The Journal of Systems & Software 181 (2021) 111038

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Investigating the performance of personalized models for software defect prediction[☆]

Beyza Eken [*], Ayse Tosun

Computer Engineering Department, Faculty of Computer and Informatics Engineering, İstanbul Technical University, 34469, İstanbul, Turkey

ARTICLE INFO

Article history:
Received 1 October 2020
Received in revised form 22 April 2021
Accepted 30 June 2021
Available online 13 July 2021

Keywords:
Personalized
Change-level
Defect prediction
Software recommendation systems

ABSTRACT

Software defect predictors exploring developer perspective reveal that code changes made by separate developers tend to have different defect patterns. Personalized defect prediction also contributes to this view and gives promising results. We aim to investigate the performance of personalized defect predictors compared to those of traditional models. We conduct an empirical study on six open-source projects for 222 developers. Personalized and traditional defect predictors are built utilizing two algorithms and cross-validation on the historical commit data, and assessed via seven performance measures and statistical tests. Our results show that personalized models (PMs) achieve an increase of up to 24% in recall for 83% of developers, while causing higher false alarm rates for 77% of developers. PMs are better for those developers who contribute to the modules with many prior contributors. Although size metrics contribute to the performance of the majority of the PMs, they significantly differ in terms of information gained from experience, diffusion and history metrics, respectively. The decision of whether a PM should be chosen over a traditional model depends on a set of factors, i.e., selected algorithm, model validation strategy or performance measures, and hence, PM performance significantly differs regarding these factors.

1. Introduction

Studies on software defect prediction (SDP) propose automated tools that mine historical development data from version control systems, source code bases and issue management systems and identify development patterns to recommend defect-prone modules in software systems. For more than two decades in empirical software engineering, researchers are building SDP models using different datasets, different input features, and different algorithms to catch more bugs compared to manual code review and other bug detection activities performed by the development teams (Moser et al., 2008; Malhotra, 2015). The recommendations given by SDP models to assist developers are discovered to improve the quality of the produced software (Robillard et al., 2014; Hall et al., 2011).

Software systems produced by different developers tend to have distinguishing defect patterns (Ostrand et al., 2010) since each developer has unique characteristics regarding their development styles, experience, and defect proneness of the code they produce. Developers' profiles and their way of working can be quantified through a set of features, including but not limited to years of experience, age, gender, collaboration with other developers, commit activity, quality of the code produced, and bug resolution activity. Some of these features have previously been used as developer metrics in SDP studies. For example, the number of developers that changed a code module (Weyuker et al., 2008) is included in the prediction models to incorporate the developer aspect into defect prediction. In another study, developer focus is quantified and included into the defect prediction models (Di Nucci et al., 2017). Gender is also studied in bug prediction to assess the effect of gender on introducing smells (Catolino et al., 2019).

Recommendation systems targeting other businesses, e.g., e-commerce websites, search engines, and social media platforms utilize users' features such as search history or locations to give customized and more useful search results or advertisements (Blog, 2009; Tucker, 2014). The software engineering field also adopts the techniques used in recommendation systems to improve the process of software systems (Avazpour et al., 2014), e.g., a recommendation system to decide which source code modules need to be modified (Gasparic and Janes, 2016). Recently, instead of quantifying developers' characteristics as a set of metrics, personalized defect predictors have been built separately for each developer by utilizing the corresponding developer's change history only (Eken, 2018; Jiang et al., 2013; Xia et al., 2016). While general defect prediction models utilize the whole change history produced by the team, a personalized

[☆] Editor: Shane McIntosh.
[*] Corresponding author.
E-mail addresses: beyzaeken@itu.edu.tr (B. Eken), tosunay@itu.edu.tr (A. Tosun).

https://doi.org/10.1016/j.jss.2021.111038
0164-1212/© 2021 Elsevier Inc. All rights reserved.

model utilizes less data during training compared to a traditional model. Moreover, predictions made by these personalized SDP models target the relevant developer only. By making developer-specific predictions, researchers aim to improve the usability and accuracy of defect prediction tools.

The personalized SDP models proposed so far (e.g., Jiang et al., 2013; Xia et al., 2016) give promising results in terms of defect detection rates. These models lead to investigate fewer files modified by a selected developer only, and still catch more defects than the general models. The conclusions drawn from these early personalized models are hard to generalize due to certain assumptions made in their methodology. In particular, the personalized model findings are limited in terms of reported performance measures, the used metrics, and model construction strategies. First, the proposed models in the literature could reach up to 7% increase in F1-measure and 8% increase in recall, however, the effect sizes of these performance values and individual personalized models' performance were not discussed in detail. Second, well-known metrics that characterize bugs in SDP, e.g., lines of code, code changes, or the number of developers, were not used while building personalized models, but the source codes were transformed into characteristic word vectors. Third, developer selection and data collection strategies are ad-hoc, i.e., 100 commits from 10 developers were used for model building. Considering that SDP models still lack generalizability in different contexts, the results suffer from external validity of their prediction performance (D'Ambros et al., 2012). We believe the need for further investigation on the benefits and the general applicability of personalization in SDP.

In this study, we aim to investigate the performance of personalized SDP models compared to those of traditional approaches in an empirical setup on six open-source software projects: Gimp, Maven-2, Perl, PostgreSQL, Rails, Rhino (Misirli et al., 2016). We utilize change-level defect prediction datasets of those projects, each has 9 to 26 years of development history and embody the state-of-the-art process metrics (Mockus and Weiss, 2000; Graves et al., 2000; Matsumoto et al., 2010; Nagappan and Ball, 2005; D'Ambros et al., 2010; Nagappan et al., 2006). Our study extends and contributes to the previous work on personalized defect predictors in the following aspects:

1. We investigate a total of 222 developers over six large-scale open source projects, and build personalized models for each of these selected developers. In RQ1 (Section 3), we analyze the performance of personalized models against the traditional approaches, namely a general model (GM) and a general model for a selected set of developers (SM). The developer selection strategy that we follow is based on previous discussions on the size of training data in SDP (Turhan et al., 2009), and data availability in recommendation systems (Schein et al., 2002).
2. To increase the conclusion validity of our empirical analyses, we evaluate our proposed models with respect to seven performance measures, namely probability of detection, probability of false alarm, precision, F1-measure, Area Under ROC Curve, Matthews Correlation Coefficient, and Brier score. We also utilize additional assessment techniques, such as effect size calculation and Nemenyi's test, for model comparisons. Our findings are based on statistically significant differences with medium to large effect size only.
3. We conduct an empirical analysis on the development characteristics of individual developers with respect to (1) the number of commits made by the developers, (2) the ratio of bug-inducing commits to the total commits of the developers, and (3) the process metric values. We would like to understand whether these characteristics reflect when personalized models are superior over traditional models, or vice versa (RQ2 in Section 3).
4. We conduct a feature analysis using Information Gain on the process metrics used to predict defects in a personalized model, and assess the distinguishing features among personalized models (RQ3 in Section 3).

Empirical analyses on 222 developers of six open-source projects show that personalized defect prediction models improve the traditional models' probability of detection rates up to 24% for 83% of the analyzed developers. Even though overall results show the superiority of the personalized approach, personalized models do not improve the prediction performance for every developer per se. We observe that the developers whose traditional model outperforms the personalized model have a higher experience (i.e., more contribution as commits) on their projects. On the other hand, the personalized models are more successful than the traditional models for those developers who contribute to the files that were modified by many developers. Also, the importance rank of the process metrics differs among 222 developers, except the fact that the size metrics (i.e., added lines of code) are the most important ones for the majority of developers. Finally, the performance of the personalized model highly depends on the selected machine learning algorithms and performance assessment criteria.

Structure of the paper. Section 2 reports the related work on personalized defect prediction literature. In Section 3, we report the experimentation conducted in this study in detail. Section 4 discusses the results of the experimentation conducted to answer our research questions, while Section 5 discusses the outcome of our experimentation from various aspects (e.g., machine learning algorithms applied, model validation techniques). In Section 6 threats to the validity of our findings are discussed. Section 7 summarizes the key take-away messages and reports future research directions.

2. Related work

In this section, we report several change-level defect prediction approaches over the history of SDP models. In the subsections below, we explain the evolution of SDP models with respect to using people related metrics, and discuss prior studies that propose personalized SDP models.

SDP models are designed to make predictions at different granularity levels, e.g., file, class, method, and code change level (Hall et al., 2011). A change-level SDP model makes it easier to find the responsible developer for a bug-prone software entity (Kamei et al., 2012). Majority of the change-level SDP studies in the literature utilize developer experience, history of the changed files, diffusion of code changes, and size of changes (Mockus and Weiss, 2000; Shihab et al., 2012; Kamei et al., 2012; Misirli et al., 2016). Mockus and Weiss (2000) propose models that predict the high risky (being prone to failure) initial maintenance requests (IMR) that contain multiple software changes. Their analysis on a large scale telecommunication system shows that the number of subsystems modified, and developer experience are found to be indicators of these high risky changes. Similarly, Shihab et al. (2012) also use change-level software metrics, such as the time when the change is made, the purpose of change (bug fix or not) in addition to size, history, and experience metrics to predict risky changes. They show that the developer experience, number of added lines, bug-proneness of the modified files, number of bugs, and the number of bug reports linked to a change are found to be good indicators of a change's risk-proneness.

Śliwerski et al. (2005) propose a method called SZZ to locate fix-inducing changes of software development by using the software archives and bug reporting systems. The SZZ algorithm contributes to the SDP field by providing a new data source to researchers and practitioners of the field: bug-inducing changes (Kamei and Shihab, 2016). Later, Kim et al. (2008) classify the code changes as bug-inducing or clean. Their prediction model utilizes textual features of the modified file and directory names, and change metadata information (i.e., commit hour, commit day), complexity metrics of the modified files in addition to the size metrics. Kamei et al. (2012) utilize diffusion, size, history, purpose, and developer experience metrics to predict bug-inducing changes of six open-source and five commercial software projects. Their empirical assessment demonstrates a reduction in the code review effort (i.e., 35% of all predicted bug-inducing changes could be identified by spending 20% of the total effort) using the prediction model and a recall of 64%.

Researchers have so far approached the SDP problem from different perspectives, i.e., from modeling the algorithm and validation techniques to enriching the data, and to modeling the people. Various statistical and machine learning techniques are utilized (Hall et al., 2011; Malhotra, 2015), i.e., Logistic Regression (LR), Naive Bayes (NB), and Random Forest (RF) algorithms are widely used and performed quite well on predicting defects. Some studies focus on assessing the experimental setup to depict the data more accurately, i.e., investigating other model validation techniques (Tantithamthavorn et al., 2017), algorithm tuning (Fu et al., 2016), data pre-processing approaches (Jiang et al., 2008), enriching the data by adding new metrics, and new class types (Tsakiltsidis et al., 2016; Misirli et al., 2016). Recently, SDP models that make predictions at a fine-granularity level are proposed, such as predicting bug-prone files (Pascarella et al., 2019) or code lines (Yan et al., 2020; Pornprasit and Tantithamthavorn, 2021) in software changes.

More recent studies consider the effect of developers on defect proneness of software modules (Schröter et al., 2006; Weyuker et al., 2008; Ostrand et al., 2010; Posnett et al., 2013; Lee et al., 2016). As people are the third essential unit in software development, in addition to product and process, researchers investigate software developers to model their development behavior, interactions among each other through the modified source code modules, and other communication channels. Besides, new studies are putting the whole focus on the developers, and propose personalized SDP models that aim to give customized prediction results to each developer in a team (Jiang et al., 2013; Xia et al., 2016). Customized feedback is provided by separately-built models for each developer instead of including developer metrics into general SDP models. Below, we report studies that use developer metrics to build defect prediction models in Section 2.1, and build personalized defect prediction models in Section 2.2.

