Automated Software Engineering (2020) 27:369–392
https://doi.org/10.1007/s10515-020-00277-4

Understanding machine learning software defect predictions

Geanderson Esteves[1] · Eduardo Figueiredo[2] · Adriano Veloso[2] · Markos Viggiato[3] · Nivio Ziviani[1]

Received: 4 August 2019 / Accepted: 14 September 2020 / Published online: 12 October 2020

Abstract

Software defects are well-known in software development and might cause several problems for users and developers aside. As a result, researches employed distinct techniques to mitigate the impacts of these defects in the source code. One of the most notable techniques focuses on defect prediction using machine learning methods, which could support developers in handling these defects before they are introduced in the production environment. These studies provide alternative approaches to predict the likelihood of defects. However, most of these works concentrate on predicting defects from a vast set of software features. Another key issue with the current literature is the lack of a satisfactory explanation of the reasons that drive the software to a defective state. Specifically, we use a tree boosting algorithm (XGBoost) that receives as input a training set comprising records of easy-to-compute characteristics of each module and outputs whether the corresponding module is defect-prone. To exploit the link between predictive power and model explainability, we propose a simple model sampling approach that finds accurate models with the minimum set of features. Our principal idea is that features not contributing to increasing the predictive power should not be included in the model. Interestingly, the reduced set of features helps to increase model explainability, which is important to provide information to developers on features related to each module of the code which is more defect-prone. We evaluate our models on diverse projects within Jureczko datasets, and we show that (i) features that contribute most for finding best models may vary depending on the project and (ii) it is possible to find effective models that use few features leading to better understandability. We believe our results are useful to developers as we provide the specific software features that influence the defectiveness of selected projects.

Keywords Software defects · Explainable models · Jureczko datasets · SHAP values

Extended author information available on the last page of the article

1 Introduction

Software defect prediction is a field of interest in both academic literature and the software industry (Nagappan and Ball 2005; Jiang et al. 2013; Agrawal and Menzies 2018). Defect prediction models are based on learned features from either (i) source code and metadata information (Menzies et al. 2007; Turhan et al. 2009; Jiang et al. 2013; Jing et al. 2014; Fukushima et al. 2014; Tantithamthavorn et al. 2015; Tantithamthavorn and Hassan 2018) or (ii) metrics used to specify software design complexity (Wang et al. 2016; Xu et al. 2018). Studies on features learned from software source code and metadata information usually use approaches based on deep neural networks (Wang et al. 2016; Xu et al. 2018). Studies that rely on software metrics use either code inspections and unit testing (Fukushima et al. 2014) or machine learning approaches, such as support vector machines (Elish and Elish 2008; Gray et al. 2009), decision trees (Knab et al. 2006; Jiang et al. 2013; Jing et al. 2014), naïve bayes (Turhan and Bener 2009; Wang and Li 2010), neural networks (Thwin and Quah 2005; Jing et al. 2014; Yang et al. 2016), or dictionary learning-based prediction (Jing et al. 2014).

Despite the considerable accuracy usually achieved by machine learning models (Menzies et al. 2010; Wang and Li 2010; Jiang et al. 2013; Fukushima et al. 2014; Wang et al. 2016), they are often overly complex and hinder the understandability of the model. In most cases, we usually cannot explain the prediction, and we still need investigation regarding the explanation of model decisions that could help developers to better understand the rationale behind the defect model (Jiang et al. 2013; Lewis et al. 2013; Jing et al. 2014). Further, explaining model decisions is also beneficial, as it enables the proper understanding of the effects in software development costs and efforts during development. Furthermore, understanding these machine learning models is especially important for software development, as assuring software system quality is expensive, and defect-fixing processes require a laborious effort from a company (Zhang et al. 2017). 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 (Nagappan et al. 2006; Agrawal and Menzies 2018).

Motivated by the benefits of predicting software defects for developers and companies apart, the goal of this study is to explore software features that can help practitioners to understand software defects affecting code quality. Further, we also want to investigate the power of these features to predict software defects. Guided by this goal, our study investigates the following research questions.

RQ1: Do optimized XGBoost using random search outperforms the state-of-the-art machine learning classifiers for defect prediction?
RQ2: How does the number of features impact the performance of defect models?
RQ3: How comparable is the predictive accuracy and variability of features in defect prediction models?

Differently from previous studies that tune a single defect prediction model (Elish and Elish 2008; Wang and Li 2010; Fukushima et al. 2014), we perform an extensive exploration of the model space, which results in hundreds of thousands of candidate models. Specifically, we learned models considering distinct combinations of Object-

Table 1 Baseline methods used in previous works about the defect prediction task

ML methods	Literature source
Logistic Regression	Nagappan et al. (2006), Jiang et al. (2013), Tantithamthavorn and Hassan (2018)
Naive Bayes	Jiang et al. (2013), Jing et al. (2014), Xuan et al. (2015), Sun et al. (2018)
K-Nearest Neighbor	Turhan et al. (2009), Jing et al. (2014), Xuan et al. (2015)
Neural Network	Stites et al. (1991), Jing et al. (2014), Yang et al. (2016)
Support Vector Machine	Elish and Elish (2008), Shuai et al. (2013), Jing et al. (2014)
Decision Trees	Knab et al. (2006), Jiang et al. (2013), Jing et al. (2014), Ferenc et al. (2018)
Random Forest	Fukushima et al. (2014), Tantithamthavorn et al. (2015), Tantithamthavorn and Hassan (2018)

Oriented features (Chidamber and Kemerer 1994; Gyimothy et al. 2005; D'Ambros et al. 2010; Couto et al. 2012; Jiang et al. 2013; Herbold and Crosspare 2015) applied to eight Java projects. The data used in this research is publicly available under the PROMISE Software Engineering Repository (Sayyad Shirabad and Menzies 2005). Hence, we compose each sampled model of a specific set of features. As a result, the learned models correspond to a myriad of explanations for the software defect phenomenon.

To evaluate our modeling approach, we compared the effectiveness of our models with other machine learning methods typically employed in software defect prediction (Table 1). For seven out of the eight projects, we could learn models that achieved similar or superior effectiveness when compared with baseline models. Our accuracy numbers range from nearly 67 to 86%. Besides the seven machine learning methods, we also added XGBoost as a baseline method (Lundberg and Lee 2017a). Differently from our approach, all baseline algorithms employ the full set of features while learning their models (Nagappan et al. 2006; Jiang et al. 2013; Jing et al. 2014; Fukushima et al. 2014; Tantithamthavorn and Hassan 2018; Tantithamthavorn et al. 2015). We found hyper-parameters using an automated parameter optimization technique (Kuhn 2015). The baseline served as a metric for comparing the target dataset.

