IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 November 2022, accepted 2 December 2022, date of publication 7 December 2022,
date of current version 13 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3227339

RESEARCH ARTICLE

Extending Developer Experience Metrics for Better Effort-Aware Just-In-Time Defect Prediction

YEONGJUN CHO[1,2], JUNG-HYUN KWON[1,3], JOOYONG YI[4], AND IN-YOUNG KO[1], (Member, IEEE)

[1]School of Computing, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, Republic of Korea
[2]NAVER, Gyeonggi-do 1784, Republic of Korea
[3]KT Telecom, Gyeonggi-do 13606, Republic of Korea
[4]Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, Republic of Korea

Corresponding authors: Jung-Hyun Kwon (junghyun.kwon@kaist.ac.kr), Jooyong Yi (jooyong@unist.ac.kr), and In-Young Ko (iko@kaist.ac.kr)

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-2020-0-01795) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation); and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) under Grant 2021R1A2C1009819 and Grant 2021R1A5A1021944; and in part by the Institute for Information and Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIP), development of automatic software error repair technology that combines code analysis and error mining, under Grant 2021-0-01001.

ABSTRACT Developers use defect prediction models to efficiently allocate limited resources for quality assurance and appropriately make a plan for software quality improvement activities. Traditionally, defect predictions are conducted at the module level, such as the class or file level. However, a more recent trend is to perform defect prediction for a single or consecutive commits to the repository, which is known as just-in-time (JIT) defect prediction. JIT defect prediction finds error-prone changes instead of error-prone modules, and as a result, the developer only needs to investigate error-prone changed lines instead of the entire module. When building JIT defect prediction models, researchers used various metrics, including developer experience metrics which measure the developer's experiences. Despite the fact that software defectiveness is likely to be affected by the experience of developers, developer metrics were understudied in the literature. In this work, we investigate the impact of various novel developer experience metrics and their combinations on JIT defect prediction. Our experimental results are positive. We found that it is possible to improve the cost-effectiveness of defect prediction models consistently and statistically significantly by using our new developer experience metrics.

INDEX TERMS Software defect prediction, just-in-time defect prediction, developer experience metric, software quality management.

I. INTRODUCTION

Defect prediction helps developers identify software artifacts that are likely to have defects. It provides a sorted list of the potentially defective artifacts (those that are likely to be more error-prone appear earlier in the list), based on which developers can make a quality improvement plan considering limited resources [1]. To predict the error-proneness of each software module,[1] most defect prediction techniques use a prediction model trained with various code metrics, such as the code complexity [2] and change histories [3] known to be effective in estimating error-proneness.

Although a defect prediction technique helps developers narrow down the modules to inspect, module inspection still requires a large effort because the module predicted

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

[1]While in many programming languages, a module is defined as a file, some languages, such as Terraform, define a module as a directory containing a collection of files.

to be error-prone may contain many lines of code (LOCs). To overcome this drawback of module-level defect prediction, recent studies have introduced just-in-time (JIT) defect prediction models. Given a set of code changes committed to the repository, JIT defect prediction models predict whether this set of code changes contains a defect. Because the number of changed lines at each commit is typically much smaller than the size of a potentially defective file, predicting defect-inducing code changes is known to be more effective in locating the defective lines. Moreover, JIT defect prediction techniques can provide feedback at the earlier stage of development compared to file-level defect prediction because defect prediction can be conducted as soon as a change is made into the source code repository, rather than waiting until a new version of the software is ready for inspection [4], [5].

To predict whether a code change induces a defect, researchers have proposed various change-level metrics, some of which are shown in Table 1. These metrics measure, for example, how big a code change is (Size), how many modules are changed (Diffusion), which developer modified the code and how experienced the developer is (Experience).

To obtain accurate and useful prediction results from a defect prediction model, *it is important to use metrics effective for defect prediction.* To improve defect prediction performance, *we propose novel metrics extending the existing developer experience metrics.* Despite the fact that software defectiveness is likely to be affected by the experience of developers, developer metrics were understudied in the literature. In this work, we investigate the impact of various novel developer experience metrics and their combinations on JIT defect prediction. In particular, we compare the cost-effectiveness between different metrics. That is, we aim to find metrics that can reveal as many defects as possible after investigating the limited number of lines of changed code.

This paper makes the following contributions:

1) We propose new developer experience metrics extending the existing ones. We extend developer metrics with two different dimensions. For the first dimension, we consider various *granularities of modules* (e.g., systems, subsystems, and files). For the second dimension, we consider various ways to measure *how recently* the developer made a change on the module (e.g., commit time and version numbers). We also consider the combinations of both dimensions.
2) We empirically evaluate our new developer experience metrics. We compare the performance (i.e., the cost-effectiveness of trained models) of our metrics with the existing ones. We find that our new experience metrics improve the cost-effectiveness of defect prediction models. In particular, we report which combination of our metrics results in consistent and statistically significant improvements.

The rest of this article is organized as follows. Section II discusses the background of our study. Section III describes the new developer experience metrics that we propose. Section IV discusses the design of our experiments, and Section V presents the results of the experiments. Section VI reports the threats to validity, and Section VII provides the related works. Finally, Section VIII presents the conclusions and future work of this study.

TABLE 1. Common change-level metrics used in the previous studies.

Group	Name	Description
Size	LA	total number of added code lines
	LD	total number of deleted code lines
	LT	average number of code lines before the change
Diffusion	NS	total number of changed subsystems
	ND	total number of changed directories
	NF	total number of changed files
	Entropy	distribution of modified code across files
Purpose	FIX	whether containing a fix-related keyword in a change log
History	NDEV	average number of developers who touched a file so far
	AGE	average number of days passed since the last modification
	NUC	total number of unique changes
Experience	EXP	the number of commits an author made for the project
	SEXP	the number of commits an author made for the subsystem containing the changed files
	REXP	the number of commits an author made for the project, weighted by the age of the changes

II. BACKGROUND

A. DEFECT PREDICTION

When developing software, software quality is one of the most critical aspects to the stakeholders of software development (e.g., customers, developers, and managers). Given limited resources, it is usually not feasible to take a close look at all parts of the software. Thus, it is important to know which part of the software is likely to be buggy so that developers can improve the software quality as much as possible within a limited time. For this purpose, defect prediction was proposed [6].

Defect prediction is typically performed using a machine learning model. Figure 1 shows the typical workflow of defect prediction consisting of two phases. In the first phase (the upper right box of the figure), a defect prediction model M is trained with data where each item of the data consists of the values of n metrics (where n represents the number of metrics used to train M) and the label indicating whether the item is defective or not. As for metrics, diverse data such as code size, code complexity, and developer experience have been used in the literature, as shown in Table 1. The ''Name'' and ''Description'' column describes the name of the metrics and

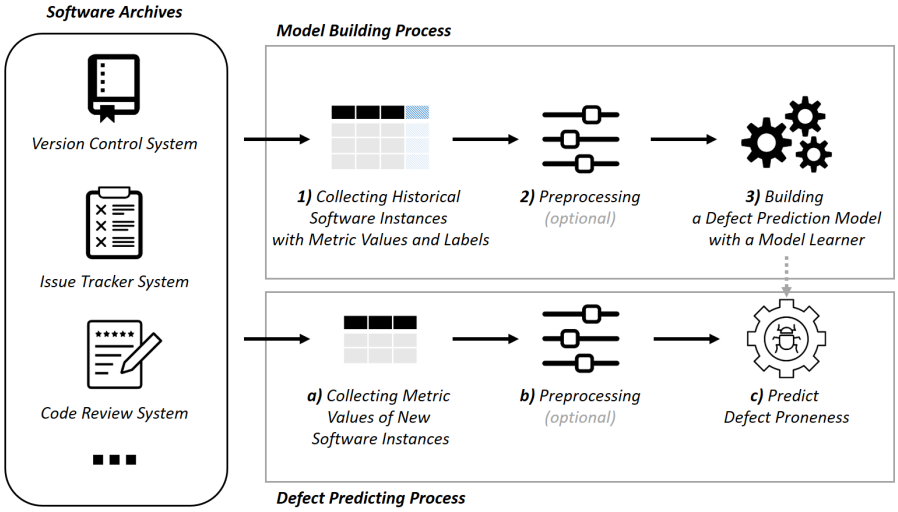


FIGURE 1. Overview of the defect prediction activity.

their descriptions, respectively. In defect prediction research, metrics are often categorized into several groups, as shown in the "Group" column of the table. Training data is typically mined from version-control systems. Once a defect prediction model M is trained, in the second phase (the lower right box of the figure), M is used to locate modules that are likely to be defective.

