Predicting Software Defects with Explainable Machine Learning

Geanderson Santos
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
geanderson@dcc.ufmg.br

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
figueiredo@dcc.ufmg.br

Adriano Veloso
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
adrianov@dcc.ufmg.br

Markos Viggiato
University of Alberta
Edmonton, Alberta, Canada
viggiato@ualberta.ca

Nivio Ziviani
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
nivio@dcc.ufmg.br

ABSTRACT

Most software systems must evolve to cope with stakeholders' requirements and fix existing defects. Hence, software defect prediction represents an area of interest in both academia and the software industry. As a result, predicting software defects can help the development team to maintain substantial levels of software quality. For this reason, machine learning models have increased in popularity for software defect prediction and have demonstrated effectiveness in many scenarios. In this paper, we evaluate a machine learning approach for selecting features to predict software module defects. We use a tree boosting algorithm that receives as input a training set comprising records of software features encoding characteristics of each module and outputs whether the corresponding module is defective prone. For nine projects within the widely known NASA data program, we build prediction models from a set of easy-to-compute module features. We then sample this sizable model space by randomly selecting software features to compose each model. This significant number of models allows us to structure our work along model understandability and predictive accuracy. We argue that explaining model predictions is meaningful to provide information to developers on features related to each module defective-prone. We show that (i) features that contribute most to finding the best models may vary depending on the project, and (ii) effective models are highly understandable based on a survey with 40 developers.

CCS CONCEPTS

• **Computing methodologies** → *Cross-validation*; • **Software and its engineering**;

KEYWORDS

software defects, explainable models, NASA datasets, SHAP values

SBQS'20, December 1–4, 2020, São Luís, Brazil

ACM ISBN 978-1-4503-8923-5/20/12…$15.00
https://doi.org/10.1145/3439961.3439979

ACM Reference Format:
Geanderson Santos, Eduardo Figueiredo, Adriano Veloso, Markos Viggiato, and Nivio Ziviani. 2020. Predicting Software Defects with Explainable Machine Learning. In *19th Brazilian Symposium on Software Quality (SBQS'20), December 1–4, 2020, São Luís, Brazil*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3439961.3439979

1 INTRODUCTION

Software defect prediction is an area of interest in both academia and the software industry [1, 13, 25]. Hence, new machine learning techniques that can develop more accurate and efficient predictive models have emerged in the past few years [2, 3, 21]. Researches base these defective prediction models on learned features from either (i) source code and metadata information [8, 13, 15, 22, 33, 35, 40] or (ii) quality metrics used to specify software design complexity [42, 46]. Studies on features learned from software source code, and metadata information usually employ approaches based on deep neural networks [42, 46]. Studies that rely on software quality metrics use either code inspections and unit testing [8] or machine learning approaches, such as Support Vector Machines (SVM) [5, 11], Decision Trees [41], Naïve Bayes [39, 43], neural networks [38], or dictionary learning-based prediction [15].

Despite the high accuracy usually achieved by machine learning models, they are often overly complicated and may hinder the understandability of the model. In most cases, we usually cannot explain the prediction of any machine learning model, and we still need investigation regarding the explanation of model decisions that could help developers to reason on the rationale behind a machine learning model that predicts a software defect [13, 20]. Further, explaining model decisions is also beneficial, as it enables the proper understanding of the effects in software development costs and efforts during development. Therefore, predicting defects while understanding the predictors help organizations to reduce development and maintenance costs and to concentrate efforts on the most defect-prone parts of the system [1]. This finding is relevant as high-quality software development is expensive, and defect-fixing processes require a laborious effort from a company [51]. Hereafter, we use the terms "explainability" and "interpretability" to refer to the same concept due to the lack of consensus in the machine learning community about these definitions [14]. For this reason, we refer to these terms as "understandability".

Differently from previous works that tune a single defect prediction model [5, 8, 43], we perform an exploration of the model space, which results in hundreds of thousands of evaluated models

[30]. Specifically, we learned prediction models considering distinct combinations of McCabe and Halstead features (see Section 2.3.2) applied to modules from nine NASA projects [22]. Since we compose each sampled model of a specific set of features, the learned models correspond to a myriad of explanations for the software defect phenomenon. We compared the effectiveness of our models with other machine learning methods typically used in software defect prediction. As a result, on average, 3.5% of the randomly generated models (considering 2,097,152 models) show superior accuracy when compared with seven baselines. We showed that some features are more relevant for defect prediction, although the importance of these features varies within projects. For eight out of nine NASA projects considered in our study, we could learn models that achieved similar or superior effectiveness when compared with baseline models. These eight models that provide higher accuracy gains are also easily understandable, according to their SHAP (SHapley Addictive exPlanation) values. Finally, we investigate model understandability through a survey with 40 developers about the model prediction. We used the survey study to understand whether or not the explanations provided by SHAP could be useful for developers to understand what is behind a defect prediction. We conclude that, in most cases, developers can reason on the features causing defects in the modules.

We organize the rest of our paper as follows. In Section 2, we describe our study setup. Next, we present and discuss our results and their implications in Section 3. Section 4 discusses the threats to validity concerning our work. Then, Section 5 presents related work of learning to predict defects from source code and metadata information. Finally, Section 6 concludes our paper with insights for further explorations.

2 STUDY SETUP

2.1 Goal and Research Questions

This paper aims to evaluate a machine learning approach for selecting features to predict software module defects using the classic NASA datasets. To achieve this goal, we discuss the following research questions:

RQ1: How randomly sampling the model space compare to state-of-the-art baselines?
RQ2: What is the relationship between the number of features and prediction effectiveness?
RQ3: How high-quality model understandability is affected by prediction effectiveness?

2.2 Defect Prediction Definition

We can define the task to predict software defects as follows. The input of our model represents a training set, which requires a set of instances known as $< x, y >$. The x is a vector of software features ($x = \{x_1, x_2, \ldots, x_n\}$). Complementary, the y denotes the outcome of our model. More specifically, NASA datasets define a defect based on the following expression, where one or more errors change the status of a module to defective.

$$defective? = errorcount >= 1$$

The training set compounds a model that reports features to the corresponding outcome. The test set comprises records $< x, ? >$ for which only the module x is available, while the corresponding label y is unknown. Thus, to find the optimal machine learning model, we need to enumerate all the combinations of features to produce the models. Another solution is to sample the model space to build a machine learning model for each set of software features. Furthermore, we construct the model space by randomly selecting the software features that compose these models. We start by creating models with a single software feature until we use the entire pool of software features. In this manner, we end up selecting each software feature evenly from the software metrics.

