The Journal of Systems & Software 206 (2023) 111853

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Boosting multi-objective just-in-time software defect prediction by fusing expert metrics and semantic metrics☆

Xiang Chen [a,*], Hongling Xia [a], Wenlong Pei [a], Chao Ni [b], Ke Liu [a]

[a] School of Information Science and Technology, Nantong University, Nantong, China
[b] School of Software Technology, Zhejiang University, Hangzhou, China

ARTICLE INFO

Article history:
Received 25 February 2023
Received in revised form 5 July 2023
Accepted 11 September 2023
Available online 16 September 2023

Keywords:
Just-in-time defect prediction
Multi-objective optimization
Expert metrics
Semantic metrics
Metric fusion

ABSTRACT

Just-in-time software defect prediction (JIT-SDP) aims to predict whether a code commit is defect-inducing or defect-clean immediately after developers submit their code commits. In our previous study, we modeled JIT-SDP as a multi-objective optimization problem by designing two potential conflict optimization objectives. By only considering expert metrics for code commits, our proposed multi-objective just-in-time software defect prediction (MOJ-SDP) approach can significantly outperform state-of-the-art supervised and unsupervised baselines. Recent studies have shown that deep learning techniques can be used to automatically extract semantic metrics from code commits and achieved promising performance for JIT-SDP. However, it is unclear how well MOJ-SDP performs when semantic metrics are used, and whether these two types of metrics are complementary and can be boosted by fusing them for MOJ-SDP. We conducted an extensive experiment using 27,319 code commits from 21 real-world open-source projects. Our results show that when using semantic features, the performance of MOJ-SDP can be slightly decreased for P_{opt}, but greatly improved for Recall@20%Effort. However, when these two types of metrics are fused based on the model-level fusion with the maximum rule, the performance can be boosted by a large margin and outperform state-of-the-art JIT-SDP baselines.

1. Introduction

During software development and maintenance, if developers can identify the defective modules in advance, they can allocate software quality assurance resources more reasonably and help to improve the software quality. In previous software defect prediction studies (Hall et al., 2011; Wan et al., 2018), most of the researchers perform software defect prediction at the coarse-grained level (such as file or module). While just-in-time software defect prediction (JIT-SDP) aims to automatically predict whether a code commit is defect-inducing or not. It is not hard to find that JIT-SDP can perform software defect prediction at the more fine-grained level (i.e., code change). In addition to the above advantage, by using the trained JIT-SDP models, the developers can get immediate feedback after submitting their code commits to the version control system. Moreover, JIT-SDP can help to find appropriate developers to localize and fix defects when the submitted code commit is predicted as a defect-inducing one.

In previous studies (Hall et al., 2011; Zhao et al., 2022), researchers have extensively studied the JIT-SDP problem. They proposed different approaches, which mainly design effective metrics to measure the code commits and then utilize traditional machine learning methods or deep learning methods to build JIT-SDP models. In the early research phase, researchers mainly resorted to expert metrics (i.e., hand-craft metrics) to construct JIT-SDP models. These expert metrics were manually designed by domain experts based on their understanding of defect-inducing commits. For example, Kamei et al. (2012) designed 14 expert metrics from five dimensions (such as code change size, code change diffusion, code change purpose, history of code changes, and developer experience). However, these expert metrics were designed based on expert knowledge and may not effectively represent the semantic meaning of the code commits. For example, two code commits with different labels (such as defect-inducing or defect-clean) may have the same metric values (such as code change size) (Hoang et al., 2019). When only considering expert metrics for JIT-SDP, Yang et al. (2016) found that there exist some simple unsupervised methods, which are comparative to previous supervised methods in terms of effort-aware performance measures. However, we still believed that supervised methods could achieve better performance. Therefore, we modeled JIT-SDP

☆ Editor: Shane McIntosh.

* Corresponding author.
E-mail addresses: xchencs@ntu.edu.cn (X. Chen), jstzxhl@yeah.net (H. Xia), wl.pei@outlook.com (W. Pei), chaoni@zju.edu.cn (C. Ni), aurora.ke.liu@outlook.com (K. Liu).

https://doi.org/10.1016/j.jss.2023.111853
0164-1212/

as a multi-objective optimization problem (Chen et al., 2018) in our previous study. In our problem formulization, we consider two potential conflict optimization objectives when constructing JIT-SDP models. The first objective is designed to maximize the number of identified defect-inducing code commits. The second objective is designed to minimize the inspection costs for code commits. Then we proposed a novel multi-objective just-in-time software defect prediction (MOJ-SDP) approach MULTI. Based on experimental subjects only considering expert metrics (Kamei et al., 2012; Yang et al., 2016), we found MULTI can significantly outperform 43 supervised methods and unsupervised methods in three different evaluation scenarios (i.e., cross-validation scenario, cross-project validation scenario, and timewise validation scenario) in terms of effort-aware performance measures.

Recently, researchers resorted to deep learning to learn semantic metrics from the code commits by directly analyzing the code changes and commit messages. For example, Hoang et al. were the first to propose an end-to-end deep learning framework DeepJIT (Hoang et al., 2019) that can automatically extract semantic metrics by using CNN (Convolutional Neural Network). Later, they proposed CC2Vec (Hoang et al., 2020), which can model the hierarchical structure of code changes with the attention mechanism. **However, the effectiveness of MOJ-SDP when considering semantic metrics is unknown.** Therefore, we want to investigate whether MOJ-SDP with semantic metrics can outperform MOJ-SDP with expert metrics in this study.

Previous empirical results (Hoang et al., 2019, 2020) showed that using semantic metrics can achieve better performance than expert metrics. However, the analysis perspectives of these two different metric types are not quite the same. For example, only analyzing the code change in a code commit cannot acquire information for the number of developers, who have modified the files of this code commit in the past (Matsumoto et al., 2010). Therefore, expert metrics and semantic metrics may be complementary to each other. An intuitive approach is to consider both types of these metrics to boost the performance of JIT-SDP. For example, Ni et al. (2022a) fully utilized these two types of metrics and proposed a unified model JIT-Fine for just-in-time defect prediction and localization. **However, the effectiveness of MOJ-SDP when fusing both types of metrics is unknown.** Therefore, we want to first investigate whether these two types of metrics are complementary by analyzing correctly identified defect-inducing code commits for MOJ-SDP. If they are complementary to a certain degree, we then want to investigate whether MOJ-SDP by fusing both types of metrics can achieve the best performance. Finally, by considering this metric fusion setting, we want to investigate whether MOJ-SDP can outperform state-of-the-art JIT-SDP baselines, such as the recently proposed JIT-SDP approach JIT-Fine (Ni et al., 2020).

Since experimental subjects shared by previous JIT-SDP studies (Kamei et al., 2012; Hoang et al., 2019, 2020) often suffered from a low-quality problem due to tangled commits, we use a large-scale JIT-SDP dataset JIT-Defect4J (Ni et al., 2022a) with higher quality as experimental subjects in our empirical design. This dataset contains 27,319 code commits, which were gathered from 21 large-scale open-source projects. To evaluate the performance of the trained JIT-SDP models, we consider three effort-aware performance measures (i.e., Recall@20%Effort, Effort@20%Recall, P_{opt}), which consider the code commit inspection cost (i.e., the number of lines related to code changes).

In our empirical study, we want to answer the following three research questions (RQs).

RQ1: For MOJ-SDP, whether using semantic metrics can achieve better performance than expert metrics?

Results. For MOJ-SDP, our experimental results show that when compared with using expert metrics, using semantic metrics can slightly decrease the performance in terms of P_{opt}, but greatly improve the performance in terms of Recall@20%Effort.

RQ2: For MOJ-SDP, whether fusing both two types of metrics can further boost the performance?

Results. For MOJ-SDP, our experimental results show that two types of metrics are complementary to a certain degree by analyzing the correctly identified defect-inducing code commits. Moreover, fusing two different types of metrics by model-level fusion with the maximum rule can further boost the performance of MOJ-SDP.

RQ3: Whether MOJ-SDP by fusing both types of metrics can outperform state-of-the-art JIT-SDP baselines?

Results. The experimental results show that MOJ-SDP by fusing both types of metrics can outperform the state-of-the-art JIT-SDP baselines. For example, in terms of P_{opt}, MOJ-SDP with metric fusion can at most improve the performance by 19.8%.

Except for the previous study (Ni et al., 2022a), our study confirms again that two different types of metrics are complementary in the context of JIT-SDP when considering the MOJ-SDP approach. Therefore, we suggest that it is better for researchers to consider both types of metrics when designing new JIT-SDP approaches.

