Software Defect Prediction via Generative Adversarial Networks and Pre-Trained Model

Wei Song, Lu Gan, Tie Bao*
College of Computer Science and Technology, Jilin University
Changchun 130012, China

***Abstract*—Software defect prediction, which aims to predict defective modules during software development, has been implemented to assist developers in identifying defects and ensure software quality. Traditional defect prediction methods utilize manually designed features such as "Lines Of Code" that fail to capture the syntactic and semantic structures of code. Moreover, the high cost and difficulty of building the training set lead to insufficient data, which poses a significant challenge for training deep learning models, particularly for new projects. To overcome the practical challenge of data limitation and improve predictive capacity, this paper presents DP-GANPT, a novel defect prediction model that integrates generative adversarial networks and state-of-the-art code pre-trained models, employing a novel bi-modal code-prompt input representation. The proposed approach explores the use of code pre-trained model as auto-encoders and employs generative adversarial networks algorithms and semi-supervised learning techniques for optimization. To facilitate effective training and evaluation, a new software defect prediction dataset is constructed based on the existing PROMISE dataset and its associated engineering files. Extensive experiments are performed on both within-project and cross-project defect prediction tasks to evaluate the effectiveness of DP-GANPT. The results reveal that DP-GANPT outperforms all the state-of-the-art baselines, and achieves performance comparable to them with significantly less labeled data.**

Keywords*—Software defect prediction; semi-supervised learning; generative adversarial networks; deep learning

I. Introduction

In this highly digitized society, software has become integral to all aspect of social life. As the fundamental element in software development, the software quality and reliability have become prominent, exerting profound impacts on various aspects of society. The growing complexity of modern software technologies, however, introduces various defects during development, compromising overall software quality and reliability [1]. The manual detection and correction of defects incur significant labor and cost burdens. Therefore, software defect prediction has emerged as a promising approach to automatically predict defective modules with existing software code and historical data [2], [3], [4], aiding developers in cutting costs and enhancing development quality. Prior work indicates that software defect prediction has been a top three research priority in software engineering [5].

Traditional defect prediction methods utilize machine learning algorithms, such as Decision Tree [6], Random Forest [7] and Naive Bayes [8]. These models rely on manually designed features, such as McCabe features [9] based on program flow chart, Halstead features [10] based on the number of opcodes and operators, and object-oriented CK metric [11]. These features are often too simplistic to effectively capture and understand the syntax, semantic structure, and contextual relationships of the code. Moreover, as the software complexity and defects grow, the costs and challenges associated with manually designing features have escalated significantly. Consequently, researches on automatically extracting program structures and semantic features from source code have been conducted. Automatic feature extraction methods primarily encompass four categories: sequence-based, tree-based, graph-based, and model-based. These features are put into deep learning models such as Deep Belief Network (DBN) [12], Convolutional Neural Network (CNN) [13] and Long Short-Term Memory (LSTM) [14] for prediction. These models outperform traditional machine learning models across various scenarios, demonstrating promising capabilities in predicting software defects.

Although deep learning models have seen success, current software defect prediction models face challenges. One is that current defect prediction models lack the ability to thoroughly comprehend the syntactic and semantic structures of the code. For example, CNNs have restricted capability in capturing contextual information. Recently, large language models (LLMs), which are trained on large scaled corpora and fine-tuned on various downstream tasks, such as GPT-series [15], [16] and BERT-series [17], [18], have set new state-of-the-art (SOTA) benchmarks on natural language processing tasks. Motivated by the achievements of LLMs, researchers have started exploring the application of language models in software engineering. Several code pre-trained language models have been proposed, such as CodeBERT [19], CODEGEN [20] and UnixCoder [21]. These models achieve SOTA in multiple software engineering tasks, demonstrating their potential for software defect prediction.

Another challenge lies in data limitation for training a defect prediction model. In practice, data collection for model training is extremely limited, which means over-fitting is likely to occur. Additionally, while fine-tuning the pre-trained model has shown to be an effective method to improve performance on downstream tasks, their discriminative ability significantly diminishes when the number of labeled samples for fine-tuning is too low. It has been exemplified that the performance of fine-tuned BERT significantly degrades when the number of labeled samples is less than 200 [17]. The limitations in data availability present a significant barrier to the application of language models and the development of defect prediction models. While a large scale of unlabeled source code is more

This work was supported by the Science and Technology Research Project of Educational Commission of Jilin Province, No. JJKH20231176KJ.

readily available within or cross the development projects, researchers have explored unsupervised and semi-supervised learning methods to gain decent results, such as those described in [22], [23] and [24].

This paper presents DP-GANPT, a software defect prediction model that leverages semi-supervised generative adversarial networks and a bi-modal code pre-trained model. DP-GANPT simultaneously leverages the generator and a pre-trained auto-encoder as dual different encoders, and the discriminator as a decoder for classification. The auto-encoder utilizes both labeled and unlabeled data for code representation in semi-supervised learning, while the generator introduces perturbation options for generating synthetic samples, augmenting the quantity and diversity of training samples. Concurrently, the discriminator serves as a decoder, enhancing the discriminative reconstruction capability and robustness of the decoder by the augmentation of GAN and semi-supervised learning techniques. A novel bi-modal dataset based on manually designed PROMISE datasets and the source files is constructed to evaluate DP-GANPT on both within-project defect prediction(WPDP) and cross-project defect prediction(CPDP) tasks. The results demonstrate that DP-GANPT outperforms all of the baselines by at least 17.8% and 3.4% on average, and it matches the performance of the SOTA models using only 100 labeled training samples.

The main contributions of this work are as follows:

- We propose a new software defect prediction model DP-GANPT, which employs GAN on pre-trained code language model for software defect prediction tasks, capable of driving both supervised and semi-supervised learning.
- We propose a novel bi-modal sequence input representation inspired by the thought of prompt learning, which enhances the adaptability of the model for downstream tasks in software defect prediction.
- We construct a software defect prediction dataset for bi-modal sequences corresponding to the PROMISE dataset to effectively facilitate the training and evaluation.

The rest of the paper is organized as follows. In Section II, we introduce the background and related work. Section III describes the proposed model DP-GANPT. We introduce the experimental setup in Section IV, and present the experiment results and a discussion in Section V. Section VI discusses threats to validity. Finally, we summarize our work and introduce the future work in Section VII.

II. Background and Related Works

A. Defect Prediction Models Based on Deep Learning

Fig. 1 illustrates the main steps of building a deep learning-based software defect prediction model. The initial step involves data collection and preprocessing. Features extracted from software modules are gathered either from the current project (*i.e.*, WPDP) or from other software projects (*i.e.*, CPDP) to serve as training samples. These features may encompass automatically extracted features or manually designed features such as code lines, complexity and comment rates. Subsequently, the collected samples are labeled to indicate the presence or absence of defects, and divided into training and test sets. Following construction of the model, training is carried out, enabling the model to learn the mapping relationship from software features to the existence of defects. Subsequent to training, the model is evaluated on the test set to assess its performance. Ultimately, the evaluated model is utilized to predict whether other software modules contain defects, providing valuable information for software development teams to identify and rectify potential defects.

Like traditional machine learning, some deep learning-based defect prediction models rely on manually-crafted feature engineering. Qiao *et al.* [25] and Manjula *et al.* [26] employed empirical manually extracted features, and utilized deeper and more complex networks to achieve better performance. More popular extraction methods include language sequences, abstract syntax tree (AST) and graph representations. Wang *et al.* [12] leveraged DBN to learn semantic features from the nodes of AST and source code, utilizing Euclidean distance on traditional numerical features to handle noise for defect prediction. Their findings demonstrate outstanding performance of automatically generated semantic features in file-level defect prediction, with promising results in cross-project defect prediction as well. Shi *et al.* [27] extracted AST information as symbol and control sequences to train Bi-LSTM. Qiu *et al.* [28] used matrices from ASTs and feed them into a CNN to extract features automatically. Zhao *et al.* [29] integrated AST and CFG features into a graph network architecture, leveraging the strengths of feature representations. Zhou *et al.* [30] employed GNN and AST for feature extraction and fusion, and CNN for defect prediction, claiming the best performance across 21 open-source datasets.

Recent studies have embraced pre-trained models as classifiers or auto-encoders. Fu *et al.* [31] proposed a Transformer-based method named LineVul, utilizing the CodeBERT pre-trained language model to generate vector representations of source code. Results indicate that LineVul achieves significantly higher F1-scores on C/C++ language datasets compared to baseline methods. Uddin *et al.* [32] introduced a novel model that combines pre-trained BERT with Bi-LSTM networks, treating BERT as an auto-encoder and using Bi-LSTM for classification. Liu *et al.* [33] introduced a model that integrates pre-trained UnixCoder and CNN, using UnixCoder as an auto-encoder, and CNN for classification prediction.

