EL-CodeBert: Better Exploiting CodeBert to Support Source Code-Related Classification Tasks

Ke Liu
School of Information Science and Technology, Nantong University
Nantong, China
aurora.ke.liu@outlook.com

Guang Yang
School of Information Science and Technology, Nantong University
Nantong, China
novelyg@outlook.com

Xiang Chen*
School of Information Science and Technology, Nantong University
Nantong, China
xchencs@ntu.edu.cn

Yanlin Zhou
School of Information Science and Technology, Nantong University
Nantong, China
1159615215@qq.com

ABSTRACT

With the development of deep learning and natural language processing techniques, the performance of many source code-related tasks can be improved by using pre-trained models. Of these pre-trained models, CodeBert is a bi-modal pre-trained model for programming languages and natural languages, which has been successfully used in current source code-related tasks. These previous studies mainly use the output vector of CodeBert's last layer as the code semantic representation for fine-tuning downstream source code-related tasks. However, this setting may miss the valuable representational information, which may be captured by other layers of CodeBert.

To better exploit the representational information in each layer of CodeBert for fine-tuning downstream source code-related tasks, we propose an approach `EL-CodeBert`. Our approach first extracts the representational information in each layer of CodeBert and views them as a representational information sequence. Then our approach learns the importance of representational information in each layer through the bidirectional recurrent neural network (i.e., Bi-LSTM) and the attention mechanism. To verify the effectiveness of our proposed approach, we select four downstream source code-related classification tasks (i.e., code smell classification, code language classification, technical debt classification, and code comment classification). After compared with state-of-the-art baselines for these tasks, `EL-CodeBert` can achieve better performance in most performance measures. Finally, we also conduct ablation studies to verify the rationality of the component setting in our proposed approach.

*Xiang Chen is the corresponding author.

Internetware 2022, June 11-12, 2022, Hohhot, China

ACM ISBN 978-1-4503-9780-3/22/06…$15.00
https://doi.org/10.1145/3545258.3545260

CCS CONCEPTS

• **Computer methodologies** → **Supervised learning; Artificial intelligence**.

KEYWORDS

Source code-related task, Pre-trained model, CodeBert, Fine-tuning

ACM Reference Format:
Ke Liu, Guang Yang, Xiang Chen, and Yanlin Zhou. 2022. `EL-CodeBert`: Better Exploiting CodeBert to Support Source Code-Related Classification Tasks. In *Proceedings of The 13th Asia-Pacific Symposium on Internetware (Internetware 2022).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3545258.3545260

1 INTRODUCTION

Large-scale pre-trained models (such as Bert [13], RoBerta [29], GPT [35], and T5 [36]) have shown promising performance in a variety of natural language processing (NLP) tasks [33]. These pre-trained models typically follow a paradigm of pre-training a model on a large general-purpose corpus and then fine-tuning the trained model for downstream tasks.

The success of the pre-trained models in NLP tasks [13, 25, 34, 46] has inspired many studies in the software engineering domain, which apply the pre-trained models (such as CodeBert [15], GraphCodeBert [19], T5 [36]) to source code-related tasks. These pre-trained models can achieve better performance on these source code-related tasks [28]. Some of the previous studies [10, 22] found that Bert could capture a rich hierarchy of linguistic information in NLP tasks, with surface features at the lower level, syntactic features at the middle level, and semantic features at the higher level. Currently, a common practice in Bert-based models is to only use the first token position ('[CLS]') of the last layer of Bert output as the representation of the whole sentence. However, this setting may miss the valuable information in NLP tasks, which can be captured by other layers (such as the low-level layers and the middle-level layers) of Bert [24, 26, 40].

To our best knowledge, whether using the representational information in each layer of CodeBert can improve the performance of downstream source code-related tasks has not been investigated in previous studies. To fill this gap, we propose a novel approach `EL-CodeBert` to fully exploit the semantic information captured in

CodeBert. Specifically, CodeBert uses a 12-layer encoder for pre-training. The shallower the encoder layer in the CodeBert model, the lower level of semantic information is represented. While the deeper the encoder layer, the higher level of semantic information is represented [22]. Therefore, we make full use of the extracted representational information from each layer through the bidirectional recurrent neural network (i.e., Bi-LSTM). Then we use the attention mechanism to assign weights for each layer based on their importance for the specific source code-related task. To verify the effectiveness of our proposed approach `EL-CodeBert`, we select four downstream tasks related to source code (i.e., code smell classification, code language classification, technical debt classification, and code comment classification). The experimental results show that our proposed approach can achieve state-of-the-art performance on these four tasks in most performance measures. Finally, we also perform ablation studies to verify the rationality of the component setting in our proposed approach for different tasks.

To our best knowledge, the main contributions of our study can be summarized as follows.

- We propose a novel approach `EL-CodeBert`, which can better exploit the representational information learned by each layer of CodeBert by using Bi-LSTM and the attention mechanism.
- We apply our approach for four classical downstream source code-related tasks. These tasks are code smell classification, code language classification, technical debt classification, and code comment classification. Empirical study results show the effectiveness of our approach and the rationality of the component setting in our approach.
- We share our scripts and trained models in our project homepage[1] to facilitate the replication of our study and encourage more follow-up studies on this research topic.