Our findings also show that random exploration of the model space results in effective models. In the Jureczko dataset, on average, 3.5% of the randomly generated models (in a space of 1,997,287 models) are superior when compared to baseline methods. Additionally, we showed that some features are more important for defect prediction, but the importance of these features varies within distinct projects. For instance, for the JEDIT project, our model explained the prediction using only the MAX CC (Maximum McCabe's Complexity) feature (McCabe 1976; McCabe and Butler 1989). This means that for the JEDIT project, classes with high McCabe's complexity have more chance of being defective than classes with low complexity. Finally, we compare the predictive accuracy and variability of the software features.

We organize the rest of this paper as follows. Section 2 presents related works regarding the task of learning from source code and metadata information and learning from software metrics using machine learning. In Sect. 3, we model the problem to predict software defects based on the Jureczko datasets. We then present and discuss our results and their implications in Sect. 4. In Sect. 5, we present the threats to validity in detail. Finally, in Sect. 6, we conclude our paper and present directions for future work.

2 Related work

Two main approaches tackle the defect prediction task (Nagappan and Ball 2005; Nagappan et al. 2006; Wang et al. 2016; Xu et al. 2018). The first approach aims at the use of source code metrics as the input of a model that learns the behavior of the software system (Nagappan and Ball 2005; Nagappan et al. 2006; Menzies et al. 2007; Jiang et al. 2013; Tantithamthavorn and Hassan 2018). The current literature applied this approach for several decades in different contexts of defective prediction. The second type comprises efforts to develop models that learn from the source code and metadata information (D'Ambros et al. 2010; Wang et al. 2016; Xu et al. 2018). In this effort, the number of possibilities is massive, and so the effort and difficulty of designing models from these metrics. This section discusses significant studies into both approaches. Furthermore, we also analyzed the distinct types of datasets we can use for the defect prediction task. Then, we discuss different algorithms used to predict defects. Finally, we present a discussion over the understandability of a defect prediction model as a recent trend into this task.

2.1 Learning from source code metrics

Machine Learning (ML) approaches have received extensive attention in the software engineering (SE) community for a considerable period. One of the efforts to create effective ML models valuable for the SE community is the classification/regression using source code metrics. Even though these efforts share the fundamentals of analyzing code metrics, they also vary in terms of accuracy, complexity, and the input data they require to predict a defect. As an example, Nagappan and Ball (2005) present a technique for the prediction of software defect density managing a collection of applicable code churn patterns. Using regression models, the authors show that absolute software measures of code churn are poor predictors of defect density. At that moment, the authors proposed a recent set of relative measures capable of predicting defect density. In a similar approach, Nagappan et al. (2006) also conducted an empirical study of the post-release defect history of five Microsoft systems. They found failure-prone software entities are statistically correlated with code complexity measures. Using Principal Component Analysis (PCA), they built regression models that accurately predict the likelihood of post-release defects for current entities.

In another direction, Jiang et al. (2013) proposed a personalized defect prediction approach, in which prediction models were built for each developer to predict software defects at the file level. They compose the features of attributes extracted from a commit describing characteristics of the source code, such as Lines of Code (LOC).

The work used three categories of software features, namely characteristic vectors, bag-of-words, and metadata. In a broader discussion, Tantithamthavorn and Hassan (2018) documented pitfalls and challenges in applying the defect modeling for ML models aiming to accurately predict defects. This model is divided into seven different steps: hypothesis formulation, designing metrics, data preparation, model specification, model construction, model validation, and model interpretation. The authors discussed pitfalls for each step of the defect modeling.

In another study in this direction, Jing et al. (2014) used the 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 baseline literature recall by 0.15. Similarly, Turhan et al. (2009) 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 code, such as complexity features and Halstead metrics. The paper concludes that cross-company data are useful in extreme cases and should be used only when within-company data are not available.

As we can notice, the above-mentioned works applied machine learning to predict software defects in unique circumstances. Moreover, they use ML algorithms to predict defects using various techniques and metrics. The main difference in our work compared to these previous studies is that we are interested in the understandability of ML models. Aiming that goal, we employed an innovative technique identified as SHapley Addictive exPlanation (SHAP) values that allowed the explanation of the features that may affect the defect prediction of selected projects. Further, we also discuss the power of these features based on their accuracy and variability for the defect prediction task.

2.2 Learning from source code and metadata information

The prediction of software defects prevails a complex task by definition. In some cases, the source code metrics, as the ones mentioned in the previous section, are not sufficient and efficient for the defect prediction task. For these reasons, many papers adopted models employing the code metadata information. As an example, Wang et al. (2016) examined the impact of using a system's semantic as the prediction model's features. The authors used deep belief networks to automatically learn these features from token vectors collected from abstract syntax trees. Then, they evaluated the model on ten open-source projects and improved the F1 score for both within-project defect prediction by 14.2% and cross-project defect prediction by 8.9%. Similarly, the works of Xu et al. (2018) employed a non-linear mapping method to extract representative features by embedding the initial data into a high-dimension space. Their results achieved average F-measure, g-mean, and balance of 0.480, 0.592, and 0.580, respectively, and outperformed nearly all baseline methods.

Our study, on the other hand, takes into consideration metadata information alongside source code metrics. The Jureczko datasets (Jureczko and Spinellis 2010; Jureczko and Madeyski 2010) employed in this research are well-known in the defect prediction literature (Sun et al. 2018; Ferenc et al. 2018). For instance, Sun et al. (2018) compared the effectiveness of predicting software defects from the Jureczko datasets. They conclude that the results are more dependable on the machine learning process, as in the case of adequate data cleaning. Similarly, Ferenc et al. (2018) gathered a wide variety of defect prediction datasets, including Jurescko, and studied the accuracy of decision trees. However, to the best of our knowledge, none of the studies using this dataset has applied explainability concepts aiming to understand the source code metrics and code metadata information guiding these projects to a defective state.

2.3 Datasets for defect prediction

In the current literature, at least two data sources are used to predict software defects. First, a set of defective modules described in the NASA data program (Menzies et al. 2007, 2010) relies on metrics from Halstead‘s operator-operand counts (Halstead 1977) and McCabe‘s dependencies and complexity (McCabe and Butler 1989; McCabe 1976). Even though research studies consider these data noisy and problematic (Gray et al. 2011; Ghotra et al. 2015; Petrić et al. 2016), many studies in the software defect prediction literature applied these data sources. The second widely used dataset relates to the CK metrics. These metrics demonstrated their effectiveness for defect prediction (Jureczko and Spinellis 2010; Jureczko and Madeyski 2010). As a result, the current literature gathers an impressive collection about the CK metrics under the Jureczko dataset publicly available under the PROMISE repositories[1] (Jureczko and Madeyski 2010; Jureczko and Spinellis 2010).

NASA datasets have been an object of study for a considerable period of time in the software engineering community. One of the first studies into these data, Menzies et al. (2007) presented defect predictors using static code attributes defined by McCabe and Halstead features. According to the authors, these metrics are *“useful, easy to use, and widely used”*. They concluded that the choice of the learning method is more important than which subset of the available data is used for learning (Menzies et al. 2007).