The main contributions of our study can be summarized as follows:

- We are the first to investigate the performance of MOJ-SDP by considering semantic metrics. Then we investigate the complementary between semantic metrics and semantic metrics for MOJ-SDP. Finally, we investigate the performance of MOJ-SDP by fusing these two types of metrics.
- We conduct experimental studies on a total of 27,319 code commits gathered from large-scale and diverse open-source projects. Final empirical results show that MOJ-SDP can achieve the best performance when fusing two types of metrics. By using this setting, MOJ-SDP can also outperform the state-of-the-art JIT-SDP baselines.

Open Science. To support the open science community, we shared our studied experimental subjects, scripts, and experimental results in our GitHub repository (https://github.com/wenlong-pei/MOJ_SDP).

Paper Organization. Section 2 introduces the code commit representation based on semantic metrics and expert metrics and the framework of the MOJ-SDP approach. Section 3 shows our experimental setup, including research questions, experimental subjects, performance measures, and the detailed experimental setup. Section 4 performs result analysis for each of our designed research questions. Section 5 conducts discussions on the influence of hyperparameter setting and the performance in terms of non-effort-aware performance measure. Section 7 summarizes the related studies for JIT-SDP and emphasizes the novelty of our study. Section 8 concludes this study and shows some potential future studies.

2. MOJ-SDP by fusing expert metrics and semantic metrics

In this section, we first introduce the preliminary knowledge about just-in-time software defect prediction. Then we show the framework of MOJ-SDP and introduce the details for each step.

2.1. Preliminary

Just-in-time software defect prediction aims to automatically predict whether a code commit induces defects or not. Compared to software defect prediction on the coarser granularity (such as the file or the module), JIT-SDP can perform defect prediction in the finer granularity. The process of JIT-SDP can be summarized as follows: (1) code commits (code changes and commit messages) can be extracted from the version control systems and are measured by metrics. (2) The types of extracted code commits are labeled as defect-inducing or defect-clean. (3) Based on the measured information and the labeled types of the gathered code commits, JIT-SDP models can be trained by machine learning methods (such as Logistic regression). (4) For a new code commit, the developer can measure this code commit and use the trained model to predict whether this code commit is defect-inducing or not.

In the early research stage, researchers mainly designed expert metrics based on their domain knowledge. The widely used classical expert metrics were designed by Kamei et al. (2012). They designed these metrics based on code change size, code change diffusion, code change purpose, history of code changes, and developer experience. In the recent research stage, researchers (Hoang et al., 2019, 2020) mainly learned semantic metrics directly from the code changes and commit messages through end-to-end deep learning.

Labeling the types of extracted code commits as defect-inducing or defect-clean mainly includes two steps. In particular, the first step is to identify a defect-fixing code commit. The second step is to locate the prior code commit that introduces the defect (i.e., defect-inducing code commit) in the first step. Śliwerski et al. (2005) were the first to propose an algorithm SZZ to implement this two-step approach. The proposed SZZ algorithm can scale up the labeling amount of code commits and has been widely used in previous JIT-SDP studies for labeling gathered code commits. However, it is still challenging to accurately identify defect-fixing code commits in the first Step and defect-inducing code commits in the second step. Therefore, this is still an open problem for JIT-SDP and many different SZZ variants have been proposed (Da Costa et al., 2016; Neto et al., 2018, 2019; Fan et al., 2019; Borg et al., 2019; Rosa et al., 2021; Herbold et al., 2022b).

2.2. Framework

In this subsection, we use Fig. 1 to show the framework of MOJ-SDP. Specifically, given code commits (i.e., code changes and corresponding commit messages), we first extract expert metrics and semantic metrics from these code commits as the code commit representation respectively (in Section 2.3). In the JIT-SDP model construction phase, for each metric type, we build the Pareto optimal solutions based on multi-objective optimization by utilizing the Logistic regression classifier based on the training set. Then, we generate multiple models *ME* for the expert metrics and multiple models *MS* for the semantic metrics based on the Pareto optimal solutions. Finally, we select the optimal model ME_o from *ME* and the optimal model MS_o from *MS* by evaluating their performance on the validation set according to a specific performance measure (in Section 2.4). In the JIT-SDP model inference phase, given a new code commit, we use the model-level fusion method to determine the final prediction value by returning the maximum output value of these two models (i.e., ME_o and MS_o) and use this prediction value to determine whether this new code commit is defect-inducing or defect-clean (in Section 2.5). In the rest of this section, we show the details for each step.

Table 1
Expert metrics used by MOJ-SDP.

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

2.3. Code commit representation

In MOJ-SDP, we learn code commit representation from two different perspectives: expert metrics and semantic metrics.

2.3.1. Expert metrics

In this study, we consider 14 expert metrics. These expert metrics can be classified into five dimensions, such as code change diffusion, code size, change purpose, modification history, and developer experience. We show the details of these expert metrics in Table 1. In this table, we show the metric name, description, and corresponding dimension. These 14 metrics are first proposed in the original study of JIT-SDP (Kamei et al., 2012). Due to the dataset sharing of this study, research on this JIT-SDP has attracted the attention of many researchers (Yang et al., 2015, 2017; Chen et al., 2018) and they all consider these 14 metrics as the expert metrics. Notice the JIT-SDP baselines (i.e., Yan et al. (2020), Deeper (Yang et al., 2015) and JIT-Fine (Ni et al., 2022a)) and our proposed approach consider all these 14 metrics as the expert metrics. While the baseline LApredict (Zeng et al., 2021) only utilizes the metric LA as the expert metric.

In our proposed MOJ-SDP approach, we gather these metric values by CommitGuru (Rosen et al., 2015). After gathering the metric values for these expert metrics, we can get the expert metric vector V_{EM}.

2.3.2. Semantic metrics

Recently, deep learning methods have been widely used in intelligent code understanding tasks (Yang et al., 2023; Wang et al., 2018; Lin et al., 2023; Li et al., 2017; Qiao et al., 2020; Chen et al., 2019; Yang et al., 2022b; Li et al., 2021; Yang et al., 2022a; Liu et al., 2022b; Yang et al., 2021b,a). For JIT-SDP, researchers also aimed to directly extract semantic representation from code changes and commit messages via deep learning (Hoang et al., 2019, 2020).

In MOJ-SDP, we extract semantic metrics of code commits by utilizing the pre-trained model CodeBERT (Feng et al., 2020), which is more effective than previous deep learning techniques (Hoang et al., 2019, 2020). CodeBERT is designed for software engineering tasks and we can fine-tune it on the JIT-SDP problem by following the solutions used by previous studies (Niu et al., 2022; Ahmad et al., 2021; Yu et al., 2022; Liu et al., 2022a). Specifically, we consider three types of information for code commits. The first type of information is the **commit message**, which represents the

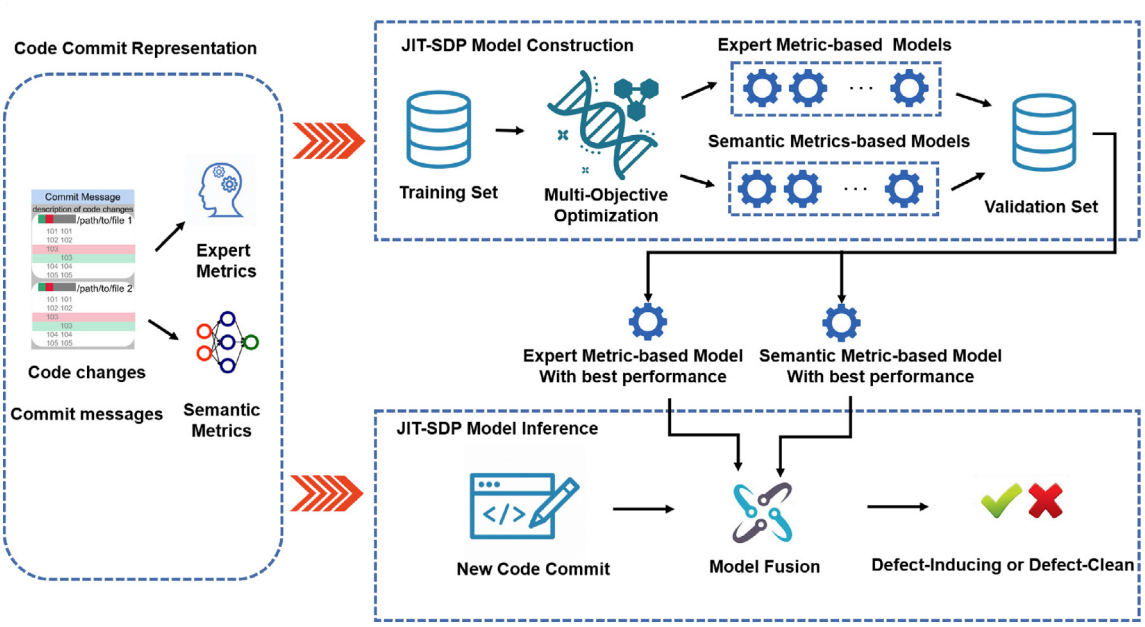

Fig. 1. The overview of MOJ-SDP, we use the model-level fusion with the maximum rule to fuse expert metrics and semantic metrics in this figure.

