Just-In-Time Defect Prediction on JavaScript Projects: A Replication Study

CHAO NI, School of Software Technology, Zhejiang University, China
XIN XIA, Software Engineering Application Technology Lab, Huawei, China
DAVID LO, Singapore Management University, Singapore
XIAOHU YANG, Zhejiang University, China
AHMED E. HASSAN, Queen's University, Canada

Change-level defect prediction is widely referred to as just-in-time (JIT) defect prediction since it identifies a defect-inducing change at the check-in time, and researchers have proposed many approaches based on the language-independent change-level features. These approaches can be divided into two types: supervised approaches and unsupervised approaches, and their effectiveness has been verified on Java or C++ projects. However, whether the language-independent change-level features can effectively identify the defects of JavaScript projects is still unknown. Additionally, many researches have confirmed that supervised approaches outperform unsupervised approaches on Java or C++ projects when considering inspection effort. However, whether supervised JIT defect prediction approaches can still perform best on JavaScript projects is still unknown. Lastly, prior proposed change-level features are programming language–independent, whether programming language–specific change-level features can further improve the performance of JIT approaches on identifying defect-prone changes is also unknown.

To address the aforementioned gap in knowledge, in this article, we collect and label the top-20 most starred JavaScript projects on GitHub. JavaScript is an extremely popular and widely used programming language in the industry. We propose five JavaScript-specific change-level features and conduct a large-scale empirical study (i.e., involving a total of 176,902 changes) and find that (1) supervised JIT defect prediction approaches (i.e., CBS+) still statistically significantly outperform unsupervised approaches on JavaScript projects when considering inspection effort; (2) JavaScript-specific change-level features can further improve the performance of approach built with language-independent features on identifying defect-prone changes; (3) the change-level features in the dimension of size (i.e., LT), diffusion (i.e., NF), and JavaScript-specific (i.e., SO and TC) are the most important features for indicating the defect-proneness of a change on JavaScript projects; and (4) project-related features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective, and Forks) have a high association with the probability of a change to be a defect-prone one on JavaScript projects.

This research/project is partially supported by the National Science Foundation of China (Grant No. U20A20173), Key Research and Development Program of Zhejiang Province (Grant No. 2021C01014), and the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative.

Authors' addresses: C. Ni, School of Software Technology, Zhejiang University; Zheda Road 38 West Lake District, Hangzhou, Zhejiang, China; email: chaoni@zju.edu.cn; X. Xia (corresponding author), Software Engineering Application Technology Lab, Huawei, Jianghong Road 410, Binjiang District, Hangzhou, Zhejiang, China; email: xin.xia@acm.org; D. Lo, Singapore Management University, 80 Stamford Road, Singapore; email: davidlo@smu.edu.sg; X. Yang, College of Computer Science and Technology, Zhejiang University; Road 38 West Lake District, Hangzhou, Zhejiang, China; email: yangxh@zju.edu.cn; A. E. Hassan, Queen's University, 25 Union St, Queen's University, Kingston, Canada; email: ahmed@cs.queensu.ca.

1049-331X/2022/08-ART76 $15.00
https://doi.org/10.1145/3508479

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Maintaining software;**

Additional Key Words and Phrases: Defect prediction, just-in-time defect prediction, empirical study, JavaScript

ACM Reference format:
Chao Ni, Xin Xia, David Lo, Xiaohu Yang, and Ahmed E. Hassan. 2022. Just-In-Time Defect Prediction on JavaScript Projects: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 76 (August 2022), 38 pages.
https://doi.org/10.1145/3508479

1 INTRODUCTION

Software defect prediction plays a crucial role in software engineering and attracts much attention from researchers. Recently, researchers have proposed many defect prediction approaches, and their effectiveness has been confirmed on various projects [43, 44, 49, 53, 86]. However, the majority of defect prediction approaches focus on identifying defect-prone entities at a coarse-grained level (i.e., class/file/module). Although these approaches can be used in some cases, their drawbacks hinder their practical application, especially for the cases with limited resources. Therefore, researchers proposed new defect prediction approaches which focus on identifying defect-prone entities at a fine-grained level (i.e., change) [16, 26, 29, 30, 39, 89]. Change-level defect prediction has attracted increasing interest in recent years, as it finely and timely helps developers to identify defect-prone entities [18, 31, 42, 70, 84].

Change-level defect prediction is widely referred to as **Just-in-Time (JIT)** defect prediction since it can identify a defect-inducing change at the check-in time. The defect-inducing changes are those which introduce one or a few defects and make software invalid [72]. Compared with coarse-grained level defect prediction, JIT defect prediction has the following advantages [34]: (1) Predicting at a fine granularity: the identified defect-inducing changes are linked to certain changes, and it hugely reduces the area of source code needed to be inspected. (2) Predicting for a concrete developer: the identified defect-inducing changes are linked to certain changes, and it quickly finds who made the modifications to those changes and then makes assignments for a developer to fix the defect. (3) Predicting at the check-in time: the changes can be timely classified as clean ones or defect-prone ones as soon as they are submitted to the code repository.

Due to the aforementioned benefits of JIT defect prediction, researchers have proposed many approaches [16, 29, 30, 34, 39] based on language-independent change-level features proposed by Kamei et al. [34], which can be classified into two groups: supervised JIT defect prediction approaches and unsupervised JIT defect prediction approaches. The effectiveness of these JIT defect prediction approaches has been verified on projects developed using Java or C++ programming languages. In practice, different projects aiming at solving specific jobs are often developed using various most suitable programming languages. JavaScript programming language is widely used for both client-side and server-side applications and becomes an extremely popular programming language according to Stack Overflow Developer Survey.[1] Therefore, software quality assurance is an important issue for JavaScript projects since its popularity among other programming languages [19, 21, 22, 55, 67]. However, whether these change-level features can effectively identify the defects of the projects developed using JavaScript programming language is unknown. Additionally, researchers conducted empirical studies on the comparison between supervised and unsupervised JIT defect prediction approaches [29, 30, 39, 89] and found that supervised approaches perform best in a holistic view when considering effort-aware performance measures

[1] https://insights.stackoverflow.com/survey/2021.

(i.e., considering inspection effort) on Java or C++ projects. However, whether supervised JIT defect prediction approaches can still perform best on JavaScript projects is also unknown. Furthermore, according to previous work [12, 61, 64, 82], different programming languages can impact not only the coding process but also the properties (e.g., source code size, the number of developers, and age/maturity) of the resulting projects. However, previously proposed 14 change-level features are programming language independent. Thus, whether programming language–specific features have an impact on identifying defect-prone changes is still unknown.

In view of the aforementioned interesting questions, in this article, we collect the top-20 most starred projects developed using JavaScript and published on GitHub,[2] propose five JavaScript-specific change-level features, and conduct a large-scale empirical study to answer the following research questions:

RQ1: How well do recently proposed effort-aware JIT defect prediction approaches perform on JavaScript projects?

We make a deep comparison between supervised (i.e., EALR [34], OneWay [16], and CBS+ [30]) and unsupervised (i.e., LT [89] and Churn [39]) effort-aware JIT defect prediction approaches on JavaScript projects when considering inspection effort and find that CBS+ always performs best among all supervised approaches. Additionally, when compared with unsupervised approaches, the supervised approach CBS+ also significantly statistically performs best in most cases.

As for Java or C++ projects, CBS+ always performs best among all supervised approaches and outperforms unsupervised JIT defect prediction approaches when considering inspection effort according to the results in previous work [29, 30].

Furthermore, we propose five JavaScript-specific change-level features and conduct a further study on the best-performing supervised approach CBS+, and find that JavaScript-specific change-level features can further improve the performance of CBS+ on identifying defect-prone changes. Therefore, language-dependent change-level features have impacts on JIT defect prediction approaches.

RQ2: What are the important features for effort-aware JIT defect prediction on JavaScript projects?

We further analyze the importance of each change-level feature on best supervised JIT defect prediction approach CBS+ and figure out the most important features (i.e., "LT," "NF," "SO," and "TC") for predicting defective changes in JavaScript projects. These features belong to the dimension of "Size," "Diffusion," and "JavaScript-specific," which indicates the importance of the three types of JIT features, especially "LT," "NF," "SO," and "TC."

As for Java or C++ projects, "NF," "FIX," and "AGE" which belong to the dimension of "Diffusion," "Purpose," and "History," respectively, are the important features for predicting defect-prone changes according to the results of preivous work [34].

Therefore, different types of change-level features have varying impacts on different projects developed by different programming languages.

RQ3: Is there an association between project-related features and the probability of a defect-prone change on JavaScript projects?

We lastly investigate the association between project-related features and the probability of a defective change using mixed effect logistic regression and find that project-related features are associated with the probability of a change to be a defect-prone one in JavaScript projects. Specifically, the seven features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective, and Forks) have the largest and statistically significant association with the probability of a defect-prone change.

[2]https://github.com/.

As for Java or C++ projects, project-related features (i.e., the number of changes, the number of developers, the number of files, the number of downloads) also have a high association with the quality of the projects according to the results in previous work [93, 94].

Therefore, project-related features do have an association with the probability of defect-prone changes on different projects.

Article contributions:

(1) We collect and label the dataset of the top-20 most popular (i.e., in terms of the number of stars) JavaScript projects using the MA-SZZ[3] algorithm since there is no JavaScript dataset available today. The dataset can be useful for future work and is publicly available on GitHub and GitLab.[4]
(2) We conduct a case study on 20 JavaScript projects with 176,902 changes to investigate the two aspects: (1) whether the programming-independent change-level features can effectively identify defect-prone changes on JavaScript projects in the context of effort-aware scenario; and (2) whether supervised JIT defect prediction approaches still have the advantages over unsupervised approaches on JavaScript projects when considering inspection effort.
(3) We firstly propose five JavaScript-specific change-level features in this article, and conduct a further study to uncover that language-dependent features have impacts on JIT prediction approach on identifying defect-prone changes.
(4) We study the importance of change-level features to effort-aware JIT defect prediction approaches on JavaScript projects. Additionally, we investigate the association between project-related features and the probability of a defect-prone change using a mixed effects logistic regression.

Article Structure. Section 2 describes the experimental dataset and design, including the studied projects, the studied change-level features, the selected defect prediction approaches, the evaluation performance measures, the data pre-processing, and the statistical analysis. Section 3 analyzes the experimental results. Section 4 compares the results obtained in this article with the results in previous work. Section 5 presents the potential threats to validity in our empirical studies. Section 6 briefly reviews the related work on JIT defect prediction. Section 7 concludes this article and presents future work.

2 EMPIRICAL STUDY SETUP

In this section, we introduce our empirical study settings. We firstly present the studied JavaScript projects. We secondly introduce the change-level features we used in our research. Following that, previously proposed approaches, the evaluation measures, data pre-processing, and statistical analysis are subsequently presented.

2.1 JavaScript Projects

Projects selection. We use *JavaScript* as the keyword to search repositories on *All GitHub* (i.e., 729,631 repository results returned). Then, we filter these repositories by *JavaScript* language (i.e., 409,432 repository results returned). Finally, we sort these queried projects by *most stars* in descending order. To choose the most suitable projects, we set up the following inclusion criteria: (1) the ratio that files ending with ".js" account for the whole files in a project is no less than 90%; (2) projects should not be one of these types: tutorials, algorithm implementations written in

[3]https://github.com/danielcalencar/ma-szz.
[4]https://github.com/jacknichao/JIT-on-JavaScript-projects.git or https://gitlab.com/shared-materials/JIT-on-JavaScript-projects/.

JavaScript, the experience of a job interview, and collections of useful code snippet since they are not conceptually software projects according to the definition.[5] After filtering by these criteria, we pick up the top-20 most starred popular projects on January 10, 2020, which means we cloned the repositories of these selected projects on January 10, 2020, and the time period of all changes in each selected project starts from the time it was created on GitHub and ends by January 10, 2020. A summary of these projects can be seen in Table 1.

In Table 1, the first column lists the name of selected JavaScript projects. The following two columns correspond to the number of stars and the number of forks for each project. One star of a project means there is one user on GitHub who is interested in the project. One fork operation of a project means there exists one user on GitHub making a copy of such a project into his/her software repository and may intend to make some contributions to this project in the future. Thus, the more number of stars or forks a project has, the more popular such a project is. The "JS Ratio" column indicates the ratio that files ending with ".js" account for the whole files in a project. The next three columns show the number of branches, the number of changes, and the number of defective changes, respectively. Branches are effectively pointers to a snapshot of changes in a project. A branch in a project means a part of the everyday development process or a dump of function collection. Following that, the defective ratio in each project is listed in column "Def Ratio." In the next two columns, the number of files and LOC are presented in sequence. After that, the number of contributors, the median number of code churn (i.e., LA+LD), and the mean number of code churn are presented. For the last column, a short description of each project is presented. In a whole view, these selected projects belong to different application domains (e.g., library, compiler, client-side application, framework), vary in size (7,509–396,587 LOC), vary in the number of contributors (191–1,362), vary in the number of branches (3–994), vary in the number of files (79–8,997), vary in the number of forks (1,460–24,164), have different popularity (26,94–141,536), and cover a long period of time (1,226–52,381 days). Therefore, these selected projects are representative projects in all JavaScript projects.

Commit analysis. We analyze all commits in our experimental projects on how many files a commit may change. We list the results in Table 2. In Table 2, the columns represent the range of numbers of files a commit changes, and the rows represent the numbers and percentage of the commit in each range (e.g., 0 Files, 1–10 Files). According to the results, we find that only 17% of all commits do not modify any file (e.g., merge commit), while 83% of all commits modify at least one file. On average, for our studied 20 JavaScript projects, a commit modifies 4.88 files.

Then, we conduct further analysis of the time interval between two adjacent commits. We statistic all commits on studied JavaScript projects, and group time interval into days. We list the results in Table 3.

According to the results shown in Table 3, we can find that the majority of the time interval between two commits belongs to the group 0–10 days. Then, we further analyze the commits in the group 0–10 days and divide them into smaller groups (i.e., one day, one group). We illustrate the results in the form of a histogram.

As shown in Figure 1, we find that the majority of the time interval between two adjacent commits is 0 days (i.e., less than 24 hours), which means that many changes in a project are submitted instantly to the corresponding GIT repository.

Therefore, the JIT defect prediction approach is quite necessary to identify defect-prone changes for JavaScript projects because of the high frequency of commit submission and the relatively small numbers of the modified file of a commit.

[5] https://en.wikipedia.org/wiki/Software_development.

