Towards Better Multilingual Code Search through Cross-Lingual Contrastive Learning

Xiangbing Huang*
National University of Defense Technology, China
xbhuang@nudt.edu.cn

Yingwei Ma*
National University of Defense Technology, China
yingwei.ywma@gmail.cn

Haifang Zhou†
National University of Defense Technology, China
haifang_zhou@nudt.edu.cn

Zhijie Jiang
National University of Defense Technology, China
jiangzhijie@nudt.edu.cn

Yuanliang Zhang
National University of Defense Technology, China
zhangyuanliang13@nudt.edu.cn

Teng Wang
National University of Defense Technology, China
wangteng13@nudt.edu.cn

Shanshan Li
National University of Defense Technology, China
shanshanli@nudt.edu.cn

ABSTRACT

Recent advances in deep learning have significantly improved the understanding of source code by leveraging large amounts of open-source software data. Thanks to the larger amount of data, code representation models trained with multilingual datasets show superior performance to monolingual models and attract much more attention. However, the entangled source code from various programming languages makes multilingual models hard to differentiate language-specific textual semantics or syntactic structures, which significantly increases the difficulty of model learning from multilingual datasets directly. On the other hand, for a given problem, developers are likely to choose similar identifiers, even if coding in different languages. However, the presence of similar identifiers in multilingual code snippets does not mean that they implement the same functionality, which may misdirect models to overemphasize these unreliable signals and ignore the semantic information of multilingual code. To tackle the above issues, we propose LAM-Code, a language-aware multilingual code understanding model. Specifically, we propose a simple yet effective method to perceive linguistic information by injecting language-specific viewer into the language models. Furthermore, we introduce a cross-lingual contrastive learning method by generating more similar training instances but with fewer overlapping features. This method prevents the models from over-relying on similar identifiers across languages. We conduct extensive experiments to evaluate the effectiveness of our approach on a large-scale multilingual dataset. The experimental results show that our approach significantly outperforms the state-of-the-art methods.

KEYWORDS

multilingual, code representation, contrastive learning

ACM Reference Format:
Xiangbing Huang, Yingwei Ma*, Haifang Zhou, Zhijie Jiang, Yuanliang Zhang, Teng Wang, and Shanshan Li. 2023. Towards Better Multilingual Code Search through Cross-Lingual Contrastive Learning. In *Proceedings of The 14th Asia-Pacific Symposium on Internetware (Internetware 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Recent advances in machine learning have benefited many software engineering-related tasks, such as code search [13, 17, 27, 47], code summarization [8, 18, 26, 34], and code synthesis [23, 35]. Open source code repository sites like Github [1] and Q&A sites like Stack Overflow [2] produce an enormous of source code data, laying the foundation for large-scale programming language models such as CodeBERT [12], GraphCodeBERT [16], CodeT5 [41], and UniXcoder [15]. These large-scale models have been pre-trained on large multilingual datasets, improving understanding of code semantics and achieving promising performance in code-related downstream tasks.

Pre-training data does not require manual annotation and is easy to collect, but fine-tuning for downstream tasks requires labeled data, which requires more time and attention. To utilize more high-quality labeled data, Ahmed *et al.*[1] pooled data across languages to create a multilingual training set and indicated that multilingual training is effective not only in the pre-training stage but also in the fine-tuning stage of the software engineering model. As pointed out by Ahmed *et al.*[1], for a given problem, even code snippets written in different programming languages have similar

*Co-first author
†Corresponding author

Internetware 2023, August 04–06, 2023, Hangzhou, China

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1]https://github.com/
[2]https://stackoverflow.com/

Python

```
def longestPalindrome(self, s: str) ->
int:
    ans = 0
    count = collections.Counter(s)
    for v in count.values():
        ans += v // 2 * 2
        if ans % 2 == 0 and v % 2 == 1:
            ans += 1
    return ans
```

C++

```
int longestPalindrome(string s) {
    unordered_map<char, int> count;
    int ans = 0;
    for (char c : s)
        ++count[c];
    for (auto p : count) {
        int v = p.second;
        ans += v / 2 * 2;
        if (v % 2 == 1 and ans % 2 == 0)
            ++ans;}
    return ans;
}
```

Figure 1: Same functionality code snippets implemented in Python and C++.

Python

```
def inorder(root: TreeNode, result):
    if not root:
        return
    inorder(root.left)
    result.append(root.val)
    inorder(root.right)
```

C++

```
public void preorder(TreeNode root,
                     List<Integer> result) {
    if(root == null){
        return;
    }
    result.add(root.val);
    preorder(root.left, result);
    preorder(root.right, result);
}
```

Figure 2: Different functionality code snippets implemented in Python and C++.

identifiers, so multilingual models can take advantage of this feature of multilingual data to improve performance in any language.

However, due to the differences and commonalities between code written in different languages, the existing multilingual training methods still have two limitations. On one hand, due to textual (*i.e.*, token) and syntactic (*e.g.*, AST) differences among programming languages, there are significant differences in the implementation way of multilingual code with the same function. However, the existing methods build a unified model with shared parameters among multiple programming languages, making the trained model insensitive to language and difficult to distinguish language-specific information. For example, Figure 1 shows code snippets for the same functionality implemented in Python and C++, respectively. We can see that although both code snippets implement the function of solving the longest palindrome, there are differences in how they are implemented, such as the language built-in data structures and looping statements (highlighted in red). These differences are due to specific syntax and keywords of different languages, so how to perceive linguistic information during multilingual training is crucial.

On the other hand, for a given problem, developers are likely to choose similar identifiers, even if coding in different languages. However, the presence of similar identifiers in multilingual code snippets does not mean that they implement the same functionality. For example, Figure 2 shows in-order and pre-order traversal of a binary tree implemented in Python and C++, respectively. Although the two pieces of code share many similar identifiers (highlighted in red), they implement different traversal way. Existing methods may misdirect models to overemphasize these unreliable signals and ignore the semantic information of multilingual code, which have not been fully explored in previous work.

