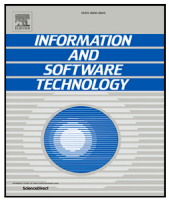
Information and Software Technology 166 (2024) 107365

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Improving domain-specific neural code generation with few-shot meta-learning

Zhen Yang [a], Jacky Wai Keung [b], Zeyu Sun [c], Yunfei Zhao [d,e], Ge Li [d,e,*], Zhi Jin [d,e,*], Shuo Liu [b], Yishu Li [b]

[a] School of Computer Science and Technology, Shandong University, Qingdao, China
[b] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[c] Zhongguancun Laboratory, Beijing, China
[d] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China
[e] School of Computer Science, Peking University, No. 5 Yiheyuan Road, Beijing, China

ARTICLE INFO

Keywords:
Code generation
Few-shot learning
Meta-learning
Transfer learning

ABSTRACT

Context: Neural code generation aims to automatically generate code snippets guided by Natural Language Descriptions (NLDs). In recent years, various neural code generation models for mainstream Programming Languages (PLs), such as Java and Python, have been proposed and demonstrated significant success in prior studies. Nonetheless, due to the scarcity of available training examples for some domain-specific PLs, such as Solidity, Bash, and Clojure, simply adopting previous neural models may lead to overfitting and inadequate learning.

Objective: To overcome this challenge, we propose MetaCoder, a novel meta-learning code generation approach that efficiently extracts general-purpose knowledge from a large-scale source language and rapidly adapts to domain-specific scenarios, even with relatively few samples.

Method: MetaCoder employs MAML, a powerful few-shot meta-learning method, to construct a transfer learning framework. This framework learns general-purpose knowledge from large-scale source languages and applies it in domain-specific target languages. To acquire more general-purpose knowledge, heterogeneous sub-tasks are constructed from the source language during the pre-training phase of MAML. As such, combining with CodeBERT and K-means, we design an unsupervised category assignment method for code generation samples, thereby exploiting the n-way k-shot rule to construct the heterogeneous sub-tasks. Consequently, MetaCoder can be applied to the code generation field.

Results: We evaluate MetaCoder with both tree-based (e.g., TreeGen) and sequence-based (e.g., CodeGPT) backbones on two domain-specific PLs, including Solidity and Bash. Extensive experiments demonstrate the superior performance of our approach compared to baselines and verified its capability of code generation visually in practice.

Conclusion: MetaCoder effectively extracts general-purpose knowledge from large-scale source languages, thereby enhancing model performance. Therefore, we highly recommend MetaCoder as a code generation approach for domain-specific PLs.

1. Introduction

Code generation techniques are designed for generating code automatically with the given Natural Language Descriptions (NLDs) of the code functionalities, which can largely lower down the programming threshold and improve the development efficiency. In recent years, numerous deep learning-based generative solutions have been proposed in this field and have seen significant progress, such as TranX [1], TreeGen [2], RECODE [3], and CodeT5 [4]. Nevertheless, none of them focus on domain-specific Programming Languages (PLs) with relatively fewer samples. A recent study [5] revealed that people had created 8,945 PLs till now, while Meyerovich et al. [6] found that the top 20 PLs account for 95% of the project repositories, showing that most of the PLs are niche, such as Solidity [7], Bash [8], and Clojure [9], and the necessity of automated code generation for domain-specific

* Corresponding authors.
E-mail addresses: zhenyang@sdu.edu.cn (Z. Yang), Jacky.Keung@cityu.edu.hk (J.W. Keung), szy_@pku.edu.cn (Z. Sun), zhaoyunfei@pku.edu.cn (Y. Zhao), lige@pku.edu.cn (G. Li), zhijin@pku.edu.cn (Z. Jin), sliu273-c@my.cityu.edu.hk (S. Liu), yishuli5-c@my.cityu.edu.hk (Y. Li).

https://doi.org/10.1016/j.infsof.2023.107365
Received 3 March 2023; Received in revised form 19 October 2023; Accepted 9 November 2023
Available online 11 November 2023

PLs is pervasive and long-standing. However, previous proposals were either dependent on massive training examples or homogeneous programming patterns. Without targeted improvement, directly migrating them to domain-specific PLs may easily overfit and hardly learn well.

One of the solutions is adopting transfer learning techniques to acquire prior knowledge from PLs with large-scale training samples, and then reusing them to fine-tune with domain-specific PLs. Hence, we propose to adopt Model-Agnostic Meta-Learning (MAML) [10], a powerful few-shot meta-learning method, to deal with this issue, and name our approach as MetaCoder. The core methodology of MAML in Natural Language Processing (NLP) is pre-training on batches of multiple heterogeneous sub-tasks constructed from the large-scale source language, by which the model can learn general-purpose knowledge efficiently across the sub-tasks and fastly adapt to the target language via fine-tuning. Sub-tasks, constructed via the n-way k-shot rule, are a series of subsets of source languages. For example, given a classification task with N classes, the n-way k-shot rule will randomly select n ($n \leq N$) classes with k samples for each from the source language to construct every sub-task in a heterogeneous manner. However, as a generative task, code generation carries a special classification paradigm compared with the usual classification. That is to say, different from usual classification tasks that predict one category per input, each input of the code generation task corresponds to predictions of a sequence of tokens, where each token is a classification on the whole vocabulary. Thus, it is difficult to distinguish the category of a code generation sample, and we cannot directly carry out the n-way k-shot rule to construct sub-tasks.

To this end, considering that we do not know the number of categories in the source language and the category that each sample belongs to, we design an unsupervised category assignment method in MetaCoder to allocate each code generation sample a category. Specifically, MetaCoder firstly uses CodeBERT [11] to vectorize the representation of each code generation sample in the source language, then adopts K-means clustering [12] to assign categories for samples based on their representations. Given the samples with assigned categories, MetaCoder constructs the heterogeneous sub-tasks via the n-way k-shot rule. Afterward, MetaCoder employs MAML to learn the general-purpose knowledge from those constructed sub-tasks and return good initialization parameters for the embedded code generation model to further fine-tune on the target language. Consequently, MetaCoder can be built as a whole and applied to domain-specific code generation.

We evaluate the effectiveness of MetaCoder on two domain-specific PLs, including Solidity and Bash. In detail, Solidity is the main-stream smart contract programming language [13], running on the top of Ethereum, one of the world's largest blockchain platforms. Bash is the default shell command language for the Linux operating system [8]. In this paper, we conduct extensive experiments on 7,878 Solidity samples and 10,592 Bash commands. Besides, we adopt two categories of code generation models as the backbones of MetaCoder. One is tree-based, such as TreeGen and TranX, and the other one is sequence-based, such as CodeGPT. Subsequently, we compare MetaCoder with three baselines, namely (1) a backbone model trained individually, (2) a backbone model with pre-training, and (3) a backbone model with MAML but constructing sub-tasks via random sampling. In the domain of Solidity, the experimental results quantitatively reveal that MetaCoder outperforms the within-domain counterparts by 33.21% in terms of BLEU and 21.81% in terms of ROUGE-L on average across different backbones. In the domain of Bash, MetaCoder with different backbones outperforms the within-domain counterparts by 16.17% and 13.32% on average in terms of BLEU and ROUGE-L, respectively. Besides, the instance analysis also demonstrates the powerful generative ability of MetaCoder towards domain-specific PLs in practice.

The contributions of this paper can be summarized in three-fold:

- We propose MetaCoder, a code generation approach enhanced with few-shot meta-learning, to migrate general-purpose knowledge from data-rich PLs to domain-specific PLs when generating code snippets. To the best of our knowledge, this is the first code generation work focusing on domain-specific PLs.
- We design an automatic unsupervised category assignment method in MetaCoder for sub-task construction. This improvement, for the first time, makes meta-learning applicable to the code generation field.
- We evaluate MetaCoder with both tree-based (e.g., TreeGen) and sequence-based (i.e., CodeGPT) backbones on 7,878 Solidity samples and 11,549 Bash samples. Extensive experiments demonstrate the effectiveness of MetaCoder against baseline models.

The remainder of this paper is organized as follows. Section 2 presents the background and preliminaries. Section 3 elaborates on the proposed approach, including the unsupervised category assignment, meta-learning, and fine-tuning. Section 4 introduces the data collection and preprocessing. Sections 5 and 6 discuss the experimental design and results. Section 7 further discusses some typical cases of MetaCoder and the performance comparison between MetaCoder with the latest techniques. Section 8 demonstrates the threats to validity. Section 9 introduces the related work of this paper. Finally, Section 10 summarizes the paper.

2. Background

2.1. Code generation models

Code generation models mainly consist of two categories, i.e., sequence-based and tree-based [14].

2.1.1. Sequence-based code generation models

Sequence-based code generation models are mainly composed of code-related pre-trained language models, such as GPT-3.5,[1] CodeGen [15], Codex [16], and CodeGPT [17]. These models are initiated from pre-trained language models and further optimized based on a series of code-related tasks, such as next-token prediction language modeling on code [18]. Therefore, the outputs of these models are sequences of tokens when carrying out the code generation task. Hence, they cannot ensure the grammar correctness of the generated code. Since most of the models of this category carry billions of parameters, further training on some of them is unrealistic. Based on our affordable computational overhead, we select CodeGPT as the backbone model of MetaCoder in this category. In detail, CodeGPT is initialized from GPT-2 [19] with the structure of the Transformer's decoder. Specifically, CodeGPT has two versions, one is optimized from Java, and the other is optimized from Python. Since we adopt Java as the source language in this paper, we select the former one for experiments.

