JIT-Coka: An Improved Framework for Just-in-Time Defect Prediction and Localization Using Fused Features of Code Change

Yuguo Liang[1][0009−0002−8738−2891], Chengcheng Wu[1], Wentao Chen[1], Guisheng Fan[1,2](✉), and Huiqun Yu[1,2](✉)

[1] Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China

[2] Shanghai Engineering Research Center of Smart Energy, Shanghai 201103, China

✉{gsfan,yhq}@ecust.edu.cn

Abstract. Just-in-Time Defect Prediction and Localization (JIT-DP and DL) play a crucial role in software quality assurance by identifying defective code changes and locating faulty lines at the time of code submission. While existing methods leverage either handcrafted expert features or semantic features extracted by deep learning models, few explicitly distinguish or effectively fuse these two types of information. In this paper, we propose JIT-Coka, an improved framework for JIT-DP and DL tasks that combines an encoder-decoder based pre-trained model of code (CodeT5) with KANLinear, which is a spline-based adaptive nonlinear classifier used to model fine-grained nonlinear relationships between semantic and expert features. We conduct comprehensive experiments on the high-quality JIT-Defects4J dataset, evaluating JIT-Coka and representative baselines using multiple metrics. Results show that JIT-Coka significantly outperforms state-of-the-art models in defect prediction, improving F1 and MCC by 6.8% and 8.5% respectively compared to JIT-Smart, while maintaining competitive performance in DL.

Keywords: Software Defect Prediction · Defect Localization · Just-in-Time · Software Quality Assurance · Pre-Trained Models

1 Introduction

Software defects, also known as bugs or faults [7], can manifest in various forms such as API misuse, coding errors, style violations, and security vulnerabilities [28]. Traditional defect prediction (DP) methods have primarily focused on identifying defective components at a coarse granularity, such as files or packages. However, these approaches face significant challenges in terms of identifying responsible personnel and delaying defect detection [12]. To address these issues, research has gradually shifted toward finer-grained Just-in-Time Defect Prediction and Localization (JIT-DP and DL), which aims to automatically detect buggy code changes and further localize the defective lines, thereby reducing the debugging and testing burden on software developers and testers. Consequently, JIT-DP and DL have become critical research directions in the field of software engineering [37, 24].

JIT-DP and DL have evolved from traditional machine learning paradigms to deep learning approaches, with the recent application of pre-trained models achieving promising results in the field. Earlier models specifically designed for JIT-DP and DL were based on traditional machine learning methods [19, 13, 12, 34, 37]. These models typically relied on carefully crafted change-level metrics, such as code churn, change diffusion, historical data, and author experience, also referred to as expert features [21], to identify defective commits, and further applied techniques like n-gram or bag-of-words to localize defective lines [33, 24]. Recent studies [38, 21, 3] have begun to introduce pre-trained models of code (CodePTMs), trained on large-scale multilingual code corpora, as the backbone to learn semantic representations (also called semantic features [21]) from code changes. These models often integrate expert features into the training process for JIT-DP and DL, achieving promising results on benchmark datasets.

However, current JIT-DP and DL research still suffers from several limitations. On the one hand, existing models [21, 3] that combine expert features with semantic features do not implement mechanisms to explicitly distinguish these two types of features; instead, they simply concatenate the expanded expert and semantic features before feeding them into the final classifier. Furthermore, although encoder-decoder architecture-based pre-trained models of code (CodePTMs) have demonstrated strong performance across multiple software engineering tasks [22, 23, 36], existing JIT-DP and DL methods are still built upon encoder-only CodePTMs (e.g., CodeBERT [5]), which are pre-trained on a limited range of programming languages and tasks. This can restrict their practical performance and limits their extensibility to generative tasks. For example, when building a multi-task model capable of both JIT-DP and commit message generation [16], additional decoders (e.g., GPT-2 [25]) must be attached to encoder-only CodePTMs, whereas encoder-decoder architectures naturally support such tasks.

On the other hand, most JIT-DP methods tend to evaluate models using only a limited subset of common classification metrics, such as threshold-independent AUC-ROC and threshold-dependent F1-score. Recent research [20] has shown that scores of these metrics convey limited information when applied to imbalanced DP datasets. For instance, AUC-ROC score often yields overly optimistic evaluations, while in practice a specific classification threshold (e.g., 0.5) must be applied. Meanwhile, an F1 score alone cannot reveal whether a model is better at precision or recall. Besides, more comprehensive metrics such as Matthews Correlation Coefficient (MCC), which captures the full picture of the confusion matrix, should also be reported for a more balanced evaluation [35].

To bridge these gaps, we propose JIT-Coka, a JIT-DP and DL model built upon **Co**deT5 and **KA**NLinear, which is a spline-based adaptive nonlinear classifier inspired by the Kolmogorov–Arnold representation theorem [17]. We conduct a comprehensive evaluation on the JIT-Defects4J dataset, using multiple evaluation metrics to compare JIT-Coka against existing dedicated models. Our key contributions are summarized as follows:

- We perform comprehensive experiments on representative JIT-DP and DL models using the high quality JIT-Defects4J dataset and evaluate them with multiple metrics to fill the evaluation gap in prior studies.
- We develop JIT-Coka, a model applicable to both JIT-DP and DL tasks. On the DP task, JIT-Coka achieves significantly better performance than the current state-of-the-art model (JIT-Smart) in terms of Precision and MCC, while maintaining comparable performance in DL.
- We conduct a comprehensive ablation study to validate the effectiveness of each component of JIT-Coka. Moreover, the implementation and trained models are made publicly available [3] to facilitate future research.

2 Related Work

2.1 Pre-Trained Models of Code

Pre-trained models (PTMs) have shown strong generalization in NLP and inspired similar approaches in software engineering. CodePTMs aim to learn transferable code representations and are typically based on the Transformer architecture [30]. According to their backbone structure, they can be categorized as encoder-only (e.g., CodeBERT [5], GraphCodeBERT [6]), decoder-only (e.g., CodeGPT [18]), or encoder-decoder (e.g., PLBART [1], CodeT5 [32]). These models are typically pre-trained in an unsupervised manner on large-scale code corpora, using tasks like Masked Language Modeling or Denoising Autoencoding to learn general code representations. They can then be fine-tuned with a small amount of labeled data to adapt to specific downstream software engineering tasks such as automatic code summarization [15].