To address the above problems, we propose a ***Language-Aware Multilingual Code*** *Understanding Model,* ***LAMCode*** in short. LAMCode originates from our observation that code written in different programming languages has both cross-language shared information and language-specific information. Therefore, we propose a simple yet efficient method to perceive linguistic information by injecting language-specific viewer into the language models. Specifically, we add different [*Language*] special tokens (*e.g.*, [*C*], [*Java*]) to the code input to capture language-specific information. Then, we integrate the language-specific information with the cross-language shared information obtained by pre-trained models such as CodeBERT [12] to generate the final code representation.

In addition, to better capture the semantic relationship between multilingual code, we introduce the cross-lingual contrastive learning method to enhance the semantic matching of multilingual code. In the previous work, the contrastive loss between positive and negative samples is widely applied [19, 29, 44]. Apart from augmenting negative samples within a mini-batch, we propose an Overlap-Reduction Augmentation method to generate more similar training instances but with fewer overlapping features. This approach can effectively prevent the model from over-relying on some unreliable signals (*e.g.*, similar identifiers across languages) while ignoring the semantic information of multilingual code, which is also validated in our experiment.

The contributions of this paper are as follows:

- We propose a simple yet effective method to generate language-aware code representations through injecting language-specific viewer into the language models.
- To better capture the semantic relationship of multilingual code, we introduce a cross-language contrastive learning method to enhance multilingual semantic matching. To prevent the model from overemphasizing unreliable signals among languages, we propose an augmentation method to generate more similar training instances but with fewer overlapping features.
- We evaluate the effectiveness of our method on two tasks, natural language code search and cross-language code search. Experimental results on a large multilingual code dataset show that our method achieves state-of-the-art performance. Further analysis demonstrated the effectiveness of the approach. All data and source code can be found in our anonymous repository. [3]

2 BACKGROUND & MOTIVATION

As mentioned in the introduction, the existing multilingual training methods still have two limitations. The first limitation is multilingual differences, which result in models that do not perceive linguistic information and leverage language-specific information to understand code. The second limitation is multilingual commonality, which may cause the model to over-rely on these unreliable

[3] https://github.com/yingweima2022/LAMCode

Table 1: Multilingual differences.

Dif/Language	Python	C++	Java
code style	parse code blocks by indentation	parse code blocks by {}	parse code blocks by {}
statement structure	End the statement with \n	End the statement with ;	End the statement with ;
data types	No need to explicit definition	explicitly define	explicitly define
loop statement	for ... in ... while	for(; ; ;) while/do while	for(; ; ;) while/do while
loop control statement	break/continue/pass	break/continue	break/continue
conditional statement	if ... elif ... else	if ... else if ... else switch case	if ... else if ... else switch case
input/output	input print	cin/scanf cout/printf	nextLine/next System.out.println
keyword	def/with using/...	goto/void auto/...	boolean/byte package/...
common third-party libraries	Django/NumPy Pytorch/...	Boost/OpenCV Caffe/...	Spring/OpenNLP ApacheCommons/...
data structure	List/Tuple Dictionary/Set	Vector/Array/List Queue/Stack Map/Set	Set/List/ArrayList Stack/Queue Map/Collection

features while ignoring the semantic information of the code. We now present some motivating evidence that further illustrates the differences and commonalities of multilingual datasets that inspired our design of LAMCode.

2.1 Multilingual Differences

As shown in Figure 1, we use an example to illustrate that there are some differences between different languages; to describe the differences among languages in further detail, we observe the three languages Python, C++ and Java in the XLCoST dataset and summarize the differences as Table 1.

These language-related keywords (*e.g.*, input/output statement) and grammatical differences (*e.g.*, there is no *do while* statement in Python) provide language-specific information that previous multilingual approaches have ignored. That said, previous methods build a unified model with shared parameters among multiple programming languages, making the trained model insensitive to language and unable to utilize language-specific information to understand multilingual code. Inspired by this, we propose the language-specific viewer method to perceive language information for better understanding of multilingual code.

2.2 Multilingual Commonality

Ahmed *et al.*[1] experimentally demonstrate that for a given problem, developers are likely to choose similar identifiers, even if they are coded in different languages. Specifically, Ahmed *et al.*[1] obtained 15 cross-language pairs from six languages, Ruby, JavaScript, Java, Go, PHP, and Python, and measured the identifier similarity between pairs of programs that solve the same problem in each language (e.g., programs for graph diameter problem in Java and Python). They compared the difference in identifier similarity between paired and randomly selected unpaired data. Specifically, for Java and Python, they found 544 matching pairs of programs

Table 2: Cross-language identifier similarity.

Language pair	#of common programs	for random programs	Overlap coefficient for common programs	increased in %
Java & Python	544	0.10	**0.32**	+210.67%
Java & PHP	282	0.08	**0.28**	+235.01%
Python & PHP	267	0.09	**0.29**	+214.21%

solving the same problem in both languages, and then they randomly paired 544 Java programs with 544 other Python programs. So they can take two sets of programs (*i.e.*, the same program implemented in different languages and different programs implemented in different languages) and check the similarity between them.

Table 2 shows that in Java, Python, and PHP, the common program pairs have 210%-235% additional identifier overlap compared to the random program pairs, which mearsures by Szymkiewicz-Simpson Overlap coefficient [4]. Although there are more overlapping identifiers between common programs across languages than random programs, this may misdirect models to overemphasize these unreliable signals and ignore the semantic information of the code, which has not been fully explored in previous work. For example, the two pieces of code in Figure 2 shown in Section 1, although there are many overlapping identifiers (*e.g.,root, left, right*), implement different functions. To solve this problem, we propose the cross-language contrastive learning method to exploit this multilingual code feature better.

