Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Bi-modal Semantic and Temporal Evolution Modeling for Just-in-Time Software Defect Prediction

Zheyuan Pan
Xinjiang Normal University
Yong Li
liyong@live.com

Xinjiang Normal University
Guanlong Yan
Xinjiang Normal University

Article

Keywords:

Posted Date: May 19th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9448100/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Bi-modal Semantic and Temporal Evolution Modeling for Just-in-Time Software Defect Prediction

Zheyuan Pan[1], Yong Li[1,2,*], and Guanlong Yan[1]

[1]School of Computer Science and Technology,Xinjiang Normal University, Urumqi, 830054, Xinjiang,China;107622024210529@stu.xjnu.edu.cn(Z.P.);107622024210521@stu.xjnu.edu.cn(G.Y.)
[2]Xinjiang Engineering Research Center for Smart Education and Application, Urumqi, 830054, Xinjiang, China
[*]corresponding:liyong@live.com

ABSTRACT

Just-in-Time Software Defect Prediction (JIT-DP) aims to identify potential defects at the code commit stage and serves as a critical technique for ensuring software quality. However, existing JIT-DP methods primarily rely on either hand-crafted metric features or deep learning-based semantic features. The former fail to capture the deep semantics of code changes, while the latter generally overlook the temporal evolution patterns between commits and the continuity of development intentions, thereby limiting prediction performance.

To this end, this study proposes a JIT defect prediction method based on bi-modal semantic and temporal evolution modeling. The method constructs a synergistic framework of semantic understanding and evolutionary analysis, achieving accurate prediction driven by both deep semantic representation and long-term temporal dependencies. First, we utilize the pre-trained BiCC-BERT model to explicitly learn the bi-modal semantic representations of commit messages and code changes via its innovative "Replaced Message Identification" objective, extracting deep semantic features for each commit. Second, historical commits are constructed into feature sequences in chronological order, and a Temporal Convolutional Network (TCN) is introduced for modeling. The TCN captures long-term dependencies within the sequence through causal dilated convolutions to learn the evolutionary patterns of code semantics. Finally, the temporal representations extracted by the TCN are fused with traditional expert features to perform classification prediction via a Multi-Layer Perceptron (MLP).

Experimental results on the JIT-Defects4J dataset demonstrate that the proposed method significantly outperforms various mainstream baseline models, including JIT-Fine and DeepJIT, across all key metrics. Specifically, the method achieves an F1-score of 0.4429 and an AUC of 0.8199, setting a new performance record for this task. Notably, benefiting from the effective capture of historical commit evolution patterns, the model reaches a Recall of 0.5460, significantly reducing the risk of defect leakage compared to baseline models. Furthermore, parameter sensitivity analysis confirms that introducing a historical time window of appropriate length significantly enhances the model's ability to understand code change intentions, verifying the necessity and effectiveness of temporal modeling in JIT defect prediction.

Please note: Abbreviations should be introduced at the first mention in the main text – no abbreviations lists. Suggested structure of main text (not enforced) is provided below.

Introduction

Software Quality Assurance (SQA) remains a core challenge in software engineering. As the scale and complexity of software systems continue to grow, identifying and remediating defects at the earliest possible stage of development is of paramount importance for reducing maintenance costs and ensuring system reliability[1]. Although Software Defect Prediction (SDP) techniques aim to anticipate risks by analyzing historical data, traditional file-level or module-level predictions typically suffer from long feedback latencies. This delay makes it difficult for developers to accurately recall the specific context in which the defect was introduced, thereby escalating remediation costs[2]. To address this latency issue, Just-in-Time Software Defect Prediction (JIT-DP) has emerged as a fine-grained and short-cycle prediction paradigm[3]. Its core premise is to immediately assess the risk of defect introduction at the moment code changes are submitted to the version control system. By directing developer attention toward high-risk changes, JIT-DP enables developers to conduct reviews or tests while the coding context remains fresh in their minds, thereby significantly enhancing defect remediation efficiency[4].

Existing JIT-DP approaches predominantly follow two main technical paradigms. The first relies on expert-defined metric features (e.g., the number of modified files, lines of code, and developer experience), employing traditional machine learning models such as Logistic Regression and Random Forest for prediction[4,5]. While these methods offer strong interpretability, the shallow statistical features they rely on are inadequate for capturing the complex semantics and development intentions underlying code changes. The second paradigm consists of deep learning-based semantic approaches, which utilize neural

networks to automatically learn distributed representations of code changes[6,7].However, most current semantic methods treat individual commits as isolated events: on one hand, they fail to adequately model and leverage the intrinsic, explicit semantic correlations between commit messages (natural language descriptions) and code diffs (structured changes); on the other hand, they generally overlook the continuous evolutionary nature of code in software projects, specifically the temporal dependencies and evolutionary patterns embedded within historical commit sequences. These limitations result in an insufficient depth of understanding regarding defect introduction mechanisms, thereby constraining further improvements in prediction performance.

To address these limitations, this study proposes a JIT defect prediction method based on bi-modal semantic and temporal evolution modeling. This method constructs a unified framework synergistically driven by semantic understanding and evolutionary analysis, aiming to achieve more precise risk assessment through a deep analysis of commit intentions and the capture of their evolutionary patterns across the temporal dimension.Specifically, we first utilize pre-training techniques (such as Transformer-based code models[8]) to explicitly learn the semantic alignment between commit messages and code changes, thereby extracting change representations rich in deep semantics. Subsequently, historical commits are organized into sequences in chronological order, and a Temporal Convolutional Network (TCN) is employed to capture long-term dependencies spanning multiple commits[9], facilitating the modeling of macro-evolutionary trends in code semantics. Finally, the learned temporal representations are fused with traditional expert features to be jointly utilized for defect prediction.

The main contributions of this paper are summarized as follows:

We propose a unified JIT-DP framework that integrates deep semantic understanding with long-term temporal analysis, effectively overcoming the bottlenecks of existing methods in semantic correlation modeling and temporal context utilization.

We innovate by combining a mechanism for explicitly learning code-message semantic alignment with Temporal Convolutional Networks (TCN)—which are adept at capturing long-term dependencies—thereby achieving an effective bridge from static semantic representations to dynamic evolutionary pattern analysis.

We conduct comprehensive experimental validation on public benchmark datasets. The results demonstrate that the proposed method significantly outperforms existing state-of-the-art (SOTA) approaches across multiple evaluation metrics. Furthermore, ablation studies confirm the individual effectiveness and synergistic effects of the bi-modal semantic features and the temporal modeling module.

This study not only provides a superior solution for JIT-DP but also offers a novel analytical perspective for deeply understanding the dynamic evolution processes of software projects.

The remainder of this paper is organized as follows: Section 2 reviews related work on code change representation learning, Just-in-Time defect prediction, and temporal modeling. Section 3 elaborates on the overall framework of the proposed method and the design and implementation of its core components. Section 4 presents the experimental setup, including research questions, datasets, baseline methods, and evaluation metrics, followed by a detailed analysis of the experimental results. Finally, Section 5 concludes the paper and outlines directions for future research.

2 Related Work

This chapter systematically reviews existing work closely related to the research presented in this paper. First, we outline the development trajectory of Just-in-Time Software Defect Prediction (JIT-DP) methods, categorizing them into two main streams for review: expert feature-based and semantic feature-based approaches. Second, we explore advancements in code change representation learning—a key enabling technology—analyzing the characteristics of two mainstream paradigms: sequence-based and graph-based approaches. Finally, we summarize the application status of temporal modeling techniques in the software engineering domain, with a specific focus on analyzing the existing gaps in temporal evolution analysis within current JIT-DP research.

2.1 Just-in-Time Software Defect Prediction Methods

Just-in-Time Software Defect Prediction (JIT-DP) aims to identify potential defect risks immediately upon code change submission by analyzing historical data. Based on the types of features utilized, existing research can be broadly categorized into two primary streams: expert feature-based methods and semantic feature-based methods.

2.1.1 Expert Feature-based JIT-DP

Early JIT-DP research heavily relied on metric features pre-defined by software engineering experts based on empirical knowledge. The core of these methods lies in abstracting complex code changes into a set of quantifiable statistical features. The 14 features defined by Kameiet al.[4] from five dimensions are widely adopted in the domain of just-in-time defect prediction and have been demonstrated for theireffectiveness on such prediction tasks.[10]Similarly, Pornprasit et al. introduced JITLine, which adopts a hybrid strategy; it combines traditional Bag-of-Words (BoW) features of code changes with expert-designed statistical metrics and employs a Random Forest classifier for defect prediction[11].

Such expert feature-based methods typically possess strong interpretability, require relatively low computational resources for training, and maintain competitive performance in certain specific scenarios. However, their fundamental limitation lies in the fact that hand-crafted features often represent an oversimplification of the complex software development process, making it difficult to capture the intrinsic, deep semantic factors that trigger software defects. Furthermore, the feature engineering process is typically time-consuming and labor-intensive, heavily reliant on project-specific domain knowledge, and thus demonstrates limited generalizability.

2.1.2 Semantic Feature-based JIT-DP

To overcome the limitations of shallow expert features, research focus has gradually shifted towards automatically learning latent semantic representations from Commit Messages and Code Diffs. Unlike expert feature-based approaches, these methods aim to enable models to directly comprehend the underlying semantics of both code and text.