2 RELATED WORK

2.1 Code Representation Learning

With the introduction of the code naturalness assumption [1] and the development of deep learning techniques, code representation learning has become an important research topic for software analytics.

In this subsection, we summarize related representative studies on code representation learning. DeFreez et al. [12] proposed the Func2Vec method to solve the path explosion problem of the code, which converts the execution path into a sequence of labels, mapping the labels into continuous real-valued vectors with the help of the Word2Vec [30] method. Xu et al. [42] extracted the features through program slicing. Based on the features, they used natural language processing to analyze programs with source code. Alon et al. [2] proposed a Code2Vec model that represents code fragments as continuous distribution vectors. Zhang et al. [47] proposed a novel AST-based Neural Network for source code representation. It encodes the statement trees to vectors by capturing the lexical and syntactical knowledge of statements. Yang et al. [43] proposed a novel method ComFormer based on Transformer and fusion method-based hybrid code presentation. Yang et al. [44] used the Transformer encoder to carry out encoding for the source code and used a CNN-based code function extractor to research the expertise of nearby features. Feng et al. [15] constructed a bimodal pre-trained model CodeBert for programming languages and natural languages, which was dedicated to learning a generic representation of source code. Recently, Jain et al. [21] proposed ContraCode. This method uses a contrastive self-supervised algorithm, which can learn representational invariants to transformations via compiler-based data augmentations.

2.2 Source Code-Related Classification Tasks

In this subsection, we introduce our considered four source code-related tasks and summarize their related studies.

Code Smell Detection. A code smell is a code symptom that is introduced into a program due to design flaws or poor coding habits. Code smells can lead to software quality problems and make software systems difficult to develop and maintain. There are different methods to detect the code smells in the source code. Fontana et al. [4, 16, 17] compared different machine learning techniques (i.e., J48, JRip, Random Forest, Naive Bayes, SMO, and LibSVM) in predicting the severity of code smells (such as God class, data class, long method, and feature envy). Later, Liu et al. [27] proposed a neural network-based approach to automatically increase the size of the corpus. Sharma et al. [39] found that CNN, RNN, and auto-encoder deep learning models can achieve promising performance on code smell detection.

Code Language Classification. Recently, the programming language of source code is either manually assigned or determined by analyzing file extensions [18]. The cost of manual classification can be significantly reduced with the help of an automatic code language classifier. Gilda [18] used multi-layer neural networks and convolutional neural networks to classify source code language. Their trained model can classify 60 programming languages. Alrashedy et al. [3] classified the code language types of code snippets in Stack Overflow. They used Random Forest Classifier (RFC) and XGBoost to build a classifier. Then, they combined the text information of the problems in Stack Overflow with the code snippets to improve the accuracy. Yang et al. [45] applied deep learning and pre-trained models (i.e., a fine-tuned RoBerta model) to source code language classification.

Technical Debt Classification. Technical debt is a metaphor that reflects the trade-off between short-term benefits and long-term stability for developers [7]. Self-admitted technical debt (SATD) is a variant of technical debt that has been proposed to identify debts intentionally introduced during software development [38, 41]. Potdar and Shihab [32] manually summarized 62 patterns that can be used to identify SATD from the comments of Java software projects. Huang et al. [20] proposed a text mining-based SATD identification method. Compared with the 62 patterns, the text mining method proposed by Huang et al. can achieve a significant performance improvement on the F1-score. Ren et al. [38] proposed a convolutional neural network-based approach to classify code comments as SATD or non-SATD.

Code Comment Classification. Code comments are an important software artifact to help developers understand code, but different code comments are aimed at different target groups and intentions.