Table 1. Summary of the Studied JavaScript Projects

Project	# Stars	# Fork	JS Ratio	# Branches	# Changes	# Defective	Def Ratio	# File	# LOC	#Contributors	# Med_size	# Mean_size	Short Introduction
vue	141,536	20,388	97.7%	33	6,156	2,081	33.8%	432	168,808	391	19	289	A progressive, incrementally adoptable JavaScript framework for building UI on the web.
react	131,183	24,164	95.2%	28	14,080	3,694	26.2%	881	203,624	1,511	1	219	A declarative, efficient, and flexible JavaScript library for building user interfaces.
axios	60,819	4,919	91.7%	9	888	201	22.6%	79	7,509	191	1	107	Promise-based HTTP client for the browser and node.js.
three.js	52,412	19,754	99.2%	4	28,196	6,447	22.9%	1,208	396,587	1,362	8	473	JavaScript 3D library.
jquery	51,761	18,224	93.5%	4	7,672	4,265	55.6%	178	88,971	347	9	226	jQuery JavaScript Library.
webpack	49,422	6,221	99.3%	49	8,961	1,770	19.8%	3,399	78,840	611	2	50	A bundler for JavaScript and friends. Packs many modules into a few bundled assets.
material-ui	47,975	10,700	98.3%	4	10,800	2,231	20.7%	6,842	148,396	1,620	0	82	React components for faster and easier web development.
express	44,310	7,442	100.0%	10	5,824	920	15.8%	152	20,655	287	3	32	Fast, unopinionated, minimalist web framework for node.
Chart.js	43,956	9,657	99.6%	3	2,775	661	23.8%	215	41,689	338	4	454	Simple HTML5 Charts using the <canvas> tag.
moment	41,481	6,231	99.8%	20	3,781	757	20.0%	583	182,540	588	5	310	Parse, validate, manipulate, and display dates in JavaScript.
meteor	41,121	5,034	98.0%	994	27,805	5,870	21.1%	1,193	272,424	501	7	80	A JavaScript App Platform.
lodash	39,648	4,115	100.0%	6	8,314	1,280	15.4%	702	35,102	320	16	253	A modern JavaScript utility library delivering modularity, performance and extras.
yarn	36,047	2,199	98.7%	58	2,787	1,110	39.8%	554	57,350	527	5	55	Fast, reliable, and secure dependency management.
babel	23,466	3,570	99.7%	16	12,519	3,284	26.2%	8,997	143,864	816	3	116	A compiler for writing next generation JavaScript.
parcel	32,055	1,460	96.9%	79	2,458	775	31.5%	743	27,073	216	14	78	Blazing fast, zero configuration web application bundler.
anime	31,418	2,323	100.0%	8	846	169	20.0%	19	11,130	35	17	136	JavaScript animation engine.
serverless	30,440	3,368	97.2%	20	10,324	2,334	22.6%	357	63,327	700	1	122	Build web, mobile, and IoT applications with serverless architectures using AWS Lambda, Azure Functions, Google CloudFunctions, and more!
Ghost	30,147	6,519	92.0%	9	9,800	2,467	25.2%	888	115,214	386	0	87	The most popular headless Node.js CMS for professional publishing.
hyper	30,067	2,462	99.9%	5	1,287	371	28.8%	100	148,974	237	2	386	A terminal built on web technologies.
pdf.js	26,946	6,413	96.6%	3	11,629	2,331	20.0%	273	104,007	364	2	62	PDF Reader in JavaScript.

Table 2. Numbers of Files Modified by One Commit

	0 Files	**1–10 Files**	**11–20 Files**	**21–30 Files**	**31–40 Files**	**41–50 Files**	**51– Files**
# Commit	35,002	163,197	5,584	1,631	824	505	2,162
% Percentage	**17%**	**83%**					

Table 3. Days Interval between Two Adjacent Commits

Days	**0–10D**	**11–20D**	**21–30D**	**31–40D**	**41–0D**	**51–60D**	**61–70D**	**71–80D**	**81–90D**	**91–100D**	**101–200D**
# Commit	176,558	230	60	25	9	6	5	2	3	1	3

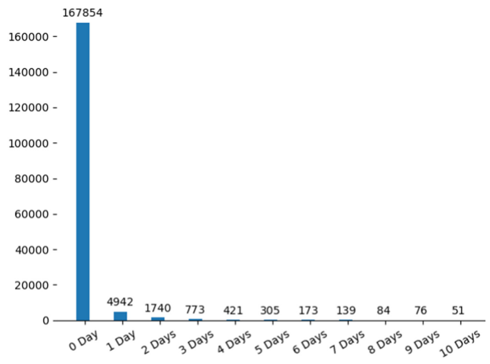

Fig. 1. Numbers of commit in different time intervals.

Data labeling. In practice, labeling historical change as defect inducing or clean is very challenging as it requires a considerable amount of manual effort (e.g., manually analyzing thousands of LOC) and in-depth domain knowledge of the project (which is only feasible by contacting the core developers of a project). Therefore, Sliwerski et al. [72] initially developed an approach named SZZ to automatically identify defect-inducting change in each project repository.

The SZZ algorithm can be organized in two subsequent phases: defect-fixing change identification phase and defect-inducing change identification phase.

Phase 1: Defect-fixing change identification. SZZ firstly searches these changes which aim at fixing previous defects by leveraging some special characteristics. In particular, SZZ searches some keywords (e.g., "bug," "fix," "wrong," "error," "fail," "problem," and "patch") in each change message to mark whether a change is a defect-fixing change or not.

Phase 2: Defect-inducing change identification phase. Firstly, for each candidate bug-fixing change, SZZ uses *git diff* command to identify all changes that previously make modifications to the same lines of code. These modified lines of code are identified as those that cause defects. Secondly, SZZ uses the *git blame* command, a powerful tool which can show what revision and author last modified each line of a file, to figure out the last changes which introduce those lines that finally cause the defects in bug-fixing change. Finally, these changes are labeled as defect-introducing changes, and the other changes are labeled as clean changes.

However, previous studies observed that the original SZZ might cause a large amount of noise, which results in mislabeled changes [10, 36, 51]. In particular, the original SZZ [72] simply considers all lines modified by bug-fixing changes as buggy lines. It uses the built-in *annotate* command in version control systems to trace back through the change history.

To reduce the mislabeled changes, Kim et al. [36] proposed an improved SZZ variant built on top of the original SZZ [72]. It discards all non-semantic lines (e.g., blank/comment lines) and those

Table 4. The MA-SZZ's Precision and Recall on Sampled Changes

Project	Prec.	Rec.	Project	Prec.	Rec.
Chart.js	100%	92%	meteor	95%	95%
Ghost	92%	100%	moment	85%	100%
anime	95%	100%	parcel	94%	100%
axios	91%	95%	pdf.js	95%	100%
babel	96%	96%	react	92%	96%
express	93%	93%	serverless	96%	100%
hyper	89%	96%	three.js	86%	100%
jquery	95%	100%	vue	97%	97%
lodash	93%	93%	webpack	90%	94%
material-ui	90%	95%	yarn	95%	100%
Avg Prec.	93%		Avg Rec.	97%	

lines involving format modifications (e.g., modifications to code indentation) [36]. This implementation of the SZZ variant applies the annotation graph to trace the change history. Notice that annotation graph is a powerful tool for tracing the evolution of lines of code along the code history as proposed by Zimmermann et al. [95]. For ease of presentation, we refer to the SZZ variant proposed by Kim et al. as **Annotation Graph SZZ** (a.k.a., **AG-SZZ**).

Subsequently, Da Costa et al. [10] proposed another improved SZZ variant which is built on top of Kim et al.'s AG-SZZ. In this variant, it improves AG-SZZ by mainly focusing on how to further mitigate the mislabeled noise caused by branch or merge operation on changes and property modification on changes. Da Costa et al. referred to their SZZ variant as **Meta-change Aware SZZ** (a.k.a., **MA-SZZ**). In this article, as suggested by Fan et al.'s work [14], we use MA-SZZ to label the changes of JavaScript projects.

To verify the effectiveness of MA-SZZ in our study, we perform a manual analysis of MA-SZZ in labeling changes. Because of the requirement of a large amount of manual effort and domain knowledge, for each project, we randomly sample 100 changes from each project and keep the class distribution the same as the original project. That is, we totally sample 2,000 changes. Then, the first two authors are required to separately determine whether the sampled changes are labeled correctly or not. Notice that both authors have many years of programming experience in JavaScript. Finally, the two authors compare their results to identify any disagreements on the labeled results. We find that, on average, in about 5.1% changes, the two authors have different decisions. It also means the two authors share the same decision on about 94.9% of the 2,000 changes. For those changes with decision conflict, the two authors further discuss whether the change contains defects or not.

In Table 4, we show the precision and recall of MA-SZZ on these sampled changes in each project. According to the results shown in the table, we notice that MA-SZZ achieves a precision of 85%–100% and a recall of 92%–100% across the top-20 JavaScript projects. Therefore, it averagely achieves a precision of 93% and a recall of 97% on the 2,000 sampled changes, and our manual analysis, to some extent, verifies the effectiveness of MA-SZZ though it cannot achieve 100% precision and 100% recall on labeling changes.

2.2 Change-Level Features

In our study, we use the 14 change-level features which were used by prior work [16, 29, 30, 34, 39, 89]. The change-level features can be classified into five dimensions according to Kamei et al. [34]: diffusion, size, purpose, history, and experience. Table 5 summarizes these change-level features

Table 5. Summary of Change-Level Features

Dimension	Feature	Description
Diffusion	NS	Number of subsystems touched by the current change.
	ND	Number of directories touched by the current change.
	NF	Number of files touched by the current change.
	Entropy	Distribution across the touched files.
Size	LA	Lines of code added by the current change.
	LD	Lines of code deleted by the current change.
	LT	Lines of code in a file before the current change.
Purpose	FIX	Whether or not the current change is a defect fix.
Histroy	NDEV	The number of developers that changed the files.
	AGE	The average time interval (in days) between the last and the change over the files that are touched.
	NUC	The number of unique last changes to the files.
Experience	EXP	Developers experience, i.e., the number of changes.
	REXP	Recent developer experience, i.e., the total experience of the developer in terms of changes, weighted by their age,
	SEXP	Developer experience on a subsystem, i.e., the number of changes the developer made in the past to the subsystems.
JavaScript-specific	HtmlCss	Sum of HTML and CSS operations included in the current change.
	Strict	Switch between opening and closing "strict" mode by the current change.
	BDom	Sum of BOM (Browser Object Model) and DOM (Document Object Model) operations included in the current change.
	SO	Number of special operators (i.e., !== or ===) in JavaScript programming language included in the current change.
	TC	Number of type check operations included in the current change.

including their short names, descriptions, and grouping into several dimensions. To make our article self-contained, we briefly introduce these features below. A more detailed description can be found in Kamei et al.'s work [34].

The diffusion dimension includes NS, ND, NF, and Entropy, which characterize the distribution of a change. Kamei et al. [34] stated that a highly distributed change is more likely to be a defect-inducing change. The size dimension is composed of LA, LD, and LT which are used to characterize the size of a change. It is believed that a larger change is expected to have a higher likelihood of being a defect-inducing change [47, 71]. The purpose dimension has only one feature: FIX. There is a belief [90] that a defect-fixing change is more likely to introduce a new defect. The history dimension is composed of NDEV, AGE, and NUC. Previous studies found that a defect is more likely to be introduced by a change if the touched files have been modified by more developers, by more recent changes, or by more unique last changes [11, 20, 24, 40]. The experience dimension, including EXP, REXP, and SEXP, characterizes a developer experience based on the number of changes made by the developer in the past. There is a belief that a change made by a more experienced developer is less likely to introduce defects [46].

Different programming languages have different characteristics. These aforementioned language-independent change-level features are only verified on projects developed by Java or C++; both of them are strongly typed programming languages. However, JavaScript is one of the most popular weakly typed programming languages. Therefore, in addition to the 14 features from four dimensions, we also propose five JavaScript-specific features (i.e., *HtmlCss*, *Strict*, *BDom*, *SO*, and *TC*) and group them into the JavaScript-specific dimension. We introduce these features in detail as follows.

HtmlCss. HtmlCss represents HTML and CSS. The front end of a web application consists of three important parts: JavaScript, HTML, and CSS. Each of these parts has different functions [79]. That is, HTML defines the structure of web pages, JavaScript defines the behavior of web pages, and CSS defines the layout of web pages. A web application will be considered as a strong coupled application if a JavaScript file frequently and directly contains a few scripts which interact with HTML script or CSS script. Therefore, for coupled applications, changes to one part of the application often inadvertently impact unrelated parts downstream [83], which will cause unexpected errors. Although the three important parts naturally need to interact with each other, there are many different state-of-the-art development approaches that can decouple them. We calculate the number of times that HTML and CSS are defined in JavaScript files included in a change by identifying their structure (e.g., <>, </>, and $selector\{style\_name : style\_value, \ldots\}$), which indicates the frequency of interaction between them.

Strict. Strict mode is a special execution setting in JavaScript programming language, which indicates the codes must be executed under strict conditions. In such a mode, scripts will not be allowed to execute if they try to use undeclared variables. We can add "use strict" to or remove "use strict" from the head of a script or the inner of a function to turn on or turn off the strict mode. "use strict" is not a simple statement, but a literal expression, which impacts both syntax and runtime behavior.[6] Switching the strict mode means an exactly same JavaScript script may have different behavior and therefore cause syntax or runtime errors [56, 59]. We calculate the number of times "use strict" is used by searching the modified context in JavaScript files in a change.

BDom. **BDom** is the abbreviation of BOM (**Browser Object Model**) and DOM (**Document Object Model**). BOM enables JavaScript to interact with the browser, while DOM enables JavaScript to access all the elements (e.g., <a>, <p>, <div>, and so on) of a HTML document. BOM and DOM can help to adjust display environment, change browsing behavior, detect browser capability, determine whether or not to use user agent, address compatibility issues, access HTML elements, and so on. BOM contains a few attributes, including *window, location, navigator, screen, and history*, while DOM contains one major attribute: *document*. Therefore, these keywords are good indicators for the usage of BOM and DOM. We conjecture that JavaScript files directly interacting with BOM and DOM without a high-quality third-party package (e.g., jQuery[7]) are more likely to introduce defect-prone changes. Therefore, we sum up the number of times these keywords (i.e., *window, location, navigator, screen, history, and document*) are used in JavaScript files in a change.

SO. **SO** is short for **Special Operators**, which represents expressions unique to JavaScript programming language compared with strongly typed programming language (e.g., C++ and Java). Although special operators (e.g., !==, ===, and *void*) provide more useful functionality, they may also cause incorrect understanding. Take "*LE*!==*RE*" as an example; "!==" is a binary operation, which means the left element (i.e., *LE*) and the right element (i.e., *RE*) are not absolutely equal. That is, *LE* is not equal to *RE* in terms of variable data or variable type, or both of them. In our article, we consider the above-mentioned three operators since we think using these operators may lead to logic errors in JavaScript scripts, especially for those developers who also have experience in other strongly typed programming languages (e.g., C++ or Java). To calculate the value of "SO" in a change, we search the number of times the three operators are used in the change.

TC. **TC** is short for **Type Check**. Different from Java or C++ programming language, JavaScript is a weakly typed programming language. Improper usage of variable types will cause the program to run abnormally and even produce unintended results. Prior work [60] stated that although the rules for coercing types are well defined [13] in JavaScript programming language, even expert JavaScript developers struggle to fully comprehend the behavior of some code corresponding to type check. Therefore, incorrect type checking may result in defect-prone codes. There are three

[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.
https://www.w3schools.com/js/js_strict.asp.
[7]https://jquery.com/.

categories of type for JavaScript: data type (i.e., *string*, *number*, *Boolean*, and so on), object type (i.e., *Object*, *Date*, and *Array*), and null object (e.g., *null* and *undefined*). Different types of variables can only be checked correctly with different operations. For example, we use "*typeof*" to check data typed variables, while using "*instanceof*" to check object typed variables. Therefore, we search the number of two keywords (i.e., "*typeof*" and "*instanceof*") to calculate the usage of type check operation.

2.3 Selected Approaches

We choose five state-of-the-art effort-aware JIT defect prediction approaches and three classical effort-unaware defect prediction approaches as our candidate approaches for different purposes in our empirical study.