B. JUST-IN-TIME DEFECT PREDICTION

In a modern software development environment where software is constantly changing, developers continue to commit their code changes to a code repository. In this environment, developers need to know whether their code changes are defective or not. For this reason, JIT (Just-In-Time) defect prediction was introduced — JIT defect prediction predicts how much the current code changes are likely to be buggy.

C. CHANGE-LEVEL DEVELOPER EXPERIENCE METRICS USED IN JIT DEFECT PREDICTION

Metrics related to developer experience have been used in previous work [7], [8], [9], [10], [11], with the rationale that more experienced developers are less likely to write faulty code. While developer experience is difficult to measure, it is commonly approximated by counting the number of changes made by the developer—typically, one commit to a version control system is considered one change.

Specifically, the following three metrics have been used widely in the literature on defect prediction to measure developer experience: EXP (developer experience metric), SEXP (subsystem developer experience metric), and REXP (recent developer experience metric).

1) EXP

First, EXP for a change c is defined as:

$$\mathrm{EXP}(c) = |Changes(d_c)|$$

where d_c and $Changes(d_c)$ represent the developer who made the change c, and the changes d_c made on the *project* until c is made (including c), respectively.

Consider Figure 2 where the developer has made <Change #4> after making three changes from <Change #1> to <Change #3>. Then, EXP(<Change #4>) is 4 since the developer made a total of 4 changes, including <Change #4>. See the EXP column of Table 2.

2) SEXP

In a modern distributed software development environment, developers typically work on specific subsystems. Thus, a developer usually has different levels of expertise on different subsystems. A developer who has substantial expertise on a subsystem S_1 may not be familiar with another subsystem S_2 and is more likely to induce fault on S_2 than on S_1. However, the EXP metric does not distinguish the developer experience between different subsystems. Meanwhile, the second metric, SEXP, measures the developer's experience with the subsystems under change. The following is the definition of SEXP.

$$\mathrm{SEXP}(c) = |Changes(d_c, S(c))|$$

where $S(c)$ and $Changes(d_c, S(c))$ represent the subsystems modified by the change c, and the changes d_c made on a *subsystem* $s \in S(c)$ until c is made (including c), respectively.

In our running example, SEXP(<Change #4>) is 3 since with <Change #4>, the developer has made a change on two subsystems, subsys1 and subsys2, and she also changed one of those two subsystems at <Change #1> and <Change #2>. See the SEXP column of Table 2.

3) REXP

As developers accumulate experience on the system under development, they are less likely to write faulty code. REXP, defined in the following, correlates the error-proneness of the

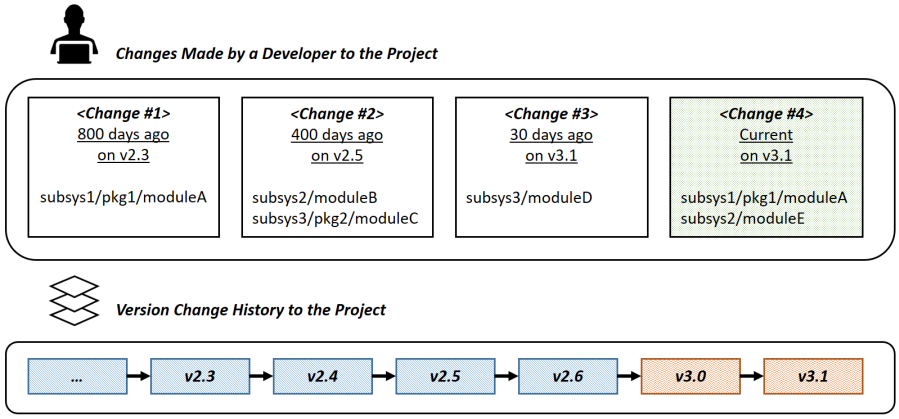


FIGURE 2. An example scenario of code changes.

TABLE 2. The values of the metrics for the example scenario shown in Figure 2.

Change ID	EXP	SEXP	REXP	MEXP	SimEXP	RVEXP	RvEXP
Change #1	1.000	1.000	0.333	1.000	0.500	0.500	0.166
Change #2	1.000	1.000	0.500	0.000	0.000	0.500	0.250
Change #3	1.000	0.000	1.000	0.000	0.000	1.000	1.000
Change #4	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Metric Value	4.000	3.000	2.833	2.000	1.500	3.000	2.416

change with how recently that change was made.

$$\mathrm{REXP}(c) = \sum_{c' \in Changes(d_c)} \frac{1}{1 + Y(c, c')}$$

where $Y(c, c')$ is $\lfloor Y(c) - Y(c') \rfloor$ where $Y(c)$ refers to the year when change c is made.

The REXP column of Table 2 shows how REXP(<Change #4>) is computed. Notice in Figure 2 that the developer made <Change #1> 800 days (less than 3 years) ago, <Change #2> 400 days (less than 2 years) ago, and <Change #3> 30 days (less than 1 year) ago. As shown in Table 2, REXP(<Change #4>) is the sum of 1/(1+2), 1/(1+1), 1/(1+0), and 1/(1+0).

III. NEW DEVELOPER EXPERIENCE METRICS

In this section, we describe our new developer experience metrics.

A. MODULE-BASED METRICS

1) MEXP

Although SEXP considers developer experience at a finer granularity level (i.e., a subsystem) than EXP, it may not be fine-grained enough. We propose a more fine-grained metric, MEXP, which considers developer experience at the module level — in this work, we define a module as a file. We define MEXP as follows:

$$\mathrm{MEXP}(c) = |Changes(d_c, M(c))|$$

where $M(c)$ and $Changes(d_c, M(c))$ refer to the modules modified by the change c, and the changes d_c made on a *module* $m \in M(c)$ until c is made (including c), respectively. MEXP counts the number of changes developer d_c made on the $M(c)$ modules.

In our running example, MEXP(<Change #4>) is 2 since at <Change #4>, moduleA and moduleE are modified, and one of these two modules is modified at <Change #1>. See the MEXP column of Table 2.

2) AVG_MEXP AND AVG_SEXP

Consider the following two scenarios. In the first scenario, the developer has created the initial version of 100 modules and committed them to the repository. In the second scenario, the developer has modified a single module she previously modified 99 times. Although the first change would be more error-prone than the second one, MEXP does not distinguish between them — in both cases, MEXP is 100. While this example is an extreme case, it reveals the limitation of MEXP. Motivated by this problem, we suggest a new metric AVG_MEXP defined as follows, which computes the average value of MEXP:

$$\mathrm{AVG\_MEXP}(c) = \frac{\mathrm{MEXP}(c)}{|M(c)|}$$

where $M(c)$ refers to the modules modified by change c. In our running example, AVG_MEXP(<Change #4>) is 1 since MEXP(<Change #4>) is 2 and <Change #4> modifies two modules.

Similar to AVG_MEXP, we also use AVG_SEXP defined as follows:

$$\mathrm{AVG\_SEXP}(c) = \frac{\mathrm{SEXP}(c)}{|S(c)|}$$

where $S(c)$ refers to the subsystems modified by change c.

3) SimEXP

Suppose that a developer d made the following two code changes. In the first code change c_1, she modified module m_1. In the second code change c_2, she modified two modules, m_1 and m_2. The developer may gain different

experiences when performing these two changes. While the task performed at c_1 is concentrated on a single module m_1, the second change c_2 is likely to involve interaction between the two modified modules. When the same developer d later makes a new change only on a module m_1, the experience she gained from c_1 is more likely to be relevant to the current task than the other experience gained from c_2. Conversely, when d modifies the two modules m_1 and m_2, the opposite is more likely to be true. More generally, we conjecture that the more similar the past experience of the developer is to the current change, that experience is likely to have a bigger influence on the current change. Here, the similarity between two changes c_1 and c_2 is proxied by the similarity between the two sets of modules modified in c_1 and c_2, which we formally describe as follows:

$$\frac{|M(c_1) \cap M(c_2)|}{|M(c_1) \cup M(c_2)|}$$

where $M(c_i)$ represents the modules modified in change c_i. Based on this notion, we define a new metric SimEXP as follows:

$$\text{SimEXP}(c) = \sum_{c' \in Changes(d_c)} \frac{\left|M(c) \cap M(c')\right|}{|M(c) \cup M(c')|}$$

In our running example, SimEXP is 1.5 as shown in Table 2. Note that for <Change #1>, $(|M(c) \cap M(c')|)/(|M(c) \cup M(c')|)$ is |{moduleA}|/|{moduleA, moduleE}|, which equals 0.5. The rest of the changes can be handled similarly.

