Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.058779

ARTICLE

Telecontext-Enhanced Recursive Interactive Attention Fusion Method for Line-Level Defect Prediction

Haitao He[1], Bingjian Yan[1], Ke Xu[1,*] and Lu Yu[1,2]

[1]School of Information Science and Engineering, Yanshan University, Qinhuangdao, 066000, China

[2]Hebei Port Group Co., Ltd., Tangshan, 063000, China

*Corresponding Author: Ke Xu. Email: xuke_kara@163.com

Received: 20 September 2024 Accepted: 04 November 2024 Published: 17 February 2025

ABSTRACT

Software defect prediction aims to use measurement data of code and historical defects to predict potential problems, optimize testing resources and defect management. However, current methods face challenges: (1) Coarse-grained file level detection cannot accurately locate specific defects. (2) Fine-grained line-level defect prediction methods rely solely on local information of a single line of code, failing to deeply analyze the semantic context of the code line and ignoring the heuristic impact of line-level context on the code line, making it difficult to capture the interaction between global and local information. Therefore, this paper proposes a telecontext-enhanced recursive interactive attention fusion method for line-level defect prediction (TRIA-LineDP). Firstly, using a bidirectional hierarchical attention network to extract semantic features and contextual information from the original code lines as the basis. Then, the extracted contextual information is forwarded to the telecontext capture module to aggregate the global context, thereby enhancing the understanding of broader code dynamics. Finally, a recursive interaction model is used to simulate the interaction between code lines and line-level context, passing information layer by layer to enhance local and global information exchange, thereby achieving accurate defect localization. Experimental results from within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) conducted on nine different projects (encompassing a total of 32 versions) demonstrated that, within the same project, the proposed methods will respectively recall at top 20% of lines of code (Recall@Top20%LOC) and effort at top 20% recall (Effort@Top20%Recall) has increased by 11%–52% and 23%–77%. In different projects, improvements of 9%–60% and 18%–77% have been achieved, which are superior to existing advanced methods and have good detection performance.

KEYWORDS

Line-level defect prediction; telecontext capture; recursive interactive structure; hierarchical attention network

1 Introduction

As software projects grow in scale and the complexity of modifications increases, the timely identification and correction of software defects become increasingly critical to maintaining software quality and minimizing maintenance costs. If defects cannot be detected and corrected promptly, they may have a serious impact on product quality [1]. In this context, software flaw forecasting

has emerged as a potent tool for software quality assurance, which can help quality assurance teams identify potential problems in the early stages of development, optimize resource allocation, and ensure that testing work is targeted and efficient. By analyzing historical data, researchers design metrics that are strongly correlated with software defects. They then utilize statistical or artificial intelligence methods to construct defect forecasting frameworks to predict the defect tendency and severity, or the distribution of defects across software modules, ultimately enhancing the robustness of the software [2].

In recent years, scholars have introduced approaches for predicting software defects that operate across multiple levels of granularity, addressing different aspects of software reliability and maintenance, which are divided based on different levels of software structure, from advanced file level [3] to more refined method level [4] and statement level [5] analysis. Xu et al. [6] used graph neural networks to capture sub tree feature information with potential defects in abstract syntax trees, and combined semantic information and contextual information learned based on abstract syntax tree (AST) representation to obtain defect feature information. Later, Uddin et al. [7] used a bidirectional long-short-term memory network (BiLSTM) to learn the contextual information in the embedded token vector represented by the source code using the bidirectional encoder representations from transformers (BERT) model, in order to capture the semantic features of the code and preserve the semantic and contextual information of the source code. Abdu et al. [3] fully utilized features extracted from different source code perspectives, using gating mechanisms to combine syntax level features extracted from AST and semantic graph features extracted from control flow graph and data dependency graph for defect prediction. Zhao et al. [8] proposed a multi stream graph neural network model that combines AST and control flow graph (CFG) with data flow extended control flow graph (ECFG) for defect prediction, utilizing the context dependency relationship and syntax structure of the code. However, they generally face the problem that the prediction granularity is too coarse [9] to directly point out the specific location of the defective code for developers.

To enhance prediction accuracy, the researchers further explored fine-grained defect prediction methods. Shao et al. [10] used association rule mining techniques to simplify association rules by considering only one feature and one class label, making the model easier to interpret and understand. Yang et al. [4] further refined the granularity of defect prediction, down to the method level, generating a method call sequence containing code context and semantic information. They used a transformer model to map the method call sequence to a low dimensional vector space, extracting semantic and syntactic features. Majd et al. [5] designed a set of statement level metrics consisting of 32 items, such as the number of times unary and binary operators are used in a statement. Then, a long-short-term memory (LSTM) network is used to learn and process sentence level features to identify sentence level defects.

These methods predict defects by analyzing method and statement level features. Although finer grained code representation improves the accuracy of defect prediction, it still cannot accurately locate specific lines of code. This means that developers need to invest extra time reviewing files one by one and locating erroneous lines of code, greatly reducing review efficiency.

Therefore, researchers have further explored row level defect prediction techniques, aiming to improve code review efficiency, more accurately locate code errors, and predict potential defect risks. Wattanakriengkrai et al. [11] used LIME technology to identify risk markers, enabling file level models to predict defect lines in code. Zhu et al. [12] combined the BiLSTM model with extended code syntax information to predict row level defects. The DeepLineDP model proposed by Pornprasit et al. [13] utilized a bidirectional gated recurrent unit (Bi-GRU) network and attention

mechanism to estimate the defect probability of code lines from defective files. These advanced methods have made significant progress in line level defect prediction, but there are still shortcomings in gaining a deeper understanding of the connections and interactions between code lines and their contexts. The existing row level defect prediction methods still need improvement when dealing with complex code structures and contextual relationships.

In order to provide a clearer explanation of the issues that may arise when a line of code lacks a global context, as well as the interaction between the line of code and its local context, we have provided a simplified example in the figure as an illustration. Fig. 1 shows an example of motivation in our research, with code snippets sourced from the dataset used in this study. The code in Fig. 1a contains a flaw on Line 6: when returning null, it may cause a NullPointerException. To address this issue, the fix code in Fig. 1b modifies it to return a Field array of length 0. Through this example, we can draw the following two important observations: **Observation 1:** The occurrence of defects depends on specific calling contexts. The code in Fig. 1a only returns null when dataList.size()==0, without considering that the caller may not have performed a null value check. Developers often assume by default that the fetchData method will not return null. Especially when the call to the fetchData method occurs outside of the DataManager class, the caller often ignores the null check on the return value. This indicates that the source code model should not be limited to a local single granularity, especially in cross method or cross module calls, where the potential risk of defects can be hidden due to context. **Observation 2:** Although these two code snippets have slight differences in details, their semantics are significantly different. As shown in Fig. 1, the difference lies in whether the return value on the 7th line chooses to return NOFINELDS or null. In Fig. 1b, since the reference to NODATA is an object (as shown in Line 2 of Fig. 1b), it does not trigger an exception, while the null in Fig. 1a may cause the caller to encounter a NullPointerException. Therefore, this distinction actually reflects the difference between objects and null. If you only look at the local code and ignore the global context (such as how the caller handles the return value), hidden defects may be overlooked. In this case, local code returning null or NO-DATA can cause different behaviors at the caller and may even lead to errors. Therefore, only by analyzing the lines of code while considering the global context can we truly understand the impact of these changes on system behavior. The interaction between global and local is crucial in locating and resolving potential defects.

```
1 class DataManager {
2     public final Data[] fetchData(String key) {
3         List<Data> dataList = new ArrayList<>();
4         // ...
5         if (dataList.size() == 0)
6             return null;
7         // ...
8     }
9 }
```

(a) The defect version of Java

```
1 class DataManager {
2     private final static Data[] NO_DATA = new Data[0];
3     public final Data[] fetchData(String key) {
4         List<Data> dataList = new ArrayList<>();
5         // ...
6         if (dataList.size() == 0)
7             return NO_DATA;
8         // ...
9     }
10 }
```

(b) The fixed version of Java

Figure 1: The motivation example studied in this article: comparison between code snippet (a) and code snippet (b)

In response to the above issues, this paper proposes a telecontext-enhanced recursive interactive attention fusion method TRIA-LineDP for line-level defect prediction. Specifically, the initial step

involves utilizing a pre-trained word to vector (Word2vec) model to transform the code file into a vector representation. This process effectively extracts the semantic content embedded within the code. At the same time, Bi-GRU is used to extract the code line feature information and context feature information. Subsequently, the telecontext capture module is employed to capture remote interaction information of line-level context based on content and location across multiple dimensions. This module effectively captures remote dependencies between lines of code, addressing the limitation of relying solely on information from neighboring lines and neglecting remote interactions. Finally, by using a recursive interaction structure to fuse and pass heuristic contextual information layer by layer, combined with an attention mechanism, different importance weights are assigned to each line of code, effectively capturing important global and local dependencies, and obtaining enhanced representations for each line of code, thereby improving the recognition accuracy of defective code lines. On the test dataset, the model can be used to obtain the probability of defects in each line of code and code file, and the performance of the model can be evaluated using traditional metrics such as Area Under the Curve (AUC), Balanced Accuracy (BA), Matthews Correlation Coefficient (MCC) and Geometric Mean (GM). In addition, by ranking code lines using defect probability, three key metrics can be used to evaluate model performance: Recall@Top20% proportion of defects identified within the first 20% of LOC measurement code lines to the total number of defects, Effort@Top20%Recall and evaluate the effort required to achieve the goal Recall@Top20%LOC and Initial False Alarm (IFA) are used to check frequency analysis. The experimental results demonstrate that the TRIA-LineDP method outperforms other leading defect prediction techniques, showcasing superior performance across several key metrics. Specifically, for file-level defect prediction, the TRIA-LineDP method achieves higher scores in the AUC (Area Under the Curve), MCC (Matthews Correlation Coefficient), BA (Balanced Accuracy) and GM (Geometric Mean) indicators. Additionally, at the row level, it excels in the Recall@Top20%LOC, Effort@Top20%Recall and IFA metrics, further underscoring its effectiveness in identifying defects efficiently and accurately.

The main contributions of this paper are as follows:

1. The telecontext capture module was designed, and through the remote interactive attention mechanism based on content and location, it is employed to enhance the global aggregation capability of line-level contexts, thereby mitigating the issue of remote dependencies within the code.
2. A recursive interaction structure is designed, which is passed and fused layer by layer through heuristic effects between lines of code and line-level contexts, to integrate line-of-code information with line-level contextual information, with a special focus on global and local dynamic interactions between lines of code and their contexts. This approach enables deeper mining and expression of core features.
3. A novel code-based telecontext-enhanced recursive interactive attention fusion method TRIA-LineDP, is proposed for line-level defect prediction. This method combines a telecontext capture module and recursive interaction structure, which can effectively capture and fuse global and local interaction dependencies in the code.
4. Experiments were carried out on multiple open-source benchmark projects, and the proposed method was compared with other advanced defect prediction techniques under both intra- and inter-project conditions. The results indicate that the TRIA-LineDP method delivers good prediction performance in defect prediction tasks.

