Automated Parameter Optimization of Classification Techniques for Defect Prediction Models

Chakkrit Tantithamthavorn[1], Shane McIntosh[2], Ahmed E. Hassan[3], Kenichi Matsumoto[1]

[1]Nara Institute of Science and Technology, Japan
{chakkrit-t,matumoto}
@is.naist.jp

[2]McGill University, Canada
shane.mcintosh@mcgill.ca

[3]Queen's University, Canada
ahmed@cs.queensu.ca

ABSTRACT

Defect prediction models are classifiers that are trained to identify defect-prone software modules. Such classifiers have configurable parameters that control their characteristics (e.g., the number of trees in a random forest classifier). Recent studies show that these classifiers may underperform due to the use of suboptimal default parameter settings. However, it is impractical to assess all of the possible settings in the parameter spaces. In this paper, we investigate the performance of defect prediction models where Caret — an automated parameter optimization technique — has been applied. Through a case study of 18 datasets from systems that span both proprietary and open source domains, we find that (1) Caret improves the AUC performance of defect prediction models by as much as 40 percentage points; (2) Caret-optimized classifiers are at least as stable as (with 35% of them being more stable than) classifiers that are trained using the default settings; and (3) Caret increases the likelihood of producing a top-performing classifier by as much as 83%. Hence, we conclude that parameter settings can indeed have a large impact on the performance of defect prediction models, suggesting that researchers should experiment with the parameters of the classification techniques. Since automated parameter optimization techniques like Caret yield substantially benefits in terms of performance improvement and stability, while incurring a manageable additional computational cost, they should be included in future defect prediction studies.

CCS Concepts

•**General and reference** → *Experimentation;* •**Software and its engineering** → **Software defect analysis;** *Search-based software engineering;*

Keywords

Software defect prediction, experimental design, classification techniques, parameter optimization

ICSE '16, May 14-22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884857

1. INTRODUCTION

The limited Software Quality Assurance (SQA) resources of software organizations must focus on software modules (e.g., source code files) that are likely to be defective in the future. To that end, defect prediction models are trained to identify defect-prone software modules using statistical or machine learning classification techniques.

Such classification techniques often have configurable parameters that control characteristics of the classifiers that they produce. For example, the number of decision trees of which a random forest classifier is comprised can be configured prior to training the forest. Furthermore, the number of non-overlapping clusters of which a k-nearest neighbours classifier is comprised must be configured prior to using the classification technique.

Since the optimal settings for these parameters are not known ahead of time, the settings are often left at default values. Prior work suggests that defect prediction models may underperform if they are trained using suboptimal parameter settings. For example, Jiang *et al.* [22] and Tosun *et al.* [58] also point out that the default parameter settings of random forest and naive bayes are often suboptimal. Koru *et al.* [13] and Mende *et al.* [36, 37] show that selecting different parameter settings can impact the performance of defect models. Hall *et al.* [15] show that unstable classification techniques may underperform due to the use of default parameter settings. Mittas *et al.* [41] and Menzies *et al.* [40] argue that unstable classification techniques can make replication of defect prediction studies more difficult.

Indeed, we perform a literature analysis that reveals that 26 of the 30 most commonly used classification techniques (87%) require at least one parameter setting. Since such parameter settings may impact the performance of defect prediction models, the settings should be carefully selected. However, it is impractical to assess all of the possible settings in the parameter space of a single classification technique [1, 16, 27]. For example, Kocaguneli *et al.* [27] point out that there are at least 17,000 possible settings to explore when training k-nearest neighbours classifier.

In this paper, we investigate the performance of defect prediction models where Caret [30] — an off-the-shelf automated parameter optimization technique — has been applied. Caret evaluates candidate parameter settings and suggests the optimized setting that achieves the highest performance. Through a case study of 18 datasets from systems that span both proprietary and open source domains, we record our observations with respect to two dimensions:

Table 1: Overview of studied parameters.

Family	Family Description	Parameter Name	Parameter Description	Classification techniques that apply with their default and candidate parameter values.
Naive Bayes	Naive Bayes is a probability model that assumes that predictors are independent of each other. **Techniques:** Naive Bayes (NB).	Laplace Correction	[N] Laplace correction (0 indicates no correction).	NB={**0**}
		Distribution Type	[L] TRUE indicates a kernel density estimation, while FALSE indicates a normal density estimation.	NB={TRUE, **FALSE**}
Nearest Neighbour	Nearest neighbour is an algorithm that stores all available observations and classifies new observations based on its similarity to prior observations. **Techniques:** *k*-Nearest Neighbour (KNN).	#Clusters	[N] The numbers of non-overlapping clusters to produce.	KNN={**1**, 5, 9, 13, 17}
Regression	Logistic regression is a technique for explaining binary dependent variables. MARS is a non-linear regression modelling technique. **Techniques:** GLM and MARS.	Degree Interaction	[N] The maximum degree of interaction (Friedman's mi). The default is 1, meaning build an additive model (i.e., no interaction terms).	MARS={**1**}
Partial Least Squares	Partial Least Squares regression generalizes and combines features from principal component analysis and multiple regression. **Techniques:** Generalized Partial Least Squares (GPLS).	#Components	[N] The number of PLS components.	GPLS={**1**, 2, 3, 4, 5}
Neural Network	Neural network techniques are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. **Techniques:** Standard (NNet), Model Averaged (AVNNet), Feature Extraction (PCANNet), Radial Basis Functions (RBF), Multi-layer Perceptron (MLP), Voted-MLP (MLPWeightDecay), and Penalized Multinomial Regression (PMR).	Bagging	[L] Should each repetition apply baggin?	AVNNet={TRUE, **FALSE**}
		Weight Decay	[N] A penalty factor to be applied to the errors function.	MLPWeightDecay, PMR, AVNNet, NNet, PCANNet={**0**, 0.0001, 0.001, 0.01, 0.1}, SVMLinear={**1**}
		#Hidden Units	[N] Numbers of neurons in the hidden layers of the network that are used to produce the prediction.	MLP, MLPWeightDecay, AVNNet, NNet, PCANNet={**1**, 3, 5, 7, 9}, RBF={**11**, 13, 15, 17, 19}
Discrimination Analysis	Discriminant analysis applies different kernel functions (e.g., linear) to classify a set of observations into predefined classes based on a set of predictors. **Techniques:** Linear Discriminant Analysis (LDA), Penalized Discriminant Analysis (PDA), and Flexible Discriminant Analysis (FDA).	Product Degree	[N] The number of degrees of freedom that are available for each term.	FDA={**1**}
		Shrinkage Penalty Coefficient	[N] A shrinkage parameter to be applied to each tree in the expansion (a.k.a., learning rate or step-size reduction).	PDA={**1**, 2, 3, 4, 5}
		#Terms	[N] The number of terms in the model.	FDA={**10**, 20, 30, 40, 50}
Rule-based	Rule-based techniques transcribe decision trees using a set of rules for classification. **Techniques:** Rule-based classifier (Rule), and Ripper classifier (Ripper).	#Optimizations	[N] The number of optimization iterations.	Ripper={1, **2**, 3, 4, 5}
Decision Trees-Based	Decision trees use feature values to classify instances. **Techniques:** C4.5-like trees (J48), Logistic Model Trees (LMT), and Classification And Regression Trees (CART).	Complexity	[N] A penalty factor to be applied to the error rate of the terminal nodes of the tree.	CART={0.0001, 0.001, **0.01**, 0.1, 0.5}
		Confidence	[N] The confidence factor used for pruning (smaller values incur more pruning).	J48={**0.25**}
		#Iterations	[N] The numbers of iterations.	LMT={**1**, 21, 41, 61, 81}
SVM	Support Vector Machines (SVMs) use a hyperplane to separate two classes (i.e., defective or not). **Techniques:** SVM with Linear kernel (SVMLinear), and SVM with Radial basis function kernel (SVMRadial).	Sigma	[N] The width of Gaussian kernels.	SVMRadial={0.1, 0.3, **0.5**, 0.7, 0.9}
		Cost	[N] A penalty factor to be applied to the number of errors.	SVMRadial={0.25, 0.5, **1**, 2, 4}, SVMLinear={**1**}
Bagging	Bagging methods combine different base learners together to solve one problem. **Technique:** Random Forest (RF), Bagged CART (BaggedCART)	#Trees	[N] The numbers of classification trees.	RF={**10**, 20, 30, 40, 50}
Boosting	Boosting performs multiple iterations, each with different example weights, and makes predictions using voting of classifiers. **Techniques:** Gradient Boosting Machine (GBM), Adaptive Boosting (AdaBoost), Generalized linear and Additive Models Boosting (GAMBoost), Logistic Regression Boosting (LogitBoost), eXtreme Gradient Boosting Tree (xGBTree), and C5.0.	#Boosting Iterations	[N] The numbers of iterations that are used to construct models.	C5.0={**1**, 10, 20, 30, 40}, GAMBoost={**50**, 100, 150, 200, 250}, LogitBoost={**11**, 21, 31, 41, 51}, GBM,xGBTree={50, **100**, 150, 200, 250}
		#Trees	[N] The numbers of classification trees.	AdaBoost={**50**, 100, 150, 200, 250}
		Shrinkage	[N] A shrinkage factor to be applied to each tree in the expansion (a.k.a., learning rate or step-size reduction).	GBM={**0.1**}, xGBTree={**0.3**}
		Max Tree Depth	[N] The maximum depth per tree.	AdaBoost, GBM, xGBTree={**1**, 2, 3, 4, 5}
		Min. Terminal Node Size	[N] The minimum terminal nodes in trees.	GBM={**10**}
		Winnow	[L] Should predictor winnowing (i.e feature selection) be applied?	C5.0={**FALSE**, TRUE}
		AIC Prune?	[L] Should pruning using stepwise feature selection be applied?	GAMBoost={**FALSE**, TRUE}
		Model Type	[F] Either tree for the predicted class or rules for model confidence values.	C5.0={**rules**, tree}