B. TEMPORAL METRICS

1) RVEXP AND RvEXP

The motivation of REXP is to assign a larger weight to a more recent change. To define how recent a past change c is, REXP compares the time when c was made and the time when the current change is made.

There is another way to measure how recently the past changes were made. Many software products are maintained using the semantic versioning scheme [12], $n_1.n_2.n_3$, where n_1, n_2 and n_3 represent the major version number, the minor version number, and the patch version number, respectively. By comparing the versions of two different changes (i.e., commits), one can measure the interval between those two changes.

In this study, we experiment with two new metrics (RVEXP and RvEXP), each of which computes the interval at the granularity of the major versions (RVEXP) and the minor versions (RvEXP), respectively. We define these two new metrics as follows:

$$\text{RVEXP}(c) = \sum_{c' \in Changes(d_c)} \frac{1}{1 + V(c, c')}$$

$$\text{RvEXP}(c) = \sum_{c' \in Changes(d_c)} \frac{1}{1 + v(c, c')}$$

where $V(c, c')$ and $v(c, c')$ represent the version difference between c and c' at the granularity of major versions and minor versions, respectively.

In our running example, RVEXP is 3 as shown in Table 2. Note that the major version at <Change #1> and <Change #2> is 2, while the major version at <Change #3> is 3. Thus, for <Change #1> and <Change #2>, value 1/(1+1) is obtained. Meanwhile, for <Change #3> whose major version is 3, value 1/(1+0) is obtained.

RvEXP is computed similarly based on minor versions, as shown in Table 2. For example, the minor version difference between <Change #3> and <Change #1> is 5 (the version change history is shown in the bottom part of Figure 2), and the value 1/(1+5) is obtained.

2) RSEXP, RMEXP, RVSEXP, RVMEXP, RvSEXP AND RvMEXP

We also define temporal metrics at the subsystem and module levels as follows, similar to before.

$$\text{RSEXP}(c) = \sum_{c' \in Changes(d_c, S(c))} \frac{1}{1 + Y(c, c')}$$

$$\text{RMEXP}(c) = \sum_{c' \in Changes(d_c, M(c))} \frac{1}{1 + Y(c, c')}$$

$$\text{RVSEXP}(c) = \sum_{c' \in Changes(d_c, S(c))} \frac{1}{1 + V(c, c')}$$

$$\text{RVMEXP}(c) = \sum_{c' \in Changes(d_c, M(c))} \frac{1}{1 + V(c, c')}$$

$$\text{RvSEXP}(c) = \sum_{c' \in Changes(d_c, S(c))} \frac{1}{1 + v(c, c')}$$

$$\text{RvMEXP}(c) = \sum_{c' \in Changes(d_c, M(c))} \frac{1}{1 + v(c, c')}$$

3) AVG METRICS

For the six metrics shown in Section III-B2, we define the AVG metrics. For example, AVG_RSEXP and AVG_RMEXP are defined as follows:

$$\text{AVG\_RSEXP}(c) = \frac{\text{RSEXP}(c)}{|S(c)|}$$

$$\text{AVG\_RMEXP}(c) = \frac{\text{RMEXP}(c)}{|M(c)|}$$

IV. EXPERIMENT DESIGN

A. RESEARCH QUESTIONS

In this paper, our main goal is to see whether using our new developer experience metrics improves the performance of defect prediction. We accordingly ask the following research questions.

1) Do our module-based metrics improve the performance of defect prediction?
2) Do our temporal metrics improve the performance of defect prediction?

TABLE 3. Dataset used in our experiment.

Name	Domain	Languages	LOC	Commits	Developers	Defect Ratio
React	UI framework	JavaScript, HTML, CSS, etc	105,096	5587	1226	0.458
Iotivity	IoT framework	C++, C, Java, etc	424,932	5426	231	0.408
Dynomite	Storage engine	C, Shell, Python, etc	27,225	699	39	0.392
Notepad++	Code editor	C++, HTML, C, etc	337,041	2126	158	0.388
Vue	UI framework	JavaScript, HTML, TypeScript, etc	128,804	2216	194	0.333

3) Does the performance of defect prediction improve when our module-based and temporal metrics are combined?

B. DATASET

We extracted the dataset from the five open-source projects listed in Table 3. The table shows meta-data about the projects, including (from left to right) the domain of the projects, the programming languages used in the projects, LOC ,[2] the number of commits, the number of developers who contributed to the projects, and the ratio of defective commits. Our dataset covers diverse domains and programming languages. Note that the datasets used in the previous studies [4], [9], [13], [14] do not contain our new metrics and cannot be directly used for our experiments.

We extracted all 14 change-level metrics shown in Table 1.[3] In addition, we extracted our new experience metrics descried in Section III. To label whether a change is defective or not, we use the standard SZZ algorithm [16].

C. JIT DEFECT PREDICTION MODELS

We train our JIT defect prediction models using random forest [17]. Random forest is a classification (and regression) technique using multiple decision trees (we used 100 decision trees in our experiments). A classification decision (e.g., whether a change is defective or not) is made by performing the majority voting with the trained multiple decision trees. Random forest is commonly used in the literature of defect prediction research due to its high effectiveness [18], [19], [20]

D. PERFORMANCE MEASURES

To measure the performance of effort-aware JIT defect prediction, we use the Area Under the Cost Effectiveness Curve (AUCEC). AUCEC is commonly used in the literature on defect prediction to measure the cost-effectiveness of defect prediction [20], [21], [22].

Figure 3 illustrates AUCEC. In the figure, the X and Y-axis represent the ratio of inspected changed lines and the ratio of detected defects, respectively. Each point (x, y) of the curve denotes the portion of the detected defects (represented with the y value) after investigating the x portion of the changes (represented with the x value). Note that the ratio of the inspected changed lines is used as a proxy for the effort the developers put in.

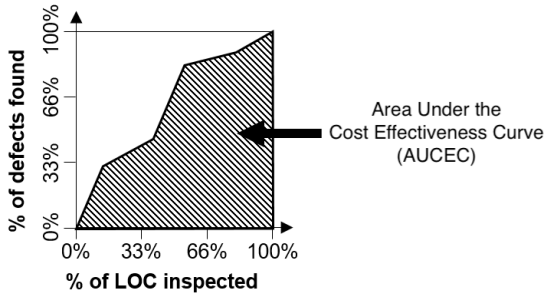


FIGURE 3. Area Under the Cost Effectiveness Curve (AUCEC).

When measuring AUCEC, we assume that the developers investigate changes c in the order of their cost-effectiveness scores $ce(c)$ computed using the following formula.

$$ce(c) = p(c) \times (1 - \frac{e(c)}{\max_{i \in Changes} e(i)}) \quad (1)$$

where $p(c)$ represents the error-proneness of c returned from the trained JIT defect prediction model,[4] $e(c)$ represents the effort proxied by the number of changed lines of c, and *Changes* represents a set of changes to investigate. Notice that as $p(c)$ increases (i.e., c is predicted to be more error-prone) and $e(c)$ decreases (i.e., c modifies the smaller number of lines), the value of $ec(c)$ increases.

If a defect prediction model A shows a higher AUCEC value than another model B, this implies that after investigating the same amount of lines of code, more defects are detected by A than B. In the literature on defect prediction, it is often assumed that developers usually investigate only $N\%$ of the changed lines within a limited time. As for the value of N, 20 is most often used, and we also use the same. We use the notation AUCEC_{20} to denote the AUCEC score obtained after inspecting the 20% of SLOC (source lines of code).

E. EVALUATION METHODS

To assess how useful our extended metrics are, we perform defect prediction with two different datasets, M_{base} and M_{target} where M_{base} contains the common existing metrics used in previous studies while M_{target} is defined as $M_{base} \cup$

[2]We obtained LOC using CLOC (https://github.com/AlDanial/cloc)

[3]We used the CodeRepoAnalyzer tool [15].