2.1. Defect prediction using developer metrics

Schröter et al. (2006) report differences in the bug density of source code files developed by different developers. According to their study, specific developers more likely to generate bugs than others, and this reflects the complexity of the code rather than a competency between developers. Later works study the relationship between the defect density of code modules and the *experience* of the developer. The more experienced developers develop more complex code because more risky, larger and thus more complex tasks are assigned to more experienced developers (Zeller, 2009; Tufano et al., 2017). Further, the results of Eyolfson et al.'s study (Eyolfson et al., 2011) show that more experienced developers tend to introduce fewer bugs. Rahman and Devanbu (Rahman and Devanbu, 2011) report that there is no clear correlation between the bugginess of the code and the developers' overall experience on the project. On the other hand, their study states that a developer's experience on one file is more important than the developer's overall experience on the project.

The *number of developers* who worked on a software module is popularly used as one of the metrics in SDP models (Weyuker et al., 2008; Ostrand et al., 2010; Bell et al., 2013; Matsumoto et al., 2010). A study by Ostrand et al. (2010) reports that including the number of developers who modified a code module into the predictor makes a modest improvement in the performance of defect prediction. Matsumoto et al. (2010) indicate that developers' defect injection rates vary, and modules edited by many developers contain more defects.

Posnett et al. (2013) proposed *developer focus* to refer to the activity of a developer. The authors measured the focus of a specific developer to a specific module, and concluded that more focused developers would cause fewer defects. Di Nucci et al. (2017) extended the *focus* metrics by adding a distance measure between the modules. The smaller the distance between the modules means that the modules are more related to each other. The proposed model outperforms their baselines including the study of Posnett et al. (2013).

Lee et al. (2016) proposed a metric set that models the developers' *interaction behavior at a micro-level*. Micro interaction metrics are collected from Mylyn, such as browsing and editing times of code files, frequencies and edited file information. They concluded that micro interaction metrics improved the prediction performance when they were used with source code and change history metrics. Also reported in Lee et al. (2016), the number of developers that edited a file in the history, the number of edit events observed for a file, and the number of selections of a file are relatively good predictors.

Calikli and Bener (2015) also modeled developers in terms of their *confirmation bias* levels to identify their relationships with the post-release defects. For instance, testers may exhibit confirmatory behavior in the form of a tendency to make the code run rather than employing a strategic approach to make it fail.

2.2. Personalized defect prediction

Bettenburg et al. (2012) claim that software engineering data contains a large amount of variability and models built with a global context are often irrelevant and less successful than models built with a local context. They applied Multivariate Adaptive Regression Splines (MARS) (Friedman, 1991) which is a global model that considers the local regions in the data. Their results show that the MARS approach outperforms the traditional global approach in defect prediction.

We identified two studies proposing personalized SDP models (Jiang et al., 2013; Xia et al., 2016). These personalized models are built at change-level rather than file/method level since the ownership can be defined easier on a code change than a file. A file may be modified by several developers, and hence, a defect in a file may be the consequence of these code changes, whereas a code change is only associated with a single developer (Kamei et al., 2012; Misirli et al., 2016).

Jiang et al. (2013) propose change-level personalized SDP models for six different open-source projects, namely Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. They identified 10 developers who contributed the most to these projects. Then, 100 consecutive commits of each developer were collected and used for building the personalized prediction models separately, while a total of 1000 commits of all 10 developers are used for building the traditional, general prediction model. In addition to the personalized and general SDP models, a weighted model which combines different developers' data, and a meta-classifier which ensembles the predictions of general and personalized approaches are built. They employed Alternating Decision Tree (ADTree), Naive Bayes (NB) and Logistic Regression (LR) algorithms to build their models. Jiang et al. (2013) built their models utilizing characteristic vectors that represent the syntactic structure of the changed source codes in a commit, and bag-of-words for both commit messages and source code. Moreover, commit hour and day, cumulative change and bug-inducing change counts, source code file/path names, and file age were the other input metrics. Their results indicate that the F1-measure of the personalized model was 1 to 6% more than the traditional (general) model among six projects. Besides, when the top 20% of defective lines of code are inspected, a personalized approach could detect up to 155 more defects than a traditional approach.

Xia et al. (2016) propose an alternative personalized approach that utilizes other developers' commit history, as other developers' commit history could be useful to predict the defects of a specific developer. That idea is similar to the weighted personalized approach in Jiang et al. (2013), but instead of randomly selecting the other developers' data to build a training set, other developers' change data was included in the training set utilizing a genetic algorithm. The authors in Xia et al. (2016) experimented on the same datasets used by Jiang et al. (2013) and filtered the same number of developers (10) and the same number of commits (100 each). They also used the same metric set used in Jiang et al.'s study (Jiang et al., 2013). The results reveal that their proposed personalized solution could reach up to 13% higher F1-measure than the traditional models and detect up to 245 more defects than the previously built models in Jiang et al. (2013).

Early personalized approaches for SDP (Jiang et al., 2013; Xia et al., 2016) report that personalized defect predictors could reach up to 13% higher F1-measure rates than the general models. On the other hand, the recall values reported in Jiang et al. (2013) are not very high, between 39% and 74%, compared to the previously reported defect predictors (Hall et al., 2011), while the precision values reflect that false alarm rates may also be higher. Their data selection and developer selection techniques are strict, i.e., only 10 developers and 100 commits from those developers were chosen to build the personalized models. Furthermore, the general models built in Jiang et al. (2013), Xia et al. (2016) only utilized the selected developers' commits whereas the non-selected developers' development history are not included in the general SDP models. To further understand the intrinsic characteristic of personalized approaches we need to observe the results of personalized approaches under different circumstances. Thus, in this study, we aim to investigate the performance of personalized defect predictors with a different experimental setup regarding datasets, metrics, model construction and additional performance measures.

3. Experimental setup

In this study, our objectives are to build *personalized* defect predictors using individual code changes of developers, to assess the models' performance against the traditional models, and to understand the factors that may have an effect on the performance of personalized predictors. We would like to evaluate the usefulness of the personalized approach by setting up an improved empirical setup that conforms to the state-of-the-art change-level SDP. We believe that the findings of this study would contribute to software practitioners while deciding whether the personalized approach in the context of SDP can be worth building. We define three research questions (RQs) as follows:

RQ1 How does a personalized SDP approach perform compared to traditional SDP approaches?

RQ2 To what extent do development characteristics have an effect on the superiority of PM?

RQ3 How does the importance of metrics used for defect prediction differ among personalized models?

Please note that for RQ2, we define development characteristics over commit activities of developers: (1) number of commits of a developer, (2) ratio of bug-inducing commits to the total number of commits of a developer, (3) metric (Table 1) values of the commits of a developer, and (4) the importance rank of the metrics for a developer. More details on the methodology of each of our research questions are reported in Section 3.2.

3.1. Dataset details

We conducted our research on six datasets containing historical commit information of six open-source projects, namely Gimp, Maven-2, Perl, PostgreSQL, Rails and Rhino. This dataset is collected in a prior study whose steps are described in Misirli et al. (2016). The dataset contains five types of metrics at commit (change) level; the size, history, experience, diffusion and purpose related metrics, and bug-inducing commits were extracted from the commit histories of the six projects. The full list of metrics is given in Table 1. The *Size* metrics represent the amount of change during a commit. The *History* metrics represent the number of developers that changed the modified files in a commit, the change history of the modified files in a commit and the average time interval between a commit and the last change time of modified files in the commit. The *Experience* metrics are calculated based on prior commits of the developer who made the commit. The higher the number of previous commits of the developer, the higher the value of experience metrics. The *Diffusion* metrics represent how a change is spread across source files. These metrics are calculated by counting the number of modified files, subsystems, directories in a commit and calculating the entropy of each commit. The *Purpose* dimension involves a single metric named as FIX that represents whether the purpose of a commit is bug-fixing or not (binary). The detailed explanations of the metrics can also be found in the previous work (Misirli et al., 2016). We think the selected projects represent rich information in terms of their development periods (e.g., from 1987 till 2013 for Perl) and the extracted metrics. The dataset contains 13 well-known process metrics which measure the commits through various aspects. Previous studies also report that the process metrics extracted from the software projects contain rich and useful information for defect prediction models, and these perform better than code metrics (Misirli et al., 2016; Kamei et al., 2012; D'Ambros et al., 2012). Therefore, we chose this dataset to make a better comparison of our approach with the prior studies in terms of prediction performance as well as to increase the replicability of our methodology on publicly available projects. Building a new dataset was not the focus of our study. Instead, we selected an existing dataset that has already been validated in the context of change-level SDP studies.

The date range of the commits, the number of total and bug-inducing commits, and the number of total developers for all the projects are given in Table 2. Besides, it also reports our developer selection statistics which we mention in the next section.

3.2. Our methodology

We share the same initial goal with Jiang et al. (2013) of building personalized defect predictors, but our methodology differs in many aspects: Selecting the developers to be modeled for personalized prediction, the data sampling approach, the algorithms applied, and the performance assessment methods. We

Table 1
Software metrics used in this study.

Metric	Description
Size	Lines of code added in a commit (ADD) Lines of code deleted in a commit (DEL)
History	Number of developers had changed the modified files (NDEV) Number of prior changes to modified files (NPC) The average time interval between the last and the current change (AGE)
Experience	Experience of the developer (EXP) Recent experience of the developer (REXP) Experience of the developer on a subsystem (SEXP)
Diffusion	Number of modified files (NF) Number of modified subsystems (NS) Number of modified directories (ND) Entropy: distribution of modified code across each file (ENT)
Purpose	Is the purpose of a commit to fix a bug? (FIX)

Table 2
Details of the dataset.

	Gimp	Maven-2	Perl	PostgreSQL	Rails	Rhino
Time period of commits in years	1997–2013	2003–2012	1987–2013	1996–2013	2004–2013	1999–2013
Total number of commits	32.875	5.399	50.485	35.005	32.866	2.955
Total number of bug-inducing commits	11.940	551	12.172	13.511	6.224	1.291
Total number of developers	499	33	1116	38	2287	35
Statistics regarding developer selection						
Number of selected developers	51	5	87	27	45	7
Total number of commits of all selected developers	28.286	4.411	45.876	34.848	22.592	2.801
Total number of bug-inducing commits of all selected developers	11.057	514	10.954	13.491	5.223	1.254
Range of commits of selected developers'	45–9.222	127–2.296	47–7.850	47–12.755	53–3.314	77–1.095
Range of bug-inducing commit ratio of selected developers' (between 0 and 1)	0.08–0.8	0.05–0.2	0.05–0.65	0.13–0.8	0.02–0.53	0.17–0.62

focus on an empirical evaluation of personalized models with traditional models on different open-source projects as the prior study, but also we analyzed whether the importance of metrics used for prediction differs across developers and whether the development characteristics (i.e., developer experience) affect the performance of the personalized approach.

All of these aspects are explained below. Fig. 1 illustrates the steps of model building.

3.2.1. *Developer selection*

We pick a specific number of developers from each project to build their corresponding personalized defect prediction models. Our aim is to select as many developers as possible while keeping a sufficient number of total and bug-inducing commits that belong to each developer for building specialized defect predictors. Indeed, a good amount of data is needed to build a successful predictor (Raudys et al., 1991). Although we cannot make a generalized comment on the number of data instances used during the training of a machine learning model, a prior study (Turhan et al., 2009) in our field report that using 100 data instances would be enough to learn for an adequate defect predictor.