3 APPROACH

We introduce LAMCode, a multi-language code understanding model. It consists of two components: the Language Specific Viewer module and the Cross-Language Contrastive Learning module. Specifically, the Language Specific Viewer module generates a language-specific code vector for multi-language code snippets. It combines the code vector obtained through the shared parameter model to get the final representation. To effectively capture the semantic relation of multilingual code, we introduce the cross-lingual contrastive learning method to enhance the semantic information of multilingual code by generating similar code snippets with fewer overlapping features as paired positive examples. The overall framework of our model is shown in Figure 3.

3.1 Preliminary

In this section, we introduce the basic model, input samples, output representations, and contrastive learning mechanism and build our model on top of these.

Since pre-trained models such as CodeBERT [12] achieve good code representation through self-supervised training on massive open source data and substantially improve many downstream tasks, we use CodeBERT as the base code encoder. CodeBERT generates high-quality text and code embeddings by pre-training on the CodeSearchNet[20] corpus with two tasks, masked language modeling and replaced token detection. Following previous research [16, 33], in order to obtain the entire sequence representation of the code, a special token [*CLS*] is usually inserted at the beginning

[4]This is a measure of similarity like the Jaccard index; it's calculated as $\frac{|X \cap Y|}{min(|X|,|Y|)}$.

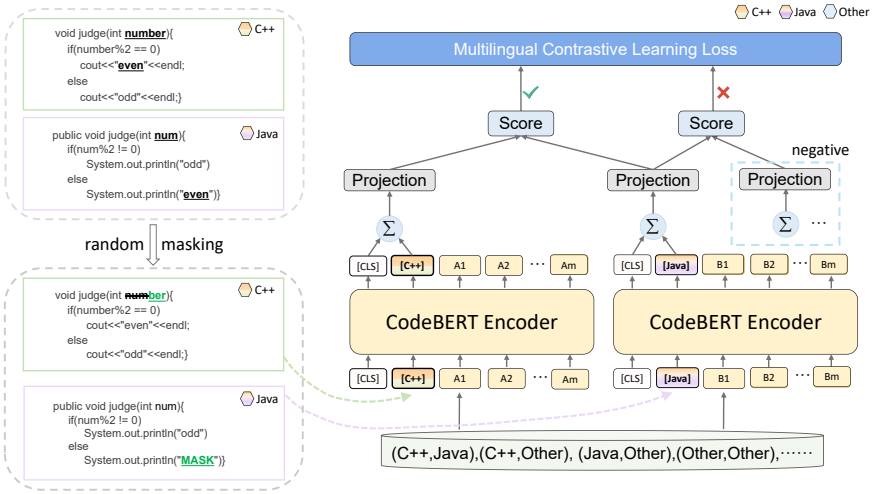

Figure 3: Architecture of LAMCode.

of the input code sequence, and the embedding of $[CLS]$ at the last layer is taken as the sequence representation of the whole code.

After obtaining the representation of the code, the model estimates the similarity between code using cosine distance or Euclidean distance. Contrastive learning is a new paradigm that learns semantic relationships between code by minimizing the distance between representations of similar code (positive examples) and pushing away the distance of dissimilar code (negative examples). InfoNCE [36] is a widely used contrastive learning loss; its form is shown in Eq. 1. For a given code c, its value is lower when it is similar to the matching code embedding of $c+$ and dissimilar to the non-matching code embedding of $c-$. The temperature hyperparameter [42] is denoted by t, and an appropriate value of t can help optimize the model better.

$$\mathcal{L}_{c,c^+,c^-} = -log\frac{exp(c \cdot c^+/t)}{exp(c \cdot c^+/t) + \sum_{c^-} exp(c \cdot c^-/t)} \tag{1}$$

3.2 Language Specific Viewer Based Architecture

Limited by the traditional way of multilingual code representation, the existing methods build a unified model with shared parameters among multiple programming languages, making the trained model insensitive to language and difficult to distinguish language-specific text semantics and syntax structures. While some previous studies [14] attempt to narrow the differences across languages to obtain uniform representations and somehow improve the effectiveness, they usually lead to more expensive preprocessing costs and generality issues. Therefore, we propose a simple yet effective method to perceive linguistic information and produce language-specific code representations; we will describe it in detail.

As pre-trained models achieve significant improvements on many downstream tasks, some work found that $[CLS]$ tends to obtain the overall semantic information of the sentence [9, 24]. However, our model tends to perceive specific language information of different language code, so we propose the Language Specific Viewer module. Instead of just using the latent representation of the $[CLS]$ token, we assign code snippets of each language a newly added language token $[Language]$, such as $[C]$ and $[Java]$, which are initialized randomly. For code sequence input, we add language-specific tokens after the $[CLS]$ token according to the language type of the code snippets. We then utilize the CodeBERT encoder to obtain two representations of the code snippets: the $[CLS]$ token captures the general semantic information of the code snippet, and the $[Language]$ token captures language-specific information within the code snippet:

$$E(a) = Encode_a([CLS] \circ [C++] \circ a \circ [SEP]) \tag{2}$$

$$E(b) = Encode_b([CLS] \circ [Java] \circ b \circ [SEP]) \tag{3}$$

Where $\circ$ is the concatenation operation. $[CLS]$ and $[SEP]$ are special tokens in the original CodeBERT. $[C++]$ and $[Java]$ are newly added special language tokens. $Encode_a$ and $Encode_b$ represent code encoder with shared parameters.

