Article

Knowledge Distillation-Based Multilingual Code Retrieval

Wen Li, Junfei Xu and Qi Chen *

College of Computer Science and Technology, Zhejiang University, Hangzhou 310013, China;
leevanleevan@163.com (W.L.); xujunfei@zju.edu.cn (J.X.)
* Correspondence: chenqi@zju.edu.cn

Abstract: Semantic code retrieval is the task of retrieving relevant codes based on natural language queries. Although it is related to other information retrieval tasks, it needs to bridge the gaps between the language used in the code (which is usually syntax-specific and logic-specific) and the natural language which is more suitable for describing ambiguous concepts and ideas. Existing approaches study code retrieval in a natural language for a specific programming language, however it is unwieldy and often requires a large amount of corpus for each language when dealing with multilingual scenarios.Using knowledge distillation of six existing monolingual Teacher Models to train one Student Model – MPLCS (Multi-Programming Language Code Search), this paper proposed a method to support multi-programing language code search tasks. MPLCS has the ability to incorporate multiple languages into one model with low corpus requirements. MPLCS can study the commonality between different programming languages and improve the recall accuracy for small dataset code languages. As for Ruby used in this paper, MPLCS improved its MRR score by 20 to 25%. In addition, MPLCS can compensate the low recall accuracy of monolingual models when perform language retrieval work on other programming languages. And in some cases, MPLCS' recall accuracy can even outperform the recall accuracy of monolingual models when they perform language retrieval work on themselves.

Keywords: multilingualities; code search; knowledge distillation

check for updates

Citation: Li, W.; Xu, J.; Chen Q. Knowledge Distillation-Based Multilingual Code Retrieval. *Algorithms* **2022**, *15*, 25. https://doi.org/10.3390/a15010025

Academic Editor: Marcos Zampieri

Received: 30 November 2021
Accepted: 12 January 2022
Published: 17 January 2022

1. Introduction

The research on code retrieval can be divided into two broad categories according to the methods used: Information Retrieval-Based Methods and Deep Learning Model-Based Methods. Information Retrieval-Based Methods are more based on traditional search methods, the main idea is to improve work based on text similarity, and to perform code retrieval through search techniques combined with code features. For example, Luan et al. [1] proposed a structured code recommendation tool called Aroma, which implements a method of searching code by using coding. They divide the retrieval process into two stages: stage 1, perform a small range lightweight search, and then stage 2, a further in-depth search based on the previous results searched in stage 1. On top of this, Lv et al. [2] discovered a way to better connect the characteristics of code by focusing on the API calls in coding. Standard APIs have specific functions and corresponding documentation descriptions, this enables them to turn code search tasks into simple similarity matches between natural language descriptions and API documentation. However, these information retrieval-based methods cannot uncover the deep connection between natural language and code language, this leads to the lack of accuracy in methods. With the rapid development of NLP, some scholars start using deep learning models to solve the problem of code retrieval to prove its accuracy.

The main strategy of the Deep Learning-Based Approach is to use a neural network approach to map code snippets and query statements in the same vector space. Husain et al. [3] came up with a plain and typical example to propose a basic framework where they simply consider the code as a text, encode it using several common methods for text embedding, and map the code and description to the same high-dimensional space for learning. Not long after, Gu et al. [4] went further in this direction, by not only considering the code text as features, but also function names and API (Application Programming Interface) sequences. After that, many scholars discovered more code-specific features to build new models. Haldar et al. [5] added both tokens and AST (Abstract Syntax Tree) information on the one hand, and improved join embedding, on the other hand, they argue that only calculating similarity in the last step (i.e., only the overall similarity is considered, not the local similarity) would cause information loss, so they fused both CAT (An AST-Based Model) and MP (A Multi-Perspective Model) methods and proposed the MP-CAT Model. Sachdev et al. [6] address an unsupervised learning task by proposing Neural Code Search (NCS), using features such as function names, function calls, enumerated quantities, string literals, annotations, TF-IDF weights, and word vectors to construct high-dimensional vectors for retrieval. Cambronero et al. [7] made improvements based on the NCS algorithm and proposed an improved idea of UNIF by adding a solution to the unsupervised learning algorithm NCS to improve model performance. With this model, a small number of supervised samples, can be comparable to some supervised learning algorithms. Meanwhile, some scholars are researching other issues related to code retrieval, for example, Yin and Neubig[8] investigated the task of code generation and argued that the current approach is to view it as a seq2seq generation task, but does not take into account that the code language has a specific syntactic structure. From this, an AST tree generation by natural language is proposed, and tools are used to convert the AST tree into code. Analogous to large pre-training models such as ELMo, GPT, BERT, and XLNet, Feng et al. [9] and Kanade et al. [10] proposed the pre-training models with codes studied.

When modern software engineers develop a software product, it often requires more than just one programming language, thus developers are faced with the need to search for multiple code languages during the development process. A recent survey of open-source projects has shown that the use of multiple languages is rather universal, with a mean number of 5 languages used per project[11]. Thus, multilanguage software development (MLSD) seems to be common, at least in the open-source world[12]. As mentioned above, Both the information retrieval-based approaches and the deep learning-based approaches deal with the mapping from a single natural language to a single programming language. Thus this paper proposed the idea of mapping a single natural language to a multi-programming language, which is new in this field. The goal that we want to achieve here is, when a query of natural language is inputted, we were able to find multiple programming languages (such as Java, PHP, Go, etc.) codes that have the functionality that matches the natural language description. And this is done with the use of knowledge distillation.

