The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization

Chao Ni
School of Software Technology, Zhejiang University
Hangzhou, Zhejiang, China
chaoni@zju.edu.cn

Wei Wang†
College of Computer Science and Technology, Zhejiang university
Hangzhou, China
wangw99@zju.edu.cn

Kaiwen Yang†
College of Computer Science and Technology, Zhejiang university
Hangzhou, China
kwyang@zju.edu.cn

Xin Xia*
Software Engineering Application Technology Lab, Huawei
Hangzhou, China
xin.xia@acm.org

Kui Liu
Software Engineering Application Technology Lab, Huawei
Hangzhou, China
brucekuiliu@gmail.com

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

To improve software quality, just-in-time defect prediction (JIT-DP) (identifying defect-inducing commits) and just-in-time defect localization (JIT-DL) (identifying defect-inducing code lines in commits) have been widely studied by learning semantic features or expert features respectively, and indeed achieved promising performance. Semantic features and expert features describe code change commits from different aspects, however, the best of the two features have not been fully explored together to boost the just-in-time defect prediction and localization in the literature yet. Additional, JIT-DP identifies defects at the coarse commit level, while as the consequent task of JIT-DP, JIT-DL cannot achieve the accurate localization of defect-inducing code lines in a commit without JIT-DP. We hypothesize that the two JIT tasks can be combined together to boost the accurate prediction and localization of defect-inducing commits by integrating semantic features with expert features. Therefore, we propose to build a unified model, **JIT-Fine**, for the just-in-time defect prediction and localization by leveraging the best of semantic features and expert features. To assess the feasibility of JIT-Fine, we first build a large-scale line-level manually labeled dataset, **JIT-Defects4J**. Then, we make a comprehensive comparison with six state-of-the-art baselines under various settings using ten performance measures grouped into two types: effort-agnostic and effort-aware. The experimental results indicate that JIT-Fine can outperform all state-of-the-art baselines on both JIT-DP and JIT-DL tasks in terms of ten performance measures with a substantial improvement (i.e., 10%-629% in terms of effort-agnostic measures on JIT-DP, 5%-54% in terms of effort-aware measures on JIT-DP, and 4%-117% in terms of effort-aware measures on JIT-DL).

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**.

KEYWORDS

Just-In-Time, Defect Prediction, Defect Localization, Deep Learning

ACM Reference Format:
Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549165

1 INTRODUCTION

Software defects are unavoidable in software development, and most of them are induced by the commits submitted in the evolution of software [54]. Once software is released with defects, the various walks of human-like could be substantially affected with unacceptable consequences on finance[1] and human life [64]. Additionally, the cost of maintaining defect-contained software will be increased sharply [3, 23]. To address this challenge, one potential feasible method is to identify and fix defects as early as possible. To this end, practitioners have been exploring various approaches of identifying defects at the coarse-grained level [5, 31, 34–36, 51, 55, 66] or fine-grained level [4, 7, 11, 12, 20, 22, 24, 29, 38, 40, 61]. Meanwhile, a few approaches are also proposed to timely and finely locate where the defects exist, which can help developers better understand the code [42, 57]. These fine-grained approaches aim at assisting developers to identify the defect-inducing commits/lines submitted by developers before merging them, which is also referred to as just-in-time (JIT) techniques. The promising results achieved by existing studies indicate that JIT defect prediction and localization (JIT-DP & JIT-DL) can provide effective hints for software participants to

*Xin Xia is the corresponding author.
† Wei Wang and Kaiwen Yang contributed equally to this research.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549165

[1] https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107

identify and locate software defects in time [22, 42], especially for the limited resource scenario.

In the literature, the state-of-the-art JIT-DP techniques mainly adopt machine learning technologies to build a prediction model that is used to predict the defect-inducing commits by learning from the semantic features or the carefully curated expert features. Semantic features are mainly mined from the semantic information and syntactic structure hidden in the faulty source code to represent the semantic characteristics of defect-prone commits [17, 18, 59]. The expert features are defined by experts according to their understanding on defect-inducing commits with their professional knowledge and experience, which are taken as the input of learning based classifiers to identify the defect-inducing commits. In the literature, Kamei et al. [22] firstly defined 14 types of expert features (cf. Section 2) with five dimensions (i.e., diffusion, size, purpose and history of code changes, as well as programmers' experience), which have been widely used as the expert features to detect the defect-inducing code changes. The state-of-the-art techniques also mainly adopt machine learning technologies with token features to identify lines (JIT-DL) that are associated with the defective commits by two-stage or two-different approaches [42, 57].

A diversity of state-of-the-art research work has been proposed to boost the JIT-DP and JIT-DL tasks for developers, and have made a great progress with promising results [17, 18, 37, 42, 57, 59, 62]. However, the state-of-the-art JIT-DP and JIT-DL techniques are still limited on the granularity and accuracy of defect prediction and localization. For example, ① the low-quality datasets with tangled commits could affect the model training and further lead to the low accuracy of identified results because of the noise in both training data and testing data [14]. ② Semantic features and expert features represent different characteristics of code changes from different dimensions, which however are not explored together for defect prediction or localization yet. We infer that the best of them could be exploited together for JIT-DP and JIT-DL tasks. ③ JIT-DP and JIT-DL tasks are respectively studied as two independent techniques, while they could be designed into a unified model from their purpose of identifying the defect-inducing code lines in code change commits. To sum up, we intuitively hypothesize that just-in-time defect prediction and localization could be proceeded with a unified learning model by integrating the semantic features with expert features behind code changes.

In this paper, based on our hypothesis, we comprehensively explore the importance of combining semantic features with expert features for building a unified JIT-DP and JIT-DL model, and present a novel approach (named **JIT-Fine**) with the widely used deep learning technique, CodeBERT [8, 10, 63]. To the best of our knowledge, JIT-Fine is the first to build a unified model for JIT-DP and JIT-DL tasks. The underlying intuition of our approach is to capture the distinguishing features of (non)defect-inducing code changes and their contexts by integrating semantic features with expert features, and consequently to identify the defective commits as well as locate the defect position at line level. More specifically, JIT-Fine mainly consists of three steps: ❶ extracting expert features and semantic features from the code changes and their contexts with pre-trained models, ❷ learning integrated features between semantic and expert features to distinguish commits from defect-inducing ones to defect-free ones, and ❸ predicting whether a commit is defect-inducing and locating which code line induce the defect in a commit. Eventually, this paper makes the main contributions as below:

- **Dataset JIT-Defects4J.** We build a large-scale line-level labeled dataset JIT-Defects4J on the top of LLTC4J, a dataset collected by Herbold et al. [14] for analyzing the tangled bug fixing commits, to support the research of just-in-time defects and to fill the gap of the low-quality dataset in the JIT-DP and JIT-DL community.
- **JIT-Fine.** We implement a unified just-in-time defect prediction and localization technique, JIT-Fine, with a unified learning model to mine the distinguishing features from the semantic features and expert features of code changes and their contexts. The replication package of JIT-Fine and the aforementioned dataset JIT-Defects4J are publicly available.[2]
- **Just-in-time defect prediction and localization.** We comprehensively investigate the value of integrating the expert features and semantic features for just-in-time defect prediction and just-in-time defect localization. The results indicate that JIT-Fine outperforms the state of the art (e.g., a higher F1-measure by 14-37 percentage points on defect prediction, and a higher Top-10 accuracy by 2-12 percentage points).
- **Analyzing the effectiveness of combined features.** We analyze the effectiveness of semantic features and expert features on JITLine and JIT-Fine, and the experimental results uncover that the combination of semantic and expert features presents obvious advantages on just-in-time defect prediction than the usage of single ones.