2.3.1 Effort-Aware Approaches. Recently, many effort-aware JIT defect prediction approaches have been proposed [16, 29, 30, 34, 39]. These approaches can be classified into two groups: supervised approaches and unsupervised approaches. Supervised approaches are widely used previously and are often expected to have a better performance since a lot of labeled data is used as training data. However, a sufficient amount of labeled data for newly started projects can be hard to obtain. Thus, unsupervised approaches are gradually proposed and receive lots of attention since these approaches are simple to implement and achieve comparable performance compared with supervised approaches [89]. The comparison of the supervised approaches vs. unsupervised approaches triggers heated discussions in the literature [16, 29, 30]. We totally consider five approaches including three supervised approaches (i.e., EALR [34], OneWay [16], and CBS+ [30]) and two unsupervised approaches (i.e., LT [89] and Churn [39]) as our candidate approaches since their effectiveness on Java or C++ projects has been confirmed. We briefly introduce these approaches to make our article self-contained in the following paragraphs.

Supervised Approaches.

EALR. The first approach, **EALR**, short for **Effort-Aware Linear Regression**, is proposed by Kamei et al. [34]. They used a regression model to address effort-aware defect prediction issues, in which LOCs (i.e., lines of code) are used as the proxy of code inspection effort. For training data, they convert the defect label $Y(x)$ into defect density $R_d = \frac{Y(x)}{Effort(x)}$, where $Y(x)$ is 1 if the change is defect inducing or 0 otherwise, and $Effort(x)$ indicates the amount of effort required by the change (i.e., the number of lines modified by a change). Then, EALR tries to learn the relationship between change-level features and defect density R_d. For testing data, EALR predicts R_d of each change and prioritizes the changes based on R_d in descending order.

OneWay. The second approach, OneWay, is proposed by Fu and Menzies [16]. OneWay, inspired by a simple unsupervised approach proposed by Yang et al. [89], aims at using supervised training data to remove all but one of the Yang et al. predictors and then applying this trained learner on the testing data. OneWay has two benefits: one is that it can fully use the information of labeled data, and another is that it can sharply decrease the number of unsupervised approaches. In particular, for training data, OneWay builds simple unsupervised models for each change-level feature, and evaluates those models in terms of a given evaluation measure. Then, the best-performing model built on a specific feature f will be obtained. For testing data, OneWay builds an unsupervised approach based on f and prioritizes changes based on $1/f$ in descending order.

CBS+. The third approach, CBS+, an improved version of **CBS** (**Classify Before Sorting**), is proposed by Huang et al. [30] and based on the observation that the relationship between the change features and defect density may be non-linear. In addition, Koru et al. [37] found that smaller modules are proportionally more defect-prone and should be inspected first. Thus, to fully leverage the advantages of supervised approach and deeply benefiting from Koru et al.'s findings, CBS+ assumes that among all changes classified to be defect-prone, small ones should be inspected first. In particular, for training data, CBS+ builds a classifier with logistic regression on the pre-processed training data (e.g., addressing imbalance, data normalization). For testing data, the same pre-processing will be applied to testing data before predicting by the classifier. Then, these changes will be classified as defect-prone or clean based on a specified threshold (e.g., 0.5). After that, the changes classified as defect-prone are sorted ahead of changes classified as clean changes. Finally, CBS+ separately sorts defect-prone changes and clean changes in descending order of defect density of each change.

Unsupervised Approaches. For JIT defect prediction, unsupervised approaches aim at figuring out a best change-level feature for sorting on the target projects. Formally, for a best sorting feature F, an unsupervised approach will sort the testing changes in descending order based on $R(c) = 1/F(c)$, where c represents a specific change, $F(c)$ is the value of feature F, and $R(c)$ is the risk value predicted by the unsupervised approach which indicates the probability of a change to be a defect-prone one. This is mainly due to a prior finding that indicates that smaller modules are proportionally more defect-prone and should be inspected first [37, 75]. Until now, two unsupervised approaches LT proposed by Yang [89] and Code Churn (referred to as Churn for easy presentation) proposed by Liu et al. [39] have been proposed.

LT. The fourth approach, LT, is proposed by Yang et al. [89]. They used feature LT representing LOC in a file before the current change as the best feature F to build an unsupervised approach, and its effectiveness in terms of Recall has been confirmed.

Churn. The fifth approach, Churn, is subsequently proposed by Liu et al. [39]. They used code churn representing the sum of LA and LD modified by the current change as the best feature F to build an unsupervised approach and its effectiveness has also been confirmed.

2.3.2 Effort-Unaware Approaches. Software defect prediction (e.g., file-level and change-level defect prediction) has received much attention, and many approaches have been proposed based on a few popular and effective classification models (e.g., logistic regression) [6, 29, 34, 52, 53, 77, 80, 86]. These approaches are used to evaluate the effectiveness of models without considering inspection efforts such as budget, time, importance, or human-resource.

In our study, we want to investigate whether the programming language–independent change-level features can effectively identify the defects in JavaScript projects in the scenario of an effort-unaware setting. We totally consider three widely used classical classifiers (Logistic Regression, Naive Bayes, and Random Forest) [5, 6, 52, 53, 58, 80, 81, 86] since their effectiveness in identifying defects in software projects without considering inspection effort. We briefly introduce these classifiers as follows: **Logistic Regression** (**LR**) [9], a linear regression model, estimates the likelihood of a classification with logistic function. For classification issues, LR chooses the class with the highest likelihood. **Naive Bayes** (**NB**) [65] estimates a score for each class based on an application of Bayes law. **Random Forest** (**RF**) [4] is composed of many random trees. Usually, each random tree is a decision tree (e.g., C4.5).

2.4 Evaluation

In this section, we introduce our evaluation plan and performance measures that we will use.

2.4.1 Experiment Setting. To evaluate the performance of the studied approaches, following previous studies [29, 89], we consider a 10-fold time-aware validation setting which makes sure that the changes used for testing are always submitted later than the changes used for training. In particular, for a given project, all changes that happened in this project will be sorted in ascending order chronologically. Then, these changes will be divided into approximately 12 equal folds and numbered as fold 0 to fold 11. That is, fold 0 contains the earliest submitted changes and fold 11 contains the latest submitted changes. For each fold i ($i \in [1, 10]$), the training data includes all the changes coming from fold 0 to fold i-1. We calculate 10 performance measures for each method on fold 1 to fold 10. Notice that, we don't consider the changes in fold 0 and fold 11 since they don't satisfy the requirement of the SZZ algorithm. The SZZ algorithm can only identify the change which has parent change and its child changes have been marked as fixed changes. Therefore, we remove the changes in fold 0 since they are the ancestor of subsequent changes, and we remove the changes in fold 11 since they are the latest change and may not be correctly labeled.

On the whole, we adopt such a time-aware experimental setting for two reasons. First, during the process of project development, the changes happened on projects are submitted to the project version control system (e.g., git) in chronological order. Thus, these changes have a definite chronological relationship with each other. Second, for a given project, the amount of changes is increasing gradually as time goes on, which is in line with objective facts. That is, we barely have changes at the initial stage of a project. We, however, can obtain many changes after a few months or years of development or maintenance.

2.4.2 Performance Measures. In this section, we introduce 10 performance measures that can be divided into two groups: effort-aware and effort-unaware.

Effort-Aware Performance Measures. This group considers code inspection effort and includes six performance measures: P_{opt}, *Precision@20%*, *Recall@20%*, *F1-measure@20%*, *PCI@20%*, and *IFA* [29, 30]. Suppose we totally have a dataset with M changes and N defective changes. After inspecting 20% of the total modified lines of code, suppose we inspected m changes and observed n defective changes. Then these evaluation measures can be defined as follows.

Precision@20%: the proportion of inspected defective changes over all the inspected changes, which is calculated as n/m.

Recall@20%: the proportion of inspected defective changes over all the actual defective changes, which is calculated as n/N.

F1-measure@20%: a summary measure that considers both *Precision@20%* and *Recall@20%*, which is calculated as $\frac{2\times Precision@20\%\times Recall@20\%}{Precision@20\%+Recall@20\%}$.

PCI@20%: the proportion of the number of inspected changes over all changes, which is calculated as m/M.

P_{opt}: it is based on the concept of the Alberg diagram [2] which shows the relationship between the Recall obtained by a prediction model and the inspection cost (e.g., the total modified LOC of changes) for a specific prediction model. To compute such a measure, two additional prediction models are required: the optimal model and the worst model. In the optimal model and the worst model, changes are respectively sorted in decreasing and ascending order according to their actual defect densities. A good prediction model is expected to outperform the random model and approximate the optimal model. For a given prediction model m, the P_{opt} can be calculated as $P_{opt}(m) = 1 - \frac{Area(optimal)-Area(m)}{Area(optimal)-Area(worst)}$; $Area(M)$ represents the area under the curve corresponding to the model M [30, 89].

IFA: is the number of Initial False Alarms encountered before we find the first defective change, which can be calculated by the number of inspected clean changes before finding the first defect-prone change.

Effort-Unaware Performance Measures. This group hardly considers code inspection effort and includes four performance measures: *F1-measure* [27, 74, 86], *Recall*, *AUC* [28, 52, 53, 66], and *PFA* [17, 45, 54, 80]. There are four possible outcomes for a change in a testing data: a change can be classified as defective when it is truly defective (**true positive**, **TP**); it can be classified as defective when it is actually non-defective (**false positive**, **FP**); it can be classified as non-defective when it is actually defective (**false negative**, **FN**); or it can be classified as non-defective and it is truly non-defective (**true negative**, **TN**). Therefore, based on the four possible outcomes, *Recall*, *PFA*, and *F1-measure* can be defined as follows:

Recall: the proportion of defective instances that are correctly labeled: $Recall = \frac{TP}{TP+FN}$.

F1-measure : a summary measure that combines both precision and recall. It not only evaluates the tradeoff between precision (i.e., $P = \frac{TP}{TP+FP}$ increase) and recall (i.e., $R = \frac{TP}{TP+FN}$) reduction, but also evaluates the opposite way: $F1\text{-}measure = \frac{2\times P\times R}{P+R}$.

AUC: the area under the **receiver operating characteristic** (**ROC**) curve [23], which is a 2D illustration of the TP rate on the y-axis versus the FP rate on the x-axis. The ROC curve is obtained by varying the classification threshold over all possible values, separating clean and defect-prone predictions. A best-performing predictor achieves an AUC value close to 1. The ROC analysis is robust in the case of imbalanced class distributions and asymmetric misclassification costs. It also represents the probability that a method will rank a randomly chosen defective module higher than a randomly chosen not defective one.

PFA: the probability of false alarm is defined as the ratio of FPs to all non-defective instances: $PF = \frac{FP}{FP+TN}$:

Notice that, for *Precision@20%*, *Recall@20%*, *F1-measure@20%*, P_{opt}, *F1-measure*, *Recall*, and *AUC*, the larger these measures' value, the better the corresponding approaches' performance. The improvement of the best approach A over the other approach B can be calculated as $\frac{A-B}{B} \times 100\%$. For *IFA*, *PCI@20%*, and *PFA*, the smaller these measures' value, the better the corresponding approaches' performance. Thus, the improvement of the best approach A over the other approach B can be calculated as $\frac{A-B}{A} \times 100\%$, which means the ratio of decrease.

2.5 Data Pre-Processing

Prior work [76] highlighted the side-effect of correlated features on models' performance. As suggested by Harrell [63], we primarily filter correlated features and redundant features before investigating the performance of models on JavaScript projects.

Filtering correlated features. For correlated features, we calculate the correlation between each pair of features by Spearman rank test [91]. Then, we apply *Hmisc*, an R tool-kit[8] for variable clustering analysis, to cluster the correlated features. During the correlation analysis, we use 0.8 as the threshold suggested by Li et al. [38]. In particular, if the correlation coefficient of two features is larger than 0.8, the two features are known as collinearity features and one of them should be removed. For all pairs of correlated features in a specific project, we first remove the feature, which is highly correlated with many other features. Then, for other pairs of correlated features, we take the advice suggested by Kamei et al. [33, 34] and Li et al. [38] and keep all the ones which have the advantages for easy understanding. Thus, for the interpretation of the JIT defect prediction approaches, when two features are correlated, we keep the one which is easier to understand and remove the other one.

Filtering redundant features. After filtering correlated features, we further filter redundant features. Redundant features represent a feature that can be predicted by the combination of other features. We filter these redundant features since they not only have no contribution to the JIT

[8]https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf.

defect prediction approach but also increase the training time for the approach building. We apply the *redun*, a function implemented in *rms* R tool-kit,[9] to identify these redundant features.

Notice that for practical application, testing data is not available when building a prediction model especially for time-aware JIT defect prediction setting. Thus, we only conduct preprocessing operations on training data to filter correlated features and redundant features.

2.6 Statistical Analysis

To check the statistical significance of the performance difference of two different methods in a 10-fold time-aware validation setting, we run the Wilcoxon signed-rank test [85] with a Bonferroni correction [1]. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test on the performance measures, while the Bonferroni correction is used to counteract the problem of multiple comparisons. For all the statistical testing, the null hypotheses are that there is no difference between two defect prediction approaches, and the significance level α is set to 0.05. If the p-value is smaller than 0.05, we reject the null hypotheses; otherwise, we accept the null hypotheses.

Additionally, we also use Cliff's delta (δ) [8], which is a non-parametric effect size measure that quantifies the amount of difference between the two approaches. The range of Cliff's delta is $[-1,1]$. $|\delta|$ equals to 1 indicates the absence of overlap between two approaches. It means all data from one group are higher than that from the other group, and vice versa. $|\delta|$ equals to zero means that the two approaches are overlapping completely. We consider $|\delta|$ which are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474, and above 0.474 as "Negligible (N)," "Small (S)," "Medium (M)," "Large (L)" effect size, respectively, following [8].

3 EMPIRICAL STUDY RESULTS

In this section, we present the results for the three research questions:

- **RQ1: How well do recently proposed effort-aware JIT defect prediction approaches perform on JavaScript projects?**
- **RQ2: What are the important features for effort-aware JIT defect prediction on JavaScript projects?**
- **RQ3: Is there an association between project-related features and the probability of a defect-prone change on JavaScript projects?**

3.1 RQ1: How Well do Recently Proposed Effort-Aware JIT Defect Prediction Approaches Perform on JavaScript Projects?

Motivation. In practice, we often have lots of work to do with limited resources such as time, human-resource, or budget. Thus, effort-aware performance measures should be preferentially taken into consideration. Many effort-aware JIT defect prediction approaches have been proposed, which can be divided into two categories: supervised approaches (e.g., EALR [34], OneWay [16], and CBS+ [30]) and unsupervised approaches (e.g., LT [89] and Churn [39]).

For supervised approaches, the comparison of the three approaches has been conducted on six open source projects, and the effectiveness of CBS+ has been confirmed [30]. However, whether CBS+ can perform best on JavaScript projects is unknown. Additionally, two unsupervised approaches were proposed by Yang et al. [89] and Liu et al. [39], and their results stated that unsupervised JIT defect prediction approaches can obtain comparable performance with supervised

[9]https://cran.r-project.org/web/packages/rms/rms.pdf.

approaches on six open source projects. However, the comparison of the supervised vs. unsupervised approaches on JavaScript projects is unknown.