[1] https://github.com/NTDXYG/EL-CodeBert

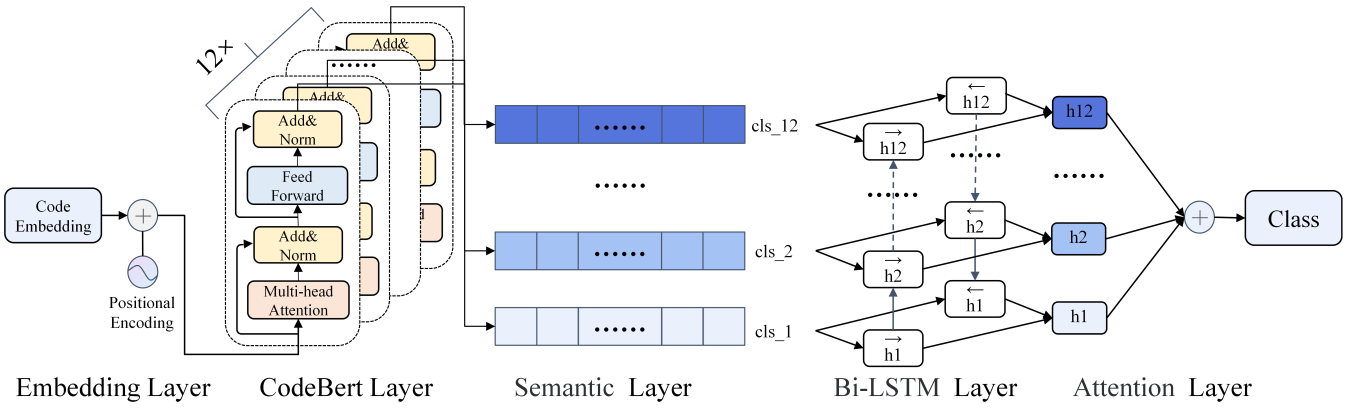


Figure 1: The model architecture of our proposed approach EL-CodeBert

Rani et al. [37] proposed a method to automatically identify class comment types using two techniques (i.e., natural language processing, and text analysis), which can classify frequent class comment information types with high accuracy for three programming languages (i.e., Python, Java, and Smalltalk). Chen et al. [8] classified code comments into six intention categories and manually label 20,000 code-comment pairs. These categories include "what", "why", "how-to-use", "how-it-is-done", "property", and "others".

2.3 Novelty of Our Study

In some of the previous Bert-based studies for NLP tasks, researchers [24, 26, 40] found that only using the representational information of the last layer may miss the valuable information, which can be captured by other layers. To our best knowledge, this issue on CodeBert-based source code-related tasks has not been thoroughly investigated. Although residual layers have been widely used in the Transformer, which can also implicitly help the model retain the information captured by different layers, the original representational information is still fused away because the vector goes through multiple residual layers. Therefore, in this study, we want to take full advantage of the representational information learned in each layer of CodeBert and then propose an approach EL-CodeBert. Our proposed approach extracts the representational information from each layer of CodeBert as the representational information sequence. Then our approach uses the bi-directional recurrent neural network (Bi-LSTM) and the attention mechanism to focus on the layers with the most important representational information for the specific task.

3 OUR PROPOSED APPROACH EL-CODEBERT

3.1 Model Architecture of EL-CodeBert

The model architecture of our proposed approach EL-CodeBert can be found in Figure 1. Our designed model architecture includes Embedding Layer, CodeBert Layer, Semantic Layer, Bi-LSTM Layer, and Attention Layer. We illustrate the details of each layer as follows.

Embedding layer. In this layer, we aim to capture the relationships between tokens by mapping the target input from a textual representation to a vector representation on a high-dimensional space. For the input to the model, we tokenize the inputs by the Byte-level BPE algorithm and obtain the sequence $x = (x_1, \cdots, x_n)$, where n is the length of this sequence. Since the sequence length is different for different inputs, we use a padding operation to unify the sequence length to facilitate the processing of the CodeBert layer. Supposing the maximum input length is N, for the sequences whose length is less than N, we pad 0 to the end of these sequences. For the sequences whose length is larger than N, we directly truncate the end of these sequences. Therefore, the output of the embedding layer is $X = (X_1, \cdots, X_N)$. To allow the model to exploit the sequence's order information, we add the absolute Positional Encoding (APE). Finally, the input to the next layer is $X = X + APE(X)$, $X \in \mathbb{R}^{batch \times N \times d_{model}}$, where $batch$ is the size of the batch size and d_{model} is the size of the embedding dimension.

CodeBert layer. In this layer, we aim to exploit the strong learning capability of the pre-trained model to extract the representational information of the input. CodeBert [15] is a bi-modal pre-training model for programming languages (PL) and natural languages (NL) based on the encoder in Transformer. CodeBert is pre-trained on a large general-purpose corpus by using two tasks: Masked Language Model (MLM) and Replaced Token Detection (RTD). Specifically, the MLM task targets bi-modal data by simultaneously feeding the code with the corresponding comments and randomly selecting positions for masking, then replacing the token with a special [Mask] token, the goal of the MLM task is to predict the original token. The RTD task targets uni-modal data with separate codes and comments and randomly replaces the token, which aims to learn whether the token is the original word utilizing a discriminator.

Semantic layer. In this layer, we aim to extract sufficient information about the semantic representation of the target input. Generally, researchers extract the vector of the first token in the encoder of the last layer of the pre-trained model as an aggregated sequence representation, and add a fully connected feed-forward network for classification. In our proposed approach, we extract the vector of the first token of each layer of CodeBert and concatenate it into a new semantic representation vector X_{sem} in order to make better use of the rich representation information learned by the layers in the pre-trained model. Since CodeBert uses a 12-layer encoder for pre-training, $X_{sem} \in \mathbb{R}^{\text{batch} \times 12 \times d_{\text{model}}}$. Jawahar et al [22] found that the shallower the encoder layer in the Bert model, the more

it represents low-level semantic information, while the deeper the encoder layer in the Bert model, the more it represents higher-level semantic information. Thus X_{sem} can hold semantic progression relationships from shallow to deep layers in CodeBert.