The rest of this paper is organized as follows. Section 2 first introduces the background and motivation of our work. Following that, Section 3 introduces the design of JIT-Fine. Section 4 describes the details of studied dataset. Section 5 presents the experimental setting including compared baselines and considered performance measures. Section 6 reports the experimental results. Following that, some threats to validity are presented in Section 7. Section 8 describes related prior work. Finally, we conclude our work and mention future plan in Section 9.

2 BACKGROUND AND MOTIVATION

2.1 Expert Features and Semantic Features

This section clarifies the notions of expert features and semantic features related to the code changes.

Expert Features are the descriptive features defined by experts according to their understanding of defect-inducing commits with their professional knowledge and experience, to measure the quality of code changes. In the literature, dozens of expert features [22, 37, 45] have been defined at change-level from various dimensions (e.g., diffusion, size, purpose, history, experience, review, programming language). The 14 features (presented in Table 1) defined by Kamei et al. [22] from five dimensions are widely adopted in the domain of just-in-time defect prediction and have been demonstrated for their effectiveness on such prediction tasks [42, 50, 62]. Therefore, we follow the state-of-the-art work [42, 50, 62] to adopt the 14 expert features as an important part to mine the integrated features of code changes for just-in-time defect prediction and localization.

[2] https://github.com/jacknichao/JIT-Fine

Table 1: Studied 14 basic change-level features.

Name	Description	Dimension
NS	The number of modified subsystems	Diffusion
ND	The number of modified directories	
NF	The number of modified files	
Entropy	Distribution of modified code across each file	
LA	Lines of code added	Size
LD	Lines of code deleted	
LT	Lines of code in a file before the change	
FIX	Whether or not the change is a defect fix	Purpose
NDEV	The number of developers that changed the modified files	History
AGE	The average time interval between the last and current change	
NUC	The number of unique changes to the modified files	
EXP	Developer experience	Experience
REXP	Recent developer experience	
SEXP	Developer experience on a subsystem	

Semantic Features represent the theoretical units of meaning-holding components which are used for representing word meaning [1], and play a crucial role in determining the lexical relationships between words in a language. In the domain of software engineering, semantic features capture the meaning of tokens in code as well as their contexts (e.g., semantic and syntactic structural information of code), that have been widely used to represent the intrinsic characteristics of code mined with deep learning techniques, like CodeBERT [10]. CodeBERT aims at learning contextual word embedding (i.e., the embedding of a word changes dynamically according to the context in which it appears) and has been widely used in multiple natural language processing (NLP) tasks [43] and software engineering (SE) tasks [63]. In this work, we extract semantic features of code changes by leveraging the popular pre-trained model, CodeBERT, and fine-tune it on commits for adapting it to downstream tasks.

2.2 Motivation

In the literature, just-in-time defect prediction and localization have been attracting more and more attention to boost just-in-time software quality analysis, and indeed has achieved promising results for developers to automatically identify defects in code change commits from coarse-grain granularity [31, 34, 51, 66] to fine-grained granularity [7, 12, 22, 24, 40]. Nevertheless, just-in-time defect prediction and localization is still an open question because of various limitations (e.g., low-quality dataset, independently mined features, and non-unified models), which motivates us to conduct this study to boost the development of just-in-time defect prediction and localization.

Low-quality datasets. The state-of-the-art JIT-DP and JIT-DL techniques are highly dependent on accurate information about code changes in the related datasets. Unfortunately, these techniques suffer from the low-quality dataset (e.g., tangled commits mislabeled by practitioners). In practice, a single commit could be tangled with several kinds of code changes, e.g., bug fix, code refactoring, testing, new features, and documentation. In addition, commits tangled with several bug fixes can heavily affect the accuracy of data extraction and labeling with the related algorithm SZZ [48] and its variants [46], which subsequently has negative impacts on JIT-DP and JIT-DL tasks since noise data could bias the learning results of corresponding models. Unfortunately, almost all prior studies were proposed based on such a low-quality dataset [9, 22, 42, 59]. To address the challenge of the low-quality dataset, more and more large-scale and accurate datasets are collected by researchers. For JIT-DL task, Yan et al. [57] and Pornprasit et al. [42] respectively built line-level datasets on the top of SZZ labeled dataset. For JIT-DP, Rosa et al. [46] used the natural language processing technique to identify bug-fixing commits by utilizing developers' information (i.e., developers explicitly referenced that the commits fixed bugs) to build a large-scale dataset with accurate bug-fixing commits. Similarly, Zeng et al. [62] built a large-scale dataset with the data labeling algorithm SZZ. However, these datasets still do not consider the side-effects of tangled bug-fixing commits (i.e., a single commit is tangled with several bug fixes). **To fill this gap, we build a large-scale high-quality line-level dataset JIT-Defects4J to alleviate the effects of tangled commits on the basis of LLTC4J [14], which can be utilized in JIT-DP and JIT-DL tasks.**

Independently mined features. Semantic features reflect the intrinsic characteristics of code changes and their contexts, which have been mined by practitioners to build the just-in-time defect prediction model with deep learning techniques (e.g., DeepJIT [17] and CC2Vec [18]). Expert features are defined on the basis of the participants' knowledge and experience, that are the extrinsic understanding from human. Researchers proposed to utilize expert features (e.g., EALR [22], CBS+ [21], OneWay [11], Churn [28], and LApredict [62]) to conduct the just-in-time defect prediction task. More recently, Pornprasit and Tantithamthavorn [42] proposed a just-in-time defect localization technique, JITLine, by considering code token features (ignoring token order as well as the deeper semantic features) and a few expert features. These research work has achieved promising results on just-in-time defect prediction and localization. Actually, semantic features and expert features represent distinguishing features of code from the aspect of expert knowledge and intrinsic structure of code, respectively. However, the state-of-the-art research work did not explore the integration of semantic features and expert features for JIT-DP and JIT-DL tasks yet. *Can a combination of both kinds of features achieve higher performance?* With this question, **we intuitively hypothesize that leveraging the best of semantic and expert features could be used to improve the just-in-time defect prediction and localization.** Eventually, our work fills this gap by investigating the combination of semantic features and expert features to verify our intuitive hypothesis.

Independently built models. Just-in-time defect prediction and localization have been studied in various ways to support the software quality assurance. Existing works either treat the two tasks independently [37, 56, 65] or address the two tasks with two-stage/two-different approaches [42, 57]. For example, Yan et al. [57] proposed a two-phase work for JIT-DP and JIT-DL. Specially, they build two independent models for the two tasks respectively: prediction model with expert features for JIT-DP and N-gram model for JIT-DL. Besides, Pornprasit and Tantithamthavorn [42] proposed