2.1.2. Tree-based code generation models

Compared with sequence-based code generation models, tree-based code generation models can generate grammatically correct code snippets because they consistently maintain a partial Abstract Syntax Tree (AST) of the already-generated code during the whole inference process. In this paper, we select two typical tree-based code generation models, i.e., TreeGen [2], and TranX [1], to study the performance of our framework.

- TreeGen: It exploits the training dataset to construct the vocabulary of grammar rules. It consists of an NL reader for encoding input NLDs, an AST reader for encoding the partial AST of the already-generated code, and a decoder for producing a grammar rule at each time step. TreeGen adopts many complex structures for feature learning, such as multi-head attention [20], tree convolution, separable convolution [21], and gating mechanism, thus, obtaining a competitive performance on some widely studied datasets, including HearthStone, ATIS, and GEO.

[1] https://platform.openai.com/docs/models/gpt-3-5

- TranX: It utilizes the Abstract Syntax Description Language (ASDL) [22] to depict the syntax of programming languages. It is a sequence-to-sequence structure with Long Short-Term Memory (LSTM) as its backbone. Similarly, the input is a piece of NLD, while the output is a sequence of grammar rules. Datasets, including GEO, ATIS, Django, and WikiSQL, have been tested via TranX. And its intuitive structure and good performance attracted broad extension in recent years, such as ML-TRANX [23], ReCode [3], ASED [24], and TRANX-RL [25].

2.2. Few-shot Learning and meta-learning

Few-Shot Learning (FSL) is a kind of machine learning technology aiming to fastly adapt trained models to new tasks with limited examples [26]. According to Wang et al. [27], existing methods of FSL can be categorized into three-fold, including (1) data augmentation to enrich the training set, (2) constraining hypothesis space to improve the optimization efficacy, and (3) altering search strategy in hypothesis space.

Meta-learning is a typical FSL method of the third category, which aims to learn good initialization parameters from a large-scale dataset for fine-tuning, thereby guiding the model to quickly adapt to the target task with few samples. Different from conventional transfer learning technologies, meta-learning trains the model on a variety of heterogeneous sub-tasks, and its model parameters are updated based on the query set (for the validation purpose) of each sub-task to ensure the generality to different tasks, which is also known as "learning to learn" [10]. In recent years, several representative meta-learning methods have been continuously proposed, such as Model-Agnostic Meta-Learning (MAML) [10], and Reptile [28].

In this paper, we adopt MAML as the few-shot meta-learning method to transfer general-purpose knowledge from data-rich PLs to domain-specific PLs. In general, MAML firstly constructs various sub-tasks from a large-scale dataset according to the n-way k-shot rule to prepare its pre-training pool. For example, given a classification task with N classes, the sampling method of MAML randomly selects n ($n \leq N$) classes with k samples for each to construct each sub-task. Subsequently, in each optimization step, a meta learner updates the model parameters based on several randomly selected sub-tasks, meaning that multiple sub-tasks jointly determine the gradient descent direction. According to [10], the intuition behind this approach is that (1) MAML carries an assumption that the pre-training data follows a distribution $P(T)$ over K tasks $\{T_1, \ldots, T_K\}$, where each T_i is capable of standing for a specific machine learning task with heterogeneous characteristics. On the other hand, (2) some prior knowledge is more general and transferable than others among sub-tasks, which may also be useful in new tasks. Thus learning from multiple sub-tasks instead of homogeneous training batches like normal pre-training can guide the model to acquire more general-purpose knowledge, such that the model can adapt to the target task more quickly.

3. Methodology

In this section, we elaborate on the details of our proposed framework.

3.1. Overview

Fig. 1 demonstrates the architecture of our proposed approach, which includes a training stage and an inference stage. More specifically, the training stage consists of four phases, i.e., (1) conducting unsupervised category assignment for the large-scale source language, e.g., Java. (2) constructing heterogeneous sub-tasks via the n-way k-shot rule, (3) carrying out meta-learning with a code generation model on the constructed sub-tasks to learn general-purpose knowledge of the source language, and (4) fine-tuning the code generation model with the meta-learned initialization parameters to implement the fast adaption to the target language, e.g., Solidity. Subsequently, given an NLD (i.e., a user requirement), we exploit the fine-tuned code generation model to generate the corresponding code snippet in the inference stage. The details of each phase are elaborated in the following sections.

3.2. Unsupervised category assignment

Tasks such as code generation cannot directly partition samples into sub-tasks via the n-way k-shot rule. Because code generation predicts a sequence of tokens for each sample, and each of its predictions is a classification among the whole vocabulary. In other words, given a sample of code generation whose target is a sequence with a length of G, it needs to be classified by G times. Thus we cannot simply designate a category for a code generation sample, leading to difficulty carrying out the n-way k-shot rule to construct heterogeneous sub-tasks. Similar dilemmas also appear in other fields of software engineering, such as code search with meta-learning [29], where they just randomly partition the code search samples into sub-tasks. However, this naive method is unreasonable because it will generate each sub-task in the same distribution, causing sub-tasks to present homogeneous properties. As such, MAML may hard to learn the general-purpose knowledge from those sub-tasks. To this end, we propose an unsupervised category assignment method to solve this problem, which consists of two steps as shown in Fig. 1, i.e., Step 1 extracts representations for code generation samples via CodeBERT [11], while Step 2 employs K-means clustering [12] to assign categories.

For Step 1, we argue that the categories of the samples in code generation should be determined by both source code and their corresponding NLDs. Besides, CodeBERT is a powerful pre-trained code representation model and is capable of encoding source code and NLDs jointly. Hence, in this work, we utilize CodeBERT to encode both source code and their corresponding NLDs of the source language as the representation prepared for clustering.

For Step 2, after we obtain the initial representations of the source language, we conduct max-pooling on each of them for dimension reduction, thereby fitting the input requirement of K-means clustering. We apply K-means for clustering in this paper because K-means is one of the most widely used and highly effective clustering algorithms. Furthermore, for large-scale data in practical environments, complicated clustering algorithms, such as Mean Shift [30], DBSCAN [31], and EM [32], are hardly applied due to their significant computational overhead. However, to exert K-means, the number of clusters should be determined in advance. Here we adopt elbow rule [33] to decide the best clustering number, where the Sum of Squared Error (SSE) is exploited as the selection indicator. As defined in Eq. (1), a squared error is computed by the square of the distance between each point (p_{ij}) and its centroid (m_i), and C_i denotes the ith cluster in C. Normally, with the growth of the clustering number, SSE decreases gradually, and the sharp elbow of the diagram can be selected as the best clustering number.

$$SSE = \sum_{C_i \in C} \sum_{p_{ij} \in C_i} |p_j - m_i|^2 \tag{1}$$

3.3. Meta-learning

Next, we adapt MAML, one of the few-shot meta-learning methods as introduced in Section 2.2, into our approach to learn general-purpose knowledge from the source language. As shown in Algorithm 1, we have prepared N clusters of code generation samples of source language from the previous phase. Then, we select a code generation model (e.g., TreeGen) as the backbone of MAML. Subsequently, we construct L heterogeneous sub-tasks via the n-way k-shot rule as specified in Line 1. According to [10], each sub-task only requires a small number of training samples from the source language. Particularly, Phase 3