Early explorations were primarily based on generic deep neural networks. A representative work is DeepJIT, proposed by Hoang et al., which utilizes Convolutional Neural Networks (CNNs) and Long Short-Term Memory networks (LSTMs) to process commit messages and code changes, respectively. This work demonstrated the effectiveness of deep semantic features in assessing the defectiveness of code changes[6]. Building upon this, they further proposed CC2Vec, which introduces a hierarchical attention mechanism. By leveraging the semantics of commit messages to guide the model to focus on critical parts of the code changes, CC2Vec extracts more discriminative representations, thereby achieving performance superior to that of DeepJIT[7].

With the advent of Pre-trained Language Models (PLMs), models based on the BERT family have gradually become mainstream. CodeBERT aims to learn contextual word embeddings and has been widely applied in numerous natural language processing (NLP) tasks[12] as well as software engineering (SE) tasks[13].

These methods benefit from the rich semantic knowledge acquired from large-scale unlabeled corpora, which significantly reduces the effort required for manual feature engineering and enhances prediction performance. However, whether it is early works like DeepJIT or the latest methods based on pre-trained models (e.g., JIT-Fine), most of them still treat each commit as an isolated static sample for analysis. Consequently, they overlook the continuous evolutionary relationships and dependency contexts of code changes across the temporal dimension.

2.1.3 Summary

In summary, Just-in-Time Software Defect Prediction has evolved from shallow, hand-crafted expert features to deep, automated semantic features. Although methods based on deep learning and pre-trained models have greatly enhanced the dimensionality of feature representation and prediction accuracy, mainstream works generally treat individual code commits as independent and identically distributed (i.i.d.) static samples. This "isolated" processing approach severs the natural contextual dependencies and temporal evolution patterns inherent in the software development process, making it difficult for models to gain insight into the deep mechanisms of defect introduction. This has become the primary limiting factor preventing current JIT-DP models from further breaking through performance bottlenecks.

2.2 Code Change Representation Learning

High-quality code change representation serves as the cornerstone for enhancing the performance of semantic-based JIT-DP. Unlike static code representation, code change representation learning aims to capture the evolutionary process of code repositories, focusing on the trajectories and patterns within the editing history to generate distributed vectors capable of reflecting fine-grained distinctions. Existing methods are primarily classified into two main categories: sequence-based and graph-based approaches.

2.2.1 Sequence-based Methods

Sequence-based methods treat code changes as streams of text or tokens and have been extensively studied due to their good compatibility with existing Natural Language Processing (NLP) techniques. Early works, such as CC2Vec by Hoang et al., utilize a hierarchical attention network guided by commit messages to learn code change representations. CoreGen, introduced by Nie et al., leverages context information to automatically generate commit messages based on the Transformer model, thereby indirectly facilitating code change understanding[14].

A prevailing trend in recent research involves designing specialized pre-training tasks tailored to the characteristics of code changes. For instance, CodeReviewer, introduced by Li et al., exploits four specific objectives within code review scenarios to jointly learn the representations of code changes and review comments[15]. Meanwhile, CCBERT proposed by Zhou et al.[16] and CCT5 proposed by Lin et al.[17]focus on developing pre-training models that rely solely on code diffs—independent of commit messages—aiming to capture finer-grained information regarding edit operations.

By learning from massive domain-specific data, these sequence-based pre-trained models are capable of capturing rich domain knowledge concerning code changes. In particular, mature models such as CodeBERT serve as powerful semantic feature extractors for JIT-DP tasks.

2.2.2 Graph-based Methods

Graph-based methods aim to preserve the structural information of code changes by analyzing Abstract Syntax Trees (ASTs) or Program Dependency Graphs (PDGs). In et al. introduced a Graph Neural Network (GNN)-based approach that learns representations from code ASTs via an auto-encoder framework[18]. Brody et al. proposed a structured model that utilizes paths within the AST to represent edit operations occurring in specific contexts[19]. Similarly, commit2Vec by Lozoya et al.[20]and the incremental edit model by Yao et al.[21]focus on leveraging the syntactic structure of ASTs to automatically extract generic representations of code changes or to learn the iterative editing process of the trees.

Theoretically, graph-based methods possess a superior capability to capture the structural semantics of code. However, the complexity of graph construction and the high computational overhead constrain their applicability in large-scale JIT prediction scenarios to some extent. In contrast, sequence-based methods strike a better balance between performance and efficiency.

2.2.3 Summary

In summary, high-quality code change representation learning provides a solid technical foundation for a deep understanding of development intent and code semantics. Among these, while graph-based methods can preserve fine-grained syntactic structures, their complex graph construction processes and high computational overhead limit their large-scale application in just-in-time prediction scenarios. Conversely, sequence-based pre-trained language models (such as the CodeBERT family) achieve an excellent balance between feature representational capability and computational efficiency, making them the most practical semantic extraction paradigm at present. However, existing representation learning schemes mostly focus on the static alignment of text and code within a single commit. The extracted feature vectors have not yet been effectively integrated into the dynamic evolutionary trajectories across commits, urgently requiring the incorporation of temporal analysis mechanisms to fully activate the value of these high-dimensional semantics in the temporal dimension.

2.3 Applications of Temporal Modeling in Software Engineering

The development of software systems is inherently a dynamic process of continuous evolution, where the codebase state, development activity characteristics, and quality attributes all undergo constant changes over time. Consequently, temporal modeling—as a pivotal technique capable of capturing and analyzing this dynamism—demonstrates significant value in software quality prediction and evolution analysis.

Early studies primarily leveraged statistical time-series models or sliding window-based metric aggregation to analyze trends in software quality. For instance, researchers predicted the defect-proneness of future modules or identified signs of Code Decay by tracking the evolutionary trajectories of metrics—such as code complexity and coupling—across successive versions, or by analyzing the time-series distribution of defect counts . Although these methods introduced the temporal dimension, they represent, in essence, a trend analysis of shallow, aggregated metrics, failing to delve into the semantic or structural evolutionary processes internal to code changes.

With the advancement of deep learning, Recurrent Neural Networks (RNNs) and their variants—particularly Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs)—have been widely incorporated into software engineering tasks due to their inherent advantages in sequence data modeling. These models have been successfully applied to: 1) predicting defects by learning from historical sequences of code metrics; 2) analyzing development activity trajectories to predict developers' next actions; and 3) forecasting maintenance effort based on sequences of historical issue reports. However, when processing long sequences of software evolution history, traditional RNN architectures face inherent limitations, such as high training complexity, difficulty in capturing extremely long-term dependencies, and challenges regarding parallelization.

In recent years, Temporal Convolutional Networks (TCNs) have garnered increasing attention as a novel architecture for sequence modeling. TCNs employ a 1D dilated causal convolution structure : their inherent causality ensures that predictions depend solely on past information, strictly adhering to the chronological order of software evolution; meanwhile, by stacking convolutional layers and utilizing exponentially increasing dilation factors, TCNs can effectively expand the receptive field to capture long-term dependencies. Furthermore, TCNs are computationally more efficient than RNNs and are more amenable to parallel training. Within the field of software engineering, TCNs have seen preliminary applications in tasks such as defect prediction and crash report classification, demonstrating superior potential.

Focusing specifically on the Just-in-Time Defect Prediction (JIT-DP) domain, the application of temporal modeling remains in its nascent stages. The few existing attempts that utilize LSTMs or simple RNNs to model dependencies between continuous commits typically limit their input features to traditional low-dimensional expert metrics (e.g., statistical values of LA, LD, and NS). Currently, there is a lack of systematic research exploring how to perform temporal evolution modeling on the deep semantic features of code changes—specifically, the high-dimensional distributed representations learned via pre-trained models[9]. In other words, existing works have failed to organically integrate two critical dimensions: "understanding the semantic content of individual commits" and "analyzing the evolutionary trajectory of semantics over time." This separation renders models incapable of identifying complex dynamic risk patterns, such as "specific semantic patterns of code changes that recur in project history and are associated with defects" or "implementation deviations of development intent that gradually

accumulate into defects across multiple commits." Therefore, integrating advanced temporal modeling architectures (such as TCNs) with deep semantic features represents a pivotal breakthrough for elevating the analytical paradigm of JIT-DP models.

2.3.1 Summary

In summary, temporal modeling has demonstrated tremendous potential in capturing the dynamic evolutionary characteristics of software systems. Compared to traditional RNN architectures, Temporal Convolutional Networks (TCNs) provide a superior solution for modeling long sequences of code evolution due to their strict causality, exponentially expanding receptive fields, and efficient parallel computing capabilities. Observing the current JIT-DP field, the exploration of temporal modeling remains in its infancy. In particular, systematic research that deeply integrates "the extraction of deep bi-modal semantic features" with "temporal evolution analysis based on advanced architectures (e.g., TCN)" remains a void. Therefore, constructing a unified framework that balances static deep semantic understanding with dynamic temporal evolution analysis to resolve the issues of feature isolation and context deficiency is a core challenge that current research urgently needs to overcome. This is precisely the fundamental motivation for proposing the T-BiCC method in this paper.

