GRAPHCODEBERT: PRE-TRAINING CODE REPRESENTATIONS WITH DATA FLOW

Anonymous authors
Paper under double-blind review

ABSTRACT

Pre-trained models for programming language have achieved dramatic empirical improvements on a variety of code-related tasks such as code search, code completion, code summarization, etc. However, existing pre-trained models regard a code snippet as a sequence of tokens, while ignoring the inherent structure of code, which provides crucial code semantics and would enhance the code understanding process. We present GraphCodeBERT, a pre-trained model for programming language that considers the inherent structure of code. Instead of taking syntactic-level structure of code like abstract syntax tree (AST), we use data flow in the pre-training stage, which is a semantic-level structure of code that encodes the relation of "where-the-value-comes-from" between variables. Such a semantic-level structure is neat and does not bring an unnecessarily deep hierarchy of AST, the property of which makes the model more efficient. We develop GraphCodeBERT based on Transformer. In addition to using the task of masked language modeling, we introduce two structure-aware pre-training tasks. One is to predict code structure edges, and the other is to align representations between source code and code structure. We implement the model in an efficient way with a graph-guided masked attention function to incorporate the code structure. We evaluate our model on four tasks, including code search, clone detection, code translation, and code refinement. Results show that code structure and newly introduced pre-training tasks can improve GraphCodeBERT and achieves state-of-the-art performance on the four downstream tasks. We further show that the model prefers structure-level attentions over token-level attentions in the task of code search.

1 INTRODUCTION

Pre-trained models such as ELMo (Peters et al., 2018), GPT (Radford et al., 2018) and BERT (Devlin et al., 2018) have led to strong improvement on numerous natural language processing (NLP) tasks. These pre-trained models are first pre-trained on a large unsupervised text corpus, and then fine-tuned on downstream tasks. The success of pre-trained models in NLP also promotes the development of pre-trained models for programming language. Existing works (Kanade et al., 2019; Karampatsis & Sutton, 2020; Feng et al., 2020; Svyatkovskiy et al., 2020; Buratti et al., 2020) regard a source code as a sequence of tokens and pre-train models on source code to support code-related tasks such as code search, code completion, code summarization, etc. However, previous works only utilize source code for pre-training, while ignoring the inherent structure of code. Such code structure provides useful semantic information of code, which would benefit the code understanding process. Taking the expression $v = max_value - min_value$ as an example, v is computed from max_value and min_value. Programmers do not always follow the naming conventions so that it's hard to understand the semantic of the variable v only from its name. The semantic structure of code provides a way to understand the semantic of the variable v by leveraging dependency relation between variables.

In this work, we present GraphCodeBERT, a pre-trained model for programming language that considers the inherent structure of code. Instead of taking syntactic-level structure of code like abstract syntax tree (AST), we leverage semantic-level information of code, i.e. data flow, for pre-training. Data flow is a graph, in which nodes represent variables and edges represent the relation of "where-the-value-comes-from" between variables. Compared with AST, data flow is neat and does not bring an unnecessarily deep hierarchy, the property of which makes the model more efficient. In order to learn code representation from source code and code structure, we introduce two new

structure-aware pre-training tasks. One is data flow edges prediction for learning representation from code structure, and the other is variable-alignment across source code and data flow for aligning representation between source code and code structure. GraphCodeBERT is based on Transformer neural architecture (Vaswani et al., 2017) and we extend it by introducing a graph-guided masked attention function to incorporate the code structure.

We pre-train GraphCodeBERT on the CodeSearchNet dataset (Husain et al., 2019), which includes 2.4M functions of six programming languages paired with natural language documents. We evaluate the model on four downstream tasks: natural language code search, clone detection, code translation, and code refinement. Experiments show that our model achieves state-of-the-art performance on the four tasks. Further analysis shows that code structure and newly introduced pre-training tasks can improve GraphCodeBERT and the model has consistent preference for attending data flow.

In summary, the contributions of this paper are: (1) GraphCodeBERT is the first pre-trained model that leverages semantic structure of code to learn code representation. (2) We introduce two new structure-aware pre-training tasks for learning representation from source code and data flow. (3) GraphCodeBERT provides significant improvement on four downstream tasks, i.e. code search, clone detection, code translation, and code refinement.

2 Related Works

Pre-Trained Models for Programming Languages Inspired by the big success of pre-training in NLP (Devlin et al., 2018; Yang et al., 2019; Liu et al., 2019; Raffel et al., 2019), pre-trained models for programming languages also promotes the development of code intelligence (Kanade et al., 2019; Feng et al., 2020; Karampatsis & Sutton, 2020; Svyatkovskiy et al., 2020; Buratti et al., 2020). Kanade et al. (2019) pre-train a BERT model on a massive corpus of Python source codes by masked language modeling and next sentence prediction objectives. Feng et al. (2020) propose CodeBERT, a bimodal pre-trained model for programming and natural languages by masked language modeling and replaced token detection to support text-code tasks such as code search. Karampatsis & Sutton (2020) pre-train contextual embeddings on a JavaScript corpus using the ELMo framework for program repair task. Svyatkovskiy et al. (2020) propose GPT-C, which is a variant of the GPT-2 trained from scratch on source code data to support generative tasks like code completion. Buratti et al. (2020) present C-BERT, a transformer-based language model pre-trained on a collection of repositories written in C language, and achieve high accuracy in the abstract syntax tree (AST) tagging task.