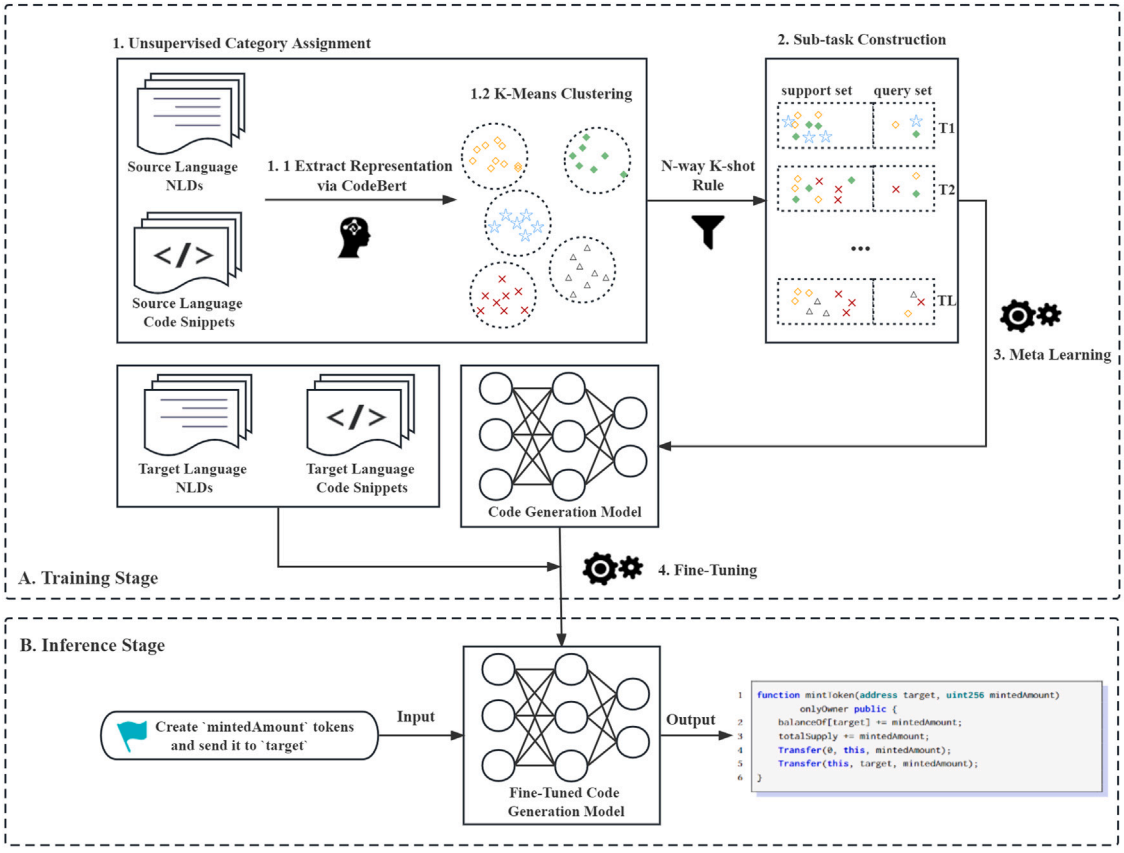


Fig. 1. The overall framework of our approach.

Algorithm 1 Meta-Learning for Code Generation

Require: α, β: learning rates, M: update steps; I: maximum pre-training steps; i: current step.
Require: N clusters partitioned from the source language, where each cluster contains at least K samples (i.e., code-NLD pairs). Each cluster is regarded as a class.
Require: Choose a code generation model f (i.e., TreeGen) with parameters θ.
1: Randomly sampling $n \leq N$ classes, each with $k \leq K$ samples to construct a sub-dataset D_j, with the jth sub-dataset assigned for the sub-task T_j, totally constructing L sub-datasets (sub-tasks).
2: Randomly initialize θ
3: $i \leftarrow 0$
4: **while** $i \leq I$ **do**
5: $i \leftarrow i + 1$
6: $D_l \leftarrow$ Randomly select $l \leq L$ sub-datasets
7: **for each** $D_j \in D_l$ **do**
8: Initialize a local $\theta' \leftarrow \theta$
9: Split D_j into $\{D_j^{spt}, D_j^{qry}\}$
10: **while** $m \leq M$ **do**
11: Run $f_{\theta'}$ on D_j^{spt} and update θ': $\theta' \leftarrow \theta' - \alpha \nabla_{\theta'} L(f_{\theta'}(D_j^{spt}))$
12: **end while**
13: $L(f_{\theta'}(D_j^{qry})) \leftarrow$ Evaluate $f_{\theta'}$ on D_j^{qry}, return loss
14: **end for**
15: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{j=1}^{l} L(f_{\theta'}(D_j^{qry}))$
16: **end while**
17: return θ

in Fig. 1 is a 3-way 4-shot example, where we randomly select three categories, each with four samples. For each sub-task j, it contains a support set D_j^{spt} and a query set D_j^{qry}. The former is for training, while the latter is for validation.

Afterward, we embed the selected code generation model into MAML. The key idea of MAML is to learn a set of good initialization parameters θ for the embedded model from a series of heterogeneous sub-tasks. In the context of code generation, we hope to transfer the coding capability from the large-scale source language (e.g., Java or Python) to the domain-specific target language (e.g., Solidity or Bash).

Specifically, MAML employs an inner loop to update the local parameters θ' which is for sub-task-specific learning (Line 10–12), and an outer loop to update the meta parameters θ, which is for general-purpose knowledge learning (Line 4–16). Each inner loop contains M steps of updating for local parameters. More formally, the updating of local parameters in each step can be defined as below, where α is the learning rate of the inner loop, $L(f_{\theta'}(D_j^{spt}))$ denotes the loss of code generation model (f) with parameters (θ') on support set of jth sub-task (D_j^{spt}), and $\nabla_{\theta'}$ represents the gradient on θ'.

$$\theta' \leftarrow \theta' - \alpha \nabla_{\theta'} L(f_{\theta'}(D_j^{spt})) \tag{2}$$

For the updating of meta parameters, we present it in Eq. (3), where β denotes the learning rate of the outer loop, and $\sum_{j=1}^{l} L(f_{\theta'}(D_j^{qry}))$ denotes the sum of losses of selected sub-tasks on their corresponding query set (D_j^{qry}).

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{j=1}^{l} L(f_{\theta'}(D_j^{qry})) \tag{3}$$

After pre-training up to the predefined maximum iteration, we stop MAML and return meta parameters θ for the preparation of fine-tuning in the next step.

Table 1
Data statistics of Java, Solidity, and Bash.

Java				
Partition	Total[b]	Total[a]	# Avg. Lines	# Avg. Tokens
Train	454,451	361,940	8.626	83.92
Valid	15,328	11,834	7.492	76.39
Test	26,909	21,675	8.628	86.37
Solidity				
Partition		Total	# Avg. Lines	# Avg. Tokens
Train		6878	5.835	63.34
Valid		500	6.154	66.41
Test		500	5.892	62.93
Bash				
Partition		Total	# Avg. Lines	# Avg. Tokens
Train		9592	1.029	10.32
Valid		500	1.040	11.03
Test		500	1.038	10.55

[Φ] Total[b] and Total[a] denote the total volume of the Java dataset before and after preprocessing.

3.4. Fine-tuning

In the fine-tuning phase, we activate the code generation model with the learned meta parameters θ and fine-tune it with the normal method on the target samples of domain-specific PLs. It is noticeable that MAML is not used in this phase because good initialization parameters are supposed to be obtained during the meta-learning phase.

4. Dataset and preprocessing

4.1. Dataset

In this paper, we select two domain-specific PLs for experiments, including Solidity and Bash. The Solidity dataset was released by Hu et al. [34], crawled from Etherscan.[2] It is worth noting that Solidity provides an illustration of comment, known as Ethereum Natural Language Specification Format (NatSpec), and this dataset uses comments behind *@notice* in NatSpec as the NLDs, called user notice. User notices only record the functionalities of the corresponding code snippets without any implementation details, which is used for facilitating end users' code comprehension. Another kind of NLD in Solidity smart contracts is annotated as *@dev* in NatSpec, which is used for developers and provides extra code implementation details without any transaction context. We argue that adopting user notices as NLDs can reduce the programming threshold of Solidity to the maximum extent because both developers and end users can easily comprehend and express their intents in such a way. The Bash dataset adopted in this paper was published by Yu et al. [35] and mainly collected from web pages such as question-answering forums and tech blogs. Each sample contains a Bash command and a corresponding NLD.

For the source language, we adopt the Java dataset of CodeSearchNet [36] for experiments. There are two reasons. Firstly, Java and Solidity are relatively similar. For example, they are both object-oriented and high-level PLs. Researchers have exploited transfer learning to convey the knowledge from Java to Solidity in code summarization task [34] and presented impressive performance. Therefore, we can explore the effectiveness of MetaCoder between similar PLs (i.e., adopting Java as the source PL while Solidity as the target PL). Secondly, Java is dissimilar to Bash both syntactically and lexically, as the former is platform-agnostic object-oriented PL while the latter is scripting PL running on the Linux operating system. According to the experiments on MetaCoder between Java (as the source PL) and Bash (as the target PL), we can uncover whether and how much general-purpose knowledge can be transferred via MetaCoder between PLs that are not similar.

4.2. Preprocessing

4.2.1. Preprocessing for Java

Towards the NLDs in the Java dataset, we truncate them at the first period or line break, as leading sentences typically describe the functionalities of methods. Then, we remove those NLDs with less than three English words for their descriptions are not informative enough. Towards the corresponding code snippets, we first remove the inline comments. Then, due to the AST requirement of those tree-based code generation models, e.g., TreeGen and TranX, we employ tree-sitter,[3] an AST parser tool-kit, to extract the Java AST of each code snippet. Finally, for both NLDs and code snippets, we change their tokens to lowercase format, such that we can shrink the vocabulary size and reduce differences across languages. After the above series of operations, the training set remains 361,940 samples, the validation set remains 11,834 samples, and the testing set remains 21,675 samples. Table 1 demonstrates their data volume and corresponding average amounts of lines and tokens per data sample.