It is worth noting that CodeBERT, as one of the early widely used CodePTMs, was pre-trained on the CodeSearchNet corpus, which includes six programming languages: Python, Java, JavaScript, Ruby, Go, and PHP. In contrast, CodeT5 further expanded its pre-training data by including C and C# code, thereby enhancing its generalization ability across different programming languages. Later models like CodeT5+ [31] further improved performance by using larger and more diverse corpora. Additionally, encoder-only models (e.g., CodeBERT) are generally more suitable for code understanding tasks, while decoder-only models (e.g., CodeGPT) excel at code generation tasks. Encoder-decoder models (e.g., CodeT5) combine the advantages of both, enabling them to handle both code understanding and generation tasks [36, 22, 23].

2.2 Just-in-Time Defect Prediction and Localization

Early studies in JIT-DP typically involved extracting key information from code changes and applying machine learning techniques to build predictive models.

[3] https://github.com/Hugo-Liang/JIT-Coka

Building upon prior work, Kamei et al. [12] proposed 14 expert-designed change-level features derived from five dimensions: size, diffusion, purpose, history, and author experience. They constructed a JIT-DP model based on logistic regression using these features. Following this, the DBN-JIT [34] model leveraged Deep Belief Networks to model the same expert features. The JITLine [24] model further combined expert features with token-level information extracted from code changes as input to a Random Forest [2] classifier for DP. Additionally, it employed LIME [26] to interpret token-level contributions, enabling DL.

Subsequent multi-task learning methods based on deep learning focused on extracting semantic information directly from the content of code changes for both JIT-DP and DL. DeepJIT [10] employed TextCNN [14] to encode both code changes and commit messages for predicting whether a commit was defective. Later, CC2Vec [11] introduced a hierarchical representation strategy that differentiated between added and deleted lines of code. However, Zeng et al. [37] observed that models like DeepJIT and CC2Vec did not consistently outperform simple logistic regression models (such as LApredict) that relied solely on basic code change metrics.

More recent JIT-DP and DL models have started integrating semantic features obtained from pre-trained code models (CodePTMs) with the 14 expert features for final classification. In particular, JIT-Fine [21] and the current state-of-the-art JIT-DP model JIT-Smart [3] feed both commit messages and code changes into CodeBERT to obtain semantic representations, which are then concatenated with dimension-expanded expert features for DP. The difference lies in their DL approach: JIT-Fine uses an attention mechanism to aggregate token-level defect contribution scores to the line level and performs localization by ranking, while JIT-Smart introduces a specially designed Defect Localization Network (DLN) composed of bidirectional LSTMs and attention mechanisms for both DP and DL tasks.

In contrast, our proposed model JIT-Coka employs a KANLinear module to explicitly distinguish semantic features from expert features, aiming to improve DP performance. Furthermore, we incorporate CodeT5 to enhance the model's overall capability in both DP and DL tasks.

3 Approach

Figure 1 illustrates the framework of our proposed JIT-DP and DL model, JIT-Coka. In the model training and testing part, the preprocessed data go through three stages: input representation, KANLinear classification, and defect prediction and localization. The core idea is to leverage an encoder-decoder based CodePTM (i.e., CodeT5) to capture semantic features of commit message and code changes, further fuse these features with expert features, and finally classify the fused features using the customized KANLinear. In the meantime, the line-level code changes are fed into the CodeT5 model to generate token embeddings, then the defect localization network (DLN) is utilized to locate defective lines.

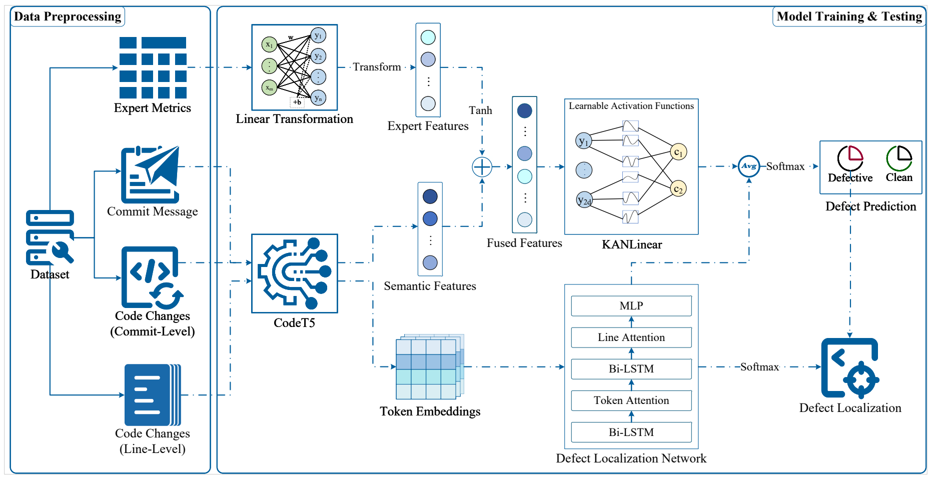

Fig. 1. Framework of the JIT-Coka model.

Eventually, the loss of both KANLinear and DLN are calculated and weighted averaged to update model parameters.

3.1 Input Representation

The model's input comprises of four parts: expert metrics, commit message, commit-level and line-level code changes. The commit message and commit-level code changes and are first tokenized into sequences of tokens and then concatenated into a unified input sequence. Specifically, let the token sequence for the commit message be:

$$\mathbf{M} = [m_1, m_2, \ldots, m_o], \tag{1}$$

and the token sequence for the commit-level code changes be:

$$\mathbf{C} = [a_1, a_2, \ldots, a_p, d_1, d_2, \ldots, d_q], \tag{2}$$

where o represents the length of the tokenized message, while p and q represent the lengths of the tokenized added code and deleted code in the code changes. The concatenated input sequence is formalized as:

$$\mathbf{X} = [[CLS], m_1, \ldots, m_o, [ADD], a_1, \ldots, a_p, [DEL], d_1, \ldots, d_q, [SEP]], \tag{3}$$

where $[CLS]$ and $[SEP]$ are the special beginning and end tokens of the model input, respectively. The $[ADD]$ and $[DEL]$ tokens serve as the separator for the code change content. The input sequence $\mathbf{X}$ is then fed into the CodeT5 encoder to obtain its hidden layer representation:

$$\mathbf{H} = \text{CodeT5}(\mathbf{X}) \in R^{n \times d}, \tag{4}$$

where n is the predetermined max input length (i.e., 512) of the input sequence, and d is the hidden layer dimension of CodeT5. The representation of the first token, $\mathbf{Y}_{[\mathrm{CLS}]} = \mathbf{H}[0] \in R^d$, is used as the semantic feature of the code change.

