Dynamic Selection of Classifiers in Bug Prediction: an Adaptive Method

Dario Di Nucci[1], Fabio Palomba[2], Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy — [2]TU Delft, The Netherlands — [3]University of Molise, Italy

Abstract—In the last decades the research community has devoted a lot of effort in the definition of approaches able to predict the defect proneness of source code files. Such approaches exploit several predictors (*e.g.*, product or process metrics) and use machine learning classifiers to predict classes into buggy or not buggy, or provide the likelihood that a class will exhibit a fault in the near future. The empirical evaluation of all these approaches indicated that there is no machine learning classifier providing the best accuracy in any context, highlighting interesting complementarity among them. For these reasons ensemble methods have been proposed to estimate the bug-proneness of a class by combining the predictions of different classifiers. Following this line of research, in this paper we propose an adaptive method, named ASCI (Adaptive Selection of Classifiers in bug predIction), able to dynamically select among a set of machine learning classifiers the one which better predicts the bug proneness of a class based on its characteristics. An empirical study conducted on 30 software systems indicates that ASCI exhibits higher performances than 5 different classifiers used independently and combined with the majority voting ensemble method.

Keywords—Bug Prediction, Classifier Selection, Ensemble Techniques

1 Introduction

Continuous changes, close deadlines, and the need to ensure the correct behaviour of the functionalities being issued are common challenges faced by developers during their daily activities [1]. However, limited time and manpower represent serious threats to the effective testing of a software system. Thus, the resources available should be allocated effectively upon the portions of the source code that are more likely to contain bugs. One of the most powerful techniques aimed at dealing with the testing-resource allocation is the creation of *bug prediction models* [2] which allow to predict the software components that are more likely to contain bugs and need to be tested more extensively.

Roughly speaking, a bug prediction model is a supervised method where a set of independent variables (the predictors) are used to predict the value of a dependent variable (the bug-proneness of a class) using a machine learning classifier (*e.g.,* Logistic Regression [3]). The model can be trained using a sufficiently large amount of data available from the project under analysis, *i.e., within-project* strategy, or using data coming from other (similar) software projects, *i.e., cross-project* strategy.

A factor that strongly influences the accuracy of bug prediction models is represented by the classifier used to predict buggy components. Specifically, Ghotra *et al.* [4] found that the accuracy of a bug prediction model can increase or decrease up to 30% depending on the type of classification applied [4]. Also, Panichella *et al.* [5] demonstrated that the predictions of different classifiers are highly complementary despite the similar prediction accuracy.

Based on such findings, an emerging trend is the definition of prediction models which are able to combine multiple classifiers (a.k.a., *ensemble* techniques [6]) and their application to bug prediction [5], [7], [8], [9], [10], [11], [12], [13]. For instance, Liu *et al.* [7] proposed the *Validation and Voting* (VV) strategy, an approach where the prediction of the bug-proneness of a class is assigned by considering the output of the majority of the classifiers. Panichella *et al.* [5] devised CODEP, an approach that uses the outputs of 6 classifiers as predictors of a new prediction model, which is trained using Logistic Regression (LOG). However, as highlighted by Bowes *et al.* [14], traditional ensemble approaches miss the predictions of a large part of bugs that are correctly identified by a single classifier and, therefore, *"ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of bugs"* [14]. An alternative approach to deal with non homogeneous data in the context of cross-project bug prediction is the *local bug prediction* [12]. This technique firstly clusters homogeneous data and then builds, for each of them, a different model using the same classifier. Unfortunately, local bug prediction is hardly applicable in the context of within-project bug prediction because it tends to create too small clusters hardly usable as training sets.

Based on the results of these previous approaches, we conjecture that a successful way to combine classifiers can be obtained by *choosing the most suitable classifier based on the characteristics of classes, rather than combining the output of different classifiers*. From this point of view, our idea is different than other ensemble methods, but also different than *local bug prediction* [12] as we build dif-

ferent models for homogeneous dataset using different classifiers.

To verify this conjecture, we propose a novel adaptive prediction model, coined as **ASCI** (**A**daptive **S**election of **C**lass**I**fiers in bug prediction), which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class (*i.e.,* product metrics). Specifically, given a set of classifiers our approach firstly trains these classifiers using the structural characteristics of the classes in the training set, then a decision tree is built where the internal nodes are represented by the structural characteristics of the classes contained in the training set, and the leafs are represented by the classifiers able to correctly classify the bug-proneness of instances having such structural characteristics. In other words, we use a decision tree learning approach to train a classifier able to predict which classifier should be used based on the structural characteristics of the classes.

To build and evaluate our approach, we firstly carry out a preliminary investigation aimed at understanding whether a set of five different classifiers, *i.e.,* Binary Logistic Regression (LOG), Naive Bayes (NB), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), and Decision Trees (DTree), is complementary in the context of within-project bug prediction: the study is designed to answer the following research question:

- **RQ$_0$:** *Are different classifiers complementary to each other when used in the context of within-project bug prediction?*

Our results corroborate previous findings achieved in the context of cross-project bug prediction [4], [5], indicating high complementarity among the five different experimented classifiers.

Afterwards, we experimented the proposed adaptive method on the data of 30 software systems extracted from the PROMISE repository [15], comparing the accuracy achieved by ASCI with the ones obtained by (i) the bug prediction models based on each of the five classifiers independently, and (ii) the VV ensemble technique combining the predictions of the five classifiers through majority voting. Specifically, our second study is steered by the following research questions:

- **RQ$_1$:** *Does the proposed adaptive technique outperform stand-alone classifiers?*
- **RQ$_2$:** *Does the proposed adaptive technique outperform the Validation and Voting ensemble technique?*

The results of our study highlight the superiority of ASCI with respect to all the baselines, indicating that the use of the adaptive model increases the F-measure of the predictions up to 7% with respect to the the best model built using a single classifiers, and up to 5% with respect to the VV model.

Structure of the paper. Section 2 discusses the background and the related literature. Section 3 presents ASCI in details. In Section 4 we report the preliminary study aimed at investigating the complementarity of classifiers, while Section 5 reports the design and the results of the empirical evaluation of the proposed approach. We discuss possible threats that could affect the validity of our empirical study in Section 6, before concluding the paper in Section 7.

2 Background and Related Work

Most of the work on bug prediction refers to the definition of models using different types of predictors, *e.g.,* CK metrics [16], process metrics [17], history-based metrics [18], [19]. Extensive surveys of the different approaches proposed in the literature can be found in [1] and [2]. In the following, we overview the main approaches based on single classifiers, the differences between within- and cross-project bug prediction, and the use of ensemble techniques in these contexts.

