The Journal of Systems & Software 204 (2023) 111753

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Semantic feature learning for software defect prediction from source code and external knowledge

Jingyu Liu, Jun Ai [*], Minyan Lu, Jie Wang, Haoxiang Shi

School of Reliability and Systems Engineering, Beihang University, Beijing, China

ARTICLE INFO

Article history:
Received 13 February 2023
Received in revised form 27 April 2023
Accepted 21 May 2023
Available online 7 June 2023

Keywords:
Software defect prediction
Semantic features
Transformer

ABSTRACT

Software defects not only reduce operational reliability but also significantly increase overall maintenance costs. Consequently, it is necessary to predict software defects at an early stage. Existing software defect prediction studies work with artificially designed metrics or features extracted from source code by machine learning-based approaches to perform classification. However, these methods fail to make full use of the defect-related information other than code, such as comments in codes and commit messages. Therefore, in this paper, additional information extracted from natural language text is combined with the programming language codes to enrich the semantic features. A novel model based on Transformer architecture and multi-channel CNN, PM2-CNN, is proposed for software defect prediction. Pretrained language model and CNN-based classifier are utilized in the model to obtain context-sensitive representations and capture the local correlation of sequences. A large and widely used dataset is utilized to verify the effectiveness of the proposed method. The results show that the proposed method has improvements in generic evaluation metrics compared with the optimal baseline method. Accordingly, external information can have a positive impact on software defect prediction, and our model effectively incorporates such information to improve detection performance.

© 2023 Elsevier Inc. All rights reserved.

1. Introduction

In the 21st century, it seems that no aspect of daily life is immune to assistance from software, however, software products with compromised quality are likely to produce errors and unexpected results. Such failures can have severe consequences not only causing property damage and monetary loss, but may also result in human casualty (Wong et al., 2017, 2010). According to the findings of a new report, leading causes of poor software quality include cybercrime resulting from software vulnerabilities and accumulated deficiencies, which are very expensive and time-consuming to mend, costing the US an estimated $2.41 trillion by 2022 (Krasner, 2022). Consequently, it is necessary to identify software defects at an early stage of development to deliver reliable software products with fewer defects at a lower cost. Software defect prediction (SDP) methods can help indicate fundamental components of software in which weaknesses, vulnerabilities, and defects are more likely to occur, thereby contributing to the reduction of overall testing and maintenance efforts.

SDP is the process of building classifiers to predict which software modules or regions of code are most likely to fail (Omri and Sinz, 2020), and one of the most important factors affecting the predictive performance is the feature representation of the source code. Early research on SDP focused on designing artificial features related to potentially defective code, such as product metrics (Chidamber and Kemerer, 1994; Purao and Vaishnavi, 2003), process metrics (Muthukumaran et al., 2015; Nagappan and Ball, 2005), design metrics (Wong et al., 2000), and network metrics (Ai et al., 2019; Li et al., 2019; Yang et al., 2018), while many researchers have evaluated various machine learning technologies to improve prediction accuracy and performance (Aleem et al., 2015; Alsaeedi et al., 2019; Elish and Elish, 2008; Özakıncı and Tarhan, 2018; Prasad et al., 2015). However, traditional features cannot distinguish codes that have different semantics but similar structures; in other words, these features do not adequately capture the syntax and semantics of different levels of source code (Akimova et al., 2021; Omri and Sinz, 2020; Pachouly et al., 2022). With the emergence of deep learning (DL) based on artificial neural networks (ANNs), researchers have proposed representation learning algorithms to automatically learn the semantic representation of programs (Dam et al., 2021; Hoang et al., 2019; Li et al., 2017; Qiu et al., 2019; Wang et al., 2020, 2016; Xu et al., 2019; Zhao et al., 2022). More advanced features can be extracted from raw data using DL, bridging the gap between the semantic information of programs and features for defect

[*] Corresponding author.
E-mail addresses: liujingyu1@buaa.edu.cn (J. Liu), aijun@buaa.edu.cn (J. Ai), lmy@buaa.edu.cn (M. Lu), wang_jie@buaa.edu.cn (J. Wang), sy2114218@buaa.edu.cn (H. Shi).

prediction. Source code tokenization and automatic feature extraction are still the main factors affecting DL for defect prediction (Giray et al., 2023). In recent years, many advances in the field of natural language processing (NLP) have been used to solve various code comprehension problems and have also demonstrated positive results in the field of SDP (Fu and Tantithamthavorn, 2022). Applying the Transformer architecture with self-attention mechanism will allow a more comprehensive consideration of the code context of the defect (Akimova et al., 2021).

Typically, token sequences and graph structures extracted from source code are used as model inputs to generate features containing semantic information Semantics refers to the meanings of the program, for code intelligence such as vulnerability detection and clone detection, the core challenge is how to empower models to understand and infer the intent behind the code. When humans read code, understanding becomes easier if the code contains descriptive text such as comments and other information. For the testing engineers, identifying and locating defects are also based on the understanding of the code. As developers increasingly use version control and source management system (SCMS) repositories such as GitHub to manage code in the software development process, researchers have access to a wealth of unstructured NL data such as code issue reports and pull requests. The software project itself also provides requirements documents that describe functional information, as well as comments within the code. In fact, the use of unstructured natural language (NL) data that does not come from source code may also help to obtain richer features containing defective semantics. However, only a few studies have explored the automatic extraction of additional semantic information from such external knowledge.

These unstructured NL data can be regarded as another view of the source code to help generate semantic features that reflect the functionality of the code. Using this kind of information can further enrich feature semantics and then support SDP models to identify defective modules (Huo et al., 2018). However, one primary problem with unstructured NL data is how to combine NL texts and programming language (PL) code to learn semantic features.

In this paper, we propose a novel model, PM2-CNN (Pretrained Model-Based Multi-Channel Convolutional Neural Network), for defect prediction, addressing the problem of learning semantic features with the help of external knowledge in an unstructured NL format. A pretrained language model fine-tuned on labeled datasets is used to extract semantic features from source code and descriptive text. Furthermore, we introduce a multi-channel CNN to simultaneously process PL features and NL features, and use multiple convolution kernels to obtain rich semantic information. The contributions of this work are presented as the following:

(1) From the perspective of better understanding and inferring code intent, defect feature is constructed by combining programming language data with natural language data, which enhances the features used in the software defect prediction process with richer information.
(2) A novel defect prediction model, PM2-CNN, is proposed, which takes both source code and descriptive text sequences as input and identifies potentially defective code modules as a result. Pre-trained model is introduced to extract features of natural and programming languages, and multi-channel CNN is built to serve as a classifier for defect prediction.
(3) The experiment result on large real-world dataset shows that the proposed method performs better than the baseline models, showing that combining PL data and NL data can yield better feature representation that reflects the defective semantics of the source code, and incorporating external knowledge in an unstructured NL format has a positive effect on SDP.

The remainder of the paper is organized as follows: Section 2 introduces works relevant to the features and techniques used for SDP. Section 3 presents the proposed defect prediction method. Section 4 describes the selection of datasets and experimental settings. Section 5 compares our work with other approaches, draws conclusions, and provides directions for future research. Finally, threats to validity and the study conclusion are presented.

2. Related work

In the following subsections, relevant works that inspired our research are briefly introduced.

2.1. Features used for software defect prediction

In the SDP field, determining the relevant features that can distinguish between defective and non-defective software entities is one of the more active research directions. Early research focused on coupling and cohesion metrics in code suites (i.e., CK metrics suite Chidamber and Kemerer, 1994). Research (Tiwari and Rathore, 2018) has provided an overview of coupling and cohesion metrics used for object-oriented suites. In addition, researchers have also introduced process metrics such as code changing metrics (Madeyski and Jureczko, 2015) and developer metrics (Bhattacharya et al., 2012) for the behaviors of each stage of the software development process, respectively, modeling code modifications and the interaction between developers when modifications occur. The designs of these metrics not only consider the code itself but also take other relevant information into consideration by mining the relationship between metrics and defects. However, artificially constructed metrics are difficult to distinguish from program semantics and therefore degrade defect prediction performance.