description of the submitted code commit. The last two types of information are **added lines** and **deleted lines**, which represent the lines added and deleted in the code commit. Then we add the token "[CLS]" before the commit message, the token "[ADD]" before the added lines, and the token "[DEL]" before the deleted lines. These three different tokens can differentiate these different information types. Later, these different types of information are tokenized into a token sequence. We input the token sequence into the CodeBERT model and then generate the corresponding embedding vector V_{SM} (i.e., semantic metric vector for the code commit). In our study, we use the following hyper-parameters' values to train the model: the learning rate is 1e-5, the batch size is 24, and the maximum epoch is 50. The dimension of the generated semantic vectors is set to 768.

2.4. JIT-SDP model construction

MOJ-SDP model the JIT-SDP problem as the bi-objective optimization problem. This model construction approach is motivated by the idea of search-based software engineering (SBSE). The concept of SBSE was first proposed by Harman et al. (2012). In particular, SBSE can use search-based evolutionary algorithms (such as simulated annealing, and genetic algorithm) to provide automated or semi-automated solutions for software engineering problems with large-scale complex search spaces, which have multiple competing or even conflicting optimization objectives. Until now, SBSE has been successfully applied to different tasks throughout the software life cycle (such as software requirement analysis, software design, software testing, and software maintenance) (Henard et al., 2015; Grunske, 2006; Zhang et al., 2007; Tawosi et al., 2022; Ni et al., 2019). For software defect prediction, Harman (2010) also suggested that SBSE (especially the multi-objective optimization algorithm) can help to build high-quality prediction models.

Solution Encoding. MOJ-SDP uses Logistic regression to construct JIT-SDP models. Logistic regression is a classical machine learning classifier and has been widely used in previous software defect prediction studies (Ghotra et al., 2015; Chen et al., 2018). Supposing there are n metrics to measure the code commit, we encode the coefficient vector $c = \langle c_0, \ldots, c_n \rangle$ of the Logistic regression model as a solution for JIT-SDP. Notice, the encoding of the coefficient vector depends on the model fusion method, which is discussed in Section 2.5. If we consider the metric-level fusion, the coefficient is related to the concatenation of the expert metric vector and the semantic metric vector. Otherwise, if we consider the model-level fusion, the coefficient is related to the expert metric vector or the semantic metric vector.

Here we use cc_i to denote the ith code commit and use $v_{i,j}$ to denote its value on the jth metric. For this code commit, we can use the function $y()$ shown in Eq. (1), which is the modeling formula used in Logistic regression, to estimate the defect-inducing probability of this code commit.

$$y(cc_i, c) = \frac{1}{1 + e^{-(c_0 + c_1 v_{i,1} + \cdots + c_n v_{i,n})}} \tag{1}$$

Since JIT-SDP is modeled as a binary classification problem, if the value of $y()$ is larger than 0.5, we classify this code commit as defect-inducing. Otherwise, we classify this code commit as defect-clean. Then we define the new function $Y()$ as follows:

$$Y(cc_i, c) = \begin{cases} 1 & if\ y(cc_i, c) > 0.5 \\ 0 & if\ y(cc_i, c) \leq 0.5 \end{cases} \tag{2}$$

Optimization Objective Setting. During the model construction process, MOJ-SDP considers two optimization objectives. Specifically, the first optimization objective is designed in terms of the benefit perspective. Based on a set of code commits CC and a solution c, it can be computed as follows:

$$benefit(c) = \sum_{cc_i \in CC} Y(cc_i, c) \times defect(cc_i) \tag{3}$$

In Eq. (3), $defect(cc_i)$ denotes whether this code commit cc_i is really defect-inducing. If this code commit is defect-inducing, the value of $defect(cc_i)$ is set to 1. Otherwise, its value is set to 0.

The second optimization objective is designed in terms of the cost perspective. Based on a set of code commits CC, it can be computed as follows:

$$cost(c) = \sum_{cc_i \in CC} Y(cc_i, c) \times SIZE(cc_i) \tag{4}$$

In Eq. (4), we use the function $SIZE()$ to measure the code inspection costs on the code commit m_i. the function $SIZE()$ returns the total number of LOC (lines of code) modified by a code commit.

Table 2
Hyperparameter setting in JIT-SDP model construction.

Hyperparameter name	Value
Population size	200
Iteration number	600
Crossover probability	0.5
Mutation probability	$1/n$

$*$ n denotes the number of the metrics.

There exists a potential conflict between these two optimization objectives. Specifically, if we want to identify more defect-inducing code commits, we may increase the costs of code inspection. On the contrary, if we want to reduce the costs of code inspection, we may miss more defect-inducing commits.

To facilitate the subsequent description of the MOJ-SDP model construction approach, we show related definitions for multi-objective optimization.

Definition 1 (*Pareto Dominance*)**.** Supposing w_i and w_j are two feasible solutions, we call w_i is Pareto dominance on w_j, if and only if: ($benefit(w_i) > benefit(w_j)$ and $cost(w_i) \leq cost(w_j)$) or ($benefit(w_i) \geq benefit(w_j)$ and $cost(w_i) < cost(w_j)$)

Definition 2 (*Pareto Optimal Solution*)**.** A feasible solution w is a Pareto optimal solution, if and only if there is no other feasible solution w^*, which is Pareto dominance on the solution w.

Definition 3 (*Pareto Optimal Set*)**.** This set contains all the Pareto optimal solutions.

Definition 4 (*Pareto Optimal Front*)**.** The surface composed of the vectors corresponding to all the Pareto optimal solutions is called the Pareto front.

Model Construction via NSGA-II. Due to potential conflicts in the optimization objectives designed from two different perspectives when evaluating the quality of the candidate solutions, MOJ-SDP utilizes a classical multi-objective optimization algorithm NSGA-II (Deb et al., 2002) to find the Pareto optimal set. This solution first initializes the population. This population contains N chromosomes, which are encoded based on the above solution encoding. In the initial population, each chromosome is randomly generated (i.e., the value of each element in the chromosome is assigned by a random value). Then this solution utilizes classical two evolutionary operators (the crossover operator, and the mutation operator) to generate new chromosomes for the next population. Specifically, the crossover operator randomly chooses two chromosomes according to a specified crossover probability, performs the crossover operation, and generates two new chromosomes. The mutation operator randomly chooses a chromosome according to a specified mutation probability, performs the mutation operation, and generates a new chromosome. Later it performs the selection operation to select high-quality chromosomes for the new population by the fast non-dominated sorting algorithm and the crowding distance (Deb et al., 2002). After satisfying the termination criterion, it may converge to stable solutions and return all the Pareto optimal solutions in the current population. In this solution, we utilize the specified number of population iterations as the termination criterion. Notice that all the Pareto optimal solutions are generated based on the training set and each solution (i.e., coefficient vector) can be used to construct a JIT-SDP model via Logistic regression. The hyperparameters of MOJ-SDP and their values are shown in Table 2.

Since we can construct multiple JIT-SDP models based on the solutions in the Pareto front, we use the validation set to further select the optimal JIT-SDP model M according to a specific performance measure.

2.5. JIT-SDP model inference

The straightforward idea is to concatenate the expert metric vector V_{EM} and the semantic metric vector V_{SM} as the final vector (i.e., the dimension of the input vector is $14 + 768$) before model construction. This information fusion method belongs to the metric-level fusion. In addition to this fusion method, there is another fusion method (i.e., model-level fusion). For this fusion method, we can use different rules (such as maximum, average, and weighted average) to determine how to generate the final prediction value by considering the output of these two models, which are constructed based on expert metrics (i.e., the dimension of the input vector is 14) or semantic metrics (i.e., the dimension of the input vector is 768). In our study, we find the model-level fusion with the maximum rule can achieve the best performance and a more detailed analysis can be found in Section 5.1. Specifically, when given a new code commit, we first generate the code commit representation for this code commit by considering expert metrics and semantic metrics respectively. Then we fuse the optimal model ME_o for expert metrics and the optimal model MS_o for semantic metrics by returning the maximum prediction value of these two models, which can finally predict whether this code commit is defect-inducing or not.