3 Methods

This chapter details the design and implementation of the proposed Just-in-Time Software Defect Prediction (JIT-DP) method based on bi-modal semantic and temporal evolution modeling . The method aims to construct a comprehensive prediction framework capable of fully leveraging deep semantic information from code commits, traditional expert domain knowledge, and the evolutionary patterns of historical commits. To achieve this objective, the proposed method systematically models the core processing workflow by dividing it into four primary phases in logical sequence: first, parallel dual-stream feature extraction to acquire heterogeneous semantic and expert features; second, bi-modal feature fusion to construct a unified sequence of commit representations; subsequently, temporal evolution modeling based on Temporal Convolutional Networks (TCN) to capture the dynamic evolutionary patterns of code changes over time; and finally, temporal context-based defect prediction. Through this end-to-end joint modeling paradigm, this chapter seeks to overcome the limitations regarding feature representation and the insufficiency of temporal context utilization in existing research, thereby achieving more precise identification of just-in-time software defects.

3.1 Overview of the General Framework

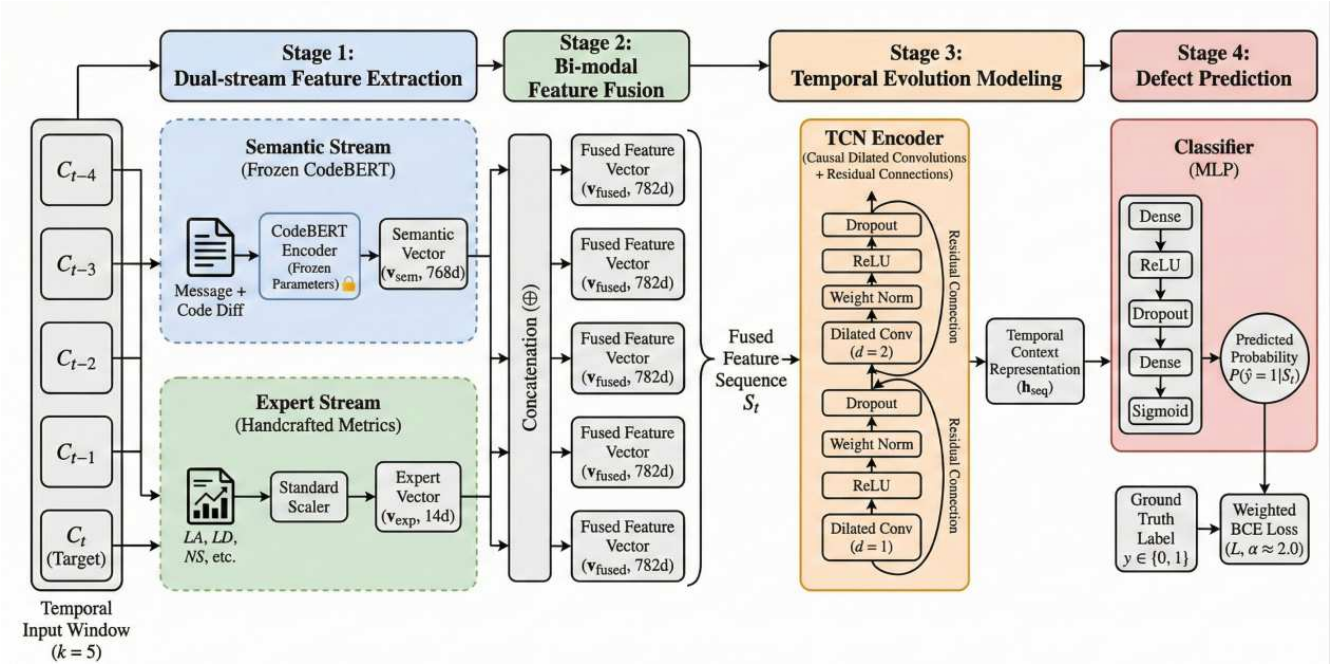


Figure 1. Overview of the T-BiCC Framework based on Bi-modal Semantics and Temporal Evolution Modeling

As shown in Figure 1, the proposed T-BiCC framework takes a temporal window containing the target commit and its adjacent historical context as input. It aims to jointly model the bi-modal features of code changes and their temporal evolution patterns through a cascaded processing workflow.

First, the framework performs parallel dual-stream feature extraction for each commit within the window. On one hand, this process utilizes a pre-trained CodeBERT model with frozen parameters to capture deep semantic information from unstructured commit messages and code changes; on the other hand, it synchronously calculates and normalizes traditional software engineering expert metrics (such as LA, LD, etc.). To obtain a comprehensive view of the commit state at each time step, these two sets of heterogeneous feature vectors are subsequently explicitly concatenated during the bi-modal fusion phase, thereby forming a time-serialized fused feature representation.

Subsequently, to capture the dynamic evolutionary patterns of these fused features over time, the sequence is fed into the core temporal evolution modeling module, namely, the Temporal Convolutional Network (TCN) encoder. Leveraging a structure of stacked causal dilated convolutions and residual connections, the TCN effectively expands the receptive field while strictly adhering to temporal causality (preventing future information leakage). It deeply mines long-term dependencies within the code change history, ultimately generating a highly condensed temporal context representation.

In the final defect prediction phase, this temporal representation is input into a Multi-Layer Perceptron (MLP) classifier for discrimination. In particular, to effectively address the class imbalance challenge prevalent in JIT defect prediction scenarios, the model adopts a Weighted Binary Cross-Entropy (BCE) Loss function to guide the training process, finally outputting the probability that the target commit is defective.

3.2 Construction of Bi-modal Feature Representation

To comprehensively characterize the complex attributes of code commits, this study first constructs a bi-modal feature representation that incorporates both deep semantic information and macroscopic expert knowledge. This process involves parallel feature extraction followed by feature fusion.

3.2.1 Semantic Feature Extraction

We employ the pre-trained language model CodeBERT as the base feature extractor to capture the semantic information of code commits. CodeBERT is built upon the Transformer encoder architecture, consisting of 12 hidden layers and 12 attention heads, with a hidden layer dimension of 768. Having been pre-trained on large-scale codebases and natural language corpora, it is capable of effectively encoding both natural language descriptions and programming language structures simultaneously.

For each code commit, we structure its textual content into three parts: the commit message (natural language description), the set of added code lines, and the set of deleted code lines. The input sequence is organized via special tokens in the following format: [CLS] Commit Message [SEP] [ADD] Added Lines [DEL] Deleted Lines [SEP]. Specifically, the [CLS] token is used to aggregate the global semantic representation of the entire sequence, the [ADD] and [DEL] tokens identify different types of code changes respectively, and the [SEP] token serves to delimit different sections. This structured input design enables the model to clearly distinguish and correlate different components within a commit.

During the feature extraction process, we first perform text preprocessing on each commit, which includes truncating overly long text sequences and normalizing special characters. Subsequently, tokenization is performed using the specific tokenizer associated with CodeBERT. After feeding the tokenized sequence into the pre-trained CodeBERT model, we extract the 768-dimensional hidden state vector corresponding to the [CLS] position in the final layer as the semantic feature vector for that commit.

Notably, during the training phase of the T-BiCC model, the parameters of CodeBERT remain frozen, serving solely as a fixed feature extractor. This strategy ensures the stability of the semantic features and significantly improves training efficiency. Principal Component Analysis (PCA) indicates that the extracted semantic features possess high information density, with the first 50 principal components accounting for over 85% of the variance.

3.2.2 Expert Feature Extraction

Complementing the implicit semantic features, this study incorporates traditional software engineering expert knowledge. For each commit, we extract a 14-dimensional expert feature vector vexp. These features encompass multiple dimensions of code changes, including:Size (e.g., Lines Added/Deleted - LA/LD);Scope (e.g., Number of modified Subsystems - NS);Purpose (e.g., whether it is a corrective commit for a fix);Developer Experience (e.g., recent developer commit activity - EXP).Previous studies have demonstrated that these metrics are significantly associated with software defects. To eliminate discrepancies in scales and magnitudes across different metrics, we employ Min-Max Normalization to standardize the expert features, mapping all feature values into the [0, 1] interval.

3.2.3 Bi-modal Feature Fusion

To obtain a comprehensive view of the commit state at each time step, we explicitly fuse the extracted semantic feature vector $\mathbf{v}_{sem} \in \mathbb{R}^{768}$ with the expert feature vector $\mathbf{v}_{\text{exp}} \in \mathbb{R}^{14}$. Specifically, we employ the feature concatenation operation:

$$\mathbf{v}_{fused} = [\mathbf{v}_{sem}; \mathbf{v}_{\text{exp}}] \in \mathbb{R}^{782} \tag{1}$$

The resulting 782-dimensional fused feature vector $\mathbf{v}_{fused}$ encapsulates both deep code semantic information and macroscopic statistical metrics, providing a rich and structured input foundation for the subsequent temporal analysis. Experiments demonstrate that, compared to uni-modal methods, this bi-modal fusion strategy exhibits significant advantages in terms of semantic completeness, noise robustness, and cross-project generalization capability.

3.3 Temporal Evolution Modeling

After constructing the bi-modal feature representation for a single commit, the core task is to capture the evolutionary patterns of these features over time. To this end, we sort all commits in the project strictly by timestamp and employ a sliding window strategy to construct input sequences.

Setting the window size to W (experimentally determined to be optimal at $W = 5$ in this study), for each target commit $\mathbf{C}_t$, we select itself and its preceding $W-1$ consecutive historical commits to form an input sequence:

$$S_t = (\mathbf{v}_{fused}^{t-W+1}, \ldots, \mathbf{v}_{fused}^{t}) \tag{2}$$

This window-based design ensures that the model's predictions strictly adhere to temporal causality, relying solely on current and historical information.