In subsequent studies, researchers used machine learning and DL methods to automatically learn semantic features from source code, thus, capturing the structural and semantic information of the program. Considerable existing work has proposed different approaches to learn program semantics for different tasks, and the approaches can be categorized into four groups: Feature-based, Sequence-based, Tree-based, and Graph-based (Siow et al., 2022). Wang et al. (2020, 2016) utilized a deep belief network to automatically learn semantic features from source code, using the program's abstract syntax tree (AST) and source code changes as input. Dam et al. (2021) represented the code as a sequence of code tokens and used the long short-term memory (LSTM) model to convert the sequence of tokens into a feature vector expressing the semantic information. Several studies (Hoang et al., 2019; Li et al., 2017; Qiu et al., 2019) have proposed feature learning techniques based on CNN architecture to tackle the problem of capturing local characteristics. Li et al. (2017) proposed an SDP framework called DP-CNN, which uses a CNN to generate discriminative features from the program's AST, and word embeddings are used to encode tokens extracted from the AST to learn the semantics of the source code. SDP approaches based on deep neural networks (Xu et al., 2019) and graph neural networks (Xu et al., 2021) have also been proposed, with the aim of learning high-level feature representation of defect data. It can be noted that these studies only consider code sequences or abstract graphs of code as the source of input for feature generation, which makes the features relatively scarce and insufficient.

A few researchers have explored the use of code annotations to enhance the semantic representation of features. Huo et al. (2018) used CNNs for annotation augmentation programs as SDP models to automatically embed code annotations when generating semantic features. Their method used a pretrained Word2Vec algorithm to encode codes and comments into numeric vectors, then fed the obtained vectors into two separate CNNs. The results demonstrated that the comment feature can help improve prediction performance, outperforming CNNs, deep belief networks, as well as standard classifications such as Logistic Regression or Naive Bayes. In another study (Miholca and Czibula, 2019), a hybrid SDP model based on ANN and gradient relational association rules was proposed, and the source code and annotations were encoded as fixed-length numeric vectors to distinguish defective and non-defective software components. The results showed that considering semantic features rather than traditional metrics significantly improves SDP performance. However, these studies use unsupervised learning models such as Word2Vec or Doc2Vec to encode the source code and annotations into numeric vectors, thus, ignoring the influence of the code context on features. Our research aims to combine unstructured NL data, which can be code comments or other descriptive text, with PL data in semantic feature generation. The proposed method learns program semantics with sequence-based code representations, which treats codes as flat sequences of tokens (Moritz et al., 2015; White et al., 2015) and transforms them into numerical vectors through distributed representations (Mikolov et al., 2013). In the proposed model, a pretrained language model is applied to capture deeper semantics, given the semantic and syntactic similarity of code and text. Depending on the context, similar code or text may have very different characteristics that may be associated with defects. In addition, we introduce a multi-channel CNN to jointly process program language features and natural language features, resulting in the enhancement of feature semantics.

2.2. Pretrained language models and their application in software defect prediction

In the field of NLP, pretrained language models have achieved great success. Massive unlabeled corpora of data are used for pretraining tasks, such as masked language modeling and next sentence prediction (Devlin et al., 2018), and finally a general language representation mapping function is available. Following the pre-training stage, the models are fine-tuned for specific tasks, and good results have been demonstrated for multiple downstream tasks, which facilitate the application of these techniques to source code and help in obtaining context-sensitive code representations.

In particular, BERT (Devlin et al., 2018) (Bidirectional Encoder Representations from Transformers) represents a milestone in the development of the NLP field. Kanade et al. (2019) attempted to introduce BERT into the field of source code and thus proposed CuBERT. They used the Python file corpus in GitHub as a dataset and performed fine-tuning on five classification tasks and one pointer prediction task. Alternatively, CodeBERT (Feng et al., 2020) is a dual-modal extension of BERT, which uses both natural language and source code as its input. CodeBERT captures the semantic connection between PL and NL, and achieves state-of-the-art results in both code search and code document generation tasks. UniXcoder (Guo et al., 2022) is a unified code pre-training model framework proposed by Microsoft. Different pre-training tasks and attention mechanisms are used in UniXcoder to enable the model to perform code understanding task and generation task as well. UniXcoder uses three language modeling tasks for pre-training: Masked Language Modeling, Unidirectional Language Modeling, and Denoising Objective Denoising. In addition, the authors propose Code Fragment Representation Learning, which includes the tasks of Multi-modal Contrastive Learning (MCL) and Cross-Modal Generation (CMG), where MCL utilizes AST to enhance the semantics of fragment embeddings, and CMG leverages code annotations to align embeddings between programming languages. The ablation experiments show that both AST and code annotations enhance UniXcoder to better capture code semantics. Considering the adaptability of the code representation obtained by this pretrained language model, our research utilizes the encoder-only mode of UniXcoder as the encoder of our proposed model to generate feature representations of PL and NL.

In the SDP field, several SDP methods based on Transformer and pretrained language models have been explored by researchers. In the research (Uddin et al., 2022), an SDP model using a bidirectional LSTM network (Bi-LSTM) and BERT-based semantic features (SDP-BB) is proposed. This model uses Bi-LSTM to mine contextual information from embedded token vectors learned by the BERT model and captures the semantic features of codes to predict defects in the corresponding software. Fu and Tantithamthavorn (2022) propose a Transformer-based line-level vulnerability prediction approach, namely LineVul, which uses the CodeBERT pretrained language model to generate a vector representation of the source code. Their results show that the F1 metric of LineVul is significantly higher than the baseline method. However, in the studies mentioned above, the pretrained language model is used to generate the feature representation of the code without considering any other external knowledge.

Our research aims to enhance code representation with richer semantic information by augmenting unstructured NL data. The proposed method extracts features based on a pretrained model and uses a multi-channel CNN to fuse PL and NL features. The use of convolution operations helps to aggregate contextual tokens and capture local semantic information, thereby enhancing the features used for SDP.

3. Methodology

In this section, the framework of the proposed model, PM2-CNN, is described in detail, with the aim of jointly considering PL features and NL features for SDP. The proposed model integrates the pretrained model and the constructed multi-channel CNN, which consists of a two-step procedure: First, based on the pretrained model, the feature embeddings of the source code in PL and the descriptive text in NL are respectively extracted. Second, the two feature embeddings perform the convolution layer operation to obtain feature vectors for predicting software defects. The workflow of the proposed model is depicted in Fig. 1. For the model inputs, the order and sequence of the tokens is preserved and fed into the model. Data pre-processing such as tokenization and feature extraction is then performed. During training, the model learns to optimize Cross Entropy Loss function that measures the difference between the predicted defect probabilities and the actual defect labels.

In the remainder of this section, some notations used in the description of the model are presented before introducing the model, and then the process of generating feature embedding based on the pretrained model UniXcoder is introduced. Finally, the multi-channel CNN classifier is described in detail.

3.1. Notation

Suppose there are N sets of data in the dataset $X = (code_i, text_i)|_{i=1}^{N_{data}}$ and a corresponding label set $Y = y_i|_{i=1}^{N_{data}}$, where $code_i$ and $text_i$ indicate the source code in PL format and the description text of the source code in NL format in ith module of X, y_i indicates the corresponding label, and N_{data} is the number of subjects in the data archives. Source code can be represented as $code_i = \left[w_i^1, w_i^2, \ldots, w_i^n\right]$ and descriptive text can be represented as $text_i = \left[c_i^1, c_i^2, \ldots, c_i^m\right]$. Here, $w_i^n \left(c_i^m\right)$ denotes the tokens with the length of $n\,(m)$, which are obtained after applying the tokenization function on the source code and descriptive text of the ith module. The notions used to describe the model are listed in Table 1.

