The Journal of Systems and Software 150 (2019) 22–36

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Fine-grained just-in-time defect prediction

Luca Pascarella[a], Fabio Palomba[b,*], Alberto Bacchelli[b]

[a] Delft University of Technology, The Netherlands
[b] University of Zurich, Switzerland

ARTICLE INFO

Article history:
Received 11 May 2018
Revised 13 October 2018
Accepted 2 December 2018
Available online 3 December 2018

Keywords:
Just-in-time defect prediction
Empirical Software Engineering
Mining software repositories

ABSTRACT

Defect prediction models focus on identifying defect-prone code elements, for example to allow practitioners to allocate testing resources on specific subsystems and to provide assistance during code reviews. While the research community has been highly active in proposing metrics and methods to predict defects on *long-term* periods (*i.e.,*at release time), a recent trend is represented by the so-called *short-term* defect prediction (*i.e.,*at commit-level). Indeed, this strategy represents an effective alternative in terms of effort required to inspect files likely affected by defects. Nevertheless, the granularity considered by such models might be still too coarse. Indeed, existing commit-level models highlight an *entire* commit as defective even in cases where only *specific* files actually contain defects.

In this paper, we first investigate to what extent commits are partially defective; then, we propose a novel *fine-grained just-in-time* defect prediction model to predict the specific files, contained in a commit, that are defective. Finally, we evaluate our model in terms of (i) performance and (ii) the extent to which it decreases the effort required to diagnose a defect. Our study highlights that: (1) defective commits are frequently composed of a mixture of defective and non-defective files, (2) our fine-grained model can accurately predict defective files with an AUC-ROC up to 82% and (3) our model would allow practitioners to save inspection efforts with respect to standard just-in-time techniques.

© 2018 Published by Elsevier Inc.

1. Introduction

During software maintenance and evolution, developers constantly modify the source code to introduce new features or fix defects (Lehman, 1980). These modifications, however, may lead to the introduction of new defects (Kim et al., 2008), thus developers must carefully verify that the performed modifications do not introduce new defects in the code. This task is usually performed directly during development (*e.g.,*by running test cases) (Sommerville, 2006) or when changes are reviewed (Bacchelli and Bird, 2013). An efficient way to allocate inspection and testing resources to the portion of source code more likely to be defective is represented by *defect prediction* (Hall et al., 2012), which involves the construction of statistical models to predict the defect-proneness of software artifacts, by mostly exploiting information regarding the source code or the development process.

The problem of defect prediction has attracted the attention of many researchers in the past decade, who tried to address it by (i) conducting empirical studies on the factors making artifacts more defect-prone (*e.g.,* Basili et al., 1996; Khomh et al., 2012; Palomba et al., 2018; Palomba et al., 2017; Posnett et al., 2013; Spadini et al., 2018b; Tufano et al., 2017) and (ii) proposing novel prediction models aimed at accurately predicting the defect-proneness of the source code (*e.g.,* Bell et al., 2013; Hassan, 2009; Menzies et al., 2013; di Nucci et al., 2017; Palomba et al., 2016; Rahman and Devanbu, 2013).

Most of the existing techniques evaluate the defectiveness of software artifacts perform *long-term* predictions. Analyzing the information accumulated in previous software releases, these models predict which artifacts are going be more prone to defect *in future releases*. For instance, Basili et al. investigated the effectiveness of Object-Oriented metrics (Chidamber and Kemerer, 1994) in predicting post-release defects (Basili et al., 1996), while other approaches consider process metrics (*e.g.,*the entropy of changes Hassan, 2009) or developer-related factors (Bell et al., 2013; di Nucci et al., 2017) for the same purpose.

Kamei et al. reported that these long-term defect prediction models—despite their good accuracy—may have a limited usefulness in practice because they do not provide developers with immediate feedback (Kamei et al., 2013), thus not avoid the introduction of defects during the commit of artifacts on the repository. To overcome this limitation, a recent trend is the investigation of

* Corresponding author.
E-mail addresses: l.pascarella@tudelft.nl (L. Pascarella), palomba@ifi.uzh.ch (F. Palomba), bacchelli@ifi.uzh.ch (A. Bacchelli).

https://doi.org/10.1016/j.jss.2018.12.001
0164-1212/© 2018 Published by Elsevier Inc.

just-in-time prediction models, *i.e.*,techniques exploiting the characteristics of a commit to perform *short-term* predictions of the likelihood of a commit introducing a defect. With this solution, a developer can limit the effort required to diagnose problems since s/he focuses on the committed artifacts only (Kamei et al., 2013). Among the studies investigating *just-in-time* prediction models, Kamei et al. (2013, 2016) defined 14 metrics characterizing a commit under five perspectives, demonstrating how such metrics can be successfully exploited for predicting defective commits either in the case the model is trained using previous data of the project (Kamei et al., 2013) and in the case the training information come from different projects (Kamei et al., 2016). Other approaches proposed the use of deep-learning (Yang et al., 2015), textual analysis (Barnett et al., 2016), and unsupervised methodologies (Yang et al., 2016).

It is reasonable to think that, in a real-world scenario, a commit may be *partially* defective, *i.e.*,it may be composed of both defective and non-defective files. In this case, despite the advantages provided by *just-in-time* defect prediction, a developer might still need to spend a considerable effort to locate the files of a commit that are actually defective. For instance, during a Modern Code Review (MCR) the reviewers iterate several times over the proposed set of changes and the amount of time spent finding a subset of defective files might substantially increase (Bacchelli and Bird, 2013). In this paper, we aim at making a further step ahead in the context of *just-in-time* defect prediction by investigating the original problem at a finer granularity. Particularly, our goal is to investigate the prominence of partially defective commits and, should they be a significant amount, devise a defect prediction model to identify the *defective files within a commit*.

To this aim, we firstly performed an exploratory study to characterize defective commits and evaluate whether fine-grained solutions are actually needed. In the second place, we built a *fine-grained just-in-time* defect prediction model adapting 24 basic features previously defined in the papers by Kamei et al. (2013) and Rahman and Devanbu (2013). Finally, we assessed the performance of the model in terms of (i) accuracy of the predictions and (ii) effort developers can save using our model with respect to state-of-the-art *just-in-time* prediction models. The study was conducted considering 10 major open source systems and 160,515 commits. Key findings of our investigations revealed that (i) almost 43% of defective commits are composed of a mixture of both defective and non-defective resources, (ii) the devised *fine-grained* model obtained an AUC-ROC up to 82% when locating defective files in a commit, and (iii) our model is more cost-effective than the state of the art *just-in-time* model.

Structure of the paper. Section 2 reports background, related work, and a concept of the envisioned solution. Section 3 reports the methodology used to address our research goal as well as possible threats that might influence our findings, while Section 4 presents the results of the study. Finally, Section 5 concludes the paper.

2. Background and related work

In this section we introduce the terminology used through the paper, discuss the related work, and motivate our study.

2.1. Terminology

Throughout the paper, we frequently refer to the following five concepts:

Defect. To define a condition in which a software system does not meet its requirements, we use the term *defect*, among all the possible terms (*e.g.*,*bug* and *fault* Hall et al., 2012).

Defect-Inducing/-Fixing Change. We identify two events in the life of a defect: (i) the *defect-inducing change, i.e.*,the code change that inserts the defect into a project and (ii) the *defect-fixing change, i.e.*,the code change that fixes the defect.

Commit. In most modern collaborative software projects, authors develop code relying on version system control tools such as Git.[1] Such tools track changes as *commits*, which are documented changes that involve one or more files.

Non-/Partially/Fully Defective Commit. We define three classes of commits: *non-defective commits* (when all the committed files are changed without introducing any defect), *fully defective commits* (when all the committed files are changed introducing defects), and *partially defective commits* (when a subset of the committed files are changed introducing a defect). The top part of Fig. 1 depicts a part of the history of an example software system, with the activities made on the versioning system after a system's release. A set of commits $C = \{c_x, \ldots, c_{x+4}\}$ are performed by developers to evolve the system; all the commits change the same three files (`A.c`, `B.c`, and `C.c`). In Fig. 1, we see examples of non-defective commits (c_x and c_{x+3}), partially defective commits (c_{x+1}, c_{x+4}), and a fully defective commit (c_{x+2}).

2.2. Related work

In this section, we discuss the related work that inspired and guided this study, considering long- and short-term defect prediction.

2.2.1. Long-term defect prediction

Long-term defect prediction pertains to models able to classify defect-prone files in future releases of a software project. Several studies addressed this problem in the recent years (a relatively recent survey has been compiled by Hall et al., 2012). Basically, the proposed models differ for the source of information used to predict the defectiveness of a class: the main distinction is between static, product information and historical, process data.

Product Information. Structural data are computed with metrics such as the McCabe's cyclomatic complexity (McCabe, 1976) or the Chidamber and Kemerer (CK) (Chidamber and Kemerer, 1994) metrics. These product metrics have been investigated in several studies (Arisholm et al., 2010; Di Nucci et al., 2017; El Emam et al., 2001; Graves et al., 2000; Herraiz et al., 2007; Leszak et al., 2002; Nagappan and Ball, 2005; Gyimóthy et al., 2005) and researchers have shown how such metrics can provide useful contribution in the prediction of defective classes. For instance, Nagappan and Ball (2005) found that a model based on code metrics may achieve up to 83% of accuracy in the identification of defect-prone classes.

Process Information. Historical data are computed with metrics such as relative code churn, entropy of changes, or developer-related factors (Hassan, 2009; Graves et al., 2000; Hall et al., 2012; Moser et al., 2008; Munson and Elbaum, 1998; di Nucci et al., 2017). Also in this case, researchers provided empirical evidence on the value of such metrics for defect prediction. For instance, Moser et al. (2008) performed a comparative study analyzing static- and historical-based predictors, concluding that metrics computed over the history of projects are better predictors and can significantly improve the performance of defect prediction models.

Combining Information. Later on, D'Ambros et al. (2012) found that combined techniques work better than models based

[1] https://git-scm.com/.