The normalized expert metrics are passed through a linear layer for dimensionality expansion to match the dimension of the semantic features:

$$\mathbf{Y}_{\mathrm{EF}} = \mathrm{Linear}(k, d)(\mathbf{Y}_{\mathrm{v}}) \in R^d, \tag{5}$$

where k is the number of expert metrics (i.e., 14) for each sample, and $\mathbf{Y}_{\mathrm{v}} \in R^k$ is the normalized value of the expert metrics.

Finally, the semantic features $\mathbf{Y}_{[\mathrm{CLS}]}$ of commit-level code changes and the expert features $\mathbf{Y}_{\mathrm{EF}}$ of expert metrics are concatenated to form the fused features for each sample:

$$\mathbf{Y}_{\mathrm{FF}} = \mathrm{Concat}(\mathbf{Y}_{[\mathrm{CLS}]}, \mathbf{Y}_{\mathrm{EF}}) \in R^{2d}. \tag{6}$$

3.2 KANLinear Classification

To enhance the model's ability to capture nonlinear relationships among fused features, we design a classification head based on the KAN, referred to as KANLinear. The fused feature vector $\mathbf{Y}_{\mathrm{FF}}$ is passed through this KANLinear layer to obtain the final prediction logits $\mathbf{Z}_{\mathrm{KAN}} \in R^c$, where $c = 2$ denotes the number of classes (Defective and Non-defective):

$$\mathbf{Z}_{\mathrm{KAN}} = \mathrm{KANLinear}(2d, c)(\mathbf{Y}_{\mathrm{FF}}) \in R^c, \tag{7}$$

Unlike a traditional linear layer, KANLinear augments linear transformations with learnable spline-based nonlinearities, allowing for expressive and adaptive feature modeling. Formally, the KANLinear transformation can be formalized as:

$$\mathrm{KANLinear}(\mathbf{x}) = \mathbf{W}_{\mathrm{base}} \cdot \phi(\mathbf{x}) + \mathbf{W}_{\mathrm{spline}} \cdot B(\mathbf{x}), \tag{8}$$

where $\mathbf{W}_{\mathrm{base}} \in R^{c\times 2d}$ is the weight matrix for the base activation branch, $\phi(\cdot)$ is a nonlinear base activation function, $\mathbf{W}_{\mathrm{spline}} \in R^{c\times 2d\times (g+s)}$ is the set of learnable spline weights, $B(\mathbf{x}) \in R^{2d\times (g+s)}$ represents the B-spline basis functions applied to $\mathbf{x}$, g is the number of grid points (i.e., grid size), and s is the spline order.

In this architecture, each input dimension is independently mapped using B-spline interpolation defined over a fixed or learnable grid. This mechanism enables localized, smooth nonlinear transformations while retaining the computational benefits of linear layers.

Both the base activation and spline basis functions are differentiable and optimized jointly with the rest of the network via backpropagation. The final output combines both the globally expressive base path and the locally adaptive spline path, allowing the model to capture both global trends and fine-grained variations in the data.

This hybrid formulation enables the KANLinear to flexibly interpolate between purely linear and highly nonlinear behavior depending on the data. Empirically, we find that this design improves classification performance, especially in scenarios where traditional fully connected layers struggle to model subtle nonlinearities in fused features.

3.3 Defect Prediction and Localization

The defective lines localization method in our proposed JIT-Coka model is primarily inherited by the design of the DLN module in JIT-Smart [3]. It aims to identify potentially defective lines within commits that are classified as defective. The key idea is to extract token embeddings for each line from CodeT5 and apply the DLN for fine-grained localization.

Formally, let the line-level code changes of a given commit be denoted as:

$$\mathbf{L} = \{l_1, l_2, \ldots, l_r\}, \tag{9}$$

where r denotes the number of changed lines, and each l_i is a sequence of tokens. Each line l_i is tokenized and encoded by the CodeT5 encoder to obtain its token embeddings:

$$\mathbf{E}_i = \text{CodeT5}(l_i) \in R^{t_i \times d}, \tag{10}$$

where t_i is the number of tokens in line l_i.

The DLN then processes these embeddings through several stages. Specifically, the Token-level Bi-LSTM layer is firstly utilized for obtaining contextualized token representations for each $\mathbf{E}_i$. Then the Token Attention layer is applied to summarize each line into a single vector. After that, the Line-level Bi-LSTM layer further processes the line vector to model contextual dependencies. Eventually, the Line Attention layer and multi-layer perceptron (MLP) layer are used to assign importance scores to different lines in a commit. Formally,

$$\mathbf{Z}_{\text{DLN}} = \text{DLN}(\mathbf{E}_{\mathbf{L}}) \in R^r. \tag{11}$$

To leverage the interaction between classification and localization, we aggregate the prediction output from DLN with the commit-level prediction. Specifically, the final prediction for DP are computed as:

$$\mathbf{P}_{\text{DP}} = \text{Softmax}\left(\frac{1}{2} \cdot (\mathbf{Z}_{\text{KAN}} + \mathbf{Z}_{\text{DLN}})\right), \tag{12}$$

where $\mathbf{Z}_{\text{KAN}}$ is the output from the KANLinear layer in the classification module.

During training, we jointly optimize the JIT-Coka model using a weighted sum of the classification and localization losses:

$$\mathcal{L}_{\text{JIT-Coka}} = \lambda_{\text{DP}} \cdot \mathcal{L}_{\text{DP}} + \lambda_{\text{DL}} \cdot \mathcal{L}_{\text{DL}}, \tag{13}$$

where λ_{DP} and λ_{DL} are the weights for the classification and localization loss terms respectively. We use cross-entropy for both loss components.

Importantly, DL is only activated when the model correctly classifies a commit as defective. This design ensures that the line-level DL module is guided by valid classification signals.