Classifiers for Bug Prediction. Several machine learning classifiers have been used in literature [2], *e.g.,* Logistic Regression (LOG), Support Vector Machines (SVM), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), Bayesian Network (BN), Decision Trees (DTree), and Decision Tables (DTable).

However, results of previous studies demonstrated no clear winner among these classifiers [20], [21], [22], [23], [24]. In particular, depending on the dataset employed, researchers have found different classifiers achieving higher performances with respect to the others, *i.e.,* (i) RBF and its modified version, namely RBF trained with enhanced Dynamic Decay Adjustment algorithm (RBF-eDDA) [20], [21], (ii) Dynamic Evolving Neuro-Fuzzy Inference System (DENFIS), Support Vector Regression (SVR), and Regression Tree (RT) [22], (iii) ADTrees [23], and (iv) Naive Bayes and Multilayer Perceptron [24].

Moreover, Lessman *et al.* [25] conducted an empirical study with 22 classification models to predict the bug proneness of 10 publicly available software development data sets from the NASA repository, reporting no statistical differences among the top-17 models. As demonstrated by Shepperd *et al.* [26], the NASA dataset that is used in the work by Lessman *et al.* [25] was noisy and biased. A subsequent investigation on the cleaned NASA dataset performed by Ghotra *et al.* [4] found that the impact of classifiers on the performance of a bug prediction model is instead relevant. Indeed, the performance of a bug prediction model can increase or decrease up to 30% depending on the type of classifier applied [4].

Within- vs. Cross-Project Bug Prediction. Prediction approaches can be defined by training a classification model on past data of the same software project (*within-project* strategy) [8], [9], [10], [14], [20], [21], [22], [23], [24], [27] or belonging to different projects (*cross-project* strategy) [5], [7], [11], [12], [13], [28], [29], [30].

Each strategy has its pros and cons. The within-project strategy can be applied only on mature projects, where a sufficiently large amount of project history (*i.e.,* past

faults) is available. Thus, such a strategy cannot be used on new projects. In this scenario, the cross-project strategy can be used. The main problem of the cross-project strategy is represented by the heterogeneity of data. Even if some approaches [11], [12] try to mitigate such a problem, the within-project strategy should still be preferred when sufficiently large amount of data is available. This means that the two strategies are *de facto* complementary to each other.

Ensemble Techniques for Bug Prediction. Ensemble techniques aim at combining different classifiers to achieve better classification performances.

Misirli *et al.* [8] used the Validation and Voting ensemble technique to combine different classifiers in the context of within-project [8] bug prediction. This technique can be considered as a Boosting [6] specialization where a function is applied on the output of the classifiers to improve the prediction performances. In the case of Validation and Voting [7], if the majority of models (obtained using different classifiers on the same training set) predicts an entity as bug-prone, then it is predicted as bug-prone; otherwise, it is predicted as non bug-prone. Wang *et al.* [9] compared the performances achieved by 7 ensemble techniques in the context of within-project bug prediction, showing that often Validation and Voting stand out among them. With the same aim Liu *et al.* [7] experimented 17 different machine learning models in the context of cross-project bug prediction reaching similar conclusions.

Other ensemble techniques have also been applied. Kim *et al.* [10], He *et al.* [11], and Menzies *et al.* [12] proposed approaches similar to the Bagging ensemble technique [6] which combines the outputs of different models trained on a sample of instances taken with a replacement from the training set. Kim *et al.* [10] combined multiple training data obtained applying a random sampling in the context of within-project bug prediction. He *et al.* [11] proposed an approach for automatically select training data from other projects in the cross-project context. Menzies *et al.* [12] introduced the concept of local bug prediction, namely an approach in which classes that will be used for training the classifier are firstly clustered into homogeneous groups in order to reduce the differences among such classes. Leveraging on Boosting, Xia *et al.* [31] devised HYDRA. This approach combines different models, obtained from different sources, using Boosting (*e.g.*, AdaBoost).

Recently some approaches [5], [13], [32] have been proposed based on the Stacking ensemble technique [6], which uses a meta-learner to induce which classifiers are reliable and which are not. These techniques use the predicted classifications by the classifier as input. In the context of cross-project defect prediction, Panichella *et al.* [5] devised an approach, named CODEP, which firstly applies a set of classification models independently, afterwards it uses the output of the first step as predictors of a new prediction model, which is trained

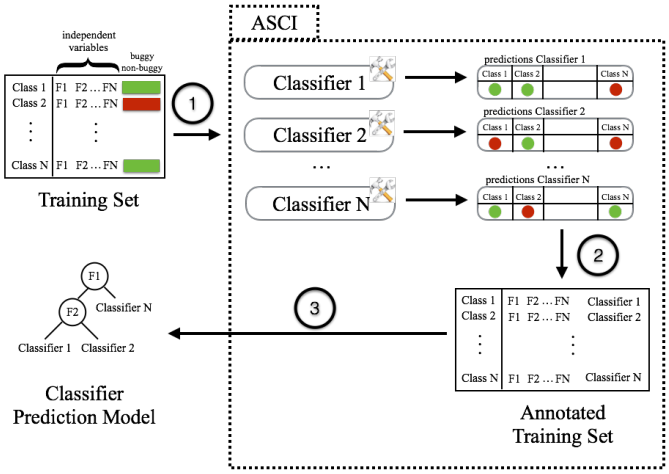

Fig. 1: The workflow of ASCI.

using LOG [3]. Zhang *et al.* [28] conducted a study similar to the one performed by Panichella *et al.* [5] comparing different ensemble approaches. Their results showed that several ensemble techniques improve the performances achieved by CODEP and that often Validation and Voting performs better. Petric *et al.* [13] used 4 families of classifiers in order to build a Stacking ensemble technique [6] based on the diversity among classifiers in the cross-project context. Their empirical study showed that their approach can perform better than other ensemble techniques and that the diversity among classifiers is an essential factor. In the context of just-in-time defect prediction, Yang *et al.* [32] proposed TLEL. This technique firstly trains different Random Forest models using Bagging and then combines them using Stacking.

