Cross-Domain Deep Code Search with Meta Learning

Yitian Chai\textsuperscript{1}, Hongyu Zhang\textsuperscript{2}, Beijun Shen\textsuperscript{1}, Xiaodong Gu\textsuperscript{1*}
\textsuperscript{1}School of Software, Shanghai Jiao Tong University, China
\textsuperscript{2}The University of Newcastle, Australia
\{sjtu_chayt,bjshen,xiaodong.gu\}@sjtu.edu.cn,hongyu.zhang@newcastle.edu.au

ABSTRACT

Recently, pre-trained programming language models such as CodeBERT have demonstrated substantial gains in code search. Despite showing great performance, they rely on the availability of large amounts of parallel data to fine-tune the semantic mappings between queries and code. This restricts their practicality in domain-specific languages that have relatively scarce and expensive data. In this paper, we propose CroCS, a novel approach for domain-specific code search. CroCS employs a transfer learning framework where an initial program representation model is pre-trained on a large corpus of common programming languages (such as Java and Python), and is further adapted to domain-specific languages such as Solidity and SQL. Unlike cross-language CodeBERT, which is directly fine-tuned in the target language, CroCS adapts a few-shot meta-learning algorithm called MAML to learn the good initialization of model parameters, which can be best reused in a domain-specific language. We evaluate the proposed approach on two domain-specific languages, namely Solidity and SQL, with model transferred from two widely used languages (Python and Java). Experimental results show that CroCS significantly outperforms conventional pre-trained code models that are directly fine-tuned in domain-specific languages, and it is particularly effective for scarce data.

CCS CONCEPTS

• Software and its engineering → Reusability: Automatic programming.

KEYWORDS

Code Search, Pre-trained Code Models, Meta Learning, Few-Shot Learning, Deep Learning

1 INTRODUCTION

Recently, deep neural networks (DNN) have been widely utilized for code search [4, 9, 13, 14, 28, 38]. Unlike traditional keyword matching methods [2, 7, 16, 17, 21, 22], deep code search models employ deep neural networks to learn the representations of both queries and code, and measure their similarities through vector distances. The application of DNNs significantly improves the understanding of code semantics, thereby achieving superb performance in code search tasks [9, 13, 18, 40].

A major challenge for deep code search is the adaptation of deep learning models to domain-specific languages. State-of-the-art code search methods are mainly designed for common languages such as Java and Python. They rely heavily on the availability of large parallel data to learn the semantic mappings between code and natural language [11]. On the other hand, there is an emerging trend of domain-specific languages such as Solidity for smart contracts [37, 39, 44] where code search is also needed. There is often insufficient training data in specific domains, causing poor fit of deep learning models. Furthermore, for each specific domain, the costs of data collection, cleaning, and model training for constructing an accurate model are all non-neglectable.

One potential route towards addressing this issue is the pre-trained code models, which pre-train a common representation model on a large, multilingual code corpus, and then fine-tune the model on task-specific data [29]. This enables code search models to transfer prior knowledge from the data-rich languages to the low-resource language. For example, CodeBERT [9], the state-of-the-art code representation model, can be pre-trained on multiple common languages and then fine-tuned in the code search task for a target language [29]. However, it is challenging to reuse knowledge from a mix of source languages for code search in the target language. Different languages have their unique characteristics, and correspond to different representations. Parameters learnt from each language can distract each other, resulting in a conflict in the shared representations. This is even more challenging in the domain-specific code search, where the target language usually has scarce training samples.

In this paper, we present CroCS (Cross-Domain Deep Code Search), a cross-domain code search technique based on few-shot meta learning. CroCS extends the "pretraining-finetuning" paradigm of CodeBERT with a meta learning phase that explicitly adapts the model parameters learnt from multiple source languages to the target language. CroCS begins by pre-training CodeBERT on a large corpus of multiple common languages such as Java and Python. Then, a meta learning algorithm named MAML (Model-Agnostic Meta-Learning) is employed in order to prevent the model parameters from falling into the local optimization of source languages.
The goal of this algorithm is to find the initialization of model parameters that enables fast adaptation to a new task with a small amount of training examples.

To evaluate the effectiveness of CroCS, we pre-train CroCS on a large corpus of common languages such as Python and Java. Then, we perform code search on two domain-specific datasets written in Solidity and SQL. We compare our approach with three baseline models, namely, a neural code search model without pre-training, a within-domain pre-training model CodeBERT [9], and a cross-language CodeBERT [29] that directly fine-tunes the target language on a pre-trained model. Experimental results show that CroCS significantly outperforms within-domain counterparts. In particular, our approach shows more strength when the data is scarce, indicating the superb effectiveness of our approach in cross-domain code search.

The contributions of this work can be summarized as:
- We propose CroCS, a novel cross-domain code search method using few-shot meta learning.
- We extensively evaluate CroCS on a variety of cross-language code search tasks. Experimental results have shown that CroCS outperforms the pre-training and fine-tuning counterparts by a large margin.

2 BACKGROUND

2.1 Code Search Based on Deep Learning

The past few years have witnessed a rapid development of deep learning for software engineering, in which code search has been one of the most successful applications. Compared with traditional text retrieval methods, deep learning based code search learns representations of code and natural language using deep neural networks, and thus has achieved superb performance [4, 9, 13, 14, 28].

2.2 Pre-trained Models for Code Search

Recently, pre-trained models such as BERT [8] and GPT-2 [26] have achieved remarkable success in the field of NLP [8, 26]. As such, researchers start to investigate the adaptation of pre-trained models to software programs [9, 36?]. Code search is one of the most successful applications of pre-trained models for programming languages.