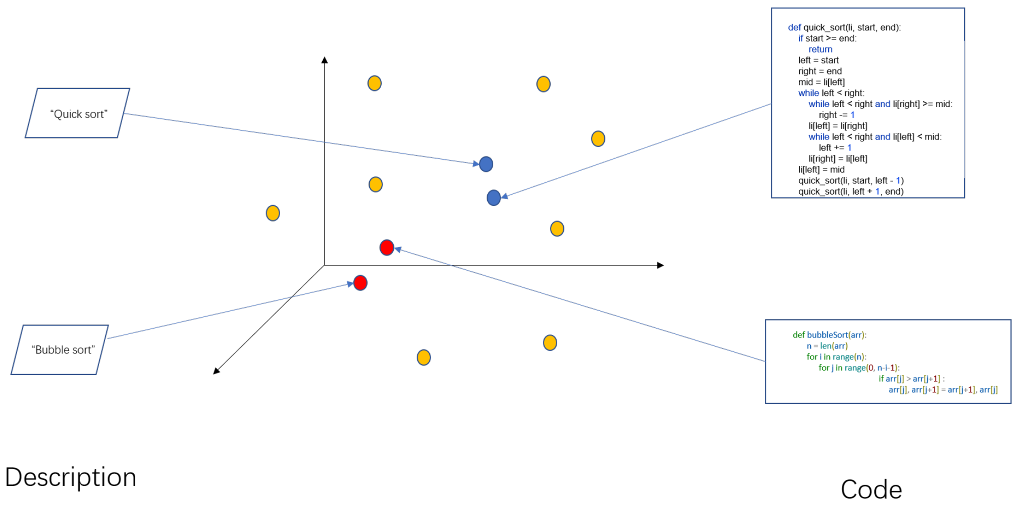
Hinton et al. [13] proposed the concept of knowledge distillation, the core idea of knowledge distillation is to first train a complex model (known as the Teacher Model) and then use the output or intermediate state of this model to train a smaller model (known as the Student Model). The main contribution of knowledge distillation is model compression, which has been widely studied and utilized in many areas of deep learning, such as natural language processing, speech recognition, and computer vision. This technique is also used for natural language translation tasks. Many scholars have done a lot of exploratory work on multilingual translation models [14–17], and NMT-based multilingual translation models have been discovered. Xu et al. [18] proposed to transfer knowledge from individual models to multilingual models using knowledge distillation, which is commonly used for studying model compression and knowledge migration and mostly fits quite well with the multilingual code search environment. A large and deep Teacher Model (or an ensemble of multiple models) is usually trained first, and then a smaller and shallower Student Model is trained to mimic the behavior of the Teacher Model. The Student Model can approach or even outperform the accuracy of the complex Teacher Model by knowledge distillation. This paper uses knowledge distillation techniques to fuse six pre-trained monolingual models into one student model. By doing so, the size of the model is reduced significantly. At the same time, the student model's performance is almost the same with each teacher model on the test set, on cases like Ruby and JavaScript, the student model even outperformed the teacher model. After redoing the experiments on different code and query encoders, we confirm that this method can be used on a wide range of encoders.

We summarize our contributions as follows:

- We propose a code search model that efficiently and accurately addresses multi-programming language fusion. A single model can solve the problem of searching for multiple programming languages.
- Compared to multiple models, our model has fewer parameters. Also, the data set requirements are lower because the data sets are complementary between different languages.
- The ability to uncover connections between different programming languages makes the model highly extensible, and this provides some support for languages with relatively small corpora.

Background

Joint Vector Representations, also known as Multimodal Embedding [19], are very common in code retrieval tasks and most deep-learning-based methods use this. Joint Vector Representations is a method to learn the connection between two heterogeneous structures, which maps data of two different structures into the same high-dimensional space [20], so that the corresponding data fall as close as possible to each other in the high-dimensional space, while making the non-corresponding data as far away from each other as possible. Such an approach also makes the query process more intuitive, and when performing a search, it is only necessary to find some points in the high-dimensional space that are closest to the target point, i.e., the nearest neighbor problem in the high-dimensional space.



```
def quick_sort(li, start, end):
    if start >= end:
        return
    left = start
    right = end
    mid = li[left]
    while left < right:
        while left < right and li[right] >= mid:
            right -= 1
        li[left] = li[right]
        while left < right and li[left] < mid:
            left += 1
        li[right] = li[left]
    li[left] = mid
    quick_sort(li, start, left - 1)
    quick_sort(li, left + 1, end)
```