Cold-start problem is also another common problem in personal recommendation systems, and it occurs when there is not available knowledge or data to make a recommendation (Aggarwal et al., 2016; Avazpour et al., 2014), i.e., making a recommendation for a not yet rated movie or a new user in a movie recommendation system (Schein et al., 2002). In our context, we face the cold-start problem in situations such as when (1) a developer might have not any prior commit, i.e., she has joined to the software team and/or just started to contribute to the software project, (2) a developer might have not any enough prior commits and/or bug-inducing commits to build a personalized model for her even though she is an active contributor to the project (i.e., 49% of all developers over six projects have only one commit, and a developer from Rails has 87 commits but only two of them are bug-inducing).

Considering all these, we select the developers among all the contributing developers whose total number of commits and bug-inducing commits are at least 45 and 10, respectively. When we apply an under-sampling (Section 3.2.3) on training data with 10-fold cross-validation, the number of total commits for a developer in his/her training set would be at least 18 (9 of 10 bug-inducing commits and 9 of 35 clean commits). This number can be quite low to train a machine learner, so we also check the prediction performance of personalized models for those developers with very few data instances. We further discuss this in Section 6.

We ended up having a total number of 222 developers over six projects. Since the developer selection criterion is applied to each project separately, the selected number of developers differs among the projects, e.g., 87 in Perl and 5 in Maven-2. Still, we cover majority of the commits i.e., 87% of the total commits over six projects.

Table 2 reports the dataset details regarding developer selection process: (i) the number of selected developers for each project, (ii) total number of commits made by all the selected developers, (iii) total number of bug-inducing commits made by all the selected developers, (iv) range of the number of commits made by each selected developer, (v) range of the ratio of bug-inducing commits to all commits of each selected developer.

The reported numbers confirm that the Pareto principle is valid in software projects (Yamashita et al., 2015): the majority of the commits are made by the minority of the developers. The percentage of the selected developers over all developers ranges between 2% to 71%, whereas the percentage of the number of commits of those selected developers over all commits ranges between 70% and 99% for six projects. 49% of all developers over six projects contributed to the project with only a single commit. Moreover, the numbers show that the bug-inducing commit ratios over the total commits of a selected developer range between 0.02 and 0.8.

3.2.2. *Model construction*

Three different SDP models are built for the selected 222 developers by using two different machine learning algorithms,

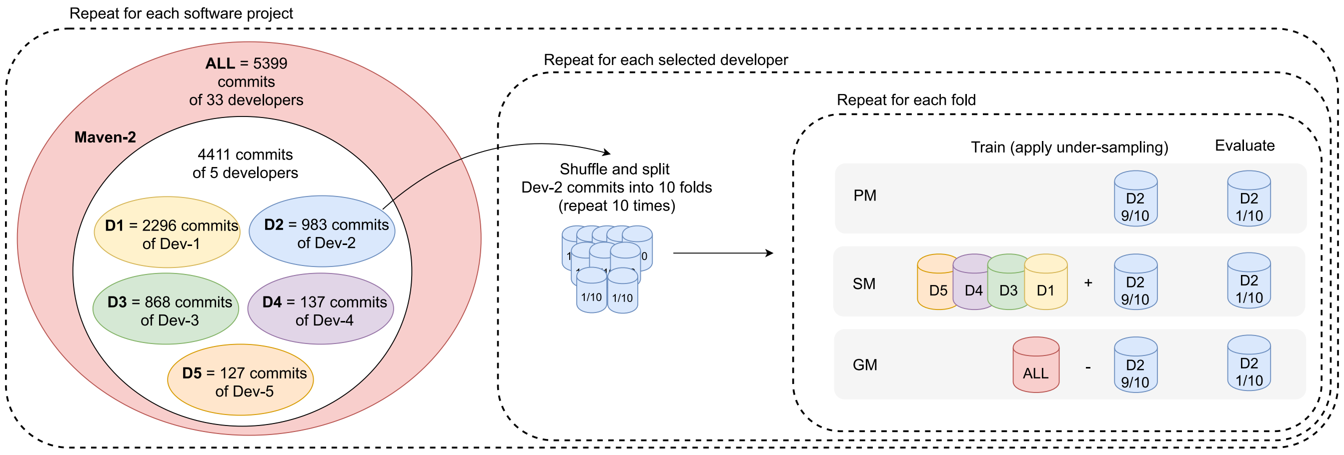
Repeat for each software project
ALL = 5399 commits of 33 developers
Maven-2
4411 commits of 5 developers
D1 = 2296 commits of Dev-1
D2 = 983 commits of Dev-2
D3 = 868 commits of Dev-3
D4 = 137 commits of Dev-4
D5 = 127 commits of Dev-5
Repeat for each selected developer
Shuffle and split Dev-2 commits into 10 folds (repeat 10 times)
1/10
Repeat for each fold
Train (apply under-sampling)
Evaluate
PM
D2 9/10
D2 1/10
SM
D5 D4 D3 D1 + D2 9/10
D2 1/10
GM
ALL - D2 9/10
D2 1/10

Fig. 1. An overview of the model building methodology.

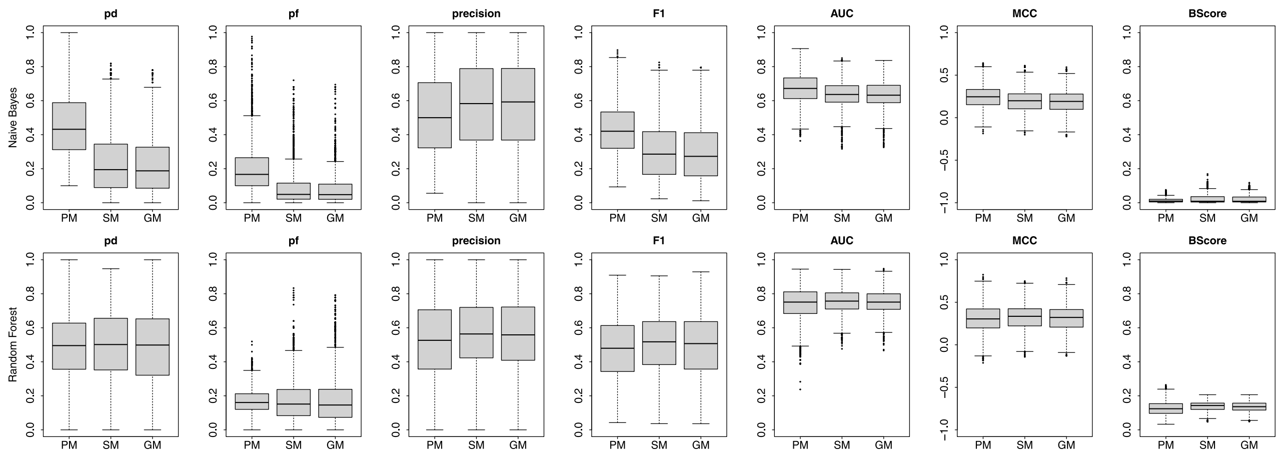

Fig. 2. Performance of PM, SM and GM.

namely Naive Bayes (NB) and Random Forest (RF). These algorithms have been known as popular and well-performing in defect prediction (Lessmann et al., 2008; Hall et al., 2011). So, we utilized these two algorithms using a 10×10-fold cross-validation technique to build our prediction models. Our models are designed to provide predictions at change level.

Personalized model (PM): We propose a personalized defect prediction model for each selected developer in six projects. A personalized model includes only the corresponding developer's commit history to training, and it is intended to make predictions only for the relevant developer at the commit level.

General model (GM): We build a general defect prediction model for each of the six open-source projects. A general model incorporates all commit history of the projects into the training set to build a single model which corresponds to the traditional defect predictors in the literature so far. This general model provides recommendations at the commit level to every contributor in the selected projects' software teams.

General model for a set of developers (SM): We also build additional general models for these projects by narrowing down the whole commit history to the commits of the most active developers instead of utilizing all developers' commit history. This approach is inspired by the general model proposed in the personalized SDP study (Jiang et al., 2013). We include this model into our empirical analyses to compare our findings with the prior study.

We apply 10-fold cross-validation to evaluate PM, SM, and GM models by following prior studies on change-level defect prediction (Kamei et al., 2012; Misirli et al., 2016; Yang et al., 2016; Young et al., 2018; Hoang et al., 2019; Qiao and Wang, 2019), and the prior personalized SDP study (Jiang et al., 2013). Recently, studies discuss the effect of dividing the commit data into 10 folds, without considering the time dependencies between the commits, on the performance of SDP models (Pascarella et al., 2019; Tan et al., 2015). Empirical studies applying different strategies for change-level SDP (e.g., Hoang et al., 2019; Yang et al., 2016), on the other hand, conclude that the findings of the selected strategies are consistent. The performance differences between the strategies are also relatively small. In particular, studies suggest that the conclusions about model performance can change when a different time period is utilized for training (Bangash et al., 2019), or when a time-sensitive validation strategy is chosen (Tan et al., 2015). We are aware of this issue, and hence, we further discuss the effects of applying cross-validation on the findings in Section 5. However, for the sake of comparability with the prior personalized models (e.g., Jiang et al., 2013), we report, in the Results Section, the statistics based on 10-fold cross-validation. Furthermore, our objective in this study is not to generalize our conclusions regarding the performance of PMs or to report the best performance of PMs, but to compare those with traditional models. Hence, we use the same experimental setup for GM, SM and PM models throughout this study. All the experiments are repeated 10 times by shuffling the data before training-test split in order to avoid data order bias (Hall and Munson, 2000).

Table 3
The confusion matrix for the defect prediction problem.

	Actually defected	Actually clean
Predicted as defected	TP (True positive)	FP (False positive)
Predicted as clean	FN (False negative)	TN (True negative)

Please note that the test folds are kept the same across all models' performance evaluation in order to conduct a fair comparison among GM, SM and PM models. During each fold, PM, SM, and GM models share the same test set which corresponds to the commits of a developer associated with the selected fold and the project as illustrated in Fig. 1. On the other hand, the training set varies among the models, i.e., it is filtered based on the criteria of the model. Fig. 1 illustrates the methodology for the three selected developers of Rhino, but the same procedure is applied for the selected developers in every project in this study.

3.2.3. *Under-sampling on training data*

In order to handle the popular problem of imbalanced class distribution in SDP datasets (e.g., Turhan et al., 2009, 2013; Kamei et al., 2012), we apply under-sampling technique on the majority class by randomly selecting the majority class instances until the size of the minority (bug-inducing) and majority class instances is the same. Note that we do not apply sampling on test data. We further discuss its effects by comparing against a no-sampling strategy and another sampling technique in Section 5.

3.2.4. *Evaluation of RQ1*

Performance of the prediction models are evaluated in terms of probability of detection (pd, also called recall), probability of false alarm (pf), precision, F1-measure, area under the ROC curve (AUC), Matthews Correlation Coefficient (MCC) and Brier score. Pd, pf, precision, F1-measure and MCC are calculated from a typical confusion matrix (see Table 3). Explanations of these evaluation metrics and their formulas are given in Table 4. Pd, pf, F1-measure, AUC are well-known measures and reported in defect prediction studies, whereas MCC and Brier score are recently used in empirical software engineering studies (Tantithamthavorn et al., 2018; Palomba et al., 2017). We report and assess the models against all these measures as we should look for a trade-off between these to conclude whether a classifier is accurate and useful in predicting defects.

To answer our RQ1, we compare the performance of PM with those of the traditional models (SM and GM). Pairwise comparisons between the three models are made using the Nemenyi significance test (Nemenyi, 1963), and the model whose performance evaluation metrics are significantly different from the others is identified (according to $p < 0.05$). Effect sizes of the comparisons are measured via Cohen's d using corrected Hedges' g (Kampenes et al., 2007). Measuring effect size is a simple way to quantify the difference between the performance values of the pairs of SDP models under comparison. The $|d| < 0.2$ is interpreted as negligible, $|d| < 0.5$ interpreted as small, $|d| < 0.8$ interpreted as medium, otherwise corresponds to large effect size. Comparisons with negligible and small effect sizes are not considered when we derive our conclusions, since larger data might be needed to claim a strong difference between the models' performance.

