JIT-Smart: A Multi-task Learning Framework for Just-in-Time Defect Prediction and Localization

XIANGPING CHEN, Sun Yat-sen University, China
FUREN XU, Sun Yat-sen University, China
YUAN HUANG*, Sun Yat-sen University, China
NENG ZHANG, Sun Yat-sen University, China
ZIBIN ZHENG, Sun Yat-sen University, China

Just-in-time defect prediction (JIT-DP) is used to predict the defect-proneness of a commit and just-in-time defect localization (JIT-DL) is used to locate the exact buggy positions (defective lines) in a commit. Recently, various JIT-DP and JIT-DL techniques have been proposed, while most of them use a post-mortem way (e.g., code entropy, attention weight, LIME) to achieve the JIT-DL goal based on the prediction results in JIT-DP. These methods do not utilize the label information of the defective code lines during model building. In this paper, we propose a unified model JIT-Smart, which makes the training process of just-in-time defect prediction and localization tasks a mutually reinforcing multi-task learning process. Specifically, we design a novel defect localization network (DLN), which explicitly introduces the label information of defective code lines for supervised learning in JIT-DL with considering the class imbalance issue. To further investigate the accuracy and cost-effectiveness of JIT-Smart, we compare JIT-Smart with 7 state-of-the-art baselines under 5 commit-level and 5 line-level evaluation metrics in JIT-DP and JIT-DL. The results demonstrate that JIT-Smart is statistically better than all the state-of-the-art baselines in JIT-DP and JIT-DL. In JIT-DP, at the median value, JIT-Smart achieves F1-Score of 0.475, AUC of 0.886, Recall@20%Effort of 0.823, Effort@20%Recall of 0.01 and Popt of 0.942 and improves the baselines by 19.89%-702.74%, 1.23%-31.34%, 9.44%-33.16%, 21.6%-53.82% and 1.94%-34.89%, respectively . In JIT-DL, at the median value, JIT-Smart achieves Top-5 Accuracy of 0.539 and Top-10 Accuracy of 0.396, Recall@20%Effort$_{line}$ of 0.726, Effort@20%Recall$_{line}$ of 0.087 and IFA$_{line}$ of 0.098 and improves the baselines by 101.83%-178.35%, 101.01%-277.31%, 257.88%-404.63%, 71.91%-74.31% and 99.11%-99.41%, respectively. Statistical analysis shows that our JIT-Smart performs more stably than the best-performing model. Besides, JIT-Smart also achieves the best performance compared with the state-of-the-art baselines in cross-project evaluation.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Just-In-Time, Defect Prediction, Defect Localization, Multi-task Learning

ACM Reference Format:
Xiangping Chen, Furen Xu, Yuan Huang, Neng Zhang, and Zibin Zheng. 2024. JIT-Smart: A Multi-task Learning Framework for Just-in-Time Defect Prediction and Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 1 (July 2024), 23 pages. https://doi.org/10.1145/3643727

*Corresponding author.

Authors' addresses: Xiangping Chen, Sun Yat-sen University, Guangzhou, China, chenxp8@mail.sysu.edu.cn; Furen Xu, Sun Yat-sen University, Guangzhou, China, xufr@mail2.sysu.edu.cn; Yuan Huang, Sun Yat-sen University, Guangzhou, China, huangyuan5@mail.sysu.edu.cn; Neng Zhang, Sun Yat-sen University, Guangzhou, China, zhangn279@mail.sysu.edu.cn; Zibin Zheng, Sun Yat-sen University, Guangzhou, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2024/7-ART1
https://doi.org/10.1145/3643727

1 INTRODUCTION

Code review is a crucial process in Software Quality Assurance (SQA) [20, 44]. Through code review, defects in the project can be identified as early as possible and be repaired [9]. With the sharp increase in the number of software projects, developers manually reviewing source code will undoubtedly bring high-cost consumption and low work efficiency [23]. To solve this problem, just-in-time defect identification technology is proposed, which can be used to aid in code review when developers submit a commit. Then, the code change in the commit will go through an automatic process of defect identification before being merged into the master branch [51]. If the code change is identified as defective, it will be returned to the developer for correction [46].

Performing analysis on change-level software systems is a difficult task, especially for the software projects with years of development history, many developers, and a multitude of software artifacts including millions of lines of code[21]. Nowadays, most of the current studies identify the defect at coarse-grained level [13, 14, 48, 51], which are limited to identifying the bug-introducing commits (i.e., just-in-time defect prediction, JIT-DP). However, not all changed code lines involved in the commits are actually defective [44]. Therefore, some studies try to identify the exact buggy positions (i.e., defective code lines) in a commit [30, 33, 46], called just-in-time defect localization (JIT-DL) . Ni et al. [30] employed the commit-level defect labels to train a supervised JIT-DP model. Then, their model used the token attention weight generated while training the JIT-DP model to locate the defective code lines. Pornprasit et al. [33] also trained a JIT-DP model, and then employed the LIME [36] model to identify the defective code lines based on the prediction label of the JIT-DP model at commit-level. In general, these methods use a post-mortem way for JIT-DL based on the prediction results in JIT-DP phase, and the line-level label is not explicitly introduced when these models are built.

In this paper, we propose JIT-Smart, a multi-task learning framework both for JIT-DP and JIT-DL. The proposed model can identify bug-introducing commits (i.e. at commit-level) and locate the code line(s) introducing the defect(i.e. at line-level). We adopt a pre-trained model to extract the semantic information of text and code. More specifically, the main difference between our model and previous ones is that we design a novel defect localization network (DLN) specifically for JIT-DL, which explicitly introduces the label information of the defective code lines for supervised learning. The output of the model is the defect probability for each modified code line in the commit.

Since we leverage commit-level and line-level labels for supervised training, we treat JIT-DP and JIT-DL as a multi-task learning process and build a unified model that integrates defect prediction and localization. The purpose of building a unified model is that the defective commit means the defective line(s) exists (similarly, a bug-free commit means that no defective lines exist), so these two tasks are logically consistent. JIT-Smart relies on the learning of supervised information (i.e. labels), so we combine the loss functions of these two tasks to guide the training process of the model. For JIT-DP and JIT-DL, there exists the class imbalance issue. We draw on the design idea of the focal loss function [26] in the field of deep learning. This strategy starts from the learning process of the model, making the model learn to pay more attention to the small number of defective samples and difficult-to-classify samples. We design a loss function separately both for JIT-DP and JIT-DL and combine the two loss functions to jointly guide model training. Experimental results show that the two tasks are a mutual learning process that promotes each other. This also verifies the rationality for building a unified model. JIT-Smart can perform defect prediction and localization simultaneously after training on these two tasks. The granularity of the feature captured by these two networks (i.e. CodeBERT and DLN) is different, and joint modeling and joint optimization are beneficial to promote their respective performances. The examples in Section 5 and experimental results prove this point.

To the best of our knowledge, JIT-Smart is the first network structure specially designed for JIT-DL, which explicitly introduces the label information of defective code lines for supervised learning for JIT-DP and JIT-DL. Experimental results show that the special design of DLN can effectively capture the contextual semantic information between code lines and achieve excellent defect localization performance, and integrate the code line structure information can also be helpful for JIT-DP. Specifically, we combine four kinds of features as input: commit message (natural language information), code changes of the commits (code language information), code changes matrix (code line structure information), 14 commit-level expert features.

We conduct experiments on a large public dataset JIT-Defects4J [30], which contains 21 software projects. The experimental results indicate that JIT-Smart can outperform the 7 state-of-the-art baselines in both JIT-DP and JIT-DL in terms of all performance measures with a substantial improvement. Specifically, in JIT-DP, at the median value, JIT-Smart achieves F1-Score of 0.475, AUC of 0.886, Recall@20%Effort of 0.823, Effort@20%Recall of 0.01 and Popt of 0.942 and improves the baselines by 19.89%-702.74%, 1.23%-31.34%, 9.44%-33.16%, 21.6%-53.82% and 1.94%-34.89%, respectively. In JIT-DL, at the median value, JIT-Smart achieves Top-5 Accuracy of 0.539 and Top-10 Accuracy of 0.396, Recall@20%Effort$_{line}$ of 0.726, Effort@20%Recall$_{line}$ of 0.087 and IFA$_{line}$ of 0.098 and improves the baselines by 101.83%-178.35%, 101.01%-277.31%, 257.88%-404.63%, 71.91%-74.31% and 99.11%-99.41%, respectively. Statistical analysis shows that our JIT-Smart performs more stably than the best-performing model. As well as in cross-project testing settings, JIT-Smart also achieves the best performance.

To facilitate research and application, the replication package of our JIT-Smart is available at: https://github.com/JIT-A/JIT-Smart.

In summary, this paper makes the following novelty and contributions:

- **A novel modeling approach for just-in-time defect prediction and localization tasks.** We regard JIT-DP and JIT-DL as a multi-task learning process and propose a unified model JIT-Smart which aims to automatically learn the semantic and expert features in commits to identify bug-introducing commits and locate defective code lines simultaneously. We design a novel multi-task learning loss function for training process to handle the class imbalance issue.
- **A novel defect localization network.** Different from previous studies, we design a novel defect localization network (DLN) for JIT-DL, which explicitly introduces the label information of defective code lines for supervised learning. This distinguishes our model by design from other models for JIT-DL through post-mortem analysis.
- **Comprehensive experimental evaluation.** We conduct experiments on a large dataset, JIT-Defects4J [30], which contains 21 software projects. To avoid the influence of random factors, we conduct multiple rounds of experiments on all baselines and perform statistical analysis. The experimental results show that JIT-Smart is statistically better than the 7 state-of-the-art baselines on both JIT-DP and JIT-DL in terms of all performance measures with a substantial improvement, even in cross-project evaluation. Statistical analysis shows that our JIT-Smart performs more stably than the best-performing model. We analyze how the different loss function weight assignments affect the performance of JIT-Smart.

