Software Quality Journal (2026) 34:12
https://doi.org/10.1007/s11219-026-09747-6

RESEARCH

CoCo-GAN: CodeBERT-driven collaborative generative adversarial learning for software defect prediction

Xiaoxing Yang[1] · Liwei Xiao[2] · Jianmin Su[3] · Bingding Huang[2]

Received: 9 July 2025 / Accepted: 20 February 2026

Abstract
Software defect prediction (SDP) is essential for improving software reliability by identifying faulty components early in the development process. However, SDP faces three major challenges: class imbalance, scarcity of labeled data, and distributional discrepancies across projects or versions. To address these challenges, we propose CoCo-GAN, a CodeBERT-based collaborative adversarial framework for SDP. CoCo-GAN consists of three main components: a bimodal CodeBERT input constructed from source code and associated comments to capture richer semantic representations; a conditional GAN (CGAN) that generates synthetic samples for the minority class, mitigating data scarcity and class imbalance; and a semi-supervised GAN (SSGAN) that leverages unlabeled data to reduce distribution gaps between source and target projects. Additionally, a block-regularized repeated half-sampling (BRHS) strategy combined with ensemble learning is employed to better utilize limited labeled data and enhance training robustness. Experiments on ten Java projects from the PROMISE benchmark show that CoCo-GAN increases the average F1 score from 51.8 and 53.4 of MFGNN and DP-CCL, respectively, to 57.8 in within-project defect prediction (WPDP), and from 43.4 and 51.0 to 53.1 in cross-project defect prediction (CPDP). This improvement arises because CoCo-GAN jointly exploits semantic-rich bimodal inputs to strengthen feature representation, uses CGAN-generated minority samples to stabilize learning under extreme imbalance, and aligns source–target domains via SSGAN's adversarial use of unlabeled data—addressing a combination of challenges that prior methods typically overlook or handle only partially.

Keywords Software defect prediction · Semi-supervised learning · Generative adversarial networks · Pre-trained models · Ensemble learning

1 Introduction

With the growing complexity of software systems, ensuring software quality has become increasingly challenging. To detect and fix potential defects early in the development cycle and minimize their impact on system stability, functionality, and user

Xiaoxing Yang and Liwei Xiao contributed equally to this work.

Extended author information available on the last page of the article

Published online: 19 March 2026

safety - preventing serious consequences such as financial losses or risks to human life - the prediction of software defects aims to automatically identify code modules faulty using existing code and historical data (Shen & Chen, 2020; Thomas & Kaliraj, 2024). This reduces the need for manual inspection and has become a prominent research focus in software engineering, attracting significant attention from both academia and industry (Li et al., 2024; Okumoto, 2016).

Early SDP approaches combined handcrafted software metrics—such as lines of code (LOC), complexity, and process-related indicators (Qiu et al., 2025)—with traditional machine learning models like logistic regression, random forest, and naive Bayes (Arar & Ayan, 2017; Wang & Li, 2010; Wang et al., 2012). However, these features often fail to capture the semantic syntax and contextual information of the source code, meaning that two semantically different code fragments may exhibit similar metric values (Abdu et al., 2022; Wang et al., 2018). Moreover, feature extraction becomes increasingly costly and impractical as software systems scale (Zhou et al., 2025).

Recent studies in software defect prediction have focused on extracting semantic and structural features from source code to enhance prediction performance. Early approaches typically represent program structure using token sequences, abstract syntax trees (ASTs) (Deng et al., 2020; Shippey et al., 2019), and control- or data-flow graphs (CFGs/DFGs) (Gupta et al., 2022; Zhao et al., 2022); however, token-based methods struggle to capture long-range dependencies, ASTs may lose contextual information during traversal, and graph-based methods—while expressive—are computationally expensive and difficult to scale. To learn patterns from these representations, deep learning models such as Deep Belief Network (DBN) (Wang et al., 2018), Convolutional Neural Network (CNN) (Abdu et al., 2024), Recurrent Neural Network (RNN) (Fan et al., 2019), and Long Short-Term Memory network (LSTM) (Khleel & Nehéz, 2024) have been employed. Yet DBN and CNN tend to capture only local or shallow patterns (Li et al., 2017), while RNN and LSTM, despite modeling sequential dependencies, often suffer from vanishing gradients over long sequences (Fang et al., 2022), limiting their ability to model rich program semantics in complex software systems.

Pre-trained models such as CodeBERT (Feng et al., 2020), UniXCoder (Guo et al., 2022), and CodeT5 (Wang et al., 2021) have demonstrated superior ability compared with traditional feature extraction methods in capturing both syntactic structures and semantic dependencies from source code, and have been successfully applied to various code intelligence tasks (Dam et al., 2023; Zhang et al., 2024). However, when directly applied to software defect prediction, their effectiveness is limited. In particular, these pre-trained models are not inherently designed to handle practical challenges in SDP, including severe class imbalance between defective and clean modules, scarcity of labeled data, and distributional discrepancies across projects or versions (Song et al., 2024; Sahar et al., 2024).

Class imbalance—where defective modules are significantly outnumbered by clean ones—is common in real-world software projects. This skew tends to bias learning algorithms toward the majority class, often reducing recall for defective modules (Borandag, 2023; Khleel & Nehéz, 2023).

Labeled data is typically scarce, as reliable defect labels require manual auditing or consistent historical bug reports, which are frequently unavailable or project-specific. This limits the ability to learn robust decision boundaries, especially in cross-project or cross-version settings where no labeled data exists for the target project.

Moreover, differences in coding conventions, development practices, and defect patterns across projects lead to distributional shifts that hinder model generalization (Sahar et al., 2024). Effective software defect prediction therefore requires an integrated approach that jointly addresses class imbalance, makes efficient use of limited labeled data, and aligns feature distributions across domains.

In this papeer, We propose CoCo-GAN, a CodeBERT-based framework for software defect prediction that jointly encodes source code and comment-derived semantics using a bimodal input strategy. The model integrates conditional and semi-supervised GANs (Mirza & Osindero, 2014; Odena, 2016) to alleviate data scarcity, class imbalance, and distribution shifts. Additionally, the BRHS strategy (Wang et al., 2019) is adopted to improve data utilization and support model ensembling. Extensive experiments on multiple projects from the PROMISE dataset show that CoCo-GAN achieves performance improvements over baseline methods in both within-project and cross-project settings.

The main contributions of this work are summarized as follows:

- Improves defect prediction by jointly using source code and its comments to better capture program semantics—without manual feature engineering.
- Enables more effective learning for defect prediction from limited, class-imbalanced data by combining synthetic minority generation with unlabeled target utilization.
- Consistently outperforms baselines on PROMISE projects in both within- and cross-project defect prediction.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 details the implementation of the proposed CoCo-GAN model. Section 4 and Section 5 describe the experimental setup and present the results, respectively. Section 6 discusses potential threats to validity. Finally, Section 7 concludes the paper and suggests directions for future work.

2 Related work

Software defect prediction builds models from historical development data to identify potentially defective modules, thereby improving software reliability and development efficiency (Basili et al., 1996). As illustrated in Fig. 1, the process begins by extracting code modules and their corresponding defect labels from version control systems or code repositories (Hassan, 2009), where data may come from different versions of

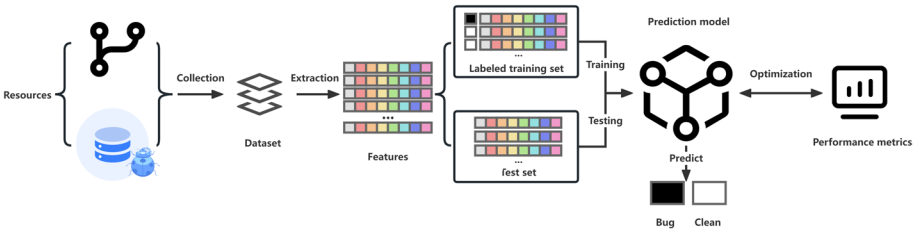

Fig. 1 Workflow of software defect prediction

the same project or from other projects. In the feature construction phase, researchers either extract traditional static code metrics or directly capture semantic and structural features from the source code (Abreu & Melo, 1996; McCabe, 1976); these features, combined with defect labels, form training and testing samples for model training. The trained model is evaluated on a retained test set or external data and can ultimately be deployed in real development environments to predict the defect tendency of newly submitted code modules.

To lay the groundwork for our approach, this section reviews the key research areas that underpin our proposed model.

2.1 Research on software defect prediction based on source code characteristics

Most methods derive source code characteristics based on AST node information. The model proposed by Song et al. (Wang et al. 2018) uses DBN to automatically learn semantic features of programs from AST nodes, significantly improving the accuracy of defect prediction in both WPDP and CPDP tasks. Shi et al. (2020) encoded AST as control and symbol sequence and input BiLSTM for modeling. Zhao et al. (2022) fused AST and CFG into a graph structure input graph neural network (GNN) to improve the ability to capture structured semantic. Qiu et al. (2023) used AST based structural matrix to input CNN to realize automatic feature extraction. Wenjun et al. (Yao et al. 2023) designed a PSFM model, combined with the syntax structure and source code text, and used CNN to automatically mine potential defect features, showing good universality, while Zhou et al. (2022) further combined GNN and CNN to achieve the best prediction performance on 21 data sets. Zhou et al. (2025) propose DP-TACT, a multi-feature fusion approach for software defect prediction that integrates AST, CFG, and traditional metrics using multiscale CNN, BiLSTM, and GCN, demonstrating superior performance on the PROMISE dataset.

With the development of pre-trained models, code representation moves from the structural level to the semantic level. Nasir et al. (Uddin et al. 2022) combined BERT with BiLSTM to build SDP-BB model, which is used to extract context representation of source code and improve model performance with data augmentation. The LineVul model proposed by Fu and Tantithamthavorn (2022) uses CodeBERT as a Transformer encoder for defect detection tasks in C/C++ projects, and the results show that the F1 score is much higher than traditional methods. Liu et al. (2023) combined UnixCoder with CNN, and improved the generalization effect of the model by using the powerful cross-language semantic modeling ability of UnixCoder. Sahar et al. (2024) used CodeBERT to extract code semantic vectors, introduced contrastive learning strategy to strengthen the discrimination ability of similar samples, and achieved significantly better prediction results than existing methods in the PROMISE project.

Nevertheless, both structure-centric and pre-trained semantic approaches exhibit limitations in practical defect prediction. Structure-based methods rely heavily on local syntactic constructs and often fail to capture long-range semantic dependencies or logical intent. Meanwhile, despite their richer semantic representations, pre-trained models are typically employed as static feature extractors—used without task-specific fine-tuning or adaptive optimization (Malhotra & Singh, 2024; Vasamsetty et al., 2022)—which limits their ability to address domain-specific challenges such as extreme class imbalance and cross-project distribution shifts.