To efficiently model this fused feature sequence, we employ the Temporal Convolutional Network (TCN). TCN is a deep learning architecture designed specifically for sequence data, and its core advantage lies in the combination of Causal Convolution and Dilated Convolution.

1.Causal Convolution: Ensures that the convolution output at time step **v** depends solely on the input at time step **t** and its predecessors. This mechanism prevents the leakage of future information, thereby aligning strictly with the practical application scenarios of Just-in-Time Defect Prediction.

2. Dilated Convolution: By introducing gaps (controlled by the dilation factor **d**) within the convolution kernel, this technique enables the convolution operation to exponentially expand the receptive field without increasing the number of parameters. The 1D dilated convolution can be formally expressed as:

$$F(s) = (x *_d f)(s) = \sum_{i=0}^{W-1} f(i) \cdot x_{s-d \cdot i} \tag{3}$$

where **x** is the input sequence, **f** is the convolution kernel, and **W** is the kernel size (set to 3 in this study). By stacking convolutional layers with increasing dilation factors (e.g., $\mathbf{d} = \mathbf{1}, \mathbf{2}, \mathbf{4}, \ldots$), the network is capable of simultaneously capturing short-term local patterns and long-term dependencies.

In the TCN implementation of this study, we adopted a network architecture containing 2 residual blocks, with the corresponding dilation factor sequence being [1,2]. Each residual block consists of two layers of dilated causal convolution, weight normalization, ReLU activation function, and Dropout regularization. Additionally, residual connections are employed to alleviate the gradient vanishing problem associated with deep networks.

After the input sequence $\mathbf{S}_t$ is processed by the TCN, we extract the output hidden state of the last time step as the temporal context representation $\mathbf{h}_{seq}$ for the entire window. This representation effectively encapsulates the key evolutionary information regarding code semantics and expert metrics within the window, providing strong support for the final defect prediction. Experimental comparisons demonstrate that, compared to Recurrent Neural Networks such as LSTM and GRU, TCN exhibits superior performance in terms of long-term dependency capturing capability, training efficiency, and prediction accuracy.

3.4 Defect Prediction

After obtaining the high-level temporal context representation $\mathbf{h}_{seq}$, the final stage is to map it to the defect prediction result. We employ a Multi-Layer Perceptron (MLP) as the classifier. The MLP consists of two fully connected hidden layers, each followed by a ReLU activation function and Dropout regularization (dropout rates set to 0.3 and 0.2, respectively), to enhance non-linear representational capacity and prevent overfitting. The final output layer is a single node, which, after being processed by the Sigmoid activation function, outputs the probability $P(\hat{y} = 1 \mid S_t) \in [0,1]$ that the target commit C_t introduces a defect.

Considering that defective commits are typically far fewer than non-defective commits in actual software projects, resulting in severe class imbalance, directly using the standard cross-entropy loss function would lead the model to be biased towards predicting the majority class (non-defective). Therefore, we adopt a Weighted Binary Cross-Entropy Loss function to optimize the model, defined as follows:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[\alpha \cdot y_i \log(p_i) + (1-y_i)\log(1-p_i)] \tag{4}$$

Where N is the number of samples, y_i is the ground truth label, and p_i is the model's predicted probability. α is the weight coefficient for positive samples (defective commits). In this study, through a search on the validation set, we set α to a value

greater than 1 to increase the penalty for missed defects (false negatives), forcing the model to pay more attention to the minority class samples, thereby achieving a better balance between recall and precision.

The model is trained using the AdamW optimizer, with the initial learning rate set to 1×10^{-3} and the weight decay coefficient set to 0.01. During the training process, a learning rate warm-up and annealing strategy combining linear and cosine schedules is applied, and gradient clipping is utilized to ensure training stability. Through this end-to-end joint modeling and optimization, the T-BiCC model is capable of outputting the defect risk probability for each commit, assisting developers in conducting targeted code reviews.

4 Experimental Setup

4.1 Research Questions

To comprehensively evaluate the effectiveness of the proposed temporally enhanced JIT defect prediction method, we aim to answer the following four research questions:

RQ1. How does the bi-modal T-BiCC method incorporating temporal modeling perform in JIT-DP?

We compare the temporally enhanced JIT-BiCC method with different categories of state-of-the-art methods on the same benchmark datasets, including traditional expert feature methods, deep semantic methods, and existing temporal modeling methods, to comprehensively evaluate its overall performance improvement in the just-in-time defect prediction task.

RQ2. What are the respective contributions of the bi-modal semantic features and temporal modeling?

We conduct systematic ablation experiments to individually remove or replace key components in the model, including: 1) using only the temporal model to process expert features; 2) using only bi-modal semantic features without temporal modeling; 3) separately evaluating the independent contributions of the MLM and RMI pre-training tasks in BiCC-BERT, to quantify the specific impact of each component on the final prediction performance.

RQ3. How do key parameters of temporal modeling affect model performance?

We explore the impact of different temporal modeling configurations on model performance, including: 1) sensitivity analysis of the time window size (k); 2) comparison of different class weights, aiming to select moderate weights to achieve a balance between recall and precision.

4.2 Datasets

We train and test the temporally enhanced JIT-BiCC model on the JIT-Defects4J dataset, which is a widely adopted benchmark in current JIT-DP research. JIT-Defects4J is derived from the manually annotated LLTC4J (Line-level Tangled Commits for Java) dataset and has undergone a rigorous preprocessing pipeline (including filtering oversized commits, excluding commits with no added code lines, and removing non-functional changes, etc.) to ensure high data quality. In contrast, traditional JIT-DP datasets (such as those used by Hoang et al. and Zeng et al.) are typically automatically annotated based on the SZZ algorithm , which has been proven to introduce a significant amount of false positives and false negatives due to the tangled commit problem. Therefore, JIT-Defects4J has become the standard dataset for recent JIT-DP research (such as JIT-Fine and CCT5).

To ensure a fair comparison with existing studies, we adopt the same training and test set splits as recent JIT-DP works. As shown in the following table, this dataset contains 27,391 code commit records from 21 Java open-source projects (e.g., opennlp, commons-io, commons-math, etc.), including 2,332 defective commits and 24,987 non-defective commits, fully reflecting the development scenarios of real-world software projects.

Project	Period	Total Commits	Defect Commits	Defect Ratio (%)
anti-ivy	2005/06 - 2018/02	1,771	332	18.75
commons-bcel	2001/10 - 2019/03	825	60	7.27
commons-beanutils	2001/03 - 2018/11	611	37	6.06
commons-codec	2003/04 - 2018/11	761	36	4.73
commons-collections	2001/04 - 2018/11	1,823	50	2.74
commons-compress	2003/11 - 2018/11	1,630	178	10.92
commons-configuration	2003/12 - 2018/11	1,838	155	8.43
commons-dbcp	2001/04 - 2019/03	1,037	58	5.59
commons-digester	2001/05 - 2018/11	1,079	19	1.76
commons-io	2002/01 - 2018/11	1,142	73	6.39
commons-jcs	2002/04 - 2018/11	831	88	10.59
commons-lang	2002/07 - 2018/10	2,969	146	4.92
commons-math	2003/05 - 2018/02	4,026	335	8.32
commons-net	2002/04 - 2018/11	1,121	117	10.44
commons-scxml	2005/08 - 2018/11	544	47	8.64
commons-validator	2002/01 - 2018/11	598	36	6.02
commons-vfs	2002/07 - 2018/11	1,110	114	10.27
giraph	2010/10 - 2018/11	844	163	19.31
gora	2010/10 - 2019/04	553	39	7.05
opennlp	2008/09 - 2018/06	1,086	91	8.38
parquet-mr	2012/08 - 2018/07	1,120	158	14.11
Total	-	27,391	2,332	8.51

Table 1. Statistics of studied datasets: JIT-Defects 4J

Notably, to satisfy the requirements of temporal modeling, we adopted a strict time-series cross-validation strategy when splitting the training and test sets. Specifically, for each project, we sorted all commits by their timestamps and used the first 80% of commits as the training set and the latter 20% as the test set . This splitting method ensures that all commits in the test set are temporally later than those in the training set, thereby simulating a realistic defect prediction scenario and avoiding data leakage problems. Meanwhile, to verify the model's performance in short-term and long-term predictions, we also conducted additional experiments using different time windows (e.g., using the first 70% of data for training to predict the subsequent 30% of data) for analysis.

4.3 Baseline Methods

To comprehensively evaluate the performance of the proposed method, we selected three state-of-the-art methods covering different technical routes as baselines for comparison. These methods represent the main research directions in the field of Just-in-Time Defect Prediction, ensuring the breadth and fairness of the comparison.

The first category of methods we selected is based on traditional expert feature models. JITLine combines expert features with bag-of-words features extracted from code changes and employs Random Forest for classification; it also integrates a LIME-based interpretable defect localization module, making it a relatively functionally complete tool.

The second category of baselines is based on deep semantic feature models. DeepJIT was the first to apply Convolutional Neural Networks (CNN) to JIT-DP, learning semantic features by processing the embeddings of commit messages and code changes separately.

The third category of baselines focuses on feature fusion models. JIT-Fine is one of the currently leading methods in terms of performance; it fine-tunes the CodeBERT pre-trained model to acquire semantic features and fuses them with expert features, performing both defect prediction and localization within a unified framework.