[4]Random forest computes the error-proneness score by computing the ratio of the number decision trees which determine the given change is defective over the total number of decision trees.

TABLE 4. AUCEC$_{20}$ of our module-based metrics; positive improvement rates (shown in the "↑ rate" column), p-values less than or equal to 0.5, and effect sizes (shown in the "effect" column) larger than or equal to 0.1 are highlighted in yellow.

(a) 30 times 10-fold time-aware cross-validation: comparison between SEXP and the metrics behind the double vertical bar (||)

Subject	SEXP	MEXP				AVG_MEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.052	0.054	4.861	0.142	0.06	0.052	0.241	0.327	0.04
Iotivity	0.024	0.024	-0.043	0.783	0.011	0.024	0.585	0.684	0.017
Vue	0.083	0.084	0.996	0.977	0.001	0.083	-0.343	0.547	0.025
React	0.046	0.049	5.282	0.135	0.061	0.047	2.991	0.407	0.034
Dynomite	0.036	0.038	6.877	0.341	0.039	0.037	3.81	0.423	0.033

Subject	SEXP	AVG_SEXP				SimEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.052	0.055	6.335	0.065	0.075	0.055	5.7	0.239	0.048
Iotivity	0.024	0.024	0.293	0.883	0.006	0.025	4.901	0.054	0.079
Vue	0.083	0.082	-1.657	0.668	0.018	0.084	0.537	0.659	0.018
React	0.046	0.044	-4.467	0.033	0.087	0.047	2.285	0.3	0.042
Dynomite	0.036	0.039	9.047	0.242	0.048	0.037	2.36	0.925	0.004

(b) Time-aware hold-out cross validation: comparison between SEXP and the metrics behind the double vertical bar (||)

Subject	SEXP	MEXP				AVG_MEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.047	0.049	4.916	0.0	0.288	0.047	0.904	0.058	0.077
Iotivity	0.042	0.043	2.213	0.089	0.069	0.043	2.032	0.021	0.094
Vue	0.110	0.110	0.372	0.932	0.003	0.115	4.438	0.0	0.178
React	0.091	0.093	2.801	0.0	0.171	0.095	4.198	0.0	0.269
Dynomite	0.025	0.031	20.664	0.0	0.311	0.035	38.131	0.0	0.531

Subject	SEXP	AVG_SEXP				SimEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.047	0.048	1.13	0.019	0.096	0.048	3.126	0.0	0.219
Iotivity	0.042	0.043	2.23	0.051	0.08	0.043	2.788	0.0	0.147
Vue	0.110	0.111	0.997	0.857	0.007	0.110	0.078	0.652	0.018
React	0.091	0.090	-0.827	0.015	0.1	0.094	3.258	0.0	0.215
Dynomite	0.025	0.022	-13.515	0.0	0.237	0.027	8.685	0.009	0.107

{extended metric(s)}. More specifically, we first add all metrics shown in Table 1 except for two developer experience metrics, SEXP and REXP, into M_{base}. To refer to these common base metrics, we use the notation M_{common}. Then, depending on which extended metrics are evaluated, we add either SEXP or REXP into M_{base}. When evaluating module-based metrics, we define M_{base} as $M_{common} \cup \{\text{SEXP}\}$. Meanwhile, when evaluating temporal metrics, we define M_{base} as $M_{common} \cup \{\text{REXP}\}$. This is to evaluate module-based (or temporal) metrics separately without them being affected by temporal (or module-based) metrics. When assessing RQ3 where we consider both module-based and temporal metrics, and accordingly define M_{base} as M_{base} as $M_{common} \cup \{\text{SEXP, REXP}\}$.

Given M_{base} and M_{target}, we compare their performance using the two validation methods described in the following.

1) 30 TIMES 10-FOLD TIME-AWARE CROSS VALIDATION

The 10-fold cross-validation method is commonly used to evaluate machine-learning models. This method splits the dataset into 10 folds and uses 9 for training and the remaining one for testing. In total, 10 different pairs of training/testing sets can be obtained, and all of them are used for validation. We repeat this process 30 times.

Considering the fact that a defection prediction model is trained with the past data, we make sure all commits in training set S_{train} are made before the commits in the testing set S_{test}, using the following method. For a given testing dataset S_{test}, we sort S_{test} in reverse chronological order. We also prepare two lists, S'_{test} and S'_{train}, initialized with an empty set and S_{train}, respectively. Then, we perform the following two tasks in a row.

1) We move the first item M_{test} from S_{test} to S'_{test}.
2) We find items $M_{train} \in S_{train}$ committed later than M_{test} and then remove M_{train} from S'_{train}.

We repeat these two steps as long as $|S'_{train}|/|S'_{test}|$ is larger than 9.

2) TIME-AWARE HOLD-OUT CROSS VALIDATION

We found that the 30 times 10-fold time-aware cross-validation often results in low statistical power. To compensate for this problem, we also apply hold-out cross validation. We sort our dataset in chronological order and use the first 90% of the data for training and the last 10% for testing. We train and test a model 300 times for each metric we evaluate.

V. EXPERIMENT RESULTS

A. RQ1. DO OUR MODULE-BASED METRICS IMPROVE THE PERFORMANCE OF DEFECT PREDICTION?

Tables 4(a) and 4(b) show the results for RQ1 from 30 times 10-fold time-aware cross validation and time-aware hold-out cross validation, respectively. The first column of the table

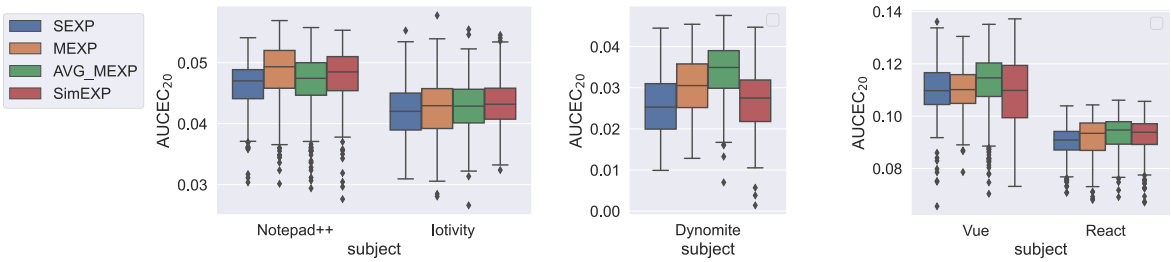


FIGURE 4. Time-aware hold-out cross-validation: performance comparison between SEXP and our three new metrics (MEXP, AVG_MEXP, SimEXP).

TABLE 5. 30 times 10-fold time-aware cross-validation: $AUCEC_{20}$ comparison between REXP and the metrics behind the double vertical bar (‖); positive improvement rates (shown in the "↑ rate" column), p-values less than or equal to 0.5, and effect sizes (shown in the "effect" column) larger than or equal to 0.1 are highlighted in yellow.

Subject	REXP	RVEXP				RvEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.053	0.054	2.868	0.535	0.025	0.054	2.89	0.738	0.014
Iotivity	0.024	0.024	-1.871	0.797	0.011	0.023	-3.611	0.533	0.025
Vue	0.074	0.083	12.42	0.047	0.081	0.085	15.586	0.024	0.092
React	0.049	0.049	0.201	0.869	0.007	0.048	-2.425	0.326	0.04
Dynomite	0.037	0.037	-1.185	0.86	0.007	0.036	-4.232	0.208	0.051

Subject	REXP	RSEXP				RVSEXP				RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.053	0.049	-6.954	0.179	0.055	0.051	-2.737	0.791	0.011	0.054	2.512	0.476	0.029
Iotivity	0.024	0.024	-1.115	0.821	0.009	0.024	-0.912	0.98	0.001	0.024	-1.839	0.615	0.021
Vue	0.074	0.082	11.043	0.017	0.097	0.086	17.03	0.002	0.127	0.079	7.541	0.198	0.053
React	0.049	0.047	-3.592	0.083	0.071	0.048	-2.197	0.366	0.037	0.050	2.411	0.9	0.005
Dynomite	0.037	0.039	4.257	0.552	0.024	0.039	5.4	0.263	0.046	0.035	-5.621	0.336	0.039