A growing line of research explicitly targets the practical data constraints inherent in SDP. To cope with limited labeled examples, some studies adopt unsupervised, semi-supervised, or transfer learning to leverage unlabeled or cross-project data (Bai et al., 2022; Majumder et al., 2024); others address class imbalance through data augmentation (including resampling and data generation) and ensemble strategies (Alqarni & Aljamaan, 2023; Borandag, 2023; Khleel & Nehéz, 2023), and mitigate distributional discrepancies by filtering software metrics or employing contrastive learning to enhance feature alignment across domains (Sahar et al., 2024).

Yet these approaches tend to address the challenges of label scarcity, class imbalance, and distribution shift in a modular or isolated manner, without considering their interdependencies or designing a unified mechanism to jointly mitigate them.

2.2 CodeBERT

CodeBERT is an encoder-only transformer model based on RoBERTa (Liu et al., 2019), pre-trained on the CodeSearchNet[1] dataset using code–comment pairs from six programming languages. By jointly encoding source code and its associated natural language comments into a unified sequence, CodeBERT learns cross-modal representations that capture both syntactic structure and semantic intent. Its bidirectional architecture enables holistic understanding of code context—modeling variable dependencies, function nesting, and control flow—making it well-suited not only for code search and documentation generation but also for downstream tasks requiring deep comprehension of program logic.

The proposed model applies CodeBERT as a trainable semantic encoder, fine-tuned to learn domain-adaptive representations for software defect prediction under limited labeled data.

2.3 Conditional and semi-supervised GANs

CGAN extend the traditional GAN framework by introducing conditional information, such as class labels or textual descriptions, into both the generator and discriminator. This allows the generator to produce samples that match specific conditions and the discriminator to evaluate their authenticity under the same context, enabling more controllable and targeted generation (Alqarni & Aljamaan, 2023). SSGAN further enhance GANs by combining generation and classification with limited labeled data. The discriminator not only distinguishes real from fake samples but also classifies them, while the generator learns to produce realistic and label-consistent samples. This joint learning helps improve both data augmentation and classification performance under scarce supervision (Xiaozhi et al., 2020).

The proposed model integrates CGAN and SSGAN into a unified generative framework to jointly address class imbalance, defect sample scarcity, and distribution shifts through conditional synthesis and semi-supervised feature alignment.

2.4 Block-regularized repeated half-sampling

BRHS is an enhanced variant of the Repeated Learning Test (RLT) that reduces the variance of generalization error estimation by controlling sample overlap across training sets. It constructs balanced partitions using two orthogonal arrays, ensuring that each instance appears

[1] https://github.com/github/CodeSearchNet

equally often across all training folds and that pairwise overlaps remain consistent—as illustrated in Table 1. Compared to conventional RLT, BRHS offers lower estimation variance, flexible partition sizes, and a rigorously balanced design, making it especially well-suited for learning under small-sample and highly imbalanced conditions.

In this work, we adopt the same block configuration as the original BRHS study without experimental tuning.

Drawing on both the insights and limitations of prior work, this study unifies several complementary technical directions into a single framework. By jointly leveraging semantic representation, conditional generation, semi-supervised learning, and balanced ensemble partitioning, the proposed approach aims to address the intertwined practical challenges of limited labeled data, severe class imbalance, and distribution shifts.

3 Methods

This section describes the motivation behind our study and details the implementation of our proposed method. Specifically, we present the dataset construction and preprocessing process, the design of feature engineering, and the training and ensemble testing strategies.

3.1 Research motivation

Software defect prediction faces three major challenges that hinder model performance and generalization. First, the inherent class imbalance between defective and non-defective samples often biases learning toward the majority class, reducing the model's ability to detect actual defects. Second, the scarcity of high-quality labeled data limits the effectiveness of data-driven approaches, particularly deep learning models. Third, significant distribution shifts between training and test data—especially in cross-version and cross-project scenarios—undermine the transferability of learned representations. To address these issues, this study proposes a unified defect prediction framework capable of handling class imbalance, data scarcity, and distribution heterogeneity, while enhancing the semantic understanding of source code to achieve accurate and robust defect identification.

3.2 Model implementation

3.2.1 Dataset

The PROMISE dataset (Jureczko & Madeyski, 2010) is one of the most widely used benchmark datasets in software defect prediction. Originally collected by the NASA Software

Table 1 Each row represents one of the six splits of dataset S into training set T and validation set V. Each block is denoted as B^i

Split	Training Blocks (T)	Validation Blocks (V)
Split 1	B^2, B^4, B^6, B^7, B^8	$B^1, B^3, B^5, B^9, B^{10}$
Split 2	$B^1, B^2, B^6, B^9, B^{10}$	B^3, B^4, B^5, B^7, B^8
Split 3	$B^2, B^3, B^5, B^7, B^{10}$	B^1, B^4, B^6, B^8, B^9
Split 4	B^1, B^3, B^7, B^8, B^9	$B^2, B^4, B^5, B^6, B^{10}$
Split 5	B^3, B^4, B^5, B^6, B^9	$B^1, B^2, B^7, B^8, B^{10}$
Split 6	$B^1, B^4, B^5, B^8, B^{10}$	B^2, B^3, B^6, B^7, B^9

Engineering Laboratory and released through the PROMISE project,[2] it contains multiple historical versions of real-world Java projects. Owing to its clear structure and standardized format, the PROMISE dataset has become an important foundation for constructing and evaluating defect prediction models, and is widely used in both within-project and cross-project studies (Chen et al., 2020; Dam et al., 2019).

In this study, ten Java projects with several versions are selected to ensure consistency with the datasets used in the baseline models, thereby providing a fair and comparable experimental setup for both prediction scenarios. Each sample in the dataset corresponds to a source code file labeled as defective or non-defective. We utilize the source code and its corresponding labels for model training and evaluation, ensuring that all features are learned directly from the raw code. The selected projects collectively cover 25 versions, with thousands of source files in total, encompassing diverse project scales and defect distributions. Detailed statistics are summarized in Table 2.

3.2.2 Feature engineering

Figure 2 illustrates the feature extraction process of the CodeBERT model. To leverage CodeBERT's pre-training paradigm, which is designed to model pairs of natural language descriptions and source code, we design a bimodal input structure that integrates both natural language (NL) and programming language (PL) components using a unified input template consistent with the model's pre-training format.

For the NL modality, we extract the class name, class-level comments, and annotated method names from Java source files to capture the high-level semantic intent of code modules. Following prior work showing that contextual prompts can enhance downstream performance (Pan et al., 2021), we prepend a fixed prompt—"*These codes have corresponding class or function information:*"—to guide the model's interpretation. The class name and annotated method names are explicitly retained because they often encode core functionality; in particular, developer-annotated identifiers tend to highlight essential behaviors of the module.

This design aligns with CodeBERT's pre-training objective, which leverages paired code–natural language segments. Moreover, by keeping the NL component concise, we reserve more token capacity for the source code, allowing the model to better attend to syntactic and structural patterns. The final NL input is constructed as:

$$\mathrm{NL} = \mathrm{Prompt} \,\|\, \mathrm{Class\_name} \,\|\, \mathrm{Method\_names}, \tag{1}$$

where $\|$ denotes concatenation. If no annotated method names are available, `Method_names` is replaced with the class-level comment to preserve semantic completeness. For minimal constructs such as interfaces or enumerations that lack both method annotations and comments, the NL input consists only of the prompt and class name. An illustrative example of this NL construction is provided in Fig. 2.

For the PL modality, we preprocess each Java file by removing single-line, multi-line, and Javadoc comments, as well as package declarations and import statements. Only func-

[2] https://openscience.us/repo/defect/

Table 2 Detailed statistics of defect prediction datasets

Project	Version	No. of files	No. of defective files	% of defective files
Ant	1.5	292	32	10.96
	1.6	350	92	26.29
	1.7	741	166	22.40
Camel	1.2	595	216	36.30
	1.4	848	145	17.10
	1.6	935	188	20.11
Ivy	1.4	241	16	6.64
	2.0	352	40	11.36
jEdit	3.2	260	90	34.62
	4.0	293	75	25.60
	4.1	300	79	26.33
Log4j	1.0	119	34	28.57
	1.1	100	37	37.00
Lucene	2.0	186	91	48.92
	2.2	234	143	61.11
	2.4	330	203	61.52
Poi	1.5	235	141	60.00
	2.5	380	248	65.26
	3.0	438	281	64.16
Synapse	1.0	157	16	10.19
	1.1	222	60	27.03
	1.2	256	86	33.59
Xalan	2.4	676	110	16.27
	2.5	762	387	50.79
Xerces	1.2	439	71	16.17
	1.3	452	69	15.27

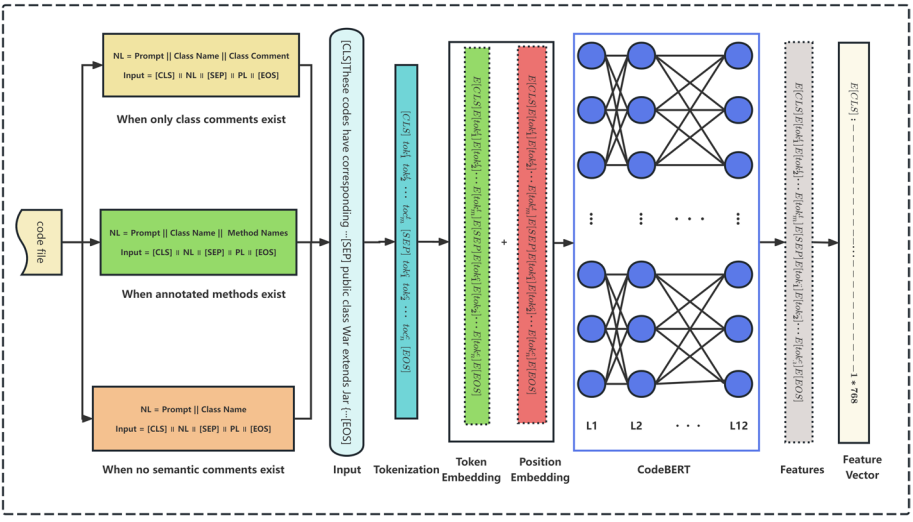

Fig. 2 Feature extraction with CodeBERT for defect prediction

tion bodies and core implementation code are retained, reducing syntactic noise and ensuring effective utilization of CodeBERT's token capacity.

Finally, NL and PL are organized into CodeBERT's standard input format as follows:

$$X_{\text{input}} = [\text{CLS}] \,\|\, NL \,\|\, [\text{SEP}] \,\|\, PL \,\|\, [\text{EOS}], \tag{2}$$

Here, `[CLS]` serves as a special token for capturing the global semantic representation used in downstream classification tasks. `[SEP]` separates the NL and PL modalities, and `[EOS]` marks the end of the input sequence.