[N] denotes a numeric value; [L] denotes a logical value; [F] denotes a factor value.
The default values are shown in bold typeface and correspond to the default values of the Caret R package.

(1) Performance improvement: Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

(2) Performance stability: Caret-optimized classifiers are at least as stable as classifiers that are trained using the default settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are more stable than classifiers that are trained using the default values.

Since we find that parameter settings can have such an impact on model performance, we revisit prior analyses that rank classification techniques by their ability to yield top-performing defect prediction models. We find that Caret increases the likelihood of producing a top-performing classifier by as much as 83%, suggesting that automated parameter optimization can substantially shift the ranking of classification techniques.

Our results lead us to conclude that parameter settings can indeed have a large impact on the performance of defect prediction models, suggesting that researchers should experiment with the parameters of the classification techniques. Since automated parameter optimization techniques like Caret yield substantially benefits in terms of performance improvement and stability, while incurring a manageable additional computational cost, they should be included in future defect prediction studies.

To the best of our knowledge, this is the first paper to study:

- A large collection of 43 parameters that are derived from 26 of the most frequently-used classification techniques in the context of defect prediction.
- The improvement and stability of the performance of defect prediction models when automated parameter optimization is applied.
- The ranking of classification techniques for defect prediction when automated parameter optimization is applied.

Paper Organization. The remainder of the paper is organized as follows. Section 2 illustrates the importance of parameter settings of classification techniques for defect prediction models. Section 3 positions this paper with respect to the related work. Section 4 presents the design and approach of our case study. Section 5 presents the results of our case study with respect to our two research questions. Section 6 revisits prior analyses that rank classification techniques by their likelihood of producing top-performing defect prediction models. Section 7 discusses the broader implications of our findings. Section 8 discloses the threats to the validity of our study. Finally, Section 9 draws conclusions.

2. THE RELEVANCE OF PARAMETER SETTINGS FOR DEFECT PREDICTION MODELS

A variety of classification techniques are used to train defect prediction models. Since some classification techniques do not require parameter settings (e.g., logistic regression), we first assess whether the most commonly used classification techniques require parameter settings.

We first start with the 6 families of classification techniques that are used by Lessmann *et al.* [32]. Based on a recent literature review of Laradji *et al.* [31], we add 5 additional families of classification techniques that have been recently used in defect prediction studies. In total, we study 30 classification techniques that span 11 classifier families. Table 1 provides an overview of the 11 families of classification techniques.

Our literature analysis reveals that 26 of the 30 most commonly used classification techniques (87%) require at least one parameter setting. Table 1 provides an overview of the 25 unique parameters that apply to the studied classification techniques.

26 of the 30 most commonly used classification techniques (87%) require at least one parameter setting, indicating that selecting an optimal parameter setting for defect prediction models is an important experimental design choice.

3. RELATED WORK & RESEARCH QUESTIONS

Recent research has raised concerns about parameter settings of classification techniques when applied to defect prediction models. For example, Koru *et al.* [13] and Mende *et al.* [36, 37] point out that selecting different parameter settings can impact the performance of defect models. Jiang *et al.* [22] and Tosun *et al.* [58] also point out that the default parameter settings of research toolkits (e.g., R [46], Weka [14], Scikit-learn [44], MATLAB [35]) are suboptimal.

Although prior work suggests that defect prediction models may underperform if they are trained using suboptimal parameter settings, parameters are often left at their default values. For example, Mende *et al.* [38] use the default number of decision trees to train a random forest classifier (provided by an R package). Weyuker *et al.* [60] also train defect models using the default setting of C4.5 that is provided by Weka. Jiang *et al.* [21] and Bibi *et al.* [2] also use the default value of the k-nearest neighbours classification technique ($k = 1$).

In addition, the implementations of classification techniques that are provided by different research toolkits often use different default settings. For example, for the number of decision trees of the random forest technique, the default settings vary from 10 for the `bigrf` R package [34], 50 for MATLAB [35], 100 for Weka [14], to 500 for the `randomForest` R package [33]. Moreover, for the number of hidden layers of the neuron networks techniques, the default settings vary from 1 for the `neuralnet` R package [11], 2 for the `nnet` R package [48] and Weka [14], to 10 for MATLAB [35]. Such a variation of default settings that are provided by different research toolkits may influence conclusions of defect prediction studies [53].

There are many empirical studies in the area of Search-Based Software Engineering (SBSE) [16] that aim to optimize software engineering tasks (e.g., software testing [20]). However, little SBSE research has been applied to optimize the parameters of classification techniques for defect prediction models. Although prior studies have explored the impact of parameter settings, they have only explored a few parameter settings. To more rigorously explore the parameter space of classification techniques for defect prediction models, we formulate the following research question:

(RQ1) How much does the performance of defect prediction models improve when automated parameter optimization is applied?

Recent research voices concerns about the stability of performance estimates that are obtained from classification techniques when applied to defect prediction models. For example, Menzies *et al.* [40] and Mittas *et al.* [41] argue that unstable classification techniques can make replication of defect prediction studies more difficult. Shepperd *et al.* [49], and Jorgensen *et al.* [23] also point out that the unstable performance estimates that are produced by classification techniques may introduce bias, which can mislead different research groups to draw erroneous conclusions. Myrtveit *et al.* [42] show that high variance in performance estimates from classification techniques is a critical problem in comparative studies of prediction models. Song *et al.* [51] also show that applying different settings to instable classification techniques will provide different results.