The rest of our paper is organized as follows. Section 2 introduces the details of our approach. Section 3 describes the experimental setup. Section 4 reports the experimental results. Section 5 is a qualitative analysis of our study. Section 6 presents the threats to the validity of our study. Section 7 discusses the related work. We draw conclusions in Section 8.

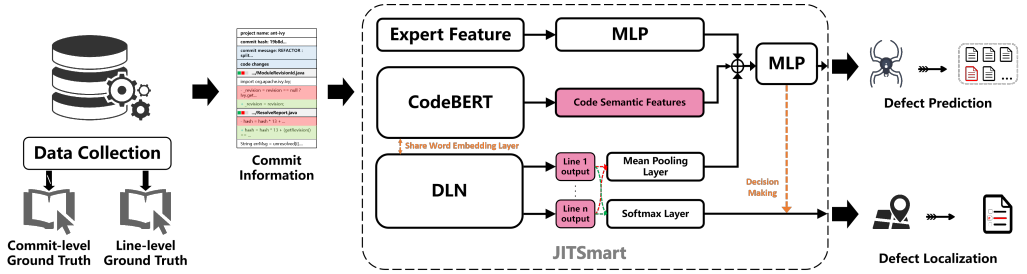


Fig. 1. An overview of JIT-Smart for the JIT-DP and the JIT-DL tasks

2 APPROACH

Figure 1 shows the main steps of our approach. Firstly, a public dataset is introduced, which contains the commits and the corresponding commit-level and line-level labels. Secondly, we construct our model JIT-Smart. Specifically, we extract 14 commit-level expert features[22] and integrate them by a nonlinear network. We adopt the pre-trained model to extract semantic information of natural language (i.e. commit messages) and source code (i.e. code changes). In particular, we design a defect localization network for JIT-DL. Thirdly, we treat JIT-DP and JIT-DL as a multi-task learning process to train JIT-Smart. Lastly, we use the trained model to perform JIT-DP and JIT-DL. Details of each part are presented in the following subsections.

JIT-Smart contains two main components: DP-Network (DPN, i.e. CodeBERT) and DL-Network (DLN). In our design, DLN is tightly coupled DPN. (1) From the perspective of training, firstly the two networks share the word embedding layer. Second, the combined loss of the two tasks can optimize the two parts of the network at the same time during training. (2) From the perspective of prediction, the results of JIT-Smart's DP network are used to assist in correcting the results of the DL network. As the commit-level decision maker, the DP network combines the mean pooling features from the DLN, and only when the DP network predicts defective commits, JIT-Smart will decide to use the prediction result of the DLN. DPN extracts more comprehensive and coarse-grained commit-level information, while DLN extracts finer-grained line-level information. DLN outputs the feature vector of each code line (the vector can be converted into the output probability of each code line through the fully connected layer).

We only utilize the strategy that the results of DLN will be adopted when the DP model predicts a commit as defective in the prediction stage we do not apply to the model training stage. Because we want DPN and DLN to automatically learn this decision-making relationship in model training and be more accurate when making decisions in the prediction stage.

2.1 Dataset Preparation and Preprocessing

Dataset Preparation. Our experiments are based on the large-scale commit-level and line-level dataset JIT-Defects4J proposed by Ni et al. [30]. The description of the dataset is shown in Table 1. The reason why we choose JIT-Defects4J is that it is larger and more fine-grained in the field of JIT-DP and JIT-DL than the dataset in [13, 14, 33, 46, 48, 51], including 21 java projects, covering multiple software directions (e.g. related to client usage network protocols, scientific computing related, Java application configuration management related, natural language text processing related). This dataset is not preprocessed, it is raw data. The dataset contains various information about commits (e.g. commit messages, code changes, etc.) and label information for defective commits and defective code lines. As shown in Table 1, JIT-Defects4J contains a total of 27,319 commits, of which 8.54% are defective. These commits involve 131,962 code lines, of which 9.54%

are defective code lines. JIT-Defects4J is based on the LLT4J dataset [12] which analyzes the tangled bug fixing commits. The collection of JIT-Defects4J data is mainly achieved through the version control system and PyDriller (a python package). Additionally, JIT-Defects4J contains 14 expert features which are directly extracted by the CommitGuru [37]. These 14 expert features proposed by Kamei et al. [22], which are defined from five dimensions (i.e. diffusion, size, purpose, history, experience), are widely used in the field of just-in-time defect prediction [30, 47, 48, 50].

Table 1. Statistics of datasets: JIT-Defects4J

Java Project	Commit	% Ratio$_c$	Code Line	% Ratio$_l$
ant-ivy	1,771	18.75%	16,503	10.00%
commons-bcel	825	7.27%	1,936	16.01%
commons-beanutils	611	6.06%	1,847	6.66%
commons-codec	761	4.73%	2,320	10.60%
commons-collections	1,823	2.74%	3,615	5.01%
commons-compress	1,630	10.92%	7,794	8.04%
commons-configuration	1,838	8.43%	7,984	8.14%
commons-dbcp	1,037	5.59%	2,692	7.13%
commons-digester	1,079	1.76%	472	18.43%
commons-io	1,142	6.39%	3,018	6.49%
commons-jcs	831	10.59%	5,246	8.58%
commons-lang	2,969	4.92%	6,895	8.17%
commons-math	4,026	8.32%	20,006	15.23%
commons-net	1,121	10.44%	5,275	8.36%
commons-scxml	544	8.64%	4,296	5.63%
commons-validator	598	6.02%	1303	8.75%
commons-vfs	1,110	10.27%	5286	7.26%
giraph	844	19.31%	18652	10.21%
gora	553	7.05%	3530	3.77%
opennlp	1,086	8.38%	4238	7.69%
parquet-mr	1,120	14.11%	9054	8.05%
ALL	**27,319**	**8.54%**	**131,962**	**9.54%**

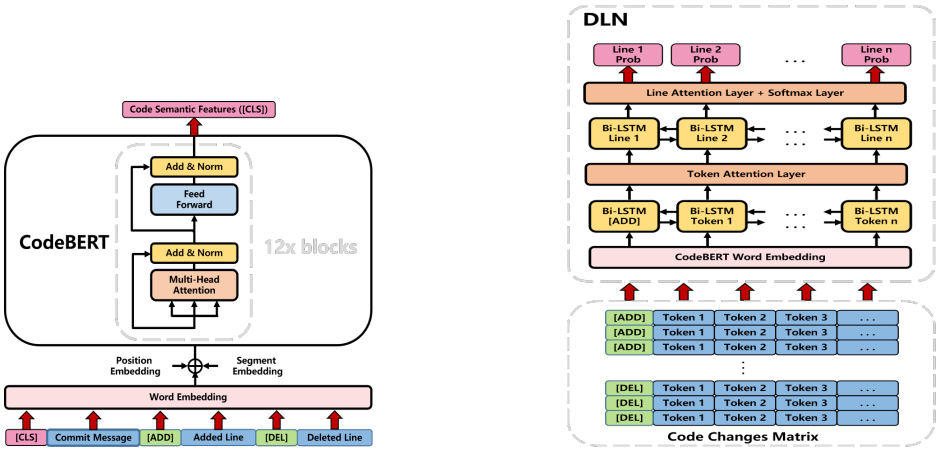
* Ratio$_c$: defective commits/ALLS. Ratio$_l$: defective lines/ALLS

Dataset Preprocessing. For the processing of the input data of the model, in the first step, we replace the real numbers such as integers in the code with an identifier <*num*>, replace the constant strings with an identifier <*str*>, and remove the special characters (e.g. (*space*), (,)), etc. In the second step, we use the popular BPE algorithm [38], which can well alleviate the problem of out-of-vocabulary to tokenize the input sequence. The third step is to convert the sequence into its index in the corresponding dictionary. For a further aggregation of the 14 expert features, we first normalize each column of features and then use a multi-layer perceptron(MLP) to perform a nonlinear mapping. The input and output dimensions of this layer are 14, 768 respectively.

2.2 Code and Natural Language Semantic Information Extraction

In the research field of the combination of software engineering and AI, most of the current research relies on how to better represent the code language or natural language to complete some basic downstream tasks such as defect detection and localization tasks [30, 31], code summary generation task [7, 27, 39, 45], and code language conversion task [7, 10]. The CodeBERT model we adopt is proposed by Feng et al. [7], which is pre-trained on large data covering six programming languages (i.e. Go, Java, Javascript, PHP, Python, Ruby) and natural languages. Since the JIT-DP task includes commit messages (natural language) and code changes (code language), the CodeBERT pre-training model can extract the semantic information of code and natural language. Then we finetune training on JIT Dataset to make the model suitable for downstream tasks (JIT-DP and JIT-DL).

CodeBERT adopts the structure of BERT [6] which consists of twelve transformer blocks [42] and the dimension of hidden states is 768. The main purpose of the model is to capture the semantic information between code tokens through the multi-head attention mechanism in Figure 2a. The multi-head attention layer in CodeBERT consists of 12 scaled dot-product attention layers.



(a) The structure of semantics extractor

(b) The structure of DLN

Fig. 2. The main network module of JIT-Smart

Specifically, we input two kinds of information to CodeBERT: commit message and code changes of the commits (i.e., added lines and deleted lines) into the model. In particular, as shown in Figure 2a, we separate each added line and deleted line by using the special separator "[ADD]" "[DEL]". At the head of the entire sequence, we add a special "[CLS] identifier" like the BERT model [6], which can be used as semantic information for the entire sequence. After the calculation of multi-head attention mechanism, the semantic information of "[CLS]" token is equivalent to the aggregated representation of other tokens, which can be used to complete some downstream tasks (e.g. classification tasks). In our work, we adopt the features of the "[CLS]" token extracted by the CodeBERT model to represent the contextual semantic information of commit messages and code changes of the commits for JIT-DP.