Before being fed into CodeBERT, the constructed bimodal input is first tokenized using its built-in BPE (Byte-Pair Encoding) tokenizer (Sennrich et al., 2016), which encodes each input at the subword level for better semantic understanding. The resulting token sequence is then mapped into semantic space through token and position embeddings, formulated as:

$$E_{\text{input}} = E_{\text{token}}(X_{\text{tok}}) + E_{\text{pos}}(X_{\text{tok}}), \tag{3}$$

where $X_{\text{tok}} = \text{Tokenizer}(X_{\text{input}})$ denotes the BPE-tokenized input.

The embedded sequence is passed through the multi-layer Transformer encoder of CodeBERT to model deep semantic and syntactic features. The output representation is given by:

$$E_{\text{output}} = \text{CodeBERT}(E_{\text{input}}) \in \mathbb{R}^{L \times d}, \tag{4}$$

We extract the hidden state at the `[CLS]` position as the final semantic feature:

$$E_{\text{final}} = E_{\text{output}}[\text{CLS}] \in \mathbb{R}^{d}. \tag{5}$$

This high-dimensional semantic vector encodes rich structural and semantic information, including logical patterns and potential defect cues, providing strong support for the downstream classification model.

3.2.3 Training and ensemble testing

Figure 3 illustrates the complete pipeline of the proposed defect prediction model, consisting of three stages: pre-training, training, and testing. In the training stage, the original labeled source dataset is first partitioned into twelve train-validation pairs using the BRHS strategy. Both the source (labeled) and target (unlabeled) code files—along with the test set—are then processed by CodeBERT to extract semantic feature representations, yielding labeled features from the source and unlabeled features from the target project.

In the pre-training stage, several datasets with known labels are used to construct training samples. Specifically, for the WPDP task, we adopt the earliest version of each Java project. For the CPDP task, the earliest versions of all projects, excluding the target project, are used. Feature vectors extracted by CodeBERT and their corresponding defect labels are then used to train CGAN, which learns to generate realistic synthetic samples.

As shown in the left part of Fig. 4, the CGAN consists of a generator G and a discriminator D, trained in an adversarial manner. The generator takes a noise vector z and a class label y

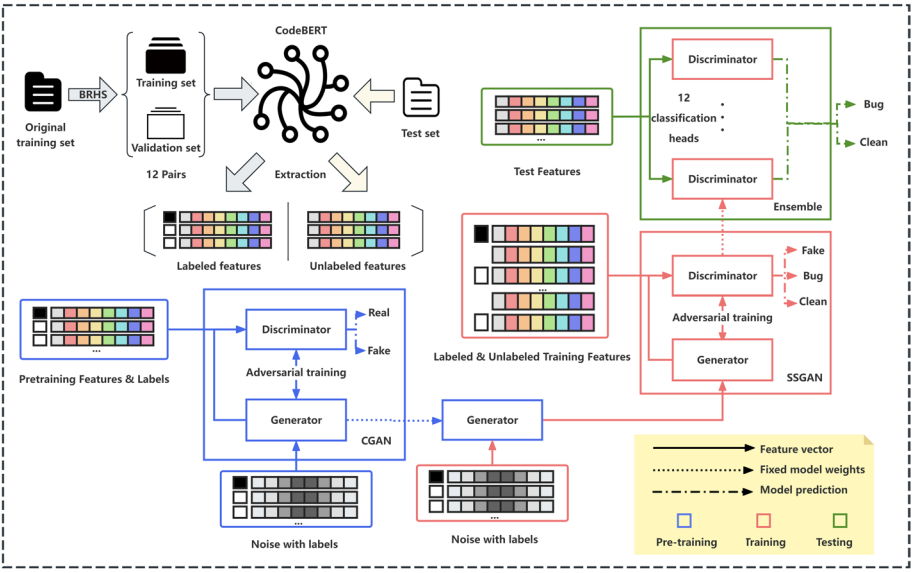

Fig. 3 Training and testing pipeline of the proposed model with CGAN-based pretraining, SSGAN-based semi-supervised training on BRHS-partitioned train-validation sets, and ensemble prediction

as input and generates a pseudo feature vector $G(\mathrm{z}, \mathrm{y})$ that mimics the real feature distribution. The discriminator receives either a real sample x or a generated sample, along with its label, and attempts to distinguish between them.The dimensionality of the input noise vector z is set to 100.

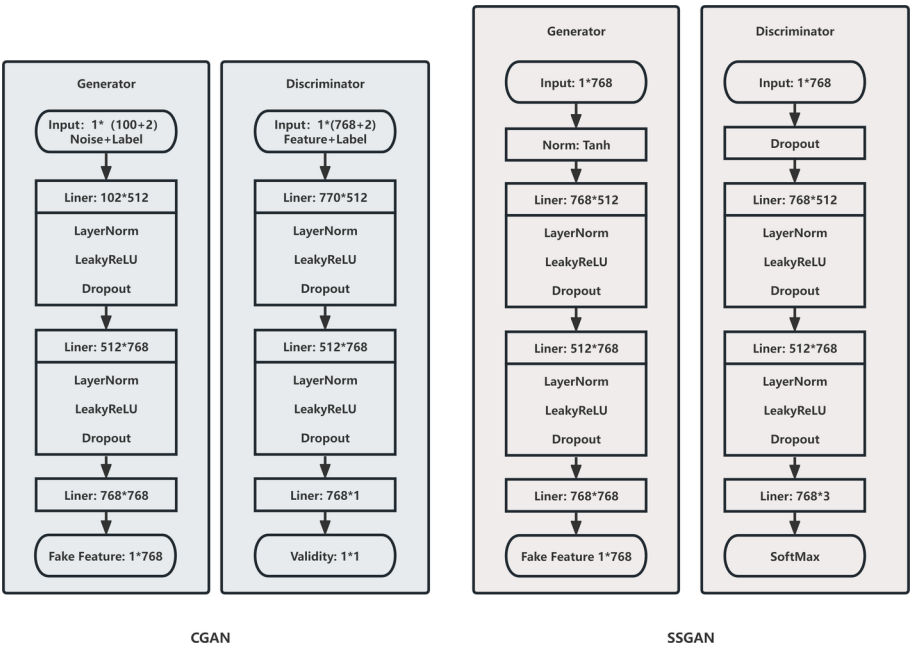

Fig. 4 Network structures of CGAN (left) and SSGAN (right)

This value was selected based on a sensitivity analysis over dimensions 50, 100, 150, 200, 250, 300, which showed that 100 yields the best average performance under our BRHS-based validation protocol, while also being consistent with standard configurations in GAN literature (Mirza & Osindero, 2014; Odena, 2016; Song et al., 2024).

The discriminator loss is defined as:

$$\mathcal{L}_D = -\mathbb{E}[\log D(\mathrm{x}, \mathrm{y})] - [\log(1 - D(G(\mathrm{z}, \mathrm{y}), \mathrm{y}))] \tag{6}$$

This encourages the discriminator to assign high probabilities to real labeled samples and low probabilities to generated ones.

Conversely, the generator aims to fool the discriminator:

$$\mathcal{L}_G = -\mathbb{E}[\log D(G(\mathrm{z}, \mathrm{y}), \mathrm{y})] \tag{7}$$

This drives the generator to produce synthetic features that are indistinguishable from real data under the same class label.

After CGAN convergence, its generator is frozen. During training, we sample new noise vectors with balanced class labels (1:1 defective/non-defective ratio) and feed them into the frozen generator to obtain pseudo-labeled features. These are then passed to the SSGAN for further refinement to better match the target project's feature distribution.

In the formal training stage (right part of Fig. 4), SSGAN builds the final defect predictor. The source project is partitioned using BRHS into multiple train-validation groups, ensuring each instance appears equally often in training sets and that training/validation splits maintain consistent overlap—thereby reducing partition bias. Each training set is resampled to a 1:1 defect ratio, while validation sets preserve the original class distribution.

During SSGAN training, the pretrained CGAN generator (with its weights fixed) takes random noise z and a conditional label $\mathrm{y}_{\mathrm{fake}}$ as input, and generates synthetic minority-class samples denoted as $(\mathrm{x}_{\mathrm{fake}}, \mathrm{y}_{\mathrm{fake}})$. These synthetic instances are then combined with real labeled source data (x, y) and unlabeled target data x to train the SSGAN discriminator. This design allows the model to jointly mitigate class imbalance through realistic minority augmentation and reduce cross-project distribution discrepancies via adversarial learning on both real and generated data.

The SSGAN generator's loss comprises three components:

$$\mathcal{L}_G = -\mathcal{L}_{\mathrm{adv}} - \mathcal{L}_{\mathrm{feat}} - \mathcal{L}_{\mathrm{cls}} \tag{8}$$

The three components are defined as follows:

(1) **Adversarial Loss:**

$$\mathcal{L}_{\mathrm{adv}} = \mathbb{E}\left[\log D(G(\mathrm{x}_{\mathrm{fake}}, \mathrm{y}_{\mathrm{fake}}))_{\mathrm{prob}}\right] \tag{9}$$

This pushes the generator to create features that the discriminator confidently classifies as "real."

(2) **Feature Regularization Loss:**

$$\mathcal{L}_{\text{feat}} = \|\mathbb{E}[D(\mathrm{x})_{\text{feat}}] - \mathbb{E}[D(G(\mathrm{x}_{\text{fake}}, \mathrm{y}_{\text{fake}}))_{\text{feat}}]\|_2^2 \tag{10}$$

This aligns the statistical moments of real and generated features in the discriminator's intermediate representation space, promoting distributional similarity.

(3) **Classification Loss:**

$$\mathcal{L}_{\text{cls}} = \text{CrossEntropy}(D(G(\mathrm{x}_{\text{fake}}, \mathrm{y}_{\text{fake}}))_{\text{logits}}, \mathrm{y}_{\text{fake}}) \tag{11}$$

This ensures that synthetic features retain correct semantic class information by enforcing accurate label prediction.

The SSGAN discriminator processes three types of inputs: labeled source data, unlabeled target data, and refined synthetic data. Its total loss is:

$$\mathcal{L}_D = -\mathcal{L}_{\text{sup}} - \mathcal{L}_{\text{unsup-real}} - \mathcal{L}_{\text{unsup-fake}} \tag{12}$$

(1) **Supervised Loss (for labeled real data):**

$$\mathcal{L}_{\text{sup}} = \text{CrossEntropy}(D(\mathrm{x})_{\text{logits}}, \mathrm{y}) \tag{13}$$

This trains the discriminator to correctly classify labeled defective and clean code samples.