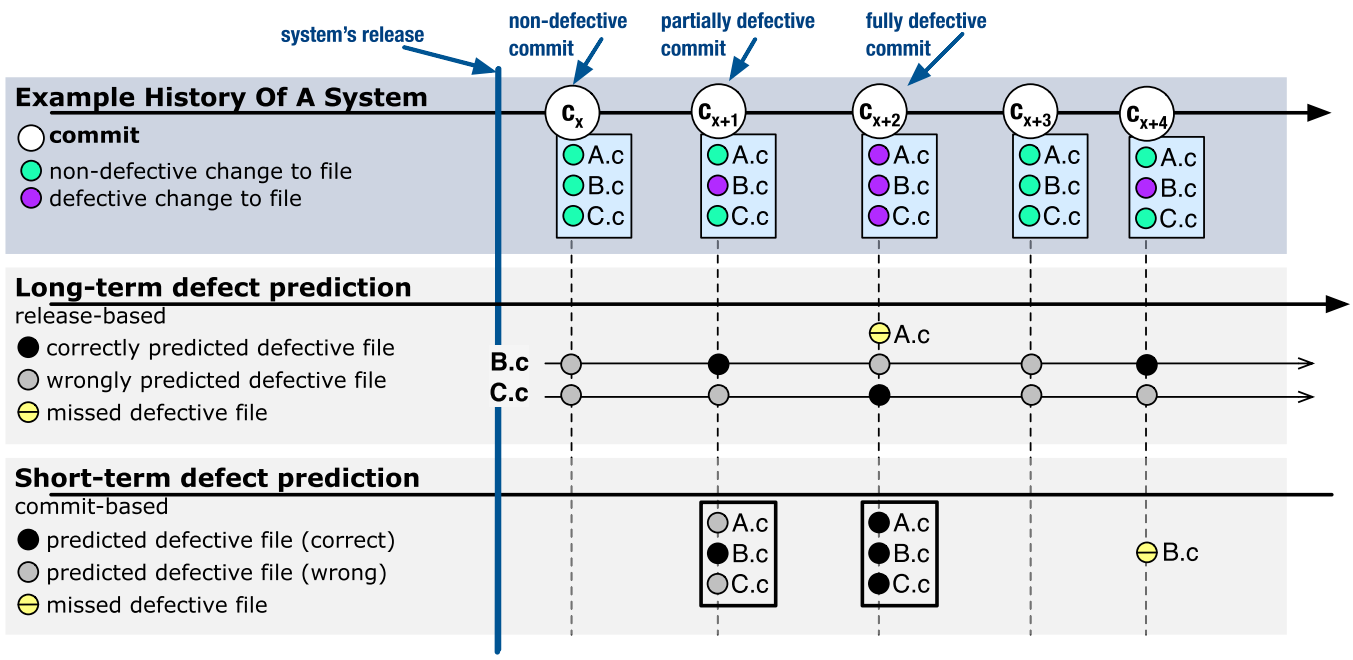

Fig. 1. An example set of (defective) commits after release to files in a software system.

on single unique set of metrics. On the basis of this result, di Nucci et al. (2017) defined a combined model based on a mixture of static and historical metrics able to outperform the prediction capabilities of single models. Finally, Menzies et al. (2013) introduced the concept of *local* defect prediction, an approach in which classes that will be used for training the classifier are firstly clustered into homogeneous groups to reduce the differences among such classes and obtain higher prediction accuracy.

We build on top of this line of work, by considering the features that are better able to predict defects at file-level—a key attribute that we include in our model.

2.2.2. Short-term defect prediction

Short-term defect prediction refers to models able to classify defect-prone at commit time. Previous work motivated the introduction of this new strategy with the need of having tools able to locate defects in the shortest possible time (Mockus and Weiss, 2000). While Madeyski and Kawalerowicz (2017) proposed the idea of continuous defect prediction, Mockus and Weiss (2000) addressed the problem by proposing a model based on the change-proneness of files to predict defects at commit-level.

Characteristics of Defect-Introducing Changes. Other studies (*e.g.*, Śliwerski et al., 2005 and Eyolfson et al., 2011) tried to localize defect-introducing changes in open source projects by means of correlation between the defectiveness of a commit and the experience of developers. Śliwerski et al. (2005) also discovered that defect-introducing changes are generally a part of large transactions. The "unnaturalness" of defective code was subsequently confirmed by Ray et al. (2016), who discovered that source code presenting defects is characterized by higher entropy than non-defective code. They also found that source code entropy might be a valid and simpler way to complement the effectiveness of static analysis tools (*e.g.*,CheckStyle[2]) in recommending to developers the areas of source code where to focus inspection activities.

Ad Hoc Models. Jiang et al. (2013) proposed the concept of "personalized" defect prediction by proposing a technique able to create a different model for each developer. Their results report that such a technique outperforms existing *just-in-time* defect prediction models. Along with this line, Xia et al. (2016) improved the aforementioned technique by Jiang et al. using a multi-objective genetic algorithm that firstly builds a defect prediction model for each developer, and then combine these models assigning different weights with the aim of maximizing F-Measure and cost-effectiveness. With respect to these papers, our approach has not the goal to build a prediction model for each developer, but instead that of providing feedback on defect-prone classes within the scope of a commit: further analysis will evaluate the possible benefits provided by the creation of personalized models in the context of fine-grained just-in-time defect prediction.

Just-In-Time. The studies by Kamei et al. (2013, 2016) are great source of inspiration for our work. They proposed a *just-in-time* quality assurance technique that predict defects at commit-level trying to reduce the effort of a reviewer (Kamei et al., 2013). Later on, they also evaluated how *just-in-time* models perform in the context of cross-project defect prediction (Kamei et al., 2016). Their main findings report good accuracy for the models in terms of both precision and recall, but also in terms of saved inspection effort. Our work is complementary to these papers. In particular, we start from their basis of detecting defective commits and complement this model with the attributes necessary to filter only those files that are defect-prone and should be more thoroughly reviewed. Rahman et al. (2011), Yang et al. (2015), and Barnett et al. (2016) proposed the usage of alternative techniques for *just-in-time* quality assurance, such as cached history, deep learning, and textual analysis, reporting promising results. We did not investigate these further in the current paper, but studies can be designed and carried out to determine if and how these techniques can be used within the model we present in this paper to further increase its accuracy.

[2] http://checkstyle.sourceforge.net.

2.3. Motivating example

We discuss an example in which a developer uses long- vs. short-term defect prediction models while inspecting a commit, in order to show some of the limitations of these approaches.

The top part of Fig. 1 depicts an example history of a software system, with the activities made on the versioning system after a system's release. A set of commits $C = \{c_x, \ldots, c_{x+4}\}$ are performed by developers to evolve the system. For sake of clarity, suppose that the files `A.c`, `B.c`, and `C.c` are always changed in the considered commits after the system's release. The small circles in the top bar represent changes made to files in each commit and the colors represent whether these changes introduce a defect (purple) or not (dark green) in these files. In addition, a black box surrounds all the files in the same commit. In the following we describe the behavior of the two aforementioned prediction models:

Long-term defect prediction. Based on the information gathered before the system's release, a long-term defect prediction model would mark certain files as defect-prone for the entire period leading to the issue of the next release. In our example, the model marks the files `B.c` and `C.c` as defect-prone and `A.c` as non-defective. The model classifies both `B.c` and `C.c` as defective starting from the system's release onward, in Fig. 1 we depict this behavior with a horizontal small arrows, thus showing that `B.c` and `C.c` are considered as defective in *every commit*. Indeed, the model does not provide any information about the exact commit that will likely lead to the introduction of a defect. This model would issue warnings about these files on each commit involving them. In our example, this represents an unjustified extra-effort for the developer inspecting the commit. As found in previous research, this unjustified extra-effort derived from using a tool can reduce the developers' confidence in the prediction (Parnas and Lawford, 2003), thus leading to miss important defects in future commits involving actual defect-prone artifacts. Finally, we see that the model does not classify as defective the code in file `A.c`, also when a defect is introduced in commit c_{x+2} (the missed defective file is depicted with a yellow circle).

Short-term Defect Prediction. As an alternative, a reviewer may adopt a short-term defect prediction model such as the *just-in-time* one proposed by Kamei et al. (2013). In this scenario, a developer is pointed to analyze more in depth only the files referring to a commit marked as potentially defective by the model. However, the number of resources to inspect might be still high depending on the number of files committed and the wasted effort on how many are defect-free. For instance, in the commit c_{x+1} shown in Fig. 1, only the file `B.c` introduces a defect, while the others are defect-free, yet a warning from the tool would be issued; the developer may need to analyze some non-defective files before finding the actual defect. Thus, while short-term solutions can significantly reduce the reviewers' effort, they might still produce extra-effort in cases a commit is partially defective. Furthermore, in our example, file `B.c` is again defective in commit c_{x+4}, but it is not marked as such, since the model does not recognize the commit as defective (in the figure, the missed defective file is depicted with a yellow circle).

The goal of our work is to make the first steps in supporting software developers during the inspection of a commit (*e.g.,*in a code review), by striving to overcoming the aforementioned limitations of existing defect prediction models in this context. The next section details our research questions and the research method.

3. Methodology

This section defines the overall goal of our study, motivates our research questions, and outlines our method.

3.1. Research questions

The *goal* of the study is to investigate how frequently commits are only partially defective and to devise a defect prediction model able to identify the files with the changes that are more likely to introduce a defect. We set up our work around three research questions. The first one is a preliminary analysis aimed at assessing the extent to which a defect prediction model is actually able to estimate the defect-proneness of files within a commit. To this aim, we investigate the ratio of commits that contain both defective and not defective files. Should the frequency of partially commits be low, standard just-in-time models, such as the one devised by Kamei et al. (2013) would be sufficient, while in case there should be a notable percentage of commits presenting both defective and non-defective files, then defect prediction models working at a finer granularity than standard just-in-time ones would be desirable.