The organizational structure of this paper is outlined as follows: Chapter 2 analyzes the current related work. Chapter 3 provides a detailed introduction to the structure of TRIA-LineDP and the

functions of its modules. Chapter 4 introduces the experimental procedure and analyzes the results. Finally, Chapter 5 summarizes the content of this article.

2 Related Work

Software defect prediction is an important research field in software engineering, with the main purpose of helping development teams effectively allocate testing resources and improve software quality and reliability. Scholars have proposed defect detection methods for different granularities, which are divided according to different levels of software structure, from high-level file level [3] to more refined method level [4] and statement level [5] for analysis. File level defect prediction is one of the earliest and most widely researched methods in this field. Due to its fundamental role in understanding and identifying potential defects in software systems, file level methods have received widespread attention. This method predicts whether a file contains defects by analyzing file level features such as code complexity, annotation density, modification history, etc.

File-level defect prediction represents one of the earliest and most extensively researched approaches within the field of defect prediction. This method has garnered significant attention due to its foundational role in understanding and identifying potential defects across various software systems. The file-level approach predicts whether a file contains defects by analyzing file-level characteristics such as code complexity, comment density, modification history, and so on. Xu et al. [6] proposed a method of integrating semantic and contextual information to identify software defects, combined with the use of AST to represent learning and graph neural networks to capture subtrees with potential defect information (GNNs-DP). Through this method, effective semantic and contextual information was extracted from the source code. Uddin et al. [7] used embedded label vectors learned from the BERT model to process contextual information and identify key features in nodes (SDP-BB) based on a bidirectional long short-term memory network (BiLSTM). This model preserves the hierarchical structure and syntax relationships of the code, thereby improving the performance of code feature representation. Abdu et al. [3] proposed a deep level convolutional neural network (DH-CNN) that combines feature representations extracted from different source code perspectives. word to vector (Word2vec) and node to vector (Node2vec) techniques are respectively used to extract syntax level features from abstract syntax trees (ASTs) and semantic graph features from Control Flow Graphs and Data Dependency Graphs for defect prediction. Liu et al. [14] using node features and basic path features extracted separately from the program dependency graph, highly correlated multi feature fusion (MF) is integrated with relevant contextual information, and an optimized support vector machine (SVM) algorithm is used to design a multi feature labeling method specifically for identifying high-risk defects. Zhao et al. [8] proposed a multi flow graph neural network model that combines AST and CFG with data flow for defect prediction (MFGNN). In the outer layer, an inter process extended control flow graph (ECFG) is used to depict the dependencies between basic blocks. In the inner layer, the structure of each basic block is represented by an Abstract Syntax Tree (AST). However, File level defect prediction can only indicate to developers which files may have defects, but they still need to further browse the entire file to locate specific lines of defective code, which invisibly increases the workload and reduces the efficiency of code review. Therefore, it is crucial to study more refined line level defect prediction in order to improve the accuracy and efficiency of code review.

With advancing research, defect prediction granularity has been refined to class and method levels. Methods at these levels use static attributes, such as the Chidamber and Kemerer Metrics Suite (CK) [15] and Metrics for Object-Oriented Design (MOOD) [16], primarily for class-level defect

prediction. For method-level prediction, metrics like McCabe Cyclomatic Complexity (McCabe) [17] and Halstead Complexity Measures (Halstead) [18] are widely applied. Shao et al. [10] used atomic association rule mining (ACAR) to explore attribute-category relationships, demonstrating that association rules contribute significantly to defect prediction. Yang et al. [4] further refining the granularity of defect prediction to the method level, a method call sequence defect prediction method based on code context and semantic information is proposed. And use transformer to build a defect prediction model that generates semantic and syntactic structural features (TSASS). Majd et al. [5] designed a set of statement level indicators consisting of 32 different indicators was designed, covering elements such as the frequency of use of unary and binary operators in statements. Then, a Long Short Term Memory (LSTM) network was used to learn and process statement level static code features, constructing a statement level deep learning model to identify statement level defects (SLDeep). Although these prediction methods are more granular than at the file level, they still fail to pinpoint specific defective lines of code. Developers still need to further search for potential defective code lines within the scope of methods or classes, which increases additional workload and affects the efficiency of defect repair. Therefore, it is crucial to develop more fine-grained defect prediction methods, especially those that can directly identify specific defect lines of code.

Recent studies have shifted focus towards row-level defect prediction, aiming to enhance the precision of defect detection through more detailed analysis. Wattanakriengkrai et al. [11] proposed a new line level defect prediction framework (Line-DP) using model independent techniques (LIME). The LIME method is used to calculate the LIME score for each label in the defect file, and labels with positive scores are considered risk labels. The lines are ranked based on the score to identify high-risk lines. However, as a local interpretation model, LIME may overlook global features and the contextual relationships between lines. Mahbub et al. [19] proposed a hierarchical encoder structure was proposed to explore code defects (a.k.a. bug) and capture code context for row level defect prediction (Bugsplorer), emphasizing the importance of contextual information. Zhu et al. [12] combined a BiLSTM model with extended syntax information for improved line-level prediction (SyntaxLineD), leveraging bidirectional LSTM to capture code dependencies. While BiLSTM recognizes nearby dependencies effectively, it struggles with long-range dependencies and capturing global context. Wang et al. [20] identified bug lines by using an n-gram model to detect unnatural tags (Bugram), but its fixed-length context limits its ability to capture extended dependencies. Thus, some defect features that require a broader context may be missed. In addition, Pornpraset et al. [13] developed the DeepLineDP model, which uses a Bi-GRU to capture adjacent tokens and line-level context, employing attention to calculate vulnerability scores for each line. However, DeepLineDP focuses on a single level of context, limiting dynamic interactions between code lines and their context.

While various methods exist for line-level defect prediction, most rely solely on a single layer of local information from individual code lines and do not fully utilize the contextual interactions between lines or a comprehensive consideration of global and local information, limiting prediction accuracy. The approach in this research effectively identifies defective lines by analyzing both global and local interaction features within the code. Its core advantage lies in capturing dynamic interactions between code lines and their context, allowing for a deeper representation of code features. Table 1 presents a summary and comparison of relevant software defect prediction methods.

Table 1: Summary of related software defect prediction approaches

Ref. No.	Published year	Approach name	Advantages	Disadvantages
Xu et al. [6]	2020	GNNs-DP	By combining AST and GNN, the model can learn latent defect information from defective subtrees and dynamically adjust according to the repaired changes, enhancing the adaptability of learning.	The coarse-grained prediction of file level defects limits its ability to locate specific defect codes, resulting in lower repair efficiency.
Uddin et al. [7]	2022	SDP-BB	Using BiLSTM and embedded token vectors learned based on BERT model to process contextual information can better capture the semantic and contextual information of the source code.	
Abdu et al. [3]	2024	DH-CNN	Multiple source code representations compensate for the limitations in software defect prediction caused by a single code representation.	
Liu et al. [14]	2023	MFSVM	By fusing multiple strongly correlated features and using label generation methods, the curse of dimensionality that may arise from multi feature fusion has been solved.	

(Continued)

Table 1 (continued)

Ref. No.	Published year	Approach name	Advantages	Disadvantages
Zhao et al. [8]	2022	MFGNN	Introducing the syntactic structure of basic blocks, namely their corresponding AST, can provide a more informative representation of basic blocks.	
Shao et al. [10]	2018	ACAR	Using Atomic Association Rule Mining (ACAR) to explore the relationship between attributes and categories, and improve the prediction of defect prone modules.	Although these prediction methods are more detailed than file level methods, they still cannot accurately locate specific lines of defective code.
Yang et al. [4]	2022	TSASS	By focusing on method call sequences and preserving the contextual structure of the code, this model is able to capture finer grained program behavior and potential defects.	
Majd et al. [5]	2020	SLDeep	SLDeep opens new avenues for applying and enhancing deep learning models in software defect prediction, offering insights for defining finer-grained statement level code metrics across languages.	

(Continued)

Table 1 (continued)

Ref. No.	Published year	Approach name	Advantages	Disadvantages
Wattana-kriengkrai et al. [11]	2020	Line-DP	The use of explanatory LIME technology makes this method independent of specific defect prediction models and has higher applicability.	LIME is a local interpretation model that may not fully capture global characteristics when approximating local models, and may overlook contextual relationships between code lines.
Mahbub et al. [19]	2024	Bugsplorer	Utilizing two Transformer models in a hierarchical structure can better capture local contextual information of software defects.	The complex training process requires adjusting multiple hyperparameters, resulting in insufficient generalization ability.
Zhu et al. [12]	2023	Syntax LineD	Consider the code line level representation of syntax node coverage, add syntax nodes to their corresponding coverage lines, and fully utilize the syntax information related to code lines.	Although BiLSTM can capture before and after dependencies, it still has limitations in capturing long-range dependencies and global contextual information.

(Continued)

Table 1 (continued)

Ref. No.	Published year	Approach name	Advantages	Disadvantages
Wang et al. [20]	2016	Bugram	Identify defective code tags based on the probability estimated by the n-gram model to alleviate the problem of relying on frequent patterns.	The n-gram model captures only fixed-length context and misses long-distance dependencies.
Pornpraset et al. [13]	2022	DeepLineDP	Consider utilizing contextual information from surrounding tags and lines of code in the code.	DeepLineDP only focuses on a single level of contextual information, making it difficult to effectively capture contextual semantics and local interactions between code lines.

3 Mehod

In this section, we introduce TRIA-LineDP, a hierarchical attention fusion method based on telecontext-enhanced recursive interaction. As shown in Fig. 2, it consists of five steps: (1) Source code preprocessing and embedding; (2) Line of code and line level contextual feature extraction; (3) Telecontext capture module enhances global context aggregation; (4) Recursive interaction fusion feature construction; (5) Defect prediction at the line level.

3.1 Source Code Preprocessing and Embedding

Preprocessing is essential for applying deep learning to software defect prediction effectively. Efficient code preprocessing removes redundant, non-logical information and reduces interference, enabling the model to focus on structural and semantic features, thereby enhancing defect prediction accuracy [21]. To streamline model input and focus on core code semantics, we implemented preprocessing steps outlined in Table 2. This typically includes formatting, cleaning, and transforming code for further processing. Preprocessed code better aligns with the model input requirements, as irrelevant information that could create noise is removed-such as special characters and blank lines.