4.2.2. Preprocessing for solidity and bash

Given the Solidity dataset has been carefully processed by Hu et al. here we directly parse the code snippets to their corresponding ASTs. Since tree-sitter does not provide the tool to parse Solidity source code, we adopt Solidity-Parser,[4] a widely used AST parser for Solidity [34,37,38], as a substitution. However, since we adopt different AST parsers for source and target languages, their node names that share the same meanings will more or less have some inconsistencies. For example, the root node of Java is "method_declaration" while that of Solidity is "function_definition". This phenomenon will definitely damage the transfer efficacy across languages because, for example, models that already learned the meaning of "method_declaration" must re-learn "function_definition" that carries the same meaning. Simply sharing the vocabulary of both languages is useless because representations of a language's particular nodes will not be invoked or learned during the training on another. Hence, referring to the AST nodes of Java, we change those with the same meanings in Solidity accordingly, which means, for example, we use "method_declaration" in Solidity ASTs instead of "function_definition" to ensure consistency. Next, since string literals in source code do not affect the program execution, we simplify them as "STR" when generating code for Solidity. Besides, for both NLDs and code snippets, we also change their tokens to lowercase format like Section 4.2.1. Towards the Bash dataset, we follow the same process of Solidity, except the AST parser used here is tree-sitter.[3] For both the Solidity and Bash datasets, we randomly select 500 samples for validation and testing, respectively. The rest of their samples are applied for training. Details can be found in Table 1, which demonstrates the partition result of Solidity and Bash, along with their corresponding average numbers of lines and tokens per sample, respectively.

5. Experimental setup

5.1. Research questions

This paper mainly focuses on the following three research questions and designs our experiments accordingly.

[2] https://etherscan.io/

[3] https://tree-sitter.github.io/tree-sitter/

[4] https://github.com/federicobond/solidity-parser-antlr

- **RQ1: Whether MetaCoder is effective for generating domain-specific PLs?**
  To verify the effectiveness of MetaCoder, we select two domain-specific PLs for experiments, including Solidity and Bash. In addition, we take Java as the source language to learn general-purpose knowledge. Subsequently, we separate our experiments into two stages. In stage 1, we adopt TranX and TreeGen as the backbone of MetaCoder, respectively, thereby exploring its performance improvement on tree-based models. In stage 2, we select CodeGPT as the backbone of MetaCoder to further verify its effectiveness on sequence-based models. In this way, we can comprehensively evaluate whether the effectiveness of MetaCoder among different backbones is consistent. For each experiment, we compare MetaCoder with various baseline models presented in Section 5.2 regarding the metrics introduced in Section 5.4.
- **RQ2: How does the number of pre-training steps during the meta-learning influence the effectiveness of MetaCoder?**
  The number of pre-training steps I is an important hyper-parameter of MetaCoder. Pre-training with excessively large I might be easy to make the model overfit to the source language, while the model may also be possible to underfit with a small I. Both situations will damage the efficacy of meta-learning. To answer RQ2, we configure a series of I for meta-learning. In addition, we also select a tree-based model (i.e., TreeGen) and a sequence-based model (i.e., CodeGPT) for experiments, thereby exploring the influence discrepancy among different kinds of backbones. Since TreeGen and CodeGPT early stop their pre-training phase mostly before 4500 steps based on our observation, we set $I = \{500, 1500, 2500, 3500, 4500\}$ to investigate the influence of pre-training steps. Besides, pre-training steps under different clustering numbers may also produce different influences on the effectiveness of MetaCoder. Therefore, we test the variation of pre-training steps jointly with different clustering numbers (i.e., $N = \{1, 5, 10, 15, 20, 25\}$). Based on the above experimental setting, we can obtain a series of corresponding initialization parameters. Afterward, we adopt each group of learned initialized parameters as the starting point to fine-tune their associated backbone models and evaluate their performances.
- **RQ3: How does the clustering number affect the performance of MetaCoder?**
  Initially, the clustering number N has been determined by the elbow rule. However, we still cannot know how the code generation performance varies with other Ns, and whether the N selected via the elbow rule is also suitable for code generation. Hence, we continue our investigation on clustering numbers based on the experimental results of RQ2, by which situations under different backbone models and pre-training steps can be considered.

5.2. Comparison methods

In this paper, we adopt the two tree-based models and one sequence-based model as the backbone of MetaCoder, respectively. Then, for each of them, we compare MetaCoder with three baselines:

(1) **Individual:** A backbone model trained individually, referring to training the backbone model only with the dataset of each target language.
(2) **Pre-training(N):** A backbone model with pre-training, referring to pre-training the backbone model with the normal approach on the Java dataset first, then fine-tuning it on the dataset of each target language based on the learned initialization parameters of the pre-training phase.
(3) **Meta-Learning(R):** A backbone model with MAML but constructing sub-tasks via random sampling, referring to constructing sub-tasks of meta-learning via random sampling, then pre-training the backbone model with meta-learning on the Java dataset, finally, fine-tuning it on the dataset of each target language based on the learned initialization parameters of the meta-learning phase.

5.3. Implementation details

For TreeGen and TranX, We try our best to re-implement both of them based on their publicly released source code and adapt them to our datasets. For CodeGPT, we follow the instructions on Hugging Face[5] to instantiate the model and load the pre-trained parameters. In addition, we craft an input template ("<NLD>\nwrite a <TARGET LANG> code snippet based on the above natural language description".) to guide CodeGPT to generate code snippets of different PLs. Particularly, "<NLD>" is a requirement of a code snippet which is the same as the input of TreeGen and TranX. "<TARGET LANG>" is the name of the target PL. We adopt this setting because we find CodeGPT has no idea which PL we hope to generate if only given a piece of requirement. Thus we use this template to instruct CodeGPT on generating our designated PLs. Besides, for each model we mentioned above, their hyper-parameter settings are kept the same as recorded in their original papers [1,2,17], which are also the settings configured in the fine-tuning phase.

For meta-learning, when constructing sub-tasks via the n-way k-shot rule, we set the n to five and the k to two. The support set and query set are assigned 50% of the data in each sub-task, respectively. Four sub-tasks constitute a pre-training batch. As such, we have 20 samples for training and 20 samples for validation per batch. The learning rates of the inner (α) and outer (β) loops are set to 1e-2 and 1e-3, respectively, when embedding TranX and TreeGen. When testing on CodeGPT, they are fixed to 5e-5 and 5e-6, respectively. The number of update steps (M) is set to four.

For pre-training with the normal approach, to try to make its setting consistent with meta-learning for a fair comparison, we set the learning rate to 1e-3 and 5e-6 for tree-based and sequence-based backbones, respectively. The batch size is set to 20.

5.4. Evaluation metrics

Since all ASTs can be fully recovered to their corresponding code snippets, we evaluate the model performance by directly comparing generated AST sequences and reference AST sequences. In this paper, we adopt BLEU [39] and ROUGE-L [40] as our evaluation metrics, which are widely used in code/text generation field [2,41–45].

- **BLEU.** BLEU-N computes the N-gram precision of a candidate sequence to the reference sequence. We report a composite BLEU which is the average of BLEU-1, BLEU-2, BLEU-3, and BLEU-4. Besides, we adopt the smoothing-1 method [46] to alleviate the computation bias in case non-overlapping N-grams appear.
- **ROUGE-L.** ROUGE-L computes the matching degree on the longest common sequences. Compared with BLEU, it is a recall score, which evaluates how much of the ratio of the reference sequences appear in the candidate sequences.

Since the outputs of sequence-based models (e.g., CodeGPT) are a series of code tokens, which cannot guarantee syntactic correctness. We evaluate CodeGPT with an extra metric, namely the Syntactic Correctness Rate (SCR), which measures the rate of syntactically correct outputs. Besides, when computing BLEU and ROUGE-L for CodeGPT, we set scores of syntactically incorrect outputs as zero, because their outputs cannot be parsed to ASTs for evaluation.

[5] https://huggingface.co/

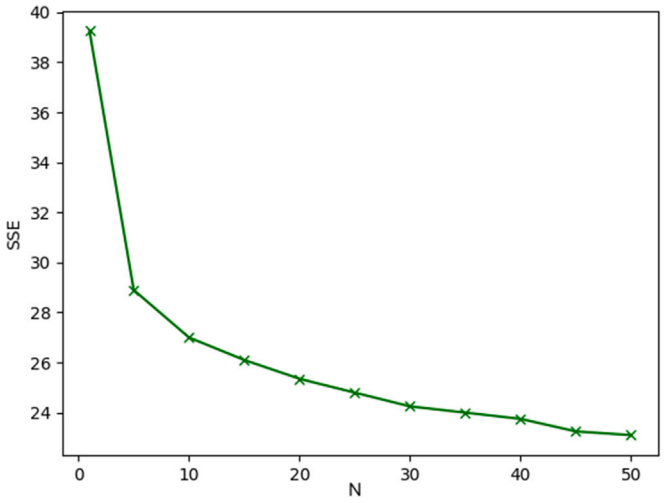


Fig. 2. Experimental result of elbow rule on Java dataset.