2.3 Data

As previously stated, we use the datasets containing metrics computed from modules of nine NASA software projects. These datasets are subject to other research studies about defect prediction [1, 22–24]. The various NASA projects comprise a broad range of NASA systems. For instance, CM1 refers to spacecraft instruments; KC1, KC3, MC2 refer to storage management for grounded data; MW1 manages the data transactions; and PC1, PC2, PC3, PC4 refer to software for an earth-orbiting satellite. Therefore, we can compare our results to other predictive models. Table 1 exemplifies the data with the 21 features. For each module within a project, there is a value assigned to each feature. The average value for a project represents the sum of values assigned to each module divided by the number of modules within the project (Table 1). These average values differ between projects as observed in Table 1, e.g., the average BRANCH_COUNT (number of branches) for project CM1 is 12.98, and for the project, KC1 is 7.24. Table 1 also shows the percentual of defective modules in each project, and it is clear the imbalanced nature of the data, i.e., defective modules are heavily under-represented in comparison with non-defective modules. We removed repeated/duplicate data points to avoid identical modules in training and testing data.

2.3.1 Data Imbalance. The data considered in this work are highly imbalanced, where approximately 11% of software modules present defects, and nearly 89% of modules are clean (Table 1). For this reason, we could not naively evaluate our models with the NASA dataset without an extensive data exploration process [32]. As a result, we employed a technique known as Synthetic Minority Oversampling Technique (SMOTE) [34]. Therefore, by using the SMOTE technique, we balanced the data 50/50 (i.e., half of the modules have defects, and the other half represents clean modules). More details about this technique are available in the replication package[1].

2.3.2 Features for Defect Prediction. NASA datasets are mostly composed of either McCabe or Halstead software quality metrics. These classic measures are module-based features originally proposed to anticipate the complexity of a module and reason on the quality of a software [44]. Table 2 describes the 21 features used in our paper with its descriptions. Metrics starting with either McCabe or Halstead are specific to these metrics (lines 2 to 11 in the Table).

[1] https://github.com/anonymous-replication/replication-nasa

Table 1: Overview of NASA Data Program Metrics.

	Projects	CM1	KC1	KC3	MC2	MW1	PC1	PC2	PC3	PC4
	Programming Language	C	C++	Java	C	C	C	C	C	C
	Number of Modules	505	1,571	458	127	403	1,059	4,505	1,511	1,347
	Defective Modules	9.5%	20.31%	9.39%	34.65%	7.69%	7.18%	0.51%	10.59%	13.21%
1	BRANCH_COUNT	12.98	7.24	10.78	16.42	10.22	13.20	7.62	12.60	8.28
2	CYCLOMATIC_COMP.	7.30	4.13	6.32	8.90	5.99	7.41	4.39	6.99	4.75
3	DESIGN_COMP.	4.94	3.63	5.43	3.09	4.51	4.32	3.18	3.62	2.88
4	ESSENTIAL_COMP.	3.08	2.17	2.64	4.40	2.44	3.46	2.19	2.97	2.28
5	HALSTEAD_CONTENT	49.91	31.37	51.19	35.39	46.71	37.52	22.95	43.52	28.46
6	HALSTEAD_DIFFICULTY	20.00	10.36	17.39	27.67	12.28	20.34	14.38	18.30	18.09
7	HALSTEAD_LEVEL	0.08	0.19	0.07	0.08	0.12	0.08	0.11	0.08	0.11
8	HALSTEAD_EFFORT	49058	9248	30806	65064	11070	42547	12995	47008	21432
9	HALSTEAD_ERR._EST	0.42	0.15	0.37	0.40	0.20	0.32	0.14	0.35	0.20
10	HALST._LENGTH	196.55	82.00	185.86	204.97	106.20	157.45	79.46	162.21	110.89
11	HALST._PROG_TIME	2725.4	513.8	1711.4	3614.6	615.05	2363.7	721.94	2611.6	1190.6
12	HALSTEAD_VOLUME	1262.2	438.11	1118.2	1202.8	600.43	964.11	426.10	1036.2	596.65
13	LOC_BLANK	16.72	2.98	4.78	11.25	6.45	8.76	11.12	8.22	7.82
14	LOC_CODE_AND_COMM.	5.64	0.21	0.23	2.38	0.27	1.43	14.55	1.75	2.38
15	LOC_COMMENTS	17.42	1.65	2.45	13.78	5.40	5.80	5.74	5.73	5.49
16	LOC_EXECUTABLE	41.06	24.17	32.20	41.55	26.66	29.85	2.62	28.26	20.47
17	LOC_TOTAL	46.70	32.28	32.43	43.94	26.92	31.27	17.17	30.01	22.85
18	NUM_OPERANDS	76.73	31.16	68.65	86.79	46.78	68.43	32.45	72.68	42.72
19	NUM_OPERATORS	119.82	50.84	117.20	118.18	59.41	89.02	47.01	89.54	68.17
20	NUM_UNIQ._OPERAN.	35.34	15.07	29.16	23.94	25.86	27.20	12.70	27.77	14.40
21	NUM_UNIQ._OPERAT.	19.03	10.58	15.99	15.58	13.51	16.46	11.88	15.21	12.73

As stated in Table 2, features are also related to other code characteristics, such as the number of operands and operators (unique or not unique) and the different variations of lines of code (for instance, comments, executable code, and blank lines).