Subject	REXP	RMEXP				RVMEXP				RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.053	0.054	2.881	0.151	0.059	0.053	1.572	0.935	0.003	0.055	3.801	0.532	0.026
Iotivity	0.024	0.024	-0.856	0.638	0.019	0.024	0.396	0.313	0.041	0.024	-0.154	0.563	0.024
Vue	0.074	0.080	9.015	0.145	0.059	0.084	14.386	0.031	0.088	0.084	13.859	0.011	0.104
React	0.049	0.049	0.171	0.665	0.018	0.048	-1.388	0.631	0.02	0.049	0.229	0.499	0.028
Dynomite	0.037	0.037	0.817	0.461	0.03	0.037	-1.181	0.816	0.01	0.038	2.864	0.533	0.025

Subject	REXP	AVG_RSEXP				AVG_RVSEXP				AVG_RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.053	0.054	3.363	0.353	0.038	0.053	-0.003	0.905	0.005	0.055	4.124	0.205	0.052
Iotivity	0.024	0.023	-3.601	0.13	0.062	0.024	-1.615	0.329	0.04	0.024	-1.803	0.731	0.014
Vue	0.074	0.084	13.191	0.013	0.101	0.084	13.796	0.047	0.081	0.082	11.067	0.075	0.073
React	0.049	0.047	-4.699	0.051	0.08	0.048	-1.571	0.203	0.052	0.048	-2.236	0.308	0.042
Dynomite	0.037	0.037	-0.12	0.615	0.021	0.036	-2.475	0.86	0.007	0.037	-0.719	0.957	0.002

Subject	REXP	AVG_RMEXP				AVG_RVMEXP				AVG_RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.053	0.053	1.094	0.369	0.037	0.053	1.714	0.701	0.016	0.055	4.271	0.655	0.018
Iotivity	0.024	0.023	-3.287	0.703	0.016	0.023	-2.672	0.726	0.014	0.023	-2.26	0.931	0.004
Vue	0.074	0.081	9.541	0.085	0.07	0.083	12.127	0.109	0.065	0.088	19.726	0.0	0.159
React	0.049	0.048	-1.957	0.188	0.054	0.048	-1.318	0.237	0.048	0.048	-2.499	0.182	0.054
Dynomite	0.037	0.036	-2.95	0.43	0.032	0.037	-1.426	0.94	0.003	0.038	1.169	0.909	0.005

shows the subjects under evaluation, and the second column shows the median $AUCEC_{20}$ score obtained when the base metrics M_{base} is used. Recall that for RQ1, we define M_{base} as $M_{common} \cup \{SEXP\}$.

To assess our extended module-based metrics, we measure the $AUCEC_{20}$ score after replacing SEXP with each of those extended metrics. The third to fifth columns of the table show the results. For each extended metric, we report the median $AUCEC_{20}$ score, the improvement rate (↑ rate) (which we describe shortly), p-value, and effect size. We compute the p-value and effect size using the Wilcoxon-Mann-Whitney test [23]. The improvement rate shows how much $AUCEC_{20}$ score improves when SEXP is replaced with the metric under consideration. We define the improvement rate as $((\text{median}_{target} - \text{median}_{base}) / \text{median}_{base}) \times 100$.

The results of the first validation (30 times 10-fold time-aware cross-validation) show that our module-based metrics tend to cause a positive effect, although statistical significance is not observed except for one (AVG_SEXP for React). However, the hold-out validation results show that in most cases, statistically significant improvement is observed. Our three metrics, MEXP, AVG_MEXP, and SimEXP outperform SEXP across all subjects. In particular, AVG_MEXP outperforms SEXP with statistical significance (i.e., ≤ 0.05) across all subjects except for Notepad++ where the p-value is 0.058.

TABLE 6. Time-aware hold-out cross-validation: $AUCEC_{20}$ comparison between REXP and the metrics behind the double vertical bar (||); positive improvement rates (shown in the "↑ rate" column), p-values less than or equal to 0.5, and effect sizes (shown in the "effect" column) larger than or equal to 0.1 are highlighted in yellow.

Subject	REXP	RVEXP				RvEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.047	-0.961	0.031	0.088	0.044	-9.138	0.0	0.498
Iotivity	0.041	0.041	-1.235	0.869	0.007	0.041	-0.854	0.565	0.024
Vue	0.108	0.103	-4.609	0.0	0.26	0.078	-28.263	0.0	0.812
React	0.096	0.096	-0.356	0.579	0.023	0.092	-4.135	0.0	0.273
Dynomite	0.023	0.023	-3.415	0.063	0.076	0.031	30.95	0.0	0.432

Subject	REXP	RSEXP				RVSEXP				RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.048	1.218	0.127	0.062	0.049	1.534	0.028	0.089	0.048	0.264	0.794	0.011
Iotivity	0.041	0.042	3.486	0.0	0.186	0.043	4.243	0.0	0.22	0.045	8.518	0.0	0.375
Vue	0.108	0.116	6.797	0.0	0.433	0.111	2.377	0.0	0.205	0.108	0.023	0.519	0.026
React	0.096	0.093	-2.988	0.0	0.199	0.091	-5.134	0.0	0.337	0.091	-5.453	0.0	0.353
Dynomite	0.023	0.026	12.47	0.0	0.176	0.024	1.959	0.726	0.014	0.029	25.099	0.0	0.38

Subject	REXP	RMEXP				RVMEXP				RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.050	4.945	0.0	0.241	0.047	-2.153	0.029	0.089	0.052	9.257	0.0	0.552
Iotivity	0.041	0.042	1.708	0.041	0.084	0.042	3.463	0.001	0.137	0.043	4.811	0.0	0.268
Vue	0.108	0.108	-0.011	0.256	0.046	0.111	2.804	0.0	0.204	0.114	5.242	0.0	0.332
React	0.096	0.093	-2.961	0.0	0.197	0.092	-3.873	0.0	0.26	0.093	-2.899	0.0	0.196
Dynomite	0.023	0.030	27.562	0.0	0.408	0.029	23.696	0.0	0.263	0.030	28.038	0.0	0.428

Subject	REXP	AVG_RSEXP				AVG_RVSEXP				AVG_RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.046	-4.461	0.0	0.217	0.048	0.801	0.842	0.008	0.046	-4.268	0.0	0.229
Iotivity	0.041	0.039	-4.3	0.0	0.228	0.039	-5.003	0.0	0.203	0.039	-4.388	0.0	0.173
Vue	0.108	0.108	0.097	0.172	0.056	0.113	4.521	0.0	0.288	0.104	-3.664	0.0	0.185
React	0.096	0.090	-6.197	0.0	0.39	0.086	-10.001	0.0	0.535	0.091	-5.311	0.0	0.376
Dynomite	0.023	0.025	7.122	0.003	0.119	0.020	-13.662	0.0	0.204	0.028	18.348	0.0	0.277

Subject	REXP	AVG_RMEXP				AVG_RVMEXP				AVG_RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.047	-1.307	0.16	0.057	0.049	2.813	0.001	0.142	0.049	2.492	0.001	0.132
Iotivity	0.041	0.041	-0.263	0.811	0.01	0.042	1.643	0.069	0.074	0.041	0.277	0.629	0.02
Vue	0.108	0.109	0.386	0.179	0.055	0.112	3.129	0.0	0.232	0.129	18.986	0.0	0.667
React	0.096	0.095	-1.093	0.506	0.027	0.096	0.326	0.182	0.054	0.096	-0.286	0.71	0.015
Dynomite	0.023	0.024	1.244	0.938	0.003	0.028	19.354	0.0	0.231	0.026	10.034	0.001	0.134

The box plots in Figure 4 illustrate the observation that MEXP, AVG_MEXP, and SimEXP outperform SEXP.

> Our three module-based metrics, MEXP, AVG_MEXP, and SimEXP outperform the existing metrics across all subjects.

B. RQ2: DO OUR TEMPORAL METRICS IMPROVE THE PERFORMANCE OF DEFECT PREDICTION?

For RQ2, we define M_{base} as $M_{common} \cup \{\text{REXP}\}$. Similar to RQ1, we measure the $AUCEC_{20}$ score after replacing REXP with each of our temporal metrics. Tables 5 and 6 show the results for RQ2 from 30 times 10-fold time-aware cross validation and time-aware hold-out cross validation, respectively. As compared to the module-based metrics, positive effects are less observed. Nonetheless, we can observe from Table 6 that AVG_RVMEXP outperforms REXP across all subjects. Also, performance improvement is observed for all subjects except for React when RSEXP, RVSEXP, RvSEXP, RvMEXP, or AVG_RvMEXP is used.