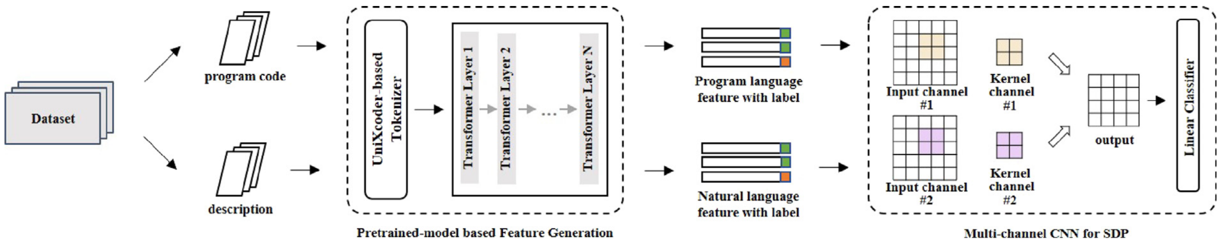


Fig. 1. An overview of the workflow of the proposed model.

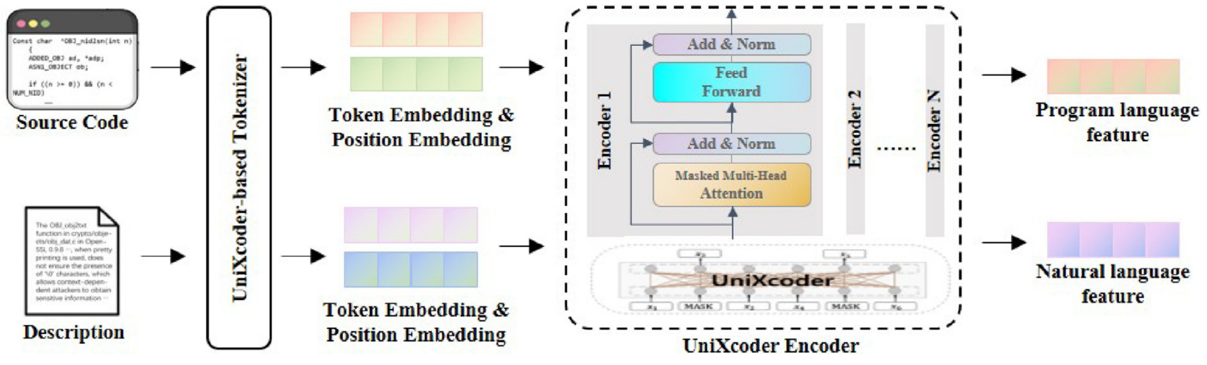


Fig. 2. UniXcoder-based feature embedding generation.

Table 1
Notations used in the description of the model.

Symbols	Details
X	Repository/Dataset
$code_i$, $text_i$	Source code and descriptive text in ith module of X
Y	The label of X
y_i	The ith label of Y
w_i^n	Represents token of programming language
c_i^m	Represents token of natural language
x_n	Represents the embedding of token n
L	Fixed length of token sequence
d	Denotes the feature embedding size

3.2. Pretrained model-based feature embedding generation

The proposed model uses UniXcoder (Guo et al., 2022), a unified cross-modal pretrained model for PL, to learn the semantic features of the source code and description text. UniXcoder utilizes cross-modal information such as AST and code annotations during pre-training, and experimental results for code comprehension and generation tasks show that it achieves SOTA in most tasks, which indicates that the representation of code fragments is enhanced. Therefore, UniXcoder was selected as the encoder in the proposed model for extracting code and text features, with each fragment being output as a vector. Fig. 2 illustrates the structure of the feature embedding generation in which there is a UniXcoder-based byte-pair encoding (BPE) tokenizer and a stack of N Transformer layers. Each layer contains a multi-headed self-attention operation followed by a feed forward layer.

3.2.1. Byte pair encoding tokenization

It is noted that UniXcoder trains a BPE vocabulary on the C4 dataset and CodeSearchNet, which consists of 50K subword units for programming languages and 1416 additional special tokens. Therefore, in this model the UniXcoder pretrained tokenizer is directly employed.

The BPE tokenizer is a typical subword-based tokenizer algorithm, which decomposes strings or words into substrings or subwords, with a granularity level between characters and words. The central idea is to continue merging subwords according to the frequency of occurrence until the vocabulary size is reduced or the probability increment is below a certain threshold, which helps to balance the vocabulary dictionary size and semantic independence. Tokenizers can divide words into their smallest components and then combine these small components into statistically interesting components. For example, “smaller” and “smallest” become “small”, ”er”, and “est”. The advantage of BPE is that it can share subwords to compress storage space and allow the tokenizer to advance further. Unmatched substrings will be replaced with special symbols such as “<unk>” for output, and string blocks classified as unknown tokens will actually disappear during training, thus, preserving more meaningful semantic information for the subsequent embedding procedure.

UniXcoder is a code pretrained model that is simultaneously compatible with “encoder-only” mode, “decoder-only”, mode and “encoder–decoder” mode. Since the encoder-only mode is used in the subsequent feature embedding generation, a special token should be added at the front of the input, which can be regarded as a “switch” to control the behavior of the model through different self-attention mask strategies. Specifically, after applying UniXcoder-pretrained BPE subword tokenization on the source code and descriptive text, a special token $[Enc]$ is added to each of the input sequences, for example, the sequence of the source code token is as follows:

$$code_i^{token} = [[cls], [Enc], [Sep], w_1, w_2, \dots, w_n, [sep]] \tag{1}$$

3.2.2. Feature embedding generation

In this step, the UniXcoder architecture is used as the encoder of the proposed model, and the pretrained weights of the UniXcoder are used as the initial weights. First, to generate embedding vectors for each token and its position, word and positional encoding are performed for the subword tokenization. The vectors are then fed into the UniXcoder architecture, which consists of a stack of Transformer layers. Each Transformer layer consists of an architecturally identical Transformer that has a multi-head self-attention layer followed by a feed forward layer. The prefix [Enc] in the output of the previous layer controls the UniXcoder to make it work as an encoder-only model.

3.2.2.1. Token embeddings and position embeddings.

In addition to the semantic meanings of the tokens in the input sequence, the position and order define the syntax of the code and the composition of the sentence, which means they are also critical for code and text understanding. Therefore, it is important to add position-related information in the input sequence to make the model aware of the order of the input. The intention of this step is to generate feature embedding that captures the semantic meanings and positional information of code and text. To do so, for each token sequence obtained by the subword-tokenized function, there is (1) a token encoding vector representing the meaning of each word and (2) a positional encoding vector identifying the position of each word in the input sequence.

The input sequence is mapped into an n-dimensional word vector sequence through a series of processing operations such as tokenization, dictionary mapping, and sequence encoding. The positional encoding information is also mapped into an n-dimensional word vector. Finally, the token embedding and the position embedding are added to form a new embedding vector x with dimension of (L, d) as the input of the UniXcoder encoder, as shown in formula (2), L is the fixed sequence length and d denotes the embedding size. Each row in x represents a token, and the dimension of x is $(512, 768)$ in the proposed model.

$$x = x^{token\ embedding} + x^{position\ embedding} \tag{2}$$

3.2.2.2. UniXcoder encoders.

In this step, the feature embedding vector x is fed into a stack of UniXcoder encoder layers. Each layer performs a multi-head self-attention operation, and then a feed forward neural network processes the output of the previous layer. The following briefly introduces the multi-head self-attention mechanism and feed-forward neural network.

The multi-head self-attention layer used in the proposed model makes each attention mechanism optimize the different characteristic parts of each vocabulary, in order to balance any deviations that may be produced by the same attention mechanism and benefit the meanings of words with more nuanced expressions.