2.3.3 Baseline Models. To compare the effectiveness of our approach, we considered seven machine learning models previously applied in the literature: Logistic Regression (LR) [26, 29, 52]. Naive Bayes (NB) [13, 15, 50], K-Nearest Neighbor (NBB) [16, 40, 47], Neural Network (NN) [15, 48], Decision Trees (CART) [6, 15, 17], Support Vector Machine (SVM) [5, 11, 31], and Random Forest [8, 35, 49]. We also included XGBoost [21, 30] as a baseline method, as this algorithm offered unique opportunities to deal with the data [3]. Differently from our approach, all the baseline algorithms use the full set of features while learning their models. We found relevant hyper-parameters using an automated parameter optimization technique [18] with a 10-fold cross-validation technique assisted by the scikit-learn package in the Python programming language [19]. The results reported are the average of the ten runs using cross-validation, and to ensure their relevance, we assess the statistical significance of our measurements using a Scott-Knott Effect Estimation Size (ESD) test [36, 37]. Further, we evaluate the effectiveness of the considered models using the standard ROC Area Under the Curve (AUC) and F1 measure. The AUC is an estimate of the probability that a prediction model will rank a randomly chosen positive instance higher than a randomly chosen negative class instance. In our context, we consider a positive class, the ones that represent a defect, while the negative class instances are the ones that are clean (non-defective). Moreover, the F1 measure represents the harmonic mean between the precision and recall [10]. We also

Table 2: NASA Data Program Software Quality Features.

	metric	description
1	Branch count	Number of branches
2	McCabe's complexity	Number of independent paths
3	McCabe's design	Complexity of a module
4	McCabe's essential	Degree of structuredness
5	Halstead content	Independent complexity of a module
6	Halstead difficult	Difficult to handle the module
7	Halstead level	Inverse of the error proneness
8	Halstead effort	Estimated mental effort
9	Halstead error	Number of errors in module
10	Halstead length	Operators and operands numbers
11	Halstead time	Estimate time to develop module
12	Halstead volume	Bits required to execute the module
13	Blank lines	Number of blank lines
14	LOC and comments	Numbers of lines of code and comments
15	Lines of comments	Number of lines of comments
16	LOC	Total lines of code
17	LOC exucutable	Total lines of code executable
18	Operands	Total number of operands
19	Operators	Total number of operators
20	Unique operands	Number of unique operands
21	Unique operators	Number of unique operators

checked for highly correlated features to avoid the multicollinearity problem from both the baseline models and our approach. Thus, we removed software features with over 99% of the correlation to other features.

2.4 Feature Importance

Machine learning models that can effectively predict defects in source code are usually hard to understand. Thus, it is relevant to understanding why a model has made such predictions because it provides valuable insights into the nature of the defects [13, 20, 30]. As an example, if a developer knows that the size of a module is an essential feature that makes the module defect-prone, it may cause the developer to focus on the refactoring of that specific module. The typical approach to understanding such predictions bases on the calculation of the impact of each software feature. Therefore, a software feature is relevant for the prediction if permuting its values increases the error. Thus, the model relies on specific features for the defect prediction. Software features interact with each other in complex ways to build accurate machine learning models. One technique to calculate the complexity between these features is applying Shapley values [28]. These values can find a significant division that characterizes the features of importances distributions. Formally, the explanation model g is a linear function of binary variables:

$$g(z) = \phi_0 + \sum_{i=1}^{m} \phi_i \times z_i, \tag{1}$$

where ϕ_i for $i = 0, 1, \ldots, m$ are parameters called Shapley values, m is the number of simplified input features, $z_i = \{z_1, z_2, \ldots, z_m\}$ is a binary vector in simplified input space where $z \in \{0, 1\}^m$. Shapley values measure how each feature contributes to the prediction. In this case, how they contribute to predicting a software defect. In theory, these values are optimal and provide an accurate attribution value. Here, we apply the implementation of these values known as SHAP (SHapley Addictive exPlanation) [21]. This technique is an approximation of Shapley values to compute the importance of each software feature.

3 RESULTS

3.1 Competitiveness of Random Models

In this section, we compare distinct baseline models to predict defects in the target data [4, 5, 13, 15, 23, 23, 39, 48]. Thus, we want to estimate the predictive capability of the data, i.e., the data quality to predict defects in the selected projects. For this experimentation, we applied the same setup to the nine NASA projects [22]. Table 3 displays the results of the baseline models for each NASA dataset. We note that in eight of the considered datasets, the random search could find models that achieve superior predictive accuracy compared to the remaining models (for both AUC numbers and F1 measure). The random search was unsuccessful only for the MC2 dataset (AUC) and PC3 (F1) for which they could not find an XGBoost model that is as effective as the baseline model.

As our set of experiments using AUC and F1 are not statistically sound to evaluate the performance of the target baselines. This happens because it is hard to distinguish the performance achieved by these algorithms looking solely in Table 3. Thus, we apply a test

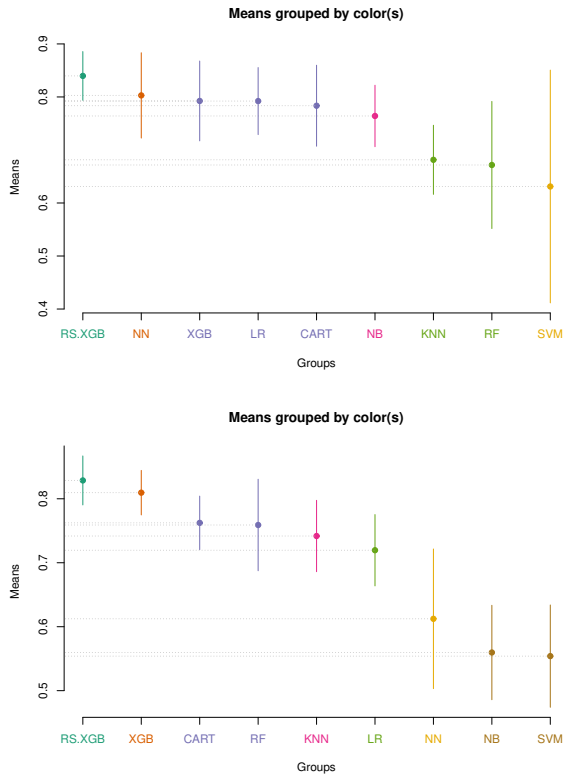

Figure 1: Scott-Knott Effect Size Estimation test. AUC Numbers (Above). F1 Measure (Below).

known as Scott-Knott Effect Size Difference (ESD) [36, 37]. The ESD test is a mean comparison approach that leverages a hierarchical clustering to partition the set of treatment means into statistically distinct groups with non-negligible difference [36, 37]. Figure 1 reveals that our approach (RS-XGB) expresses the lowest treatment means compared to the remaining baseline models. Therefore, the ESD test unveils that out of the nine classifiers used in this experiment, we find six clusters for the AUC evaluation metric (the left portion of Figure 1). While, for the F1 evaluation metric (the right portion of Figure 1), we detected seven clusters. The best performing model separates from the baseline models for each of the evaluation metrics. As a result, we may conclude that RS-XGB is slightly more effective to predict defects using the NASA data.