> Our temporal metric, AVG_RVMEXP, outperforms the existing metrics across all subjects.

C. RQ3: DOES THE PERFORMANCE OF DEFECT PREDICTION IMPROVE WHEN OUR MODULE-BASED AND TEMPORAL METRICS ARE COMBINED?

Since combining 6 module-based metrics and 18 temporal metrics results in too many combinations (108), we combine the three best module-based metrics with which performance improvement is observed across all subjects (i.e., AVG_MEXP, SimEXP, and AVG_RVMEXP) and the six temporal metrics with which performance improvement is observed in at least 4 subjects (i.e., AVG_RVMEXP, RSEXP, RVSEXP, RvSEXP, RvMEXP, and AVG_RvEXP). We compare each of the 18 combinations with the base case where we define M_{base} as $M_{common} \cup \{\text{SEXP, REXP}\}$. Note that SEXP and REXP are the existing module-level and temporal metrics, respectively.

Tables 7 and 8 show the results. Table 7 shows the result of the 30 times 10-fold time-aware cross-validation, and it is observed that SimEXP+AVG_RvMEXP outperforms M_{base} across all subjects.

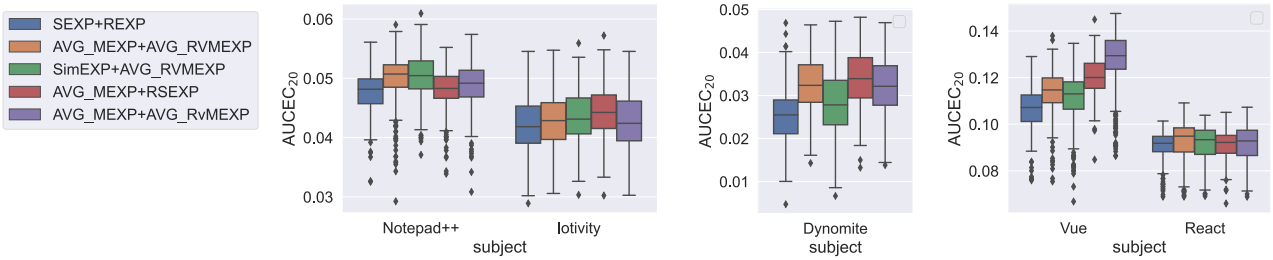


FIGURE 5. Time-aware hold-out cross-validation: performance comparison between SEXP+REXP and the four combination of our metrics.

Table 8 shows the result of the time-aware hold-out cross-validation. It is observed that in most cases, p-values are less than 0.05, and effect sizes are larger than 0.1, indicating statistically-significant non-negligible results are obtained. As compared to RQ1 and RQ2, combining our module-based and temporal metrics tends to cause more visible changes in performance.

SimEXP+AVG_RVMEXP outperforms M_{base} across all subjects, with statistical significance. In addition to that, in three more combinations (i.e., AVG_MEXP+AVG_RVMEXP, AVG_MEXP+RSEXP, and AVG_MEXP+AVG_RvMEXP), performance improves across all subjects. The box plots in Figure 5 illustrate the observation that these four combinations outperform the base case using the combination of SEXP and REXP.

> When combining our module-based and temporal metrics, statistically significant improvement is observed. In particular, SimEXP+AVG_RVMEXP outperforms the existing metrics across all subjects, with statistical significance.

VI. THREATS TO VALIDITY

A. CONSTRUCT VALIDITY

We collected independent variables (i.e, the change-level metrics) based on the CodeRepoAnalyzer [15], and the dependent variable (i.e., the variable indicating whether a commit is defective or not) using the SZZ algorithm. Although these algorithms have been widely used in defect prediction studies [4], [24], they may produce incorrect results (e.g., non-defective change may be labeled defective). The computation of REXP and its extended metrics (i.e., RSEXP, RMEXP) are computed based on the commit history. There is a potential threat to the validity in case the developers "squash" (merge) multiple commits since by doing so, the commit order between commits is lost.

B. INTERNAL VALIDITY

When measuring AUCEC, we used 20% as the cutoff point, as commonly conducted in the literature on defect prediction. Nonetheless, it is unknown which cutoff point is best. To mitigate this threat, we also evaluated the performance with the 10% cutoff point and observed the same general tendency.

C. EXTERNAL VALIDITY

We conducted the experiments with data from five open-source projects. Although we carefully chose various projects with different sizes, domains, and programming languages used, our subjects may not represent all software projects. Nonetheless, to the best of our knowledge, this is the first study that investigates the impact of extended developer experience metrics on defect prediction. We expect our positive results to foster further studies on developer experience metrics.

VII. RELATED WORKS

A. IDENTIFICATION OF BUGGY PATTERNS BASED ON DEVELOPER EXPERIENCE FACTORS

Matsumoto et al. [25] defined five metrics that characterize a developer's activities for a specific version of the software and analyzed the correlation between those metrics and the ratio of the buggy commits authored by a developer. The five metrics they defined for each developer for a specific software version are 1) the number of commits made by a developer 2) the number of lines revised by a developer 3) the number of unique modules revised by a developer, 4) the number of unique packages revised by a developer, and 5) the ratio of buggy commits by a developer for the previous version. Analysis results showed that the number of unique modules revised and the ratio of buggy commits for the previous version significantly correlated with the ratio of buggy commits for the chosen version. Although the authors used version information in collecting a developer's experience, those developer experience metrics are defined per developer for a specific version rather than per change. Moreover, whether these metrics are a good predictor of defect prediction was not determined, even though the authors showed that developer experiences may have an impact on software quality.

Bird et al. [26] examined the effects of code ownership on software quality. For each file, they counted the number of contributors; the number of minor or major contributors, which is distinguished by the ratio of the contribution, whether it is higher than 5%; and the ownership, which is computed by using the top contributor's contribution

TABLE 7. 30 times 10-fold time-aware cross-validation: $AUCEC_{20}$ comparison between SEXP+REXP and the metrics behind the double vertical bar (||); positive improvement rates (shown in the "↑ rate" column), p-values less than or equal to 0.5, and effect sizes (shown in the "effect" column) larger than or equal to 0.1 are highlighted in yellow.

Subject	SEXP+REXP	AVG_MEXP+AVG_RVMEXP				SimEXP+AVG_RVMEXP				MEXP+AVG_RVMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.051	4.769	0.159	0.058	0.051	5.304	0.2	0.052	0.053	8.89	0.03	0.088
Iotivity	0.024	0.024	0.403	0.73	0.014	0.025	2.777	0.51	0.027	0.024	-1.459	0.35	0.038
Vue	0.081	0.077	-4.191	0.033	0.087	0.083	2.141	0.59	0.022	0.078	-3.732	0.13	0.062
React	0.046	0.048	4.747	0.11	0.065	0.047	2.755	0.698	0.016	0.050	9.177	0.034	0.087
Dynomite	0.039	0.039	0.031	0.593	0.022	0.036	-7.597	0.362	0.037	0.039	1.866	0.306	0.042

Subject	SEXP+REXP	AVG_MEXP+RSEXP				SimEXP+RSEXP				MEXP+RSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.050	3.421	0.764	0.012	0.051	4.224	0.817	0.009	0.052	5.56	0.502	0.027
Iotivity	0.024	0.024	1.628	0.67	0.017	0.024	0.448	0.906	0.005	0.023	-2.214	0.097	0.068
Vue	0.081	0.083	2.375	0.941	0.003	0.083	2.603	0.702	0.016	0.086	6.491	0.332	0.04
React	0.046	0.048	4.844	0.408	0.034	0.046	-0.227	0.634	0.019	0.048	3.528	0.308	0.042
Dynomite	0.039	0.039	-0.109	0.681	0.017	0.037	-3.386	0.731	0.014	0.038	-1.924	0.729	0.014