The self-attention mechanism used in Transformer integrates the entire context into each word and helps to learn the dependencies of words and the internal structure of a sentence. The calculation of self-attention introduces three vectors, Query (Q), Key (K), and Value (V), which are the result of multiplying the input vector with matrices W_q, W_k, and W_v respectively. The corresponding matrices are randomly initialized and then continuously optimized through model training with a default dimension of (768, 768).

$$x \times W_q = \text{Query}\,(Q) \tag{3}$$

$$x \times W_k = \text{Key}\ (K) \tag{4}$$

$$x \times W_v = \text{Value}\ (V) \tag{5}$$

The attention score is calculated by the dot production of the Query and Key vectors of each word, which is divided by the square root of the Key vector dimension to make the gradient more stable. The formula is given in Eq. (6), where d_k denotes the number of columns of the Q, K matrix, i.e., the vector dimension.

$$\text{Attention}\,(Q, K, V) = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) V \tag{6}$$

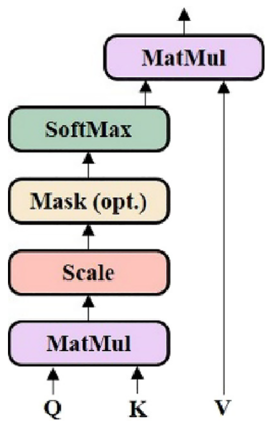


Fig. 3. Scaled dot-product attention.

The masking operation in the Transformer removes the influence of various paddings during the training process; in "encoder-only" mode, all elements of the mask matrix are set as 0 to allow all tokens to attend to each other. Finally, the SoftMax function is performed on the result and multiplied with the Value vector. The specific process and calculation formula are shown in Fig. 3:

To capture richer semantic embedding for the input sequence, self-attention is combined with the multi-head mechanism. Segmenting into multiple heads allows the model to focus on different aspects of information. Multi-Head Attention is to do the Scaled Dot-Product Attention process H times and then merge the output, as shown in formula (7).

$$\text{MultiHead}\,(Q, K, V) = Concat\,(head_1, \ldots, head_h)\, W^o$$
$$\text{where } head_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{7}$$

In the feed-forward neural network, a unit that can perform nonlinear transformation is provided to make the output feature embedding more meaningful. The unit maps the attention result of each position to a larger dimensional feature space, then utilizes a nonlinear activation function such as ReLU and finally returns to the original dimension. The pseudo-code for pretrained model-based feature embedding generation is shown in Algorithm I.

Algorithm I. Pretrained model-based feature embedding generation

```
    Input: $(code_i,\ text_i)|_{i=1}^{N_{data}}$, a set of PL & NL pairs
    Input: $\boldsymbol{\theta}$, initial pretrained model parameters
    Output: $(x_i^{code}, x_i^{text})|_{i=1}^{N_{data}}$, the feature embeddings of PL & NL tokens
1   for $n = 1,2,\cdots,N_{data}$ do
        /***take $code_i$ for example, same procedure as $text_i$***/
2       $code_i^{token} \leftarrow tokenization(code_i)$
3       $x_i = Token\_Embedding(code_i^{token}) + Position\_Embedding$
4       $Q_{x_i}, K_{x_i}, V_{x_i} \leftarrow x_i$
5       $x_i = LayerNorm(x_i + MultiHead\_Attention(Q_{x_i}, K_{x_i}, V_{x_i}))$
6       $x_i = LayerNorm(x_i + Feed\_Forward(x_i))$
7   end
8   return $(x_i^{code}, x_i^{text})|_{i=1}^{N_{data}}$
```

3.3. Multi-channel CNN classifier

In this subsection, the classification procedure based on a multi-channel CNN is demonstrated. The multi-channel CNN is used to jointly process PL features and NL features for SDP. The main structure consists of four parts: input layer, convolutional layer, pooling layer, and fully connected layer. The predicted probability is obtained at the end, indicating whether the software module is defective or clean.

The basic idea of the CNN is to capture local features, which are sliding windows consisting of several tokens for code or text. Multiple kernels of different sizes are used to extract key information in sentences so that local correlations can be better captured. The advantage of a CNN is that it can automatically combine and filter N-gram features to obtain semantic information at different levels of abstraction.

"Multi-channel" refers to how many sets of data the convolution layer in the CNN needs to process at one time. In computer vision, multi-channel CNNs can be used to classify color images. Since there are three colors – red, green, and blue – in color images, each color represents a channel. With regard to the source code and descriptive text, the convolutional layer needs to process two sets of data. Hence, it is natural to think of building one channel for one language type.

The following paragraphs use Fig. 4 as an example to briefly describe the process of the constructed multi-channel CNN. The input for the multi-channel CNN comes from the combination of outputs from the previous layer, which is constructed for the subsequent joint processing. Assuming that there are PL feature x_{pl} and NL feature x_{nl}, the length of each input sequence is L, and the feature vector of each word is d-dimensional.

The formal description of the convolutional layer is as follows: Use a kernel $w \in \mathbb{R}^{\boldsymbol{hd}}$ and a sliding window $x_{i:i+h-1}$ to perform a convolution operation on the input matrix to generate a feature c_i, denoted as

$$c_i = f\left(w \cdot x_{i:i+h-1} + b\right) \tag{8}$$

where $x_{i:i+h-1}$ represents a window of size $h \times d$ composed of row i to row $i+h-1$ of the input matrix, which is spliced by $x_i, x_{i+1}, \ldots; x_{i+h-1}$. The kernel region size is indicated by h, w represents the $h \times d$-dimensional weight matrix, b is the bias parameter, f is denoted as the nonlinear function, and w and x_{i+h-1} perform a dot product operation. The filter is applied to the input matrix and moves one-by-one from top to bottom: $i = 1, 2, \ldots, L-h+1$. For example, c_1 is calculated by the convolution layer operation on $x_{1:h}$, c_2 is obtained by the convolution layer operation on $x_{2:h+1}$, and so on. Then the feature map $\boldsymbol{c}$ is built by concatenating c_i together: $\boldsymbol{c} = [c_1, c_2, \ldots, c_{L-h+1}]$. Each convolution operation is equivalent to the extraction of a feature vector. By defining different windows, different feature vectors can be extracted. As can be seen in Fig. 4, there are two color matrices with kernel size [3, 5], which constitute two channels, collectively called a convolution kernel. By defining different windows, different feature vectors can be extracted. The result obtained after convolution is a vector whose shape is $((L-h+1) \times k_{num}, 1)$, where k_{num} is denoted as the number of convolution kernels.

The pooling layer uses max pooling, which extracts the maximum value in the column vector and obtains a fixed-length vector representation of vectors with different lengths. The pooling operation is for the entire vector, so its shape is $[L-h+1,1]$, and $h_{num} * k_{num}$ elements will be obtained, where h_{num} indicates the number of kernel region sizes. The elements will be concatenated together to form a $h_{num} * k_{num}$ dimension vector.

After obtaining the vector representation of the input sequences, the subsequent network structure is related to the specific task. In the SDP task, a fully connected layer is built, and the SoftMax activation function is used to output the probability of each category.

Algorithm II presents the pseudo-code of pretrained model based multi-channel CNN.

Algorithm II. Pretrained model based multi-channel convolutional neural network for defect prediction

```
Input: semantic representation of source code and descriptive text $(x_i^{code}, x_i^{text})\big|_{i=1}^{N_{data}}$ and the
corresponding labels $y_i\big|_{i=1}^{N_{data}}$, test data $X_t$,
Output: The prediction result of $X_t$
Hyperparameters: $N_{epochs} \in \mathbb{N}$
1   for $i = 1,2,\cdots,N_{epochs}$ do
2       for $n = 1,2,\cdots,N_{data}$ do
3           $x_i = concat(x_i^{code}, x_i^{text})$
4           /***Forward propagate via network and calculate the training loss***/
5           $output = [conv(x_i)$ for $con$ in $convs]$
6           $output = concat(output)$
7           $y_p = Linear\_Classification(output)$
8           $\mathcal{L} = CrossEntorpyLoss(y_p, y_i)$
9           /***Back propagate and update the network parameters***/
10          $w \leftarrow w - \eta \cdot \nabla \mathcal{L}$
11      end
12  end
13  Encode the test data $X_t$ as vector representations based on pretrained model and use trained
    model multi-channel CNN to predict the test data.
```

4. Experiment

4.1. Research questions

To systematically evaluate the proposed PM2-CNN approach, we formulated the following three research questions (RQs):
RQ1. Does the proposed approach outperform the state-of-the-art function-level SDP approach?
RQ2. How does the use of multi-channel CNN as a classifier affect the model?
RQ3. What are the effects of unstructured NL data on the prediction performance?