From these experiments, we may assume that the superiority of random models occurs due to some reasons. In the first place, we use a flexible tree boosting algorithm based on training a large number of low-accuracy models and then combining the predictions produced by those weak models to obtain a high-accuracy model. Further, we implement a particular subset of features instead of requiring the algorithm to use all available software features. In the remainder of this paper, we show the results considering the AUC evaluation metric as it is very similar to the F1 measure in our context.

Table 3: AUC Numbers / F1 Measure score for different NASA projects. Numbers in bold indicate the best models for each evaluation metric.

Baseline Models Performance (AUC / F1)									
Models	CM1	KC1	KC3	MC2	MW1	PC1	PC2	PC3	PC4
LR	0.753 / 0.690	0.795 / 0.708	0.691 / 0.704	0.761 / 0.601	0.742 / 0.747	0.852 / 0.717	0.822 / 0.755	0.813 / 0.755	0.901 / 0.799
NB	0.702 / 0.518	0.790 / 0.509	0.677 / 0.584	0.739 / 0.514	0.724 / 0.660	0.799 / 0.442	0.805 / 0.543	0.780 / 0.661	0.861 / 0.606
KNN	0.666 / 0.736	0.689 / 0.723	0.670 / 0.711	0.783 / 0.657	0.656 / 0.722	0.734 / 0.787	0.554 / 0.843	0.649 / 0.706	0.732 / 0.791
NN	0.744 / 0.397	0.797 / 0.707	0.707 / 0.576	**0.835** / 0.660	0.689 / 0.534	0.829 / 0.594	**0.905** / 0.721	0.799 / 0.579	**0.921** / 0.743
SVM	0.667 / 0.565	0.767 / 0.653	0.572 / 0.588	0.747 / 0.472	0.659 / 0.633	0.774 / 0.469	0.139 / 0.638	0.476 / 0.445	0.879 / 0.523
CART	0.674 / 0.760	**0.818** / 0.766	0.754 / 0.705	0.709 / 0.719	0.703 / 0.761	0.819 / 0.732	0.832 / 0.787	**0.842** / 0.787	0.900 / 0.844
RF	0.706 / 0.719	0.584 / 0.752	0.605 / 0.610	0.483 / 0.729	0.735 / 0.771	0.696 / 0.764	**0.901** / 0.799	0.734 / **0.835**	0.601 / 0.852
XGB	0.722 / 0.776	**0.814** / **0.799**	0.655 / 0.781	0.766 / 0.779	0.779 / 0.791	0.802 / 0.811	**0.899** / 0.876	0.811 / 0.821	0.884 / 0.852
RS-XGB	**0.801** / **0.799**	**0.815** / **0.803**	**0.839** / **0.821**	0.781 / **0.796**	**0.849** / **0.805**	**0.868** / **0.851**	**0.905** / **0.888**	**0.842** / 0.802	**0.917** / **0.891**

3.2 Features and Prediction Effectiveness

We devote the following set of experiments to evaluating the number of features that contribute to the understandability of our machine learning model. Figure 2 shows AUC numbers obtained by models varying the number of software features within the model, and also average AUC numbers when considering all models of a particular size. The plots show AUC numbers obtained by best and worst-performing XGBoost models of each size. Interestingly, the best performing models are composed of up to six features in all datasets, which enables good understandability with high AUC numbers. Results show that as the number of features increases, AUC values often decreases.

To get an overview of which features are most important for a model, we plotted the SHAP values of every feature within the model for every module. Figure 3 shows SHAP summary plots associated with the best models for eight datasets. The plot sorts feature by the sum of SHAP value magnitudes overall modules and use SHAP values to show the distribution of the impacts each feature has on the model output. The feature value in red is high and the feature value in blue is low. We understand model decisions for modules in PC2 and PC3 datasets with only two features, four of the datasets used four features (CM1, KC3, MW1, PC4), one used five features (MC2), and one used eight features (KC1). Notice that the relevant features may vary depending on the dataset. Some of the most relevant features are LOC_COMMENTS (number of comment lines), a feature already discussed in the literature as a source of "bad smells" because of its capacity to hide the real code complexity [7]. Figure 3 also shows that Halstead's software metrics and NUM_OPERANDS (total number of operands) are often present in variations of the model explanation. As the NASA dataset focuses on the Halstead and McCabe metrics, we can infer that the Halstead metrics are more useful to understand machine learning model prediction.

3.3 Model Understandability

We devote the final set of experiments to verifying the implications of the model for software developers' understandability. Figure 4 illustrates an explanation for the decision of an arbitrary model randomly gathered from the model space composed of the combination of all features. This randomly selected model represents one instance of one module in one of the nine NASA datasets. Thus, software features in red increase the model output, while features in blue decrease the output. As a result, features LOC_TOTAL=284 (total number of lines of code) and BRANCH_COUNT=45 (number of branches) yield the most significant increase for predicting a defective module and features NUM_UNIQUE_OPERANDS (unique operands) and LOC_EXUCUTABLE (number of lines of code executable) have a minor impact on the model prediction in the same direction. There are features in blue that contribute to the module not being defective. However, as their influence is too low, the figure does not show these features. Such type of explanation is helpful when the developer of the module needs to evaluate a single defective module.

Based on this scenario, we conducted an online survey with 40 developers from unique backgrounds. The developers are all based in Brazil. The example used in the survey study is the same as presented in Figure 4. The key idea of the survey is to understand whether developers can comprehend the results of our models generated from our approach to feature selection with SHAP. After understanding the local explanation, we check if developers are undertaking the proper actions on the defective module based on multiple-choice questions. Even though the survey was multiple-choice, we provide the participants with a text-box to express any opinion about the study. The entire survey is available in the replication package of this paper.