Bi-LSTM layer. In this layer, we aim to take full advantage of the extracted semantic information for further learning. Since the semantic information in X_{sem} has a shallow to deep characteristic, it is natural that we would like to learn all the semantic information through the Bi-LSTM. Based on Figure 1, we can find that for a given $X_{sem} = (X_1, \cdots, X_{12})$, we use a two-layer bi-directional LSTM, each layer using 12 cells for learning respectively. For each semantic message in X_{sem}, it is expressed by the forward-implicit vector and backward-implicit vector learned by the Bi-LSTM as follows.

$$\overrightarrow{h}_i = \overrightarrow{\text{LSTM}}(X_i), i \in [1, 12] \tag{1}$$

$$\overleftarrow{h}_i = \overleftarrow{\text{LSTM}}(X_i), i \in [12, 1] \tag{2}$$

Finally, we use the element-wise sum to combine the forward and backward pass outputs, so that the implicit vector learned by the Bi-LSTM can be expressed as follows.

$$h_i = \left[\overrightarrow{h_i} \oplus \overleftarrow{h_i}\right] \tag{3}$$

Attention layer. In this layer, we aim to combine the semantics of each layer according to the assigned weights of each layer. For a given target input, not all representational information contributes equally to the input. Some source code-related tasks may focus more on the low-level representational information of the code, while others may focus more on the high-level representational information of the code. Therefore we need to use the attention mechanism to extract the more important representational information. Specifically, We first convert h_i to u_i via the full connection layer, $u_i = tanh(Wh_i + b)$. Then the similarity between u_i and the context vector u_w can be calculated and transformed into a probability distribution by Softmax.

$$\alpha_i = \frac{\exp\left(u_i^T u_w\right)}{\sum_i \exp\left(u_i^T u_w\right)} \tag{4}$$

α_i can be treated as the importance of the input for each token, therefore using α_i as a global weighted summation over h_i can generate the input vector X_{out}.

$$X_{out} = \sum_i \alpha_i h_i \tag{5}$$

Finally, for X_{out}, we can classify it by using a two-layer fully connected feedforward network.

$$p(y \mid X_{out}) = W_1(gelu(W_0 X_{out} + b_0)) + b_1 \tag{6}$$

$$\hat{y} = \arg\max_y \hat{p}(y \mid X_{out}) \tag{7}$$

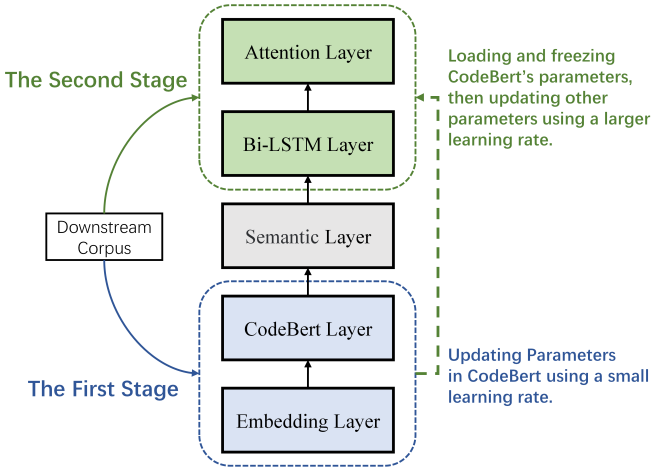


Figure 2: The two-stage fine-tuning process in our proposed approach `EL-CodeBert`

3.2 Two-Stage Fine-tuning Method

As shown in Figure 2, due to the difference in model training between CodeBert and Bi-LSTM, we propose a two-stage fine-tuning method. We use Cross-Entropy Loss as the loss function for model training. Specifically, for the binary-classification task, the model predicts the probability of each class with the formula:

$$L = \frac{1}{N}\sum_{i=1}^{N} -\left[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)\right] \tag{8}$$

where y_i is the label of the i-th sample (we use 1 to denote the positive class, and use 0 to denote the negative class. Then p_i is the probability that the i-th sample is predicted to be positive). N is the number of samples in the corpus.

For the multi-classification task, the formula is

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{M} -y_{ic}\log(p_{ic}) \tag{9}$$

where M represents the number of classes, y_{ic} represents the sign function that takes 1 if the true class of the i-th sample is equal to c, and 0 otherwise. p_{ic} represents the predicted probability that the i-th sample belongs to the class c.

When fine-tuning the model, we first train CodeBert based on the corpus related to the downstream task. We extract the vector corresponding to the first token [CLS] of the last layer of CodeBert, input it to a linear layer, and perform the classification. After the first stage, we can get the parameters' values of the fine-tuned CodeBert model. In the second stage, we first load and freeze these parameters' values, and then fine-tune the other parameters of the model at a larger learning rate.

4 EXPERIMENTAL STUDY

In our empirical studies, we aim to answer the following three research questions (RQs)

RQ1: Can our proposed approach outperform state-of-the-art baselines for different source code-related tasks?

RQ2: Whether using Bi-LSTM and the attention mechanism can improve the performance of our proposed approach?

RQ3: Whether using two-stage fine-tuning can improve the performance of our proposed approach?

In RQ1, we aim to verify the effectiveness of EL-CodeBert on four source code-related classification tasks. To answer this RQ, we conducted a series of empirical studies and compared EL-CodeBert with state-of-the-art baselines. In RQ2 and RQ3, we aim to investigate the performance influence of different components in our proposed approach. To answer these two RQs, we performed extensive ablation studies.

4.1 Experimental Subjects

To verify the effectiveness of our proposed approach, we select four downstream source code-related tasks. Among these tasks, two tasks are related to source code classification and the remaining two tasks are related to code comment classification. To ensure a fair comparison with current state-of-the-art approaches, we followed the original partitioning of the datasets used in these four downstream tasks.