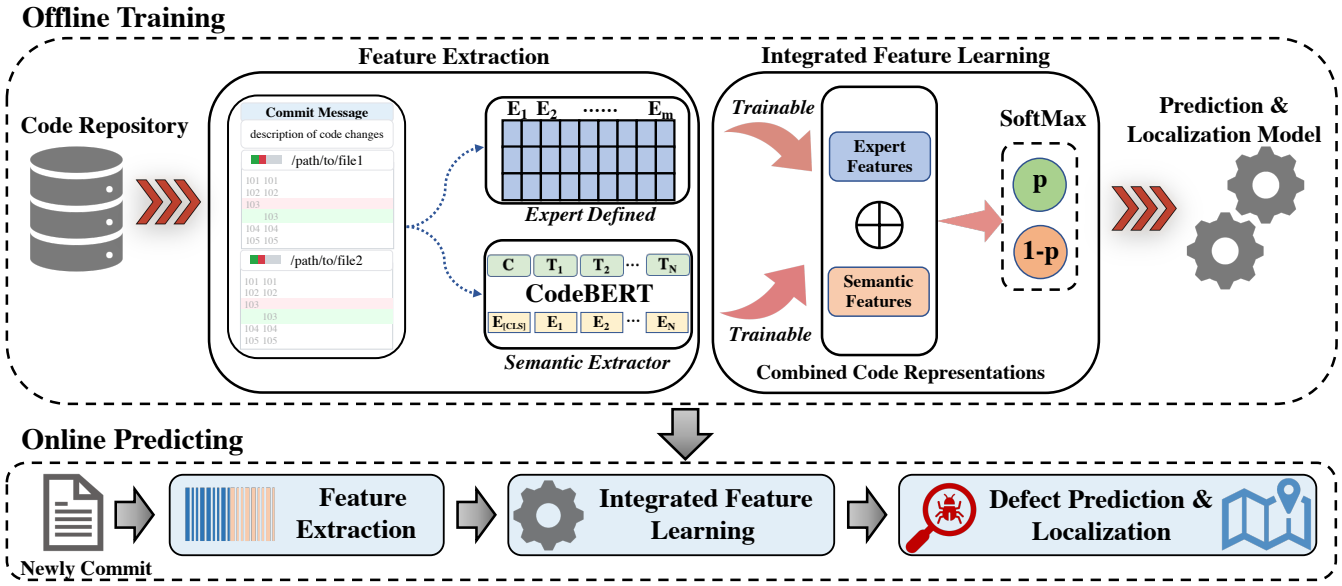

Figure 1: An overview of our approach JIT-Fine of predicting and localizing defects.

the JITLine model to address the JIT-DP task with a prediction model by combining expert features with code token features, and to address the JIT-DL task with another model LIME [27] by calculating the contribution score of each token for its defect-inducing interpretation. Moreover, the existing JIT-DL techniques can identify the defect-inducing code changes at the fine-grained code line level, but cannot obtain the high accuracy on identifying the exact defect-inducing code lines in code changes without considering the identification of defect-inducing commits [42, 57] (e.g., the most recent state-of-the-art JITLine [42] can only achieve 10.4% accuracy for the identified top-5 most suspicious defect-inducing code lines in code changes in our dataset). *Can the JIT-DP and JIT-DL tasks be integrated into a unified model to improve their performance?* **This work is thus conducted to investigate the feasibility of designing a unified model to proceed the JIT-DP and JIT-DL tasks.**

3 JIT-FINE: JUST-IN-TIME DEFECT PREDICTION AND LOCALIZATION

To investigate the feasibility of our intuitive hypothesis, we propose a unified approach, **JIT-Fine**, that integrates the semantic features with expert features for the just-in-time defect prediction and localization. As illustrated in Figure 1, JIT-Fine consists of three main steps: ❶ feature extraction, where expert features are extracted by adopting the widely used 14 change-level features defined by Kamei et al. [22] and semantic features are extracted with the popular pre-trained model CodeBERT; ❷ integrated feature learning is proceeded with a fully connected layer, and ❸ defect prediction and localization is to predict the defect-inducing commits and locate defective code lines with previously learned features. Details of JIT-Fine are presented in the following subsections.

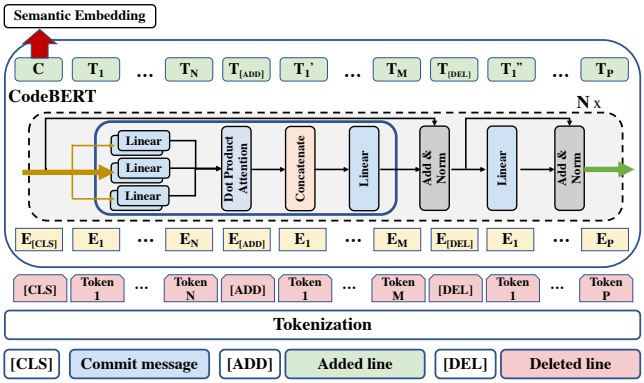

Figure 2: The structure of semantic feature extractor.

3.1 Feature Extraction

Feature extraction aims at converting the statistical data and code tokens into numeric representation that can be adapted to the deep neural network models to capture the distinguishing characteristics for the later prediction and localization tasks. In the community, semantic feature extraction mainly relies on the pre-trained model CodeBERT [10] that aims at learning contextual word embedding (i.e., the embedding of a word changes dynamically according to the context in which it appears) and has been widely used in multiple natural language processing (NLP) tasks [43] and software engineering (SE) tasks [63]. Therefore, we also leverage CodeBERT to extract the semantic features for code changes.

The structure of the semantic feature extractor in JIT-Fine is shown in Figure 2, which takes as input three types of information (e.g., commit message, added lines, deleted lines). "*commit message*" represents the description of the submitted commit, "*added line*" and "*deleted line*" represent the lines added and deleted in the commit, respectively. For the classification task, a special token

"[CLS]" is always added in front of each input instance to obtain the semantic representation of the whole sentence [8], we follow this workaround and add the token "[CLS]" before the commit message. In addition, we add the other two tokens (i.e., [ADD] and [DEL]) before the "*added line*" and the "*deleted line*" to differentiate them. Then, commit message, added lines, deleted lines are tokenized into a token sequence that is fed into the CodeBERT model to generate the corresponding embedding vector that is referred to as the semantic vector of a code change commit. CodeBERT [10] is designed for the general software engineering tasks, we thus fine-tune it on commits to adjust downstream tasks by following the workaround verified in the prior work [39, 63]. For expert features, we directly adopt the widely used 14 change-level metrics proposed by Kamei et al. [22] to extract the expert features from code change commit by directly using the CommitGuru [47] that is an online service of automatically extracting and labeling commit-level datasets.

3.2 Integrated Feature Learning

With the semantic and expert features extracted in different ways, JIT-Fine further needs to learn their integrated features. In the literature, learning integrated features can be proceeded in different ways (e.g., building a model by simply combining expert features and semantic features [42, 57], or training semantic features jointly with expert features to build a model [19, 39, 41]). Recently, Pan et al. [39] built a model by training semantic features jointly with expert features to automatically identify the information type in developer chatrooms with a promising result. Therefore, we also leverage the straightforward way of jointly training the semantic features with the expert features. Note that, the initial expert feature vector only contains 14 numeric elements extracted with the 14 feature factors, while the semantic feature vector generated by CodeBERT has 768 numeric elements. To avoid that the semantic feature could overwhelm the expert feature, and to treat the two kinds of features equally, we follow Yang et al.'s [59] work to obtain the high-dimensional representation vectors for expert features with the deep learning technology (a fully connected layer is used in this study). The extended expert feature vector is denoted as V_{EF}, and the semantic vector outputted by CodeBERT is denoted as V_{SF}. JIT-Fine concatenates the two vectors (i.e., V_{EF} and V_{SF}) to generate a new one (denoted as V_F) for constructing the features of the code changes in a given commit and fine-tune them together with another fully connective layer during the training phase.

3.3 Defect Prediction and Localization

With the learned integrated features, the last step of JIT-Fine is to train a defect prediction and localization model that will be applied to commits under review. To this end, the learned integrated features are first fed into one fully connected layer for the binary classification task. The model is trained by iterating it on all training datasets, monitoring the loss function, and optimizing the weights of feature relationship by back propagation mechanism. Such progress is executed by a few epochs (i.e., at most 50 times since we optimize our model by early stop strategy). Meanwhile, based on the *attention mechanism* in CodeBERT, we can utilize the weights of each input code token to locate where the defect exists.