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
                    if arr[j] > arr[j+1] :
                        arr[j], arr[j+1] = arr[j+1], arr[j]
```

Figure 1. Conceptual diagram of joint embedding in a code search task.

This paper used joint embedding to learn the relevance between natural language description and code. As shown in Figure 1, code segments and natural language representations are mapped to the same high-dimensional space. The code for bubble sort and "bubble sort" is mapped to relatively close locations, as is the case for "quick sort".

Paper structure

The remainder of the paper is organized as follows. Section 2 presents our proposed framework, this includes the teacher model network structure and the learning process of distilling knowledge using models to train student model. Section 3 describes the experimental setup and details. Section 4 presents an analysis of the experimental results. Section 5 concludes the paper.

2. Multi-Programming Language Code Search

Inspired by multilingual translation models, we propose a novel deep learning model, MPLCS, to solve the task of multi-programming language code search. Each programming language has its own syntactic structure and thus programming languages are heterogeneous from one another. For the heterogeneous problem, the method of joint embedding mentioned above is used to map them into the same high-dimensional space, and the semantic similarity of each heterogeneous data is measured by the similarity degree. For the problem of multi-model fusion, the solution we adopt is to use knowledge distillation to handle it.

2.1. Overview

There are two main parts, one for training the Teacher Model and one for training the Student Model. We will be discussing the network structure of the Teacher model in Section 2.2. As for the Student Model, it has the same structure of the Teacher Model and can take corresponding Teacher Model as input, then fuse the properties of different language models. The components are elaborated in Section 2.3.

2.2. Teacher Model

We follow Hamel et al's study and use the same monolingual model structure, i.e., 1dCNN, NBOW, and self-attention, these are commonly used methods based on token sequences. In this paper, we will only introduce the model structure of self-attention and the subsequent experiments will be based on this model, since it performed best among the 3 methods. We embedded the code in the same way as embedding the natural language—an encoding of their token sequences with an added attention vector. In general, the model needs to be trained for the following parts as shown in Figure 2: code_vocab and query_vocab of the code and the description of the embedding layers, the fully connected layers of the corresponding code and description, and the attention vector.

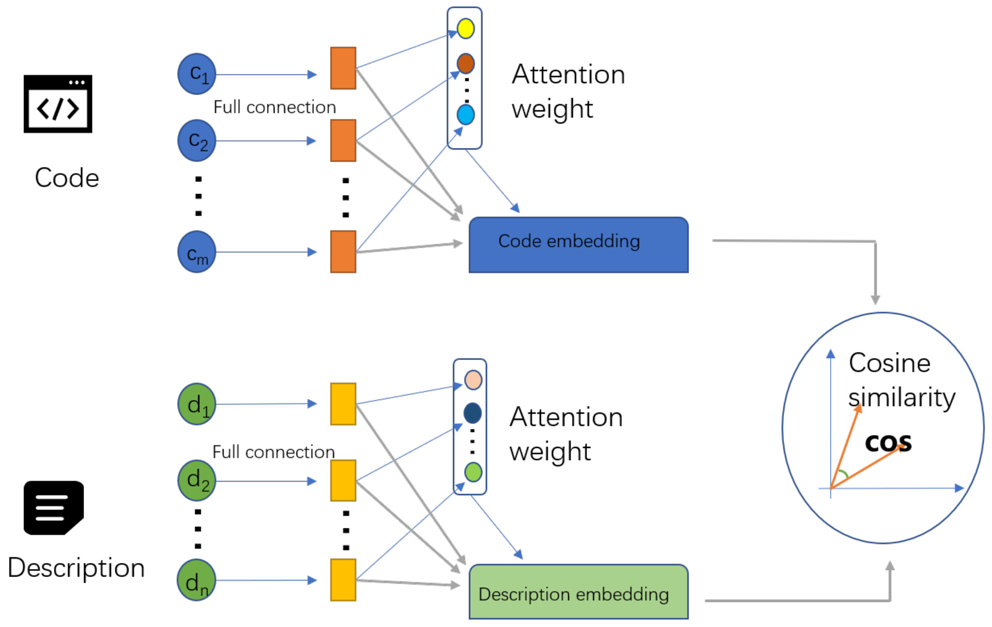


Figure 2. Details in Teacher Models whose encoder is self-attentions.

Here, we define two embedding lexicons: code_vocab and query_vocab, each line corresponds to a specific code token or objects of the description token:

$$code\_vocab \in \mathbb{R}^{|X| \times d} \tag{1}$$

$$query\_vocab \in \mathbb{R}^{|Q| \times d} \tag{2}$$

X is the set of code token dictionaries, Q is the set of natural language description dictionaries, and d is the dimension of the embedding, which we set to 128 in our experiments. Finding the embedding of a code or query is a simple matter of finding the corresponding line.

For a line of code $C = ctoken_1, ctoken_2, , ctoken_n$ and its corresponding natural language description $Q = qtoken_1, qtoken_2, \cdots, qtoken_m$,with the $ctoken_i(i = 1, \cdots, n)$ as code token and $qtoken_i(i = 1, \cdots, m)$ as description token. After embedding (random initialization) we can obtain:

$$\begin{aligned} c_i &= embedding(ctoken_i), \\ q_i &= embedding(qtoken_i) \in \mathbb{R}^d \end{aligned} \tag{3}$$

And after going through a full connected layer, we have:

$$\tilde{c}_i = tanh(W_c \cdot c_i), \tilde{q}_i = tanh(W_q \cdot q_i) \tag{4}$$

In which $W_c, W_q \in \mathbb{R}^{d \times d}$, $tanh$ is the hyperbolic tangent function, which is a common monotonous nonlinear activation function, taking the value from $(-1, 1)$.

Finally, we use the attention vector to aggregate these token vectors, it is essentially a weighted average aggregation. The main process is to calculate the weight of each token in the current block of code or natural language sentence, in terms of goals, we naturally hope that the token that can represent the code or sentence will occupy a greater weight, and this is where "attention" comes into play. In the beginning, the attention vector $a \in \mathbb{R}^d$ will be randomly initialized and study the model simultaneously during the training process. The weight of each token is calculated by dotting the vector of the token with the attention vector and then normalizing it to ensure that the weights sum up to 1, the weights of each corresponding code token vector $\tilde{c}_i$ are calculated as follow:

$$\alpha_i = \frac{exp(\tilde{c}_i^T \cdot a)}{\sum_{j=1}^{n} exp(\tilde{c}_j^T \cdot a)} \tag{5}$$

The purpose of using exp is to ensure that the weights are positive, as in the standard softmax function, and we divide by the sum of all terms to ensure that the sum of the weights is 1, the calculation of natural language description token vector follows a similar pattern. Once $\alpha_i(i = 1, \cdots, n)$ is calculated, the final code vector can be obtained by summing the linear weights of the code token vectors $\{\tilde{c}_1, \tilde{c}_2, \cdots, \tilde{c}_n\}$. Code Vector represents the entire code segment, and it's expressed as follow:

$$v_c = \sum_{i=1}^{n} \alpha_i \cdot \tilde{c}_i \tag{6}$$

Similarity Model

After obtaining the code vector *wait* and the description vector, we want the code and description vectors to be co-embedding, so we measure the similarity between these two vectors. We measure this by using the cosine similarity formula, which is defined as