To boost generalization, we replace string constants and numbers with universal tags String (<STR>) and Number (<NUM>). A pre-trained Word2Vec model using the Continuous Bag of Words (CBOW) algorithm is employed to capture features related to code defects. The CBOW model generates vector representations of tokens based on surrounding context, effectively capturing token relationships within code. Customized language models for each project are trained with the Gensim

library. To maintain code structure, each file is converted into a sequence of code lines, represented as $L = [l_1, l_2, \ldots, l_n]$, where l_i is the embedded vector for the i-th line.

Table 2: Source code preprocessing and embedding measures

Category	Method	Description
Preprocessing process	Formatting code	Remove comments and blank lines: Remove comments (single-line comments and multi-line comments) and unnecessary blank lines from the code.
	Cleanup code	Delete useless code: Remove variables, functions, or code snippets that are no longer in use.
	Convert code	1. Split lines of code: Split complex multi-line code into simple single-line code for easy processing in the future. 2. Generate code tags: Convert code into easily analyzable tag forms, such as tagging code as functions, variables, operators, etc.
Specific measures	Remove special characters	Remove all special characters such as ”, ’()’, ’,’, ’;’, etc.
	Remove blank lines	Remove blank lines.
Improve generalization ability	Replace string constants and numbers in the code	Use String (<STR>) and Number (<NUM>) universal tags instead of string constants and numbers.
Token vector representation	Using pre-trained Word2Vec models	Choose the Continuous Bag of Words (CBOW) model in Word2Vec as the training algorithm.

3.2 Line of Code and Line Level Contextual Feature Extraction

To accurately extract code lines and contextual features, this section uses a Bi-GRU network to capture the before and after dependencies and contextual information in the code lines. The detailed implementation is illustrated in Algorithm 1.

Algorithm 1: Bidirectional GRU-based feature extraction

Input: Code file f with lines $L = [l_1, \ldots, l_n]$; tokens $T = [t_1, \ldots, t_m]$; embedding matrix W
Output: Line representation $V_l = [h_{l_1}, \ldots, h_{l_n}]$; context sequence $H_l = [h_{l_1}, \ldots, h_{l_n}]$

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to len(T) **do**
3: $E_i = [W[t_{i1}], \ldots, W[t_{ij}]]$
4: **end for**
5: **end for**
6: **for** $i = 1$ to n **do**
7: $h_{ij} = \text{concat}(\text{forward\_GRU}\ (E_i), \text{backward\_GRU}\ (E_i))$
8: **end for**
9: **for** $j = 1$ to len(h_{ij}) **do**
10: $\alpha_{ij} = \text{attention\_weight}(h_{ij})$
11: V_l.append(sum($\alpha_{ij} \times h_{ij}$))
12: **end for**
13: **for** $i = 1$ to n **do**
14: H_l.append(concat(forward_GRU (V_l), backward_GRU (V_l)))
15: **end for**
16: **Return** V_l, H_l

The input is a source code file that contains a series of code line representations and a word embedding matrix, while the output includes a sequence of code lines and line level contextual representations. Lines 1–5 iterate through each token in a code line, obtaining its vector representation from the embedding matrix. Lines 6–8 use forward and backward gated recurrent unit (GRU) to process these token vectors, generating different hidden states, and then connecting these two hidden states to capture dependencies. Lines 9–12 apply attention mechanisms to hidden states to focus on important markers, thereby generating vector representations of code lines. Lines 13–15 use bidirectional GRU to further process the code line sequence and generate line level contextual representations. This algorithm effectively captures code lines and their contextual information, providing a solid foundation for code analysis and understanding.

3.3 Telecontext Capture Module Enhances Global Context Aggregation

Bidirectional GRU processes sequences based on local context, capturing dependencies only within limited windows. This restricts its ability to utilize global context and limits its focus to the preceding and following time steps, making it challenging to capture long-range dependencies and potentially missing important contextual details. To address these limitations, we propose a remote context capture module. This module employs a multi-head attention mechanism to achieve remote interactions through both content-based and location-based interactions.

Content based interaction. The row-level context is aggregated as a global background vector, linearly projected to calculate the key (K) and value (V). Using the normalized key ($\overline{K}$) and value V, the content-based interaction vector H_c is derived.

Location based interaction. While content-based interaction considers context, it overlooks the relative positions within the sequence, lack mechanisms to represent data order. To address this, the module introduces positional embeddings to represent data order, allowing structured interactions between query and context. This captures relationships between both adjacent and distant positions

in the sequence, enabling global content and position interactions. A matrix R for relative positional embeddings encodes all possible relative positions (n, m), which is then remapped to a three-dimensional tensor P. The positional interaction H_n^p is calculated from the positional embedding and value V.

Finally, content-based and location-based interactions are combined into a matrix $H_n = H_c + H_n^p$, applied to the query $q_n \in \mathbb{R}^{|k|}$ to obtain the output, as shown in Eq. (1).

$$y_n = \left(H_c + H_n^p\right)^T q_n = \left(\overline{K}^T V + P_n^T V\right)^T q_n \tag{1}$$

Here, q_n is derived from the input contextual sequence via a learned linear projection. Each column of H_n represents a contextual feature that combines content and structural information. The query q_n then performs weighted allocation on these features to produce the output y_n.

Algorithm 2: Telecontext capture module enhanced global context aggregation algorithm

Input: Context sequence X, global background vector G
Output: Global context sequence Y_n
Initialization:
1: Initialize weight matrices W_k, W_q, W_v
2: Initialize position embedding matrix R
3: Compute key K, value V and query Q from G and X using W_k, W_v and W_q
4: $H_c \leftarrow K^T @ V$
5: Compute positional tensor P from R
6: **for** $i = 1$ to n **do**
7: $\quad H_n^p[i] \leftarrow P[i]^T @ V$
8: **end for**
9: $H_n \leftarrow H_c + H_n^p$
10: $Y \leftarrow []$
11: **for** $i = 1$ to n **do**
12: $\quad q_n \leftarrow Q[i]$
13: $\quad y_n \leftarrow H_n^T[i] \cdot Q[i]$
14: $\quad Y$.append(y_n)
15: **end for**
16: **return** Y

As shown in Algorithm 2, the input includes a row-level contextual sequence and a global background vector, with the output being a global row-level context sequence enhanced through remote interaction. Lines 1–3 initialize parameters and perform linear projections to generate the key K, value V and query Q for multi-head attention, along with matrix R for position embeddings. Line 4 compute content-based interaction vectors, while Lines 5–8 calculate and combine location-based interaction vectors using position embeddings. Line 9 merges the content-based and location-based interactions, and Lines 10–15 compute the final output.

As depicted in Fig. 3, In the content interaction phase, the global background G is aggregated with row-level context, K and V are derived via linear transformations to construct the content interaction matrix H_c. For position interaction, relative position embeddings produce P_n, used with V to compute the position interaction matrix H_n^p. These two interactions are then fused and applied to the query output.

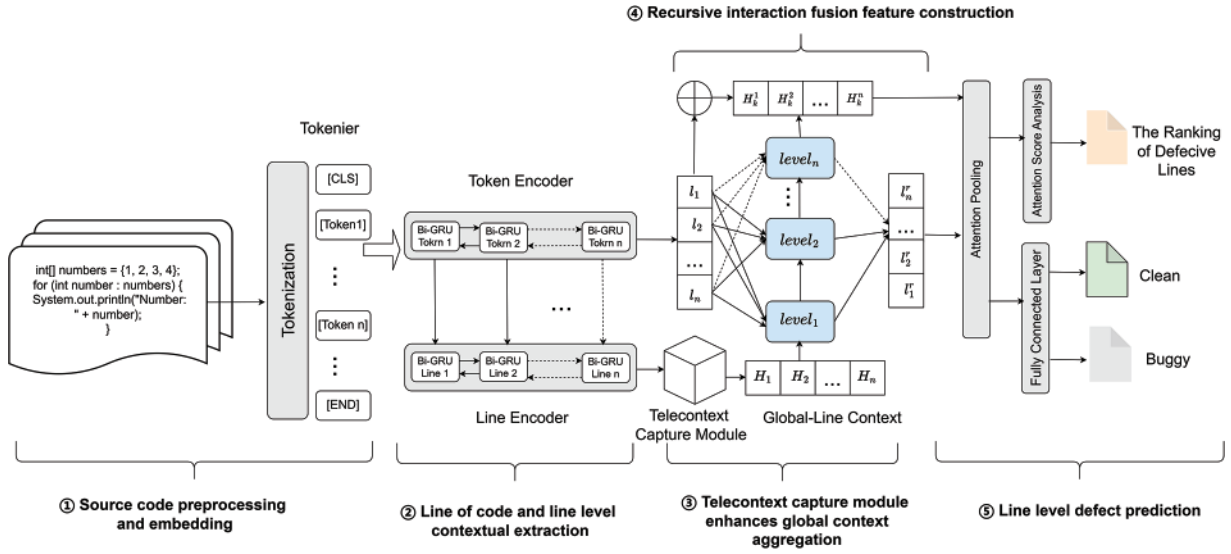


Figure 2: Overview of TRIA-LineDP model

3.4 Recursive Interaction Fusion Feature Construction

In source code analysis, it's essential to capture both global structure and local complexity, including dependency relationships. Single-layer code line information provides limited semantics and may overlook the global structure. To address this, we propose a high-order interactive recursive model.

The core of the model is an information exchange structure that processes and updates layer by layer, gradually integrating features from different levels. By fusing code lines with line-level context information, each layer enhances the contextual understanding of each code line. This recursive model generates new code line representations and multi-scale contextual interactions using attention and gating mechanisms, capturing both global and local information to identify defect-prone features.

For the i-th layer ($i \in [1, 2, \ldots, k]$), the interaction representation combines original code lines with context from the previous layer, recursively enhancing the contextual representation of each layer. An element-wise gating mechanism integrates core information from the original input l and the enriched context representation $\hat{s}^{(i)}$ from the previous layer, forming an interaction representation for layer i. This adaptive mechanism fine-tunes inter-layer features as they are passed through each layer, as shown in Eqs. (2) and (3).

$$\omega_1^{(i)} = \text{sigmoid}([l; \hat{s}^{(i)}]W^{g2} + b^{g2}) \tag{2}$$

$$\hat{l}^{(i)} = \omega_1^{(i)} \circ l + (1 - \omega_1^{(i)}) \circ \hat{s}^{(i)} \tag{3}$$

During each update, the comprehensive representation $\hat{s}^{(i)}$ is merged with the prior layer heuristic information c_{i-1}, generating updated information c_i to pass recursively to the next layer, as shown in Eqs. (4) and (5).

$$\omega_2^{(i)} = \text{sigmoid}([c_{i-1}; \hat{s}^{(i)}]W^{g3} + b^{g3}) \tag{4}$$

$$c_i = \omega_2^{(i)} \circ c_{i-1} + (1 - \omega_2^{(i)}) \circ \hat{s}^{(i)} \tag{5}$$

This model effectively integrates code line and contextual information, progressively building richer relationship representations. Finally, the enhanced representations from each layer are cascaded into a relational code line representation $l^r = [l^{(1)}; l^{(2)}; l^{(3)}; \ldots; l^{(k)}]$, capturing both global and local interactions.