3.2.5. *Evaluation of RQ2*

We think that the development characteristics such as the ratio of bug-inducing commits to the total number of commits of a developer, the total number of commits of a developer, the size, history, and diffusion of the developers' changes, and the developers' experience might have an impact on the performance of PM. To understand the effect of these data characteristics on PM performance, we split 222 developers into three groups based on their success on PM:

1. *PM > SM/GM:* The developers whose PM model's performance is significantly better than that of SM or GM or both.
2. *PM < SM/GM:* The developers whose PM model's performance is significantly worse than that of both SM and GM.
3. *PM = SM/GM:* The developers whose PM model's performance is statistically not different from that of SM and GM.

We form these groups according to the Nemenyi pairwise tests conducted on all the performance evaluation metrics (pd, pf, precision, F1, AUC, MCC and BScore) in RQ1, and according to the test results with medium to large effect sizes. Then we compare the development characteristics across the three groups. For each group, the characteristic to be analyzed, e.g., number of commits of a developer, is aggregated over all developers in that group. Then, the aggregated values of the three groups are compared with each other by applying the Mann–Whitney U Test (Conover and Conover, 1980). Later, the effect sizes of comparisons are measured via Cohen's d using corrected Hedges' g (Kampenes et al., 2007).

3.2.6. *Evaluation of RQ3*

Our motivation for RQ3 is to assess if a) PM models utilize a common metric set to predict bug-inducing changes of the corresponding developers, or b) each PM model has its own, unique metric set that provides the highest amount of information to predict bug-inducing changes for the corresponding developer. We applied Information Gain (InfoGain) feature ranking technique (Quinlan, 1986) on the personal commit data of each of the 222 developers in order to analyze the effect of each process metric (Table 1) on the prediction of bug-inducing changes for that developer. The metrics are ranked according to the information provided for bug prediction, and later, the rank values of each metric are compared to each other by using Scott-Knott ESD test to analyze if rank values of each metric are statistically different or the same among the developers.

4. Results

Empirical results conducted on six projects to answer our RQs are reported and discussed in this section.

4.1. *RQ1: How does a personalized SDP approach perform compared to traditional SDP approaches?(PM vs. SM and PM vs. GM)*

In this section, we discuss the findings of PM versus traditional models, namely SM and GM, by considering the algorithms' performance (NB and RF) individually. Later, we discuss the performance values of PM on an individual basis. As explained in our methodology, PM is trained using historical code changes of the selected developer only, whereas GM and SM are trained using all or a subset of the change history of the project respectively. Overall, we observe that the personalized SDP models are better at predicting bug-inducing changes compared to the traditional models. A more detailed discussion is provided below

There are two comparisons, namely PM vs. SM and PM vs. GM, based on the performance achieved with two machine learning algorithms applied (NB, RF), and based on seven performance evaluation metrics (pd, pf, precision, F1-measure, AUC, MCC and Brier Score). The winner model of each comparison according to the Nemenyi test is given in the corresponding cells in Table 5. The "-" sign in the table means that the two models performed statistically the same.

Table 4
The equations and explanations of performance metrics used in this study.

Name(s)	Description	Formula
Probability of detection (pd) Recall	Ratio of correctly predicted defected modules to actually defected modules	$\frac{TP}{TP+FN}$
Probability of false alarm (pf)	Ratio of incorrectly non-defected modules to actually non-defected modules	$\frac{FP}{FP+TN}$
Precision	Ratio of the actually defected modules to predicted as defected	$\frac{TP}{TP+FP}$
F1-measure	Harmonic mean of the precision and recall	$\frac{2*precision*recall}{precision+recall}$
Matthews correlation coefficient (MCC)	How well a defect predictor makes a binary classification as bug-inducing commit or not (Matthews, 1975) (the value of 1 represents perfect classification while −1 represents a completely wrong classification)	$\frac{TP*TN-FP*FN}{\sqrt{(TP+FP)*(TP+FN)*(TN+FP)*(TN+FN)}}$
Brier score	The accuracy of predictions based on the prediction probabilities (Brier, 1950; Rufibach, 2010) (N: number of commits in the validation set, p_i: the probability of the prediction made by the SDP model, a_i: actual outcome of the commit (1 if it is a bug-inducing commit, 0 otherwise))	$\frac{1}{N}\sum_{i=1}^{N}(p_i-a_i)^2$
Area under curve (AUC)	How much the predictor is capable of distinguishing between classifying a commit as bug-inducing and non-bug-inducing	–

Table 5
Win/Loss results of the comparisons between PM and SM, and PM and GM. Bold cells indicate medium to large effect sizes.

	NB		RF	
	PM vs SM	PM vs GM	PM vs SM	PM vs GM
Pd	PM	PM	SM	GM
Pf	SM	GM	SM	GM
Prec.	SM	GM	SM	GM
F1	PM	PM	SM	GM
AUC	PM	PM	SM	GM
MCC	PM	PM	SM	-
Brier	PM	PM	SM	GM

The performance of PM, SM and GM aggregated over six projects are also provided as boxplots in Fig. 2. The results of the models built with the NB algorithm are shown on top of the figure, while the results of the models built with the RF algorithm are given at the bottom of the figure. From left to right, the boxplots correspond to pd, pf, precision, F1-measure, AUC, MCC and Brier Score values.

According to the reported performance of the SDP models built with the NB algorithm in Fig. 2 and the performance comparisons in Table 5, PM outperforms both of the traditional models in terms of pd, F1, AUC, MCC and Brier Score. Although PM increases the pd value, it also increases the pf value for 77% of the developers. The increase in pf also triggers a decrease in precision and hence PM cannot win over the traditional models in terms of pf and precision. The effect sizes of the comparisons between PM and the traditional models are large, around 1.3, 0.9, 1.0 in terms of pd, pf and F1 respectively. For the rest of the evaluation metrics, the effect sizes of the comparisons are small or negligible.

According to the performance with the RF algorithm in Fig. 2 and statistical test comparisons in Table 5, PM under-performs than SM and GM in terms of all model assessment metrics. The measured effect sizes of the comparisons between PM and traditional models, on the other hand, are negligible to small (0.1 to 0.3).

Personalized models in detail: The performance of PM models for 222 selected developers vary between 10 and 100% in terms of pd according to the results reported in Fig. 2. This figure also shows us that while prediction performance of PMs for some developers are very low, others are high in terms of pd. Here, we investigate the PM performance of developers in detail. PM significantly differs from the traditional models in terms of pd, pf, and F1 when NB is utilized. These differences have large effect sizes, and hence here, we report the performance of PM versus SM/GM using NB algorithm and in terms of pd, pf and F1.

Listing the performance results of all of the 222 developers in this paper would take too much space.[1] Thus, we chose nine developers with the best, worst and medium performing PM models, and report their PM, SM and GM model performance in Fig. 3.

Each developer's ID, the project name that the developer contributed are given on the left of the figure.[2] The total number of commits done by the developers and their bug-inducing commit ratio are also given respectively under their aliases. Each PM boxplot in Fig. 3 has a specific color that represents if the PM wins over the traditional models (SM and GM) according to Nemenyi pairwise tests. If PM outperforms at least one of the traditional models, the corresponding boxplot is colored in blue, while it is red if SM and/or GM outperforms PM. If the personalized and traditional models perform the same, the boxplot is colored in black.

Fig. 3 shows that the individual PM performance could reach very high median values in terms of pd, e.g., 85% for Dev-4 and 92% for Dev-51 from Rails. On the other hand, PM models could not detect more than 11% of the defects for some developers, e.g., Dev-13 from Gimp. PM, SM and GM perform the same for the other developers, e.g., Dev-34 from Perl and Dev-4 from PostgreSQL in terms of pd.

Besides, among all the 222 developers, there is not any developer whose PM results are better than the SM and/or GM in terms of all performance measures. While a developer's PM performance is better than at least one of the traditional models in terms of pd, it may not be better in terms of pf. For example, in Fig. 3, PM for Dev-6 from Gimp significantly outperforms the traditional models in pd, but the traditional models outperform

1 The full list is available in https://kovan.itu.edu.tr/index.php/s/cR4dJpn6nL8wvdb.

2 Developer names are replaced with aliases within each project due to privacy.

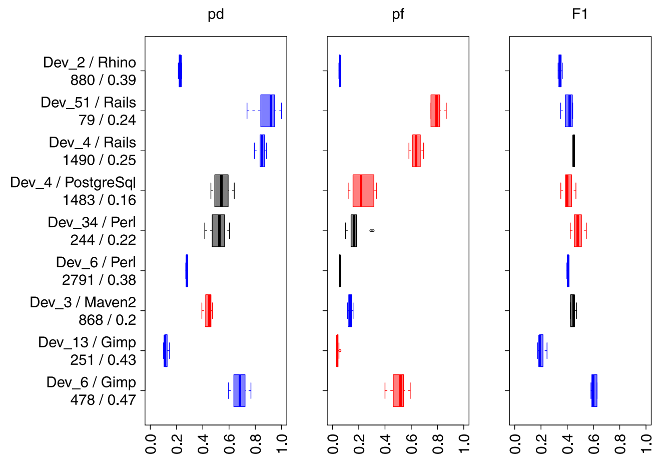

Fig. 3. PM performance of a sample of developers (blue color indicates superiority of PM over traditionals, red color indicates inferiority of PM, whereas black color indicates the equality of models). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 6
Number of developers belong to each group.

	PM >SM/GM	PM <SM/GM	PM = SM/GM
pd	184	17	21
pf	28	172	22
F1	169	17	36

the personalized approach in terms of pf. This trade-off can be observed in all the developers' PM versus SM/GM comparisons. Thus, we believe the decision of the type of SDP model to be used for a developer seems to be highly dependent on the prioritized performance metric.

Building a PM is more convenient for majority of the developers contributed to six open-source projects: PM outperforms the traditional models in terms of predicting defects (pd) and F1-measure, while PM produces higher false alarm rates (pf) than traditional models.

4.2. RQ2: To what extent do development characteristics have an effect on the superiority of pm?

To understand what factors lead to the success of PMs in predicting bug-inducing changes, we would like to analyze development characteristics in detail. As explained in our methodology, we first identify development characteristics in terms of the size and bug-proneness of the development activity, and process metrics used as input features of our model. Later, we formed the groups of developers considering the performance of PM versus SM and GM. The number of developers that belongs to each group is reported in Table 6. We know from our findings in RQ1 that PM outperforms traditional models in terms of pd and F1 when NB is applied, whereas PM is worse in terms of pf. Therefore the groups are formed according to pd, pf and F1 separately. The first group illustrates that there are 184 developers and 169 developers whose PMs are the best when pd and F1 values are compared, respectively. For 172 developers, PMs perform the worst in terms of pf. Once the groups are formed, we analyze the significant differences in terms of development characteristics among these groups.

First of all, we compare the total number of commits of the developers across the three groups. Fig. 4 shows the boxplots

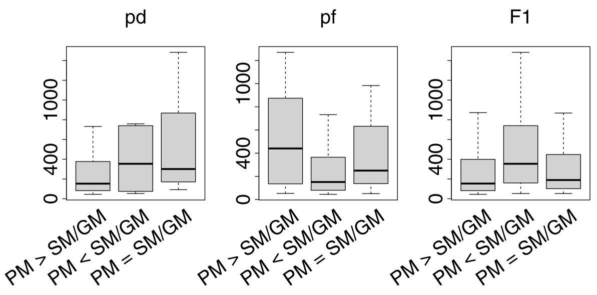

Fig. 4. Number of commits of developers belonging to each group.