RQ$_1$. *What is the ratio of partially defective commits?*

Once assessed the actual need for finer grained solutions, we devise a defect prediction model to predict defective files at commit scope.

RQ$_2$. *To what extent can we predict defect-inducing changes at file-level in a commit?*

In addition to assessing the model as a whole, we also evaluate which features provide the highest contribution to the achieved performance.

RQ$_3$. *What are the features of the devised model that the most to its performance?*

Finally, we are also interested in understanding how much effort could be saved when using the proposed model, comparing it to the just-in-time defect prediction model proposed by Kamei et al. (2013) as our baseline.

RQ$_4$. *How much effort can be saved using a fine-grained just-in-time defect prediction model with respect to a standard just-in-time model?*

In the following sections, we describe the steps we perform to answer our three research questions.

3.2. Subject systems

To conduct our analysis, we focused on open-source software systems and defined multiple selection criteria: We selected software systems (i) written in the most common programming languages (C, C++, Java, JavaScript, Ruby, and Perl, i.e., the most popular programming languages Cass and Bulusu, 2018), (ii) having different size and scope, and (iii) having a change history composed of at least 1000 commits. We preferred open-source systems where a versioning system is used to track all changes. The access to the source code history enables the computation of metrics with static analysis tools. Moreover, to increase the generalizability of our research, we selected software projects having different domains and programming languages. Note that our selection is not intended to be statistically significant, but rather we just aim at selecting a various set of systems to assess the performance of our prediction model in different contexts (*e.g.,*when considering projects having different change history sizes). In practice, we

Table 1
Characteristics of the subject software systems.

Project	KLOC	Developers	Commits	Defective commits
Accumulo	102	66	8747	1399
Angular-js	87	1589	8467	1525
Bugzilla	239	99	9788	3621
Gerrit	79	38	22,232	4001
Gimp	102	216	38,240	8412
Hadoop	291	92	15,556	2606
JDeodorant	70	9	1105	199
Jetty	88	70	12,638	3286
JRuby	129	322	38,894	8945
OpenJPA	822	25	4848	1502
Overall	**2009**	**2526**	**160,515**	**35,496**

started from the entire list of open source projects available on GitHub; then we filtered out systems not implemented in the considered programming languages and with less than 1000 commits in their history. Successively, among 2,362,287 project candidates, we considered only the most popular projects for a given domain or scope; finally, we randomly selected the ten open-source software systems reported in Table 1. For each system, the table reports size (in terms of KLOCs), number of contributors, number of commits, and the information on the number of defective commits.

3.3. RQ_1 – *Investigating defective commits*

To answer our first research question, we analyze the ratio of the defective files (*i.e.,*source code, configuration, and auxiliary files) contained in defective commits. To this aim, for each commit c_i of the change history of a system S, we identify the set *defectiveFiles*(c_i) composed of the defective resources contained in c_i. To the best of our knowledge, there is not a publicly available dataset reporting this information: Previous work defined datasets of defective commits (Kamei et al., 2013), without providing details on which of the resources in a certain commit were actually defective. For this reason, we build our own dataset as detailed in the following.

Data extraction. To automatically identify the set of defective files in each of the commits of the considered systems, we rely on the SZZ algorithm (Śliwerski et al., 2005; Williams and Spacco, 2008). SZZ exploits the annotation/blame feature of a versioning system to *estimate* the lines of code of a file that induced a certain defect, thus retrieving files that are defect-inducing in each commit. More formally, the algorithm implements the following steps:

1. For each file f_i (where $i = 1 \ldots n$) involved in a defect fixing commit *dfc*, the algorithm *prevVersion*(*commit, file*) extracts the last version of the file before the defect fixing commit: *prevVersion*(*dfc*, f_i);
2. Starting from the commit *prevVersion*(*dfc*, f_i), for each line of code in f_i changed to fix the defect in *dfc*, the algorithm uses `git blame` to detect the file revision where the last change to that line occurred. We identify comments and empty lines using island parsing (Moonen, 2001) and we exclude f_i if no other code is touched. This step outputs the commits in which a defect in file f_i is introduced.

The SZZ algorithm takes as input the list of defects that are *already fixed* by developers, excluding the open ones,[3] but the analysis and the effect of considering open issues will be considered in future work (*e.g.,*exploiting tools such as ReLink Wu et al., 2011).

Data analysis. Once extracted the defective files involved in defective commits, we answer $\mathbf{RQ_1}$ in two ways. First, we measure how many defective commits are *partially* defective, *i.e.,*they contain a mixture of both defective and non-defective resources. This analysis allow us to understand the magnitude of the problem investigated: If the vast majority of defective commits is composed of only defective artifacts, then standard *just-in-time* defect prediction models would suffice; conversely, if a significant part of defective commits is *partially* defective, then the introduction of fine-grained solutions might be worthwhile. Second, we further analyzed the set of *partially* defective commits, by measuring the ratio between defective and non-defective files they contain. More formally, we computed the $defectiveFiles_{dc}$ ratio as follow:

$$defectiveFiles_{dc} = \frac{\#defectiveFiles(dc)}{\#files(dc)} \quad (1)$$

where $\#defectiveFiles(dc)$ represents the number of defective files in the defective commit *dc*, and $\#files(dc)$ the total number of files in *dc*. This analysis helped us to understand the intrinsic characteristics of *partially* defective commits. Also in this case, if the resulting ratio is high (most files are defective in *partially* defective commits), then the adoption of fine-grained solutions would be not worthwhile.

3.4. RQ_2 – *The fine-grained JIT model*

To answer our second research question, we build a *fine-grained just-in-time* defect prediction model and evaluate its performance. In the following, we describe (i) the independent variables, *i.e.,*the metrics on which the model relies, (ii) the dependent variable, *i.e.,*the characteristic that the model have to predict, (iii) the machine learner performing the predictions, and (iv) the validation methodologies to estimate the accuracy.

I. Independent variables. This step consists in extracting and quantifying the characteristics of each file involved in a commit. To this purpose, we considered the 24 basic features shown in Table 2. These features represent a modified version of those previously proposed by Kamei et al. (2013) and Rahman and Devanbu (2013). We adapted the previous metrics to work at file-level in a commit. The column 'Description' in Table 2 details the implementation of the metrics in our context. The choice of the independent variable is driven by two goals: (i) to understand the value of standard *just-in-time* measures in a fine-grained context; (ii) to investigate whether metrics originally proposed in the context of long-term defect prediction to predict defective files may also provide useful contributions when employed in the prediction of defective files contained in a change set.
Furthermore, the chosen metrics help us to characterize commits under different perspectives, thus allowing us to evaluate which metric types are more relevant in our context. Specifically, we selected metrics to measure (i) the developers' experience (e.g., the experience of the committer Kamei et al., 2013), (ii) structural and process factors of the files in the commit (e.g., the lines of code added or the number of previous changes of a committed file Rahman and Devanbu, 2013), and (iii) factors related to the neighbors' of a committed file, which have been shown to be relevant for predicting the defectiveness of files Rahman and Devanbu (2013). Although other metrics have been proposed in the contexts of both code review (*e.g.,* by McIntosh et al., 2014; Kononenko et al., 2015) and defect prediction (*e.g.,* D'Ambros et al., 2012; di Nucci et al., 2017), the selected metrics better allow us to verify the role of a larger set of metrics that have been previously adopted for traditional short- and long-term defect prediction. Further studies can be conducted to investigate the addition of other metrics in our context.

[3] Open issues might be not verified by developers (*i.e.,*they might be not real defects).

Table 2
List of the independent/predicting variables adapted from Rahman and Devanbu (2013)* and Kamei et al. (2013)**.

Acronym	Name	Enumerate	Ref.
COMM	Commit Count	Number of changes to the file up to the considered commit	*
ADEV	Active Dev Count	Number developers who modified to the file up to the considered commit	*
DDEV	Distinct Dev Count	Cumulative number of distinct developers contributed to the file up to the considered commit	*
ADD	Normalized Lines Added	Normalized number of lines added to the file in the considered commit	*
DEL	Normalized Lines Deleted	Normalized number of lines removed to the file in the considered commit	*
OWN	Owner's Contributed Lines	Boolean value indicating whether the commit is done by the owner of the file	*
MINOR	Owner's Contributed Lines	Number of contributors who contributed less than 5% of the file up to the considered commits	*
SCTR	Changed Code Scattering	Number of packages modified by the committer in the commit	*
NADEV	Neighbor's Active Dev Count	Number of developers who changed the files involved in commits where the file has been modified	*
NDDEV	Neighbor's Distinct Dev Count	Cumulative number of distinct developers who changed the files involved in commits where the file has been modified	*
NCOMM	Neighbor's Commit Count	Number of commits made to files involved in commits where the file has been modified	*
NSCTR	Neighbor's Commit Count	Number of different packages touched by the developer in commits where the file has been modified	*
OEXP	Neighbor's Commit Count	Percentage of lines authored in the project	*
EXP	All Committer's Experience	Mean of the experiences of all the developers	*
ND	Number of modified directories	Number of modified directories	**
Entropy	Distribution of modified code across each file	Entropy of changes of the file up to the considered commit	**
LA	Lines of code added	Number of lines added to the file in the considered commit (absolute number of the ADD metric)	**
LD	Lines of code deleted	Number of lines removed to the file in the considered commit (absolute number of the DEL metric)	**
LT	Lines of code in a file before the change	Lines of code in the file before the change	**
AGE	Average interval between the last and the current change	The average time interval between the last and the current change	**
NUC	Number of unique changes to the modified files	Number of times the file has been modified alone up to considered commit	**
CEXP	Experience of the committer	Number of commits made on the file by the committer up to the considered commit	**
REXP	Recent developer experience (last x months)	Number of commits made on the file by the committer in the last month	**
SEXP	Developer experience on a subsystem	Number of commits made by the developer in the package containing the file	**

From a methodological standpoint, the process metrics adapted from Rahman and Devanbu (2013) (*i.e.*,COMM, ADEV, DDEV, ADD, DEL, OWN, MINOR, SCTR, NADEV, NDDEV, NCOMM, NSCTR, OXEP, and EXP) were always evaluated considering the commits up to the commit of interest. Similar adjustments were applied for the metrics proposed by Kamei et al. (2013). For instance, the NUC metric represents the number of unique changes to the files modified in a commit. In our case, we adjust NUC to represent the number of times a single file involved in a commit is modified alone up to the considered commit. Descriptions of how we adapted the Kamei et al. (2013) and Rahman and Devanbu (2013) metrics are reported in Table 2.