Despite the use of the NASA datasets in the current literature, many studies consider these data as being noisy and problematic to predict defects (Gray et al. 2011; Ghotra et al. 2015; Petrić et al. 2016). One of the first criticism about the NASA datasets come from Gray et al. (2011). The authors argue that the data presented in the NASA datasets are problematic. They derive this conclusion from the fact that many data points are duplicated in the public dataset (Gray et al. 2011). Likewise, Ghotra et al. (2015) discuss other problems with the NASA datasets. They conclude that the data is not only erroneous as previously detected by the literature but also biased. For the first conclusion (erroneous data), they show that many entries in the dataset are not correct about the software modules. Then, they also discuss the bias nature in the dataset, as they show it collects the data from only one software setting. Therefore, the authors show that distinct cleaning steps could transform the result achieved by the classifiers (Ghotra et al. 2015). Conclusively, Petrić et al. (2016) discuss how problematic the NASA dataset is for executing any software task. The authors applied an extensive data cleaning process that did not significantly improve the noise included in the data.

[1] http://promise.site.uottawa.ca/SERepository/.

The authors conclude that erroneous data points are unavoidable for the dataset. As a result, they do not suggest using the NASA dataset for defect prediction.

The arguments against NASA datasets are enough to not use the data in our study (Gray et al. 2011; Ghotra et al. 2015; Petrić et al. 2016). Considerably, the static nature of the dataset does not allow the feature engineering process, which could be used to compose other features from the existing ones. Opportunely, the literature shows other alternatives for exploring software defect prediction.

Besides the use of NASA datasets to predict defects, one notable solution is the CK metrics related to Object-Oriented Programming (OOP). The works of Jureczko and Madeyski (2010) described an analysis of several open-source projects. The authors used software metrics to generate clusters that could join projects that have similar defect causes. They found at least six clusters in the dataset, but statistically, only two of them demonstrated to be true. Then, the authors analyzed the clusters with the state-of-art about defect prediction. Along those lines, Jureczko and Spinellis (2010) also proposed a new tool to collect software metrics from Object-Oriented Programming systems. They discuss the inefficiency of current methods of extracting these metrics from existing software projects. Then, the authors developed a model able to find 80% of defective classes within the investigated software projects. They report that two of the CK metrics are class size factors: Weighted Methods per Class (WMC) and Lines of Code (LOC). Despite the high effectiveness to predict defective classes presented in these papers (Jureczko and Madeyski 2010; Jureczko and Spinellis 2010), none of these papers applied explainability techniques aiming at identifying which CK metrics are important for the prediction. We know, for instance, that LOC and WMC are class size factors. However, we could not draw any conclusion about the real impact of each CK metric. In this work, we take into consideration this subject about the CK metrics for defect prediction.

Table 1 presents relevant studies that applied classic machine learning methods for defect prediction in distinct datasets. As we can notice, at least seven algorithms are employed to predict the likelihood of software defects.

2.4 Explaining software defects

The explainability of software defects is a relatively recent topic (Mori and Uchihira 2018; Jiarpakdee et al. 2020). Mori and Uchihira (2018) analyzed the trade-off between accuracy and interpretability of different classifiers. The experimentation displays a comparison between the balanced output that satisfies both accuracy and interpretability criteria. Jiarpakdee et al. (2020) empirically evaluated three model-agnostic procedures: Local Interpretability Model-agnostic Explanations (LIME), and BreakDown techniques. They improve the results found with LIME using hyperparameter optimization, which they called LIME-HPO. This work concludes that (i) model-agnostic techniques are necessary to explain individual predictions of defect models; (ii) instance explanations generated by model-agnostic techniques are mostly overlapping with the global explanation; (iii) model-agnostic techniques take less than a minute to generate instance explanations, and (iv) more than half of the practitioners achieved a contractive explanation for the defect models.

Unlike these papers, we apply another state-of-the-art technique for interpretability and prediction explanation (Wang et al. 2019) known as SHAP values. This technique allows us to understand the predictions made by our models. Furthermore, the current literature employed solely parameter optimization to support the interpretability of these models (Mori and Uchihira 2018; Jiarpakdee et al. 2020). In this work, we apply a novel algorithm to select the most performant features from the power-set of software features, i.e., software metrics.

3 Learning to predict software defects

The task of learning to predict software defects is defined as follows. We have as input the training set (referred to as $\mathcal{D}$), which consists of a set of records in the form $< x, y >$, where x is a module represented as a vector of features $x = \{x_1, x_2, \ldots, x_n\}$, in which each x_i encodes a particular characteristic of the module, and y is the corresponding outcome, i.e., whether the corresponding module is defective or not. The training set is used to construct a model that relates features of the modules to the corresponding outcome. The test set (referred to as $\mathcal{T}$) consists of records $< x, ? >$ for which only the module x is available, while the corresponding outcome y is unknown. The model learned from $\mathcal{D}$ is used to predict the outcomes for modules in $\mathcal{T}$.

3.1 Sampling the model space

Finding the optimal machine learning model, i.e., the subset of features for which we achieve the best prediction accuracy, would require the exhaustive enumeration of all combinations of features. Alternatively, we sampled the model space to obtain a model for each combination of features. Specifically, we sample the model space by randomly selecting a set of features to compose the model. We enumerate models composed of a single feature and generate models of increasing size until we achieve a significant sample size. We chose evenly at random features that compose each module. Thus, our approach differs from the classic implementation of the XGBoost algorithm. We describe the main steps of our sampling approach in Algorithm 1.

To better understand our algorithm approach, Algorithm 1 shows the basic structure of our strategy to model selection. Lines 2 to 7 perform the main loop responsible for going through each set of features from the dataset. As Algorithm 1 displays in the main loop, we search the entire feature space combining all possible sequences of features. Here, we guarantee that the entire set of features is tested with all others. After this process, in Line 8, we stored the highest predictive accurate model for each project.

The features we consider may have a variety of complex nonlinear interactions. Capturing these interactions requires a classification algorithm with significant flexibility, and thus we chose the gradient boosting machines algorithm (Chen et al. 2016). XGBoost belongs to a family of machine learning boosting algorithms. This model uses the gradient boosting (GBM) framework at its core. The boosting technique is a sequential technique that works on the principle of an ensemble. Then, it combines

Algorithm 1 Sampling the Model Space

```
Require: training set $D$
Require: pool of features $F$
Require: number of candidate models $n$
Ensure: the most performant model $m(f^*)$
$i \leftarrow 0$
while $i < n$ do
    $i \leftarrow i + 1$
    $k \leftarrow$ random integer between 1 and $|F|$
    $f \leftarrow k$ distinct features randomly selected from $F$
    compute the predictive performance of $m(f)$ in $D$
end while
$m(f^*) \leftarrow$ model with the highest predictive performance
```

a set of weak learners and delivers improved prediction accuracy. More specifically, models are iteratively trained so that each model is trained on errors of previous models, thus giving more importance to the difficult cases. At each iteration, errors are computed and a model is fitted to these errors. Finally, the contribution of each base model to the final one is found by minimizing the overall error of the final model. Fitting the base models is computationally challenging and, so, we use XGBoost, a recent fast implementation of gradient boosting machines (Lundberg and Lee 2017a).