Algorithm 3: Recursive interaction feature construction

Input: Code line sequence l, context sequence c, layers k, dimensions d and d_f
Output: Extended code line representation l^r
Initialize: Weights W^{g1}, W^{g2}, W^{g3}; Biases b^{g1}, b^{g2}, b^{g3}
1: **for** i in range(k) **do**
2: **1.** $\theta_l, \theta_c = \text{softmax}(l@V), \text{softmax}(c@V)$
3: $s_l, s_c = \theta_l@V, \theta_c@V$
4: **2.** $\omega_1 = \text{sigmoid}(\text{concat}([l, s_l])@W^{g1} + b^{g1})$
5: $s_{\text{hat}} = \omega_1 \circ s_l + (1 - \omega_1) \circ s_c$
6: **3.** $\omega_2 = \text{sigmoid}(\text{concat}([l, s_{\text{hat}}])@W^{g2} + b^{g2})$
7: $l_{\text{hat}} = \omega_2 \circ l + (1 - \omega_2) \circ s_{\text{hat}}$
8: $l = \text{LayerNorm}(l + \text{MLP}(l_{\text{hat}}))$
9: **4.** $\omega_3 = \text{sigmoid}(\text{concat}([c, s_{\text{hat}}])@W^{g3} + b^{g3})$
10: $c = \omega_3 \circ c + (1 - \omega_3) \circ s_{\text{hat}}$
11: **5.** l^r.append(l)
12: **end for**
13: **return** concat(l^r, axis = 1)

As shown in Algorithm 3, the input includes the sequence of code lines V_l and line contexts V_c, with the output is a related extended sentence representation formed by cascading enhanced code line representations from all layers, initialized as an embedding matrix and gating weights. Lines 1–3 extract query vectors and calculate attention scores. Lines 4–5 fuse code lines with previous layer context using a gating mechanism. Lines 6–8 update features by merging the comprehensive representation with original code lines. Lines 9–10 update context information, and Line 11 saves the enhanced code line representation for the current layer.

The recursive unit, shown in Fig. 4, processes code lines l and context c_{i-1} from the previous layer as inputs. Using the embedding matrix $V^{(i)}$, it generates relationship aggregation features for each layer. An element-wise gating mechanism controls the fusion of code lines and context to create a comprehensive relationship representation $\hat{s}^{(i)}$ for layer i. This representation $\hat{s}^{(i)}$ integrates into l, producing an extended code line representation $l^{(i)}$ at layer i. Additionally, $\hat{s}^{(i)}$ updates the context of the previous layer c_{i-1} to form the new heuristic context c_i.

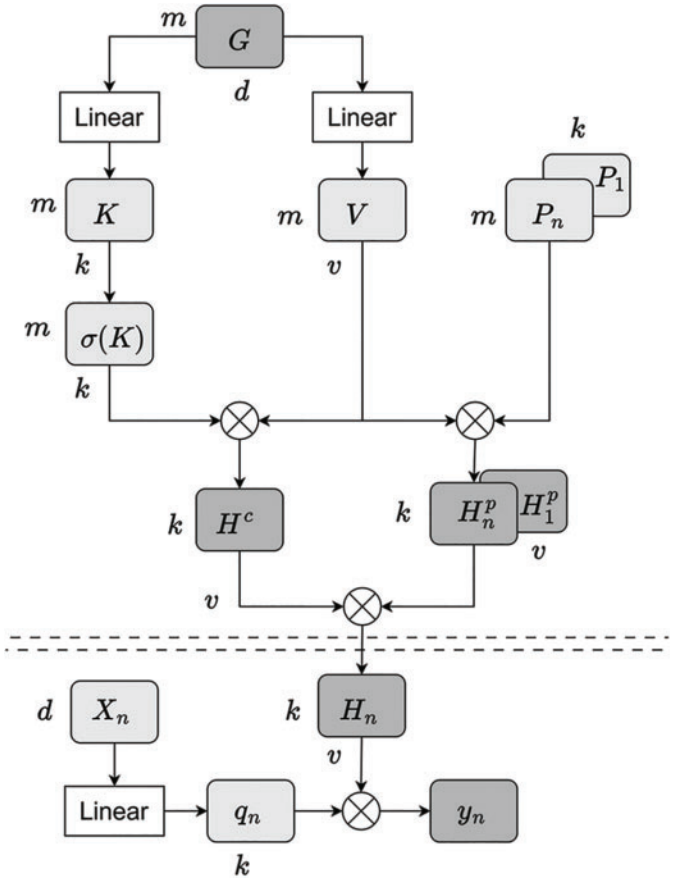


Figure 3: Telecontext capture module aggregates global context process

3.5 Defect Prediction at the Line Level

For each source code file F, all enhanced code lines are represented as $F = [l_1^r, l_2^r, \ldots, l_m^r]$. To generate an abstract file-level vector representation f, we use attention pooling [22], which calculates accurate representations based on the attention scores of each code line. This approach leverages multi-level contextual semantics to ensure the attention scores reflect the importance of code lines at various contextual scales. We calculate the attention score of each line by combining its original representation l with the final line-level context representation c_i obtained through recursive interaction fusion. This allows the attention mechanism to assess both fundamental and contextual details when calculating the attention score. The specific formula is as follows: $A = \text{softmax}\left([L; C]W_{\text{att}}^T\right)$, where $[L;C]$ is the concatenation of all code line representations and their final context representations, and W_{att} is a learnable weight matrix.

Using the attention scores, we weight and combine code lines to obtain the comprehensive file-level vector $f = FA$, which is passed through a fully connected layer serving as the predictor. This layer outputs prediction scores, which are converted into defect probabilities using the Sigmoid function, with weights W_0 and biases b_0. The corresponding calculation is presented in Eq. (6):

$$P = \text{Sigmoid}(W_0 f + b_0) \tag{6}$$

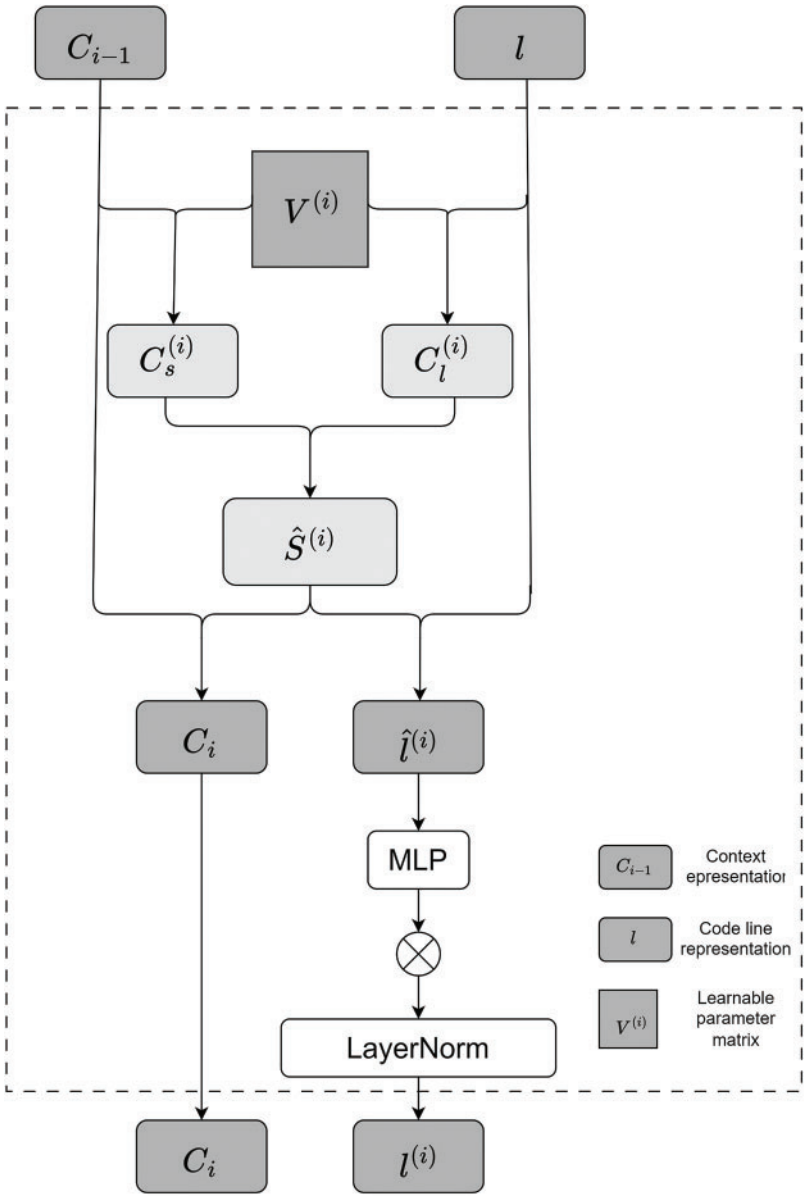


Figure 4: Recursive interaction unit

To pinpoint code lines susceptible to defects, we initially extract the attention score for each line in the defective file. This score, computed through the attention mechanism, serves as an indicator of code risk. These attention scores range from -1 to 1, reflecting the relative importance and risk level of each line of code in the current context. We calculate the attention score for each code line based on the original input representation l and the final line level context representation c_i serving as the definitive risk factor for each line of code. Among them, the final line level contextual representation c_i reflects the comprehensive characteristics of the code line in both global and local contexts, which can provide deeper semantic information for the model. This ensures that the attention score of each line of code reflects its importance at different contextual scales, thereby enabling a more accurate evaluation of

the importance of each line of code in weight calculations to improve the accuracy of risk assessment. Arrange all lines within the source code document in order of their risk coefficients, and mark the lines with higher risk coefficients as potential defect lines. These markers help developers prioritize high-risk lines of code during code review and maintenance, thereby improving software quality and reliability.

4 Experimental Setup and Results

In this section, we will outline the research methodology and present the experimental findings. This will include a detailed description of the dataset used, assessment criteria, reference methods for comparison, specifics of the experimental procedures, and a comprehensive analysis of the results corresponding to every posed question.