4.1.1 Code Smell Classification. For this task, we use the corpus shared by Fakhoury et al. [14]. The code smells in this corpus come from the use of misleading identifier names or violations of common naming conventions. In this corpus, they labeled about 1,700 code snippets, following the taxonomy of linguistic smells [5]. Therefore, this task is a binary-classification task and the aim of this task is to determine the existence of code smell in the target code snippet.

4.1.2 Code Language Classification. For this task, we use the corpus shared by Alrashedy et al. [3]. They collected code snippets from Stack Overflow, which include 21 programming languages. After using the preprocessing steps proposed by Yang et al. [45], the remaining code snippets only include 19 common programming languages. Therefore, this task is a multi-classification task and the aim of this task is to predict the type of the programming language in the target code snippet.

4.1.3 Technical Debt Classification. For this task, we use the corpus shared by Maldonado et al. [11]. They extracted the code comments from ten Java projects on Github. Each sample in the corpus is assigned to one of five self-admitted technical debt categories (i.e., design debt, requirement debt, defect debt, documentation debt, or test debt) or does not contain any technical debt. In our study, we model this task as a binary classification problem. Specifically, for a given code comment, we aim to identify whether this code comment contains technical debt or not.

4.1.4 Code Comment Classification. For this task, we use the corpus shared by Pascarella et al. [31]. They mined over 11,000 code comments from open-source Java projects and classified them into 16 categories based on their intent and target developers. Therefore, this task is a multi-classification task and this task aims is to predict the code comment as one of 16 categories.

In our study, we use stratified sampling to split the corpus into a training set and a test set according to 80%:20% for each task. Detailed statistical information of these corpora is shown in Table 1. This statistical information includes (1) the number of the training set and the test set, (2) the average, mode, median value of the code/comment length, and (3) the percentage of the samples with size <32, <64, <128, and <256.

4.2 Performance Measures

For the binary-classification tasks, we select Accuracy, Precision, Recall, and F1-Score as the performance measures. These performance measures can be calculated as follows.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{11}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{12}$$

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{13}$$

where TP means that a positive sample is predicted as the positive class, TN means that a negative sample is predicted as the negative class, FP means that a negative sample is predicted as the positive class, and FN means that a positive sample is predicted as the negative class.

For the multi-classification tasks, we use the Macro method to calculate these performance measures. Specifically, we count TP, FP, FN, and TN for each class, and then calculate Precision, Recall, and F1-Score respectively, and finally average them to obtain the Macro-Precision, Macro-Recall, and Macro-F1-Score.

4.3 Baselines

In our study, we consider seven state-of-the-art baselines (i.e., Random Forest, XGBoost, TextCNN, AttBLSTM, Bert, RoBerta, and CodeBert) for our considered four source code-related classification tasks. The brief introduction of these baselines is listed as follows.

Random Forest. This baseline [6] is based on the decision tree algorithm and bagging algorithm in ensemble learning. We implement this baseline by scikit-learn[2].

XGBoost. This baseline [9] is based on the regression tree algorithm and the boosting algorithm in ensemble learning. We implement it by xgboost[3].

TextCNN. This baseline [23] is based on convolutional neural networks. We implement it[4] according to its original description.

AttBLSTM. This baseline [48] is based on the recurrent neural network and the attention mechanisms. We implement it[5] according to its original description.

Bert. This baseline [13] is based on Transformer and pretrain tasks. It performs self-supervised learning on a large-scale corpus with two pre-training tasks (i.e., MLM and NSP). We implement it by Bert-base model [6].

[2] https://github.com/scikit-learn/scikit-learn
[3] https://github.com/dmlc/xgboost
[4] https://github.com/NTDXYG/Text-Classify-based-pytorch/blob/master/model/TextCNN.py
[5] https://github.com/NTDXYG/Text-Classify-based-pytorch/blob/master/model/TextRNN_Attention.py
[6] https://huggingface.co/bert-base-cased

Table 1: Corpus Statistics for our considered four source code-related classification tasks

Task	Problem Type	Training	Test	Avg	Mode	Median	<32	<64	<128	<256
Code Smell	Binary-Classification	1439	350	41.97	8	20.0	68.19%	85.56%	93.14%	97.64%
Code Language	Multi-Classification	179,556	44,889	58.58	2	31.0	51.04%	73.76%	89.78%	96.94%
Technical Debt	Binary-Classification	31,708	6,652	9.27	3	6.0	95.99%	99.23%	99.87%	99.98%
Code Comment	Multi-Classification	8,895	2,247	15.82	2	7.0	88.46%	93.88%	97.72%	99.97%

Table 2: Comparison results on code smell classification task

Approach	ACC (%)	P (%)	R (%)	F1 (%)
Random Forest	78.286	78.880	77.548	77.756
XGBoost	75.714	75.667	75.305	75.409
TextCNN	80.000	80.016	79.641	79.761
AttBLSTM	78.857	78.810	78.537	78.631
Bert	79.714	79.580	79.653	79.610
RoBerta	81.143	81.014	81.067	81.038
CodeBert	85.429	85.516	85.128	85.264
EL-CodeBert	**86.000**	**85.990**	**85.795**	**85.874**

RoBerta. This baseline [29] is also based on Transformer and pre-train tasks. It uses a larger corpus than Bert and is pre-trained using only the MLM method. We implement it by Roberta-base model [7].
CodeBert. This baseline [15] is based on Transformer and source code-related pretraining tasks (i.e., MLM and RTD). We implement it by CodeBert-base model [8].