RQ1 focuses on the effectiveness of the proposed model by comparing it with other baseline models. RQ2 explores the impact of using a multi-channel CNN as the classifier in the proposed model, two sets of comparison experiments are conducted using different classifiers to evaluate the impact of the selected classifiers on the model performance. RQ3 investigates the impact that the addition of NL data has on the performance of the prediction model by controlling the inputs to the model.

4.2. Dataset

The Big-Vul dataset provided by Fan et al. (2020) was used as a benchmark in the experiments for two reasons: First, to make a fair comparison with methods such as IVDetect (Li et al., 2021), LineVul (Fu and Tantithamthavorn, 2022), and others. Second, the dataset correlates code changes with public Common Vulnerabilities and Exposures (CVE) description information, providing descriptions of the code function in NL format for both clean and defective samples, which meets the input requirements. The Big-Vul (Fan et al., 2020) dataset is a large C/C++ dataset in the field of code vulnerabilities. A total of 3754 code vulnerabilities are included, covering 91 different vulnerability types. All the information is stored in CSV format.

4.3. Evaluation measures

To evaluate the predictive performance of the proposed model, the experiment uses four evaluation indicators: *Accuracy*, *Precision*, *Recall*, and *F1-score* . Accuracy is the most common metric in the binary classification. In addition, F1-score is also widely used in SDP due to imbalanced datasets.

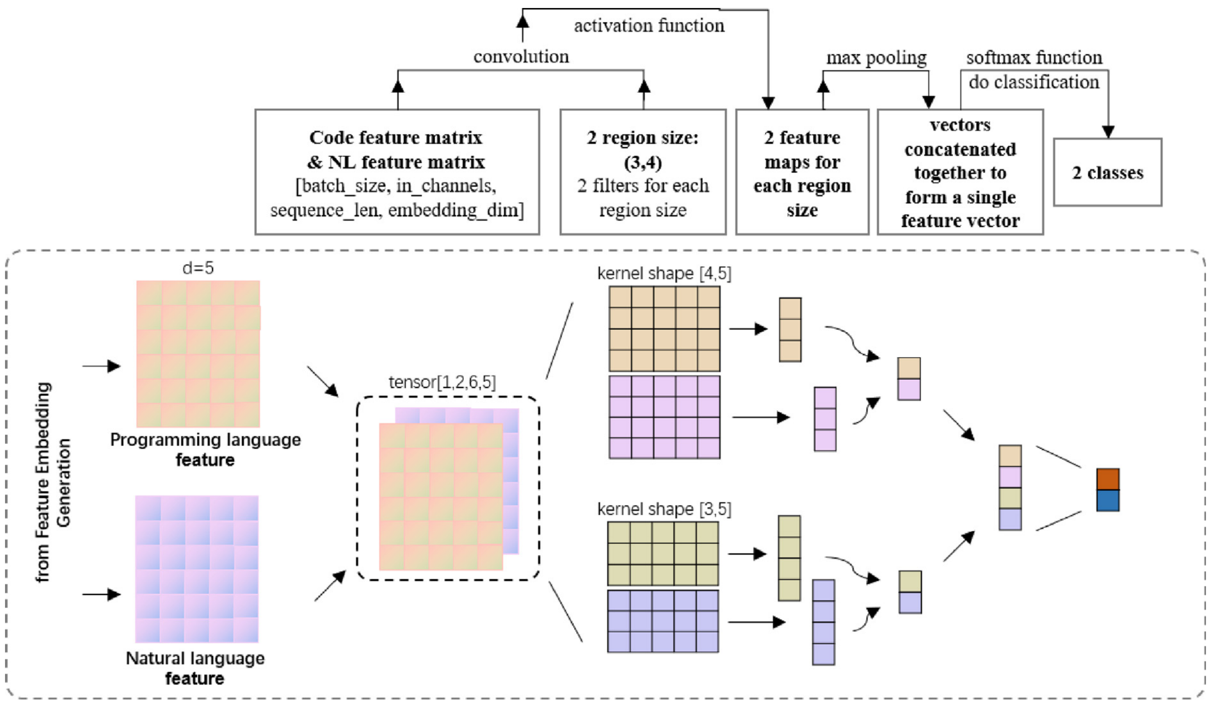

activation function
convolution
max pooling
softmax function
do classification
Code feature matrix & NL feature matrix [batch_size, in_channels, sequence_len, embedding_dim]
2 region size: (3,4) 2 filters for each region size
2 feature maps for each region size
vectors concatenated together to form a single feature vector
2 classes
from Feature Embedding Generation
d=5
Programming language feature
Natural language feature
tensor[1,2,6,5]
kernel shape [4,5]
kernel shape [3,5]

Fig. 4. Multi-channel CNN for jointly processing the PL and NL features. . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
Confusion matrix.

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

Usually, a confusion matrix is constructed to calculate the performance measures, as shown in Table 2. In the confusion matrix, the positive and negative row names represent the true value of the sample, and the positive and negative column names represent the predicted results of the sample. Therefore, True Positive (TP) means the defective samples that are predicted as defective samples. False Positive (FP) denotes the clean samples predicted as defects. True Negative (TN) indicates clean samples predicted as clean, and False Positive (FN) represents defective samples predicted as clean.

The accuracy rate is the probability of correct predictions, which represents the ratio of the number of samples correctly classified by the classifier to the total number of samples for a given test set. This can be defined as the following:

$$Accuracy(Acc) = \frac{TP + FN}{TP + FP + TN + FN}$$

Precision focuses on the prediction results and refers to the probability of correct predictions in the predicted positive samples. *Recall* indicates the proportion of defective cases correctly identified among the actually defective cases. The formulas of *Precision* and *Recall* are denoted as the following:

$$Precision(P) = \frac{TP}{TP + FP}$$

$$Recall\,(R) = \frac{TP}{TP + FN}$$

It should be noted that *Precision* and *Recall* are a pair of contradictory measures. In order to be able to comprehensively consider these two indicators, the F-measure (the weighted harmonic average of Precision and Recall) is proposed, denoted as the following:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

4.4. Experiment setting

In the experiment, the pretrained model UniXcoder (Guo et al., 2022) was used as the encoder of the proposed model. The pretrained model contained 12 multi-head attention. The multi-channel CNN was constructed to perform binary classification tasks. The size of the windows selected by the two channels were 3 and 4, and the number of each convolution kernel was 200. The dropout mechanism was used during the training process and set to 0.5. The batch size was 16. The AdamW (Loshchilov and Hutter, 2019) optimizer was utilized for training, and the initial learning rate was 2E−05. The parameter settings are shown in Table 3.

All the models used in the experiment were implemented in Python 3.8, primarily relying on two Python libraries, PyTorch (version 1.12.1) and Transformers (version 4.23.1). PyTorch is a high-performance library that provides optimization support for scientific computing in Python. It uses a graphics processing unit to accelerate computing and provides tools to support numerical optimization of general mathematical expressions. The Transformers library is also called pytorch-transformers and pytorch-pretrained-bert; it provides many state-of-the-art pretrained models, such as BERT, GPT-2 and RoBERTa. Based on the training dataset, the parameters of the overall model were fine-tuned to make it more suitable for the defect prediction task, and the best F1 score of the validation set was used to decide the model to preserve. The model was trained on an NVIDIA RTX A6000 graphics card.

The training, validation, and test data came from 188,636 function-level instances of the Big-Vul (Fan et al., 2020) dataset. In the experiment, the dataset was randomly divided into a training set, a verification set, and a test set, according to the ratio of 8:1:1. Specifically, 150,908 instances were selected for the training subset, and both the validation and test subsets

Table 3
Parameter settings of the proposed model.

Parameter	Value
Head number of multi-head attention	12
Kernel size	3, 4
Number of convolution kernel	200
Dropout rate	0.5
Batch size	16
Learning rate	0.00002

contained 18,864 instances each. The training subset was used to build the model, and the validation subset was utilized to preserve the optimal fine-tuning parameters. Finally, the test subset evaluated the performance of the proposed model.