3. Experimental design

In this section, we first show our designed three research questions and their design motivation. Then we introduce the characteristics of our considered experimental subjects, performance measures, and our experimental setup details.

3.1. Research questions

In our empirical study, we design the following three research questions (RQs).

RQ1: For MOJ-SDP, whether using semantic metrics can achieve better performance than expert metrics?

Motivation. The effectiveness of MOJ-SDP has been confirmed when considering expert metrics for JIT-SDP. Specifically, in our previous study (Chen et al., 2018), we found that MOJ-SDP can achieve better performance than 43 unsupervised methods and supervised methods considered by Yang et al. (2016) in terms of effort-aware performance measures. Their model performance evaluation is conducted in three different evaluation scenarios, such as the cross-validation scenario, the cross-project validation scenario, and the timewise validation scenario. In this RQ, we want to investigate whether MOJ-SDP with semantic metrics can outperform MOJ-SDP with expert metrics.

RQ2: For MOJ-SDP, whether fusing both two types of metrics can further boost the performance?

Motivation. Since the analysis perspectives of these two types of metrics are not quite the same. Therefore, expert metrics and semantic metrics may be complementary to each other. An intuitive approach is to consider both types of these metrics to boost the performance of JIT-SDP. In a previous study, Ni et al. (2022a) fully utilized these two types of metrics and proposed a unified model JIT-Fine for just-in-time defect prediction and localization. Their empirical results confirmed the effectiveness of this metric fusion setting. In this RQ, we want to analyze whether these two types of metrics are complementary in our investigated

MOJ-SDP approach by analyzing the correctly identified defect-inducing code commits. If this complementary exists, we want to further investigate whether fusing these two different types of metrics can help to boost the performance of MOJ-SDP.

RQ3: Whether MOJ-SDP by fusing both types of metrics can outperform state-of-the-art JIT-SDP baselines?

Motivation. If fusing both types of metrics can help MOJ-SDP to achieve the best performance, we want to further compare MOJ-SDP with state-of-the-art JIT-SDP baselines (especially the recently proposed JIT-SDP approach JIT-Fine (Ni et al., 2022a)) in this RQ.

3.2. Experimental subjects

In previous studies on JIT-SDP, researchers have shared many experimental subjects and these shared experimental subjects have significantly promoted the research advancement of this research topic. For example, Kamei et al. (2012) gathered experimental subjects from six open-source projects (i.e., Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla, and PostgreSQL). These experimental subjects were then considered in the follow-up JIT-SDP studies (Yang et al., 2015, 2016, 2017; Chen et al., 2018). Recently, for JIT-SDP studies concerning semantic metrics, researchers (Hoang et al., 2019, 2020) considered two new large-scale software projects (i.e., QT and OpenStack).

However, previously shared experimental subjects may suffer from a low-quality problem with tangled commits, especially code commits with some kinds of code changes (such as bug fixing, code refactoring, software testing, new feature adding, and documentation) (Herzig and Zeller, 2013). These tangled commits can have a negative impact on the code commit labeling accuracy when using SZZ and its variants (Śliwerski et al., 2005; Fan et al., 2019). Therefore, we used a large-scale high-quality dataset JIT-defects4J (Ni et al., 2022a), which was recently gathered by Ni et al. to alleviate the effects of tangled commits on the basis of LLTC4J (Herbold et al., 2022a). The characteristics of our considered experimental subjects can be found in Table 3. In this table, we show the project name, the time frame, the number of defect-inducing commits (DIC), the number of defect-clean commits (DCC), and the ratio of the defect-inducing commits (Ratio). As shown in this table, JIT-Defects4J has a varying number of code commits from 544 to 4026, and its defect-inducing commit ratio is from 1.76% to 18.75%. By following the recent study (Ni et al., 2022a), we split the dataset by following the time-aware setting. Specifically, for each project, we split the dataset by sorting all the code commits by their timestamp in an ascending way. Then we get the training set, the validation set, and the testing set according to the proportion of 80%, 10%, and 10%. Finally, we combine the data of all projects into the final training set, the validation set, and the testing set.

3.3. Performance evaluation measures

Since the optimization objectives of MOJ-SDP are designed in the effort-aware context, we mainly consider three effort-aware performance measures. These performance measures evaluate the performance of JIT-SDP models by considering the code inspection cost (i.e., the number of lines) for the code commits. In the rest of this subsection, we simply introduce these performance measures and more details can be found in the previous study (Ni et al., 2022a).

Recall@20%Effort (R@20%E). This performance evaluation measure computes a proportion between the actual number of defect-inducing code commits found with the 20% of the code inspection costs and the total number of defect-inducing code commits. The higher value of R@20%E means more actual defect-inducing code commits can be identified when spending 20% of the code inspection costs.

P_{opt}. This performance evaluation measure is on the basis of the concept of the Alberg diagram, which indicates the relationship between the Recall obtained by a JIT-SDP model and the code inspection cost for the specific JIT-SDP model.

Effort@20%Recall (E@20%R). This performance evaluation measure computes the amount of code inspection cost that developers have to spend when 20% actual defect-inducing code commits are identified.

The value range of these three performance evaluation measures is between 0 and 1. The larger the value of the first two measures, the better the performance of the corresponding approach. The last measure is the opposite, which means the smaller the value of this measure, the better the performance of the corresponding approach.

4. Result analysis

4.1. Result analysis for RQ1

RQ1: For MOJ-SDP, whether using semantic metrics can achieve better performance than expert metrics?

Approach. In this RQ, we first use MOJ-SDP to train the JIT-SDP model by only considering the expert metrics. Then we use MOJ-SDP to train the JIT-SDP model by only considering the semantic metrics. To evaluate the performance of these JIT-SDP models, we consider three effort-aware performance measures.

Results. The comparison results are shown in Table 4. In this table, we can find MOJ-SDP with expert metrics (MOJ-SDP$_{EM}$) can achieve the performance of 0.959, 0.027, 0.838 in terms of R@20%E, E@20%R, and P_{opt}. While MOJ-SDP with semantic metrics (MOJ-SDP$_{SM}$) can achieve the performance of 0.957, 0.015, 0.802 in terms of R@20%E, E@20%R, and P_{opt}. In terms of R@20%E, using two different metrics can achieve similar performance. In terms of E@20%R, using semantic metrics can achieve better performance. This means using semantic metrics can reduce 44.4% code inspection costs after identifying 20% defect-inducing code commits. However, in terms of P_{opt}, the performance of using semantic metrics can be slightly decreased.

Answer to RQ1: Compared to MOJ-SDP with expert metrics, MOJ-SDP with semantic metrics can reduce 44.4% code inspection costs if only identifying 20% defect-inducing code commits. However, in terms of P_{opt}, the performance of MOJ-SDP with semantic metrics can be slightly decreased.

4.2. Result analysis for RQ2

RQ2: For MOJ-SDP, whether fusing both two types of metrics can further boost the performance?

Approach. In this RQ, we first want to investigate whether these two types of metrics are complementary in correctly identifying defect-inducing code commits for MOJ-SDP. Specifically, we use the Venn diagram to show the prediction diversity on the identified defect-inducing code commits by following the previous study (Bowes et al., 2018). If these two types of metrics are complementary to a certain degree for MOJ-SDP, we want to investigate whether fusing these two types of metrics can boost the performance of MOJ-SDP. There exist different information fusion methods. In our study, we simply use the model-level fusion with the maximum rule to perform the metric fusion.

Results. Fig. 2 shows the prediction diversity in terms of the performance measure P_{opt}. In this figure, we find only 2906 defect-inducing code commits can be correctly identified by both

Table 3
Statistical information of our used experimental subjects.