For making the decision of a given commit, we can not only obtain the corresponding label but also obtain its defect density, which can be calculated as the ratio between the probability outputted by the model ($Y(c)$) and the total modified lines of that commit ($\#LOC(c)$). The defect density of a commit is a good metric since different commit needs different costs of applying quality assurance activities [22, 30]. For localizing the defect-inducing line in each commit, we calculate the contribution (i.e., weights) of each token in modified code to the final classification task and summary up all tokens' contributions in one line. Finally, we rank defective lines that are associated with a given commit based on the total contribution.

4 CLEAN DATASET CONSTRUCTION

As presented in Section 2.2, the JIT-DP and JIT-DL community suffer from the low-quality dataset with tangled commits [15, 32, 37, 42, 57, 62], especially a commit tangled with several kinds of code changes [16], e.g., bug fix, code refactoring, testing, new features and documentation. Therefore, we build a clean line-level dataset for both JIT-DP and JIT-DL on the basic of LLTC4J [14] to alleviate the impact of tangled commit on the evaluation of different approaches.

LLTC4J, Line-Labelled Tangled Commits for Java, is manually built by Herbold et al. [14], which only focus on bug-fixing commits in Java projects. In particular, they manually validated 2,328 bugs from 28 projects and these bugs are fixed by 3,498 commits. To decrease the high degree of uncertainty when determining whether a commit is tangled or not [25, 26], they first recruited 45 participants with two criteria (i.e., majoring in computer science or a closely related subject and at least one-year of programming experience in Java). Then, they assigned each commit to four different participants to label the data independently. Each commit was shown to four participants and the consensus of each line in a commit can be achieved if at least three participants agreed on the same label. Otherwise, no consensus for the line was achieved. Following that, they used the agreement among the participants to decrease the uncertainty involved in the labeling. In particular, each modified line in a commit can be labeled as one of the following types: 1) *contributing to the bug-fixing*; 2) *changing whitespace*; 3) *changing documentation*; 4) *changing test*; 5) *unrelated improvement not required for the bug-fixing*; and 6) *no consensus*.

LLTC4J is a good starting-point for collecting a high-quality, fine-grained, comprehensive dataset for just-in-time defect prediction and line-level localization research. However, as aforementioned, it only collects the bug-fixing commits and labels the lines in each bug-fixing commit. Therefore, we extend this dataset from two-sides: 1) extracting both clean commits and buggy commits; 2) extracting the line label in defect-introducing commits. Just-in-time defect prediction is typically treated as a binary-classification task, the dataset should contain both clean instances and buggy instances for building a model. Besides, to investigate the effectiveness of just-in-time defect localization model, we need the ground truth label for each line in a commit. Therefore, the former one enables us to construct a comprehensive dataset for just-in-time defect prediction research, while the latter one enables us to construct a line-level bug-introducing or bug-free dataset for just-in-time defect

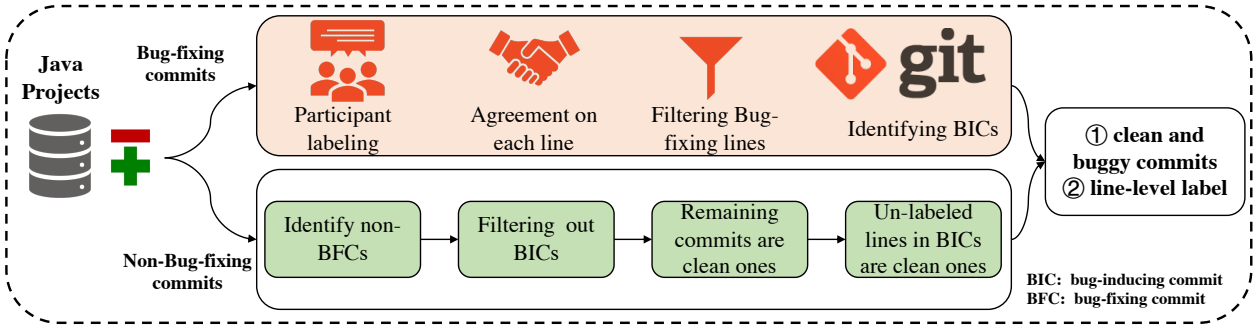

Figure 3: The process of data extension and line-level labeling.

Table 2: Statistics of studied datasets: JIT-Defects4J.

Java Project	Timeframe	Commit-level			Line-level		
		# BC	# CC	% Ratio (Bugs / ALL)	# BL	# CL	% Ratio (Bugs / ALL)
ant-ivy	2005-06-16 - 2018-02-13	332	1,439	18.75% (332 / 1,771)	1,650	14,853	10.00% (1,650 / 16,503)
commons-bcel	2001-10-29 - 2019-03-12	60	765	7.27 % (60 / 825)	310	1,626	16.01% (310 / 1,936)
commons-beanutils	2001-03-27 - 2018-11-15	37	574	6.06 % (37 / 611)	123	1,724	6.66 % (123 / 1,847)
commons-codec	2003-04-25 - 2018-11-15	36	725	4.73 % (36 / 761)	246	2,074	10.60% (246 / 2,320)
commons-collections	2001-04-14 - 2018-11-15	50	1,773	2.74 % (50 / 1,823)	181	3,434	5.01 % (181 / 3,615)
commons-compress	2003-11-23 - 2018-11-15	178	1,452	10.92% (178 / 1,630)	627	7,167	8.04 % (627 / 7,794)
commons-configuration	2003-12-23 - 2018-11-15	155	1,683	8.43 % (155 / 1,838)	650	7,334	8.14 % (650 / 7,984)
commons-dbcp	2001-04-14 - 2019-03-12	58	979	5.59 % (58 / 1,037)	192	2,500	7.13 % (192 / 2,692)
commons-digester	2001-05-03 - 2018-11-16	19	1,060	1.76 % (19 / 1,079)	87	385	18.43% (87 / 472)
commons-io	2002-01-25 - 2018-11-16	73	1,069	6.39 % (73 / 1,142)	196	2,822	6.49 % (196 / 3,018)
commons-jcs	2002-04-07 - 2018-11-16	88	743	10.59% (88 / 831)	450	4,796	8.58 % (450 / 5,246)
commons-lang	2002-07-19 - 2018-10-10	146	2,823	4.92 % (146 / 2,969)	563	6,332	8.17 % (563 / 6,895)
commons-math	2003-05-12 - 2018-02-15	335	3,691	8.32 % (335 / 4,026)	3,046	16,960	15.23% (3,046 / 20,006)
commons-net	2002-04-03 - 2018-11-14	117	1,004	10.44% (117 / 1,121)	441	4,834	8.36 % (441 / 5,275)
commons-scxml	2005-08-17 - 2018-11-16	47	497	8.64 % (47 / 544)	242	4,054	5.63 % (242 / 4,296)
commons-validator	2002-01-06 - 2018-11-19	36	562	6.02 % (36 / 598)	114	1,189	8.75 % (114 / 1,303)
commons-vfs	2002-07-16 - 2018-11-19	114	996	10.27% (114 / 1,110)	384	4,902	7.26 % (384 / 5,286)
giraph	2010-10-29 - 2018-11-21	163	681	19.31% (163 / 844)	1,904	16,748	10.21% (1,904 / 18,652)
gora	2010-10-08 - 2019-04-10	39	514	7.05 % (39 / 553)	133	3,397	3.77 % (133 / 3,530)
opennlp	2008-09-28 - 2018-06-18	91	995	8.38 % (91 / 1,086)	326	3,912	7.69 % (326 / 4,238)
parquet-mr	2012-08-31 - 2018-07-12	158	962	14.11% (158 / 1,120)	729	8,325	8.05 % (729 / 9,054)
ALL		**2,332**	**24,987**	**8.54 % (2,332 / 27,319)**	**12,594**	**119,368**	**9.54 % (12,594 / 131,962)**

"BC" refers to "Buggy Commit", "CC" refers to "Clean Commit", "BL" refers to "Buggy Line", and "CL" refers to "Clean Line".

localization research. The data extraction and extension process is illustrated in Figure 3.