II. Dependent variable. The characteristic to measure is the defectiveness of files contained in a commit. To this aim, we exploited the dataset built in the context of $\mathbf{RQ_1}$ (*i.e.*,we used the output of the SZZ algorithm as a dependent variable to predict).

III. Machine learner. In this stage, we needed to select a machine learning classifier able to use the independent variables to infer the defectiveness of files in a change set (Friedman et al., 2001). To this aim, we tested different classifiers (using the validation methodologies described later in this section), *i.e.,Binary Logistic Regression* (le Cessie and van Houwelingen, 1992), *J-48* (Malhotra, 2015), *ADTree* (Freund and Mason, 1999), *Multilayer Perceptron* (Van Der Malsburg, 1986), *Naive Bayes* (John and Langley, 1995), and *Random Forest* (Liaw and Wiener, 2002). As a result, we found that the *Random Forest* technique (Liaw and Wiener, 2002) is the one having the highest performance, in line with previous findings (Jiang et al., 2008; Robnik-Sikonja, 2004). A complete report of such analysis is available in our online appendix (Pascarella et al., 2018).
Such classifiers builds several decision trees, each of them containing nodes representing a condition on a certain feature that splits the dataset into two. A condition is chosen based on the so-called *Mean Decrease in Impurity* (MDI) (Grabmeier and Lambe, 2007), a metric able to measure the extent to which the value of a feature can correctly discriminate the dependent variable. It is important to point out that the selected classifier automatically performs a feature selection, thus avoiding the well-known problem of multi-collinearity (O'Brien, 2007) that occurs when two or more independent variables correlate with each other, possibly affecting the performance of the classifier.

IV. Validation methodologies. The final step to answer $\mathbf{RQ_2}$ is related to the validation of the model. Commonly used techniques such as *ten-fold cross* (Devijver and Kittler, 1982; Tan et al., 2015), or *leave-out-out cross-validation* (Sammut and Webb, 2010) are not suitable for the validation of just-in-time defect prediction models because the data points (*i.e.*,the commits) follow a certain time order: *Time-insensitive* validation strategies might cause a model to be trained using future data that should not be known at the time of the prediction (Tan et al., 2015). For this reason, we adopt a *time-sensitive* analysis where the defectiveness of a commit c_i is evaluated by a model trained using the data coming from the previous three months of history of the system considered. In other words, while the training

set is composed of three-month data, the test set is represented by each commit singularly. Doing so, we exclude the first three months of change history, because of the lack of data needed to perform a proper validation (Tan et al., 2015). Our choice of considering three-month periods is based on: (i) choices made in previous work (Hassan, 2009; di Nucci et al., 2017; Tan et al., 2015) and (ii) the results of an empirical assessment we performed on such a parameter. The empirical assessment showed that the best performance for the devised model is achieved by using three-month periods. In particular, we experimented with time windows of one, two, three, and six months. The complete results are available in our replication package (Pascarella et al., 2018).
Afterward, we measure the performance of the model using *precision* and *recall* (Baeza-Yates and Ribeiro-Neto, 1999):

$$precision = \frac{|TP|}{|TP + FP|} \tag{2}$$

$$recall = \frac{|TP|}{|TP + FN|} \tag{3}$$

where *TP*, *FP*, and *FN* are:

- True Positives (*TP*): elements that are correctly retrieved by the *fine-grained just-in-time* prediction model (*i.e.*,defective files correctly classified as such);
- False Positives (*FP*): elements that are wrongly classified by the *fine-grained just-in-time* prediction model (*i.e.*,non-defective files misclassified as defective by the model);
- False Negatives (*FN*): elements that are not retrieved by the *fine-grained just-in-time* prediction model (*i.e.*,defective files misclassified as non-defective by the model).

In addition, to have a unique value that synthesizes precision and recall we also measure the *F-measure, i.e.*,the harmonic mean of precision and recall:

$$F - Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{4}$$

While the metrics described so far have been widely used in the past to evaluate defect prediction models (Hall et al., 2012), most of the classifiers output a probability ranging between 0 and 1 representing the likelihood of a code component to be part of a certain class (*i.e.*,in our case, to be *defective* or *non-defective*). The threshold used to discriminate the two classes (in most cases—as well as in this work—such threshold is set to 0.5) influences the computation of both precision and recall, and as a consequence of F-Measure. ROC plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1; the diagonal represents the expected performance of a random classifier. AUC computes the area below the ROC and allows us to have a comprehensive measure for comparing different ROCs: An area of 1 represents a perfect classifier (all the defective methods are recognized without any error), whereas for a random classifier an area close 0.5 is expected (since the ROC for a random classifier tends to the diagonal). To have a detailed view of the performance of the model in the different cases found in **RQ$_1$**, in Section 4.2 we report the evaluation metrics achieved when ran the model over the set of (i) all the defective commits in the dataset, (ii) partially defective commits only, and (iii) fully defective commits only.

3.5. ***RQ$_3$*** – *Investigating the importance of the features*

While in **RQ$_2$** we provide an overview of the accuracy of the devised model in predicting defective files within a commit, **RQ$_3$** has the goal of investigating which features contribute the most to the prediction capabilities. To address this point, we use an *information gain* algorithm (Quinlan, 1986) to quantify the gain provided by each independent variable to the prediction of defective files within commits. Formally, let M be the devised *fine-grained just-in-time* prediction model, let $F = \{f_1, \ldots, f_n\}$ be the set of features used by M, the *information gain* algorithm (Quinlan, 1986) applies the following formula to compute the difference in entropy:

$$InfoGain(M|f_i) = H(M) - H(M|f_i) \tag{5}$$

where the function $H(M)$ indicates the entropy of the model that includes the feature f_i, while the function $H(M|f_i)$ measures the entropy of the model that does not include f_i. Entropy is computed as follow:

$$H(M) = -\sum_{i=1}^{n} prob(f_i) \log_2 prob(f_i) \tag{6}$$

The algorithm quantifies the degree of uncertainty in M that is reduced by considering the feature f_i. In our work, we employ the *Gain Ratio Feature Evaluation* algorithm (Quinlan, 1986), which ranks $f_1, \ldots, f_n$ in descending order based on the contribution provided by each feature to the decisions made by M. More specifically, the output of the algorithm is represented by a ranked list in which the features having the highest expected reduction in entropy are placed at the top. During this step, we also verified—through the `evaluateAttribute` function of the Weka[4] implementation of the algorithm—whether a certain feature mainly contributes to the identification of defective or non-defective files, *i.e.*,if there exists a positive or negative relationship between the feature and the defect-proneness of files.

3.6. ***RQ$_4$*** – *Measuring the saved effort*

For **RQ$_4$** we investigate the potential benefits in terms of saved effort that the *fine-grained just-in-time* defect prediction model provides to a developer analyzing the committed files to discover possible defects (*e.g.*,in a code review). Specifically, we perform an effort-aware validation as recommended by Ostrand et al. (2005). In this formulation, a technique is assessed on the fraction of defects it can detect while varying the effort required to locate them. As done in previous work (Kamei et al., 2013), we first rank the files to inspect according to their probability of being defective, as it is assigned by the automated classifier (in our case, *Random Forest*); then we measure the percentage of defects that a developer would identify as the effort spent in analyzing the suggested defective files increases. To approximate such an effort, we use the *number of lines of code to inspect*; this metric has been shown to be a surrogate measure of the effort needed for testing or reviewing a module, as code and cognitive complexity are strongly related to size (Nagappan and Ball, 2005). Thus, size can be considered as a lightweight and efficient solution to estimate the developer's effort in inspecting a code change (Arisholm et al., 2010; Kamei et al., 2010; Mende and Koschke, 2009; 2010).

We compare our model to the traditional *just-in-time* defect prediction model proposed by Kamei et al. (2013). The selection of this baseline is driven by experimental tests, where we found that this approach works better than the twelve unsupervised techniques proposed by Yang et al. (2016). In particular, the model by Kamei et al. (2013) achieves an AUC-ROC 6% higher than the best

[4] https://www.cs.waikato.ac.nz/ml/weka/.

unsupervised technique, which was the one that predicts a commit as defective in case of a number of committed files higher than eight. We report the results of this additional analysis in our online appendix (Pascarella et al., 2018).

To perform a fair comparison, the baseline relies on the same predictors used by Kamei et al. in their experiments and is trained using the best performing classifier (*i.e.,Random Forest*, the same used by our approach). We also empirically evaluate the performance of the several classifiers, namely *Binary Logistic Regression* (le Cessie and van Houwelingen, 1992), *J-48* (Malhotra, 2015), *ADTree* (Freund and Mason, 1999), *Multilayer Perceptron* (Van Der Malsburg, 1986), *Naive Bayes* (John and Langley, 1995), and *Random Forest* (Liaw and Wiener, 2002), when applied on the model by Kamei et al. (2013). Also in this case, *Random Forest* classifier outperforms the others.

As the baseline can only assign defect probabilities to commits (because of the commit-level granularity), we assume that the same probability holds for all files within that commit. In other words, if a commit is considered defective by Kamei et al.'s technique, then all the files within that commit are considered as potentially defective and have the same probability to require further inspection by a developer. To determine in which sequence the developer would inspect the files in the same commit, we use the *alphabetical* order, because it is the normal order offered by both IDEs and code review tools (Baum et al., 2017). Once we assign the probabilities/order to all the files, we rank them in descending order and compare it with the ranking provided by our technique. It is worth noting that we expect our technique to outperform this baseline, as by definition it aims at lowering the granularity of the information presented to developers. Nevertheless, we still consider this comparison useful because we can verify whether and how much our approach actually meet the expected goal.