Like any form of classifier optimization, automated parameter optimization may increase the risk of *overfitting*, i.e., producing a classifier that is too specialized for the data from which it was trained to apply to other datasets. To investigate whether parameter optimization is impacting the stability of defect prediction models, we formulate the following research questions:

(RQ2) How stable is the performance of defect prediction models when automated parameter optimization is applied?

4. CASE STUDY APPROACH

In this section, we discuss our selection criteria for the studied systems and then describe the design of our case study experiment that we perform in order to address our research questions.

4.1 Studied Datasets

In selecting the studied datasets, we identified three important criteria that needed to be satisfied:

- **Criterion 1 — Publicly-available defect datasets from different corpora**: Our recent work shows that researchers tend to reuse experimental components (e.g., datasets, metrics, and classifiers) [56]. Song *et al.* [52] and Ghotra *et al.* [12] also show that the performance of defect prediction models can be impacted by the dataset from which they are trained. To combat potential bias in our conclusions and to foster replication of our experiments, we choose to train our defect prediction models using datasets from different corpora and domains that are hosted in publicly-available data repositories.

- **Criterion 2 — Dataset robustness**: Mende *et al.* [36] show that models that are trained using small datasets may produce unstable performance estimates. An influential characteristic in the performance of a classification technique is the number of *Events Per Variable* (EPV) [45, 55], i.e., the ratio of the number of occurrences of the least frequently occurring class of the dependent variable (i.e., the events) to the number of independent variables that are used to train the model (i.e., the variables). Our recent work shows that defect prediction models that are trained using datasets with a low EPV value are especially susceptible to unstable results [55]. To mitigate this risk, we choose to study datasets that have an EPV above 10, as suggested by Peduzzi *et al.* [45].

- **Criterion 3 — Sane defect data**: Since it is unlikely that more software modules have defects than are free of defects, we choose to study datasets that have a rate of defective modules below 50%.

To satisfy criterion 1, we began our study using 101 publicly-available defect datasets. 76 datasets are downloaded from the Tera-PROMISE Repository,[1] 12 clean NASA datasets are provided by Shepperd *et al.* [50], 5 are provided by Kim *et al.* [26, 61], 5 are provided by D'Ambros *et al.* [6, 7], and 3 are provided by Zimmermann *et al.* [64]. To satisfy criterion 2, we exclude the 78 datasets that we found to have EPV values below 10. To satisfy criterion 3, we exclude an additional 5 datasets because they have a defective rate above 50%.

Table 2 provides an overview of the 18 datasets that satisfy our criteria for analysis. To strengthen the generalizability of our results, the studied datasets include proprietary and open source systems of varying size and domain.

Figure 1 provides an overview of the approach that we apply to each studied system. We describe each step in the approach below.

Table 2: An overview of the studied systems.

Domain	System	Defective Rate	#Files	#Metrics	EPV
NASA	JM1[1]	21%	7,782	21	80
	PC5[1]	28%	1,711	38	12
Proprietary	Prop-1[2]	15%	18,471	20	137
	Prop-2[2]	11%	23,014	20	122
	Prop-3[2]	11%	10,274	20	59
	Prop-4[2]	10%	8,718	20	42
	Prop-5[2]	15%	8,516	20	65
Apache	Camel 1.2[2]	36%	608	20	11
	Xalan 2.5[2]	48%	803	20	19
	Xalan 2.6[2]	46%	885	20	21
Eclipse	Platform 2.0[3]	14%	6,729	32	30
	Platform 2.1[3]	11%	7,888	32	27
	Platform 3.0[3]	15%	10,593	32	49
	Debug 3.4[4]	25%	1,065	17	15
	SWT 3.4[4]	44%	1,485	17	38
	JDT[5]	21%	997	15	14
	Mylyn[5]	13%	1,862	15	16
	PDE[5]	14%	1,497	15	14

[1] Provided by Shepperd *et al.* [50].
[2] Provided by Jureczko *et al.* [24].
[3] Provided by Zimmermann *et al.* [62].
[4] Provided by Kim *et al.* [26, 61].
[5] Provided by Ambros *et al.* [6].

4.2 Generate Bootstrap Sample

In order to ensure that the conclusions that we draw about our models are robust, we use the out-of-sample bootstrap validation technique [8, 55], which leverages aspects of statistical inference [9]. The out-of-sample bootstrap is made up of two steps:

(Step 1) A bootstrap sample of size N is randomly drawn with replacement from an original dataset, which is also of size N.

(Step 2) A model is trained using the bootstrap sample and tested using the rows that do not appear in the bootstrap sample. On average, 36.8% of the rows will not appear in the bootstrap sample, since it is drawn with replacement [8].

The out-of-sample bootstrap process is repeated 100 times, and the average out-of-sample performance is reported as the performance estimate.

Unlike the ordinary bootstrap, the out-of-sample bootstrap technique fits models using the bootstrap samples, but rather than testing the model on the original sample, the model is instead tested using the rows that do not appear in the bootstrap sample [55]. Thus, the training and testing corpus do not share overlapping observations.

Unlike k-fold cross-validation, the out-of-sample bootstrap technique fits models using a dataset that is of equal length to the original dataset. Cross-validation splits the data into k equal parts, using k - 1 parts for fitting the model, setting aside 1 fold for testing. The process is repeated k times, using a different part for testing each time. However, Mende *et al.* [36] point out that the scarcity of defective modules in the small testing corpora of 10-fold cross validation may produce biased and unstable results. Prior studies have also shown that 10-fold cross validation can produce unstable results for small samples [3]. On the other hand, our recent research demonstrates that the out-of-sample bootstrap tends to produce the least biased and most stable performance estimates [55]. Moreover, the use of out-of-sample bootstrap is recommended for high-skewed datasets [17], as is the case in our defect prediction datasets.

[1] http://openscience.us/repo/

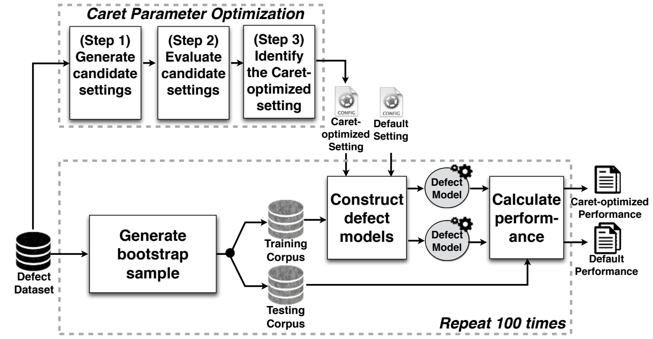
Caret Parameter Optimization
(Step 1) Generate candidate settings
(Step 2) Evaluate candidate settings
(Step 3) Identify the Caret-optimized setting
CONFIG
Caret-optimized Setting
CONFIG
Default Setting
Defect Dataset
Generate bootstrap sample
Training Corpus
Testing Corpus
Construct defect models
Defect Model
Defect Model
Calculate performance
Caret-optimized Performance
Default Performance
Repeat 100 times

Figure 1: An overview of our case study approach.

4.3 Caret Parameter Optimization

Since it is impractical to assess all of the possible parameter settings of the parameter spaces, we use the optimized parameter settings suggested by the `train` function of the `caret` R package [30]. Caret suggests candidate settings for each of the studied classification techniques, which can be checked using the `getModelInfo` function of the `caret` R package [30]. The optimization process is made up of three steps.

