.* [28] study 22 classifiers and conclude that the classification model is less important than generally assumed, giving researchers more freedom in choosing models when building bug predictors. Actually simple models like naïve Bayes or C4.5 classifiers perform as well as more complicated models [13][34]. Other studies suggest that there are certain models which perform better than others in predicting bugs. Elish and Elish [14] compare SVM against 8 machine learning and statistical models and show that SVM performs classification generally better. Guo *et al.* [18] compare Random Forest with other classifiers and show how it generally achieves better prediction. Ghotra *et al.* [15] show that there are four statistically distinct groups of classification techniques suggesting that the choice of the classification model has a significant impact on the performance of bug prediction. Our findings confirm the superiority of certain models over others. Specifically, we show that Random Forest is indeed the best machine learning model, followed by Support Vector Machines.

Mende and Koschke [32] studied the concept of effort-aware defect prediction. They compared models with predicting defect density using only Random Forest trained only on source code metrics. They took the effort into account during the training phase by predicting bug density as a response variable. Although we agree with Mende and Koschke on the importance of building effort-aware prediction models, our results actually advise against using source code metrics and bug density as independent and dependent variables respectively. Canfora *et al.* also consider cost in the training phase [9]. Using genetic algorithms, they trained a multi-objective logistic regressor that, based on developer preferences, is either highly effective, with low cost, or balanced. They treated bug prediction as a classification problem and they used only source code metrics as independent variables. We agree with Canfora *et al.* that bug predictors should be tuned to be cost-effective, but our study shows that source code metrics are rarely a good choice of independent variables, and predicting bug count is more cost-effective than predicting bug proneness.

Kamei *et al.* [25] also evaluated the predictive power of history and source code metrics in an effort-sensitive manner. They used regression model, regression tree, and Random Forest as models. They found that history metrics significantly outperform source code metrics with respect to predictive power when effort is taken into consideration. Our results confirm their findings but also add that the use of both metrics is even more cost-effective.

Arisholm *et al.* [2] studied several prediction models using history and source code metrics in bug prediction. They also dealt with the class imbalance problem and performed effort-aware evaluation. They found that I) history metrics perform better than source code metrics and II) source code metrics are no better than random class selection. Our results confirm their first finding but contradict the second. We show that indeed using source code metrics is less cost-effective than using change metrics or a mix of both. However, using source code metrics with the right options of other

configurations is certainly better than random class selection. This contradiction with our findings comes from the fact that the dependent variable in their study is bug proneness and, as opposed to our study, they did not consider other dependent variables.

Jiang *et al.* [24] surveyed many evaluation techniques for bug prediction models and concluded that the system cost characteristics should be taken into account to find the best model for that system. The cost Jiang *et al.* considered is the cost of misclassification because they dealt with the bug prediction problem as a classification problem. In our study, we treat bug prediction as a regression problem (*i.e.*, bug count and bug density) and as a classification problem (bug proneness and entity class) and the cost of the model is the actual cost of maintenance using LOC as a proxy.

Finally, there are many other studies that look into the effect of experimental setup on bug prediction studies. Tantithamthavorn *et al.* show how different evaluation schemes can lead to different outcome in bug prediction research [50]. In this study we focus on configuring bug predictors to be cost-effective. Thus, we fix the evaluation scheme to the one the reflects our purpose (*i.e.*, *CE*). In a systematic literature review, Hall *et al.* [20] define some criteria that makes a bug prediction paper and its results reproducible. Surprisingly, out of 208 surveyed papers, only 36 were selected. In this paper we adhere to these criteria by extensively describing our empirical setup. Shepperd *et al.* [46] raised concerns about the conflicting results in previous bug prediction studies. They surveyed 42 primary studies. They found out that the choice of the classification technique, the metric family, and the data set have small impact on the variability of the results. The variability comes mainly from the research group conducting the study. Shepperd *et al.* conclude that who is doing the work matters more than what is done, meaning that there is a high level of researcher bias. We agree with the authors that there are many factors that might affect the outcome of a bug prediction study. In fact this is the main motivation behind our study. However, Shepperd *et al.* looked at studies that treat bug prediction as a classification problem, ignoring the fact that the response variable is itself a factor that affects the outcome. In our study we include more factors and emphasize on the interplay among them.

6 CONCLUSIONS & FUTURE WORK

Bug prediction is used to reduce the costs of testing and code reviewing by directing maintenance efforts towards the software entities that most likely contain bugs. From this point of view, a successful bug predictor finds the largest number of bugs in the least amount of code. Using the cost effectiveness evaluation scheme, we carry out a large-scale empirical study to find the most efficient bug prediction configurations, as building a bug predictor entails many design decisions, each of which has many options. We summarize the findings of this study as follows:

(1) Using a mix of source code and change metrics is the most cost-effective option for the independent variables. Change metrics alone is a good option.
(2) Random Forest results is the best machine learning model, followed by Support Vector Machines.
(3) Bug count is the most cost-effective option for the response variable. Bug density and bug proneness are the least cost-effective and should be avoided.
(4) The combination of the above configurations results in the most cost-effective bug predictor in all systems in the used dataset.