$$\cos(v_c, v_q) = \frac{v_c^T \cdot v_q}{\|v_c\| \|v_q\|} \tag{7}$$

The higher the similarity, the higher the correlation between the code vector and the description vector. In summary, the MPLCS model takes a pair of $\langle$codes, descriptions$\rangle$ as input, and calculates their cosine similarity to measure the strength of their correlation.

Teacher Model Learning

Contrastive representation learning is often used for code retrieval tasks [4,5,21–23], and our experiment will use the contrastive loss function as the loss function of the model.We now describe how to train the MPLCS model in two stages, the first stage is to train the six Teacher Models. Both codes and descriptions are embedded into a unified vector space. The ultimate goal of joint embedding is that if a code fragment and a description have similar semantics, their embedding vectors should be close to each other. In other words, given an arbitrary code fragment C and an arbitrary description D, we want it to predict high similarity (close to 1) if D is the correct description of C, otherwise, it will only little similarity (close to 0).

Therefore, we need to use negative samples to construct our loss function. Essentially we consider this problem: we construct each training instance as a triad $\langle C, D+, D-\rangle$: for each code fragment C, there is a positive description $D+$ (the correct description of C) and a negative description $D-$ (the wrong description of C) chosen randomly from a pool of all $D+$(negative samples are derived from positive samples). When trained on the $\langle C, D+, D-\rangle$ set, MPLCS predicts the cosine similarity of the $\langle C, D+\rangle$ and $\langle C, D-\rangle$ pairs and minimizes the rank loss, that is, minimizing the following equation.

$$Loss_{teacher} = \max(0, 1 - \cos(c, d+) + \cos(c, d-)) \tag{8}$$

In this formula, $d+$ represents positive description, $d-$ represents negative description. In practice, we use the strategy of obtaining the cosine between two of the N code vectors and the N corresponding description vectors. This gives us an $N * N$ matrix, with positive sample values on the main diagonal and negative sample values everywhere else. We want the positive sample value to be as large as possible and the negative sample value to be as small as possible, so we subtract the value on each diagonal by 1, and the main goal is to make all values as small as possible. The formula is described as follows:

$$\begin{aligned} submax(X, i) &= \max_{\substack{j \neq i \\ 1 \leq j \leq n}} X[i, j] \\ Loss(X) &= \frac{1}{n} \sum_{i=1}^{n} ((1 - X[i, j]) + submax(X, i)) \end{aligned} \tag{9}$$

2.3. Student Model

After training the six Teacher Models, we begin to train the Student Model. Every language input will obtain two sets of vectors during the encoding step, one set constructed by the Student Model and the other one constructed by the Teacher Model for the corresponding language, as shown in Figure 3. The composition of the loss function consists of three parts: (1) STUDENT's code vector and STUDENT's description vector, (2) the code vector of the TEACHER and the description vector of the STUDENT, (3) the code vector of the STUDENT and the description vector of the TEACHER.

$$Loss_{student-self} = \max(0, 1 - \cos(c_{student}, d_{student}+) + \cos(c_{student}, d_{student}-)) \tag{10}$$

$$Loss_{KD} = \sum_{teacher_i \in Teacher} \begin{pmatrix} \max(0, 1 - \cos(c_{student}, d_{student}+) + \cos(c_{student}, d_{student}-)) + \\ \max(0, 1 - \cos(c_{student}, d_{teacher}+) + \cos(c_{student}, d_{teacher}-)) \end{pmatrix} \tag{11}$$

$$Loss_{student-ALL} = (1 - \lambda) Loss_{student-self} + \lambda Loss_{KD} \tag{12}$$

In the formula, *Teacher* is the set of Teacher Models, which is the set of six Teacher Models in this paper. For each of the Teacher Models, two additional sets of loss functions are computed, as shown in $Loss_{KD}$, we replace code vector to the loss function in Teacher Model's and replace description vector to Teacher Models'. The parameter λ is used to adjust the contribution of the teacher model in the student model. We explore the effect of this parameter on Teacher Models in our experiment. Even the two parts of $Loss_{KD}$ can be scaled differently to serve as a focus for one part of the work, and this part of the work can be further elaborated for future studies.

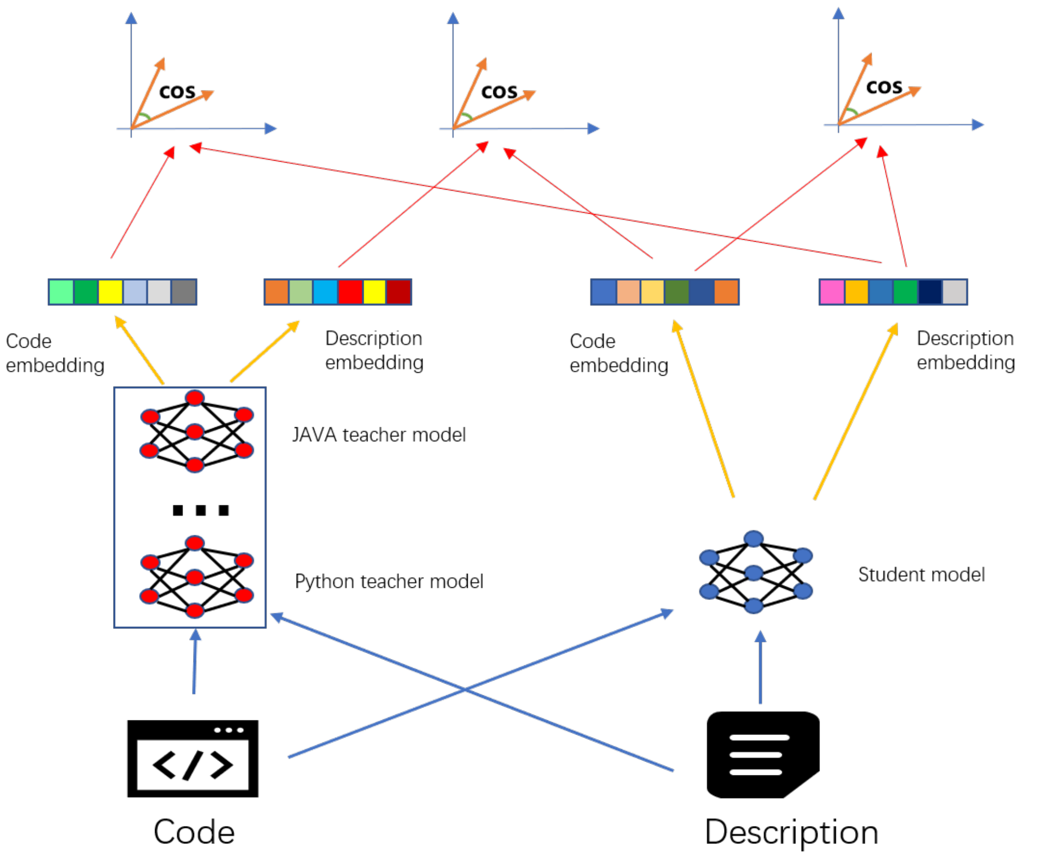


Figure 3. Schematic diagram of the overall model.

The training process is shown in Algorithm 1. L is the number of language varieties, which is taken as 6 in this paper, $l \in [L]$ denotes the language number, D^l denotes the training set for the language with the number l, θ_M represents the parameter of the multi-lingual Student Model, the corresponding $\theta^l_{teacher}$ represents a parameter of Teacher Model for the language with the number l. Our algorithm takes the pre-trained Teacher Models as input. It is important to note that the training set for the Teacher Model can either be shared with the Student training set for that language or choose a separate training set. Similarly, the structure of the student network model can be set to be the same or different from that of the teacher network model. For convenience, the same data set and network model structure are chosen in this paper. Notice that lines 7–10 of Algorithm 1 made a loss function selection. This is based on the strategy that: if the Student Model is already performing better than the Teacher Model for a particular language, the Teacher Model will not be introduced, but this setting is not fixed, and the accuracy of this language may be reduced later when training other languages, that is then the Teacher Model will be reintroduced. The setting of whether or not to introduce a Teacher Model involves parameter τ, as described in lines 14–22, where the accuracy of the Student Model is higher than that of the Teacher Model, then the Teacher Model is not introduced.

Algorithm 1 Knowledge distillation in multiple code languages