We use the last layer embedding of $[CLS]$ and $[Language]$ as the sequence representation of the whole code. Specifically, we first obtain the last layer embedding of the $[CLS]$ token and $[Language]$ token. Then we compute the final code representation by summing the two representations and mapping this representation to a unified space through a linear layer. Finally, we use the inner product to calculate their similarity score. The overall similarity score is in the form of the following:

$$Score(a, b) = Sim(F(a), F(b)) \tag{4}$$

$$F(a) = liner(sum(E(a)_{CLS}, E(a)_{C++})) \tag{5}$$

$$F(b) = liner(sum(E(b)_{CLS}, E(b)_{Java})) \tag{6}$$

3.3 Cross-Language Contrastive Learning Method

How to effectively capture the semantic relation of multilingual code is essential for code understanding. In this work, we introduce the cross-lingual contrastive learning method to enhance the semantic matching of multilingual code. These include the Overlap-Reduction Augmentation module and the In-Batch Multilingual Augmentation module. We will describe it in detail.

3.3.1 Overlap-Reduction Augmentation. To prevent the model from over-relying on similar identifiers across languages, as shown in Figure 3, we propose augmenting the positive examples with the random masking technique. Inspired by the observation of Section 2.2, the more similar tokens across languages may not necessarily represent semantic consistency; however, they may mislead the models to overemphasize these unreliable signals and ignore the semantic information of the code. Therefore, we hypothesize that a certain degree of similar token masking or deletion for pairs of samples may alleviate this phenomenon.

Specifically, we match pairs of code-code pairs or query-code pairs between pairs and then process similar tokens in two ways. If the same token exists between pairs, we randomly select a token on one side to replace it with the <*mask*>token. Since the identifiers in the code are often abbreviated, for example, the *maximum* token is abbreviated as the *max* token. In this case, we use the longest common substring method for matching. If the length of the longest common substring is greater than l, we will randomly remove the public substring on one side. For example, after deleting the common substring *max* token, the *maximum* token is modified to the *imum* token and used as a new positive example. We will randomly select one of these two methods to enhance each positive sample.

3.3.2 In-Batch Multilingual Augmentation. In-Batch Multilingual Augmentation generates negative samples by combining code or queries with randomly sampled multilingual code within a mini-batch. For a given positive pair, the other N-1 code snippets of different languages are treated as negative samples and forced away by the models. Specifically, for one sample (q_i, c_i) in the n samples $(q_1, c_1), (q_2, c_2), ..., (q_n, c_n)$ of the mini-batch, we pair q_i and other n-1 $c_{j(j!=i)}$ samples as negative samples. In-Batch Multilingual Augmentation enables the model to distinguish the semantic relations of different languages better and make semantically different code farther apart in the representation space.

4 EXPERIMENTAL SETUP AND RESULTS

In this paper, we consider two common and challenging source code understanding tasks: natural-language (NL) code search and cross-language (XL) code search. We will introduce the objectives of each task, the datasets we use, baseline methods, and experimental settings. Finally, we ask four research questions and design experiments to answer them.

4.1 Tasks

NL code search: The code search task aims to find the most semantically relevant code snippet from the code collection for a given natural language query. Specifically, the code search task usually requires two encoders. One is a natural language encoder, which encodes natural language queries into query embeddings. The other is a code encoder, which encodes source code to code embedding. During the retrieval process, for a given natural language query, we retrieve all code embeddings to find the closest code embedding to its embedding and use the corresponding code snippet as the retrieval result. In this paper, we use the initialization parameters of CodeBERT [12] to initialize two encoders. We evaluate the performance of our method on this task using the Mean Reciprocal Rank (MRR) metric, which is widely used in previous work [13, 27]. MRR is the average of reciprocal ranks of the correct code snippets for given queries Q. The higher the MRR value, the better the code search performance is. The computation of MRR is : $MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank_i}$, where Q denotes the set of queries in the automatic evaluation; $Rank_i$ denotes the rank of the ground-truth code corresponding to the i_{th} query.

XL code search: For code snippets in a given language, the XL code search task aims to retrieve functionally similar code in multiple other languages. The XL code search also requires two encoders, but different from the NL code search, the two encoders of the XL code search are source code encoders, which encode the source code into the code embedding. We also use the initialization parameters of CodeBERT [12] to initialize both encoders. During the retrieval process, for a given code snippet, we retrieve all multilingual code embeddings to find the closest code embedding to its embedding and return its corresponding code snippet as the return result. Unlike NL code searches, a code snippet may return multiple correct results in different languages. To account for multiple correct answers, we use a modified MRR [46] for evaluation. The computation of modified MRR is: $MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} MRR_{q_i}$, $MRR_{q_i} = \frac{1}{k}\sum_{j=1}^{n} r_{ij}(c_{ij} \in A_i)$, where q_i denotes a query from the set of queries $Q = \{q_1, q_2, ...q_{|Q|}\}$; candidate set $C_i = \{c_{i1}, c_{i2}, ..., c_{in}\}$ corresponding to q_i and the answer set $A_i = \{a_{i1}, a_{i2}, ..., a_{ik}\}$ where $k \in [1, 6]$, r_{ij} is the reciprocal rank of the j_{th} candidate c_{ij} for $c_{ij} \in A_i$.

4.2 Datasets

We conduct experiments on the large-scale benchmark dataset XLCoST [46]. It contains seven programming languages - C++, JavaScript, C, Python, Java, C# and PHP. This dataset is oriented toward cross-language code intelligence tasks and is the largest parallel dataset in source code in terms of size and the number of languages. Given a program in one language, the dataset contains the same program in up to six other programming languages. Moreover, each of the snippets is accompanied by a comment, and the comment for a particular snippet is the same across all the languages. The training and test data contain query-code and code-code pairs for the NL Code Search and XL Code Search tasks. According to previous research [26, 27, 32], the model retrieves the correct code snippet from the candidate code for a given query/code when performing the evaluation. Since the test set of C language in the dataset is small, to ensure accuracy, we use other six programming languages for training in the NL Code Search task; in XL Code Search, we use all the languages in the dataset because the C language test set includes test data from C to the other six languages.

Table 3: Effectiveness of LAMCode for NL Code Search task.