However, these approaches have several limitations that hinder their effectiveness and generalizability. Firstly, data scarcity usually occurs in practice, but is often overlooked for these models, leading to their poor performance in the software development. Secondly, although relevant unlabeled data is more readily available, these methods do not fully exploit its potential. Lastly, these approaches do not fully capitalize on the natural language components. Software artifacts, such as source code and documentation, often contain rich natural language information that can be leveraged to improve the understanding and representation of software vulnerabilities. Our proposed DP-GANPT leverages GAN and a pre-trained model, and is trained with semi-supervised learning method to exploit the employment of relevant unlabeled data. Additionally, we propose a novel input representation to better utilize the deeper characteristics of both programming and natural

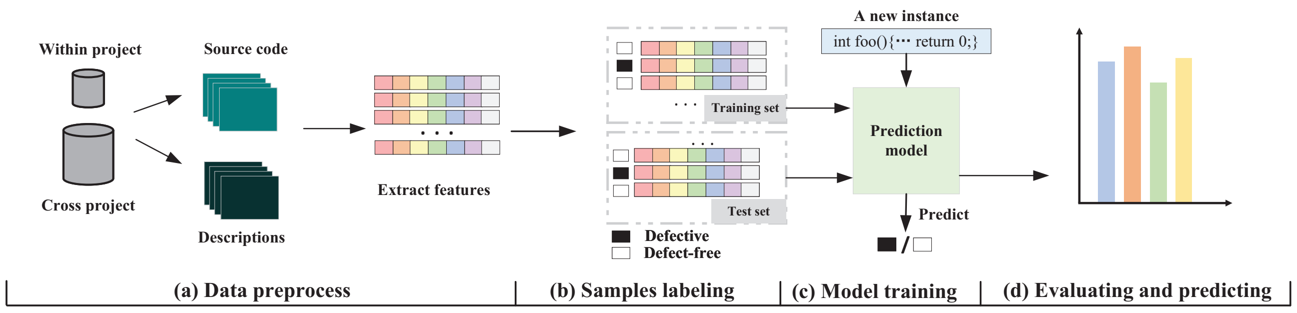


Fig. 1. Workflow of software defect prediction model based on deep learning.

language information. By leveraging these methods, we aim to find a more effective and efficient approach to software defect prediction.

B. Code Pre-trained Model

Recent advances in deep learning have enabled the development of LLMs. Trained on ultra-large-scale corpora, these models are able to better understand the underlying connections and connotations of data. Code pre-trained models aim to learn a robust representation of source code, which can be used for various programming tasks and show the effectiveness. The main difference between LLMs and code pre-trained models is the training data. LLMs are typically trained on natural language text, while code pre-trained models are trained on source code or both of code and natural language. The architectures of code pre-trained models are same as LLMs, including encoder-only, decoder-only and encoder-decoder architectures.

Encoder-only architecture models takes in a sequence of tokens and outputs a continuous representation of the input code. They are usually pre-trained on masked language model and other unsupervised tasks, and are ideal for classification and code search. Kanade *et al.* [34] presented CuBERT to train BERT models on large-scale Python source code, while CBERT [35] trains BERT on a large C language corpus. GPT-C [36] is trained on Python, C#, JavaScript and TypeScript for code completion task. Furthermore, Feng *et al.* [19] proposed CodeBERT whose architecture is same as RoBERTa[37] with bi-modal input.

Decoder-only architecture models are left-to-right models that generates a sequence of tokens to produce the output code, and usually used for generation tasks. CodeGPT [38], CodeParrot [39] and CODEGEN [20] are examples of such models. In the past two years, there has been a growing body of researches focused on the study of decoder-only models, driven by their demonstrated effectiveness in code generation tasks.

Encoder-decoder architecture models adapt pre-training objectives of both encoder-only and decoder only architectures. Encoder-decoder models includes UnixCoder [21], CodeT5 [40] and the enhanced version CodeT5+ [41]. The architecture of UnixCoder adopts the framework pattern of UniLM [42], supporting multiple tasks through manipulation of input attention masks. CodeT5+, an enhanced version of CodeT5, aims to efficiently expand model capacity while avoiding training from scratch. This objective is achieved by initializing the model with a pre-trained frozen offline language model.

C. SS-GANs

Semi-supervised Generative Adversarial Networks (SS-GANs) [43] is an effectual technique to implement semi-supervised learning, and a variate Generative Adversarial Networks(GANs) [44], which leverages labeled data to train the discriminator, and a large scale of unlabeled data to enhance the structural understanding and internal representations. In GANs, the generator generates fake samples that imitate the distribution of real samples, while the discriminator determines whether the sample is a real sample or not. To train a SS-GAN, the discriminator not only needs to discriminate the authenticity of the samples, but also acts as a classifier to classify real samples into different classes. Specifically, all samples are divided into $K + 1$ categories, where the real samples are classified into a certain class in $(1, ..., K)$, and the generated samples are classified into the $K + 1$ class.

III. METHODOLOGY

A. Motivation

This paper identifies two key challenges in software defect prediction. The first is the lack of comprehension and discrimination of models. Large scaled code pre-trained have been successful in various downstream tasks such as code search. Code pre-trained model as the auto-encoder employs an unsupervised learning method that enhances the generalization by learning low-dimensional representations of the input data. The other problem is data limitation. By introducing unlabeled data, semi-supervised learning enhances the learning of relevant clustered data, thereby improving the prediction performance. Furthermore, GAN architecture conducts data augmentation to improve the robustness, which has been used widely [45], [46], [47].

Therefore, we explore the integration of generative adversarial algorithms into semi-supervised learning methods to optimize pre-trained auto-encoder and train a discriminator, to address challenges posed by insufficient data in real-world scenarios and improve the capacity of prediction. The application

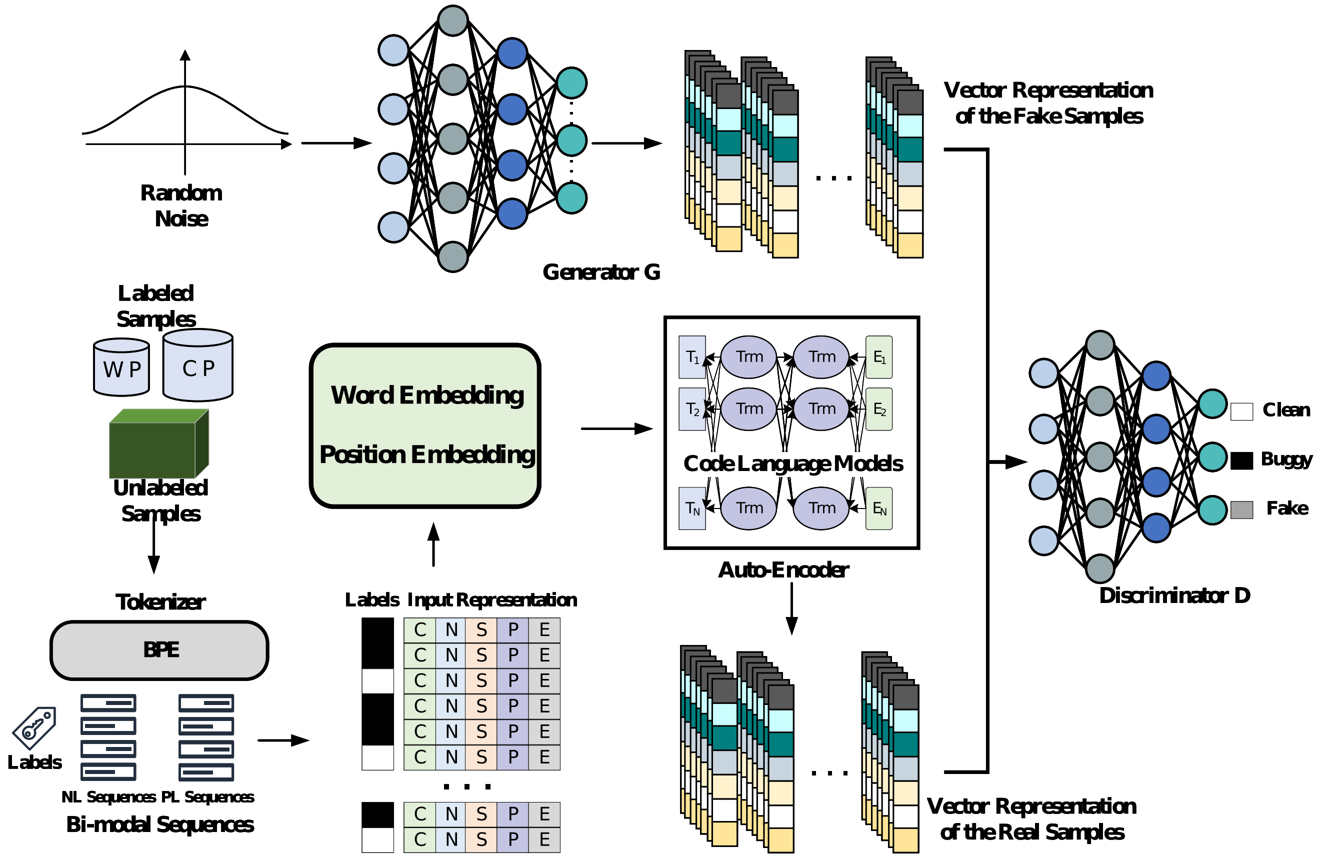


Fig. 2. The architecture of DP-GANPT.

