Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment

Gemma Catolino[1], Dario Di Nucci[2], Filomena Ferrucci[1]
[1]University of Salerno, Italy, [2] Vrije Universiteit Brussel, Belgium
gcatolino@unisa.it, dario.di.nucci@vub.be, fferrucci@unisa.it

***Abstract*—Bug Prediction is an activity aimed at identifying defect-prone source code entities that allows developers to focus testing efforts on specific areas of software systems. Recently, the research community proposed *Just-in-Time* (JIT) Bug Prediction with the goal of detecting bugs at commit-level. While this topic has been extensively investigated in the context of traditional systems, to the best of our knowledge, only a few preliminary studies assessed the performance of the technique in a mobile environment, by applying the metrics proposed by Kamei *et al.* in a within-project scenario. The results of these studies highlighted that there is still room for improvement. In this paper, we faced this problem to understand (i) which Kamei *et al.*'s metrics are useful in the mobile context, (ii) if different classifiers impact the performance of cross-project JIT bug prediction models and (iii) whether the application of ensemble techniques improves the capabilities of the models. To carry out the experiment, we first applied a feature selection technique, *i.e.,* InfoGain, to filter relevant features and avoid models multicollinearity. Then, we assessed and compared the performance of four different well-known classifiers and four ensemble techniques. Our empirical study involved 14 apps and $42,543$ commits extracted from the COMMIT GURU platform. The results show that *Naive Bayes* achieves the best performance with respect to the other classifiers and in some cases outperforms some well-known ensemble techniques.**

***Index Terms*—JIT Bug Prediction; Metrics; Empirical Study**

I. INTRODUCTION

Continuous Integration (CI) [1] is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to code base; it aims to provide integration errors or bugs earlier, thus improving software quality and speeding up the issue resolution activities carried out by developers. Although CI is originated from the Extreme Programming paradigm [2], [3], born in the early nineties, its principles can be applied to any iterative modern development model.

Given its nature, CI naturally fits well in the context of emerging release planning strategies where software systems are not released following a clearly defined road-map, but rather *continuous* releases become available on weekly/daily basis [4]. Nowadays, such emerging development release practices is widely adopted by mobile applications: indeed, with over two billion users relying on smart-phones and tablets for social and emergency connectivity [5], mobile app developers need to continuously update their apps in order to satisfy users proposing new features or fixing important bugs [6].

In such a context, mobile developers require appropriate tools to manage continuous changes and avoid the introduction of bugs [7], [8]. While the research community is devising many tools to simplify maintenance and evolution of mobile apps [9]–[11], there is still a lack of tools that help developers in timely spotting possible bugs. Toward this direction, a promising area is represented by software bug prediction. This technique relies on a set of metrics (*i.e.,* independent variables) to characterise a certain piece of software in order to predict its bug-proneness (*i.e.,* dependent variable) by using a machine learning technique (a.k.a., classifier) [12]. As recently shown by Kamei *et al.* [13], such models can be even more useful if they provide feedback as soon as new changes are committed to repositories. Being able to notably reduce the effort needed for code inspection, this fine-grained technique, coined as *Just-in-Time* (JIT) [13], [14], has been shown to be effective in the context of traditional applications.

Despite the previous work to define JIT bug prediction models, there is still *lack of empirical evidence on the performance of such models in the context of mobile apps*. Indeed, the different way in which mobile apps are developed (*e.g.,* continuous releases, newcomers issuing their own apps, etc.) might have a strong influence on the performance of models that have been not designed for taking into account mobile-related factors.

To the best of our knowledge, this topic has been treated only by Kaur *et al.* [15] and Catolino [16]. The former work explored the usefulness of code and process metrics to predict mobile app bugs occurring in the next release, while the latter preliminarily assessed a within-project JIT bug prediction model (*i.e.,* a model relying on previous data of the considered apps) on five mobile apps, finding that the performance are very low (*i.e.,* $\approx$35% in terms of F-Measure). As a result, these authors highlighted how further studies are needed to apply these models in the mobile context. In this paper, we start bridging this gap by analysing three aspects:

- **The selection of metrics.** Most of the metrics used for traditional systems have been adapted into the mobile context [17], [18]. However, it is still unclear to what extent these metrics are effective [15], [16].
- **The impact of classifiers.** Recent findings [19], [20] showed that the choice of the machine learning technique to build prediction models has a strong impact on the performance (*i.e.,* by up to 30%), as well as the use of ensemble techniques usually improve the performance of prediction models [19], [21]–[23]. Nevertheless, in the context of mobile, it is still unclear which classifiers offer

the best performance.

- **The need for cross-project bug prediction models.** As suggested by Hall *et al.* [12], cross-project models should be further investigated to ensure a wider applicability of bug prediction: indeed, lack of previous data might render the application of within-project strategies not feasible. This is particularly true in the context of mobile, where new applications are born on daily basis and, therefore, developers do not have information to build within-project bug prediction models (as also shown in [16]).

To address these arguments and to assess the usefulness of JIT bug prediction for mobile applications, we conducted an empirical study with the aim of understanding (i) which Kamei *et al.*'s metrics are relevant in mobile context, (ii) if different classifiers impact the performance of cross-project JIT bug prediction models, and (iii) whether the application of ensemble techniques improves the capabilities of the models. The study involved 14 applications for a total of $42,543$ commits extracted from the COMMIT GURU platform [24].

To understand which metrics have more influence on the bugginess of mobile apps commits, we first applied a feature selection technique, *i.e., InfoGain* [25] on the set of metrics proposed by Kamei *et al.* [13].

Once extracted this set, we experimented four standard machine learning techniques (*i.e., Logistic Regression* [26], *Decision Table* [27], *Support Vector Machine* [28], and *Naive Bayes* [29]) and four ensemble methods (*i.e., Random Forests* [30], *Voting* [31], *Bootstrap Aggregating* [32], and *Boosting* [33]). We assessed each classifier and ensemble technique adopting the *Leave-One-Out Cross-Validation* [34].

Key results of our study indicate that metrics related to code churn, *e.g., Lines of code deleted*, diffusion of changes, *e.g., Number of modified directories*, and history, *e.g., Number of unique changes*, are important to identify risky commits. Despite the differences between traditional systems and mobile apps [8], these results are partially in line with traditional bug research in traditional system [12]. We obtained similar results when comparing the performance of JIT bug prediction models. Indeed, we observed that the best performance for standard classifiers is achieved by *Naive Bayes*, *i.e.,* 50% of F-Measure, while *Logistic Regression* performed worse than other classifiers by up to 40% in terms of *F-measure*. This means that the choice of classifier has impacted on the performance of JIT bug prediction models [19], [20]. Finally, applying ensemble methods does not provide evident benefits with respect to simpler classifiers. For example, the model built using the *Boosting* ensemble technique performed only 4-5% better than the one relying on *Naive Bayes*.