4 Experimental Setup

4.1 Baselines

We compare our proposed JIT-Coka against the following baseline methods, which include both joint JIT-DP and DL models as well as classical JIT-DP-specific models:

JITLine [24]. This model feeds expert features along with token-level features extracted from code changes into a Random Forest classifier for DP. It then applies a trained LIME model to estimate the contribution of each token, enabling DL at the line level.

JIT-Fine [21]. JIT-Fine concatenates high-dimensional semantic features extracted by CodeBERT with expanded expert features, and applies a linear layer for DP. For DL, it computes the contribution of each token in the modified code to the final classification result and ranks code lines based on the aggregated contribution of their tokens.

JIT-Smart [3]. As the current state-of-the-art (SOTA) model for JIT-DP and DL tasks, JIT-Smart extends JIT-Fine by incorporating the focal loss to better handle class imbalance. Moreover, it introduces a specialized DLN, significantly improving performance in the DL task.

In addition to the above three models that perform both JIT-DP and DL, we also evaluate the following three classical models designed solely for JIT-DP:

DBN-JIT [34]. This model inputs expert features into a deep belief network to classify whether a code change introduces a defect.

DeepJIT [10]. As the first deep learning model specifically designed for JIT-DP, DeepJIT uses TextCNN to extract semantic features from commit messages and code changes for binary classification.

LApredict [37]. A simple logistic regression model that uses only one expert feature—the number of added lines of code—to predict defectiveness.

It is worth noting that the CC2Vec [11] model was excluded from our evaluation. During its initial representation learning phase, CC2Vec incorporates data from the test set, which violates the temporal constraints of the JIT-DP task [29, 24]. Additionally, as it is not specifically designed for JIT-DP and is computationally expensive, we chose not to include it in our comparative analysis.

4.2 Dataset

We conduct experiments on the **JIT-Defects4J** dataset, which was released by the Ni et al. [21]. This dataset is constructed based on the manually annotated LLT4J dataset [9], where buggy lines have been labeled with high accuracy, enabling reliable evaluation of both JIT-DP and DL tasks.

JIT-Defects4J consists of 27,319 commit samples from 21 Java projects, totaling 177,125 lines of code. These projects span multiple software domains and cover commit histories from 2001 to 2019. The defect ratios at the commit and line levels are 8.54% and 7.11%, respectively. Each commit sample contains the commit message, added and deleted lines of code, and a set of 14 expert features extracted using the CommitGuru [27] tool, as originally proposed by Kamei et al. [12]. These expert features capture various aspects of code changes across five dimensions: size, diffusion, purpose, history, and author experience, and have been widely adopted in ML-based JIT-DP studies.

Following prior work, we divide the dataset into training, validation, and test sets in an 80%-20%-20% split by project. We also ensure that the commits in the test set occur chronologically after those in the training and validation sets to satisfy the time-sensitive requirements of JIT-DP evaluation. Summary statistics of the dataset are presented in Table 1. More detailed distributions for each project and descriptions of expert features can be found in our open-source reproduction package.

Table 1. Statistics of the JIT-Defects4J dataset

Partition	**Commit-Level**		**Line-Level**	
	Commit	**Defective (Ratio %)**	**Line**	**Defective (Ratio %)**
Train	16,374	1,390 (8.49%)	117,818	7,872 (6.68%)
Valid	5,465	467 (8.55%)	32,642	2,611 (8.00%)
Test	5,480	475 (8.67%)	26,665	2,111 (7.92%)
Total	27,319	2,332 (8.54%)	177,125	12,594 (7.11%)

4.3 Research Questions

To evaluate the effectiveness of our proposed JIT-Coka model, we aim to answer the following three research questions (RQs):

- **RQ1**: What is the best performance that JIT-Coka and relevant baselines can achieve on the JIT-Defects4J dataset?
- **RQ2**: How do JIT-Coka and baselines perform over multiple runs on the JIT-Defects4J dataset?
- **RQ3**: What is the effectiveness of each component in the JIT-Coka model?

4.4 Evaluation Metrics

We evaluate the DP performance of JIT-Coka and related baselines using five common metrics derived from the confusion matrix:

Precision. The proportion of true positives among all predicted positives: Precision = $\frac{TP}{TP+FP}$. **Recall** (or True Positive Rate, TPR). The proportion of actual positives that are correctly identified: Recall = $\frac{TP}{TP+FN}$. **F1-Score**. The harmonic mean of precision and recall: F1-Score = $\frac{2\times(\text{Precision}\times\text{Recall})}{\text{Precision}+\text{Recall}}$. **Matthews Correlation Coefficient (MCC)**. A balanced metric suitable for imbalanced datasets: MCC = $\frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

To complement these metrics, we also report the **Area Under the Receiver Operating Characteristic Curve (AUC-ROC)**, a threshold-independent metric that measures a classifier's performance across all thresholds. The ROC curve plots the False Positive Rate (FPR) against the TPR, where FPR $= \frac{FP}{TN+FP}$. Note, however, that the AUC-ROC score provides limited discriminative power unless one ROC curve fully dominates another. Moreover, the use of AUC as a metric for classification of imbalanced data has been found to be misleading [8].

For DL, we adopt **Top-5 Accuracy** and **Top-10 Accuracy** to evaluate the model's ability to rank truly defective lines near the top of its predictions.

Due to space limitations, we omit effort-aware metrics, and instead provide the complete evaluation results in our open-source reproduction package.

To assess the statistical significance of performance differences across multiple runs, we conduct one-sided non-parametric **Wilcoxon Rank-Sum test**, which is also known as the **Mann-Whitney U Test**, and compute the corresponding **effect size** r to measure the magnitude of the difference. The effect size r is calculated based on the Mann–Whitney U statistic as: $r = \frac{|z|}{\sqrt{2n}}$, where z is the corresponding standardized test statistic and n is the number of repeated runs per model (i.e., 5).

This effect size r complements p-values by providing insight into the practical significance of observed differences, which is especially useful when comparing model performance across repeated experiments.

4.5 Parameter Settings

Before feature fusion, both the expanded expert features and the semantic features have a dimensionality of 768. A Dropout with a rate of 0.1 is applied before feeding the concatenated features into the final classification head. In the KAN-Linear module, the grid size g is set to 5, the spline order s is set to 3, and the base activation function $\phi(\cdot)$ is implemented using SiLU [4]. The loss weights of λ_{DP} and λ_{DL} are set to 0.3 and 0.7, respectively.