To some extent, ASCI is similar to Stacking [6], because it builds a meta-model combining a set of base models. However, unlike Stacking, our meta-model classifier does not use the predictions of the base classifiers to better predict the bugginess of a class, but uses the characteristics of the class (*i.e.*, the original predictors) to predict the classifier able to better predict the bug-proneness of the class. For the same reason ASCI is different from Validation and Voting, and Boosting. Moreover it is different with respect to Bagging as it does not work on the composition of the training set.

3 ASCI: An Adaptive Method for Bug Prediction

In this section we present ASCI (**A**daptive **S**election of **C**lassifiers in bug pred**I**ction), our solution to dynamically select classifiers in bug prediction. The implementation of ASCI is available in our online appendix [33].

Figure 1 depicts the three main steps that our approach employs to recommend which classifier should be used to evaluate the bugginess of a given class. In particular:

1) Let $C = \{c_1, ..., c_n\}$ be a set of n different classifiers, and let $T = \{e_1, ..., e_m\}$ be the set of classes composing the training set. Each $c_i \in C$ is experimented

against the set T, in order to find its configuration. As an example, for the *Logistic Regression* [3], this step will configure the parameters to use in the logistic function. At the same time, each $c_i \in C$ outputs the predictions regarding the bug-proneness of each $e_j \in T$. Note that the proposed technique is independent from the underlying pool of classifiers, however it would be desirable that the base classifiers exhibit some level of complementarity. The time efficiency of this step is influenced by (i) the number and type of classifiers to configure and (ii) the size of the training set: however, on systems similar to those we analyzed in the evaluation of the model (see Section 5) it requires few seconds.

2) At the end of the first step, each $e_j \in T$ is labeled with the information about the best classifier $c_i \in C$ which correctly identified its bugginess. In this stage, there are two possible scenarios to consider. If a unique machine learning classifier c_i is able to predict the bug-proneness of e_j, then e_j will be associated with c_i. On the other hand, if more classifiers or none of them correctly identified the bugginess of e_j, we assign to e_j the classifier c_i having the highest F-Measure on the whole training set. The output of this step is represented by an annotated training set T'. Note that, while there would be other alternatives to build the annotated training set (*e.g.,* the usage of multi-label classifiers [34]), we empirically evaluated them observing that our solution results in higher performances. A detailed comparison between our approach, the multi-label solution, and a baseline where the choice is made randomly is available in our online appendix [33].

3) Finally, based on T', we build a *classifier prediction model* using a decision tree DT as classifier. Specifically, given the structural characteristics of the classes in the annotated training set (independent variables), the goal of this final step is to build a model able to predict the classifier $c_i \in C$ to use (dependent variable). In other words, the role of the DT is to predict a nominal variable indicating the name of the classifier $c_i \in C$ most suitable for classifying a class that is characterized by the structural properties reported as independent variables. Note that we decide to use a decision tree learning approach since a decision tree captures the idea that if different decisions were to be taken, then the structural nature of a situation (and, therefore, of the model) may have changed drastically [35]. This is in line with what we want to obtain, namely a model where a change in the structural properties of a class implies a different evaluation of suitability of a classifier. In order to build DT, we propose to use Random Forest [36], a classifier widely used in literature built from a combination of tree predictors.

Once the adaptive model has been built, the bugginess of a new class is predicted by using the classifier that the DT has selected to be the most suitable one and not all the base classifiers as required by other ensemble techniques, such as VV, Boosting, and Stacking.

4 On the Complementarity of Machine Learning Classifiers

This section describes the design and the results of the empirical study we conducted in order answer our $\mathbf{RQ_0}$ with the purpose of verifying whether the investigated classifiers are complementary and thus good candidates for being combined by ASCI.

- **$\mathbf{RQ_0}$:** *Are different classifiers complementary to each other when used in the context of within-project bug prediction?*

TABLE 1: Characteristics of the software systems used in the study

#	Project	Release	Classes	KLOC	Buggy Classes	(%)
1	Ant	1.7	745	208	166	22%
2	ArcPlatform	1	234	31	27	12%
3	Camel	1.6	965	113	188	19%
4	E-Learning	1	64	3	5	8%
5	InterCafe	1	27	11	4	15%
6	Ivy	2.0	352	87	40	11%
7	jEdit	4.3	492	202	11	2%
8	KalkulatorDiety	1	27	4	6	22%
9	Log4J	1.2	205	38	180	92%
10	Lucene	2.4	340	102	203	60%
11	Nieruchomosci	1	27	4	10	37%
12	pBeans	2	51	15	10	20%
13	pdfTranslator	1	33	6	15	45%
14	Poi	3.0	442	129	281	64%
15	Prop	6.0	660	97	66	10%
16	Redaktor	1.0	176	59	27	15%
17	Serapion	1	45	10	9	20%
18	Skarbonka	1	45	15	9	20%
19	SklepAGD	1	20	9	12	60%
20	Synapse	1.2	256	53	86	34%
21	SystemDataManagement	1	65	15	9	14%
22	SzybkaFucha	1	25	1	14	56%
23	TermoProjekt	1	42	8	13	31%
24	Tomcat	6	858	300	77	9%
25	Velocity	1.6	229	57	78	34%
26	WorkFlow	1	39	4	20	51%
27	WspomaganiePI	1	18	5	12	67%
28	Xalan	2.7	909	428	898	99%
29	Xerces	1.4	588	4	437	74%
30	Zuzel	1	39	14	13	45%

4.1 Empirical Study Design

The *goal* of the empirical study is to assess the complementarity of different classifiers when used to predict bugs at class level granularity, with the *purpose* of investigating whether they classify different sets of software components as bug-prone. The *quality focus* is on the improvement of the effectiveness of bug prediction approaches in the context of *within-project* bug prediction, while the *perspective* is of a researcher who is interested to understand to what extent different classifiers complement each other when used to predict buggy classes. Indeed, if classifiers are complementary it could be worth to combine them using an ensemble technique. The *context* of the study consists of 30 software systems from the `Apache Software Foundation` ecosystem[1]. Table 1 reports the specific release taken into account as well as the characteristics of the projects considered

1. http://www.apache.org

in the study in terms of size, expressed as number of classes and KLOC, and number and percentage of buggy classes. All the systems are publicly available in the PROMISE repository [15], which provides for each project (i) the independent variables, *i.e.,* LOC and CK metrics [37], and (ii) the dependent variable, *i.e.,* a boolean value indicating whether a class is buggy or not.