(Step 1) Generate candidate parameter settings: The `train` function will generate candidate parameter settings based on a given budget threshold (i.e., tune length) for evaluation. The budget threshold indicates the number of different values to be evaluated for each parameter. As suggested by Kuhn [28], we use a budget threshold of 5. For example, the number of boosting iterations of the C5.0 classification technique is initialized to 1 and is increased by 10 until the number of candidate settings reaches the budget threshold (e.g., 1, 10, 20, 30, 40). Table 1 shows the candidate parameter settings for each of the studied parameters. The default settings are shown in bold typeface.

(Step 2) Evaluate candidate parameter settings: Caret evaluates all of the potential combinations of the candidate parameter settings. For example, if a classification technique accepts 2 parameters with 5 candidate parameter settings for each, Caret will explore all 25 potential combinations of parameter settings (unless the budget is exceeded). We use 100 repetitions of the out-of-sample bootstrap to estimate the performance of classifiers that are trained using each of the candidate parameter settings. For each candidate parameter setting, a classifier is fit to a subsample of the training corpus and we estimate the performance of a model using those rows in the training corpus that do not appear in the subsample that was used to trained the classifier.

(Step 3) Identify the Caret-optimized setting: Finally, the performance estimates are used to identify which parameter settings are the most optimal. The Caret-optimized setting is the one that achieves the highest performance estimate.

4.4 Construct Defect Models

In order to measure the impact that automated parameter optimization has on defect prediction models, we train defect models using the Caret-optimized settings and the default settings. To ensure that the training and testing corpora have similar characteristics, we do not re-balance or re-sample the training data, as suggested by Turhan [59].

Normality Adjustment. Analysis of the distributions of our independent variables reveals that they are right-skewed. As suggested by previous research [22], we mitigate this skew by log-transforming each independent variable ($ln(x+1)$) prior to using them to train our models.

4.5 Calculate Performance

Prior studies have argued that threshold-dependent performance metrics (i.e., precision and recall) are problematic because they: (1) depend on an arbitrarily-selected threshold [32, 47] and (2) are sensitive to imbalanced data [18]. Instead, we use the *Area Under the receiver operator characteristic Curve (AUC)* to measure the discrimination power of our models as suggested by recent research [32].

The AUC is a threshold-independent performance metric that measures a classifier's ability to discriminate between defective and clean modules (i.e., do the defective modules tend to have higher predicted probabilities than clean modules?). AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine whether a file is classified as defective or not. Values of AUC range between 0 (worst performance), 0.5 (random guessing performance), and 1 (best performance).

5. CASE STUDY RESULTS

In this section, we present the results of our case study with respect to our two research questions.

(RQ1) How much does the performance of defect prediction models improve when automated parameter optimization is applied?

Approach. To address RQ1, we start with the AUC performance distribution of the 26 classification techniques that require at least one parameter setting (see Section 2). For each classification technique, we compute the difference in the performance of classifiers that are trained using default and Caret-optimized parameter settings. We then use boxplots to present the distribution of the performance difference for each of the 18 studied datasets. To quantify the magnitude of the performance improvement, we use Cohen's d effect size [4], which is the difference between the two means divided by the standard deviation of the data ($d = \frac{\bar{x}_1 - \bar{x}_2}{s.d.}$). The magnitude is assessed using the thresholds provided by Cohen [5]:

$$\text{effect size} = \begin{cases} \textit{negligible} & \text{if Cohen's } d \leq 0.2 \\ \textit{small} & \text{if } 0.2 < \text{Cohen's } d \leq 0.5 \\ \textit{medium} & \text{if } 0.5 < \text{Cohen's } d \leq 0.8 \\ \textit{large} & \text{if } 0.8 < \text{Cohen's } d \end{cases}$$

Furthermore, understanding the most influential parameters would allow researchers to focus their optimization effort. To this end, we investigate the performance difference

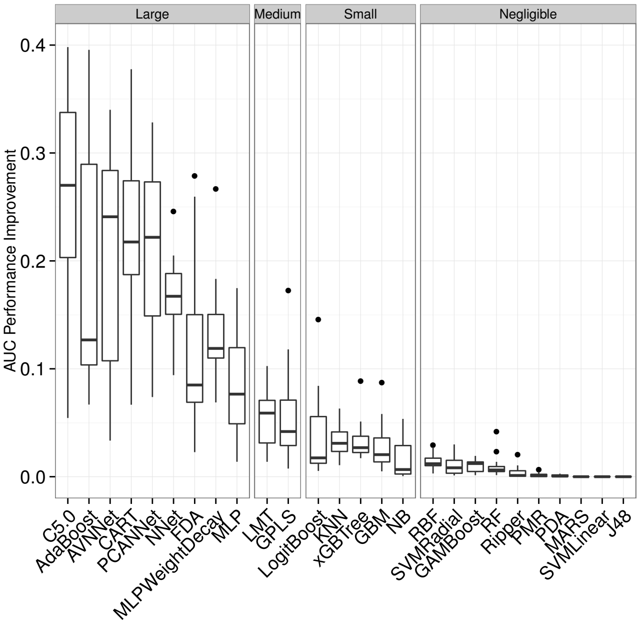

Figure 2: The performance improvement and its Cohen's d effect size for each of the studied classification techniques.

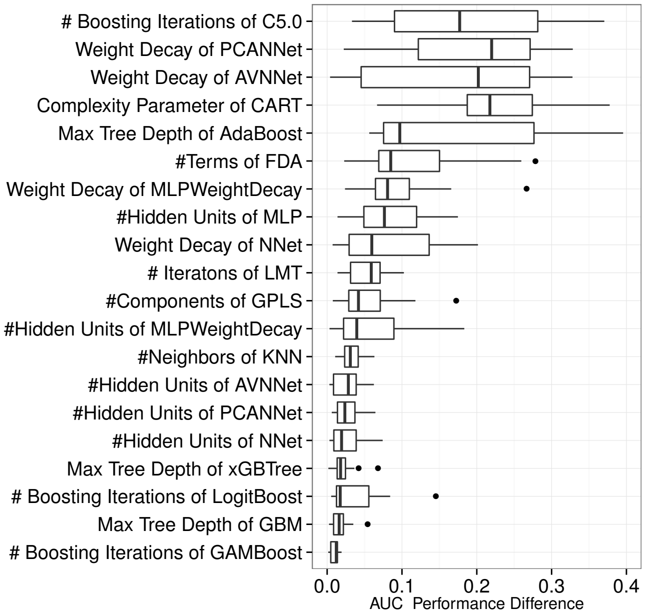

Figure 3: The AUC performance difference of the top-20 most sensitive parameters.

for each of the studied parameters. To quantify the individual impact of each parameter, we train a classifier with all of the studied parameters set to their default settings, except for the parameter whose impact we want to measure, which is set to its Caret-optimized setting. We estimate the impact of each parameter using the difference of its performance with respect to a classifier that is trained entirely using default parameter settings.

Results. **Caret improves the AUC performance by up to 40 percentage points.** Figure 2 shows the performance improvement for each of the 18 studied datasets and for each of the classification techniques. The boxplots show that Caret can improve the AUC performance by up to 40 percentage points. Moreover, the performance improvement provided by applying Caret is non-negligible (i.e., $d > 0.2$) for 16 of the 26 studied classification techniques (62%). This indicates that parameter settings can substantially influence the performance of defect prediction models.

C5.0 boosting yields the largest performance improvement when Caret is applied. According to Cohen's d, the performance improvement provided by applying Caret is large for 9 of the 26 studied classification techniques (35%). On average, Figure 2 shows that the C5.0 boosting classification technique benefits most by applying Caret, with a median performance improvement of 27 percentage points. Indeed, the C5.0 boosting classification technique improves from 6 to 40 percentage points.