of GAN aids the model in supplementing the original dataset by generating new data samples, thereby augmenting the quantity and diversity of training examples. Semi-supervised learning enables the model to effectively utilize unlabeled data, enhancing its generalization capability and robustness. Furthermore, with prompt learning becoming a new paradigm in natural language process [48], we aim to incorporate the thought of prompt learning into the natural language and programming language (NL-PL) bi-modal input representation to improve the comprehension of defect prediction objective and accelerate the convergence.

B. Architecture

As depicted in Fig. 2, DP-GANPT primarily consists of tokenizer, embedding layer, code pre-trained auto-encoder, generator, discriminator, and output layer. The NL-PL bi-modal sequences are extracted from training data within the project or across projects. After tokenization, the labeled and unlabeled data is represented as input representation sequences. Subsequently, these input representation sequences pass through token embedding and positional embedding layers, being mapped into vector representations of a specified dimension. These vectors are then put into a pre-trained code language model-based auto-encoder for the extraction of semantic and structural information, yielding output vector representations. Simultaneously, the generator takes random noise as input and maps it to samples that conform to the actual data distribution,which are described as fake samples in Fig. 2. Finally, both the real and fake samples are put into the discriminator to identify them clean, buggy or fake.

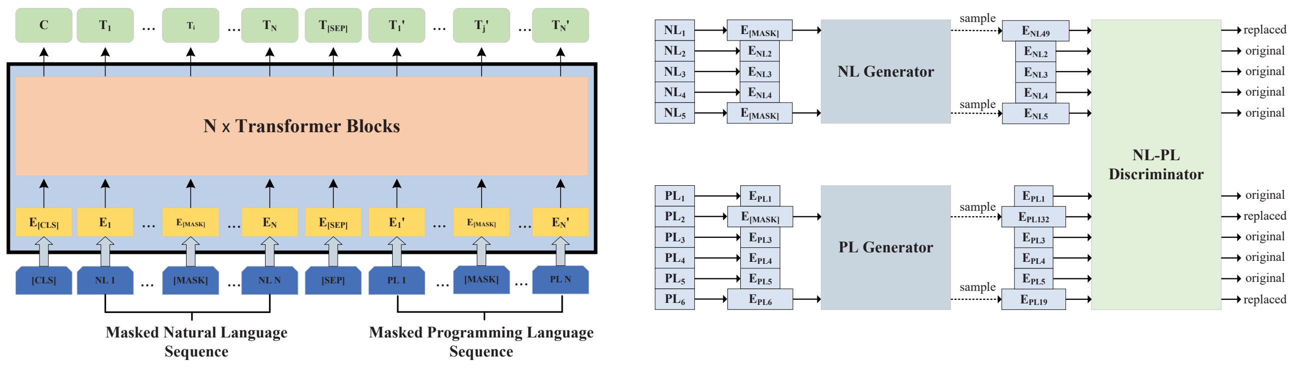
1) Input representation: In our study, we utilize source code and descriptions collected from the same project or other projects as training data, which are denoted as WP and CP in Fig. 2, respectively. Each labeled or unlabeled sample is a concatenation of two segments depending on modals, where one segment is a natural language sequence, represented as [NL], and the other is a programming language sequence, represented as [PL]. Like the standard input representation of BERT, [CLS] is placed at the beginning to describe the characteristics of the aggregated sequence, and the [SEP] token is placed between two sequences (*i.e.*, between [NL] and [PL]) to indicate the separation. Finally the [EOS] token is placed at the end to signify the end of the sequence. Therefore, the input representation of DP-GANPT is defined as:

$$INPUT = SEQ([CLS][NL][SEP][PL][EOS]), \quad (1)$$

where C, N, S, P and E are short for [CLS], [NL], [SEP], [PL], [EOS], respectively.

During the fine-tuning process, prompt learning emerges as an effective technique that guides the model in learning task-specific representations by incorporating prompts or cues into the input data. Inspired by prompt learning, this paper introduces content for the natural language modality subsequence, comprising various prompts. Formally, [NL] is defined as:

$$[NL] = \{[NP] \oplus [AP] \oplus [OP] \oplus ...\}, \quad (2)$$



(a) Masked Language Model **(b) Replaced Token Detection**

Fig. 3. Two objectives of pre-training CodeBERT. (a) illustrates the masked language model objective, and (b) illustrates the replaced token detection objective.

where [NP], [AP] and [OP] are name-prompts, annotations-prompts and objective-prompts, respectively. Name-prompts refer to the names of modules, functions, or classes in the software code, while annotations-prompts are used to document the purpose of a function, describe how a particular piece of code works, and provide guidance on how to modify or extend the code. Objective-prompts suggest the training objective, such as “Is there any bug, defect, error, fail or patch in the software module?” for software defect prediction tasks. The ellipsis indicates that the [NL] can be expanded depending on different designs, allowing for flexibility and adaptability in our approach.

By employing this input representation, additional prompt information is incorporated into the original input data, which guides the model to focus on portions of the input data relevant to the software defect prediction task, facilitating the learning of task-specific representations and enhancing performance in defect prediction.

2) *Tokenization and embbeding:* Before inputting the bi-modal sequences into the auto-encoder, the Byte Pair Encoding (BPE) algorithm [49] is employed for tokenizing the sequences. The core of BPE involves two stages, the generation of a merge-operation set, followed by the concrete application of these operations to a subword vocabulary.

The primary task in the first stage is to identify the most frequent character pairs within words, and construct the merge-operation set based on this information. Initially, each word is decomposed into individual character sequences. Frequent character pairs, which could be merged to form new symbol pairs, are identified through a search process. Following this, the character pairs are merged into new subwords, resulting in a more refined tokenization. This approach ensures the integrity of common vocabulary while breaking down rare vocabulary into a collection of its constituent subwords. The BPE mechanism is applied to the pre-training corpus, creating a subword tokenizer specifically designed for source code. BPE effectively handles complex vocabulary within the code, breaking it down into subwords containing rich semantic information, thereby optimizing the subsequent language model training process.

The embedding process includes word embedding and position embedding. After tokenization, tokens are embedded by One-Hot algorithm, and then pass through a linear layer for word embedding, resulting in vectors of fixed dimensions. For software defect prediction task, capturing code context and position information is crucial. Position embedding is employed to represent the position information of each element in the sequence, capturing the positional relationships among tokens in the input sequence. Position embedding is calculated as follows:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_m}}}\right), \tag{3}$$

$$PE(pos, 2i+1) = \sin\left(\frac{pos}{10000^{\frac{2i+1}{d_m}}}\right), \tag{4}$$

where pos represents the position of the token in the sequence, i is the dimension index of the positional vector, and d_k is the dimension of the vector representation after word embedding. Through this encoding method, the position of tokens in the sequence is uniquely determined, and the distance between adjacent tokens is approximately constant.

3) *Pre-trained auto-encoder:* Our preliminary work has proved that code language models with encoder-only architecture are the most effective and efficient for software defect prediction among the three language model architectures. Therefore, we utilize pre-trained CodeBERT with encoder-only architecture as the auto-encoder to generate high-quality real samples representations. It is worth noting that other outstanding models can also be implied, given the continuous emergence of large-scale code pre-trained models.

The CodeBERT auto-encoder consists of multi-layer Transformer blocks with self-attention mechanism, which trains RoBERTa [37] architecture on Codesearchnet [50], an open-source collection of over 4,000 open-source repositories providing both bi-modal data and uni-modal data. The dataset consists of software modules in six programming languages, including Python, Java, JavaScript, PHP, Ruby, and Go. At the pre-training state, two objectives are conducted as shown in Fig. 3. One is masked language model (MLM) proposed in BERT [17] to predict the masked token in a sequence, enhancing the comprehension of the context and relationships between tokens. This objective is trained on bi-modal data, which includes both code and natural language descriptions. The other objective is replaced token detection (RTD) that is proposed in ELECTRA [51] to identify whether a token is replaced, strengthening the capacity to recognize alterations in the code. This objective is trained on uni-modal data, consisting solely of code.