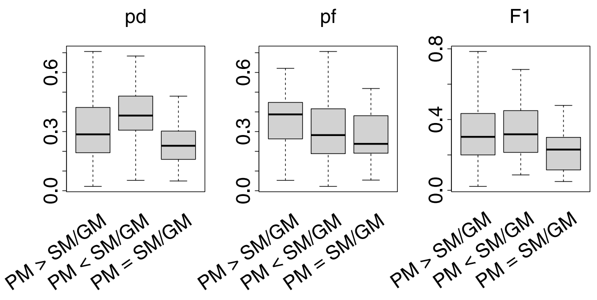

Fig. 5. Bug-inducing commit ratios of developers belong to each group.

of the total number of commits made by the developers associated to the three groups. Pairwise comparisons among the three groups are conducted with Nemenyi tests in terms of pd, pf and F1. The tests show that the total number of commits made by the developers in the second group, where SMs or GMs win over PMs, is significantly different and larger than those of developers in the other groups. The effect size of this finding is medium to large. We observe many developers in the second group, i.e., four developers from PostgreSQL, three developers from Rails, and one developer from Gimp, who are among the top contributors to their project. The number of changes made by those developers are in the range of 53 - 12755 (average is 2060). In such a case, it seems the traditional models perform better than PMs.

Second, we compare the ratio of bug-inducing commits to the total number of commits of developers that belong to each group. Fig. 5 shows the boxplots of the ratios of bug-inducing commits over total commits of developers associated to the three groups. According to the boxplots, the third group of developers, where PM, SM and GM performs statistically the same, has a lower ratio of bug-inducing commits when compared to the other groups of developers with a medium to large effect sizes. The difference among the first and the second group of developers have small and negligible effect sizes. Therefore, we conclude that bug-inducing commit ratio of developers do not seem to have an effect on which model (PM or traditional) performs better.

Third, we investigate if there are differences or similarities among the contributions made by developer groups in terms of 13 process metrics listed in Table 1. We aggregated the changes of developers within each group, and compared the values of each metric calculated from these changes among the groups. Results show that there are differences in five out of 13 metric values calculated from the changes among the three groups: NDEV, NPC, EXP, REXP, and SEXP. In Fig. 6, we only report the values of these five metrics calculated for the three groups.

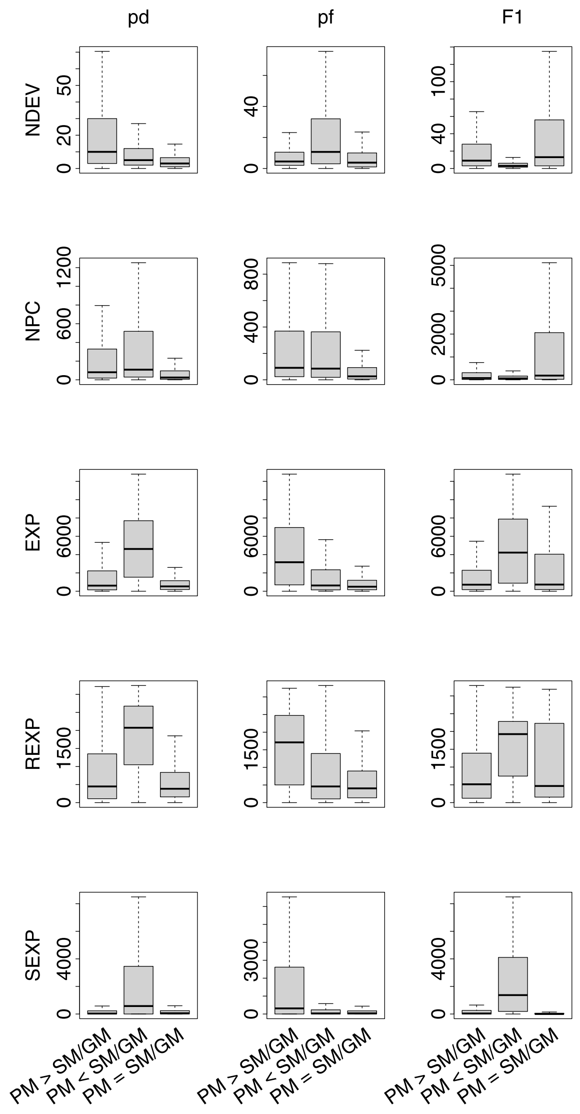

Fig. 6. Metric values of developers belong to each group.

According to Fig. 6, traditional SDP models perform better than PMs for the developers who have higher EXP, REXP, and SEXP values. This finding is consistent with our analysis on the first development characteristics, i.e., number of commits made by a developer. Combining both, we can argue that the traditional models perform better than PMs for those developers who have more experience on the project (contributed as the bulk of commits). On the other hand, PMs is better than the traditional models when the associated developers contribute to the modules which are modified by many developers (higher NDEV). Furthermore, when developers commit to modules which are modified many times (higher NPC) the difference between the PM and traditional models has a larger effect size. However, the winning model might change with respect to the performance measure.

We would like to highlight the fact that the findings regarding the developer group-based analysis also have similar patterns when the analysis setup is extended by including the performance results measured by other model assessment metrics and when the RF algorithm is used.

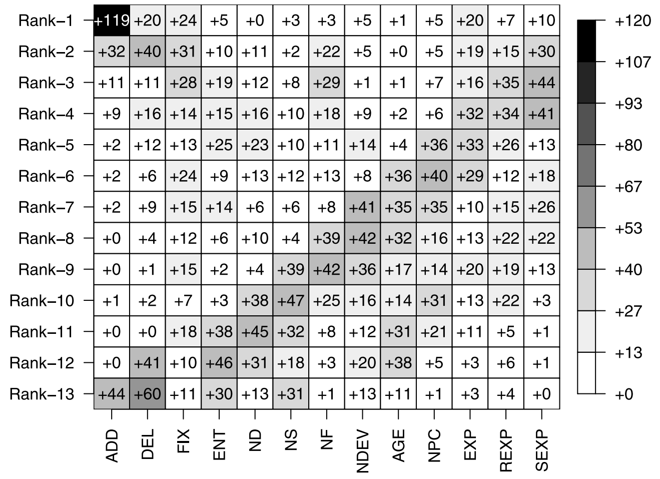

Fig. 7. Ranks of metrics based on InfoGain over each PM.

The development characteristics significantly reflect under which settings PM performs better than SM and/or GM. PM is a more successful approach for predicting defects of the developers that contribute to the modules that have been changed by more developers. When a developer is among the most experienced developers in a project, PM underperforms compared to SM and/or GM.

4.3. *RQ3: How does the importance of metrics used for defect prediction differ among personalized models?*

The heat map in Fig. 7 reports InfoGain results for all 222 selected developers' changes across all six projects. The columns represent the process metrics, whereas the rows represent the ranks one to thirteen. Values in cells represent *"how many times a metric is ranked as the first (or second to thirteenth) during an InfoGain evaluation?"*. The bigger values are represented with darker gray, whereas the smaller values are represented with a lighter gray. Fig. 8 also reports the comparison of metric ranks based on Scott-Knott tests: The metrics that belong to the same group are represented with the same color. The lower the rank of a metric in this figure, the higher its contribution to the prediction model.

Apart from the size metrics, Fig. 7 shows that each PM gains different amount of information from different metrics. When we look at the Scott-Knott ESD clustering in Fig. 8, we observe that the highest ranked metric is the added lines of code in a commit (ADD). ADD is ranked as the first for 119 out of 222 developers. The subsystem experience (SEXP) is the second best metric and followed by the experience (EXP), the purpose (FIX) and the recent experience (REXP) metrics. The diffusion metrics, NF in particular, also appear in higher ranks than the history metrics (NPC, NDEV, and AGE).

Please note that we perform similar analyses on GM and SM, and observe that the *experience* dimension is more important when predicting defects with PM compared to SM and GM. In contrast, the diffusion dimension is more important in predicting defects with SM and GM. Particularly, the mean ranks of SEXP, EXP and REXP metrics are between five and six (Fig. 8), while those metrics are ranked around six to 11 for SM and GM. In terms of the diffusion dimension, ENT, NS, NF, and ND metrics are one to two ranks higher in SM and GM compared to PM. Another important observation is that FIX becomes the last dimension in

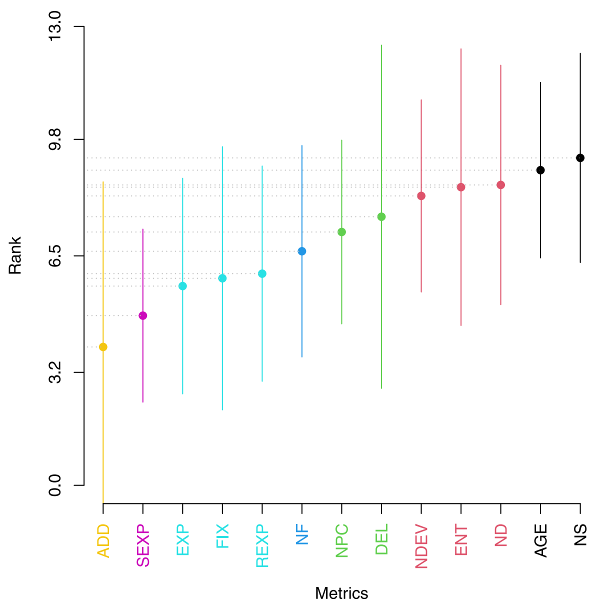

Fig. 8. Statistical comparison of the metric ranks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

SM and GM models, whereas it is ranked as the third important dimension for PM models.

> PMs differ from each other in terms of the amount of information gained from the experience, diffusion and history metric dimensions respectively. According to the majority of PMs, the process metric representing the number of added lines is the most contributing one to the performance.

5. Discussion

Empirical results demonstrate that the selection of which SDP model (i.e. PM or GM) to use in real life may depend on several factors, such as machine learning algorithms, performance metrics and validation strategies. In this section, we discuss how all of these factors might affect the performance of the proposed personalized SDP models, and compare our findings with the prior personalized models in the literature. We also discuss the applicability of the PMs in industrial settings.

5.1. NB versus RF

Figs. 2 given in Section 4.1 indicate that the performance values of an SDP model vary depending on the machine learning algorithms utilized during training. For example, the range of median of pd values across all three models is 0.19–0.43 when models are built with NB, whereas the median pd values obtained with RF algorithm are 0.5 for all three models.

Pairwise statistical comparison between RF and NB for each of the three models points out that RF is better at predicting defects in terms of F1-measure, AUC, MCC, and Brier Score than NB. In terms of pd, PM performs statistically the same when it is built with NB or RF, whereas SM and GM perform better when they are built with RF. Although statistical tests on the models' performance values suggest that RF must be chosen over NB for answering our research questions, the effects of the models' performance differences are not large enough in the case of RF.

Both machine learning algorithms are widely used by the SDP researchers. A recent benchmark on the performance of defect predictors reports that RF may perform better according to one measure, namely H-measure (Li et al., 2019), but the best performing classifier significantly depends on the project. The authors suggest that instead of utilizing a complex learning algorithm like RF, using a simpler one like NB would be more convenient. Based on the literature and our analyses, we also suggest using a simpler algorithm like NB since it reports significant differences with larger effect sizes in the context of PM versus traditional models.

5.2. Recall (pd) versus false alarm rate (pf)

The personalized approach often increases pd rates while it also increases pf rates, according to the performance values given in Fig. 2. It is different than the common pattern observed in data-oriented SDP studies, e.g. using cross-company or cross-project data to predict defects (Turhan et al., 2009). Prior studies show that using other projects' data (global context) increases both recall and pf compared to using a project's data only (local context). But here in our context, using a developer-specific data increases both recall and pf compared to using all developers' commit data. On the other hand, when we look at the false alarm rates of PM model, it has a median around 20%. Having a false alarm rate of 20% is acceptable among many state-of-the-art defect predictors, considering that it achieves a better recall rates. Thus personalized approaches in SDP might be preferable over the traditional approaches as the former gives a better prediction performance.