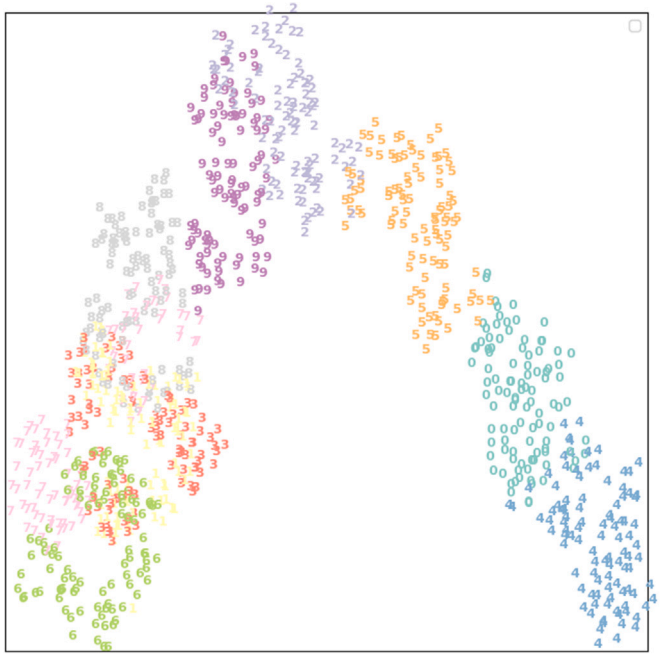


Fig. 3. Clustering results visualization (N=10).

6. Experimental results

6.1. Effectiveness of MetaCoder (RQ1)

In order to evaluate the performance of MetaCoder, we first implement the K-means clustering on the whole Java dataset and adopt the elbow rule to determine the best clustering number. Fig. 2 demonstrates the experimental result of the elbow rule with clustering number N from 5 to 50, where the N grows by 5 at each step. The first point (N=1) denotes no clustering. Based on our observation in Fig. 2, N=10 is selected as the best clustering number because the SSE no longer significantly decreases when the N grows to 10. Subsequently, we take the category assignment result of code generation samples under $N = 10$ to construct sub-tasks and evaluate MetaCoder as elaborated in Section 5.1. Fixed the setting of N=10, we further randomly select a hundred samples for each cluster and visualize their distributions by applying t-SNE [47] in Fig. 3, where clusters are numbered from 0 to 9 with diverse colors. Apparently, although the cluster 1 and 3 are overlapped, the clustering results with K-means (N=10) can clearly partition most of the clusters.

Table 2 demonstrates the experimental results of each method towards Solidity in Stage 1, where the first three, i.e., (1) Individual, (2) Pre-Training(N), and (3) Meta-Learning(R), represent the three baselines we mentioned in Section 5.2 in order. The pre-training step of Pre-Training(N), Meta-Learning(R), and MetaCoder are fixed to 500 in this stage, thereby ensuring comparison fairness. As can be seen, regardless of embedding either TreeGen or TranX for MetaCoder, it consistently achieves the best performance. Specifically, MetaCoder with TreeGen outperforms baselines on average by 26.32% and 7.98% in terms of BLEU and ROUGE-L, respectively. Meanwhile, MetaCoder with TranX also outperforms baselines on average by 11.67% and 4.58% in terms of each metric, respectively. Based on Table 3, we can observe a similar result. MetaCoder still performs best against baselines when generating Bash commands. Specifically, MetaCoder with TreeGen outperforms baselines on average by 10.28% and 6.62% in terms of BLEU and ROUGE-L, respectively. In addition, MetaCoder with TranX outperforms baselines on average by 24.41% and 28.21% in terms of each evaluation metric. The superiority of MetaCoder among baselines can attribute to our adoption of MAML and our adaptation of MAML in the code generation field because we semantically partition the code generation samples in the source language via clustering, thereby constructing heterogeneous sub-tasks for each backbone model to learn the general-purpose knowledge via MAML.

It is noticeable that Meta-Learning(R) performs slightly poorer than Pre-Training(N), as shown in Table 2 and 3, showing that random sampling for sub-task construction dramatically damages the performance of meta-learning. A potential explanation is random sampling makes each sub-task homogeneous, reducing the learned generalization capability of meta-learning. Besides, methods equipped with transfer learning, i.e., Pre-Training(N), Meta-Learning(R), and MetaCoder, all perform better than the individual backbone models, proving that they indeed learn the prior knowledge more or less from the source language. In general, it is natural that embedding with TreeGen performs better than that with TranX because TreeGen carries a much more complex network structure for program feature learning compared with TranX, as we mentioned in Section 2.1.2.

Table 4 demonstrates the experimental results of MetaCoder with CodeGPT on Solidity and Bash, respectively. The pre-training steps of Pre-Training(N), Meta-Learning(R), and MetaCoder are fixed to 1500 in this stage, thereby ensuring comparison fairness. As can be seen, training CodeGPT individually on domain-specific PLs performs worst, and even cannot generate syntactic correct code snippets. Interestingly, we find the generated code snippets of this method are mixed formats of source and target languages. Thus, the syntax is definitely incorrect, and we ignore them during the comparison. Compared with the rest of the methods, MetaCoder still manifests its superiority. To be specific, MetaCoder outperforms baselines on average by 61.66%, 52.87%, and 48.36% in terms of BLEU, ROUGE-L, and SCR, respectively, when generating Solidity code snippets. Besides, When generating Bash code snippets, although there is a little decline in SCR, MetaCoder still outperforms baselines on average by 13.81% and 5.14% in terms of BLEU and ROUGE-L, respectively.

Afterward, we make comparisons between tree-based and sequence-based models. Based on Table 2, 3, and 4, we find that, in general, MetaCoder with tree-based backbones performs better on generating Solidity code snippets, while MetaCoder with sequence-based models carries its superiority when generating Bash commands. A potential explanation is that tree-based models have smaller and more accurate search space for their rule restrictions when code generation. Thus they present better generation capability on relatively complicated PLs, such as Solidity. On the other hand, sequence-based models, such as CodeGPT, carry much more neurons and have more powerful generality, but cannot guarantee syntactic correctness. Thus, they

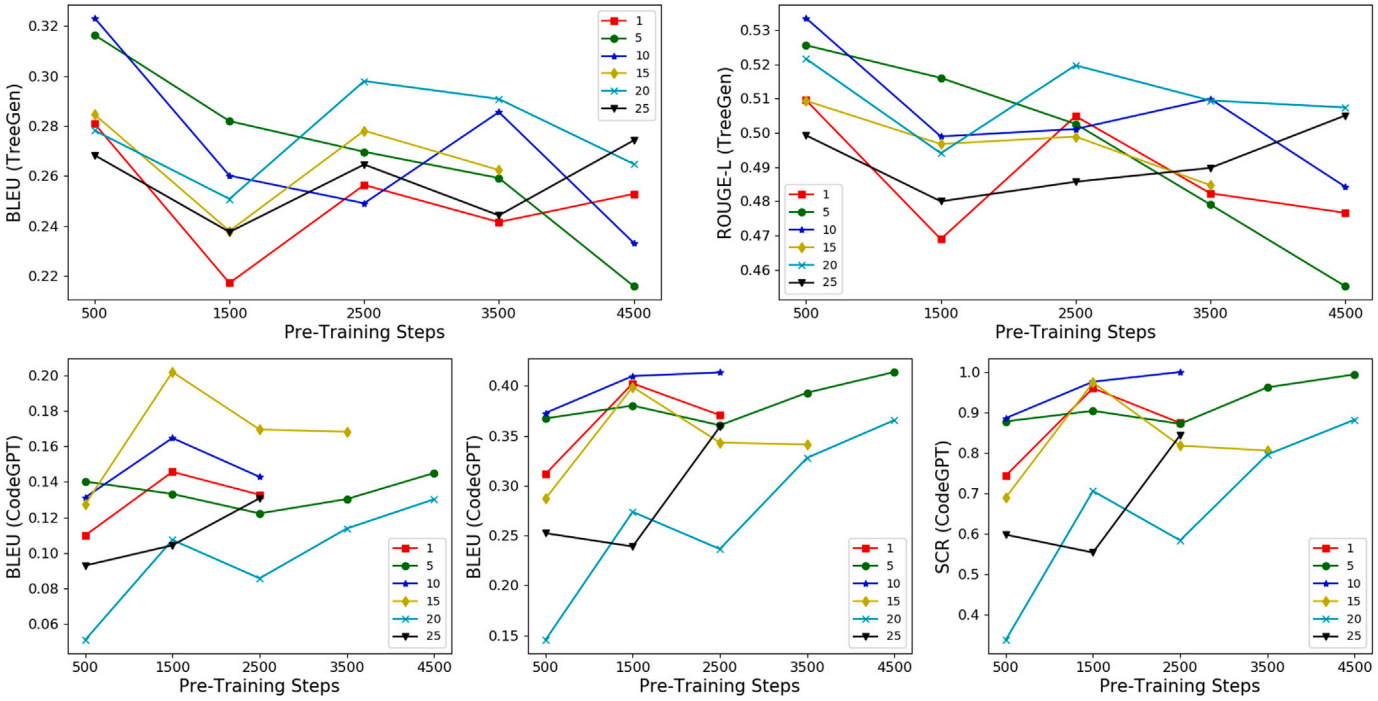


Fig. 4. The Influence of pre-training steps on MetaCoder.

Table 2
Stage 1: Experimental results of each method towards Solidity.