We focus the study on the background of the participants and their understandability of an arbitrary predictive model. Among the participants, 18 (45%) holds a master's degree in some computer science areas, e.g., information systems, computer science, or computer engineering. The other 11 (27.5%) holds an undergraduate degree in some computing area, and 7 (17.5%) are still undergraduates in computer-related areas. The remaining 4 (10%) has a Ph.D. in computer science. Twenty-seven developers (67.5%) studied computer science, eleven (27.5%) studied information systems, and two (5%) studied computer engineering in their latest achieved degree. The survey also questioned how long the participants developed code in their careers. The results showed that 19 (47.5%) developed software for over five years, ten (25%) participants developed code between three and four years, and another ten (25%) developers worked in the industry for less than three years. Only one participant (representing 2.5%) opted to not respond to this question. As the results indicate, most developers that participated in the survey

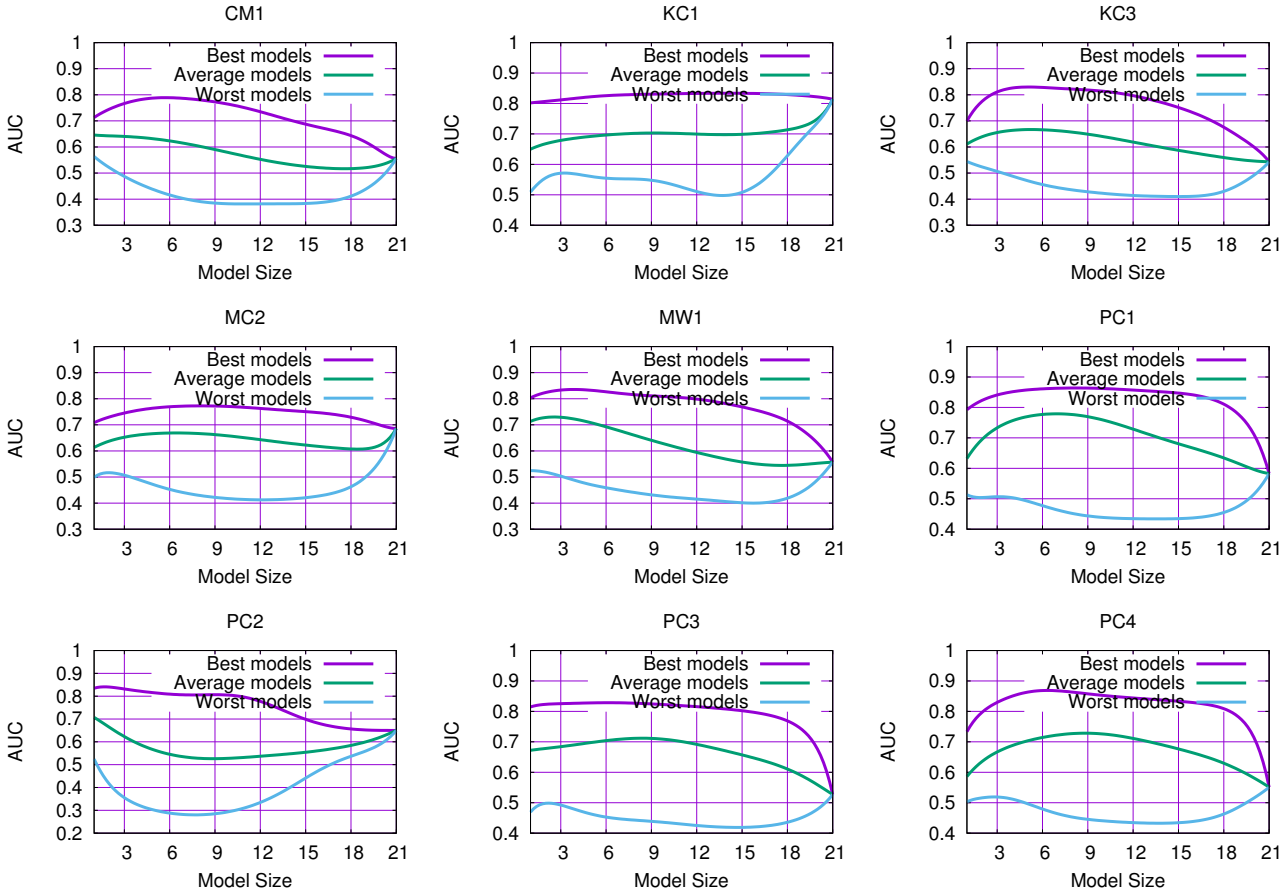

Figure 2: AUC numbers for different datasets and varying number of features (model size) within the models.

hold a master's degree in computer science and developed software for over five years.

3.3.1 Developers Understandability. The second part of the survey evaluates the extent to which the developers understood SHAP feature importances. To do so, we opted for showing Figure 4 and asking the developers three questions to understand the results of our model, as we discuss next.

(a) In the first question (Q1), we wanted to know if the participants would increase the number of lines of code based on the local explanation shown in Figure 4. As we can observe in Figure 4, the total number of lines of code is associated with defective modules. Therefore, we expected that the developers would not increase the size of the module that is already defect-prone. To analyze the data, we have applied a Likert-type scale with five options: (1) strongly disagree, (2) disagree, (3) neither agree nor disagree (4) agree, and (5) strongly agree, as shown in Figure 5. Among the developers, 19 (or 47.5% of participants) strongly disagree (1) with the increase of lines of code in the module. Eleven developers (27.5%) disagree with the statement (2), and six developers (15%) agree (4) in increasing lines of codes. Other options had a minor impact for the developers, e.g., only two developers (5%) strongly agree (5) in adding more lines of code, and two developers (5%) (3) could not give an opinion about the subject. From this question, we may conclude that the developers could understand the local explanation, as 30 developers (75%) disagree or strongly disagree that they should increase the lines of code in this module.

(b) In the second question (Q2), we wanted to explore whether the developers understood the order of importance of the features that affect the defectiveness of the module. Figure 4 shows that the number of lines of code is the feature that contributes the most to classifying the module as defective. Thus, we asked if the developers consider that the total number of lines of code was more important than the number of unique operands. We chose this comparison because these two features are similar. Hence, we want to avoid confusion from the participants. This question used the Likert type scale from the last question. We also show the results in Figure 5, thirteen developers (32.5% of participants) (5) strongly agree that the total number of lines of code is more important for the model. Twelve developers (30%) (4) agree that the total number of lines of code is more relevant to the