One of the most successful pre-trained models for code is the CodeBERT [9]. CodeBERT is built on top of BERT [8] and Roberta [20], two popular pre-trained models for natural language. Unlike pre-trained models in NLP, CodeBERT is designed to represent bi-modal data [5], namely, programming and natural languages. Figure 2 shows the architecture of CodeBERT. In general, the model is built upon a Transformer encoder. The training involves two pre-training tasks in six programming languages. One is the masked language modeling (MLM), which trains the model to fill the masked token in the input sequences. The other task is the replaced token detection (RTD), which trains the model to detect the replaced tokens in the input sequences. These two pre-training tasks endow CodeBERT with generalization ability, so that it can be fine-tuned to adapt to downstream tasks such as code search and code summarization.

As a code representation model, CodeBERT has been successfully employed for code search [9]. Specifically, a binary classifier is employed which takes as input the representation of the [CLS] token and predicts whether a given <NL, PL> pair is semantically related. This classifier is then fine-tuned on a code search dataset by minimizing the cross-entropy loss. In the search phase, the classifier

\[ L(c, d^+, d^-) = \max(\cos(c, d^+) - \cos(c, d^-) + \epsilon, 0) \] (1)
predicts the matching score between an NL query and each code snippet in the codebase. The search engine returns the top-k code snippets that have the highest matching scores.

Due to the superb performance, researchers have also applied CodeBERT for cross-language code search [29]. They pre-trained CodeBERT with multiple languages such as Python, Java, PHP, Javascript, and Go, and then fine-tuned a code search model on an unseen language such as Ruby. Results have shown that cross-language code search achieves better performance than training in a single language from scratch. This further supports the effectiveness of transfer learning for code search [29].

2.3 Meta Learning and Few-Shot Learning

Few-shot learning is a machine learning technology that aims to quickly adapt a trained model to new tasks with less examples [30]. Despite the superb performance, deep learning models are often data-hungry [11]. They rely on the availability of large-scale data for training. That means, the performance can be limited due to the scarcity of data in specific domains [11]. By contrast, humans can learn knowledge from a few examples. For example, a child can learn to distinguish between lions and tigers when provided with a few photos, probably because human beings have prior knowledge before learning new data or because human brains have a special way to process knowledge. Based on this intuition, researchers have proposed few-shot learning.

Few-shot learning methods can be roughly classified into the following two categories:

1) Metric-based methods, which learn a distance function between data points so that new test samples can be classified through comparison with the K labeled examples [42]. There are a few typical algorithms for metric-based few-shot learning, such as Siamese Network [6], Prototypical Network [30], and Relation Network [32].

2) Meta Learning, also known as “learning-to-learn”, which trains a model on a variety of learning tasks, such that it can solve new learning tasks using only a small number of training samples [10]. Unlike the conventional machine learning prototype that a model is optimized in the training set to minimize the training loss, meta learning updates model parameters using the validation loss in order to enhance the generalization to different tasks. There are some typical algorithms for few-shot meta learning, such as MAML [10] and Reptile [24].

MAML (Model-Agnostic Meta-Learning) is a few-shot meta learning algorithm which aims at learning a good initialization of model parameters so that the model can quickly reach the optimal point in a new task with a small number of data samples [10, 42]. The algorithm assumes that the data used for training follows a distribution \( p(T) \) over \( k \) tasks \( \{T_1, ..., T_k\} \), where \( T_i \) stands for a specific machine learning task on the data. The intuition is that some data features are more transferrable than others. In other words, they are broadly applicable to all tasks in \( p(T) \), rather than a single individual task \( T_i \). To find such general-purpose representations, MAML updates model parameters that are sensitive to changes in the task, such that small changes in the parameters will produce large improvements on the loss function of any task drawn from \( p(T) \). Motivated by this, MAML separates data into individual tasks. A meta learner is employed to update parameters using gradients on each local task \( T_i \) [10]. A more detailed description of the algorithm and how it is adapted to code search will be presented in Section 3.3.

3 APPROACH

3.1 Overview

Figure 3 shows the architecture of CroCS. In general, CroCS takes CodeBERT [9] as the backbone, and extends it with a meta learning phase. The core component of CroCS is RoBERTa [20], which is built upon a multi-layer bidirectional Transformer [33] encoder.

The pipeline of CroCS involves four phases. Similar to CodeBERT, we start by pre-training CroCS to learn code representations in a large corpus of multiple source languages. Next, we perform meta learning to explicitly transfer the representations of source languages into the target language. After the domain adaptation, we fine-tune it on the code search data of the target language in order to train the semantic mapping between code and natural language. We finally perform code search using the fine-tuned model. We will describe the detailed design of each phase in the following sections.

3.2 Pre-training

The pre-training phase aims to learn code and NL representations from a large corpus of multiple common languages such as Java and Python. Similar to CodeBERT, we use the pre-training task of masked language modeling (MLM). We did not use the RTD (replaced token detection) pre-training task of CodeBERT because the effect of this task has been shown to be marginal [9].

In the pre-training phase, the model takes as input an (NL, PL) pair which is formatted into a sequence of

\[
[\text{CLS}], w_1, w_2, ..., w_n, [\text{SEP}], c_1, c_2, ..., c_m, [\text{EOS}]
\]

where \( w_1, w_2, ..., w_n \) denotes a sequence of \( n \) words in the natural language text, while \( c_1, c_2, ..., c_m \) represents a sequence of \( m \) tokens in the code snippet. The special [CLS] token at the beginning is...
a placeholder for the representation of the entire input sequence. The \([SEP]\) token indicates the border of the code snippet and the natural language text. The \([EOS]\) token indicates the end of the sequence.