```
Input: training set $\{D^l\}_{l=1}^{L}$, trained Teacher Models for L languages $\{\theta^l_{teacher}\}_{l=1}^{L}$, learning rate $\eta$,total number of training steps $\tau$ ,distillation inspection step length $\tau_{check}$, distillation accuracy threshold $\tau$
Output: The trained Student Model
1:  Randomly initialized Student Model parameters $\theta_M$, current step count set to $T = 0$, cumulative gradient $g = \mathbf{0}$, For each Teacher Model, mark $f^l = True, l \in [L]$
2:  while $T < \mathcal{T}$ do
3:      $T = T + 1$
4:      $g = \mathbf{0}$
5:      for $l \in [L]$ do
6:          Randomly select a batch of data $(\mathbf{c}^l, \mathbf{d}^l)$ from the training set $D^l$
7:          if $f^l == True$ then
8:              Calculating gradient on loss function, $g+ = \partial Loss_{student-ALL} / \partial\theta_M$
9:          else
10:             Calculating gradient on loss function, $g+ = \partial Loss_{student-self} / \partial\theta_M$
11:         end if
12:     end for
13:     Update model parameter : $\theta_M - = \eta * g$
14:     if $T \% \mathcal{T}_{check} == 0$ then
15:         for $l \in [L]$ do
16:             if $Accuracy(\theta_M) < Accuracy(\theta^l_{teacher}) + \tau$ then
17:                 $f^l = True$
18:             else
19:                 $f^l = False$
20:             end if
21:         end for
22:     end if
23: end while
```

3. Experiments

3.1. Data Preparation

The experimental data was selected from the publicly available dataset collected by Hamel et al. [3]. They collected corpus from publicly available open-source GitHub repositories, and to weed out a portion of low-quality project code, libraries.io was used to identify all projects that were quoted by at least one other project, and were ranked by the number of stars and forks indicated by "popularity" ranking. The statistic information of the database is listed in Table 1. Data set is divided into training set, validation set, and test set according to an 80:10:10 ratio.

However, the obtained data through the corpus cleaning is still unsatisfied. First of all, function annotation is essentially different from inquired sentences, so the format of language is not the same.

Table 1. Sample size.

	Number of Functions
Java	542,991
Go	347,789
PHP	717,313
Python	503,502
JavaScript	157,988
Ruby	57,393
Total	2,326,976

Code and annotations are often written by the same author at the same time and therefore they appear to be the same vocabulary, unlike search queries which cover many different terms. Secondly, despite we put enough effect on data cleaning, the extent to which each annotation accurately describes its relevant code fragment is still uncertain. For example, some annotations are obsolete in terms of the code they describe or the object of the comment is a localized part that the author wants to focus on rather than the whole function. Finally, we are aware of some annotations are written in other languages such as Japanese and Russian, and that our evaluation dataset focuses on English queries. In order to address this issue, some scholars have chosen to add some other conditional features to strengthen the characteristics of samples. When collecting and sorting corpus, they tend to select previously available code and corresponding descriptive annotations, and in addition, they also collect relevant query information, such as gathering asked questions from Stack Overflow and attracting the code and corresponding annotations from the answers, this method can help to propose some models that make good use of this new information, whereas this type of data is not mainstream (most codes do not have query information) For example, a company's internal code does not have relevant query questions, so this paper is still experimenting with the original dataset.

3.2. Vocabulary

For a fixed-size dictionary, the traditional tokenization based technique of simply segmenting the text with spaces and symbols has many drawbacks, such as the inability to deal well with unknown or rare words (the OOV out-of-vocabulary problem); the nature of the language's own lexical construction, and the difficulty of learning the root word associations with the traditional method. This leads to the lack of generalization ability with the traditional approach. Byte-Pair-Encoding (BPE) is a method for solving such issues. unknown or rare words can be classified as unregistered words. Unregistered is known as words that do not appear in the training corpus, but appear in the test corpus. When we work with NLP tasks, we usually generate a vocabulary list (dictionary) based on the corpus, for the words in the corpus that have a frequency greater than a certain threshold, they will be put into the dictionary, and encode all words below that threshold as "#UNK". The advantage of this approach is its simplicity, but the problem is that it's difficult for our model to handle unregistered words if they appear in the test corpus. Usually, our dictionaries are word-level, meaning that they are based on words or phrases, but this inevitably leads to the problem of unregistered words, because it is impossible to design a very large dictionary that covers all words. In addition, another type of lexicon is character-level, which is to design a lexicon with a single letter or Chinese character as the basic word. This approach can theoretically solve the problem of unregistered words because all words are composed of letters, but the disadvantage of this is that the model granularity is too fine and lacks semantic information. Rico et al. [24] proposed a sub-word based approach to generate lexicon, which combines the advantages of word-level and character-level by learning the substrings of characters with high frequency in all words from the corpus and then merging these substrings of characters with high frequency into a lexicon, this dictionary contains both word-level substrings and character-level substrings. We used the BPE technique to generate both query and code vocabulary.

3.3. Evaluation

Mean Reciprocal Rank (MRR)

MRR is commonly used in the recommended system as an evaluation metric. It evaluates the performance of the retrieval system by using the ranking of the correct retrieval results in retrieval results. The formula is as follows.

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{rank_i} \tag{13}$$

SuccessRate@k