In order to answer $\mathbf{RQ}_0$, we run five different machine learning classifiers [3], namely Binary Logistic Regression (LOG), Naive Bayes (NB), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), and Decision Trees (DTree). The selection of the machine learning classifiers is not random. On the one hand, they have been used in many previous work on bug prediction [5], [7], [14], [21], [24], [28], while on the other hand they are based on different learning peculiarities (*i.e.,* regression functions, neural networks, and decision trees). This choice increases the generalizability of our results.

As evaluation procedure, we adopt the 10-fold cross validation strategy [38]. This strategy randomly partitions the original set of data, *i.e.,* data of each system, into 10 equal sized subset. Of the 10 subsets, one is retained as *test* set, while the remaining 9 are used as *training* set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the *test* set exactly once [38]. We use this test strategy since it allows all observations to be used for both training and test purpose, but also because it has been widely-used in the context of bug prediction (e.g., see [39], [40], [41], [42]).

As evaluation methodology, we firstly evaluate the performances of the experimented classifiers using widely-adopted metrics, such as accuracy, precision and recall [43]. In addition, we also computed (i) the F-measure, *i.e.,* the harmonic mean of precision and recall, and (ii) the Area Under the Curve (AUC), which quantifies the overall ability of a prediction model to discriminate between buggy and non-buggy classes. Note that the analysis of the accuracy achieved by different classifiers is necessary to corroborate previous findings which report how different classifiers exhibit similar accuracy, even if they are complementary to each other [5], [14].

Due to space limitations, we report and discuss the boxplots of the distributions of the accuracy, the F-Measure, and the AUC achieved by the single classifiers independently on the 30 considered systems. A complete report of the results is available in our online appendix [33]. Furthermore, we verified whether the differences are statistically significant by exploiting the Mann-Whitney U test [44]. The results are intended as statistically significant at $\alpha = 0.05$. We also estimated the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [45] for ordinal data. We followed the guidelines in [45] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

After the analysis of the performance of the different classifiers, we analyze their complementarity by computing the overlap metrics. Specifically, given two sets of predictions obtained by using classifiers c_i and c_j, we compute:

$$corr_{c_i \cap c_j} = \frac{|corr_{c_i} \cap corr_{c_j}|}{|corr_{c_i} \cup corr_{c_j}|}\% \quad (1)$$

$$corr_{c_i \setminus c_j} = \frac{|corr_{c_i} \setminus corr_{c_j}|}{|corr_{c_i} \cup corr_{c_j}|}\% \quad (2)$$

where $corr_{c_i}$ represents the set of bug-prone classes correctly classified by the classifier c_i, $corr_{c_i \cap c_j}$ measures the overlap between the set of buggy classes correctly identified by both classifiers c_i and c_j, and $corr_{c_i \setminus c_j}$ measures bug-prone classes correctly classified by c_i only and missed by c_j. Also in this case, we aggregated the results of the 30 considered systems by summing up the single $corr_{c_i \cap c_j}$, $corr_{c_i \setminus c_j}$, and $corr_{c_j \setminus c_i}$ obtained after the evaluation of the complementarity between two classifiers on a system. The fine-grained results are available in our online appendix [33].

4.2 Analysis of the Results

The results achieved running the stand-alone classifiers over all the considered software systems are reported in Figure 2.

Looking at the boxplots, we can confirm previous findings in the field [5], [14], demonstrating once again that there is no a clear winner among different classifiers in bug prediction. Indeed the differences in the terms of accuracy, F-Measure, and AUC achieved by the classifiers are quite small, as highlighed by the median values presented in Figure 2. Despite this, the average F-Measure (0.56) achieved by MLP is slightly higher with respect to the other classifiers (*i.e.,* NB=+3%, LOG=+4%, RBF=+2%, DTree=+4%). As shown in Table 2 such superiority is statistically significant when considering the differences between the performances of MLP and the ones achieved by LOG and RBF, even though with negligible effect size.

Particularly interesting is the discussion of the results achieved on the `Apache Xalan` project. Here all the classifiers achieve good precision and recall. This is due to the fact that in this project there is 99% of buggy classes. The classifiers can be mainly trained using data related to buggy components and, thus, they do not have enough information to distinguish those components not affected by bugs. However, the presence of an extremely low number of non-buggy classes (11) does not influence too much the performances of the classifiers, which correctly predicted most of the buggy classes.

Another interesting result concerns LOG. As reported in recent papers [1], [5], this classifier is the more suitable in the context of cross-project bug prediction. According to our findings, this is not true when the training set is built using a within-project strategy. A possible explanation of the result is that LOG generally requires

Fig. 2: Boxplots of the accuracy, F-Measure, and AUC-ROC achieved by the single classifiers.

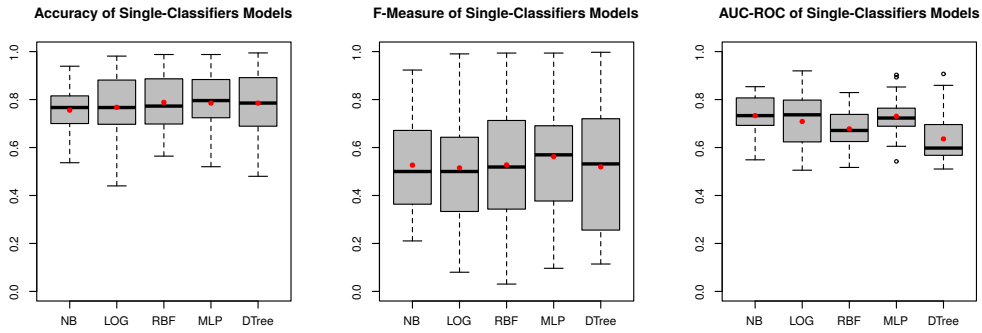

TABLE 2: p-values and Cliff's Delta obtained between each pair of single classifiers. $p-values$ that are statistically significant are reported in bold face.

	NB			LOG			RBF			MLP			DTree		
	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$
NB	-	-	-	0.28	0.03	neg.	0.16	0.04	neg.	0.44	-0.04	neg.	0.34	0.02	neg.
LOG	0.73	-0.03	neg.	-	-	-	0.18	0.01	neg.	0.96	-0.07	neg.	0.64	-0.04	neg.
RBF	0.73	-0.03	neg.	1.00	-0.00	neg.	-	-	-	0.96	-0.07	neg.	0.64	-0.04	neg.
MLP	0.57	0.04	neg.	**0.04**	**0.07**	**neg.**	**0.04**	**0.07**	**neg.**	-	-	-	0.49	0.05	neg.
DTree	0.67	-0.02	neg.	0.36	0.04	neg.	0.36	0.04	neg.	0.52	-0.05	neg.	-	-	-

TABLE 3: Overlap analysis among the classifiers considered in the preliminary study in all the considered classes.