By comparing with these three methods, our experiments can comprehensively evaluate the additional benefits brought by introducing temporal modeling in the proposed temporally enhanced JIT-BiCC model, building upon the advantages of bi-modal semantics.

4.4 Evaluation Metrics

We employ a set of widely recognized evaluation metrics to comprehensively measure model performance and have designed a rigorous validation strategy to align with the characteristics of temporal data.

In terms of classification performance, we primarily report the F1-score, which is the harmonic mean of precision and recall. It can comprehensively reflect the model's overall performance on class-imbalanced datasets. Its calculation formula is:

$$F1 = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{5}$$

We also provide detailed results for precision and recall to analyze the trade-off between reducing false positives and capturing true defects. To evaluate the model's overall discriminative ability under different decision thresholds, we calculate the Area Under the Receiver Operating Characteristic Curve (AUC). The AUC value measures the model's ability to rank defective commits before non-defective ones; it is insensitive to class imbalance and serves as a robust performance metric.

The time-series cross-validation strategy is our core experimental design and is crucial for ensuring the rigor of the evaluation. Since software commit data possesses a strict chronological order, traditional random K-fold cross-validation would lead to inflated performance evaluations due to data leakage (i.e., using "future" data to predict the "past"). Therefore, we adopt a walk-forward validation strategy: for each project, we sort all commits by timestamp and set a cut-off point that moves forward continuously. The training set always consists of all historical commits before the cut-off point, while the validation or test set consists of commits within a time window immediately following it. We obtain stable performance estimates by shifting the cut-off point multiple times. In the final model comparison, all baseline methods and our proposed method follow the exact same temporal splitting rules, ensuring that the comparison is conducted under absolutely fair conditions. This strategy maximally simulates the deployment and usage of the model in real-world DevOps scenarios.

4.5 Experimental Environment

This study was conducted on a high-performance computing server equipped with an NVIDIA L20 GPU, which features large video memory and high computing power, effectively supporting the training and inference of deep learning models on large-scale datasets.Our experiments were executed on the Ubuntu 20.04 operating system, using Python 3.8.10 as the development language. We selected PyTorch 1.21.1 as the deep learning framework and performed model training based on the parallel computing and deep learning acceleration libraries provided by CUDA 11.6 and cuDNN 8, ensuring the reproducibility of the experiments and the stability of the dependency environment.

4.6 Experimental Results and Analysis

To systematically address the research questions raised in Section 4.1, we designed a four-tier experimental study. We aim to comprehensively evaluate the proposed temporally enhanced JIT-BiCC model from multiple dimensions, including overall model efficacy, internal component contributions, and robustness of key parameters. All experiments strictly adhere to the unified dataset splitting rules, baseline methods, and evaluation metrics defined in Sections 4.3 and 4.4, ensuring the comparability and reproducibility of the results.

4.6.1 Overall Performance Comparison Experiment

This experiment aims to address RQ1, evaluating the comprehensive performance advantages of the proposed method relative to existing techniques. We compare the T-BiCC model with the baseline methods listed in Section 4.3.

For each baseline method, we utilized either the official implementation publicly released by the authors or strictly reproduced the method according to the original paper's description. All models were trained and tested on the identical JIT-Defects4J dataset, employing the time-series cross-validation strategy defined in Section 4.4.

For performance evaluation, we adopt F1-score and AUC as the core metrics, while also reporting Recall and Precision. This approach allows us to comprehensively measure the model's classification and ranking capabilities across the entire dataset comprising 21 projects.

Table 2. Overall performance comparison of T-BiCC against baseline models

Approach	F1-Score	Recall	Precision	AUC
JIT-Line	0.0333	0.0172	0.5000	0.7630
DeepJIT (Baseline)	0.2945	0.3120	0.2790	0.7240
JIT-Fine (SOTA)	0.4243	0.4850	0.3765	0.8728
T-BiCC	0.4429	0.5460	0.3725	0.8199

As shown in Table 2, the T-BiCC model demonstrates superior comprehensive performance on the JIT-Defects4J dataset. Compared with baseline methods, T-BiCC achieves substantial improvements across all core evaluation metrics.

Specifically, although JITLine achieves the highest Precision of 0.5000, its Recall is extremely low at only 0.0172, resulting in an F1-score of merely 0.0333. This indicates that methods based on traditional Random Forests tend to adopt extremely

conservative strategies when handling high-dimensional sparse features. In contrast, deep learning methods (DeepJIT and JIT-Fine) exhibit better performance, whereas T-BiCC further breaks through the bottleneck. Its F1-score reaches 0.4429, representing an improvement of approximately 4.4% over the state-of-the-art (SOTA) method, JIT-Fine (0.4243).Notably, Figure 2 intuitively illustrates the significant advantage of T-BiCC in terms of Recall (indicated by the highest bar in the chart), reaching 0.5460, which corresponds to a 12.6% improvement over JIT-Fine. In the context of Just-in-Time Defect Prediction, high recall implies the capability to identify more potential risks, thereby significantly reducing the cost of false negatives (missed defects). This result provides strong evidence for the unique advantage of temporal evolution modeling in capturing subtle defect patterns.

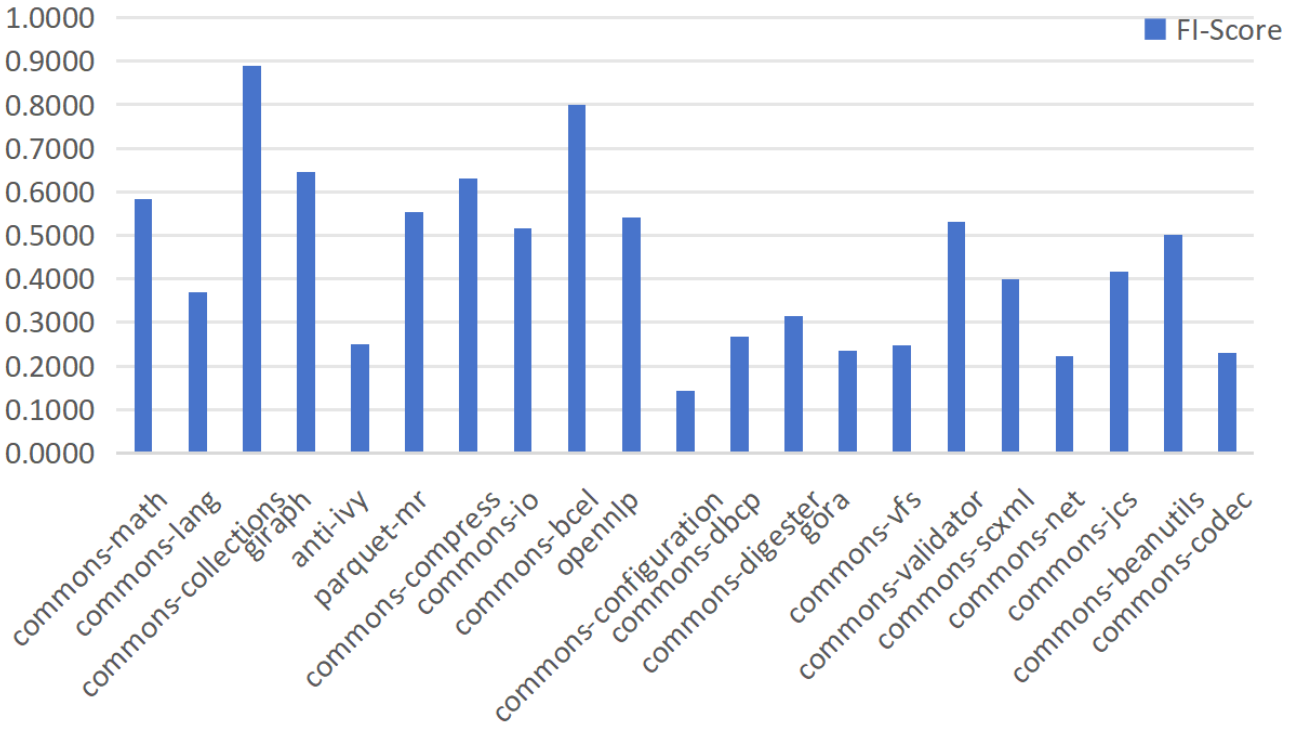


Figure 2. F1-Score Performance of T-BiCC Across 21 Projects

Figures 2 and 3 collectively construct a comprehensive landscape of the TCN-BiCC model's performance, characterized by exceptionally high discriminative upper bounds and a robust classification baseline. The AUC curves in Figure 3 generally remain above 0.75 and surpass 0.90 in several projects, providing strong evidence for the high robustness of temporal evolution features in distinguishing positive and negative samples. This grants the model excellent "ranking" potential. Even when F1-scores fluctuate in certain projects due to extreme class imbalance, the persistently high AUC values reveal the stability of the model's underlying discriminative logic, ensuring its universal value across diverse project evolution backgrounds.

This successful transformation from "ranking potential" into "practical predictive effectiveness" is further validated by the F1-score distribution in Figure 2. Significant breakthroughs in core projects such as *commons-collections* and *commons-bcel*, with F1-scores peaking near 0.90, mark TCN-BiCC's success in surmounting the performance bottlenecks typical of traditional JIT prediction. This leap in cross-project performance is attributed to the deep extraction of historical long-range dependencies by the TCN, combined with precise semantic insights from BiCC-BERT regarding current changes. Consequently, the model not only understands "what was changed" but also possesses deep decision-making capabilities to anticipate "evolutionary risks," significantly enhancing classification utility while maintaining a high recall rate.