Different with previous works, GraphCodeBERT is the first pre-trained model that leverages code structure to learn code representation to improve code understanding. We further introduce a graph-guided masked attention function to incorporate the code structure into Transformer and two new structure-aware pre-training tasks to learn representation from source code and code structure.

Neural Networks with Code Structure In recent years, some neural networks leveraging code structure such as AST have been proposed and achieved strong performance in code-related tasks like code completion (Li et al., 2017; Alon et al., 2019; Kim et al., 2020), code generation (Rabinovich et al., 2017; Yin & Neubig, 2017; Brockschmidt et al., 2018), code clone detection (Wei & Li, 2017; Zhang et al., 2019; Wang et al., 2020), code summarization (Alon et al., 2018; Hu et al., 2018) and so on (Nguyen & Nguyen, 2015; Allamanis et al., 2018; Hellendoorn et al., 2019). Nguyen & Nguyen (2015) propose an AST-based language model to support the detection and suggestion of a syntactic template at the current editing location. Allamanis et al. (2018) use graphs to represent programs and graph neural network to reason over program structures. Hellendoorn et al. (2019) propose two different architectures using a gated graph neural network and Transformers for combining local and global information to leverage richly structured representations of source code. However, these works leverage code structure to learn models on specific tasks from scratch without using pre-trained models. In this work, we study how to leverage code structure for pre-training code representation.

3 Data Flow

In this section, we describe the basic concept and extraction of data flow. In next section, we will describe how to use data flow for pre-training.

Data flow is a graph that represents dependency relation between variables, in which nodes represent variables and edges represent where the value of each variable comes from. Unlike AST, data flow is same under different abstract grammars for the same source code. Such code structure provides crucial code semantic information for code understanding. Taking $v = max_value - min_value$ as an example, programmers do not always follow the naming conventions so that it is hard to understand the semantic of the variable. Data flow provides a way to understand the semantic of the variable v to some extent, i.e. the value of v comes from max_value and min_value in data flow. Besides, data flow supports the model to consider long-range dependencies induced by using the same variable or function in distant locations. Taking Figure 1 as an example, there are four variables with same name (i.e. x^3, x^7, x^9 and x^{11}) but with different semantic. The graph in the figure shows dependency relation between these variables and supports x^{11} to pay more attention to x^7 and x^9 instead of x^3. Next, we describe how to extract data flow from a source code.

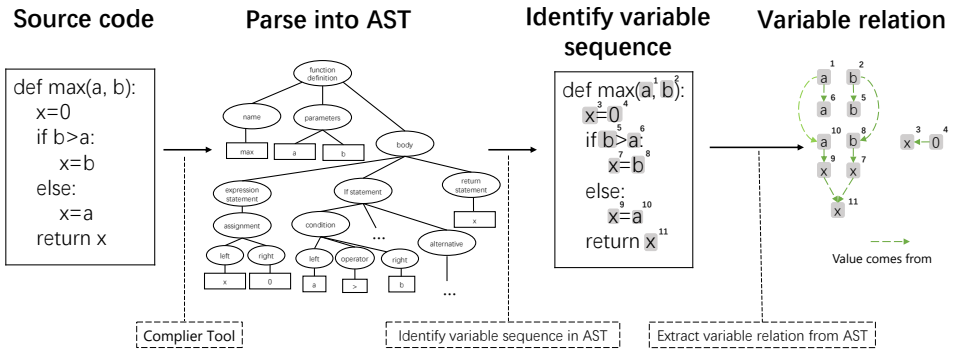

Figure 1: The procedure of extracting data flow given a source code. The graph in the rightmost is data flow that represents the relation of "where-the-value-comes-from" between variables.

Figure 1 shows the extraction of data flow through a source code. Given a source code $C = \{c_1, c_2, ..., c_n\}$, we first parse the code into an abstract syntax tree (AST) by a standard compiler tool[1]. The AST includes syntax information of the code and terminals (leaves) are used to identify the variable sequence, denoted as $V = \{v_1, v_2, ..., v_k\}$. We take each variable as a node of the graph and an direct edge $\varepsilon = \langle v_i, v_j \rangle$ from v_i to v_j refers that the value of j-th variable comes from i-th variable. Taking $x = expr$ as an example, edges from all variables in $expr$ to x are added into the graph. We denote the set of directed edges as $E = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_l\}$ and the graph $\mathcal{G}(C) = (V, E)$ is data flow used to represent dependency relation between variables of the source code C.

4 GRAPHCODEBERT

In this section, we describe GraphCodeBERT, a graph-based pre-trained model based on Transformer for programming language. We introduce model architecture, graph-guided masked attention and pre-training tasks including standard masked language model and newly introduced ones. More details about model pre-training setting are provided in the Appendix A.

4.1 MODEL ARCHITECTURE

Figure 2 shows the model architecture of GraphCodeBERT. We follow BERT (Devlin et al., 2018) and use the multi-layer bidirectional Transformer (Vaswani et al., 2017) as the model backbone. Instead of only using source code, we also utilize paired comments to pre-train the model to support more code-related tasks involving natural language such as natural language code search (Feng et al., 2020). We further take data flow, which is a graph, as a part of the input to the model.