After pre-training, the auto-encoder encodes the labeled and unlabeled samples into 768-dim vector representations, named real examples. These vectors are then fed into the decoder, *i.e*., the discriminator, for prediction.

4) Generator and discriminator: The generator and discriminator enable semi-supervised and adversarial learning on the output of the auto-encoder. The generator transforms random noises into vector representations that mirror the structure of authentic samples, which we describe as fake examples. During the training process, the discriminator is a ternary classifier trained to differentiate among three distinct classes: real samples with defects, real samples without defects, and fake samples. Once the training process is complete, the generator is discarded while the discriminator is retained for prediction in practice.

The architecture of generator and discriminator is depicted in Fig. 4. According to the theoretical and experimental proof of Dai *et al*. [52], a bad generator improves generalization for semi-supervised learning. In this work, both the generator and discriminator are deep feed-forward networks with a hidden fully connected layer, rather than more complex models. We use LeakyReLU as the activation function and a dropout layer to avoid overfitting. The input noise vector has a size of 100, while the hidden layers of both generator and discriminator have a size of 512. The output size of generator is 768 that mirrors real examples from the output of the pre-trained CodeBERT. Before the SoftMax layer in the discriminator is another fully connected layer with the same numbers of classes for predicting. The output of the discriminator is a 3-dim vector, where the value of each dimension is the probability of the corresponding class. The class with the highest probability is the prediction of the model.

Training GANs is a process of finding Nash equilibrium in a zero-sum game between two players. However, because the loss function is non-convex, the parameters are continuous and the dimension of the parameter space is extremely high, so that it is very tough to find the equilibrium. Therefore, the loss function is usually minimized by gradient descent on the cost functions of both generator and discriminator or by using a heuristic algorithm to try to achieve convergence.

Formally, let G and D denote the generator and discriminator, respectively. At the training state, $P_g(x)$ is the generator's generation of the real data distribution $P_d(x)$. A three dimensional output vector of the input sample x is represented as:

$$l = \{l_1, l_2, l_3\}, \tag{5}$$

where l_1 and l_2 denote real example with and without defects, and l_3 denotes fake examples. We use $p_m(y = i|x)$ to denote

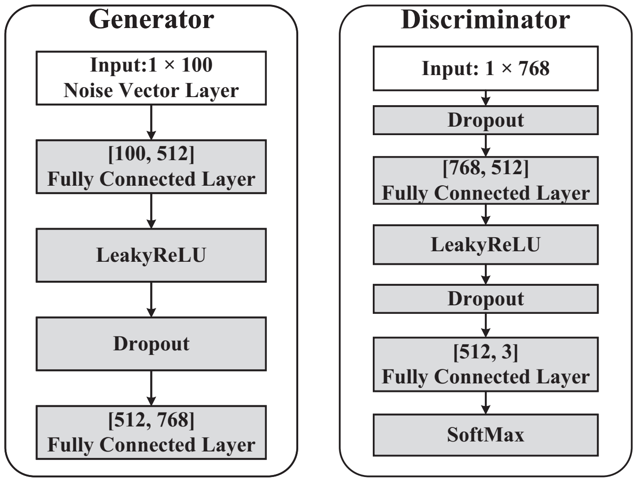


Fig. 4. The architecture of the generator and discriminator.

the probability that the sample x is predicted to be the i-th class, calculated by the model using SoftMax function as follows:

$$p_m(y = i|x) = \frac{exp(l_i)}{\sum_{k=1}^{3} exp(l_k)}. \tag{6}$$

Therefore, $p_m(y = 3|x)$ represents the probability that the sample x is judged to be a generated sample. L_D is defined as the loss function of discriminator D with cross-entropy. L_D is composed of two parts, the loss function of supervised learning L_{D_s} and the loss function of unsupervised learning L_{D_u}. L_{D_s} represents the penalty of misclassifying during training for the labeled real samples, which is defined as:

$$L_{D_s} = -\mathbb{E}_{x,y \sim p_d(x,y)} log\, p_m(y|x, y <= 2). \tag{7}$$

Unsupervised loss L_{D_u} consists of two parts, misjudging unlabeled real samples as fake and misjudging generated samples as defective or non-defective. By inputting each single real sample, a fake example would be generated in accompany during the training stage. This means half of the input to the discriminator is real samples from pre-trained model and the other half is fake samples from the generator. So the unsupervised learning loss function is define as:

$$\begin{aligned} L_{D_u} = &- \mathbb{E}_{x \sim p_d(x)} log\, [1 - p_m(y = 3|x)] \\ &- \mathbb{E}_{x \sim p_g(x)} log[p_m(y = 3|x)], \end{aligned} \tag{8}$$

the loss function of discriminator is defined as follows:

$$L_D = L_{D_s} + L_{D_u}, \tag{9}$$

where L_{D_u} is equal to zero for supervised learning.

For generator G, the goal is not just to minimize the third output, *i.e*., the fake dimension of the D, but to generate data that is as similar to the real data as possible. Therefore, we train G to match the output of the auto-encoder, because D needs to find features that best distinguish real data from the generated. This process is named feature matching. Let $f(x)$ denote activations such as average, on the output of the auto-encoder, then the objective function of this process is defined as:

$$L_{G_{fm}} = ||\mathbb{E}_{x \sim p_d(x)} f(x) - \mathbb{E}_{x \sim p_g(x)} f(x)||_2^2, \qquad (10)$$

Meanwhile, we reward when the samples generated by G are judged as non-defective by D. The loss function of this process is defined as:

$$L_{G_u} = -\mathbb{E}_{x \sim p_g(x)} log[1 - p_m(\hat{y} <= 2|x, y = 3)]. \qquad (11)$$

To sum up, the loss function of G is defined as:

$$L_G = L_{G_{fm}} + L_{G_u}. \qquad (12)$$

During the test and prediction stage, the fake dimension, *i.e.*, the third dimension l_3, is omitted when calculating SoftMax function for classification. This deliberate exclusion prevents samples from being misclassified as fake in real-world applications.

IV. Experimental Setups

A. Research Questions

To evaluate the effectiveness of our proposed method, the following four research questions are designed:

- RQ1: How does DP-GANPT perform in WPDP compared with the SOTA methods?
- RQ2: How does DP-GANPT perform in CPDP compared with the SOTA methods?
- RQ3: How does DP-GANPT perform under labeled data limitation?
- RQ4: Why does DP-GANPT work?

B. Construction of Bi-modal Dataset for Defect Prediction

Table I presents the projects in PROMISE dataset used for experimentation and evaluation, comprising a total of 10 projects with 25 versions utilized in the experiments. The PROMISE dataset has been widely utilized in researches on software defect prediction. However, the existing PROMISE dataset consists of static manually designed features, and lack specialized version designed specifically for bi-modal sentence sequences. Therefore, we construct a novel software defect prediction dataset of bi-modal input sequence based on the existing dataset and its source code project files, aiming to reflect the data characteristics and task requirements in real-world software engineering environments more accurately.

Specifically, we crawl the source engineering files corresponding to each version of each project in the PROMISE dataset. Subsequently, it extracts version, name, and defect information from each table for each version of each project. Leveraging the naming characteristics of JAVA project files, it extracts path information from the names and matches them with the source files. Irrelevant information for the experiments, such as licenses, authors, and format details (*e.g.*, spaces, extra spaces, line breaks), is removed. Then, it extracts files, function names, comment information, and source code sequences into new files to construct a new dataset using state machine transitions and pattern matching. The labels, defective or non-defective, are represented as bi-modal sequences. For instance, when encountering the "//" symbol, the state machine transitions to the corresponding single-line comment state, storing the subsequent sequence in the comment information string until encountering "\n" to conclude. Pattern matching involves merging and rewriting the path information of project files with the file names, followed by matching with the names in the static dataset to identify the corresponding labels for the files. Through these operations, a bi-modal PROMISE dataset with multiple prompts is constructed, providing robust support for subsequent experiments.