Due to the limitation of paper length, we show the parameter setting in our proposed approach and baselines in our project homepage.

We run all the experiments on a computer with an Intel(R) Xeon(R) Silver 4210 CPU and a GeForce RTX3090 GPU with 24 GB memory. The running OS platform is Linux OS.

5 RESULT ANALYSIS

In this section, we first compare our proposed approach EL-CodeBert with baselines on four downstream tasks. Then we verify the effectiveness of the component setting in our proposed approach through ablation studies.

5.1 Result Analysis for RQ1

Code Smell Classification. The comparison results between EL-CodeBert and baselines can be found in Table 2. In this table, the first two approaches are traditional machine learning approaches, while the last six approaches are deep learning approaches. For this task, deep learning approaches can achieve better performance than traditional machine learning approaches. EL-CodeBert can achieve the best performance in terms of four performance measures.
Code Language Classification. The comparison results between EL-CodeBert and baselines can be found in Table 3. For this task, we can find that the approaches based on pre-trained models (i.e., Bert, RoBerta, CodeBert, and EL-CodeBert) can achieve better performance than the other approaches. One possible reason is that the pre-trained models can help to extract effective information from the code. Of these pre-trained model-based approaches, EL-CodeBert can achieve the best performance in terms of four performance measures.

Table 3: Comparison results on code language classification task

Approach	ACC (%)	P (%)	R (%)	F1 (%)
Random Forest	78.728	79.362	78.825	78.874
XGBoost	78.803	79.925	78.891	79.217
TextCNN	82.662	83.561	82.706	82.964
AttBLSTM	79.035	79.801	79.107	79.272
Bert	86.865	87.129	86.938	86.985
RoBerta	87.202	87.424	87.276	87.135
CodeBert	87.418	88.042	87.450	87.614
EL-CodeBert	**87.959**	**88.177**	**88.023**	**88.077**

Table 4: Comparison results on technical debt classification task

Approach	ACC (%)	P (%)	R (%)	F1 (%)
Random Forest	97.278	**95.080**	87.019	90.564
XGBoost	97.294	93.901	88.516	90.991
TextCNN	96.978	93.015	87.258	89.881
AttBLSTM	97.114	93.979	87.177	90.227
Bert	96.783	91.229	88.004	89.534
RoBerta	97.595	93.352	91.110	92.279
CodeBert	97.835	94.197	91.991	93.059
EL-CodeBert	**97.850**	94.024	**92.310**	**93.146**

Technical Debt Classification. The comparison results between EL-CodeBert and baselines can be found in Table 4. For this task, we find EL-CodeBert can achieve the best performance except for the performance measure Precision. However, EL-CodeBert can achieve better performance in terms of F1-Score by improving the performance in terms of Recall.
Code Comment Classification. The comparison results between EL-CodeBert and baselines can be found in Table 5. For this task, we find EL-CodeBert can still achieve the best performance except for the performance measure Precision. However, EL-CodeBert can also achieve better performance in terms of F1-Score by improving the performance in terms of Recall.

Summary for RQ1: EL-CodeBert can outperform state-of-the-art baselines in our consider four source code-related classification tasks in terms of most performance measures.

[7] https://huggingface.co/roberta-base
[8] https://huggingface.co/microsoft/codebert-base

Table 5: Comparison results on code comment classification task

Approach	ACC (%)	P (%)	R (%)	F1 (%)
Random Forest	90.921	83.783	75.104	74.618
XGBoost	90.565	77.561	68.661	71.645
TextCNN	91.945	87.541	78.596	80.977
AttBLSTM	92.345	85.578	78.268	80.596
Bert	94.482	87.149	83.935	85.275
RoBerta	94.393	**90.525**	86.121	86.875
CodeBert	94.838	87.916	86.301	86.820
EL-CodeBert	**95.238**	89.395	**87.280**	**87.977**

Table 6: Ablation study results on using the attention mechanism and Bi-LSTM

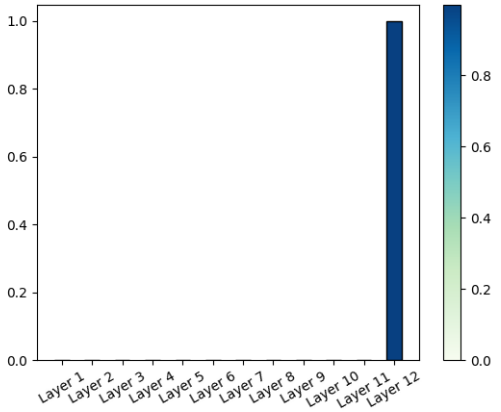
Task	Approach	ACC (%)	P (%)	R (%)	F1 (%)
Code Smell	Full Model	**86.000**	**85.990**	**85.795**	**85.874**
	w/o Bi-LSTM	85.714	85.724	85.484	85.578
	w/o Attention	85.714	85.840	85.392	85.544
Code Language	Full Model	**87.959**	**88.177**	**88.023**	**88.077**
	w/o Bi-LSTM	87.754	87.974	87.826	87.878
	w/o Attention	87.736	87.935	87.813	87.855
Technical Debt	Full Model	97.850	94.024	92.310	93.146
	w/o Bi-LSTM	97.850	93.480	**93.008**	93.242
	w/o Attention	**97.910**	**94.743**	91.878	**93.252**
Code Comment	Full Model	**95.238**	89.395	87.280	87.977
	w/o Bi-LSTM	95.060	88.330	87.250	87.466
	w/o Attention	94.927	**92.043**	**87.416**	**88.243**