False alarm rate (pf) of an SDP model is an important measure that should be evaluated according to the context. We believe the false alarm rates should be considered in terms of two perspectives, namely project and people perspective. Earlier studies discuss the project perspective (Hall et al., 2011; Menzies et al., 2007) and state that in a safety-critical or a mission-critical software project, developers would prefer to have a model with high pd rates with the cost of high false alarms. However, a more cost-effective SDP solution would be to have fewer false alarms with the cost of low pd rates. Similarly, the developer's personal choice and/or development methodologies used by the team may affect the choice of having high pf or low pd rates. A developer may dislike the situation of frequent false alarm triggers, or even true positives if she chooses to review her code regularly, and detects the defects by herself instead of following the output of a commit-level defect predictor. In that case, only the bug-inducing commits that have a high probability should be given as recommendations to the developers using traditional models. This would in turn reduce the pd rates but eliminates potential false alarms significantly. On the other hand, a person who wants to explore all potential issues may prefer to use PM with the cost of false alarms.

Please note that, we chose seven performance measures to report the prediction performance of the three types of models in this study. We picked the most commonly used ones, namely pd, pf, precision, F1 and AUC, to make a fair comparison of our models' performance with the related studies. We also report other two measures, namely MCC and Brier Score, that are proposed to avoid biased assessment of F1-measure (Yao and Shepperd, 2020). Unfortunately, the statistical pairwise comparisons among the models with respect to the measures, precision, AUC, MCC and Brier Score report small to negligible effect sizes. Thus, the conclusions derived from those performance measures in particular could be biased due to sample size or other data characteristics. It is our future goal to investigate ways to increase the effect sizes of the significance test outcomes for MCC and Brier Score.

5.3. Data sampling:

During our empirical analysis, we applied random under-sampling on the training data to balance the number of data instances that belong to different classes. We observe that under-sampling significantly improves the prediction performance of PM, SM and GM by 5% in terms of pd compared to a no-sampling strategy when models are built with NB. In addition to under-sampling, we also applied another well-known and successful data balancing technique from the literature, i.e., Synthetic Minority Oversampling Technique (SMOTE) (Chawla et al., 2002). Both SMOTE and under-sampling produce very similarly performing PMs, i.e., a median of 44% in terms of pd. Although SM and GM using SMOTE achieves 13% higher pd values compared to the no-sampling strategy, our test results on the superiority of PM over SM and GM do not change. In our online appendix,[3] we share the prediction performance obtained by applying under-sampling, SMOTE, and no-sampling.

SMOTE does not lead to a better prediction performance for PM when compared to the under-sampling technique. In fact, applying SMOTE takes longer time than applying under-sampling, as the former technique creates synthetic data instances of the minority class using the k-nearest neighbor technique on data instances (Chawla et al., 2002). Under-sampling, on the other hand, simply selects random data instances from the majority class until the sample sizes of both classes are equal. Due to its simplicity and performance, random under-sampling gives us more advantage over SMOTE. We choose the under-sampling as our data sampling technique and report our performance results obtained with under-sampling in the paper.

5.4. Model validation strategy

We conduct our empirical analysis on personalized SDP models using 10-fold cross-validation strategy. We discuss our rationale behind this in Section 3.2. Cross-validation technique has been widely used in SDP literature to evaluate the performance of the change-level defect predictors (Kamei et al., 2012; Misirli et al., 2016; Yang et al., 2016; Young et al., 2018; Hoang et al., 2019; Qiao and Wang, 2019), as well as the personalized defect predictors (Jiang et al., 2013; Xia et al., 2016). A study by Falessi et al. (2018) also reports that the majority of the SDP studies (61%) uses k-fold-cross-validation, while a very few of them (9%) are validated by considering release-based data splitting. Recent studies argue that a time-sensitive approach that preserves the temporal order of commit activities, i.e., learning from past commits to make defect predictions on future commits, would resemble real-life scenarios. Thus, several studies adopt the time-sensitive approach to their training and validation steps of the SDP models (Pascarella et al., 2019; Tan et al., 2015; Hoang et al., 2019; Yang et al., 2016). Two studies compare change-level SDP models by using both time-aware validation and cross-validation strategies (Hoang et al., 2019; Yang et al., 2016). Their results demonstrate that both strategies yield similar performance, and their conclusions are consistent among different settings. The ongoing discussion on the validation methodologies of defect predictors remarks that the conclusions derived from the experimentation should be limited within the experimentation context (Falessi et al., 2018; Bangash et al., 2019).

We believe both cross-validation and time-sensitive validation strategies are required to draw more generalized conclusions on the personalized SDP. The former is important to be consistent with the earlier studies in the field, whereas the latter is important in order to understand the practical applicability of personalized defect predictors. Due to the ongoing discussion on the validation strategy in the SDP field, we would also like to investigate the performance of the personalized models against traditional models (RQ1) in a time-sensitive validation strategy. In this subsection we report the experimental setup for this analysis and its results.

A typical time-sensitive model construction makes a prediction on a commit at time *t*, by learning from the commits before time *t*. Considering that, new train–test data pairs are generated with a sliding window technique. For each developer (d), we stratify all his/her commits into time splits ($d_0, d_1, \ldots, d_n$). Instead of having a fixed length time splits, i.e., six-month, we follow a strategy that produces consistent number of commits in each time split. Stratifying the data with a fixed length time window produces data subsets having no bug-inducing instances. Therefore, we ensure that our splits include at least 10 bug-inducing commits and 10 clean commits. This way, we maximize the number of selected developers and their commits. While one time split (d_n) of a developer becomes a test set, all the commits of that developer prior to that split (d_0 to d_{n-1}) constitute the train set of PM. All commits of *all* developers between the splits d_0 and d_{n-1} constitute of the training set of GM, whereas all commits of the *selected* developers between the splits d_0 and d_{n-1} constitute of the training set of SM. Similar to our setup in Fig. 1, we keep the test set the same among three SDP models (PM, SM, and GM). The developer selection criteria and under-sampling on training are also applied similarly. However, due to the time-sensitive stratification strategy, we can build PMs for 179 out of 222 developers. We report the performance of PM, SM, and GM trained with NB using a time-sensitive validation strategy in Fig. 9.

Findings on the personalized SDP approach are similar among both model validation approaches. We observe that median performance values of models are consistent with those models trained with NB in Fig. 2. The superiority of PM over traditional models is also observed when predictors are evaluated in a time-sensitive validation setup. According to pd and F1, PM significantly outperforms the traditional models, but PM also produces higher false alarm rates, and hence both traditional models significantly outperform PM in terms of pf and precision. The statistical comparisons between PM and traditional models has large effect sizes in terms of pd (1.2), pf (0.8) and F1 (0.9), and small effect sizes for precision (0.2).

It might be possible to reach different conclusions on PM by changing the time-sensitive model validation strategy, i.e., stratifying data with fixed length time intervals and/or changing the training set size. Therefore, our results are open to discussion. Besides the model validation strategy, development characteristics of projects would also affect the conclusions on PM. For instance, in our experimentation, the contributors of a project and their contribution amount to the project changes over time. Fluctuation of the developers involved in the training of general models slightly reduces the prediction performance of general models. Although our results support the applicability of personalized defect prediction over the traditional approach in real life, further analysis and discussion are necessary. We consider applying other time-sensitive data stratification strategies as future research directions for assessing the success of PM in real-life settings.

5.5. Further insights on PM combined with the prior studies

As we report in Section 2.2 there are two studies on personalized SDP models (Jiang et al., 2013; Xia et al., 2016). Xia et al. (2016) complements the work by Jiang et al. (2013) by following

[3] https://kovan.itu.edu.tr/index.php/s/cR4dJpn6nL8wvdb

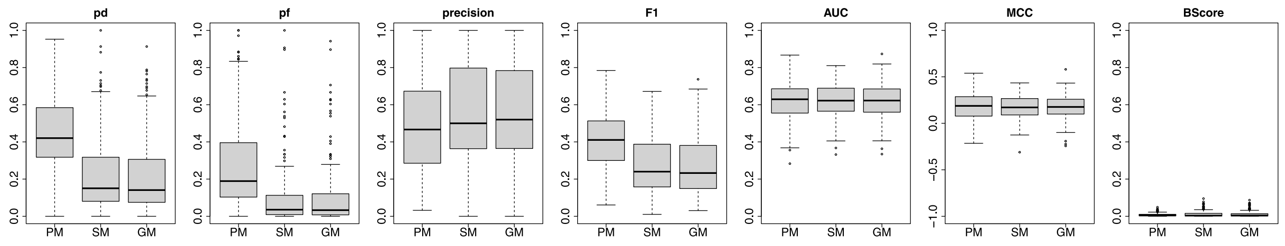

Fig. 9. Performance of PM, SM and GM using a time-sensitive validation strategy.

the same methodology except the algorithm. The improved PM model in Xia et al. (2016) achieves an average of 64% recall and 63% F1-measure over 60 developers. The prior studies and ours follow a completely different experimental design: We utilize different open-source projects except PostgreSQL, different machine learning techniques and process metrics to build personalized our PM and traditional models. Therefore, we cannot report here a one-to-one comparison between the earlier PM models and our proposed PM models. Even though both studies and ours share one open-source project, namely PostgreSQL, the time period of the collected data, the selected developers as well as the sampled data to train the PM models might not be the same. We are, in fact, covering a larger period of commit activity in our work. Our PM models report a median of 40% recall and 40% F1-measure, when NB is utilized, and 50% F1-measure, when RF is utilized, over 222 developers. We have lower PM performance in our context, and we think this might be due to incorporating more commits and developers into our PMs. However, our conclusions have a large effect size (RQ.1), and they support the success of PMs for defect prediction over traditional models. Furthermore we did not restrict the commit size during training set of PM models, which, we have seen, has a major effect on the performance of PM against the traditional models (RQ2). We further provide insights on development characteristics, and discuss under which circumstances PM could reach a better prediction than a traditional model.

Furthermore, prior studies propose collective personalized models to improve the prediction performance of PM and state the superiority the collective models over PM. There are multiple collective models used by the prior studies. One of the methods is leveraging a collective training set in which half of the commits belongs to an individual, while the other half of the commits are taken from other developers' (Jiang et al., 2013). Another method is making an ensemble of the predictions of personalized and traditional models (Jiang et al., 2013). Xia et al. (2016) also use genetic algorithms to create training sets by collecting various amount of commits from various developers.

Although our focus in this study is to assess PM against traditional models (SM and GM), we also set up collective models to validate their success in our context. Similar to Jiang et al. (2013), we built a weighted personalized model (WPM) that utilizes a collective training set that 50% of the training commits belong to a developer, whereas the other 50% belong to other developers. We also built an ensemble model called PM+ that combines predictions of PM, SM, GM and WPM using the majority voting technique. WPM and PM produce very similar performance values and their statistical difference has small or negligible effect sizes. Depending on the algorithm, the superiority of PM+ or PM changes. When models are built with NB, PM outperforms the PM+ in terms of pd, whereas PM+ outperforms PM in terms of pf and Brier Score. On the contrary, when models are built with RF, we observe that PM+ outperform PM in terms of pd and F1 with the cost of higher pf values. These conclusions have medium to large effect sizes only. We conclude that ensemble of PM, SM, GM and WPM models may perform better than PM when they are built with NB, and better than all when they are built with RF. Although the prior studies also support the usage of collective models (especially the ensemble approaches) over the personalized or traditional models, our empirical analysis highlights the importance of the selected performance measures on the final conclusion. The collective models' performance plots are available in our online appendix.[4]

5.6. Effort-aware performance assessment:

Both prior personalized defect prediction studies (Jiang et al., 2013; Xia et al., 2016) and the state-of-the-art change-level defect prediction studies (Kamei et al., 2012; Qiao and Wang, 2019; Yang et al., 2016) utilize an effort-aware performance assessment for their prediction models. Therefore, we also assess our prediction models' performances using an effort-aware measurement.