Structure of the paper. Section II overviews the literature related to bug prediction in the context of mobile apps. In Section III we discuss the research methodology adopted to build and evaluate JIT bug prediction models, while in Section IV we report the results of the empirical study. Section V examines the threats to the validity of the study and the way we mitigated them. Finally, Section VI concludes the paper and provides insights on our future research agenda.

II. RELATED WORK

Bug prediction represents a topic extensively investigated in the context of standard systems [12], [35]–[37], using both traditional and JIT techniques. As for mobile applications, at the best of our knowledge, there are only two studies addressing the topic [15], [16]. In the following, we discuss the literature related to *Just-in-Time* bug prediction and cross-project bug prediction. Furthermore, we analyze the main results related to the role of machine learners and ensemble techniques for bug prediction

A. Just-in-Time Bug Prediction for Traditional Systems

The research conducted by Kamei *et al.* [13], [14], [38], [39] plays a key role for JIT bug prediction of traditional systems. Indeed, they proposed a *Just-in-Time* quality assurance technique as a more practical alternative to traditional bug prediction techniques being able to provide defect feedback at commit-time. To this aim they devised a set of metrics, named Kamei *et al.*'s metrics in the following. In particular, in the first study [13] they analysed the performance of JIT bug prediction models considering 11 software systems; they built a logistic regression model and validate it using 10 fold-cross validation. In [14], [38] they performed additional analysis considering also cross-project models. The results show that models trained on other projects are a good solution for systems with limited historical data. However, JIT bug prediction models perform better in cross-project context when the training instances are carefully selected. Finally, McIntosh and Kamei [39] showed how fluctuations in the properties of fix-inducing changes can impact the performance and interpretation of JIT models.

Other researchers tried to propose alternatives for *Just-in-Time* quality assurance, using deep learning and textual analysis [40], [41], while others proposed new approaches [42]–[45] exploiting the Kamei *et al.*'s metrics [13]. In particulars, Yang *et al.* [44] showed that simple unsupervised JIT bug prediction models perform better than the state-of-the-art supervised models in JIT defect prediction, and that deep learning is suitable for JIT models [40]. Afterwards, they proposed a two-layer ensemble learning approach for *Just-in-Time* defect prediction [42]. In the inner layer, they combine Decision Tree and Bagging to build a Random Forest model, while in the outer layer, they use random under-sampling to train many different Random Forest models and ensemble them once more using stacking. The evaluation of the approach, conducted on six open source projects, showed good results in terms of prediction accuracy.

Huang *et al.* [45] replicated the study of Yang *et al.* [44] while proposing a simpler but improved supervised model which obtained similar results in terms of Recall, but performed significantly better in terms of Precision [44]. Chen *et al.* [43] proposed a multi-objective optimisation supervised method to build JIT bug prediction models. They conducted a large-scale empirical study to compare their approach with 43 state-of-the-art supervised and unsupervised methods under three commonly used performance evaluation scenarios: cross-validation, cross-project-validation, and time wise-cross-validation. The results

confirm that supervised methods are still promising in effort-aware JIT bug prediction models. Finally, Nayrolles and Hamou-Lhadj [46] proposed an approach, called CLEVER (Combining Levels of Bug Prevention and Resolution techniques) that intercepts risky commits (commits that could contain bugs) before they reach the central repository. They evaluated their approach on 12 Ubisoft systems, achieving good results.

B. Just-in-Time Bug Prediction for mobile apps

In the context of *Just-in-Time* bug prediction of mobile applications, Kaur *et al.* [15] evaluated the accuracy of code and process metrics to predict defects in open source mobile apps, showing that models based on process metrics are better than those based on code metrics. In general, in the field of mobile apps, the research community has tried to adapt metrics used in standard software systems [15]–[18], [47], more than proposing metrics or guidelines to measure mobile applications. Starting from the work by Kamei *et al.* [13], [14], [38], only one recent study [16] tried to adapt their set of metrics, in the context of mobile applications. This work considered five mobile apps from the COMMIT GURU platform [24] and applied a feature selection technique, *i.e., Wrapper* [48], in order to extract the most significant metrics to predict commit bugginess. Finally, a logistic regression model was built, adopting *Ten-fold cross validation* and a within-project strategy. Results showed that the models were able to discover only a limited number of bugs (Recall $\approx 25\%$), highlighting the need of further analysis. The work of Catolino [16] represents the baseline of the current paper. In particular, we performed a large scale empirical study (i) involving more apps, (ii) aiming to deeply analysing which Kamei *et al.*'s metrics can be adapted in mobile context using a different type of feature selection analysis, (iii) understanding if different classifiers impact the performance of JIT bug prediction models using a cross-project strategy, and (iv) investigating whether the application of ensemble techniques improves the capabilities of the models.

C. Cross-Project Bug Prediction

Several studies addressed the problem of cross-project bug prediction, *e.g.,* [49]–[54]. However, as highlighted by Hall *et al.* [12] within-project techniques need to be gradually replaced by cross-project solutions to ensure a wider applicability of bug prediction. In this subsection, we recall the main papers on the topic. In particular, Zimmermann *et al.* [49] conducted a large experiment on 12 systems, showing the pitfalls of cross-project defect prediction. Turhan *et al.* [50] compared the performance of cross- and within-project strategies, concluding that within-project ones performed better. Watanabe *et al.* [51] defined a method able to adapt external data to the project taken into account. Their results showed that it is possible to re-use prediction models among projects developed with different programming languages. Jureczko and Madeyski [53] showed that a selection of external data can be successfully exploited to predict bugs. To this aim they exploited a clustering technique to split different projects based on their characteristics. Nagappan *et al.* [52] studied the generalisability of cross-project bug prediction models for identifying post-release bugs. The empirical investigation of five Microsoft systems revealed that no single set of predictors properly fits all projects. Furthermore, they showed that within-project models tend to perform better than cross-project ones. Finally, He *et al.* [54] conducted a large experiment on 34 datasets obtained from ten open source projects. Their results showed that cross-project strategy worked better than within-project and those results are strictly related with the ability to select valuable data for training models.

D. Ensemble Techniques for Bug Prediction

Ghotra *et al.* [20] recently showed that the selection of the machine learner is relevant for the performance of the bug prediction model (it can increase/decrease up to 30%). Aiming to exploit the best contribution of different techniques, the research community moved their attention to the application of ensemble methods [33]. Wang *et al.* [21] compared the performances achieved with seven ensemble techniques in the context of within-project bug prediction, finding that Voting performed better with respect to the others. The same results were obtained by Misirli *et al.* [55], who improved the performance of within-project bug prediction using the Voting technique. Similar results were obtained by Zhang *et al.* [56] and Panichella *et al.* [19] that compared different ensemble approaches, highlighting how Voting performs better. This technique turned out to be the best ensemble technique also in the context of cross-project bug prediction, as shown by Liu et al *et al.* [57] who evaluated 17 different models.