During model training, the batch size is set to 8, the learning rate is set to 1e-5, and the total number of training steps is 50. Early stopping is employed with a patience value of 5 to reduce training time. All experiments are conducted on a server equipped with an NVIDIA Tesla P40 GPU (24GB), Intel(R) Xeon(R) Gold 5118 CPU, 100GB of RAM, and running CentOS 7.6. Notably, JIT-Coka completes training in about 7.5 hours, which is roughly 1.4× faster than JIT-Smart ($\approx$ 10.5 hours). For testing, JIT-Coka processes 5,480 samples in 205 seconds ($\sim$ 0.037s/sample), nearly identical to JIT-Smart ($\sim$ 0.035s/sample).

For the model comparison experiments in RQ1 and RQ2, each model is trained and evaluated five times using five randomly selected seeds: 42, 88, 1234, 2024, and 2048. It is worth noting that while the original JIT-Smart paper reports the best performance with a seed value of 0, our reproduction results show that the model performs poorly on DP tasks under this seed and is consistently outperformed by our proposed JIT-Coka model in most evaluation metrics.

5 Experimental Results and Analysis

5.1 RQ1: What is the best performance that JIT-Coka and relevant baselines can achieve on the JIT-Defects4J dataset?

The best performance of JIT-Coka and related baselines on the JIT-Defects4J dataset for the JIT-DP task and their corresponding DL performance are shown in Table 2 and Table 3, respectively. Best-performing metrics are highlighted in **bold** for each metric. Notably, we primarily select the best-performing experimental results based on the F1 score for the DP task, and report all evaluation metrics from that experiment. Under this setting, the scores for some metrics other than F1 may not be the highest. For example, although JIT-Coka achieves the best F1 score on the DP task, its Top-5/10 Accuracy scores for the DL task may not be the best among all experiments.

Table 2. Optimal JIT-DP performance of JIT-Coka and related baselines on the JIT-Defects4J dataset

Model	**Precision**	**Recall**	**F1-score**	**MCC**	**AUC-ROC**
Deeper	0.1748	0.4295	0.2485	0.1629	0.6772
LApredict	0.4545	0.0316	0.0591	0.1018	0.6938
DeepJIT	0.2126	**0.6632**	0.3219	0.2724	0.7911
JITLine	**0.6391**	0.1789	0.2796	0.3096	0.8087
JIT-Fine	0.4792	0.3874	0.4284	0.3829	0.8777
JIT-Smart	0.5023	0.4611	0.4808	0.4343	**0.8916**
JIT-Coka	0.5463	0.4842	**0.5134**	**0.4713**	0.8887

Table 3. DL performance of JIT-Coka and related baselines when achieving their optimal JIT-DP performance on the JIT-Defects4J dataset

Model	**Accuracy**	
	Top-5	**Top-10**
JITLine	0.1339	0.1214
JIT-Fine	0.1749	0.1672
JIT-Smart	0.5409	0.3943
JIT-Coka	**0.5459**	**0.4038**

From Table 2, we observe that although JIT-Coka does not achieve the highest scores in either Precision or Recall individually, it ranks second on both metrics and outperforms JIT-Smart, leading to the best F1 and MCC scores overall. In contrast, although JITLine and DeepJIT attain the best scores in

Precision and Recall respectively, their performance on the other metric is significantly worse, resulting in much lower F1 and MCC scores compared to the CodePTMs-based models (JIT-Fine, JIT-Smart, and JIT-Coka). Furthermore, while JIT-Smart achieves the best AUC-ROC score among multiple runs, its advantage over JIT-Coka is minimal, with a difference of only 0.3%.

As shown in Table 3, JIT-Coka also achieves the best Top-5/10 Accuracy scores. However, the improvements over JIT-Smart are marginal (0.9% and 2.4%, respectively). We note that improving DL performance on top of strong DP performance is not trivial, as the computation of DL metrics depends on the initial DP results. In multiple runs, we observed that the model often performs worse in DP when it achieves the best DL performance. This pattern is observed across all three CodePTMs-based models.

Answer to RQ1: JIT-Coka achieves the best performance on several key metrics for both JIT-DP and DL tasks. Notably, compared to the current SOTA model JIT-Smart, JIT-Coka improves the F1 and MCC scores by 6.8% and 8.5%, respectively, and also yields better results in Top-5/10 Accuracy.

5.2 RQ2: How do JIT-Coka and baselines perform over multiple runs on the JIT-Defects4J dataset?

Figure 2 shows the boxplots of the JIT-DP and DL results for JIT-Coka and related baselines over five random seeds. Given the clear significant performance differences between the three CodePTMs-based models (JIT-Fine, JIT-Smart, and JIT-Coka) and other baselines on multiple metrics, we mark only the significance results among these three for simplicity.

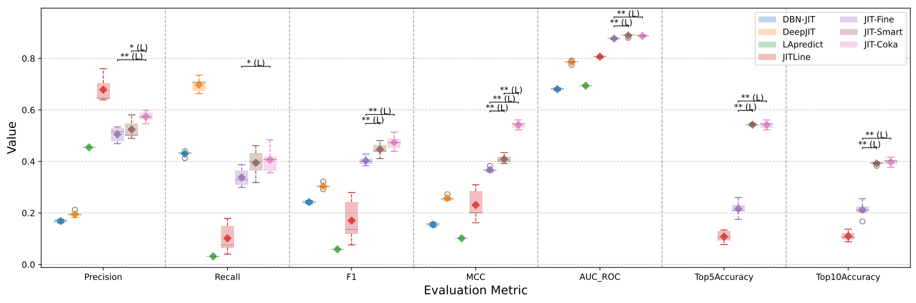

Fig. 2. Distributions of JIT-DP and DL performance of JIT-Coka and related baselines on the JIT-Defects4J dataset.

Consistent with the observations in RQ1, JIT-Coka consistently outperforms JIT-Smart in F1 and MCC metrics across multiple experiments, indicating its superior defect identification and overall classification capability on the JIT-Defects4J dataset. Specifically, JIT-Coka significantly outperforms JIT-Smart on the MCC metric with a large effect size (L), partly due to its significantly better Precision. While its advantages in Recall and F1 are not statistically significant, JIT-Coka still achieves higher minimum, average, and maximum scores than JIT-Smart on both metrics. The average MCC and F1 scores of JIT-Coka are 0.4417 and 0.4736, which are 8.0% and 6.1% higher than the corresponding scores of JIT-Smart (0.4089 and 0.4465). Furthermore, although JIT-Coka significantly outperforms JIT-Fine on AUC-ROC, the absolute differences are small and not statistically significant.