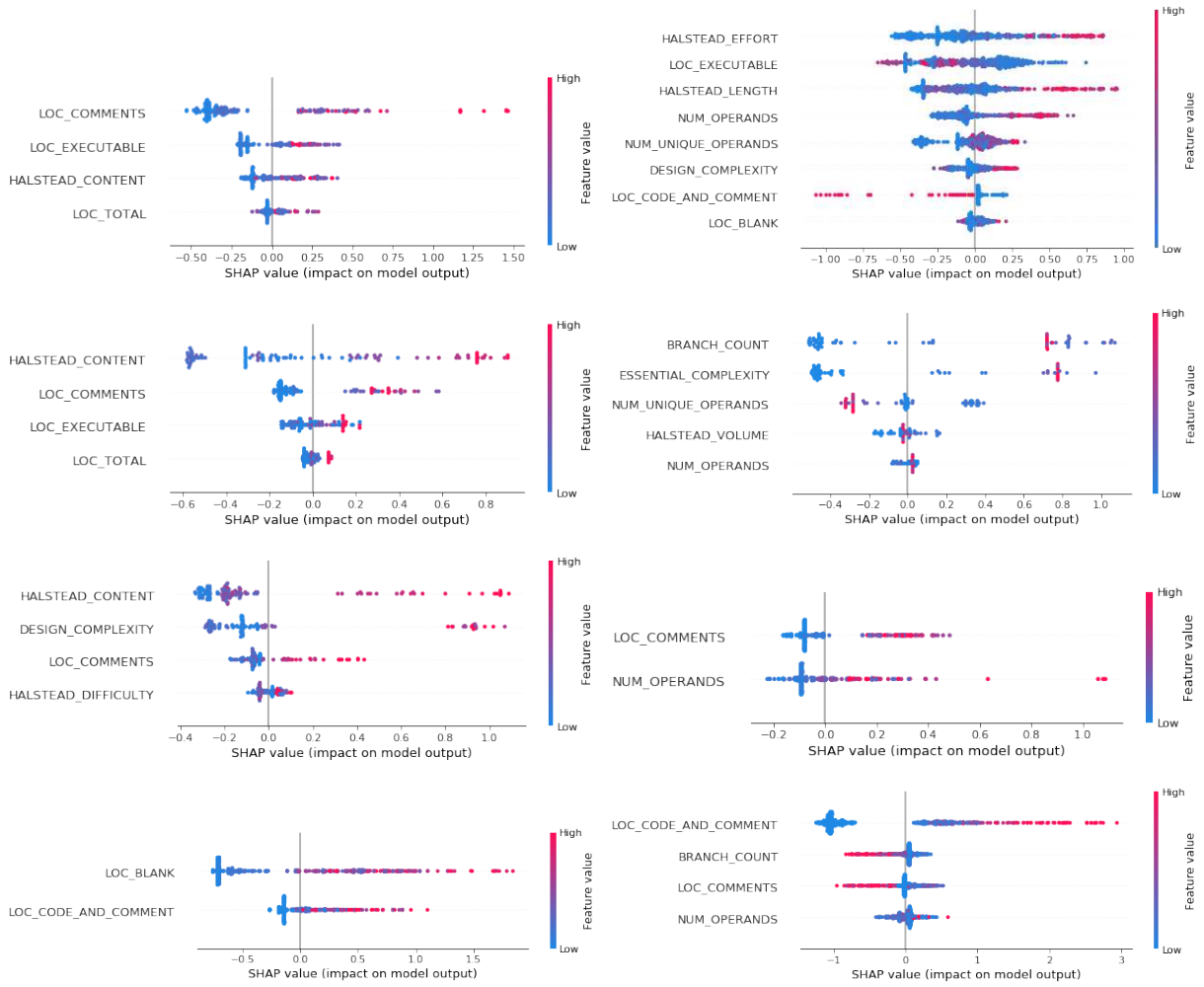

Figure 3: (Color online) From top to bottom, left to right: CM1, KC1, KC3, MC2, MW1, PC2, PC3, PC4. Best overall performing models for each dataset.

Figure 4: Local explanation randomly selected from the predictor.

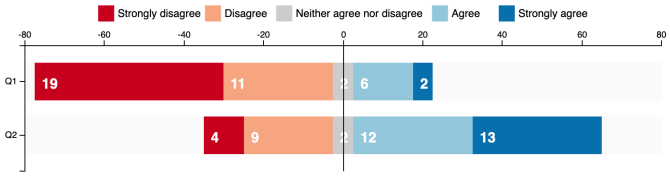

Figure 5: Questions 1 and 2 about the defectiveness of the model.

model than the number of unique operands. Nine developers (22.5%) (2) disagree with the importance of lines of code. Participants chose other options, as four developers (10%) (1) strongly disagree the total number of lines of code is more important to the model, while two developers (5%) (3) could not express an opinion about the subject. We conclude that most developers (25 or 62.5%) understood the output and considered the number of lines of code more important than the number of unique operands. It is important to note that

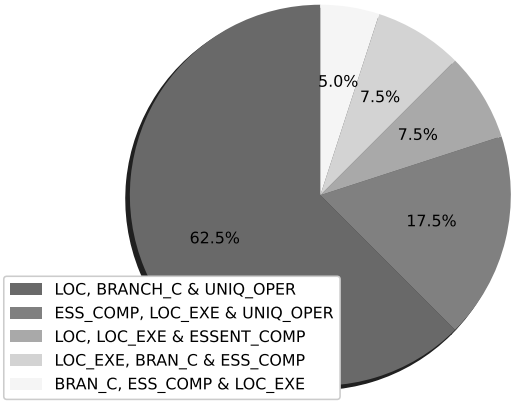

Figure 6: Set of Important Features.

little to no training was provided to the developers to respond to the survey. We adopted a basic textual description of the features and generated models as the only training to capacitate the developers about the survey. For this reason, we believe training would improve the understandability of developers about SHAP importances. However, the results from both questions (Q1 and Q2) show that the developers could comprehend the local explanation.

(c) Finally, we asked developers which are the top three features that bring the module to a defective state (Q3) (based on Figure 4). We give the participants several combinations of three options. The right answer was the combination of the total number of lines of code, number of branches, and number of unique operands. Figure 6 shows that 25 developers (62.5%) chose the right combination of defect-prone features. At the end of the survey, we concluded that most developers could understand the local explanation generated by SHAP using RS−XGBoost. We also conclude that if participants had the local explanation during the development of a module, they could anticipate problems that may arise in the module based on the local explanations provided in this survey.

3.3.2 Implications for Developers. The end of the survey and the experiments with machine learning models provided insights into the implications of our work for developers.

(i) Our study infers how hard it is to build models that are understandable from different software projects. For this reason, developers should not expect identical machine learning models to explain various software projects built from distinct programming languages. We recommend training models under the same software project, or at least, training models using the same programming language.

(ii) Our study indicates that it is possible to apply a technique similar to the one used in our experimental phase to build a tool to classify defects in software projects. We found that the only restriction to such a tool relates to the feature classification under Halstead and McCabe metrics.

(iii) Although developers were not given detailed training about the survey (or SHAP), they were able, in most cases (75% in Q1, 62.5% in Q2, and 62.5% in Q3), to understand the output provided by SHAP. We may conclude that we reached considerable model understandability because the developers could assess the scenario only accounting the knowledge about software development. Again, the proposed tool could output SHAP graphs, like the ones used in the survey, to support the developers' understanding of their project.