Task	Model	C++	Java	Py	C#	JS	PHP
NL Code Search	RoBERTa	51.47	50.40	48.98	52.24	50.05	62.01
	CodeBERT	59.13	56.07	57.91	56.65	54.37	65.13
	polyglotCodeBERT	63.84	63.80	61.97	63.92	62.79	72.76
	LAMCode	**65.69**	**65.48**	**64.54**	**65.41**	**65.49**	**74.47**

Table 4: Effectiveness of LAMCode for XL Code Search task.

Task	Model	C++	Java	Py	C#	JS	PHP	C
XL Code Search	RoBERTa	47.02	48.79	47.25	46.73	47.06	43.92	39.86
	CodeBERT	47.25	48.67	47.05	46.98	47.17	44.02	38.22
	polyglotCodeBERT	47.43	48.85	47.60	47.20	47.45	44.52	41.43
	LAMCode	**47.58**	**49.01**	**47.77**	**47.29**	**47.45**	**44.54**	**41.69**

4.3 Baselines

To evaluate the effectiveness of our approach, we compare LAMCode with three pre-training-based methods: RoBERTa [28], CodeBERT [12], and polyglotCodeBERT [1].

RoBERTa [28] is built on a multi-layer bidirectional Transformer [69] encoder and pre-trained with masked language modeling (MLM) task, which learns language representations by predicting the original tokens of the masked positions. RoBERTa is pre-trained on massive natural language corpora.

CodeBERT [12] used the BERT [10] architecture to generate high-quality text and code embeddings by pretraining on the CodeSearchNet [20] corpus with two tasks, masked language modeling and replaced token detection, which uses a discriminator to identify the replaced token.

polyglotCodeBERT [1] uses CodeBERT's architecture and pre-training parameters. Unlike the original CodeBERT, polyglotCodeBERT uses multilingual mixed data for fine-tuning instead of single-language data in the fine-tuning stage and has achieved improvements in the corresponding single-language test set.

4.4 Experimental Settings

All baselines are initialized with pre-trained weights and default configurations (including hyperparameters) published by the respective original authors. Referring to [1], we changed the source and target sequence lengths to align with the dataset according to the tasks, and we set the max length as 512. The longest common substring l is 4. The temperature coefficient is 0.05. The models were trained using 4 Tesla V100 GPUs with 32GB of memory on each GPU. The code for training and evaluation is published in the dataset's GitHub repository.

4.5 Research Questions and Results

In this section, to evaluate our proposed LAMCode model, we conduct experiments to answer the following research questions:

RQ1: What is the effectiveness of LAMCode when compared with state-of-the-art models?

To answer this research question, we evaluate the effectiveness of our method by comparing three pre-trained models (RoBERTa [28], CodeBERT [12], and polyglotCodeBERT [1]).

Table 5: Effectiveness of each component in LAMCode model for NL Code Search task.

Task	Model	C++	Java	Py	C#	JS	PHP
NL Code Search	polyglotCodeBERT	63.84	63.80	61.97	63.92	62.79	72.76
	LAMCode	**65.69**	**65.48**	**64.54**	**65.41**	**65.49**	74.47
	-w.o Language Viewer	63.01	63.15	61.78	63.01	62.64	73.13
	-w.o Overlap-Reduction	64.85	64.27	63.51	64.70	63.26	72.39
	-w.o In-Batch	64.05	64.26	63.03	64.06	62.79	**75.74**

NL Code Search: Table 3 shows the overall performance of our model and other baselines, measured in MRR, and the values in the table represent percentages. (RoBERTa and CodeBERT are the published values in [46]). For polyglotCodeBERT, we reproduce its results in the XLCoST dataset. From this table, we can observe that our LAMCode outperforms other baselines in all languages. Specifically, compared with the state-of-the-art results on the C++, Java, Python, C#, JavaScript and PHP datasets, our approach improves the MRR by 1.85, 1.68, 2.57, 1.49, 2.7, and 1.71, respectively. Overall, LAMCode improves MRR by 1.49–2.70 across the six programming languages. In addition, we can see that polyglotCodeBERT has achieved better results than other baselines such as CodeBERT and obtained the same conclusion with [1], indicating that multilingual training can make the model use more training data to improve the performance of the model. Our method achieves better results than polyglotCodeBERT, indicating that LAMCode can better learn the representations of different programming languages and utilize multilingual data to improve model performance.

XL Code Search: Table 4 shows the overall performance of our model and other baselines. We reproduce the results for all baselines and measure the performance of the model using the modified MRR as a metric, and the values in the table represent percentages. The modified MRR metric is the average of search results from one language to six other languages, reflecting the average search performance across all languages. From this table, we can observe that our LAMCode outperforms other baselines in all languages. Cross-language code search is a programming language-related task. It needs to use the code of a given language to search for code with similar functions in different languages, which has high requirements for cross-language code understanding ability. It shows that our method understands multilingual code better than other methods, which produces a performance improvement.

RQ2: How much do different components contribute?

In this section, we design an experiment on the NL Code Search task to test the effects of two components of LAMCode: the Language Specific Viewer method and the Cross-Language Contrastive Learning method. The experimental results are shown in Table 5.

Language Specific Viewer Method: We construct experiments to evaluate the effectiveness of the Language Specific Viewer module. Specifically, we evaluate its impact on the model by removing the Language Specific Viewer. After removing the Language Specific Viewer, the model degenerates to only use [*CLS*] to represent the code vector, and the model cannot perceive language information. In Table 5, the item *-w.o Language Viewer* in the second row is the result of removing the Language Specific Viewer. It can be seen from the results that removing the Language Specific Viewer will lead to a decrease in the MRR score, which shows that the Language

Specific Viewer can help the model to perceive language information and make better use of multilingual data during training to improve the performance of the model.