Name	Time frame	DIC	DCC	DCC ratio
ant-ivy	2005-06-16∼2018-02-13	332	1,439	18.75%
commons-bcel	2001-10-29∼2019-03-12	60	765	7.27%
commons-beanutils	2001-03-27∼2018-11-15	37	574	6.06%
commons-codec	2003-04-25∼2018-11-15	36	725	4.73%
commons-collections	2001-04-14∼2018-11-15	50	1,773	2.74%
commons-compress	2003-11-23∼2018-11-15	178	1,452	10.92%
commons-configuration	2003-12-23∼2018-11-15	155	1,683	8.43%
commons-dbcp	2001-04-14∼2019-03-12	58	979	5.59%
commons-digester	2001-05-03∼2018-11-16	19	1,067	6.39%
commons-io	2002-01-25∼2018-11-16	73	1,069	6.39%
commons-jcs	2002-04-07∼2018-11-16	88	743	10.59%
commons-lang	2002-07-19∼2018-10-10	146	2,823	4.92%
commons-math	2003-05-12∼2018-02-15	335	3,691	8.32%
commons-net	2002-04-03∼2018-11-14	117	1,004	10.44%
commons-scxml	2005-08-17∼2018-11-16	47	497	8.64%
commons-validator	2002-01-06∼2018-11-19	36	562	6.02%
commons-vfs	2002-07-16∼2018-11-19	114	996	10.27%
giraph	2010-10-29∼2018-11-21	163	681	19.31%
gora	2010-10-08∼2019-04-10	39	514	7.05%
opennlp	2008-09-28∼2018-06-18	91	995	8.38%
parquer-mr	2012-08-31∼2018-07-12	158	962	14.11%
ALL		2332	24,987	8.54%

Table 4
Comparison results between MOJ-SDP with expert metrics and MOJ-SDP with semantic metrics.

Method	R@20%E (↗)	E@20%R (↘)	P_{opt} (↗)
MOJ-SDP$_{EM}$	0.959	0.027	0.838
MOJ-SDP$_{SM}$	0.957	0.015	0.802

"EM" refers to "Expert Metrics" and "SM" refers to "Semantic Metrics".

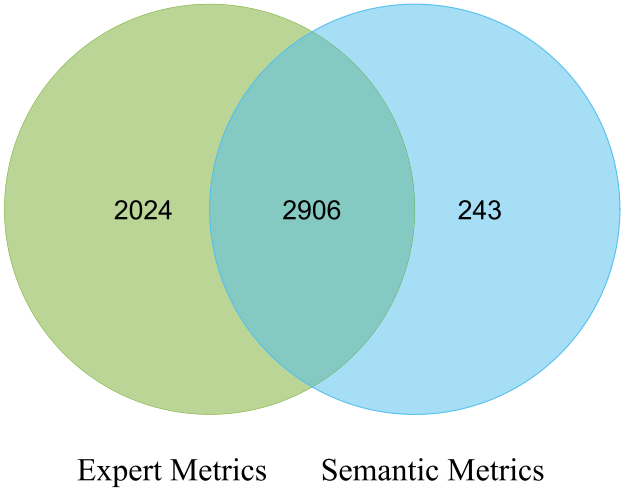

Fig. 2. The prediction diversity on correctly identifying defect-inducing code commits for MOJ-SDP with expert metrics and MOJ-SDP with semantic metrics by the Venn diagram in terms of P_{opt}.

methods, 2024 defect-inducing code commits can be only correctly identified by the MOJ-SDP with expert metrics, and 243 defect-inducing code commits can be only correctly identified by the MOJ-SDP with semantic metrics. This result indicates that learning the vector representation of code commits from different perspectives has certain limitations, as expert metrics or semantic metrics can only capture partial information about the code commit. This result is our motivation to further improve the performance of MOJ-SDP by fusing these two types of metrics.

Then we show the performance of MOJ-SDP by fusing two types of metrics and compare this setting with MOJ-SDP by only considering one type of metric. The comparison results can be found in Table 5. In this table, we can find MOJ-SDP by fusing two types of metrics (MOJ-SDP$_{MF}$) can achieve the performance of 0.970, 0.010, and 0.975 in terms of R@20%E, E@20%R, and P_{opt}. For example, in terms of P_{opt}, fusing two types of metrics can at least improve the performance by 21.6%. In terms of E@20%R, fusing two types of metrics can at least reduce the code inspection cost by 33.3%.

Table 5
Comparison results between only considering one type of metrics and considering both types of metrics.

Method	R@20%E (↗)	E@20%R (↘)	P_{opt} (↗)
MOJ-SDP$_{EM}$	0.959	0.027	0.838
MOJ-SDP$_{SM}$	0.957	0.015	0.802
MOJ-SDP$_{MF}$	0.970	0.010	0.975

"EM" refers to "Expert Metrics", "SM" refers to "Semantic Metrics", and "MF" refers to "Metric Fusion".

Answer to RQ2: For MOJ-SDP, expert metrics and semantic metrics are complementary to each other. Therefore, by fusing both types of metrics, MOJ-SDP can further boost performance.

4.3. Result analysis for RQ3

RQ3: Whether MOJ-SDP by fusing both types of metrics can outperform state-of-the-art JIT-SDP baselines?

Approach. In this RQ, we want to compare MOJ-SDP by metric fusion with state-of-the-art JIT-SDP baselines. Specifically, we select the following six state-of-the-art JIT-SDP baselines.

- **LApredict.** LApredict (Zeng et al., 2021) builds a JIT-SDP model by utilizing only the information of an expert metric LA (i.e., "lines of code added") with the traditional Logistic regression classifier.
- **Yan et al.** Yan et al. (2020) explore the expert metrics to build a JIT-SDP model with the Logistic regression and build a defect localization method with the n-gram technique.
- **Deeper.** Deeper (Yang et al., 2015) uses the deep neural model (i.e., deep belief networks) to generate and integrate advanced metrics from the initial expert metrics. In Deeper, the advanced metrics are linear combinations of the initial expert metrics.

- **DeepJIT.** DeepJIT (Hoang et al., 2019) automatically extracts semantic metrics by using CNN (Convolutional Neural Network).
- **CC2Vec.** CC2Vec (Hoang et al., 2020) models the hierarchical structure of code changes with the attention mechanism.
- **JITLine.** JITLine (Pornprasit and Tantithamthavorn, 2021) extracts source code tokens of code changes as metrics and applies a SMOTE technique that is optimized by a differential evolution algorithm to solve the class imbalance issue.
- **JIT-Fine.** JIT-Fine (Ni et al., 2022a) fully utilizes these two types of metrics and proposes a unified model JIT-Fine for just-in-time defect prediction and localization.

Results. The comparison results can be found in Table 6. We first revisit the performance differences between previous baselines based on our large-scale high-quality experimental subject, which can alleviate the effects of tangled commits. Specifically, similar to previous findings (Hoang et al., 2020), two classical semantic metrics-based baselines (i.e., DeepJIT Hoang et al. (2019) and CC2Vec Hoang et al. (2020)) can still outperform expert metrics-based baselines (i.e., Yan et al. (2020) and Deeper Yang et al. (2015)). Different from two classical semantic metrics-based baselines, The underlying intuition of JITLine (Pornprasit and Tantithamthavorn, 2021) is that code tokens that frequently appeared in defect-inducing code commits in the past may have a high probability to be fixed in the future. Therefore, JITLine simply extracts source code tokens of code changes as semantic features and we can still find JITLine can outperform DeepJIT (Hoang et al., 2019) and CC2Vec (Hoang et al., 2020) in our experimental subject. However, the simple but effective baseline LApredict (Zeng et al., 2021) shows negative results in our study (i.e., only outperforms the expert metrics-based baseline Yan et al. (2020) but worse than semantic metrics-based baselines), which means data quality is an important internal threat validity for the effectiveness of LApredict.