During the pre-training process, we randomly replace 15% of the tokens in the input sequence with a special \([MASK]\) token and let the model predict the original token. The task can be optimized by minimizing the cross-entropy loss between the predicted and the original tokens.

The pre-trained model can be used to produce the contextual vector representation of each token for both natural language descriptions and code snippets. In particular, the representation of the \([CLS]\) token stands for the aggregated sequence representation which can be used for classifying the entire input sequence.

### 3.3 Meta Learning

We next perform meta learning to adapt the pre-trained code model to the target domain. We employ a meta-learning algorithm named MAML (Model-Agnostic Meta-Learning) [10] which is a typical algorithm for few-shot learning [10, 12, 31]. The key idea of MAML is to use a set of source tasks \(\{T_1, ..., T_k\}\) to find the initialization of parameters \(\theta_0\) from which learning a target task \(T_0\) would require only a small number of training samples [10]. In the context of code search, this amounts to using large data of common languages to obtain \(\theta_0\) and use a set of source tasks \{NL, PL\} to find the initialization of \(\theta_0\) for the target language. We fine-tune the model on the code search task, which can be formulated as a binary classification problem. For a dataset \(\mathcal{D}^{\text{train}}_{\text{large}}\) of each language, we construct the \(\mathcal{D}^{\text{train}}_{\text{small}}\) which is randomly selected from the \(\mathcal{D}^{\text{train}}_{\text{large}}\), and then randomly replace NL or PL in the original pairs. We assemble different \(\mathcal{D}^{\text{valid}}\) for each language, with the \(\mathcal{D}^{\text{valid}}\) used for training follows a distribution \(\mathcal{D}^{\text{valid}}\) on large data of common languages.

Hence, it splits each training set \(\mathcal{D}^{\text{train}}_{\text{large}}\) into a training and validation set \(\mathcal{D}^{\text{train}}_{\text{small}}\) at training a code search model with small sized data, therefore the meta learner.

The procedure of our algorithm is summarized in Algorithm 1.

In order to learn a good model initialization of multiple source languages, we construct the \(\mathcal{D}^{\text{train}}_{\text{large}}\) from multiple source languages. We segment the original dataset of each language into batches. This results in a pool of batches that involves multiple languages. During meta learning, we randomly select \(k\) batches from the batch pool.

![Figure 4: An overview of the MAML algorithm.](image)

**Algorithm 1 Meta Learning for Code Search**

**Require:** \(\alpha, \beta\): step size; \(M\): meta update steps
1. Pre-train the global model on source languages and obtain the initial parameters \(\theta\)
2. Create \(k\) copies of \(\theta\) with each \(\theta_i\) being the local parameters for \(T_i\).
3. **while** not done **do**
4. Divide the dataset of each source language into batches
5. Construct \(\mathcal{D}^{\text{train}}_{\text{large}}\) by randomly selecting \(k\) batches from the batch pool, with the \(i\)-th batch \(D_i\) assigned for task \(T_i\)
6. **for each** \(D_i \in \mathcal{D}^{\text{train}}_{\text{large}}\) **do**
7. Split \(D_i\) into \(\mathcal{D}^{\text{train}}_{\text{large}}, \mathcal{D}^{\text{valid}}_{\text{large}}\)
8. Run \(T_i\) on \(\mathcal{D}^{\text{train}}_{\text{large}}, \mathcal{D}^{\text{valid}}_{\text{large}}\) and evaluate local gradients \(\nabla_{\theta} L_{T_i}(f_\theta)\) using the cross-entropy loss \(L_{T_i}\)
9. Update local parameters \(\theta_i\) with gradient descent:
   \[
   \theta_i = \theta - \alpha \nabla_{\theta} L_{T_i}(f_\theta)
   \]
10. **if** \(i \mod M = 0\) **then**
11. Evaluate gradients \(\nabla_{\theta} L_{T_i}(f_\theta)\) using the cross-entropy loss \(L_{T_i}\) in \(\mathcal{D}^{\text{valid}}_{\text{large}}\)
12. Update the global parameters \(\theta\) using the gradients on the validation set:
   \[
   \theta = \theta - \beta \nabla_{\theta} L_{T_i}(f_\theta)
   \]
13. **end if**
14. **end for**
15. **end while**

### 3.4 Fine-Tuning

In the fine-tuning phase, we adapt CroCS to the code search task in the target language. We fine-tune the model on the code search task, which can be formulated as a binary classification problem. For a corpus of \(\langle\text{NL}, \text{PL}\rangle\) pairs, we create the same number of negative samples by randomly replacing NL or PL in the original pairs. We assign a label to each pair to indicate whether the NL is corresponding to the PL in the pair (1=relevant, 0=irrelevant).
For each training instance, we build an input sequence with the same format as in the pre-training phase. We take the hidden state in the $[CLS]$ position of CodeBERT as the aggregated representation of the input sequence. The representation is further taken as input to a fully connected neural classifier to predict whether the given $(NL, PL)$ pair is relevant. We fine-tune the model by minimizing the binary cross-entropy loss between predictions and labels.

### 3.5 Domain-Specific Code Search

Finally, we perform code search based on the fine-tuned model in a domain-specific codebase. The code search engine works with the following steps:

1. A natural language query $Q$ is provided to the code search system.
2. Splice $Q$ separately with each code snippet $C_i$ in the codebase to obtain a series of input sequences

$$< Q, C_1 >, \ldots, < Q, C_n >$$

3. Input these sequences into the trained model and obtain their matching scores.
4. Sort code snippets according to their matching scores.
5. Return the top-k code snippets as the results.