3.2 Features

Estimating software defects is a task related to learning from either source code metrics or code metadata. In this section, we present an overview of the features used to predict the defects in the source code. All features contained in the Jureczko datasets relate to class-level metrics (D'Ambros et al. 2010; Couto et al. 2012; Herbold and Crosspare 2015). We then introduce these metrics before presenting a complete description of the features.

Several relevant features are present for each project class included in the Jureczko dataset (Jureczko and Spinellis 2010). These features comprise two distinct groups, CK and Object-Oriented metrics (D'Ambros et al. 2010; Couto et al. 2012; Herbold and Crosspare 2015). Notably, the current literature considers Lines of Code (LOC) as one of the most important features for the defect prediction in distinct datasets (Gyimothy et al. 2005).

Jureczko datasets are CK metrics extracted from software projects (Jureczko and Spinellis 2010; Jureczko and Madeyski 2010). These metrics are related to Object-Oriented Programming (OOP) and are ultimately based on several features that may impact on the defectiveness of target source code. The datasets used in this research for CK metrics are all obtained from Java code. Next, we explain each feature used in this study about the CK metrics.

1. Weighted methods per class (WMC): complexity of methods in the class.
2. Depth of Inheritance Tree (DIT): each class has a measure of the inheritance levels from the object hierarchy top.
3. Number of Children (NOC): the number of immediate descendants of the class.

4. Coupling between object classes (CBO): the number of classes coupled to a given class (efferent couplings and afferent couplings). These couplings can occur through method calls or field accesses.
5. Response for a Class (RFC): number of different methods that can be executed when an object of that class receives a message.
6. Lack of cohesion in methods (LCOM): counts the sets of methods from a class that are not related across the sharing of some of the class fields.
7. Lack of cohesion in methods (LCOM3): divided into three aspects.
   (a) m: number of methods in a class.
   (b) a: number of attributes in a class.
   (c) μ(A): number of methods that access the attribute A.
8. Number of Public Methods (NPM): counts all the methods in a class that are declared as public.
9. Data Access Metric (DAM): the ratio of the number of private (or protected) attributes to the total number of attributes declared in target class.
10. Measure of Aggregation (MOA): count of the number of class fields whose types are user defined classes.
11. Measure of Functional Abstraction (MFA): the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class.
12. Cohesion Among Methods of Class (CAM): computes the relatedness among methods of a class based on the parameter list of the methods. The metric is computed using the summation of the number of different types of method parameters in every method divided by multiplication of the number of different method parameter types in the whole class and number of methods.
13. Inheritance Coupling (IC): the number of parent classes in which a given class is coupled. A class considered coupled if one of the following conditions is satisfied:
    (a) One of its inherited methods uses an attribute that is defined in a new/redefined method.
    (b) One of its inherited methods calls a redefined method.
    (c) One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.
14. Coupling Between Methods (CBM): total number of new/redefined methods to which all the inherited methods are coupled.
15. Average Method Complexity (AMC): measures the average method size for each class. The size is measured as the number of lines of code in the method.
16. Afferent couplings (Ca): the number of classes that depend upon the measured class.
17. Efferent couplings (Ce): the number of classes that the measured class is dependent upon.
18. McCabe's cyclomatic complexity (CC): the greatest value of CC among methods of the investigated class.
19. McCabe's cyclomatic complexity (CC): the arithmetic mean of the CC value in the investigated class.

20. Lines of Code (LOC): total number of lines of code in the target class.

3.3 Data and setup

Table 2 shows the data for eight projects and the 20 features described in Sect. 3.2. For each Java module, there is a value attached to each feature and the average value is the sum of values referred to each module divided by the number of Java modules in the project. These average values differ between projects, e.g., the feature lcom (lack of cohesion of methods) for project *jedit* is 233.00 and for project *log4j* is only 37.17. Table 2 also reveals the percent of defective modules in each project and the imbalanced nature of the data. It is also important to note that the proportion of defective modules varies in the dataset. The lowest number of defects is only 8.87% from the *tomcat* project, while we find the highest number of defects in the *log4j* project, where 57.90% of the modules had defects.

We tested the effectiveness of the considered models applying the standard Area Under the Curve (AUC) and the F1 score, as adopted by Caret: Classification and Regression Training (Kuhn 2015). The AUC is an assessment of the probability that a prediction model ranks a randomly chosen positive instance higher than a randomly chosen negative instance. While the F1 score is the harmonic mean of precision and recall (Sokolova et al. 2006). The AUC is important for our study as both datasets are imbalanced. We used ten-fold cross-validation and relevant hyper-parameters were found using a validation set during training. The results reported are the average of the ten runs, and to ensure their relevance we assess the statistical significance of our measurements using the Scott-Knott Effect Difference test Tantithamthavorn et al. (2017, 2018). This test represents the mean comparison that leverages a hierarchical clustering to partition the set of treatment means into statistically distinct groups with a non-negligible difference.

3.4 Feature importance and shapley additive explanations

Effective models generate predictions that are often hard to explain. A key challenge in software defect prediction is to understand the reason why a model has made a specific prediction because it provides insight into potential solutions by focusing on the features of the code that are more associated with the defect (Jiang et al. 2013; Lewis et al. 2013). For instance, a developing team may be able to focus on modifying a specific module if they know that the complexity of that module is contributing to the likelihood that the module presents with a defect.

The typical approach to explain the predictions of a model involves calculating the impact each feature has on the prediction. Feature importance can be defined as the increase in the model prediction error after feature values undergo permutation. This operation breaks the relationship between the feature and the outcome. Therefore, a feature is important if changing its values increases the model error. The increase in model error shows that the model relied on that feature for the prediction. On the other hand, a feature is not important if changing its values does not result in a change in