For identifying bug-introducing commits, we start from all bug-fixing commits in the original dataset. Those commits have at least one agreed "*contributing to the bug-fixing*" line, which means the line is labeled by at least three participants with same label and can be selected as candidate bug-fixing commits. For each "*contributing to the bug-fixing*" line in the candidate bug-fixing commits, we use *git blame* to find its corresponding bug-introducing commit and also label the corresponding lines which are modified in bug-fixing commit as buggy line. Such an operation can greatly reduce the scope of defect-introducing candidates and accurately label the line of code. The remaining commits (i.e., not classified as bug-introducing commits) are treated as clean ones.

We also need to collect the modification in hunks[3] of each commit. We use PyDriller [49] to extract the corresponding code changes and commit messages. For the purpose of high-quality dataset, we set a few criteria to filter some extremities according to the suggestion in [29]: 1) keeping the code changes made to Java files only and filtering those non-functional code changes (e.g. comments); 2) ignoring large commits, which change more than 10,000 lines of code or change more than 100 files since those commits are likely noise caused by routine maintenance (e.g., copyright updates); 3) ignoring changes that do not add any new lines since the SZZ algorithm has an assumption that defects are introduced by adding new lines.

Notice that we remove five projects from the original dataset[4] since there have insufficient participants to label the commits in

[3] https://git-scm.com/
[4] https://smartshark.github.io/dbreleases/

these projects. Besides, we also filter out another two projects with some issues, where many commits cannot be found in their corresponding repositories downloaded from GitHub. Finally, 21 Java projects are used for this study and we name the extension of LLTC4J as JIT-Defects4J for easier reference. The statistical information of the final dataset can be found in Table 2.

As shown in Table 2, we analyze two statistical information of our studied dataset: commit-level and line-level. JIT-Defects4J has a varying number of commits ranging from 544 to 4,026, and its bug ratio is 1.76%-18.75%. It also has a varying number of lines of codes ranging from 472 to 20,006, and its bug ratio is 3.77%-18.43%.

Apart from extracting the modified lines (i.e., added lines and deleted lines) from commits, we also extract the 14 change-level defect features (presented in Table 1) from five dimensions (i.e., diffusion, size, purpose, history and experience) as proposed by Kamei et al. [22] with CommitGuru [47], which are widely used in just-in-time defect prediction scenario [6, 42, 60].

5 EXPERIMENTAL SETTINGS

In this section, we first briefly introduce the studied baselines and present the two types of performance measures.

5.1 Baselines

To comprehensively evaluate the performance of JIT-Fine with prior work, in this paper, we totally consider six state-of-the-art approaches (i.e., LApredict [62], Deeper [59], DeepJIT [17], CC2Vec [18], Yan et al.'s work [57] and JITLine [42]). As presented in Table 3, LApredict [62] aims at building a defect prediction model by leveraging the information of "lines of code added" expert feature with the traditional logistic regression classifier. Deeper [59] focuses on building a defect prediction model by learning expert features with the deep belief networks. DeepJIT [17] is to build a defect prediction model with convolutional neural networks by learning the semantic features from the commit message and corresponding code changes. CC2Vec [18] relies on the hierarchical attention network to learn semantic features for the defect prediction. These works only consider the expert features or semantic features to address the JIT defect prediction task. Yan et al. [57] explore the expert features to build a defect prediction model with the logistic regression and building a defect localization model with N-gram technique, respectively. While JITLine [42] leverages the combination of expert features and token features to build a defect prediction model with random forest classifier and leverages the token features to build a defect localization model with the LIME model. Different from the existing work, JIT-Fine aims at building a unified model for defect prediction and localization by exploring the best of the semantic features and expert features.

Table 3: Six baselines used in the comparison.

Baselines	Features			Model		JIT Tasks		Venue
	EF	SF	TF	TM	DLM	DP	DL	
LApredict [62]	✔			✔		✔		ISSTA 2021
Deeper [59]	✔				✔	✔		QRS 2015
DeepJIT [17]		✔			✔	✔		MSR 2019
CC2Vec [18]		✔			✔	✔		ICSE 2020
Yan et al. [57]	✔		✔	✔		✔*	✔*	TSE 2020
JITLine [42]	✔		✔	✔		✔*	✔*	MSR 2021
JIT-Fine	✔	✔			✔	✔	✔	This work

*"EF": Expert Feature, "SF": Semantic Feature, "TF": Token Feature, "TM": Traditional Machine Learning Model, and "DLM": Deep Learning Model.
✔* means that the related technique address the JIT-DP and JIT-DL tasks with two independently ways.

5.2 Evaluation Measures

To evaluate the effectiveness of JIT-Fine, we adopt two types of widely used performance measures: effort-agnostic performance measures and effort-aware performance measures.

Effort-agnostic Performance Measures. Effort-agnostic performance measures evaluate the prediction performance without considering cost, which means SQA team has enough resources to inspect all potential defective lines of code.

This group considers two widely used performance measures: F1-score and AUC [36, 42, 59]. There are four possible prediction results for a commit in the testing dataset: A commit can be predicted as defective one when it is truly defective (true positive, TP); it can be predicted as defective one when it is actually clean (false positive, FP); it can be predicted as clean one when it is actually defective (false negative, FN); or it can be predicted as clean one and it is truly clean (true negative, TN). Therefore, based on the four possible results, F1-score can be defined as follows:

F1-score: is a harmonic mean of $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$. It is computed as $F1\text{-}score = \frac{2 \times Precision \times Recall}{Precision+Recall}$. It is often used as a summary measure to evaluate if an increase in precision outweighs a reduction in recall and vice versa.

AUC: represents the area under the receiver operating characteristic (ROC) curve [13], which is a 2D illustration of true positive rate (TPR) on the y-axis versus false positive rate (FPR) on the x-axis. ROC curve is generated by varying the classification threshold over all possible values, which can separates clean and buggy predictions. AUC ranges from 0 to 1, and a good prediction model can obtain an AUC value close to 1. The ROC analysis is robust especially for imbalanced class distributions and asymmetric misclassification costs. It also represents the probability that a model ranks a randomly chosen defective instance higher than a randomly chosen clean one.

Effort-aware Performance Measures. Effort-aware performance measures evaluate the prediction performance by considering the given cost threshold, e.g., a certain number of lines of code to inspect. Such type of performance measures are extremely important especially when SQA team has limited resources to inspect potential defective lines of code. Developers want to discover as many defects as possible by manually inspecting the top percentages of lines that are likely to be defective. Similar to prior work [36, 42], we use the 20% of total lines of code as the proxy of inspection effort.

This group totally considers seven widely used performance measures [36, 42, 57], which can be sub-divided into two groups: performance measures for identifying defect-prone commits and performance measures for localizing defects in commits. In this former group, three performance measures (i.e., Recall@20%Effort, Effort@20%Recall and P_{opt}) are considered, while four performance

measures (i.e., Top-N, Recall@20%Effort$_{line}$, Effort@20%Recall$_{line}$ and IFA$_{line}$) are considered in the latter group.

Group 1: Performance measures for identifying defect-prone commits.