Kim *et al.* [58] combined multiple training data and they applied a random sampling in the context of within-project bug prediction. Petric *et al.* [23] built a Stacking ensemble technique [33] starting from four families of classifiers in the cross-project bug prediction. Their results showed the improvements given by their approach with respect to other ensemble techniques. At the same time, Panichella *et al.* [19] devised CODEP, a technique that applies a set of classification models independently and then uses the output of the first step as predictors of a new prediction model.

Finally, Di Nucci *et al.* [22] developed 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. They performed an empirical study on 30 systems, showing that their approach reaches higher performances than five different classifiers used independently and combined with the majority voting ensemble method.

III. RESEARCH METHODOLOGY

This section describes the design of the empirical study.

A. Research Questions

The *goal* of the study is to identify risky code changes, using Kamei *et al.*'s metrics [13], that should be reviewed and/or tested more carefully before their integration in the

code base in the *context* of mobile applications, with the *purpose* of reducing the effort required for testing activities, focusing on those components that are more bug-prone. The *quality focus* is on the prediction accuracy of *Just-in-Time* bug prediction models trained using different standard classifiers and ensemble methods in cross-project strategy. The *perspective* is of researchers, who want to evaluate the effectiveness of using the model when predicting bug-prone components in mobile apps. In details, we formulated the following research questions:

RQ_1: *Which of the metrics proposed by Kamei et al. are more relevant in the context of JIT bug prediction for mobile apps?*

With this research question, we analysed the relevance of the metrics proposed by Kamei *et al.* in the context of mobile apps. The goal is to understand whether the metrics used in *Just-in-Time* bug prediction for traditional software are suitable also in the mobile context. We used the feature selection algorithm described in one of the next subsection to understand which metrics are more suitable as predictors in JIT bug prediction.

RQ_2: *How different classifiers work in the context of cross-project JIT bug prediction for mobile apps?*

The aim of this research question is to assess the impact of using different classifiers in the performance of *Just-in-Time* bug prediction models. We choose several classifiers that have been used in many previous works for bug prediction [19], [56], and are based on different learning peculiarities (*e.g.,* regression functions, neural networks, and decision trees). This choice increases the generalisability of our results. Finally, the motivation behind the application of a cross-project strategy depends on the fact that in the context of mobile apps, new applications are born on daily basis and, therefore, developers cannot have enough information to build a prediction model, making the within-project strategy not always feasible, as also shown by Catolino [16].

RQ_3: *Do ensemble techniques improve the effectiveness of cross-project prediction models?*

Finally, we compare the performance of the models built using ensemble techniques with those built using single classifiers. Indeed, as shown by the research community, different classifiers are suitable in different contexts and their combination can lead to better performance [22], [59]. For this reason, ensemble techniques have been used for bug prediction [22], change prediction [60], and effort estimation [61]. As done by Kamei *et al.* [14], we compare the performance of the best model coming from $\mathbf{RQ}_2$ with the models trained using ensemble techniques.

B. Context Selection

The *context* of the study consists of 14 open source mobile Android applications with different size and scope. To reduce the threats related to the generalisability of the results, we selected mobile apps having different application domains and size from the COMMIT GURU platform [24]. Table I shows for each app (i) the URL of the *Play Store* page, (ii) the URL of the *GitHub* repository, (iii) the number of commits, and (iv) the percentage of buggy commits. The dataset used in this study is available in the online appendix [62].

C. Building Just-in-Time Bug Prediction Models

To answer our research questions, we first need to design *Just-in-Time* bug prediction models. This step requires the definition of several aspects such as (i) the selection of the independent variables to use in the models, (ii) the formulation of the dependent variable that the model will predict, (iii) the choice of a feature selection technique to avoid multi-collinearity, (iv) the definition of training and validation strategies. The following sub-sections report the choices that we adopt for each aspect.

Independent Variables. As previously mentioned, we use as independent variables the metrics proposed by Kamei *et al.* [13] for *Just-in-Time* bug prediction. Indeed, these metrics have been shown to perform well in the context of tradition software systems to predict whether or not a change will induce a bug in the future [13], [14], [38]. They take into account a wide range of factors based on the characteristics of a software change (*i.e.,* a commit), such as the number of added lines, and committers' and developers' experience. Table II reports the metrics grouped according to their scope. The *Diffusion* scope refers to the diffusion of changes. In general, metrics within this scope are considered very relevant in bug prediction [63] along with those belonging to the *Size* scope [64]. For the latter, the intuition is that larger changes are more prone to introduce bugs. Another interesting scope is the *History* of the files involved in the changes. Indeed Matsumoto *et al.* [65] showed how files previously modified by many developers, (*i.e., NDEV* metrics) usually contain more bugs. Furthermore, a recent study showed how developers-factors can influence the change proneness of the code components [66]. Regarding the *Purpose* scope, previous work [67] highlighted that bug-fixing changes are more likely to induce new bugs. Finally, regarding the *Experience* scope, previous work [63] showed that high-experienced developers tend to introduce less bugs.

Dependent Variable. We considered the bugginess of a commit as dependent variable and we analyzed $42,543$ commits. In particular, we considered a commit as *buggy* if it introduced a defect that was fixed by a later commit, otherwise we consider it *clean*. It is important to note that the dependent and independent variables used in this study are available on the COMMIT GURU platform [24].

Feature Selection. Although the independent metrics have already been experimented in *Just-in-Time* bug prediction models [13], we believe that a simple combination, obtained by featuring together all the predictors, might lead to sub-optimal results because of over-fitting [68]. To avoid this issue, we study the subsets of predictors able to lead to the best prediction performance. To this aim, we perform feature selection [69] by applying *information gain* [25]. This algorithm quantifies for each metric the gain obtained by adding it to the prediction model. Our choice has been driven by the ability of the algorithm to quantify the gain provided by each metric,