SuccessRate@k is a common metric to evaluate whether an approach can retrieve the correct answer in the top k returning results. It is widely used by many studies on the code search task. The metric is calculated as follows:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(Rank_q \leqslant k) \tag{14}$$

where Q denotes the set of queries, $Rank_q$ denotes the highest rank of the hit snippets in the returned snippet list for the query; $\delta()$ denotes an indicator function that returns 1 if the Rank of the qth query ($Rank_q$) is smaller than k otherwise returns 0. SuccessRate@k is important because a better code search engine should allow developers to find the desired snippet by inspecting fewer results.

3.4. Experiment Setup

3.4.1. Data Pre-Processing

First, we tokenize the code and description of the training set for all six languages and use the BPE method to construct a code dictionary and a description dictionary, the size of both dictionaries is set to 30,000. Then, codes and descriptions in the dataset are then transformed into index sequences, the code length is set to 200, the description length is set to 30, and a pad operation is used if the length is insufficient.

3.4.2. Teacher Model Training

We shuffle the training set, the batch size is set to 512, the epoch limit is set to 500, the optimization algorithm is Adam, and the learning rate is 0.1. Early Stop mechanism is used during the training process, i.e., setting a tolerance value of patience = 5, when a model trained on one epoch performed better on the validation set than a model trained on the next 5 epochs, stop training and save the model corresponding to this epoch. After training for each language, six teacher models are obtained. In addition, in order to establish that the method of knowledge distillation is indeed effective, We also set up a dataset that fused six languages together to train a model, which is shown by the ALL rows in Tables 2 and 3.

3.4.3. Student Model Training

The network structure of the student model is consistent with that of the teacher model. The results of the teacher models' encoder are used to guide the student model during the training process. The details are described in Algorithm 1 in the previous section.

3.4.4. Eevaluation Setting

The evaluation method for the teacher model and the student model was described in the previous section. The batch size for both MRR and SuccessRate@k is set to 1000, that is $|Q| = 1000, k = 1, 5, 10$ for SuccessRate@k.

3.4.5. Lambda Parameter Exploration

In order to find out the magnitude of impact the teacher model has on the student model, the parameters in Equation (12) were set to different values, to explore the effect of different weightings of the teacher model on the student model. As the results under different encoders show a similar pattern, We explore the problem of λ only for the teacher model and student model that both code and query encoder are self-attention.

3.4.6. Experiment Equipment

The equipment uses 3 RTX 1080Ti 11GB, and the training time per epoch is around 200–400 s. The number of epochs for a teacher model or student model is around 20–40.

Table 2. MRR for each monolingual model and MPLCS model on different language test sets.

CODE ENCODE	QUERY ENCODE	Model	Go	Java	Javascript	Php	Python	Ruby
SELF-ATT	SELF-ATT	GO	**0.7756**	0.5400	0.4591	0.4552	0.5649	0.4760
		Java	0.6485	**0.6632**	0.4806	0.5390	0.6047	0.5157
		Javascript	0.5688	0.5187	0.5304	0.4494	0.5719	0.4816
		Php	0.6432	0.6005	0.5068	**0.6424**	0.6915	0.5572
		Python	0.6397	0.5691	0.4968	0.5602	**0.7613**	0.5791
		Ruby	0.4849	0.4319	0.3494	0.3673	0.5167	0.4773
		ALL	0.7356	0.6350	0.5240	0.6191	0.7177	0.5717
		MPLCS	0.7472	0.6404	**0.5492**	0.6079	0.7289	**0.5977**
CNN	CNN	GO	**0.7780**	0.5593	0.4767	0.4872	0.6002	0.4943
		Java	0.6691	**0.6776**	0.5106	0.5644	0.6446	0.5381
		Javascript	0.6038	0.5510	0.5546	0.4822	0.6148	0.5122
		Php	0.6718	0.6181	0.5257	**0.6539**	0.7152	0.5707
		Python	0.6783	0.5994	0.5140	0.5658	**0.7748**	0.5918
		Ruby	0.5628	0.4738	0.3937	0.4208	0.5802	0.5239
		ALL	0.7405	0.6458	0.5363	0.6268	0.7301	0.5805
		MPLCS	0.7451	0.6531	**0.5656**	0.6217	0.7457	**0.6111**
NBOW	NBOW	GO	**0.6777**	0.5181	0.4256	0.4081	0.5280	0.4420
		Java	0.5408	**0.5981**	0.4354	0.4456	0.5414	0.4590
		Javascript	0.5312	0.4844	0.4799	0.4118	0.5031	0.4087
		Php	0.5645	0.5359	0.4442	**0.5569**	0.5720	0.4727
		Python	0.5746	0.5212	0.4334	0.4499	**0.6560**	0.4987
		Ruby	0.4645	0.4248	0.3465	0.3465	0.4977	0.4539
		ALL	0.6466	0.5660	0.4602	0.5251	0.6117	0.4911
		MPLCS	0.6710	0.5882	**0.5024**	0.5283	0.6389	**0.5369**
SELF-ATT	NBOW	GO	**0.7599**	0.5315	0.4594	0.4379	0.5549	0.4647
		Java	0.6392	**0.6571**	0.4849	0.5423	0.6021	0.5110
		Javascript	0.5754	0.5136	0.5354	0.4740	0.5600	0.4687
		Php	0.6391	0.5899	0.4956	**0.6424**	0.6678	0.5436
		Python	0.6340	0.5646	0.4944	0.5473	**0.7563**	0.5646
		Ruby	0.4840	0.4202	0.3476	0.3446	0.5072	0.4704
		ALL	0.7266	0.6291	0.5271	0.6175	0.7127	0.5661
		MPLCS	0.7403	0.6429	**0.5604**	0.6148	0.7356	**0.6022**

Table 3. SuccessRate@k for each monolingual model and MPLCS model on different language test sets.

SuccessRate@k	CODE ENCODE	QUERY ENCODE	Model	Go	Java	Javascript	Php	Python	Ruby
SuccessRate@1	SELF-ATT	SELF-ATT	Go	**0.7233**	0.4535	0.3750	0.3615	0.4645	0.3740
			Java	0.5629	**0.5859**	0.3980	0.4496	0.5064	0.4150
			Javascript	0.4828	0.4329	0.4400	0.3614	0.4715	0.3755
			Php	0.5630	0.5208	0.4260	**0.5645**	0.6017	0.4555
			Python	0.5594	0.4851	0.4158	0.4758	**0.6781**	0.4785
			Ruby	0.3886	0.3428	0.2648	0.2787	0.4156	0.3710
			ALL	0.6709	0.5542	0.4362	0.5362	0.6255	0.4700
			MPLCS	0.6800	0.5643	**0.4670**	0.5295	0.6434	**0.5035**
	CNN	CNN	Go	**0.7238**	0.4730	0.3891	0.3889	0.4965	0.3830
			Java	0.5862	**0.5998**	0.4229	0.4757	0.5453	0.4365
			Javascript	0.5154	0.4616	0.4627	0.3925	0.5120	0.4045
			Php	0.5933	0.5389	0.4434	**0.5743**	0.6240	0.4650
			Python	0.6006	0.5144	0.4301	0.4782	**0.6915**	0.4890
			Ruby	0.4671	0.3840	0.3032	0.3293	0.4794	0.4210
			ALL	0.6758	0.5651	0.4461	0.5431	0.6376	0.4790
			MPLCS	0.6801	0.5685	**0.4723**	0.5331	0.6510	**0.5075**

Table 3. *Cont.*

SuccessRate@k	CODE ENCODE	QUERY ENCODE	Model	Go	Java	Javascript	Php	Python	Ruby
	NBOW	NBOW	Go	**0.5934**	0.4268	0.3320	0.3156	0.4254	0.3385
			Java	0.4399	**0.5072**	0.3412	0.3480	0.4392	0.3520
			Javascript	0.4359	0.3929	0.3825	0.3177	0.4018	0.3055
			Php	0.4694	0.4453	0.3528	**0.4630**	0.4686	0.3680
			Python	0.4796	0.4297	0.3397	0.3537	**0.5537**	0.3945
			Ruby	0.3673	0.3347	0.2592	0.2577	0.3955	0.3465