Thus, we not only want to figure out the best supervised JIT defect prediction approach on JavaScript projects, but also try to display how well supervised approaches perform when compared with unsupervised approaches.

Moreover, the 14 change-level features proposed by Kamei et al. [34] are programming language–independent features, which can help to build a JIT defect prediction model on identifying defect-prone changes. However, whether programming language–dependent change-level features (e.g., JavaScript-specific change-level features) can further improve the performance on identifying defect-prone changes is still unknown.

Method. To address the aforementioned issues, we investigate two specific sub-questions:

- **Question 1**: How well do recently proposed effort-aware JIT defect prediction approaches perform on JavaScript projects using 14 prior proposed programming language–independent change-level features?
- **Question 2**: Can JavaScript-specific change-level features improve the performance of the effort-aware JIT defect approach on identifying defect-prone changes?

In Question 1, first, we implement the five studied JIT defect prediction approaches introduced previously: EALR [34], OneWay [16], CBS+ [30], LT [89], and Churn [39]. Second, we execute all five approaches on JavaScript projects considering six effort-aware performance measures after two data pre-processing steps (i.e., filtering correlated features and filtering redundant features). Notice that for the two unsupervised approaches, they do not need to build a prediction model with the help of labeled changes on the training data, but directly predict whether a change is defect-prone or not using one or a few change-level features on the testing data. More details about LT and Churn can be found in Section 2.3. Therefore, for a fair comparison, we only execute two unsupervised approaches on testing data. Third, we compare the performance of three supervised approaches and figure out the best one. Then, we make a comparison between the best-performing supervised approach and two unsupervised approaches.

In Question 2, we explore the performance difference between the effort-aware JIT defect prediction approach using 14 original change-level features and the one using the combination of 14 original change-level features and five JavaScript-specific change-level features.

Results for Question 1:

How well do recently proposed effort-aware JIT defect prediction approaches perform on JavaScript projects using 14 prior proposed programming language–independent change-level features?

Table 6 presents the average performance of three supervised effort-aware JIT defect prediction approaches in terms of six effort-aware performance measures. Table 7 shows the comparison between the best-performing supervised approach (i.e., CBS+) and the two unsupervised approaches in terms of six effort-aware performance measures. The statistical results are shown in the bottom few rows of each table, and the best approaches are listed in the last row.

By analyzing the comparison among supervised approaches, we obtain the following observations:

(1) CBS+ statistically significantly outperforms EALR and OneWay with a medium or large effect size in terms of six effort-aware performance measures in most cases. In particular, CBS+ improves EALR and OneWay by 108% and 65% in terms of *F1-measure*@20%, by 39% and 63% in terms of *IFA*, by 12% and 26% in terms of *PCI@20%*, by 19% and 13% in terms of P_{opt}, by 93% and 80% in terms of *Precision@20%*, and by 60% and 28% in terms of *Recall@20%*, respectively.

Table 6. The Average Performance of the Three Newly Proposed Supervised Effort-Aware JIT Defect Prediction Approaches on JavaScript Projects in Terms of Six Studied Performance Measures

Project	F1-measure@20%↑			IFA↓			PCI@20%↓			Popt↑			Precision@20%↑			Recall@20%↑		
	CBS+	EALR	OneWay	CBS+	EALR	OneWay	CBS+	EALR	OneWay	CBS+	EALR	OneWay	CBS+	EALR	OneWay	CBS+	EALR	OneWay
Chart.js	**0.46**	0.24	0.28	**6.3**	27.1	75.2	**0.39**	0.59	0.77	0.65	0.59	**0.69**	**0.44**	0.17	0.18	0.56	0.43	**0.65**
Ghost	**0.60**	0.37	0.31	**3.8**	32.9	12.9	**0.22**	0.23	0.24	**0.67**	0.65	0.51	**0.65**	0.45	0.35	**0.57**	0.35	0.30
anime	**0.39**	0.29	0.20	**6.7**	7	12	0.40	0.46	**0.34**	0.64	**0.72**	0.48	**0.32**	0.27	0.20	**0.57**	0.51	0.26
axios	**0.48**	0.13	0.32	**2.6**	8.5	13.8	0.41	**0.31**	0.74	0.68	0.44	**0.74**	**0.47**	0.09	0.24	0.63	0.21	**0.67**
babel	**0.59**	0.17	0.27	**3.9**	71.1	28.3	**0.24**	0.37	0.37	**0.70**	0.50	0.48	**0.62**	0.24	0.28	**0.56**	0.24	0.34
express	**0.34**	0.16	0.18	**21**	26.3	8.6	**0.28**	0.31	0.24	**0.65**	0.51	0.50	**0.28**	0.14	0.17	**0.46**	0.24	0.23
hyper	**0.48**	0.31	0.36	**1.6**	3.4	34	**0.26**	0.66	0.88	**0.52**	0.40	0.42	**0.58**	0.24	0.25	0.48	0.55	**0.73**
jquery	**0.69**	0.27	0.55	**1.6**	6	75.4	0.47	**0.35**	0.78	**0.84**	0.44	0.63	**0.78**	0.36	0.46	0.63	0.27	**0.69**
lodash	**0.25**	0.22	0.14	40	**6.1**	64	0.69	0.52	**0.50**	0.79	**0.80**	0.66	0.16	**0.18**	0.12	**0.72**	0.60	0.45
material-ui	**0.49**	0.28	0.21	6	**5.4**	7	**0.18**	0.50	0.43	0.52	**0.64**	0.50	**0.59**	0.32	0.19	0.45	**0.53**	0.34
meteor	**0.39**	0.14	0.19	25.7	51.2	**13.1**	**0.40**	0.42	0.27	**0.69**	0.58	0.53	**0.31**	0.12	0.20	**0.58**	0.25	0.24
moment	**0.39**	0.29	0.30	19.6	8.7	**3.2**	0.48	0.59	**0.36**	**0.72**	0.63	0.66	**0.30**	0.21	0.29	**0.65**	0.62	0.46
parcel	**0.44**	0.19	0.30	7.9	9.1	**4.2**	**0.26**	0.39	0.31	**0.58**	0.45	0.54	**0.52**	0.20	0.35	**0.42**	0.23	0.31
pdf.js	**0.46**	0.23	0.21	**9.9**	93.2	354.8	**0.26**	0.39	0.88	**0.58**	0.51	0.55	**0.42**	0.20	0.13	0.52	0.29	**0.57**
react	**0.64**	0.32	0.29	9.1	**7.4**	13.7	**0.23**	0.45	0.84	0.74	0.78	**0.90**	**0.66**	0.27	0.20	0.62	0.55	**0.71**
serverless	**0.47**	0.13	0.28	5.5	18.2	**4.1**	**0.24**	0.25	0.26	**0.60**	0.50	0.58	**0.46**	0.23	0.28	**0.49**	0.15	0.32
three.js	**0.41**	0.05	0.28	99.4	4.8	**3.4**	0.66	**0.09**	0.28	**0.87**	0.36	0.63	0.28	**0.34**	0.27	**0.86**	0.07	0.36
vue	**0.57**	0.49	0.44	9.1	**3.1**	63.7	**0.65**	0.70	0.80	**0.91**	0.84	0.87	**0.45**	0.37	0.31	**0.82**	0.78	0.79
webpack	**0.39**	0.11	0.23	11.6	82.7	**1.9**	0.27	0.45	**0.21**	**0.56**	0.47	0.51	**0.34**	0.13	0.25	**0.47**	0.21	0.26
yarn	**0.54**	0.14	0.39	5.1	16.7	**1.2**	0.28	**0.25**	0.32	**0.58**	0.45	**0.58**	**0.70**	0.31	0.48	**0.46**	0.13	0.35
Average	**0.47**	0.23	0.29	**14.82**	24.45	39.73	**0.36**	0.41	0.49	**0.67**	0.56	0.60	**0.47**	0.24	0.26	**0.58**	0.36	0.45
Improvement		108%	65%		39%	63%		12%	26%		19%	13%		93%	80%		60%	28%
p-value		<0.001	<0.001		>0.05	<0.005		<0.005	<0.001		<0.001	<0.001		<0.001	<0.001		<0.001	−0.001
Cliff's delta		0.73	0.64		0.03	0.06		0.14	0.16		0.33	0.25		0.61	0.58		0.50	0.34
Effect size		L	L		N	N		N	S		S	S		L	L		L	M
Winner	**CBS+**			**CBS+**			**CBS+**			**CBS+**			**CBS+**			**CBS+**		

The best-performing results are highlighted in bold. "↓" indicates "the smaller the better"; "↑" indicates "the larger the better."

Table 7. The Average Performance of the Unsupervised Approaches Compared with CBS+ on JavaScript Projects in Terms of Six Studied Performance Measures

Project	F1-measure@20%↑			IFA↓			PCI@20%↓			Popt↑			Precision@20%↑			Recall@20%↑		
	CBS+	Churn	LT	CBS+	Churn	LT	CBS+	Churn	LT	CBS+	Churn	LT	CBS+	Churn	LT	CBS+	Churn	LT
Chart.js	**0.46**	0.36	0.21	**6.3**	82.2	59.7	**0.39**	0.95	0.56	0.65	**0.81**	0.62	**0.44**	0.23	0.15	0.56	**0.86**	0.44
Ghost	**0.60**	0.32	0.14	**3.8**	416.2	168.7	**0.22**	0.92	0.69	0.67	**0.73**	0.41	**0.65**	0.20	0.10	0.57	**0.74**	0.26
anime	**0.39**	0.28	0.07	**6.7**	24.5	19.9	0.40	0.84	**0.36**	0.64	**0.73**	0.25	**0.32**	0.18	0.10	0.57	**0.72**	0.08
axios	**0.48**	0.33	0.22	**2.6**	29.3	24.9	**0.41**	0.93	0.69	0.68	0.82	**0.85**	**0.47**	0.21	0.16	0.63	**0.80**	0.46
babel	**0.59**	0.35	0.12	**3.9**	283.3	67.7	**0.24**	0.94	0.52	0.70	**0.81**	0.56	**0.62**	0.23	0.09	0.56	**0.82**	0.18
express	**0.34**	0.19	0.15	**21**	171.1	95.5	**0.28**	0.86	0.58	0.65	**0.75**	0.60	**0.28**	0.12	0.10	0.46	**0.62**	0.33
hyper	**0.48**	0.41	0.21	**1.6**	42.8	37.5	**0.26**	0.93	0.64	0.52	**0.65**	0.47	**0.58**	0.28	0.16	0.48	**0.84**	0.40
jquery	**0.69**	0.68	0.16	**1.6**	72.9	67.7	0.47	0.92	**0.28**	0.84	**0.87**	0.42	**0.78**	0.55	0.22	0.63	**0.89**	0.14
lodash	**0.25**	0.24	0.07	**40**	109.7	81.7	0.69	0.94	**0.20**	0.79	**0.90**	0.28	**0.16**	0.14	0.06	0.72	**0.90**	0.09
material-ui	**0.49**	0.30	0.08	**6**	220.3	83.5	**0.18**	0.96	0.57	0.52	**0.76**	0.51	**0.59**	0.19	0.05	0.45	**0.84**	0.22
meteor	**0.39**	0.29	0.12	**25.7**	224	91.1	**0.40**	0.91	0.41	0.69	**0.84**	0.49	**0.31**	0.18	0.10	0.58	**0.76**	0.19
moment	**0.39**	0.30	0.19	19.6	104	**17.2**	**0.48**	0.95	0.54	0.72	**0.84**	0.72	**0.30**	0.19	0.16	0.65	**0.86**	0.42
parcel	**0.44**	0.32	0.14	**7.9**	44.7	36.4	**0.26**	0.84	0.40	**0.58**	0.52	0.48	**0.52**	0.23	0.13	0.42	**0.59**	0.18
pdf.js	**0.46**	0.24	0.11	**9.9**	313.1	171.3	**0.26**	0.92	0.62	0.58	**0.66**	0.52	**0.42**	0.15	0.07	0.52	**0.66**	0.22
react	**0.64**	0.39	0.18	**9.1**	617.8	117.3	**0.23**	0.97	0.72	0.74	**0.97**	0.62	**0.66**	0.26	0.13	0.62	**0.96**	0.38
serverless	**0.47**	0.27	0.14	**5.5**	254.7	107.7	**0.24**	0.90	0.62	0.60	**0.72**	0.58	**0.46**	0.19	0.11	0.49	**0.66**	0.23
three.js	**0.41**	0.34	0.22	99.4	165.5	**29.5**	0.66	0.97	**0.51**	0.87	**0.91**	0.75	**0.28**	0.21	0.16	0.86	**0.93**	0.38
vue	**0.57**	0.49	0.33	**9.1**	86.2	40.5	0.65	0.94	**0.56**	0.91	**0.94**	0.83	**0.45**	0.33	0.26	0.82	**0.92**	0.52
webpack	**0.39**	0.22	0.15	**11.6**	115.4	21.6	**0.27**	0.88	0.61	0.56	**0.63**	0.52	**0.34**	0.14	0.10	0.47	**0.62**	0.32
yarn	**0.54**	0.46	0.18	**5.1**	71.1	43.3	**0.28**	0.85	0.47	0.58	**0.60**	0.41	**0.70**	0.35	0.18	0.46	**0.70**	0.20
Average	**0.47**	0.34	0.16	**14.82**	172.44	69.14	**0.36**	0.92	0.53	0.67	**0.77**	0.54	**0.47**	0.23	0.13	0.58	**0.78**	0.28
Improvement		40%	198%		91%	79%		60%	31%	15%		42%		105%	262%	36%		179%
p-value		<0.001	<0.001		<0.001	<0.001		<0.001	<0.001	<0.001		<0.001		<0.001	<0.001	<0.001		<0.001
Cliff's delta		0.50	0.90		0.82	0.61		0.91	0.47	0.35		0.58		0.67	0.88	0.62		0.87
Effect size		L	L		L	L		L	M	M		L		L	L	L		L
Winner	**Supervised**			**Supervised**			**Supervised**			**Unsupervised**			**Supervised**			**Unsupervised**		

The best-performing results are highlighted in bold. "↓" indicates "the smaller the better"; "↑" indicates "the larger the better."

(2) According to the average performance of CBS+, we find that CBS+ can help developers to inspect only 36% of changes and identify about 58% of all defect-prone changes with 47% accuracy.