Table I
CHARACTERISTICS OF THE MOBILE APPS CONSIDERED IN THE STUDY

Project	URL of Play Store	URL of Repository	#SLOC	#Developers	# of Commits	% of Buggy Commits
Afwall	https://tinyurl.com/opd8628	https://tinyurl.com/m722ouo	77,243	20	1,127	37%
Alfresco	https://tinyurl.com/kfv93ez	https://tinyurl.com/ya533yya	152,047	5	1,449	21%
Android Sync	https://tinyurl.com/yafbk6f2	https://tinyurl.com/y9vcudjt	275,637	27	280	51%
Android Walpaper	https://tinyurl.com/hpl65mr	https://tinyurl.com/y7f5bjpt	35,917	1	605	22%
AnySoftKeyboard	https://tinyurl.com/k9s97zl	https://tinyurl.com/lqzxc3v	114,784	38	3,250	26%
Apg	https://tinyurl.com/cxwqp5n	https://tinyurl.com/y6vxu57x	151,204	56	4.363	30%
Atmosphere	https://tinyurl.com/ybdofq5h	https://tinyurl.com/y9ovae5z	56,686	101	5,757	38%
Chat Secure Android	https://tinyurl.com/lero26e	https://tinyurl.com/pxcupkk	98,768	35	2,869	30%
Facebook Android SDK	https://tinyurl.com/6ueeu7y	https://tinyurl.com/yctphsxw	103,802	64	636	28%
Flutter	https://tinyurl.com/y9q57wa8	https://tinyurl.com/yd25noy3	639,350	352	13,067	1%
Kiwix	https://tinyurl.com/ognzugp	https://tinyurl.com/ycvufxwn	32,598	60	1,571	22%
Own Cloud Android	https://tinyurl.com/8vfuemy	https://tinyurl.com/ptflhfe	115,169	70	7,144	22%
Page Turner	https://tinyurl.com/y9qgcffz	https://tinyurl.com/yc26yklh	30,943	16	193	22%
Notify Reddit	https://tinyurl.com/nd835ec	https://tinyurl.com/ydce46ge	9,506	2	231	26%

Table II
THE METRICS DEFINED BY KAMEI *et al.* [13]

Scope	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Number of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a bug fix
History	NDEV	Number of developers working on the files
	AGE	Average number of days since the last change
	NUC	Number of unique change to modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

thus allowing us to study which ones have more influence in *Just-in-Time* bug prediction. More formally, let M be the model using all the predictors and $F = \{f_1, \ldots, f_n\}$ the set of features composing M, *information gain* [25] measures the difference in terms of entropy between the set that includes f_i and the one that does not, using the following formula:

$$InfoGain(M, f_i) = E(M) - E(M|f_i) \quad (1)$$

where the function $E(M)$ represents the entropy of M when it includes f_i, and the function $E(M|f_i)$ represents the entropy of M that does not include f_i. $E(M)$ is computed as reported in the following equation:

$$E(M) = -\sum_{i=1}^{n} prob(f_i) \log_2 prob(f_i) \quad (2)$$

The output of the algorithm is represented by a ranked list where the more relevant features, *i.e.,* the ones having the highest expected reduction in entropy, are placed on the top. To ensure that only the relevant features are used, we set the cut-off point of the ranked list equal to 0.1, as suggested by previous studies [25], [60].

In a previous work, Catolino [16] used *Wrapper* [48] as feature selection algorithm. This method measures the "usefulness" of features based on the classifier performance, while *information gain* is a filtering method that picks up the intrinsic properties of the features (*i.e.,* the "relevance" of the features) measured via univariate statistics instead of cross-validation performance. Since our goal was to deeper analyze the reasons behind the extraction of particular metrics, we decided to use this kind of method.

It is important to note that the result of this step has been used for the evaluation analysis of RQ_1 and the selection of the independent variables used to build JIT bug prediction models in RQ_2 and RQ_3.

Data Balancing. *Just-in-Time* bug prediction is an imbalanced problem [70]. This means that the number of data available in the training set for a certain class (*e.g.,* the number of buggy commits) is far less than the amount of data available for another class (*e.g.,*, the number of clean commits). The skewness of the dataset is highlighted in Table I. To handle this issue, we apply *Synthetic Minority Over-sampling Technique* (SMOTE), proposed by Chawla *et al.* [71] to make the training set uniform with respect to the number of buggy commits. Since this approach can be run once per time to over-sample a certain minority class, we repeated the over-sampling until all the classes considered have a similar number of instances.

Training Strategy. We experimented four standard machine learning techniques (*i.e., Logistic Regression* [26], *Decision Table* [27], *Support Vector Machine* [28], and *Naive Bayes* [29]) and four ensemble methods (*i.e., Random Forests* [30], *Voting* [31], *Bootstrap Aggregating* [32], and *Boosting* [33]).

Decision Table [27] describes all possible combinations of conditions and the decision appropriate to each combination. Each decision corresponds to a variable, relation or predicate whose possible values are listed among the condition alternatives. Each action is a procedure or operation to perform, and the entries specify whether (or in what order) the action is to be performed for the set of condition alternatives the entry corresponds to.

Logistic Regression [26] is used to explain the relationship between one dependent binary variable and one or more metrics (interval or ratio scale) independent variables. Logistic regression can be binomial (binary), ordinal or multinomial. In this work, we use *Binary Logistic Regression*, indeed it deals

with situations in which the observed outcome for a dependent variable can have only two possible values.

Support Vector Machine (SVM) [28] is primarily a classier method that performs classification tasks by constructing hyperplanes in a multidimensional space that separates cases of different class labels. SVM can be applied for regression and classification tasks and can handle multiple continuous and categorical variables.

Naive Bayes [29] methods are a set of supervised learning algorithms based on the application of Bayes' theorem with the naive assumption that the value of a particular feature is independent from the value of any other feature, given the class variable.

Random Forests [30] are a mixture of tree predictors such that each tree depends on the values of a random vector sampled autonomously and with the same distribution for all trees in the forest. The generalization error for forests converges to a limit as the number of trees in the forest becomes large, while the generalization error of each tree depends on the strength of the individual trees in the forest and the association between them. A different subset of the training data is selected, with replacement, to train each tree. Remaining training data are used to estimate error and variable importance. Class assignment is made by the number of votes from all of the trees and for regression, the average of the results is used.

Voting [31] represents the simplest ensemble algorithm, and is often very effective. It can be used for classification or regression problems. Voting works by creating two or more sub-models. Each sub-model makes predictions which are combined in some way, such as by taking the mean or the mode of the predictions, allowing each sub-model to vote on what the outcome should be. In our case, the weak learners are the four standard classifiers described above.

Bootstrap Aggregating [32], also called *Bagging*, combines the output of various models in a single prediction. During the training phase, *m* datasets with the same size as the original one are generated by performing sampling with replacement (bootstrap) from the training set. Hence for each dataset, a model is trained using a weak classifier. During the test phase, for each instance, the composite classifier uses a majority voting rule to combine the output of the models into a single prediction. The weak learner chosen is *REPTree*, since a previous study reported good result using this technique [72].

Adaptive Boosting (*AdaBoost*) [33] is a well-known *Boosting* technique. During the training phase, *AdaBoost* repetitively trains a weak classifier on subsequent training data. At each iteration, a weight is assigned to each instance of the training set, with the purpose of assigning higher weights to misclassified instances that should have more chances to be correctly predicted by the new models. At the end of the training phase, a weight is assigned to each model in order to reward models having higher overall accuracy. During the test phase, the prediction of a new instance is performed by voting all models. The results are thus combined using the weights of the models, in the case of binary classification a threshold of 0.5 is applied. The weak learner chosen is *DecisionStump* that has been widely used in combination with *Boosting* and *Bagging* [72], [73].