5. Experiment results and analysis

5.1. Does the proposed approach outperform the state-of-the-art function-level SDP approach?

In this section, the proposed model is compared with SDP models with the same granularity to answer RQ1. We conducted an experiment comparing PM2-CNN with four state-of-the-art function-level defect prediction methods, including Devign (Zhou et al., 2019), ReVeal (Chakraborty et al., 2022), IVDetect (Li et al., 2021), and LineVul (Fu and Tantithamthavorn, 2022), to verify the effectiveness of the proposed method.

(1) Devign (Zhou et al., 2019): This model integrates a gated graph neural network (GGNN) and a convolutional module is proposed for vulnerability detection. This model takes joint graph representation as input and uses AST as the basic graph, combining a control flow graph and a data flow graph.
(2) ReVeal (Chakraborty et al., 2022): This model is a GGNN-based code vulnerability detection method. A code property graph is used as the model input. The GGNN is trained with the goal of generating graph embedding, then a multi-layer perceptron layer serves to obtain classification results.
(3) IVDetect (Li et al., 2021): In this model, feature-attention graph convolution networks are used for classification. The authors abstract the code representation based on a program dependency graph, thus, extracting different information and embedding it to obtain vector representations of a program dependency graph node (a statement).
(4) LineVul (Fu and Tantithamthavorn, 2022): This model uses the BERT structure and self-attention model to capture long dependencies of code sequences. Instead of training the code representation from the dataset of the specific project, it directly uses the language model pretrained by CodeBERT.

In this experiment, Big-Vul dataset (Fan et al., 2020) was used as input for the model. Table 4 presents the experimental results obtained by the five models. The best values in the model are highlighted in bold. Aiming at fair comparison with baseline models such as IVDetect, three metrics: Precision, Recall and F1-score were used to evaluate the performance of the proposed model.

According to the experimental results, the proposed method (PM2-CNN) outperforms the four baseline methods on three metrics. Specifically, the F1-score of the classification results is 2.78% higher than the best comparison model, thereby proving the effectiveness of the proposed model.

Table 4
Results of PM2-CNN and four baseline methods.

Model	Big-Vul dataset		
	Precision	Recall	F1-score
Devign	0.18	0.52	0.26
ReVeal	0.19	0.74	0.30
IVDetect	0.23	0.72	0.35
LineVul	0.97	0.86	0.91
PM2-CNN	**0.98**	**0.91**	**0.94**

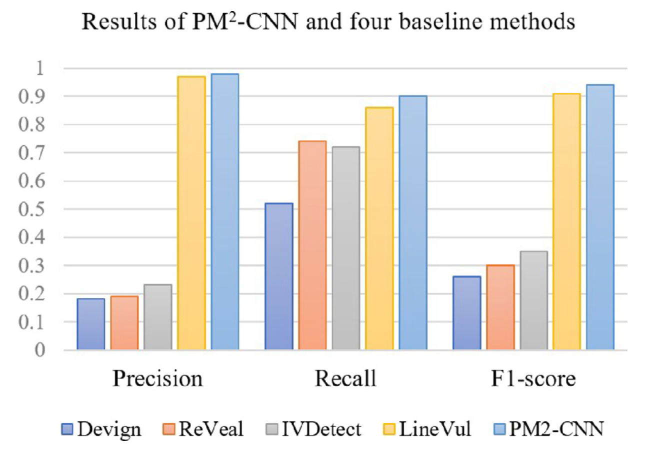


Fig. 5. Results of PM2-CNN and baseline models on Big-Vul (Fan et al., 2020) dataset.

From Fig. 5, it can be found that the SDP models based on the pretrained language model (PM2-CNN and LineVul) show better model performance than the other models, with an approximately 70% improvement in Precision indicators and 10%–30% improvement in Recall measures. The results show that using the pretrained language model to generate features not only realizes the automatic mining of software characteristics, but the obtained features also express more defect semantics.

Furthermore, compared with the LineVul model, the proposed method uses a multi-channel CNN module rather than a linear classifier. The convolution layer in multi-channel CNN processes data on multiple channels at one time. It can be seen that for the Recall and F1-score metrics, the performance increased by five and three percentage points, respectively. In terms of the F1-score, that is, the trade-off between the two metrics of Precision and Recall, PM2-CNN achieves the best F1-score of 0.9420. PM2-CNN uses a pretrained language model to extract the abstract features of PL and NL, and add them to the CNN. The final convolution result is the comprehensive information result of each channel. The use of a CNN as a classifier exhibits a better classification performance on the SDP task.

From the discussion above, we can conclude that the semantic features mined by the proposed method produce better prediction results on function-level SDP. The results show that among all the compared methods, the PM2-CNN model proposed in this paper achieved the best results in comparative experiments on three metrics, thus, proving the effectiveness of the model.

5.2. How does the use of multi-channel CNN as a classifier affect the model?

This subsection analyzes the effect of a multi-channel CNN as a classifier on the proposed model. When constructing the features, the UniXcoder is utilized to construct features from the inputs.

We conducted three comparative experiments to evaluate the effect of the CNN classifier used in the proposed model. In Experiment I, the CNN classifier was removed, and a simple linear

Table 5
Settings of comparison experiments.

	Input		Classifier
	Programming language	Natural language	CNN
PM2-CNN	●	●	●
I	●	●	/
II	●	/	●
III	●	/	/

'●' denotes the data or module **is included** in the experiment.
'/' denotes the data or module **is not included** in the experiment.

Table 6
Results of PM2-CNN and three comparison experiments on four metrics.

	Big-Vul dataset			
	Accuracy	Precision	Recall	F1-score
PM2-CNN	0.9937	0.9785	0.9081	0.9420
I	0.9927	0.9703	0.8976	0.9325
II	0.9933	0.9707	0.9100	0.9393
III	0.9908	0.9472	0.8844	0.9147

classifier similar to the classifier used in LineVul was established, and the model inputs included programming language and natural language. Considering that the inputs used to generate the feature embeddings include both NL data and PL data, when the multi-channel CNN is not used as a classifier, the inputs of the two sources will be merged into a sequence form in Experiment I due to the limitations of the input format. Therefore, in order to reduce the effects of the input forms on the classifier, we added two more comparison experiments that only take programming data as the input. Experiment II and Experiment III used only programming data as input, with the former utilizing a CNN classifier and the latter employing a simple linear classifier. Because NL cannot support the identification of defects, we did not set up any experiments that only fed NL data into the model.

The specific experimental setup can be found in Table 5, where '●' denotes that the data or module is included in the experiment and '/' indicates that it is not included.

The results of PM2-CNN and three comparison experiments are listed in Table 6. The first row shows the results of the PM2-CNN model, which are compared with Experiment I below. When the input is PL and NL data, the model performance increases by 0.1%–1% on the four metrics. In addition, when the input is only PL data, according to Experiment II and Experiment III, there is a 2.35% improvement on the *Precision* indicator and a 2.46% improvement on the F1-score indicator.

Consequently, it can be seen that using the CNN as a classifier has a positive effect on SDP tasks.

5.3. What are the effects of unstructured NL data on the prediction performance?

This subsection investigates whether unstructured NL features can help improve defect prediction performance. We evaluated the role of the NL characteristics by controlling the input of the model.

We compared the prediction results of PM2-CNN when the input is a combination of PL and NL, and when the input is only PL. The structure and initial parameters of the models in the experiments were exactly the same, except that the first experiment used program code and descriptive text as features, while the second experiment used only the source code as features for evaluation.

Table 7 shows the results obtained on the four indicators by carrying out the experiment 10 times and taking the average value on the Big-Vul dataset. It can be observed that the use of PL and NL combined feature produced better results in terms of *Accuracy*, *Precision*, and *F1-Score* metrics than the use of PL features alone, but slightly lower in *Recall* indicator. To better evaluate the effect of both inputs on the effectiveness of SDP, we used *Receiver Operating Characteristic* curve and calculated the respective Area Under Curve (AUC) score values, as shown in Fig. 6. As result, the AUC values for the combined PL and NL features are higher than those for the PL-only input, which indicates that for a defective sample, the NL features can help the model improve the probability of being predicted as defective.