Moreover, Figure 3 shows that the `#boosting iterations` parameter of the C5.0 classification technique is the most influential parameter, while the `winnow` and `model type` parameters tend to have less of an impact. Indeed, the default `#boosting iterations` setting that is provided by the `C5.0` R package [29] is 1, indicating that only one C5.0 tree model is used for prediction. Moreover, we find that, when large datasets of more than 1,000 modules are analyzed, the performance of C5.0 boosting with the default setting tends to underperform. Nevertheless, we find that the optimal `#boosting iterations` parameter is 40, suggesting that the default parameter settings of the research toolkits are suboptimal for defect prediction datasets, which agrees with the suspicion of prior studies [15, 22, 51, 58].

In addition to C5.0 boosting, other classifiers also yield a considerably large benefit. Figure 2 shows that the performance of the adaptive boosting (i.e., AdaBoost), advanced neural networks (i.e., AVNNet, PCANNet, NNet, MLP, and MLPWeightDecay), CART, and flexible discriminant analysis (FDA) classification techniques also have a large effect size with a median performance improvement from 13-24 percentage points. Indeed, Figure 3 shows that the fluctuation of the performance of the advanced neural network techniques is largely caused by changing the `weight decay`, but not the `#hidden units` or `bagging` parameters. Moreover, the `complexity` parameter of CART and `max tree depth` of adaptive boosting classification techniques are also sensitive to parameter optimization.

Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

(RQ2) How stable is the performance of defect prediction models when automated parameter optimization is applied?

Approach. To address RQ2, we start with the AUC performance distribution of the 26 studied classification techniques on each of the 18 studied datasets. The stability of a classification technique is measured in terms of the variability of the performance estimates that are produced by the 100 iterations of the out-of-sample bootstrap. For each classification technique, we compute the standard deviation

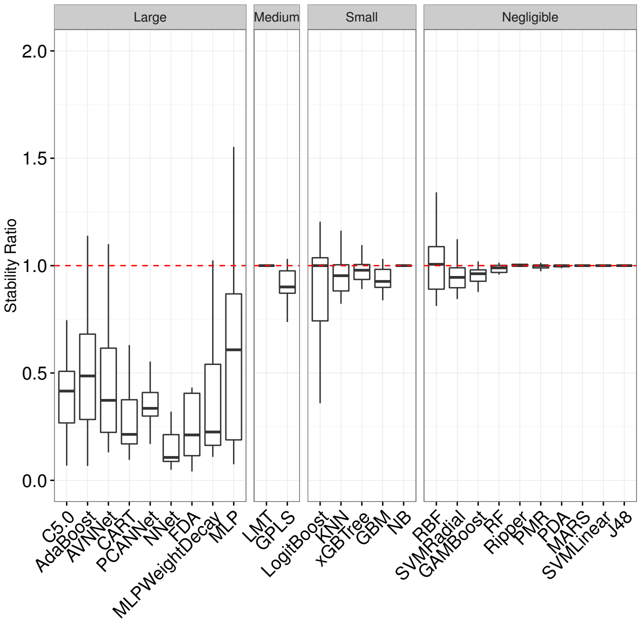

Figure 4: The stability ratio of the classifiers that are trained using Caret-optimized settings compared to the classifiers that are trained using default settings for each of the studied classification techniques.

(S.D.) of the bootstrap performance estimates of the classifiers where Caret-optimized settings have been used and the S.D. of the bootstrap performance estimates of the classifiers where the default settings have been used. To analyze the difference of the stability between two classifiers, we present distribution of the stability ratio (i.e., S.D. of the optimized classifier divided by the S.D. of the default classifier) of the two classifiers when apply to 18 studied datasets.

Similar to RQ1, we analyze the parameters that have the largest impact on the stability of the performance estimates. To this end, we investigate the stability ratio for each of the studied parameters. To quantify the individual impact of each parameter, we train a classifier with all of the studied parameters set to their default settings, except for the parameter whose impact we want to measure, which is set to its Caret-optimized setting. We estimate the impact of each parameter using the stability ratio of its S.D. of performance estimates with respect to a classifier that is trained entirely using default settings.

Results. **Caret-optimized classifiers are at least as stable as classifiers that are trained using the default settings.** Figure 4 shows that there is a median stability ratio of at least one for all of the studied classification techniques. Indeed, we find that the median ratio of one tends to appear for the classification techniques that yield negligible performance improvements in RQ1. These tight stability ratio ranges that are centered at one indicate that the stability of classifiers is not typically impacted by Caret-optimized settings.

Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are more stable than classifiers that are trained using the default values. Indeed, Figure 4 shows that there is a median stability ratio of 0.11 (NNet) to 0.61 (MLP) among the 9 classification techniques where the stability has improved. This equates to a 39%-89% stability improvement for these Caret-optimized classifiers. Indeed, Figure 5 shows that the stability of the performance of the advanced neural network techniques is largely caused by changing the `weight decay`, but not the `#hidden units` or `bagging` parameters, which consistent with our findings in RQ1.

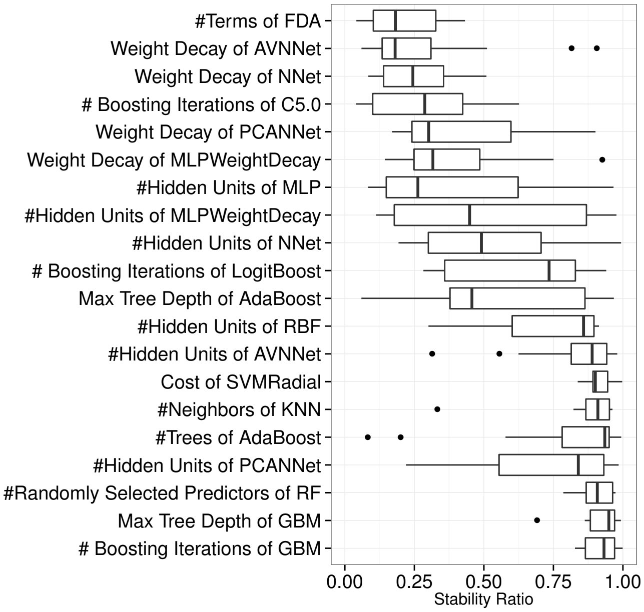

Figure 5: The stability ratio of the top-20 most sensitive parameters.

Caret-optimized classifiers are at least as stable as classifiers that are trained using the default settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are more stable than classifiers that are trained using the default values.

6. REVISITING THE RANKING OF CLASSIFICATION TECHNIQUES FOR DEFECT PREDICTION MODELS

Prior studies have ranked classification techniques according to their performance on defect prediction datasets. For example, Lessmann *et al.* [32] demonstrate that 17 of 22 studied classification techniques are statistically indistinguishable. On the other hand, Ghotra *et al.* [12] argue that classification techniques can have a large impact on the performance of defect prediction models.

However, these studies have not taken parameter optimization into account. Since we find that parameter settings can improve the performance of the classifiers that are produced (see RQ1), we set out to revisit the findings of prior studies when Caret-optimized settings have been applied.