5.2 Result Analysis for RQ2

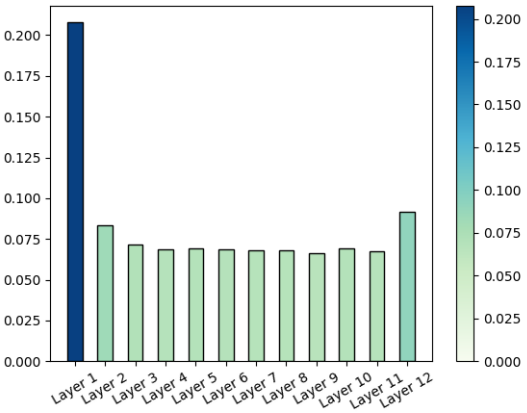
In RQ2, we conduct an ablation study to investigate the contribution of two components (i.e., the attention mechanism and Bi-LSTM) in our proposed approach. Specifically, we compared the performance of EL-CodeBert on four selected tasks: code smell classification, code language classification, technical debt classification, and code comment classification.

As shown in Table 6, we find that removing either of these components can reduce the performance of two tasks (i.e., code smell classification and code language classification). On the contrary, in the two tasks (i.e., technical debt classification and code comment classification), removing the attention mechanism component may instead achieve better performance. This suggests that all components help our approach to better capture syntactic and semantic information from the code, but the effect of each component differs across different tasks. Specifically, both the attention mechanism and Bi-LSTM improve the performance of our approaches for code smell classification and code language classification, but for technical debt classification and code comment classification, our approach performs better with only using the Bi-LSTM component.

Then we aim to explain why using Bi-LSTM and the attention mechanism can further learn the semantic information extracted by the CodeBert model and achieve better performance. Taking the code smell classification task as an example, we provide two in-depth examples of the contributions of Bi-LSTM and the attention mechanism to our approach. We can find from the visualization of Example 1 that our approach focuses almost exclusively on the vectors extracted by the last encoder layer of the CodeBert model, but in Example 2, our approach focuses on both the vectors extracted by the first layer and the last encoder layer of the CodeBert model. This suggests that while most of the samples in the code smell classification task require sufficient information to be extracted from only the vectors extracted by the last layer of the CodeBert model's encoder, there are still some samples that require not only the semantic vectors extracted by the last layer of the encoder but also lower-level semantic information from the other layers.



(a) Example1. public void <w> To Source </w> (<w> String Builder </w> sb) { sb . append (<w> string Map </w> . get (Op)) ; Child . <w> To Source </w> (sb) ;



(b) Example2. public int <w> get Doc Count </w> ( ) return <w> doc Count </w> ;

Figure 3: Two examples of our approach to the attention of CodeBert's 12-layer encoder vector in the code smell classification task. Notice <w> and </w> are used to mark the identifier.

Table 7: Ablation study results for different fine-tuning methods

Task	Approach	ACC (%)	P (%)	R (%)	F1 (%)
Code Smell	Two Stage	**86.000**	**85.990**	**85.795**	**85.874**
	Single Stage	84.286	84.180	84.346	84.229
Code Language	Two Stage	**87.959**	**88.177**	**88.023**	**88.077**
	Single Stage	87.553	87.873	87.602	87.694
Technical Debt	Two Stage	**97.850**	**94.024**	**92.310**	**93.146**
	Single Stage	97.670	93.498	91.668	92.559
Code Comment	Two Stage	**95.238**	89.395	**87.280**	**87.977**
	Single Stage	95.060	**93.363**	84.960	86.867

Summary for RQ2: Using Bi-LSTM and the attention mechanism can help our proposed approach `EL-CodeBert` to achieve better performance, especially for the code smell classification and code language classification tasks.

5.3 Result Analysis for RQ3

In RQ3, we further conduct experiments by analyzing the influence of the two-stage fine-tuning method (details of this method can be found in Section 3.2) on the model performance. Here our comparison method is the single-stage fine-tuning method, which fine-tunes all the parameters in `EL-CodeBert` with the same learning rate. Based on the comparison results in Table 7, we can find that the two-stage fine-tuning method can help our approach achieve better performance.

Summary for RQ3: In each of four source code-related downstream tasks, the two-stage Fine-tuning method can help our approach to achieve better performance.

6 THREATS TO VALIDITY

In this section, we mainly analyze the potential threats to the validity of our empirical study.

Internal Threats. The first internal threat is the potential faults in the implementation of our approach. To alleviate this threat, we check our implementation carefully and use mature libraries (such as PyTorch and transformers). The second internal threat is the implementation of our considered baselines. To alleviate this threat, we mainly use the implementation shared by an open-source repository. The third internal threat is the randomness of the initialization of deep learning models. To alleviate this threat, we fixed the random seed for model initialization to ensure the replication of our study.