Finally, we perform a comparison with the *optimal* approach that ranks all the actual defective files first, starting from the smallest to the largest. In this way, we can investigate how far our technique is with respect to optimal scenario as well as how much it improves upon existing *just-in-time* approaches.

Data analysis. To quantify the differences between our model and the baselines, we use the P_{opt} and P_k evaluation metrics (Mende and Koschke, 2009). P_{opt} is defined as the Δ_{opt} between the effort-based cumulative lift charts of the optimal model and the devised prediction model. Similarly, Δ_k is defined as the Δ_k between our technique and the one by Kamei et al. (2013). Larger values of P_{opt} and P_k indicate smaller differences between the compared techniques. Such values are normalized in the range [0,1] to ease their interpretation (Kamei et al., 2013).

3.7. Threats to validity

The results of our study may be affected by a number of threats.

Threats to construct validity. As for factors threatening the relation between theory and observation, in our context, these are mainly concerned with the measurements we performed. Above all, we rely on the results of the SZZ algorithm (Śliwerski et al., 2005) to answer our research questions. Although the intrinsic imprecisions of SZZ (da Costa et al., 2016) still represent a threat for the validity of our results, it is the most effective algorithm available in literature.

To compute the CEXP, REXP, SEXP metrics, we mined commits to count the number of modifications applied by a developer in different time windows. However, it might be possible that the actual author of a commit is not the same person as the committer. That may be especially true in large projects where sometimes developers (*e.g.,*newcomers) can modify the source code but do not have rights to perform a push onto the repository. This potential problem might have influenced the way the metrics are computed and used within the devised prediction model. To verify the extent to which this represents an actual issue for our analyses, we quantified in how many cases there was a mismatch between author and committer in the analyzed commits. Specifically, for each commit of the considered projects, we ran the command `git show --format=full`[5] to obtain the full set of information available for the commit. That includes data on both author and committer email addresses. Thus, we could compute the number of times in which the two email addresses differ, *i.e.,*in how many cases the author of a change was not the actual committer. Out of the 160,515 total commits considered in our study, we found 4173 mismatches, meaning that we are not accurate in only 2.6% of the cases. Based on this result, we can argue that such mismatches represent corner cases rather than systematic problems that threats our analysis. To further verify the impact of this potential threat, we completely re-ran our study excluding those 4173 commits. However, we did not observe any difference for the results achieved when including the commits. That indicates that mismatches between authors and committers do not influence our findings. A complete overview of this additional analysis is available in our online appendix (Pascarella et al., 2018).

Threats to conclusion validity. Although the metrics used to evaluate the performance of the *fine-grained just-in-time* defect prediction model, (*i.e.,*precision, recall, F-measure, and AUC-ROC), are widely used in the field (D'Ambros et al., 2012), future studies can be conducted to validate our model from a different angle, *e.g.,*by evaluating its industrial impact.

A possible threat concerning the results achieved in $\mathbf{RQ_1}$ is related to the co-presence of production and test files within a commit, which may lead to a over-estimation of the number of partially defective commits. We conducted an additional analysis to assess the effect of excluding test files on our findings. We could not find differences with respect to the results reported in the original submission (a complete report of this additional analysis is available in our online appendix Pascarella et al., 2018). These results are in line with recent work: Even test code may be defective (Vahabzadeh et al., 2015) and test files have the same proneness of production files to be affected by functional issues (Spadini et al., 2018a). It seems reasonable to keep test files in our analysis/approach to maintain developers' awareness also on bugs in tests.

Another threat regards how we assess the cost-effectiveness of the models experimented. As done in previous research (Canfora et al., 2015; Panichella et al., 2016; Arisholm et al., 2010; Kamei et al., 2010; Mende and Koschke, 2009; 2010), we measure the inspection cost in terms of lines of code to be inspected by a reviewer. LOC has been evaluated as a valid proxy measure (Arisholm et al., 2010) since it is correlated with code and cognitive complexity (Nagappan and Ball, 2005). However, this is an approximation. Function points (FPs) (Matson et al., 1994) represent an alternative that we do not consider in this study, since it requires setting parameters that only original developers/managers or expert effort estimation consultants might properly set and a third-party analysis done by the authors

[5] https://git-scm.com/docs/git-show.

Table 3
Results for RQ1 on partially defective commits.

Systems	Ratio		
	Partially defective commits	Defective files	Avg. files per commit
Accumulo	46%	44%	4.1
Angular-js	51%	38%	2.2
Bugzilla	47%	37%	5.4
Gerrit	38%	43%	3.3
Gimp	44%	45%	4.3
Hadoop	49%	38%	3.1
JDeodorant	39%	47%	3.4
Jetty	53%	46%	3.8
JRuby	45%	42%	3.5
OpenJPA	40%	37%	4.0
Overall	**43%**	**42%**	**3.7**

of this paper would introduce noise/bias. Future work can be designed and conducted to investigate how much size approximates defect inspection effort.

In the context of $\mathbf{RQ_2}$, we adopt a time-sensitive validation strategy where a single commit c_i represents the test set and the data of the previous three months form the training set. We select this strategy because this is the most similar to a real-case scenario where developers use the devised approach as soon as a new commit is performed, for example in a code review. While other researchers adopted slight variations of this strategy (*e.g.,*Tan et al., 2015 used a gap between training and test sets to add in the training set defective commits that were discovered and fixed), we preferred it for its stronger ecological validity.

We statistically compare the differences between our model and the standard *just-in-time* model proposed by Kamei et al. (2013). We do not perform statistical tests with the Bonferroni correction (Abdi, 2007): This is a conscious decision taken on the basis of the findings by Perneger (1998), who explained why such a correction is unnecessary and deleterious for sound statistical inference. Finally, we assess the model for the presence of multicollinearity (O'Brien, 2007), relying on *Random Forest*, which can automatically remove non-relevant features.

As a final note, we compare our model with the one proposed by Kamei et al. (2013) in the context of $\mathbf{RQ_4}$ (the cost-effectiveness analysis) but not in $\mathbf{RQ_2}$ (the accuracy analysis). On the one hand, the model by Kamei et al. (2013) targets a different problem (*i.e.,*detecting defective commits rather than defective files within commits), thus it cannot be fairly compared with the proposed model in terms of accuracy. This statement is supported by experimental data, which showed that the model by Kamei et al. (2013) achieved an overall F-Measure of 31% and AUC-ROC of 53% when employed in our context (by considering all the files within an identified defective commit as defective). On the other hand, the comparison performed regarding cost-effectiveness allows us to understand and quantify the gain provided by our approach against state of the art.

Threats to external validity. The main issue concerns the generalizability of the results. To alleviate this issue, we take into account a variety of projects having different characteristics, scope, and size. Nevertheless, future studies must be devised to replicate and extend our investigation on a larger set of systems, possibly taking into consideration industrial projects as well.

4. Results and analysis

In this section, we present the results of the study by research question.

4.1. $\mathbf{RQ_1}$. *What is the ratio of partially defective commits?*

The analysis of the results associated to the first research question aims to understand the prominence of *partially* defective commits, hence the importance of devising a *fine-grained* solution for just-in-time defect prediction. Table 3 reports the results for each considered system: The second column reports the percentage of *partially* defective commits contained in the considered systems, the third column shows the percentage of defective files for each projects (computed using Formula 1), and the fourth column reports the average number of files per commit in the considered systems. The last row ("Overall") represents the average ratio computed taking into account all the projects as a single dataset.

Among all the defective commits investigated we found that 43% of them are *partially* defective, *i.e.,*they contain a mixture of both defective and non-defective files, while 57% of defective commits only contain one resource. Thus, while standard *just-in-time* models can be adopted in most cases, there still exists a consistent part of defective commits for which they cannot provide developers with detailed information.

Investigating the *partially* defective commits more in depth, we found that on overall only 42% of committed files are defective; this is quite surprising, since it implies that less than the half of the elements in a *partially* defective commit is actually defective. Considering the perspective of a developer who has to inspect the files in a change set, she might spend more than half of the time inspecting non-defective resources before finding an actual defect.

For instance, let us consider the commit `a0641ea475` belonging to the ANGULAR.JS project.[6] In this case, the developer committed 9 different files with the aim of making configurable the errors to show in case of wrong usage of the tool. However, there was only one defective file in the whole change set, *i.e.,*the `minErr.js` one. As a consequence, the usage of *coarse-grained just-in-time prediction* model such as the one proposed by Kamei et al. might not provide the adequate support in these cases. The observations made until now still hold when considering the "best" scenario reported in the table, *i.e.,*the one of the JDEODORANT project, where we found that 47% of the resources in a defective commit is affected by a problem, enforcing a developer to inspect many non-defective resources before diagnosing the defect.

With the aim of further understanding the characteristics of defective commits, we also computed the Kendall's τ correlation

[6] https://github.com/angular/angular.js/pull/15881.

Table 4
Results of the RQ2 considering all commits in the history of the subject software systems.

	RQ2				RQ3
Systems	Precision	Recall	F-measure	AUC-ROC	Cost-effectiveness
Accumulo	71%	66%	69%	82%	**16% (L)**
Angular-js	73%	62%	68%	79%	**18% (L)**
Bugzilla	65%	65%	65%	73%	**4% (M)**
Gerrit	69%	62%	65%	72%	**8% (L)**
Gimp	61%	59%	60%	69%	**16% (L)**
Hadoop	67%	58%	63%	73%	**7% (L)**
JDeodorant	75%	61%	68%	74%	**11% (L)**
Jetty	60%	65%	62%	77%	**17% (L)**
JRuby	64%	61%	62%	70%	**21% (L)**
OpenJPA	63%	60%	61%	72%	**7% (M)**
Overall	**67%**	**62%**	**65%**	**76%**	**13% (L)**

Table 5
Results of the RQ2 considering only partially defective commits in the history of the subject software systems.