Similarly, in the DL task, both JIT-Coka and JIT-Smart significantly outperform other DL-enabled baselines in Top-5/10 Accuracy, though the performance difference between them is not statistically significant. JIT-Coka's average Top-5/10 Accuracy scores are 0.5417 and 0.3977, slightly different from JIT-Smart's (0.5426 and 0.3922). It is worth noting that although JIT-Coka achieves the best Top-5/10 Accuracy (0.5612 and 0.4167), its F1 and MCC scores are only 0.4529 and 0.4293. Likewise, JIT-Smart's best Top-5/10 Accuracy scores (0.5593 and 0.4097) correspond to F1 and MCC of 0.4645 and 0.4188. This reinforces the idea that DL capability should be evaluated on top of DP performance, rather than in isolation.

It is also worth mentioning that several specialized models for JIT-DP or DL show instability on specific metrics, which may limit their practical applicability. Future work could focus on improving their robustness. Moreover, the machine learning-based JITLine and the first deep-learning-based DeepJIT model achieve the best Precision and Recall respectively. Future research could explore ways to improve their Recall and Precision to enhance overall classification performance.

Answer to RQ2: JIT-Coka consistently achieves significantly better Precision and MCC scores than JIT-Smart with large effect size, while maintaining strong DL capability. Conducting multiple experiments and evaluating models with diverse metrics helps reveal each model's strengths and weaknesses, providing valuable insights for further optimization.

5.3 RQ3: What is the effectiveness of each component in the JIT-Coka model?

To assess the contribution of each component in JIT-Coka, we conduct comprehensive ablation studies. Specifically, we test four variants: using CodeBERT instead of CodeT5 (w/o CodeT5), using a standard Linear layer for DP instead of KANLinear (w/o KANLinear), removing the DLN module and ranking token-level attention contributions as in JIT-Fine (w/o DLN), and removing the 14 expert features to rely only on CodeT5-based semantic features (w/o EF). Due to resource constraints, we perform ablations using the best-performing seed (2048) of JIT-Coka. The results are shown in Table 4.

Table 4. Ablation study of different components in the JIT-Coka model

Model	JIT-DP Metrics					DL Accuracy	
	Precision	**Recall**	**F1-score**	**MCC**	**AUC-ROC**	**Top-5**	**Top-10**
JIT-Coka	0.5463	**0.4842**	**0.5134**	**0.4713**	0.8887	0.5459	0.4038
– w/o CodeT5	0.5045	0.4674	0.4852	0.4388	**0.8905**	0.5435	**0.4063**
– w/o KANLinear	0.5875	0.3958	0.4730	0.4433	0.8881	**0.5483**	0.4003
– w/o DLN	**0.5893**	0.4168	0.4883	0.4565	0.8894	0.1992	0.2033
– w/o EF	0.3789	0.4547	0.4134	0.3539	0.8610	0.5424	0.3953

The results indicate that CodeT5 affects both Precision and Recall, with a greater impact on Precision. Interestingly, this configuration yields the best AUC-ROC and Top-10 Accuracy scores. In contrast, KANLinear significantly affects Recall, leading to notable declines in F1 and MCC even though Precision remains relatively high. This setup achieves the best Top-5 Accuracy in DL.

Removing the DLN module leads to the best Precision, but poor Recall, resulting in suboptimal F1 and MCC. Notably, while DP performance is nearly the second-best, DL performance (Top-5/10 Accuracy) is the worst, suggesting DLN plays a more critical role in DL than DP, consistent with observations from the JIT-Smart study.

When expert features are excluded, the model's Precision drops significantly, and despite a slight decrease in Recall, the resulting F1, MCC, and AUC-ROC are the worst. However, DL performance decreases only slightly, indicating that expert features are more crucial for DP.

Additionally, since the DLN module influences JIT-Coka's DP performance through a weighted loss during training, we further investigate the effect of varying the weight ratios between DP and DL losses. The results are summarized in Table 5.

Table 5. Performance of JIT-Coka with different loss weight ratios for DP and DL

Loss Weight		JIT-DP Metrics					DL Accuracy	
$\lambda_{\mathbf{DP}}$	$\lambda_{\mathbf{DL}}$	**Precision**	**Recall**	**F1-score**	**MCC**	**AUC-ROC**	**Top-5**	**Top-10**
0.1	0.9	0.4837	0.4695	0.4765	0.4277	0.8853	0.5466	0.4028
0.2	0.8	**0.5769**	0.4105	0.4797	0.4467	0.8868	0.5379	0.3973
0.3	0.7	0.5463	**0.4842**	**0.5134**	**0.4713**	**0.8887**	0.5459	0.4038
0.4	0.6	0.5083	0.4526	0.4788	0.4334	0.8882	0.5474	0.4051
0.5	0.5	0.5174	0.4695	0.4923	0.4473	0.8881	0.5478	0.4033
0.6	0.4	0.5208	0.4484	0.4819	0.4382	0.8831	0.5453	0.4069
0.7	0.3	0.4903	0.4779	0.4840	0.4358	0.8885	0.5435	0.4005
0.8	0.2	0.4784	0.4653	0.4717	0.4224	0.8819	**0.5518**	**0.4118**
0.9	0.1	0.4715	0.4695	0.4705	0.4203	0.8863	0.5402	0.4004
1.0	0.0	0.5403	0.4379	0.4837	0.4431	0.8818	0.5328	0.3934

It is evident that when the loss weights for DP and DL are set to 0.3–0.7, the JIT-Coka model achieves the best DP and overall classification performance on the JIT-DP task, while also maintaining good performance on the DL task. Under this setting, the model attains the highest scores in Recall, F1, MCC, and AUC-ROC.

Although the model achieves the best Precision score when the DP and DL loss weights are set to 0.2–0.8, its Recall is the lowest under this setting, which negatively impacts its performance on other comprehensive evaluation metrics. Similarly, when the DP and DL loss weights are set to 0.8–0.2, the model achieves the best Top-5/10 Accuracy scores on the DL task, but its DP and overall classification performance are suboptimal.