4 THREATS TO VALIDITY

This work has some limitations that could potentially threaten our conclusions. In this section, we discuss four types of threats to our investigation. First, we examine the external threats to validity. Then, we review the internal threats to validity. Next, we consider the construct threats to validity. Finally, we show the conclusion threats to validity.

External Validity: Threats to external validity represent situations that limit our ability to generalize the results of our study [45]. In our study, a threat to the external validity concerns the limited number of projects we analyzed (only nine NASA projects). Furthermore, the scope of these projects probably limits the capability to generalize to various contexts of software development. For this reason, the findings of our investigation may not generalize well to other software projects, especially the ones implemented in various programming languages (as most of NASA projects are based on C language).

Internal Validity: Threats to internal validity are practices that can affect the independent variable to causality [45]. In our context, this threat refers to the selected NASA datasets because we naively applied the data reported in the NASA data program [22]. However, we could not validate the data on how the NASA team collected the data [23]. As a result, the data may be incomplete or even wrongly selected by the authors [12, 27]. Other papers already demonstrated minor inconsistencies in this data [9, 12]. To mitigate the effect of imbalanced data, which is a known problem in the NASA data [9, 12, 27], we apply a machine learning technique called SMOTE to balance the data. However, we cannot guarantee that the data reflects the actual nature of the software projects used in our study.

Construct Validity: Construct validity assumes the result of the experiments to the concept or theory [45]. The current literature acknowledges SHAP values as a valid technique to understand machine learning predictions [21]. However, other agnostic models may find different explanations based on a series of software features and data internal arrangement. As an example, the current explainable defect prediction literature focus on techniques such as LIME and BreakDown [14] to understand defects in source code.

Conclusion Validity: Threats to the conclusion validity relate to issues to express the correct conclusion between the treatment and the outcome [45]. In our study, this threat also links to the explanations generated by SHAP. Our models depend upon the defect labels of the NASA dataset [22]. Other research studies learned that many projects rely on a six months post-release period to generate defects in the source code [49]. Therefore, our study did not take into consideration the post-release windows and may not generalize well in instances reported in previous works.

5 RELATED WORK

Software defect prediction applying machine learning techniques has received extensive recognition in the software engineering community for a long time. Several research studies rely on source code metadata [42] and software metrics [15, 22] as features to machine learning-based algorithms. For instance, Wang et al. [42] studied the impact of using the program's semantic as the prediction model's features. The authors used deep learning networks to automatically learn semantic features from token vectors obtained from abstract syntax trees. In a similar approach, Xu et al. [46] employed a non-linear mapping method to extract representative features by embedding the original data into a high-dimension space. Their results achieved average F-measure, g-mean, and balance of 0.480, 0.592, and 0.580. Our work, on the other hand, aimed at using features learned from the classic Halstead and McCabe metrics.

The current literature applies several software metrics for defect prediction. As an example, Menzies et al. [22] presented defect classifiers using code attributes defined by McCabe and Halstead metrics. They concluded that the choice of the learning method is more important than which subset of the available data we use for learning. From a different perspective, Jing et al. [15] used a dictionary learning technique to predict software defects by using characteristics of software metrics mined from open-source software. They used datasets from NASA projects as test data to evaluate the proposed method, which achieved a recall value of 0.79, improving the recall by 0.15 when compared to other methods. In this paper, we also used the McCabe and Halstead software metrics and the NASA datasets. However, unlike Menzies et al. [22] and Jing et al. [15], we focused on understandable machine learning models for predicting software defects.

Some studies investigate cross-project and cross-company defect prediction [8, 40]. For instance, Fukushima et al. [8] explored cross-project prediction models within the context of just-in-time prediction. Their results indicated no relationship between project prediction performance and cross-project prediction performance, and just-in-time prediction models built using projects with similar characteristics or using ensemble methods usually perform well in a cross-project context. In a similar approach, Turhan et al. [40] used cross-company data for building localized defect predictors. They used principles of analogy-based learning to cross-company data to fine-tune these models for localization. The authors used static code features extracted from the source code, such as complexity features and Halstead metrics. The paper concludes that cross-company data are useful in extreme cases, and when within-company data is not available. Unlike these previous papers, we did not aim at analyzing defects across different projects or companies.

Defect prediction is challenging, and previous work addresses these matters [33, 35]. For instance, Tantithamthavorn and Hassan [33] documented pitfalls and difficulties in applying novel defect modeling. The authors divided their model into seven steps: hypothesis formulation, designing metrics, data preparation, model specification, model construction, model validation, and model interpretation. Then, they discussed pitfalls for each step of the proposed defect modeling. In a different paper, Tantithamthavorn et al. [35] showed the impact of noisy data on the creation of defect prediction models. They argue that mislabelled data could impact not only the effectiveness but also the reliability of the model. The authors apply a case study with thousands of manually-curated issue reports. Unlike Tantithamthavorn work [33, 35], we did not focus on the pitfalls and challenges of dealing with noisy data. Although, we tried to mitigate or overcome them in our study with correlation analysis, feature importance, and proper data cleaning.

In this work, we used features composed of software metrics proposed by McCabe and Halstead to built understandable defect prediction models using the tree boosting algorithm XGBoost [3]. We compared the effectiveness of XGBoost against seven well-known machine learning methods, which usually perform well on the defect prediction task. We went beyond the aforementioned works by using SHAP values [21] to optimally compute the importance of each feature in the prediction, which allows identifying the most impactful software quality features when predicting software defects.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we examined the space for software defect prediction models using an efficient implementation of the ensemble method known as the XGBoost algorithm, which resulted in millions of randomly generated machine learning models. Therefore, we evaluated these models considering their accuracy and understandability. Thus, our study found that only 3.5% of the target models (out of a space of 2 097 152 models) achieved AUC numbers superior compared to the baseline. We also demonstrated that software defect prediction represents a project-specific task. In this sense, software features composing the effective models may vary depending on the project characteristics. We conclude that it is helpful to understand the software features contributing to model decisions. Finally, we applied SHAP values to understand model decisions and observed that best performing models are simple to understand because they are composed of a few features and well-distributed numbers. Thus, model explanations may provide insight on which software quality features of the code are more prone to defect.