Then we compare MOJ-SDP with state-of-the-art JIT-SDP baselines. In terms of R@20%E, MOJ-SDP with metric fusion can at least improve the performance by 25.5% and at most improve the performance by 57.7%. in terms of P_{opt}, MOJ-SDP with metric fusion can at least improve the performance by 5.0% and at most improve the performance by 19.8%. in terms of E@20%R, MOJ-SDP with metric fusion can at most reduce the code inspection cost by 54.5%. Notice R@20%E indicates that the larger the better while E@20%R indicates that the smaller the better. We conduct a depth analysis of these comparison results, when compared to expert metrics-based baselines, we find MOJ-SDP$_{EM}$ (i.e., only consider expert metrics) can achieve better performance, especially in terms of R@20%E. This means with the 20% of the code inspection costs, MOJ-SDP$_{EM}$ can identify 95.9% defect-inducing commits. This finding is similar to our previous finding (Chen et al., 2018) on the earliest experimental subjects shared by Kamei et al. (2012) in 2012. When compared to semantic metrics-based baselines, we find MOJ-SDP$_{SM}$ (i.e., only consider semantic metrics) can also achieve better performance in terms of R@20%E (i.e., 0.957). The high performance in terms of R@20%E is contributed to our modeling method. In previous studies (such as Yan et al. (2020)), they trained the JIT-SDP models by Logistic regression, and the coefficient vector is optimized by the cross entropy-based loss function. While in our MOJ-SDP, the optimal values of the coefficient vector are searched by using the multi-objective optimization algorithm NSGA-II (Deb et al., 2002). During the evolutionary process, we select the higher-quality coefficient vectors through two optimization objectives, which aim to identify more defect-inducing code commits but decrease the cost of code inspection (more details can be found in Section 2.4). Therefore, MOJ-SDP has obvious advantages in optimizing the value of this performance measure. Moreover, by using metric fusion, MOJ-SDP$_{MF}$ can still hold the high value of R@20%E, while the value of E@20%R can be slightly improved, and the value of $\mathbf{P_{opt}}$ can be substantially improved. The effectiveness of metric fusion is also confirmed in the previous study of the baseline JIT-Fine (Ni et al., 2022a). In their study, they simply considered the metric-level fusion method (i.e., simply concatenate the expert metric vector and the semantic metric vector) and achieve better performance than only using one type of metric. Compared to this baseline, we consider a more effective metric fusion method (i.e., model-level fusion) (analyzed in Section 5.1) and use the multi-objective optimization algorithm NSGA-II as the backbone modeling method (help to achieve higher R@20%E). Final comparison results between MOJ-SDP$_{MF}$ and JIT-Fine also confirm the effectiveness of our approach setting.

Table 6
Comparison results between MOJ-SDP with metric fusion and state-of-the-art JIT-SDP baselines.

Method	R@20%E (↗)	E@20%R (↘)	P_{opt} (↗)
LApredict	0.625	0.020	0.814
Yan et al.	0.615	0.022	0.819
Deeper	0.638	0.021	0.827
DeepJIT	0.676	0.014	0.860
CC2Vec	0.676	0.014	0.861
JITLine	0.705	0.015	0.883
JIT-Fine	0.773	0.010	0.927
MOJ-SDP$_{MF}$	0.970	0.010	0.975

"MF" refers to "Metric Fusion".

Table 7
The Performance Influence of Different Information Fusion Methods on MOJ-SDP.

Fusion method	R@20%E (↗)	E@20%R (↘)	P_{opt} (↗)
Concatenation	0.964	0.011	0.962
Average	0.971	0.008	0.963
Weight (0.3:0.7)	0.970	0.009	0.962
Weight (0.4:0.6)	0.970	0.008	0.963
Weight (0.6:0.4)	0.970	0.008	0.963
Weight (0.7:0.3)	0.970	0.008	0.968
Max	0.970	0.010	0.975

Answer to RQ3: When fusing both types of metrics, MOJ-SDP can achieve better performance than state-of-the-art JIT-SDP baselines.

5. Discussion

5.1. The influence of the information fusion methods

In this section, we investigated the influence of different information fusion methods. For the metric-level fusion method, we simply concatenate the expert metric vector and the semantic metric vector as the final vector before training the model. For the model-level fusion method, we consider different rules. Given the outputs of the model based on expert metrics (M_e) and the model based on semantic metrics M_s, the model-level fusion method can generate the final prediction value based on the maximum, the average, and the weighted average of these two outputs. For the weighted average rule, we consider four different weight proportions (i.e., 0.3:0.7, 04:0.6, 0.6:0.4, 0.7:0.3) for the model M_e and the model M_s. The final results can be found in Table 7. In this table, we can find the model-level fusion with the maximum rule can achieve the best performance in our study.

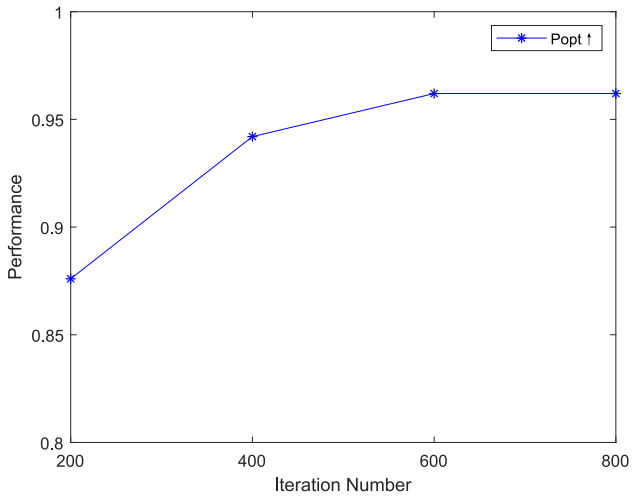

Fig. 3. Performance for MOJ-SDP with different iteration numbers.

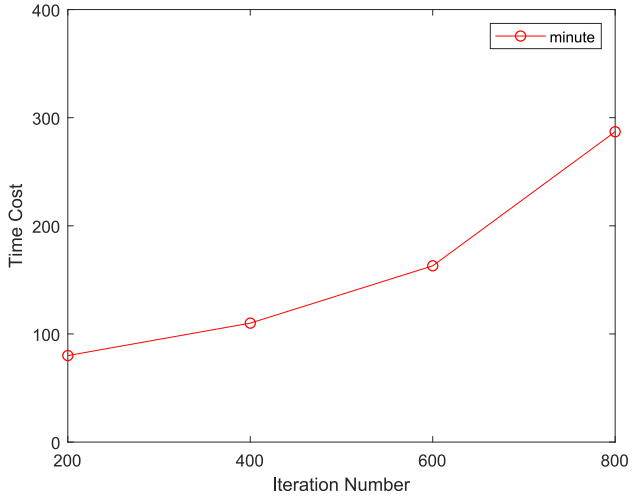

Fig. 4. JIT-SDP model construction time cost for MOJ-SDP with different iteration numbers.

5.2. The influence of the iteration number on MOJ-SDP

The iteration number is an important hyperparameter for the prediction model construction step in MOJ-SDP. Intuitively, increasing the iteration number can help to find better models, but it will also greatly increase the search computational costs. In this subsection, we want to investigate the influence of the iteration number on MOJ-SDP and find the cost-effectiveness value for this hyperparameter. Here, we consider four different iteration numbers (i.e., 200, 400, 600, 800) in terms of the performance measure P_{opt}. The results can be found in Fig. 3. In this figure, we can find when the iteration number reaches 600, the performance improvement for MOJ-SDP tends to be slow.

At the same time, as the number of iterations increases, the time cost is also growing fast. The results can be found in Fig. 4. For example, it only takes 80 min to train the JIT-SDP model when the number of iterations is set to 200. However, it takes 287 min when the number of iterations is set to 800. It is not hard to find when the iteration number exceeds 600, the time cost for JIT-SDP model construction increases rapidly.

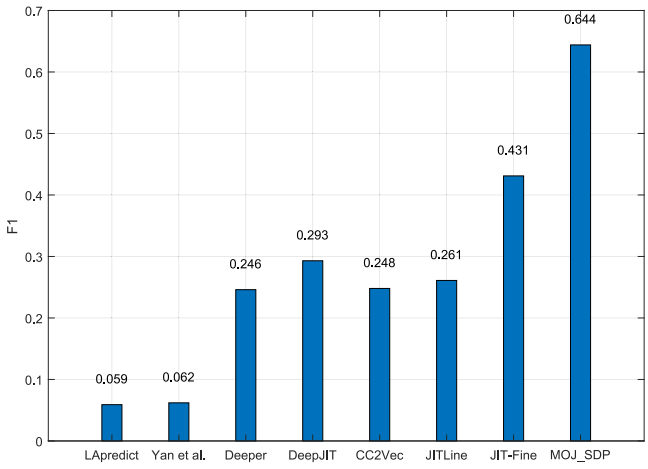

Fig. 5. Performance comparisons between MOJ-SDP with metric fusion and JIT-Fine in terms of F1.