### 4 EXPERIMENTAL SETUP

We evaluate the performance of CroCS in domain-specific code search tasks and explore the effect of training data size on the performance. Finally, we extend our method to other backbone pre-trained models such as GPT-2 [26]. In summary, we evaluate CroCS by addressing the following research questions:

- **RQ1: How effective is CroCS in cross-domain code search?**
  To verify the effectiveness of CroCS in cross-domain code search tasks, we take Python and Java as the source languages and adapt the learned model to two domain-specific languages, namely, Solidity and SQL. We compare the accuracy of code search by various approaches in the two target languages.

- **RQ2: What is the impact of data size on the performance of cross-domain code search?**
  As mentioned, one of the challenges for cross-domain code search is the scarcity of data in the domain-specific language. In RQ2, we aim to study the effect of data size on the performance. We vary the size of dataset and compare the performance under different data sizes.

- **RQ3: How effective is CroCS applied to other pre-trained programming language models?**
  Besides CodeBERT, there are other pre-trained models that also achieve outstanding results in software engineering tasks [1, 23, 25]. We wonder whether other pre-trained models can have the same effectiveness on code search when equipped with meta learning. We replace the backbone pre-trained model with GPT-2 [3, 26], which is also a popular pre-trained language model based on Transformer. GPT-2 differs from BERT in that it is an autoregressive language model built on top of the Transformer decoder. We evaluate

| Phase       | Python       | Java        |
|-------------|--------------|-------------|
| pre-train   | # functions  | 412,178     | 454,451     |
|             | # comments   | 412,178     | 454,451     |
| meta learning | # functions  | 824,342     | 908,886     |
|             | # comments   | 824,342     | 908,886     |

the effectiveness of CroCS and examine their impact to the performance of code search.

### 4.1 Implementation Details

We build our models on top of the RoBERTa [20] using the same configuration as RoBERTa-base ($H=768, A=12, L=12$). The rate of masked tokens is set to 15%. We use the default CodeBERT tokenizer, namely, Microsoft/codebert-base-MLM with the same vocabulary size (50265). We set the maximum sequence length to 256 to fit our maximum computational resources. The default batch size is set to 64. The three hyperparameters $\alpha, \beta, M$ in Algorithm 1 are empirically set to $1e-5, 1e-4$, and 100, respectively. Our experimental implementation is based on the tool provided by Huggingface Transformers$^1$ and the higher library provided by Facebook Research$^2$.

All models are trained on a GPU machine with Nvidia Tesla V100 32G using the Adam [15] algorithm. We use a learning rate of $5e-5$ [9] in the pre-training phase which warms up in the first 1,000 steps and linearly decays. We measure the performance on the validation set during the training process, and select the checkpoint of the model which has the best accuracy on the validation set for testing.

### 4.2 Datasets

#### 4.2.1 Data Used for Pre-training and Meta Learning

We pre-train and perform meta learning using the training data for the code search task provided by CodeBERT [9]. We select two popular languages, namely, Python and Java as the source languages. The statistics of the dataset are shown in Table 1. For each language, the dataset contains parallel data of (NL, PL) pairs, including both positive and negative samples. In order to prevent the training from falling into a local optimum of one source language, we use only positive samples for pre-training and use the entire set of pairs for meta learning.

#### 4.2.2 Data Used for Fine-tuning and Code Search

We fine-tune and test the code search task using two domain-specific languages,

---

$^1$https://huggingface.co/transformers/

$^2$https://higher.readthedocs.io/
We measure the performance of code search using two popular quantitative criteria on the test set, including MRR (Mean Reciprocal Rank) and the top-k accuracy. They are commonly used for evaluating code search engines [9, 13].

MRR [22, 41] aims to let a search algorithm score search results in turn according to the search content, and then arrange the results according to the scores in a descend order. For N test queries, the MRR can be computed as

$$\text{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\text{Rank}(i)}$$

where Rank(i) represents the position of the correct code snippet in the returned results for query i. The greater the MRR score, the better the performance on the code search task.

We preprocess the dataset by removing all inline comments from functions. We remove duplicate pairs, namely, two (NL, PL) pairs that have the same comment but differ only in the number of position in the dataset and a few variable names in code. We also balance positive and negative samples where the negative samples are generated by randomly replacing NL (i.e. (c, d)) and PL (i.e. (c, d)) of positive samples.

SQL is a well-known language that is specifically designed for manipulating database systems. The dataset we used for fine-tuning and testing SQL is provided by [43] for cross-domain semantic parsing and SQL code generation (text-to-SQL). The original data is in a JSON format and contains the following fields:

- question: the natural language question.
- question_toks: the natural language question tokens.
- db_id: the database id to which this question is addressed.
- query: the SQL query corresponding to the question.
- query_toks: the SQL query tokens corresponding to the question.
- sql: parsed results of this SQL query.

We preprocess the SQL dataset by selecting the “question” and “query” fields from the .json data as our NL and PL, respectively. We remove duplicate data that has the same code from the original test set. We also balance positive and negative samples where the negative samples are generated by randomly disrupting descriptions and code based on positive samples.

### 4.3 Evaluation Metrics

We measure the performance of code search using two popular quantitative criteria on the test set, including MRR (Mean Reciprocal Rank) and the top-k accuracy. They are commonly used for evaluating code search engines [9, 13].

MRR [22, 41] aims to let a search algorithm score search results in turn according to the search content, and then arrange the results according to the scores in a descend order. For N test queries, the MRR can be computed as

$$\text{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\text{Rank}(i)}$$

where Rank(i) represents the position of the correct code snippet in the returned results for query i. The greater the MRR score, the better the performance on the code search task.

### 5 EXPERIMENTAL RESULTS

#### 5.1 Effectiveness in Cross-Domain Deep Code Search (RQ1)

Table 3 and 4 show the performance of different approaches in the cross-domain code search task. We take Python and Java as the source languages and test the performance on two domain-specific languages, namely, SQL and Solidity.