	A=NB			A=Log			A=RBF			A=MLP			A=DTree		
	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A
B=NB	-	-	-	83	12	5	91	5	4	92	4	4	90	5	5
B=Log	83	5	12	-	-	-	83	5	12	81	6	13	80	6	14
B=RBF	91	4	5	83	12	5	-	-	-	89	5	6	89	5	6
B=MLP	92	4	4	81	13	6	89	6	5	-	-	-	90	5	5
B=DTree	90	5	5	80	14	6	89	6	5	90	5	5	-	-	-

much more data in the training set to achieve stable and meaningful results [46]. In the cross-project strategy, the training set is built using a bunch of data coming from external projects, while in the within-project solution the construction of the training set is limited to previous versions of the system under consideration. As a consequence, LOG is not more suitable than other classifiers in this context.

Concerning the analysis of the complementarity, Table 3 summarizes the results achieved. In particular, for each pair of classifiers, the table reports the percentage of classes that are correctly classified as bug-prone by (i) both the compared classifiers (i.e., column A∩B), (ii) only the first classifier (i.e., column A-B), and (iii) only the second classifier (i.e., column B-A). From this table, we can observe that the overlap between the set of correctly classified instances by a classifiers pair is at least 80%, *i.e.*, 80% of the instances are correctly classified by both the classifiers in the comparison. This means that the bug-proneness of most of the classes may be correctly predicted by an arbitrary classifier, while less than the remaining 20% of predictions are correctly classified by only one of the classifiers. However, the majority of the non-overlapping predictions are related to buggy classes (83%): such missing predictions can result in a reduction up to 35% of the performances of a bug prediction model.

More importantly, we noticed that these performances may be increased by analyzing how different classifiers behave on classes having different structural characteristics. As an example of the impact of the characteristics of classes on the effectiveness of bug prediction classifiers, consider the predictions provided by NB and MLP in the `Apache Velocity` project. We found the former more effective in predicting the bugginess of classes having more than 500 lines of code: indeed, although the F-Measure achieved by NB on this system is quite low (35%), the correct predictions refer to large classes containing bugs. An example is the `io.VelocityWriter`, which has 551 LOCs, and a low values cohesion and coupling metrics (*e.g.*, Coupling Between Methods (CBM) = 5). On the other hand, MLP is the classifier obtaining the highest F-Measure (62%), however its correct predictions refer to classes having a high level of coupling: indeed, most of the classes correctly predicted as buggy are the ones having the value of the Coupling Between Methods (CBM) metric higher than 13 and a limited number of lines of code (<400). For instance, the class `velocity.Template` contains 320 LOCs, but it has a CBM = 19. Thus, an adequate selection of the classifiers based on the characteristics of classes may increase the effectiveness of bug prediction.

Summary for RQ$_0$. Despite some differences, the five experimented machine learning classifiers achieve comparable results in terms of accuracy. However, their complementarity could lead to combined models able to achieve better results.

Fig. 3: Boxplots of the accuracy, F-Measure, and AUC-ROC achieved by MLP, VV, and ASCI.

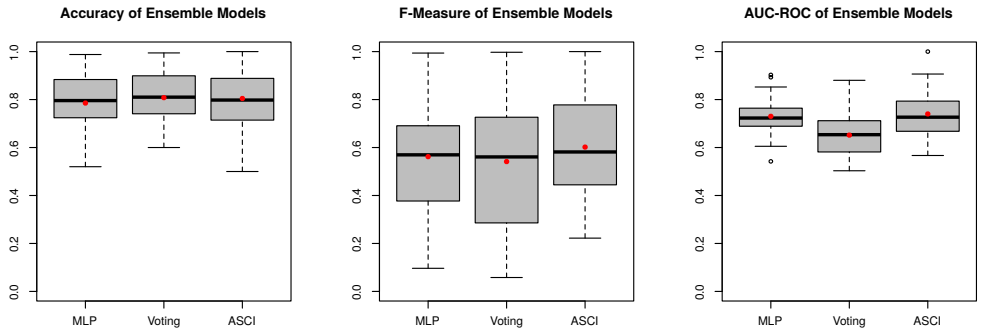

TABLE 4: $p-values$ and Cliff's Delta obtained between VV and ASCI along with MLP as baseline. $p-values$ that are statistically significant are reported in bold face.

	MLP			VV			ASCI		
	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$
MLP	-	-	-	0.91	-0.02	neg.	1.00	-0.17	small
VV	0.09	0.02	neg.	-	-	-	1.00	-0.12	neg.
ASCI	**<0.01**	**0.17**	**small**	**<0.01**	**0.12**	**neg.**	-	-	-

5 Evaluating the Proposed Adaptive Method

In this section, we describe the empirical study conducted in order to evaluate the performances of the proposed adaptive method. In particular, we aim at providing answers to the following **RQs**:

- **RQ_1:** *Does the proposed adaptive technique outperform stand-alone classifiers?*
- **RQ_2:** *Does the proposed adaptive technique outperform the Validation and Voting ensemble technique?*

5.1 Empirical Study Design

The *goal* of the study is to evaluate the proposed adaptive method ASCI in within-project bug prediction and to compare it to the base classifiers, as well as to a different ensemble technique. Consequently, the *quality focus* is on improving the effectiveness of bug prediction approaches, while the *perspective* is of researchers, who want to evaluate the effectiveness of using the proposed adaptive method when identifying bug-prone classes. The *context* of the study consists of the 30 software systems used in Section 4.

In order to answer the two research questions, we used the same classifiers evaluated in Section 4, *i.e.*, NB, LOG, RBF, MLP, and DTree, as basic classifiers for both ASCI and *Validation and Voting*, and the same independent variables (*i.e.*, LOC and CK metrics). Also in this case, we used as evaluation procedure the 10-fold cross-validation strategy [38]. Then, we evaluated the performances achieved by our model with the ones achieved by the base classifiers ($\mathbf{RQ}_1$) and by Validation and Voting ($\mathbf{RQ}_2$) in terms of the metrics used in Section 4, *i.e.*, accuracy, precision, recall, F-measure, and AUC-ROC [43]. We also verified the statistical significance and the effect size of the differences using the Mann-Whitney test [44] and the Cliff's delta [45], respectively.