Recall@20%Effort(R@20%E): measures a proportion between the actual number of defect-introducing commits found with given inspection effort and the total number of commits. In this paper, similar to prior work [36, 42], we use the line of code (LOC) as the proxy of inspection effort, i.e., the 20% LOC of the whole project. A high value of R@20%E means more actual defect-introducing commits are ranked at the top of list to be inspected. Therefore, developers will find more actual defect-introducing commits with less effort.

Effort@20%Recall(E@20%R): measures the amount of effort that developers have to spend when 20% actual defect-introducing commits in testing dataset are found. A low value of E@20%R means developers will find the 20% actual defect-introducing commits with less effort.

P$_{opt}$: is on the basic of the concept of the Alberg diagram [2] which indicates the relationship between the Recall obtained by a prediction model and the inspection effort for a specific prediction model. To compute this measure, two additional prediction models are required: the optimal one and the worst one. In the optimal model and the worst model, commits are sorted in decreasing and ascending order by defect densities, respectively. A good prediction model is expected to perform better than the random one and approximate the optimal one. For a given prediction model M, the p_{opt} can be calculated as: $1 - \frac{Area(Optimal)-Area(M)}{Area(Optimal)-Area(Worst)}$, where $Area(M)$ represents the area under the curve corresponding to the model M. In this paper, the defect density of commit is estimated as $\frac{Y(m)}{\#LOC(c)}$.

Group 2: Performance measures for localizing defects in commit.

Top-N Accuracy: measures the proportion of actual defective lines that are ranked in the top-N (N={5,10} in our study) ranking. In general, developers need to inspect all modified lines for a given commit. However, it is not a ideal situation especially for limited SQA resources. Therefore, a high Top-N accuracy means more actual defective lines are ranked at the top.

Recall@20%Effort$_{line}$(R@20%E$_{line}$): measures the proportion of defective lines which can be found (i.e., correctly predicted) with a given effort, i.e., the top 20% loc of modified lines of a given defect-introducing commit. A high value of R@20%E$_{line}$ means more actual defective lines can be ranked at the top.

Effort@20%Recall$_{line}$(E@20%R$_{line}$): measures the percentage of effort that developers spend to find the actual 20% defective lines for a given defect-introducing commit. A low value of E@20%R$_{line}$ means the developers can find the 20% actual defective lines with a little amount of effort.

Initial False Alarm Lines (IFA$_{line}$): measures the number of clean lines before developers find the first actual defective line for a given commit. A low value of IFA$_{line}$ means that developers can find the first actual defective line by inspecting a few number of clean lines.

6 EXPERIMENTAL RESULTS

To investigate the feasibility of JIT-Fine with integrated features on the just-in-time defect prediction and localization, our experiments focus on the following three research questions:

- **RQ-1.** *To what extent just-in-time defect prediction performance can JIT-Fine achieve?*
- **RQ-2.** *How do the integrated semantic and expert features affect the performance of JIT defect prediction models?*
- **RQ-3.** *Can JIT-Fine with the unified model be used to proceed just-in-time defect localization accurately?*

6.1 [RQ-1]: Just-In-Time Defect Prediction

Objective: Various advanced approaches have been proposed for JIT defect prediction, which either uses expert change-level features (e.g., 14 change-level features) or uses semantic features (e.g., learned with deep learning techniques) to build a prediction model in various settings (i.e., effort-aware setting or effort-agnostic setting). JIT-Fine integrates expert features and semantic features for the just-in-time defect prediction. Therefore, we investigate to what extent the prediction performance can JIT-Fine achieve with the integrated features.

Experiment Design: We totally consider six state-of-the-art baselines: LApredict [62], Yan et al. [57], DeepJIT [17], CC2Vec [18], Deeper [59], and JITLine [42]. Besides, to comprehensively compare the performance among baselines and JIT-Fine, we consider five widely used performance measures from two types: effort-agnostic ones and effort-aware ones. Considering the heavy impact of time as proposed by McIntosh and Kamei [29], we follow the same time-aware strategy to build the training data and testing data from the dataset as prior work does [17, 18, 42]. Specifically, for each project, we first sort all commits by their timestamp in ascending. Then, the top 80% of commits in each project are treated as training data, while the rest 20% of commits in each project are treated as test data. We also keep the distribution as same as the original ones in training and testing data. Finally, the training data from each project are combined into the target training data, so does the testing data. As for CC2Vec, we also retrain the model without any information from the testing dataset according to Pornprasit et al.'s [42] suggestion.

Results: The evaluation results are reported in Table 4. The best performances are also highlighted in bold. According to the results, we find that our approach JIT-Fine outperforms all baselines

Table 4: Defect prediction results of JIT-Fine compared against six baselines.

Methods	F1-score$^{\uparrow}$	AUC$^{\uparrow}$	R@20%E$^{\uparrow}$	E@20%R$^{\downarrow}$	Popt$^{\uparrow}$
LApredict	0.059	0.694	0.625	0.020	0.814
Yan et al.	0.062	0.675	0.615	0.022	0.819
Deeper	0.246	0.682	0.638	0.021	0.827
DeepJIT	0.293	0.775	0.676	0.014	0.860
CC2Vec	0.248	0.791	0.676	0.014	0.861
JITLine	0.261	0.802	0.705	0.015	0.883
JIT-Fine	**0.431**	**0.881**	**0.773**	**0.010**	**0.927**

*R@20%E: Recall@20%Effort; E@20%R: Effort@20%Recall. The outperforming results are highlighted in bold. '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better', the same as Tables 5 and 6.

methods on all performance measures. In particular, as for effort-agnostic performance measures, JIT-Fine obtains 0.431 and 0.881 in terms of F1-score and AUC, which improves baselines by 47%-629% and by 10%-30% in terms of F1-score and AUC, respectively. As for effort-aware performance measures, JIT-Fine obtains 0.773, 0.010, and 0.927 in terms of Recall@20%Effort, Effort@20%Recall, and P_{opt}, which improves baselines by 10%-26%, 26%-54%, and 5%-14% in terms of Recall@20%Effort, Effort@20%Recall, and P_{opt}, respectively. Besides, by comparing the performance of LApredict with other baselines, we find that building a simple supervised model cannot better capture the characteristics of defect-inducing commits. Moreover, we can also observe that the prediction models, from the top one (LApredict) to the bottom one (JIT-Fine), can achieve better performance as they use more and more information and build with a more and more complex model in most cases.

✍ **RQ-1** ► *JIT-Fine outperforms the state-of-the-art baselines on the just-in-time defect prediction, especially achieving the overwhelming results at F1-score. It indicates that the unified model learning the integrated semantic and expert features can achieve better performance on just-in-time defect prediction than the independent models with single feature.*◄

6.2 [RQ-2]: Effectiveness of Integrated Features

Objective: Expert features and semantic features are extracted considering different aspects of code change commits. The expert features carry the characteristics of commits based on expert professional knowledge and experience, while semantic features present the intrinsic characteristics of code change from their semantic and syntactic structural contexts, that are often pre-trained on large-scale source code datasets. Therefore, in this research question, we explore how the integrated semantic and expert features affect the effectiveness of just-in-time defect prediction.

Experiment Design: We set three training scenarios (i.e., semantic features, expert features, and their integration) to train JIT-Fine for assessing the effectiveness of integrated features on boosting defect prediction. The experimental dataset is set the same as the experiment of RQ-1 (i.e., 80% for training and 20% for testing). Moreover, we also consider two types of performance measures (i.e., effort-agnostic and effort-aware) for comprehensively studying the impact of each individual type of feature. In addition, JITLine utilizes the two types of features, i.e., expert features and token features, where the token features can be considered as a kind of coarse-grained semantic features as JITLine leverages the bag-of-words technology to extract the semantic-representative information. We thus consider JITLine as a baseline for this research question.