	RQ2				RQ3
Systems	Precision	Recall	F-measure	AUC-ROC	Cost-effectiveness
Accumulo	76%	69%	73%	85%	**19% (L)**
Angular-js	75%	63%	69%	77%	**23% (L)**
Bugzilla	69%	68%	68%	74%	**4% (M)**
Gerrit	77%	66%	72%	77%	**9% (L)**
Gimp	65%	63%	64%	69%	**17% (L)**
Hadoop	68%	61%	64%	76%	**11% (L)**
JDeodorant	83%	68%	76%	79%	**16% (L)**
Jetty	64%	69%	67%	83%	**20% (L)**
JRuby	66%	63%	65%	73%	**28% (L)**
OpenJPA	66%	64%	65%	71%	**7% (M)**
Overall	**72%**	**65%**	**69%**	**77%**	**16% (L)**

(Kendall, 1938) between the number of files per commit and the number of defective files. This is a non-parametric statistical test used to measure the ordinal association between two measured quantities, with a value ranging between −1 and +1.[7] In our case, the correlation between number of files per commit and number of defective files turned to be equals to 0.42, thus indicating a positive concordance between the two variables. This confirms previous findings reporting that the more resources a developer changes the higher the chances to introduce defects (Hindle et al., 2008).

In conclusion, the results show the need of fine-grained techniques to reduce the number of resources to inspect in a defective commit.

Result 1. 42% of defective commits in our subjects are *partially* defective, *i.e.,*composed of both files that are changed without introducing defects and files that are changed introducing defects. Further, in almost 43% of the changed files a defect is introduced, while the remaining files are defect-free.

4.2. ***RQ$_2$***. *To what extent can the model predict defect-inducing changes at file-level?*

To answer our second research question we evaluate the effectiveness of the prediction model described in Section 3.4 based on a machine learning algorithm built using the *Random Forest* classifier. For sake of clarity, we report the results of both RQ$_2$ and RQ$_3$ in three separated tables that have a similar structure. The columns "RQ$_2$" report the evaluation metrics, *i.e.,precision, recall, F-measure*, and *AUC-ROC*, for each system. Table 4 is obtained evaluating our model considering indiscriminately all commits in the history of the projects, instead Table 5 considers only *partially* defective commits and Table 6 represents only *fully-defective* commits.

Looking at the full-inclusive results of Table 4, we observe that the precision ranges between 60% and 75% (overall = 67%), the recall between 58% and 66% (overall = 62%), while the overall F-measure is equal to 65%. Interesting are the results in terms of precision: in a context where the recommendations are given when developers are committing their changes on the repository, having a tool able to pinpoint the files that are likely defective can avoid the introduction of a consistent number of defects in a system. Assuming that developers can recognize a defect should they get a true positive warning from our model, the adoption of our model has the potential to be useful in practice, since its precision is higher than 60% in most of the cases. The recall values tell us that our model locates more than half of the defects actually present in the subject systems.

Considering also the AUC-ROC we observe that the model obtains levels between 69% and 82% (overall = 76%). The worst case observed in our dataset regards the Gimp project, where our model achieves the lowest F-measure (60%). Investigating the likely causes behind this result, we found that our model was not able to work on the other projects because of the peculiar characteristics of the *C* programming language used. In particular, 39% of the change sets were composed of interacting files (*e.g.,*a file *C* including functions from other files): consistent modifications to files including several external files were often performed, while minor defective changes were performed on the other files. As a consequence, the metrics computed (*e.g.,*the normalized number of lines of code added) were not effective.

[7] (i) −1 represents a perfect negative linear relationship, (ii) +1 a perfect positive linear relationship, and (iii) the values in between indicate the degree of linear dependence between the two measured quantities.

Table 6
Results of the RQ2 considering only fully defective commits in the history of the subject software systems.

	RQ2				RQ3
Systems	Precision	Recall	F-measure	AUC-ROC	Cost-effectiveness
Accumulo	65%	63%	64%	72%	**7% (M)**
Angular-js	71%	61%	66%	80%	**13% (L)**
Bugzilla	61%	62%	61%	71%	**3% (S)**
Gerrit	62%	58%	60%	66%	**6% (M)**
Gimp	57%	58%	57%	68%	**13% (L)**
Hadoop	66%	56%	60%	64%	**6% (M)**
JDeodorant	74%	59%	67%	73%	**9% (L)**
Jetty	61%	64%	62%	71%	**11% (L)**
JRuby	62%	59%	61%	69%	**18% (L)**
OpenJPA	60%	60%	60%	67%	**6% (M)**
Overall	**63%**	**61%**	**62%**	**70%**	**10% (L)**

We do not observe large decays between the overall metric values and the highest/lowest ones (*i.e.,*the difference is always within 21%). This means that the *fine-grained just-in-time* model is consistent across the projects.

When analyzing the results obtained by the models built only considering *partially* and *fully* defective commits (Tables 5 and 6), we observe that the former outperforms the latter by 7% in terms of both F-measure and AUC-ROC. Since the goal of this paper is to assess the extent to which a prediction model can identify defective files within commits, we consider the performance of the technique built on *partially* defective commits as encouraging because the proposed approach is actually able to meet the intended goal. At the same time, we consider the performance degradation noticed on *fully* defective commits as reasonable. Unfortunately, we are not able to speculate on the specific reasons causing such degradation. Likely, the addition of further independent variables able to characterize the defectiveness of commits as a whole (*e.g.,*the metrics devised by Kamei et al., 2013) can be beneficial to improve the performance of the model further. A future research effort can be devoted to the potential combination between just-in-time and fine-grained just-in-time models. In any case, our results show that in the majority of the cases the model can provide further recommendations also when considering fully-defective commits. Finally, the model including all commits inherits pros and cons observed in the cases of the models built on *partially* and *fully* defective commits only. In other words, it can predict *partially* defective commits better than *fully* defective ones, having higher performance on the former and lower on the latter; that shifts performance in the middle to the individual models.

Result 2. The proposed model achieves an overall *AUC-ROC* of 76% and obtains stable performance across the considered projects.

4.3. ***RQ₃***. *What are the features of the devised model that the most to its performance?*

Table 7 reports the results achieved when applying the *Gain Ratio Feature Evaluation* algorithm (Quinlan, 1986) to understand which are the most relevant features that allow the model to identify defect-inducing changes within the files of a commit. In particular, for each variable we report (i) the expected entropy reduction it gives to the model and (ii) the shape of the relationship with the dependent variable, *i.e.,*whether the feature contributes more to the prediction of defective or non-defective files.

Four key factors give the highest contribution to the performance of the model, *i.e.,*experience of the committer, lines of code added, neighbor's commit count, and recent experience of the committed. All of them have an expected entropy reduction higher than 0.6, which means that they are highly relevant to discriminate defective files. The number of modified directories is also pretty relevant, with an entropy reduction of 0.58, while the result of the changed code scattering (entropy reduction = 0.55) confirms that non-focused modifications tend to have an adverse effect on source code quality (di Nucci et al., 2017). Other factors, even though lower in ranking, can characterize defective files within a commit, such as entropy or owner's contributed lines. The remaining independent variables tend to be less/poorly related to defective files (*e.g.,*the average interval between the last and the current change).

Table 7
Gain provided by each feature to the prediction model.

Variable name	Expected entropy reduction	Shape
CEXP	0.76	Defective
LA	0.71	Defective
NCOMM	0.68	Non-defective
REXP	0.64	Defective
ND	0.58	Non-defective
SCTR	0.55	Defective
Entropy	0.49	Non-defective
OWN	0.48	Non-defective
SEXP	0.43	Defective
LD	0.33	Non-defective
MINOR	0.31	Defective
DEL	0.28	Non-defective
ADD	0.21	Defective
COMM	0.19	Defective
NSCTR	0.17	Defective
DDEV	0.12	Defective
NDDEV	0.10	Non-defective
OEXP	0.09	Non-defective
EXP	0.09	Defective
ADEV	0.06	Non-defective
NADEV	0.04	Non-defective
NUC	0.04	Non-defective
LT	0.03	Defective
AGE	0.02	Defective

Looking more in depth into the results, we observe that the model relies on different types of information to discriminate defective and non-defective files. For example, experience-related metrics have a positive relationship with the dependent variable, meaning that they mostly help in detecting defective files. It seems reasonable to think that this is due to a substantial difference in the behavior of developers expert and non-expert of a particular piece of code, which the model can correctly interpret for the identification of defect-inducing changes. Similarly, the amount of lines added mainly indicates the defectiveness of an artifact. On the other hand, several factors contribute more to the prediction of non-defective files. Among them, the NCOMM feature, which represents the neighbor's commit count, has a strong impact on the predictions made by the model on non-defective files: this

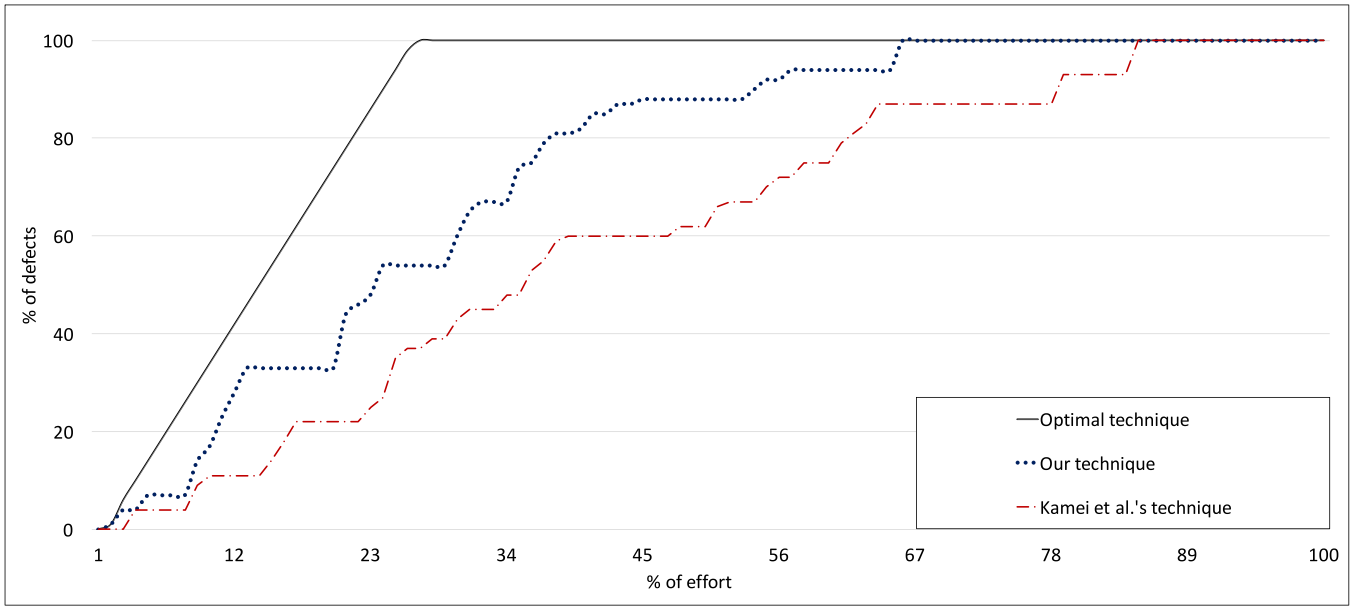

Fig. 2. Results achieved for **RQ$_4$**.

indicates that the number of changes applied to files connected to a specific file can particularly characterize the lack of defects.