As future work, we plan to mine data from public repositories on GitHub or similar platforms. We could label this data and then apply similar models used in this research. Therefore, we would provide the software quality community with additional case studies of the models proposed in this paper. The end product of this study could be a tool for developers to analyze their projects. Thus, we would like to test how developers would apply such machine learning tools to track defects in their projects. However, a prominent issue to using data publicly available at GitHub is the precise labeling of the data using McCabe and Halstead's metrics. Furthermore, the public data available on GitHub could provide different perspectives in terms of predicting software defects using machine learning models. For example, we would like to classify a commit to generate a temporal analysis concerned with the evolution of the software throughout time.

ACKNOWLEDGMENTS

This research was partially supported by Brazilian funding agencies: CNPq (Grant 424340/2016-0), CAPES, and FAPEMIG (grant PPM-00651-17).

REFERENCES

[1] A. Agrawal and T. Menzies. 2018. Is better data better than better data miners?: on the benefits of tuning SMOTE for defect prediction. In *International Conference of Software Engineering (ICSE)*.

[2] T. Chen and C. Guestrin. 2015. XGBoost : Reliable Large-scale Tree Boosting System.

[3] T. Chen and C. Guestrin. 2016. Xgboost: A scalable tree boosting system. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*.

[4] D. Cruz, A. Santana, and E. Figueiredo. 2020. Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. In *International Conference on Technical Debt (TechDebt '20)*.

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

[6] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. 2018. A Public Unified Bug Dataset for Java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. Association for Computing Machinery.

[7] M. Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[8] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Working Conference on Mining Software Repositories (MSR)*.

[9] B. Ghotra, S. McIntosh, and A. E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*.

[10] C. Goutte and E. Gaussier. 2005. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *Proceedings of the 27th European Conference on Advances in Information Retrieval Research (ECIR)*.

[11] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. 2009. Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics. In *Engineering Applications of Neural Networks*.

[12] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. 2011. The misuse of the NASA metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation Assessment in Software Engineering (EASE)*.

[13] T. Jiang, L. Tan, and S. Kim. 2013. Personalized defect prediction. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[14] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy. 2020. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models.

[15] X. Jing, S. Ying, Z. Zhang, S. Wu, and J. Liu. 2014. Dictionary learning based software defect prediction. In *International Conference of Software Engineering (ICSE)*.

[16] R. Kaur and S. Sharma. 2019. An ANN Based Approach for Software Fault Prediction Using Object Oriented Metrics. (2019).

[17] P. Knab, M. Pinzger, and A. Bernstein. 2006. Predicting Defect Densities in Source Code Files with Decision Tree Learners. In *Proceedings of the International Workshop on Mining Software Repositories (MSR) (MSR)*.

[18] M. Kuhn. 2015. Caret: Classification and regression training. http://topepo.github.io/caret/index.html.

[19] Buitinck L., Louppe G., Blondel M., Pedregosa F., Mueller A., Grisel O., Niculae V., Prettenhofer P., Gramfort A., Grobler J., Layton R., VanderPlas J., Joly A., Holt B., and Varoquaux G. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*.

[20] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E J. Whitehead Jr. 2013. Does bug prediction support human developers? findings from a google case study. In *International Conference of Software Engineering (ICSE)*.

[21] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Annual Conference on Neural Information Processing Systems (NIPS)*.

[22] T. Menzies, J. Greenwald, and A. Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering* (2007).

[23] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* (2010).

[24] T. Menzies and J. S. Di Stefano. 2004. How good is your blind spot sampling policy. In *IEEE International Symposium on High Assurance Systems Engineering*.

[25] G. Moreira, R. Mellado, R. Junior, A. Cunha, and L. Dias. 2012. Predicting Post-Release Defects in OO Software using Product Metrics. In *IX Experimental Software Engineering Latin American Workshop*.

[26] N. Nagappan, T. Ball, and A. Zeller. 2006. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*.

[27] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. 2016. The Jinx on the NASA Software Defect Data Sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Article 13, 5 pages.

[28] Shapley L. S. 1953. A Value for n-Person Games. In *Annals of Mathematical Studies*, H. W. Kuhn and A. W. Tucker (Eds.). Princeton University Press.

[29] G. E. Santos and E. Figueiredo. 2020. Commit Classification using Natural LanguageProcessing: Experiments over Labeled Datasets. In *XXIII Ibero-American Conference on Software Engineering*.

[30] G. E. Santos and E. Figueiredo. 2020. Failure of One, Fall of Many: An Exploratory Study of Software Features for Defect Prediction. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation*.

[31] B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang. 2013. Software Defect Prediction Using Dynamic Support Vector Machine. In *Ninth International Conference on Computational Intelligence and Security*.

[32] M. Tan, L. Tan, S. Dara, and C. Mayeux. 2015. Online defect prediction for imbalanced data. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*.

[33] C. Tantithamthavorn and A. E. Hassan. 2018. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[34] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *CoRR* (2018).

[35] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *International Conference on Software Engineering (ICSE)*.

[36] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* (2017).

[37] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2019. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* (2019).

[38] M. T. Thwin and T. Quah. 2005. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of systems and software* (2005).

[39] B. Turhan and A. Bener. 2009. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering* (2009).

[40] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* (2009).

[41] J. Wang, B. Shen, and Y. Chen. 2012. Compressed C4.5 models for software defect prediction. In *International Conference on Quality Software (QSIC)*.

[42] S. Wang, T. Liu, and L. Tan. 2016. Automatically learning semantic features for defect prediction. In *International Conference of Software Engineering (ICSE)*.

[43] T. Wang and W. Li. 2010. Naive bayes software defect prediction model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)*.

[44] E. J. Weyuker. 1988. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering* (1988).

[45] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. 2012. *Experimentation in Software Engineering*. Springer.

[46] Z. Xu, J. Liu, X. Luo, and T. Zhang. 2018. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[47] X. Xuan, D. Lo, X. Xia, and Y. Tian. 2015. Evaluating Defect Prediction Approaches Using a Massive Set of Metrics: An Empirical Study. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*. 4.

[48] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. 2016. Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better than Supervised Models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[49] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn. 2019. Mining Software Defects: Should We Consider Affected Releases?. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[50] Sun Z., Li J., and Sun H. 2018. An empirical study of public data quality problems in cross project defect prediction. *Computing Research Repository (CoRR)* (2018).

[51] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. 2017. The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models. *IEEE Transactions on Software Engineering* (2017).

[52] T. Zimmermann and N. Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *International Conference of Software Engineering (ICSE)*.