Method	TreeGen		TranX	
	BLEU	ROUGE-L	BLEU	ROUGE-L
Individual	21.28%	46.48%	18.89%	40.50%
Pre-Training(N)	28.84%	51.06%	21.29%	41.93%
Meta-Learning(R)	28.08%	50.95%	19.23%	40.67%
MetaCoder	**32.31%**	**53.35%**	**22.02%**	**42.90%**

Table 3
Stage 1: Experimental results of each method towards Bash.

Method	TreeGen		TranX	
	BLEU	ROUGE-L	BLEU	ROUGE-L
Individual	17.45%	58.64%	8.17%	44.50%
Pre-Training(N)	16.28%	56.76%	10.97%	55.12%
Meta-Learning(R)	15.16%	61.69%	9.44%	43.12%
MetaCoder	**17.94%**	**62.87%**	**11.68%**	**60.28%**

achieve superior performance on PLs (e.g., Bash) with relatively simple syntactic complexity. In fact, if the computational overhead is affordable, we can extend our experiments to some more powerful sequence-based models that are optimized based on GPT-3 [48], such as Codex [16] and CodeGen [15], we believe the performance can be further improved. In addition, no matter whether MetaCoder performs on Solidity or Bash, it consistently improves code generation capabilities towards these domain-specific PLs. But, referring to the improvement among different backbones, in general, domain-specific PLs carrying similar characteristics to the source language obtain relatively more improvements.

6.2. Influence of the pre-training steps (RQ2)

In this section, we discuss how the number of pre-training steps during the meta-learning phase affects the performance of MetaCoder. Fig. 4 demonstrates the performance variation of MetaCoder with TreeGen and CodeGPT under different numbers of pre-training steps, where each line represents a fixed clustering number setting. Points not shown in the figures denote an early stopping at a corresponding pre-training step. Therefore, the associated experimental results are unavailable. For MetaCoder embedded with TreeGen, BLEU and ROUGE-L scores overall decline with the increment of the pre-training steps during the meta-learning phase. Because the data volume of domain-specific PLs is relatively limited, pre-training with excessive steps in the source language overwhelms MetaCoder. However, we also find that such a declining trend becomes gradually not obvious as the clustering number increases. A potential explanation is that since the best clustering number is around N=10, with the clustering number increasing, MetaCoder with TreeGen needs more pre-training steps to learn the general-purpose knowledge, thus postponing the overwhelming efficacy of the source language. Interestingly, for MetaCoder embedded with CodeGPT, both BLEU and ROUGE-L scores increase as a whole when we provide more pre-training steps in the meta-learning phase. We attribute this phenomenon to the fact that CodeGPT needs to adapt to the input context with more pre-training steps. Because we feed CodeGPT with the template of "<NLD>\nwrite a <TARGET LANG> code snippet based on the above natural language description". as we mentioned in Section 5.3, which is different from its original training paradigm.

6.3. Influence of the clustering number (RQ3)

In this section, we explore the influence of the clustering number on MetaCoder. Although we have determined the best clustering number in Section 6.1 via the elbow rule, this result is not guided by the code generation performance. We still cannot know whether a good clustering from the view of the elbow rule is suitable for meta-learning adapted for code generation. Hence, the exploration of the clustering number becomes necessary.

The investigation of clustering numbers is based on the experimental results of RQ2, where we reformulate their manifestation in Fig. 5. Each line denotes a score variation of an evaluation metric with different clustering numbers under a fixed pre-training step. Points not shown in the figures denote an early stopping at a corresponding pre-training step. Therefore, the associated experimental results are unavailable. It is noticeable that the clustering number equals one means

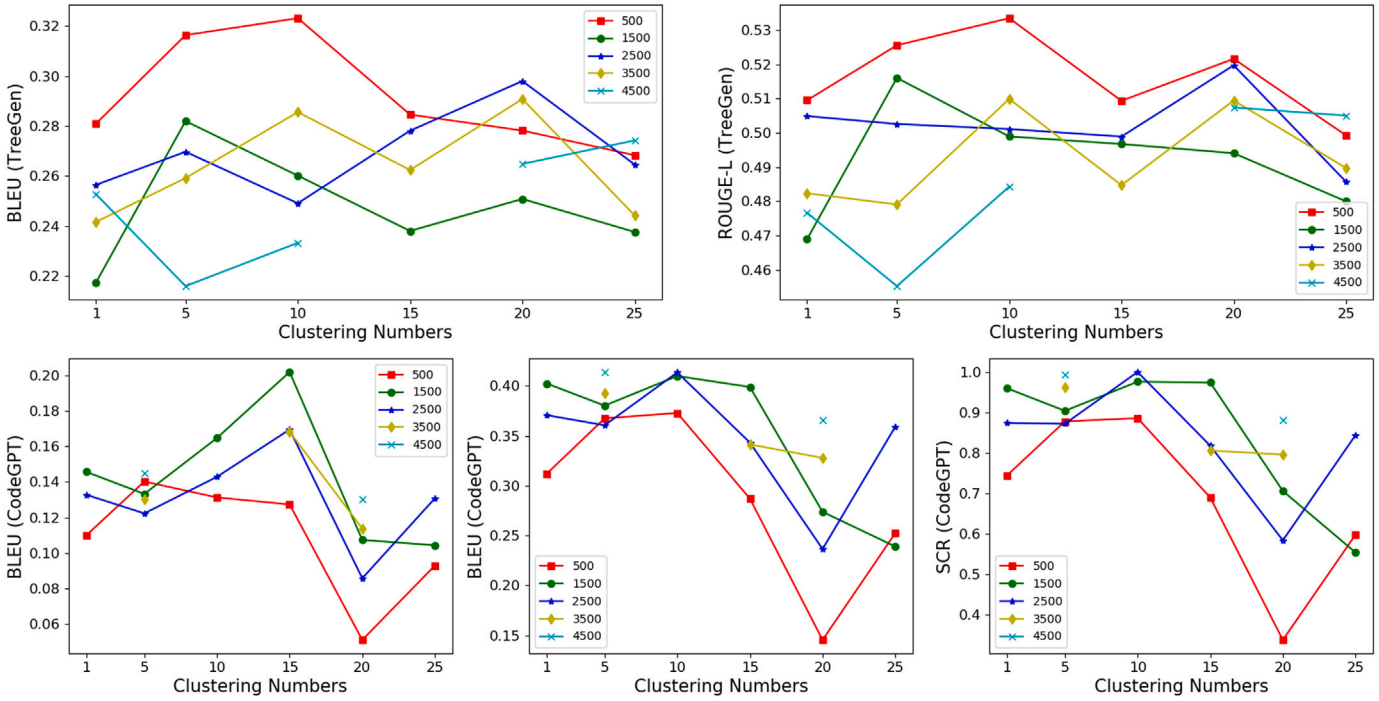


Fig. 5. The influence of clustering numbers on MetaCoder.

Table 4
Stage 2: Experimental results of each method.

Method	CodeGPT on Solidity			CodeGPT on Bash		
	BLEU	ROUGE-L	SCR	BLEU	ROUGE-L	SCR
Individual	/	/	0%	/	/	0%
Pre-Training(N)	7.83%	20.10%	50.04%	24.90%	55.08%	**100%**
Meta-Learning(R)	14.56%	40.24%	96.00%	29.99%	56.87%	**100%**
MetaCoder	**16.47%**	**40.99%**	**97.60%**	**30.97%**	**58.94%**	98.40%

we do not conduct the clustering, and the sub-tasks are constructed via random sampling. Based on our observation, we find that MetaCoder with TreeGen performs best at around clustering numbers N=10 when the pre-training steps are less than or equal to 1500. However, when continue increasing the pre-training steps, the best clustering number moves forward to around N=20, but they cannot reach the peak performance anymore. On the other hand, MetaCoder with CodeGPT consistently performs the best at around clustering number N=10. In conclusion, regardless of MetaCoder with either TreeGen or CodeGPT, it always performs best in terms of each evaluation metric when we set the clustering number at around N=10. Thus determining the best clustering number via the elbow rule is still reasonable and practical for meta-learning adapted for code generation.

7. Discussion

7.1. Domain-specific code generation with latest techniques

As we mentioned in the experimental setup of RQ1 (Section 5.1), we involve CodeGPT in our experiments, instead of the latest Large Language Models (LLMs), such as GPT-3.5 and Codex, considering the computational overhead of pre-training/fine-tuning. Although the included backbones have demonstrated the effectiveness of MetaCoder, it is still unknown how those latest LLMs perform on domain-specific PLs. To this end, we introduce GPT-3.5 with default settings and zero-shot learning in our experiments for comparison, showcasing the latest techniques' basic performance in generating domain-specific PLs. GPT-3.5 is one of the latest LLMs in the GPT-3 family trained by OpenAI, we select the latest version for experiments, namely gpt-3.5-turbo-0613.

Table 5
The performance of GPT-3.5 with zero-shot learning.