6.1 Approach

As Keung *et al.* [25] point out, dataset selection can be a source of bias in an analysis of top-performing classification techniques. To combat the bias that may be introduced by

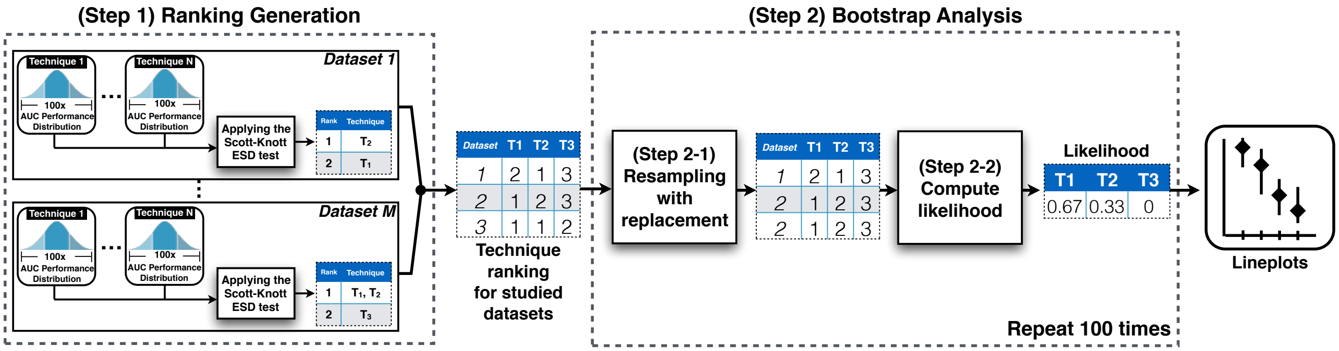
(Step 1) Ranking Generation
Dataset 1
Technique 1
Technique N
100x
AUC Performance Distribution
Applying the Scott-Knott ESD test
Rank
Technique
1
T₂
2
T₁
Dataset M
1
T₁, T₂
2
T₃
Dataset T1 T2 T3
1 2 1 3
2 1 2 3
3 1 1 2
Technique ranking for studied datasets
(Step 2) Bootstrap Analysis
(Step 2-1) Resampling with replacement
Dataset T1 T2 T3
1 2 1 3
2 1 2 3
2 1 2 3
(Step 2-2) Compute likelihood
Likelihood
T1 T2 T3
0.67 0.33 0
Repeat 100 times
Lineplots

Figure 6: An overview of our statistical comparison over multiple datasets.

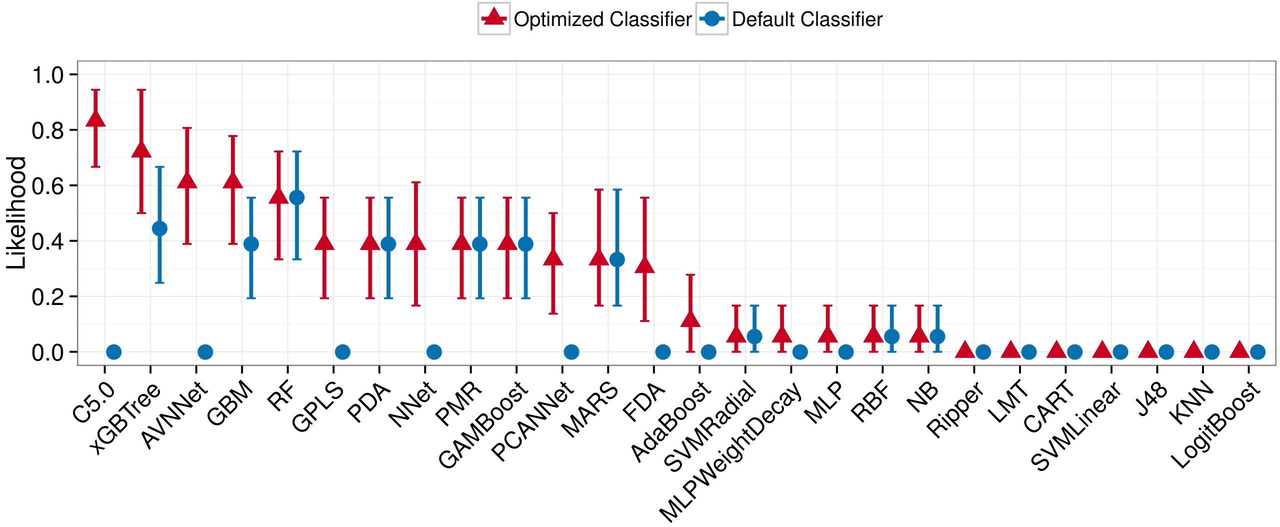

Figure 7: The likelihood of each technique appearing in the top Scott-Knott ESD rank. Circle dots and triangle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval of the likelihood of the bootstrap analysis. A likelihood of 80% indicates that a classification technique appears at the top-rank for 80% of the studied datasets.

dataset selection, we use a bootstrap-based Ranking Likelihood Estimation (RLE) experiment. Figure 6 provides an overview of our RLE experiment. The experiment uses a statistical comparison approach over multiple datasets that leverages both effect size differences and aspects of statistical inference [9]. The experiment is divided into two steps that we describe below.

(Step 1) Ranking Generation. We first start with the AUC performance distribution of the 26 studied classification techniques with the Caret-optimized parameter settings and the default settings. To find statistically distinct ranks of classification techniques within each dataset, we provide the AUC performance distribution of the 100 bootstrap iterations of each classification technique with both parameter settings to a Scott-Knott Effect Size Difference (ESD) test ($\alpha = 0.05$) [55]. The Scott-Knott ESD test is a variant of the Scott-Knott test that is effect size aware. The Scott-Knott ESD test uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups.

Unlike the traditional Scott-Knott test [19], the Scott-Knott ESD test will merge any two statistically distinct groups that have a negligible Cohen's d effect size [4] into one group. The Scott-Knott ESD test also overcomes the confounding issue of overlapping groups that are produced by several other post-hoc tests [12, 41], such as Nemenyi's test [43], which were used in prior studies [32]. We implement the Scott-Knott ESD test based on the implementation of the Scott-Knott test provided by the `ScottKnott` R package [19] and the implementation of Cohen's d provided by the `effsize` R package [57].

We use the Scott-Knott ESD test in order to control for dataset-specific model performance, since some datasets may have a tendency to produce over- or under-performing classifiers. Finally, for each classification technique, we have 18 different Scott-Knott ranks (i.e., one from each dataset).

(Step 2) Bootstrap Analysis. We then perform a bootstrap analysis to approximate the empirical distribution of the likelihood that a technique will appear in the top Scott-Knott ESD rank [8]. The key intuition is that the relationship between the likelihood that is derived from studied datasets and the true likelihood that would be derived from the population of defect datasets is asymptotically equivalent to the relationship between the likelihood that is derived from bootstrap samples and the likelihood that is derived from studied datasets. We first input the ranking of the studied classification techniques on 18 studied datasets to the bootstrap analysis, which is comprised of two steps:

(Step 2-1) A bootstrap sample of 18 datasets is randomly drawn with replacement from the ranking table, which is also of comprised of size 18 studied datasets.

(Step 2-2) For each classification technique, we compute the likelihood that a technique appears in the top Scott-Knott ESD rank in the bootstrap sample.

The bootstrap analysis is repeated 100 times. We then present the results with its 95% confidence interval, which is derived from the bootstrap analysis.

6.2 Results

C5.0 boosting tends to yield top-performing defect prediction models more frequently than the other studied classification techniques. Figure 7 shows the likelihood of each technique appearing in the top Scott-Knott ESD rank. We find that there is a 83% likelihood of C5.0 appearing in the top Scott-Knott rank. Furthermore, the bootstrap-derived 95% confidence interval ranges from 67% to 94%. On the other hand, when default settings are applied, C5.0 boosting has a 0% likelihood of appearing in the top rank. This echoes the findings of RQ1, where C5.0 boosting was found to be the classification technique that is most sensitive to parameter optimization.