It is important to note that all the techniques described above have been already used for bug prediction [22], [35], [74]–[76]. Finally, before running the models, we also identified their best configuration using the MULTI SEARCH, a generalized version of the *Grid Search* algorithm [77]. Such an algorithm represents a brute force method to estimate hyper-parameters of a machine learning approach. Suppose that a certain classifier C has k parameters, and each of them has N possible values. A grid search basically considers a Cartesian product $f_{|_{k\times N}}$ of these possible values and tests all of them. We selected this algorithm because recent work in the area of machine learning has shown that it is among the most effective methods to configure machine learning algorithms [77].

Validation Strategy. To evaluate the models, we adopted the *Leave-One-Out Cross-Validation* [34] as done by Kamei *et al.* in their study [14]. *Leave-one-out cross validation* is K-fold cross validation taken to its logical extreme, with K equal to N, the number projects in the set. That means that N separate times, the function approximator is trained on all the data except for one project and a prediction is made for that point.The validation was repeated 10 times, so that each project formed the test set once.

D. Evaluation of RQ_1

To perform a comprehensive empirical analysis of the applicability of defect prediction models for mobile apps, we started with the evaluation of which and why particular metrics can be adopted in the mobile context. For this reason, starting from Kamei *et al.*'s metrics, we deeper analysed the result of *Information Gain* [25] described above, to understand the factors that lead some metrics to be more useful than others and trying to provide a possible interpretation.

E. Evaluation of Performance RQ_2 and RQ_3

To evaluate the performance of the models and answer to the latter research questions, we first computed the harmonic mean of precision and recall (*i.e., F-Measure*):

$$F\text{-}Measure = 2 * \frac{precision * recall}{precision + recall} \quad (3)$$

In addition, we computed the *Matthews Correlation Coefficient* (MCC) [78], defined in the following

$$MCC = \frac{(TP*TN)-(FP*FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (4)$$

where *TP*, *TN*, and *FP* represent the number of (i) true positives, (ii) true negatives, and (iii) false positives, respectively, while *FN* is the number of false negatives. Its value ranges between -1 and +1. A coefficient equal to +1 indicates a perfect prediction; 0 suggests that the model is no better than a random

one; and -1 indicates total disagreement between prediction and observation.

Finally, we report the *Area Under the ROC Curve* $AUC - ROC$ obtained by the experimented prediction models. This metric quantifies the overall ability of a prediction model to discriminate between *buggy* and *clean* commits. The closer $AUC - ROC$ to 1, the higher the discrimination power of the classifier, while an $AUC - ROC$ closer to 0.5 means that the model is not very different from a random one.

We statistically verified the validity of our findings by using the Scott-Knott ESD test [79]. This extension of original Scott-Knott test [80] applies a hierarchical clustering algorithm to group together the model performances based on the statistically significance of the differences between them. Afterwards, it refines the clusters by merging together the groups whose differences are negligible in terms of effect size. We executed the test to verify to what extent the differences in terms of *Matthew Correlation Coefficient* were statistically significant.

IV. Results

This section reports the analysis of the results achieved for the three research questions formulated in our study.

Table III
Gain Provided by each Feature when Predicting Buggy Commits.

Metric	Info Gain
nuc	**0.25**
ld	**0.25**
la	**0.22**
nf	**0.19**
nd	**0.17**
ndev	**0.15**
ns	0.08
entropy	0.06
sexp	0.06
rexp	0.06
exp	0.05
lt	0.05
age	0.01
fix	< 0.01

A. RQ_1: Which of the metrics proposed by Kamei et al. are more relevant in the context of JIT bug prediction for mobile apps?

The aim of the first research question was to deeper analysing which Kamei *et al.*'s metrics, used for traditional systems, are relevant in the context of mobile apps. Previous studies analysed how mobile apps and traditional systems result very different, in terms of size, component etc., [8] however, the research community showed how some metrics belonging to traditional systems can be also adopted in the mobile context [17], [18]. The first step of this analysis consisted of pruning the non-relevant metrics through the application of the *information gain* algorithm [25], whose results are reported in Table III.

As it is possible to see, the *Number of unique change to modified files* (nuc) provides a significant contribution to the overall reduction of the entropy of the model: in particular, the gain provided is quantifiable at 0.25. As explained in the paper by Kamei *et al.* [13], "the larger the nuc, the more likely a defect is introduced, because developers will have to recall and track many changes". A possible explanation behind this relevance is that mobile apps are stressed with many changes given their presence in the app market [11], [81]; indeed, developers have to keep up with user feedback and reviews (all different between each other) to compete on the app market, this lead to change several times the application. Such ready-to-commit changes are more bug-prone.

Also the *Line added* (la) and *Line deleted* (ld) metrics provide a strong contribution to the model (*i.e.,* 0.25 for the former, 0.22 for the latter). Code churn is considered an important factor in the prediction of bug prone components [64], [82], this could be true in the case of a bug-prone commit. Indeed, given the motivation described above for the *nuc* metric, in addition to the rapid development process of mobile apps [83], developers could be more inclined to introduce a bug.

The results also show how *Number of modifies files* (nf) and *Number of modified directories* (nd) reduce the uncertainty of the model during the classification of bug-prone commits (*i.e.,* 0.19 for the former, 0.17 for the latter). A possible explanation behind this result could be related to the massive usage and updating of libraries by developers in mobile apps, as also confirmed in the study by Salza *et al.* [84]. This might possibly lead to miss the adaptation of some source code files to the newly available library, thus increasing the likelihood to introduce defects.

Finally, we found that *Number of developers working on the file* (ndev) contributes in reducing the entropy of the model (*i.e.,* 0.15). Being a mobile app smaller than a traditional application in term of size, the proportion between number of developers and app size could be obvious. However, we found some examples where the number of contributors (developers) is very high; *e.g.,* Kiwix (an offline reader for Web content) and Own Cloud (open source file sync and share software) have 52 and 67 contributors respectively. Thus, it is possible that more developers work on the same files, making the components more prone to errors.

Other metrics provided an information gain lower than the threshold of 0.10, thus their usefulness can be considered limited. It is important to point out that metrics related to the developers' experience (*exp, rexp, sexp*) give a limited contribution when predicting bug-prone commits. A possible explanation could be given by the rapid development of an app as well as the high number of developers within an app [83]. Moreover, since these metrics are based on the contribution of developers in terms of number of commits (and it is clearly visible how the number of commits in a mobile app is extremely low with respect to a traditional application), the computation of these factors could be not effective. Since research community highlighted how developers experience represents a critical factor [35], [60], [85], we believe that new metrics related to experience are needed.

Summary for RQ$_1$. Metrics related to code churn (*i.e., la* and *ld*), diffusion of changes (*i.e., nd* and *nf* and *nd*), and history (*i.e., nuc* and *ndev*) are important in risky commits identification.