Table 7
Results of PL-NL and PL-only features on four metrics.

Model	Big-Vul dataset			
	Accuracy	Precision	Recall	F1-score
PL-only	0.9933	0.9707	**0.9100**	0.9393
PL-NL	**0.9937**	**0.9785**	0.9081	**0.9420**

It can be seen that NL features have a positive but small effect on defect prediction, which may be limited to the quality of the text sequences.

6. Threats to validity

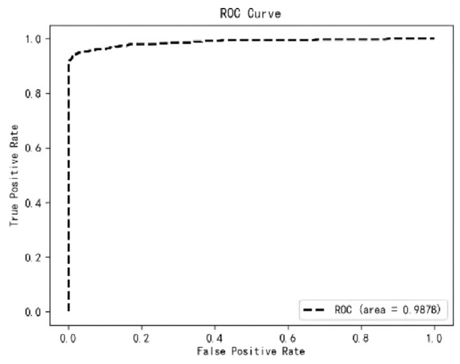
Threats to External Validity are related to the generalization of the proposed model. We conducted experiments using the Big-Vul dataset (Fan et al., 2020), which is a large-scale line-level vulnerability dataset. Other datasets (Zhou et al., 2019; Chakraborty et al., 2022) were not used because they only provided the ground truth at the functional level without including other knowledge, which did not meet our needs. While the dataset used in the study by Xu et al. (2021) provides descriptive information such as commit messages in the repositories, the small sample size carried the risk of overfitting. Therefore, we finally chose to use the Big-Vul dataset and made a fair comparison with other models using this dataset. In future work, datasets containing external knowledge related to defects can be constructed, and the generality of the model could be explored on other line-level vulnerability datasets.

Threats to Internal Validity are concerned with factors that may affect a dependent variable, which is the hyperparameter setting in our study. Different values of hyperparameters were experimented manually. Considering the use of pretrained language models in the method, we used the default hyperparameter settings, and the learning rate was set to 2e−5. In our approach, the multi-channel CNN sets up two sizes of convolution kernels at 3 and 4. We tried to adjust the batch_size to 4, 8, and 16, respectively; set the parameter *weight_decay* of the optimizer AdamW to four values of 0, 1e−4, 5e−4, and 1e−5; and experimented the number of filters of 4, 100, and 200. The best performing results were selected based on the above combinations, in this case it means that better combinations of hyperparameters may exist. More rigorous hyperparameter search methods, such as grid search and random search, will be adopted in future work to continue exploring the selection of hyperparameters.

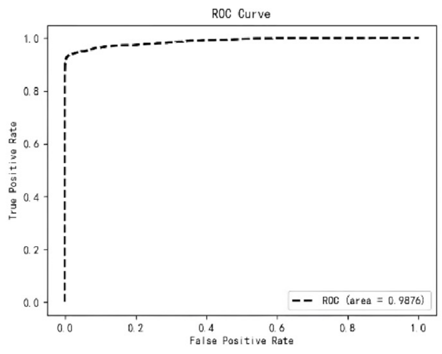
In RQ1, the results of IVDetect (Li et al., 2021) were directly reused from the experimental results in the paper without our validation. We tried the provided open source code to the best of our ability, but were not able to find a way to rerun it, which has been also mentioned by others (Fu and Tantithamthavorn, 2022). Therefore, we followed the identical experimental setup as the baseline model to eliminate bias and chose the same metrics to evaluate the performance of our proposed model.

7. Conclusion

In this paper, a vulnerability prediction method integrating Transformer architecture and a multi-channel CNN, PM2-CNN, is



(a) ROC curve for PL-NL feature



(b) ROC curve for PL-only feature

Fig. 6. ROC curve for PL-NL feature and PL-only feature.

proposed, which feed source code and text sequences into the model to identify potentially defective code modules. PM2-CNN performs better than the baseline model on large commonly used datasets, showing that incorporating external knowledge can produce better feature representations that reflect the semantics of source code defects, which has a positive impact on SDP. In the future, different types and different ways of adding external knowledge in feature embeddings are considered to be explored to further improve the performance of SDP models.

CRediT authorship contribution statement

Jingyu Liu: Conceptualization, Methodology, Software, Writing – original draft. **Jun Ai:** Supervision, Conceptualization. **Minyan Lu:** Supervision, Writing – review & editing. **Jie Wang:** Writing – review & editing. **Haoxiang Shi:** Investigation, Data curation.

Declaration of competing interest

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.

Data availability

Data will be made available on request

References

Ai, J., Su, W., Zhang, S., Yang, Y., 2019. A software network model for software structure and faults distribution analysis. IEEE Trans. Reliab. 68, 844–858. http://dx.doi.org/10.1109/TR.2019.2909786.

Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P., Misilov, V.E., 2021. A Survey on Software Defect Prediction using Deep Learning, Mathematics. Multidisciplinary Digital Publishing Institute, http://dx.doi.org/10.3390/math9111180.

Aleem, S., Capretz, L.F., Ahmed, F., 2015. Benchmarking machine learning technologies for software defect detection. Int. J. Softw. Eng. Appl. 6, 11–23. http://dx.doi.org/10.48550/arxiv.1506.07563.

Alsaeedi, A., Khan, M.Z., Alsaeedi, A., Khan, M.Z., 2019. Software defect prediction using supervised machine learning and ensemble techniques: A comparative study. J. Softw. Eng. Appl. 12, 85–100. http://dx.doi.org/10.4236/JSEA.2019.125007.

Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M., 2012. Graph-based analysis and prediction for software evolution. In: Proc. - Int. Conf. Softw. Eng.. pp. 419–429. http://dx.doi.org/10.1109/ICSE.2012.6227173.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2022. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48, 3280–3296. http://dx.doi.org/10.1109/TSE.2021.3087402.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493. http://dx.doi.org/10.1109/32.295895.

Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2021. Automatic feature learning for predicting vulnerable software components. IEEE Trans. Softw. Eng. 47, 67–85. http://dx.doi.org/10.1109/TSE.2018.2881961.

Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. In: NAACL HLT 2019-2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf., Vol. 1. pp. 4171–4186. http://dx.doi.org/10.48550/arxiv.1810.04805.

Elish, K.O., Elish, M.O., 2008. Predicting defect-prone software modules using support vector machines. J. Syst. Softw. 81, 649–660. http://dx.doi.org/10.1016/J.JSS.2007.07.040.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. - 2020 IEEE/ACM 17th Int. Conf. Min. Softw. Repos. MSR 2020. pp. 508–512. http://dx.doi.org/10.1145/3379597.3387501.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Find. Assoc. Comput. Linguist. Find. ACL EMNLP 2020. pp. 1536–1547. http://dx.doi.org/10.48550/arxiv.2002.08155.

Fu, M., Tantithamthavorn, C., 2022. LineVul: A transformer-based line-level vulnerability prediction. In: Proceedings - 2022 Mining Software Repositories Conference, MSR 2022. Association for Computing Machinery, http://dx.doi.org/10.1145/3524842.3528452.

Giray, G., Bennin, K.E., Köksal, Ö., Babur, Ö., Tekinerdogan, B., 2023. On the use of deep learning in software defect prediction. J. Syst. Softw. 195, 111537. http://dx.doi.org/10.1016/J.JSS.2022.111537.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. UniXcoder: Unified cross-modal pre-training for code representation. pp. 7212–7225. http://dx.doi.org/10.18653/v1/2022.acl-long.499.

Hoang, T., Khanh Dam, H., Kamei, Y., Lo, D., Ubayashi, N., 2019. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In: IEEE Int. Work. Conf. Min. Softw. Repos. 2019-May. pp. 34–45. http://dx.doi.org/10.1109/MSR.2019.00016.