5.3. Evaluation on non-effort-aware performance measure

Except for three effort-aware performance measures, we also consider a non-effort-aware performance measure F1. We use Precision to denote the percentage of actual defect-inducing code commits to all the predicted defect-inducing code commits, and use Recall to denote the percentage of predicted defect-inducing code commits to all the actual defect-inducing code commits. The F1 value is the harmonic average of the Precision value and the Recall value. Notice in this Section, we select the optimal model from the multiple models generated by NSGA-II for each type of metrics according to the F1 value on the validation set. In this subsection, we compare MOJ-SDP with metric fusion with the state-of-the-art baselines, which are introduced in Section 4.3. The final comparison results can be found in Fig. 5. Consistent with the results of the previous JIT-SDP studies (Yang et al., 2015; Pornprasit and Tantithamthavorn, 2021; Ni et al., 2022a), these baselines also achieved low performance in terms of F1. However, in this figure, we can find the F1 value of MOJ-SDP with metric fusion can achieve 0.644 (the Precision value is 0.61 and the Recall value is 0.68). Therefore, MOJ-SDP can at most improve the performance by 49.4% when compared with JIT-Fine and at least improve the performance by 1091% when compared with the baseline LApredict.

6. Threats to validity

In this section, we mainly analyze the potential threats to the validity of our empirical findings.

6.1. Internal threats

The main internal threat is the potential faults in the implementation of MOJ-SDP and baselines. To alleviate this threat, we use mature frameworks to guarantee implementation correctness and implement our proposed approach by pair programming. For baselines, we reuse existing implementations of these baselines and use the same values of the hyperparameters when their codes are available.

6.2. External threats

The first external threat is the selection of the experimental subjects. To alleviate this threat, we select JITDefect4J as our experimental subject, since previously shared experimental subjects

may suffer from a low-quality problem with tangled commits (Ni et al., 2022a). The second external threat is the selection of the Logistic regression classifier. The reason for this setting is that the candidate solution encoding (details can be found in Section 2) of MOJ-SDP is designed based on the Logistic regression classifier. However, our empirical results show that MOJ-SDP with metric fusion can achieve better performance than state-of-the-art JIT-SDP baselines when considering the Logistic regression classifier. In the future, we will continue to consider other classifiers and design corresponding solution encoding methods to further improve the performance of MOJ-SDP.

6.3. Construct threats

The main construct threat is related to the selected measures for JIT-SDP model performance evaluation. Since MOJ-SDP is designed in the effort-aware context, we mainly consider three performance measures to evaluate the performance of JIT-SDP models to alleviate this threat. These performance measures have also been widely used in previous studies (Ni et al., 2020, 2022a; Chen et al., 2018). Moreover, we also investigate the performance of JIT-SDP models in terms of the performance measure F1, which is a non-effort-aware performance measure in Section 5.

7. Related work

In this section, we simply classified previous studies for JIT-SDP into two categories (i.e., JIT-SDP based on expert metrics and JIT-SDP based on semantic metrics). In addition to these research studies, there are many studies concerning other issues for JIT-SDP. These issues include label noise (McIntosh and Kamei, 2018; Cabral et al., 2019; Fan et al., 2019), cross-project prediction (Kamei et al., 2016; Tabassum et al., 2020; Zhang et al., 2022), fine-grained prediction (Pornprasit and Tantithamthavorn, 2021; Pascarella et al., 2019; Trautsch et al., 2020), semi-supervised learning (Li et al., 2020), model interpretability (Pornprasit et al., 2021; Zheng et al., 2022), other application domains (such as JavaScript projects (Ni et al., 2022b), Android mobile apps (Zhao et al., 2021)).

7.1. Just-in-time defect prediction based on expert metrics

To our best knowledge, Mockus and Weiss (2000) were the first to study the problem of JIT-SDP. They predicted the risk of new code commits by utilizing historical information from previous code commits. Kim et al. (2008) extracted metrics from commit messages, source codes, and file names by using text mining methods. Kamei et al. (2012) designed 14 expert metrics and proposed an effort-aware prediction method based on Logistic regression. Yang et al. (2015) proposed the Deeper method, which utilized a deep belief network to extract high-level information from previously designed expert metrics. Later, they Yang et al. (2017) further proposed a two-layer ensemble learning method TLEL.

In 2016, Yang et al. (2016) found that there exist some simple unsupervised methods, which are comparative to previous supervised methods in terms of effort-aware performance measures. They evaluated the performance of these methods in three different evaluation scenarios (i.e., cross-validation scenario, cross-project validation scenario, and timewise validation scenario). This study has attracted researchers to pay continuous attention to the problem of JIT-SDP. For example, Fu and Menzies (2017) proposed a new supervised method OneWay, which is based on simple methods proposed by Yang et al. (2016) and can automatically select the potential best method. Liu et al. (2017) investigated a new unsupervised method based on code churn (i.e. the code change size). Huang et al. (2017, 2019) performed a holistic evaluation of previous JIT-SDP methods from two different perspectives: context switches and developer fatigue due to initial false alarms. However, we Chen et al. (2018) still believed that supervised methods should have better performance. Therefore, they applied multi-objective optimization to JIT-SDP and propose a novel MOJ-SDP method MULTI.

7.2. Just-in-time defect prediction based on semantic metrics

In recent years, driven by the progress of deep learning research, researchers use deep learning to extract semantic metric values from code commits. Hoang et al. (2019) proposed DeepJIT, which can use deep learning to extract semantic features from the commit logs and code changes. Later, Hoang et al. (2020) proposed CC2Vec, which can model the hierarchical structure of code changes with the attention mechanism. Recently, Zeng et al. (2021) evaluated DeepJIT and CC2Vec on an extended dataset and found that deep learning-based approaches cannot outperform a simple model LApredict.

7.3. Novelty of our study

In this study, we extended our previous study on MOJ-SDP (Chen et al., 2018) in three aspects. First, we investigate the effectiveness of MOJ-SDP by considering semantic metrics by deep learning. Second, we investigate the complementary of the expert metrics and semantic metrics for MOJ-SDP. Then we propose a simple metric fusion method, which can further improve the performance of MOJ-SDP and outperform state-of-the-art JIT-SDP baselines. Finally, we evaluate the effectiveness of MOJ-SDP on the higher-quality dataset JIT-Defects4J, which contains 27,319 code commits gathered from 21 real-world open-source software projects.

8. Conclusion and future work

In this study, we study multi-objective just-in-time software defect prediction by fusing the expert metrics designed in a manual way and the semantic metrics extracted by deep learning. By conducting an extensive experiment with 31,783 code commits gathered from 21 real-world open-source software projects, we find these two types of metrics are complementary in correctly identifying defect-inducing code commits for MOJ-SDP. Therefore, by simply fusing these two types of metrics, we can further boost the performance of MOJ-SDP and the boosted performance can also outperform state-of-the-art JIT-SDP baselines. Our results show that it is promising to combine the expert metrics and the semantic metrics to achieve better performance for MOJ-SDP.

In the future, we first want to investigate the effectiveness of MOJ-SDP by further considering more real-world open-source projects. We second want to consider other classifies for MOJ-SDP and design corresponding solution encoding methods. We third want to investigate more advanced multi-objective optimization methods and metric fusing methods to further improve the performance of MOJ-SDP. Finally, we want to learn higher-quality semantic metrics by considering more advanced deep code representation methods.

CRediT authorship contribution statement

Xiang Chen: Software, Conceptualization, Methodology, Writing – review & editing, Supervision. **Hongling Xia:** Data curation, Software, Validation. **Wenlong Pei:** Data curation, Software, Validation. **Chao Ni:** Data curation, Software, Writing – review & editing. **Ke Liu:** Software, Validation.

Declaration of competing interest

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

Data availability

Data will be made available on request.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (Grant no 62202419).

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K., 2021. Unified pre-training for program understanding and generation.. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.

Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation. pp. 7–12.

Bowes, D., Hall, T., Petrić, J., 2018. Software defect prediction: do different classifiers find the same defects? Softw. Qual. J. 26, 525–552.

Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S., 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 666–676.

Chen, D., Chen, X., Li, H., Xie, J., Mu, Y., 2019. Deepcpdp: Deep learning based cross-project defect prediction. IEEE Access 7, 184832–184848.

Chen, X., Zhao, Y., Wang, Q., Yuan, Z., 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. Inf. Softw. Technol. 93, 1–13.

Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6 (2), 182–197.

Fan, Y., Xia, X., da Costa, D., Lo, D., Hassan, A., Li, S., 2019. The impact of changes mislabeled by SZZ on just-intime defect prediction. IEEE Trans. Softw. Eng. 15 (1), 1–26.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 72–83.

Ghotra, B., McIntosh, S., Hassan, A.E., 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, IEEE, pp. 789–800.