Unlike prior work in the data mining domain, we find that random forest is not the most frequent top performer in our defect prediction datasets. Indeed, we find that there is a 55% likelihood of random forest appearing in the top Scott-Knott rank with a bootstrap-derived 95% confidence interval that ranges from 33% to 72%. A one-tailed bootstrap t-test reveals that the likelihood of C5.0 producing a top performing classifier is significantly larger than the likelihood of random forest producing a top-performing classifier ($\alpha = 0.05$). This contradicts the conclusions of Fernandez-Delgado *et al.* [10], who found that random forest tends to yield top-performing classifiers the most frequently. The contradictory conclusions indicate that the domain-specifics play an important role.

Automated parameter optimization increases the likelihood of appearing in the top Scott-Knott ESD rank by as much as 83%. Figure 7 shows that automated parameter optimization increases the likelihood of 11 of the studied 26 classification techniques by as much as 83% (i.e., C5.0 boosting). This suggests that automated parameter optimization can substantially shift the ranking of classification techniques.

C5.0 boosting tends to yield top-performing defect prediction models more frequently than the other studied classification techniques. This disagrees with prior studies in the data mining domain, suggesting that domain-specifics play a key role. Furthermore, automated parameter optimization increases the likelihood of appearing in the top Scott-Knott ESD rank by as much as 83%.

7. DISCUSSION

7.1 Cross-Context Defect Prediction

The performance improvement of defect prediction models is estimated using a bootstrap resampling approach (see RQ1). While this bootstrap resampling approach is common in other research areas [3, 8, 9], recent studies in software engineering tend to estimate the performance of defect models using data from different contexts [63]. Hence, we perform an additional analysis in order to investigate whether Caret still improves AUC performance in a cross-context setting. We analyze the performance of defect prediction models that are trained in one context, but tested in another context. We then compute the performance improvement between the models that are trained with Caret-optimal and default settings.

Caret still improves the performance of cross-context defect prediction models by up to 30 percentage points. Based on an analysis of 5 releases of proprietary systems, 2 releases of Apache Xalan, and 3 releases of Eclipse Platform, we find that the performance of cross-context classifiers that are trained using Caret outperform classifiers that are trained using default settings. For example, we find that when neural network classifiers are trained using Eclipse Platform 2.1 and tested using Eclipse Platform 3.0, the AUC performance improves by 30 percentage points when compared to classifiers that are trained using default settings. This suggests that automated parameter optimization also yields a large benefit in terms of cross-context defect prediction.

7.2 Computational Cost

Our case study approach is computationally-intensive (i.e., 450 parameter settings × 100 out-of-sample bootstrap repetitions × 18 systems = 810,000 results). However, the results can be computed in parallel. Hence, we design our experiment using a High Performance Computing (HPC) environment. Our experiments are performed on 43 high performance computing machines with 2x Intel Xeon 5675 @3.1 GHz (24 hyper-threads) and 64 GB memory (i.e., in total, 24 hyper-threads × 43 machines = 1,032 hyper-threads). Each machine connects to a 2 petabyte shared storage array via a dual 10-gigabit fibre-channel connection.

For each of the classification techniques, we compute the average amount of execution time that was consumed by Caret when producing suggested parameter settings for each of the studied datasets.

Caret adds less than 30 minutes of additional computation time to 65% of the studied classification techniques. We find that the optimization cost of 17 of the 26 studied classification techniques (65%) is less than 30 minutes. We find that the C5.0 and extreme gradient boosting classification techniques, which yield top-performing classifiers more frequently than other classification techniques, fall into this category. This indicates that applying Caret tends to improve the performance of defect models while incurring a manageable additional computational cost.

On the other hand, 12% of the studied classification techniques require more than 3 additional hours of computation time to apply Caret. Only AdaBoost, MLPWeightDecay, and RBF incur this large overhead. Nonetheless, the computation could still be completed if it was run overnight. Since defect prediction models do not need to be built very often in practice, this cost should still be manageable.

8. THREATS TO VALIDITY

We now discuss the threats to the validity of our study.

8.1 Construct Validity

The datasets that we analyze are part of several collections (e.g., NASA and PROMISE), which each provide different sets of metrics. Since the metrics vary, this is a point of variation between the studied systems that could impact our results. However, our within-family datasets analysis shows that the number and type of predictors do not influence our findings. Thus, we conclude that the variation of metrics does not pose a threat to our study. On the other hand, the variety of metrics also strengthens the generalization of our results, i.e., our findings are not bound to one specific set of metrics.

The Caret budget, which controls the number of settings that we evaluate for each parameter, limits our exploration of the parameter space. Although our budget setting is selected based on the literature [30], selecting a different budget may yield different results. However, the results of our study show that a modest exploration of the parameter space can already lead to a large change in the performance of defect prediction models.

Our results from RQ1 show that Caret improves the performance of defect prediction models. However, the performance improvement may increase the complexity of defect prediction models. Thus, we plan to investigate the relationship between model complexity and performance in future work.

8.2 Internal Validity

We measure the performance of our classifiers using AUC. Other performance measures may yield different results. We plan to expand the set of measures that we adopt in our future work.

The generalizability of the bootstrap-based Ranking Likelihood Estimation (RLE) is dependent on how representative our sample is. To combat potential bias in our samples, we analyze datasets of different sizes and domains. Nonetheless, a larger sample may yield more robust results.

Prior work has shown that dirty data may influence conclusion that are drawn from defect prediction studies [12, 53, 54]. Hence, noisy data may be influencing our conclusions. However, we conduct a highly-controlled experiment where known-to-be noisy NASA data [50] has been cleaned. Nonetheless, dataset cleanliness should be inspected in future work.

8.3 External Validity

We study a limited number of systems in this paper. Thus, our results may not generalize to all software systems. However, the goal of this paper is not to show a result that generalizes to all datasets, but rather to show that there are datasets where parameter optimization matters. Nonetheless, additional replication studies may prove fruitful.

9. CONCLUSIONS

Defect prediction models are classifiers that are trained to identify defect-prone software modules. The characteristics of the classifiers that are produced are controlled by configurable parameters. Recent studies point out that classifiers may under-perform because they were trained using suboptimal default parameter settings. However, it is impractical to explore all of the possible settings in the parameter space of a classification technique.

In this paper, we investigate the performance of defect prediction models where Caret [30] — an automated parameter optimization technique — has been applied. Through a case study of 18 datasets from systems that span both proprietary and open source domains, we make the following observations:

- Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

- Caret-optimized classifiers are at least as stable as classifiers that are trained using the default settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are more stable than classifiers that are trained using the default values.

- Caret increases the likelihood of producing a top-performing classifier by as much as 83%, suggesting that automated parameter optimization can substantially shift the ranking of classification techniques.

Our results lead us to conclude that parameter settings can indeed have a large impact on the performance of defect prediction models, suggesting that researchers should experiment with the parameters of the classification techniques. Since automated parameter optimization techniques like Caret yield substantial benefits in terms of performance improvement and stability, while incurring a manageable additional computational cost, they should be included in future defect prediction studies.

Finally, we would like to emphasize that we do not seek to claim the generalization of our results. Instead, the key message of our study is that there are datasets where there are statistically significant differences between the performance of classification techniques that are trained using default and Caret-optimized parameter settings. Hence, we recommend that software engineering researchers experiment with automated parameter optimization (e.g., Caret) instead of relying on the default parameter setting of the research toolkits, assuming that other parameter settings are not likely to lead to significant improvements. Given the availability of automated parameter optimization in commonly-used research toolkits (e.g., Caret for R [30], MultiSearch for Weka [14], GridSearch for Scikit-learn [44]), we believe that our recommendation is a rather simple and low-cost recommendation to adopt.