2.3 Defect Localization Network

The previous studies do not make full use of the label information of the defective code lines to train [30, 33, 46]. The main reason is that they all rely on some existing deep learning models to complete JIT-DP at a coarse-grained level. There is no suitable model structure specially designed for the defect localization problem.

Specifically, the main difference between ours and previous JIT-DP and JIT-DL studies is that we design a novel defect localization network (DLN) for JIT-DL, as shown in Figure 2b. The input of the DLN is Code Changes Matrix (CCM), while the output of the DLN is Code Changes Matrix Features (CCMF). The structure of DLN adopts a hierarchical network design. DLN mainly consists of five parts: a word embedding layer, a code token sequence encoder, a token-level attention layer, a code line sequence encoder, and a line-level attention layer. We will introduce the working principles of these five parts as follows.

Word embedding layer. The word embedding layer of DLN and CodeBERT models is used as a shared structure and the dimension of the embedding vector is 768. In this layer, the input is the code changes matrix. DLN encodes the code lines into a two-dimensional matrix which preserves the structure information of the code lines.

Code token sequence encoder. Since there is a certain contextual relationship between a token and its surrounding tokens, we adopt a bidirectional long short-term memory network(Bi-LSTM) [15] to capture the semantic relationship. Bi-LSTM has been widely used in software engineering [16–19, 43]. The dimension of hidden states in LSTM is 256. The number of LSTM layers is 1.

Token-level attention layer. Since not all code tokens have an equal contribution to the semantic representation of code lines, we adopt an attention mechanism [42] to assign different attention weights to different code tokens, and then aggregate them to form a vector representing the semantic information of the code line.

Code line sequence encoder. Since there are contextual correlations between different code lines, we adopt Bi-LSTM to capture the contextual semantic features of code lines. The dimension of hidden states in LSTM is 512. The number of Bi-LSTM layers is 1.

Line-level attention layer. Since not all code lines have an equal contribution to the semantic representation of code changes, we adopt an attention mechanism to assign weight to different line-level semantic features. We do not aggregate the semantic features of these lines of code in this step. Finally, we pass the features of these code lines to the classifier for JIT-DL. Moreover, we perform an average pooling of these features to obtain a feature representing the structural semantic information of code changes, which can be used to assist JIT-DP (experimental results show that this feature can contribute to JIT-DP.)

In general, DLN explicitly utilizes the label information of the defective code lines for supervised learning, and output the defect probability of each modified code line in a commit.

2.4 Handling the Class Imbalance Issue

In JIT-DP and JIT-DL, the JIT defect datasets we use are highly imbalanced. Previous JIT-related research [3, 33] focuses on using the under-sampling or over-sampling strategy (such as SMOTE [4]) to alleviate the class imbalance issue. The disadvantage of alleviating the imbalance problem through the sampling method is that the distribution of the actual data categories is changed, making the category distribution of the training set and the test set inconsistent. These methods do not actually alleviate the model's ability to discriminate between samples with a small proportion of classes and samples that are difficult to classify.

We draw on the idea of the focal loss function [26] to alleviate the class imbalance issue in JIT-DP and JIT-DL. We are the first to apply focal loss to the field of defect prediction and localization. The goal of focal loss is to solve the problem of imbalanced sample categories and imbalanced sample classification difficulty. Since there exists the class imbalance issue in JIT-DP and JIT-DL, we refer to the idea of focal loss to design a loss function for JIT-DP and JIT-DL, respectively. We call them JIT-DP loss and JIT-DL loss.

The loss function in the JIT-DP task is as follows:

$$JIT_{DP}\ loss = -\alpha_t(1-p_t)^{\gamma}\log(p_t) \tag{1}$$

where p_t is the predicted probability of t class (there are only two classes in JIT-DP, so $t \in [0,1]$), $\alpha_t \in [0,1]$, $\gamma \in [0,5]$. According to previous experience [26], the performance is the best when they set: $\alpha_t = 0.25$, $\gamma = 2.0$. Since the purpose of focal loss is the same as ours, which is to solve the class imbalance issue, we adopt the default optimal parameters. The balance factor α_t is used to balance the unbalanced proportion of positive and negative samples, $(1-p)$ makes the model pay more attention to the samples with a small number of categories and difficult to classify. γ regulates the rate at which the weight of simple samples is reduced.

In JIT-DL, the loss function as follows:

$$JIT_{DL}\ loss = -\alpha_t(1-p_t)^{\gamma}\log(p_t) \tag{2}$$

Since the existence of defective lines indicates the existence of defective commits and if commits are not defective, there must be no defective code lines, these two tasks are related. Therefore we combine the loss functions of these two tasks to guide the training process of the model. Lastly, we add the weighted JIT-DP loss and JIT-DL loss as the loss function for our unified model training as follows:

$$JIT\ loss = \omega_{dp} * JIT_{DP}\ loss + \omega_{dl} * JIT_{DL}\ loss \tag{3}$$

In the experiment, we find that when $\omega_{dp} = 0.3$ and $\omega_{dl} = 0.7$, the experimental performance is the best.

3 EXPERIMENTAL SETUP

In this section, we first introduce the studied baselines. Then we describe 10 evaluation metrics we adopt. Finally, we present the key parameter settings of the model.

3.1 Baselines

We use the source code provided in the corresponding research instead of implementing these baselines ourselves. We set the default optimal parameters and operating methods from the corresponding papers to ensure the accuracy of our reproduction.

LApredict [51]. LApredict simply builds a logistic regression classifier with adopting the "added-line-number" features to perform JIT-DP.

CC2Vec [14]. CC2Vec is a code language representation framework, which is based on the hierarchical attention network [49] to extract the semantic information of code changes of the commits for JIT-DP.

DeepJIT [13]. DeepJIT extracts the interactive semantic information of commit messages and code changes based on Text-CNN model [24] for JIT-DP.

Deeper [48]. Deeper is proposed to leverage a Deep Belief Network with 14 commit-level expert features for JIT-DP.

JIT-Fine [30] JIT-Fine adopts the CoderBERT pre-trained model to preform defect prediction and uses the attention weights generated by the CodeBERT to perform defect localization by combining the semantic features and expert features.

JITLine [33]. JIT-Line utilizes random forest model [2] and LIME method [36] for defect prediction and localization by combining expert features and token features.

TPF-JIT [46]. Yan et al. propose a two-phase framework (we call "TRF-JIT" in our paper) for JIT-DP and JIT-DL. TRF-JIT uses a logistic regression model with expert features for JIT-DP and leverage the N-gram model with code entropy for JIT-DL.

3.2 Evaluation Metrics

In order to evaluate our model comprehensively, we adopt the following 10 performance metrics which include 5 commit-level and 5 line-level metrics. Since in the actual SQA code review stage, the SQA team has limited resources to inspect where the defects locate, the 10 evaluation metrics we adopt include effort-agnostic and effort-aware types of metrics.

The commit-level metrics for JIT-DP:

F1-Score. F1-Score is a commonly used metric in binary classification tasks and is a reasonable metric for class imbalance problems. It is a trade-off between $recall = \frac{TP}{TP+FN}$ and $precision = \frac{TP}{TP+FP}$ (TP: true positive, FP: false positive, TN: true negative, FN: false negative). $F1 - Score = \frac{2*recall*precision}{recall+precision}$

AUC. AUC is the area under the receiver operating characteristic curve (ROC) [11]. AUC is a reasonable metric for class imbalance problems and comprehensively considers the performance

under all thresholds, so in the case of unbalanced samples, AUC is a more reasonable evaluation measure. The closer the AUC is to 1.0, the higher the authenticity of the detection method is. The authenticity is the lowest when it is less than or equal to 0.5.

Recall@20%Effort (R@20%E). R@20%E measures the proportion of the actual bug-introducing commits that model finds with 20% lines of code (LOC). A high value of R@20%E indicates that the more defective commits are identified by the model with less effort, and are ranked at the top of the list under code review process.

Effort@20%Recall (E@20%R). E@20%R is used to measure the proportion of code lines that need to be reviewed for 20% of the bug-introducing commits found by the model to the total number of lines of code. A higher value means that the model can find the bug-introducing commits with less effort.

Popt. Popt [1] is used to evaluate the relationship between effort (number of code lines reviewed) and recall. The closer the Popt is to 1.0, the best the model.

The line-level metrics for JIT-DL:

Top-5/10 Accuracy. Top-5/10 Accuracy is calculated by only considering the prediction results corresponding to the real defect commits. It measures the proportion of actual defective lines that are ranked in the top-5/10, a larger value means that the model can more accurately rank the actual defective lines higher for developers to review efficiently. Top 5/10 Accuracy has been widely used in JIT-DL, such as in [30, 34, 46].

Recall@20%Effort$_{line}$ (R@20%E$_l$). Given a defective commit, we first sort all the code lines in the commit in descending order according to the defect scores, and then calculate how many real defective code lines are recalled in the top 20% of the code lines.

Effort@20%Recall$_{line}$(E@20%R$_l$). Given a defective commit, we first sort all the code lines in the commit in descending order according to the defect scores. Then calculate to find out 20% of the defective lines, the proportion of the number of defective lines to be reviewed to the total number of lines of code.

Initial False Alarm$_{line}$ (IFA$_l$). Given a defective commit, we first sort all the lines in the commit in descending order according to the defect scores and then calculate the number of real clean lines that have been reviewed when the first defective line is found.

3.3 Statistical Analysis

In order to reduce the impact of random factors on evaluating the performance of each model, we conduct 5 rounds of experiments on each model and then perform statistical analysis on the experimental results to more accurately measure the performance and stability of each model. Specifically, we analysis the following two aspects.