Table 2 Jureczko dataset

	Projects	TOMCAT	ANT	LOG4J	PROP	XALAN	CAMEL	JEDIT	LUCENE
	Programming Language	Java	Java	Java	Java	Java	Java	Java	Java
	Number of Modules	835	745	449	69653	3320	2784	1749	782
	Defective Modules	8.97%	22.28%	57.90%	12.28%	54.39%	20.18%	17.32%	56.01%
1	wmc	12.95	11.07	7.72	5.22	11.15	8.43	12.78	9.78
2	dit	1.68	2.52	1.67	3.02	2.54	1.95	2.61	1.78
3	noc	0.36	0.73	0.26	0.59	0.55	0.52	0.45	0.65
4	cbo	7.65	11.04	7.20	15.03	12.76	10.68	13.33	10.25
5	rfc	33.47	34.36	23.58	24.65	29.52	20.87	39.48	23.96
6	lcom	176.27	89.14	37.17	37.59	127.51	70.40	233.00	51.22
7	ca	3.86	5.65	3.93	2.82	6.07	5.13	8.10	5.75
8	ce	0.0	5.74	3.61	12.27	7.39	6.13	6.77	4.99
9	npm	10.77	8.36	5.22	3.46	9.08	6.84	7.75	6.69
10	lcom3	1.08	1.01	1.00	1.35	1.14	1.08	1.03	0.95
11	loc	350.43	280.07	177.45	170.23	412.72	111.76	457.30	277.52
12	dam	0.57	0.64	0.22	0.19	0.43	0.61	0.52	0.50
13	moa	0.94	0.72	0.81	0.091	0.80	0.65	1.05	1.18
14	mfa	0.29	0.50	0.29	0.61	0.54	0.39	0.49	0.33
15	cam	0.48	0.47	0.43	0.55	0.47	0.49	0.45	0.43
16	ic	0.27	0.72	0.34	1.08	0.80	0.37	0.64	0.52
17	cbm	0.59	1.31	0.66	1.71	2.87	0.71	1.53	1.14
18	amc	25.57	23.64	20.25	30.27	57.36	10.94	30.64	22.78
19	max_cc	4.27	4.66	3.43	3.30	4.35	2.17	6.72	4.68
20	avg_cc	1.25	1.36	1.34	1.28	1.32	0.94	1.83	1.28

the model error. The lack of a significant change in the model error confirms that the feature was not a factor in the prediction.

Features often interact with each other in many complex ways to create models that provide accurate predictions. Thus, the feature importance is also given as a function of the interplay between the features. In this case, Shapley values (Shapley 1953) can be used to find a fair division scheme that defines how the total importance should be distributed among features. More specifically, samples are transformed into a space of simplified binary features. Explanation models are restricted to the so-called additive feature attributions methods, which means that values predicted by the explanation model are linear combinations of these binary input vectors. 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. Shapley values are theoretically optimal and are unique consistent and locally accurate attribution values. In this work, we use SHAP (SHapley Addictive exPlanation) values Lundberg and Lee (2017b) as an approximation of Shapley values to compute the importance of each feature in the prediction.

Differently from previous papers, we use SHAP values to understand the CK metrics that influence the defectiveness of the classes. Then, we build models using the tree boosting algorithm (Chen et al. 2016). We compare the effectiveness of XGBoost against seven well-known machine learning models that have performed well on the defect prediction task. Our XGBoost implementation differs from the usual use case of the algorithm because we focus on the random search of the models. Furthermore, this mechanism allows the development of accurate models using fewer features from the datasets. We went beyond the aforementioned works by using SHAP (SHapley Addictive exPlanation) values (Lundberg and Lee 2017b) to compute the importance of each feature in the prediction.

SHAP assigns an importance value (positive or negative) to each feature in a particular prediction. The output value comprises the sum of the base value (average prediction over the validation set) and these dominant values. Otherwise, SHAP allows us to summarize important features, and to associate low and high feature values to an increase/decrease in output values (i.e., prediction). As a result, SHAP applies a color-coded violin plot built from all predictions. The color red shows significant numbers and blue insignificant numbers.

4 Results

In the next sections, we present our results and discuss each research question presented in Sect. 1 of this paper.

4.1 Competitiveness of random models

We devote the first set of experiments to answering the research question RQ1: *"Do optimize XGBoost using random search outperforms the state-of-the-art ML classifiers for defect prediction?"*. To answer this question, we applied seven baseline algorithms described in Table 1. We find these base models in the literature in prominent works about the defect prediction task. Note that the literature considers the Random Forest algorithm as a relevant model for defect prediction in many datasets (Fukushima et al. 2014; Tantithamthavorn et al. 2015; Tantithamthavorn and Hassan 2018). As we discuss next, this algorithm was not the best performing model for the defect prediction in the Jurescko datasets. Hereafter, we refer to our approach as US−XGBoost (US−XGB), standing for Unbiased Search for XGBoost models.

Tables 3 and 4 show the results of the experiments. Our unbiased search found efficient models for the defective prediction task. Results suggest that both AUC and F1 numbers are higher when fewer features have been used to create the model. For both evaluation metrics (AUC and F1 score), we did not find better results using the unbiased search for only one project (LUCENE). In that case, the Logistic Regression was fairly superior in both metrics. Thus, in approximately 85% of the projects, US-XGB was the best model for the prediction. For the remaining six projects, our approach could increase the AUC numbers by a large margin. Out of all the possibilities of features, our model could overcome the baseline models in around 3.5% of the cases.

To statistically test the soundness of the baseline results, we applied a Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al. 2017, 2018). Figure 1 shows that US-XGB has the lowest treatment means compared to the seven baseline algorithms for both metrics (i.e., AUC and F1). Out of the eight predictors used in this experiment, we find seven clusters for the AUC metric and five clusters with the F1 score using the target test. The best performing model (US-XGB) was an isolated cluster, separated from the other models in both cases.

RQ1. We conclude that our optimized version of XGBoost (US-XGB) usually outperforms classic baseline models in the defect prediction task.

4.2 Model interpretability

This section focuses on answering RQ2: *"How does the number of features impact the performance of defect models?"*. To answer this question, we use SHAP values to explain our prediction. SHAP values are generated from global models to generate local explanations. These explanations are used to assign importance values to each feature. The SHAP value is relevant because we could provide satisfactory accuracy numbers (i.e., AUC) for our algorithm as discussed in the previous sections (Sect. 4.1). Figure 2 shows SHAP summary plots associated with the most superior models for the selected projects (*tomcat*, *ant*, *log4j*, *prop*, *xalan*, *camel*, *jedit*, and *lucene*). A vertical line shows that points emerging along the right are contributing to increase the likelihood of a defect. Additionally, points appearing on the left side lead to decreasing

Table 3 AUC numbers for different prediction models

	ML methods	TOMCAT	ANT	LOG4J	PROP	XALAN	CAMEL	JEDIT	LUCENE
1	Logistic Regression	0.785	0.722	**0.722**	0.706	**0.668**	0.665	0.807	**0.732**
2	Naive Bayes	0.781	0.704	0.704	0.677	0.648	0.607	0.771	0.705
3	K-Nearest Neighbor	0.689	0.584	0.584	0.698	0.620	0.633	0.711	0.669
4	Neural Network	0.788	0.657	0.601	0.744	0.655	0.613	0.561	0.688
5	Support Vector Machine	0.775	0.487	0.487	0.523	0.561	0.662	0.761	0.585
6	Decision Trees	0.602	0.566	0.558	0.635	0.570	0.639	0.630	0.570
7	Random Forest	0.766	0.592	0.595	0.739	0.611	0.723	0.753	0.647
8	XGBoost	0.776	0.636	0.626	0.777	0.662	0.762	0.821	0.670
	US−XGBoost	**0.859**	**0.731**	**0.715**	**0.889**	**0.665**	**0.802**	**0.836**	0.677