[3] https://github.com/microsoft/CodeBERT
Table 4: Performance of each method in the Solidity dataset.

| Model                  | Acc@1 | Acc@5 | Acc@10 | MRR  |
|------------------------|-------|-------|--------|------|
| No-Pretraining         | 0.002 | 0.008 | 0.014  | 0.0101|
| CodeBERT (NL-based)    | 0.453 | 0.732 | 0.821  | 0.5801|
| CodeBERT (within-domain)| 0.515 | 0.798 | 0.857  | 0.6383|
| CodeBERT (cross-language)| 0.532 | 0.779 | 0.848  | 0.6436|
| CroCS                  | **0.658** | **0.829** | **0.879** | **0.7336**|

Overall, CroCS achieves the best performance among all the methods. From the results on the SQL dataset, we can see that CroCS outperforms the baseline models in terms of all metrics, especially the top-1 accuracy and MRR, which are about 11% and 7% greater than the strong baselines, respectively.

The improvement is more significant on the Solidity dataset (Table 4). We can see that CroCS substantially outperforms strong baselines especially in the top-1 accuracy and MRR, which are about 20% and 18% stronger, respectively.

There is a large margin between CodeBERT (NL-based) and CodeBERT (within-domain). We hypothesize that this is because the SQL corpus is too scarce, so that the pre-training may not provide sufficient prior knowledge to the code-search model. CroCS obtains more significant improvement against CodeBERT (NL-based) in SQL than that in the Solidity dataset, probably because SQL is much closer to natural language than Solidity.

The results demonstrate that CroCS is remarkably effective in domain-specific code search tasks.

5.2 Effect of Data Size (RQ2)

Figure 5 and 6 show the performance of CroCS under different data sizes compared with the cross-language CodeBERT [29]. We vary the size of training data from 0 to full data.

As the result shows, CroCS outperforms the baseline model under all data sizes, which supports the significance of the improvement achieved by CroCS. In particular, we note that when the data size gets smaller (e.g., <500), the improvement of CroCS against the baseline model becomes more significant. That means that CroCS is particularly effective in scarce data, indicating the outstanding ability of CroCS on domain specific languages. By contrast, the baseline model without meta learning can not adapt to the task well due to the insufficiency of data.

5.3 Performance on other Pre-trained Models (RQ3)

Table 5: Performance of each method based on GPT-2.

| Language | Model                  | Acc@1 | Acc@5 | Acc@10 | MRR  |
|----------|------------------------|-------|-------|--------|------|
| SQL      | No-Pretraining         | 0.002 | 0.010 | 0.022  | 0.0124|
|          | GPT2 (NL-based)        | 0.481 | 0.808 | 0.889  | 0.6204|
|          | GPT2 (within-domain)   | 0.470 | 0.785 | 0.877  | 0.6088|
|          | GPT2 (cross-language)  | 0.447 | 0.767 | 0.875  | 0.5899|
|          | CroCS_{GPT-2}          | **0.511** | **0.823** | **0.905** | **0.6464**|
| Solidity | No-Pretraining         | 0.002 | 0.008 | 0.014  | 0.0101|
|          | GPT2 (NL-based)        | 0.484 | 0.751 | 0.830  | 0.6079|
|          | GPT2 (within-domain)   | 0.487 | 0.772 | 0.848  | 0.6073|
|          | GPT2 (cross-language)  | 0.481 | 0.760 | 0.827  | 0.6057|
|          | CroCS_{GPT-2}          | **0.561** | **0.781** | **0.846** | **0.6607**|
We evaluate the performance of CroCS\textsubscript{GPT}−2 and compare it with baseline models that are also based on GPT-2. We experiment with (\texttt{Python, Java}) as the source languages and test the performance in Solidity and SQL. The training differs a little bit in the meta learning phase: we formulate the input for code search as:

\[
[BOS], w_1, \ldots, w_N, c_1, \ldots, c_m, [EOS]
\]

where \([BOS]\) and \([EOS]\) represent the “beginning” and “ending” of the sequence, respectively. The representation of the \([EOS]\) token stands for the aggregated sequence representation and is used for classification. We implement CroCS\textsubscript{GPT} based on the Huggineface repository\textsuperscript{1}. The hyperparameters are set as follows: we set the batch size to 44, learning rate to 2.5\textsuperscript{-5} which warms up in the first 1,000 steps and decays according to a cosine curve.

Table 5 shows the performance of CroCS\textsubscript{GPT}−2 compared against baseline models. Clearly, CroCS\textsubscript{GPT}−2 works better than all the baseline models. The MRR scores of CroCS\textsubscript{GPT}−2 are about 5\% and 10\% greater than those of the baseline model in the SQL and Solidity languages, respectively. This affirms the effectiveness of CroCS\textsubscript{GPT}−2 when equipped with meta learning.

We notice that the GPT-2 pre-trained in natural language corpus shows a comparable performance to ours in the SQL language. We conjecture that SQL is simple and similar to natural languages, hence pre-training on massive text corpus is effective for the target task without heavy adaptation. Another notable point we observe is that the results of CroCS\textsubscript{GPT}−2 are lower than those of CroCS\textsubscript{BERT}, presumably because GPT-2 is a unidirectional language model, which dynamically estimates the probability of text sequences and can be more suitable for generation than search tasks. GPT-2 processes each input text from left to right sequentially, thus can be limited in representing context-sensitive features. By contrast, BERT-style models are trained with de-noising strategies (e.g., the MLM task) which enable them to obtain bidirectional, context-sensitive features.