Acknowledgments. This study would not have been possible without the data shared in the Tera-PROMISE repository [39], Shepperd *et al.* [50], Zimmermann *et al.* [64], Kim *et al.* [26, 61], and D'Ambros *et al.* [6, 7], as well as High Performance Computing (HPC) systems provided by the Compute Canada[2] and HPCVL.[3] This work was supported by the JSPS Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers: Interdisciplinary Global Networks for Accelerating Theory and Practice in Software Ecosystem, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

[2] https://www.computecanada.ca/

[3] http://www.hpcvl.org/

10. REFERENCES

[1] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[2] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas. Software Defect Prediction Using Regression via Classification. In *Proceedings of the International Conference on Computer Systems and Applications*, pages 330–336, 2006.

[3] U. M. Braga-Neto and E. R. Dougherty. Is cross-validation valid for small-sample microarray classification? *Bioinformatics*, 20(3):374–380, 2004.

[4] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* 1988.

[5] J. Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[6] M. D'Ambros, M. Lanza, and R. Robbes. An Extensive Comparison of Bug Prediction Approaches. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 31–41, 2010.

[7] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[8] B. Efron. Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[9] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap.* CRC Press, 1993.

[10] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[11] S. Fritsch and F. Guenther. neuralnet: Training of neural networks. http://CRAN.R-project.org/package=neuralnet, 2015.

[12] 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 International Conference on Software Engineering*, pages 789–800, 2015.

[13] A. Günes Koru and H. Liu. An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, pages 1–5, 2005.

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

[15] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *Transactions on Software Engineering*, 38(6):1276–1304, nov 2012.

[16] M. Harman, P. McMinn, J. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. pages 1–59, 2012.

[17] F. E. Harrell Jr. *Regression Modeling Strategies.* Springer, 1st edition, 2002.

[18] H. He and E. A. Garcia. Learning from Imbalanced Data. *Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[19] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman. *The ScottKnott Clustering Algorithm.* Universidade Estadual de Santa Cruz - UESC, Ilheus, Bahia, Brasil, 2014.

[20] Y. Jia, M. Cohen, and M. Petke. Learning Combinatorial Interaction Test Generation Strategies using Hyperheuristic Search. In *Proceedings of the International Conference on Software Engineering*, pages 540–550, 2015.

[21] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

[22] Y. Jiang, B. Cukic, and T. Menzies. Can Data Transformation Help in the Detection of Fault-prone Modules? In *Proceedings of the workshop on Defects in Large Software Systems*, pages 16–20, 2008.

[23] M. Jorgensen and M. Shepperd. A Systematic Review of Software Development Cost Estimation Studies. *Transactions on Software Engineering*, 33(1):33–53, 2007.

[24] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, pages 9:1–9:10, 2010.

[25] J. Keung, E. Kocaguneli, and T. Menzies. Finding conclusion stability for selecting the best effort predictor in software effort estimation. *Automated Software Engineering*, 20(4):543–567, 2013.

[26] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with Noise in Defect Prediction. In *Proceedings of the International Conference on Software Engineering*, pages 481–490, 2011.

[27] E. Kocaguneli, T. Menzies, A. B. Bener, and J. W. Keung. Exploiting the essential assumptions of analogy-based effort estimation. *Transactions on Software Engineering*, 38(2):425–438, 2012.

[28] M. Kuhn. Building Predictive Models in R Using caret Package. *Journal of Statistical Software*, 28(5), 2008.

[29] M. Kuhn. C50: C5.0 decision trees and rule-based models. http://CRAN.R-project.org/package=C50, 2015.

[30] M. Kuhn. caret: Classification and regression training. http://CRAN.R-project.org/package=caret, 2015.

[31] I. H. Laradji, M. Alshayeb, and L. Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.

[32] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction : A Proposed Framework and Novel Findings. *Transactions on Software Engineering*, 34(4):485–496, 2008.

[33] A. Liaw and M. Wiener. randomforest: Breiman and cutler's random forests for classification and regression. http://CRAN.R-project.org/package=randomForest, 2015.

[34] A. Lim, L. Breiman, and A. Cutler. bigrf: Big random forests: Classification and regression forests for large data sets. http://CRAN.R-project.org/package=bigrf,

2015.

[35] MATLAB. *version 8.5.0 (R2015a)*. The MathWorks Inc., Natick, Massachusetts, 2015.

[36] T. Mende. Replication of Defect Prediction Studies: Problems, Pitfalls and Recommendations. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.

[37] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, page 7, 2009.

[38] T. Mende, R. Koschke, and M. Leszak. Evaluating Defect Prediction Models for a Large Evolving Software System. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 247–250, 2009.

[39] T. Menzies, C. Pape, R. Krishna, and M. Rees-Jones. The Promise Repository of Empirical Software Engineering Data. http://openscience.us/repo, 2015.

[40] T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17(1-2):1–17, 2012.

[41] N. Mittas and L. Angelis. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *Transactions on Software Engineering*, 39(4):537–551, 2013.

[42] I. Myrtveit, E. Stensrud, and M. Shepperd. Reliability and Validity in Comparative Studies of Software Prediction Models. *Transactions on Software Engineering*, 31(5):380–391, 2005.

[43] P. Nemenyi. *Distribution-free multiple comparisons*. PhD thesis, Princeton University, 1963.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] P. Peduzzi, J. Concato, E. Kemper, T. R. Holford, and A. R. Feinstein. A Simulation Study of the Number of Events per Variable in Logistic Regression Analysis. *Journal of Clinical Epidemiology*, 49(12):1373–1379, 1996.

[46] R Core Team. R: A language and environment for statistical computing. http://www.R-project.org/, 2013.

[47] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the International Conference on Software Engineering*, pages 432–441, 2013.

[48] B. Ripley. nnet: Feed-forward neural networks and multinomial log-linear models. http://CRAN.R-project.org/package=nnet, 2015.

[49] M. Shepperd, D. Bowes, and T. Hall. Researcher Bias : the Use of Machine Learning in Software Defect Prediction. *Transactions on Software Engineering*, 40(6):603–616, 2014.

[50] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[51] L. Song, L. L. Minku, and X. Yao. The Impact of Parameter Tuning on Software Effort Estimation Using Learning Machines. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, pages 1–10, 2013.

[52] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A General Software Defect-Proneness Prediction Framework. *Transactions on Software Engineering*, 37(3):356–370, 2011.

[53] C. Tantithamthavorn. Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling. In *Proceedings of the International Conference on Software Engineering*, page To appear, 2016.

[54] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *Proceedings of the International Conference on Software Engineering*, pages 812–823, 2015.

[55] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An Empirical Comparison of Model Validation Techniques for Defect Prediction Model. Under Review at IEEE Transactions on Software Engineering (TSE), http://sailhome.cs.queensu.ca/replication/kla/model-validation.pdf, 2015.

[56] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Comments on "Researcher Bias: the Use of Machine Learning in Software Defect Prediction". *PeerJ PrePrints*, page 3:e1543, 2015.

[57] M. Torchiano. effsize: Efficient effect size computation. http://CRAN.R-project.org/package=effsize, 2015.

[58] A. Tosun and A. Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 477–480, 2009.

[59] B. Turhan. On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17(1-2):62–74, 2011.

[60] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Comparing negative binomial and recursive partitioning models for fault prediction. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, pages 3–9, 2008.

[61] R. Wu, H. Zhang, S. Kim, and S. C. Cheung. ReLink: Recovering Links between Bugs and Changes. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, pages 15–25, 2011.

[62] T. Zimmermann and N. Nagappan. Predicting Subsystem Failures using Dependency Graph Complexities. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 227–236, 2007.

[63] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction. In *Proceedings of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering*, pages 91–100, 2009.

[64] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, pages 9–20, 2007.