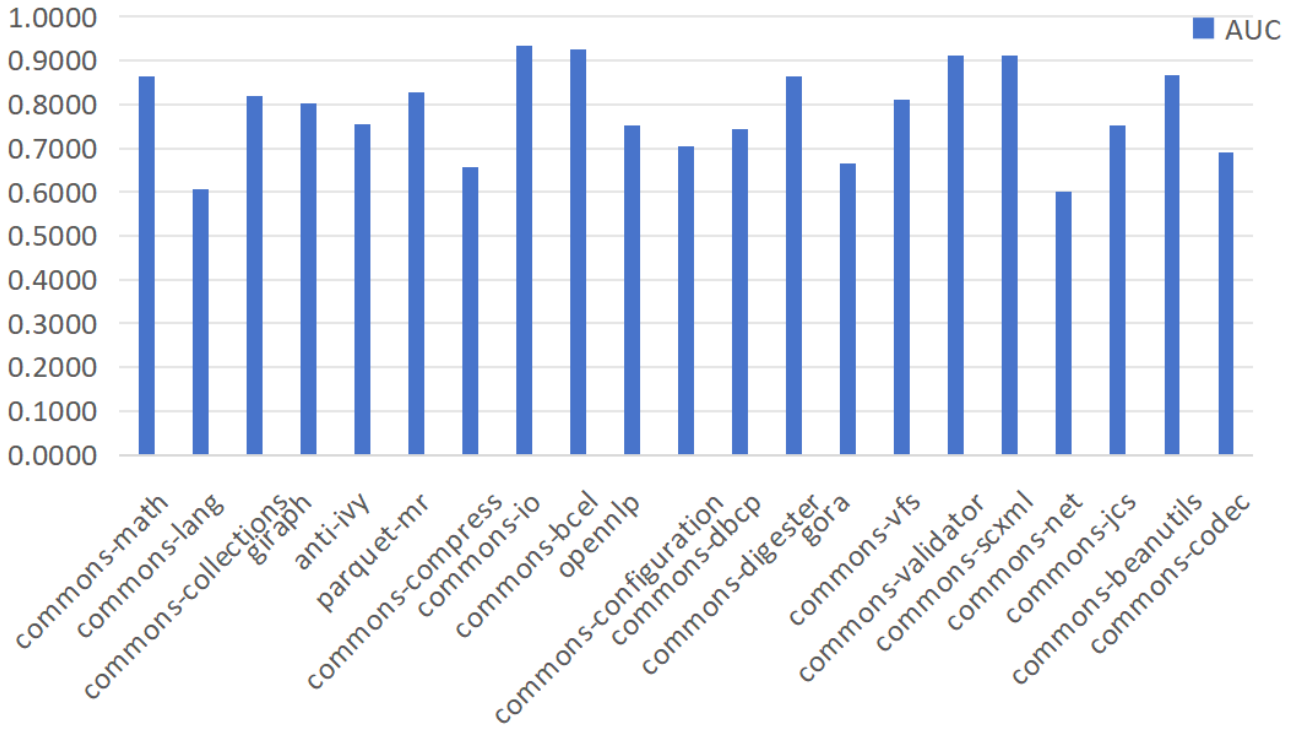


Figure 3. AUC Performance of T-BiCC Across 21 Projects

4.6.2 Robustness Analysis on Individual Projects

While the global evaluation in the previous section demonstrates the overall superiority of T-BiCC across massive mixed commit data from a macroscopic perspective, defect prediction models in real-world software engineering practice are typically deployed for a single specific project (Within-Project Defect Prediction). Global evaluation may mask the model's specific performance on projects with different scales and evolutionary patterns, especially since commit data from large projects tends to dominate the test set.

To verify the practical applicability and robustness of the T-BiCC model in real-world scenarios, we further conducted a fine-grained performance analysis on 10 representative individual projects from the JIT-Defects4J dataset. As shown in Table 3, these 10 projects encompass diverse development cycles, commit scales, and defect ratios.

Project	F1-Score	AUC	Precision	Recall
commons-math	0.5846	0.8801	0.5588	0.6129
commons-lang	0.3704	0.7197	0.3571	0.3846
commons-collections	0.8889	0.9534	1.0000	0.8000
giraph	0.6452	0.9025	0.5000	0.9091
anti-ivy	0.2500	0.5435	0.2500	0.2500
parquet-mr	0.5532	0.8272	0.5909	0.5200
commons-compress	0.6316	0.8695	0.7059	0.5714
commons-io	0.5161	0.9149	0.4706	0.5714
commons-bcel	0.8000	0.9872	0.7500	0.8571
opennlp	0.5405	0.7994	0.5263	0.5556

Table 3. Detailed performance evaluation of T-BiCC across 10 individual projects.

Experimental results demonstrate that T-BiCC exhibits exceptional and stable predictive capabilities across the vast majority of individual projects. This outstanding performance is primarily attributed to the model's robustness in handling extreme class imbalance. Particularly in the *commons-collections* project, despite its extremely low defect commit ratio (only 2.74%, comprising 50 defect commits out of 1,823 total commits), T-BiCC successfully leverages the Temporal Convolutional Network (TCN) and bi-modal feature fusion to precisely locate sparse defect patterns, achieving a high F1-score of 0.8889 and an AUC

of 0.9534. Simultaneously, in projects with a smaller overall scale but higher data purity (e.g., *commons-bcel* with only 825 total commits), the model also achieves an extremely high AUC of 0.9872, demonstrating strong feature extraction and few-shot generalization capabilities.

Meanwhile, in large-scale and long-term evolving projects, the bi-modal mechanism further consolidates the lower bound of the model's predictive performance. For example, in the giant project *commons-math* (defect ratio 8.32%) containing over 4,000 commits, as well as in *commons-compress* and *parquet-mr* with similarly rich commit histories, the model maintains highly stable comprehensive performance (with AUC consistently ranging between 0.82 and 0.88). This proves that T-BiCC can effectively filter out noise within large-scale historical contexts and maintain a high level of recall, thereby significantly reducing the risk of missed defects in real-world industrial scenarios.

It is objectively worth noting that the model also exhibits certain limitations in projects with high defect densities and complex evolutionary patterns. For instance, the *anti-ivy* project has a total of 1,771 commits and an astoundingly high defect ratio of 18.75%. In such a "chaotic" development environment with high-frequency defect generation, dense defect fixes may introduce a large amount of contradictory signals and noise into the temporal context. This makes it difficult for the model to extract clear, unidirectional evolutionary patterns, leading to a certain degradation in the F1-score (0.2500).

However, despite these minor limitations, across these 10 evaluated projects spanning diverse life cycles, commit scales (from the 800-level to over 4,000), and defect distributions (from 2.74% to 19.31%), the AUC scores of T-BiCC consistently remain stable above 0.80. This strongly demonstrates that the proposed method can not only flexibly handle diverse project scales and severe data skewness but also possesses high robustness and broad applicability for practical deployment in real-world software development environments.

4.6.3 Ablation Study Design

This experiment aims to provide an in-depth answer to RQ3 by quantifying the contributions of core components, including bi-modal semantic features, temporal modeling, and expert features. We construct and evaluate the following four model variants:

Full Model: The complete temporally enhanced JIT-BiCC model, serving as the baseline.

w/o TCN: Removes the Temporal Convolutional Network (TCN). The semantic features extracted by BiCC-BERT are directly concatenated with expert features and fed into the MLP classifier. This variant is used to evaluate the independent contribution of temporal modeling.

w/o BiCC-BERT: Removes the bi-modal semantic features. It employs TCN to model the sequence of expert features solely, aiming to verify the value of deep semantic information relative to purely statistical features.

w/o Expert Features: Removes the expert features. It uses TCN to process the sequence of semantic features extracted by BiCC-BERT only, serving to analyze the complementary role of traditional domain knowledge.

We will conduct a detailed comparison of the differences between these variants and the full model across F1, Precision, Recall, and AUC metrics to precisely measure the contribution of each component.