### 5.4 Impact of Different Hyperparameters (RQ4)

Figure 7(a) and 7(b) show the performance of CroCS under different batch sizes on the SQL and Solidity datasets. We vary batch sizes to 64, 32, 16 and 8, respectively. The results show that larger batch sizes have slight impact on the performance, while smaller batch sizes have evident effect on the performance.

Figure 7(c) and 7(d) show the performance of CroCS under different learning rates on the SQL and Solidity datasets. We vary the learning rate to 2\textsuperscript{-5}, 1\textsuperscript{-5}, and 5\textsuperscript{-6}, respectively. As we can see, the performance is insensitive to learning rates lower than 1\textsuperscript{-5}. However, learning rates that are larger than 1\textsuperscript{-5} have significant impacts on performance.

To sum up, the impact of hyperparameters on CroCS is limited to a certain range. The performance is sensitive to the hyperparameters when the batch size is less than 32 or the learning rate is greater than 1\textsuperscript{-5}. In addition, our model is more sensitive to both batch size and learning rate on the Solidity dataset than SQL.

### 5.5 Case Study

We now provide specific search examples to demonstrate the effectiveness of CroCS in domain specific code search.

Listing 1 and 2 compare the top-1 results for the query “what is the smallest city in the USA” returned by CroCS and the cross-language CodeBERT, respectively. The query involves complex semantics such as the word \texttt{smallest}. A code search system is expected to associate “small” with the corresponding SQL keyword \texttt{MIN}. They are different but are semantically relevant. Listing 1 shows that CroCS can successfully understand the semantics of \texttt{smallest}, while the cross-language CodeBERT cannot. The example suggests that CroCS is better than the cross-language CodeBERT [29] in terms of semantic understanding.

Listing 3 and 4 show the results returned by CroCS and the cross-language CodeBERT for the query “Reset all the balances to 0 and the state to false” in the Solidity language. The keywords in the query are balances, state, and false. It can be seen that both approaches return code snippets that hit some of the keywords. However, the snippet returned by CroCS is clearly more relevant than that returned by the cross-language CodeBERT. For example, it explicitly states \texttt{beneficiary.balance=0} and \texttt{filled = false} in the source code. On the other hand, the snippet provided by the cross-language CodeBERT is vague in semantics. Cross-language CodeBERT may pay more attention to similar words and is limited in understanding semantics.

These examples demonstrate the superiority of CroCS in cross-domain code search, affirming the strong ability of learning representations at both token and semantic levels.

### Listing 1: The first result of query “what is the smallest city in the USA” returned by CroCS.

```
SELECT city_name
FROM city
WHERE population = (SELECT MIN(population) FROM city);
```
Across all the experiments, the performance of the experimental GPT-2 approach can be generalized to other pre-trained models such as CodeBERT. Furthermore, the results of RQ3 suggest that our parameters and gathering meta gradients. For example, in our experiments, it requires around 50% extra hours for meta-learning.

### 6 DISCUSSION

#### 6.1 Why does CroCS work better than the cross-language CodeBERT?

We believe that the advantage of CroCS mainly comes from the difference between meta learning and simply pre-training & fine-tuning. As Figure 8 illustrates, the traditional pre-training & fine-tuning paradigm tries to learn the common features of multiple source languages in the pre-training phase, and directly reuses the pre-trained parameters to specific tasks through fine-tuning. The features of different source languages distract each other, leading to an ill-posed representation to be reused by the target language. By contrast, meta learning employed by CroCS tries to adapt the pre-trained parameters to new tasks during the learning process, resulting in representations that take into account all source languages.

In a view of machine learning, both the pre-training & fine-tuning paradigm and meta learning aim to enhance the generalization ability of deep neural networks in multiple tasks. However, in the pre-training & fine-tuning paradigm, the model will not obtain task information before fine-tuning on specific downstream tasks, while meta learning focuses on learning information in specific tasks and can enhance the generalization ability of the model. CroCS successfully combines the two methods.

#### 6.2 Limitations

Although effective, we recognize that the adaptation of meta-learning to code search might not be a perfect fit. Meta-learning is usually used for classification tasks on scarce data [10, 42], whereas we adapt it to the context of code search. These two concepts (i.e., classification vs. ranking) are not a natural fit. Hence, meta-learning might not perfectly solve the root problem of cross-domain code search. More adaptations are demanded to fit the two concepts.

In order to efficiently adapt code search tasks to scarce data scenarios, we follow the MAML paper [10] and divide the data into machine learning “tasks”, with each task aiming at training a code search model with small sized data. Such an approach has a few benefits. For example, it is easy for task adaptations since it does not introduce any learned parameters. Furthermore, adaptation can be performed with any amount of data since it aims at producing an optimal weight initialization [10]. The limitation is that, the division of the data into “tasks” is random and there needs a concrete explanation on how split tasks are related to cross-language code search. It remains to investigate how such divisions turn out to be effective in scarce data.

Another downside of CroCS is that the MAML algorithm it employs can bring more time and computational cost in the large-scale data set. Different from the conventional gradient descent methods, MAML needs to compute a meta gradient based on multiple losses computed from sub-tasks. This costs extra time for saving model parameters and gathering meta gradients. For example, in our experiments, it requires around 50% extra hours for meta-learning.
We have identified the following threats to our approach:

**The number of source languages.** Due to the restriction of computational resources, we only selected two source languages and two domain-specific target languages. Meta learning with more source languages could have different results. In our future work, we will evaluate the effectiveness of our approach with more source and target languages.