(2) **Unsupervised Loss (for real and generated data):**

$$\mathcal{L}_{\text{unsup-real}} = \mathbb{E}\left[\log D(\mathrm{x})_{\text{prob}}\right] \tag{14}$$

This encourages the discriminator to assign high "realness" scores to all unlabeled (target) code features.

$$\mathcal{L}_{\text{unsup-fake}} = \mathbb{E}\left[\log\left(1 - D(G(\mathrm{x}_{\text{fake}}, \mathrm{y}_{\text{fake}}))_{\text{prob}}\right)\right] \tag{15}$$

This penalizes the discriminator if it fails to recognize synthetic features as "fake," strengthening its ability to distinguish real from generated data.

The complete training objective is:

$$\min_G \mathcal{L}_G \quad , \quad \min_D \mathcal{L}_D \tag{16}$$

To better align CodeBERT-extracted features with the defect prediction task, we embed the pre-trained CodeBERT Transformer encoder directly into the discriminator and jointly fine-tune its parameters during SSGAN training. This preserves CodeBERT's semantic modeling capacity while adapting it to capture defect-relevant patterns. After convergence, the entire discriminator—including the adapted CodeBERT encoder—is frozen and reused in the ensemble stage.

For ensemble evaluation, we leverage twelve such discriminators, each trained on a distinct train-validation split generated by the BRHS strategy. As shown in Table 1, the source project is partitioned into six groups; by alternating each group as validation set (with the rest as training), we obtain twelve unique training configurations and thus twelve fixed-weight discriminators.

During testing, the target project's code is first encoded by the fine-tuned CodeBERT to produce feature vectors. These features are then fed into all twelve frozen discriminators—now used purely as binary classifiers—to obtain defect probabilities. Finally, predictions are aggregated via a weighted ensemble, where each model's weight equals its F1 score on its respective validation set. The class with the highest aggregated probability is selected as the final prediction.

4 Experimental setup

4.1 Research questions

To evaluate the effectiveness of CoCo-GAN and understand the contribution of each component, we design the following research questions:

- **RQ1**: How does CoCo-GAN perform in within-project defect prediction tasks compared with baseline models?
- **RQ2**: How does CoCo-GAN perform in cross-project defect prediction tasks compared with baseline models?
- **RQ3**: How do the core components of CoCo-GAN—including CGAN-based data augmentation, SSGAN-based semi-supervised learning, and the BRHS strategy—contribute to addressing the key challenges of data scarcity, distribution discrepancy, and class imbalance across both WPDP and CPDP tasks?

4.2 Training configuration

All experiments are conducted on an NVIDIA A800 SXM4 80GB GPU and an Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz. The base weights of the CodeBERT pre-trained model are obtained from `microsoft/codebert-base` on HuggingFace[3], which follows the RoBERTa architecture.For training the adversarial generative networks, the following hyperparameters are used: the random seed is set to 64, batch size is 16, learning rate is 1×10^{-5}, dropout rate is 0.3, and the slope of LeakyReLU is 0.2. The training process employs the AdamW optimizer and a cosine annealing with warm restarts learning rate scheduler.

The Conditional Generative Adversarial Network is trained for 30 epochs on the training set to ensure convergence. For the Semi-Supervised Generative Adversarial Network, we set the maximum number of training epochs to 30 as well, but apply early stopping based on the performance on the validation set to avoid overfitting and reduce training time.

[3] https://huggingface.co/microsoft/codebert-base

4.3 Evaluation metrics

To evaluate the performance of the proposed defect prediction model, we adopt two widely used binary classification metrics: F1 score and AUC (Area Under the ROC Curve).

F1 score is the harmonic mean of precision and recall, particularly suitable for imbalanced datasets. Based on the confusion matrix (Table 3), precision (P) and recall (R) are computed as:

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN} \tag{17}$$

The F1 score is then calculated as:

$$F1 = \frac{2 \cdot P \cdot R}{P + R} \tag{18}$$

AUC measures the area under the ROC curve, which plots the true positive rate (TPR) against the false positive rate (FPR). It evaluates the model's ability to distinguish between defective and non-defective samples:

$$\text{AUC} = \int_0^1 \text{TPR}(\text{FPR})\, d\text{FPR} \tag{19}$$

where TPR and FPR are defined as:

$$\text{TPR} = \frac{TP}{TP + FN}, \quad \text{FPR} = \frac{FP}{FP + TN} \tag{20}$$

4.4 Baseline models

We selected five models as baseline models, including DBN (Wang et al., 2018), Tree-LSTM (Dam et al., 2019), DTL-DP (Li et al., 2019), MFGNN (Zhao et al., 2022), and the latest DP-CCL (Sahar et al., 2024). These models are all based on source code text for feature extraction and have been widely verified in cross-version and cross-project defect prediction tasks, thus having good comparability. The DBN model extracts token vectors from the abstract syntax tree and source code change metrics, and uses a Deep Belief Network to automatically extract features, followed by classification using logistic regression. Tree-LSTM parses the abstract syntax tree using JavaParser and inputs it into a tree-structured LSTM network. The output of the last root node is used for classification through logistic regression. It achieved good results on both WPDP and CPDP datasets. Both DTL-DP and MFGNN use CFG and AST, where DTL-DP further combines DFG to capture context dependencies, while MFGNN treats AST basic blocks as nodes of the CFG for

Table 3 Confusion matrix for binary classification

	Actual positive	Actual negative
Predicted positive	TP (True Positive)	FP (False Positive)
Predicted negative	FN (False Negative)	TN (True Negative)

further integration. Graph Attention Networks (GAT) are then used to extract features, followed by classification using logistic regression. DP-CCL extracts node information from AST and combines it with software metrics, utilizing supervised contrastive learning to capture semantic features from source code, followed by classification using logistic regression. Based on these existing defect prediction task studies, we adopt the same WPDP and CPDP experimental group design, and compare the prediction performance of these models. These baselines collectively cover various feature extraction paradigms explored in previous defect prediction studies.

5 Experimental results

To answer **Questions 1 and 2**, this study compares and analyzes the performance of five mainstream baseline models across multiple datasets, primarily using F1 score and AUC as evaluation metrics. F1 score is suitable for imbalanced datasets, measuring the model's ability to identify defect samples. AUC evaluates the model's ability to distinguish between positive and negative samples under varying thresholds, focusing on stability and generalization across complex settings like cross-project and cross-version tasks.

To ensure consistency between WPDP and CPDP experimental setups, we re-implemented both MFGNN and DP-CCL, and conducted comparative experiments under a unified evaluation protocol. It should be noted that in the original implementation of MFGNN, 30% of the target project's test set was used as a validation set to fine-tune the logistic regression classifier for cross-project defect prediction experiments, with the remaining 70% used for final prediction. However, in this study, to ensure consistency in experimental design and the rationality of generalization evaluation, we fully used the source project training set for fine-tuning, keeping the target project data unlabeled.

For the comparison results of the other baseline methods, we referenced the data provided in the original MFGNN paper, as their experimental group design aligns with ours. Since MFGNN did not provide results for Tree-LSTM in cross-project prediction, we used the data from the original paper that reported cross-project and cross-version experimental results for this model. Considering that the original paper used bar charts and did not provide specific values, we used the open-source tool WebPlotDigitizer[4] for image data extraction. This tool has been widely adopted in various studies (Burda et al., 2017; Drevon et al., 2017; Marin et al., 2017), and its extraction accuracy has been verified for reliability. The final extracted values are consistent with the general trends and average values described in the original paper, ensuring the credibility and reproducibility of the comparison data.

To thoroughly answer **Question 3**, this study designs and implements multiple ablation experiments to validate the specific contribution of different modules in the proposed model to the overall prediction performance. All experiments are conducted based on the complete CoCo-GAN framework. By systematically removing or replacing key components and comparing their F1 scores and AUC values across multiple datasets, we evaluate how each module affects the model's capability to handle data scarcity, distribution discrepancy, and class imbalance, as well as its performance consistency across both WPDP and CPDP tasks. The specific experimental settings are as follows:

[4] https://automeris.io/WebPlotDigitizer

- **CB-LR**: A baseline model that uses CodeBERT embeddings with a logistic regression classifier.
- **–CGAN**: Removes the CGAN module to evaluate the contribution of adversarial data generation to model performance.
- **Sup.**: Trains only on labeled data to assess the influence of semi-supervised learning.
- **–FT**: Disables joint fine-tuning of CodeBERT and uses its pretrained weights instead, to analyze the impact of feature adaptability.
- **–BRHS**: Replaces BRHS-based partitioning with random splits to examine its effect on distribution stability and cross-project consistency.
- **–Ens.**: Removes the ensemble voting mechanism and averages the discriminator outputs instead, to test the effect of ensemble voting.

5.1 RQ1 and RQ2

As shown in Tables 4 and 5, the proposed method, CoCo-GAN, demonstrates significant performance advantages in both WPDP and CPDP tasks. In the 15 cross-version prediction experiments presented in Table 4, CoCo-GAN achieves the highest average F1 score of 57.8, and in the 22 cross-project prediction settings shown in Table 5, CoCo-GAN also obtains the highest average F1 score of 53.1, indicating strong performance across both scenarios.The statistical "Win" counts further support CoCo-GAN's overall superiority and stability, where it outperforms other methods in the majority of cases.