Huo, X., Yang, Y., Li, M., Zhan, D.C., 2018. Learning semantic features for software defect prediction by code comments embedding. In: Proc. - IEEE Int. Conf. Data Mining, ICDM 2018-November. pp. 1049–1054. http://dx.doi.org/10.1109/ICDM.2018.00133.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2019. Learning and evaluating contextual embedding of source code. In: 37th Int. Conf. Mach. Learn. ICML 2020 PartF168147-7. pp. 5066–5077. http://dx.doi.org/10.48550/arxiv.2001.00059.

Krasner, H., 2022. The cost of poor software quality in the US: A 2022 report. Consort. Inf. Softw. Qual. (CISQ) http://dx.doi.org/10.1515/9781400871308-001.

Li, Y., Eric Wong, W., Lee, S.Y., Wotawa, F., 2019. Using tri-relation networks for effective software fault-proneness prediction. IEEE Access 7, 63066–63080. http://dx.doi.org/10.1109/ACCESS.2019.2916615.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: Proc. - 2017 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS 2017. pp. 318–328. http://dx.doi.org/10.1109/QRS.2017.42.

Li, Y., Wang, S., Nguyen, T.N., 2021. Vulnerability detection with fine-grained interpretations. In: ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, Inc, pp. 292–303. http://dx.doi.org/10.1145/3468264.3468597.

Loshchilov, I., Hutter, F., 2019. Decoupled weight decay regularization. In: 7th International Conference on Learning Representations, ICLR 2019.

Madeyski, L., Jureczko, M., 2015. Which process metrics can significantly improve defect prediction models? An empirical study. Softw. Qual. J. 23, 393–422. http://dx.doi.org/10.1007/s11219-014-9241-7.

Miholca, D.L., Czibula, G., 2019. Software defect prediction using a hybrid model based on semantic features learned from the source code. In: Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). In: LNAI, vol. 11775, pp. 262–274. http://dx.doi.org/10.1007/978-3-030-29551-6_23/COVER.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. ArXiv e-prints, page. arXiv Prepr. arXiv:1310.4546 26, 1–9.

Moritz, K., Tomáš, H.†, Kočisk´y, K., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., Blunsom, P., Deepmind, G., 2015. Teaching machines to read and comprehend. Adv. Neural Inf. Process. Syst. 28, 1693–1701.

Muthukumaran, K., Choudhary, A., Murthy, N.L.B., 2015. Mining github for novel change metrics to predict buggy files in software systems. In: Proc. - 1st Int. Conf. Comput. Intell. Networks, CINE 2015. pp. 15–20. http://dx.doi.org/10.1109/CINE.2015.13.

Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Proc. - Int. Conf. Softw. Eng. 2005. pp. 284–292. http://dx.doi.org/10.1109/ICSE.2005.1553571.

Omri, S., Sinz, C., 2020. Deep learning for software defect prediction: A survey. In: Proc. - 2020 IEEE/ACM 42nd Int. Conf. Softw. Eng. Work. ICSEW 2020. pp. 209–214. http://dx.doi.org/10.1145/3387940.3391463.

Özakıncı, R., Tarhan, A., 2018. Early software defect prediction: A systematic map and review. J. Syst. Softw. 144, 216–239. http://dx.doi.org/10.1016/j.jss.2018.06.025.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., Abraham, A., 2022. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. Eng. Appl. Artif. Intell. http://dx.doi.org/10.1016/j.engappai.2022.104773.

Prasad, M.C.M., Florence, L.F., Arya3, A., 2015. A study on software metrics based software defect prediction using data mining and machine learning techniques. Int. J. Database Theory Appl. 8, 179–190. http://dx.doi.org/10.14257/ijdta.2015.8.3.15.

Purao, S., Vaishnavi, V., 2003. Product metrics for object-oriented systems. ACM Comput. Surv. http://dx.doi.org/10.1145/857076.857090.

Qiu, S., Lu, L., Cai, Z., Jiang, S., 2019. Cross-project defect prediction via transferable deep learning-generated and handcrafted features. In: Proc. Int. Conf. Softw. Eng. Knowl. Eng. SEKE 2019-July. pp. 431–436. http://dx.doi.org/10.18293/SEKE2019-070.

Siow, J.K., Liu, S., Xie, X., Meng, G., Liu, Y., 2022. Learning program semantics with code representations: An empirical study. In: Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022. pp. 554–565. http://dx.doi.org/10.1109/SANER53432.2022.00073.

Tiwari, S., Rathore, S.S., 2018. Coupling and cohesion metrics for object-oriented software: A systematic mapping study. In: ACM Int. Conf. Proceeding Ser.. http://dx.doi.org/10.1145/3172871.3172878.

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 Comput. 1–15. http://dx.doi.org/10.1007/s00500-022-06830-5.

Wang, S., Liu, T., Nam, J., Tan, L., 2020. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46, 1267–1293. http://dx.doi.org/10.1109/TSE.2018.2877612.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proc. - Int. Conf. Softw. Eng. 14-22-May-2016. pp. 297–308. http://dx.doi.org/10.1145/2884781.2884804.

White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: IEEE International Working Conference on Mining Software Repositories. pp. 334–345. http://dx.doi.org/10.1109/MSR.2015.38.

Wong, W.E., Debroy, V., Surampudi, A., Kim, H., Siok, M.F., 2010. Recent catastrophic accidents: Investigating how software was responsible. In: SSIRI 2010-4th IEEE International Conference on Secure Software Integration and Reliability Improvement. pp. 14–22. http://dx.doi.org/10.1109/SSIRI.2010.38.

Wong, W.E., Horgan, J.R., Syring, M., Zage, W., Zage, D., 2000. Applying design metrics to predict fault-proneness: A case study on a large-scale software system. Softw. Pract. Exp. 30, 1587–1608. http://dx.doi.org/10.1002/1097-024X(20001125)30:14<1587::AID-SPE352>3.0.CO;2-1.

Wong, W.E., Li, X., Laplante, P.A., 2017. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. J. Syst. Softw. 133, 68–94. http://dx.doi.org/10.1016/J.JSS.2017.06.069.

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., Tang, Y., 2019. LDFR: Learning deep feature representation for software defect prediction. J. Syst. Softw. 158, 110402. http://dx.doi.org/10.1016/J.JSS.2019.110402.

Xu, J., Wang, F., Ai, J., 2021. Defect prediction with semantics and context features of codes based on graph representation learning. IEEE Trans. Reliab. 70, 613–625. http://dx.doi.org/10.1109/TR.2020.3040191.

Yang, Y., Ai, J., Wang, F., 2018. Defect prediction based on the characteristics of multilayer structure of software network. In: Proc. - 2018 IEEE 18th Int. Conf. Softw. Qual. Reliab. Secur. Companion, QRS-C 2018. pp. 27–34. http://dx.doi.org/10.1109/QRS-C.2018.00019.

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. J. Syst. Softw. 184, 111108. http://dx.doi.org/10.1016/J.JSS.2021.111108.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems.

Jingyu Liu received her B.S. degree in software engineering from the school of Information and Software Engineering at University of Electronic Science and Technology of China in 2019. She is currently a Ph.D. student in the School of Reliability and Systems Engineering at Beihang University. Her research interests include software defect prediction, and deep learning.

Jun Ai received his Ph.D. degree in engineering from the School of Reliability and System Engineering at Beihang University in 2006. He is currently a professor in the School of Reliability and Systems Engineering at Beihang University. His research interests include software reliability, software defect prediction, software quality evaluation, and software networks.

Minyan Lu has been a professor and Ph.D. supervisor of Beihang University, China, since 2006. Her main research interests include software reliability engineering, software reliability evaluation and testing, software dependability.

Jie Wang is a Ph.D. candidate in School of Reliability and Systems Engineering at Beihang University, Beijing, China. She received the master's degree in communication and information system from Communication University of China, Beijing, China, in 2018. Her research interests include software reliability engineering, artificial intelligence system.

Haoxiang Shi received his B.S. degree in Air Vehicle Quality and Reliability from the School of Reliability and System Engineering at Beihang University, Beijing, China in 2021. His research interests include machine learning, software defect prediction, and deep learning.