Subject	SEXP+REXP	AVG_MEXP+RVSEXP				SimEXP+RVSEXP				MEXP+RVSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.052	5.938	0.126	0.063	0.050	3.417	0.649	0.019	0.053	8.829	0.063	0.076
Iotivity	0.024	0.024	0.993	0.465	0.03	0.024	0.283	0.918	0.004	0.024	-0.727	0.381	0.036
Vue	0.081	0.086	6.64	0.668	0.018	0.082	1.896	0.938	0.003	0.075	-6.652	0.04	0.084
React	0.046	0.047	2.741	0.332	0.04	0.046	-0.645	0.992	0.0	0.047	3.427	0.173	0.056
Dynomite	0.039	0.037	-4.65	0.691	0.016	0.038	-1.088	0.592	0.022	0.036	-6.865	0.656	0.018

Subject	SEXP+REXP	AVG_MEXP+RvSEXP				SimEXP+RvSEXP				MEXP+RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.053	8.22	0.14	0.06	0.051	4.266	0.646	0.019	0.053	8.077	0.06	0.077
Iotivity	0.024	0.024	-0.234	0.838	0.008	0.025	3.003	0.815	0.01	0.024	-0.045	0.626	0.02
Vue	0.081	0.080	-1.41	0.048	0.081	0.084	3.868	0.772	0.012	0.082	1.49	0.35	0.038
React	0.046	0.048	4.29	0.502	0.027	0.047	2.446	0.603	0.021	0.048	5.535	0.189	0.054
Dynomite	0.039	0.036	-6.348	0.296	0.043	0.035	-9.273	0.264	0.046	0.038	-1.515	0.716	0.015

Subject	SEXP+REXP	AVG_MEXP+RvMEXP				SimEXP+RvMEXP				MEXP+RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.052	6.516	0.22	0.05	0.053	8.519	0.138	0.061	0.054	10.006	0.086	0.07
Iotivity	0.024	0.024	1.416	0.793	0.011	0.024	2.695	0.345	0.039	0.024	1.271	0.579	0.023
Vue	0.081	0.085	5.021	0.682	0.017	0.085	5.123	0.52	0.026	0.079	-2.65	0.357	0.038
React	0.046	0.048	5.369	0.182	0.054	0.048	5.57	0.206	0.052	0.047	2.561	0.391	0.035
Dynomite	0.039	0.037	-5.309	0.277	0.044	0.037	-5.467	0.514	0.027	0.037	-4.64	0.712	0.015

Subject	SEXP+REXP	AVG_MEXP+AVG_RvMEXP				SimEXP+AVG_RvMEXP				MEXP+AVG_RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.049	0.053	9.251	0.037	0.085	0.053	8.27	0.022	0.093	0.053	8.384	0.036	0.085
Iotivity	0.024	0.024	1.165	0.671	0.017	0.024	1.439	0.983	0.001	0.024	-0.866	0.619	0.02
Vue	0.081	0.086	6.018	0.766	0.012	0.085	5.722	0.853	0.008	0.090	10.986	0.18	0.055
React	0.046	0.049	6.645	0.067	0.075	0.048	3.559	0.203	0.052	0.049	7.697	0.029	0.089
Dynomite	0.039	0.037	-3.097	0.656	0.018	0.039	0.505	0.579	0.023	0.038	-1.695	0.503	0.027

ratio. To evaluate the effects of four ownership metrics on software quality, they conducted a correlation analysis of the pre- and post-release failures and built linear regression models using code metrics and ownership metrics as the independent variables and failures as the dependent variable. They specified that their purpose for building the linear regression models was not to predict whether a file contains any defect but to check whether the ownership metrics can be effectively used in classification models. Based on their experimental results, they recommended that developers should review the changes made by minor contributors more carefully since their limited experiences may induce defects. Unlike our work, this work is conducted at the file level, not at the change level.

Eyolfson et al. [27] studied the correlation between the error-proneness of a commit and the developer's experience, which was proxied by the days passed after the first commit the developer made on the Linux kernel and PostgreSQL projects. They reported that there are several threats that may affect the interpretation of the relationship between the developer experience metric and the bugginess of a commit, such as more experienced developers working on more complex source code or inflation of the developer experience metric value caused by his/her extremely low commit frequency. Nonetheless, the authors observed that data from both projects showed that the error-proneness of a commit decreases as the author's experience increases in general, and they reported that this correlation could be exploited in predicting the locations of buggy code. Although the authors showed the possibility of using developer experience metrics in defect prediction, they used a very basic method in quantifying a developer's experience.

TABLE 8. Time-aware hold-out cross-validation: $AUCEC_{20}$ comparison between SEXP+REXP and the metrics behind the double vertical bar (||); positive improvement rates (shown in the "↑ rate" column), p-values less than or equal to 0.5, and effect sizes (shown in the "effect" column) larger than or equal to 0.1 are highlighted in yellow.

Subject	SEXP+REXP	AVG_MEXP+AVG_RVMEXP				SimEXP+AVG_RVMEXP				MEXP+AVG_RVMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.051	5.307	0.0	0.359	0.050	4.747	0.0	0.362	0.044	-7.604	0.0	0.491
Iotivity	0.042	0.043	2.452	0.091	0.069	0.043	3.053	0.001	0.14	0.042	1.024	0.826	0.009
Vue	0.107	0.115	7.03	0.0	0.392	0.113	5.451	0.0	0.286	0.084	-21.963	0.0	0.817
React	0.092	0.095	3.325	0.0	0.199	0.093	1.674	0.01	0.104	0.093	1.719	0.0	0.143
Dynomite	0.025	0.032	26.991	0.0	0.482	0.028	9.175	0.0	0.197	0.031	22.464	0.0	0.396

Subject	SEXP+REXP	AVG_MEXP+RSEXP				SimEXP+RSEXP				MEXP+RSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.048	0.253	0.225	0.05	0.043	-11.161	0.0	0.692	0.044	-9.413	0.0	0.597
Iotivity	0.042	0.044	5.751	0.0	0.248	0.045	7.708	0.0	0.302	0.043	3.533	0.001	0.131
Vue	0.107	0.120	12.016	0.0	0.644	0.093	-12.845	0.0	0.664	0.088	-17.68	0.0	0.761
React	0.092	0.092	0.403	0.745	0.013	0.091	-0.416	0.09	0.069	0.092	0.084	0.622	0.02
Dynomite	0.025	0.034	33.091	0.0	0.574	0.030	16.291	0.0	0.257	0.030	19.39	0.0	0.341

Subject	SEXP+REXP	AVG_MEXP+RVSEXP				SimEXP+RVSEXP				MEXP+RVSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.050	3.066	0.0	0.189	0.043	-10.208	0.0	0.649	0.044	-8.283	0.0	0.531
Iotivity	0.042	0.044	6.042	0.0	0.238	0.042	1.036	0.355	0.038	0.041	-1.059	0.083	0.071
Vue	0.107	0.116	8.537	0.0	0.484	0.082	-23.205	0.0	0.813	0.077	-27.826	0.0	0.839
React	0.092	0.090	-2.269	0.0	0.189	0.093	0.851	0.149	0.059	0.093	1.709	0.002	0.127
Dynomite	0.025	0.032	25.006	0.0	0.39	0.023	-8.376	0.001	0.133	0.026	3.558	0.046	0.082

Subject	SEXP+REXP	AVG_MEXP+RvSEXP				SimEXP+RvSEXP				MEXP+RvSEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.048	0.528	0.96	0.002	0.045	-7.406	0.0	0.517	0.046	-4.973	0.0	0.365
Iotivity	0.042	0.045	7.239	0.0	0.308	0.044	4.183	0.002	0.124	0.041	-1.625	0.044	0.082
Vue	0.107	0.113	5.302	0.0	0.307	0.081	-24.541	0.0	0.828	0.078	-27.113	0.0	0.845
React	0.092	0.089	-3.391	0.0	0.246	0.090	-1.589	0.001	0.138	0.090	-1.458	0.001	0.132
Dynomite	0.025	0.032	27.185	0.0	0.534	0.031	23.069	0.0	0.426	0.033	31.203	0.0	0.55

Subject	SEXP+REXP	AVG_MEXP+RvMEXP				SimEXP+RvMEXP				MEXP+RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.045	-7.415	0.0	0.494	0.045	-6.79	0.0	0.442	0.046	-3.994	0.0	0.304
Iotivity	0.042	0.045	6.546	0.0	0.245	0.044	5.35	0.0	0.236	0.043	2.681	0.009	0.106
Vue	0.107	0.090	-15.804	0.0	0.73	0.088	-18.028	0.0	0.76	0.085	-20.915	0.0	0.801
React	0.092	0.092	0.711	0.531	0.026	0.091	-0.639	0.164	0.057	0.091	-0.415	0.448	0.031
Dynomite	0.025	0.033	29.267	0.0	0.568	0.030	18.978	0.0	0.352	0.031	20.649	0.0	0.458

