CODE-MVP: Learning to Represent Source Code from Multiple Views with Contrastive Pre-Training

Xin Wang1 Yasheng Wang2 Yao Wan3 Jiawei Wang4
Pingyi Zhou2 Li Li4 Hao Wu5 Jin Liu5

1School of Computer Science, Wuhan University, China 2Huawei Noah’s Ark Lab
3Faculty of Information Technology, Monash University, Australia
4School of Computer Science, Wuhan University, China
5School of Information Science and Engineering, Yunnan University, China

{xinwang0920, jinliu}@whu.edu.cn

Abstract

Recent years have witnessed increasing interest in code representation learning, which aims to represent the semantics of source code into distributed vectors. Currently, various works have been proposed to represent the complex semantics of source code from different views, including plain text, Abstract Syntax Tree (AST), and several kinds of code graphs (e.g., Control/Data Flow Graph). However, most of them only consider a single view of source code independently, ignoring the correspondences among different views. In this paper, we propose to integrate different views with the natural-language description of source code into a unified framework with Multi-View contrastive Pre-training, and name our model as CODE-MVP. Specifically, we first extract multiple code views using compiler tools, and learn the complementary information among them under a contrastive learning framework. Inspired by the type checking in compilation, we also design a fine-grained type inference objective in the pre-training. Experiments on three downstream tasks over five datasets demonstrate the superiority of CODE-MVP when compared with several state-of-the-art baselines. For example, we achieve 2.4/2.3/1.1 gain in terms of MRR/MAP/Accuracy metrics on natural language code retrieval, code similarity, and code defect detection tasks, respectively.

1 Introduction

Code intelligence that utilizes machine learning techniques to promote the productivity of software developers, has attracted increasing interest in both communities of software engineering and artificial intelligence (Lu et al., 2021; Feng et al., 2020; Wang et al., 2022; Wan et al., 2022a; Wu et al., 2021). To achieve code intelligence, one fundamental task is code representation learning (also known as code embedding), which aims to preserve the semantics of source code in distributed vectors (Alon et al., 2019). It can support various downstream tasks about code intelligence, including code defect detection (Omri and Sinz, 2020; Zhao et al., 2021b,a), code summarization (Wan et al., 2018), code retrieval (Wan et al., 2019), and code clone detection (White et al., 2016).

Current approaches to code representation borrow ideas from the successful deep learning methods in natural language processing, mainly attributed to the naturalness hypothesis in source code (Allamanis et al., 2018). From our investigation, existing approaches mainly represent the source code from different views of code, including code token in plain text (Iyer et al., 2016), Abstract Syntax Tree (AST) (Bui et al., 2021a), and Control/Data Flow Graphs (CFGs/DFGs) of code (Cummins et al., 2020; Wang and Su, 2020). Recently, many attempts have been made to pretrain a masked language model for source code, such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), SynCoBERT (Wang et al., 2021), CodeGPT (Lu et al., 2021), PLBART (Ahmad et al., 2021), ContraCode (Phan et al., 2021), CoTextT (Phan et al., 2021), and CodeT5 (Wang et al., 2021b). Table 1 shows the contribution of our work when compared with current pre-trained language models for source code.

Table 1: Comparison with current pre-trained code models. PT: Program Transformation.
Despite much progress in code representation learning, most of them only consider a single view of source code independently, ignoring the consistency among different views (Feng et al., 2020; Lu et al., 2021; Ahmad et al., 2021; Wang et al., 2021b). Usually, a program, accompanied by a corresponding natural-language comment (NL), can be parsed into multiple views, e.g., the source code tokens, AST, and CFG. We argue that these different views contain complementary semantics of the program. For example, the source code tokens (e.g., method name identifiers) and natural-language comments always reveal the lexical semantics of code, while the intermediate structures of code (e.g., AST and CFG) always reveal the syntactic and executive information of code. In addition, a program can also be transformed (or rewritten) into different variants that have equivalent functionality. We think that different variants of the same program reveal the functional information of code. That is, those different program variants with the same functionality are expected to represent the same semantics.