Given a source code $C = \{c_1, c_2, ..., c_n\}$ with its comment $W = \{w_1, w_2, ..., w_m\}$, we can obtain the corresponding data flow $\mathcal{G}(C) = (V, E)$ as discussed in the Section 3, where $V = \{v_1, v_2, ..., v_k\}$ is a set of variables and $E = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_l\}$ is a set of direct edges that represent where the value of each variable comes from. We concatenate the comment, source code and the set of variables as the

[1] https://github.com/tree-sitter/tree-sitter

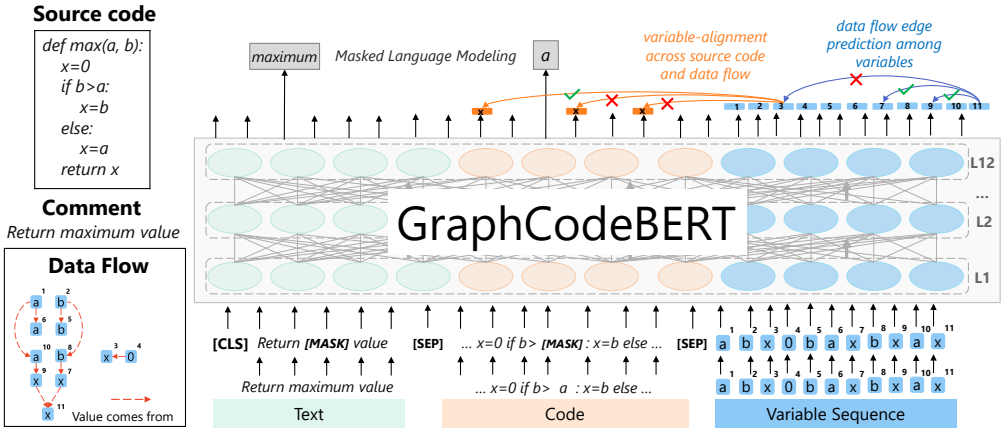

Figure 2: An illustration about GraphCodeBERT pre-training. The model takes source code paired with comment and the corresponding data flow as the input, and is pre-trained using standard masked language modeling (Devlin et al., 2018) and two structure-aware tasks. One structure-aware task is to predict where a variable is identified from (marked with orange lines) and the other is data flow edges prediction between variables (marked with blue lines).

sequence input $X = \{[CLS], W, [SEP], C, [SEP], V\}$, where $[CLS]$ is a special token in front of three segments and $[SEP]$ is a special symbol to split two kinds of data types.

GraphCodeBERT takes the sequence X as the input and then converts the sequence into input vectors H^0. For each token, its input vector is constructed by summing the corresponding token and position embeddings. We use a special position embedding for all variables to indicate that they are nodes of data flow. The model applies N transformer layers over the input vectors to produce contextual representations $H^n = transformer_n(H^{n-1}), n \in [1, N]$. Each transformer layer contains an architecturally identical transformer block that applies a multi-headed self-attention operation (Vaswani et al., 2017) followed by a feed forward layer over the input H^{n-1} in the n-th layer. For the n-th transformer layer, the output H^n of a multi-headed self-attention is computed via:

$$Q_i = H^{n-1}W_i^Q,\ K_i = H^{n-1}W_i^K,\ V_i = H^{n-1}W_i^V \tag{1}$$

$$head_i = \mathrm{softmax}(\frac{\mathrm{Q_i K_i^T}}{\sqrt{\mathrm{d_k}}} + \mathrm{M})\mathrm{V_i} \tag{2}$$

$$H^n = [head_1; ...; head_u]W_n^O \tag{3}$$

where the previous layer's output $H^{n-1} \in \mathbb{R}^{|X| \times d_h}$ is linearly projected to a triplet of queries, keys and values using model parameters W_i^Q,W_i^K,$W_i^V \in \mathbb{R}^{d_h \times d_k}$, respectively. u is the number of heads, d_k is the dimension of a head, and $W_n^O \in \mathbb{R}^{d_h \times d_h}$ is the model parameters. $M \in \mathbb{R}^{|X| \times |X|}$ is a mask matrix, where M_{ij} is 0 if i-th token is allowed to attend j-th token otherwise $-\infty$.

4.2 Graph-guided Masked Attention

To incorporate the graph structure into Transformer, we define a graph-guided masked attention function to filter out irrelevant signals. The attention masking function could avoid the key k_i attended by the query q_j by adding the attention score $q_j^T k_i$ an infinitely negative value so that the attention weight becomes zero after using a softmax function. To represent dependency relation between variables, a node-query q_{v_i} is allowed to attend to a node-key k_{v_j} if there is a direct edge from the node v_j to the node v_i (i.e. $\langle v_j, v_i \rangle \in E$) or they are the same node (i.e. $i = j$). Otherwise, the attention is masked by adding an infinitely negative value into the attention score. To represent the relation between source code tokens and nodes of the data flow, we first define a set $E^{'}$, where $\langle v_i, c_j \rangle / \langle c_j, v_i \rangle \in E^{'}$ if the variable v_i is identified from the source code token c_j. We then allow the node q_{v_i} and code k_{c_j} attend each other if and only if $\langle v_i, c_j \rangle / \langle c_j, v_i \rangle \in E^{'}$. More formally, we use the following graph-guided masked attention matrix as the mask matrix M in the equation 2:

$$M_{ij} = \begin{cases} 0 & \text{if } q_i \in \{[CLS], [SEP]\} \text{ or } q_i, k_j \in W \cup C \text{ or } \langle q_i, k_j \rangle \in E \cup E^{'} \\ -\infty & \text{otherwise} \end{cases} \tag{4}$$

4.3 PRE-TRAINING TASKS

We describe three pre-training tasks used for pre-training GraphCodeBERT in this section. The first task is masked language modeling (Devlin et al., 2018) for learning representation from the source code. The second task is data flow edge prediction for learning representation from data flow, where we first mask some variables' data flow edges and then let GraphCodeBERT predict those edges. The last task is variable-alignment across source code and data flow for aligning representation between source code and data flow, which predicts where a variable is identified from.