B. RQ$_2$: How different classifiers work in the context of cross-project JIT bug prediction for mobile apps?

Figures 1 shows the results achieved by JIT bug prediction models built using standard classifiers. Each box-plot is based on the distribution of the 14 mobile apps considered in this study and $42,543$ commits. Detailed results for each mobile app are available on the online appendix [62]. Looking at the F-Measure 1, we notice that there is no a clear winner among the different classifiers, especially if we consider *Naive Bayes*, *SVM* and *Decision Table*. Indeed, we can notice that the difference in terms of F-Measure (49%, 44% and 45% respectively), and AUC-ROC (77%, 69% and 69 % respectively) achieved by three of the experimented models is quite small. Analysing the detailed results in [62], *Naive Bayes* obtained the best performance in 12 of 14 mobile apps, with a median value of MCC around 0.33, slightly better than *Support Vector Machine* and *Decision Table* (*i.e.,* +6% and +5% respectively). The better performance of Naive Bayes partially confirms the results reported by Hall *et al.* [12] in their systematic literature review, which highlights how *Naive Bayes* and Logistic regression resulted the best performing among several classifiers. However, in our study the latter achieved the worst performance. Analysing the detailed results in [62] we observe that the recall value is very low (the maximum value achieved is 15%). So, it seems that the model cannot correctly identify the bugginess of most of the actual buggy commits present in the considered mobile apps; it is not able to define a clear separation at the decision boundary, thus not being able to distinguish between commits actually affected by bugs and those not introducing any bug. In other words, the model cannot properly set its coefficients since it cannot adequately predict the gain/loss from each individual feature.

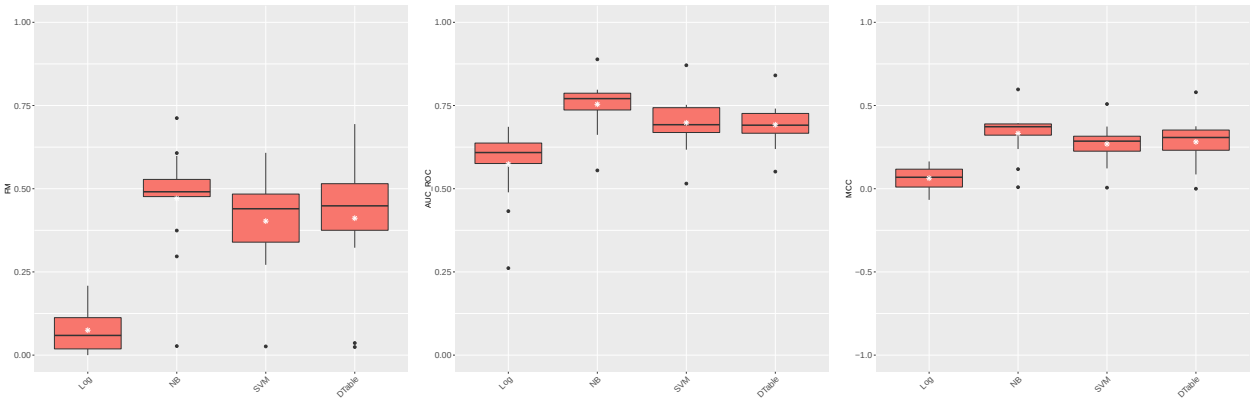
We believe that this not surprising performance could depend on the presence of a particular app namely, *Flutter*, that leads to critically decrease the general performance (see the detailed results in [62]); indeed, as it is possible to see from Table I, it contains only 1% of risky commits on a total of $13,067$ and the operation of balancing was not so effective; indeed a disadvantage of SMOTE is that it does not take into consideration neighbouring examples from other classes. This could lead to an increase of class overlapping [70], [71], so despite the usage of different classifiers, the performances of the models are very low (*e.g.,* 0.03 of F-Measure). Another possible reason is related to the nature of metrics; in particular, they belong to the category of product metrics. It is worth noting that process metrics work better in bug prediction as shown by Kaur *et al.* [15] in the mobile context, but more in general also in standard applications [12].

The performance is even clearer when analysing the results of the Scott-Knott ESD test, shown in Figure 3(a). Despite the statistical test, in terms of F-Measure there is no a clear winner among the models, *Naive Bayes* is statistically better in terms of MCC with respect to all the other classifiers. This means that although it is able to identify a slightly higher number of true positive, it has a stronger correlation with the dependent variable that lead to the best performance. *Decision Table* and *Support Vector Machine* are equivalent, while *Logistic Regression* is the classifier with the worst performance. When analysing in details precision and recall achieved by the classifiers (see online appendix [62]), we noted that all the considered classifiers have a similar precision (*i.e.,* $\approx$0.50). This means that half of the results reported by the models are false positives and we are still far to models exploitable in industrial contexts. Thus the difference reported in terms of *F-Measure* is due to the difference in terms of recall. These results confirm the ones reported in previous studies on bug prediction [19], [22], [59].

Summary for RQ$_2$. *Naive Bayes* obtained the best performance. Although in terms of F-Measure, there is no a clear winner among different classifiers, *i.e., SVM* and *DecisionTable* (49%, 44% and 45% respectively), the improvement of *Naive Bayes* are statically significant with respect to the other classifiers, as shown by results of the Scott-Knott ESD test.

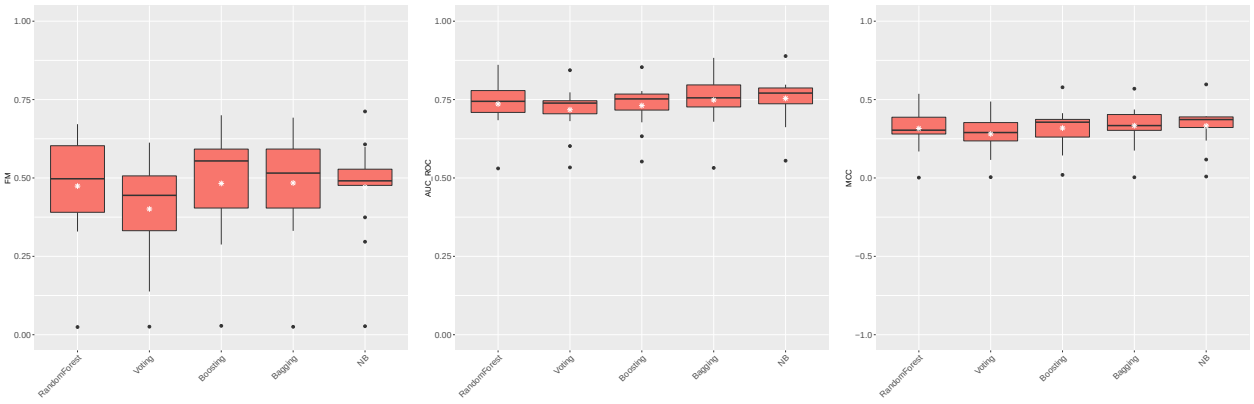
C. RQ$_3$: Do ensemble techniques improve the effectiveness of cross-project prediction models?