Performance Gain[44]. To determine whether our JIT-Smart is better than the baselines, we use the performance gain metric to compute the percentage of the performance difference between our JIT-Smart and the baselines.

$$\%Diff = \frac{\sum (Perf_{JIT-Smart} - Perf_{baseline})}{\sum Perf_{baseline})} \tag{4}$$

Statistical Test. We employ the one-sided Wilcoxon-signed rank test to validate the statistical distinction. In detail, we assess the efficacy of our JIT-Smart in relation to the baseline methods. We choose the Wilcoxon signed-rank test because it is a non-parametric method suited for pairwise evaluation of two distributions. This enables us to compare the performance of our JIT-Smart with the baselines on the same dataset setting.

We also measure the effect size (r) which is the magnitude of the difference between two distributions using the following calculation:

$$r = \frac{Z}{\sqrt{n}} \tag{5}$$

where Z is a statistic Z-score from the Wilcoxon signed-rank test and n is the total number of samples [41]. $r > 0.5$ indicates large, $0.3 < r \leq 0.5$ indicates medium, and $0.1 < r \leq 0.3$ is small, otherwise negligible[8].

3.4 Key Parameter Settings

We mentioned above that the input data (i.e. code changes matrix) dimension of the DLN is a two-dimensional form: $[n_l, n_t]$ (n_l: number of lines in code changes, n_t: number of tokens in each code line). Since the neural network model is trained in mini batches, it is necessary to set a fixed value for the two dimensions of the code changes matrix. If the number of lines in code changes or number of tokens in each code line is smaller than this fixed value, it needs to be filled, otherwise, it needs to be truncated.

First, we separately count the distribution of the number of lines in code changes and the number of tokens in each code line in the dataset. As shown in Table 2, each row represents the statistics of the number of code lines in the commits or the number of tokens in each line. The statistical indicators include the statistics of the average value, standard deviation, minimum value, and maximum value. Finally, we decide to set the first dimension of the code changes matrix to 256 and the second dimension to 64, because the coverage ratio can reach more than 95%.

Table 2. Statistics of the length distribution of code changes

Statistic Type	Mean	Std	Min	Max	Coverage Ratio
Code lines (train)	55.52	141.41	1	6092	95.8%
Code tokens (train)	14.12	9.38	1	570	99.8%
Code lines (valid)	47.4	118.61	1	2515	97%
Code tokens (valid)	15.1	9.31	1	635	99.8%
Code lines (test)	59.78	152.26	1	1969	95.3%
Code tokens (test)	15.87	10.33	1	917	99.8%

* Coverage Ratio: the proportion of the number below the threshold of 256 or 64.

4 EXPERIMENTAL RESULTS

In this section, we present the motivation, approach and results for the following four research questions:

RQ1: How effective is JIT-Smart in just-in-time defect prediction?

RQ2: How well can JIT-Smart locate defective lines in just-in-time defect localization?

RQ3: What is the accuracy of JIT-Smart compared to the state-of-the-art baseline in the cross-project experimental?

RQ4: How do the different loss function weight assignments affect the performance of JIT-Smart?

4.1 RQ1: How effective is JIT-Smart in just-in-time defect prediction?

Motivation. Various methods have been proposed in the research of just-in-time defect prediction, JIT-DP models have been built based on traditional machine learning or deep learning methods by extracting different features (i.e. semantic features, expert features, token features). Our JIT-Smart is a unified model which treats JIT-DP and JIT-DL as a multi-task learning process, hence our model can identify both bug-introducing commits and defective code lines. In this RQ, we try to explore how JIT-Smart performs in JIT-DP compared to other state-of-the-art models.

Approach. For a fair comparison, we use the training set, validation set, and test set divided from the JIT-Defects4J dataset. Specifically, 60% of commits in each project are treated as training data, 20% of commits in each project are treated as validating data, 20% of commits in each project are treated as testing data. All studied baselines we compare use the same dataset for training and testing. We use 5 commit-level evaluation metrics for JIT-DP.

Finally, we also evaluate whether the code changes matrix features that preserve the structural semantic features of code lines generated by DLN are helpful for JIT-DP.

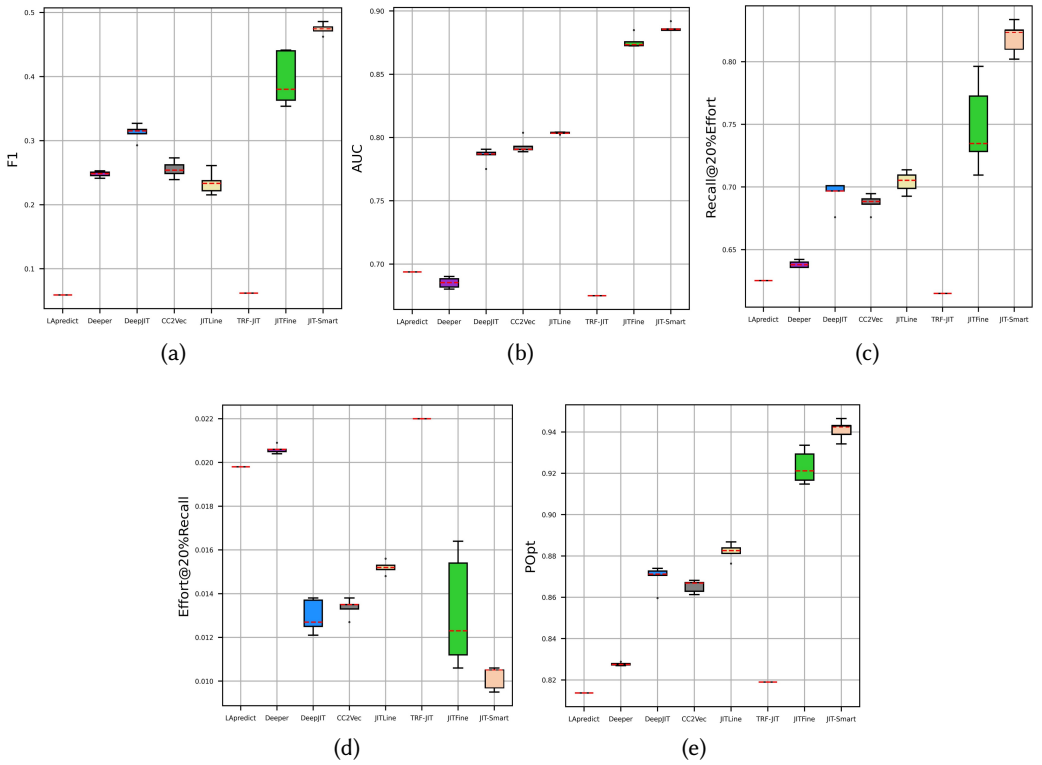


Fig. 3. The JIT-DP performance of JIT-Smart and baselines

Table 3. A comparative summary of the JIT-DP performance between JIT-Smart and the baselines.

JIT-Smart vs Baseline	F1-Score↑		AUC↑		R@20%E↑		E@20%R↓		Popt↑	
	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)
LApredict	**702.74%**	L (p<0.05)	**27.78%**	L (p<0.05)	**30.97%**	L (p<0.05)	**-48.69%**	L (p<0.05)	**15.65%**	L (p<0.05)
Deeper	**91.48%**	L (p<0.05)	**29.38%**	L (p<0.05)	**28.30%**	L (p<0.05)	**-50.68%**	L (p<0.05)	**13.69%**	L (p<0.05)
DeepJIT	**51.78%**	L (p<0.05)	**12.86%**	L (p<0.05)	**17.96%**	L (p<0.05)	**-21.60%**	L (p<0.05)	**34.89%**	L (p<0.05)
CC2Vec	**85.77%**	L (p<0.05)	**11.75%**	L (p<0.05)	**19.18%**	L (p<0.05)	**-23.95%**	L (p<0.05)	**8.75%**	L (p<0.05)
TPF-JIT	**665.19%**	L (p<0.05)	**31.34%**	L (p<0.05)	**33.16%**	L (p<0.05)	**-53.82%**	L (p<0.05)	**14.90%**	L (p<0.05)
JITLine	**102.93%**	L (p<0.05)	**10.34%**	L (p<0.05)	**16.33%**	L (p<0.05)	**-33.16%**	L (p<0.05)	**6.67%**	L (p<0.05)
JIT-Fine	**19.89%**	L (p<0.05)	**1.23%**	L (p<0.05)	**9.44%**	L (p<0.05)	**-22.91%**	L (p<0.05)	**1.94%**	L (p<0.05)

Effect Size: Large(L) r>0.5, Medium(M) 0.3 <r $\leq$ 0.5, Small(S) 0.1<r $\leq$ 0.3, Negligible(N) r<0.1
The bold text indicates that JIT-Smart is better than the baselines.

Results. Figure 3 shows that our JIT-Smart achieves the best performance compared to the baselines on all evaluation metrics. The red dotted line in the figure represents the median. At the median value, our JIT-Smart achieves F1-Score of 0.475 and AUC of 0.886, which are better

than the baselines (F1-Score of 0.059 to 0.38, AUC 0.675 to 0.873). Table 3 shows that the F1-Score values of our JIT-Smart are 19.89%-702.74% higher than all baselines and the AUC values of our JIT-Smart are 1.23%-31.34% higher than all baselines. This verifies that our approach JIT-Smart can accurately identify more bug-introducing commits and better distinguish defective samples from non-defective samples.

Moreover, at the median, our JIT-Smart achieves Recall@20%Effort of 0.823, Effort@20%Recall of 0.011, and Popt of 0.942, which are better than the baselines (Recall@20%Effort of 0.615 to 0.7347, Effort@20%Recall of 0.012 to 0.022, and Popt of 0.8137 to 0.9212). Table 3 shows that the Recall@20%Effort values of our JIT-Smart are 9.44%-33.16% higher than all baselines, the Effort@20%Recall values of our JIT-Smart are 21.6%-53.82% lower than all baselines, and the Popt values of our JIT-Smart are 1.94%-34.89% higher than all baselines. This shows that our model can find more bug-introducing commits with less effort.