Masked Language Modeling We follow Devlin et al. (2018) to apply masked language modeling (MLM) pre-training task. Specially, we sample randomly 15% of the tokens from the source code and paired comment. We replace them with a [MASK] token 80% of the time, with a random token 10% of the time, and leave them unchanged 10% of the time. The MLM objective is to predict original tokens of these sampled tokens, which has proven effective in previous works (Devlin et al., 2018; Liu et al., 2019; Feng et al., 2020). In particular, the model can leverage the comment context if the source code context is not sufficient to infer the masked code token, encouraging the model to align the natural language and programming language representations.

Edge Prediction To learn representation from data flow, we introduce a pre-training task of data flow edges prediction. The motivation is to encourage the model to learn structure-aware representation that encodes the relation of "where-the-value-comes-from" for better code understanding. Specially, we randomly sample 20% of nodes V_s in data flow, mask direct edges connecting these sampled nodes by add an infinitely negative value in the mask matrix, and then predict these masked edges E_{mask}. Taking the variable x^{11} in Figure 2 for an example, we first mask edges $\langle x^7, x^{11}\rangle$ and $\langle x^9, x^{11}\rangle$ in the graph and then let the model to predict these edges. Formally, the pre-training objective of the task is calculated as Equation 5, where $E_c = V_s \times V \cup V \times V_s$ is a set of candidates for edge prediction, $\delta(e_{ij} \in E)$ is 1 if $\langle v_i, v_j\rangle \in E$ otherwise 0, and the probability $p_{e_{ij}}$ of existing an edge from i-th to j-th node is calculated by dot product following a sigmoid function using representations of two nodes from GraphCodeBERT.

$$loss_{EdgePred} = -\sum_{e_{ij} \in E_c} [\delta(e_{ij} \in E_{mask}) log p_{e_{ij}} + (1 - \delta(e_{ij} \in E_{mask})) log(1 - p_{e_{ij}})] \quad (5)$$

Node Alignment To align representation between source code and data flow, we introduce a pre-training task of node alignment across source code and data flow, which is similar to data flow edge prediction. Instead of predicting edges between nodes, we predict edges between code tokens and nodes. The motivation is to encourage the model to align variables and source code according to data flow. Taking the variable x^3 in Figure 2 for an example, we first mask the edge between x^3 and source code, and then predict which code token the variable x^3 is identified from. Specially, we randomly sample 20% nodes in the graph, mask edges between code tokens and sampled nodes, and then predict masked edges. The pre-training objective of this task is same as Equation 5.

5 EXPERIMENTS

We evaluate our model on four downstream tasks, including code search, clone detection, code translation and code refinement. Detailed experimental settings can be found in the Appendix.

5.1 NATURAL LANGUAGE CODE SEARCH

Given a natural language as input, the task aims to find the most semantically related code from a collection of candidate codes. We conduct experiments on the CodeSearchNet dataset (Husain et al., 2019), which includes six programming languages. We use Mean Reciprocal Rank (MRR) as our evaluation metric and report results of existing methods in the Table 1.

The first group calculates inner product of code and query encodings as relevance scores to rank candidate codes. Husain et al. (2019) implement four methods to obtain the encodings, including bag-of-words, convolutional neural network, bidirectional recurrent neural network, and multi-head attention. The second group is the results of pre-trained models. Roberta (Liu et al., 2019) is a pre-trained model on text corpus with MLM learning objective, while **RoBERTa (code)** is pre-trained

model	Ruby	Javascript	Go	Python	Java	Php	Overall
NBow	0.429	0.461	0.641	0.581	0.514	0.484	0.518
CNN	0.245	0.352	0.627	0.571	0.527	0.529	0.475
BiRNN	0.084	0.153	0.452	0.321	0.287	0.251	0.258
selfAtt	0.365	0.451	0.681	0.692	0.587	0.601	0.563
RoBERTa	0.625	0.606	0.820	0.809	0.666	0.658	0.697
RoBERTa (code)	0.661	0.640	0.819	0.844	0.721	0.671	0.726
CodeBERT	0.693	0.706	0.840	0.869	0.748	0.706	0.760
GraphCodeBERT	**0.732**	**0.711**	**0.841**	**0.879**	**0.757**	**0.725**	**0.774**

Table 1: Results on natural language code search.

only on code. **CodeBERT** (Feng et al., 2020) is pre-trained on code-text pairs with MLM and replaced token detection learning objectives. As we can see, **GraphCodeBERT** that leverages code structure for pre-training could bring a 1.4% gain of MRR, achieving the state-of-art performance.

5.2 Code Clone Detection

Code clones are multiple code fragments that output similar results when given the same input. The task aims to measure the similarity between two code fragments, which can help reduce the cost of software maintenance and prevent bugs. We conduct experiments on the BigCloneBench dataset (Svajlenko et al., 2014) and report results in the Table 2.