Table 5: Comparing results on defect prediction with different features.

Methods	Setting	F1-score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑
JITLine	EF	0.147	0.672	0.617	0.024	0.818
	SF_t	0.191	0.783	**0.722**	**0.012**	**0.894**
	$EF+SF_t$	**0.261**	**0.802**	0.705	0.015	0.883
JIT-Fine	EF	0.230	0.661	0.632	0.020	0.825
	SF	0.375	0.856	0.741	0.015	0.917
	EF+SF	**0.431**	**0.881**	**0.773**	**0.010**	**0.927**

Results: The comparison results are reported in Table 5 and the best performances are highlighted in bold for each approach on three different settings: EF (using expert features only), SF (using semantic features only), and EF+SF (combining expert features and semantic features). According to the results, we can obtain the following observations: 1) Both expert features and semantic features have their own advantages in building a prediction model. 2) Semantic features seem to have a better understanding of code characteristics than expert features in the domain of defect prediction. 3) JIT-Fine can better utilize the advantages from both expert features and semantic features to build a performance-better prediction model on both effort-agnostic and effort-aware settings. 4) JITLine performs better on effort-agnostic settings while performing badly on effort-aware settings when combining expert features and semantic features. 5) Complex models (such as deep learning models) have a better ability to learn knowledge from input information than relatively simple traditional machine learning models if they have the identical input information when we compare JITLine and JIT-Fine on EF and SF settings, independently.

✍ **RQ-2** ► *Semantic features and expert features present their own advantages in identifying defect-inducing commits. Integrating the best of them can help JIT-Fine and JITLine achieve better performance on defect prediction than the model fed with single features.* ◄

6.3 [RQ-3]: Just-In-Time Defect Localization

Objective: Localizing the defect that exists in modified codes can help developers better understand these issues and help developers faster address defects with fewer efforts. However, there exists a few machine learning approaches for fine-grained JIT defect prediction at the line level [42, 57]. Different from JITLine [42] and Yan et al's work [57], JIT-Fine is to build a unified model for addressing defect prediction and defect localization, simultaneously. In this research question, we thus investigate whether JIT-Fine with the unified model can be used to proceed just-in-time defect localization accurately.

Experiment Design: To evaluate the effectiveness of JITLine, N-gram, and JIT-Fine, we firstly need to label the high-quality line-level ground-truth dataset. Low-quality datasets, especially built on tangled datasets, can heavily affect the evaluation of different approaches' performance on defect localization. Therefore, as introduced in Section 4, we start from the line-level manually labeled dataset. We clone all git repositories of the studied projects and use PyDriller toolkit [49] to identify the defect-introducing commits since our dataset has manually validated the bug-fixing commit and their specific bug-fixing lines of code. Different from previous work [42], our work has exact labels of lines of code in bug-fixing commit (i.e., manually labeling) and we can finely identify the bug-inducing lines of code using *git blame*, which extremely decreases the noise. Therefore, according to prior work [7, 44], those deleted lines in defect-fixing commits are labeled as defective ones, otherwise, they are labeled as clean ones. Those deleted lines are further used to identify the bug-inducing commits.

As for localizing the defective lines, JITLine first uses LIME technology to generate synthetic instances and builds a local sparse

linear regression model to identify the importance of each feature (i.e., token in bag-of-words features) to the model decision. Once the importance score of each token is computed, JITLine generates the ranking of defect-prone lines by summarizing the importance score for all tokens that appear in that line. As for Yan's approach, they directly build a source code language model and train it on clean source code lines (i.e., a clean model) by using the N-gram model. Different from the two approaches, JIT-Fine directly utilizes the weights of each token in our fine-tuned CodeBERT model to calculate its impact on classification. That is, JITLine/N-gram locates defective lines of code after the defect prediction model building, while JIT-Fine locates defective lines of codes during the process of the defect prediction model building.

Result: The evaluation results are presented in Table 6 and the best performances are highlighted in bold. According to the results, we find that our approach JIT-Fine performs best on all performance measures. In particular, JIT-Fine achieves 0.212, 0.214, 0.208, 0.318 and 10.8 in terms of Top-5, Top-10, Recall@20%Effort$_{line}$, Effort@20%Recall$_{line}$ and IFA$_{line}$, respectively, which improves JITLine and N-gram by 105% and 10%, by 117% and 9%, by 32% and 46%, by 4% and 8%, by 55% and 29%, in terms of of Top-5, Top-10, Recall@20%Effort$_{line}$, Effort@20%Recall$_{line}$ and IFA$_{line}$, respectively. The results also indicate the advantages of a unified model for defect prediction and defect localization.

Table 6: Defect localization results of JIT-Fine compared against JITLine and Yan et al.'s work.

Methods	Accuracy↑		R@20%E$_l$↑	E@20%R$_l$↓	IFA$_l$↓
	Top-5	Top-10			
JITLine	0.104	0.098	0.157	0.332	24.2
Yan et al.	0.193	0.195	0.143	0.345	15.3
JIT-Fine	**0.212**	**0.214**	**0.208**	**0.318**	**10.8**

∗**R@20%E$_l$**: Recall@20%Effort$_{line}$, **E@20%R$_l$**: Effort@20%Recall$_{line}$, **IFA$_l$**: IFA$_{line}$.

✍ **RQ-3** ► *JIT-Fine achieves the better performance on localizing defects compared with state-of-the-art approaches, which indicates that building a unified model with the integrated features can benefit both defect prediction task and defect localization task.*◄

7 THREATS TO VALIDITY

Threats to Internal Validity mainly correspond to the potential mistakes in our implementation of our approach and other baselines. To minimize such threat, we not only implement these approaches by pair programming but also directly use the original source code from the GitHub repositories shared by corresponding authors. Besides, we use the same hyperparameters in the original papers. The authors also carefully review the experimental scripts to ensure their correctness.

Threats to External Validity mainly correspond to the studied dataset. Even we have tested our model on the so far largest fine-gained evaluation in the literature to ensure a fair comparison with baselines, the project diversity is also limited from three aspects. The first one is the programming language used in studied projects, which is developed Java programming language. However, projects developed by other popular programming languages (e.g., C/C++ and Python) have not been considered. The second one is the studied projects are open-source projects, the performance of JIT-Fine on commercial projects is unknown. Thus, more diverse commit-level datasets can be explored in future work. The last one is that in reality not all bugs can be identified in the code repository. Though Herbold et al. [14] manually labeled their dataset, they still may miss a few bug-fix commits.

Threats to Construct Validity mainly correspond to the performance metrics in our evaluations. To minimize such threat, we consider a few types of metrics for different comparison tasks. In particular, we generally consider two kinds of performance metrics (i.e., effort-agnostic and effort-aware) with tens of performance metrics (i.e., F1-score, AUC, P$_{opt}$).