Numbers in bold indicate the best models for each Jureczko dataset

Table 4 F1 numbers for different prediction models

	ML methods	TOMCAT	ANT	LOG4J	PROP	XALAN	CAMEL	JEDIT	LUCENE
1	Logistic Regression	0.237	0.642	0.642	0.511	0.622	0.316	0.288	**0.703**
2	Naive Bayes	0.296	0.556	0.556	0.254	0.411	0.473	0.381	0.467
3	K-Nearest Neighbor	0.170	0.568	0.568	0.216	0.601	0.436	0.296	0.671
4	Neural Network	0.541	0.625	0.547	0.516	0.591	0.519	0.541	0.603
5	Support Vector Machine	0.022	0.625	0.625	0.151	0.625	0.167	0.102	0.615
6	Decision Trees	0.224	0.585	0.591	0.298	0.583	0.498	0.379	0.588
7	Random Forest	0.253	0.625	0.626	0.288	0.614	0.459	0.452	0.632
8	XGBoost	0.328	0.631	0.631	0.433	**0.674**	0.414	0.612	0.662
	US−XGBoost	**0.687**	**0.667**	**0.692**	**0.655**	**0.669**	**0.601**	**0.693**	0.696

Numbers in bold indicate the best models for each Jureczko dataset

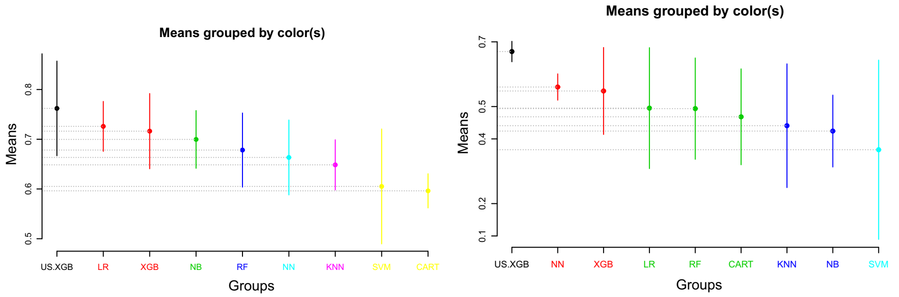


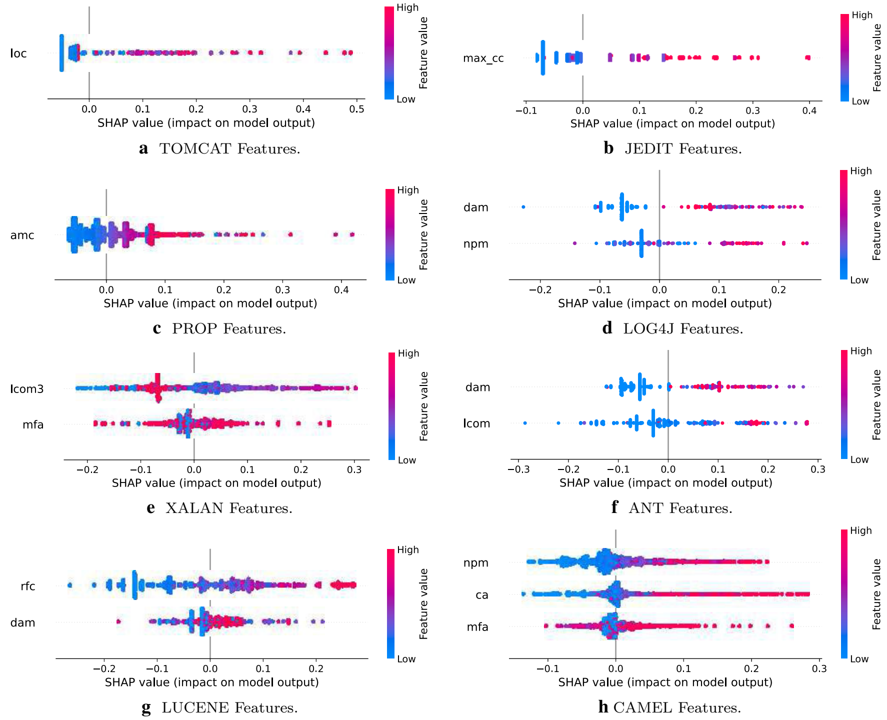
Fig. 1 Scott-Knott ESD test for the Jureczko dataset. AUC Numbers (On the Left) and F1 Score (On the Right)

the probability of a defect. Hence, this reveals that for the *tomcat* project (Fig. 2a), higher Lines of Code (LOC) values increase the chance of our model predicting defects in that specific project.

Machine learning decisions for the TOMCAT, JEDIT, and PROP projects are explained with solely one feature. Four of the selected projects used only two features (LOG4J, XALAN, ANT, and LUCENE), and the remaining project used three features (CAMEL). We also remark that important features may vary depending on the project. Some of the most relevant features derived were LOC, AMC (Average Method Complexity), Data Access Metric (DAM), Response for a Class (RFC), and Number of Public Methods (NPM).

These results indicate that our approach (US-XGB) to search the feature space is relevant to improve the AUC numbers as discussed in Sect. 4.1. Moreover, this algorithm also contributed to the generation of explainable models (Fig. 2). The models that we create are composed of up to only three features. Therefore, these models are simpler than models generated for the exploration of the entire feature space. We argue that models composed of fewer features are more explainable because if a developer received the model explanation derived from Fig. 2, it would be easier for them to work on the features that are producing more defects in the specific project. For example, a *camel* developer could use our results to acknowledge that the number of public methods (i.e., NPM feature) (Fig. 2) may contribute to defects in that project. Also, the SHAP graph indicates that higher numbers of NPM are the main cause of defects. With this information, the developer could use these insights to work on the reduction of public methods for the *camel* project.

RQ2. We conclude that the optimal number of features to predict software defects is never the full pool of features. The best-performing models were composed by up to 3 features.



a TOMCAT Features.

b JEDIT Features.

c PROP Features.

d LOG4J Features.

e XALAN Features.

f ANT Features.

g LUCENE Features.

h CAMEL Features.

Fig. 2 Best overall performing models for each dataset

4.3 Accuracy and variability of software features

In this section, we explore the following RQ3: *"How comparable is the predictive accuracy and variability of features in defect prediction models?"*. To determine the predictive accuracy of software features, we quantified all models that included the target feature. In this case, the predictive accuracy represents the average AUC number of all models that included the target feature. Similarly, the variability accounts for the average Mean Absolute Deviation (MAD) value of all models that incorporate the software feature. Again, our implementation (US−XGB) generated millions of models. Figure 3 shows the predictive accuracy and variability of the target software features. Specifically, around 3.5% of the features are part of models in which the average AUC numbers are higher than 82%. Our approach to feature selection associated most of the features with significantly lower average AUC numbers (only around 77%). We observe a similar leaning while investigating the distribution of features taking into consideration the model variability. In this case, around 3% of the features associate to models with low variability.