To sum up, this analysis let emerge that defective commits are strongly explained by developer-related factors (thus corroborating the need for methodologies and tools for an efficient allocation of resources) and that developers should perform small changes. Our results are in partial agreement with the findings by Kamei et al. (2013): indeed, only the experience of the committer is a powerful predictor in both traditional and fine-grained just-in-time defect prediction. Instead, when comparing our results with those of Rahman and Devanbu (2013), we confirm that process metrics are generally better predictors than product ones. More in general, we observe that no single family of metrics (*i.e.,*product, process, or developer-related) provides the best predictors; this is in line with recent findings reporting the importance of exploiting a combination of metrics to improve the performance of prediction models (Catolino et al., 2018; di Nucci et al., 2017).

Result 3. Developer-related factors are those that generally provide the highest contribution to the prediction of defective files within commits. Similarly, also the amount of lines added influences the prediction.

4.4. ***RQ$_4$***. *How much effort can be saved using a fine-grained just-in-time defect prediction model with respect to a standard just-in-time model?*

This analysis is intended to provide evidence on the effort developers can save using our model to guide the inspection of commits for defects. We consider the state of the art *just-in-time* model and the optimal results for comparison. Fig. 2 plots the effort-based cumulative lift charts of the experimented techniques. The devised *fine-grained just-in-time* solution presents a larger curve with respect to the technique of Kamei et al. (2013): this confirms the ability of our model to work better than the baseline, and thus it can optimize the effort required by a developer to locate actual defects. For instance, our results show that 54% of all defects can be identified by investigating an effort of 24% in terms of lines of code to inspect. With the same budget, only 27% of them can be found by relying on the technique by Kamei et al. (2013). This observation is confirmed when considering the P_k metric, that is equal to 0.43. Thus, the devised technique seems to represents a more viable solution for predicting defects at commit-level. A clear example can be found in the JRuby project and is represented by the pull request #4371[8] where two reviewers needed to inspect four committed files. In one of these, *i.e.,*the `mx_jruby.py` file composed of 171 lines of code, a defect was identified. On this commit, our model correctly marked the file as defective. At the same time, the model by Kamei et al. also correctly pointed out the defectiveness of the commit. However, while the effectiveness of the models is the same their code review cost is largely different: using the proposed model a reviewer should have focused on the `mx_jruby.py` file only, while using the baseline one she should have potentially investigated all the four files, leading to the analysis of a total of 1947 LOCs (*i.e.,*+1776 LOCs).

If we consider the differences between our technique and the optimal model, the P_{opt} is 0.31. This means that, as expected, the optimal model outperforms ours. Nonetheless, we can also see that the difference is closer when considering a reduced effort budget, *i.e.,*in cases where developers have limited time to dedicate to defect fixing activities. For example, let consider a hypothetical limited budget of 10%: in this case, using our technique, it is possible to identify 23% of the defective files, while with the optimal technique 35%. This indicates that, at least in the first phases, our model can be considered as a valid solution to speed up the identification of defects. At the same time, we argue that more research on the topic would be needed, as there is still room for further improvements.

When considering the individual projects, we observe that the Bugzilla and OpenJPA projects follow a different trend. Further analyzing these cases, we found that the limited improvement with respect to the baseline was due to the characteristics of the commits in that repositories. Even though in **RQ$_1$** we found that average commit defectiveness ratio of the systems was 37% in both the cases, often the commits on these repositories contain few resources, *i.e.,*the average number of resources per commit was 2.3 and 3.4 for the two systems, respectively. This aspect limited the difference in the inspection costs achieved by the experimented models since in cases of defective commits composed of few

[8] https://github.com/jruby/jruby/pull/4371 (associated with the commit `9c921e`).

resources the lines of code to inspect with the two models is similar. Nevertheless, also in these systems, the *fine-grained* solution outperforms the baseline: this result indicates that the proposed model may be useful also on systems which follow restrictive commit policies or whose developers tend to commit fewer changes (*e.g.,*to avoid the introduction of tangled changes (Herzig et al., 2016)).

Finally, it is worth remarking that the results achieved on the entire set of defective commits were also confirmed when considering partially and fully defective commits independently.

Result 4. Considering an effort-budget of 24%, 54% of the defects can be identified by our technique. In comparison with the state of the art, we observe that the devised technique represents a more viable solution to locate defects at commit-time. Our model improves upon the baseline also with a few files per commit.

5. Conclusion

Many defect prediction models have been proposed to locate defect-prone files or commits exploiting *long-term* or *short-term* techniques, respectively. Nevertheless, such models suffer from limitations due to the coarse-grained granularity of the predictions performed, which hinder their practical applicability (*e.g.,*in code review). For this reason, we investigated the possibility to devise a fine-grained *just-in-time* defect prediction model to locate defective files contained in a commit. The study considered 10 open-source systems written in different programming languages and having different size and scope. In total we analyzed 160,515 commits of which 35,496 defective.

The main contributions made by this paper are:

1. An empirical validation aimed at understanding the prominence of partially defective commits, *i.e.,*commits containing both defective and non-defective files on a set of 10 different open source software projects. The results highlight that almost half of defective commits contain both defect-inducing and defect-free changes.
2. A fine-grained *just-in-time* defect prediction model and its empirical evaluation, which showed performance up to 82% in terms of AUC-ROC.
3. An assessment of the *cost-effectiveness* of our model and its comparison with the standard *just-in-time* model proposed by Kamei et al. (2013), with evidence that our model is more cost-effective.
4. An online appendix (Pascarella et al., 2018) that reports all the additional analyses mentioned in the paper.

Based on the results, our future agenda includes the replication of our study on a larger set of systems, possibly performing an in-depth study in an industrial context. At the same time, future studies can be designed and conducted to investigate (i) the role of other independent variables, *e.g.,*those reported by McIntosh et al. (2014), on the performance of fine-grained defect prediction, (ii) the model in the context of cross-project defect prediction, and (iii) the benefits provided by the usage of personalized defect prediction (Jiang et al., 2013; Xia et al., 2016) as well as more sophisticated ensemble techniques (Yang et al., 2017). Moreover, we plan to evaluate the extent to which standard just-in-time approaches working at commit-level can be combined with the fine-grained solution we proposed, *e.g.,*through a multi-stage classification process where the defective commits are identified first and then the specific defective files are detected. Furthermore, the effectiveness of our model should be evaluated in-field, through a controlled study with practitioners to incorporate in our model some of the guidelines suggested by Lewis et al. (2013) to make defect prediction more actionable in practice and support human activities (*e.g.,*by introducing a graphical user interface supporting code reviewers when diagnosing defect-prone code components).

References

Abdi, H., 2007. Bonferroni and Šidák corrections for multiple comparisons. In: Encyclopedia of Measurement and Statistics, 3, pp. 103–107.

Arisholm, E., Briand, L.C., Johannessen, E.B., 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J. Syst. Software 83 (1), 2–17.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Proceedings - International Conference on Software Engineering, pp. 712–721. doi:10.1109/ICSE.2013.6606617.

Baeza-Yates, R., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley.

Barnett, J.G., Gathuru, C.K., Soldano, L.S., McIntosh, S., 2016. 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, New York, NY, USA, pp. 496–499. doi:10.1145/2901739.2903496.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Software Eng. 22 (10), 751–761.

Baum, T., Schneider, K., Bacchelli, A., 2017. On the optimal order of reading source code changes for review. In: 33rd International Conference on Software Maintenance and Evolution, pp. 329–340.

Bell, R., Ostrand, T., Weyuker, E., 2013. The limited impact of individual developer data on software defect prediction. Empir. Software Eng. 18 (3), 478–505. doi:10.1007/s10664-011-9178-4.

Canfora, G., Lucia, A.D., Penta, M.D., Oliveto, R., Panichella, A., Panichella, S., 2015. Defect prediction as a multiobjective optimization problem. Softw. Test. Verif. Reliab. 25 (4), 426–459. doi:10.1002/stvr.1570.

Cass, S., Bulusu, P., 2018. Interactive: the top programming languages 2018. IEEE Spectr.. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., Zaidman, A., 2018. Enhancing change prediction models using developer-related factors. J. Syst. Software 143, 14–28.

le Cessie, S., van Houwelingen, J., 1992. Ridge estimators in logistic regression. Appl. Stat. 41 (1), 191–201.

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

da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A., 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans. Software Eng. PP (99). doi:10.1109/TSE.2016.2616306. 1–1.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Software Eng. 17 (4–5), 531–577.

Devijver, P.A., Kittler, J., 1982. Pattern Recognition: A Statistical Approach.