Deckard (Jiang et al., 2007) is to compute vectors for structural information within ASTs and then a Locality Sensitive Hashing (LSH) (Datar et al., 2004) is used to cluster similar vectors for detection. **RtvNN** (White et al., 2016) trains a recursive autoencoder to learn representations for AST. **CDLH** (Wei & Li, 2017) learn representations of code fragments via AST-based LSTM and hamming distance is used to optimize the distance between the vector representation of AST pairs. **FA-AST-GMN** (Wang et al., 2020) uses GNNs over a flow-augmented AST to leverages explicit control and data flow information for code clone detection. Results show that our **GraphCodeBERT** that leverages code structure information outperforms other methods, which demonstrates the effectiveness of our pre-trained model for the task of code clone detection.

Model	Precision	Recall	F1
Deckard	0.93	0.02	0.03
RtvNN	0.95	0.01	0.01
CDLH	0.92	0.74	0.82
ASTNN	0.92	0.94	0.93
FA-AST-GMN	0.96	0.94	0.95
RoBERTa (code)	0.960	0.955	0.957
CodeBERT	0.967	0.963	0.965
GraphCodeBERT	**0.973**	**0.968**	**0.971**

Table 2: Results on code clone detection.

5.3 Code Translation

Code translation aims to migrate legacy software from one programming language in a platform to another. Following Nguyen et al. (2015) and Chen et al. (2018), we conduct experiments on a dataset crawled from the same several open-source projects as them and report results in the Table 3.

The **Naive** method is directly copying the source code as the translation result. **PBSMT** is short for phrase-based statistical machine translation (Koehn et al., 2003), and has been exploited in previous works (Nguyen et al., 2013; Karaivanov et al., 2014). As for the **Transformer**, we use the same number of layers and hidden size as pre-trained models. To leverage the pre-trained models for translation, we initialize the encoder with pre-trained models and randomly initialize parameters of the decoder and the source-to-target attention. Results show that the models initialized with pre-trained models (i.e the second group) significantly outperform PBSMT and Transformer models. Among them, **GraphCodeBERT** achieves state-of-art performance, which demonstrates the effectiveness of our model for code translation.

Method	Java→C#		C#→Java	
	BLEU	Acc	BLEU	Acc
Naive	18.54	0.000	18.69	0.000
PBSMT	43.53	0.125	40.06	0.161
Transformer	55.84	0.330	50.47	0.379
RoBERTa (code)	77.46	0.561	71.99	0.579
CodeBERT	79.91	0.590	72.13	0.580
GraphCodeBERT	**80.58**	**0.594**	**72.64**	**0.588**

Table 3: Results on code translation.

5.4 CODE REFINEMENT

Code refinement aims to automatically fix bugs in the code, which can contribute to reducing the cost of bug-fixes. We use the dataset released by Tufano et al. (2019) and report results in the Table 4.

The **Naive** method directly copies the buggy code as the refinement result. For the **Transformer**, we use the same number of layers and hidden size as the pre-trained models. Same as the Section 5.3, we initialize the encoder with pre-trained models and randomly initialize parameters of the decoder and the source-to-target attention. Then we use the training data to fine-tune the whole model. In the table, we see that the **Transformer** significantly outperforms **LSTM**. Results in the second group shows that pre-trained models outperform Transformer models further, and **GraphCodeBERT** achieves better performance than other pre-trained models on both datasets, which shows leveraging code structure information are helpful to the task of code refinement.

Method	small		medium	
	BLEU	Acc	BLEU	Acc
Naive	78.06	0.000	90.91	0.000
LSTM	76.76	0.100	72.08	0.025
Transformer	77.21	0.147	89.25	0.037
RoBERTa (code)	77.30	0.159	90.07	0.041
CodeBERT	77.42	0.164	90.07	0.052
GraphCodeBERT	**80.02**	**0.173**	**91.31**	**0.091**

Table 4: Results on code refinement.

5.5 MODEL ANALYSIS

Ablation Study We conduct ablation study on the task of natural language code search to understand various components in our approach impact overall performance. We remove two pre-training tasks and data flow, respectively, to analyze their contribution. Table 5 shows that the overall performance drops from 77.4% to 76.6%$\sim$76.8% when removing Node Alignment and Edge Prediction pre-training tasks, respectively, which reveals the importance of two structure-aware pre-training tasks. After ablating the data flow totally, we can see that the performance drops from 77.4% to 76.0%, which means leveraging data flow to learn code representation could improve GraphCodeBERT.

Methods	Ruby	Javascript	Go	Python	Java	Php	Overall
GraphCodeBERT	**0.732**	**0.711**	**0.841**	**0.879**	**0.757**	**0.725**	**0.774**
-w/o NodeAlign	0.713	0.703	0.839	0.873	0.753	0.718	0.766
-w/o EdgePred	0.715	0.710	0.839	0.874	0.752	0.719	0.768
-w/o Data Flow	0.693	0.706	0.840	0.869	0.748	0.706	0.760

Table 5: Ablation study on natural language code search

Node-vs. Token-level Attention Table 6 shows how frequently a special token $[CLS]$ that is used to calculate probability of correct candidate attends to code tokens (Codes) and variables (Nodes). We see that although the number of nodes account for 5%$\sim$20%, attentions over nodes overwhelm node/code ratio (around 10% to 32%) across all programming languages. The results indicate that data flow plays an important role in code understanding process and the model pays more attention to nodes in data flow than code tokens.