Inspired by the aforementioned insights, this paper proposes a novel CODE-MVP for code representation, which aims to integrate multiple views of the code into a unified framework with multi-view contrastive pre-training. Concretely, we first extract multiple views of code using several compiler tools, and learn the complementary information among them under a multi-view contrastive learning framework. Meanwhile, inspired by the type checking in compilation process, we also introduce fine-grained type inference as an auxiliary task in the pre-training process to encourage the model to learn more fine-grained type information.

To summarize, the contributions of this paper are two-fold: (1) We are the first to represent source code from multiple views, including the code tokens, AST, CFG, and various program equivalents, under a unified multi-view contrastive pre-training framework. Meanwhile, we also introduce an auxiliary task of inferring type annotations for variables. (2) We extensively evaluate CODE-MVP on three program comprehension tasks. Experimental results demonstrate the superiority of CODE-MVP when compared with several state-of-the-art baselines. Specifically, CODE-MVP achieves 2.4/2.3/1.1 gain on MRR/MAP/Accuracy metrics in natural language code retrieval, code similarity, and code defect detection tasks, respectively.

2 Multiple Views of Code

We borrow ideas from the way that computers process the source code in compilation, where a program would be converted into multiple views. Figure 1 shows the process of converting a program from source code to machine code. During this process, the compiler would automatically utilize some program analysis techniques to verify the correctness of source code, including lexical, syntax, and semantic analyses. In the lexical analysis, a program is treated as a sequence of tokens and checked for spelling problems. In the syntax analysis, syntactic rules of programs are defined by the context-free grammar (Javed et al., 2004). Then the program could be parsed as an AST, based on which many program transformation heuristics can be applied to rewrite the program while maintaining the same desired functionality. In the semantic analysis, semantic rules of the program are defined by the attribute grammar (Paakki, 1995). Then the compiler could check the types of code tokens, and a decorated AST could be obtained. After the three stages above, a translator will convert the source code to its Intermediate Representation (IR), which is then considered as the basis for building Control/Data Flow Graphs (CFGs/DFGs) for further optimizations in the static analysis. Finally, the IR of the source code should be converted into machine code to execute through a code generator. Next, we introduce how we extract different views of the source code. Figure 2 illustrates multiple views of source code with an example.

Abstract Syntax Tree (AST). An AST, which is composed of leaf nodes, non-leaf nodes and edges between them, contains rich syntactic structural information of source code. In the AST, an assignment statement \( y = 0 \) can be represented by a non-leaf node assignment that points to three
leaf nodes (0, y, and =). In this paper, we parse a snippet of source code into an AST using a standard compiler tool tree-sitter.\(^1\) To feed an AST into our model, we apply depth-first traversal to convert it into a sequence of AST tokens (Kim et al., 2021).

**Control Flow Graph (CFG).** CFG, which represents the execution semantics of the program in the form of a graph, is one intermediate representation of programs. A CFG consists of basic blocks and directed edges between them, where each directed edge reflects the execution order of the two basic blocks in the program. We can easily traverse the CFG along directed edges to parse it into a token sequence, which reveals the execution semantics of the program. In this paper, we use a static analyzer Scalpel\(^2\) (Li et al., 2022) to construct the CFGs for Python code snippets.

**Program Transformation (PT).** The program transformation operations aim to produce multiple variants for a given program that satisfy the same desired functionality (Rabin et al., 2020). These different variants of a program can help the model capture functional semantics. In this work, we employ the following program transformation heuristics on ASTs and rewrite one program into another equivalent variant.

- **Function and Variable Renaming.** We randomly take new names from a set of candidates, such as VAR\(_1\), FUNC\(_1\), to rename the names of variables and functions in a program. This heuristic will not change the AST structure of the program, except for the textual appearance of variable and function names in the AST.

- **Loop Exchange.** The for and while loops represent the same functionality in a program. We traverse the AST to identify the for and while loop nodes, and replace for loops with while loops or vice versa. We also modify the initialization, condition and afterthought simultaneously.

- **Dead Code Insertion.** We first traverse the AST to identify several basic blocks (Mendis et al., 2019), and then randomly select a basic block and insert dead code snippets into it. Note that the dead code snippets are predefined and selected from a set of candidates.