Di Nucci, D., Palomba, F., Oliveto, R., De Lucia, A., 2017. Dynamic selection of classifiers in bug prediction: an adaptive method. IEEE Trans. Emerg. Topics Comput. Intell. 1 (3), 202–212.

El Emam, K., Melo, W., Machado, J.C., 2001. The prediction of faulty classes using object-oriented design metrics. J. Syst. Software 56 (1), 63–75.

Eyolfson, J., Tan, L., Lam, P., 2011. Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, pp. 153–162.

Freund, Y., Mason, L., 1999. The alternating decision tree learning algorithm. In: Proceedings of the Sixteenth International Conference on Machine Learning. Morgan Kaufmann Publishers Inc., pp. 124–133.

Friedman, J., Hastie, T., Tibshirani, R., 2001. The Elements of Statistical Learning. Springer Series in Statistics, 1. Springer, Berlin.

Grabmeier, J.L., Lambe, L.A., 2007. Decision trees for binary classification variables grow equally with the gini impurity measure and pearson's chi-square test. Int. J. Bus. Intell. Data Min. 2 (2), 213–226.

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Software Eng. 26 (7), 653–661.

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

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Software Eng. 38 (6), 1276–1304.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 78–88. doi:10.1109/ICSE.2009.5070510.

Herraiz, I., Gonzalez-Barahona, J.M., Robles, G., 2007. Towards a theoretical model for software growth. Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on. IEEE. 21–21

Herzig, K., Just, S., Zeller, A., 2016. The impact of tangled code changes on defect prediction models. Empir. Software Eng. 21 (2), 303–336. doi:10.1007/s10664-015-9376-6.

Hindle, A., German, D.M., Holt, R., 2008. What do large commits tell us?: A taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. ACM, pp. 99–108.

Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 279–289.
Jiang, Y., Cukic, B., Menzies, T., 2008. Can data transformation help in the detection of fault-prone modules? In: Proceedings of the 2008 Workshop on Defects in Large Software Systems. ACM, pp. 16–20.
John, G.H., Langley, P., 1995. Estimating continuous distributions in bayesian classifiers. In: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., pp. 338–345.
Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E., 2016. Studying just-in-time defect prediction using cross-project models. Empir. Software Eng. 21 (5), 2072–2106.
Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., Hassan, A.E., 2010. Revisiting common bug prediction findings using effort-aware models. In: Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, pp. 1–10.
Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Software Eng. 39 (6), 757–773.
Kendall, M.G., 1938. A new measure of rank correlation. Biometrika 30 (1/2), 81–93.
Khomh, F., Penta, M.D., Guéhéneuc, Y.-G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empir. Software Eng. 17 (3), 243–275. doi:10.1007/s10664-011-9171-y.
Kim, S., Whitehead Jr, E.J., Zhang, Y., 2008. Classifying software changes: clean or buggy? IEEE Trans. Software Eng. 34 (2), 181–196.
Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W., 2015. Investigating code review quality: do people and participation matter? In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, pp. 111–120.
Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. Proc. IEEE 68 (9), 1060–1076. doi:10.1109/PROC.1980.11805.
Leszak, M., Perry, D.E., Stoll, D., 2002. Classification and evaluation of defects in a project retrospective. J. Syst. Software 61 (3), 173–187.
Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., Whitehead Jr., E.J., 2013. Does bug prediction support human developers? Findings from a Google case study. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 372–381.
Liaw, A., Wiener, M., 2002. Classification and regression by randomforest. R News 2 (3), 18–22.
Madeyski, L., Kawalerowicz, M., 2017. Continuous defect prediction: the idea and a related dataset. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, pp. 515–518.
Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. 27 (C), 504–518. doi:10.1016/j.asoc.2014.11.023.
Matson, J.E., Barrett, B.E., Mellichamp, J.M., 1994. Software development cost estimation using function points. IEEE Trans. Software Eng. 20 (4), 275–287.
McCabe, T.J., 1976. A complexity measure. IEEE Trans. Software Eng. (4) 308–320.
McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2014. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 192–201.
Mende, T., Koschke, R., 2009. Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering. ACM, p. 7.
Mende, T., Koschke, R., 2010. Effort-aware defect prediction models. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on. IEEE, pp. 107–116.
Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., Turhan, B., Zimmermann, T., 2013. IEEE Trans. Software Eng. 39 (6), 822–834. doi:10.1109/TSE.2012.83.
Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.
Moonen, L., 2001. Generating robust parsers using island grammars. In: Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. IEEE, pp. 13–22.
Moser, R., Pedrycz, W., Succi, G., 2008. 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, pp. 181–190.
Munson, J.C., Elbaum, S.G., 1998. Code churn: a measure for estimating the impact of code change. In: Software Maintenance, 1998. Proceedings., International Conference on. IEEE, pp. 24–31.
Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE, pp. 284–292.
Nucci, D.D., Palomba, F., Rosa, G.D., Bavota, G., Oliveto, R., Lucia, A.D., 2017. A developer centered bug prediction model. IEEE Trans. Software Eng. PP (99). doi:10.1109/TSE.2017.2659747. 1–1.
O'Brien, R., 2007. A caution regarding rules of thumb for variance inflation factors. Qual. Quant. 41 (5), 673.
Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2005. Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng. 31 (4), 340–355.
Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Software Eng. 23 (3), 1188–1221.
Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A., 2017. The scent of a smell: an extensive comparison between textual and structural smells. IEEE Trans. Software Eng..
Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2016. Smells like teen spirit: improving bug prediction performance using the intensity of code smells. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016). Raleigh, USA: IEEE.
Panichella, A., Alexandru, C.V., Panichella, S., Bacchelli, A., Gall, H.C., 2016. A search-based training algorithm for cost-aware defect prediction. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. ACM, New York, NY, USA, pp. 1077–1084. doi:10.1145/2908812.2908938.
Parnas, D.L., Lawford, M., 2003. The role of inspection in software quality assurance. IEEE Trans. Software Eng. 29 (8), 674–676. doi:10.1109/TSE.2003.1223642.
Pascarella, L., Palomba, F., Bacchelli, A., 2018. Fine-Grained Just-in-Time Defect Prediction: Online Appendix. Technical Report.
Perneger, T.V., 1998. What's wrong with Bonferroni adjustments. BMJ 316 (7139), 1236.
Posnett, D., D'Souza, R., Devanbu, P., Filkov, V., 2013. Dual ecological measures of focus in software development. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 452–461.
Quinlan, J.R., 1986. Induction of decision trees. Mach. Learn. 1 (1), 81–106. doi:10.1023/A:1022643204877.
Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 432–441.
Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P., 2011. Bugcache for inspections: hit or miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 322–331. doi:10.1145/2025113.2025157.
Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P., 2016. On the "naturalness" of buggy code. In: Proceedings of the 38th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 428–439. doi:10.1145/2884781.2884848.
Robnik-Sikonja, M., 2004. Improving random forests. In: ECML, 3201, pp. 359–370.
Sammut, C., Webb, G.I. (Eds.), 2010. Leave-One-Out Cross-Validation. Springer US, Boston, MA. pp. 600–601.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: ACM SIGSOFT Software Engineering Notes, 30. ACM, pp. 1–5.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? SIGSOFT Software Eng. Notes 30 (4), 1–5. doi:10.1145/1082983.1083147.
Sommerville, I., 2006. Software Engineering: (Update) (8th Edition) (International Computer Science). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A., 2018a. On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 1–12.
Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A., 2018b. On the relation of test smells to software code quality. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE.
Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2. IEEE Press, pp. 99–108.
Tufano, M., Bavota, G., Poshyvanyk, D., Di Penta, M., Oliveto, R., De Lucia, A., 2017. An empirical study on developer-related factors characterizing fix-inducing commits. J. Software: Evol. Process 29 (1), e1797–n/a. doi:10.1002/smr.1797. E1797 JSME-15-0185.R2
Vahabzadeh, A., Fard, A.M., Mesbah, A., 2015. An empirical study of bugs in test code. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, pp. 101–110.
Van Der Malsburg, C., 1986. Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Springer Berlin Heidelberg, pp. 245–248.
Williams, C., Spacco, J., 2008. Szz revisited: verifying when changes induce fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems. ACM, New York, NY, USA, pp. 32–36. doi:10.1145/1390817.1390826.
Wu, R., Zhang, H., Kim, S., Cheung, S.-C., 2011. Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 15–25. doi:10.1145/2025113.2025120.
Xia, X., Lo, D., Wang, X., Yang, X., 2016. Collective personalized change classification with multiobjective search. IEEE Trans. Reliab. 65 (4), 1810–1829.
Yang, X., Lo, D., Xia, X., Sun, J., 2017. Tlel: a two-layer ensemble learning approach for just-in-time defect prediction. Inf. Software Technol. 87, 206–220.
Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on. IEEE, pp. 17–26.
Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. 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, New York, NY, USA, pp. 157–168. doi:10.1145/2950290.2950353.

Luca Pascarella My research interests remark the European "Software Engineering in Enterprise Cloud Applications Homepage" (SENECA) project goals. This is a research project which I belong to. My research consists in a dynamically and continuous investigation aimed at understanding how structured and unstructured data can be manipulated to improve development processes of software projects. Currently, I am investigating corner aspects that involve Modern Code Reviews (MCR), such as how developers use and produce code comments. My goal is to understand how unexplored aspects of MCR may affect the development process with the purpose to address issues and propose considerable solutions for industrial companies. This research is supported by collaborating with several universities all over the world and in particular with an important Dutch company, the Software Improvement Group (SIG).

Fabio Palomba is a Senior Research Associate at the University of Zurich, Switzerland. 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.

Alberto Bacchelli My primary research interest is in Empirical Software Engineering. My focus is on developing and applying data science processes and techniques to conduct empirical experiments on complex knowledge-based activities, particularly the development of software, and improve their state of the practice. My goal is understanding and improving complex knowledge- based activities, such as software engineering, with processes, practices, and tools informed by data and actors' needs, and by engaging with other disciplines such as social sciences. I tackle research questions with both qualitative and quantitative research approaches.