			ALL	0.5586	0.4728	0.3627	0.4286	0.5067	0.3825
			MPLCS	0.5849	0.4935	**0.3973**	0.4287	0.5312	**0.4260**
	SELF-ATT	NBOW	Go	**0.7021**	0.4425	0.3718	0.3429	0.4530	0.3620
			Java	0.5532	**0.5791**	0.3992	0.4543	0.5011	0.4015
			Javascript	0.4866	0.4262	0.4445	0.3853	0.4566	0.3595
			Php	0.5588	0.5095	0.4128	**0.5645**	0.5745	0.4430
			Python	0.5529	0.4792	0.4103	0.4589	**0.6713**	0.4630
			Ruby	0.3879	0.3312	0.2630	0.2560	0.4072	0.3635
			ALL	0.6582	0.5473	0.4378	0.5331	0.6201	0.4625
			MPLCS	0.6735	0.5600	**0.4703**	0.5271	0.6415	**0.4945**
SuccessRate@5	SELF-ATT	SELF-ATT	Go	**0.8302**	0.6418	0.5535	0.5626	0.6850	0.5980
			Java	0.7491	**0.7567**	0.5762	0.6450	0.7215	0.6335
			Javascript	0.6710	0.6180	0.6370	0.5499	0.6898	0.6075
			Php	0.7388	0.6947	0.6015	**0.7356**	0.8002	0.6770
			Python	0.7373	0.6684	0.5870	0.6612	**0.8639**	0.7000
			Ruby	0.5925	0.5303	0.4407	0.4661	0.6343	0.6070
			ALL	0.8066	0.7327	0.6245	0.7179	0.8319	0.6855
			MPLCS	0.8171	0.7441	**0.6617**	0.7214	0.8524	**0.7290**
	CNN	CNN	Go	**0.8354**	0.6617	0.5778	0.6041	0.7253	0.6290
			Java	0.7685	**0.7715**	0.6113	0.6696	0.7648	0.6600
			Javascript	0.7083	0.6562	0.6643	0.5854	0.7383	0.6380
			Php	0.7634	0.7115	0.6189	**0.7493**	0.8279	0.6925
			Python	0.7701	0.7015	0.6123	0.6709	**0.8776**	0.7175
			Ruby	0.6729	0.5771	0.4991	0.5229	0.6996	0.6490
			ALL	0.8119	0.7431	0.6411	0.7284	0.8455	0.7030
			MPLCS	0.8179	0.7558	**0.6778**	0.7306	0.8656	**0.7450**
	NBOW	NBOW	Go	**0.7729**	0.6247	0.5358	0.5136	0.6476	0.5650
			Java	0.6561	**0.7063**	0.5423	0.5581	0.6604	0.5835
			Javascript	0.6399	0.5898	0.5920	0.5190	0.6190	0.5195
			Php	0.6744	0.6413	0.5502	**0.6677**	0.6954	0.5930
			Python	0.6833	0.6293	0.5350	0.5632	**0.7815**	0.6220
			Ruby	0.5723	0.5263	0.4403	0.4442	0.6150	0.5865
			ALL	0.7492	0.6773	0.5770	0.6401	0.7380	0.6200
			MPLCS	0.7713	0.7048	**0.6287**	0.6475	0.7731	**0.6770**
	SELF-ATT	NBOW	Go	**0.8213**	0.6353	0.5588	0.5489	0.6736	0.5850
			Java	0.7391	**0.7491**	0.5810	0.6450	0.7203	0.6365
			Javascript	0.6774	0.6150	0.6402	0.5764	0.6821	0.5915
			Php	0.7329	0.6840	0.5893	**0.7356**	0.7801	0.6625
			Python	0.7282	0.6636	0.5912	0.6523	**0.8620**	0.6865
			Ruby	0.5912	0.5210	0.4362	0.4408	0.6221	0.5995
			ALL	0.8038	0.7269	0.6285	0.7186	0.8290	0.6905
			MPLCS	0.8163	0.7438	**0.6667**	0.7209	0.8537	**0.7325**

Table 3. *Cont.*

SuccessRate@k	CODE ENCODE	QUERY ENCODE	Model	Go	Java	Javascript	Php	Python	Ruby
SuccessRate@10	SELF-ATT	SELF-ATT	Go	**0.8581**	0.6970	0.6100	0.6301	0.7476	0.664
			Java	0.7936	**0.7998**	0.6325	0.6995	0.7826	0.6945
			Javascript	0.7229	0.6749	0.6967	0.6129	0.7563	0.6755
			Php	0.7831	0.7405	0.6550	**0.7759**	0.8486	0.7355
			Python	0.7809	0.7215	0.6475	0.7154	**0.9030**	0.751
			Ruby	0.6611	0.5995	0.5125	0.5355	0.7058	0.6785
			ALL	0.8399	0.7777	0.6832	0.7636	0.8755	0.7455
			MPLCS	0.8494	0.7910	**0.7287**	0.7703	0.8969	**0.7930**
	CNN	CNN	Go	**0.8622**	0.7146	0.6318	0.6672	0.7891	0.6945
			Java	0.8104	**0.8131**	0.6694	0.7236	0.8215	0.7260
			Javascript	0.7626	0.7114	0.7203	0.6492	0.7987	0.7020
			Php	0.8054	0.7581	0.6771	**0.7890**	0.8722	0.7620
			Python	0.8101	0.7507	0.6708	0.7234	**0.9155**	0.7710
			Ruby	0.7373	0.6406	0.5624	0.5895	0.7678	0.7240
			ALL	0.8447	0.7898	0.7008	0.7741	0.8906	0.7640
			MPLCS	0.8523	0.8067	**0.7410**	0.7803	0.9117	**0.8040**
	NBOW	NBOW	Go	**0.8208**	0.6841	0.6033	0.5832	0.7196	0.6280
			Java	0.7253	0.7624	0.6077	0.6280	0.7320	0.6595
			Javascript	0.7025	0.6546	0.6593	0.5884	0.6902	0.5965
			Php	0.7349	0.7003	0.6165	**0.7291**	0.7622	0.6660
			Python	0.7449	0.6900	0.6068	0.6321	**0.8422**	0.6920
			Ruby	0.6448	0.5913	0.5105	0.5153	0.6894	0.6565
			ALL	0.8002	0.7361	0.6453	0.7034	0.8030	0.7045
			MPLCS	**0.8217**	**0.7626**	**0.6975**	0.7156	0.8397	**0.7455**
	SELF-ATT	NBOW	Go	**0.8508**	0.6919	0.6198	0.6176	0.7395	0.6670
			Java	0.7896	**0.7934**	0.6372	0.7000	0.7831	0.7010
			Javascript	0.7328	0.6732	0.7017	0.6421	0.7465	0.6570
			Php	0.7789	0.7322	0.6457	**0.7755**	0.8319	0.7205
			Python	0.7779	0.7183	0.6488	0.7069	0.8994	0.7480
			Ruby	0.6594	0.5858	0.5063	0.5126	0.6946	0.6775
			ALL	0.8367	0.7745	0.6892	0.7637	0.8750	0.7465
			MPLCS	0.8482	0.7918	**0.7312**	0.7715	**0.9011**	**0.7865**

4. Experiment Results

We have prepared several different sets of code and query encoding methods and combined them into different models, including the common 1dCNN, NBOW, and self-attention. We have also prepared a test set for each language and tested it on each monolingual model and MPLCS respectively, the MRR results and SuccessRate@k results are shown in Tables 2 and 3 respectively. The results indicate that the monolingual model's prediction performed better only for its own language, however, it did not perform well for other languages (Ruby is because the training set is too small). Such a result holds in all models. Otherwise, we can observe the similarity between programming languages through this table, for example between Python and PHP. Notice that the ALL model trained by fusing the six data, has consistent test results across various languages. Although the accuracy rate is not as high as monolingual to itself, it is more accurate than monolingual to other languages. Our model outperforms ALL on almost every language. It can be confirmed that the student model does indeed study the knowledge of the teacher model through knowledge distillation techniques. Notice that MPLCS is superior to the teacher model on the Ruby and JavaScript test sets. This is due to the fact that the training sets for both languages are relatively small, and the multilingual fusion model can compensate for the small training set to some extent.

The effect of λ on the student model can be seen in Table 4. We can see that as λ increases from 0 and the student model receives more guidance from the teacher model, which leads to the gradual increase in MMR across all models. When $\lambda = 0.8$, The average MRR for the six languages reached a maximum, relatively close to the MMR when $\lambda = 0.9$ and $\lambda = 1.0$. The results indicate that as more teacher models are instructed, the more knowledge student models are learned.

Table 4. The effect of λ on the student model.

Lambda	Go	Java	Javascript	Php	Python	Ruby	Avg
0.0	0.7304	0.6365	0.5520	0.6041	0.7208	0.5925	0.6535
0.1	0.7386	0.6415	0.5567	0.6086	0.7274	0.5957	0.6591
0.2	0.7385	0.6444	0.5594	0.6084	0.7320	0.5980	0.6611
0.3	0.7345	0.6443	0.5602	0.6130	0.7362	0.6005	0.6628
0.4	0.7393	0.6460	0.5617	0.6138	0.7354	0.6033	0.6642