By analyzing the comparison between the best-performing supervised approach and two unsupervised approaches, we obtain the following observations:

(1) CBS+ statistically significantly outperforms Churn and LT with large effect size in most cases. In particular, CBS+ improves Churn and LT by 40% and 198% in terms of *F1-measure@20%*, by 91%

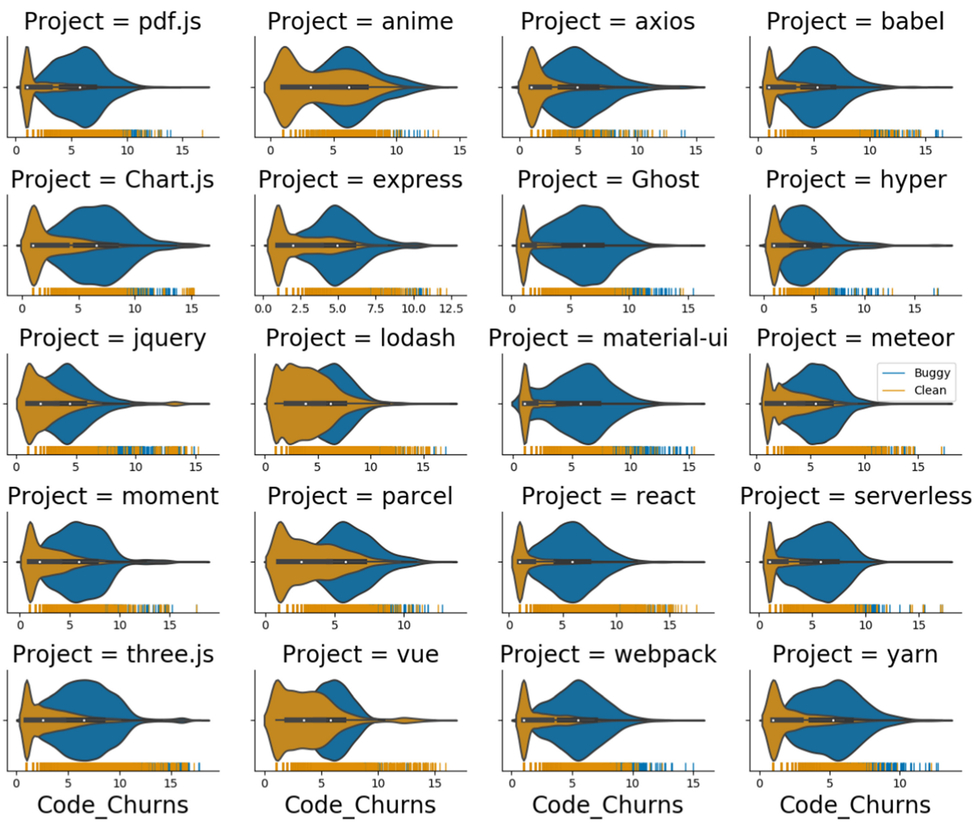

Fig. 2. Code_Churns distribution (after log transformation based on 2) of studied JavaScript projects.

and 79% in terms of *IFA*, by 60% and 31% in term of *PCI@20%*, and by 105% and 262% in terms of *Precision@20%*.

(2) In terms of *Recall@20%* and P_{opt}, the unsupervised method Churn performs the best. However, in terms of *Recall@20%*, Churn outperforms CBS+ at the cost of *Precision@20%*. Thus, when considering *F1-measure@20%*, which is a harmonic mean of *Recall@20%* and *Precision@20%*, we find that Churn has poor performance compared with CBS+. In terms of P_{opt}, Churn outperforms CBS+ with medium effect size.

(3) Churn obtains a high *Recall@20%* and P_{opt} with the help of skewed distribution of change sizes. In particular, we plot the distribution of each project by a violin plot as shown in Figure 2 according to the code churn (i.e., LA+LD). Since each project has a wide range of code churn of changes, we adopt a log transformation (base 2) to the values of code churn. In Figure 2, we split the changes into two groups and draw two types of violin plots: defect-prone violin and clean violin. Additionally, we also draw a rug figure for each project in the x-axis, which presents the whole distribution of changes in each project according to their code churns. According to Figure 2, we find that (1) the distribution of changes in each project is extremely skewed. Specifically, the majority of changes modifies less than 2^7 LOC. (2) The defect-prone changes modify more LOC than the clean changes do. Thus, Churn achieves better performances in terms of *Recall@20%* and P_{opt} because of the highly skewed distribution of change size.

(4) As a whole, when considering *F1-measure@20%*, *IFA*, *PCI@20%*, and P_{opt}, the supervised method CBS+ statistically significantly outperforms unsupervised methods: Churn and LT. We

exclude *Recall@20%* and *Precision@20%* since *F1-measure@20%* can holistically evaluate a model's performance.

Conclusion 1.1

Among the six effort-aware performance measures, the supervised method CBS+ statistically significantly performs the best among all supervised JIT defect prediction approaches. When compared with two unsupervised approaches (i.e., Churn and LT), CBS+ also statistically significantly performs the best in most cases. In terms of *Recall@20%* and P_{opt}, Churn performs better than CBS+ with the help of highly skewed distribution of dataset and at the loss of precision. Therefore, the supervised method such as CBS+ should be the first choice for JIT defect prediction when considering effort-aware performance measures for JavaScript projects.

Results for Question 2:

Can JavaScript-specific change-level features improve the performance of effort-aware JIT defect approach on identifying defect-prone changes?

Based on the results of Question 1, we find that supervised methods can achieve better performance than unsupervised methods, and CBS+ performs best among all supervised ones. In addition, according to the introduction of two unsupervised approaches (i.e., LT built only based on lt and Churn built based on la and ld) in Section 2.3, the five JavaScript-specific change-level features have no impact on the performance of two unsupervised approaches. Thus, in this question, we conduct a further experiment on whether the best-performing supervised approach CBS+ can achieve better performance when considering another five JavaScript-specific change-level features than CBS+ which only considers 14 language-independent change-level features.

Table 8 presents the comparison results of CBS+ in such a setting that CBS+ is built with or without JavaScript-specific change-level features. In particular, the column named "JIT" represents CBS+ built with 14 prior proposed language-independent change-level features, while the column named "JIT+JS" represents CBS+ built on the combination of 14 language-independent change-level features and five JavaScript-specific change-level features. The statistical results are shown in the bottom few rows of the table, and the changing trend of performance is illustrated in the last row.

According to the comparison results, we find that CBS+, on average, can be further improved by using JavaScript-specific features on all the six performance measures. In particular, CBS+$_{(JIT+JS)}$ statistically significantly improves CBS+$_{(JIT)}$ by 16% in terms of *F1-measure@20%*, by 76% in terms of *IFA*, by 6% in terms of P_{opt}, by 11% in terms of *Precision@20%*, and by 12% in terms of *Recall@20%*. In terms of *PCI@20%*, CBS+$_{(JIT+JS)}$ improves CBS+$_{(JIT)}$ by 6% on average but with no statistical significance.

Conclusion 1.2

JavaScript-specific change-level features can further improve the performance of JIT defect prediction approach.

3.2 RQ2: What are the Important Features for Effort-Aware JIT Defect Prediction in JavaScript Projects?

Motivation. To predict whether or not a change is a defect-prone one, Kamei et al. [34] consider 14 factors (i.e., change-level features) grouped into five dimensions derived from the source control

Table 8. The Average Performance Comparison Between CBS+ Built on Language-Independent Change-Level Features and CBS+ Built on the Combination of Language-Independent Features and JavaScript-Specific Features

Project	CBS+											
	F1-measure@20%↑		**IFA**↓		**PCI@20%**↓		**Popt**↑		**Precision@20%**↑		**Recall@20%**↑	
	JIT	JIT+JS	JIT	JIT+JS	JIT	JIT+JS	JIT	JIT+JS	JIT	JIT+JS	JIT	JIT+JS
Chart.js	0.46	**0.59**	6.3	**5.8**	0.39	**0.30**	0.65	**0.68**	0.44	**0.56**	0.56	**0.65**
Ghost	0.60	**0.65**	3.8	**2.0**	**0.22**	0.25	0.67	**0.72**	0.65	**0.66**	0.57	**0.65**
anime	0.39	**0.44**	6.7	**5.7**	0.40	**0.37**	0.64	**0.70**	0.32	**0.38**	0.57	**0.61**
axios	0.48	**0.59**	2.6	**1.5**	**0.41**	**0.41**	0.68	**0.81**	0.47	**0.52**	0.63	**0.76**
babel	0.59	**0.64**	3.9	**1.6**	**0.24**	0.28	0.70	**0.76**	0.62	**0.63**	0.56	**0.66**
express	0.34	**0.38**	21	**5.2**	0.28	**0.27**	**0.65**	0.64	0.28	**0.31**	0.46	**0.50**
hyper	0.48	**0.62**	1.6	**1.3**	**0.26**	0.39	0.52	**0.59**	0.58	**0.59**	0.48	**0.71**
jquery	0.69	**0.76**	1.6	**0.2**	**0.47**	0.55	**0.84**	**0.84**	0.78	**0.79**	0.63	**0.75**
lodash	0.25	**0.37**	40	**11.8**	0.69	**0.39**	**0.79**	0.75	0.16	**0.27**	**0.72**	0.68
material	0.49	**0.58**	6	**1.9**	**0.18**	0.21	0.52	**0.62**	0.59	**0.61**	0.45	**0.57**
meteor	0.39	**0.45**	25.7	**4.0**	0.40	**0.38**	0.69	**0.75**	0.31	**0.36**	0.58	**0.62**
moment	0.39	**0.50**	19.6	**3.6**	0.48	**0.41**	0.72	**0.78**	0.30	**0.41**	0.65	**0.73**
parcel	0.44	**0.47**	7.9	**3.3**	0.26	**0.25**	0.58	**0.62**	0.52	**0.57**	0.42	**0.44**
pdf.js	0.46	**0.50**	9.9	**4.6**	0.26	**0.25**	0.58	**0.60**	0.42	**0.46**	0.52	**0.55**
react	0.64	**0.74**	9.1	**2.7**	**0.23**	0.27	0.74	**0.84**	0.66	**0.72**	0.62	**0.78**
serverless	0.47	**0.52**	5.5	**2.4**	**0.24**	0.29	0.60	**0.67**	0.46	**0.48**	0.49	**0.57**
three.js	0.41	**0.54**	99.4	**5.0**	0.66	**0.37**	**0.87**	0.82	0.28	**0.43**	**0.86**	0.74
vue	0.57	**0.60**	9.1	**2.6**	0.65	**0.60**	0.91	**0.92**	0.45	**0.50**	**0.82**	**0.82**
webpack	0.39	**0.44**	11.6	**5.2**	0.27	**0.26**	0.56	**0.59**	0.34	**0.39**	0.47	**0.51**
yarn	0.54	**0.60**	5.1	**0.9**	**0.28**	0.31	0.58	**0.59**	0.70	**0.72**	0.46	**0.54**
Average	0.47	**0.55**	14.82	**3.6**	0.36	**0.34**	0.67	**0.71**	0.47	**0.52**	0.58	**0.64**
Imporvment		16%		76%		6%		6%		11%		11%
p-value		<0.001		<0.001		>0.05		<0.001		<0.001		<0.001
Cliff's delta		−0.29		0.42		−0.04		−0.16		−0.15		−0.26
Cliff's size		S		M		N		S		S		S
Trend	↗		↗		↗		↗		↗		↗	

The bestperforming results are highlighted in bold. "↓" indicates "the smaller the better"; "↑" indicates "the larger the better."

repository data of a project. In this article, we also propose five JavaScript-specific change-level features. These features, totally 19 features, describe a project from different perspectives, and they play varying degrees of importance to such a project. Thus, previous studies [34, 35] have realized the importance of features and want to understand the exact impact of various features. For example, Kamei et al. analyzed the most important features to their approach (i.e., EALR). Understanding the importance of features can help developers avoid the pitfalls which are strongly associated with the incidence of future defects. Thus, we want to investigate the most important features on JavaScript projects.

According to the results of RQ1, we find that CBS+ performs best when compared with other approaches. Thus, we investigate the impact on the performance of CBS+ to figure out the most important features on JavaScript projects in the context of effort-aware settings.

Method. Different projects are developed for different purposes, which indicates different features may play various roles in each project. Thus, we analyze how change-level features affect the performance of CBS+ after two data pre-processing steps: filtering correlated features and filtering redundant features.

Identification. After filtering all correlated change-level features, we rebuild the best-performing supervised JIT defect prediction approach CBS+ with the remaining features. As suggested by a previous study [15], we investigate feature importance by a 10 × 10-fold cross-validation experimental setting. In a whole, two phases are involved for identifying the important features as follows:

Calculating importance scores. First, we use training data to build CBS+ in each fold. Then, we calculate the generic feature importance score proposed by Tantithamthavorn et al. [62, 77]

for CBS+. The generic feature importance score can be calculated using the following two steps: (1) In the testing data, for each feature, we randomly permutate the values of the feature. That is, all the values of other features are kept as they are in the testing data except the value of one specific feature which is permutated. (2) For each feature, we calculate the total performance difference between the results obtained on the original testing data and the results obtained on the randomly permutated testing data. The larger the difference's value, the higher the corresponding feature's importance. Thus, we use the difference value as the proxy of the feature's importance.

Calculating importance ranks. After we obtain the importance value of each feature, we further calculate the importance ranks of each feature. We totally obtain 100 importance scores for each feature (i.e., 10 × 10-fold cross-validation). Then, we apply the *Scott-Knott ESD* (*SK-ESD*) test, an enhanced variant of the *Scott-Knott* test [68], on the feature importance scores. *Scott-Knott ESD* mitigates the skewness of input data and thus relaxes the assumption of normally distributed data which is strictly required by the *Scott-Knott*. Additionally, *Scott-Knott ESD* considers the effect size of the input data and merges any two statistically distinct groups with a negligible effect size into one group.

Finally, we obtain the ranks of each change-level feature on each JavaScript project. Based on the ranks of each feature, we can quickly figure out what are the important features for these projects. Additionally, to easily obtain the statistical information, we sum up the number of JavaScript projects where a feature is ranked as top-1 and one of the top-3 or top-5 important features.

Results. Table 9 shows the ranks of studied features in JavaScript projects. In Table 9, the first column presents the name of each project. The following 11 columns, named as Rank 1 to Rank 11, list the features in each group according to the level of their importance to each project. For the convenience of analysis, we sum up the number of each feature when they are ranked as the top-1, top-3, or top-5 important feature as shown in Table 10.

From Table 10, when considering top-1 important features, we find that the features "LT" (which is the most important feature in 11 projects), "NF" (which is the most important feature in 8 projects), "SO" (which is the most important feature in 1 project), and "TC" (which is the most important feature in 1 project) have more importance than other features in JavaScript projects. For example, if a project contains many files (i.e., "NF"), the business logic of the project will be distributed in different files, which will increase the difficulty for developers to understand and memorize the code, thus affecting the quality of the software. At the same time, if a developer wants to modify a large file (i.e., "LT"), it also affects the developer's comprehension of the whole file since it already contains many functionalities, which will induce unknown errors. For different programming languages, developers need to deal with their special language features carefully. For example, some special operators in JavaScript (i.e., "SO") can simplify the business logic and reduce the workload of development if they are used correctly. However, the incorrect use of these operators will lead to unknown logic errors. In addition, for weakly typed programming languages (i.e., "TC"), developers must carefully check the type of variables to make a complete and correct judgment.

When considering top-5 important features, we find that "LA," "LT," "EXP," and "SO" are the top-3 important features and are important in 17, 17, 16, and 15 projects, respectively, on JavaScript projects. It is obvious that the quality of code written by experienced developers is higher than that of beginners. In addition, the number of code lines will also affect the comprehension of a developer. It is easier for developers to understand smaller code changes than to understand larger ones.

As a whole, we find that for JavaScript projects, the features belonging to "Diffusion," "Size," and "JavaScript-specific" dimension are important to JIT defect prediction.