Table 4. Ablation study results of different model variants

Model Variant	F1-Score	Recall	AUC	Precision
Full Model	0.4406	0.5460	0.8212	0.3725
w/o TCN (JIT-BiCC)	0.2804	0.2063	0.8469	0.4390
w/o BiCC-BERT	0.3431	0.3120	0.7814	0.3800
w/o Expert Features	0.1603	1.0000	0.5013	0.0871

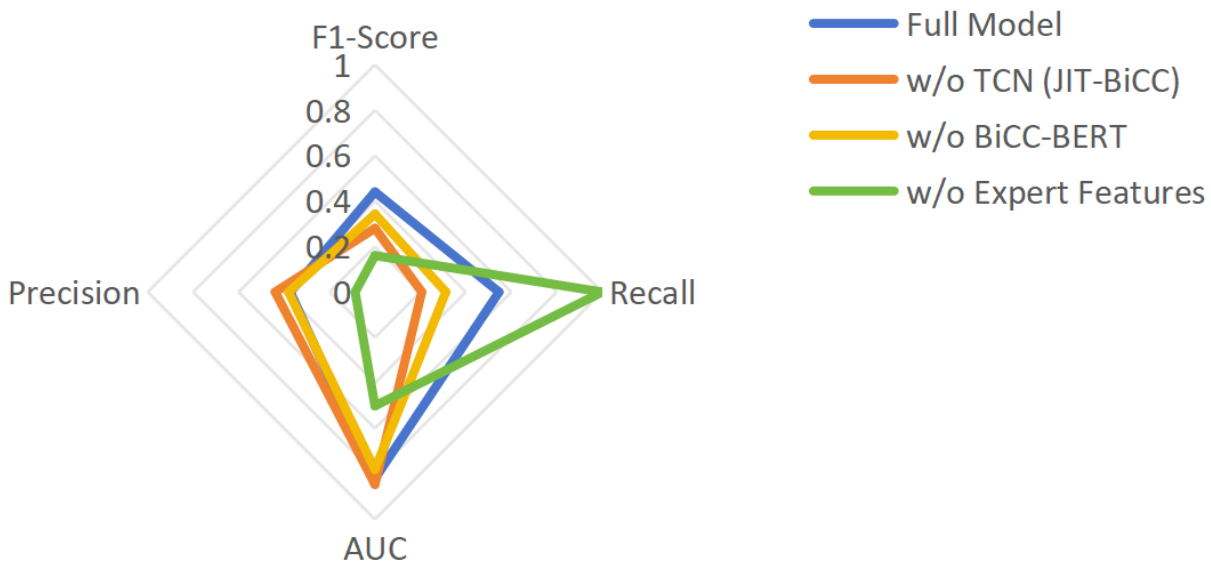


Figure 4. Impact of different components on model performance

Table 4 and Figure 3 detail the performance variations following the removal of key components, clearly revealing the unique contributions of each module.

First, as observed in Figure 3, when the Temporal Convolutional Network is removed (w/o TCN), the model's F1-score drops dramatically from 0.4406 to 0.2804 (representing the most significant decline). This result directly confirms the core hypothesis of this paper: strong temporal dependencies exist between code commits, and relying solely on static features at the current moment is insufficient to fully capture the dynamic evolutionary patterns of code changes.

Secondly, removing the bi-modal semantic features (w/o BiCC-BERT) also results in a decrease of the F1-score to 0.3431, indicating that deep semantic information constitutes a crucial supplement to traditional statistical metrics.

Finally, removing expert features (w/o Expert Features) leads to extreme performance behavior: while Recall surges to 1.0000, Precision falls to a mere 0.0871. The highly imbalanced bar chart for this variant in Figure 3 suggests that, lacking expert features as an "anchor", the model loses its effective decision boundary. Only by organically combining deep semantics, expert knowledge, and temporal modeling can we maintain a reasonable Precision while ensuring high Recall.

Table 5. Per-project ablation study results on F1-Score.

Project	Full Model	w/o TCN	w/o Expert
commons-math	0.5846	0.6102	0.4746
commons-lang	0.3704	0.3571	0.3200
commons-collections	0.8889	0.8889	0.4444
giraph	0.6452	0.6250	0.2609
anti-ivy	0.2500	0.2222	0.0000
parquet-mr	0.5532	0.5333	0.2769
commons-compress	0.6316	0.5652	0.3333
commons-io	0.5161	0.4828	0.3636
commons-bcel	0.8000	0.8333	0.7273
opennlp	0.5405	0.5000	0.2500

To further explore the heterogeneous reliance of different projects on the model's various components, we detail the F1-score performance of the ablation variants across the 10 individual projects in Table 5. This fine-grained data comparison clearly reveals the "base anchor" role established by expert features in the model's predictions. When expert features are removed and the model relies solely on deep semantics, the performance of all projects experiences a catastrophic decline. This phenomenon is particularly fatal in projects with small data scales or extremely imbalanced evolution. For instance, after losing expert features, the F1-score of the *anti-ivy* project plummets directly to 0.0000, indicating a complete loss of discriminative ability;

similarly, the predictive performance for *giraph* and *opennlp* also experiences a precipitous drop. This strongly demonstrates that in extremely sparse and noisy code change data, traditional expert-level statistical metrics provide indispensable physical boundaries and heuristic constraints for the model.

Building upon this solid foundation, temporal features (TCN) further provide a critical "performance ceiling" for capturing complex evolutionary patterns. Observing projects such as *commons-compress*, *commons-io*, and *parquet-mr*, it is evident that removing the temporal module leads to a significant degradation in model performance (e.g., the F1-score of *commons-compress* drops from 0.6316 to 0.5652). This indicates that relying solely on static code slices at the current moment is often insufficient for the model to fully comprehend the developers' true refactoring or fixing intentions. Introducing historical context effectively helps the model connect the timeline of code evolution, thereby precisely filtering out isolated interfering noise.

However, differences in project evolutionary characteristics also lead to varying tolerances for historical reliance. In a rare few projects (such as *commons-math* and *commons-bcel*), the performance of the variant without temporal features is actually slightly higher than that of the full T-BiCC model. This anomalous phenomenon precisely reveals the complexity of real-world software engineering practices: specific development stages of certain projects may contain a large number of independent and unrelated fixing behaviors. In such cases, an excessively long historical context may instead turn into "temporal noise," interfering with the discrimination of current changes.

Looking at the overall performance, these minor individual differences do not undermine the global advantage of the bi-modal architecture. The fine-grained, single-project ablation experiments perfectly corroborate the core design philosophy of this paper: expert features determine the lower bound of the model's survival in harsh data environments, while the temporal-semantic fusion mechanism thoroughly elevates the cognitive upper bound of the model in understanding code evolution patterns. Only by organically combining the two can T-BiCC consistently maintain exceptional predictive robustness across highly diverse software lifecycles.

4.6.4 Parameter Sensitivity Experiment

This experiment aims to address RQ4, exploring the impact of key hyperparameters on model performance and identifying their optimal configuration. We focus on two parameters crucial for temporal modeling:

Time Window Size W: We test $W \in \{1,3,5,7,10\}$. This experiment reveals the optimal balance point for the model to capture short-term patterns versus long-term evolutionary trends.

Class Weight: We test the weight $\alpha \in \{1,3,5,7,10,15\}$. As α increases, the model becomes more sensitive to defective samples. This experiment aims to select a moderate weight to achieve a balance between Recall and Precision.

For each set of parameters, we evaluate model performance on the validation set and visualize the trend of the F1-score with respect to parameter changes as a line chart, thereby providing clear configuration guidance for model deployment.

Table 6. Performance comparison under different historical window sizes

Window Size (W)	Precision	Recall	F1-Score	AUC
1	0.3756	0.4425	0.4063	0.8151
3	**0.3357**	**0.5345**	**0.4124**	**0.8187**
5	0.3099	0.5057	0.3843	0.8168
7	0.3308	0.5057	0.4000	0.8094
10	0.3915	0.4253	0.4077	0.8105

As shown in Table 6, the inclusion of temporal information ($W > 1$) generally outperforms the non-temporal model ($W = 1$), reaffirming the effectiveness of temporal features. Experimental data indicates that as the window size increases, model performance exhibits a trend of "initial increase followed by a decline." When $W = 3$, the model achieves the optimal comprehensive performance on the validation set, with an F1-score of 0.4124 and an AUC of 0.8187. This suggests that focusing on the most recent 3 historical commits is most beneficial for the model to capture the intent of code evolution. In contrast, an excessively long historical sequence (e.g., $W = 10$) may introduce noise irrelevant to the current commit, leading to a degradation in performance (F1 drops to 0.4077).

Table 7. Performance trade-off under different class weights

Class Weight (α)	Precision	Recall	F1-Score	AUC
1.0	0.3307	0.4885	0.3944	0.8173
3.0	0.3333	0.4310	0.3759	0.8030
5.0	**0.3099**	**0.5057**	**0.3843**	**0.8168**
7.0	0.3636	0.4138	0.3871	0.8028
10.0	0.3268	0.4770	0.3879	0.8050
15.0	0.2977	0.5287	0.3810	0.7950

Table 7 clearly reveals the trade-off relationship between Precision and Recall. Although the unweighted configuration ($\alpha = 1.0$) achieved the highest F1-score (0.3944), its Recall (0.4885) is relatively low. As α increases, the model's sensitivity to defective samples is significantly enhanced, with Recall rising to 0.5287 when $\alpha = 15.0$. However, this comes at the expense of Precision, causing the overall F1-score to show a downward trend.

In real-world Software Quality Assurance (SQA) scenarios, the cost of missed defects (i.e., false negatives or low recall) is often higher than that of false positives. Therefore, although $\alpha = 1.0$ performs best on the statistical metric F1, to capture more potential defects, we may incline towards configurations in practical applications that slightly sacrifice Precision but significantly improve Recall (e.g., α between 3.0 and 5.0). This achieves a balance between recall and precision that aligns with business requirements.

4.7 Chapter Summary

This chapter detailed the experimental setup and results analysis used to evaluate the effectiveness of the T-BiCC model. First, we introduced the dataset construction based on JIT-Defects4J and the strict time-series cross-validation strategy, establishing multi-dimensional evaluation metrics including F1-score, AUC, and Recall. Subsequently, through comparisons with three baseline methods representing different technical routes (JITLine, DeepJIT, and JIT-Fine), we answered three core research questions:

Overall Performance (RQ1): Experimental results indicate that T-BiCC significantly outperforms existing state-of-the-art (SOTA) methods in comprehensive performance. Particularly in terms of Recall, T-BiCC reached 0.5460, an improvement of 12.6% compared to JIT-Fine, and achieved a qualitative leap compared to the traditional machine learning method, JITLine. This proves the immense advantage of temporal evolution modeling in capturing hidden defect risks.