PLs	BLEU	ROUGE-L	SCR
Solidity	19.35%	37.45%	76.00%
Bash	36.06%	46.91%	80.80%

The prompt used is the same as the input template of CodeGPT in Section 5.3. Table 5 presents the performance of GPT-3.5 with zero-shot learning on the code generation of Solidity and Bash. As can be seen from Tables 2–5, even though we do not train GPT-3.5 on either Solidity or Bash dataset, it still performs better than MetaCoder with tree-based backbones on Bash in terms of BLEU. Besides, it also outperforms MetaCoder with CodeGPT in terms of BLEU on both domain-specific PLs. Nonetheless, MetaCoder with various backbones also manifests its superiority, even with much fewer parameters. For example, MetaCoder with CodeGPT carries higher scores of ROUGE-L and SCR than GPT-3.5 on both domain-specific PLs. In addition, MetaCoder with tree-based backbones also performs better on Solidity generation in terms of BLEU and ROUGE-L. The above findings demonstrate that although the latest LLMs (e.g., GPT-3.5) carry much more powerful generality on domain-specific PLs, MetaCoder still can surpass them to some extent, even with much smaller backbones. Nevertheless, training an LLM with such huge parameters is unrealistic for most practitioners, but there will always be new PLs appearing in the future, and existing LLMs cannot generalize to them. To that end, our proposal will be beneficial for such a situation and reach better results with much less computational overhead.

7.2. Instance analysis

In this section, we conduct a simple analysis of instances handled by each approach. Towards the two tree-based backbone models, i.e., TreeGen and TranX, we embed them into MetaCoder with their best settings, respectively, and showcase code generation examples of Solidity language. Following the same setting, corresponding examples are also chosen from the experimental results of baselines accordingly. The generated ASTs are manually translated to source code and shown in Table 6.

The first example uses TreeGen as the backbone model. The given NLD requires setting the transfer fee, anywhere within the range 0%–10%. As can be seen, the generated code snippet of MetaCoder (N=10, I=500) is almost the same as the reference code, except for the misprediction of the type of the parameter "_transferfeerate" and ignorance of the error message of the first "require" function. However, the generated code snippet of Pre-Training(N) (I=500) cannot infer a complete logic based on the given NLD, where it neglects to report the fee update log via the event emitting statement. Even worse, apart from the mistakes made by MetaCoder and Pre-training(N), both Meta-Learning(R) (I=500) and Individual lose more details, where the former incorrectly constrains the "_transferfeerate" while the latter totally cannot understand the requirement of the given NLD.

The second example, using TranX as the backbone model, is dictated to get the title of the module. Obviously, MetaCoder (N=10, I=3500) performs best, which correctly generates the function name and return statement, except for the return variable. However, the repeated generation problem occurs in the return variables of Pre-Training(N) (I=3500). Besides, both Individual and Pre-Training(N) cannot correctly predict the return value of this function either, which should be a string literal. Here, we do not show the generated results of Meta-learning(R) because it stops the pre-training early after 1500 steps during the meta-learning phase. Thus, we cannot conduct a fair comparison.

Subsequently, we continue selecting one example of Bash commands generated by MetaCoder with CodeGPT as shown in Table 6, namely Example 3. Apparently, the Individual method cannot discriminate Bash from Java without a pre-training phase. Thus, it even cannot generate Bash-like commands when given a requirement. As a pre-training phase is involved with our formulated instruction template, CodeGPT starts to understand the given requirement and generate Bash-like code snippets as shown in Pre-Training(N) (I=1500), Meta-Learning(R) (I=1500), and MetaCoder (N=10, I=1500). Although MetaCoder does not capture the compression order of the requirement in this example, its integrity and quality in Bash command generation still outperform the baselines.

In conclusion, the above instances visually demonstrate the superiority of MetaCoder compared with three baselines, showing that our novel method of meta-learning on code generation is effective.

8. Threats to validity

This section clarifies the threats to internal, external, and construct validity, respectively.

8.1. Internal validity

First of all, the threats to internal validity involve the replication of each approach. Due to the unavailability of ASDL in smart contracts, we do not adopt the grammar rules parsed from ASDL. Instead, we collect them from the training set for TranX as TreeGen does, and accordingly make some modifications on its input end for adaptation. While for the implementation of TreeGen, we replicate it strictly following its publicly released source code. Besides, we carefully follow the instructions in Hugging Face to implement CodeGPT with default settings. Therefore, we believe the threats of replication are limited.

Next, owing to the restriction of computation resources, we do not exhaustively tune the hyper-parameters of MetaCoder. Thus, other hyper-parameters settings may bring better results, which may become another threat to internal validity.

8.2. External validity

The threat to external validity mainly concerns the dataset we adopt in the experiments. The Java dataset, as a part of CodeSearchNet [36], is collected from publicly available open-source GitHub repositories. On the other hand, the Solidity dataset is collected from Etherscan.io,[6] composed of up-to-date verified smart contract source code deployed on Ethereum. Besides, the Bash dataset is curated from web pages such as question-answering forums and tech blogs. Hence, the Java, Solidity, and Bash datasets we adopt are all real-world examples of high quality, which means the threat to the external validity is minimal.

8.3. Construct validity

The threat to construct validity relates to the utilization of evaluation metrics. In this paper, we employ BLEU and ROUGE-L to evaluate the performance of MetaCoder and baselines, which are widely used metrics in code generation field [2,41,42]. Although these automatic evaluation metrics cannot fully substitute human judgment, they quantify the performance of each approach efficiently.

9. Related work

In this section, we summarize the related work of this paper from the following two perspectives.

9.1. Neural code generation

Neural code generation aims to automate the process of generating code snippets via deep neural networks. In the early stage, researchers adapt the sequence-to-sequence structure to develop sequence-based neural models [49,50]. However, programming languages are different from natural languages for their highly structural formats. Therefore, lots of studies have emerged to substitute the sequence-based methods with tree-based methods and train the neural networks to generate AST rules, thereby ensuring syntactic correctness for code [1,2,25,51,52]. For example, Yin et al. [52], for the first time, proposed a data-driven tree-based neural network, where both LSTM and Recurrent Neural Network (RNN) are incorporated. On the basis of their proposal, researchers proposed a lot of variants to further improve the performance. Sun et al. [2,51] successively adopted a Convolutional Neural Network (CNN) and Transformer to resolve the long-term dependency problem when generating code snippets. Subsequently, Jiang et al. [25] designed a dynamic branch selection method as a further improvement when decoding. Recently, owing to the development of pre-trained language models, sequence-based models achieve significant improvement and attract attention in academia again [15–17,53]. However, most of the previous work focus on general-purpose code generation. This paper concentrates on improving code generation performance for domain-specific PLs.

[6] https://etherscan.io/

Table 6
Instance analysis.

Example 1: MetaCoder with TreeGen on Solidity Code Generation	
Description	set the transfer fee, anywhere within the range 0%–10%.
Reference Code	function setTransferFeeRate (uint _transferFeeRate) external { require (_transferFeeRate<= MAX_TRANSFER_FEE_RATE, "Transfer fee rate must be below MAX_TRANSFER_FEE_RATE"); transferFeeRate = _transferFeeRate; emit TransferFeeUpdated (_transferFeeRate); }
Individual	function transfer (uint256 _tokenid) { require(0-10 == 0-10); require(0-10 == 0-10); }
Pre-Training(N)	function settransferfeerate (uint256 _transferfeerate) external {require (_transferfeerate <= max_transfer_fee_rate); require (_transferfeerate != max_transfer_fee_rate); _transferfeerate = _transferfeerate; }
Meta-Learning(R)	function settransferfeerate (address _who) public { require (readonly >= 0); require (_transferfeerate != fee); require (_transferfeerate != fee); }
MetaCoder	function settransferfeerate (address _transferfeerate) external {require (_transferfeerate <= max_transfer_fee_rate); transferfeerate = _transferfeerate; emit Transferfeeupdated (_transferfeerate); }
Example 2: MetaCoder with TranX on Solidity Code Generation	
Description	get the title of the module.
Reference Code	function getTitle() public view returns(string) {return "General Permission Manager";}
Individual	function gettitle() public view returns(byte32 question_id, uint8 _gameid, address _securitytoken) { uint256 _tokenid = assets[_id].y; return address(gav); }
Pre-Training(N)	function getdescription() public view returns(uint256 _index, uint 256 _team, uint256 _gameid, uint256 _team, uint256 _gameid) { return c.oldpledge; }
MetaCoder	function gettitle() public view returns(address owner) { return "STR"; }
Example 3: MetaCoder with CodeGPT on Bash Command Generation	
Description	recursively finds and compresses all files in a current folder.
Reference Code	find . -type f -exec bzip2 +
Individual	@SuppressWarnings("unchecked") public <T>T get(String key, Class<T> type) return (T)get(key);
Pre-Training(N)	find -type f
Meta-Learning(R)	find -type f
MetaCoder	find . -type f -exec ls -l \;

9.2. Meta-learning in software engineering

Meta-learning, as a kind of FSL method, has attracted more and more attention in the software engineering field, such as code search and code representation learning. For example, Chai et al. [29] adopted meta-learning to resolve the problem of cross-domain code search. Pian et al. [54] proposed a meta-learning approach, namely *MetaTP-Trans*, to learn language-specific and language-agnostic information from multilingual source code data. However, the above studies only directly partitioned sub-tasks in a random fashion without considering how to make them heterogeneous. In this paper, we propose an unsupervised category assignment method for heterogeneous sub-task construction during the meta-learning phase, thereby more efficiently adapting meta-learning to the code generation field and improving the performance of backbone models on domain-specific PLs.