	Ruby	Javascript	Go	Python	Java	Php
Codes/Nodes	90.1/9.9	94.6/5.4	95.0/5.03	80.6/19.4	93.2/6.8	87.5/12.5
$[CLS]$ $\rightarrow$Codes/Nodes	82.3/17.7	89.7/10.3	91.0/9.0	67.7/32.3	87.8/12.2	79.4/20.6

Table 6: Attention distribution (%) between code tokens (codes) and variables (nodes) across different programming language on natural language code search test sets. The first row is the ratio of the number of code tokens to nodes, and the second row is attention distribution of $[CLS]$ token.

Comparison between AST and Data Flow Figure 3 shows MRR score with respect to input sequence length on the validation dataset of Ruby programming language for the task of code search. **AST Pre-order Traversal** regards AST as a sequence by linearizing all AST nodes using pre-order traversal algorithm. **AST Subtree Masking** regards AST as a tree and introduce subtree masking (Nguyen et al., 2019) for self-attention of the Transformer. In subtree masking, each node-query in AST attends only to its own subtree descendants, and each leaf-query only attends to leaves of AST.

Transformer has a self-attention component with $O(n^2)$ time and memory complexity where n is the input sequence length, and thus is not efficient to scale to long inputs. We observe that injecting AST even hurts the performance when the sequence length is short (e.g. shorter than 128), while GraphCodeBERT consistently brings performance boost on varying sequence length and obtains better MRR score than AST-based methods. The main reason is that data flow is neat and the number of nodes account for $5\% \sim 20\%$ (see Table 6), which does not bring an unnecessarily deep hierarchy of AST and makes the model more accurate and efficient.

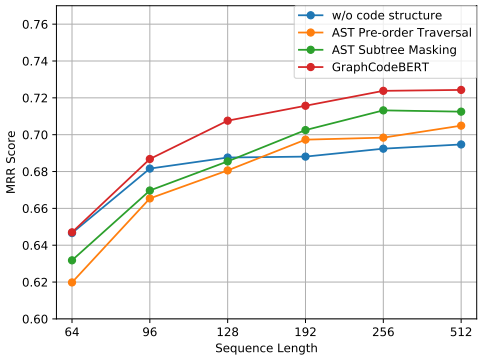

Figure 3: MRR score on the validation dataset of Ruby for code search with varying length of input sequence.

Case Study We also give a case study to demonstrate that data flow would enhance the code understanding process. Given a source code and a comment, we use GraphCodeBERT with and without data flow to predict whether the comment correctly describes the source code. Results are given in Figure 4. We can see that both models make correct prediction in the original example, where the threshold is 0.5 (left panel). To study the code understanding ability of models, we change the source code (center panel) and the comment (right panel), respectively. Although we make a small change on the source code ($return\ a \rightarrow return\ b$) and the comment ($sum\ value \rightarrow mean\ value$), the semantic of the source code and the comment are completely different and corresponding gold labels change from 1 to 0. As we can see in the figure, GraphCodeBERT without using data flow fails these tests and still outputs high probability for negative examples. After leveraging data flow, GraphCodeBERT better understands the semantic of source code and makes correct predictions on all tests, which demonstrates that data flow could improve the code understanding ability of the model.

	Unchanged	Code: $return\ a \rightarrow return\ b$	NL: $sum\ value \rightarrow mean\ value$
Input	**NL:** Return sum value of an array **Code:** import numpy as np def f(array): a=np.sum(array) b=np.mean(array) return a	**NL:** Return sum value of an array **Code:** import numpy as np def f(array): a=np.sum(array) b=np.mean(array) return **b**	**NL:** Return **mean** value of an array **Code:** import numpy as np def f(array): a=np.sum(array) b=np.mean(array) return a
Label	1	0	0
Prediction	**GraphCodeBERT:** 0.6563 (1) **GraphCodeBERT:** 0.8728 (1) **(w/o Data Flow)**	**GraphCodeBERT:** 0.4615 (0) **GraphCodeBERT:** 0.8608 (1) **(w/o Data Flow)**	**GraphCodeBERT:** 0.2884 (0) **GraphCodeBERT:** 0.9048 (1) **(w/o Data Flow)**

Figure 4: We take a comment and a source code as the input (first row), and use GraphCodeBERT with and without data flow to predict the probability of the source code matching the comment (third row). The label is 1 if the comment correctly describes the source code otherwise 0 (second row).

6 Conclusion

In this paper, we present GraphCodeBERT that leverages data flow to learn code representation. To the best of our knowledge, this is the first pre-trained model that considers code structure for pre-training code representations. We introduce two structure-aware pre-training tasks and show that GraphCodeBERT achieves state-of-the-art performance on four code-related downstream tasks, including code search, clone detection, code translation and code refinement. Further analysis shows that code structure and newly introduced pre-training tasks boost the performance. Additionally, case study in the task of code search shows that applying data flow in the pre-trained model improves code understanding.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. *arXiv*, pp. arXiv–1910, 2019.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness throughneural language models. *arXiv preprint arXiv:2006.12641*, 2020.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pp. 2547–2557, 2018.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pp. 253–262, 2004.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2019.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010. IEEE, 2018.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105. IEEE, 2007.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. *arXiv preprint arXiv:2001.00059*, 2019.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 173–184, 2014.