Regarding $\mathbf{RQ}_2$, we choose the *Validation and Voting* (VV) ensemble classifier defined by Liu *et al.* [7], which predicts the bug-proneness of a class based on the majority of "votes" of the base classifiers. The choice is driven by the findings provided by by Wang *et al.* [9] and Zhang *et al.* [28], which demonstrated that the VV method is able to outperform other ensemble classifiers in the contexts of within-project [9] and cross-project [28] bug prediction.

5.2 Analysis of the Results

Figure 3 shows the results achieved on the 30 subject systems by (i) ASCI, (ii) VV, and (iii) MLP, the base classifier achieving the best prediction performances in Section 4. For sake of readability, we do not report in Figure 3 the results obtained by the other base classifiers which are shown in Figure 2. Indeed if the performance of ASCI are better than MLP, then they are also better than the other base classifiers. However, the comparison of ASCI with all the base classifiers is reported in our online appendix [33].

Does the proposed adaptive technique outperform stand-alone classifiers? The proposed adaptive method shows an F-Measure ranging between 18% and 100%, while the accuracy of the classification is between 44% and 100%. Moreover the average AUC-ROC is 73%. These results indicate that the proposed solution has a good prediction accuracy, and it is not negatively influenced by the intrinsic degree of uncertainty created by the usage of a *classifier* prediction model. Moreover, when compared to the MLP model, we observed that ASCI performs better in 77% of the cases (23 out of the total 30 systems), with an F-Measure, an accuracy, and

an AUC-ROC 7%, 2%, and 1% higher than the baseline model, respectively. Table 4 shows that the superiority of our method is statistically significant ($\alpha < 0.01$) even though with a small effect size ($d = 0.17$). As MLP is the model achieving the best performances with respect to all the other classifiers experimented in Section 4, we can claim that on our dataset ASCI also works better than all the other classifiers, *i.e.,* NB, LOG, RBF, and DTree. However, complete results aimed at comparing ASCI with the other classifiers are available in our online appendix [33].

It is interesting to discuss the case of the `Redaktor` project. In this case, NB is the best classifier and the resulting model has 45% of F-Measure, 77% of accuracy, and 92% of AUC-ROC, while the model built using MLP achieves slightly lower performances. The two classifiers have a quite high complementarity (54% of buggy classes are correctly classified by only one of the two models) that does not allow the single stand-alone models to correctly capture the bug-proneness of all the classes of the system. In this case, the use of ASCI helps the prediction model to increase its F-measure up to 66%, *i.e.,* the adaptive model is able to correctly identify almost 21% more buggy classes. Moreover, the benefits provided by the use of our adaptive method are visible even when considering the other accuracy indicators. Indeed, there is an increase in precision values by 29% (from 35% to 64%), in accuracy values by 12%, and in the AUC-ROC values by 4%. As an example, let us consider the predictions provided by the different models on the classes `article.EditArticleForm` and `contact.NewContactForm`. The former class contains 367 lines of code and has a cyclomatic complexity equals to 6. The latter is composed of 132 lines of code and has a lower complexity (*i.e.,* cyclomatic complexity=4). In the case of stand-alone models, the first class is correctly classified as buggy only by NB, while the second one only by MLP. Thus, the individual classifiers correctly predict only one instance. Our model, instead, is able to correctly predict the bug-proneness of such components since it is able to correctly identify the right classifiers to use in both the situations. Specifically, the decision tree built on this system (step 3 in Section 3) recommends, for classes having more than 268 LOCs and cyclomatic complexity higher than 5, the application of the NB classifier, while the MLP is suggested when a class has less than 144 LOCs and cyclomatic complexity lower than 5. This example is quite representative of the ability of the proposed method to suggest which classifier should be used based on the structural characteristics of the classes, and highlights the beneficial effect of ensemble classifiers in the context of within-project bug prediction.

Another interesting observation can be made by looking at the results for the `JEdit` system. This project contains only 11 buggy classes (*i.e.,* 2% of the total classes), and we observed that the stand-alone models behave as a constant classifier which always predicts instances as non-buggy. Therefore, none of the experimented classifiers is able to correctly classify the 11 buggy classes of the system, while, instead, all of them correctly predict the results for non-buggy classes. By construction, in this case the annotated training set used by our adaptive method is built only using the results of the best classifier on the whole training set, *i.e.,* MLP. As a consequence, our model has exactly the same performances as the MLP model (F-Measure=18%, Accuracy=62%, AUC-ROC=69%). This example raises the limitation of our method. Indeed, when none of the stand-alone models is able to correctly predict the bug-proneness of a class, the choice of the classifier to use made by the adaptive model is useless, since all of them will perform bad. However this limitation affects also other ensemble classifiers, such as the VV technique. Moreover, as part of our future agenda we plan to combine ASCI with a Bagging technique able to reduce the variance of the predictions.

Summary for RQ$_1$. The proposed adaptive method outperforms the performances achieved by stand-alone models over all the software systems in our dataset. On average, the performances against the best stand-alone classifier increases up to 2% in terms of accuracy, 7% in terms of F-measure.

Does the proposed adaptive technique outperform the Validation and Voting ensemble technique? To answer this research question we first compare the VV ensemble technique with MLP, *i.e.,* the best classifier resulting from **RQ**$_0$. As it is possible to see from Figure 3, the VV technique, on average, performs 1% worse in terms of F-Measure, 2% better in terms of accuracy, and almost 10% worse in terms of AUC-ROC with respect to the stand-alone classifier: these results, especially the ones achieved considering the AUC-ROC, demonstrate how the VV model has often a low ability in discriminating classes affected and not by a bug. Furthermore, the VV technique actually outperforms the MLP model only on 50% of the systems (15/30), while on the other cases the stand-alone model works slightly better than the ensemble technique. The reason behind this result can be found looking at the way the VV technique provides its predictions. Specifically, the technique predicts a class as buggy when the majority of classifiers predict the class as buggy, while it predicts a class as bug-free in the opposite case. As pointed out by Bowes *et al.* [14], the VV technique does not take into account the success of the individual classifiers in predicting bugs. Thus, when the number of classifiers that provide a correct prediction is low, the results of the VV method are quite unreliable. On our dataset, this often happens. For instance, on the `Camel` project, all the classifiers have low F-Measure (ranging between 3% and 32%), and the overlap between them is very high (90%, on average). Thus, in this case the VV technique often answered by taking into account the wrong predictions made by the majority of classifiers.