Finally, our findings reveal a compelling evidence that bug prediction configurations are interconnected. Changing one configuration can render a bug predictor useless. Hence, we advise that future bug prediction studies take this factor into account.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Analysis” (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

REFERENCES

[1] Harald Altinger, Steffen Herbold, Friederike Schneemann, Jens Grabowski, and Franz Wotawa. 2017. Performance Tuning for Automotive Software Fault Prediction. In *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.
[2] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. 2010. A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. *J. Syst. Softw.* 83, 1 (Jan. 2010), 2–17. https://doi.org/10.1016/j.jss.2009.06.055
[3] V.R. Basili, L.C. Briand, and W.L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on* 22, 10 (Oct. 1996), 751–761. https://doi.org/10.1109/32.544352
[4] Claudia Beleites, Richard Baumgartner, Christopher Bowman, Ray Somorjai, Gerald Steiner, Reiner Salzer, and Michael G Sowa. 2005. Variance reduction in estimating classification error using sparse datasets. *Chemometrics and intelligent laboratory systems* 79, 1 (2005), 91–100.
[5] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. 2007. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting (IWPSE '07)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/1294948.1294953
[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[7] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software* 51, 3 (2000), 245–273. https://doi.org/10.1016/S0164-1212(99)00102-8
[8] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. 1999. Investigating Quality Factors in Object-oriented Designs: An Industrial Case Study. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 345–354. https://doi.org/10.1145/302405.302654
[9] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. 2013. Multi-objective Cross-Project Defect Prediction. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. 252–261. https://doi.org/10.1109/ICST.2013.38
[10] Shyam R. Chidamber and Chris F. Kemerer. 1991. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA '91)*. ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/117954.117970
[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An Extensive Comparison of Bug Prediction Approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*. IEEE CS Press, 31–40.
[12] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.
[13] Pedro Domingos and Michael Pazzani. 1997. On the Optimality of the Simple Bayesian Classifier Under Zero-One Loss. *Mach. Learn.* 29, 2-3 (Nov. 1997), 103–130. https://doi.org/10.1023/A:1007413511361
[14] Karim O Elish and Mahmoud O Elish. 2008. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software* 81, 5 (2008), 649–660.
[15] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. 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, 789–800.

[16] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 171–180.

[17] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering* 26, 2 (2000).

[18] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 417–428.

[19] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[20] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[21] Ahmed E. Hassan and Richard C. Holt. 2005. The Top Ten List: Dynamic Fault Prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 263–272. https://doi.org/10.1109/ICSM.2005.91

[22] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug Prediction Based on Fine-grained Module Histories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 200–210. http://dl.acm.org/citation.cfm?id=2337223.2337247

[23] Steffen Herbold. 2013. Training Data Selection for Cross-project Defect Prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering (PROMISE '13)*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/2499393.2499395

[24] Yue Jiang, Bojan Cukic, and Yan Ma. 2008. Techniques for Evaluating Fault Prediction Models. *Empirical Softw. Engg.* 13, 5 (Oct. 2008), 561–595. https://doi.org/10.1007/s10664-008-9079-3

[25] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A.E. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. 1–10. https://doi.org/10.1109/ICSM.2010.5609530

[26] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 489–498. https://doi.org/10.1109/ICSE.2007.66

[27] M. Klas, F. Elberzhager, J. Munch, K. Hartjes, and O. von Graevemeyer. 2010. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, Vol. 2. 119–128. https://doi.org/10.1145/1810295.1810313

[28] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 485–496. https://doi.org/10.1109/TSE.2008.35

[29] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[30] Thilo Mende. 2010. Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 5.

[31] Thilo Mende and Rainer Koschke. 2009. Revisiting the Evaluation of Defect Prediction Models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE '09)*. ACM, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/1540438.1540448

[32] Thilo Mende and Rainer Koschke. 2010. Effort-Aware Defect Prediction Models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE Computer Society, Washington, DC, USA, 107–116. https://doi.org/10.1109/CSMR.2010.18

[33] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/1453101.1453106

[34] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *Software Engineering, IEEE Transactions on* 33, 1 (Jan. 2007), 2–13. https://doi.org/10.1109/TSE.2007.256941

[35] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Automated Software Engg.* 17, 4 (Dec. 2010), 375–407. https://doi.org/10.1007/s10515-010-0069-5

[36] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. 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)*. ACM, New York, NY, USA, 181–190. https://doi.org/10.1145/1368088.1368114

[37] Niclas Ohlsson and Hans Alberg. 1996. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Trans. Softw. Eng.* 22, 12 (Dec. 1996), 886–894. https://doi.org/10.1109/32.553637

[38] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Automatic Feature Selection by Regularization to Improve Bug Prediction Accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*. 27–32. https://doi.org/10.1109/MALTESQUE.2017.7882013

[39] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Hyperparameter Optimization to Improve Bug Prediction Accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*. 33–38. https://doi.org/10.1109/MALTESQUE.2017.7882014

[40] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. The Impact of Feature Selection on Predicting the Number of Bugs. *Information and Software Technology* (2017), in review.

[41] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. 2005. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on* 31, 4 (April 2005), 340–355. https://doi.org/10.1109/TSE.2005.49

[42] A Panichella, R. Oliveto, and A De Lucia. 2014. 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*. 164–173. https://doi.org/10.1109/CSMR-WCRE.2014.6747166

[43] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 2–12. https://doi.org/10.1145/1453101.1453105

[44] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 341–350. https://doi.org/10.1145/1368088.1368135

[45] ANDREW JHON Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[46] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[47] R. Subramanyam and M.S. Krishnan. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on* 29, 4 (April 2003), 297–310. https://doi.org/10.1109/TSE.2003.1191795

[48] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. 1999. An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*. 242–249. https://doi.org/10.1109/METRIC.1999.809745

[49] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 321–332. https://doi.org/10.1145/2884781.2884857

[50] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2017), 1–18.

[51] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. 2008. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software* 81, 5 (2008), 823–839.

[52] Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443.

[53] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2007. Using Developer Information As a Factor for Fault Prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07)*. IEEE Computer Society, Washington, DC, USA, 8–. https://doi.org/10.1109/PROMISE.2007.14

[54] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2008. Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Softw. Engg.* 13, 5 (Oct. 2008), 539–559. https://doi.org/10.1007/s10664-008-9082-8

[55] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. 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)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/1595696.1595713