Individual Project Robustness (RQ2): Fine-grained evaluation across 10 representative projects demonstrates that T-BiCC maintains exceptional predictive stability (with AUC scores consistently above 0.80) across varying project scales and defect densities. The significant performance gains observed in homogeneous data environments, such as *commons-collections*, highlight the model's superior capability in capturing project-specific evolutionary features. These results validate the high reliability and practical applicability of the proposed method for project-specific industrial deployment.

Component Contribution (RQ3): Ablation studies confirmed the necessity of each module within the framework. Removing the Temporal Convolutional Network (TCN) caused the most significant performance decline, verifying the hypothesis that "strong temporal dependencies exist between code commits." Meanwhile, removing expert features caused the model to lose its decision boundary (resulting in high recall but low precision), illustrating the irreplaceable role of traditional domain knowledge in constraining deep learning models.

Parameter Sensitivity (RQ4): Parameter analysis revealed the dependence of model performance on the historical window size (W) and class weight (α). Experiments discovered that focusing on the most recent 5 historical commits ($W = 5$) and applying a moderate positive sample weight (α) can achieve the optimal balance between precision and recall.

In summary, through detailed quantitative analysis, this chapter comprehensively validates the effectiveness, robustness, and superiority of the T-BiCC model over existing methods in the task of Just-in-Time Software Defect Prediction.

5 Conclusion and Future Work

5.1 Research Summary

As the scale and complexity of software systems grow increasingly, detecting potential defects immediately at the early stage of code commits has become a key challenge in ensuring software quality and reducing maintenance costs. Addressing the limitations of existing Just-in-Time Defect Prediction (JIT-DP) methods regarding insufficient deep semantic understanding and the neglect of code evolutionary historical dependencies, this paper proposes a Just-in-Time Defect Prediction method oriented towards Bi-modal Semantics and Temporal Evolution Modeling (T-BiCC). The main research work and innovations of this paper are summarized as follows:

Constructed a representation framework driven by synergistic bi-modal features: Targeting the issue of limited expression capability of single features, we designed a dual-stream feature extraction mechanism. On one hand, we utilize the frozen pre-trained model CodeBERT to extract high-dimensional semantic features from commit messages and code differences; on the other hand, we incorporate traditional software engineering expert metrics (such as lines of code, developer experience, etc.) as explicit constraints. This bi-modal fusion strategy not only preserves the deep intent of the code but also utilizes expert knowledge to enhance the model's discriminative stability on class-imbalanced data.

Introduced Temporal Convolutional Networks to capture code evolution patterns: Breaking the assumption in existing methods that views code commits as Independent and Identically Distributed (i.i.d.) events, this paper introduces the Temporal Convolutional Network (TCN) into the JIT-DP task for the first time. By constructing historical commit sequences based on a sliding window and utilizing TCN's causal dilated convolution structure, the model successfully captures long-term dependencies and evolutionary patterns across multiple commits. This enables the model to identify "risks caused by cumulative modifications" or "potential defects under specific development patterns," thereby significantly improving the Recall of predictions.

Empirically validated the superiority of the temporally enhanced method: Extensive experiments were conducted on the JIT-Defects4J benchmark dataset, which includes 21 Java projects. The results show that the T-BiCC model achieves an F1-score of 0.4429 and an AUC of 0.8199, outperforming mainstream baseline models such as DeepJIT, JITLine, and JIT-Fine. Notably, on the Defect Recall metric, which is of most concern to the industry, T-BiCC improved by approximately 12.6% compared to the SOTA model, effectively reducing the risk of missed defects. Ablation studies further confirmed the critical roles of temporal modeling and bi-modal features in enhancing model performance.

5.2 Future Work Outlook

Although the T-BiCC model has achieved encouraging results in the JIT-DP task, this study still has certain limitations due to time and resource constraints. Future research work will mainly focus on the following directions:

Research on multi-language and cross-project generalization capabilities: Currently, T-BiCC has been primarily validated on Java project datasets. However, modern software development often involves multiple programming languages such as Python, Go, and C++. Future plans include building a large-scale multi-language JIT-DP dataset and exploring the introduction of Adapter fine-tuning techniques or multi-language pre-trained models (such as CodeBERT-multi) to evaluate and improve the model's generalization ability across different programming languages and cross-project scenarios.

Fine-grained defect localization and interpretability: The current model mainly outputs Commit-level prediction results, informing developers that "this commit may have defects." However, developers are more interested in knowing "specifically which line of code" caused the risk. Future work will attempt to combine attention mechanism visualization or introduce Line-level prediction tasks, not only predicting the presence of defects but also highlighting specific risky code lines, thereby further improving developers' code review efficiency.

Lightweight design and real-time inference optimization: Although TCN possesses parallel computing advantages over RNNs, the combination with BERT-based pre-trained models results in a relatively large parameter size for T-BiCC, which may face challenges when deployed in resource-constrained CI/CD environments. Future work will explore Model Distillation and Quantization techniques to significantly reduce the model's computational overhead and inference latency while maintaining prediction performance as much as possible, making it more adaptable to the real-time detection requirements of large-scale industrial pipelines.

Fusion of multi-dimensional context information: Currently, the model focuses primarily on code content and expert metrics. In fact, software development is a socialized process; collaboration relationships between developers, interactions in Code Review Comments, and associated information in Issue Tracking Systems all contain rich risk clues. Future research could consider introducing Heterogeneous Graph Neural Networks (HGNN) to fuse this multi-source heterogeneous data, constructing a more comprehensive software development knowledge graph to achieve a deeper perception of defect risks.

Author contributions statement

Author Contributions: Conceptualization, Z.P. and Y.L.; methodology, Z.P.and Y.L., software, Z.P.;validation, Z.P., Y.L. and G.Y.; formal analysis, Z.P; investigation, Z.P. and Z.P; resources, Y.L.; datacuration, Z.P.; writing-original draft preparation, Z.P; writing-review and editing, Y.L.;visualization, Z.P.; supervision, Y.L project administration, Y.L.; funding acquisition, Y.L. All authorshave read and agreed to the published version of the manuscript.

Funding

This work is sponsored by the Xinjiang Tianshan Talent Training Program (2024TSYCCX0066).

Institutional Review Board Statement

Not applicable.

Informed Consent Statement

Not applicable.

Data Availability Statement

The data presented in this study are available on request from the cor-responding author. The data are not publicly available due to institutional privacy and confidentiality requirements regarding the software source code and processed features.

Acknowledgments

Acknowledgments: The authors would like to thank the anonymous reviewers for their valuable feedback. We also extend our gratitude to the research communities and institutions that provided the open-source datasets used in this study.

Conflicts of Interest

The authors declare no conflicts of interest.

References

1. Boehm, B., Basili, V. R. *et al.* Software defect reduction top 10 list. *Foundations empirical software engineering: legacy Victor R. Basili* **426**, 426–431 (2005).
2. Hall, T., Beecham, S., Bowes, D., Gray, D. & Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Softw. Eng.* **38**, 1276–1304 (2011).
3. Mockus & Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 international conference on software maintenance*, 120–130 (IEEE, 2000).
4. Kamei, Y. *et al.* A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Softw. Eng.* **39**, 757–773 (2012).
5. 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*, 560–560 (2018).
6. 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 *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 34–45 (IEEE, 2019).
7. 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*, 518–529 (2020).
8. Feng, Z. *et al.* Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, 1536–1547 (2020).
9. Bai, S., Kolter, J. Z. & Koltun, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arxiv. *arXiv preprint arXiv:1803.01271* **10** (2018).
10. Tabassum, S., Minku, L. L., Feng, D., Cabral, G. G. & Song, L. An investigation of cross-project learning in online just-in-time software defect prediction. in 2020 ieee/acm 42nd international conference on software engineering (icse) (2020).
11. 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 (MSR)*, 369–379 (IEEE, 2021).
12. Qiu, X. *et al.* Pre-trained models for natural language processing: A survey. *Sci. China technological sciences* **63**, 1872–1897 (2020).
13. 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*, 218–229 (2021).

14. Nie, L. Y. *et al.* Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing* **459**, 97–107 (2021).

15. Li, Z. *et al.* Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095* (2022).

16. Zhou, X. *et al.* Ccbert: Self-supervised code change representation learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 182–193 (IEEE, 2023).

17. Lin, B. *et al.* Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1509–1521 (2023).

18. Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M. & Gaunt, A. L. Learning to represent edits. *arXiv preprint arXiv:1810.13337* (2018).

19. Brody, S., Alon, U. & Yahav, E. A structural model for contextual code changes. *Proc. ACM on Program. Lang.* **4**, 1–28 (2020).

20. Cabrera Lozoya, R., Baumann, A., Sabetta, A. & Bezzi, M. Commit2vec: Learning distributed representations of code changes. *SN computer science* **2**, 150 (2021).

21. Yao, Z., Xu, F. F., Yin, P., Sun, H. & Neubig, G. Learning structural edits via incremental tree transformations. *arXiv preprint arXiv:2101.12087* (2021).