8 RELATED WORK

JIT defect prediction has attracted extensive attention of researchers in recent years since it can identify defect-inducing commit at a fine-grained level at check-in time. Mockus and Weiss [33] firstly extracted historical information (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) in commits to build a classifier to predict the risk of new commits. Kamei et al. [22] then proposed 14 change-level features and used them to build an effort-aware JIT prediction model. The 14 change-level features are widely used in the following studies. Yang et al. [58, 59] subsequently proposed two approaches. In particular, Yang et al. [59] firstly used Deep Belief Network (DBN) to extract higher-level information from the initial change-level features. Then, Yang et al. [58] combined decision tree and ensemble learning to build an ensemble learning model for JIT defect prediction. To further improve Yang et al's model, Young et al. [61] proposed a new deep ensemble method by using arbitrary classifiers in the ensemble and optimizing the weights of the classifiers. Later, Liu et al. [28] proposed a new unsupervised approach named code churn and evaluated it in effort-aware settings. Following that, Chen et al. [6] treated the effort-aware JIT defect prediction task as a multi-objective optimization problem and consequently a set of effective features are selected to build the prediction model. McIntosh et al. [29] investigated the impact of systems evolution on JIT defect prediction models via a longitudinal case study of 37,524 changes from the rapidly evolving QT and OpenStack systems. They found that the interval between training periods and testing periods has side effects on the performance of JIT models and JIT models should be trained using six months (or more) of historical data. Besides, Wan et al. [52] discussed the drawbacks of existing defect prediction tools and highlighted future research directions through literature review and a survey of practitioners. Moreover, Cabral et al. [4] utilized a new sampling technology to address the issues of verification latency and class imbalance evolution in online JIT defect prediction setting. Recently, Hoang et al. [17, 18] proposed two new approaches, which use a modern deep learning model to learn the representation of commit message and code changes.

Apart from the above approaches, researchers also conduct studies on fine-grained prediction and focus more attention on where the defect exists. Pascarella et al. [40] proposed a fine-grained JIT

defect prediction model based on handcrafted features to prioritize which changed files in a commit are the riskiest ones. Yan et al. [57] proposed a two-phase approach for defect identification and defect localization. In particular, they firstly trained a prediction model on software metrics to identify which commits are the most risky ones, then they trained the N-gram model on textual features, which is subsequently used to locate the riskiest lines. In addition, Wattanakriengkrai et al. [53] stated that a machine learning approach can achieve better performance than the N-gram approach. However, these work mainly focused on file-level defect localization. Recently, Chanathip et al. [42] proposed a new approach JITLine, which is a machine learning-based JIT defect approach for predicting defect-introducing commits and for localizing defective lines that are related to defective commit.

Prior studies either focus on defect prediction or focus on defect localization using only expert features or only semantic features [9, 17, 18, 22]. A few studies try to address the two tasks simultaneously with two sub-approaches or with two sub-phases [42, 57]. Different from prior work, in this paper, we focus on building a unified model using both expert features and semantic features for addressing both defect prediction and defect localization.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose a unified approach **JIT-Fine**, which fully utilizes both expert features and contextual semantic features of modified source code to build a performance-better model for just-in-time defect prediction and just-in-time defect localization simultaneously. We also build a large-scale line-level labeled dataset **JIT-Defects4J** for both JIT-DP and JIT-DL research. To investigate the effectiveness of JIT-Fine, we make a comprehensive comparison with six baselines on ten performance measures. The results empirically demonstrate the value of integrating the expert features and semantic features for two kinds of just-in-time software quality assurance.

Our future work involves extending our evaluation by considering more open source and commercial projects developed in other programming languages (e.g., C/C++, Python, etc.). We also plan to implement JIT-Fine into a tool (e.g., a GitHub plugin) to assess its usefulness in practice.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China (Grant No.62202419 and No. 62172214), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), the Key Research and Development Program of Zhejiang Province (No.2021C01105), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06).

REFERENCES

[1] Maha Al-Yahya, Hend Al-Khalifa, Alia Bahanshal, and Iman Al-Oudah. 2011. Automatic generation of semantic features and lexical relations using OWL ontologies. In *Proceedings of the International Conference on Application of Natural Language to Information Systems*. Springer, 15–26.

[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] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Technique Report in University of Cambridge* (2013).

[4] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676.

[5] 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.

[6] 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.

[7] 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.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] 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* (2019).

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. [n. d.]. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[11] 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.

[12] 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.

[13] 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.

[14] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. 2021. A Fine-grained Data Set and Analysis of Tangling in Bug Fixing Commits. *Empirical Software Engineering* (2021).

[15] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.

[16] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 121–130.

[17] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[18] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[19] Shenda Hong, Yuxi Zhou, Meng Wu, Junyuan Shang, Qingyun Wang, Hongyan Li, and Junqing Xie. 2019. Combining deep neural networks and engineered features for cardiac arrhythmia detection from ECG recordings. *Physiological measurement* 40, 5 (2019), 054009.

[20] 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)*. IEEE, 159–170.

[21] 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.

[22] 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.

[23] Sunghun Kim and E. James Whitehead Jr. 2006. How long did it take to fix bugs?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*. ACM, 173–174. https://doi.org/10.1145/1137983.1138027

[24] 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.

[25] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proceedings of the 22nd International Conference on Program Comprehension*. 262–265.

[26] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting commits via past code changes. In *Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 129–136.

[27] Marco Tú lio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.

[28] 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.

[29] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

[30] Thilo Mende and Rainer Koschke. 2010. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 107–116.

[31] Tim Menzies, Andrew Butcher, Andrian Marcus, and David Zimmermann, Thomas a nd 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.

[32] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia Haiduc. 2018. Are bug reports enough for text retrieval-based bug localization?. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 381–392.

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

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

[35] 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.

[36] 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* 48, 3 (2020), 786–802.

[37] 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 Transactions on Software Engineering and Methodology* (2022).

[38] Chao Ni, Kaiwen Yang, Xin Xia, David Lo, Xiang Chen, and Xiaohu Yang. 2022. Defect Identification, Categorization, and Repair: Better Together. *arXiv preprint arXiv:2204.04856* (2022).

[39] Shengyi Pan, Lingfeng Bao, Xiaoxue Ren, Xin Xia, David Lo, and Shanping Li. 2021. Automating developer chat mining. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 854–866.

[40] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[41] Rahul Paul, Samuel H Hawkins, Yoganand Balagurunathan, Matthew Schabath, Robert J Gillies, Lawrence O Hall, and Dmitry B Goldgof. 2016. Deep feature transfer learning in combination with traditional features predicts survival among patients with lung adenocarcinoma. *Tomography* 2, 4 (2016), 388–395.

[42] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2021. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 369–379.

[43] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* (2020), 1–26.

[44] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology* 99 (2018), 164–176.

[45] Gema Rodriguezperez, Meiyappan Nagappan, and Gregorio Robles. 2020. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering* (2020).

[46] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 436–447.

[47] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 966–969.

[48] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[49] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911.

[50] Sadia Tabassum, Leandro L Minku, Danyi Feng, George G Cabral, and Liyan Song. 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 554–565.

[51] 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.

[52] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1241–1266.

[53] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1480–1496.

[54] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 326–337.

[55] Fei Wu, Xiao-Yuan Jing, Ying Sun, Jing Sun, Lin Huang, Fangyi Cui, and Yanfei Sun. 2018. Cross-Project and Within-Project Semisupervised Software Defect Prediction: A Unified Approach. *IEEE Transactions on Reliability* (2018).

[56] Yan Xiao and Jacky Keung. 2018. Improving bug localization with character-level convolutional neural network and recurrent neural network. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 703–704.

[57] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2022. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* 48, 2 (2022), 82–101.

[58] 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.

[59] 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.

[60] 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.

[61] Steven Young, Tamer Abdou, and Ayse Bener. 2018. A replication study: just-in-time defect prediction with ensemble learning. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 42–47.

[62] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 427–438.

[63] Gao Zhipeng, Xia Xin, Lo David, Grundy John, and Zimmermann Thomas. 2021. Automating the Removal of Obsolete TODO Comments. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[64] Michael Zhivich and Robert K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy* 7, 2 (2009), 87–90.

[65] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.

[66] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction. *ACM Trans. Softw. Eng. Methodol.* 27, 1 (2018), 1:1–1:51.