The aforementioned reasons still explain the differences in the performances achieved by the VV method

and by ASCI. Indeed, we observed that, on average, the VV technique provides an F-Measure 5% lower than our approach. Moreover, ASCI achieves higher values with respect to all the other indicators (recall = +2%, precision = +12%, AUC-ROC = +8%), confirming its superiority on 25 of the 30 total systems considered in this study (*i.e.*, 83% of the cases). Table 4 shows that the differences are statistically significant ($p - value < 0.01$), even though with a small effect size ($d = 0.12$). Interesting are the cases of `Systemdata` and `Apache Velocity`, where our approach performs 31% and 13%, respectively, better than the VV method in terms of F-Measure.

To better understand the difference between our approach and the VV technique, consider the class `resource.ResourceManager` of the `Apache Velocity` project mentioned in Section 4: it is affected by a bug which is correctly classified by only two classifiers that we experimented in our study, *i.e.*, MLP and J48, while the three remaining classifiers predicted this instance as non-buggy. In this case, VV clearly provides an incorrect prediction, since the majority of the classifiers wrongly predict the bugginess of the class. Conversely, our approach is able to identify the class as buggy because it selects MLP as the classifier to use in this situation: in particular, the `resource.ResourceManager` class has several incoming dependencies, thus having high values for the Afferent Coupling (CA) and Coupling Between Methods (CBO) metrics [37]. For classes having CA > 8 and CBO > 13, our model suggested the usage of MLP as classifier, thus being able to correctly mark the class as buggy.

Other examples are represented by the previously cited `article.EditArticleForm` and `contact.NewContactForm` classes of the `Redaktor` project. While our model is able to correctly predict their bug-proneness (as explained above), the VV model cannot perform well because there is only one classifier out of the total five that correctly classify these instances (NB in the first case, MLP in the second one).

When VV works better than ASCI, the improvement in terms of F-Measure is between 3% and 14%. It is worth noting that the latter case refers to `SklepAGD`, a system composed of only 20 classes (12 of them containing bugs). As already explained before, in situations like this one it is possible that our model behaves in the same manner as the best stand-alone model. Thus, we can conclude that higher bug prediction performances can be obtained by taking into account the structural characteristics of the classes of a software project, rather than combining the output of different classifiers.

Summary for RQ$_2$. Our technique achieves performances higher than the Validation and Voting ensemble technique on 83% of the cases. This confirms that selecting the classifiers based on the characteristics of the class might be more effective than combining the results of different classifiers.

6 Threats to Validity

This section discusses the threats to the validity of our empirical evaluation, classifying them into *construct*, *external*, and *conclusion* validity.

Construct Validity. Threats to *construct validity* regard the relation between theory and observation. In our context, a threat in this category regards the dataset used in the empirical study. All the datasets come from the PROMISE repository [15], which is widely recognized as reliable, and it has been also used in several previous work in the field of bug prediction [4], [5], [7], [9], [14], [25], [28]. However, we cannot exclude possible imprecision or incompleteness. Another threat is related to the re-implementation of the Validation and Voting baseline approach which we used in the context of **RQ$_2$**. However, our re-implementation uses the exact algorithm defined by Liu *et al.* [7]. We know that parameter tuning is an important factor for defect prediction models. In order to minimize this threat we used the default parameters for each classifier used in our study.

External Validity. Threats to *external validity* concern the generalizability of our findings. To reduce this threat, we analyzed 30 different software projects that arise from very different application domains and present different characteristics (*i.e.*, developers, size, number of components, *etc.*). However, replication of the study on a larger set of systems would be desirable. Another threat is related to the variations in the prediction obtained by classifiers that we did not investigate. For this reason, we chose five classifiers that are quite representative of those used in previous studies on bug prediction (e.g., [5], [14]). Besides this, also the choice of the decision tree builder algorithm (*e.g.*, Random Forest) could have impacted our results. As future work we plan to replicate our study using more classifiers. Furthermore, we compared ASCI only with the VV ensemble method [7]: this choice was driven by recent findings showing the superiority of this approach with respect to other ensemble techniques [9], [28], future work will be devoted to compare ASCI with other ensemble techniques to improve the generalizability of the results. Finally, we tested the proposed approach in a within-project scenario, while its evaluation in a cross-project environment is part of our future agenda.

Conclusion Validity. Threats to *conclusion validity* are related to the relation between treatment and outcome. The metrics used in order to evaluate the approaches (*i.e.*, AUC-ROC, accuracy, precision, recall, and F-measure) have been widely used in the evaluation of the performances of bug prediction classifiers [4], [5], [7], [14]. However, we complement the quantitative analysis by reporting several qualitative examples aimed at showing the potential benefits provided by the use of the proposed adaptive model. Furthermore, we support our findings by using appropriate statistical tests, *i.e.*, the Mann-Whitney U and Cliff's Delta tests.

7 Conclusion and Future Work

In this paper we proposed ASCI, an approach able to dynamically recommend the classifier to use to predict the bug-proneness of a class based on its structural characteristics. To build our approach, we firstly performed an empirical study aimed at verifying whether five different classifiers correctly classify different sets of buggy components: as a result, we found that even different classifiers achieve similar performances, they often correctly predict the bug-proneness of different sets of classes. Once assessed the complementarity among the classifiers, we experimented ASCI on 30 open-source software systems, comparing its performances with the ones obtained by (i) the bug prediction models based on each of the five classifiers independently, and (ii) the Validation and Voting ensemble technique. Key results of our experiment indicate that:

- Our model achieves higher performances than the ones achieved by the best stand-alone model over all the software systems in our dataset. On average, the performances increases up to 7% in terms of F-Measure.
- Ensemble techniques such as Validation and Voting may fail in case of a high variability among the predictions provided by different classifiers. Indeed, in these cases the majority of them might wrongly classify the bug-proneness of a class, negatively influencing the performances of techniques which combine the output of different classifiers.
- An ensemble technique that analyzes the structural characteristics of classes to decide which classifier should be used might be more effective than ensemble techniques that combine the output of different classifiers. Indeed, our model exhibits performances which are on average 5% better than the Validation and Voting technique in terms of F-measure.