10. Conclusion

In this paper, we conduct the first code generation work focusing on domain-specific PLs. We propose a novel approach, namely MetaCoder, which brings meta-learning techniques into the code generation field. MetaCoder assigns categories for code generation samples via clustering, such that we can construct heterogeneous sub-tasks for meta-learning algorithms to learn efficiently. Extensive experimental results have verified the superior performance of MetaCoder against baseline models and shown its capability of generating high-quality code snippets in domain-specific PLs.

CRediT authorship contribution statement

Zhen Yang: Conceptualization, Methodology, Software, Writing – original draft. **Jacky Wai Keung:** Supervision, Methodology. **Zeyu Sun:** Conceptualization, Methodology. **Yunfei Zhao:** Software, Validation. **Ge Li:** Supervision, Methodology. **Zhi Jin:** Supervision, Methodology. **Shuo Liu:** Validation. **Yishu Li:** Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported in part by the General Research Fund (GRF) of the Research Grants Council of Hong Kong, and the industry research funds of the City University of Hong Kong (7005217,9220097,9220103, 9229029,9229098,9678149) The National Natural Science Foundation of China under Grant Nos 62072007, 62192733, 61832009, 62192731, 62192730, the Key Program of Hubei under Grant JD2023008.

References

[1] P. Yin, G. Neubig, TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (Demo Track), 2018.
[2] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, L. Zhang, Treegen: A tree-based transformer architecture for code generation, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, (05) 2020, pp. 8984–8991.
[3] S.A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, G. Neubig, Retrieval-based neural code generation, in: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018.
[4] Y. Wang, W. Wang, S. Joty, S.C. Hoi, CodeT5: 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, 2021, pp. 8696–8708.
[5] D. Pigott, Online historical encyclopaedia of programming languages, 2020.
[6] L.A. Meyerovich, A.S. Rabkin, Empirical analysis of programming language adoption, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, 2013, pp. 1–18.

[7] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, F.-Y. Wang, An overview of smart contract: architecture, applications, and future trends, in: 2018 IEEE Intelligent Vehicles Symposium (IV), IEEE, 2018, pp. 108–113.

[8] C. Newham, Learning the Bash Shell: Unix Shell Programming, "O'Reilly Media, Inc.", 2005.

[9] S. Halloway, A. Bedra, Programming Clojure, Pragmatic Bookshelf, 2012.

[10] C. Finn, P. Abbeel, S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, in: International Conference on Machine Learning, PMLR, 2017, pp. 1126–1135.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547, http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139, URL https://aclanthology.org/2020.findings-emnlp.139.

[12] R. Xu, D. Wunsch, Survey of clustering algorithms, IEEE Trans. Neural Netw. 16 (3) (2005) 645–678, http://dx.doi.org/10.1109/TNN.2005.845141.

[13] C. Dannen, Introducing Ethereum and Solidity, Vol. 1, Springer, 2017.

[14] S. Shen, X. Zhu, Y. Dong, Q. Guo, Y. Zhen, G. Li, Incorporating domain knowledge through task augmentation for front-end JavaScript code generation, 2022, arXiv preprint arXiv:2208.10091.

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, Codegen: An open large language model for code with multi-turn program synthesis, 2022, arXiv preprint arXiv:2203.13474.

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.

[17] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021, arXiv preprint arXiv:2102.04664.

[18] W.X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al., A survey of large language models, 2023, arXiv preprint arXiv:2303.18223.

[19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., Language models are unsupervised multitask learners, 2019.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[21] F. Chollet, Xception: Deep learning with depthwise separable convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1251–1258.

[22] D.C. Wang, A.W. Appel, J.L. Korn, C.S. Serra, The zephyr abstract syntax description language, in: DSL, Vol. 97, 1997, p. 17.

[23] B. Xie, J. Su, Y. Ge, X. Li, J. Cui, J. Yao, B. Wang, Improving tree-structured decoder training for code generation via mutual learning, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35, (16) 2021, pp. 14121–14128.

[24] H. Jiang, L. Song, Y. Ge, F. Meng, J. Yao, J. Su, An AST structure enhanced decoder for code generation, IEEE/ACM Trans. Audio Speech Lang. Process. 30 (2021) 468–476.

[25] H. Jiang, C. Zhou, F. Meng, B. Zhang, J. Zhou, D. Huang, Q. Wu, J. Su, Exploring dynamic selection of branch expansion orders for code generation, in: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2021, pp. 5076–5085.

[26] J. Snell, K. Swersky, R. Zemel, Prototypical networks for few-shot learning, Adv. Neural Inf. Process. Syst. (2017).

[27] Y. Wang, Q. Yao, J.T. Kwok, L.M. Ni, Generalizing from a few examples: A survey on few-shot learning, ACM Comput. Surv. (CSUR) 53 (3) (2020) 1–34.

[28] A. Nichol, J. Achiam, J. Schulman, On first-order meta-learning algorithms, 2018, arXiv preprint arXiv:1803.02999.

[29] Y. Chai, H. Zhang, B. Shen, X. Gu, Cross-domain deep code search with meta learning, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 487–498.

[30] Y. Cheng, Mean shift, mode seeking, and clustering, IEEE Trans. Pattern Anal. Mach. Intell. 17 (8) (1995) 790–799, http://dx.doi.org/10.1109/34.400568.

[31] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al., A density-based algorithm for discovering clusters in large spatial databases with noise, in: Kdd, Vol. 96, (34) 1996, pp. 226–231.

[32] T.K. Moon, The expectation-maximization algorithm, IEEE Signal Process. Mag. 13 (6) (1996) 47–60.

[33] R.L. Thorndike, Who belongs in the family, in: Psychometrika, Citeseer, 1953.

[34] X. Hu, Z. Gao, X. Xia, D. Lo, X. Yang, Automating user notice generation for smart contract functions, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 5–17.

[35] C. Yu, G. Yang, X. Chen, K. Liu, Y. Zhou, BashExplainer: Retrieval-augmented bash code comment generation based on fine-tuned CodeBERT, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 82–93, http://dx.doi.org/10.1109/ICSME55016.2022.00016.

[36] H. Husain, et al., Codesearchnet challenge: Evaluating the state of semantic code search, 2019, arXiv preprint arXiv:1909.09436.

[37] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, M. Zhang, A multi-modal transformer-based code summarization approach for smart contracts, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, 2021, pp. 1–12.

[38] Z. Yang, J. Keung, M. Zhang, Y. Xiao, Y. Huang, T. Hui, Smart contracts vulnerability auditing with multi-semantics, in: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2020, pp. 892–901.

[39] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, BLEU: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[40] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text Summarization Branches Out, 2004, pp. 74–81.

[41] A. Svyatkovskiy, S.K. Deng, S. Fu, N. Sundaresan, Intellicode compose: Code generation using transformer, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1433–1443.

[42] M.R. Parvez, W.U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Retrieval augmented code generation and summarization, 2021, arXiv preprint arXiv:2108.11601.

[43] Z. Yang, J.W. Keung, X. Yu, Y. Xiao, Z. Jin, J. Zhang, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. 32 (2) (2023) 1–41.

[44] X. Ma, J.W. Keung, X. Yu, H. Zou, J. Zhang, Y. Li, AttSum: A deep attention-based summarization model for bug report title generation, IEEE Trans. Reliab. (2023).

[45] F. Zhang, X. Yu, J. Keung, F. Li, Z. Xie, Z. Yang, C. Ma, Z. Zhang, Improving Stack Overflow question title generation with copying enhanced CodeBERT model and bi-modal information, Inf. Softw. Technol. 148 (2022) 106922.

[46] B. Chen, C. Cherry, A systematic comparison of smoothing techniques for sentence-level bleu, in: Proceedings of the Ninth Workshop on Statistical Machine Translation, 2014, pp. 362–367.

[47] L. Van der Maaten, G. Hinton, Visualizing data using t-SNE, J. Mach. Learn. Res. 9 (11) (2008).

[48] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Adv. Neural Inf. Process. Syst. 33 (2020) 1877–1901.

[49] L. Mou, R. Men, G. Li, L. Zhang, Z. Jin, On end-to-end program generation from user intention by deep neural networks, 2015, arXiv preprint arXiv:1510.07211.

[50] W. Ling, E. Grefenstette, K.M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, P. Blunsom, Latent predictor networks for code generation, 2016, arXiv preprint arXiv:1603.06744.

[51] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, L. Zhang, A grammar-based structural cnn decoder for code generation, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, (01) 2019, pp. 7055–7062.

[52] P. Yin, G. Neubig, A syntactic neural model for general-purpose code generation, 2017, arXiv preprint arXiv:1704.01696.

[53] W.U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, 2021, arXiv preprint arXiv:2103.06333.

[54] W. Pian, H. Peng, X. Tang, T. Sun, H. Tian, A. Habib, J. Klein, T.F. Bissyandé, MetaTPTrans: A meta learning approach for multilingual code representation learning, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 37, (4) 2023, pp. 5239–5247.