The results achieved by the ensemble methods are quietly surprising. In general, these techniques should be able to improve the model performance with respect to single classifiers, as also previous study showed [19], [21]–[23]. In our case, we observed that there is no a clear winner; indeed, looking at Figure 2, the differences with respect to the *Naive Bayes* model are quite small (49% of F-Measure of *Naive Bayes* with respect to 50% *Random Forest*, 54% *Boosting* and 51% *Bagging*). *Boosting* and *Bagging* have slightly better performance than *Random Forest*: this is quite strange, since previous literature frequently showed that *Random Forest* is among the top machine learning approaches to use [35], [64]. We performed the Scott-Knott ESD test, shown in Figure 3(b), that confirmed our hypothesis. In particular, the performance of *Random Forest*, *Boosting*, *Bagging* and *Naive Bayes* are not statistically different, while *Voting* is statistically worse than them. This result confirms a previous study by Bowes *et al.* [59] who showed that applying a *Voting* ensemble classifier could lead to worse results with respect to the weak classifiers on which it is built. Thus, we can conclude that on the one hand, ensemble techniques do not seem to provide important improvements in the prediction of risky commits. This means that future research should be devoted to understand the reasons behind this partially negative result and provide novel methodologies to address it, as also suggested by [86], [87]. On the other hand, it confirms the importance of hyper-parameter tuning [88]. Indeed, it is worth to note that all the single classifiers were tuned using the MULTISEARCH algorithm. It is worth considering that, in case



(a) F-Measure (b) AUC-ROC (c) Matthew Correlation Coefficient

Figure 1. Results achieved by JIT bug prediction models built using standard classifiers



(a) F-Measure (b) AUC-ROC (c) Matthew Correlation Coefficient

Figure 2. Results achieved by JIT bug prediction models built using ensemble classifiers

of ensemble techniques, it is not possible to tune the weak classifier on which they relied.

Summary for RQ$_3$. *Bagging* methods obtained the best performance, *i.e.,* 54% overall value of F-Measure. However, the differences with respect to *Naive Bayes* seems to be quite small, *i.e.,* 49%. Although *Random Forest*, *Boosting* and *Bagging* result statistically significant with respect of *Naive Bayes*, ensemble techniques do not always guarantee better performance with respect to a single classifier.

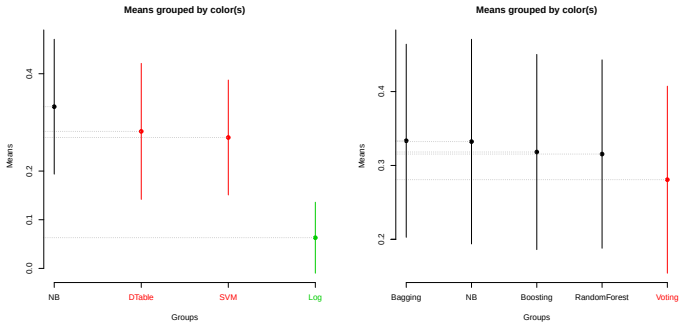
V. Threats to Validity

In this section, we discuss factors that might have affected the validity of the empirical study.

Threats to Construct Validity. As for threats related to the relationship between theory and observation, a first factor is related to the dataset exploited. In our study, we relied on data coming from Commit Guru platform [24] and in order to ensure quality and robustness of data we conducted a preliminary data preprocessing following the guidelines provided by Shepperd *et al.* [89] in order to remove noisy data. Of course, we cannot exclude possible imprecision in the computation of the dependent variable. Moreover about the choice of the machine learning classifier we decided to use techniques that have been widely used in the context of bug prediction [22], [35], [74]–[76].

Threats to Conclusion Validity A first threat regards the validation technique adopted when testing the different JIT bug prediction models experimented. The usage of the Leave-One-Out Cross-Validation [34] was driven by recent results showing that such validation technique is among the ones that are more stable and reliable [90]. In addition, we ran the *SMOTE* algorithm [71] to balance the training set.

As for the evaluation of the performances of the experimented models, we computed *precision*, *recall*, *F-Measure*, *AUC-ROC* and *MCC*, able to provide an overview of the performance of the devised models and of the compared baselines under different perspectives, thus allowing us to better report the performance of JIT bug prediction models. Finally, in both **RQ$_2$** and **RQ$_3$** we statistically confirmed our observations by exploiting the Scott-Knott ESD test [79].



(a) Single Classifiers (b) Ensemble Classifiers and NB

Figure 3. The likelihood of each technique appearing in the top Scott-Knott ESD rank in terms of MCC. The circle dots in the figure indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a technique is in the top-rank for 50% of the studied datasets

Threats to External Validity Threats in this category mainly concern the generalisation of results. In the context of our work, we performed the empirical study that took into account 14 open source mobile apps having different size and scope. At the same time, we considered different types of machine learning techniques. Of course, we cannot claim the generalisability, however part of our future research agenda is to extend the study with more apps that include also non open source software.

VI. Conclusions

Just-in-Time bug prediction [13] is an alternative to traditional bug prediction that is able to predict the presence of bugs at commit-level, *i.e.,* as soon as a new change is committed by developers on the repository. While this technique has been pretty explored in the context of traditional systems, to the best of our knowledge only few previous studies addressed this issue in the context of mobile apps [15], [16]. In particular, Catolino [16] conducted a preliminary study in order to understand the adaptability of the within-project JIT bug prediction model devised by Kamei *et al.* [13] in the context of mobile apps, thus showing a clear limitation of the model in finding bug prone commits and highlighting that further analysis is needed.

In this paper, we (i) deeper analysed which Kamei *et al.*'s metrics are important in the mobile context, (ii) evaluated the performance of JIT bug prediction in a cross-project scenario, and (iii) assessed the impact of classifiers and ensembles selection on the overall bug prediction capabilities on a larger set of mobile apps.

To this aim, first we applied the *InfoGain* algorithm [25] that filtered only the metrics that are positively related to commit defectiveness in the mobile context. Then, we compared the performance of four standard machine learning techniques (*i.e., Logistic Regression* [26], *Decision Table* [27], *Support Vector Machine* [28], and *Naive Bayes* [29]) and four ensemble methods (*i.e., Random Forests* [30], *Voting* [31], *Bootstrap Aggregating* [32], and *Boosting* [33]. The study was conducted on 14 mobile apps and a total of $42,543$ commits.

The achieved results provide the following findings:

- Metrics related to code churn (*e.g., Lines of code deleted*), diffusion of changes (*e.g., Number of modified directories*), and history (*e.g., Number of unique changes*) are important to identify risky commits.
- These results are in line with bug prediction research for traditional systems [12].
- The best performance for single classifiers is achieved by *Naive Bayes*, *i.e.,* 49% of F-Measure, (well appreciated and performing in the context of bug prediction for traditional systems [12]), while *Logistic Regression* performed worst than other classifiers by up to 40% in terms of *F-measure*. This means that the choice of classifier impacted on the performance of the JIT bug prediction models [19], [20].
- Ensemble methods do not provide evident benefits with respect to the single classifiers. Indeed, looking at the performances achieved, the model built using the *Boosting* classifier performed 4-5% better than the one relying on *Naive Bayes*.