4.1 Experimental Setup

Dataset selection. This study utilized the publicly available defect dataset corpus provided by Wattanakriengkrai et al. [11] collected from Yatish et al. [23], which covers 32 software versions of 9 open source systems of the Apache open source project. We conducted a brief information analysis on 32 versions, as shown in Table 3. The project name (Project) indicates the name of each dataset. The count of files (#File) corresponds to the file quantity in each project across various versions, ranging from 731 to 8846. Line of Code (#LOC) signifies the line count in each project across different versions, with figures spanning from 74 k to 567 k. The proportion of defective files (#Defective Files) reflects the percentage of files with defects in each project, ranging from 2% to 28% (#Defective LOC) refers to the proportion of lines of code in each project that contain defects, ranging from 0.03% to 2.90%. Versions list the version information contained in the dataset for each project.

Table 3: Summary information statistics of dataset

Project	#File	#LOC	#Defective file	#Defective LOC	Versions
ActiveMQ	1884–3420	142–299 k	2%–7%	0.08%–0.44%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	1515–8846	75–485 k	2%–8%	0.09%–0.24%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	1963–2705	412–533 k	6%–28%	0.10%–0.63%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	757–884	74–94 k	2%–4%	0.10%–0.17%	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	1059–1834	246–537 k	7%–11%	0.17%–1.02%	0.94.0, 0.95.0, 0.95.2
Hive	1416–2662	290–567 k	6%–19%	0.31%–2.90%	0.9.0, 0.10.0, 0.12.0
JRuby	731–1614	106–240 k	2%–13%	0.03%–0.09%	1.1, 1.4, 1.5, 1.7
Lucene	805–2806	101–342 k	2%–8%	0.07%–0.39%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	1672–2578	106–165 k	2%–16%	0.05%–0.46%	1.3.0.beta1, 1.3.0beta2, 1.5.3

There are three main reasons for using this dataset in this study. Firstly, the dataset contains multiple versions of data and ground truth markers, indicating which versions are affected by defects and providing rich information on data changes between versions for research. Secondly, the dataset comes from open-source software systems used in practical applications, which has high value for research in the field of defect prediction. Finally, this dataset has been widely used in previous

experiments on line level software defect prediction research [11], indicating that it has been validated and recognized, making it a reliable benchmark for evaluating the performance of TRIA-LineDP.

Evaluation Indicators. In order to rigorously evaluate the effectiveness of our file level defect prediction method, we implemented four widely recognized evaluation metri. These include the area under the curve (AUC), Balance accuracy (BA), Matthews correlation coefficient (MCC) and geometric mean (GM).

1. **AUC.** AUC metric evaluates the overall performance of binary classification models, which can achieve better evaluation performance when dealing with class imbalanced data. A higher AUC value indicates that the model is highly effective in distinguishing positive and negative samples.
2. **BA.** The Balanced accuracy index measures the ability of a model to distinguish between defective and non defective instances by determining the average rate of true positives and true negatives.
3. **MCC.** The Matthews correlation coefficient (MCC) evaluates classifier performance by considering true positives, true negatives, false positives, and false negatives. Its value range is -1 to 1. A score of 1 indicates a perfect prediction, 0 indicates the result is equivalent to a random guess, and -1 indicates a completely incorrect prediction.
4. **GM.** The geometric mean (GM) of true positive rate (TPR) and true negative rate (TNR) reflects the overall performance of the model in positive and negative sample classification. The value of GM is between 0 and 1. The closer the GM value is to 1, the better the balance of the model in identifying positive and negative instances.

For the line-level defect prediction task, we used workload-aware metrics: recall in the top 20% of code lines (Recall@Top20%LOC) [13], effort required for the top 20% recall (Effort@Top20%Recall) [13] and Initial False Alarm (IFA) [24].

1. **Recall@Top20%LOC.** This metric evaluates how well the predictive model identifies defects within the top 20% of code lines in the software. A high value indicates that the model can effectively identify key defect lines, while a low value indicates the need for additional effort.
2. **Effort@Top20%Recall.** This metric assesses the percentage of lines of code needed to achieve the top 20% of recall, where lower figures suggest that most defects are detectable with minimal effort. In contrast, higher figures imply that more extensive code review is necessary to uncover these defects.
3. **IFA.** Initial False Alarm (IFA) counts the number of defect-free lines reviewed before finding the first defect in a file. A low IFA value means defects are found quickly, requiring minimal review of defect-free lines, while a high IFA value indicates that more time is spent on inspecting non-defective lines.

Baseline method. To evaluate the TRIA-LineDP method rigorously, we benchmarked it against leading defect prediction techniques using established evaluation metrics. Our experimental setup follows the protocol by Pornprasit et al. [13], using the first version of each project as the training dataset to establish a strong model foundation. The second version of each project serves as a validation set to fine-tune model parameters for optimal performance, while subsequent versions are used for testing.The study compares nine mainstream file-level defect prediction techniques: DH-CNN [3], convolutional neural network (CNN) [15], MFSVM [14], Bi-LSTM [7], DeepLineDP [13], statement-level vulnerability detection using graph neural networks (LineVD) [25], deep learning for bug detection (DeepBugs) [26], code text-to-text transfer transformer (CodeT5) [27] and intermediate

variable detection (IVDetect) [28]. Additionally, four advanced line-level defect prediction methods were introduced: DeepLineDP [13], ErrorProne [29], N-gram [20] and LineVD [25].

4.2 Experimental Environment and Statistical Testing

The experimental model was implemented in PyTorch and run on a server with an NVIDIA RTX 3090 GPU (24 GB RAM). Parameters were optimized using binary cross-entropy loss. A pre-trained Word2Vec model generated 50-dimensional embeddings for code tokens. Training was performed with mini-batches of size 16 and a learning rate of 0.001. To mitigate overfitting, a dropout rate of 0.2 and normalization techniques were applied. The recursive hierarchy comprised 5 levels, with 4 attention heads to capture detailed patterns. The Bi-GRU network had a hidden layer with 64 nodes, and the model final output size was 128 dimensions. Table 4 lists the experimental parameters for our model and the comparative models.

Table 4: Summary of model parameters

Model	Parameters	Model	Parameters
TRIA-LineDP	Embedding Size: 50 Batch Size: 16 Dropout: 0.2 Number of Recursive Layers: 5 Heads: 4 Hidden Size: 64 Learning Rate (Lr): 0.001	DH-CNN	Batch Size: 128 Dimensions: 100 Window: 10 Batch Word: 4 Context Window Sizes: 5 Vector Size: 100 Batch Words: 50 Negative Sampling: 10 Minimum Word Frequency: 5
MFSVM	Vector Size: 30 Window Size: 8 Learning Rate (Lr): 0.01 Iterations: 3 K: 58	CodeT5	Vocabulary Size: 32,000 Sequence Lengths: 512, 256 Batch Size: 64 Learning Rate (Lr): 2e-4
BiLSTM	Number of Layers: 2 Hidden Size: 64 Dropout: 0.2 Vocabulary Size: 30,000 Sequence Length: 512 Batch Size: 32 BERT Learning Rate: 2e-5 BiLSTM Learning Rate: 1e-3	IVDetect	Hidden Node: 100 Number of Layers: 3 Learning Rate (Lr): 0.001 Batch Size: 32
CNN	Batch Size: 32 Embedding Size: 30 Number of Hidden Layers: 10 Hidden Node: 100 Filter Length: 5	DeepLineDP	Bi-GR Hidden Size: 64 MLP Hidden Size: 64 Learning Rate (Lr): 0.001 Batch Size: 32

(Continued)

Table 4 (continued)

Model	Parameters	Model	Parameters
LineVD	Number of Layers: 3 Hidden Size: 64 Learning Rate (Lr): 0.001 Batch Size: 32 Dropout: 0.3	N-gram	Gram Sizes: 3 Sequence Lengths: 3–8 Reporting Sizes: 20

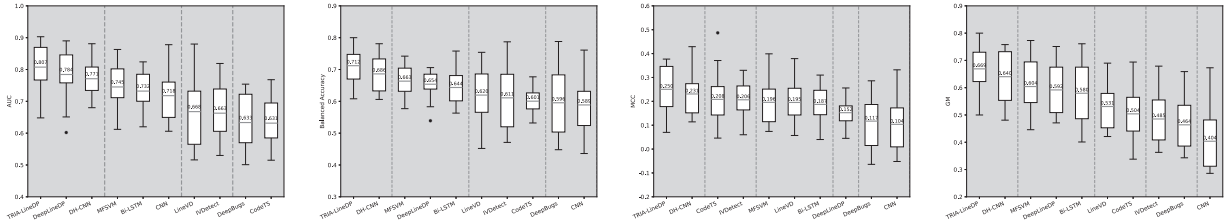
To ensure robust experimental findings and enable precise comparisons between defect prediction methods, we used the Scott-Knott Effect Size Difference (Scott-Knott ESD) test [30]. This statistical method categorizes performance metrics, such as AUC, into groups that are statistically distinct, highlighting meaningful differences among methods. The Scott-Knott ESD test involves two steps: first, adjusting data to correct non-normal distribution characteristics, ensuring that data meets statistical testing assumptions; then, combining results with small effect size differences into new groups with statistically significant distinctions. This approach minimizes performance variation within each group and highlights substantial differences between groups, effectively distinguishing the performance of defect prediction methods across datasets. Further details on the Scott-Knott ESD test are available in the literature [30].

4.3 RQ1: How Effective and Cost-Effective Is TRIA-LineDP in the Same Project Defect Prediction (WPDP) Scenario?

Software defect prediction is crucial for helping developers detect and resolve issues early in development, improving software quality. Despite advancements in defect prediction techniques to optimize software quality assurance (SQA) resources, most methods focus on file-level predictions, while line-level prediction has received less attention, creating a gap in research for more precise defect identification. The recent DeepLineDP method leverages a bidirectional GRU network to capture code structure and context effectively, showing strong defect prediction performance at both file and line levels in WPDP contexts. However, this approach does not fully capture the importance of contextual relationships and local interactions between code lines, particularly for line-level defect prediction. Thus, this study examines whether the TRIA-LineDP approach can surpass existing techniques in predicting defects at both the file and line levels in WPDP scenarios, with an added focus on cost-effectiveness.