Subject	SEXP+REXP	AVG_MEXP+AVG_RvMEXP				SimEXP+AVG_RvMEXP				MEXP+AVG_RvMEXP			
	median	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect	median	↑ rate	p-value	effect
Notepad++	0.048	0.049	2.106	0.0	0.179	0.050	4.526	0.0	0.368	0.045	-7.063	0.0	0.489
Iotivity	0.042	0.042	1.355	0.137	0.061	0.043	2.256	0.29	0.043	0.042	-0.221	0.253	0.047
Vue	0.107	0.129	20.744	0.0	0.668	0.130	21.12	0.0	0.54	0.091	-14.938	0.0	0.715
React	0.092	0.093	1.108	0.035	0.086	0.092	0.482	0.418	0.033	0.093	1.07	0.067	0.075
Dynomite	0.025	0.032	26.12	0.0	0.458	0.028	10.579	0.0	0.169	0.029	13.75	0.0	0.3

Moreover, the performance impact of the developer experience metric they proposed for defect prediction models was not evaluated.

Tufano et al. [28] analyzed the effect of the experience level of developers on the bugginess of a commit on five Java open-source projects. They defined four different developer experience metrics at the change level that consider the lexical experience and the frequency of experience on modified files. More specifically, the lexical experience metric is calculated by using the textual similarity between the texts in a modified file and the concatenated texts from all files modified by an author of the change. After obtaining the lexical experience on the files that were modified in a commit, the authors computed the mean value of the lexical experience on multiple files to ensure that the metric is defined at the change level. The frequency of experience was computed by counting the number of commits that were made by the author on the file modified in the target commit, and then dividing that counted number by the number of the commits the author made in the past. Furthermore, two additional developer experience metrics were defined in the same manner as the previous two metrics, except that these metrics only consider the commits from the past six months. The authors concluded that the mean value of the four developer experience metrics from fix-inducing commits and clean commits was significantly different. Although they defined four new developer experience metrics and showed the possibility of the usefulness of these metrics in defect prediction models, they used fixed time windows for calculating the developer's recent experience, and the performance

impact of these metrics on JIT defect prediction was not shown.

B. DEVELOPER EXPERIENCE METRIC ON JIT DEFECT PREDICTION

Mockus and Weiss [8] suggested various change-level metrics, including EXP, SEXP, and REXP described in Section II. Kamei et al. [4] used various metrics, including the developer experience metrics of Mockus and Weiss [8] to evaluate the performance of JIT defect prediction. However, they did not extend developer experience metrics. McIntosh and Kamei [9] proposed the author awareness metrics, which is defined as the proportion of past changes that were made to a subsystem that the reviewer has authored or reviewed. They did not find this metric useful in improving the performance of the JIT defect prediction. In this work, we proposed another developer experience metrics that show a positive effect on the performance of JIT defect prediction.

VIII. CONCLUSION AND FUTURE WORK

In this work, we have proposed novel developer experience metrics. In particular, we extended the widely-used two experience metrics, SEXP and REXP. SEXP is defined at the granularity of subsystems, and we have proposed MEXP defined at the file granularity. We also proposed SimEXP, which measures the similarities between commits. Regarding REXP, which measures how recently the developer made a change with the unit of the year, we have suggested RVEXP and RvEXP, which measures the same with the unit of the major and minor versions, respectively. We also suggested the variation of those metrics (i.e., AVG_RVMEXP) by averaging the experiences, instead of summing them up. We also combined these metrics together when conducting experiments.

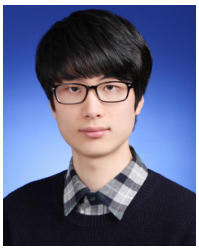
Our experimental results show that our new metrics often improve the cost-effectiveness of defect-prediction models. When we combined module-based metrics and temporal metrics, we obtained stronger results. In particular, when combining SimEXP and AVG_RVMEXP, the statistically significant performance improvement was observed across all 5 subjects. In future work, we plan to experiment with more subjects to study how general our findings are.

ACKNOWLEDGMENT

This work is done while Stavros Yeongjun Cho and Jung-Hyung Kwon were at KAIST.

REFERENCES

[1] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Nov. 2004, pp. 417–428.
[2] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, May 1992.
[3] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 181–190.
[4] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
[5] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.
[6] F. Akiyama, "An example of software system debugging," in *Proc. IFIP Congr.*, vol. 71, 1971, pp. 353–359.
[7] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 157–168.
[8] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, Apr. 2000.
[9] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018. [Online]. Available: https://ieeexplore.ieee.org/document/7898457/
[10] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," 2017, *arXiv:1703.00132*.
[11] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, Apr. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121218302656
[12] T. Preston-Werner. (2017). *Semantic Versioning 2.0.0*. [Online]. Available: https://semver.org/spec/v2.0.0.html
[13] H. Jahanshahi, D. Jothimani, A. Başar, and M. Cevik, "Does chronology matter in JIT defect prediction?: A partial replication study," in *Proc. 15th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, Sep. 2019, pp. 90–99.
[14] J. Gesi, J. Li, and I. Ahmed, "An empirical examination of the impact of bias on just-in-time defect prediction," in *Proc. 15th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2021, pp. 1–12.
[15] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug. 2015, pp. 966–969.
[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories (MSR)*, 2005, pp. 1–5.
[17] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R News*, vol. 2, no. 3, p. 6, 2002.
[18] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.
[19] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on defect models," 2018, *arXiv:1801.10271*.
[20] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the 'Naturalness' of buggy code," in *Proc. 38th Int. Conf. Softw. Eng.* New York, NY, USA: ACM, 2016, pp. 428–439.
[21] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, Jan. 2010.
[22] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng. (PROMISE)*, 2009, pp. 7:1–7:10.
[23] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, Mar. 1947.
[24] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 279–289.
[25] S. Matsumoto, Y. Kamei, A. Monden, K.-I. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng. (PROMISE)*, 2010, p. 18.
[26] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (SIGSOFT/FSE)*, 2011, pp. 4–14.
[27] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess," in *Proc. 8th Work. Conf. Mining Softw. Repositories (MSR)*, 2011, pp. 153–162.
[28] M. Tufano, G. Bavota, D. Poshyvanyk, M. D. Penta, R. Oliveto, and A. D. Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits: Developer-related factors characterizing fix-inducing commits," *J. Softw., Evol. Process*, vol. 29, no. 1, Jan. 2017, Art. no. e1797.



YEONGJUN CHO received the master's degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2019. He is currently a Site Reliability Engineer at Naver Corporation, South Korea. His research interests include software engineering and site reliability engineering. More information about him is available at: https://www.linkedin.com/in/yeongjuncho/.

JUNG-HYUN KWON received the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2018. He is currently a Research Engineer at Korea Telecom, South Korea. His research interests include software engineering and machine learning. More information about him is available at: https://www.linkedin.com/in/jung-hyun-kwon.

JOOYONG YI received the Ph.D. degree in computer science from Aarhus University, in 2007. He is currently an Assistant Professor at the Ulsan National Institute of Science and Technology (UNIST). Before joining the present position, he worked in various places, such as Kansas State University, the National University of Singapore, and Innopolis University. His research interests include software engineering and programming languages. His current research focus is on automated software engineering, such as automated bug fixing. He is one of the winners of the Facebook Testing and Verification Research Award, in 2018. He has been serving as a Program Committee Member of various premier conferences on software engineering, such as the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), the ACM International Symposium on Software Testing and Analysis (ISSTA), and the International Conference on Software Engineering (ICSE). More information about him is available at: http://www.jooyongyi.com/.

IN-YOUNG KO (Member, IEEE) received the Ph.D. degree in computer science from the University of Southern California (USC), in 2003. He is currently a Professor with the School of Computing and the Director of the Research Center for Big Data Edge Cloud Services (BECS), Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. His research interests include services computing, web engineering, and software engineering. His recent research focuses on service-oriented software development in large-scale and distributed system environments, such as web, the Internet of Things (IoT), and edge-cloud environments. More information about him is available at: https://webeng.kaist.ac.kr/webengpress/professor/.