Our future research agenda includes the comparison of ASCI with other ensemble techniques. Furthermore, we plan to extend our study in order to analyze how the proposed model works in the context of cross-project bug prediction. In this context, the combination of ASCI with a technique able to reduce the variance of the training set (*e.g.,* Bagging) could improve the prediction performances. For example, it would be interesting to evaluate whether the local bug prediction proposed by Menzies *et al.* [12] could complement our model. Indeed, local models exploit similar information with respect to those used in our approach (structural characteristics of classes) to partially solve the problem of data heterogeneity. Hence, an adaptive method may be complementary to local bug prediction by selecting the most suitable classifier for each data cluster in the context of cross-project bug prediction.

References

[1] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

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

[3] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[4] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 789–800.

[5] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. IEEE, 2014, pp. 164–173.

[6] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1, pp. 1–39, 2010.

[7] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 852–864, Nov 2010.

[8] A. T. Mısırlı, A. B. Bener, and B. Turhan, "An industrial case study of classifier ensembles for locating software defects," *Software Quality Journal*, vol. 19, no. 3, pp. 515–536, 2011.

[9] T. Wang, W. Li, H. Shi, and Z. Liu, "Software defect prediction based on classifiers ensemble," *Journal of Information & Computational Science*, vol. 8, no. 16, pp. 4241–4254, 2011.

[10] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of International Conference on Software Engineering*. IEEE, 2011, pp. 481–490.

[11] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[12] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 822–834, 2013.

[13] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo, "Building an ensemble for software defect prediction based on diversity selection," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 46.

[14] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" *Software Quality Journal*, pp. 1–28, 2017.

[15] "The promise repository of empirical software engineering data," 2015.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*. IEEE, 2008, pp. 181–190.

[18] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, 2017.

[19] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1810–1829, 2016.

[20] M. E. Bezerra, A. L. Oliveira, P. J. Adeodato, and S. R. Meira, *Enhancing RBF-DDA algorithm's robustness: Neural networks applied to prediction of fault-prone software modules*. Springer, 2008, pp. 119–128.

[21] M. E. Bezerra, A. L. Oliveira, and S. R. Meira, "A constructive rbf neural network for estimating the probability of defects in software modules," in *2007 International Joint Conference on Neural Networks*. IEEE, 2007, pp. 2869–2874.

[22] M. O. Elish, "A comparative study of fault density prediction in aspect-oriented systems using mlp, rbf, knn, rt, denfis and svr models," *Artificial Intelligence Review*, vol. 42, no. 4, pp. 695–703, 2014.

[23] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 675–686, Oct 2007.

[24] R. Malhotra, "An empirical framework for defect prediction using machine learning techniques with android software," *Applied Soft Computing*, vol. 49, pp. 1034–1050, 2016.

[25] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, July 2008.

[26] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.

[27] A. Panichella, C. V. Alexandru, S. Panichella, A. Bacchelli, and H. C. Gall, "A search-based training algorithm for cost-aware defect prediction," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, 2016, pp. 1077–1084.

[28] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Proceedings of the IEEE Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 264–269.

[29] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 252–261.

[30] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th ACM SIGSOFT ESEC/FSE*. ACM, 2009, pp. 91–100.

[31] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.

[32] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, 2017.

[33] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method - replication package," 2016. [Online]. Available: http://figshare.com/articles/Dynamic_Selection_of_Classifiers_in_Bug_Prediction_an_Adaptive_Method/4206294

[34] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Int J Data Warehousing and Mining*, vol. 2007, pp. 1–13, 2007.

[35] L. M. Y. Freund, "The alternating decision tree learning algorithm," in *Proceeding of the International Conference on Machine Learning*, 1999, pp. 124–133.

[36] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[37] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 629–639, 1998.

[38] P. A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*, 1982.

[39] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

[40] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. ACM, 2008, p. 23.

[41] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, *Bugs as deviant behavior: A general approach to inferring errors in systems code*. ACM, 2001, vol. 35, no. 5.

[42] E. J. W. J. Sunghun Kim and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[43] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[44] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[45] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[46] C. Perlich, F. Provost, and J. S. Simonoff, "Tree induction vs. logistic regression: A learning-curve analysis," *J. Mach. Learn. Res.*, vol. 4, pp. 211–255, Dec. 2003.

Dario Di Nucci Dario Di Nucci received the masters degree in Computer Science from the University of Salerno, Italy, in 2014. He is currently working toward the PhD degree at the University of Salerno, Italy, under the supervision of Prof. Andrea De Lucia. His research interests include software maintenance and evolution, software testing, search based software engineering, green mining, mining software repositories, and empirical software engineering. He is a student member of the IEEE and ACM.

Fabio Palomba Fabio Palomba is a Postdoctoral Researcher at the Delft University of Technology, The Netherlands. He received the PhD degree in computer science from the University of Salerno, Italy, in 2017. His research interests include software maintenance and evolution, empirical software engineering, change and defect prediction, green mining and mining software repositories. He serves and has served as a program committee member of international conferences such as MSR, ICPC, ICSME, and others. He is member of the IEEE and ACM.

Rocco Oliveto Rocco Oliveto is Associate Professor at University of Molise (Italy), where he is also the Chair of the Computer Science program and the Director of the Software and Knowledge Engineering (STAKE) Lab. He is also one of the co-founders and CEO of datasounds, a spin-off of the University of Molise aiming at efficently exploiting the priceless heritage that can be extracted from big data analysis. He co-authored over 100 papers on topics related to software traceability, software maintenance and evolution, search-based software engineering, and empirical software engineering. His activities span various international software engineering research communities. He has served as organizing and program committee member of several international conferences in the field of software engineering. He was program co-chair for ICPC 2015, TEFSE 2015 and 2009, SCAM 2014, WCRE 2013 and 2012. He will be general chair for SANER 2018.

Andrea De Lucia Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the MSc degree in computer science from the University of Durham, United Kingdom, in 1996, and the PhD degree in electronic engineering and computer science from the University of Naples Federico II, Italy, in 1996. He is a full professor of software engineering at the Department of Computer Science, University of Salerno, Italy, the head of the Software Engineering Lab, and the director of the International Summer School on Software Engineering. His research interests include software maintenance and testing, reverse engineering and reengineering, source code analysis, code smell detection and refactoring, defect prediction, empirical software engineering, search-based software engineering, collaborative development, workflow and document management, visual languages, and elearning. He has published more than 200 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. He also serves on the editorial boards of international journals and on the organizing and program committees of several international conferences. He is a senior member of the IEEE and the IEEE Computer Society. He was also at-large member of the executive committee of the IEEE Technical Council on Software Engineering (TCSE).