TABLE I. Description of Dataset Selected for the Experiments

Project	Version	Average Samples	Average Defect Rate (%)
Ant	1.5 1.6 1.7	422	22.5
Camel	1.2 1.4 1.6	891	23.6
Ivy	1.4 2.0	454	10.3
jEdit	4.0 4.1	276	21.5
Log4j	1.0 1.1	200	41.4
Lucene	2.0 2.2 2.4	247	56.7
Poi	1.5 2.5 3.0	328	66.8
Synapse	1.0 1.1 1.2	208	27.7
Xalan	2.4 2.5	816	35.6
Xerces	1.2 1.3	323	18.3

TABLE II. The Projects and Versions used as Training and Test Sets for Defect Prediction Experiments

	WPDP		CPDP	
Projects	Training Set	Test Set	Training Set	Test Set
Ant	1.5	1.6	Camel 1.4	jEdit 4.1
	1.6	1.7	jEdit 4.1	Camel 1.4
Camel	1.2	1.4	Lucene 2.2	Xalan 2.5
	1.4	1.6	Xalan 2.5	Lucene 2.2
Ivy	1.4	2.0	Poi 2.5	Synapse 1.1
jEdit	4.0	4.1	Synapse 1.2	Poi 3.0
Lucene	2.0	2.2	Xerces 1.3	Xalan 2.5
	2.2	2.4	Xalan 2.5	Xerces 1.3
Log4j	1.0	1.1	Camel 1.4	Ant 1.6
Poi	1.5	2.5	Ant 1.6	Camel 1.4
	2.5	3.0	Xerces 1.3	Ivy 2.0
Synapse	1.0	1.1	Ivy 2.0	Xerces 1.3
	1.1	1.2	jEdit 4.1	Log4j 1.1
Xalan	2.4	2.5	Log4j 1.1	jEdit 4.1
Xerces	1.2	1.3	Ivy 2.0	Synapse 1.2
			Synapse 1.2	Ivy 2.0

Based on the dataset constructed from the 10 projects and 25 different versions above, we conduct evaluation the proposed method on both WPDP and CPDP, as shown in Table II, comprising a total of 31 distinct experimental groups. For WPDP, the models are trained on older versions and tested on more recent versions, resulting in 15 different experimental groups. For CPDP, the study utilizes datasets from 16 different projects for both training and testing.

C. Evaluation Metrics

F1-score is widely employed in experiments involving imbalanced datasets, and is also widely utilized for prior works. Consequently, we opts for F1-score as the metric for assessment. F1-score is the harmonic mean of precision and

TABLE III. CONFUSION MATRIX

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

recall. Precision denotes the proportion of samples predicted as defects by the model that are indeed defects among all predictions. Recall, on the other hand, indicates the proportion of actual defect samples that the model correctly predicts. Precision and recall often stand in opposition, with a high value for one metric potentially leading to a reduction in the other. To strike a balance between these two metrics, F1-score is utilized as a comprehensive evaluation metric in experiments. Its computation is articulated as follows:

$$Precision = \frac{TP}{TP + FP}, \tag{13}$$

$$Recall = \frac{TP}{TP + FN}, \tag{14}$$

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall}, \tag{15}$$

where TP, FP, FN are true positive, false positive, false negative in confusion matrix shown in Table III, respectively. TP is the number of defective samples that are predicted buggy, while FP is the number of samples without defect but predicted to be buggy. FN means the number of defective examples that are undetected.

D. Implement Details

We run all experiments on three NVIDIA RTX 3090 24G GPUs with Intel Xeon Silver 4210R 64GB RAM. The maximum sequence length for experiments is configured as 512, the batch size is set to 16, and the learning rate is established at 1×10^{-5}. The maximum number of epochs for the experimental training process is set to 30. A 10-fold cross validation is employed on the training set, and early stopping is applied to mitigate overfitting. For the generator and discriminator, we employ the LeakyReLU activation function with a slope of 0.2 to introduce non-linearity without causing the vanishing gradient problem. Additionally, we apply a dropout rate of 0.3 to both the generator and discriminator, which helps to prevent overfitting and improve generalization performance. In terms of optimization, we selected AdamW as our optimizer, which has been shown to be effective in various natural language processing tasks. The auto-encoder in DP-GANPT is implyed by CodeBERT-base with 125M parameters based on microsoft/codebert-base in HuggingFace Transformers [53]. Partial configuration of the model is listed in Table IV. The input size of the auto-encoder is 514 including [CLS] and [EOS], and 12 Transformer blocks are trained in CodeBERT.

For semi-supervised learning, given the limitation of a small number of versions within the project, experiments on both WPDP and CPDP follow a methodology where, after excluding subsequent versions following the test set, three datasets are randomly sampled as unlabeled data for each experiment. This process is repeated for five times, and the average results are taken to mitigate the impact of randomness on the experiments. For instance, if the test set is "Ant 1.6", the unlabeled data will be randomly sampled three times from the remaining datasets after excluding "Ant 1.7" and the training set, repeating this process five times for a comprehensive evaluation.

TABLE IV. PARTIAL CONFIGURATION OF PRE-TRAINED CODEBERT AS AUTO-ENCODER

Name	microsoft/codebert-base
Architectures	Roberta Model
Attention_dropout_prob	0.1
Activation_function	GELU
Hidden_dropout_prob	0.1
Hidden_size	768
Intermediate_size	3072
Layer_norm_eps	1×10^{-5}
Max_position_embeddings	514
Num_attention_heads	12
Num_hidden_layers	12
Position_embedding_type	absolute
Vocab_size	50265
Parameter_size	125M

TABLE V. MANUALLY DESIGNED FEATURES FOR ADABOOST

Features	**Description**
AMC	Average Method Complexity
CA	Afferent Couplings
CAM	Cohesion Among Methods of class
CBM	Coupling Between Methods
CBO	Coupling Between Object class
CE	Efferent Couplings
DAM	Data Access Metric
DIT	Depth of Inheritance Tree
IC	Inheritance Coupling
LCOM	Lack of COhesion in Methods
LCOM3	Another typer of Lack of COhesion in Methods
LOC	Lines Of Code
MFA	Measure of Functional Abstraction
MOA	Measure Of Aggregation
NOC	Number Of Children
NPM	Number of Public Methods
RFC	Response For a Class
WMC	Weighted Methods of Class

E. Baselines

Five models are utilized as the baselines for evaluation, including one of the best-performing methods using manually designed features, the ensemble learning algorithm AdaBoost, and four SOTA defect prediction models based using deep learning method, including DBN [54], BugContext [55], Tree-LSTM [56], and MFGNN [29].

AdaBoost is an adaptive boosting ensemble learning method that constructs multiple weak classifiers on the same dataset, ultimately yielding a strong classifier. In the experiments, AdaBoost utilizes manually designed features as shown in Table V. DBN extracts semantic information from ASTs of the source code and metrics of code change features using deep belief networks. BugContext enhances the feature representation of programs by integrating semantic information from Context-Free Grammars (CFGs) and Dependency-Free Grammars (DFGs). Tree-LSTM trains a multi-layer LSTM network in the form of a tree structure corresponding to the AST of the source code. MFGNN embeds AST and context-free methods into a unified code representation, integrates them into a hierarchical model, and proposes a neural network architecture that effectively explores the top-down hierarchical structure using a graph attention mechanism.

V. Experiment Results and Discussion

A. Answer to RQ1 and RQ2

Table VI illustrates the comparison between five baselines and DP-GANPT on WPDP, with the best result on bold. Among all the models considered, DP-GANPT exhibits superior average F1-score across the 15 groups of experiments, demonstrating its outstanding performance. The five baselines, AdaBoost, DBN, BugContext, Tree-LSTM and MFGNN, achieve 46.4, 35.2, 42.9, 50.7 and 52.9 on the average F1-score. The top-performing model, DP-GANPT, achieves the value of 62.3, outperforming the five baseline models on F1-score by 34.3%, 77.0%, 48.3%, 22.9%, and 17.8%, respectively. More specifically, it achieves the top position on 11 out of 15 experimental groups.

This substantial performance gap demonstrates the advantage of DP-GANPT in capturing the underlying structures and syntax of source code. The results affirm that DP-GANPT successfully leverages its capability to enhance defect prediction accuracy and highlights its effectiveness in addressing challenges inherent on WPDP.

CPDP task primarily assesses whether the semantic and contextual features extracted by defect prediction models can be applied to different projects. The comparison between five baseline models and DP-GANPT on CPDP is presented in Table VII, where DP-GANPT achieves the highest average F1-score of 54.6. Among the 16 experimental groups, DP-GANPT demonstrates superior performance on 8 groups, while MFGNN exhibits the best on 6 groups, showcasing its respective strengths. Specifically, DP-GANPT outperforms the five baselines by 42.6%, 38.9%, 56.9%, 17.4% and 3.4%, respectively.

Another advantage of DP-GANPT lies in its ease of deployment, requiring minimal additional cost and effort. Firstly, fine-tuning a model based on pre-trained models incurs low time consumption costs. DP-GANPT achieves performance better than the SOTA models within less training time, often as few as two to three epochs. Additionally, the training samples in the form of sequences is easily obtainable, while the construction process of the required graph structure in MFGNN demands higher time and resource costs.