### 3 Code-MVP

#### 3.1 Tasks and Notations

We define the set of program samples in multiple views (i.e. NL, PL, AST, CFG, PT) as \(S = \{S^1, \ldots, S^m\}\), where \(m\) represents the number of views, \(S^a \in S^n\) represents a program in the view of \(a\). Given a program, the PL view denotes its textual appearance, the NL view denotes its corresponding natural-language comment, and the PT denotes the variants of this program based on program transformation. The AST and CFG are extracted from a program using several compiler tools. Code-MVP adopts two forms of input, i.e., single-view input \(x^a = \{<CLS>, s^a\}\) and dual-view input \(x^{ab} = \{<CLS>, s^a, <SEP>, s^b\}\), where \(a\) and \(b\) denote two different views of the program. Following (Devlin et al., 2019), a special token <CLS> is appended at the beginning of each input sequence, and <SEP> is used to concatenate two sequences. Subsequently, the representation of <CLS> is used to represent the entire sequence, and <SEP> is used to split two views of sub-sequences. Given a set of programs with their corresponding multiple
views, we aim to learn the code representation by utilizing the mutual information existing in different views. Our intuition is to learn complementary information from multiple views of code by pulling the code under different views together and pushing the dissimilar ones apart.

### 3.2 Framework Overview

Figure 3 shows a simple example of our multi-view contrastive pre-training framework. Given a program $s_i$, we use the same program to construct a pair of positive samples $(x^a_i, x^b_i) = \{<CLS>, s^a_i\} \text{ vs } \{<CLS>, s^b_i\}$ in the form of views $a$ and $b$, as described above. We take $x^a_i$ and $x^b_i$ as the input of CODE-MVP respectively. The last hidden representations of $<CLS>$ tokens in the two inputs can be formulated as $h^a_i = \text{CODE-MVP}(x^a_i)$ and $h^b_i = \text{CODE-MVP}(x^b_i)$. We utilize a projection head (a two-layer MLP) to map hidden representations to a space, i.e., $v^a_i = f(h^a_i), v^b_i = f(h^b_i)$. Then the multi-view contrastive objective can be performed. During the pre-training process, we also design other two pre-training tasks, i.e., fine-grained type inference (FGTI) task and multi-view masked language modeling (MMLM).

### 3.3 Multi-View Contrastive Learning

We train CODE-MVP with paired data and unpaired data. Paired data refers to those program samples with paired NL, while unpaired data stands for those isolated program samples without paired NL. Next, we explain how we construct positive and negative samples for these two cases.

**Multi-View Positive Sampling.** We design Single-View (for paired and unpaired data) and Dual-View (for paired data only, which needs the NL) methods to construct multi-view positive samples for the MVCL objective:

- **Single-View.** To bridge the gap between different views of a same program, we consider the view of a program $x^a_i$ as a positive sample w.r.t another view $x^b_i$. That is, $(x^a_i, x^b_i)$ forms an inter-view positive pair, since $x^a_i$ and $x^b_i$ are two different views of a same program $x_i$.

- **Dual-View.** There are a total of $C^2_n$ combinations for two views of a same program. For efficiency, we focus on the features of the program itself, and propose the NL-conditional dual-view contrastive pre-training strategy, freezing the position of NL. Concretely, we construct a NL-conditional inter-view positive pair by replacing the second view in the input $\{<CLS>, s^a_i, <SEP>, s^b_i\}$ to be $\{<CLS>, s^a_i, <SEP>, s^a_j\}$, where $\forall a, b \neq \text{NL}$.

It is worth mentioning that there are many combinations to construct positive pairs. Some combinations are not considered in this work, such as the AST vs PT of the same program, and the CFG vs PT of the same program. Simultaneously, for training efficiency and downstream applications, we comprehensively consider eight combinations. They are (1) single-view: (NL vs PL), (NL vs PT), (PL vs AST), (PL vs CFG), and (PL vs PT); and (2) dual-view: (NL-PL vs NL-AST), (NL-PL vs NL-CFG), and (NL-PL vs NL-PT).