Effort-aware performance criterion basically assesses how much bug-inducing changes could be detected by inspecting only 20% of the total lines of code (LOC) during the review of changes predicted as bug-inducing. We calculated a benefit–cost ratio for each commit in the test set using the formula $P(c)/Effort(c)$ (Kamei et al., 2012). $P(c)$ is a binary value that represents the prediction made for commit c by a defect predictor (i.e., PM): 1 means (c) is bug-inducing, 0 means c is clean. $Effort(c)$ is the total number of changed lines (added and deleted) in the commit c. Then, we rank the commits in the test set based on their benefit–cost ratio in descending order. Later, we count the number of bug-inducing commits that could be detected when only 20% of the total effort is spent on code inspection (when the commits are inspected in descending order of their benefit–cost ratio).

Our effort-aware performance assessment shows that PM catches more bug-inducing commits than SM and GM when only 20% of the total code inspection effort is spent. While 30% of the total bug-inducing commits could be detected with PM, SM detects and GM detects 16% of the total bug-inducing commits over six projects. Our effort-aware performance results also support the prior studies' findings on PM against the traditional approaches. While prior studies report that the bug-inducing commit detection rates are increased by 12% (Jiang et al., 2013) and by 21% (Xia et al., 2016) when PM is chosen over the traditional models, we observe an increase by 14%.

5.7. Applicability to industrial settings

Building SDP models for the industry has several challenges, some of which are data availability for training, imbalanced bug-inducing versus not buggy commits, and input features (Tosun et al., 2010; Hryszko and Madeyski, 2015). Deploying these models to industrial settings, on the other hand, has other challenges, such as training data size, training/update period, and changes in the development team. Personalized SDP models would be

[4] https://kovan.itu.edu.tr/index.php/s/cR4dJpn6nL8wvdb

affected more than traditional models from the data availability during training and changes in the development team. For example, lack of commit history for a developer or lack of bug-inducing commits in a developer's commit history would prevent the developer from utilizing a personalized approach which was trained and customized for her. In such a scenario, only traditional models can provide recommendations to the developer until a sufficient amount of training data with two classes is collected. Another challenge of deploying personalized SDP models is related to the changes in the development team. During offline studies on historical data, we do not encounter cases when a new developer starts making commits, or when a senior developer stops contributing to the project. We aggregate all the commits, group those according to the developer who made these commits, and train personalized models for each developer. On the other hand, in a real-life scenario, some developers leave the project, whereas some join the project in the middle of the development process. Therefore there is a high possibility that some of the developers in the selected project would not have commit history for building a personalized model, while there would be former developers whose previous commit data would no longer be useful to anybody. Hence, a mechanism that utilizes the other developer's data to build a mixed model, or an alternative mechanism that switches between the general model and the personalized model for a specific developer depending on the data availability could be useful. Also, grouping developers based on their development similarities and building group-customized models would be another solution to a cold-start problem. In this study, we report our findings on open source projects by collecting their historical commit data. We are also working with our industrial partner, for which we had already built SDP models (Eken et al., 2019), to design personalized SDP models. We are currently working with them to address some of the issues mentioned above as well as time-sensitive model validation strategy mentioned in Section 5.4 in order to deploy such models into real-life industrial settings.

6. Threats to validity

Construct validity: The descriptive characteristics of the commits should be quantified through a metric set to measure the defect proneness of the commits. The metric set utilized in this study is widely used in change-level SDP studies in the literature (e.g. Misirli et al., 2016; Kamei et al., 2012; Shihab et al., 2012; Mockus and Weiss, 2000; Graves et al., 2000; Matsumoto et al., 2010; Nagappan and Ball, 2005; D'Ambros et al., 2010; Nagappan et al., 2006). The metrics quantify four different aspects of the changes, namely size, history and diffusion of the commits, and experience of the developers who made the commit. These aspects of the commits are measured by multiple metrics to avoid mono-metric bias in modeling, e.g., history dimension is measured in terms of three metrics, as it can be seen in Table 1: The number of developers who changed the modified files, the number of prior changes to modified files, and the average time interval between the last and the current change. We did not use complexity metrics or other object-oriented metrics used in the SDP literature because recent benchmarking studies (D'Ambros et al., 2012; Li et al., 2019) show that process metrics extracted from the code changes perform better than the other metrics on multiple projects.

Bug-fixing and bug-inducing commits were previously identified in Misirli et al. (2016) by using SZZ, and hence, we also used that dataset. The SZZ algorithm was used in the selected six open-source projects to define which parts of the code introduced the defects. The SZZ algorithm is frequently used in the literature to identify the bug-inducing commits by linking those with bug-fixing commits (Śliwerski et al., 2005). Bug-fixing commits are the locations where a bug is fixed, and fix locations are determined using the issue tracking system used by the development team. Later, the SZZ algorithm uses bug-fix-locations and traces back from bug-fix-locations to the previous commits to identify the commit where the bug was first injected into the software product. We are aware that there are still limitations of the SZZ algorithm, such as high false alarms, lack of identifying causes of new code additions, and variations in implementations by different researchers (Borg et al., 2019). As prior research on SZZ (Da Costa et al., 2016; Fan et al., 2019) indicated that SZZ implementations still need improvement to reduce the noise that causes mislabeled changes. Fan et al. (2019) empirically assessed the various SZZ implementations, and they reported that the original SZZ approach (Śliwerski et al., 2005) does not yield a significant performance reduction in just-in-time defect prediction models due to the noise in the change labels (bug-inducing or not). Therefore, we rely on the results of the SZZ algorithm used to collect the projects' data.

We try to cover as many developers as possible in our empirical analysis. Accordingly, PMs built in this study are trained with various numbers of commits, i.e., between 45 and 12.755. Due to the under-sampling, training set sizes also vary from 18 to 8.000. Some PMs have very few data instances in their training sets, and it might be possible to observe poorly performing PMs for those developers. We checked the performance of those PMs built with very few data instances, i.e., between 18 and 100 commits, against the other PMs, which are trained with more than 100 commits. We obtained a median of 0.5 pd rate for PMs in the first group (trained with less than 100 commits), whereas PMs in the second group (trained with more than 100 commits) produce a median of 0.35 pd rate when NB is used. Moreover, 89% of PMs in the first group statistically outperform SM and GM, while 75% of the PMs in the second group outperform SM and GM. These findings also support our answer for RQ2 on the commit counts of developers (Fig. 4): PM underperforms compared to SM and/or GM for those developers who contributed to a project with higher number of commits. A more strict developer selection procedure might be a primary choice considering that a good amount of data is needed to build a successful predictor (Raudys et al., 1991). However, our empirical analysis shows that we can build successful PMs with less than 100 commits.

Internal validity: In this study, we assume that each developer who contributed to each project made his/her commits according to some development principles. One of these assumptions is that developers linked their commits with the right issues in the issue repository. Another assumption is that each developer only commits his/her development codes. In some cases, the development of different developers may be committed by the authorized personnel only. We did not collect the dataset, however, we double-checked the mentioned issues and ensured all were correctly addressed in the prior studies using the same dataset.

We build a PM based on a developer's commits including bug-inducing ones and others, such as bug-fixing commits and other code changes. The author of a bug-fixing commit may not be the developer who has introduced the bug into the software system. Thus, a PM built for that developer does not include bug-fixing commit of another developer. Further, a bug-fixing commit may contain changes related to legacy code, i.e., the code is fixed due to a list of changes over the years, and the developer who fixed is not responsible in this case. SZZ is used to trace back to all prior commits which are made by the selected developers, and the data is added to their PMs respectively. Those situations may affect the performance of the personalized SDP models. However, there is not an accurate approach that can detect the exact reason and/or the source of the bugs in the software system. We believe that utilizing information from the historical aspect of the commits (i.e. number of prior changes to the modified files, number of developers had changed the modified files) helps to partially capture the indicators of above-mentioned situations (Arisholm and Briand, 2006).

Assessment of PM against traditional models is conducted on the commits of the selected developers as we described in Fig. 1. To assess personalized models, we have to focus on a set of developers whose available commits are enough to build PMs for them. Therefore, those developers' commits with limited contributions are not included into the analysis we conducted to answer our RQs. However, we also checked the prediction performance of both traditional models (SM and GM) on those developers' commits with the limited contribution, and obtained an average of 20% pd with NB. This ratio is very close to the value reported for traditional models in Fig. 2, and confirms that the even a state-of-the-art general model with all commits would reach a similar performance for the selected open source projects.

External validity: We performed our analyses on a set of open-source projects. Applying the proposed approach to different projects in industrial settings may not yield the same results obtained in this study. We think the project characteristics is an important factor that could affect the applicability and accuracy of the personalized SDP approaches. For instance, when the contributions in a project are too scattered among too many developers, PMs would perform better than traditional models. This is because, developers having bulk commits do not dominate the project (e.g. Rails), and each developer's commits contain sufficient amount of data indicating defect proneness. On the other hand, when the project has few developers responsible for majority of the all commits (e.g. PostgreSQL), traditional models perform better. Furthermore, the time period selected to collect the commits, the distribution of commits over this time period and the number of active developers should be considered during project selection. A similar conclusion is also reported in the latest benchmark (Li et al., 2019). To further understand the personalization in the SDP field, we plan to assess the performance of the proposed personalized SDP approach on a commercial project in the future.

Conclusion validity: During our experimentation, we use well-known open-source projects, model building and assessment techniques, and statistical tests (Section 3). The findings of our work are valid under the projects selected for this study, the metrics used, the algorithms used, and the strategies followed during model construction. To increase the conclusion validity of our research, we applied statistical tests on the differences between models' performance, and only base our conclusions on the differences with medium to large effect sizes. This way, we believe that we avoid potential bias related to sample size. The selected validation strategy, 10-fold cross-validation, might have influenced the results as it does not consider the temporality between changes. In Section 5, we elaborate on the validation strategy in detail, discuss prior works that compare different strategies, and we perform all empirical analyses regarding RQ1 with a time-sensitive data split strategy. Time-sensitive results also confirm that PM models are significantly better than traditional models for defect prediction, but we need more sample for large effect sizes. As the dataset is already publicly available, our results can also be reproduced and refuted in future studies.

7. Conclusion and future work

In this paper, we investigate the performance of personalized change-level software defect predictors by defining three research questions (RQs). Our personalized models (PM) achieve 24% higher probability of detection and 14% higher F1-measure rates than the two traditional models used in our study (SM and GM) (RQ1). Furthermore, we provide valuable insights for both researchers and practitioners regarding the set of development characteristics which highlight the superiority of PM over traditional models (RQ2). Furthermore, we investigate the common and the best indicators of bugs across different PMs (RQ3). We derive our conclusions based on a cross-validation setup built with NB algorithm, seven performance assessment metrics and statistically significant differences with medium/high effect sizes, but we also consider the experimental setup details that may affect our conclusions in our discussions. Over 222 developers from six open-source projects, we summarize our key take-away messages below:

- Even though the overall comparison between the performance of PM and traditional models leads us to prefer PMs over SMs or GMs, PM may not be suitable for every developer in a team. We observe that PMs built for most of the developers using NB algorithm detect more bugs, but general approaches may still be more suitable for the other developers.
- According to our empirical analysis, both traditional models (SM and GM) are more successful than PM for those developers who dominate the project's development activity with many commits. Those developers whose PM is similar to or more successful than at least one traditional model have less experience than others in software projects.
- PMs would give better predictions than traditional models for the developers who work on the software modules priorly modified by many developers. The group of developers whose PM is significantly better than SM/GM has higher NDEV (the number of developers had changed the modified files) metric values than the group of developers whose PM is equal to or significantly worse than SM/GM.
- Building a PM for a developer would need careful consideration on the process metrics since our analysis shows that the best indicators of bug-inducing changes differ among 222 PMs. Nevertheless, the number of added lines of code seems to be the most successful indicators of bugs. The developer experience metrics and the purpose metric are also the next successful indicators of bug-prone changes for the majority PMs after the number of added lines of code.
- The selection of PM over other models is also dependent on which performance measure is selected/prioritized: a personalized model could be better in detecting defects (pd) whereas it is worse in terms of false alarms (pf). Hence, the model assessment should be made according to the selected/prioritized performance measures.
- PM looks promising on its adoption to real-life since PM also outperforms the traditional models in a time-sensitive model validation setting, which resembles the real-life by preserving the temporal order of data during training and prediction. As we discussed in Section 5.4, further model validation scenarios are needed on the time-sensitive evaluation of PM to make a generalized conclusion for real-life applicability of PMs.