Combining the results of WPDP and CPDP tasks, DP-GANPT exhibits superior performance compared with the existing SOTA baselines. Conversely, AdaBoost, utilizing manually designed features, demonstrates suboptimal F1-scores in both tasks, suggesting that manually crafted features struggle to capture deeper semantic characteristics. Additionally, the performance of DBN, employing abstract syntax trees, is unsatisfactory. This can be attributed to its reliance on AST paths for establishing relationships between source code components, which only captures latent connections among code identifiers. However, software defect prediction, as a question of program classification, needs the identification of the actual control and data flow information during program execution. DP-GANPT is proficient at modeling source code, excels in capturing contextual and semantic information, making them more effective for software defect prediction tasks. Furthermore, MFGNN utilizing graph architecture demonstrates commendable performance in experiments. Nonetheless, as mentioned earlier, constructing graph models entails higher time and resource costs. These findings underscore the effectiveness and efficiency of DP-GANPT in addressing the challenges inherent in software defect prediction, as they offer a more nuanced understanding of context and semantics in source code, thereby outperforming alternative approaches.

Comparing the performance of the listed models in WPDP and CPDP with the same test set, it is evident that the results excel in WPDP. This demonstrates the importance of prioritizing data from the same project when feasible, as the consistency and correlation of data distributions between different versions of the same project are stronger. In practical applications, effort should be placed on collecting data from the same project for optimal results.

In conclusion, DP-GANPT performs better than the baselines on both WPDP and CPDP, demonstrating the effectiveness of the model. Furthermore, DP-GANPT is also an efficient and convenient method, and achieves more appropriate performance on WPDP.

B. Answer to RQ3

Deep learning models often exhibit lower performance with limited training data. Therefore, this section reduces the number of labeled samples in the dataset to 100 and 50 to demonstrate the performance with fewer labeled training samples. DP-GANPTs trained with 100 and 50 labeled samples are described as GANPT-100 and GANPT-50 for distinction. The experiments are conducted by randomly selecting samples from the training set for five iterations to obtain averaged results. The performance under data limitation is explored by comparing with the SOTA MFGNN, GANPT-S employing supervised learning without unlabeled training data, and DP-GANPT.

Across the 15 groups of experiments on WPDP depicted in Table VIII, GANPT-100 outperforms MFGNN by 1.9% on average, and achieves higher performance on 8 groups. In the CPDP task, as shown in Table IX, both GANPT-50 and GANPT-100 outperform Tree-LSTM, while they perform slightly below MFGNN. Throughout the experimental groups, GANPT-100 surpasses MFGNN in 6 out of 16 experiments. The performance of GANPT-50 and GANPT-100 on both WPDP and CPDP is reduced by 14.6% and 9.0% compared to DP-GANPT, and by 12.4% and 7.6% compared to GANPT-S, respectively. These results shed light on the performance of DP-GANPT under conditions of data limitation, revealing its resilience and competitive edge on WPDP, while also showcasing its comparative performance in the challenging context of CPDP.

The above analysis indicates that, under conditions of data limitation, DP-GANPT exhibits a decline in performance on both WPDP and CPDP. However, it still manages to perform comparably to the SOTA models, robustly demonstrating its

TABLE VI. Comparison of Performance on WPDP between DP-GANPT and Five Baselines. F1-scores are Measured as Percentages. The Best F1-scores are Highlighted in Bold

Project	Training-Test Set	AdaBoost	DBN	BugContext	Tree-LSTM	MFGNN	DP-GANPT
Ant	1.5-1.6	37.8	40.7	31.1	29.7	33.1	**65.7**
	1.6-1.7	52.2	51.7	45.1	44.2	53.7	**56.0**
Camel	1.2-1.4	40.2	16.5	36.2	53.1	**54.3**	53.3
	1.4-1.6	40.2	32.0	27.8	55.9	**56.8**	51.2
Ivy	1.4-2.0	14.3	27.3	**31.9**	15.9	22.9	30.7
jEdit	4.0-4.1	57.0	41.6	38.5	62.0	**65.0**	63.6
Lucene	2.0-2.2	58.5	36.6	43.0	60.9	64.6	**75.2**
	2.2-2.4	64.8	37.4	68.0	68.1	68.8	**76.0**
Log4j	1.0-1.1	66.7	60.5	**75.5**	73.3	73.3	73.2
Poi	1.5-2.5	77.3	8.4	79.7	81.6	83.1	**87.4**
	2.5-3.0	54.6	27.0	65.2	73.9	73.3	**82.9**
Synapse	1.0-1.1	28.9	43.0	18.8	28.2	30.4	**53.3**
	1.1-1.2	40.3	41.5	42.4	50.3	50.3	**56.3**
Xalan	2.4-2.5	32.9	30.8	17.4	34.5	33.1	**69.4**
Xerces	1.2-1.3	29.6	32.4	9.4	29.4	30.9	**40.9**
Average		46.4	35.2	42.0	50.7	52.9	**62.3**

TABLE VII. Comparison of Performance on CPDP between DP-GANPT and Five Baselines. F1-scores are Measured as Percentages. The Best F1-scores are Highlighted in Bold

Training Set	Test Set	AdaBoost	DBN	BugContext	Tree-LSTM	MFGNN	DP-GANPT
Camel 1.4	jEdit 4.1	34.8	32.3	45.2	39.6	41.5	**53.3**
jEdit 4.1	Camel 1.4	25.7	23.4	11.7	31.8	**39.8**	37.7
Lucene 2.2	Xalan 2.5	63.6	57.2	43.2	67.3	**67.4**	66.8
Xalan 2.5	Lucene 2.2	46.5	56.4	65.4	74.4	64.3	**75.4**
Poi 2.5	Synapse 1.1	28.3	49	37.0	42.3	48.5	**49.5**
Synapse 1.2	Poi 3.0	57.7	48.5	66.2	78.5	81.4	**81.5**
Xerces 1.3	Xalan 2.5	38.4	26.8	23.6	**67.8**	63.5	67.0
Xalan 2.5	Xerces 1.3	35.4	32.4	34.4	33.7	**50.0**	45.7
Camel 1.4	Ant 1.6	54.3	56.1	22.2	44.1	50.3	**62.4**
Ant 1.6	Camel 1.4	23.9	31.9	22.6	32.6	36.3	**38.0**
Xerces1.3	Ivy2.0	34.6	30.5	25.3	27.6	**37.4**	29.5
Ivy2.0	Xerces1.3	12.5	36.6	32.1	27.4	**47.8**	41.7
jEdit 4.1	Log4j 1.1	26.3	37.8	31.6	57.2	57.1	**76.9**
Log4j 1.1	jEdit 4.1	57.7	48.4	38.0	39.3	57.8	**61.3**
Ivy2.0	Synapse 1.2	39.7	32.4	17.5	52.7	**62.0**	56.6
Synapse 1.2	Ivy2.0	33.3	29.6	**40.7**	28.5	39.0	30.8
Average		38.3	39.3	34.8	46.5	52.8	**54.6**

feasibility and effectiveness in scenarios where labeled samples are limited. Throughout the experimentation process, we observe that, in CPDP, the performance gap between models utilizing fewer labeled samples and model with all labeled samples is relatively lower than WPDP. This suggests that the reduction in label information on CPDP has a less pronounced impact on performance. This phenomenon may be attributed to the fact that, the data distribution is more inconsistent with the training set on CPDP compared with WPDP.

C. Answer to RQ4

To answer RQ4, we conduct ablations to investigate the roles of individual components of DP-GANPT. More precisely, we delve into the functions of the following components: the Transformer-based auto-encoder, model pre-training, input representation, generative adversarial augmentation, generator and discriminator architectures, and semi-supervised learning. The compared models include: 1) CB-NT, utilizing only CodeBERT architecture; 2) CB-FT, fine-tuning CodeBERT that incorporates both architecture and pre-trained weights; 3) CB-FR, integrating pre-trained model and the input representation proposed in this paper; 4) GANPT-S, supervised DP-GANPT without unlabeled samples; 5) GANPT-LSTM, employing a more intricate LSTM network instead of a single hidden layer feed-forward neural network as the generator and discriminator; 6) DP-GANPT, semi-supervised learning model we proposed.

Table X and Table XI illustrate the performance of the compared models in both WPDP and CPDP experiments. By contrasting the ablation results of different constituent modules in the tables, insights into the roles and impacts of auto-encoder architecture, model pre-training, input representation, generative adversarial techniques, generator and discriminator architectures, as well as semi-supervised learning, can be gleaned.