The results of this study highlighted that on the one hand, the majority of solid and verified concepts of traditional and JIT bug prediction can be adopted also for mobile apps (*e.g.,* choice of classifier, independent variable), while on the other hand some of them should be modified given the nature of the development process of such applications.

This study opens new challenges for the future that are part of our research agenda. First we plan to compare within- and cross-project training strategies. In this context, we will exploit the concept of local bug prediction [36], a technique that has been already successfully applied to improve the effectiveness of traditional bug prediction. Based on the results of RQ_1, we believe that new metrics suitable for the mobile context should be developed. Finally, given the uneven performance of JIT bug prediction models, we will explore new approaches that combine classifiers such as the one proposed by Di Nucci *et al.* [22] to select the most suitable classifier based on the commits characteristics.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] G. Booch, *Object oriented analysis & design with application*. Pearson Education India, 2006.

[3] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[4] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[5] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 12–21.

[6] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *Requirements Engineering Conference (RE), 2013 21st IEEE International*. IEEE, 2013, pp. 125–134.

[7] A. Holzer and J. Ondrus, "Mobile application market: A developer's perspective," *Telematics and informatics*, vol. 28, no. 1, pp. 22–31, 2011.

[8] A. I. Wasserman, "Software engineering issues for mobile application development," in *FSE/SDP*, 2010.

[9] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 767–778.

[10] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 499–510.

[11] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 106–117.

[12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[13] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[14] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[15] A. Kaur, K. Kaur, and H. Kaur, "Application of machine learning on process metrics for defect prediction in mobile application," in *Information Systems Design and Intelligent Applications*. Springer, 2016, pp. 81–98.

[16] G. Catolino, "Just-in-time bug prediction in mobile applications: the domain matters!" in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 201–202.

[17] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 232–243.

[18] H. van Heeringen and E. Van Gorp, "Measure the functional size of a mobile app: Using the cosmic functional size measurement method," in *IWSM-MENSURA*, 2014.

[19] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 164–173.

[20] 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 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 789–800.

[21] 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.

[22] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

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

[24] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 966–969.

[25] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[26] S. Le Cessie and J. C. Van Houwelingen, "Ridge estimators in logistic regression," *Applied statistics*, pp. 191–201, 1992.

[27] R. Kohavi, "The power of decision tables," in *European conference on machine learning*. Springer, 1995, pp. 174–189.

[28] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.

[29] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*, ser. A Wiley-Interscience publication. Wiley, 1973. [Online]. Available: http://www.worldcat.org/oclc/00388788

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

[31] T. G. Dietterich, "Ensemble methods in machine learning," *International workshop on multiple classifier systems*, pp. 1–15, 2000.

[32] L. Breiman, "Bagging predictors," *Machine learning*, pp. 123–140, 1996.

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

[34] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," in *Encyclopedia of database systems*. Springer, 2009, pp. 532–538.

[35] 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.

[36] 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.

[37] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, 2018.

[38] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 172–181.

[39] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2018.

[40] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 17–26.

[41] J. G. Barnett, C. K. Gathuru, L. S. Soldano, and S. McIntosh, "The relationship between commit message detail and defect proneness in java projects on github," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 496–499.

[42] 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*, vol. 87, pp. 206–220, 2017.

[43] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *Information and Software Technology*, vol. 93, pp. 1–13, 2018.

[44] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.

[45] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 159–170.

[46] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," 2018.

[47] L. D'Avanzo, F. Ferrucci, C. Gravino, and P. Salza, "Cosmic functional measurement of mobile applications and code size estimation," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1631–1636.

[48] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[49] 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 the 7th joint meeting of the European software engineering conference*. ACM, 2009, pp. 91–100.

[50] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[51] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter language reuse," in *Proc. of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 19–24.

[52] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[53] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 9.

[54] 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.

[55] 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.

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

[57] 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.

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

[59] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" *Software Quality Journal*, vol. 26, no. 2, pp. 525–552, 2018.

[60] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *Journal of Systems and Software*, vol. 143, pp. 14–28, 2018.

[61] G. Seni and J. F. Elder, "Ensemble methods in data mining: improving accuracy through combining predictions," *Synthesis Lectures on Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 1–126, 2010.

[62] G. Catolino, D. Di Nucci, and F. Ferrucci. (2019) Cross-project just-in-time bug prediction for mobile app: An empirical assessment - online appendix https://figshare.com/s/9a075be3e1fb64f76b48.

[63] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[64] 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 30th international conference on Software engineering*. ACM, 2008, pp. 181–190.

[65] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

[66] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: an empirical assessment," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 186–195.

[67] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 495–504.

[68] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[69] H. Liu and H. Motoda, *Feature selection for knowledge discovery and data mining*. Springer Science & Business Media, 2012, vol. 454.

[70] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 534–550, 2018.

[71] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[72] K. Bańczyk, O. Kempa, T. Lasota, and B. Trawiński, "Empirical comparison of bagging ensembles created using weak learners for a regression problem," *Asian Conference on Intelligent Information and Database Systems*, pp. 312–322, 2011.

[73] L. Reyzin and R. E. Schapire, "How boosting the margin can also boost classifier complexity," *Proceedings of the 23rd international conference on Machine learning*, pp. 753–760, 2006.

[74] 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, 2010.

[75] E. Ceylan, F. O. Kutlubay, and A. B. Bener, "Software defect identification using machine learning techniques," in *Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on*. IEEE, 2006, pp. 240–247.

[76] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.

[77] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[78] P. Baldi, S. Brunak, Y. Chauvin, C. A. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview," *Bioinformatics*, vol. 16, no. 5, pp. 412–424, 2000.

[79] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transanctions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[80] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[81] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Crowdsourcing user reviews to support the evolution of mobile apps," *Journal of Systems and Software*, vol. 137, pp. 143–162, 2018.

[82] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.

[83] V. N. Inukollu, D. D. Keshamoni, T. Kang, and M. Inukollu, "Factors influencing quality of mobile apps: Role of mobile app development life cycle," *arXiv preprint arXiv:1410.4537*, 2014.

[84] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" 2018.

[85] J. Li, T. Stålhane, J. M. Kristiansen, and R. Conradi, "Cost drivers of software corrective maintenance: An empirical study in two companies," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–8.

[86] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1403–1416, 2012.

[87] G. Catolino and F. Ferrucci, "Ensemble techniques for software change prediction: A preliminary investigation," in *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2018 IEEE Workshop on*. IEEE, 2018, pp. 25–30.

[88] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 321–332.

[89] 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.

[90] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 1–11.