Cross-Language Contrastive Learning Method: To investigate the impact of the Cross-Language Contrastive Learning method on code understanding, we construct an ablation study on the NL Code Search task to analyze the major components of the contrastive learning method that are important to help achieve good performance. Specifically, we remove Overlap-Reduction Augmentation and In-Batch Multilingual Augmentation, respectively, to verify their impact on the model. In Table 5, the item *-w.o Overlap-Reduction* in the third row is the result of removing Overlap-Reduction Augmentation, and the item *-w.o In-Batch* in the fourth row is the result of removing In-Batch Multilingual Augmentation. We can see from the results that after removing the two methods, the MRR scores in most languages decreased, indicating that the two methods helped the model to capture the semantic relationship of multilingual code better and improve the comprehension ability of multilingual code. Although the MRR on PHP enhanced after removing In-Batch Multilingual Augmentation, In-Batch Multilingual Augmentation played a significant positive role in other languages.

RQ3: What is the effectiveness of LAMCode in the case study?

In this section, we show some cases to demonstrate the effectiveness of our model LAMCode.

As shown in Figure 4, there are two examples retrieved by polyglotCodeBERT and LAMCode for qualitative analysis. Listing1 and Listing2 are the results returned by LAMCode and polyglotCodeBERT for the query "Maximum difference between a pair of adjacent elements by excluding every element once." from the Python dataset. To implement the function of this query, it is usually necessary to traverse adjacent elements for comparison. Our model LAMCode can successfully understand the semantics of the whole code snippet and query and return the correct result, while polyglotCodeBERT cannot. This is because polyglotCodeBERT pays more attention to overlapping keywords between code and query, such as [$maximum, difference, adjacent$] of the query and the function name $maxAdjacentDifference$ of the wrong return result have many overlapping words. Paying more attention to overlapping keywords will mislead the model to overemphasize these unreliable signals and ignore the semantic information of the code and query.

The above concrete case demonstrate the advantages of LAMCode in understanding the multilingual code snippets.

RQ4: How effective is the Language Specific Viewer?

To qualitatively analyze the effect of the Language Specific Viewer method in LAMCode, we refer to the practice in [39] to analyze the attention weight, which is the core mechanism based on the transformer model. Intuitively, the attention score defines the closeness of each pair of code tokens. The higher the attention score between two tokens, the greater the mutual attention they have. From an attention analysis perspective, we aim to analyze how language-specific viewer and [*CLS*] capture source code information.

Specifically, at each Transformer layer, we can obtain a series of attention weights over the input code, where attention weight is the degree of attention from the i-th code token to the j-th code token. In this analysis, we use the weights of the last layer of the Transformer for visualization. Figure 5 shows the weight scores of

```python
def maxAdjacent(arr, N):
    res=[]
    for i in range(1, N-1):
        prev=arr[0]
        maxi=-1*float(inf)
        for j in range(1, N):
            if(i==j):
                continue
            maxi=max(maxi,abs(arr[j]-
                              prev))
            prev=arr[j]
            res.append(maxi)
    for x in res:
        print(x)

arr = [1,3,4,5,6]
N=len(arr)
maxAdjacent(arr, N)
```

(a) Listing1: The first result returned by LAMCode.

```python
def maxAdjacentDifference(N, K):
    if(N==1):
        return 0
    if(N==2):
        return K
    return 2*K

N=6
k=11
print(maxAdjacentDifference(N, k))
```

(b) Listing2: The first result returned by ploygotCodeBERT.

Figure 4: The results returned for the query "Maximum difference between a pair of adjacent elements by excluding every element once".

[*CLS*] and [*Language*] in the Python code, respectively, and selects five tokens with the largest attention scores for visualization. The darker the color of the token block, the higher the attention score.

From Figure 5, we can see that [*CLS*] focuses on the identifiers *is* and *Perfect*, which shows that identifiers such as function name are helpful for understanding the semantics of the code. In addition, we can see from the code snippet that the return value part plays a key role in program judgment, and [*CLS*] effectively captures this information. Furthermore, [*Python*] focuses on the parts related to a specific language, such as Python-specific keywords (*e.g.,def* and *from*), this shows that the Language Specific Viewer method effectively captures language-related information, which helps the model distinguishes between different languages, and better understands the characteristics of different languages. Taken together, these two complementary representations constitute the representation of the source code, improving the model's ability to understand multilingual code.

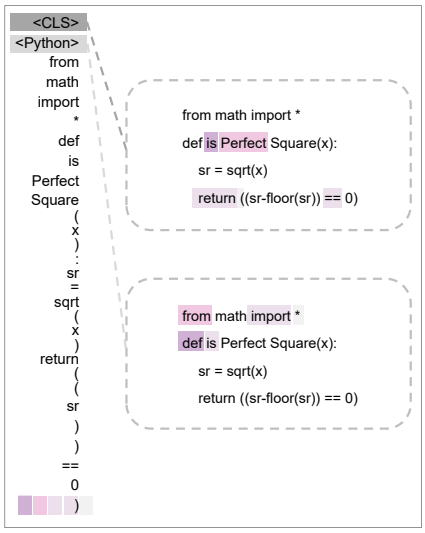

Figure 5: Visualization of the Language Specific Viewer.

5 DISCUSSION

5.1 strength

Our proposed model has three apparent advantages in the learning of source code-related tasks. We will describe them in detail.

5.1.1 A general model for multiple programming languages. Unlike previous work that fine-tunes pre-trained models to obtain multiple monolingual models when handling downstream tasks, our model directly fine-tunes on multilingual data, so it can deal with multiple programming languages without training multiple times. Compared with previous single-language models, our method can effectively reduce the cost of model deployment and maintenance. For example, in natural language code search tasks, Our model can return code results from multiple programming languages for a given natural language query without training and deploying multiple single-language models.