Figure 5 presents the Scott-Knott boxplots for the AUC scores across WPDP and CPDP tasks. It clearly shows that CoCo-GAN outperforms competing models in terms

Table 4 Comparison of F1 scores between CoCo-GAN and baseline models on the WPDP task. Bold values indicate the highest F1 score in each experimental group. [∗] Proposed method in this work; [†] Reimplemented baselines; [‡] Results extracted from figures in the original papers; [§] Models trained using data provided in MFGNN

Train	Test	CoCo-GAN*	DP-CCL†	MFGNN†	Tree-LSTM‡	DTLDP§	DBN§
ant-1.5	ant-1.6	45.4	**55.1**	29.5	47.1	45.3	40.7
ant-1.6	ant-1.7	54.4	42.9	**54.7**	35.2	35.5	51.7
camel-1.2	camel-1.4	28.8	52.4	**53.5**	32.5	32.9	16.5
camel-1.4	camel-1.6	41.8	51.9	**53.7**	40.2	34.7	32.0
ivy-1.4	ivy-2.0	**39.5**	19.5	20.4	19.0	21.1	27.3
jedit-4.0	jedit-4.1	61.5	51.9	**62.6**	49.4	23.8	41.6
log4j-1.0	log4j-1.1	**77.9**	69.2	74.6	52.7	24.0	60.5
lucene-2.0	lucene-2.2	71.5	70.3	66.7	**75.2**	58.9	36.6
lucene-2.2	lucene-2.4	**76.2**	74.9	68.4	75.2	68.8	37.4
poi-1.5	poi-2.5	**83.1**	80.8	82.6	81.2	81.9	8.4
poi-2.5	poi-3.0	79.4	**82.5**	72.0	78.2	77.7	27.0
synapse-1.0	synapse-1.1	**50.3**	27.8	26.7	45.7	41.0	43.0
synapse-1.1	synapse-1.2	**58.2**	54.3	47.0	51.4	54.4	41.5
xalan-2.4	xalan-2.5	57.3	60.2	33.1	**66.1**	50.4	30.8
xerces-1.2	xerces-1.3	**41.7**	7.2	31.5	26.2	14.8	32.4
Avg		57.8	53.4	51.8	51.7	44.3	35.2
Win			10	11	11	14	15
P-Value			>0.16	<0.05	<0.05	<0.05	<0.05

Table 5 Comparison of F1 scores between CoCo-GAN and baseline models on the CPDP task. Bold values indicate the highest F1 score in each experimental group. [∗] Proposed method in this work; [†] Reimplemented baselines; [‡] Results extracted from figures in the original papers; [§] Models trained using data provided in MFGNN

Train	Test	CoCo-GAN*	DP-CCL†	MFGNN†	Tree-LSTM‡	DTLDP§	DBN§
ant-1.6	camel-1.4	**37.8**	36.0	33.0	32.2	22.8	31.9
ant-1.6	poi-3.0	70.1	**79.7**	60.1	78.9	33.3	43.5
camel-1.4	ant-1.6	**61.4**	45.2	45.1	45.2	47.8	56.1
camel-1.4	jedit-4.1	**53.6**	38.1	44.1	39.9	38.4	32.3
ivy-1.4	synapse-1.1	**51.4**	48.1	13.7	46.3	15.7	9.7
ivy-2.0	synapse-1.2	**57.7**	49.3	43.7	26.8	45.7	32.4
ivy-2.0	xerces-1.3	36.5	42.7	39.4	**53.7**	29.4	36.6
jedit-4.1	camel-1.4	**38.1**	15.0	20.2	32.1	31.3	23.4
jedit-4.1	log4j-1.1	**68.3**	58.5	42.6	58.2	59.6	37.8
log4j-1.1	jedit-4.1	**51.1**	44.1	45.9	39.2	39.9	48.4
log4j-1.1	lucene-2.2	64.7	64.9	57.8	75.4	**76.4**	52.7
lucene-2.2	log4j-1.1	61.3	**73.5**	66.7	58.4	46.5	45.2
lucene-2.2	xalan-2.5	66.1	64.6	54.4	**68.6**	37.8	57.2
poi-2.5	synapse-1.1	45.4	48.8	48.3	44.3	35.2	**49.0**
poi-3.0	ant-1.6	**56.5**	52.6	46.6	39.1	44.8	48.2
poi-3.0	synapse-1.2	**57.3**	56.7	27.4	50.9	29.4	49.5
synapse-1.2	ivy-2.0	**35.0**	33.7	20.4	26.8	22.0	29.6
synapse-1.2	poi-3.0	61.9	76.6	**81.0**	79.0	73.9	48.5
xalan-2.5	lucene-2.2	54.6	75.6	75.1	**75.7**	74.5	56.4
xalan-2.5	xerces-1.3	38.2	36.4	**46.3**	34.6	15.7	32.4
xerces-1.3	ivy-2.0	38.4	15.7	13.0	**68.2**	11.3	30.5
xerces-1.3	xalan-2.5	62.4	**65.6**	29.3	26.4	64.9	26.8
Avg		53.1	51.0	43.4	50.0	40.7	39.9
Win			14	16	15	18	19
P-Value			>0.16	<0.05	<0.16	<0.05	<0.05

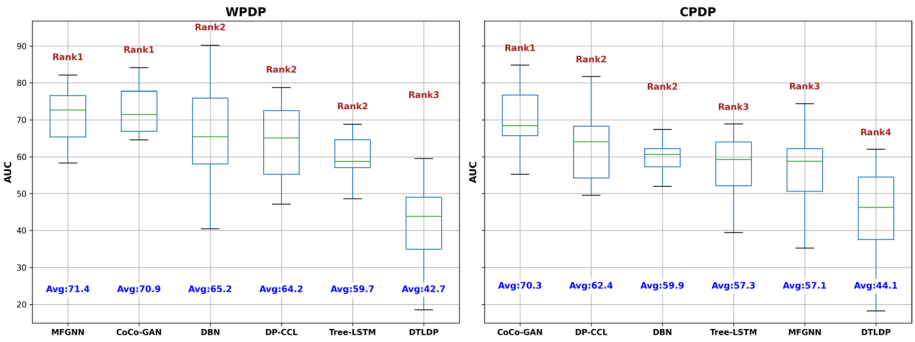

Fig. 5 Scott-Knott boxplots for the AUC scores across WPDP and CPDP tasks

of AUC, with a higher median and tighter variance range, highlighting its robust and stable predictive ability. This reflects CoCo-GAN's superior capability in modeling defect-related semantics and handling different data distributions.

Particularly noteworthy is that CoCo-GAN consistently achieves strong performance in challenging defect prediction scenarios, especially when class imbalance is severe or cross-project distribution shifts are pronounced. This strength stems from the coordinated operation of its core components: the bimodal CodeBERT encoder captures richer semantic cues by jointly modeling source code and comments; the CGAN generates high-quality synthetic defect samples to alleviate minority scarcity; the SSGAN leverages unlabeled target instances to align source and target feature distributions; and the BRHS-based ensemble strategy stabilizes training under limited labeled data. For instance, in cross-version tasks such as ivy-1.4 (6.64%) → ivy-2.0 (11.36%), xerces-1.2 (16.17%) → xerces-1.3 (15.27%), and notably synapse-1.0 (10.19%) → synapse-1.1 (27.03%)—where the target defect rate nearly triples—CoCo-GAN achieves the highest F1 scores among all methods. In these cases, CGAN effectively compensates for the extreme scarcity of defective samples in the source, while SSGAN reduces version-induced distribution drift using unlabeled target code, enabling robust knowledge transfer despite evolving defect prevalence.

This capability extends to cross-project settings with significant functional divergence. Consider the transfer from ivy (a build dependency manager focused on configuration resolution) to synapse (an enterprise service bus handling message mediation). Despite their distinct architectures and API ecosystems, CoCo-GAN outperforms all baselines. Here, the bimodal encoder proves critical: by fusing method-level comments with implementation logic, it extracts higher-level semantic patterns—such as error-handling intent or resource management—that transcend syntactic differences. Simultaneously, SSGAN's adversarial alignment on unlabeled target data mitigates domain shift, allowing the model to generalize across heterogeneous codebases where traditional feature-based or graph-based approaches falter.

That said, CoCo-GAN's effectiveness is bounded by the quality of semantic signals and the degree of structural alignment between projects. In the xalan-2.5 (50.97%) → lucene-2.2 (61.11%) task, it achieves only 54.6 F1—comparable to DBN and substantially below DP-CCL and MFGNN. This underperformance arises because Xalan (an XSLT processor) and Lucene (a search engine) share almost no common design idioms or library usage, and Lucene's defects are concentrated in low-level concurrency modules that lack informative comments. Consequently, the bimodal encoder receives weak supervision, and even CGAN cannot generate meaningful minority samples without reliable semantic anchors. Similarly, in ivy-2.0 (11.36%) → xerces-1.3 (15.27%), CoCo-GAN yields a low F1 of 36.5—below DP-CCL, Tree-LSTM, and MFGNN. The issue here lies in structural opacity: Xerces' parser relies on tightly coupled state transitions that are poorly reflected in linearized code-comment pairs, whereas tree- or graph-based models inherently capture such control-flow dependencies. These cases highlight that our sequence-based framework, while powerful in semantically rich settings, may struggle when projects are both functionally orthogonal and structurally non-compositional.

Even in adverse conditions, CoCo-GAN often exhibits behavior aligned with practical inspection priorities. For example, in camel-1.2 → camel-1.4, degraded comment quality—where method documentation is replaced by generic class banners—leads to high recall (89.2%) but low precision (17.3%), resulting in a modest F1. While this reflects a limitation in noisy signal contexts, the high recall ensures that most defective files are flagged, which is consistent with the principle that minimizing false negatives is often more critical than optimizing composite metrics in real-world defect triage (Dam et al., 2019).

5.2 RQ3

Tables 6 and 7 present the performance of the proposed method and its ablation variants on the WPDP and CPDP tasks, respectively, including F1 score, precision, recall, and average AUC. Based on a comparison with the complete model, the following observations can be made:

- **CB-LR Performance:** The CB-LR model, combining CodeBERT embeddings with a logistic regression classifier, already shows strong results (average F1: WP = 53.7, CP = 45.1). In WPDP, it even surpasses most traditional baselines. This confirms that CodeBERT's bimodal encoding effectively captures syntactic and semantic code features, providing powerful representations even with a simple linear classifier.
- **Addressing Data Scarcity and Class Imbalance:** Comparing the complete CoCo-GAN model with the variant without the CGAN module (–CGAN) reveals a performance decline of 1.7 in WP and 1.5 in CP average F1. This demonstrates that pretraining with a conditional GAN to generate label-conditioned and class-balanced pseudo feature vectors effectively improves prediction accuracy. The generated synthetic samples enrich the representation of minority classes and compensate for limited labeled data, thereby enhancing the model's generalization ability across both within-project and cross-project prediction.

  This benefit is consistently observed across diverse defect rate configurations. In CPDP, CoCo-GAN outperforms –CGAN when transferring from a low-defect source to a higher-defect target—e.g., ivy-2.0 (11.36%) → synapse-1.2 (33.59%): 57.7 vs. 56.5—and also when predicting a lower-defect target from a high-defect source—e.g., poi-3.0 (64.16%) → synapse-1.2 (33.59%): 57.3 vs. 55.7. Similarly, in WPDP, where version-to-version defect rates can shift dramatically, CoCo-GAN shows robust gains: for ant-1.5 (10.69%) → ant-1.6 (26.29%), F1 improves from 42.0 to 45.4; even more strikingly, for synapse-1.0 (10.19%) → synapse-1.1 (27.03%), F1 jumps from 40.4 to 50.3. These results confirm that the CGAN component effectively mitigates class imbalance and data scarcity regardless of whether the target project/version has higher or lower defect prevalence than the source, making it broadly applicable across realistic software evolution and reuse scenarios.
- **Incorporating Unlabeled Data for Domain Alignment:** Compared with the supervised-only variant (Sup.), CoCo-GAN improves F1 scores by 1.4 on WPDP and 0.4 on CPDP, indicating that incorporating unlabeled target samples via semi-supervised adversarial learning enhances domain alignment. In our SSGAN framework, the discriminator is trained to distinguish source from target features, while the generator (i.e., the feature encoder) is optimized to fool it—effectively minimizing the divergence between source and target feature distributions. We adopt the Wasserstein distance as a stable and meaningful metric to quantify this discrepancy, as it reflects the minimal "cost" of transforming one distribution into another in the feature space and correlates well with GAN training stability.