**Multi-View Negative Sampling.** Since the processes of unpaired data and paired data are similar, here we take the unpaired data as an example. We leverage in mini-batch and cross mini-batch sampling strategies (Chen et al., 2020) to construct intra-view and inter-view negative samples, respectively. Given a mini-batch of training data $b_1 = [x^a_1, \ldots, x^a_n]$ in the view of $a$ with size $n$, we can easily get another positive mini-batch data $b_2 = [x^b_1, \ldots, x^b_n]$ in the view of $b$, where $(x^a_i, x^b_i)$ denotes an inter-view positive pair. For $x^a_i$, the intra-view negative samples are $\{x^a_i\}, \forall i \neq j$, and the inter-view negative samples are $\{x^b_i\}, \forall i \neq j$. Finally, for each $x_i$, we can get a set of $2n - 2$ negative samples.
We calculate the loss for the same pair twice with order switched, i.e., \((x^a_i, x^b_i)\) is changed to \((x^b_i, x^a_i)\) as the dot product of their representations. Then the loss function for a positive pair \((x^a_i, x^b_i)\) can be defined as:

\[
l(x^a_i, x^b_i) = -\log \frac{\exp(v^a_i \cdot v^b_i)}{\exp(v^a_i \cdot v^b_i) + \sum_{k=2}^{2n-2} \exp(v^a_i \cdot v^b_k)}.
\]

We calculate the loss for the same pair twice with order switched, i.e., \((x^a_i, x^b_i)\) is changed to \((x^b_i, x^a_i)\) as the dot product with negative samples for \(x^a_i\) and \(x^b_i\) are different. Overall, the MVCL loss function is defined as follows:

\[
\mathcal{L}_{MVCL} = -\frac{1}{|\mathcal{N}|} \sum_{i} \left[ l(x^a_i, x^b_i) + l(x^b_i, x^a_i) \right],
\]

where \(\mathcal{N}\) denotes the set of all program samples covering all different views.

### 3.4 Pre-Training with Type Inference

Figure 4 shows the other two pre-training tasks, including fine-grained type inference and multi-view masked language modeling.

#### Fine-Grained Type Inference

Several previous works (Wang et al., 2021; Wang et al., 2021b) have proven the importance of symbolic properties in programming languages. Two concurrent works, SynCoBERT (Wang et al., 2021) and CodeT5 (Wang et al., 2021b) let the model divide the code token types into identifier or non-identifier. Inspired by the type checking in compilation process, we propose a fine-grained type inference (FGTI) objective to capture the fine-grained type information of variables (Li et al., 2022; An et al., 2011). First, we parse all source codes into ASTs. Then, we traverse the AST and use the type checker to obtain fine-grained identifier types. We employ BPE tokenizer (Sennrich et al., 2016) to tokenize tokens and let sub-tokens inherit the type information of the token. Finally, we define the loss function as follows:

\[
\mathcal{L}_{FGTI} = -\frac{1}{|Z|} \sum_{i} \sum_{j} Y_{ij} \log P_{ij},
\]

where \(Z\) denotes the set of all tokens that need to infer type, \(T\) represents the set of all types contained in the pre-training corpus, \(Y_{ij}\) denotes the label of token \(i\) in type \(j\), and \(P_{ij}\) denotes the predicted probability of token \(i\) in type \(j\).

#### Multi-View Masked Language Modeling

In addition to the multi-view contrastive learning objective and fine-grained type inference objective, we also extend the Masked Language Modeling (MLM) to the multi-view program corpus, named MMLM. Given a data point \(x\), we randomly select 15% of tokens in \(x\) and replace them with a special token \(<\text{MASK}>\), following the same settings in (Devlin et al., 2019). The MMLM objective aims to predict original tokens which are masked out. We calculate the MMLM loss as follows:

\[
\mathcal{L}_{MMLM} = -\frac{1}{|\mathcal{M}|} \sum_{i} \sum_{j} Y_{ij} \log P_{ij},
\]

where \(\mathcal{M}\) denotes the set of masked tokens, \(V\) represents the vocabulary, \(Y_{ij}\) denotes the label of the masked token \(i\) in class \(j\), and \(P_{ij}\) denotes the predicted probability of token \(i\) in class \(j\).