5.1.2 A model with higher performance. Compared to previous models, our method can leverage more training data from multilingual codes to improve model performance. We demonstrate the superiority of our approach on seven different programming languages and two common code-related tasks. Although Ahmed *et al.*[1] proposed polyglotCodeBERT to enhance the performance of downstream tasks by fine-tuning multilingual data, it did not fully explore the characteristics of multilingual data and take advantage of them. Our approach significantly improves the performance of multilingual models through the language-specific viewer method and cross-language contrastive learning methods.

5.1.3 It is easy to extend to other pre-trained models. Our approach enhances the ability of pre-trained models to understand multilingual code by adding language-specific viewers and cross-language code contrastive learning methods. We verified the effectiveness of our method on the CodeBERT pre-trained model, and we will conduct experiments on other pre-trained models in the future. Our method does not modify the structure of the original pre-training model so that it can be easily adapted to other pre-training models. With the development of pre-training model technology, our method can achieve better performance.

Table 6: The same queries returned by MulCS and ChatGPT.

Query	MulCS	ChatGPT
1.Distance between two parallel Planes in 3-D	✓	×
2.Maximum Prefix Sum possible by merging two given arrays	✓	×
3.Nth natural number after removing all numbers consisting of the digit 9	✓	×
4.Count ways to place M objects in distinct partitions of N boxes	×	✓
5.Mode in a stream of integers (running integers)	×	✓
6.Find the sum of the first Nth Icosagonal Numbers	×	✓
7.Find the sum of the first Nth Centered Tridecagonal Numbers	×	✓
8.Check whether a + b = c or not after removing all zeroes from a,b and c	✓	✓
9.Find nth Hermite number	✓	✓
10.Count numbers from range whose prime factors are only 2 and 3	✓	✓

5.2 Threats to Validity

5.2.1 Internal threats. Internal threats mainly come from the hyperparameters of the model. Hyperparameters have a great significance on the performance of deep learning models, but the hyperparameters of LAMCode are not adjusted experimentally but are set according to the default hyperparameters in [46]. Therefore, other hyperparameter settings may yield better results.

5.2.2 External threats. External threats mainly come from datasets and evaluation benchmarks. We implement LAMCode based on an existing dataset and reproduce the state-of-the-art method. We trained and evaluated as many different languages in the existing dataset as possible. The experimental results indicate the effectiveness of our approach, but we cannot guarantee the effectiveness of LAMCode in programming languages other than the seven languages in the dataset. In the future, we will evaluate the effectiveness of our method in other programming languages.

5.3 Exploring Code Search in the Age of Large-Scale Language Models

ChatGPT has strong code understanding and generation capabilities, but it also has certain limitations, such as usually serious nonsense. However, the code search database are verified correct code, so in some ways, it will be more accurate. As shown in the table 6, we randomly selected 10 queries from the test set, ChatGPT could not answer all the questions correctly, and MulCS performed better on some questions. From Figure 6, we can see that for a description "Distance between two parallel planes in 3-D", MulCS searched for the correct code, while ChatGPT generated the wrong code because of the misuse of the formula. Therefore, we argue that the semantic knowledge emphasized in this paper show a promising prospect for deep code representation, especially for further combining with large-scale pre-trained ChatGPT through code search-based auxiliary prompt.

6 RELATED WORK

6.1 Representation Learning for Source Code

Source code representation learning is crucial for the development of code intelligence and has been extensively studied [2, 3, 6, 13, 21, 38, 43, 45]. Allamanis *et al.*[2] did an understandable survey of source code representation. Gu *et al.*[13] combined the embedding

```
void distance(float a1, float b1, float c1,
              float d1,float a2, float b2,
              float c2, float d2){
  float x1, y1, z1, d;
  if(a1/a2==b1/b2 && b1/b2==c1/c2){
    x1=y1=0;
    z1=-d1/c1;
    d=fabs((c2*z1+d2))/
          (sqrt(a2*a2+b2*b2+c2*c2));
    return d;
  }
  else{
    return -1;
  }
}
```

(a) Listing1: The correct result returned by MulCS.

```
void distance(float a1, float b1, float c1,
              float d1,float a2, float b2,
              float c2, float d2){
  float numerator = fabs(d1-d2);
  float denominator = sqrt(a1*a1+b1*b1+c1*c1);
  return numerator / denominator;
}
```

(b) Listing2: The error result returned by ChatGPT(Misuse of math formula).

Figure 6: Different results returned for NL "Distance between two parallel planes in 3-D".

of plain texts and source code to improve code search. Iyer *et al.*[21] combined the attention mechanism with an LSTM network model for code summarization, and Yin *et al.*[43] trained their models on natural language texts and corresponding code from Stack Overflow. They also added the AST structure information of the code as the input to the model. Alon *et al.*[4] used paths in the AST to learn the representation of code. They verified their model in the task of predicting the method name of the code fragment. They also proposed code2seq [3]. The difference from previous work is that this model generates a series of word sequences instead of individual words. Wan *et al.*[38] proposes MMAN to combine multiple semantic information of code, including tokens, AST, and CFG of source code. They achieved excellent results in the code search task. These researches have gradually made full use of multiple views of programming languages. Compared with these works, our work learns code representations from another perspective, *i.e.*, multilingual code representation learning, aiming to improve the performance of the model by utilizing more training data in multiple languages. Experiments show that our method produces state-of-the-art results compared to single-language models and other multilingual models.

6.2 Pre-Trained Models for Programming Languages

In recent years, large-scale pre-trained models have made significant progress in natural language processing [5, 10, 25, 28, 32, 37]. The most representative neural network is Transformer [37], which uses a multi-head attention mechanism for end-to-end learning under an encoder-decoder framework. Inspired by this, pre-trained models for programming languages have been proposed [11, 12, 15, 16, 40, 48] to facilitate the development of code intelligence. feng *et al.*[12] proposed CodeBERT, which is pre-trained on CodeSearchNet [20], a large-scale dataset with six programming languages, and introduces two objectives: masked language modeling and replaced token detection. In addition, to utilize the semantic information of the code, guo *et al.*[16] proposed GraphCodeBERT, which integrates the program's data flow into the model's training process through the method of edge relation prediction pre-trains on multilingual datasets. guo *et al.*[15] present UniXcoder, a unified cross-modal pre-trained model for the programming language that enhances code representation by leveraging cross-modal information such as AST and code comments.