Interestingly, within the 0.3–0.7 interval for the DP loss weight, only the value of 0.4 leads to a relatively poor F1 score for the JIT-Coka model (i.e., below 0.48), while in all other cases, the model consistently achieves F1 scores superior to those of JIT-Smart.

Moreover, when the loss weights are set to 1–0 (i.e., DL is not optimized), the JIT-Coka model still performs well on the DP task, which is consistent with the ablation study observations of the DLN module in RQ2. The difference, however, is that the model continues to perform reasonably well on the DL task instead of suffering a significant drop. These findings suggest that future work may consider decoupling the DLN module to construct a dedicated JIT-DL model.

Answer to RQ3: All components of JIT-Coka contribute effectively to its overall performance in both JIT-DP and DL tasks. While some components may reduce performance on specific metrics, they remain essential for achieving strong overall results, indicating that there is still room for further model optimization.

6 Discussion: How do JIT-Coka and baselines perform on dataset comprising multiple programming languages?

To further evaluate the performance of JIT-Coka on dataset involving multiple programming languages, we conducted experiments on the dataset released by literature [37]. This dataset consists of approximately 9.5k samples collected from six projects with varying programming languages and defect ratios. Among them, Gerrit, JDT, and Platform are Java projects, Go is a Golang project, OpenStack is a Python project, and QT is a C++ project. Since this dataset does not provide line-level defect annotations, we only assessed the JIT-DP performance of JIT-Coka and the relevant baselines. Following the experimental procedure in 5.1, the distributions of F1 and MCC scores across multiple runs are illustrated in Figure 3, and the complete results of all evaluation metrics are provided in our open-source replication package. For consistency with the original paper, we refer to this dataset as *LApredict*.

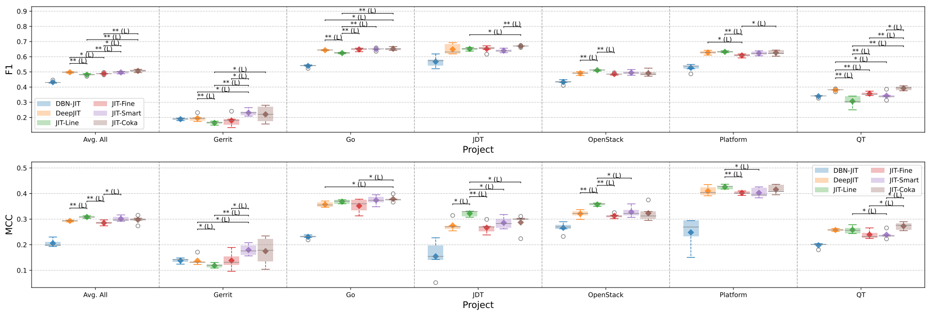

Fig. 3. Distributions of JIT-DP performance of JIT-Coka and related baselines on the *LApredict* dataset.

Overall, on the average performance across all projects (Avg. All), JIT-Coka achieves significantly better F1 scores than JIT-Line and JIT-Fine. In addition, JIT-Coka consistently outperforms JIT-Smart in terms of the minimum, maximum, and mean F1 scores. By contrast, JIT-Smart shows a slight advantage in MCC, but this difference is not statistically significant. It is worth noting that DeepJIT and JIT-Line achieve the best overall performance on F1 and MCC, respectively, but their advantages over JIT-Coka are not statistically significant, and both models perform relatively worse on the other metric.

More specifically, JIT-Coka achieves significantly better F1 and MCC scores than JIT-Fine and JIT-Smart on the QT project. This result is expected, since the backbone model of JIT-Coka, CodeT5, was pre-trained on corpora including the C programming language, whereas the backbone of the other two models, CodeBERT, was not. In addition, JIT-Coka shows a significant improvement over JIT-Smart on the JDT project. Although JIT-Coka does not always demonstrate significant improvements over the best or second-best baselines on other projects, it still exhibits unique advantages. For instance, it achieves a significantly better F1 score than DeepJIT on the Go project, while JIT-Fine and JIT-Smart, with comparable performance, do not.

In summary, JIT-Coka demonstrates overall superior performance on the multi-language dataset, further confirming the advantage of using CodeT5 rather than the earlier CodeBERT as the backbone model. Although it does not always achieve statistically significant improvements over the second-best baseline in all projects, JIT-Coka consistently maintains mid-to-high ranking performance across projects, and often outperforms other models in terms of the minimum, maximum, or mean values of evaluation metrics.

7 Threats To Validity

Construct Validity. In this study, we selected widely accepted metrics such as Precision, Recall, F1, MCC, AUC-ROC, and Top-K Accuracy to evaluate model performance. These metrics are consistent with those used in prior studies, ensuring comparability and credibility of our results. However, these metrics may still have limitations. For example, F1 score balances Precision and Recall but may obscure performance in severely imbalanced datasets. Similarly, Top-K Accuracy may not fully reflect developer needs in real debugging scenarios. To mitigate these threats, we relied on both standard benchmarks (JIT-Defects4J) and multiple metrics, and performed comprehensive ablation experiments to validate the role of each component. Still, future work could explore more fine-grained, developer-centric metrics and feature representations to better reflect real-world constructs.

Internal Validity. Although the architecture of JIT-Coka is designed to be as simple as possible and allows for easy replacement of relevant modules, certain parameters still need to be adjusted to validate the results. While we have conducted experiments on key parameters such as DP and DL loss weights, resource constraints prevented us from exhaustively exploring all experimental settings and parameter combinations. For instance, parameters such as the number of training steps, early stopping strategies, and learning rates may still pose threats to the validity of our experimental findings. Moreover, the raw values of expert features may contain noise or redundant information, affecting the model's classification performance. Similarly, the quality of commit messages and code changes can also impact model performance. To mitigate these threats, we applied Dropout to randomly discard certain neurons before classification to enhance model robustness.

External Validity. Since model performance may vary on datasets with a larger or smaller number of samples, practitioners should re-evaluate the model on their target datasets before deployment. Furthermore, the input methods and utilization of expert features that work well with JIT-Coka may not be directly applicable to other models. Future research on JIT-DP and DL models should consider that the methods optimized for JIT-Coka may not be the optimal choice for their own models. To mitigate these threats, we have demonstrated the effectiveness of JIT-Coka both theoretically and through comprehensive experiments. Moreover, our experimental results indicate that JIT-Coka may complement existing models. Therefore, the proposed approach in this paper exhibits strong generalizability.