  Crucially, reducing this distance directly benefits defect prediction under domain shift: when source and target features occupy similar regions in the embedding space, the classifier trained on labeled source data can generalize more reliably to

Table 6 Precision, Recall, and F1 scores of different ablation versions of CoCo-GAN on the WPDP task. Bold values indicate the highest F1 score in each experimental group

Train	Test	CoCo-GAN			CB-LR			–CGAN			Sup.			–FT			–BRHS			–Ens.		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
ant-1.5	ant-1.6	45.4	65.3	34.8	45.6	59.7	37.0	42.0	63.0	31.5	44.9	67.4	33.7	**57.2**	42.4	88.0	54.5	61.6	48.9	46.2	55.3	43.9
ant-1.6	ant-1.7	54.4	39.2	89.2	52.3	41.0	72.3	53.9	39.0	87.3	**54.5**	39.5	88.0	50.5	35.6	86.7	54.3	39.2	88.6	52.8	38.4	84.9
camel-1.2	camel-1.4	28.8	17.3	86.2	**44.1**	33.8	63.5	27.4	17.5	62.8	27.1	16.8	69.0	29.2	17.1	100.0	18.1	11.6	41.4	24.9	15.5	67.3
camel-1.4	camel-1.6	41.8	36.5	48.9	**46.8**	41.4	53.7	44.6	42.5	46.8	44.3	42.4	46.3	35.9	24.3	69.1	39.1	38.3	39.9	39.4	34.1	50.4
ivy-1.4	ivy-2.0	39.5	25.8	85.0	39.6	31.8	52.5	**40.5**	26.6	85.0	27.3	20.8	40.0	36.8	23.4	85.0	40.5	26.6	85.0	28.3	23.0	46.7
jedit-4.0	jedit-4.1	61.5	46.7	89.9	63.0	55.9	72.2	62.9	52.5	78.5	**63.2**	50.8	83.5	46.8	30.8	97.5	61.8	50.4	79.7	57.0	45.4	79.2
log4j-1.0	log4j-1.1	**77.9**	75.0	81.1	71.2	72.2	70.3	76.3	74.4	78.4	77.9	75.0	81.1	52.5	35.6	100.0	74.7	73.7	75.7	72.5	66.8	79.7
lucene-2.0	lucene-2.2	71.5	67.9	75.5	62.9	75.5	53.9	71.1	70.8	71.3	71.9	70.5	73.4	**76.3**	62.0	99.3	70.6	70.6	70.6	71.2	70.0	72.7
lucene-2.2	lucene-2.4	76.2	61.5	100.0	57.3	65.0	51.2	76.8	64.6	94.6	76.4	66.8	89.2	**76.6**	62.4	99.0	75.7	64.7	91.1	73.1	66.1	83.3
poi-1.5	poi-2.5	83.1	75.6	92.3	79.0	84.4	74.2	83.6	77.8	90.3	84.7	75.8	96.0	83.3	73.8	95.6	**85.1**	78.3	93.1	81.8	74.7	91.2
poi-2.5	poi-3.0	79.4	71.6	89.0	73.5	80.5	67.6	77.6	73.6	82.2	77.9	75.2	80.8	**81.2**	70.8	95.0	77.4	72.6	82.9	76.7	71.6	83.5
synapse-1.0	synapse-1.1	50.3	38.3	73.3	40.0	46.7	35.0	40.4	47.7	35.0	**52.3**	43.0	66.7	50.3	37.0	78.3	42.0	42.4	41.7	46.7	39.6	62.2
synapse-1.1	synapse-1.2	**58.2**	45.7	80.2	44.4	47.4	41.9	52.7	50.0	55.8	53.7	47.0	62.8	56.7	43.5	81.4	51.3	45.1	59.3	52.3	44.3	67.2
xalan-2.4	xalan-2.5	57.3	63.8	51.9	49.4	67.1	39.0	57.4	64.2	51.9	**57.6**	63.6	52.7	60.1	58.8	61.5	56.9	63.0	51.9	55.2	63.0	50.2
xerces-1.2	xerces-1.3	**41.7**	40.0	43.5	37.0	32.3	43.5	33.9	40.8	29.0	32.8	40.4	27.5	27.5	15.9	100.0	35.3	42.0	30.4	31.7	31.6	36.6
F1 Avg		57.8			53.7			56.1			56.4			54.7			55.8			54.0		
AUC Avg		70.9			73.0			72.0			71.9			65.0			71.0			68.8		
F1 Win					10			9			6			8			11			14		
P-Value					<0.05			>0.07			>0.16			>0.18			<0.05			<0.05		

Table 7 Precision, Recall, and F1 scores of different ablation versions of CoCo-GAN on the CPDP task. Bold values indicate the highest F1 score in each experimental group

Train	Test	CoCo-GAN			CB-LR			–CGAN			Sup.			–FT			–BRHS			–Ens.		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
ant-1.6	camel-1.4	37.8	30.3	50.3	34.4	29.1	42.1	**38.7**	30.9	51.7	38.1	30.5	51.0	36.2	27.9	51.7	37.7	30.4	49.7	37.5	29.6	51.3
ant-1.6	poi-3.0	70.1	81.9	61.2	69.8	85.9	58.7	70.3	82.0	61.6	69.9	81.5	61.2	**77.4**	81.5	73.7	69.4	82.0	60.1	69.9	82.1	61.1
camel-1.4	ant-1.6	**61.4**	47.1	88.0	57.5	42.6	88.0	58.2	43.7	87.0	54.2	45.9	66.3	54.8	39.0	92.4	58.4	43.4	89.1	51.9	38.8	80.7
camel-1.4	jedit-4.1	**53.6**	40.0	81.0	47.7	39.8	59.5	51.9	38.4	79.7	45.4	36.7	59.5	48.7	32.5	97.5	49.2	35.8	78.5	44.5	33.8	68.5
ivy-1.4	synapse-1.1	51.4	38.7	76.7	46.1	38.0	58.3	51.4	38.7	76.7	51.2	38.1	78.1	49.7	36.4	78.3	**51.7**	38.5	78.3	48.4	36.8	71.7
ivy-2.0	synapse-1.2	**57.7**	47.1	74.4	46.9	46.1	47.7	56.5	46.0	73.3	57.4	46.7	74.4	56.8	44.7	77.9	57.7	47.1	74.4	53.4	45.3	67.2
ivy-2.0	xerces-1.3	36.5	26.1	60.9	**47.3**	50.0	44.9	36.7	26.3	60.9	37.4	27.6	58.0	35.6	24.7	63.8	37.3	27.2	59.4	36.1	26.1	59.8
jedit-4.1	camel-1.4	38.1	28.9	55.9	27.1	26.0	28.3	38.9	29.9	55.9	**39.6**	30.7	55.3	38.1	28.7	56.6	39.1	29.9	56.6	36.7	29.1	52.0
jedit-4.1	log4j-1.1	68.3	62.2	75.7	62.7	70.0	56.8	65.9	60.0	73.0	**71.6**	62.3	83.9	70.8	57.6	91.9	67.4	59.2	78.4	70.3	65.0	77.0
log4j-1.1	jedit-4.1	51.1	36.4	86.1	49.2	36.1	77.2	51.1	36.6	84.8	53.2	37.8	89.9	**52.1**	36.1	93.7	50.8	36.2	84.8	52.0	37.6	85.1
log4j-1.1	lucene-2.2	64.7	69.9	60.1	59.4	69.8	51.8	64.1	70.6	58.7	67.2	72.0	62.9	**67.9**	71.9	64.3	63.8	70.9	58.0	63.4	70.0	58.9
lucene-2.2	log4j-1.1	61.3	45.9	91.9	63.4	57.8	70.3	52.2	36.1	94.6	**63.6**	48.6	91.9	52.5	35.6	100.0	54.8	39.1	91.9	59.8	44.3	94.4
lucene-2.2	xalan-2.5	66.1	53.9	85.5	38.9	55.5	30.0	63.4	51.6	82.2	66.9	56.1	82.9	**67.7**	51.4	99.2	65.8	54.3	83.5	65.7	53.0	87.7
poi-2.5	synapse-1.1	45.4	30.3	90.0	11.9	20.8	8.3	**46.5**	31.5	88.3	45.3	30.2	89.8	44.4	29.3	91.7	46.2	31.0	90.0	45.7	30.9	88.1
poi-3.0	ant-1.6	56.5	40.3	94.6	21.7	37.8	15.2	**57.3**	41.3	93.5	54.5	38.2	94.6	52.0	35.4	97.8	57.0	40.8	94.6	55.4	39.6	92.9
poi-3.0	synapse-1.2	**57.3**	43.2	84.9	37.8	45.2	32.6	55.7	42.0	82.6	56.5	42.4	83.7	57.1	41.7	90.7	55.6	42.2	81.4	56.2	42.1	84.9
synapse-1.2	ivy-2.0	35.0	24.3	62.5	31.6	20.6	67.5	35.4	23.4	72.5	**37.6**	24.1	85.0	27.8	16.4	92.5	35.8	23.3	77.5	30.7	20.4	65.0
synapse-1.2	poi-3.0	61.9	81.9	49.8	66.1	84.9	54.1	64.6	82.8	53.0	74.7	80.9	69.4	**81.9**	78.1	86.1	70.3	80.6	62.3	60.3	80.3	49.7
xalan-2.5	lucene-2.2	54.6	64.2	47.6	43.2	65.7	32.2	35.9	67.3	24.5	37.1	70.6	25.2	**77.0**	62.8	99.3	62.3	65.4	59.4	59.3	64.1	60.1
xalan-2.5	xerces-1.3	**38.2**	24.7	84.1	33.3	24.5	52.2	36.7	23.0	91.3	34.0	21.2	85.5	26.5	15.3	100.0	35.6	22.7	82.6	33.6	21.7	78.3
xerces-1.3	ivy-2.0	38.4	24.8	85.0	38.4	32.2	47.5	**42.5**	28.3	85.0	41.7	32.0	60.0	34.1	21.2	87.5	36.2	23.0	85.0	35.3	23.6	72.1
xerces-1.3	xalan-2.5	62.4	62.0	62.8	57.9	61.7	54.5	61.5	60.1	63.0	63.4	59.5	67.7	**65.1**	59.4	72.1	64.6	60.9	68.7	58.7	61.1	57.4
F1 Avg		53.1			45.1			51.6			52.7			53.4			53.0			51.1		
AUC Avg		70.3			68.6			70.5			70.3			67.9			70.1			66.9		
F1 Win					19			11			10			14			12			18		
P-Value					<0.05			>0.08			>0.24			>0.28			>0.29			<0.05		

unlabeled target instances. This is especially important in cross-project settings, where differences in coding style, API usage, or module structure often cause the decision boundary learned on the source to misalign with the true defect distribution in the target—leading to poor recall or precision. By aligning the marginal feature distributions via adversarial training on unlabeled target data, CoCo-GAN mitigates this misalignment, enabling the classifier to transfer its knowledge more effectively. As shown in Table 8, CoCo-GAN generally achieves smaller Wasserstein distances than Sup., and this reduction consistently corresponds to higher F1 scores. For example, in xalan-2.5 → lucene-2.2, the distance decreases substantially, and F1 rises from 37.1 to 54.6; similarly, in camel-1.4 → ant-1.6, reduced discrepancy yields a 7.2-point F1 gain. Conversely, in the few cases where the distance slightly increases (e.g., ant-1.6 → camel-1.4), F1 marginally drops. Across the 22 CPDP tasks, CoCo-GAN achieves lower Wasserstein distance in 19 settings, and 13 of these show concurrent F1 improvement, revealing a strong negative correlation. This pattern confirms that feature distribution alignment—enabled by unlabeled target data—is a key mechanism through which CoCo-GAN improves robustness in cross-project defect prediction.