From the results of statistical analysis, among all evaluation metrics in Table 3, the one-sided Wilcoxon-signed rank tests also confirm the statistical significance (p-value < 0.05) with a large effect size. It is worth noting that from the distribution of the box plot, we can see that our model is more stable than the best-performing model (i.e. JIT-Fine) in JIT-DP, which means that our model will not be affected by random factors that cause large fluctuations in performance.

Besides, LApredict and TRF-JIT perform poorly on all evaluation metrics, because these two models are based on traditional machine learning with expert features or token features, which does not fully mine the deep semantic information of the commits. For this motivation, we explore features that are helpful for our model. In this experiment, we use the optimal random seed (i.e. 0) to conduct one round of experiments for each model. The results from Table 4 show that on the basis of code semantic features and expert features, adding the structural-semantic features of code lines can further improve the model on F1-Score and AUC (JIT-Smart vs. JIT-Smart$_{NO\ CCMF}$ in Table 4). Finally, we verify whether the expert features are helpful for our JIT-Smart, and it can be seen from Table 4 that the performance of the model is degraded without using the expert features.

Table 4. The impact of different features for JIT-Smart

Models	F1-Score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑
JIT-Smart$_{NO\ CCMF}$	0.473	0.881	0.819	0.01	0.935
JIT-Smart$_{NO\ EF}$	0.383	0.866	0.794	0.011	0.930
JIT-Smart	**0.486**	**0.885**	**0.823**	**0.01**	**0.942**

"↑" indicates the larger the better; "↓" indicates the smaller the better.
"*NO CCMF*": not using the output of DLN mean-pooling features."*NO EF*": not using expert features.

4.2 RQ2: How well can JIT-Smart locate defective lines in just-in-time defect localization?

Motivation. Since not all changed lines are defective in bug-introducing commits, just-in-time defect localization is needed to help developers quickly locate defective code lines with less effort. However, there exist only a few approaches for just-in-time defect localization. Different from previous work, JIT-Smart can perform defect prediction and localization simultaneously. We design a novel defect localization network (DLN) for JIT-DL by explicitly introducing the label information of the defective code lines for supervised learning. Therefore, we investigate whether JIT-Smart is able to locate defective code lines more accurately and cost-effectively than the state-of-the-art baselines in JIT-DL.

Approach. We utilize the label of the existing defective code lines in the JIT-Defects4J dataset for JIT-Smart in supervised training, and transform the code lines of code changes into the form introduced in Section 2.3 and put it into the model for training. During the testing phase, we use the prediction results obtained by the trained model for evaluation. Specifically, to evaluate

the accuracy and the cost-effectiveness of JIT-Smart in JIT-DL, we use 5 widely used line-level evaluation metrics for a fair comparison.

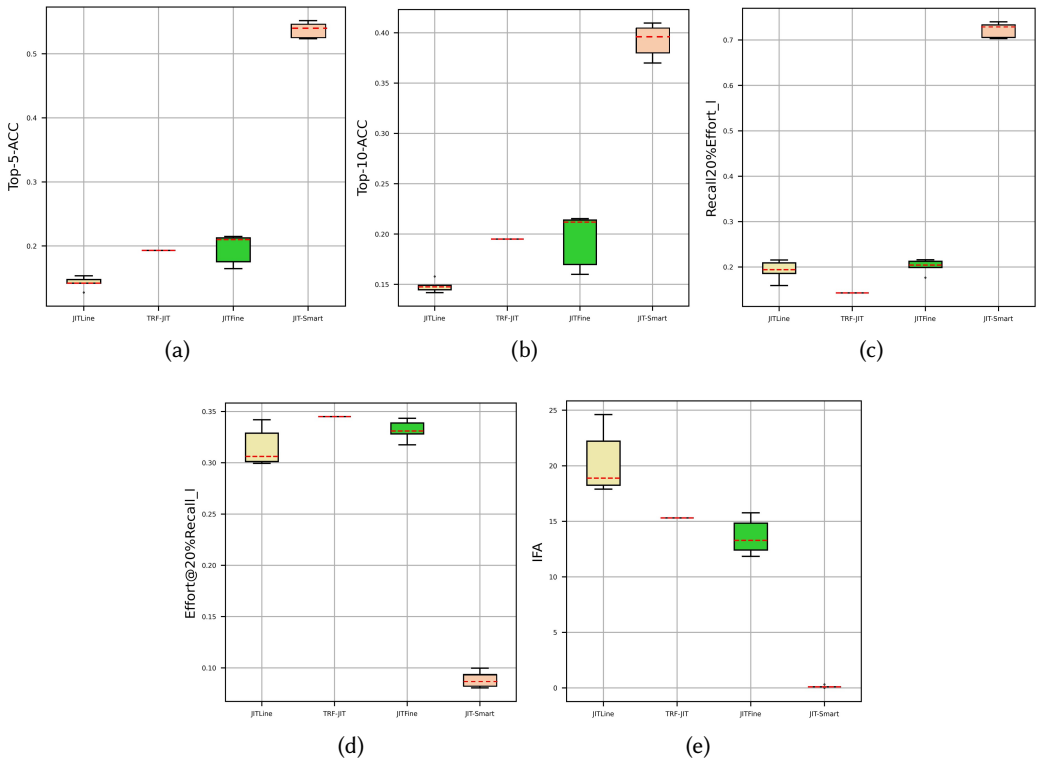


Fig. 4. The JIT-DL performance of JIT-Smart and baselines

Table 5. A comparative summary of the JIT-DL performance between JIT-Smart and the baselines.

JIT-Smart vs Baseline	Top-5 ACC ↑		Top-10 ACC ↑		R@20%E$_l$ ↑		E@20%R$_l$ ↓		IFA$_l$ ↓	
	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)	%Diff	Effect Size (p-value)
JITLine	**164.48%**	L (p<0.05)	**277.31%**	L (p<0.05)	**274.32%**	L (p<0.05)	**-71.91%**	L (p<0.05)	**-99.41%**	L (p<0.05)
TPF-JIT	**178.35%**	L (p<0.05)	**101.01%**	L (p<0.05)	**404.63%**	L (p<0.05)	**-74.31%**	L (p<0.05)	**-99.21%**	L (p<0.05)
JIT-Fine	**101.83%**	L (p<0.05)	**174.79%**	L (p<0.05)	**257.88%**	L (p<0.05)	**-73.28%**	L (p<0.05)	**-99.11%**	L (p<0.05)

Effect Size: Large(L) r>0.5, Medium(M) 0.3 <r ≤ 0.5, Small(S) 0.1<r ≤ 0.3, Negligible(N) r<0.1
The bold text indicates that JIT-Smart is better than the baselines.

Results. Figure 4 shows that our JIT-Smart achieves the best performance compared to the baselines on all evaluation metrics. At the median value, our JIT-Smart achieves Top-5 Accuracy of 0.539 and Top-10 Accuracy of 0.396, which are better than the baselines (Top-5 Accuracy of 0.142 to 0.21, Top-10 Accuracy of 0.148 to 0.212). Table 5 shows that the Top-5 Accuracy values of our JIT-Smart are 101.83%-178.35% higher than all baselines and the Top-10 Accuracy values of our JIT-Smart are 101.01%-277.31% higher than all baselines. This also verifies that JIT-Smart can more accurately locate and rank defective lines at the top.

Moreover, at the median, our JIT-Smart achieves Recall@20%Effort$_{line}$ of 0.726, Effort@20%Recall of 0.087, and IFA$_{line}$ of 0.098, which are better than the baselines (Recall@20%Effort$_{line}$ of 0.143 to 0.204, Effort@20%Recall$_{line}$ of 0.306 to 0.345, and IFA$_{line}$ of 13.282 to 18.884). Table 5 shows that

the Recall@20%Effort$_{line}$ values of our JIT-Smart are 257.88%-404.63% higher than all baselines, the Effort@20%Recall$_{line}$ values of our JIT-Smart are 71.91%-74.31% lower than all baselines, and the IFA$_{line}$ values of our JIT-Smart are 99.11%-99.41% higher than all baselines. This shows that JIT-Smart can identify more defective lines with less effort than all baselines. For developers, JIT-Smart will help them spend less effort to find the first actual defective line. JIT-Smart can identify more actual defective lines.

From the results of statistical analysis, among all evaluation metrics in Table 5, the one-sided Wilcoxon-signed rank tests also confirm the statistical significance (p-value < 0.05) with a large effect size. It is worth noting that from the distribution of the box plot, we can see that our model is more stable than the best-performing model (i.e. JIT-Fine) in JIT-DL, which means that our model will not be affected by random factors that cause large fluctuations in performance.

Table 6. The impact of attention mechanism for JIT-Smart

Models	Accuracy↑		R@20%E$_l$ ↑	E@20%R$_l$ ↓	IFA$_l$ ↓
	Top-5	Top-10			
JIT-Smart$_{Att}$	0.265	0.239	0.337	0.247	6.559
JIT-Smart$_{Att+DLN}$	0.424	0.315	0.647	0.108	0.476
JIT-Smart	**0.552**	**0.41**	**0.74**	**0.081**	**0.098**

"↑" indicates the larger the better; "↓" indicates the smaller the better.
"Att": only utilize the attention weight of CodeBERT to locate defective-lines;
"$Att + DLN$": combines attention weights and DLN for JIT-DL;
The last row indicates JIT-Smart only uses DLN for JIT-DL.

JIT-Fine which is a single-task learning approach (commit-level supervised learning) utilizes the attention weights of each token in CodeBERT to calculate its impact in locating the defective lines. JIT-Smart leverages the information of defective code lines in combination with DLN for supervised training (a multi-task learning modeling method (commit-level and line-level supervised learning)). In JIT-DP, the main framework we use is also CodeBERT.

Therefore, we investigate if the localization approach can contribute to our approach. In this experiment, we use the optimal random seed (i.e. 0) to conduct one round of experiments for each model. Specifically, we make a comparison and the results are shown in Table 6. The results indicate that the defect localization method based on the attention mechanism can not contribute to our JIT-Smart in JIT-DL. The reason may be that JIT-Fine uses only commit-level labels for learning, and then uses the attention weights generated after the model prediction ends, which may lead to more false positives than ours, thus reducing the performance. The multi-task learning method of our JIT-Smart can effectively promote the learning of defect localization structures and enable the model to learn more fine-grained information during training.

4.3 RQ3: What is the accuracy of JIT-Smart compared to the state-of-the-art baseline in the cross-project experimental?

Motivation. In practice, in the initial stages of a software project, the amount of historical data is limited. It is difficult to train an accurate model for JIT-DP and JIT-DL based on a small amount of data. The experiments of Zimmermann et al. [52] found that the real performance of the model can be further verified in the cross-project scenario. Therefore, we investigate whether our JIT-Smart can achieve a reasonable performance under cross-project settings for JIT-DP and JIT-DL.

Approach. Since the JIT-Defects4J dataset includes 21 software projects, we use one project as the test set and the other 20 projects as the training set and validation set (divided by 8:2 ratio). We conduct 21 sets of experiments under the cross-project setting. We adopt 5 commit-level and 5 line-level evaluation metrics to evaluate the results. In this experiment, we use the optimal random seed for JIT-Fine and JIT-Smart to conduct one round of experiments. From the results of RQ1 and

RQ2, the model with the closest performance to ours is JIT-Fine, hence we only compare JIT-Smart with JIT-Fine in the cross-project evaluation.

Table 7. The performance of JIT-Smart and JIT-Fine in the cross-project JIT-DP and JIT-DL tasks

Java Project	F1-Score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑	Top-5 Acc↑	Top-10 Acc↑	R@20%E_l↑	E@20%R_l↓	IFA$_l$↓
JIT-Fine										
commons-math	0.2046	0.8902	0.7134	0.0211	0.9181	0.2825	0.2778	0.265	0.2682	26.35
commons-lang	0.3849	0.8792	0.7808	0.0154	0.9365	0.1627	0.1458	0.2583	0.3585	14.0392
commons-configuration	0.5067	0.9382	0.8	0.0188	0.9522	0.1439	0.1389	0.1392	0.3643	20.8596
commons-collections	0.2432	0.9162	0.84	0.0181	0.9529	0.1333	0.119	0.1736	0.4919	31.1111
ant-ivy	0.4916	0.8511	0.5301	0.0444	0.8142	0.1841	0.1762	0.2561	0.3085	11.9542
commons-compress	0.3859	0.8404	0.6854	0.0307	0.8976	0.1956	0.1815	0.1705	0.324	18.8833
commons-io	0.5038	0.9171	0.7671	0.0214	0.9388	0.1091	0.1136	0.2562	0.3279	23.6061
commons-net	0.3053	0.774	0.6752	0.0213	0.8853	0.2316	0.2351	0.0994	0.3726	24.7931
parquet-mr	0.4048	0.8433	0.538	0.0345	0.8407	0.1725	0.1884	0.1938	0.2644	13.5098
commons-vfs	0.4072	0.8394	0.5526	0.0315	0.8518	0.1156	0.12	0.2062	0.3358	23.8222
opennlp	0.3362	0.8036	0.7363	0.0256	0.9057	0.159	0.1237	0.2935	0.2855	16.4872
commons-digester	0.1509	0.7576	0.6842	0.0297	0.9071	0.1125	0.0875	0.1667	0.4075	14.75
commons-dbcp	0.1159	0.8021	0.7069	0.0342	0.8744	0	0	0.0227	0.4798	36.5
giraph	0.4913	0.8216	0.4969	0.0413	0.7986	0.1021	0.0908	0.2153	0.6778	29
commons-jcs	0.4348	0.8271	0.3977	0.0964	0.7629	0.1171	0.125	0.1507	0.407	32
commons-bcel	0.2472	0.79	0.5167	0.0447	0.8022	0.2727	0.2848	0.167	0.299	7
commons-codec	0.4375	0.9244	0.7222	0.0234	0.9322	0.181	0.1801	0.1919	0.4201	30.2857
commons-beanutils	0.4872	0.9313	0.7838	0.015	0.9453	0.1825	0.1642	0.1771	0.3849	11.6316
commons-validator	0.3562	0.8793	0.8611	0.0046	0.9608	0.2154	0.1908	0.2524	0.3245	28.5385
gora	0.3793	0.8508	0.7692	0.009	0.9127	0.0818	0.0867	0.2012	0.3427	28.4091
commons-scxml	0.4211	0.8557	0.5106	0.0365	0.8119	0.12	0.1236	0.2011	0.3321	37.45
JIT-Smart										
commons-math	0.4336	0.8839	0.7164	0.0204	0.9148	0.7171	0.5594	0.7354	0.0827	0
commons-lang	0.449	0.9114	0.9315	0.0061	0.9753	0.6103	0.413	0.8189	0.143	0.4182
commons-configuration	0.5617	0.9353	0.8129	0.0196	0.9362	0.5848	0.4253	0.7618	0.0626	0
commons-collections	0.3962	0.9102	0.78	0.0104	0.9388	0.5048	0.3522	0.8323	0.0587	0
ant-ivy	0.4589	0.838	0.6446	0.0299	0.887	0.5081	0.3585	0.612	0.1297	0.0092
commons-compress	0.4762	0.8659	0.7978	0.0217	0.942	0.5592	0.3851	0.7032	0.1047	0
commons-io	0.5469	0.909	0.8767	0.0177	0.9641	0.4443	0.2707	0.7803	0.0728	0.6286
commons-net	0.3761	0.8031	0.6752	0.0179	0.8728	0.5333	0.3817	0.5987	0.1261	0.2683
parquet-mr	0.4597	0.861	0.6899	0.0306	0.9097	0.4848	0.3235	0.6353	0.0877	0.1579
commons-vfs	0.4364	0.8504	0.614	0.0341	0.8681	0.4114	0.2779	0.678	0.112	5.8
opennlp	0.2973	0.8207	0.7802	0.0248	0.9187	0.4636	0.2966	0.805	0.0834	0
commons-digester	0.1944	0.7979	0.7895	0.0071	0.9412	0.7786	0.6357	0.6984	0.0769	0
commons-dbcp	0.2655	0.8471	0.7931	0.0072	0.898	0.4533	0.3224	0.7111	0.0669	0
giraph	0.5349	0.861	0.6012	0.0356	0.8862	0.508	0.3575	0.588	0.4973	3.8261
commons-jcs	0.5263	0.8511	0.625	0.038	0.8629	0.44	0.3028	0.6666	0.0772	0.74
commons-bcel	0.2759	0.7841	0.6	0.0376	0.8414	0.575	0.4542	0.5784	0.0981	0
commons-codec	0.5067	0.9312	0.7778	0.0228	0.9421	0.6632	0.503	0.7301	0.1083	0
commons-beanutils	0.4878	0.9139	0.8108	0.0198	0.9389	0.5025	0.3568	0.8212	0.1034	0.9389
commons-validator	0.3117	0.9058	0.8056	0.007	0.9542	0.4333	0.3	0.7735	0.0673	0
gora	0.4198	0.8838	0.7436	0.0205	0.9142	0.3882	0.2181	0.805	0.0368	0
commons-scxml	0.4466	0.869	0.5532	0.0375	0.8517	0.5326	0.3922	0.5409	0.1165	0.6087
AVG VALUE (JIT-Fine)	0.3665	0.8539	0.6699	0.0294	0.8858	0.156	0.1502	0.1932	0.3703	22.9038
AVG VALUE (JIT-Smart)	**0.422**	**0.8683**	**0.7342**	**0.0222**	**0.9123**	**0.5284**	**0.3756**	**0.7083**	**0.1101**	**0.6379**

Results. Table 7 shows the performance comparison of our JIT-Smart and JIT-Fine under all cross-project experimental settings. From the performance of each testing project, JIT-Smart almost outperforms JIT-Fine on each evaluation metrics. From the overall average level of JIT-DP, JIT-Smart achieves F1-Score of 0.422, AUC of 0.868, Recall@20%Effort of 0.734, Effort@20%Recall of 0.022, Popt of 0.912, which improves the best performing baseline (i.e. JIT-Fine) by 15.30%, 1.69%, 9.60%, 24.39%, 2.99%, respectively.

From the average results, JIT-Smart achieves Top-5 Accuracy of 0.528, Top-10 Accuracy of 0.376, Recall@20%Effort$_{line}$ of 0.708, Effort@20%Recall$_{line}$ of 0.11, IFA$_{line}$ of 0.638, which improves JIT-Fine by 238.72%, 150.07%, 266.61%, 70.27%, 97.21%, respectively. Comparing the results of RQ2, JIT-Smart has a greater improvement on line-level evaluation metrics than JIT-Fine under the cross-project setting. The experimental results fully demonstrate that our JIT-Smart can achieve a reasonable performance under the cross-project setting. JIT-Smart can help developers identify the bug-introducing commits and locate where the defective lines with less effort.

4.4 RQ4: How do the different loss function weight assignments affect the performance of JIT-Smart?

Motivation. Previous research only performs supervised training at the commit-level for JIT-DP and then uses the intermediate results generated by the model to locate the defective lines. Different from their work, JIT-Smart can perform JIT-DP and JIT-DL simultaneously. We design a novel DLN for JIT-DL. As introduced in Section 2.4, we design a loss function for multi-task learning of JIT-Smart. Intuitively, a larger weight given to the loss function of a task indicates that the model tends to pay more attention to a certain task and its performance. Therefore, we investigate whether different loss function weight assignments will affect the performance of JIT-Smart.

Approach. We conduct 10 sets of weight assignment experiments and assign different weights to JIT-DP loss and JIT-DL loss respectively (the sum of the weights is 1), as shown in Table 8. In this experiment, we use the optimal random seed (i.e. 0) to conduct one round of experiments for JIT-Smart. In particular, we set w_{dp}=0, w_{dl}=1 to investigate whether only using JIT-DL loss to guide the model for training is effective since the defective commit means the defective line(s) exists. We do not set w_{dp}=1, w_{dl}=0 for experimenting because there is no use of line-level labels for supervised learning, which does not serve the purpose of our original design model. Lastly, we combine the results of all evaluation metrics to determine the best weight parameters.

Table 8. The impact of different loss function weight assignments for JIT-Smart

Weight	F1-Score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑	Top-5 Acc↑	Top-10 Acc↑	R@20%E_l ↑	E@20%R_l ↓	IFA_l ↓
w_{dp}=0, w_{dl}=1	0.125	0.429	0.63	0.019	0.823	0.516	0.405	0.5	0.293	1.414
w_{dp}=0.1, w_{dl}=0.9	0.481	0.885	0.829	0.009	0.945	0.53	0.388	0.704	0.099	0.08
w_{dp}=0.2, w_{dl}=0.8	0.48	0.887	0.832	0.01	0.948	0.533	0.399	0.689	0.106	0.078
w_{dp}=0.3, w_{dl}=0.7	**0.486**	**0.885**	**0.823**	**0.01**	**0.942**	**0.552**	**0.41**	**0.74**	**0.081**	**0.098**
w_{dp}=0.4, w_{dl}=0.6	0.479	0.896	0.823	0.01	0.942	0.544	0.4	0.711	0.1	0.071
w_{dp}=0.5, w_{dl}=0.5	0.475	0.887	0.823	0.01	0.942	0.539	0.394	0.718	0.095	0.079
w_{dp}=0.6, w_{dl}=0.4	0.476	0.884	0.832	0.011	0.945	0.528	0.391	0.698	0.103	0.068
w_{dp}=0.7, w_{dl}=0.3	0.477	0.883	0.827	0.011	0.943	0.536	0.392	0.706	0.093	0.08
w_{dp}=0.8, w_{dl}=0.2	0.481	0.889	0.83	0.01	0.944	0.542	0.398	0.697	0.098	0.08
w_{dp}=0.9, w_{dl}=0.1	0.48	0.887	0.808	0.01	0.94	0.538	0.395	0.708	0.091	0.129

Results. The experimental results are shown in Table 8. w_{dp}, w_{dl} indicates the weight of JIT-DP and JIT-DL loss function, respectively. Except the experimental results in the first row, the line-level evaluation metrics are not much different, so we select the weight parameters that perform best in the commit-level evaluation metrics (i.e. w_{dp}=0.3, w_{dl}=0.7). It can be seen from the results in the first row that when we only use JIT-DL loss to guide the iterative training of the model, the model decreases in most of the metrics, and the commit-level evaluation metrics decreases greatly. From the above analysis, we can see that it is beneficial for JIT-DP and JIT-DL to be a multi-task learning process, and the two tasks will complement each other. The semantic information of code structure extracted by DLN is helpful for JIT-DP since the performance is the best when w_{dl}=0.7.

5 QUALITATIVE ANALYSIS

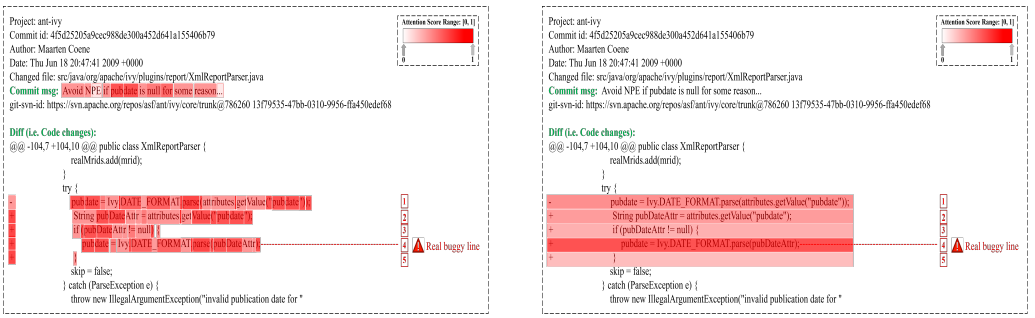
JIT-DP and JIT-DL serve different stages of code review. At the commit-level defect prediction stage, JIT-DP can help code reviewers prioritize the most likely defective commits. When the most likely defective commit is identified, in the second stage, the prediction result of JIT-DL will be able to cost-effectively locate the most likely defective code lines for code reviewers to help them make decisions. JIT-DL task is more challenging. JIT-DP will undoubtedly be more accurate. Therefore, JIT-DP and JIT-DL can function on different levels.

The design motivation for our model structure is that the predicted probability at the commit-level can be used in practice to help code reviewers prioritize commits that are more likely to be defective more efficiently. After the likely defective commit is identified, JIT-DL is used to identify

the most likely defective lines, we sort all the lines in descending order according to the defect probability of each line. Lines that are sorted at the top are more likely to be defective. Similar work to our design motivation is Yan et al.'s work [46] which proposes a two-stage framework. Different models are used for prediction and localization (i.e. logistic regression model for JIT-DP, N-gram trained on clean source code lines for JIT- DL), and their two-stage framework is trained separately (only commit-level labels are used). Different from them, we propose a novel framework JIT-Smart, which integrates two-stage label information for multi-task training. Experimental results demonstrate the advantage of modeling through this multi-task training in JIT-DP and JIT-DL tasks. Therefore we design such a framework and assume that the granularity of the feature captured by these two networks (i.e. CodeBERT and DLN) in the framework is different. Joint modeling and joint optimization are beneficial to promote their respective performances. It can also be confirmed from the experimental results that JIT-Smart outperforms the existing optimal methods. Additionally, we tried some additional model structures before (details shown in the link of our replication package).

The example in Figure 5a illustrates what the problem is with the state of the art that led to our work from the results of JIT-Smart and the current best baseline (i.e. JIT-Fine).

The following is an example of a defective commit that is successfully predicted by JIT-Fine (there is only one real buggy line in this commit). JIT-Fine locates defective lines based on the attention score generated during the defective commits identified. As shown in Figure 5a, we highlight each token according to the attention score of each token (the brighter means that JIT-Fine pays more attention to the token; the sum of the attention scores of tokens in each code line is the defect risk score of the code line, that is a higher attention score for a token in a line of code indicates that the line of code is more likely to be defective). Although the following JIT-Fine predicts that this is a defective commit, it is not so accurate and cost-effective in locating the defective code line. It can be seen that the code lines labeled 2 and 3 are judged to have a high defect risk score. Modeling methods similar to JIT-Fine are JITLine[33], TPF-JIT[46]. One of the main problems with these methods is that they use commit-level labels to model and train, and this leads to the different granularity of model attention. From the distribution results of attention, JITFine will pay attention to a relatively large range of tokens, which leads to more false positive code lines).



(a) JIT-Fine (b) JIT-Smart

Fig. 5. The prediction result of JIT-Fine and JIT-Smart

Figure 5b shows JIT-Smart predicting the same example above, successfully predicting this defective commit, so JIT-Smart further decided to use the output of the DLN for locating. The results show that JIT-Smart locates defective lines of code more accurately than JIT-Fine (fewer

false positives). Due to the particularity of JIT-DL, it needs to identify the defective line from the defective commit. That is to do a binary classification problem for each line in the defective commit, and there is a certain dependency between the lines of code. Intuitively, commit-level decisions can be used to decide whether to adopt the line-level output. Therefore, DLN organizes the input of code changes into a two-dimensional matrix and performs context-dependent learning on each code line at the token-level, and then aggregates through the token-level attention mechanism. Then DLN performs line-level context-dependent learning and finally completes the prediction through line-level attention mechanism aggregation. So our special design of DLN is more conducive to accurate and cost-effective localization of defective code lines (fewer false positives). The output of the DP network can assist in determining whether the DLN has caused a misjudgment. The granularity of the features captured by the DP network and the DLN is different. The two networks are not independent of each other but promote each other.

6 THREATS TO VALIDITY

Threats to Construct Validity relate to the evaluation metrics we adopt in experiment. In order to mitigate this threat to fairly compare with studied baselines, we adopt 5 commit-level and 5 line-level evaluation metrics for JIT-DP and JIT-DL. Similar to previous work [1, 30, 32, 40], we measure the effort of code review with LOC, but in fact given two code changes with the same number of lines of code, it may have different code complexity. Factors such as code complexity can be considered to measure the effort of code review in future work to mitigate this threat.

Threats to Internal Validity relate to potential mistakes in our reproduced studied baselines. To minimize this threat, we use the code provided in the corresponding research instead of implementing these models ourselves. We set the default optimal parameters and operating methods from the corresponding papers to ensure the accuracy of our reproduction.

Threats to External Validity relate to the generalizability of our results. We train and test our model on a large dataset (i.e. JIT-Defects4J [30]). But JIT-Defects4J only involves data in Java language, other programming languages are not considered. Future research is needed to expand the dataset including software projects written in other programming languages.

7 RELATED WORK

7.1 Just-in-Time Defect Prediction

JIT-DP which can help developers identify the bug-introducing commits has been extensively studied in recent years. Mockus et al. [29] firstly utilized the properties of the software changes including its size, duration, diffusion, type, and the experience of the developers in commits to build a logistic regression model to perform for JIT-DP. Following that, Kamei et al. [22] proposed 14 kinds of change-level features and bulit a logistic regression model for JIT-DP. Moreover, Kononenko et al. [25] conducted an empirical study and found that the change-level features extracted from the commits can contribute to JIT-DP. Based on these 14 change-level features, Yang et al. [48] leveraged deep belief network to extract a set of expressive features. Then, they used a logistic regression model for defect prediction. Subsequently, Yang et al. [47] proposed a two-layer ensemble learning approach which combines decision tree and ensemble learning to improve the performance in JIT-DP. Inspired by Yang et al.'s work [47], Young et al. [50] proposed a deep ensemble approach to generalize their approach to allow the use of any arbitrary set of classifiers in the ensemble and optimize the weights of the classifiers and allow additional layers. Meanwhile, Liu et al. [28] built an unsupervised model for the effort-aware JIT task based on the code churn metrics. Chen et al. [5] proposed a multi-objective optimization (i.e. the benefit and the cost) based method, of which the main framework is a logistic regression model for JIT-DP. Besides, Cabral et al. [3] proposed a

novel class imbalance evolution approach for the specific context of JIT-SDP. Recently, CC2vec [14] which is a code language representation framework is proposed to extract the semantic information of code changes of the commits. It is based on a hierarchical attention network for JIT-DP. DeepJIT [13] is proposed to extract the interactive semantic information of commit messages and code changes based on CNN for JIT-DP. Zeng et al. [51] conducted a comprehensive analysis on CC2vec and DeepJIT, then built a logistic regression classifier with the "added-line-number" features to perform JIT-DP. Different from their work, we propose a unified model JIT-Smart that can identify the defective commits and locate where the defective lines are.

7.2 Defect Localization

Fine-grained (i.e. line-level) prediction approaches can be more cost-effective and help developers locate defective lines with less effort. In recent years, researchers have focused on how to design methods for fine-grained defect prediction (i.e defect localization). Ray et al. [35] first applied the N-gram language model to locate defective lines for the defect localization task. Then, Yan et al. [46] proposed a two-phase framework for JIT-DP and JIT-DL. They also leveraged the N-gram language model to identify defective lines. Qiu et al. [34] also built a tool based on the N-gram language model to locate where the defective lines are. Besides, Wattanakriengkrai et al. [44] combined the logistic regression model and the LIME method for file-level and line-level post-release defect prediction. Similar to Wattanakriengkrai et al.'s work [44], Pornprasit et al. [33] combined the random forest model and the LIME method to perform JIT-DP and JIT-DL. To further improve their work, Pornprasit et al. [32] proposed to leverage HAN [49] to perform file-level defect prediction and utilize the attention mechanism to locate defective lines. More recently, Ni et al. [30] adopted CodeBERT and attention mechanism using both semantic features and expert features for JIT-DP and JIT-DL. However, they only use the label at commit-level for supervised learning. Different from their work, we propose a unified model JIT-Smart that treats JIT-DP and JIT-DL as a multi-task learning process. Specifically, we specially design a novel DLN which is part of JIT-Smart and explicitly introduces label information of defective code lines for supervised learning. Moreover, we improve the class imbalance issue from the model training level. Our JIT-Smart can perform defect prediction and defect localization simultaneously.

8 CONCLUSION

We propose a novel modeling approach for just-in-time defect prediction and localization. We design a unified model JIT-Smart, which makes the training process of JIT-DP and JIT-DL a multi-task learning process. Specifically, we design a novel defect localization network, which explicitly introduces the label information of defect code lines for supervised learning in JIT-DL. The experiment is conducted on a large-scale commit-level and line-level dataset JIT-Defects4J. We compare JIT-Smart with 7 state-of-the-art baselines in JIT-DP and JIT-DL under 5 commit-level and 5 line-level evaluation metrics. To avoid the influence of random factors, we conduct multiple rounds of experiments on all baselines and perform statistical analysis. The results demonstrate that JIT-Smart is statistically better than all the baselines, especially in JIT-DL. Statistical analysis shows that our JIT-Smart performs more stably than the best-performing model. JIT-Smart also achieves a reasonable performance in the cross-project experimental setting. Additionally, we analyze how the different loss function weight assignments affect the performance of JIT-Smart.

ACKNOWLEDGMENTS

The research is supported by National Key R&D Program of China (No. 2023YFB2703600), the National Natural Science Foundation of China (62372492), the Natural Science Foundation of Guangdong Province (2023A1515010746, 2023A1515011474).

A APPENDIX

A.1 How Does the Model Perform without any Fine-Tuning (E.g., Only with CodeBERT Embedding)?

We freeze the entire CodeBERT weight (i.e. without fine-tuning), and all indicators are much lower than JIT-Smart. We chose the best random seed for the experiment. For example, model without fine-tuning achives the F1 of 0.109, which is 345.87% lower than JIT-Smart(F1 of 0.486). The results verified the importance of our fine-tuning.

Table 9. The performance of JIT-Smart and JIT-Smart without finetuning CodeBERT

Models	F1-Score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑	Top-5 Acc↑	Top-10 Acc↑	R@20%E_l ↑	E@20%R_l ↓	IFA$_l$ ↓
$JIT-Smart_{w/o finetune-CodeBERT}$	0.109	0.733	0.642	0.02	0.831	0.49	0.334	0.649	0.08	0.621
$JIT-Smart$	0.486	0.885	0.823	0.01	0.942	0.552	0.41	0.74	0.081	0.098

"↑" indicates the larger the better; "↓" indicates the smaller the better.

A.2 Model Cost Consumption Evaluation

Information about our model size, training and test costs can be seen in the link. Compared with the best model (i.e. JITFine), the model size, training cost and testing cost of our model are not much different, but our method achieves statistically significant improvement (shown in the results of RQ1, RQ2, and RQ3 in the paper). For example, in terms of the inference time of testset(5450 samples), JIT-Smart is 40s and JITFine is 36, but JIT-Smart perfoms more stably and statistically better than JITFine, such as improving JITFine by 19.89% and 174.79% in terms of F1 and Top-10 accuracy.

Table 10. Model cost consumption evaluation

Models	Model Size	Training Time	Inference Time	GPU Graphics Card for Training
JITFine	1.39 GB	1h46mins	36s	1 NVIDIA GeForce RTX 3090
JIT-Smart	1.44 GB	2h06mins	40s	1 NVIDIA GeForce RTX 3090

A.3 Ablation Study of Focal Loss

We compared the performance of JIT-Smart and JITFine with or without focal-loss. We chose the best random seed for the experiment. Results show the advantages of JIT-Smart with focal-loss is better than JITFine under all tested conditions. For exmaple, focal-loss improved JITFine and JIT-Smart by 7.96% and 4.52% in terms of F1.

Table 11. The impact of different loss functions on the model

Models	F1-Score↑	AUC↑	R@20%E↑	E@20%R↓	Popt↑	Top-5 Acc↑	Top-10 Acc↑	R@20%E_l ↑	E@20%R_l ↓	IFA$_l$ ↓
$JITFine_{cross-entroyloss}$	0.402	0.873	0.691	0.015	0.908	0.198	0.204	0.191	0.341	12.352
$JITFine_{focal-loss}$	0.434	0.891	0.808	0.01	0.936	0.23	0.218	0.231	0.318	11.124
$JIT-Smart_{cross-entroyloss}$	0.465	0.875	0.787	0.011	0.929	0.551	0.406	0.74	0.079	0.107
$JIT-Smart_{focal-loss}$	0.486	0.885	0.823	0.01	0.942	0.552	0.41	0.74	0.081	0.098

"↑" indicates the larger the better; "↓" indicates the smaller the better.

REFERENCES

[1] 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.

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

[3] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676.

[4] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[5] 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.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[8] Andy Field. 2013. *Discovering statistics using IBM SPSS statistics*. sage.

[9] Ali Ghanbari and Andrian Marcus. 2022. Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/3533767.3534368

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[11] 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.

[12] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering* 27, 6 (2022), 1–49.

[13] 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 *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[14] 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.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[17] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[18] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Zibin Zheng, and Xiapu Luo. 2020. CommtPst: Deep learning source code for commenting positions prediction. *Journal of Systems and Software* 170 (2020), 110754.

[19] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020. Towards automatically generating block comments for code snippets. *Information and Software Technology* 127 (2020), 106373.

[20] Yuan Huang, Nan Jia, Xiangping Chen, Kai Hong, and Zibin Zheng. 2020. Code review knowledge perception: Fusing multi-features for salient-class location. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1463–1479.

[21] Yuan Huang, Jinyu Jiang, Xiapu Luo, Xiangping Chen, Zibin Zheng, Nan Jia, and Gang Huang. 2021. Change-patterns mapping: A boosting way for change impact analysis. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2376–2398.

[22] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 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*. 173–174.

[24] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar,

1746–1751. https://doi.org/10.3115/v1/D14-1181

[25] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 111–120.

[26] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.

[27] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[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 *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19.

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

[30] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1–12.

[31] Cong Pan, Minyan Lu, and Biao Xu. 2021. An empirical study on software defect prediction using codebert model. *Applied Sciences* 11, 11 (2021), 4793.

[32] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3144348

[33] 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.

[34] Fangcheng Qiu, Meng Yan, Xin Xia, Xinyu Wang, Yuanrui Fan, Ahmed E Hassan, and David Lo. 2020. JITO: a tool for just-in-time defect identification and localization. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1586–1590.

[35] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[36] Marco Tulio 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*. 1135–1144.

[37] 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*. 966–969.

[38] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[39] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[40] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1200–1219.

[41] Maciej Tomczak and Ewa Tomczak. 2014. The need to report effect size estimates revisited. An overview of some recommended measures of effect size. (2014).

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[43] Hao Wang, Weiyuan Zhuang, and Xiaofang Zhang. 2021. Software defect prediction based on gated hierarchical LSTMs. *IEEE Transactions on Reliability* 70, 2 (2021), 711–727.

[44] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* (2020).

[45] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

[46] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* (2020).

[47] 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.

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

[49] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[50] 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.

[51] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 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*. 427–438.

[52] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 91–100.

Received 2023-09-18; accepted 2024-01-23