### 3.5 Overall Training Objective

The overall loss function in CODE-MVP is the integration of several components we have defined before.

\[
\mathcal{L} = \mathcal{L}_{MVCL} + \mathcal{L}_{FGTI} + \mathcal{L}_{MMLM} + \lambda \|\Theta\|^2,
\]

where \(\Theta\) contains all trainable parameters of the model, and \(\lambda\) is the coefficient of \(L_2\) regularizer.
Table 2: Statistics of datasets for downstream tasks.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Datasets</th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural Language Code Retrieval</td>
<td>AdvTest</td>
<td>251K</td>
<td>9.6K</td>
<td>19.2K</td>
</tr>
<tr>
<td></td>
<td>CosQA</td>
<td>19.6K</td>
<td>0.5K</td>
<td>0.5K</td>
</tr>
<tr>
<td></td>
<td>CoNaLa</td>
<td>2.4K</td>
<td>-</td>
<td>0.5K</td>
</tr>
<tr>
<td>Code-to-Code Retrieval Python800</td>
<td>72K</td>
<td>4K</td>
<td>4K</td>
<td></td>
</tr>
<tr>
<td>Code Clone Detection Python800</td>
<td>144K</td>
<td>8K</td>
<td>8K</td>
<td></td>
</tr>
<tr>
<td>Code Defect Detection GREAT</td>
<td>100K</td>
<td>5K</td>
<td>5K</td>
<td></td>
</tr>
</tbody>
</table>

4 Experimental Setup

We conduct experiments to answer the following research questions: (1) *How effective is CODE-MVP compared with the state-of-the-art baselines?* (2) *How do different components and different views affect our CODE-MVP?*

4.1 Pre-Training Dataset and Settings

Different programming languages often require different program analyzers. Existing program analysis tools rarely support multiple programming languages and multi-view program transformations. For convenience, we choose Python for our experiments, as it is very popular and used in many projects. We pre-train CODE-MVP on the Python corpus of CodeSearchNet dataset (Husain et al., 2019), which consists of 0.5M bimodal Python functions with their corresponding natural-language comments, as well as 1.1M unimodal Python functions.

Code-MVP is built on the top of Transformer (Vaswani et al., 2017), and consists of a 12-layer encoder with 768 hidden sizes and 12 attention heads. The pre-training procedure is conducted on 8 NVIDIA V100 GPUs for 600K steps, with each mini-batch containing 128 sequences up to 512 tokens including special tokens. According to the length distribution of samples in the training corpus, we set the lengths of PL/AST/CFG/PT in unpaired data to 512, and set the lengths of NL and PL/AST/CFG/PT in paired data to 96 and 416 respectively. The learning rate of CODE-MVP is set to 1e-4 with a linear warm up over the first 30K steps and a linear decay. CODE-MVP is trained with a dropout rate of 0.1 on all layers and attention weights. We initialize the parameters of CODE-MVP by GraphCodeBERT (Guo et al., 2021) and utilize a BPE tokenizer (Sennrich et al., 2016).

4.2 Evaluation Tasks, Datasets and Metrics

We select several program comprehension tasks to evaluate CODE-MVP, including natural language code retrieval, code similarity, and code defect detection. We pre-train CODE-MVP on Python corpus, and choose several public Python datasets to evaluate it, as shown in Table 2.

**Natural Language Code Retrieval.** This task aims to find the most relevant code snippet from a collection of candidates, given a natural language query. We choose three datasets to evaluate this task, including AdvTest (Lu et al., 2021), CoNaLa (Yin et al., 2018), and CoSQA (Huang et al., 2021). We adopt the Mean Reciprocal Rank (MRR) metric to evaluate the performance of code retrieval. In AdvTest dataset, we set the learning rate as 5e-5, the batch size as 32, the maximum fine-tuning epoch as 20, the maximum length of both query and code sequence as 256. In CoNaLa and CoSQA datasets, we set the learning rate as 5e-5, the batch size as 32, the maximum fine-tuning epoch as 30, the maximum length of query and code sequence as 128. In AdvTest and CoSQA datasets, we save the optimal checkpoint on the validation set, and test it on the testing set. In CoNaLa dataset, we report the best results on the testing set.