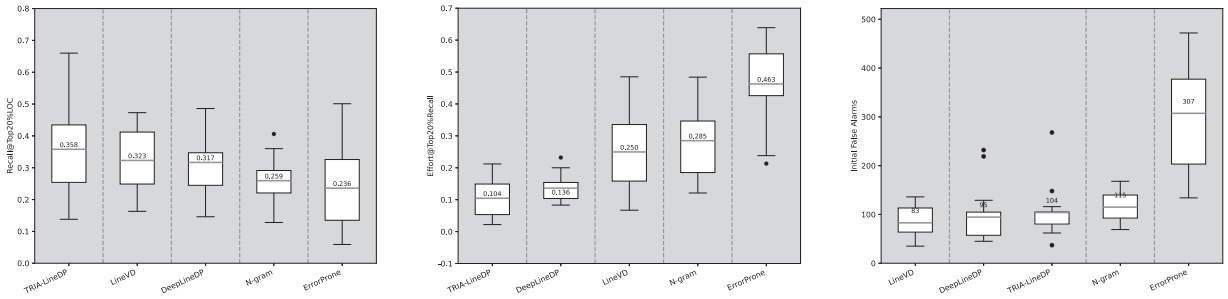
To this end, we selected twelve advanced defect prediction methods: eight focused on file-level prediction (DN-CNN, CNN, MFSVM, Bi-LSTM, LineVD, DeepBugs, CodeT5, IVDetect) and four on line-level prediction (DeepLineDP, ErrorProne, N-gram, LineVD). File-level performance was evaluated using four conventional metrics: AUC, MCC, BA and GM, while line-level performance was assessed using three workload-aware metrics: Recall@Top20%LOC, Effort@Top20%Recall and IFA, to validate TRIA-LineDP effectiveness in online defect prediction. Our experimental setup follows a design similar to Pornprasit et al. [13], using the initial project version as the training set, the subsequent version for validation, and remaining versions for testing. This study evaluates TRIA-LineDP effectiveness across both file and line levels using a comprehensive approach to training, validation and testing, and compares its performance against other methods. To highlight statistical

differences across techniques, we used the Scott-Knott ESD test, with Figs. 5 and 6 showing Scott-Knott ESD rankings and indicator distributions for TRIA-LineDP and other advanced methods for defect prediction at file and line levels in WPDP scenarios.



(a) AUC-ROC(↗) (b) Balanced Accuracy(↗) (c) MCC(↗) (d) GM(↗)

Figure 5: (Applicable to RQ1) ScottKnott ESD evaluations are conducted to analyze the AUC, BA, MCC and GM metrics within the context of Within-Project Defect Prediction (WPDP)



(a) Recall@Top20%LOC(↗) (b) Effort@Top20%Recall(↘) (c) IFA(↘)

Figure 6: (Applicable to RQ1) ScottKnott ESD analysis for the metrics Recall@Top20%LOC, Effort@Top20%Recall and IFA within the WPDP environment

Based on the results presented in Fig. 5, TRIA-LineDP performs better than other advanced methods in file-level defect prediction. The method achieved mean AUC, BA, MCC and GM scores of 0.807, 0.71, 0.25 and 0.669, respectively, reflecting improvements of 3% to 27%, 4% to 20%, 8% to 140% and 4% to 65% over other advanced defect prediction techniques at the file level. The reason for this excellent performance is that TRIA-LineDP effectively captures global context and local interaction information through a high-order interactive recursive model, thereby more accurately extracting defect code features. These results suggest that TRIA-LineDP surpasses current advanced methods in defect prediction at the file level. Furthermore, the Scott-Knott ESD test reinforced that TRIA-LineDP consistently achieved the top ranking across AUC, BA and MCC metrics, highlighting that the differences in performance are statistically meaningful.

Referring to the data illustrated in Fig. 6, it is evident that in the metrics of Recall@Top20%LOC and Effort@Top20%Recall, the TRIA-LineDP method demonstrates a cost-effectiveness improvement of 11% to 52% and 23% to 77%, respectively, when compared to other row-level defect prediction approaches. Specifically for Recall@Top20%LOC, the mean performance value of TRIA-LineDP stands at 0.358, indicating that TRIA-LineDP can detect the most defective rows (35.8%) when inspecting 20% of the total LOC. In addition, TRIA-LineDP achieved an average of 0.104 in Effort@Top20%Recall, indicating that under the condition of finding 20% defective lines, TRIA-LineDP only needs to check about 10.4% of the entire version code lines. There are two main reasons

for these outstanding performances: firstly, the Telecontext capture module, through its multi head attention mechanism, can more accurately capture broad contextual information related to specific lines of code, optimizing its ability to capture local context. The model can not only capture directly related context, but also understand the long-range dependencies and interactions between codes, which other models often overlook. By introducing positional embedding, the Telecontext capture module can not only consider the similarity of code content, but also reflect the positional relationships between code blocks. This structured interaction allows the model to more accurately reflect the actual execution process and potential logical errors of the program during prediction. Secondly, high-order interactive recursive models are designed to overcome the limitations of single level code line information processing. By introducing a deep recursive structure, the model can effectively handle and integrate global structure and local complexity, ensuring that the model not only focuses on the specific information of a single line of code, but also captures contextual information throughout the entire code body. Recursive structures allow models to re evaluate and integrate information at each level, which can more accurately simulate the actual decision-making process in software development, where high-level design decisions are typically based on a broader context rather than a single line of code. Enable the model to not only capture detailed information of local code segments, but also effectively integrate complex dependencies across multiple functions or modules, thereby showcasing superior performance in both the Recall@Top20%LOC and Effort@Top20%Recall metrics. Based on the outcomes from the Scott Knott ESD test, the performance differences between these two key indicators are not only statistically significant but also demonstrate significant effect sizes, further demonstrating the excellent performance of TRIA-LineDP.

We observed that under WDPD conditions, for the ranking of risk lines in defect files, the initial average false alarm value of TRIA-LineDP was 104, which was slightly inferior to LineVD and DeepLineDP. This means that it may rank some non defective lines before the truly defective lines. However, in actual development, developers are more focused on identifying as many defect lines as possible throughout the entire version. TRIA-LineDP is available Recall@Top20%Recall and Effort@Top20%, the excellent performance on the Recall metric demonstrates its high cost-effectiveness in comprehensive code reviews.

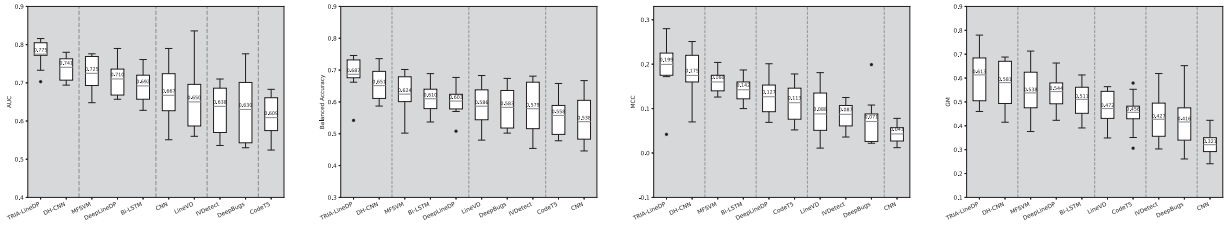
Answer to RQ1: The experimental results under WPDP conditions illustrate that the proposed TRIA-LineDP method exceeds the capabilities of existing defect prediction techniques at the file level, as well as line-level metrics such as Recall@Top20%LOC and Effort@Top20%Recall, ranking first in performance. TRIA-LineDP not only excels in predictive performance but also achieves better cost-effectiveness and lower overhead. This indicates that TRIA-LineDP has significant advantages in WPDP scenarios, enabling more effective allocation of SQA resources and improving software quality.Although not as good as LineVD and DeepLineDP in terms of IFA metrics, the reason for their higher IFA values may be due to the wide attention range of the hierarchical attention network. Compared to the total number of lines of code in the entire version, this result is still within an acceptable range.

4.4 RQ2: What Is the Effectiveness and Cost-Efficiency of TRIA-LineDP within the CPDP Framework?

Although many defect prediction methods perform well in the same project defect prediction (WPDP), practical applications often face challenges due to limitations in sample availability for new projects and differences in structure and complexity between projects. Therefore, it is necessary to evaluate the applicability and efficiency of these methods in the context of cross-project defect prediction (CPDP). This subsection compares the TRIA-LineDP method with other advanced

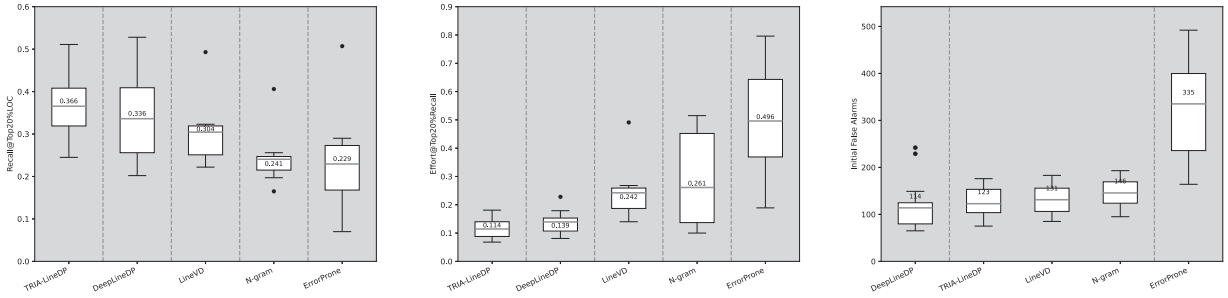
document-level and row-level defect methods in a CPDP environment to evaluate the ability and cost-effectiveness of TRIA-LineDP to transfer learning between different projects.

For this purpose, this paper examines eight methods for predicting defects at the file level, including (DH-CNN, CNN, MFSVM, Bi-LSTM, LineVD, DeepBugs, CodeT5, IVDetect) and four-line level defect prediction methods (DeepLineDP, ErrorProne, N-gram, LineVD) for comparative analysis. Among them, the performance of defect prediction at the file level is evaluated using four conventional metrics: AUC, MCC, BA and GM, while the row-level defect prediction performance is evaluated using Recall@Top20%LOC, Effort@Top20%Recall and IFA, three workload perceptions for measurement. When conducting tests in the CPDP environment, we employ this dataset partitioning strategy: the initial release of each project serves as the training dataset, the subsequent release functions as the validation dataset, and testing is ultimately performed on a different project. This cross-project testing method can simulate transfer learning requirements in practical application scenarios. Through this approach, we compared the TRIA-LineDP method with the aforementioned defect prediction methods. To statistically evaluate the performance variances among various methods, we employed the Scott Knott ESD test once more. Figs. 7 and 8 offer a comprehensive depiction of the Scott Knott ESD rankings and associated metrics for TRIA-LineDP, along with additional defect prediction approaches at both file and line levels within the context of Cross Project Defect Prediction (CPDP).