From the best-performing models in terms of accuracy (AUC numbers), 73% of features relate to the Object-Oriented metrics, the remaining 27% relates to CK metrics. In terms of individual features, Measure Functional Abstraction (MFA) feature appeared in around 14% of the best models. Completing the top-3, Lines of Code

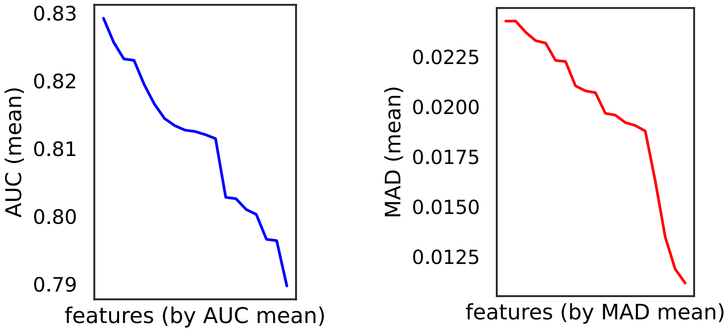


Fig. 3 Distribution of features in the models. Left—Predictive accuracy (AUC numbers). Right—Variability (MAD)

(LOC) feature appeared in about 12% of the models, while the Cohesion Among Methods of a Class (CAM) feature was in nearly 11% of the models.

In terms of feature variability, we take the models with the lowest variability. We note that around 67% of these models are composed of Object-Oriented features like the ones described in Table 2. The remaining 33% of the software features relate to the CK metrics (Table 2). Among those features, the most significant occurrence was the Number of Children of a class (NOC) feature. This feature appeared in around 13% of the top 10% models. Closing the top-3 features from the generated models, the Measure of Aggregation (MOA) feature appeared in around 12% of the models, and the Inheritance Coupling (IC) features appeared in 10% of the models. Note that even though the CK metrics represented only 33% of the total features in the lowest variability models, the NOC feature, which is a CK metric, is the top feature in these models.

The experiments with predictive accuracy and variability confirmed two important characteristics of our approach to model selection. First, a small set of features produces accurate models based on AUC numbers. Second, the software metrics vary greatly among the most accurate models. Features related to Object-Oriented are prominent in the top-performing models, and they produce almost 68% of the important models for the variability and almost 75% for the accuracy numbers.

> RQ3. We conclude that the accuracy and variability of software features vary greatly among the best-performing models.

5 Threats to validity

The study presented in this paper has some limitations that could potentially threaten our results, as we discuss next. First, we present the external threats to validity. Then, we review the internal threats. And finally, we examine the construct threats to validity and the conclusion validity.

5.1 External Validity

Threats to external validity are conditions that limit our ability to generalize the results of our paper (Wohlin et al. 2012). In our study, a threat to the external validity of our research is related to the limited number of projects we analyzed. Furthermore, all projects are related to the Java programming language. As a result, the results may not generalize to other projects especially when they are developed in other programming languages. Furthermore, our results depend on defects within the project context. Thus, we could not draw any conclusion about cross-project defects. As a future step of this paper, we would like to apply our approach to model selection to distinct scenarios.

Furthermore, we applied a limited number of baseline algorithms to classify a software module as defective. Hence, we are not able to guarantee that our results are generalized in all existing classification algorithms. For this reason, we may study other classification algorithms in future steps of this paper.

5.2 Internal validity

Threats to internal validity are influences that can affect the independent variable to causality (Wohlin et al. 2012). In our context, this threat refers to the chosen datasets, we naively applied the data reported in (Jureczko and Madeyski 2010; Jureczko and Spinellis 2010). However, we could not validate the data in terms of how it was obtained by the authors. For example, the data may be incomplete or even wrongly collected. We follow machine learning techniques to mitigate the effects of imbalanced data, as in the case of cross-validation, but we cannot guarantee that the data reflects the actual nature of the eight Java projects applied in our study.

5.3 Construct validity

Construct validity concerns inferring the result of the experiments to the concept or theory (Wohlin et al. 2012). The current literature accepts SHAP values as an agnostic method to explain machine learning models (Lundberg et al. 2018). However, other methods in the literature may have different explanations based on a series of characteristics. For instance, LIME and BreakDown have already been discussed in the current literature about software defect prediction (Jiarpakdee et al. 2020). At this moment, we could not guarantee that the results are replicable using these tools (i.e., LIME and BreakDown). Again, we may use these tools in the future works of this paper.

5.4 Conclusion validity

Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion between the treatment and the outcome (Wohlin et al. 2012). In our study, this threat also relates to SHAP values. Our explanations provided by SHAP depend upon the defect labels of the Jureczko datasets (Jureczko and Spinellis

2010; Jureczko and Madeyski 2010). Other studies (Yatish et al. 2019) discovered that many datasets rely on the six months post-release window period to predict a defect effectively when compared to the use of affected releases of issue reports, like the ones used in Jureczko datasets.

6 Conclusions and future work

We explored the space for software defect prediction models using an efficient implementation of the XGBoost algorithm, named US−XGBoost, which resulted in millions of random models. We evaluated these models considering their accuracy and interpretability. We found that 3.5% of the models (out of 1 997 287) are superior to the seven classic baseline models in the Jureczko datasets.

Our findings also indicate that software defect prediction is a project-specific task, i.e., features composing the best performing models may vary greatly depending on the project. Thus, it is particularly important to understand the factor contributing to model decisions. We used SHAP values to explain model decisions, and we found that best performing models are very simple to understand, being composed of few features and well-distributed values. Thus, model explanations may provide insight on which features of the code are more prone to defect.

As future work, we want to mine data from public repositories on Github. These data could be labeled and then analyzed using the same models applied in this research. Thus, we would provide to the community additional case studies of the model proposed in this paper. Another case study to validate the present research would be a qualitative research with developers searching for how the explanations provided in this paper would support real projects. The output of this study could even be used to propose a tool for developers to analyze their projects and check which features may indicate defective classes. Furthermore, the public data available on Github may be analyzed from many different perspectives in terms of predicting software defects using machine learning. For instance, we would like to classify commits in order to generate a temporal analysis concerned not only to the defects but also to the bugs and the inclusion of test cases on public source code.

Acknowledgements We thank the support given by the Project: Models, Algorithms and Systems for the Web (Grant by FAPEMIG/PRONEX/MASWeb APQ-01400- 14) and authors' individual grants and scholarships from CNPq and Fapemig.