Table 9. Ranks of the Studied Features in the JavaScript Projects

Projects	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6	Rank 7	Rank 8	Rank 9	Rank 10	Rank 11
vue	NF	SEXP	EXP, LA	SO, LD	LT, NDEV, FIX, Entropy, NS, AGE, NUC, HtmlCss, BDom	TC	Strict				
react	LT	LA	SEXP	EXP	LD	Entropy, SO	FIX, AGE	HtmlCss, ND, Strict, BDom	TC		
axios	LT	ND	SO	SEXP, Strict	LA, EXP, TC	BDom, NDEV, FIX, Entropy, HtmlCss					
three.js	NF	EXP	SEXP	LA	TC	NS, LD	AGE, LT	SO, FIX, Strict	HtmlCss, BDom		
jquery	NF, LT	SEXP	EXP	TC, SO	NDEV	Strict	HtmlCss, LA	LD	FIX	AGE, BDom	ND
webpack	NF	LA	EXP, SO	Strict, SEXP	BDom, TC, FIX, ND, LT, AGE	HtmlCss, Entropy					
material-ui	LT	SEXP	ND	EXP	Entropy, SO	FIX, LD	BDom, TC	Strict, HtmlCss			
express	NF	LA	SEXP	FIX	TC	AGE, ND, EXP, SO	LD, HtmlCss, Strict, BDom	Entropy, LT			
Chart.js	LT	ND	SO, LA	TC, EXP	Strict, BDom	Entropy, FIX, HtmlCss, SEXP, AGE					
moment	NF	EXP, SEXP	LA, SO	TC, BDom, FIX, AGE, LT, Strict, HtmlCss							
meteor	NF	LT	NDEV	ND	SO, LA	BDom, FIX, NS	Strict	AGE, EXP, HtmlCss	LD	TC, SEXP	
lodash	TC	EXP, NF	LT	FIX	BDom, Strict, LA	ND, AGE	HtmlCss, SO				
yarn	NF	LT	SEXP, EXP	AGE	LA	SO, TC	BDom	LD, NS, Strict, HtmlCss	FIX		
babel	LT	Entropy	LA	SO	ND, NS	FIX	Strict, LD, SEXP	TC, HtmlCss, BDom	AGE, EXP		
parcel	LT	Entropy, LA	ND, NDEV	NS, LD, SO	EXP	FIX, Strict	BDom	AGE, TC, HtmlCss			
anime	SO	NF	NDEV, NUC	FIX, EXP, SEXP	Strict, BDom, AGE, ND, HtmlCss, LA						
serverless	LT	SEXP	EXP, LA	ND	Entropy, SO	TC, FIX, Strict	AGE, BDom, HtmlCss				
Ghost	LT	NS	Entropy	SO	SEXP	EXP, BDom, HtmlCss, TC, Strict, FIX	LD				
hyper	LT	LA	SO	HtmlCss, TC, Entropy, LD, FIX, EXP, BDom, ND, AGE, Strict							
pdf.js	LT	LA	SO, Entropy, BDom, EXP	TC, AGE, Strict	FIX, HtmlCss	ND, SEXP					

These features are divided into a few groups based on their level of ranks.

Conclusion 2

For JavaScript projects, "LT," "NF," "SO," and "TC" are the most important features for JIT defect prediction, which indicates the importance of the three types of JIT features: "Diffusion," "Size," and "JavaScript-specific." Additionally, "LT" ranks top-1 in the majority of JavaScript projects, which demonstrates "LT" is the most important feature and should be further considered in future studies.

3.3 RQ3: Is There an Association Between Project-Related Features and the Probability of a Defect-Prone Change in JavaScript Projects?

Motivation. We have deeply investigated the relationship between the features and the probability of a change to be a defect-prone one. These features describe a project from different perspectives, and they play varying degrees of importance to such a project. However, these features can only characterize a project from an inner-side view of a project. That is, we previously analyzed the inner characteristics (i.e., 14 language-independent change-level features and 5 JavaScript-specific

Table 10. Number of Studied JavaScript Projects, in Which a Feature is Ranked as a Top-1, Top-3, or Top-5 Important Feature

Dimension	**Features**	**Top-1**		**Top-3**		**Top-5**	
		# Projects	**# Sum**	**# Projects**	**# Sum**	**# Projects**	**# Sum**
Diffusion	NS	0	**8**	1	**19**	4	**32**
	ND	0		4		10	
	NF	8		10		10	
	Entropy	0		4		8	
Size	LA	0	**11**	11	**25**	17	**38**
	LD	0		0		4	
	LT	11		14		17	
Purpose	FIX	0	**0**	0	**0**	8	**8**
History	NDEV	0	**0**	3	**4**	5	**14**
	AGE	0		0		7	
	NUC	0		1		2	
Experience	EXP	0	**0**	9	**18**	16	**29**
	REXP	0		0		0	
	SEXP	0		9		13	
JavaScript-specific	HtmlCss	0	**2**	0	**9**	5	**46**
	Strict	0		0		8	
	BDom	0		1		8	
	SO	1		7		15	
	TC	1		1		10	

change-level features) on the probability of a defect-prone change. Actually, there are some other features that may have an impact on the probability of a defect-prone change, such as the number of files, the number of contributors, and the number of branches. These features can characterize a project from an outer-side view of a project. Thus, in this RQ, we want to further investigate the association between outer-side features and the probability of a defective change on JavaScript projects.

Method. Project-related features. We want to investigate 11 project-related context features (as shown in Table 1) which are widely adopted by previous studies [92, 94, 96]. These features are the number of stars (Stars), the number of fork (Forks), the number of project branches (Branches), the number of changes (Changes), the number of defect-prone change (Defective), the ratio of defect-prone changes (Def Ratio), the number of files (Files), the total lines of code (LOC), the median size of code churn (Med_size), the mean size of code churn (Mean_size), and the number of contributors (Contributors), respectively. For all the project-related features, we divide the values into four groups based on the first, second, and third quartiles (i.e., least, less, more, most), as suggested by prior work [33].

Mixed effects model. To investigate the association between the project-related features and the probability of a defect-prone change, as suggested by Hassan et al. [25], we adopt mixed effects logistic regression [78], which has the ability to capture the variation of the interpretation of models among different projects. The mixed effects logistic regression model is an instance of **generalized linear mixed models** (i.e., **GLMMs**), which includes both fixed effects and random effects [41]. In the JIT defect prediction scenario, the fixed effects usually represent the explanatory features which are used to explain the data at the change level (i.e., 19 change-level features), while the random effects usually represent the project-related features which are used to describe the information of a project with a holistic perspective (i.e., 11 statistical project-related features). Using explanatory features and project-related features, a mixed effects model estimates the effects of the change-level features on the probability of a change to be a defect-prone one, while accounting for the different project-related features.

Building a mixed effects model involves two phases: data pre-processing and model building.

Phase 1: Data pre-processing. Firstly, we combine all studied JavaScript projects into one dataset. Then, we add project-related features into the combined dataset. That is, each instance in the combined dataset contains two groups of information: change-level features and project-related features. After that, we do an analysis on the correlation and redundancy of features since strongly correlated and redundant features would have a serious impact on the interpretations of a mixed effects model as stated in previous studies [25, 69]. We use the methods as introduced in Section 2.5, and finally find six highly correlated change-level features (i.e., NS, Entropy, NDEV, NUC, LD, and REXP) and find no redundant feature. Therefore, these features are removed for accurate analysis.

Phase 2: Mixed effects model building. Two types of mixed effect models, as suggested by previous work [73], are widely used: **random intercept models** (i.e., **RIMs**) and **random slope and intercept models** (i.e., **RSIMs**). The RIM possesses various intercepts for project-related features and fixes slopes for explanatory features, while the RSIM possesses various intercepts for project-related features and distinct slopes for explanatory features. In our study, we prefer to RSIMs since we assume that change-level features from different projects have different relationships with the probability of a defect-prone change.

In our RSIM, we treat project names as the random effect and LT as the random slope against the project, respectively. Once adopting this setting, different projects obtain a various basic probability of a defect-prone change, and meanwhile, LT obtains a various association with the probability of a change to be a defect-prone one. We choose LT as the random slope since it is the most important feature for JavaScript projects according to the results discussed in RQ2. The remaining change-level features are treated as fixed effects. Finally, we use *glmer*, a function in R tool-kit *lme*4,[10] to implement the mixed effects logistic regression model.

Results. The results of our mixed effects model are shown in Table 11. We firstly analyze the goodness-of-fit of our mixed model. Then, we further respectively analyze the association between project-related features or change-level features and the probability of a defect-prone change.

Goodness-of-Fit. We apply the widely used conditional coefficient of determination for generalized logistic regression and mixed effects models (i.e., R^2 *or* R^2_{GLMM}) [32, 48] to evaluate how well our mixed effects model fits the combined dataset. In particular, we use *r.squaredGLMM*, a function in the *MuMIn* [11] took-kit, to calculate coefficients and this function spits out two types of values: marginal R^2 values and conditional R^2 values. In particular, the marginal R^2 values are those associated with the fixed effects, while the conditional ones are those of the fixed effects plus the random effects. In our study, the former considers the change-level features, while the latter considers both change-level features and project-related features.

According to the results reported by the *r.squaredGLMM* function, we obtain the R^2 of the full mixed effect model to be 0.64, while the R^2 of the model with just fixed effects to be 0.43. The results mean the model with full mixed effects has the ability to explain the variability of the combined dataset by 64%, and improves the model with just fixed effects by 49%.

[10] https://cran.r-project.org/web/packages/lme4/lme4.pdf.
[11] https://cran.r-project.org/web/packages/MuMIn/MuMIn.pdf.

Table 11. Summary of the Mixed Effects Model

Type	Variable	Variance	Estimate	χ^2	Pr(>χ^2)
Random slope	LT	0.30	-	9,503.37	***
Random effects	Stars	0.67	-	567.10	***
	Branches	0.54	-	207.23	***
	Def Ratio	<0.01	-	158.20	***
	Changes	0.07	-	120.22	***
	Files	<0.01	-	92.97	***
	Defective	<0.01	-	20.13	***
	Forks	0.35	-	18.54	***
	Contributors	0.41	-	0.19	o
	LOC	<0.01	-	<0.01	o
	Mean_size	0.72	-	<0.01	o
	Med_size	0.71	-	<0.01	o
Fixed effects	NF	-	1.05	3,675.22	***
	LA	-	1.95	1,840.95	***
	EXP	-	−0.20	947.31	***
	SEXP	-	0.21	760.51	***
	SO	-	0.16	329.37	***
	FIX	-	0.24	239.84	***
	Strict	-	−0.34	213.49	***
	HtmlCss	-	−0.32	146.78	***
	Bdom	-	0.08	28.32	***
	TC	-	−0.04	11.89	***
	AGE	-	<0.01	5.06	*

Statistical significance of χ^2;
Significance codes (p-value): ***< 0.001 < **< 0.01 < *< 0.05 < o.

Association between project-related features and the probability of a defect-prone change. To evaluate the association between project-related features and the probability of a defect-prone change, we adopt the χ^2 value of each project-related feature as suggested by Bolker et al. [3]. Notice that the larger the value of a project-related feature's χ^2, the stronger the association that the feature has a defect-prone change.

Table 11 presents the summary of statistics for our mixed effects logistic regression model. According to the results shown in Table 11, we find that seven project-related features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective, and Forks) have a significant association with the probability of a change to be a defect-prone one. Specifically, among all associated features, the top three are the number of stars (Stars), the number of branches (Branches), and the ratio of defect-prone changes (Def Ratio), which indicates that the more popular a project is, the more likely it is to induce defects. For example, if a project attracts more attention from participants (such as developers), it obtains more stars on the GitHub website. Then, more new functionality features will be introduced in the next version. For implementing these features, the project manager seems to create more branches of such a project. As the development process continues, more modifications will occur in the project, and thus more change will be submitted into the code repository for the publication of a new version. With the birth of new functions in the project, it will obtain more and more attention. This process continues to cycle and increase the complexity of this project. Therefore, as time goes by, the probability of inducing defects will increase. As a whole, the findings in this subsection indicate that a mixed effects model which takes project-related features into consideration can provide a deeper understanding of the characteristics of defect-introducing changes.

Association between change-level features and the probability of a defect-prone change. As shown in Table 11, LT, NF, and LA have the strongest association with the probability of a change to be defect-prone one. Our finding indicates that the three change-level features (i.e., LT, NF, and LA) are highly associated with the probability of defect-prone changes, which is basically consistent with the results observed in RQ2 (i.e., the importance of features to CBS+).

Conclusion 3

Project-related features are associated with the probability of a change to be a defect-prone one in JavaScript projects. Specifically, the seven features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective, and Forks) have the largest and statistically significant association with the probability of a defect-prone change for studied JavaScript projects.

4 DISCUSSION

4.1 JIT Defect Prediction in Effort-Unaware Setting

We have verified the effectiveness of 14 prior proposed programming language–independent features in identifying defect-prone changes in the effort-aware setting. However, whether these language-independent features proposed based on Java or C++ projects can still achieve good performance in the effort-unaware setting is unknown. Moreover, whether the five proposed JavaScript-specific features can further improve the performance of the defect prediction model built on 14 language-independent features is still unknown. Thus, we want to investigate how these change-level features (i.e., 14 programming language–independent features and five JavaScript-specific features) affect the performance of effort-unaware JIT defect prediction approaches in the effort-unaware setting.

We evaluate three classical effort-unaware approaches (i.e., LR, NB, and RF) on the JavaScript projects after two data pre-processing steps (i.e., filtering correlated features and filtering redundant features), and analyze four effort-unaware performance measures of each approach. Then, we compare the performance of these three approaches and figure out the best one. Lastly, we conduct a further experiment on whether the five JavaScript-specific features can further improve the performance of the best one.

Table 12 presents the average performance of three classical approaches in terms of four effort-unaware performance measures on JavaScript projects. Table 13 presents the comparison results of a best-performing effort-unaware approach (i.e., RF) in such a setting that RF is built with or without JavaScript-specific change-level features. In particular, the column named "JIT" represents RF is built with 14 prior proposed language-independent change-level features, while the column named "JIT+JS" represents RF is built on the combination of 14 language-independent change-level features and five JavaScript-specific change-level features. The bottom few rows of Tables 12 and 13 show the statistical information. The best approaches are listed in the last row in Table 12, and the changing trend of performance is illustrated in the last row in Table 13. From the results shown in Table 12, we can achieve the following observations:

(1) Three classical effort-unaware approaches perform similarly on average, and RF statistically significantly outperforms LR and NB with a medium or small effect size in most cases (except for LR in terms of *PFA* and NB in terms of *Recall*).