**Code Similarity.** This task is always categorized into two groups: code-to-code retrieval and code clone detection. We conduct experiments on the Python800 dataset (Puri et al., 2021), which is composed of 800 problems with each problem having 300 unique Python solution files. We remove those files not in UTF-8 encoding formats and randomly select 100 solutions for each problem. In code-to-code retrieval, the filtered dataset is split to 720/40/40 problems for training, validation, and testing. Given a program, this task aims to retrieve other programs that solve the same problem; we evaluate using Mean Average Precision (MAP). Regarding the task of code clone detection, we treat it as binary classification and evaluate it using the Accuracy score, following (Puri et al., 2021).

To train these two tasks, we set the learning rate as 2e-5, the batch size as 32, the epoch number as 20. In code-to-code retrieval, we set the maximum length of both query and code sequence as 256. In code clone detection, we set the maximum concatenation sequence length of the two code snippets to 512. We save the optimal checkpoint on the validation set, and test it on the testing set.
Table 3: Results on the natural language code retrieval task evaluating with MRR, using the AdvTest, CoNaLa, and CoSQA datasets.

<table>
<thead>
<tr>
<th>Models</th>
<th>AdvTest</th>
<th>CoNaLa</th>
<th>CoSQA</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>18.3</td>
<td>30.7</td>
<td>57.6</td>
<td>35.5</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>27.2</td>
<td>38.9</td>
<td>64.2</td>
<td>43.4</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>35.2</td>
<td>47.3</td>
<td>68.2</td>
<td>50.2</td>
</tr>
<tr>
<td>PLBART</td>
<td>34.3</td>
<td>45.5</td>
<td>65.3</td>
<td>48.4</td>
</tr>
<tr>
<td>CodeT5</td>
<td>36.5</td>
<td>47.7</td>
<td>67.7</td>
<td>50.6</td>
</tr>
<tr>
<td>SynCoBERT</td>
<td>38.1</td>
<td>48.4</td>
<td>69.6</td>
<td>52.0</td>
</tr>
<tr>
<td>CODE-MVP</td>
<td>40.4</td>
<td>50.6</td>
<td>72.1</td>
<td>54.4</td>
</tr>
</tbody>
</table>

Table 4: Results on the code-to-code retrieval and code clone detection tasks evaluating with MAP and Accuracy score, using the Python800 dataset.

<table>
<thead>
<tr>
<th>Models</th>
<th>MAP@R</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>82.9</td>
<td>94.4</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>86.1</td>
<td>95.2</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>88.8</td>
<td>95.9</td>
</tr>
<tr>
<td>PLBART</td>
<td>86.7</td>
<td>95.5</td>
</tr>
<tr>
<td>CodeT5</td>
<td>88.1</td>
<td>95.7</td>
</tr>
<tr>
<td>SynCoBERT</td>
<td>89.2</td>
<td>96.1</td>
</tr>
<tr>
<td>CODE-MVP</td>
<td>91.5</td>
<td>97.4</td>
</tr>
</tbody>
</table>

5 Results and Analysis

5.1 Performance on Downstream Tasks (RQ1)

Natural Language Code Retrieval. Table 3 shows the results of natural language code retrieval on three datasets. We can observe that CODE-MVP outperforms all baseline models on all datasets. Specifically, it outperforms CodeT5 by 3.8 points on average. Compared to the previous state-of-the-art SynCoBERT, CODE-MVP also performs better with an average improvement of 2.4 points. This significant performance improvement indicates that the code representation learned by CODE-MVP preserves more code semantics. We attribute this improvement to our introduced multi-view contrastive pre-training strategy.

Code Similarity. Table 4 presents the results for code similarity calculation, including code-to-code retrieval and code clone detection. We can see that CODE-MVP significantly outperforms all baseline models on these two tasks. In the task of code-to-code retrieval, CODE-MVP outperforms CodeT5 and SynCoBERT by 3.4 points and 2.3 points, respectively. In the task of code clone detection, CODE-MVP achieves 1.5 and 1.3 points higher compared to GraphCodeBERT and SynCoBERT, respectively. These results show that CODE-MVP can better identify those programs with the same semantics and distinguish those programs with different semantics.