- **Domain Adaptation of CodeBERT:** Disabling joint fine-tuning of CodeBERT (–FT) leads to a significant drop of 3.1 in WP F1, indicating that domain-adaptive fine-tuning helps capture project-specific semantics and improves within-project prediction. However, for CPDP, the fine-tuned model slightly underperforms compared to the non-fine-tuned version. This suggests that while fine-tuning enhances specialization for similar domains, it may also weaken the original generalization capability of CodeBERT, thereby reducing performance when detecting defects in heterogeneous target projects trained on different source distributions.
- **Effect of BRHS and Ensemble:** Replacing the BRHS-based partitioning strategy with random splits (–BRHS) causes a 2.0-point F1 decrease in WP while having

Table 8 Wasserstein Distance between source and target project feature distributions in CPDP tasks. Smaller WD indicates a closer feature distribution between domains. Bold values denote the smaller WD in each comparison

Model	**ant-1.6→poi-3.0**	**synapse-1.2→poi-3.0**	**ant-1.6→camel-1.4**	**jedit-4.1→camel-1.4**
CoCo-GAN	**10.4395**	**11.7311**	29.6867	29.2633
Sup.	10.4578	17.0444	**22.3616**	**28.8837**
Method	**xerces-1.3→xalan-2.5**	**xalan-2.5→lucene-2.2**	**log4j-1.1→lucene-2.2**	**xalan-2.5→xerces-1.3**
CoCo-GAN	**18.5030**	**27.7522**	**11.6049**	**25.7260**
Sup.	28.4579	38.1981	18.2449	27.1038
Method	**camel-1.4→jedit-4.1**	**log4j-1.1→jedit-4.1**	**jedit-4.1→log4j-1.1**	**lucene-2.2→log4j-1.1**
CoCo-GAN	**17.2484**	**16.1614**	**15.7105**	8.5589
Sup.	27.8166	31.1936	17.3481	**8.5326**
Method	**ivy-1.4→synapse-1.1**	**poi-2.5→synapse-1.1**	**ivy-2.0→synapse-1.2**	**poi-3.0→synapse-1.2**
CoCo-GAN	**12.9438**	**16.9287**	**14.6589**	**26.2829**
Sup.	16.7939	21.5420	17.9846	31.4689
Method	**camel-1.4→ant-1.6**	**poi-3.0→ant-1.6**	**synapse-1.2→ivy-2.0**	**lucene-2.2→xalan-2.5**
CoCo-GAN	**16.9885**	**21.3198**	**19.7819**	**7.0551**
Sup.	28.5554	24.4548	20.2085	7.8512
Method	**xerces-1.3→ivy-2.0**	**ivy-2.0→xerces-1.3**		
CoCo-GAN	**10.8184**	**7.6697**		
Sup.	12.6002	12.7296		

little impact on CP. This shows that BRHS preserves the data's inherent defect ratio and overlap structure, improving stability in cross-version prediction where inter-version continuity exists. Removing the ensemble voting mechanism (–Ens.) and instead averaging classifier outputs further degrades performance (–3.8 WP, –2.0 CP). This confirms that combining multiple discriminators trained on different BRHS partitions through ensemble voting significantly enhances robustness and discriminative consistency.

The ablation study shows that several components yield statistically significant gains: CoCo-GAN outperforms CB-LR, –BRHS, and –Ens., confirming the value of our architecture, structure-aware BRHS partitioning, and ensemble voting. Disabling CodeBERT fine-tuning (–FT) also causes a significant 3.1-point drop in WPDP ($p < 0.05$), though it slightly benefits CPDP—suggesting a trade-off between specialization and generalization.

Furthermore, one-tailed Wilcoxon signed-rank tests were conducted to assess statistical significance. CoCo-GAN achieves significant improvements ($p<0.05$) over most baselines—particularly DTLDP, DBN, and MFGNN—in both WPDP and CPDP tasks. Although the gains over DP-CCL and Tree-LSTM are not statistically significant ($p>0.16$), CoCo-GAN consistently attains higher average F1 scores and secures the largest number of "wins" across nearly all datasets, indicating that its performance advantages are both stable and practically meaningful.

In contrast, the average improvements from CGAN (+1.7/+1.5 F1) and semi-supervised learning (+1.4/+0.4 F1) are not statistically significant ($p>0.07$). However, in software defect prediction, even modest average gains can reflect substantial practical impact, especially when they arise from improved robustness across heterogeneous projects or under extreme data conditions. For instance, while the average F1 gain from CGAN appears small, it prevents catastrophic failure in high-imbalance scenarios: on synapse-1.0 → synapse-1.1 (target defect rate: 27.03%), CGAN yields a +9.9 F1 improvement, directly enabling the detection of critical defects that would otherwise be missed. Similarly, semi-supervised learning reduces domain discrepancy in 19 out of 22 CPDP tasks, stabilizing performance when labeled target data is absent—a common real-world constraint.

Moreover, CoCo-GAN with CGAN wins on 9/15 WPDP and 11/22 CPDP tasks against its ablated variant, and with semi-supervision wins on 6/15 and 10/22 tasks, indicating consistent directional benefit. Given that false negatives in defect prediction often incur far higher inspection or failure costs than false positives, any mechanism that reliably improves minority-class recall—even at the cost of slight precision loss—can translate into meaningful savings in testing effort or risk mitigation. Thus, while the aggregate F1 differences may appear modest, their distributional and contextual effects align closely with practical software quality assurance priorities.

6 Threats to validity

This section outlines two main threats to the validity of our study: potential differences in experimental reproduction and the limited generalizability of our method to other programming languages.

6.1 Reproducibility issues

Despite detailed descriptions of experimental settings, parameters, and data preprocessing, some implementation discrepancies are inevitable. For example, DP-CCL and MFGNN were reimplemented based on their open-source code and papers, but certain training details (e.g., feature node selection, parameter choices) were not fully disclosed. For the Tree-LSTM model, since MFGNN did not report results for cross-project settings, we extracted them from the original paper using WebPlotDigitizer, which may introduce minor deviations. These factors may slightly affect the accuracy of comparisons.

6.2 Language generalizability

All experiments are limited to Java projects from the PROMISE dataset, which constitutes a threat to external validity regarding language generalizability. That said, CoCo-GAN builds upon CodeBERT—a foundation model pretrained on a large multilingual code corpus (including Python, JavaScript, Ruby, Go, and PHP)—which has demonstrated transferable semantic understanding across languages in diverse code intelligence tasks (Abid et al., 2023; Feng et al., 2020; Zhou et al., 2021). Since our method relies on CodeBERT's bimodal code-comment encoding as its semantic core, the architecture is in principle extensible to other languages supported by the model. However, practical deployment would require language-specific adaptations in parsing, comment extraction, and data preprocessing due to differences in syntax, typing, and documentation conventions. We leave such cross-language validation as an important direction for future work

6.3 Computational cost and scalability

CoCo-GAN incurs significant computational overhead: it requires CGAN pretraining (4M parameters) followed by twelve sequentially trained SSGAN models—one per BRHS-derived split—each involving a fine-tuned CodeBERT encoder (476M parameters) and a 3.1M-parameter discriminator, trained up to 30 epochs with early stopping. All twelve models are retained for ensemble prediction, resulting in an average training time of 40 minutes per project on two NVIDIA A800 GPUs. This cost may hinder deployment in time-sensitive or resource-constrained scenarios.

7 Conclusion and future work

This paper addresses three key challenges in software defect prediction: scarce labeled data, class imbalance, and distribution shifts in cross-version and cross-project settings. We propose CoCo-GAN, a method that integrates CodeBERT with conditional and semi-supervised GANs. CodeBERT captures syntactic and semantic features from code and comments; CGAN augments minority-class samples, while SSGAN aligns source and target distributions using unlabeled data. Combined with BRHS-based partitioning and ensemble learning, our approach achieves superior F1 and AUC scores across WPDP and CPDP tasks, demonstrating strong accuracy and generalization.

Our study is currently limited to Java-based defect prediction using the PROMISE dataset. To enhance generalizability, future work will extending CoCo-GAN to other programming languages and evaluating its effectiveness on Python, JavaScript, or C++ projects is a key direction for future research. This extension will also involve evaluating the effectiveness of alternative pre-trained models beyond CodeBERT for defect prediction in different programming languages.

Separately, to address the computational overhead, we plan to improve training efficiency through three directions: (1) designing lightweight generator and discriminator architectures; (2) applying parameter-efficient fine-tuning (e.g., LoRA, adapter modules) or progressive layer-wise freezing to minimize redundant updates in the large pre-trained encoder; and (3) developing intelligent data selection strategies that restrict adversarial training to high-utility or representative samples, thereby accelerating convergence without sacrificing performance.

Author Contributions Xiaoxing Yang proposed the research direction and supervised the entire research process. Liwei Xiao put forward the preliminary implementation idea, completed the specific experiments, and wrote the first draft of the paper. Jianmin Su and Bingding Huang provided experimental conditions for the study and supported the experimental verification work. All authors contributed to the revision of the paper, and reviewed and approved the final manuscript.

Funding This work is supported by Shenzhen Science and Technology Program (Grant No. 20220715114836001).

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

Arar, Ö. F., & Ayan, K. (2017). A feature dependent naive bayes approach and its application to the software defect prediction problem. *Applied Soft Computing, 59*, 197–209.

Alqarni, A., & Aljamaan, H. (2023). Leveraging ensemble learning with generative adversarial networks for imbalanced software defects prediction. *Applied Sciences, 13*(24), 13319.