CB-NT shows the superiority of Transformer-based architecture with attention mechanism, which achieves performance

TABLE VIII. PERFORMANCE AND COMPARISON OF DP-GANPT ON WPDP UNDER DATA LIMITATION. F1-SCORES ARE MEASURED AS PERCENTAGES. THE BEST F1-SCORES ARE HIGHLIGHTED IN BOLD

Project	Training-Test Set	GANPT-50	GANPT-100	MFGNN	GANPT-S	DP-GANPT
Ant	1.5-1.6	15.4	32.5	33.1	63.2	**65.7**
	1.6-1.7	48.6	54.6	53.7	54.8	**56.0**
Camel	1.2-1.4	37.0	37.0	**54.3**	51.6	53.3
	1.4-1.6	31.3	36.2	**56.8**	50.3	51.2
Ivy	1.4-2.0	11.8	27.7	22.9	29.4	**30.7**
jEdit	4.0-4.1	54.3	56.6	65.0	67.5	**68.6**
Lucene	2.0-2.2	71.9	71.4	64.6	74.4	**75.2**
	2.2-2.4	72.1	75.1	68.8	**77.1**	76.0
Log4j	1.0-1.1	73.3	73.0	**73.3**	73.0	73.2
Poi	1.5-2.5	84.5	85.6	83.1	86.5	**87.4**
	2.5-3.0	80.5	79.9	73.3	82.1	**82.9**
Synapse	1.0-1.1	39.5	46.0	30.4	50.3	**53.3**
	1.1-1.2	41.1	40.9	50.3	54.8	**56.3**
Xalan	2.4-2.5	66.0	69.0	33.1	**69.7**	69.4
Xerces	1.2-1.3	20.7	23.3	30.9	38.9	**40.9**
Average		49.9	53.9	52.9	61.6	**62.7**

TABLE IX. PERFORMANCE AND COMPARISON OF DP-GANPT ON CPDP UNDER DATA LIMITATION. F1-SCORES ARE MEASURED AS PERCENTAGES. THE BEST F1-SCORES ARE HIGHLIGHTED IN BOLD

Training Set	Test Set	GANPT-50	GANPT-100	MFGNN	GANPT-S	DP-GANPT
Camel 1.4	jEdit 4.1	35.6	52.5	41.5	53.2	**53.3**
jEdit 4.1	Camel 1.4	36.1	36.7	**39.8**	36.7	37.7
Lucene 2.2	Xalan 2.5	65.5	65.5	**67.4**	67.6	66.8
Xalan 2.5	Lucene 2.2	72.4	74.9	64.3	74.5	**75.4**
Poi 2.5	Synapse 1.1	48.7	49.0	48.5	49.5	**49.5**
Synapse 1.2	Poi 3.0	74.5	80.6	81.4	**82.1**	81.5
Xerces 1.3	Xalan 2.5	59.4	61.8	63.5	64.9	**67.0**
Xalan 2.5	Xerces 1.3	39.0	39.6	**50.0**	44.5	45.7
Camel 1.4	Ant 1.6	55.6	54.9	50.3	60.8	**62.4**
Ant 1.6	Camel 1.4	30.6	37.0	36.3	36.3	**38.0**
Xerces1.3	Ivy2.0	29.4	29.5	**37.4**	28.8	29.5
Ivy2.0	Xerces1.3	11.9	12.3	**47.8**	41.3	41.7
jEdit 4.1	Log4j 1.1	73.5	70.8	57.1	76.7	**76.9**
Log4j 1.1	jEdit 4.1	54.1	53.7	57.8	59.5	**61.3**
Ivy2.0	Synapse 1.2	42.9	50.5	**62.0**	54.7	56.6
Synapse 1.2	Ivy2.0	26.4	26.5	**39.0**	31.6	30.8
Average		47.2	49.7	52.8	53.9	**54.6**

better than MFGNN on WPDP, and subtly under on CPDP. Comparing the performance of fine-tuned CB-FT with CB-NT, CB-FT reveals better results in 22 out of 31 experiments across the two tasks. It outperforms CB-NT by an average of 4.0% and 4.9%, indicating that the use of a code pre-trained model is a significant contributor to DP-GANPT. More powerful pre-trained models usually mean better results on downstream tasks. The improvement of performance on software defect prediction is inseparable from artificial intelligence, especially code pre-trained language models at this stage. The pre-trained model performs as an auto-encoder, deeply enhancing the understanding of program semantics and contextual information Additionally, the significantly lower cost of fine-tuning pre-trained models underscores its practical feasibility and advantages in real-world practice, compared with training new models and constructing graph structures.

Furthermore, CB-FR, trained with proposed input representation, exhibits a performance improvement of 6.3% over models not employing this approach in WPDP and CPDP. This demonstrates that the bi-modal input representation, built upon language understanding capability of the pre-trained model, guides the model training tasks, outputs directives and focuses attention, thereby enhancing the predictive capability of the model.

In comparison to the supervised learning models GANPT-S and CB-IR, both datasets exhibit slight improvements in performance, with average F1-scores increasing by 1.1% and 2.7%, respectively. DP-GANPT is able to generate imitations of real data distributions from generator which makes the data more diverse. However, when utilizing a more intricate LSTM network as the hidden layer for the generator and discriminator, the performance experiences a decline. This suggests that samples generated through generative adversarial processes may become overly specific, incorporating noise or excessively specific features present in the training data. Moreover, the excessive strength of the generator may hinder the discriminator's effective learning of the true distribution of real data. This imbalance in equilibrium could result in generated samples that fail to effectively enhance model performance.

DP-GANPT, leveraging semi-supervised learning, exhibits an improvement of 1.8% and 1.3%, respectively, compared with GANPT-S which do not utilize unlabeled data. Additionally, it surpasses models without GAN by 3.0% and 4.0%, respectively. The experiments above demonstrate that, under the intricate interplay of its components, DP-GANPT attains remarkable performance.

VI. THREATS TO VALIDITY

There are three main threats to validity as follows.

Implementation to baselines. To make a fair comparison, we reimplement CodeBERT sharing the same hyperparameters as our proposed DP-GANPT from HuggingFace Transformers. Although a slight difference may arise, we are confident since Transformers is generally accepted and used by a wide range of scholars. As for baselines that do not provide program codes, we reimplement them after rigorous argumentation.

Projects selection. In our experiments, we select 25 datasets from open-source PROMISE, which are fully or partly adopted in extensive software defect prediction researches. The experiments do not fully demonstrate the full performance of the bi-modal model due to the limitation of datasets we use. For example, we only use Java projects which do not generalize to other projects and other programming languages.

Labeled samples for semi-supervised learning. We use 100 and 50 labeled samples from labeled training set, which may not be enough. Furthermore, some projects have a small sample size, leaving fewer samples to perform semi-supervised learning. The difference in data distribution between the labeled samples we used and the test set might affect the results. Even though we have done multiple experiments, it could still have an impact on validity.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present DP-GANPT, a software defect prediction model that employs semi-supervised generative adversarial learning and a pre-trained model. DP-GANPT utilizes

TABLE X. Ablations of DP-GANPT on WPDP. F1-scores are Measured as Percentages. The Best F1-scores are Highlighted in Bold

Project	Training-Test Set	CB-NT	CB-FT	CB-FR	GANPT-S	GANPT-LSTM	DP-GANPT
Ant	1.5-1.6	48.2	55.4	62.8	63.2	61.4	**65.7**
	1.6-1.7	52.5	55.5	55.7	54.8	**56.6**	56.0
Camel	1.2-1.4	48.6	49.3	55.1	51.6	50.1	**53.3**
	1.4-1.6	50.9	49.3	49.5	50.3	45.3	**51.2**
Ivy	1.4-2.0	26.9	25.0	29.4	29.4	26.7	**30.7**
jEdit	4.0-4.1	56.7	64.2	68.2	67.5	67.1	**68.6**
Lucene	2.0-2.2	65.3	70.7	73.6	74.4	66.7	**75.2**
	2.2-2.4	69.4	73.5	73.1	**77.1**	76.0	76.0
Log4j	1.0-1.1	72.6	**73.5**	**73.5**	73.0	**73.5**	73.2
Poi	1.5-2.5	84.6	85.7	86.2	86.5	86.8	**87.4**
	2.5-3.0	70.2	72.0	80.5	82.1	77.5	**82.9**
Synapse	1.0-1.1	43.0	45.8	47.7	50.3	46.7	**53.3**
	1.1-1.2	47.7	44.3	52.8	54.8	55.2	**56.3**
Xalan	2.4-2.5	59.3	62.7	**71.9**	69.7	69.4	69.4
Xerces	1.2-1.3	30.9	32.1	33.3	38.9	27.0	**40.9**
Average		55.1	57.3	60.9	61.6	59.1	**62.7**

TABLE XI. Ablations of DP-GANPT on CPDP. F1-scores are Measured as Percentages. The Best F1-scores are Highlighted in Bold