(a) AUC-ROC(↗) (b) Balanced Accuracy(↗) (c) MCC(↗) (d) GM(↗)

Figure 7: (Applicable to RQ2) ScottKnott ESD evaluations are conducted to analyze the AUC, BA, MCC and GM metrics within the context of Cross-Project Defect Prediction (CPDP)



(a) Recall@Top20%LOC(↗) (b) Effort@Top20%Recall(↘) (c) IFA(↘)

Figure 8: (Applicable to RQ2) ScottKnott ESD analysis for the metrics Recall@Top20%LOC, Effort@Top20%Recall and IFA across the CPDP environment

Based on the results in Fig. 7, the mean AUC, BA, MCC and GM scores of TRIA-LineDP are 0.775, 0.687, 0.199 and 0.613, respectively, representing improvements of 4%–27%, 5%–27%, 13%–362% and 5%–91% over other advanced file-level defect prediction methods. These results indicate

that TRIA-LineDP outperforms other file-level SDP techniques in Cross Project Defect Prediction (CPDP) environments. This strong performance is attributed to the model multi-level recursive processing, which enhances the interactive representation of code lines layer by layer, retaining core information from the original input while adding contextual insights to enable feedback and adaptive adjustment across layers. This layered strategy improves the model ability to recognize complex code relationships. Additionally, Scott-Knott ESD testing has confirmed that TRIA-LineDP consistently achieves top ranks across AUC, BA, MCC and GM metrics, indicating not only statistically significant performance differences but also substantial effect sizes.

Moreover, Fig. 8 shows that TRIA-LineDP significantly enhances cost-efficiency in line-level defect prediction, surpassing other methods by 9%–60% for Recall@Top20%LOC and 18%–77% for Effort@Top20%Recall. The average Recall@Top20%LOC of TRIA-LineDP is 0.366, meaning it can detect 36.6% of defective lines within the top 20% of LOC. Additionally, with an average Effort@Top20%Recall of 0.114, TRIA-LineDP requires reviewing only 11.4% of total code lines to identify 20% of defects. These results demonstrate the strong performance of TRIA-LineDP on workload-related metrics.

There are two main reasons for these outstanding performances: firstly, the multi head attention mechanism adopted by the Telecontext capture module can simultaneously focus on multiple different contextual information, capture a wider range of features that are not dependent on a single project specific context, and improve the generalization ability of the model. The Telecontext capture module can not only capture the local context of each code entity, but also consider remote contextual information. In the context of CPDP, there may be significant differences in code organization and style between projects. Telecontext capture module can generate more comprehensive code representations by integrating these remote and local contextual information, Secondly, recursive structures enable the model to reprocess and adjust the received information at each level through their multi-level processing approach, continuously adjusting and optimizing the representation of information. This layer by layer information processing and fusion process enables the model to more effectively integrate knowledge from various levels and adapt to data features from different projects. This structure is particularly suitable for handling diverse data, as it can capture common features at the initial level and adapt and optimize specific features at higher levels, thus adapting to the specific style and structure of new projects. Based on the outcomes from the Scott Knott ESD test, the difference in performance between these two key indicators is not only statistically significant but also demonstrates significant effect sizes, further demonstrating the outstanding performance of TRIA-LineDP.

Under CPDP conditions, although TRIA-LineDP initial average false alarm value (IFA) for ranking risk lines in defect files is 123, slightly higher than DeepLineDP, TRIA-LineDP superior performance in Recall@Top20%Recall and Effort@Top20%Recall highlights its high cost-effectiveness in comprehensive code reviews.

Answer to RQ2: Under CPDP conditions, the experimental findings indicate that the TRIA-LineDP approach proposed in this study excels in file-level metrics such as AUC, BA, MCC and GM, as well as in row-level indicators like Recall@Top20%LOC and Effort@Top20%Recall. It surpasses existing defect prediction methods and secures a leading position. TRIA-LineDP not only shows outstanding predictive accuracy but also delivers superior cost-effectiveness and reduced expense. These results highlight the considerable benefits of TRIA-LineDP in the CPDP scenario, facilitating effective cross-project transfer learning and enhancing both software quality and development efficiency. Although not as good as DeepLineDP in terms of IFA metrics, the reason for their higher

IFA values may be due to the wide attention range of the hierarchical attention network. Compared to the total number of lines of code in the entire version, this result is still within an acceptable range.

4.5 Discussion

4.5.1 The Impact of Key Modules in TRIA-LineDP

This subsection uses ablation studies to highlight the contributions of key components in enhancing model performance. We focus on three critical components: the Bi-GRU, the telecontext capture module, and the Recursive Interaction Module, each playing a vital role in capturing defect semantics in code lines, including global line-level context and interactions at both global and local levels. This analysis emphasizes line-level defect prediction due to its higher practical value compared to file-level prediction.

The data presented in Table 5 demonstrate the specific effects of removing each key component on model performance across different scenarios (WPDP and CPDP) through ablation studies. Significantly, variations in the mean Recall@Top20%LOC and Effort@Top20%Recall metrics underscore the effects of omitting specific components. In this context, "AbI" refers to the performance of a simplified TRIA-LineDP iteration, while "Diff." indicates the performance difference relative to the full TRIA-LineDP model. This analysis provides valuable insights into the significance of each component within the overall model, serving as a foundation for further optimization efforts.

Table 5: Results of TRIA-LineDP Ablation Study (WPDP and CPDP)

Approach	WPDP (Line-Level)				CPDP (Line-Level)			
	Recall@Top20%LOC		**Effort@Top20%Recall**		**Recall@Top20%LOC**		**Effort@Top20%Recall**	
	AbI	**Diff.**	**AbI**	**Diff.**	**AbI**	**Diff.**	**AbI**	**Diff.**
Bi-GRU	0.315	−12%	0.121	+16.3%	0.336	−8.2%	0.117	+2.6%
Telecontext capture module	0.326	−8.9%	0.118	+13.4%	0.317	−13.3%	0.125	+9.6%
Recursive Interaction	0.307	−14.2%	0.136	+30.7%	0.293	−19.9%	0.151	+32.4%
TRIA-LineDP	**0.358**	–	**0.104**	–	**0.366**	–	**0.114**	–

Eliminate Bi-GRU. The elimination of the Bi-GRU module was conducted to assess the impact of contextual information in code lines on model performance. Without the Bi-GRU, the model no longer had the capability to extract contextual information. As reflected in the data, the model performance experienced a significant decline in both WPDP and CPDP scenarios, with a decrease in economic performance for predicting defects at the line granularity by 12% and 8.2%, respectively. This finding underscores the critical role of the Bi-GRU in capturing the contextual information of code lines, and its removal leads to a substantial decrease in prediction effectiveness, accompanied by an increase in cost overhead. Specifically, the cost expenditure increased by 16.3% in WPDP and 2.6% in CPDP.

Eliminate telecontext capture module. The removal of the telecontext capture module was conducted to evaluate the importance of capturing global line-level context. Without the telecontext capture module, the model no longer enhances single-line context information. The data shows that the model performance significantly declines in both WPDP and CPDP scenarios, indicating the critical role of the telecontext capture module in capturing global line-level context. The absence of this module

leads to a substantial decrease in cost-effectiveness, with reductions of 8.9% and 13.3%, respectively, and increases in WPDP and CPDP costs by 13.4% and 9.6%. The removal of the telecontext capture module not only diminishes prediction performance but also significantly increases cost overhead.

To illustrate the impact of losing global context processing, Fig. 9 provides an example. In this case, the OrderProcessing service must verify user authentication status to process orders, relying on the validateToken method of the SessionManager service, which checks the user session token. If SessionManager implementation changes (e.g., altering validation logic or token format) without corresponding updates to OrderProcessing, it could lead to authentication errors or security vulnerabilities. The telecontext capture module monitors changes in SessionManager, such as updates to token validation logic or format, treating them as global state updates. Through multi-head attention, the module focuses on information flows from multiple points, allowing a comprehensive view of the impact of these changes on other parts, like OrderProcessing. In this process, changes in SessionManager are mapped to key information points, while session token demands in OrderProcessing form queries. By calculating attention scores between these keys and queries, the module identifies critical changes for OrderProcessing to consider. The telecontext capture module also uses positional embedding to understand the spatial relationship between method calls and logical flow in code. This integration of positional information enables it to account for content changes and the effects of changes in method call order. For instance, if the validation logic order changes in SessionManager, the module can detect this and evaluate its impact on OrderProcessing. Overall, the telecontext capture module ensures synchronization and consistency across system components during code and logic updates, reducing errors and vulnerabilities due to inconsistent information between components.

Eliminate recursive interaction module. To assess the impact of the recursive interaction module on the TRIA-LineDP model, this module was removed. The recursive interaction module is essential for gathering both global and nearby interaction details among code lines and their environments. Data indicate a significant performance decline in both WPDP and CPDP scenarios after its removal, highlighting the critical role of the module in integrating code lines with their context. Without the recursive interaction module, cost-effectiveness dropped by 14.2% and 19.9%, with WPDP costs rising by 30.7% and CPDP costs by 32.4%. Removing this module severely impairs prediction accuracy and significantly increases costs.

Similarly, we use the example in Fig. 9 to illustrate the global and local interaction issues between the lack of code lines and the global line level context. The user authentication check performed in the processOrder method of OrderProcessing, although seemingly correct in local logic, is highly dependent on the global context, that is, how the SessionManager handles validateToken. If the token verification logic of the SessionManager becomes stricter or looser, it may affect the behavior of the OrderProcessing service, especially when dealing with boundary situations. A single level of code analysis may not be able to capture such dynamic dependencies and changes in global context. The high-order interactive recursive model can effectively capture and integrate the actual impact of changes in the verification logic of the SessionManager on OrderProcessing by fusing information layer by layer through its multi-layer structure. Each layer of the model not only processes individual lines of code information, but also comprehensively considers contextual information from the previous layer, so that the entire model can reflect changes in global logic. Allow the model to evaluate globally how these changes affect the behavior of OrderProcessing. After each layer is processed, the generated contextual information is passed on to the next layer, gradually building and enhancing the understanding of global validation logic changes. This inter layer information transfer and recursive structure ensure a gradual comprehensive analysis from local to global. Through the application of this high-order interactive recursive model, the impact caused by the verification logic changes of

the SessionManager can be more effectively handled and adapted, ensuring that OrderProcessing can accurately reflect the latest verification strategies and global context requirements when processing orders. By better integrating global and local contextual information, errors and security issues caused by information silos and logical inconsistencies can be reduced.

```
1  public class SessionManager {
2      public boolean validateToken(String token) {
3          return token != null && token.equals("expected_token_value");
4      }
5  }

6  public class OrderProcessing {
7      public void processOrder(Order order) {
8          if (!Authentication.isUserAuthenticated(order.getUser())) {
9              throw new SecurityException("User not authenticated.");
10         }
11         System.out.println("Order processed successfully.");
12     }
13 }

14 public class Authentication {
15     public static boolean isUserAuthenticated(User user) {
16         return user.getSessionToken() != null &&
17                SessionManager.validateToken(user.getSessionToken());
18     }
19 }
```

Figure 9: Examples of the impact of Telecontext capture module and Recursive interaction module on actual code

In conclusion, the integration of the Bi-GRU, telecontext capture module, and recursive interaction module substantially improves both the precision and efficiency of defect prediction at the line level. These elements are crucial for capturing essential features including the semantics of individual code statements, the overarching context at the line level, and the dynamics between global and local information within code lines. Their combined effectiveness makes the software quality assurance process more efficient and cost-effective, thereby alleviating the burden on development teams when addressing defects.