The above models demonstrate the superiority of multilingual datasets in the pre-training stage. Unlike the above models, ahmed *et al.*[1] pointed out that multilingual training is effective not only in the pre-training stage but also in the fine-tuning stage. However, ahmed *et al.*[1] only does the same processing for multilingual data and does not fully capture the characteristics of multilingual code. Although pian *et al.*[31] tried to capture language-specific information using a meta-learning approach, this approach did not adapt to most pre-trained models because it changed the model structure. Compared to these works, our method takes full advantage of multilingual code for training and requires no modification to the model structure.

6.3 Contrastive Learning for Code Representation

Unlike the MLM self-supervised learning approach used in models such as CodeBERT, the contrastive learning approach does not train by predicting labels or reconstructing data. Instead, it learns code representations by closing the distance between similar example (positive) representations while maximizing the distance between different example (negative) representations. It has been widely applied to natural language and image fields. Inspired by this, some works [7, 19, 22, 30] try to apply the contrastive learning method to code representation learning. Bui *et al.*[7] and Jain *et al.*[22] mainly use semantic-preserving program transformations to generate functionally equivalent code snippets and pre-train the model with a contrastive learning technique to identify semantically equivalent (positive examples) and non-equivalent (negative examples) code snippets. Huang *et al.*[19] proposed CoCLR, which incorporates code contrastive learning into the CodeBERT. CoCLR obtains more artificially generated training instances by rewriting query statements, such as randomly deleting a word, randomly swapping, and so on, which significantly improves the code understanding ability of CodeBERT. Ding *et al.*[11] used data augmentation to generate functionally equivalent code as positive samples and injected bugs into code snippets as negative samples, to understand the general

representation of the source code and the specific characteristics of vulnerability and code clone detection.

The above models demonstrate the effectiveness of contrastive learning methods in code representation learning, both in the pre-training and fine-tuning stages. However, when solving downstream tasks, the above approaches use single-language data for fine-tuning, and multi-language data is not fully utilized. Therefore, we propose a cross-language contrastive learning method to better understand the semantic relationship of multilingual code through in-batch negative example augmentation and overlap-reduction positive example augmentation.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new language-aware multilingual code understanding model. Specifically, we propose a simple yet effective method to perceive linguistic information by injecting language-specific viewer into the language models. To enhance the semantic relations of multilingual codes, we present a cross-lingual code contrastive learning framework, which prevents the model from over-reliance on similar identifiers across languages by generating positive examples with Overlap-Reduction Augmentation. We conduct experiments on a large-scale dataset containing seven programming languages and achieve state-of-the-art performance on two tasks. Our further analysis proves the effectiveness of different components in our method.

For future work, we might extend our LAMCode to solve other software engineering problems, *e.g.*, code summarization and code generation. Moreover, we attempt to construct larger-scale multilingual datasets and extend our approach to other programming languages.

ACKNOWLEDGMENTS

This research was funded by NSFC No. 62272473 and No. 62202474.

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. 1443–1455.

[2] 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.

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

[4] 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.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.

[7] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[8] YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning Sequential and Structural Information for Source Code Summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. 2842–2851.

[9] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341* (2019).

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

[11] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6300–6312.

[12] 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. *arXiv preprint arXiv:2002.08155* (2020).

[13] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[14] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. 2022. Cross-Language Binary-Source Code Matching with Intermediate Representations. *arXiv preprint arXiv:2201.07420* (2022).

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[17] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. *arXiv preprint arXiv:2005.06980* (2020).

[18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[19] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239* (2021).

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

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[22] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973* (2020).

[23] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[24] Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the dark secrets of BERT. *arXiv preprint arXiv:1908.08593* (2019).

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[26] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1571–1575.

[27] Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. CodeRetriever: Unimodal and Bimodal Contrastive Learning. *arXiv preprint arXiv:2201.10866* (2022).

[28] 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).

[29] Yue Liu, Wenxuan Tu, Sihang Zhou, Xinwang Liu, Linxuan Song, Xihong Yang, and En Zhu. 2022. Deep Graph Clustering via Dual Correlation Reduction. In *Proc. of AAAI*.

[30] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs?. In *International Conference on Machine Learning*. PMLR, 8476–8486.

[31] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning. *arXiv preprint arXiv:2206.06460* (2022).

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 140 (2020), 1–67.

[33] Ensheng Shi, Wenchao Gub, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation. *arXiv preprint arXiv:2204.03293* (2022).

[34] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987* (2021).

[35] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[36] Aaron Van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv e-prints* (2018), arXiv–1807.

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

[38] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[39] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture?–A Structural Analysis of Pre-Trained Language Models for Source Code. *arXiv preprint arXiv:2202.06840* (2022).

[40] Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. 2022. CODE-MVP: Learning to Represent Source Code from Multiple Views with Contrastive Pre-Training. *arXiv preprint arXiv:2205.02029* (2022).

[41] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[42] Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3733–3742.

[43] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 476–486.

[44] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph contrastive learning with augmentations. *Advances in Neural Information Processing Systems* 33 (2020), 5812–5823.

[45] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2021. degraphcs: Embedding variable-based flow graph for neural code search. *ACM Transactions on Software Engineering and Methodology* (2021).

[46] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. *arXiv preprint arXiv:2206.08474* (2022).

[47] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: an overlapping-aware code retriever. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 883–894.

[48] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318* (2021).