Table 12. The Average Performance of Three Classical Classifiers on JavaScript Projects in Terms of Four Studied Effort-Unaware Performance Measures in the Effort-Unaware Setting

Project	**AUC**$^{\uparrow}$			**F1-measure**$^{\uparrow}$			**PFA**$^{\downarrow}$			**Recall**$^{\uparrow}$		
	LR	**NB**	**RF**	**LR**	**NB**	**RF**	**LR**	**NB**	**RF**	**LR**	**NB**	**RF**
Chart.js	0.61	0.62	**0.69**	0.38	0.38	**0.52**	**0.06**	0.08	**0.06**	0.28	0.31	**0.44**
Ghost	0.72	0.68	**0.79**	0.59	0.52	**0.70**	**0.03**	0.04	0.05	0.47	0.40	**0.63**
anime	0.52	0.56	**0.59**	0.09	0.24	**0.27**	0.06	0.11	**0.02**	0.09	**0.23**	0.20
axios	0.60	0.62	**0.69**	0.31	0.37	**0.49**	**0.05**	**0.05**	0.06	0.25	0.30	**0.45**
babel	0.62	0.54	**0.74**	0.37	0.16	**0.62**	0.04	**0.02**	0.07	0.28	0.09	**0.54**
express	0.55	**0.62**	0.57	0.18	**0.35**	0.24	**0.02**	0.08	0.03	0.12	**0.33**	0.17
hyper	0.58	0.60	**0.65**	0.33	0.32	**0.44**	**0.06**	0.05	**0.06**	0.23	0.24	**0.35**
jquery	0.70	0.68	**0.77**	0.70	0.57	**0.80**	0.24	**0.07**	0.25	0.65	0.43	**0.78**
lodash	0.50	0.51	**0.53**	0.01	**0.23**	0.11	**0.00**	0.67	0.03	0.00	**0.68**	0.09
material-ui	0.63	0.56	**0.73**	0.41	0.21	**0.59**	0.03	**0.01**	0.05	0.30	0.13	**0.51**
meteor	0.53	0.53	**0.59**	0.12	0.15	**0.31**	**0.02**	0.07	0.04	0.07	0.13	**0.23**
moment	0.54	0.55	**0.63**	0.16	0.22	**0.39**	**0.04**	0.12	**0.04**	0.11	0.22	**0.30**
parcel	0.58	0.57	**0.62**	0.26	0.25	**0.35**	**0.01**	0.07	0.02	0.16	0.22	**0.25**
pdf.js	0.62	**0.71**	0.69	0.37	**0.52**	0.51	0.05	0.16	**0.04**	0.28	**0.58**	0.41
react	0.67	0.73	**0.79**	0.49	0.53	**0.70**	**0.05**	0.23	0.06	0.39	**0.69**	0.65
serverless	0.64	**0.67**	0.65	0.42	**0.49**	0.44	**0.03**	0.08	0.04	0.31	**0.43**	0.34
three.js	0.51	0.55	**0.59**	0.06	0.18	**0.32**	**0.01**	0.08	0.03	0.03	0.19	**0.22**
vue	0.62	0.58	**0.68**	0.43	**0.55**	0.53	**0.12**	0.80	0.13	0.36	**0.95**	0.48
webpack	0.57	0.59	**0.61**	0.25	0.29	**0.36**	**0.01**	0.03	0.03	0.15	0.22	**0.26**
yarn	0.68	0.59	**0.69**	0.52	0.34	**0.56**	0.06	**0.04**	0.07	0.41	0.23	**0.46**
Average	0.60	0.60	**0.66**	0.32	0.34	**0.46**	**0.05**	0.14	0.06	0.25	0.35	**0.39**
Improvement	11%	10%		43%	34%			65%	15%	57%	11%	
p-value	<0.001	<0.001		<0.001	<0.001			<0.001	<0.001	<0.001	<0.05	
Cliff's delta	0.38	0.35		0.34	0.32			0.32	0.18	0.34	0.12	
Effect size	M	M		M	S			S	S	M	N	
Winner	**RF**			**RF**			**RF**			**RF**		

The bestperforming results are highlighted in bold. "↓" indicates "the smaller the better"; "↑" indicates "the larger the better."

(2) RF improves LR and NB by 11% and 10% in terms of *AUC*, by 43% and 34% in terms of *F1-measure*, and by 57% and 11% in terms of *Recall*.

(3) In terms of *PFA*, LR performs best and improves NB and RF by 65% and 15%, respectively.

From the results shown in Table 13, we find that RF, on average, can be further improved by using JavaScript-specific features in terms of three effort-unaware performance measures. In particular, $\text{RF}_{(JIT+JS)}$ statistically significantly improves $\text{RF}_{(JIT)}$ by 9% in terms of *AUC*, by 21% in terms of *F1*, and by 53% in terms of *Recall*. However, $\text{RF}_{(JIT+JS)}$ performs worse in terms of *PFA*. In a whole view, JavaScript-specific change-level features can further improve the performance of the effort-unaware defect prediction approach in the effort-unaware setting.

Among the four effort-unaware performance measures, three classical effort-unaware classifiers perform similarly on average, and RF statistically significantly outperforms LR and NB in most cases. In addition, JavaScipt-specific change-level features can further improve the performance of the effort-unaware defect prediction approach in the effort-unaware setting.

4.2 Results Comparison

Researchers have proposed many JIT defect prediction approaches based on the change-level features and have conducted an empirical study on six projects [29, 30, 34, 39, 89] to investigate many

Table 13. The Average Performance Comparison Between RF Built on Language-Independent Change-Level Features and RF Built on the Combination of Language-Independent Features and JavaScript-Specific Features

Project	**AUC**$^{\uparrow}$		**F1-measure**$^{\uparrow}$		**PFA**$^{\downarrow}$		**Recall**$^{\uparrow}$	
	JIT	**JIT+JS**	**JIT**	**JIT+JS**	**JIT**	**JIT+JS**	**JIT**	**JIT+JS**
Chart.js	0.69	**0.71**	0.52	**0.56**	**0.06**	0.11	0.44	**0.54**
Ghost	0.79	**0.85**	0.70	**0.76**	**0.05**	0.12	0.63	**0.82**
anime	**0.59**	**0.59**	0.27	**0.29**	**0.02**	0.04	0.20	**0.23**
axios	0.69	**0.70**	0.49	**0.51**	**0.06**	0.11	0.45	**0.51**
babel	0.74	**0.78**	0.62	**0.68**	**0.07**	0.12	0.54	**0.70**
express	0.57	**0.64**	0.24	**0.38**	**0.03**	0.16	0.17	**0.46**
hyper	0.65	**0.69**	0.44	**0.49**	**0.06**	0.13	0.35	**0.54**
jquery	**0.77**	**0.76**	**0.80**	0.79	0.25	**0.23**	**0.78**	0.77
lodash	0.53	**0.61**	0.11	**0.32**	**0.03**	0.15	0.09	**0.37**
material	0.73	**0.77**	0.59	**0.62**	**0.05**	0.08	0.51	**0.62**
meteor	0.59	**0.71**	0.31	**0.51**	**0.04**	0.29	0.23	**0.69**
moment	0.63	**0.69**	0.39	**0.49**	**0.04**	0.14	0.30	**0.52**
parcel	0.62	**0.63**	0.35	**0.39**	**0.02**	0.05	0.25	**0.30**
pdf.js	0.69	**0.79**	0.51	**0.60**	**0.04**	0.17	0.41	**0.72**
react	0.79	**0.85**	0.70	**0.72**	**0.06**	0.11	0.65	**0.81**
serverless	0.65	**0.78**	0.44	**0.63**	**0.04**	0.17	0.34	**0.73**
three.js	0.59	**0.73**	0.32	**0.57**	**0.03**	0.22	0.22	**0.68**
vue	0.68	**0.71**	0.53	**0.64**	**0.13**	0.24	0.48	**0.67**
webpack	0.61	**0.73**	0.36	**0.55**	**0.03**	0.17	0.26	**0.63**
yarn	0.69	**0.74**	0.56	**0.64**	**0.07**	0.09	0.46	**0.58**
Average	0.66	**0.72**	0.46	**0.56**	**0.06**	0.14	0.39	**0.59**
Improvement		9%		21%	59%			53%
p-value		<0.001		<0.001	<0.001			<0.001
Cliff's delta		0.31		0.23	0.52			0.44
Effect size		S		S	L			M
Trend	$\nearrow$		$\nearrow$		$\searrow$		$\nearrow$	

The bestperforming results are highlighted in bold. "↓" indicates "the smaller the better"; "↑" indicates "the larger the better."

important aspects involving (1) the effectiveness comparison between supervised and unsupervised JIT defect prediction approaches in the effort-aware setting, (2) the important change-level features for indicating defect-prone changes, (3) the association between project-related features and the project quality, and (4) the performance of classical classifiers in the effort-unaware JIT defect prediction setting. For a better comparison between findings reported in this article and findings reported in previous work, we summarize the results from both this article and previous work, and discuss the similarities and differences in this section. The details are shown in Table 14.

In Table 14, we summarize our findings, which correspond to the three research questions introduced in Section 3. In the table, we list the results obtained in our empirical studies and the results collected from previous work. Then, we briefly analyze the similarities or differences between our conclusions and the conclusions of previous work. According to the results shown in Table 14, we can achieve the following conclusions based on the analysis of projects developed using Java, C++, or JavaScript programming languages:

(1) When considering inspection effort, supervised JIT defect prediction approaches can achieve better performance than unsupervised approaches in a holistic view and CBS+ proposed by Huang et al. [30] is the outstanding one among all supervised approaches.

(2) Programming language–specific change-level features can further improve the performance of the JIT defect prediction approach on identifying defect-prone changes compared with the approach built with programming language–independent change-level features in both effort-aware and effort-unaware settings.

(3) Different change-level features have a varying impact on different projects developed by different programming languages.

(4) Project-related features (i.e., the number of changes, the number of files) have a high association with the quality of the project.

Based on these conclusions and the analysis in Section 3, we provide a few practical guidelines for developers:

— Developers may not use programming-language special operators in practical development if they do not fully understand their special meanings.
— Developers may strictly check the type of variable with the specific statements if they need conditional judgment, especially for a weakly typed programming language.
— Developers may concentrate related functions in the same file, which can reduce the number of files in a project and reduce the complexity of the project.
— Developers may use CBS+ as a quality assurance assistant when project development has limited resources (e.g., time, human-resource, or budget).

5 THREATS TO VALIDITY

Threats to internal validity mainly consist in the potential errors in the implementation of the approaches when we re-implement the supervised and unsupervised approaches using Python language. In particular, two unsupervised approaches (i.e., LT and Churn) are both implemented by their authors using R language. Two supervised approaches (i.e., OneWay and CBS+) are implemented by Java programming language. For EALR, the programming language used for implementation is unknown. To minimize the internal threats, we not only implement these approaches by pair programming but also make full use of third-party implementations such as the scikit-learn [57]. For these studied approaches, although our code is written in Python language, we have carefully read the published papers and strictly follow the description of these approaches. Besides, it is very challenging to retrieve 100% truly clean data that contains no mislabeled changes. In this paper, as suggested by Fan et al. [14], we use MA-SZZ algorithm with minor modification to label our studied top-20 JavaScript projects and conduct all experimental studies on the labeled dataset. From our manual analysis results on the sampled changes, we find that the effectiveness of MA-SZZ is acceptable.

Threats to external validity mainly consist in the quality and generalizability of our datasets. We use 20 JavaScript projects, which belong to different application domains, vary in change size, number of contributors, and cover a long period of time. However, there are still many other projects in other domains with a few stars on GitHub, which are not considered in our study. Besides, in our experiment, all these projects are open source projects. Thus, it is still unknown whether our conclusions are generalizable to commercial projects. In the future, we plan to reduce this threat by considering more additional software projects, especially for commercial projects.

Table 14. Summary and Comparison of Findings: Answers to RQ1, RQ2, and RQ3

RQ1: The effectiveness comparison between supervised and unsupervised JIT defect prediction approaches.	
RQ1.1: Comparison among all supervised approaches.	
Findings in this article:	**Findings in previous work** [29, 30]:
• CBS+ statistically significantly outperforms EALR and OneWay with a medium or large effect size in terms of six effort-aware performance measures in most cases.	• CBS+ can find about 46% of all defective changes and significantly outperforms EALR in terms of *Recall@20%* with an average improvement of 47%.
• CBS+ improves EALR and OneWay by 108% and 65% in terms of *F1-measure*@20%, by 39% and 63% in terms of *IFA*, by 12% and 26% in terms of *PCI@20%*, by 19% and 13% in terms of P_{opt}, by 93% and 80% in terms of *Precision@20%*, and by 60% and 28% in terms of *Recall@20%*, respectively.	• CBS+ performs better than OneWay in terms of *Precision@20%*, *F1-measure*@20%, and *IFA* when inspecting 20% LOC.
• CBS+ is the best approach among all supervised JIT defect prediction approaches.	• CBS+ performs better than EALR and OneWay in different experimental settings.
Similarity: ☑ When considering inspection effort (i.e., 20% LOC), CBS+ always outperforms EALR and OneWay in terms of many different effort-aware performance measures.	
RQ1.1: Comparison between supervised and unsupervised approaches.	
Findings in this article:	**Findings in previous work**:
• CBS+ significantly statistically outperforms Churn and LT with large effect size in most cases.	• Unsupervised JIT defect prediction approaches (i.e., LT [89] and Churn [39]) can achieve comparable or better performance than the supervised approach (i.e., EALR) in terms of *ACC* and P_{opt} due to the skewed distribution of change sizes.
• CBS+ improves Churn and LT by 40% and 198% in terms of *F1-measure@20%*, by 91% and 79% in terms of *IFA*, by 60% and 31% in terms of *PCI@20%*, and by 105% and 262% in terms of *Precision@20%*.	• CBS+ performs better than LT in terms of *Recall*, *Precision*, *F1-measure*, *IFA*, and *PCI@20%*.
• In terms of *Recall@20%* and *Popt*, the unsupervised method Churn performs best. • In a holistic view, considering *F1-measure@20%*, *IFA*, *PCI@20%*, and P_{opt}, the supervised method CBS+ statistically significantly outperforms unsupervised methods Churn and LT.	
Similarities: ☑ Supervised JIT defect prediction approach can outperform unsupervised approach in terms of most effort-aware performance measures in most cases. ☑ Unsupervised approach (i.e., Churn) obtains a high *Recall@20%* and P_{opt} due to the skewed distribution of change sizes.	

(Continued)

Table 14. Continued

RQ1.2: The impacts of programming language–specific change-level features on identifying defect-prone changes.	
Findings in this article:	**Findings in previous work**:
• JavaScript-specific change-level features can further improve the JIT defect prediction model compared with the ones which are built on prior language-independent change-level features.	
Differences: ☒ This article proposes five JavaScript-specific change-level features and confirms their usefulness on identifying defect-prone changes.	
RQ2: The important change-level features for indicating defect-prone changes.	
Findings in this article:	**Findings in previous work** [34]:
• These features (i.e., "LT," "NF," "SO," and "TC") which belong to the dimension of "Diffusion," "Size," and "JavaScript-specific" are the most important features for indicating defect-prone changes, especially for "LT."	• The features which belong to the dimension of "Diffusion," "Purpose," and "History" are the important features for indicating defect-prone changes, especially for "NF," "FIX," and "AGE."
Similarities: ☑ These features in the dimension of "Diffusion" are important features for identifying defect-prone changes in projects developed by Java, C++, or JavaScript projects, which shows the importance of such features especially for "NF." **Differences**: ☒ Different features have varying impacts on different projects developed by different programming languages. For example, the features in the dimension of "Purpose" and "History" are important to Java or C++ projects, while the features in the dimension of "Size" and "JavaScript-specific" are important to JavaScript projects.	
RQ3: The association between project-related features and the project quality.	
Findings in this article:	**Findings in previous work** [93, 94]:
• Project-related features are associated with the probability of a change to be a defect-prone one in JavaScript projects. Specifically, the seven features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective, and Forks) have the largest and statistically significant association with the probability of a defect-prone change for studied JavaScript projects.	• Project-related features can affect the distribution of software maintainability (i.e., NC (the number of changes), ND (the number of downloads)) [94] and increase the predictive power of the defect prediction model (i.e., TLOC (total lines of code), TNC (total number of commit), TND (total number of developers), and TNF (total number of files)) [93].
Similarities: ☑ Project-related features (i.e., the number of changes, the number of files) have an association with the quality of the projects. **Differences**: ☒ In addition to common project-related features, different projects have project-specific features, which may have different effects on the quality of projects. For example, project-specific features (i.e., the number of branches, the number of stars) used in this article affect the probability of a defect-prone change, while project-specific features (i.e., the number of downloads) used in previous work may affect the distribution of software maintainability.	