0.5	0.7394	0.6469	0.5602	0.6184	**0.7400**	**0.6089**	0.6668
0.6	0.7386	0.6471	0.5579	0.6165	0.7384	0.6024	0.6655
0.7	**0.7470**	0.6469	0.5563	**0.6193**	0.7364	0.6013	0.6669
0.8	0.7400	**0.6473**	**0.5621**	0.6191	0.7391	0.6059	**0.6670**
0.9	0.7421	0.6445	0.5533	0.6156	0.7357	0.6046	0.6642
1.0	0.7453	0.6456	0.5561	0.6155	0.7356	0.6042	0.6651

5. Conclusions

In this paper we present a new idea for semantic code retrieval - multi-code language code retrieval. By introducing the knowledge distillation technique, we established a Multi-Programming Language Code Search (MPLCS) model. The model can fuse several monolingual teacher models into a single student model, it supports multi-language code retrieval and also compensates for the deficiencies for languages where the training set is too small. In addition, MPLCS has no restrictions on the encoding method, it can be applied using a variety of different encoding methods. This paper only applied a general knowledge distillation technique and used only the results encoded from the teacher's model, thus the model is not significant in terms of accuracy improvement. However, this paper could have an intriguing effect on multi-code language code retrieval tasks.

Open Questions

- In this paper, only the simplest features of the code are obtained, which treats it as a new natural language, other features such as API sequences, information from AST trees were not used in this paper, further research on these features could better improve the accuracy.
- As mentioned before, a high-quality training set can also greatly improve the practical meaning of the conclusions.
- Translation between different programming languages is also a very interesting research direction.
- Multi-natural language to multi-programming language is also a valuable research direction, but it will require a more comprehensive dataset as support.

Author Contributions: Conceptualization, W.L.; Data curation, W.L.; Formal analysis, W.L. and J.X.; Funding acquisition, Q.C.; Investigation, W.L. and J.X.; Methodology, W.L.; Project administration, Q.C.; Resources, W.L.; Software, W.L.; Supervision, J.X. and Q.C.; Validation, W.L. and J.X.; Visualization, W.L.; Writing—original draft, W.L.; Writing— review & editing, W.L. and J.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by the National Key Research and Development Program of China, grant number 2018YFB2101200.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Dataset is available on https://s3.amazonaws.com/code-search-net/CodeSearchNet/v2/python.zip. The content in brackets is the programming language name, such as java.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Luan, S.; Yang, D.; Barnaby, C.; Sen, K.; Chandra, S. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.* **2019**, *3*, 1–28. [CrossRef]
2. Lv, F.; Zhang, H.; Lou, J.g.; Wang, S.; Zhang, D.; Zhao, J. Codehow: Effective code search based on api understanding and extended boolean model (e). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 260–270.
3. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv* **2019**, arXiv:1909.09436.
4. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944.
5. Haldar, R.; Wu, L.; Xiong, J.; Hockenmaier, J. A multi-perspective architecture for semantic code search. *arXiv* **2020**, arXiv:2005.06980.
6. Sachdev, S.; Li, H.; Luan, S.; Kim, S.; Sen, K.; Chandra, S. Retrieval on source code: a neural code search. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Philadelphia, PA, USA, 18 June 2018; pp. 31–41.
7. Cambronero, J.; Li, H.; Kim, S.; Sen, K.; Chandra, S. When deep learning met code search. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 964–974.
8. Yin, P.; Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv* **2017**, arXiv:1704.01696.
9. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
10. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and evaluating contextual embedding of source code. In Proceedings of the International Conference on Machine Learning, PMLR, Virtual Event, 13–18 July 2020; pp. 5110–5121.
11. Mayer, P.; Bauer, A. An empirical analysis of the utilization of multiple programming languages in open source projects. In Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, Nanjing, China, 27–29 April 2015; pp. 1–10.
12. Mayer, P.; Kirsch, M.; Le, M.A. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *J. Softw. Eng. Res. Dev.* **2017**, *5*, 1–33. [CrossRef]
13. Hinton, G.; Vinyals, O.; Dean, J. Distilling the knowledge in a neural network. *arXiv* **2015**, arXiv:1503.02531.
14. Johnson, M.; Schuster, M.; Le, Q.V.; Krikun, M.; Wu, Y.; Chen, Z.; Thorat, N.; Viégas, F.; Wattenberg, M.; Corrado, G.; et al. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 339–351. [CrossRef]
15. Firat, O.; Cho, K.; Bengio, Y. Multi-way, multilingual neural machine translation with a shared attention mechanism. *arXiv* **2016**, arXiv:1601.01073.
16. Ha, T.L.; Niehues, J.; Waibel, A. Toward multilingual neural machine translation with universal encoder and decoder. *arXiv* **2016**, arXiv:1611.04798.
17. Lu, Y.; Keung, P.; Ladhak, F.; Bhardwaj, V.; Zhang, S.; Sun, J. A neural interlingua for multilingual machine translation. *arXiv* **2018**, arXiv:1804.08198.
18. Tan, X.; Ren, Y.; He, D.; Qin, T.; Zhao, Z.; Liu, T.Y. Multilingual neural machine translation with knowledge distillation. *arXiv* **2019**, arXiv:1902.10461.
19. Xu, R.; Xiong, C.; Chen, W.; Corso, J. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In Proceedings of the AAAI Conference on Artificial Intelligence, Austin, TX, USA, 25–30 January 2015; Volume 29.
20. Karpathy, A.; Fei-Fei, L. Deep visual-semantic alignments for generating image descriptions. In Proceedings of the IEEE conference on computer vision and pattern recognition, Boston, MA, USA, 7–12 June 2015, pp. 3128–3137.
21. Wan, Y.; Shu, J.; Sui, Y.; Xu, G.; Zhao, Z.; Wu, J.; Yu, P.S. Multi-modal attention network learning for semantic source code retrieval. *arXiv* **2019**, arXiv:1909.13516.
22. Zeng, C.; Yu, Y.; Li, S.; Xia, X.; Wang, Z.; Geng, M.; Xiao, B.; Dong, W.; Liao, X. deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search. *arXiv* **2021**, arXiv:2103.13020.
23. Gu, J.; Chen, Z.; Monperrus, M. Multimodal Representation for Neural Code Search. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 483–494.
24. Sennrich, R.; Haddow, B.; Birch, A. Neural machine translation of rare words with subword units. *arXiv* **2015**, arXiv:1508.07909.