Rafael-Michael Karampatsis and Charles Sutton. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*, 2020.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*, 2020.

Philipp Koehn, Franz J Och, and Daniel Marcu. Statistical phrase-based translation. Technical report, UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST, 2003.

Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

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

Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pp. 858–868. IEEE, 2015.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 651–654, 2013.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 585–596. IEEE, 2015.

Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*, 2019.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480. IEEE, 2014.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*, 2020.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. *arXiv preprint arXiv:2002.08653*, 2020.

Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pp. 3034–3040, 2017.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 87–98. IEEE, 2016.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, Vancouver, Canada, July 2017. URL https://arxiv.org/abs/1704.01696.

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

A PRE-TRAINING DETAILS

GraphCodeBERT includes 12 layers Transformer with 768 dimensional hidden states and 12 attention heads. For fair comparison, we use the same dataset as CodeBERT (Feng et al., 2020) to pretrain our model. The dataset is the CodeSearchNet dataset[2] (Husain et al., 2019), which includes 2.4M functions with document pairs for six programming languages. We train the model on two DGX-2 machines, each having 16 NVIDIA Tesla V100 with 32GB memory. We set the max length of sequences and nodes as 512 and 128, respectively. We use the Adam optimizer to update model parameters with 1,024 batch size and 2e-4 learning rate. To accelerate the training process, we adopt the parameters of CodeBERT released by Feng et al. (2020) to initialize the model. The model is trained with 200K batches and costs about 83 hours.

At each iteration, we alternate EdgePred and NodeAlign objectives in combination with MLM to pre-train the model. And we follow Lample & Conneau (2019) to sample each batch from the same programming language according to a multinomial distribution with probabilities $\{q_i\}_{i=1...N}$, where n_i is number of examples for i-th programming language and α=0.7. Sampling with this distribution could alleviates the bias towards high-resource languages.

$$q_i = \frac{p_i^\alpha}{\sum_N^{j=1} p_j^\alpha} \; with \;\; p_i = \frac{n_i}{\sum_N^{k=1} n_k} \tag{6}$$

B NATURAL LANGUAGE CODE SEARCH

Given a natural language as input, code search aims to find the most semantically related code from a collection of candidate codes. We conduct experiments on the CodeSearchNet dataset (Husain et al., 2019), which includes six programming languages. We list data statistics about the CodeSearchNet corpus in Table 7. For each query in the test data, the answer is retrieved from a set of 1,000 candidate codes. We treat the task as a binary classification to fine-tune language-specific models for each programming language, where we use query, source code and data flow as the input. The probability of correct candidate is calculated by adding a linear layer followed by a sigmoid function on the representation of the special token $[CLS]$. In the fine-turning step, we set the learning rate as 5e-5, the batch size as 128, the max sequence length as 256 and the max number of nodes as 64. We use the Adam optimizer to update model parameters and perform early stopping on the development set.

Code Search	Training	Dev	Testing
Go	635,635	28,483	14,000
Java	908,886	30,655	26,000
JavaScript	247,773	16,505	6,000
PHP	1,047,406	52,029	28,000
Python	824,342	46,213	22,000
Ruby	97,580	4,417	2,000

Table 7: Data statistics about the CodeSearchNet corpus.

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

We give two cases of the GraphCodeBERT output for this task in Figure 5. In the first example, the model successfully finds Python source code that correctly matches the sementic of the query "Scans through a string for substrings matched some patterns". The source code finds all substrings by calling $re.findall()$ build-in fucntion. In the second case, the query is "Combing the individual byte arrays into one array", and the model searches a source code from Java candidate codes. As we can see, the source code concatenates multiple arrays into one array by calling $System.arraycopy()$ build-in fucntion.

Input: A query

Scans through a string for substrings matched some patterns.

Output: Searched Python source code (score: 0.940, rank: top1)

```
def matchall(text, patterns):
    ret = []
    for pattern in patterns:
        match = re.findall(pattern, text)
        ret += match
    return ret
```

Input: A query

Combine the individual byte arrays into one array.

Output: Searched Java source code (score: 0.979, rank: top1)

```
public static byte[] concatenate(byte[]... arrays) {
    int length = 0;
    for (byte[] array : arrays) {
        length += array.length;
    }
    byte[] newArray = new byte[length];
    int destPos = 0;
    for (byte[] array : arrays) {
        System.arraycopy(array, 0, newArray, destPos, array.length);
        destPos += array.length;
    }
    return newArray;
}
```

Figure 5: Two cases of the GraphCodeBERT output for the natural language code search task.

C CODE CLONE DETECTION

Code clone detection aims to measure the similarity between two code fragments. We use BigCloneBench dataset (Svajlenko et al., 2014), which contains over 6,000,000 true clone pairs and 260,000 false clone pairs from 10 different functionalities. We follow the settings in Wei & Li (2017), discarding code fragments without any tagged true and false clone pairs and using 9,134 remaining code fragments. Finally, the dataset provided by Wang et al. (2020) includes 901,724/416,328/416,328 examples for training/validation/testing. We treat the task as a binary classification to fine-tune GraphCodeBERT, where we use source code and data flow as the input. The probability of true clone is calculated by dot product from the representation of $[CLS]$. In the fine-turning step, we set the learning rate as 5e-5, the batch size as 128, the max sequence length as 256 the max number of nodes as 64. We use the Adam optimizer to update model parameters and tune hyper-parameters and perform early stopping on the development set.