Code Defect Detection. Table 5 shows the experimental results of code defect detection. CODE-MVP consistently outperforms all models. Specifically, it outperforms GraphCodeBERT and SynCoBERT by 1.8 and 1.1 points, respectively. These results indicate that CODE-MVP can effectively preserve the semantics of programs, which is beneficial for code defect detection.

Code Defect Detection. This task aims to identify whether a given piece of code snippet is vulnerable or not, which is usually treated as a binary classification task. We evaluate all models on the GREAT dataset (Hellendoorn et al., 2020), which is originally built from the ETH Py150 dataset (Raychev et al., 2016). We evaluate the performance of code defect detection using the Accuracy score. We randomly select 100K samples for training, 5K samples for validation and 5K samples for testing, respectively. We set the learning rate as $5 \times 10^{-5}$, the batch size as 32, the maximum fine-tuning epoch as 50, the maximum length of both query and code sequence as 256. We save the optimal checkpoint on the validation set, and test it on the testing set.
<table>
<thead>
<tr>
<th>Models</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>81.9</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>85.5</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>87.5</td>
</tr>
<tr>
<td>PLBART</td>
<td>86.8</td>
</tr>
<tr>
<td>CodeT5</td>
<td>87.4</td>
</tr>
<tr>
<td>SynCoBERT</td>
<td>88.2</td>
</tr>
<tr>
<td>CODE-MVP</td>
<td>89.3</td>
</tr>
</tbody>
</table>

Table 5: Results on the code defect detection task evaluating with Accuracy score, using the GREAT dataset.

<table>
<thead>
<tr>
<th>Models</th>
<th>AdvTest</th>
<th>CoNaLa</th>
<th>CoSQA</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE-MVP</td>
<td>40.4</td>
<td>50.6</td>
<td>72.1</td>
<td>54.4</td>
</tr>
<tr>
<td>w/o MVCL</td>
<td>36.2</td>
<td>47.7</td>
<td>69.2</td>
<td>51.0</td>
</tr>
<tr>
<td>w/o FGTI</td>
<td>38.0</td>
<td>48.9</td>
<td>70.8</td>
<td>52.6</td>
</tr>
<tr>
<td>- w/o AST</td>
<td>39.1</td>
<td>48.5</td>
<td>71.3</td>
<td>53.0</td>
</tr>
<tr>
<td>w/o PT</td>
<td>38.2</td>
<td>48.6</td>
<td>70.8</td>
<td>52.5</td>
</tr>
<tr>
<td>w/o CFG</td>
<td>37.8</td>
<td>47.9</td>
<td>70.5</td>
<td>52.1</td>
</tr>
</tbody>
</table>

Table 6: Ablation study on the task of natural language code retrieval, evaluated using MRR.

5.2 Ablation Study (RQ2)

We empirically study several simplified variants of CODE-MVP to understand the contributions of each component, including the Multi-View Contrastive Learning (MVCL), Fine-Grained Type Inference (FGTI), Abstract Syntax Tree (AST), Program Transformation (PT), and Control Flow Graph (CFG). Taking the natural language code retrieval task as an example, Table 6 shows the experimental results of each variant on that task. The setting of w/o (MVCL, FGTI) indicates that these pre-training objectives are removed from CODE-MVP respectively. The setting of w/o (AST, PT, CFG) indicates that different views of programs are removed from CODE-MVP respectively. From Table 6, several meaningful observations can be drawn. (1) Both MVCL and FGTI effectively increase the performance, which confirms that the two proposed pre-training objectives can indeed improve the ability of the model for program comprehension. (2) Exploiting different views of programs can bring performance improvements to the model as arbitrarily discarding any view of programs degrades the performance. Additionally, the introduction of CFG brings more performance improvements, indicating the importance of execution information for program understanding.