External Threats. The main external threat is the choice of the corpus for our considered downstream tasks. To alleviate this threat, we select the popular corpus provided by Fakhoury et al. [14], Alrashedy et al. [3], Maldonado et al. [11] and Pascarella et al. [31] for these four tasks respectively. In the future, we want to gather more corpus from more commercial or open-source projects and verify the effectiveness of our proposed approach.

Construct Threats. The construct threat in this study is the performance measures used to evaluate the performance of our proposed approach. To alleviate these threats, we choose four performance measures, which have been used by the previous related studies [3, 11, 14, 31].

7 CONCLUSION AND FUTURE WORK

In this study, to better exploit the representational information in each layer of CodeBert to support source code-related classification tasks, we propose a novel approach `EL-CodeBert` that extracts the representational information in each layer of CodeBert and views them as a semantic sequence. Then our approach learns the importance of each layer of representational information for different tasks through Bi-LSTM and the attention mechanism. Empirical results showed the effectiveness of our proposed approach. Moreover, we also conduct ablation studies to verify the rationality of the component settings in our proposed approach. There are still many follow-up studies in the future. For example, we first want to extend our study to other code-related pre-trained methods (such as GraphCodeBert [19]). We second want to extend our study to support source code-related generation tasks (such as source code summarization).

ACKNOWLEDGMENTS

Ke Liu and Guang Yang have contributed equally to this work and they are co-first authors. This work is supported in part by the National Natural Science Foundation of China (Grant no. 61872263), The Nantong Application Research Plan (Grant No. JC2021124).

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[3] Kamel Alrashedy, Dhanush Dharmaretnam, Daniel M German, Venkatesh Srinivasan, and T Aaron Gulliver. 2020. Scc++: Predicting the programming language of questions and snippets of stack overflow. *Journal of Systems and Software* 162 (2020), 110505.
[4] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.
[5] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.
[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[7] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 47–52.
[8] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.
[9] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.
[10] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. In *ACL 2018-56th Annual Meeting of the Association for Computational Linguistics*, Vol. 1. Association for Computational Linguistics, 2126–2136.
[11] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[12] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 423–433.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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)*. 4171–4186.

[14] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. 2018. Keep it simple: Is deep learning good for linguistic smell detection?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 602–611.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 1536–1547.

[16] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58.

[17] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. 2013. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 396–399.

[18] Shlok Gilda. 2017. Source code classification using Neural Networks. In *2017 14th international joint conference on computer science and software engineering (JCSSE)*. IEEE, 1–6.

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[20] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[21] Paras Jain and Ajay Jain. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

[22] Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. 2019. What does BERT learn about the structure of language?. In *ACL 2019-57th Annual Meeting of the Association for Computational Linguistics*.

[23] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1746–1751.

[24] Dan Kondratyuk and Milan Straka. 2019. 75 Languages, 1 Model: Parsing Universal Dependencies Universally. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2779–2795.

[25] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations*.

[26] Meiling Li, Xiumei Li, Junmei Sun, and Xinrui He. 2021. STCP: An Efficient Model Combining Subject Triples and Constituency Parsing for Recognizing Textual Entailment. In *International Conference on Artificial Neural Networks*. Springer, 284–296.

[27] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* (2019).

[28] Ke Liu, Guang Yang, Xiang Chen, and Chi Yu. 2022. SOTitle: A Transformer-based Post Title Generation Approach for Stack Overflow. In *Proceedings of The 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 566–577.

[29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[30] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[31] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.

[32] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 91–100.

[33] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (2020), 1872–1897.

[34] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[37] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of Systems and Software* 181 (2021), 111047.

[38] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM transactions on software engineering and methodology (TOSEM)* 28, 3 (2019), 1–45.

[39] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software* 176 (2021), 110936.

[40] Ta-Chun Su and Hsiang-Chih Cheng. 2020. SesameBERT: attention for anywhere. In *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 363–369.

[41] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 179–188.

[42] Aidong Xu, Tao Dai, Huajun Chen, Zhe Ming, and Weining Li. 2018. Vulnerability detection for source code using contextual LSTM. In *2018 5th International Conference on Systems and Informatics (ICSAI)*. IEEE, 1225–1230.

[43] Guang Yang, Xiang Chen, Jinxin Cao, Shuyuan Xu, Zhanqi Cui, Chi Yu, and Ke Liu. 2021. ComFormer: Code Comment Generation via Transformer and Fusion Method-based Hybrid Code Representation. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 30–41.

[44] Guang Yang, Yanlin Zhou, Xiang Chen, and Chi Yu. 2021. Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer. In *28th Asia-Pacific Software Engineering Conference (APSEC)*. 213–222.

[45] Guang Yang, Yanlin Zhou, Chi Yu, and Xiang Chen. 2021. DeepSCC: Source Code Classification Based on Fine-Tuned RoBERTa. *arXiv preprint arXiv:2110.00914* (2021).

[46] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[47] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[48] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. 2016. Attention-based bidirectional long short-term memory networks for relation classification. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*. 207–212.