In this paper, we provide more insights than prior studies on the performance of personalized models. Still, we need to continue investigating the factors affecting the performance of PM as future work. Although we did not cover the project characteristics on the performance of PM, we think they are also important. For example, the time period of collected commits and the distribution of the commits of developers through the project time period may be some of the factors affecting the performance of personalized models. We have recently observed that a combination of statistical and machine learning techniques to extract multi-dimensional information from commit history would perform better than extracting a single aspect (processing through code changes) (Eken et al., 2019). We plan to understand the effect of such a combined technique on the performance of personalized models. Plus, industrial case studies will provide more insight on the applicability of personalized SDP models and the usability of the personalized recommendations. As we discussed in Section 5.7, our future plan is to investigate the practical side of personalized SDP approach by using an online model training strategy, and industrial software projects.

CRediT authorship contribution statement

Beyza Eken: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing - original draft. **Ayse Tosun:** Conceptualization, Methodology, Writing - review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank Ass. Prof. Emad Shihab and Ass. Prof. Yasukata Kamei for creating and sharing the dataset. This research is supported in part by the Scientific Research Project Unit of İstanbul Technical University under project number: MOA-2019-42321.

References

Aggarwal, C.C., et al., 2016. Recommender Systems, Vol. 1. Springer.
Arisholm, E., Briand, L.C., 2006. Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ACM, pp. 8–17.
Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J., 2014. Dimensions and metrics for evaluating recommendation systems. In: Recommendation Systems in Software Engineering. Springer, pp. 245–273.
Bangash, A.A., Sahar, H., Hindle, A., Ali, K., 2019. On the time-based conclusion stability of software defect prediction models. arXiv preprint arXiv:1911.06348.
Bell, R.M., Ostrand, T.J., Weyuker, E.J., 2013. The limited impact of individual developer data on software defect prediction. Empir. Softw. Eng. 18 (3), 478–505.
Bettenburg, N., Nagappan, M., Hassan, A.E., 2012. Think locally, act globally: Improving defect and effort prediction models. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, pp. 60–69.
Blog, G.O., 2009. Personalized search for everyone. https://googleblog.blogspot.com.tr/2009/12/personalized-search-for-everyone.html.
Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. SZZ unleashed: An open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation. MaLTeSQuE 2019, ACM, pp. 7–12.
Brier, G.W., 1950. Verification of forecasts expressed in terms of probability. Mon. Weather Rev. 78 (1), 1–3.
Calikli, G., Bener, A., 2015. Empirical analysis of factors affecting confirmation bias levels of software engineers. Softw. Qual. J. 23 (4), 695–722.
Catolino, G., Palomba, F., Tamburri, D.A., Serebrenik, A., Ferrucci, F., 2019. Gender diversity and women in software teams: How do they affect community smells? In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society. ICSE-SEIS, IEEE, pp. 11–20.
Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.
Conover, W.J., Conover, W.J., 1980. Practical Nonparametric Statistics. Wiley New York.
Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657.
D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories. MSR 2010, IEEE, pp. 31–41.
D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4–5), 531–577.
Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., De Lucia, A., 2017. A developer centered bug prediction model. IEEE Trans. Softw. Eng.
Eken, B., 2018. Assessing personalized software defect predictors. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM, pp. 488–491.
Eken, B., Atar, R., Sertalp, S., Tosun, A., 2019. Predicting defects with latent and semantic features from commit logs in an industrial setting. In: Proceedings of the 1st International Workshop on Software Engineering Intelligence. ACM.
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.
Falessi, D., Huang, J., Narayana, L., Thai, J.F., Turhan, B., 2018. On the need of preserving order of data when validating within-project defect classifiers. arXiv preprint arXiv:1809.01510.
Fan, Y., Xia, X., da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. IEEE Trans. Softw. Eng.
Friedman, J.H., 1991. Multivariate adaptive regression splines. Ann. Statist. 1–67.
Fu, W., Menzies, T., Shen, X., 2016. Tuning for software analytics: Is it really necessary? Inf. Softw. Technol. 76, 135–146.
Gasparic, M., Janes, A., 2016. What recommendation systems for software engineering recommend: A systematic literature review. J. Syst. Softw. 113, 101–113.
Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. 26 (7), 653–661.
Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.
Hall, G.A., Munson, J.C., 2000. Software evolution: code delta and code churn. J. Syst. Softw. 54 (2), 111–118.
Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 34–45.
Hryszko, J., Madeyski, L., 2015. Bottlenecks in software defect prediction implementation in industrial projects. Found. Comput. Decis. Sci. 40 (1), 17–33.
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. pp. 16–20.
Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 279–289.
Kamei, Y., Shihab, E., 2016. Defect prediction: Accomplishments and future challenges. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 5. SANER, IEEE, pp. 33–45.
Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.
Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I., 2007. A systematic review of effect size in software engineering experiments. Inf. Softw. Technol. 49 (11–12), 1073–1086.
Kim, S., Whitehead, Jr., E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.
Lee, T., Nam, J., Han, D., Kim, S., In, H.P., 2016. Developer micro interaction metrics for software defect prediction. IEEE Trans. Softw. Eng. 42 (11), 1015–1035.
Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.
Li, L., Lessmann, S., Baesens, B., 2019. Evaluating software defect prediction performance: an updated benchmarking study. arXiv preprint arXiv:1901.01726.
Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. 27, 504–518.
Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.i., Nakamura, M., 2010. An analysis of developer metrics for fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, p. 18.
Matthews, B.W., 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochim. Biophys. Acta (BBA)-Prot. Struct. 405 (2), 442–451.
Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1).

Misirli, A.T., Shihab, E., Kamei, Y., 2016. Studying high impact fix-inducing changes. Empir. Softw. Eng. 21 (2), 605–641.
Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.
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.
Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. Proceedings. 27th International Conference on. ICSE 2005, IEEE, pp. 284–292.
Nagappan, N., Ball, T., Zeller, A., 2006. Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering. pp. 452–461.
Nemenyi, P., 1963. Distribution-Free Multiple Comparisons (Ph.D. thesis). Princeton University, Princeton, New Jersey. unpublished.
Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2010. Programmer-based fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, p. 19.
Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2017. Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng.
Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. J. Syst. Softw. 150, 22–36.
Pornprasit, C., Tantithamthavorn, C., 2021. Jitline: a simpler, better, faster, finer-grained just-in-time defect prediction. arXiv preprint arXiv:2103.07068.
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, pp. 452–461.
Qiao, L., Wang, Y., 2019. Effort-aware and just-in-time defect prediction with neural network. PLoS One 14 (2).
Quinlan, J.R., 1986. Induction of decision trees. Mach. Learn. 1 (1), 81–106.
Rahman, F., Devanbu, P., 2011. Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 491–500.
Raudys, S.J., Jain, A.K., et al., 1991. Small sample size effects in statistical pattern recognition: Recommendations for practitioners. IEEE Trans. Pattern Anal. Mach. Intell. 13 (3), 252–264.
Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T., 2014. Recommendation Systems in Software Engineering. Springer Science & Business.
Rufibach, K., 2010. Use of brier score to assess binary predictions. J. Clin. Epidemiol. 63 (8), 938–939.
Schein, A.I., Popescul, A., Ungar, L.H., Pennock, D.M., 2002. Methods and metrics for cold-start recommendations. In: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 253–260.
Schröter, A., Zimmermann, T., Premraj, R., Zeller, A., 2006. If your bug database could talk. In: Proceedings of the 5th International Symposium on Empirical Software Engineering, Vol. 2. Citeseer, pp. 18–20.
Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M., 2012. An industrial study on the risk of software changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: ACM Sigsoft Software Engineering Notes, Vol. 30, No. 4. ACM, pp. 1–5.
Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 99–108.
Tantithamthavorn, C., Hassan, A.E., Matsumoto, K., 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Softw. Eng.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.
Tosun, A., Bener, A., Turhan, B., Menzies, T., 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Inf. Softw. Technol. 52 (11), 1242–1257.
Tsakiltsidis, S., Miranskyy, A., Mazzawi, E., 2016. On automatic detection of performance bugs. In: Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on. IEEE, pp. 132–139.
Tucker, C.E., 2014. Social Networks, Personalized Advertising, and Privacy Controls. American Marketing Association.
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. Softw.: Evol. Process 29 (1), e1797.
Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J., 2009. On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. 14 (5), 540–578.
Turhan, B., Mısırlı, A.T., Bener, A., 2013. Empirical evaluation of the effects of mixed project data on learning defect predictors. Inf. Softw. Technol. 55 (6), 1101–1118.
Weyuker, E.J., Ostrand, T.J., Bell, R.M., 2008. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empir. Softw. Eng. 13 (5), 539–559.
Xia, X., Lo, D., Wang, X., Yang, X., 2016. Collective personalized change classification with multiobjective search. IEEE Trans. Reliab. 65 (4), 1810–1829.
Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A.E., Ubayashi, N., 2015. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In: Proceedings of the 14th International Workshop on Principles of Software Evolution. ACM, pp. 46–55.
Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S., 2020. Just-in-time defect identification and localization: a two-phase framework. IEEE Trans. Softw. Eng..
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. pp. 157–168.
Yao, J., Shepperd, M., 2020. Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. In: Proceedings of the Evaluation and Assessment in Software Engineering. pp. 120–129.
Young, S., Abdou, T., Bener, A., 2018. A replication study: just-in-time defect prediction with ensemble learning. In: Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. pp. 42–47.
Zeller, A., 2009. Why Programs Fail: A Guide to Systematic Debugging. Elsevier.

Beyza Eken is a research assistant at the Faculty of Computer and Informatics Engineering, İstanbul Technical University, İstanbul, Turkey. She is pursuing her Ph.D. at Software Modeling and Analysis Laboratory, under the supervision of Dr. Ayşe Tosun. Her Ph.D. thesis focuses on the people aspect of software defect prediction, more specifically building personalized recommenders for software team. She received a B.S. degree (2011) in Computer Engineering from Sakarya University, Turkey and the M.S. degree (2015) in Computer Engineering from İstanbul Technical University, Turkey.

Ayşe Tosun is an assistant professor at the Faculty of Computer and Informatics Engineering, İstanbul Technical University (ITU) İstanbul, Turkey. Before joining ITU, she worked as a post-doctoral research fellow at the Department of Information Processing Science, University of Oulu, Finland. She received her Ph.D. in 2012, and MSc degree in 2008 from the Department of Computer Engineering, Boğaziçi University, Turkey. During her Ph.D., she worked as a research intern at Software Reliability Lab, Microsoft Research in Cambridge. Her research interests are empirical software engineering, specifically mining software data repositories, software measurement, software process improvement, software quality prediction models, and applications of AI on building recommendation systems for software engineering. Dr. Tosun serves as organizers and program chairs for the majority of international software engineering conferences, and as editors and reviewers for the leading journals of empirical software engineering.