Abid, S., Cai, X., & Jiang, L. (2023). Interpreting CodeBERT for semantic code clone detection. In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, (pp. 229–238). https://doi.org/10.1109/APSEC60848.2023.00033.

Abdu, A., Zhai, Z., Algabri, R., Abdo, H. A., Hamad, K., & Al-antari, M. A. (2022). Deep learning-based software defect prediction via semantic key features of source code-systematic survey. *Mathematics, 10*(17), 3120.

Abdu, A., Zhai, Z., Abdo, H. A., Algabri, R., Al-Masni, M. A., Muhammad, M. S., & Gu, Y. H. (2024). Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model. *Scientific Reports, 14*(1), 14771.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761. https://doi.org/10.1109/32.544352

Abreu, F., & Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. In: *Proceedings of the 3rd International Software Metrics Symposium*, (pp. 90–99). doi: https://doi.org/10.1109/METRIC.1996.492446.

Bai, J., Jia, J., & Capretz, L. F. (2022). A three-stage transfer learning framework for multi-source cross-project software defect prediction. *Information and Software Technology, 150*, 106985.

Borandag, E. (2023). Software fault prediction using an RNN-based deep learning approach and ensemble machine learning techniques. *Applied Sciences, 13*(3), 1639.

Burda, B. U., O'Connor, E. A., Webber, E. M., Redmond, N., & Perdue, L. A. (2017). Estimating data from figures with a web-based program: Considerations for a systematic review. *Research Synthesis Methods, 8*(3), 258–262.

Chen, J., Hu, K., Yu, Y., Chen, Z., Xuan, Q., Liu, Y., & Filkov, V. (2020). Software visualization and deep transfer learning for effective software defect prediction. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, (pp. 578–589).

Drevon, D., Fursa, S. R., & Malcolm, A. L. (2017). Intercoder reliability and validity of WebPlotDigitizer in extracting graphed data. *Behavior Modification, 41*(2), 323–339.

Deng, J., Lu, L., & Qiu, S. (2020). Software defect prediction via LSTM. *IET Software, 14*(4), 443–450.

Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T., Kim, C.-J. (2019). Lessons learned from using a deep tree-based model for software defect prediction in practice. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, (pp. 46–57). IEEE.

Dam, T., Izadi, M., Deursen, A. (2023). Enriching source code with contextual data for code completion models: An empirical study. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, (pp. 170–182). https://doi.org/10.1109/MSR59073.2023.00035.

Fan, G., Diao, X., Yu, H., Yang, K., & Chen, L. (2019). Software defect prediction via attention-based recurrent neural network. *Scientific Programming, 2019*(1), 6230953.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*, (pp. 1536–1547).

Fang, D., Liu, S., & Liu, A. (2022). Gated homogeneous fusion networks with jointed feature extraction for defect prediction. *IEEE Transactions on Reliability, 71*(2), 512–526.

Fu, M., & Tantithamthavorn, C. (2022). LineVul: A transformer-based line-level vulnerability prediction. In: *Proceedings of the 19th International Conference on Mining Software Repositories*, (pp. 608–620).

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (pp. 7212–7225).

Gupta, M., Rajnish, K., & Bhattacharjee, V. (2022). Cognitive complexity and graph convolutional approach over control flow graph for software defect prediction. *IEEE Access, 10*, 108870–108894.

Hassan, A.E. (2009). Predicting faults using the complexity of code changes. In: *2009 IEEE 31st International Conference on Software Engineering*, (pp. 78–88). doi: https://doi.org/10.1109/ICSE.2009.5070510.

Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, (pp. 1–10).

Khleel, N. A. A., & Nehéz, K. (2023). A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method. *Journal of Intelligent Information Systems, 60*(3), 673–707.

Khleel, N. A. A., & Nehéz, K. (2024). Software defect prediction using a bidirectional LSTM network combined with oversampling techniques. *Cluster Computing, 27*(3), 3615–3638.

Liu, J., Ai, J., Lu, M., Wang, J., & Shi, H. (2023). Semantic feature learning for software defect prediction from source code and external knowledge. *Journal of Systems and Software, 204*, 111753.

Li, J., He, P., Zhu, J., & Lyu, M.R. (2017). Software defect prediction via convolutional neural network. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, (pp. 318–328). IEEE.

Li, Z., Niu, J., & Jing, X.-Y. (2024). Software defect prediction: Future directions and challenges. *Automated Software Engineering, 31*(1), 19.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). *Roberta: A robustly optimized BERT pretraining approach*. arXiv:1907.11692.

Li, Y., Wang, S., Nguyen, T.N., & Van Nguyen, S. (2019). Improving bug detection via context-based code representation learning and attention-based neural networks. In: *Proceedings of the ACM on Programming Languages. 3(OOPSLA)*, 1–30.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering, 4*, 308–320.

Majumder, S., Chakraborty, J., & Menzies, T. (2024). When less is more: on the value of "co-training" for semi-supervised software defect predictors. *Empirical Software Engineering*, *29*(2), 51.

Mirza, M., & Osindero, S. (2014). *Conditional generative adversarial nets*. arXiv:1411.1784.

Marin, F., Rohatgi, A., & Charlot, S. (2017). WebPlotDigitizer, a polyvalent and free software to extract spectra from old astronomical publications: application to ultraviolet spectropolarimetry. In: *SF2A-2017: Proceedings of the Annual Meeting of the French Society of Astronomy and Astrophysics*.

Malhotra, R., & Singh, P. (2024). CodeBERT-BiGRU for software defect prediction. In: *International Conference on Artificial Intelligence and Speech Technology*, (pp. 277–289). Springer.

Odena, A. (2016). *Semi-supervised learning with generative adversarial networks*. arXiv:1606.01583.

Okumoto, K. (2016). Experience report: Practical software availability prediction in telecommunication industry. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, (pp. 321–331). IEEE.

Pan, C., Lu, M., & Xu, B. (2021). An empirical study on software defect prediction using CodeBERT model. *Applied Sciences*. *11*(11).

Qiu, S., E. B., He, J., & Liu, L. (2025). Survey of software defect prediction features. *Neural Computing and Applications*, *37*(4), 2113–2144.
Qiu, S., Huang, H., Jiang, W., Zhang, F., & Zhou, W. (2023). Defect prediction via tree-based encoding with hybrid granularity for software sustainability. *IEEE Transactions on Sustainable Computing*.
Shippey, T., Bowes, D., & Hall, T. (2019). Automatically identifying code features for software defect prediction: Using AST N-grams. *Information and Software Technology, 106*, 142–160.
Shen, Z., & Chen, S. (2020). A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks, 2020*(1), 8858010.
Song, W., Gan, L., & Bao, T. (2024). Software defect prediction via generative adversarial networks and pre-trained model. *International Journal of Advanced Computer Science & Applications*, *15*(3).
Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (pp. 1715–1725).
Shi, K., Lu, Y., Chang, J., & Wei, Z. (2020). PathPair2Vec: An AST path pair-based code representation method for defect prediction. *Journal of Computer Languages, 59*, 100979.
Sahar, S., Younas, M., Khan, M. M., & Sarwar, M. U. (2024). DP-CCL: A supervised contrastive learning approach using CodeBERT model in software defect prediction. *IEEE Access, 12*, 22582–22594.
Thomas, N. S., & Kaliraj, S. (2024). An improved and optimized random forest based approach to predict the software faults. *SN Computer Science, 5*(5), 530.
Uddin, M. N., Li, B., Ali, Z., Kefalas, P., Khan, I., & Zada, I. (2022). Software defect prediction employing BiLSTM and BERT-based semantic feature. *Soft Computing, 26*(16), 7877–7891.
Vasamsetty, C., Kadiyala, B., et al. (2022). Leveraging GraphCodeBERT for enhanced bug prediction and code quality analysis in software development using machine learning. *International Journal of Multidisciplinary Research and Explorer, 2*(6), 49–59.
Wang, T., & Li, W.-H. (2010). Naive Bayes software defect prediction model. In: *2010 International Conference on Computational Intelligence and Software Engineering*, (pp. 1–4). IEEE.
Wang, S., Liu, T., Nam, J., & Tan, L. (2018). Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering, 46*(12), 1267–1293.
Wang, R., Li, J., Yang, X., & Yang, J. (2019). Block-regularized repeated learning-testing for estimating generalization error. *Information Sciences, 477*, 246–264.
Wang, J., Shen, B., & Chen, Y. (2012). Compressed c4. 5 models for software defect prediction. In: *2012 12th International Conference on Quality Software*, (pp. 13–16). IEEE.
Wang, Y., Wang, W., Joty, S., & Hoi, S.C. (2021). Codet 5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, (pp. 8696–8708).
Xiaozhi, L., Yinan, W., & Yinghua, Y. (2020). Fault diagnosis based on sparse semi-supervised GAN model. In: *2020 Chinese Control And Decision Conference (CCDC)*, (pp. 5620–5624). IEEE.
Yao, W., Shafiq, M., Lin, X., & Yu, X. (2023). A software defect prediction method based on program semantic feature mining. *Electronics, 12*(7), 1546.
Zhou, X., Han, D., & Lo, D. (2021). Assessing generalizability of CodeBERT. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (pp. 425–436). https://doi.org/10.1109/ICSME52107.2021.00044.
Zhou, C., He, P., Zeng, C., & Ma, J. (2022). Software defect prediction with semantic and structural information of codes based on graph neural networks. *Information and Software Technology, 152*, 107057.
Zhang, H., Lu, S., Li, Z., Jin, Z., Ma, L., Liu, Y., & Li, G. (2024). CodeBERT-attack: Adversarial attack against source code deep learning models via pre-trained model. *Journal of Software: Evolution and Process, 36*(3), 2571.
Zhao, Z., Yang, B., Li, G., Liu, H., & Jin, Z. (2022). Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks. *Journal of Systems and Software, 184*, 111108.
Zhou, Z., Zhu, Y., Yu, Q., & Li, J. (2025). Software defect prediction method based on multi-feature fusion. In: *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*, (pp. 529–539).

Authors and Affiliations

Xiaoxing Yang[1] · Liwei Xiao[2] · Jianmin Su[3] · Bingding Huang[2]

✉ Bingding Huang
huangbingding@sztu.edu.cn

Xiaoxing Yang
xxyang@chuhai.edu.hk

Liwei Xiao
2310413022@stumail.sztu.edu.cn

Jianmin Su
jackysura@163.com

[1] Department of Computer Science, Hong Kong Chu Hai College, Hong Kong, China

[2] School of Artificial Intelligence, Shenzhen Technology University, Shenzhen 518118, China

[3] School of Mechanical and Automation Engineering, Wuyi University, Jiangmen 529020, China