**The selection of pre-training tasks.** The original CodeBERT uses two pre-training tasks, namely, masked language model (MLM) and replaced token detection (RTD) [9]. However, in our experiments, we only use the MLM as the pre-training task. Combining MLM with RTD may have effects on the results. However, we believe that the results of the MLM task can stand for the performance of pre-training because the objective of RTD is similar to MLM in that both are based on the idea of de-noising. More importantly, the RTD task requires too much cost of time and computational resources, while the improvement it brings is marginal according to the ablation experiments in the CodeBERT paper [9]. Moreover, compared with RTD, the MLM task is more widely used [34] in domains other than programming languages.

**Generalization to other pre-trained models.** We have built and evaluated our approach on top of two pre-trained models, namely, BERT and GPT-2. Thus, it remains to be verified whether or not the proposed approach is applicable to other pre-trained models such as BART [1] and T5 [23, 36].

6.3 Threats to Validity

We have identified the following threats to our approach:

**The number of source languages.** Due to the restriction of computational resources, we only selected two source languages and two domain-specific target languages. Meta learning with more source languages could have different results. In our future work, we will evaluate the effectiveness of our approach with more source and target languages.

**The selection of pre-training tasks.** The original CodeBERT uses two pre-training tasks, namely, masked language model (MLM) and replaced token detection (RTD) [9]. However, in our experiments, we only use the MLM as the pre-training task. Combining MLM with RTD may have effects on the results. However, we believe that the results of the MLM task can stand for the performance of pre-training because the objective of RTD is similar to MLM in that both are based on the idea of de-noising. More importantly, the RTD task requires too much cost of time and computational resources, while the improvement it brings is marginal according to the ablation experiments in the CodeBERT paper [9]. Moreover, compared with RTD, the MLM task is more widely used [34] in domains other than programming languages.

**Generalization to other pre-trained models.** We have built and evaluated our approach on top of two pre-trained models, namely, BERT and GPT-2. Thus, it remains to be verified whether or not the proposed approach is applicable to other pre-trained models such as BART [1] and T5 [23, 36].

7 RELATED WORK

7.1 Deep Learning Based Code Search

With the development of deep learning, there is a growing interest in adapting deep learning to code search [4, 13, 19]. The main idea of deep learning based code search is to map natural and programming languages into high-dimensional vectors using bi-modal deep neural networks, and train the model to match code and natural language according to their vector similarities. NCS (Neural Code Search) [28] proposed by Facebook learns the embeddings of code using unsupervised neural networks. Gu et al. [13] proposed CODEnn (Code-Description Embedding Neural Network), which learns the joint embedding of both code and natural language. CODEnn learns code representations by encoding three individual channels of source code, namely, method names, API sequences, and code tokens. UNIF [4] developed by Facebook can be regarded as a supervised version of NCS. Similar to CODEnn, UNIF designs two embedding networks to encode natural and programming languages, respectively. Semantic Code Search (SCS) [14] first trains natural language embedding network and programming language embedding network respectively and then trains the code search task by integrating the two embedding network with similarity function. CodeMatcher [19], which is inspired by DeepCS [13], combines query keywords with the original order and performs a fuzzy search on method names and bodies. Zhu et al. [45] proposed OCoR, a code retriever that handles the overlaps between different names used by different developers (e.g., “message” and “msg”). Wang et al. [35] proposed to enrich query semantics for code search with reinforcement learning.

While these methods are mainly designed for common languages, CroCS focuses on domain-specific code search, where training data is often scarce and costly. CroCS extends pre-trained models with meta learning to extract prior knowledge from popular common programming language for searching code written in domain-specific languages.

7.2 Pre-trained Language Models for Code

In recent years, pre-trained language models for source code have received much attention [1, 9, 23, 25]. CodeBERT [9], built on top of the popular model of BERT [8], is one of the earliest attempts that adapt pre-trained models for programming languages. CodeBERT is trained with six common programming languages (Python, Java, JavaScript, PHP, Ruby, and Go). Besides, they creatively proposed the replaced token detection (RTD) task for the pre-training of programming language. CoText [25] is a pre-trained Transformer model for both natural language and programming languages. It follows the encoder-decoder architecture proposed by [33]. PLBART [1] learns multilingual representations of programming and natural language jointly. It extends the scope of pre-training to denoising pre-training, which involves token masking, deletion, and infilling. Mastropaolo et al. [23] empirically investigated how T5 (Text-to-Text Transfer Transformer), one of the state-of-the-art PLMs in NLP, can be adapted to support code-related tasks. The authors pre-trained T5 using a dataset composed of English texts and source code, and then fine-tuned the model in four code-related tasks such as bug fix and code comment generation.

Although these pre-trained models for source code can be used for cross-language code search [29] through pre-training in multiple languages and fine-tuning in the domain-specific language, they do not take into account the difference between source and target...
languages, and are limited in performing domain-specific code search. By contrast, CroCS explicitly transfers representations of multiple source languages to the target language through meta learning.

7.3 Transfer Learning for Code Search

To our knowledge, there is only one previous work that is closely related to ours. Salza et al. [29] investigated the effectiveness of transfer learning for code search. They built a BERT-based model, which we refer to as cross-language CodeBERT, to examine how BERT pre-trained on source code of multiple languages can be transferred to code search tasks of another language. Their results show that the pre-trained model performs better than those without pre-training, and transfer learning is particularly effective in cases where a large amount of data is available for pre-training while data for fine-tuning is insufficient [29].

CroCS differs significantly from theirs. We employ a meta learning algorithm to explicitly adapt the parameters from source languages to the target domain, while their work directly fine-tunes the pre-trained model in the target language.

8 CONCLUSION

In this paper, we present CroCS, a cross-domain code search approach that reuses prior knowledge from large corpus of common languages to domain-specific languages such as SQL and Solidity. CroCS extends pre-trained models such as CodeBERT with meta learning. It employs a meta-learning algorithm named MAML which learns a good initialization of model parameters so that the model can quickly reach the optimal point in a new task with a few data samples. Experimental results show that CroCS achieves significant improvement in domain-specific code search, compared to "pre-training & fine-tuning" counterparts. In the future, we will investigate our method in more languages and other software engineering tasks.