8 Conclusion and Future Work

In this paper, we proposed JIT-Coka, a unified model for Just-in-Time Defect Prediction and Localization, which effectively integrates pre-trained semantic features and handcrafted expert features through an adaptive nonlinear classification module. Leveraging the encoder-decoder architecture of CodeT5 and a robust KANLinear classifier, JIT-Coka improves the Precision, F1 and MCC for defect detection. Moreover, the use of the DLN module enables effective line-level localization, tightly coupled with commit-level classification.

JIT-Coka provides a solid foundation for fine-grained software quality assurance and has the potential for broader adoption in practice. Future directions include: (1) improving the defect localization module, e.g., by redesigning or modularizing the DLN to explicitly enhance localization without harming prediction, and (2) extending multi-task capabilities by leveraging the encoder-decoder structure of CodeT5 for related tasks such as commit message generation or repair suggestion.

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China (No. 62372174), the Computational Biology Program of Shanghai Science and Technology Commission (No. 23JS1400600), 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 (No. 2021-GYHLW-01007), and the Shanghai 2024 Science and Technology Innovation Action Plan Star Cultivation (Sailing Program, No. 24YF2719900 and 24YF2720000).

References

1. Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W.: Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 2655–2668 (Jun 2021)
2. Breiman, L.: Random forests. Machine Learning **45**, 5–32 (2001)
3. Chen, X., Xu, F., Huang, Y., Zhang, N., Zheng, Z.: Jit-smart: A multi-task learning framework for just-in-time defect prediction and localization. Proc. ACM Softw. Eng. **1**(FSE) (Jul 2024)
4. Elfwing, S., Uchibe, E., Doya, K.: Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. Neural networks **107**, 3–11 (2018)
5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547 (Nov 2020)
6. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M.: Graphcodebert: Pre-training code representations with data flow. In: International Conference on Learning Representations (2021)
7. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. **38**(6), 1276–1304 (2012)
8. Hancock, J., Khoshgoftaar, T., Johnson, J.: Evaluating classifier performance with highly imbalanced big data. Journal of Big Data **10** (04 2023)
9. Herbold, S., Trautsch, A., Ledel, B., Aghamohammadi, A., Ghaleb, T.A., Chahal, K.K., Bossenmaier, T., Nagaria, B., Makedonski, P., Ahmadabadi, M.N., Szabados, K., Spieker, H., Madeja, M., Hoy, N., Lenarduzzi, V., Wang, S., Rodríguez-Pérez,

G., Colomo-Palacios, R., Verdecchia, R., Singh, P., Qin, Y., Chakroborti, D., Davis, W., Walunj, V., Wu, H., Marcilio, D., Alam, O., Aldaeej, A., Amit, I., Turhan, B., Eismann, S., Wickert, A.K., Malavolta, I., Sulír, M., Fard, F., Henley, A.Z., Kourtzanidis, S., Tuzun, E., Treude, C., Shamasbi, S.M., Pashchenko, I., Wyrich, M., Davis, J., Serebrenik, A., Albrecht, E., Aktas, E.U., Strüber, D., Erbel, J.: A fine-grained data set and analysis of tangling in bug fixing commits. Empirical Softw. Engg. **27**(6) (nov 2022)

10. Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N.: Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the 16th International Conference on Mining Software Repositories. p. 34–45 (2019)
11. Hoang, T., Kang, H.J., Lo, D., Lawall, J.: Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 518–529 (2020)
12. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. **39**(6), 757–773 (Jun 2013)
13. Kim, S., Whitehead, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. **34**(2), 181–196 (Mar 2008)
14. Kim, Y.: Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1746–1751 (Oct 2014)
15. Liang, Y.G., Fan, G.S., Yu, H.Q., Li, M.C., Huang, Z.J.: Automatic code summarization using abbreviation expansion and subword segmentation. Expert Systems **42**(2), e13835 (2025)
16. Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X.: Neural-machine-translation-based commit message generation: How far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 373–384 (2018)
17. Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljacic, M., Hou, T.Y., Tegmark, M.: Kan: Kolmogorov-arnold networks. In: The Thirteenth International Conference on Learning Representations (2025)
18. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S.: Codexglue: A machine learning benchmark dataset for code understanding and generation. CoRR **abs/2102.04664** (2021)
19. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5**(2), 169–180 (2000)
20. Moussa, R., Sarro, F.: On the use of evaluation measures for defect prediction studies. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 101–113 (2022)
21. 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. p. 672–683 (2022)
22. Niu, C., Li, C., Luo, B., Ng, V.: Deep learning meets software engineering: A survey on pre-trained models of source code. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22. pp. 5546–5555 (7 2022)

23. Niu, C., Li, C., Ng, V., Chen, D., Ge, J., Luo, B.: An empirical comparison of pre-trained models of source code. In: Proceedings of the 45th International Conference on Software Engineering. p. 2136–2148 (2023)
24. Pornprasit, C., Tantithamthavorn, C.K.: Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. pp. 369–379 (2021)
25. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019)
26. Ribeiro, M.T., Singh, S., Guestrin, C.: " 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. pp. 1135–1144 (2016)
27. Rosen, C., Grawi, B., Shihab, E.: Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 966–969 (2015)
28. Spacco, J., Hovemeyer, D., Pugh, W.: Tracking defect warnings across versions. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. p. 133–136 (2006)
29. Tan, M., Tan, L., Dara, S., Mayeux, C.: Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2, pp. 99–108 (2015)
30. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 6000–6010 (2017)
31. Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., Hoi, S.: Codet5+: Open code large language models for code understanding and generation. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. pp. 1069–1088 (2023)
32. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 8696–8708 (Nov 2021)
33. Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S.: Just-in-time defect identification and localization: A two-phase framework. IEEE Transactions on Software Engineering **48**(1), 82–101 (2022)
34. 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 (QRS). pp. 17–26 (2015)
35. Yao, J., Shepperd, M.: Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In: Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering. p. 120–129 (2020)
36. Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., Zhang, L.: An extensive study on pre-trained models for program understanding and generation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 39–51 (2022)
37. 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. p. 427–438 (2021)
38. Zhou, X., Han, D., Lo, D.: Assessing generalizability of codebert. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 425–436 (2021)