(Continued)

Table 14. Continued

Discussion: JIT defect prediction in effort-unaware setting.	
Findings in this paper:	**Findings in previous work** [34]:
• Random Forest statistically significantly outperforms Logistic Regression and Naive Bayes, and achieves a good performance of 0.66 in terms of AUC, 0.46 in terms of F1-measure, and 0.39 in terms of Recall. • Random Forest built with both 14 change-level features and 5 JavaScipt-specific features outperforms the Random Forest built only with 14 change-level features (i.e., 0.56 of F1-measure and 0.72 of AUC).	• EALR effectively identifies the defect-prone changes (i.e., 0.45 of F1-measure and 0.76 of AUC).
Similarities: ☑ The change-level features can effectively identify defect-prone changes in Java projects, C++ projects, or JavaScript projects in effort-unaware setting.	
Differences: ☒ JavaScript-specific features can further improve the performance of defect prediction approach compared with those built with programming language–independent change-level features.	

Threats to construct validity mainly consist in the suitability of our performance measures. We consider six effort-aware performance measures (*Precision@20%*, *Recall@20%*, *F1-measure@20%*, *IFA*, *PCI@20%* and P_{opt}). We have carefully discussed the motivation for using these performance evaluation measures and cited prior studies to support our assumptions. Besides, the non-parametric statistical hypothesis Wilcoxon signed-rank test and non-parametric effect size measure Cliff's δ are conducted to ensure the confidence of performance comparison among the approaches. Therefore, this construct validity should be acceptable.

6 RELATED WORK

The classical defect prediction approaches mainly focus on identifying defect-prone software entities at a coarse-grained level (e.g., class/file/module) [7, 49, 52, 86] , which makes it hard to apply them to practical usage since these approaches identify enormous scope in the source code for finding defect-prone lines of code. Recently, fine-grained level (e.g., change) defect prediction approaches have attracted extensive attention of researchers, which afterward widely referred to as Just-in-time (JIT) defect prediction [16, 29, 30, 34, 89].

Mockus and Weiss [46] firstly predict the risk of a software change in an industrial project using change-level measures (i.e., the number of touched subsystems, the number of modified files, the number of added lines of code, and the number of modification requests). However, labeling data is extremely time-consuming and human-resource-consuming, which hinders, to a certain degree, the relevant research on JIT defect prediction. Subsequently, Sliwerski et al. [72] proposed an approach named SZZ to identify defect-introducing changes automatically. They investigated SZZ on two open-source projects and found that the changes that are committed on Friday had a higher probability of being defect-inducing changes. Since then, many approaches have been proposed to progress the research of JIT defect identification.

Many researches focus on supervised JIT defect identification approaches. Kim et al. [35] proposed a model to classify whether a change is defect-prone or not using a few change features such as file names, change meta-data, change log, source code and complexity metrics. Yin et al. [90] investigated the relationship between defect-fixing changes and defect-introducing changes on a few operating systems including Linux, OpenSolaris, FreeBSD, and a mature commercial OS. Shihab et al. [70] conducted an industrial study for better understanding of defect-prone changes. They developed a tool to help developers labeling a change as defect-prone or clean at check-in time. Since the limitation of resource in practice, Kamei et al. [34] firstly proposed the effort-aware approach named EALR and conducted a large-scale empirical study on both open source and commercial projects. They used the number of modified lines as the proxy to measure the required effort for inspecting a change. Yang et al. [87, 88] successively proposed two JIT identification approaches. They preferred to using advanced and complexed techniques (i.e., ensemble learning and deep learning) for JIT defect prediction. Nayrolles and Hamou-Lhadj [50] used code clone detection technology to propose an approach named CLEVER to intercept defect-prone changes on 12 Ubisoft projects. CLEVER contains two phases. In the first phase, CLEVER assesses the likelihood of a change to be defect-prone one. In the latter phase, CLEVER adopts clone detection to intercept the defect-prone changes identified in the previous phase. Huang et al. [29] proposed a simple but improved supervised model named CBS based on the assumption that smaller modules are proportionally more defect-prone and should be inspected first. CBS includes two phases: building a classifier and sorting testing changes. Then, Huang et al. [30] further improved the performance of CBS and called it as CBS+. Fu and Menzies [16] proposed a JIT defect prediction named OneWay, which has two phases. In the former, OneWay selects best change-level feature by using the information of labeled data. In the latter, OneWay uses the features to build a model to identify defect-prone changes in testing data.

Apart from supervised JIT defect prediction approaches, some researchers also proposed unsupervised approaches since its simplicity and comparable performance. Yang et al. [89] firstly proposed a simple unsupervised approach named LT and conducted a large-scale comparison between supervised approaches and unsupervised approaches on six widely used open-source projects. Subsequently, Lit et al. [39] proposed another unsupervised approach, and their experimental results indicated their approach performed better than all the prior supervised approach and unsupervised approach LT.

7 CONCLUSION AND FUTURE WORK

In this paper, we first use MA-SZZ algorithm to label the 20 most popular JavaScript projects on GitHub based on language-independent change-level features. To investigate whether the change-level features can effectively identify defects in JavaScript projects, we conduct a case study on 20 JavaScript projects with 176,902 changes. We make a comparison between supervised JIT defect prediction approaches (i.e., CBS+, OneWay, EALR) and unsupervised JIT defect prediction approaches (i.e., LT and Churn) when considering six effort-aware performance measures. We find that in a holistic view, CBS+ statistically significantly performs better than other supervised approaches and unsupervised approaches. Additionally, we propose five JavaScirpt-specific change-level features and conduct a further experiment on whether the performance of the best supervised approach CBS+ can be further improved when considering language-dependent change-level features (i.e., *HtmlCss*, *BDom*, *Strict*, *SO* and *TC*). We find that JavaScipt-specific features further improve CBS+'s ability on identifying defect-prone changes. Afterwards, we further investigate which change-level features are the important ones to the best-performing approach CBS+. We find that the features in the dimensions of "Size", "Diffusion" and "JavaScript-specific" are the most important ones. Especially, "LT" is the most important feature since it ranks as the top-1 most important feature in many projects. Following that, we deeply investigate the association between project-related features and the probability of a change to be a defect-prone one. We find that project-related features have an association with the probability of a defect-prone change on JavaScript projects. Especially, the seven features (i.e., Stars, Branches, Def Ratio, Changes, Files, Defective and Forks) have the largest and statistical significant association with the probability of a defect-prone change on studied JavaScript projects. Lastly, we investigate the impact of change-level features on classical defect prediction model in the setting of effort-unaware and make a comparison between the results obtained in this paper with the results collected from previous work.

In future work, we will do more research to verify the conclusion of this article, and promote the use of JIT defect prediction in the JavaScript community (e.g., developing a plug-in in IDE). In addition, we plan to collect projects developed by different programming languages and commercial projects to verify the generality of JIT defect prediction approaches. Lastly, we plan to investigate more programming-language–specific features to improve the performance of existing JIT defect prediction approaches.

ACKNOWLEDGMENTS

Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

REFERENCES

[1] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of Measurement and Statistics* 3 (2007), 103–107.

[2] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (2010), 2–17.

[3] Benjamin M. Bolker, Mollie E. Brooks, Connie J. Clark, Shane W. Geange, John R. Poulsen, M. Henry H. Stevens, and Jada-Simone S. White. 2009. Generalized linear mixed models: A practical guide for ecology and evolution. *Trends in Ecology & Evolution* 24, 3 (2009), 127–135.

[4] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[5] Ana Erika Camargo Cruz and Koichiro Ochimizu. 2009. Towards logistic regression models for predicting fault-prone code across software projects. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 460–463.

[6] Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, and Chao Ni. 2019. Software defect number prediction: Unsupervised vs supervised methods. *Information and Software Technology* 106 (2019), 161–181.

[7] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[8] Norman Cliff. 2014. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press.

[9] David R. Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)* (1958), 215–242.

[10] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2016. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*. IEEE, 31–41.

[12] Daniel P. Delorey, Charles D. Knutson, and Scott Chun. 2007. Do programming languages affect productivity? A case study using data from open source projects. In *1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07), ICSE Workshops 2007*. IEEE, 8–8.

[13] ECMAScript language specification, 5.1 edition, June 2011. https://262.ecma-international.org/5.1/.

[14] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1559–1586.

[15] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E. Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering* 46, 5 (2018), 495–525.

[16] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 72–83.

[17] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information and Software Technology* 76 (2016), 135–146.

[18] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 172–181.

[19] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: Quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17).* IEEE, 758–769.

[20] Todd L. Graves, Alan F. Karr, James S. Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26, 7 (2000), 653–661.

[21] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: A benchmark of JavaScript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST'19).* IEEE, 90–101.

[22] Quinn Hanam, Fernando S. de M Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 144–156.

[23] James A. Hanley and Barbara J. McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[24] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, 78–88.

[25] Safwat Hassan, Chakkrit Tantithamthavorn, Cor-Paul Bezemer, and Ahmed E. Hassan. 2018. Studying the dialogue between users and developers of free apps in the Google Play store. *Empirical Software Engineering* 23, 3 (2018), 1275–1312.

[26] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 200–210.

[27] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering* 19, 2 (2012), 167–199.

[28] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. 2017. Global vs. local models for cross-project defect prediction. *Empirical Software Engineering* 22, 4 (2017), 1866–1902.

[29] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17).* IEEE, 159–170.

[30] Qiao Huang, Xin Xia, and David Lo. 2018. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* (2018), 1–40.

[31] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 279–289.

[32] Paul C. D. Johnson. 2014. Extension of Nakagawa & Schielzeth's R2GLMM to random slopes models. *Methods in Ecology and Evolution* 5, 9 (2014), 944–946.

[33] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[34] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773.

[35] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[36] Sunghun Kim, Thomas Zimmermann, Kai Pan, E. James, Jr., et al. 2006. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06).* 81–90.

[37] Gunes Koru, Hongfang Liu, Dongsong Zhang, and Khaled El Emam. 2010. Testing the theory of relative defect proneness for closed-source software. *Empirical Software Engineering* 15, 6 (2010), 577–598.

[38] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017), 1831–1865.

[39] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* IEEE Press, 11–19.

[40] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. 2010. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering.* 1–9.

[41] Charles E. McCulloch and John M. Neuhaus. 2005. Generalized linear mixed models. *Encyclopedia of Biostatistics* 4 (2005).

[42] Shane McIntosh and Yasutaka Kamei. 2015. Are fix-inducing changes a moving target?: A longitudinal case study of just-in-time defect prediction. *Proceedings of the 40th International Conference on Software Engineering* 44, 5 (2015), 412–428.

[43] Tim Menzies, Andrew Butcher, Andrian Marcus, and David Zimmermann, Thomas and David Cok. 2011. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 343–351.

[44] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering* 1 (2007), 2–13.

[45] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. 2008. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*. 47–54.

[46] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[47] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 284–292.

[48] Shinichi Nakagawa and Holger Schielzeth. 2013. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution* 4, 2 (2013), 133–142.

[49] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 382–391.

[50] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 153–164.

[51] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. IEEE, 380–390.

[52] Chao Ni, Xiang Chen, Fangfang Wu, Yuxiang Shen, and Qing Gu. 2019. An empirical study on Pareto based multi-objective feature selection for software defect prediction. *Journal of Systems and Software* (2019), 215–238.

[53] Chao Ni, Wang-Shu Liu, Xiang Chen, Qing Gu, Dao-Xu Chen, and Qi-Guo Huang. 2017. A cluster based feature selection method for cross-project software defect prediction. *Journal of Computer Science and Technology* 32, 6 (2017), 1090–1107.

[54] Chao Ni, Xin Xia, David Lo, Xiang Chen, and Qing Gu. 2020. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering* (2020).

[55] Frolin S. Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2016. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering* 43, 2 (2016), 128–144.

[56] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 346–356.

[57] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct. (2011), 2825–2830.

[58] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. 409–418.

[59] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A tested semantics for getters, setters, and eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*. 1–16.

[60] Michael Pradel and Koushik Sen. 2015. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP'15)*. 999–1021.

[61] Lutz Prechelt. 2000. An empirical comparison of seven programming languages. *Computer* 33, 10 (2000), 23–29.

[62] Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei, and Ahmed E. Hassan. 2017. The impact of using regression models to build defect classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. IEEE, 135–145.

[63] Frank E. Harrell. 2001. Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer.

[64] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 155–165.

[65] Stuart J. Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Pearson Education Limited, Malaysia.

[66] Duksan Ryu, Okjoo Choi, and Jongmoon Baik. 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering* 21, 1 (2016), 43–71.

[67] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in JavaScript projects. In *Proceedings of the 2017 IEEE 24th International Cnference on Software Analysis, Evolution and Reengineering (SANER'17)*. IEEE, 294–305.

[68] Andrew Jhon Scott and M. Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[69] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[70] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 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*. 1–11.

[71] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2010), 772–787.

[72] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 1–5.

[73] Tom A. B. Snijders. 2005. Fixed and random effects. *Encyclopedia of Statistics in Behavioral Science* (2005).

[74] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. 2017. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability* 66, 1 (2017), 17–37.

[75] Mark D. Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. 2014. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering* 41, 2 (2014), 176–197.

[76] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 286–295.

[77] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.

[78] A. B. Tom and Roel J. Bosker. 1999. *Multilevel Analysis: An Introduction to Basic and Advanced Multilevel Modeling*. Sage.

[79] Andrzej Tucholka and Prem Gurbani. 2010. A highly decoupled front-end framework for high trafficked web applications. In *Proceedings of the 2010 5th International Conference on Internet and Web Applications and Services*. IEEE, 32–36.

[80] Burak Turhan, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.

[81] Burak Turhan, Ayse Tosun, and Ayse Bener. 2011. Empirical evaluation of mixed-project defect prediction models. In *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11)*. IEEE, 396–403.

[82] Stefan Wagner and Emerson Murphy-Hill. 2019. Factors that influence productivity: A checklist. In *Rethinking Productivity in Software Engineering*. Springer, 69–84.

[83] Philip Walton. 2013. Decoupling your HTML, CSS, and JavaScript. (2013). https://philipwalton.com/articles/decoupling-html-css-and-javascript.

[84] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1241–1266.

[85] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[86] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. 2016. HYDRA: Massively compositional model for cross-project defect prediction. *IEEE Transactions on Software Engineering* 42, 10 (2016), 977–998.

[87] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[88] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[89] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 157–168.

[90] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 26–36.

[91] Jerrold H. Zar. 2005. Spearman rank correlation. *Encyclopedia of Biostatistics* 7 (2005).

[92] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 182–191.
[93] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2016. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* 21, 5 (2016), 2107–2145.
[94] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan. 2013. How does context affect the distribution of software maintainability metrics?. In *2013 IEEE International Conference on Software Maintenance.* IEEE, 350–359.
[95] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead, Jr. 2006. Mining version archives for co-changed lines. In *Proceedings of the 2006 International Workshop on Mining Software Repositories.* 72–75.
[96] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM, 91–100.

Received August 2020; revised October 2021; accepted December 2021