4.5.2 Threat to Effectiveness

(a) Model parameter selection: Choosing appropriate hyperparameters is essential for achieving optimal predictive performance. While hyperparameter weights were manually adjusted during the experiment, this approach may not yield the best possible settings. Fine-tuning parameters in deep learning is computationally intensive, and variations in hyperparameter choices can lead to fluctuations in model performance.

(b) Implementation of comparative methods: To minimize the impact of implementation errors, several precautions were taken. For benchmark methods like DeepLineDP and ErrorProne, publicly available code was used directly, reducing variations due to implementation differences and ensuring reliable results. For other comparison methods lacking open-source code, we strictly followed the methodologies outlined in published papers during reproduction and experimentation. Despite this careful approach, minor differences in implementation may still influence results, potentially causing slight deviations from the actual outcomes.

(c) Diversity of experimental data: This study uses data from nine projects for experimentation. While these projects cover different types of software, they do not fully represent the diversity of all software projects. Open-source datasets typically come from smaller, structurally independent projects, whereas real-world software often involves larger, more complex dependencies. Although datasets with multiple versions and rich labeling offer some generalizability, future applications may need to adapt further to the complexity of various project scales. Differences in coding styles and defect patterns between open-source and real-world projects can also impact model performance. While the dataset from Yatish et al. includes real open-source system projects with common coding styles, future work could introduce model fine-tuning and data preprocessing to better accommodate commercial code styles. Consequently, model effectiveness may vary across different software project types.

(d) Limitations of evaluation indicators: This study used various metrics, including AUC, MCC, BA, GM, Recall@Top20%LOC, Effort@Top20%Recall and IFA, to assess model performance. However, these metrics may not fully capture model effectiveness in practical applications. Although widely used in academic research, the true effectiveness of the model must ultimately be evaluated within specific real-world scenarios and requirements.

4.5.3 Limitation

While the TRIA-LineDP model is effective at capturing complex dependencies in code and improving defect prediction accuracy, it has certain limitations in practical applications, mainly concerning model scalability, handling of complex code libraries, and challenges in deployment.

(a) Scalability issues: As project size grows, the model must handle an increasing number of code files and complex dependencies simultaneously. Expanding recursive models does not always scale linearly, which may lead to performance degradation.

(b) The challenge of handling complex code libraries: Software projects often involve multiple programming languages and styles, and code repositories may be frequently updated. Recursive models require constant updates and retraining to adapt to these changes, increasing maintenance costs and potentially lowering predictive accuracy within certain periods.

(c) Challenges of actual deployment: Integrating recursive models into existing development tools and workflows can be complex, often requiring additional development and maintenance work. This integration may also face compatibility issues with existing systems.

5 Conclusion

This article proposes a telecontext-enhanced recursive interactive attention fusion method (TRIA-LineDP) for line-level defect prediction. This method fully utilizes the heuristic relationship between code lines and line-level context, taking into account the context at the line level, and the dynamics interaction between global and local information within code lines to pinpoint defects in code files

and individual lines. Experimental results from within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) conducted on nine different projects (encompassing a total of 32 versions) demonstrated that TRIA-LineDP had accuracy rates 3%–27%, 4%–20%, 8%–140% and 4%–65% higher than existing methods on the traditional indicators of AUC, BA, MCC and GM within the project, respectively. Compared with recent line-level defect prediction methods, improvements in detecting defects at the line granularity ranged from 11%–52% for Recall@Top20%LOC and 23%–77% for Effort@Top20%Recall. In cross-project tests, AUC, BA, MCC and GM accuracy increased by 4%–27%, 5%–27%, 13%–362% and 5%–90%, respectively, with further improvements of 9%–60% in Recall@Top20%LOC and 18%–77% in Effort@Top20%Recall. These results indicate that TRIA-LineDP outperforms existing methods for both file-level and line-level defect prediction, making it an effective and economical tool for software quality assurance teams.

While the proposed model shows promise for line-level defect prediction, current research primarily relies on static code datasets, and the model has yet to be tested in dynamic, real-time development environments. However, TRIA-LineDP has potential for integration into dynamic and real-time settings. For example, in a dynamic execution environment, the model could utilize runtime data (such as performance logs or anomaly information) to improve defect prediction accuracy, capturing potential defects during software runtime and assisting developers with real-time feedback for system repair and optimization. In continuous integration (CI/CD) pipelines, defect prediction can be applied to each code submission or incorporated into code review tools to help developers detect potential issues in real-time. Future research could further explore combining static and dynamic analysis to support complex real-time feedback mechanisms.

Acknowledgement: We are grateful to the valuable comments and suggestions from all the reviewers and editors for proof reading and making corrections to this article. Without their support, it would have not been possible to submit this in the current form.

Funding Statement: This work was supported by National Natural Science Foundation of China (no. 62376240).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Bingjian Yan, Haitao He, Ke Xu; data collection: Bingjian Yan, Lu Yu; analysis and interpretation of results: Bingjian Yan, Lu Yu; draft manuscript preparation: Bingjian Yan, Haitao He, Ke Xu; manuscript final layout and preparation for submission: Bingjian Yan, Ke Xu; supervision: Haitao He. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

[1] F. Meng, W. Cheng, and J. Wang, "Semi-supervised software defect prediction model based on tri-training," *KSII Trans. Internet Inform. Syst. (TIIS)*, vol. 15, no. 11, pp. 4028–4042, 2021.

[2] E. N. Akimova *et al.*, "A survey on software defect prediction using deep learning," *Mathematics*, vol. 9, no. 11. 2021, Art. no. 1180. doi: 10.3390/math9111180.

[3] A. Abdu, Z. Zhai, H. A. Abdo, and R. Algabri, "Software defect prediction based on deep representation learning of source code from contextual syntax and semantic graph," *IEEE Trans. Reliab.*, vol. 73, no. 2, pp. 820–834, 2024. doi: 10.1109/TR.2024.3354965.

[4] F. Yang, Y. Huang, H. Xu, P. Xiao, and W. Zheng, "Fine-grained software defect prediction based on the method-call sequence," *Comput. Intell. Neurosci.*, vol. 2022, no. 1, 2022, Art. no. 4311548. doi: 10.1155/2022/4311548.

[5] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert. Syst. Appl.*, vol. 147, no. 2, 2020, Art. no. 113156. doi: 10.1016/j.eswa.2019.113156.

[6] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Trans. Reliab.*, vol. 70, no. 2, pp. 613–625, 2020. doi: 10.1109/TR.2020.3040191.

[7] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan and I. Zada, "Software defect prediction employing BiLSTM and BERT-based semantic feature," *Soft Comput.*, vol. 26, no. 16, pp. 7877–7891, 2022. doi: 10.1007/s00500-022-06830-5.

[8] Z. Zhao, B. Yang, G. Li, H. Liu, and Z. Jin, "Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks," *J. Syst. Softw.*, vol. 184, no. 1, 2022, Art. no. 111108. doi: 10.1016/j.jss.2021.111108.

[9] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1241–1266, 2018. doi: 10.1109/TSE.2018.2877678.

[10] Y. Shao, B. Liu, S. Wang, and G. Li, "A novel software defect prediction based on atomic class-association rule mining," *Expert. Syst. Appl.*, vol. 114, no. 2, pp. 237–254, 2018. doi: 10.1016/j.eswa.2018.07.042.

[11] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1480–1496, 2020. doi: 10.1109/TSE.2020.3023177.

[12] J. Zhu, Y. Huang, X. Chen, R. Wang, and Z. Zheng, "SyntaxLineDP: A line-level software defect prediction model based on extended syntax information," in *2023 IEEE 23rd Int. Conf. Softw. Qual. Reliab. Secur. (QRS)*, IEEE, 2023, pp. 83–94.

[13] C. Pornprasit and C. K. Tantithamthavorn, "DeepLineDP: Towards a deep learning approach for line-level defect prediction," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 84–98, 2022. doi: 10.1109/TSE.2022.3144348.

[14] J. Liu *et al.*, "A multi-feature fusion-based automatic detection method for high-severity defects," *Electronics*, vol. 12, no. 14, 2023, Art. no. 3075. doi: 10.3390/electronics12143075.

[15] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE Int. Conf. Softw. Qual. Reliab. Secur. (QRS)*, IEEE, 2017, pp. 318–328.

[16] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th Int. Conf. Min. Softw. Repos. (MSR)*, IEEE, 2019, pp. 34–45.

[17] M. Nevendra and P. Singh, "Software defect prediction using deep learning," *Acta Polytechn. Hung.*, vol. 18, no. 10, pp. 173–189, 2021. doi: 10.12700/APH.18.10.2021.10.9.

[18] G. Gharibi, V. Walunj, R. Nekadi, R. Marri, and Y. Lee, "Automated end-to-end management of the modeling lifecycle in deep learning," *Empir. Softw. Eng.*, vol. 26, no. 2, pp. 1–33, 2021. doi: 10.1007/s10664-020-09894-9.

[19] P. Mahbub and M. M. Rahman, "Predicting line-level defects by capturing code contexts with hierarchical transformers," in *2024 IEEE Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, IEEE, 2024, pp. 308–319.

[20] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Engi.*, 2016, pp. 708–719.

[21] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *2019 IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, IEEE, 2019, pp. 37–48.

[22] Z. Lin *et al.*, "A structured self-attentive sentence embedding," 2017, *arXiv:1703.03130*.

[23] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, IEEE, 2019, pp. 654–665.

[24] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, IEEE, 2017, pp. 159–170.

[25] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. 19th Int. Conf. Min. Softw. Repos.*, 2022, pp. 596–607.

[26] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," in *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–25, 2018. doi: 10.1145/3276517.

[27] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.

[28] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meet. Europ. Soft. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 292–303.

[29] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *2012 IEEE 12th Int. Work. Conf. Source Code Anal. Manipulation*, IEEE, 2012, pp. 14–23.

[30] S. Herbold, "Comments on ScottKnottESD in response to "An empirical comparison of model validation techniques for defect prediction models"," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1091–1094, 2017. doi: 10.1109/TSE.2017.2748129.