References

Agrawal, A., Menzies, T.: Is better data better than better data miners? On the benefits of tuning smote for defect prediction. In: International Conference of Software Engineering (ICSE), pp. 1050–1061 (2018)

Chen, T., Guestrin, C.: Xgboost: a scalable tree boosting system. In: International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp. 785–794 (2016)

Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)

Couto, C., Silva. C., Valente, M.T., Bigonha, R., Anquetil, N.: Uncovering causal relationships between software metrics and bugs. In: 2012 16th European Conference on Software Maintenance and Reengineering (2012)

D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (2010)

Elish, K.O., Elish, M.O.: Predicting defect-prone software modules using support vector machines. J. Syst. Softw. **81**(5), 649–660 (2008)

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, pp. 12–21 (2018)

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N.: An empirical study of just-in-time defect prediction using cross-project models. In: Working Conference on Mining Software Repositories (MSR), pp. 172–181 (2014)

Ghotra, B., McIntosh, S., Hassan, A.E.: Revisiting the impact of classification techniques on the performance of defect prediction models. In: IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), vol 1, pp. 789–800 (2015)

Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: Using the support vector machine as a classification method for software defect prediction with static code metrics. In: International Conference on Engineering Applications of Neural Networks (EANN), pp. 223–234 (2009)

Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: 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 2011), pp. 96–103 (2011)

Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc, Amsterdam (1977)

Herbold, S.: Crosspare: a tool for benchmarking cross-project defect predictions. In: 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW) (2015)

Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 279–289 (2013)

Jiarpakdee, J., Tantithamthavorn, C., Dam, H.K., Grundy, J.: An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans. Softw. Eng. pp 1–1 (2020)

Jing, X.Y., Ying, S., Zhang, Z.W., Wu, S.S., Liu, J.: Dictionary learning based software defect prediction. In: International Conference of Software Engineering (ICSE), pp. 414–423 (2014)

Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp. 9:1–9:10 (2010)

Jureczko, M., Spinellis, D.D.: Using object-oriented design metrics to predict software defects. In: In Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej, pp. 69–81 (2010)

Knab, P., Pinzger, M., Bernstein, A.: Predicting defect densities in source code files with decision tree learners. In: Proceedings of the International Workshop on Mining Software Repositories (MSR), MSR, pp. 119–125 (2006)

Kuhn, M.: Caret: Classification and regression training. http://topepo.github.io/caret/index.html (2015)

Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., Whitehead Jr, EJ.: Does bug prediction support human developers? findings from a google case study. In: International Conference of Software Engineering (ICSE), pp. 372–381 (2013)

Lundberg, S.M., Lee, S.: Consistent feature attribution for tree ensembles. arXiv:1706.06060 (2017a)

Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: Annual Conference on Neural Information Processing Systems (NIPS) (2017b)

Lundberg, S.M., Erion, G.G., Lee, S.: Consistent individualized feature attribution for tree ensembles. arXiv:1802.03888 (2018)

McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)

McCabe, T.J., Butler, C.W.: Design complexity measurement and testing. Commun. ACM **32**(12), 1415–1425 (1989)

Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. **1**, 2–13 (2007)

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Automated Softw. Eng. **17**(4), 375–407 (2010)

Mori, T., Uchihira, N.: Balancing the trade-off between accuracy and interpretability in software defect prediction. Empir. Softw. Eng. **24**, 779–825 (2018)

Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 284–292 (2005)

Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, pp. 452–461 (2006)

Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N.: The jinx on the nasa software defect data sets. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE (2016)

Sayyad Shirabad, J., Menzies, T.: The PROMISE Repository of Software Engineering Databases. University of Ottawa, Canada, School of Information Technology and Engineering (2005)

Shapley, L.S.: A value for n-person games. In: Kuhn, H.W., Tucker, A.W. (eds.) Annals of Mathematical Studies, pp. 307–317. Princeton University Press, Princeton (1953)

Shuai, B., Li, H., Li, M., Zhang, Q., Tang, C.: Software defect prediction using dynamic support vector machine. In: Ninth International Conference on Computational Intelligence and Security, pp. 260–263 (2013)

Sokolova, M., Japkowicz, N., Szpakowicz, S.: Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. AI 2006: Advances in Artificial Intelligence pp. 1015–1021 (2006)

Stites, R.L., Ward, B., Walters, R.V.: Defect prediction with neural networks. In: Proceedings of the Conference on Analysis of Neural Network Applications, ANNA, pp. 199-206 (1991)

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

Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**(10), 897–910 (2005)

Tantithamthavorn, C., Hassan, A.E.: An experience report on defect modelling in practice: pitfalls and challenges. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 286–295 (2018)

Tantithamthavorn, C., McIntosh, S., Hassan, AE., Ihara, A., Matsumoto, K.: The impact of mislabelling on the performance and interpretation of defect prediction models. In: International Conference on Software Engineering (ICSE), pp. 812–823 (2015)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models (2017)

Tantithamthavorn, C., McIntosh, S., Hassan, AE., Matsumoto, K.: The impact of automated parameter optimization for defect prediction models (2018)

Thwin, M.M.T., Quah, T.S.: Application of neural networks for software quality prediction using object-oriented metrics. J. Syst. Softw. **76**(2), 147–156 (2005)

Turhan, B., Bener, A.: Analysis of naive bayes' assumptions on software fault data: an empirical study. Data Knowl. Eng. **68**(2), 278–290 (2009)

Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. **14**(5), 540–578 (2009)

Wang, D., Yang, Q., Abdul, A., Lim, B.Y.: Designing theory-driven user-centric explainable ai. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19, pp. 1–15 (2019)

Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: International Conference of Software Engineering (ICSE), pp. 297–308 (2016)

Wang, T., Li, W.H.: Naive bayes software defect prediction model. In: International Conference on Computational Intelligence and Software Engineering (CiSE), pp. 1–4 (2010)

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer, Berlin (2012)

Xu, Z., Liu, J., Luo, X., Zhang, T.: Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 209–220 (2018)

Xuan, X., Lo, D., Xia, X., Tian, Y.: 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 (2015)

Yang, Y., Zhou, Y., Liu, J,. Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H.: 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, FSE, pp. 157–168 (2016)

Yatish, S., Jiarpakdee, J., Thongtanunam, P., Tantithamthavorn, C.: Mining software defects: Should we consider affected releases? In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 654–665 (2019)

Zhang, F., Hassan, A.E., McIntosh, S., Zou, Y.: The use of summation to aggregate software metrics hinders the performance of defect prediction models. IEEE Trans. Softw. Eng. **43**(5), 476–491 (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Geanderson Esteves[1] · **Eduardo Figueiredo**[2] · **Adriano Veloso**[2] · **Markos Viggiato**[3] · **Nivio Ziviani**[1]

✉ Geanderson Esteves
geanderson@dcc.ufmg.br

Eduardo Figueiredo
figueiredo@dcc.ufmg.br

Adriano Veloso
adrianov@dcc.ufmg.br

Markos Viggiato
viggiato@ualberta.ca

Nivio Ziviani
nivio@dcc.ufmg.br

[1] Department of Computer Science, Universidade Federal de Minas Gerais and Kunumi, Belo Horizonte, Brazil

[2] Department of Computer Science, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

[3] Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada