ORIGINAL ARTICLE

A Code Change-Oriented Approach to Just-In-Time Defect Prediction with Multiple Input Semantic Fusion

Teng Huang[1] | **Hui-Qun Yu**[1] | **Gui-Sheng Fan**[1] | **Zi-Jie Huang**[1] | **Chen-Yu Wu**[1]

[1]School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China.

Correspondence
Hui-Qun Yu, School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China.
Email:yhq@ecust.edu.cn
Gui-Sheng Fan, School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China.Email:gsfan@ecust.edu.cn
Zi-Jie Huang, School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China.
Email: hzj@mail.ecust.edu.cn

Funding information
This work was partially supported by the National Natural Science Foundation of China (No.62372174), the Natural Science Foundation of Shanghai (No.21ZR1416300), the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality (No. 22010504100), the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies, and the Shanghai Municipal Special Fund for Promoting High Quality Development (No.2021-GYHLW-01007).

Recent research found that fine-tuning pre-trained models is superior to training models from scratch in just-in-time (JIT) defect prediction. However,existing approaches using pre-trained models have their limitations. First, the input length is constrained by the pre-trained models.Secondly, the inputs are change-agnostic.To address these limitations, we propose JIT-Block, a JIT defect prediction method that combines multiple input semantics using changed block as the fundamental unit. We restructure the JIT-Defects4J dataset used in previous research. We then conducted a comprehensive comparison using eleven performance metrics, including both effort-aware and effort-agnostic measures, against six state-of-the-art baseline models. The results demonstrate that on the JIT defect prediction task, our approach outperforms the baseline models in all six metrics, showing improvements ranging from 1.5% to 800% in effort-agnostic metrics and 0.3% to 57% in effort-aware metrics. For the JIT defect code line localization task, our approach outperforms the baseline models in three out of five metrics, showing improvements of 11% to 140%.

KEYWORDS
Software Defect,Just-In-Time,Defect Prediction,Deep Learning

1 | INTRODUCTION

Software defect prediction aims at locating the most defect-prone code components, in order to provide suggestion for more efficient allocation of software quality assurance resources. Recently, practitioners discovered that the cost of software maintenance resulting from defects increases over time [1, 2] and the traditional defect prediction approaches may not be responsive and effective enough to resolve bugs. Thus, JIT (Just-In-Time) defect prediction approaches are proposed to defect prediction in the granularity of code commits submitted by developers. Research indicated that the majority of defects in software systems are introduced through the submitted commits during the software development process [3]. Therefore, there is a pressing need for high-performance JIT defect prediction models to assist code reviewers in their work.

In the past, researchers typically trained deep learning models from scratch to build JIT defect prediction models, and these works can be categorized into coarse-grained [4, 5, 6, 7, 8, 9] or fine-grained [10, 11, 12, 13, 14, 15] defect prediction based on the detection granularity. Although these methods made some progress, their performance is still not promising. This could be due to the limited size of the training dataset, which prevented the models from learning sufficient knowledge for accurate JIT defect prediction [16].

Recently, Ni et al. introduced the JIT-Fine model [17], which differs from previous approaches that involve training models from scratch. JIT-Fine utilizes CodeBERT [18, 19, 20] (A pre-trained model that possesses common knowledge in both the programming and natural language domains) as a pre-trained model to extract semantic features from the changed code, then it combines semantic features with expert features defined by Kamei et al.[21] for the JIT defect prediction task(JIT-DP)(identifying defect-inducing commits) and the JIT defect localization task(JIT-DL)(identifying defect-inducing code lines in commits). Semantic features are learned from the semantic information and syntax structure of the source code. They represent the code semantics that are prone to defects. Expert features, on the other hand, are characteristics derived from the expertise and experience of professionals. JIT-Fine outperforms all previous JIT defect prediction models, with performance improvements ranging from 10% to 629% compared to baseline models. However, JIT-Fine has two limitations. First, the input length is constrained by the pre-training model (CodeBERT typically has a token limit of 512). To meet this requirement, Ni et al. [17] directly truncate the source code data. However, a considerable number of commits exceed the length limitation. For example,about 60% of commits exceed the length limitation in JIT-Defect4J dataset. As a result, the dataset is not completely learned by the model. Furthermore, the token length distribution of defect-introducing commits is similar to that of all commits. Therefore, the truncated defect-introducing commits may become training noise (cf. Section 2). Second, JIT-Fine's input is change-agnostic. This can confuse the model and diminish its ability to learn the semantic meaning of the changed code (cf. Section 2). To overcome these limitations, we propose JIT-Block, a JIT defect prediction model that combines multiple input semantics based on changed block (cf. Section 2). This model has the ability to handle longer inputs and extract the semantic meaning of changed code more effectively, resulting in improved prediction performance. Specifically, we reconstruct the JIT-Defect4J dataset used by the JIT-Fine model [17], organizing it based on changed blocks. This reconstructed dataset is referred to as JIT-Block-Defect4J in this paper. Similar to previous work, JIT-Block combines expert features with extracted semantic features to jointly learn the features for JIT defect prediction tasks. Furthermore, an attention mechanism is used to locate the lines of defective code. The main contributions of this study are as follows:

1.**JIT-Block-Defect4J dataset**. We reconstruct the JIT-Defect4J dataset proposed by Ni et al. [17] and obtain the JIT-Block-Defect4J dataset. We also introduce a novel approach to organizing JIT defect prediction datasets, using changed blocks as the fundamental unit.

2.**JIT-Block**. We propose JIT-Block, a method that can handle longer inputs and better extract semantic infor-

mation from code changes. JIT-Block outperforms the baseline on all six metrics for JIT-DP task and achieves improvement in three out of five metrics for JIT-DL task. The implementation of the JIT-Block model and the JIT-Block-Defect4J dataset are publicly available [1].

3.**Analyzing the effectiveness of using changed block**. We compare the impact of using changed block as the fundamental unit for input organization with traditional methods on model performance. The results demonstrate the advantages of our approach (i.e., higher F1 and MCC performance).

The rest of the paper is organized as follows. Section 2 presents the background and motivation of our research. Section 3 provides a detailed description of the design of JIT-Block. Section 4 discusses the experimental setup. In Section 5, we present and analyze the experimental results. Section 6 discusses potential threats to the validity of our findings. Section 7 introduces related research work. Finally, Section 8 summarizes our work and proposes future research directions.

2 | BACKGROUND AND MOTIVATION

2.1 | Traditional JIT defect prediction datasets lack relative positional information

Typically, the main content of a commit primarily consists of the modified source files. The intention of the submitter could be to develop a new feature or fix a source code bug, among other things. Fig.1 shows an example of a commit submission from the Apache-ant-ivy repository. Each code line in the commit is marked with a special symbol to indicate its status (no marking indicates no modification, "+" indicates an added code line, and a "-" sign indicates a deleted code line). From the code changes, we can clearly understand the general intention of the submitter. In this case, the submitter attempted to add new predicate expressions within two "if" condition statements. However, in traditional JIT datasets used in the field of JIT defect prediction, such as JIT-Defect4J proposed by Ni [17] and the Qt and OpenStack datasets proposed by McIntosh and Kamei [22], this information is not clear. The reason is that the relative positional information of the deleted and added code lines is lost in the datasets. We assume that this would hinders the model's ability to learn the semantic information hidden in the changed code. For example, in the JIT-Defect4J dataset, this commit is organized in the form depicted in Fig.2. In the JIT-Defect4J or Qt and OpenStack datasets, the code change information in each commit is simply divided into two parts: the added code part and the deleted code part. This simplistic division results in the loss of relative positional information between the deleted and added code lines. Consequently, it becomes challenging for the model to extract the commit author's intent from the data (i.e., semantic information). For example, when the model receives an input like "+169 +288 -169 -288 [Sep]", where the "+" denotes added code line, the "-" sign denotes deleted code line, and the numbers represent the code lines with their corresponding line numbers in the commit, the "[Sep]" is a special token used to separate source code. The model may struggle to understand the author's intent because it cannot determine whether "-169" represents a change relative to "+169" or "+288", and the same ambiguity applies to "-288".

To address this issue, we propose organizing the JIT defect prediction dataset using changed block as the fundamental unit. This is illustrated in Fig.3. Our core idea is to consider **a continuous block of added or deleted code lines as a changed block** to preserve the relative positional information between added and deleted code. Thus, we transform the input to a format like "-169+169 [Sep] -288+288". We hypothesize that this would enable the model to easily recognize that "+169" represents changes relative to "-169" and "+288" represents changes relative to "-288". Consequently, it would facilitate the model in learning the semantic information embedded within the changed code more effectively.

[1] https://github.com/hangters/JIT-Block

```
169     -           if (!mrid.isExactRevision()) {
    169 +           if (!mrid.isExactRevision() && ModuleRevisionId.isExactRevision(ivyRef.getRevision())) {
170 170                 resolvedMrid = new ModuleRevisionId(mrid.getModuleId(), ivyRef.getRevision());
171 171                 IvyNode node = data.getNode(resolvedMrid);
172 172                 if (node != null && node.getModuleRevision() != null) {
    @@ -285,7 +285,7 @@ public ResolvedModuleRevision getDependency(DependencyDescriptor dd, ResolveData
285 285         if (!resolvedMrid.isExactRevision()) {
286 286             resolvedMrid = md.getResolvedModuleRevisionId();
287 287             if (resolvedMrid.getRevision() == null || resolvedMrid.getRevision().length() == 0) {
288     -               if (ivyRef.getRevision() == null || ivyRef.getRevision().length() == 0) {
    288 +               if (ivyRef.getRevision() == null || ivyRef.getRevision().length() == 0 ||
    !ModuleRevisionId.isExactRevision(ivyRef.getRevision())) {
```

FIGURE 1 A commit from Apache-ant-ivy repository

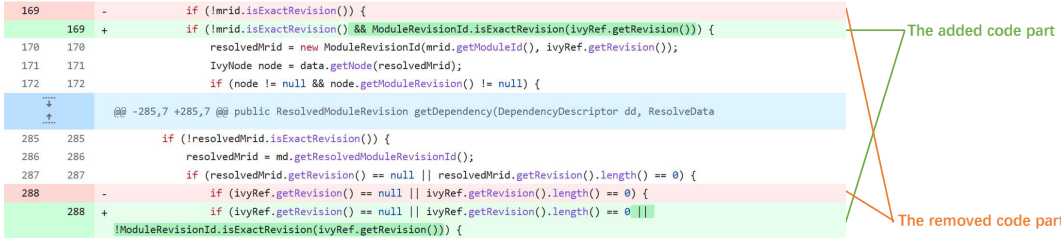

FIGURE 2 An example of how the changed code is organized in the JIT-Defect4J dataset

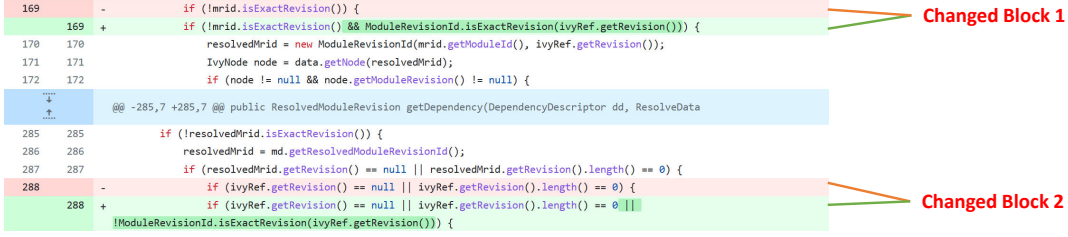

FIGURE 3 An example of how the changed code is organized using changed block as the unit

2.2 | The limited input length poses a challenge for JIT defect prediction models

Existing methods based on pre-trained models often have limitations on input length. For example, JIT-Fine [17] utilizes the pre-trained model CodeBERT [19], which often restricts its input length to 512 tokens. However, our research findings indicate that the 512-token input length cannot meet the input requirements for the majority of commit data. Taking the JIT-Defect4J dataset as an example, the distribution of the length in data samples is shown in Fig.4. (The length refers to the total number of tokens including added and deleted code lines, excluding unchanged code lines). It reveals that nearly 60% of data samples have a valid length exceeding this length limitation. We also investigate the distribution of the length in defect-introducing commits. As shown in Fig.5, we conduct a KS test to compare these two distributions (the length distribution of all commits and the length distribution of defect-introducing commits). The resulting p-value of 0.81 suggests that these two data categories follow a similar distribution. Consequently, whether a defect-introducing commit or not, directly truncating the input length of the commit to meet the 512-input length (similar to the approach used in JIT-Fine) would result in nearly 60% of the data being unused. We consider this to be an unacceptable waste, as it significantly impacts the performance of JIT defect prediction models. Furthermore, if a truncated defect-introducing commit has its corresponding defective code changes beyond 512 tokens, that positive sample (defect-introducing commits are positive samples and clean commits are negative samples) not only remains unlearned by the model but may introduce noise. This occurs because the sample is labeled as defective, yet the model does not receive the corresponding defective code change input. Moreover, since the JIT defect prediction dataset is an imbalanced dataset (cf. Section 3), where each positive sample is valuable and should be learned by the model, it becomes crucial to find a method that can handle multiple inputs to accomplish the JIT defect prediction task. We hypothesize that finding an effective approach to handle multiple inputs could improve the model's performance.

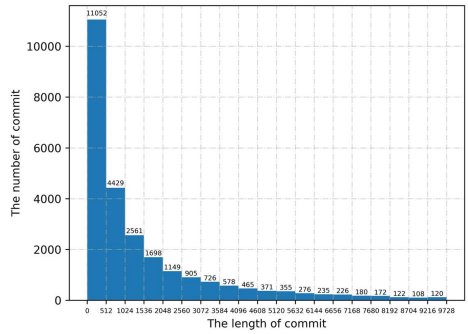

FIGURE 4 The length distribution of all commit in JIT-Defect4J dataset (with intervals of 512)

3 | JIT-BLOCK

3.1 | An Overview of JIT-Block

As shown in Fig.6, the JIT-Defect4J dataset is reconstructed, then similar to previous works, we utilize the widely used 14 change-level expert features defined by Kamei et al.[21]. We also extract semantic features of the changed

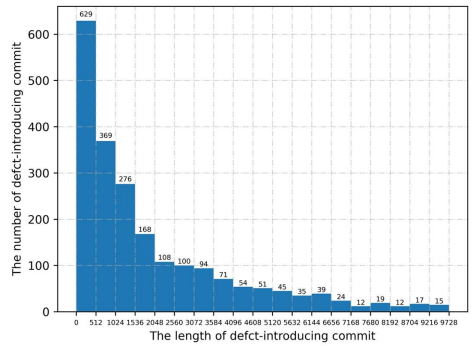

FIGURE 5 **The length distribution of defect-introducing commit in JIT-Defect4J dataset (with intervals of 512)**

code using CodeBERT. However, in contrast to previous approaches, this study incorporates multi-input semantic features related to the changes and expert features for joint learning. The joint feature learning stage mainly consists of two fully connected layers. Finally, the joint features from multiple inputs are employed to predict defect-inducing commits and locate the defective code lines. Please note that in Fig.6, the Changed Block union 1-n comes from the same commit. In other words, the JIT-Block divides a single, overly long commit into several Changed Block unions to alleviate input length limitations.The detailed information about JIT-Block will be presented in this section.

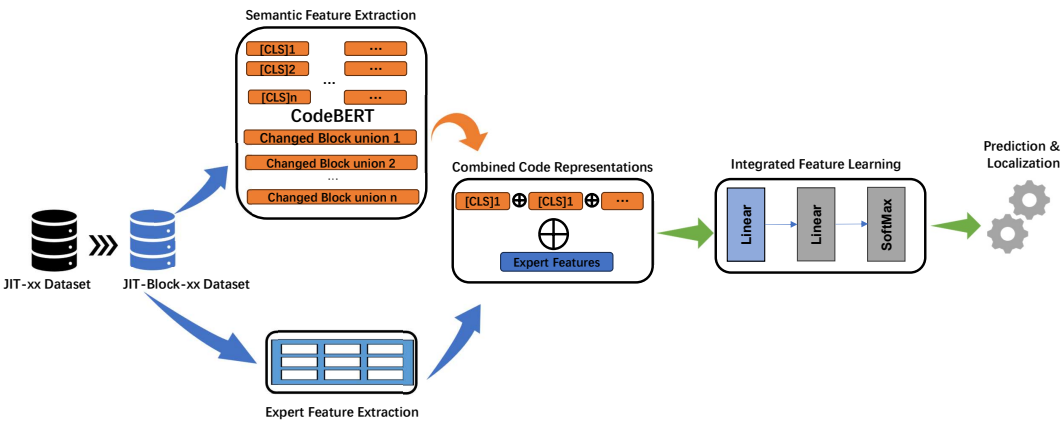

FIGURE 6 **An overview of JIT-Block**

3.2 | Refactoring JIT-Defect4J dataset

The JIT defect prediction community has been plagued by low-quality datasets due to tangled commits [23]. A single commit can contain multiple types of code changes, such as bug fixes, code refactoring, and new feature development. Due to tangled commits, the traditional methods of collecting JIT defect prediction datasets may not be accurate

enough [22]. Ni et al. [17] addressed this issue by building JIT-Defect4J rely on the LLTC4J dataset [24]. LLTC4J was manually annotated by 45 experienced participants to determine the purpose of each changed line. They considered commits that were marked by participants as contributing to bug fixing as defective commits, which greatly alleviated the impact of tangled commits on JIT defect prediction datasets. The JIT-Defect4J dataset consists of 27,319 commits from 21 large-scale open-source Java projects, of which 2,332 code commits are defective, resulting in a bug rate of approximately 8.5%.

To implement JIT-Block, the JIT-Defect4J dataset is reconstructed in the form of changed blocks to preserve relative position information. The reconstruction method is illustrated in Fig.7. The Git interface is utilized by providing the commit IDs from the JIT-Defect4J dataset to collect the original commit files. The original commit files are then read to obtain the line numbers of each added or deleted code line in the JIT-Defect4J dataset. Finally, all added code lines and deleted code lines are sorted in ascending order based on their line numbers and reorganized into changed blocks to preserve the relative positional information between added and deleted code lines. It is important to note that no additional information is added to the original dataset (JIT-Defect4J). We only change the organization of added and deleted code lines (using the changed block format). Similar to the JIT-Defect4J dataset, JIT-Block organizes all commits in chronological order. Additionally, it is worth noting that due to the removal of certain characters (such as comments or special characters) by Ni et al. [17] in a small number of code lines in JIT-Defect4J, the script is unable to match the line numbers for these code lines. Therefore, the commits that failed to match are removed, totaling approximately 0.6% of the total number of commits. Specifically, 121 out of 21,839 training data commits (approximately 0.5%) and 57 out of 5,480 testing data commits (approximately 1%) are removed. We confirm that these removed commits do not contain defect-introducing commits, as defect-introducing commits are valuable in an imbalanced dataset and should not be removed. For fair comparison, the corresponding commits are also removed from the JIT-Defect4J dataset in all baseline replication and ablation experiments.

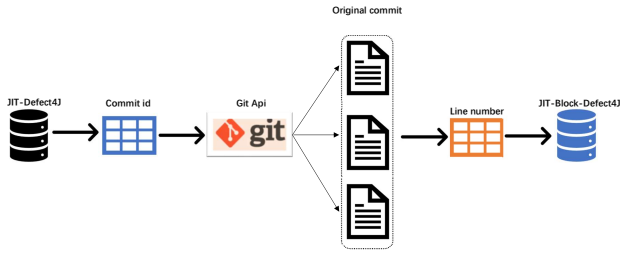

FIGURE 7 **The process of refactoring JIT-Defect4J dataset**

3.3 | The Extraction and Merging of Multi-Input Semantic Features

Recently, state-of-the-art JIT defect prediction methods commonly rely on pre-trained models for extracting semantic features from the inputs [17, 25]. Pre-trained models offer the advantage of leveraging the knowledge acquired from large-scale datasets. Consistent with prior research, this study employs CodeBERT [19], a widely adopted pre-trained model that has been extensively utilized in various Natural Language Processing (NLP) [26] and Software Engineering (SE) [20] tasks. Afterwards, the learned semantic features from each input are concatenated to form the multi-input semantic features.

The structure of the semantic feature extractor in JIT-Block is depicted in Fig.8. In JIT-Block, we refer to each input as a "changed block union." It incorporates two types of information: the “commit message” and the “changed block”. The commit message provides a description of the intentions behind the code changes made by the submitter. Please note that JIT-Block has added a commit message in each Changed Block union. The purpose is to exchange information between the commit message of a single commit and as many associated changes as possible. The changed block refers to a continuous segment of added or deleted code lines. Additionally, the special token “[CLS]” is widely used in classification tasks and is typically added at the beginning of each input [18]. Moreover, the special token “[SEP]” is used to separate different code segments. In this study, we utilize [SEP] to delimit the different changed blocks. The internal organization of each changed block is illustrated in Fig.9. The "Added line" and "Deleted line" represent the code lines that are added and deleted in the commit, respectively. The special tokens “[ADD]” and “[DEL]” are used to help the model distinguish between added and deleted code lines. After extracting the semantic features of each "changed block union" using CodeBERT, the “[CLS]" token is considered as the representative semantic feature of that changed block union. Subsequently, multiple such “changed block union” (the number of “change block union” is a tunable parameter) are fed into the CodeBERT to generate embedded vectors that represent the semantic code changes.

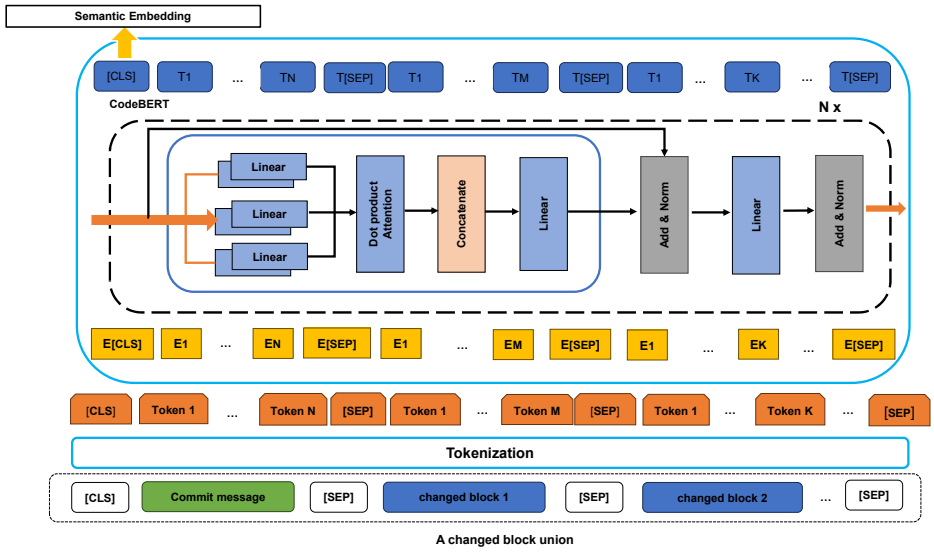

FIGURE 8 The structure of semantic feature extractor

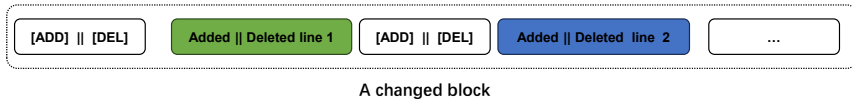

FIGURE 9 The internal organization of a changed block

3.4 | Joint Feature Learning

As shown in Fig.6, after extracting the semantic features of all changed block union, the semantic features of each changed block union are fused together. We experiment with different fusion methods, including taking the maximum, average, sum, and direct concatenation. Ultimately, direct concatenation is found to be the most effective fusion approach. To avoid the semantic features overpowering the expert features, similar to JIT-Fine, the approach proposed by Yang et al. [27] is followed. A fully connected layer is utilized to increase the dimensionality of the expert features to match the total semantic feature dimensionality. Since CodeBERT represents semantic embeddings in 768 dimensions, the total semantic features have a dimensionality of $n \times 768$ (where n is the number of changed block union), the expert features are also increased to $n \times 768$ dimensions. The total semantic feature vector is then concatenated with the expert feature vector to generate the joint features and joint feature learning is performed.

The joint feature learning stage consists of two fully connected layers. In the first fully connected layer, we reduce the dimensionality of the joint features to 768 dimensions, which are then fed into the second fully connected layer. Subsequently, the joint features are further reduced to one dimension and passed through a SoftMax function to obtain prediction scores. During the training phase, we fine-tune the aforementioned neural network.

3.5 | Defect prediction and localization

For the JIT-DP task, the joint features are used for classification prediction. For the JIT-DL task, JIT-Block leverages CodeBERT to calculate the contribution of each token in each changed block union to the semantic representation of the input, specifically, the contribution to the special token [CLS] in each change block union. In a given defect-introducing commit, the contributions are summarized for each line, and all code lines are ranked based on their total contributions (higher contribution leads to a higher rank). We hypothesize that JIT-Block's defect localization strategy is more effective than JIT-Fine because JIT-Fine only accepts single input, if the tokens associated with a defect code line are not included in the single input of 512 tokens, the overall contribution of that defect code line may decrease, potentially affecting the final defect code line localization results.

Following the previous work, we can obtain defect density for a given commit [21, 28], which can be calculated as the ratio between the probability outputted by the model (Y (c)) and the total modified lines of that commit (#LOC(c)).

4 | EXPERIMENTAL SETTINGS

According to the research by McIntosh et al.[21], the JIT defect prediction dataset exhibits temporal dependencies. Therefore, in this study, we do not perform cross-validation to avoid using test data in the training process. To ensure a fair comparison, no additional information is added to the JIT-Block-Defect4J dataset compared to the JIT-Defect4J dataset. We use the same training parameters and strategies as JIT-Fine, and follow the same data splitting strategy. Specifically, 80% of the commits from each project are used for training (60% as the training set and 20% as the validation set), while the remaining 20% are used as the test set. To provide a comprehensive evaluation of our model, in addition to using the evaluation metrics used in JIT-Fine, the Matthews correlation coefficient (MCC) is also introduced as an additional metric, since MCC is a robust measure for assessing model performance on imbalanced class datasets [29]. In this section, a brief introduction to the baseline models compared against will be provided, along with a description of the evaluation metrics used for comparison.

4.1 | Baselines

In order to accurately assess the performance of JIT-Block, we compare JIT-Block with six state-of-the-art JIT defect prediction methods from previous research:LApredict [30],Deeper [27],DeepJIT [31],CC2Vec [32], JIT-Line [33], and JIT-Fine [17].

- **LApredict** [30] uses the number of added code lines as a feature and employs a traditional logistic regression classifier for JIT defect prediction task.
- **Deeper** [27] utilizes deep belief networks to learn expert features related to JIT defect prediction.
- **DeepJIT** [31] uses a convolutional neural network to capture the semantic representation of commit messages and corresponding changed code.
- **CC2Vec** [32] employs a hierarchical attention mechanism and five comparative functions to capture the semantic representation of changed code.
- **JITLine** [33] utilizes the DE-SMOTE algorithm, to handle the class imbalance issue and uses a random forest model combined with token features and expert features for defect prediction. It also applies the Lime algorithm to locate the defective code lines.
- **JIT-Fine** [17] uses CodeBERT to extract semantic features from the changed code and combines them with expert features for defect prediction tasks and locate the defective code lines.

Unlike existing approaches, JIT-Block captures the change semantics based on changed blocks and effectively combines multiple input change semantics for JIT-DP task and JIT-DL task.

4.2 | Evaluation Measures

We evaluate the JIT-Block model using two widely used evaluation metrics: effort-aware metrics and effort-agnostic metrics.

Effort-agnostic metrics mean that software companies can examine the predictions of the defect prediction model without considering the cost. We use three widely used evaluation metrics: F1-score, AUC [34, 27], and MCC [29]. These metrics are calculated based on the results of the confusion matrix. In the confusion matrix, TP (true positive) represents the number of defect-introducing commits correctly predicted as defective, FP (false positive) represents the number of clean commits incorrectly predicted as defective, FN (false negative) represents the number of defect-introducing commits incorrectly predicted as clean, and TN (true negative) represents the number of clean commits correctly predicted as clean. We use a threshold of 0.5 to distinguish between defect-introducing commit and clean commit.

- **F1-score** is a metric that combines precision and recall to provide an overall measure of model performance. It is computed as

$$\frac{2 \times Precision \times Recall}{Precision + Recall} \tag{1}$$

- **AUC** (Area Under the Curve) is a metric used to evaluate the accuracy of a model at different thresholds. It is computed based on the Receiver Operating Characteristic (ROC) curve, which illustrates the trade-off between true positive rate and false positive rate. AUC values range from 0 to 1, with a value of 1 indicates perfect

discrimination, while a value of 0.5 indicates random guessing.

- **MCC** (Matthew's Correlation Coefficient) is a metric that comprehensively evaluates the performance of a binary classification model. It takes into account TP, FP, TN, and FN, is computed as

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{2}$$

Effort-aware metrics consider evaluating the model's performance under a certain amount of effort. Similar to previous work [33, 17], we use 20% of the total lines of code as the representative quantity for the inspection effort. For the JIT-DP task, we use the following performance metrics:

- **Recall@20%Effort (R@20%E)**. It measures the proportion between defect-introducing commits that can be discovered given 20% of the total lines of code with the total number of commits.
- **Effort@20%Recall. (E@20%R)**. It measures the amount of effort that developers need to inspect when identifying 20% of the total defect-introducing commits.
- **P_{opt}**. It compares the quality of inspection work based on the model's predicted results with the ideal inspection work. To compute this measure, we need two additional prediction models: the optimal model and the worst model. It computed as

$$1 - \frac{Area(Optimal) - Area(Our)}{Area(Optimal) - Area(Worst)} \tag{3}$$

where Area (Our) represents the area under the curve corresponding to our model. The optimal model and our model rank the commits in descending order of defect density, while the worst model ranks the commits in ascending order of defect density. For a given commit, the defect density is calculated as

$$\frac{Y(m)}{\#LOC(c)} \tag{4}$$

The value of P_{opt} ranges between 0 and 1, with a value closer to 1 indicating that the inspection work based on the model's predicted results is closer to the ideal scenario.

For the JIT-DL task, we use the following performance metrics:

- **Top-N Accuracy**. It measures the proportion of actual defect code lines included in the top N predictions made by the model (similar to JIT-Fine, we use N=5, 10).
- **Recall@20%Effort$_{line}$ (R@20%E$_{line}$)**. It measures the proportion of defect code lines that can be detected when reviewing 20% of the total code lines in a given defect-introducing commit. A higher Recall@20%Effort$_{line}$ indicates that code reviewers can detect more defect code lines with the same amount of effort.
- **Effort@20%Recall$_{line}$ (E@20%R$_{line}$)**. It quantifies the proportion of code lines that code reviewers need to review in order to detect 20% of the defect code lines in a given defect commit. A lower E@20%R$_{line}$ implies that code reviewers require less effort to identify 20% of the total defect code lines.
- **Initial False Alarm (IFA)**. It calculates the number of clean code lines that code reviewers need to inspect before finding the first defect code line. A lower IFA value indicates that code reviewers spend less effort before identifying the first defect code line.

5 | EXPERIMENTAL RESULTS

To evaluate the performance of the JIT-Block model and investigate the effectiveness of input using changed block as unit, we address the following three questions.

1. For the JIT-DP task, how does the performance of JIT-Block compare to the six baseline models?
2. For the JIT-DL task, how does the performance of JIT-Block compare to the two existing state-of-the-art JIT-DL methods?
3. Is input that using changed block as unit more effective compared to the traditional form of input?

By answering these questions, we aim to evaluate the effectiveness of the JIT-Block model in JIT-DP task and JIT-DL task and assess the advantages of using changed block unit as input representation.In the next three sets of experiments, we will consistently set the epoch to 50, the training batch size to 12, and the evaluation batch size to 128. For detailed information on other parameters, please refer to the provided source code.

5.1 | JIT-DP Task

Objective. Previous works have suffered from the issues of underutilized datasets or inputs that are unrelated to code changes. To the best of our knowledge, JIT-Block is the first JIT defect prediction method that combines multiple input semantics specifically designed for code changes. Therefore, in this experiment, we investigate the performance of JIT-Block compared to six baselines in the JIT-DP task.

Experimental Design. For the JIT-DP task, we compare JIT-Block with six state-of-the-art baseline models: LApredict [30], Deeper [27], DeepJIT [31], CC2Vec [32], JITLine [33], and JITFine [17], using six evaluation metrics including effort-agnostic metrics (F1, AUC, MCC) and effort-aware metrics (R@20%E, E@20%R, P_{opt}). All replication experiments are conducted using the publicly available source code provided by the authors of the baselines, and the experimental parameters are set according to the recommendations of the baseline authors. Additionally, due to the multi-input capability of JIT-Block, we investigate the model performance of JIT-Block with varying numbers of inputs (ranging from 1 to 6).

Results. The results are presented in Table.1, where the optimal values for each metric are highlighted in bold. For the JIT-DP task, JIT-Block model outperforms the six baseline models across all metrics, including both effort-agnostic and effort-aware measures. Specifically, in terms of effort-agnostic metrics, JIT-Block achieves an F1 score of 0.531, showing a substantial improvement of 37%-800% compared to the baselines. The MCC value of 0.494 represents a notable improvement of 45%-384%, while the AUC score of 0.883 demonstrates an improvement of 1.5%-27%. Regarding effort-aware metrics, JIT-Block achieves an R@20%E of 0.76, outperforming the baselines by 4%-21%. The E@20%R value of 0.009 indicates a considerable improvement of 30%-57%, while the P_{opt} value of 0.914 demonstrates an enhancement of 0.3%-12%. Importantly, it is worth highlighting the improvements relative to JIT-Fine, as both JIT-Block and JIT-Fine utilize Code-BERT for semantic extraction and incorporate expert features. However, JIT-Block's advantage lies in its ability to accept multiple inputs and its unique organization using changed blocks as unit. Notably, JIT-Block achieves improvements of 37%, 45%, 1.5%, 4%, 30%, and 0.3% over JIT-Fine in the six JIT-DP metrics (F1, MCC, AUC, R@20%E, E@20%R, P_{opt}), respectively. These findings validate our hypothesis that increasing the number of inputs and effectively merging them through joint learning can enhance dataset utilization and improve model performance.

We investigate how the performance of the JIT-Block model in the JIT-DP task changes with varying input num-

TABLE 1 Defect prediction results of JIT-Block compared against six baselines

Method	F1	AUC	MCC	R@20%E	E@20%R	P_{opt}
LApredict	0.059	0.694	0.102	0.625	0.02	0.814
Deeper	0.246	0.682	0.159	0.638	0.021	0.827
DeepJIT	0.295	0.775	0.253	0.676	0.014	0.859
CC2Vec	0.248	0.791	0.185	0.676	0.014	0.861
JITLine	0.261	0.802	0.303	0.705	0.015	0.883
JIT-Fine	0.388	0.87	0.34	0.731	0.013	0.911
JIT-Block$_5$	**0.531**	**0.883**	**0.494**	**0.76**	**0.009**	**0.914**

R@20%E: Recall@20%Effort; **E@20%R**: Effort@20%Recall. The subscript number in JIT-Block corresponds to the number of inputs used. JIT-Block$_5$ indicates that it uses 5 inputs and all subsequent tables follow the same rule.

bers. Table.2 presents the performance of the JIT-Block model with input numbers ranging from 1 to 6 and the best performances are highlighted in bold. It can be observed that as the input number increases from 1 to 5, both F1 and MCC gradually increase. The peak values of F1 (0.531) and MCC (0.494) are achieved when the input number is 5. Furthermore, as the input number increases from 1 to 4, all six metrics show a gradual improvement. This indicates that JIT-Block exhibits good scalability and can adapt well to variations in JIT defect prediction datasets. It can adapt to different input lengths to achieve optimal performance for a given dataset. Additionally, it can be anticipated that, under effective combination, the model's performance will improve with increasing input length up to a certain threshold, reaching its peak at a specific input length. However, once the threshold is exceeded (in this experiment, the threshold is 5), the model's performance tends to decrease with further increase in input length. This decrease in performance may be attributed to overfitting, which weakens the model's generalization ability.

TABLE 2 The performance of JIT-Block on JIT-DP using different numbers of inputs

	F1	AUC	MCC	R@20%E	E@20%R	P_{opt}
JIT-Block$_1$	0.401	0.846	0.34	0.758	0.01	0.902
JIT-Block$_2$	0.45	0.864	0.394	0.779	0.009	0.915
JIT-Block$_3$	0.481	0.871	0.43	0.794	**0.009**	0.926
JIT-Block$_4$	0.503	**0.884**	0.456	**0.811**	**0.009**	**0.928**
JIT-Block$_5$	**0.531**	0.883	**0.494**	0.76	**0.009**	0.914
JIT-Block$_6$	0.473	0.881	0.43	0.796	0.012	0.923

Conclusion. For the JIT-DP task, JIT-Block not only outperforms all six state-of-the-art baseline models in performance but also demonstrates excellent scalability.

5.2 | JIT-DL Task

Objective. Fine-grained defect localization can help code reviewers save more costs. JIT-Line [33] utilizes token features and Lime algorithm for defect code line localization, while JIT-Fine [17] uses attention scores provided by CodeBERT for defect code line localization. Similarly to JIT-Fine, JIT-Block also utilizes attention scores provided by CodeBERT for JIT-DP task. However, the difference lies in the broader defect localization perspective of JIT-Block, as it can accept multiple inputs. This allows JIT-Block to provide attention scores for a larger number of tokens. Therefore, in this experiment, we investigate the performance of JIT-Block compared to two baselines in the JIT-DP task.

Experimental Design. For the JIT-DL task, we select two baseline models, JITLine [33] and JIT-Fine [17], which have shown outstanding performance on the JIT-DP task. We reproduce the baseline experiments using the source code provided by the authors and the experiment parameters recommended by the authors. Additionally, we also investigate the performance of JIT-Block on the JIT-DL task with varying input numbers (from 1 to 6).

Results. The experimental results are shown in Table.3 and the best performances are highlighted in bold. JIT-Block achieves a TOP-10-ACC value of 0.236, which represents an improvement of 14%-140% compared to the baselines. The TOP-5-ACC value of JIT-Block is 0.235, showing an improvement of 11%-125% compared to the baselines. The IFA value of JIT-Block is 10.412, indicating an improvement of 22%-56% compared to the baselines. JIT-Block achieves the best performance in these three metrics. However, JIT-Block perform lower than the top-performing baseline models in terms of Recall@20%Effortline and Effort@20%Locline. It shows a decrease of 3% and 14% relative to JIT-Fine, respectively. This suggests that as the input length increases, the model has a broader view of JIT-DL localization, however, on average, locating the defective code lines becomes more challenging. Nevertheless, JIT-Block achieved the best results in three out of the five metrics in the JIT-DL task (TOP-10-ACC, TOP-5-ACC, IFA). This indicates that the approach of combining multiple inputs has great potential in the JIT-DP task.

TABLE 3 **Defect localization results of JIT-Block compared against two baselines**

Method	TOP-10-ACC	TOP-5-ACC	R@20%E$_{line}$	E@20%R$_{line}$	IFA
JITLine	0.098	0.104	0.157	0.33	24.208
JIT-Fine	0.206	0.21	**0.196**	**0.31**	13.488
JIT-Block$_5$	**0.235**	**0.235**	0.19	0.356	**10.412**

TOP-10-ACC: TOP-10 Accuracy; **TOP-5-ACC**: TOP-5 Accuracy; **R@20%E**$_{line}$: Recall@20%Effort$_{line}$; **E@20%R**$_{line}$: Effort@20%Recall$_{line}$.

We investigate how the performance of JIT-Block in the JIT-DL task varies with the number of inputs (ranging from 1 to 6). The specific results are shown in Table.4, with the best results highlighted in bold. It can be observed that JIT-Block consistently outperforms JIT-Fine in the TOP-10-ACC, TOP-5-ACC, and IFA metrics. Although there are some fluctuations, JIT-Block shows an increasing trend in the TOP-N-ACC (5 or 10) metrics as the number of inputs increases. The Recall@20%Effortline metric slightly decreases with the increase in the number of inputs, and overall, it is slightly lower than JIT-Fine. The IFA metric shows a decreasing trend with the increase in the number of inputs, but it remains better than JIT-Fine. The Effort@20%Loc metric also shows a decreasing trend overall, and it is consistently lower than JIT-Fine, which once again highlight the challenge of precisely locating the defective code lines as the input length increases.

Conclusion. For the JIT-DL task, JIT-Block simultaneously exhibits its advantages (better TOP-10-ACC, TOP-5-ACC,

TABLE 4 The performance of JIT-Block on JIT-DP using different numbers of inputs

	TOP-10-ACC	TOP-5-ACC	R@20%E$_{line}$	E@20%R$_{line}$	IFA
JIT-Block$_1$	0.228	0.235	**0.210**	**0.330**	**8.469**
JIT-Block$_2$	0.228	0.233	0.187	0.779	0.350
JIT-Block$_3$	0.236	0.231	0.194	0.794	0.345
JIT-Block$_4$	0.211	0.214	0.196	0.811	0.352
JIT-Block$_5$	0.235	0.235	0.190	0.76	0.356
JIT-Block$_6$	**0.240**	**0.253**	0.187	0.363	12.873

and IFA) and limitations (lower Recall@20%E$_{line}$ and Effort@20%R$_{line}$).

5.3 | The effectiveness of using changed block

Objective. JIT-Fine [17] utilizes semantic information from inputs that lack relative positional information for JIT defect prediction task. In contrast, JIT-Block organizes inputs in the form of changed blocks, preserving relative positional information. JIT-Block outperforms the baseline model significantly in the JIT-DP task and demonstrates certain advantages in the JIT-DL task (achieving the best results in terms of TOP-10-ACC, TOP-5-ACC, and IFA). In this experiment, we aim to investigate how the performance of the JIT-Block model would be affected by directly increasing the quantity of inputs that lack relative positional information in the JIT-DP task, without organizing them as changed blocks.

Experimental Design. We directly increase the quantity of traditional inputs (inputs lacking relative positional information). To ensure a fair comparison. We compare the results using three metrics: F1, AUC, and MCC, considering the same number of inputs in both cases (5 inputs).

Results. The experimental results are shown in Table.5 and the best performances are highlighted in bold. It can be observed that the inputs organized in the form of changed blocks achieve the best results in terms of F1 and MCC, with improvements of 6% and 10% respectively, compared to using traditional inputs lacking relative positional information. However, the AUC values are slightly lower by 0.3% compared to the traditional input form. Considering these three metrics, we conclude that organizing inputs in the form of changed blocks allows the model to correctly differentiate more positive and negative examples (as indicated by higher F1 and MCC values). However, compared to the regular input form, it assigns slightly lower scores to correctly predicted positive examples, and higher scores to correctly predicted negative examples (which is why the AUC value is slightly lower). One possible reason for this observation is that organizing inputs as changed block, although facilitating the learning of semantic changes (as it retains relative positional information for added and deleted code), increases the difficulty for the model to distinguish code changes. For example, in "changed block 2" in Fig.3, the deleted code line and the added code line that follow it form a changed block. However, the similarity between the deleted code line and the added code line is high (as the added code is based on modifications to the deleted code). This limited variation within many changed blocks makes it challenging for the model to differentiate change samples effectively, resulting in slightly lower scores for correctly predicted positive examples, and higher scores for correctly predicted negative examples. Conversely, this issue does not exist in the traditional input form, leading to a higher AUC value for JIT-Block*. However, considering the improvements in F1 and MCC values with the use of changed block, we believe that organizing inputs in the form

of changed blocks still holds potential.

TABLE 5 Experimental Comparison of JIT-Block using changed block input organization and traditional input

	F1	AUC	MCC
JIT-Block$_5$*	0.501	**0.886**	0.451
JIT-Block$_5$	**0.531**	0.883	**0.494**

JIT-Block* represents the JIT-Block model using the traditional input.

Conclusion. The inputs that retain relative position information (i.e., organized in changed block format) compared to traditional inputs allows the model to learn change semantics more easily, leading to more accurate prediction results (higher F1, MCC). However, it slightly increases the difficulty in distinguishing changes (lower AUC). The experiments indicate that relative position information is a valuable feature to consider for the JIT-DP task.

6 | THREATS TO VALIDITY

Internal Threats. In order to minimize errors while replicating the baselines, the source code shared by the authors on GitHub is directly used along with their recommended hyperparameter settings. Additionally, as JIT-Defect4J [17] is refactored to create JIT-Block-Defect4J, approximately 0.6% of the commits are removed (as explained in Section 3). To mitigate the impact of this action, the same commits are also removed from JIT-Defect4J while replicating the baseline experiments. Moreover, to maintain the number of defect-introducing commits in the dataset, it is manually confirmed that the removed commits do not include any defect-introducing commits.

External Threats. The JIT-Block-Defect4J dataset is a refactored version of JIT-Defect4J [17], and it shares the same limitations and biases of the original dataset. Specifically, the projects included in both datasets are primarily written in the Java programming language, and other programming languages such as C/C++ and Python are not considered. Another issue that needs to be discussed is that JIT-Block may require retraining to predict commits for a new software project. This could potentially make JIT-Block less practical. However, JIT-Block relies on a pre-trained code representation model to accomplish JIT defect prediction tasks. This allows it to leverage common knowledge learned by the pre-trained model. When predicting new changes in a new software project, we can extract expert features based on existing automated methods. Subsequently, we can fine-tune the model using a small amount of existing commits, reducing the required training data and training time. Additionally, in the scenario of predicting changes in a completely new software project without any available historical commit data, we can use the commit data from previously developed projects by the same software development team to train the model. Then, we can use this trained model to predict changes in the new project. Furthermore, if none of the mentioned information is available, we can attempt a zero-shot approach, as erroneous changes might share certain similarities. We believe that these methods can alleviate the problem to some extent.

7 | RELATED WORKS

In the past, researchers typically used a training from scratch approach to train JIT defect prediction models. Kamei et al. [21] proposed 14 change-level features that have been widely used in subsequent works. Relying on Kamei's

work, Yang et al. [27] first constructed the Deeper model using deep belief networks to learn change-level features. In subsequent works, they achieved promising results by using ensemble learning to learn change-level features [35]. In particular, unlike using supervised learning methods, Liu et al. proposed an unsupervised learning approach to accomplish JIT defect prediction tasks. Later, Chen et al. [36] treated the JIT defect prediction task as a multi-objective optimization problem and used evolutionary algorithms to select a set of effective features. Cabral et al. [37] provided a novel sampling method to address the issues of validation delay and class imbalance in JIT defect prediction. Following this, Hoang et al. [31] attempted to use modern advanced deep learning models for JIT defect prediction and proposed DeepJIT. DeepJIT utilized convolutional neural networks to automatically extract features from commit messages and changed code. Furthermore, in their subsequent work, Hoang et al. [32] introduced CC2Vec, which leverages hierarchical attention mechanisms and designed comparison functions to automatically extract high-level features from change code and commit messages for JIT defect prediction. They also demonstrated that CC2Vec can be combined with DeepJIT (i.e., using CC2Vec to learn the features extracted by DeepJIT), achieving state-of-the-art results compared to previous works. However, Pornprasit et al. [33] pointed out an error in the JIT defect prediction experiments with CC2Vec, where the test set was misused in the training set. They proposed JITLine, which uses random forests to learn expert features and combines token features for defect prediction, followed by using the Lime algorithm for defect localization.

Recently, fine-tuning existing pre-trained models has achieved state-of-the-art performance in both JIT-DP and JIT-DL tasks. Ni et al. [17] introduced JIT-Fine, which utilizes CodeBERT to extract semantic features from changed code combined with the 14 expert features proposed by Kamei et al. [21] for JIT defect prediction and employs attention mechanisms for defect localization.

In contrast to previous works, JIT-Block is the first JIT defect prediction method that combines multiple input semantics specifically designed for code changes. Our work emphasizes the importance of input quantity and retaining the relative positional information of added and deleted code lines in the inputs.

8 | CONCLUSION AND FUTURE WORK

We propose a novel JIT defect prediction method called JIT-Block, which is based on the combination of multiple input semantics using changed block as unit. JIT-Block replaces the traditional inputs that lack relative position information with inputs that retain the positional information. It then leverages the combined semantics to perform JIT defect prediction tasks (JIT-DP) and JIT defect localization tasks (JIT-DL). To validate the effectiveness of JIT-Block, we refactor the JIT-Defect4J dataset to create the JIT-Block-Defect4J dataset. In our experiments, we compare JIT-Block against six baseline methods using 11 performance metrics. The results demonstrate that JIT-Block outperforms all baseline methods comprehensively in JIT-DP task, achieving better performance in three metrics compared to previous methods. In our ablation experiments, the advantages of retaining relative positional information in the inputs compared to traditional inputs are showcased.

Our future work involves exploring the possibility of more effective methods for combining multiple input semantics to further enhance JIT-Block's performance. We will also address the limitations of JIT-Block in JIT-DL task and investigate ways to overcome them. Additionally, we plan to evaluate our method using datasets from other programming languages to further validate its applicability and generalization.

CONFLICT OF INTEREST

All authors declare that they have no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available at https://github.com/hangters/JIT-Block. The raw data are available at https://github.com/jacknichao/JIT-Fine.

ORCID

Teng Huang https://orcid.org/0009-0002-3909-6778

REFERENCES

[1] Britton T, Jeng L, Carver G, Cheak P, Katzenellenbogen T. Reversible debugging software-quantify the time and cost saved using reversible debuggers. University of Cambridge 2013;.

[2] Kim S, Whitehead EJ, Zhang Y. Classifying software changes: Clean or buggy? IEEE Transactions on software engineering 2008;34(2):181–196.

[3] Wen M, Wu R, Liu Y, Tian Y, Xie X, Cheung SC, et al. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2019. p. 326–337.

[4] Chen X, Zhang D, Zhao Y, Cui Z, Ni C. Software defect number prediction: Unsupervised vs supervised methods. Information and Software Technology 2019;106:161–181.

[5] Menzies T, Butcher A, Marcus A, Zimmermann T, Cok D. Local vs. global models for effort estimation and defect prediction. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011) IEEE; 2011. p. 343–351.

[6] Nam J, Pan SJ, Kim S. Transfer defect learning. In: 2013 35th international conference on software engineering (ICSE) IEEE; 2013. p. 382–391.

[7] Turhan B, Menzies T, Bener AB, Di Stefano J. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 2009;14:540–578.

[8] Wu F, Jing XY, Dong X, Cao J, Xu M, Zhang H, et al. Cross-project and within-project semi-supervised software defect prediction problems study using a unified solution. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C) IEEE; 2017. p. 195–197.

[9] Zhou Y, Yang Y, Lu H, Chen L, Li Y, Zhao Y, et al. How far we have progressed in the journey? an examination of cross-project defect prediction. ACM Transactions on Software Engineering and Methodology (TOSEM) 2018;27(1):1–51.

[10] Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering 2016;43(7):641–657.

[11] Fu W, Menzies T. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering; 2017. p. 72–83.

[12] Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N. An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th working conference on mining software repositories; 2014. p. 172–181.

[13] Huang Q, Xia X, Lo D. 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; 2017. p. 159–170.

[14] Pascarella L, Palomba F, Bacchelli A. Fine-grained just-in-time defect prediction. Journal of Systems and Software 2019;150:22–36.

[15] Young S, Abdou T, Bener A. 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; 2018. p. 42–47.

[16] Lin B, Wang S, Liu Z, Xia X, Mao X. Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach. IEEE Transactions on Software Engineering 2022;49(4):1640–1660.

[17] Ni C, Wang W, Yang K, Xia X, Liu K, Lo D. 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; 2022. p. 672–683.

[18] Devlin J, Chang MW, Lee K, Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:181004805 2018;.

[19] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:200208155 2020;.

[20] Gao Z, Xia X, Lo D, Grundy J, Zimmermann T. Automating the removal of obsolete TODO comments. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2021. p. 218–229.

[21] Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, et al. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 2012;39(6):757–773.

[22] McIntosh S, Kamei Y. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In: Proceedings of the 40th International Conference on Software Engineering; 2018. p. 560–560.

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

[24] Herbold S, Trautsch A, Ledel B, Aghamohammadi A, Ghaleb TA, Chahal KK, et al. A fine-grained data set and analysis of tangling in bug fixing commits. Empirical Software Engineering 2022;27(6):125.

[25] Lin B, Wang S, Liu Z, Liu Y, Xia X, Mao X. CCT5: A Code-Change-Oriented Pre-Trained Model. arXiv preprint arXiv:230510785 2023;.

[26] Qiu X, Sun T, Xu Y, Shao Y, Dai N, Huang X. Pre-trained models for natural language processing: A survey. Science China Technological Sciences 2020;63(10):1872–1897.

[27] Yang X, Lo D, Xia X, Zhang Y, Sun J. Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security IEEE; 2015. p. 17–26.

[28] Mende T, Koschke R. Effort-aware defect prediction models. In: 2010 14th European Conference on Software Maintenance and Reengineering IEEE; 2010. p. 107–116.

[29] Chicco D, Jurman G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC genomics 2020;21(1):1–13.

[30] Zeng Z, Zhang Y, Zhang H, Zhang L. Deep just-in-time defect prediction: how far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2021. p. 427–438.

[31] Hoang T, Dam HK, Kamei Y, Lo D, Ubayashi N. 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; 2019. p. 34–45.

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

[33] Pornprasit C, Tantithamthavorn CK. 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; 2021. p. 369–379.

[34] Ni C, Xia X, Lo D, Chen X, Gu Q. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. IEEE Transactions on Software Engineering 2020;48(3):786–802.

[35] Yang X, Lo D, Xia X, Sun J. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology 2017;87:206–220.

[36] Chen X, Zhao Y, Wang Q, Yuan Z. MULTI: Multi-objective effort-aware just-in-time software defect prediction. Information and Software Technology 2018;93:1–13.

[37] Cabral GG, Minku LL, Shihab E, Mujahid S. 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; 2019. p. 666–676.