Grunske, L., 2006. Identifying" good" architectural design alternatives with multiobjective optimization strategies. In: Proceedings of the 28th International Conference on Software Engineering. pp. 849–852.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Harman, M., 2010. The relationship between search based software engineering and predictive modeling. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 1–13.

Harman, M., Mansouri, S.A., Zhang, Y., 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45 (1), 1–61.

Henard, C., Papadakis, M., Harman, M., Le Traon, Y., 2015. Combining multiobjective search and constraint solving for configuring large software product lines. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, IEEE, pp. 517–528.

Herbold, S., Trautsch, A., Ledel, B., Aghamohammadi, A., Ghaleb, T.A., Chahal, K.K., Bossenmaier, T., Nagaria, B., Makedonski, P., Ahmadabadi, M.N., et al., 2022a. A fine-grained data set and analysis of tangling in bug fixing commits. Empir. Softw. Eng. 27 (6), 1–49.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2022b. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. Empir. Softw. Eng. 27 (2), 1–49.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, pp. 121–130.

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, pp. 34–45.

Hoang, T., Kang, H.J., Lo, D., Lawall, J., 2020. Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 518–529.

Huang, Q., Xia, X., Lo, D., 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 159–170.

Huang, Q., Xia, X., Lo, D., 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir. Softw. Eng. 24 (5), 2823–2862.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E., 2016. Studying just-in-time defect prediction using cross-project models. Empir. Softw. Eng. 21 (5), 2072–2106.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 318–328.

Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., Yu, D., 2021. SeCNN: A semantic CNN parser for code comment generation. J. Syst. Softw. 181, 111036.

Li, W., Zhang, W., Jia, X., Huang, Z., 2020. Effort-aware semi-supervised just-in-time defect prediction. Inf. Softw. Technol. 126, 106364.

Lin, H., Chen, X., Chen, X., Cui, Z., Miao, Y., Zhou, S., Wang, J., Su, Z., 2023. Gen-FL: Quality prediction-based filter for automated issue title generation. J. Syst. Softw. 195, 111513.

Liu, K., Yang, G., Chen, X., Yu, C., 2022a. Sotitle: A transformer-based post title generation approach for stack overflow. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 577–588.

Liu, K., Yang, G., Chen, X., Zhou, Y., 2022b. EL-CodeBert: Better exploiting CodeBert to support source code-related classification tasks. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware. pp. 147–155.

Liu, J., Zhou, Y., Yang, Y., Lu, H., Xu, B., 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 11–19.

Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.-i., Nakamura, M., 2010. An analysis of developer metrics for fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 1–9.

McIntosh, S., Kamei, Y., 2018. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans. Softw. Eng. 44 (5), 412–428.

Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.

Neto, E.C., da Costa, D.A., Kulesza, U., 2019. Revisiting and improving szz implementations. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 1–12.

Neto, E.C., Da Costa, D.A., Kulesza, U., 2018. The impact of refactoring changes on the szz algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 380–390.

Ni, C., Chen, X., Wu, F., Shen, Y., Gu, Q., 2019. An empirical study on pareto based multi-objective feature selection for software defect prediction. J. Syst. Softw. 152, 215–238.

Ni, C., Wang, W., Yang, K., Xia, X., Liu, K., Lo, D., 2022a. 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. pp. 672–683.

Ni, C., Xia, X., Lo, D., Chen, X., Gu, Q., 2020. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. IEEE Trans. Softw. Eng. 48 (3), 786–802.

Ni, C., Xia, X., Lo, D., Yang, X., Hassan, A.E., 2022b. Just-in-time defect prediction on JavaScript projects: A replication study. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4), 1–38.

Niu, C., Li, C., Ng, V., Ge, J., Huang, L., Luo, B., 2022. SPT-code: sequence-to-sequence pre-training for learning source code representations. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2006–2018.
Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. J. Syst. Softw. 150, 22–36.
Pornprasit, C., Tantithamthavorn, C.K., 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, pp. 369–379.
Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., Fu, M., Thongtanunam, P., 2021. Pyexplainer: Explaining the predictions of just-in-time defect models. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 407–418.
Qiao, L., Li, X., Umer, Q., Guo, P., 2020. Deep learning based software defect prediction. Neurocomputing 385, 100–110.
Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., Oliveto, R., 2021. Evaluating szz implementations through a developer-informed oracle. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 436–447.
Rosen, C., Grawi, B., Shihab, E., 2015. Commit guru: analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 966–969.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM Sigsoft Softw. Eng. Notes 30 (4), 1–5.
Tabassum, S., Minku, L.L., Feng, D., Cabral, G.G., Song, L., 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, pp. 554–565.
Tawosi, V., Sarro, F., Petrozziello, A., Harman, M., 2022. Multi-objective software effort estimation: A replication study. IEEE Trans. Softw. Eng. 48 (8), 3185–3205.
Trautsch, A., Herbold, S., Grabowski, J., 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 127–138.
Wan, Z., Xia, X., Hassan, A.E., Lo, D., Yin, J., Yang, X., 2018. Perceptions, expectations, and challenges in defect prediction. IEEE Trans. Softw. Eng. 46 (11), 1241–1266.
Wang, S., Liu, T., Nam, J., Tan, L., 2018. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46 (12), 1267–1293.
Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S., 2020. Just-in-time defect identification and localization: A two-phase framework. IEEE Trans. Softw. Eng. 48 (1), 82–101.
Yang, G., Chen, X., Cao, J., Xu, S., Cui, Z., Yu, C., Liu, K., 2021a. Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In: 2021 8th International Conference on Dependable Systems and their Applications (DSA). IEEE, pp. 30–41.
Yang, G., Chen, X., Zhou, Y., Yu, C., 2022a. Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 361–372.
Yang, G., Liu, K., Chen, X., Zhou, Y., Yu, C., Lin, H., 2022b. CCGIR: Information retrieval-based code comment generation method for smart contracts. Knowl.-Based Syst. 237, 107858.
Yang, X., Lo, D., Xia, X., Sun, J., 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. Inf. Softw. Technol. 87, 206–220.
Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, pp. 17–26.
Yang, G., Zhou, Y., Chen, X., Yu, C., 2021b. Fine-grained pseudo-code generation method via code feature extraction and transformer. In: 2021 28th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 213–222.
Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2023. ExploitGen: Template-augmented exploit code generation based on codebert. J. Syst. Softw. 197, 111577.
Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 157–168.
Yu, C., Yang, G., Chen, X., Liu, K., Zhou, Y., 2022. BashExplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 82–93.
Zeng, Z., Zhang, Y., Zhang, H., Zhang, L., 2021. Deep just-in-time defect prediction: how far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 427–438.
Zhang, Y., Harman, M., Mansouri, S.A., 2007. The multi-objective next release problem. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. pp. 1129–1137.
Zhang, T., Yu, Y., Mao, X., Lu, Y., Li, Z., Wang, H., 2022. FENSE: A feature-based ensemble modeling approach to cross-project just-in-time defect prediction. Empir. Softw. Eng. 27 (7), 1–41.
Zhao, Y., Damevski, K., Chen, H., 2022. A systematic survey of just-in-time software defect prediction. ACM Comput. Surv..
Zhao, K., Xu, Z., Zhang, T., Tang, Y., Yan, M., 2021. Simplified deep forest model based just-in-time defect prediction for android mobile apps. IEEE Trans. Reliab. 70 (2), 848–859.
Zheng, W., Shen, T., Chen, X., Deng, P., 2022. Interpretability application of the just-in-time software defect prediction model. J. Syst. Softw. 188, 111245.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively.

He is currently an Associate Professor at the Department of Information Science and Technology, Nantong University, Nantong, China.

He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software - Practice and Experience, Automated Software Engineering, Journal of Computer Science and Technology, IET Software, Software Quality Journal, Knowledge-based Systems, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER).

His research interests include software engineering, in particular software testing and maintenance, software repository mining, and empirical software engineering.

He received ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023.

He is the editorial board member of Information and Software Technology. More information about him can be found at: https://smartse.github.io/index.html.

Hongling Xia is currently pursuing the Master. degree with School of Information Science and Technology, Nantong University. His research interests include software defect prediction.

Wenlong Pei is currently pursuing the Master. degree with School of Information Science and Technology, Nantong University. His research interests include software defect prediction.

Chao Ni is with School of Software Technology, Zhejiang University as an assistant professor. His research interests are mainly in software engineering.

Ke Liu is currently pursuing the Master. degree with School of Information Science and Technology, Nantong University. Her research interests include software repository mining.