We give a case of the GraphCodeBERT output for this task in Figure 6. In this example, two Java source codes both download content from a given URL and convert the type of the content into string type. Therefore, two codes are semantically similar since they output similar results when given the same input. As we can see, our model gives a high score for this case and the pair is classified as true clone pair.

D CODE TRANSLATION

Code translation aims to migrate legacy software from one programming language in a platform to another. We conduct experiments on a dataset crawled from the same several open-source projects as Nguyen et al. (2015) and Chen et al. (2018), i.e. Lucene[3], POI[4], JGit[5] and Antlr[6]. We do not use Itext[7] and JTS[8] as they do because of the license problem. Those projects have both Java and C#

[3] http://lucene.apache.org/
[4] http://poi.apache.org/
[5] https://github.com/eclipse/jgit/
[6] https://github.com/antlr/
[7] http://sourceforge.net/projects/itext/
[8] http://sourceforge.net/projects/jts-topo-suite/

Input: Two source codes

```
protected String downloadURLtoString(URL url) throws IOException
{
    BufferedReader in = new BufferedReader(new
                        InputStreamReader(url.openStream()));
    StringBuffer sb = new StringBuffer(100 * 1024);
    String str;
    while ((str = in.readLine()) != null) {
        sb.append(str);
    }
    in.close();
    return sb.toString();
}
```

Output: Semantically similar (score: 0.983)

```
public static String fetchUrl(String urlString)
{
    try {
        URL url = new URL(urlString);
        BufferedReader reader = new BufferedReader(new
                                InputStreamReader(url.openStream()));
        String line = null;
        StringBuilder builder = new StringBuilder();
        while ((line = reader.readLine()) != null) {
            builder.append(line);
        }
        reader.close();
        return builder.toString();
    } catch (MalformedURLException e) {
    } catch (IOException e) {
    }
    return "";
}
```

Figure 6: A case of the GraphCodeBERT output for the code clone detection task.

implementation. We pair the methods in the two languages based on their file names and method names. After removing duplication and methods with null function body, the total number of method pairs is 11,800, and we split 500 pairs from them as the development set and another 1,000 pairs for test. To demonstrate the effectiveness of GraphCodeBERT on the task of code translation, we adopt various pre-trained models as encoders and stay hyperparameters consistent. We set the learning rate as 1e-4, the batch size as 64, the max sequence length as 256 and the max number of nodes as 64. We use the Adam optimizer to update model parameters and tune hyper-parameters and perform early stopping on the development set.

We give a case of the GraphCodeBERT output for this task in Figure 7. In this example, the model successfully translates a piece of Java code into its C# version. The differences include the type name (from "boolean" to "bool") and the usage of getting a string value of a bool variable (from "String.valueOf(b)" to "b.ToString()").

Input: A Java method

```
public void print(boolean b)
{
    print(String.valueOf(b));
}
```

→

Output: Its C# version

```
public void print(bool b)
{
    print(b.ToString());
}
```

Figure 7: A case of the GraphCodeBERT output for the code translation task.

E Code Refinement

Code refinement aims to automatically fix bugs in the code. We use the dataset released by Tufano et al. (2019). The source is buggy Java functions while the target is the according fixed ones. Almost all the names of variables and custom methods are normalized. The dataset contains two subsets based on the code length. For the *small* dataset, the numbers of training, development and test samples are 46,680, 5,835 and 5,835. For the *medium* dataset, the numbers are 52,364, 6,545 and 6,545. We also use the sequence-to-sequence Transformer model to conduct the experiments. In the fine-tuning step, we adopt various pre-trained models as encoders. We set the learning rate as 1e-4, the batch size as 64, the max sequence length as 256 and the max number of nodes as 64. We use the Adam optimizer to update model parameters and perform early stopping on the development set.

We give two cases of the GraphCodeBERT output for this task in Figure 8. In the first example, the model successfully fixes the operation bug (from "*" to "+") to match the function name "add". In the second case, the source function and type names are normalized. The return type of this function

is “void” but the buggy code gives a return value. Our model successfully removes the “return” word so that the return type of the function matches its declaration.

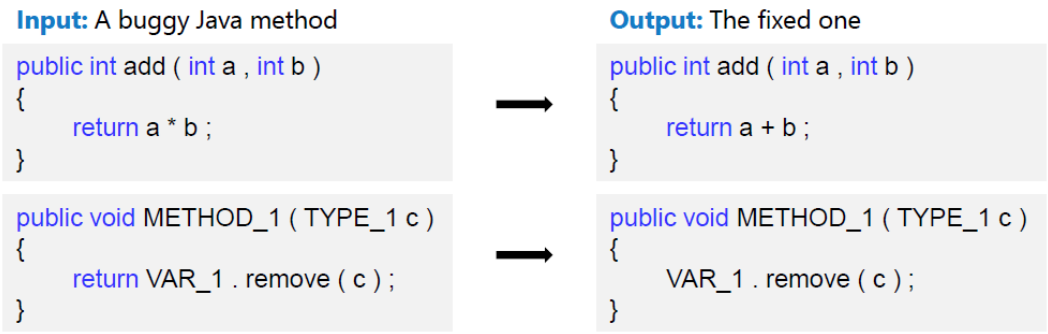

Figure 8: Two cases of the GraphCodeBERT output for the code refinement task.