Source code and datasets to reproduce our work are available at: https://github.com/fewshodetes/CDCS.

9 ACKNOWLEDGE

This work was sponsored by the National Natural Science Foundation of China under 62102244 and the CCF-Baidu Open Fund No.2021PP15002000. Xiaodong Gu is the corresponding author.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL].
[2] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcezer: a search engine for open source code supporting structure-based search. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. 681–682.
[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Dario Amodei, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL].
[4] Jose Cambraouero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 964–974.
[5] Casey Casalnuovo, Kenji Sagar, and Prem Devanbu. 2018. Studying the Difference Between Natural and Programming Language Corpora. arXiv:1806.02437 [cs.CL].
[6] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05). Vol. 1. IEEE, 539–546.
[7] Premkumar T. Devanbu. 1995. On ‘A framework for source code search using program patterns’. IEEE Transactions on Software Engineering 21, 12 (1995), 1099–1109.
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL].
[9] Zhangyi Feng, Dayu Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings. 1536–1547.
[10] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In International Conference on Machine Learning. PMLR, 1126–1135.
[11] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In Proceedings of the 2017 11th joint meeting on foundations of software engineering. 49–68.
[12] Jiaotu Gu, Yong Wang, Yun Chen, Kyunghyun Cho, and Victor Ok Li. 2018. Meta-learning for low-resource neural machine translation. arXiv preprint arXiv:1808.08437 (2018).
[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] Hamel Husain and Ho-Hsiung Wu. 2018. How to create natural language semantic search for arbitrary objects with deep learning. Retrieved November 5 (2018), 2019.
[15] Diedier P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG].
[16] Christoph Lange and Michael Kohlhaase. 2008. SWIM: A semantic wiki for mathematical knowledge management. In Emerging Technologies for Semantic Work Environments: Techniques, Methods, and Applications. IGI Global, 47–68.
[17] Otavio AL Lemos, Adriano C de Paula, Felipe C Zanchelli, and Cristina V Lopes. 2014. Theaurus-based automatic query expansion for interface-driven code search. In Proceedings of the 11th working conference on mining software repositories. 212–221.
[18] Wei Li, Haozhe Qin, Shuhun Yan, Beijun Shen, and Yuting Chen. 2020. Learning Code-Query Interaction for Enhancing Code Searches. In IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 115–126.
[19] Chao Liu, Xin Xu, David Lo, Zhawen Liu, Ahmed F. Hassan, and Shaping Li. 2020. Simplifying Deep-Learning-Based Model for Code Search. arXiv preprint arXiv:2005.14373 (2020).
[20] 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:1907.11692 [cs.CL].
[21] Merili Lu, Xiaohong Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 545–549.
[22] Fei Lv, Hongyu Zhang, Jiang-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 260–270.
[23] Antonio Masnou-Polo, Simonale Scalabrino, Nathan Cooper, David Nader Palacio, Denys Podshuvanyk, Rocco Oliveto, and Gabriele Barota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 336–347.
[24] Alex Nichol, Joshua Achiam, and John Schulman. 2018. On first-order meta-learning algorithms. arXiv preprint arXiv:1803.06999 (2018).
[25] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anhål, Alec Peltekian, and Yanfang Ye. 2020. CodeTf: Multi-task Learning with Code-Text Transformer. arXiv preprint arXiv:2005.08645 (2021).
[26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.
[27] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using siamese BERT-networks. arXiv preprint arXiv:1908.10084 (2019).
[28] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In Proceedings of the Conference on Empirical Methods in Natural Language Processing. PMLR, 1126–1135.
[29] Pasquale Salza, Christoph Schwierz, Jian Gu, and Harald C Gall. 2021. On the Effectiveness of Transfer Learning for Code Search. arXiv preprint arXiv:2108.05390 (2021).
[30] Jake Snell, Kevin Swersky, and Richard S. Zemel. 2017. Prototypical Networks for Few-shot Learning. arXiv:1703.05175 [cs.LG]
[31] Qianru Sun, Yaoyao Liu, Tat-Seng Chua, and Bernt Schiele. 2019. Meta-transfer learning for few-shot learning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 403–412.
[32] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. 2018. Learning to compare: Relation network for few-shot learning. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1199–1208.
[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in neural information processing systems. 5998–6008.
[34] Alex Wang and Kyunghyun Cho. 2019. BERT has a mouth, and it must speak: BERT as a markov random field language model. arXiv preprint arXiv:1902.04094 (2019).
[35] Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhen-chang Xing, and Yang Liu. 2022. Enriching query semantics for code search with reinforcement learning. Neurocomputing 145 (2022), 22–32.
[36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeTS: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. 8696–8708.
[37] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2–8.
[38] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 344–354.
[39] Zhen Yang, Jacky Keung, Xiao Ye, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. arXiv preprint arXiv:2103.07164 (2021).
[40] Ziyu Yao, Jayavardhan Reddy Peddamall, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In The World Wide Web Conference. 2203–2214.
[41] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 689–699.
[42] Wenpeng Yin. 2020. Meta-learning for Few-shot Natural Language Processing: A Survey. arXiv:2007.09604 [cs.CL]
[43] Tao Yu, Rui Zhang, Kau Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingming Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.
[44] Jakub Zakrzewski. 2018. Towards verification of Ethereum smart contracts: a formalization of core of Solidity. In Working Conference on Verified Software: Theories, Tools, and Experiments. Springer, 229–247.
[45] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: an overlapping-aware code retriever. In 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 883–894.