6 Related Work

Pre-Trained Models for Source Code. Benefiting from the strong power of pre-trained models in natural language processing (Liu et al., 2019; Devlin et al., 2019; Wang et al., 2021a, 2020a,b), several recent works attempt to use the pre-training techniques on programs (Svyatkovskiy et al., 2020). Kanade et al. (2020) proposed CuBERT which follows the architecture of BERT (Devlin et al., 2019), and is pre-trained with a masked language modeling objective on a large-scale Python corpus. Feng et al. (2020) proposed CodeBERT, which is pre-trained on NL-PL pairs in six programming languages, introducing the replaced token detection objective (Clark et al., 2020). Furthermore, Guo et al. (2021) proposed GraphCodeBERT, which incorporates the data flow of programs into the model pre-training process. Wang et al. (2021) proposed SynCoBERT, which incorporates ASTs via edge prediction to enhance the structural information of programs. They also used contrastive learning to maximize the mutual information among programs, documents, and ASTs. Lu et al. (2021) proposed CodeGPT for code completion, which is pre-trained using a unidirectional language modeling objective. Ahmad et al. (2021) proposed PLBART based on BART (Lewis et al., 2020), which is pre-trained on a large-scale corpus of Java and Python programs paired with their corresponding comments via denoising autoencoding. Wang et al. (2021b) proposed CodeT5 following the architecture of T5 (Raffel et al., 2020). It employs denoising sequence-to-sequence pre-training on seven programming languages. Recently, Wan et al. (2022b) conducted a thorough structural analysis aiming to provide an interpretation of pre-trained language models for source code (e.g., CodeBERT and GraphCodeBERT).

Program Analysis for Code Intelligence. In addition to the lexical information of programs, many recent works attempt to leverage program analysis techniques to capture the structural and syntactic representations of programs (Cummins et al., 2020). Kim et al. (2021) designed several strategies to feed the ASTs of programs into Transformer (Vaswani et al., 2017). Li et al. (2019) proposed a graph matching network, which utilizes the CFG of the program to deal with the challenge of binary function similarity search. Ling et al. (2021) proposed a deep graph matching and searching model based on graph neural networks (Kipf and Welling, 2017; Wang et al., 2021b,a; Yu et al., 2022; Zhao et al., 2022) for code retrieval. They represented both natural language queries and code snippets based on...
the unified graph-structured data. Iyer et al. (2020) presented the program-derived semantic graph to capture the semantics of programs at multiple levels of abstraction. Ben-Nun et al. (2018) presented inst2vec, which locally embeds individual statements in LLVM intermediate representations by processing a contextual flow graph with a context prediction objective (Mikolov et al., 2013).

**Contrastive Learning on Programs.** Recently, several attempts have been made to leverage contrastive learning for better code semantics. ContraCode (Jain et al., 2021) and Corder (Bui et al., 2021b) first utilized semantic-preserving program transformations such as identifier renaming, dead code insertion, to build positive instances. Then a contrastive learning objective is designed to maximize the mutual information among the positive and negative instances. Ding et al. (2021) presented a self-supervised pre-training technique called BOOST based on contrastive learning. They inject real-world bugs to build hard negative pairs. In CODE-MVP, we construct the positive pairs throughout the compilation process of programs, including lexical analysis, syntax analysis, semantic analysis, and static analysis. It is the first pretrained model that integrates multi-views of programs for program comprehension.

7 Conclusion

In this paper, we have proposed CODE-MVP, a novel approach to represent the source code with multi-view contrastive pre-training learning. We extract multiple code views with compiler tools and learn the complement among them under a contrastive learning framework. We also propose a fine-grained type inference task in the pre-training process. Comprehensive experiments on three downstream tasks over five datasets verify the effectiveness of CODE-MVP when compared with several state-of-the-art baselines.

Acknowledgements

We would like to thank Gerasimos Lampouras and Ignacio Iacobacci from Huawei London Research Institute for their constructive comments on this paper. Jin Liu is supported by National Natural Science Foundation of China under Grant No. 61972290. Yao Wan is partially supported by National Natural Science Foundation of China under Grant No. 62102157. Hao Wu is supported by National Natural Science Foundation of China under Grant No. 61962061, and partially supported by Yunnan Provincial Foundation for Leaders of Disciplines in Science and Technology (202005AC160005).

References


Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The python static analysis framework. CoRR, abs/2202.11840.