Training set	Test set	CB-NT	CB-FT	CB-FR	GANPT-S	GANPT-LSTM	DP-GANPT
Camel 1.4	jEdit 4.1	38.5	36.0	52.1	53.2	52.7	**53.3**
jEdit 4.1	Camel 1.4	30.6	35.4	35.8	36.7	36.0	**37.7**
Lucene 2.2	Xalan 2.5	65.9	66.0	65.5	67.6	**67.8**	66.8
Xalan 2.5	Lucene 2.2	62.0	64.8	66.2	74.5	66.7	**75.4**
Poi 2.5	Synapse 1.1	44.3	43.1	48.7	49.5	49.0	**49.5**
Synapse 1.2	Poi 3.0	49.3	58.7	75.8	82.1	70.0	**81.5**
Xerces 1.3	Xalan 2.5	62.7	65.9	66.1	64.9	66.7	**67.0**
Xalan 2.5	Xerces 1.3	38.8	44.5	42.6	44.5	44.0	**45.7**
Camel 1.4	Ant 1.6	60.1	60.6	60.7	60.8	61.5	**62.4**
Ant 1.6	Camel 1.4	32.7	36.1	36.9	36.3	36.7	**38.0**
Xerces1.3	Ivy2.0	27.0	26.7	29.2	28.8	27.4	**29.5**
Ivy2.0	Xerces1.3	36.1	39.8	**42.6**	41.3	39.3	41.7
jEdit 4.1	Log4j 1.1	60.6	76.1	76.3	76.7	76.2	**76.9**
Log4j 1.1	jEdit 4.1	54.9	53.3	58.0	59.5	53.2	**61.3**
Ivy2.0	Synapse 1.2	56.2	54.2	55.0	54.7	53.5	**56.6**
Synapse 1.2	Ivy2.0	33.7	29.8	28.6	**31.6**	28.7	30.8
Average		47.1	49.4	52.5	53.9	51.8	**54.6**

GAN to generate a wealth of samples and employs a pre-trained model to encode the novel code-prompt bi-modal data, which includes both labeled and unlabeled samples. The discriminator in GAN predicts whether a sample is generated, defective, or defective-free.

We evaluate DP-GANPT on 31 groups of experiments on both WPDP and CPDP tasks are conducted for evaluation, using the new bi-modal dataset derived from the PROMISE dataset. The results reveal that DP-GANPT outperforms the SOTA methods, with an improvement of at least 17.8% on WPDP and 3.4% on CPDP. Furthermore, we reduce the labeled samples to 100 and 50 to investigate the performance of DP-GANPT under data limitation. The results demonstrate that it achieves decent performance compared with the baselines. Finally, reasons for the effectiveness are analyzed that individual components of DP-GANPT plays a role, including the Transformer-based auto-encoder, model pre-training, input representation, generative adversarial augmentation, generator and discriminator architectures, and semi-supervised learning.

In the future, with more powerful pre-trained models proposed continuously, we would like to apply them to software defect prediction tasks to pursue better enhancement. It is worthwhile generalizing DP-GANPT to other programming languages such as Python and Go, since the pre-trained model is trained on several programming languages. Additionally, we would continue to investigate more effective input representations of models.

Acknowledgment

The authors would like to thank anonymous reviewers for their thoughtful comments. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. This work was supported by the Science and Technology Research Project of Educational Commission of Jilin Province, Grant Number: JJKH20231176KJ.

References

[1] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[2] A. Perera, A. Aleti, B. Turhan, and M. Boehme, "An experimental assessment of using theoretical defect predictors to guide search-based software testing," *IEEE Transactions on Software Engineering*, 2022.

[3] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-g. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3187–3196, 2018.

[4] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104773, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622000616

[5] F. Ferreira, L. L. Silva, and M. T. Valente, "Software engineering meets deep learning: a mapping study," in *Proceedings of the 36th annual ACM symposium on applied computing*, 2021, pp. 1542–1549.

[6] J. Wang, B. Shen, and Y. Chen, "Compressed c4. 5 models for software defect prediction," in *2012 12th International Conference on Quality Software*. IEEE, 2012, pp. 13–16.

[7] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, and N. Ubayashi, "Empirical evaluation of cross-release effort-aware defect prediction models," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2016, pp. 214–221.

[8] T. Wang and W.-h. Li, "Naive bayes software defect prediction model," in *2010 International conference on computational intelligence and software engineering*. IEEE, 2010, pp. 1–4.

[9] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[10] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*, 1977.

[11] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pp. 69–81, 2010.

[12] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.

[13] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE international conference on software quality, reliability and security (QRS)*. IEEE, 2017, pp. 318–328.

[14] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via lstm," *IET Software*, vol. 14, no. 4, pp. 443–450, 2020.

[15] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[16] J. Scheurer, J. A. Campos, J. S. Chan, A. Chen, K. Cho, and E. Perez, "Training language models with natural language feedback," *arXiv preprint arXiv:2204.14146*, 2022.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, jun 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[18] P. He, J. Gao, and W. Chen, "Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing," *arXiv preprint arXiv:2111.09543*, 2021.

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, nov 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[20] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[21] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "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)*, 2022, pp. 7212–7225.

[22] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, "Software defect prediction based on kernel pca and weighted extreme learning machine," *Information and Software Technology*, vol. 106, pp. 182–200, 2019.

[23] N. Li, M. Shepperd, and Y. Guo, "A systematic review of unsupervised learning techniques for software defect prediction," *Information and Software Technology*, vol. 122, p. 106287, 2020.

[24] F. Wu, X.-Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, and Y. Sun, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Transactions on Reliability*, vol. 67, no. 2, pp. 581–597, 2018.

[25] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing*, vol. 385, pp. 100–110, 2020.

[26] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Computing*, vol. 22, no. 4, pp. 9847–9863, 2019.

[27] K. Shi, Y. Lu, J. Chang, and Z. Wei, "Pathpair2vec: An ast path pair-based code representation method for defect prediction," *Journal of Computer Languages*, vol. 59, p. 100979, 2020.

[28] S. Qiu, H. Huang, W. Jiang, F. Zhang, and W. Zhou, "Defect prediction via tree-based encoding with hybrid granularity for software sustainability," *IEEE Transactions on Sustainable Computing*, 2023.

[29] 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," *Journal of Systems and Software*, vol. 184, p. 111108, 2022.

[30] C. Zhou, P. He, C. Zeng, and J. Ma, "Software defect prediction with semantic and structural information of codes based on graph neural networks," *Information and Software Technology*, vol. 152, p. 107057, 2022.

[31] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[32] 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 Computing*, vol. 26, no. 16, pp. 7877–7891, 2022.

[33] J. Liu, J. Ai, M. Lu, J. Wang, and H. Shi, "Semantic feature learning for software defect prediction from source code and external knowledge," *Journal of Systems and Software*, p. 111753, 2023.

[34] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[35] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, "Exploring software naturalness through neural language models," *arXiv preprint arXiv:2006.12641*, 2020.

[36] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[37] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[38] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. [Online]. Available: https://openreview.net/forum?id=6lE4dQXaUcb

[39] L. Tunstall, L. Von Werra, and T. Wolf, *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[40] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[41] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint arXiv:2305.07922*, 2023.

[42] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, “Unified language model pre-training for natural language understanding and generation,” *Advances in neural information processing systems*, vol. 32, 2019.

[43] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” *Advances in neural information processing systems*, vol. 29, 2016.

[44] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” *Advances in neural information processing systems*, vol. 27, 2014.

[45] L. Chen, S. Dai, C. Tao, H. Zhang, Z. Gan, D. Shen, Y. Zhang, G. Wang, R. Zhang, and L. Carin, “Adversarial text generation via feature-mover's distance,” in *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[46] A. Bissoto, E. Valle, and S. Avila, “Gan-based data augmentation and anonymization for skin-lesion analysis: A critical review,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 1847–1856.

[47] X. Guo, U. Anjum, and J. Zhan, “Cyberbully detection using bert with augmented texts,” in *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2022, pp. 1246–1253.

[48] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[49] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[50] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.

[51] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, “Electra: Pre-training text encoders as discriminators rather than generators,” in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=r1xMH1BtvB

[52] Z. Dai, Z. Yang, F. Yang, W. W. Cohen, and R. R. Salakhutdinov, “Good semi-supervised learning that requires a bad gan,” *Advances in neural information processing systems*, vol. 30, 2017.

[53] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, oct 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[54] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep semantic feature learning for software defect prediction,” *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2020.

[55] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, “Improving bug detection via context-based code representation learning and attention-based neural networks,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[56] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, “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)*. IEEE, 2019, pp. 46–57.