CodePrompt: Task-Agnostic Prefix Tuning for Program and Language Generation

YunSeok Choi, Jee-Hyong Lee
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
{ys.choi, john}@skku.edu

Abstract

In order to solve the inefficient parameter update and storage issues of fine-tuning in Natural Language Generation (NLG) tasks, prompt-tuning methods have emerged as lightweight alternatives. Furthermore, efforts to reduce the gap between pre-training and fine-tuning have shown successful results in low-resource settings. As large Pre-trained Language Models (PLMs) for Program and Language Generation (PLG) tasks are constantly being developed, prompt tuning methods are necessary for the tasks. However, due to the gap between pre-training and fine-tuning different from PLMs for natural language, a prompt tuning method that reflects the traits of PLM for program language is needed. In this paper, we propose a Task-Agnostic prompt tuning method for the PLG tasks, CodePrompt, that combines Input-Dependent Prompt Template (to bridge the gap between pre-training and fine-tuning of PLMs for program and language) and Corpus-Specific Prefix Tuning (to update the parameters of PLMs for program and language efficiently). Also, we propose a method to provide richer prefix word information for limited prefix lengths. We prove that our method is effective in three PLG tasks, not only in the full-data setting but also in the low-resource setting and cross-domain setting.

1 Introduction

As the software engineering field continues to grow, the use of AI to increase the efficiency of developers through code intelligence is becoming increasingly important. In particular, Program and Language Generation (PLG) tasks, such as code summarization, code generation, and code translation, are essential for developers to maximize their productivity. Code summarization allows developers to quickly understand the structure and purpose of a code, code generation assists by automatically generating code given a natural language description, and code translation facilitates the translation of code from one programming language to another, such as from Java to C#, and vice versa.

The recent success of Pre-trained Language Models (PLMs) for the PLG tasks, such as CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021), and CodeT5 (Wang et al., 2021), has been attributed to their utilization of large-scale code and text corpora. The "*pre-training then fine-tuning*" approach has been widely used to derive program language representations by self-supervised training on large-scale unlabeled data, which can then be transferred to multiple downstream tasks with limited data annotation. These approaches have proven to be successful on code-related downstream tasks. However, fine-tuning large pre-trained models can be expensive and time-consuming in terms of both updating and storing all parameters. Furthermore, there is a discrepancy between pre-training and fine-tuning in the viewpoint of the inputs and the training objectives (Brown et al., 2020; Wang et al., 2020). It makes the model difficult to fully utilize the knowledge of pre-trained models, resulting in suboptimal performance on downstream tasks (Lester et al., 2021; Gu et al., 2022; Han et al., 2022).

In order to address the issues of fine-tuning, prompt tuning approaches have recently been proposed in Natural Language Generation (NLG) tasks. To reduce the gap between pre-training and fine-tuning, Schick and Schütze (2021) proposed a prompt tuning method that combined manually crafted templates with the input. However, finding the optimal manual prompt template for each natural language task is arduous and laborious. Furthermore, due to the updating of all parameters of the language model, it also requires updating full model parameters for each task, similar to fine-tuning. It can also easily lead to sub-optimal language model parameters in low-resource settings. Li and Liang (2021) proposed a lightweight method, prefix tuning, to freeze the language model

Findings of the Association for Computational Linguistics: ACL 2023, pages 5282–5297

"In the mid-1970s punk rock was born in a dank little New York nightclub called CBGB\'s. It all started when rockers like Television, the Ramones and Patti Smith launched a frontal assault on the monolith of corporate rock \'n roll. Now another artistic revolt, Remodernism, is about to widen its offensive from the birthplace of punk.“ On May 10, 2006, the Stedelijk Museum and the University of Amsterdam staged a talk on remodernism by Daniel Birnbaum, contributing editor of Artforum, and Alison Gingeras, Assistant Curator, Guggenheim Museum. **The summary is:** In August 2006, an online group called "The Remodernists of Deviantart" was founded by Clay Martin. The group is composed of artists who are active on the website deviantart.com. In 2006, artist Matt Bray said,

(a) An example of Wikipedia

```
def change_return_type(f):
    # Converts the returned value of wrapped function to the
    # type of the first arg or to the type specified by a kwarg key
    # return_type's value.
    @wraps(f)
    def wrapper(*args, **kwargs):
        if kwargs.has_key('return_type'):
            return_type = kwargs['return_type']
            kwargs.pop('return_type')
            return return_type(f(*args, **kwargs))
        elif len(args) > 0:
            return_type = type(args[0])
            return return_type(f(*args, **kwargs))
        else:
            return f(*args, **kwargs)
    return wrapper
```

(b) An example of CodeSearchNet (Husain et al., 2019)

Figure 1: (a) PLMs for natural language are pre-trained using text from Wikipedia. (b) PLMs for program and language are pre-trained using code and comment from CodeSearchNet. The text of PLMs for program and language is agnostic to task.

parameters and instead optimize a sequence of continuous task-specific vectors (prefix). This method has shown performance comparable to fine-tuning in the full data setting while updating much fewer training parameters of a large pre-trained model. Also, even in the low-resource setting, this method has been proven to be effective by prefix initialization with words specific to the task.

However, those prompt tuning approaches for most NLG tasks are difficult to apply directly to the PLG tasks. The pre-training of PLMs for natural language involves the use of large amounts of text data consisting of a series of sentences. This dataset contains task-specific natural language instructions (templates). As shown in Figure 1a, task-specific natural language instructions, such as "Summarize _", "TL;DR _", and "The summary is _", appear in data for pre-training. These templates can bridge the gap between pre-training and fine-tuning. However, datasets of PLMs for program and language hardly contain such task-specific textual instructions (templates). PLMs for program and language are usually pre-trained with unimodal data (code-only) or bimodal data (code-comment). Input is either code or its corresponding comment, but there are very few task-specific natural language instructions, shown in Figure 1b.

Due to the lack of task-specific instructions in the pre-training stage, it is hard to manually select task-dependent prompt templates for the PLG tasks during the fine-tuning stage. Prefix tuning in the NLG tasks improved performance by initializing prefix embedding from task-specific words, especially in low-resource settings. However, for PLG tasks, we can hardly adopt such initialization approaches because there are very few task-specific words in data for PLG tasks, as mentioned. Also, unlike NLG tasks, PLG tasks are two cross-modal generation tasks. It is not appropriate to initialize encoder and decoder prefixes with the same words in the same language. Prefix embeddings of encoder and decoder need to be initialized with different words of their corresponding language.

Therefore, we propose a task-agnostic prompt tuning method, CodePrompt, applicable to any PLG tasks. Our method consists of three components: input-dependent prompt template, corpus-specific prefix tuning, and multi-word prefix initialization. First, we propose the input-dependent prompt template by combining the template with input to bridge the gap between pre-training and fine-tuning in PLMs for program and language. Input-dependent prompt template contains unified backbone words and input-specific words, regardless of the task. Second, we propose the corpus-specific prefix tuning to reduce the number of parameters for update considering the traits of two cross-modal tasks. They can effectively transfer the task and corpus specific information in cross-modal tasks, especially in low-resource settings and zero-shot settings. Third, we propose the multi-word prefix initialization to provide richer information to prefix embeddings while maintaining the number of parameters within the limited prefix length. Our CodePrompt shows great performances on three PLG tasks in full data, low-resource, and cross-domain settings.

2 Related Work

Pre-trained Model for Program and Language As the pre-trained models based on the Transformer architecture (Vaswani et al., 2017) have achieved great success in NLG tasks, the methods on extending natural language-based methods to code

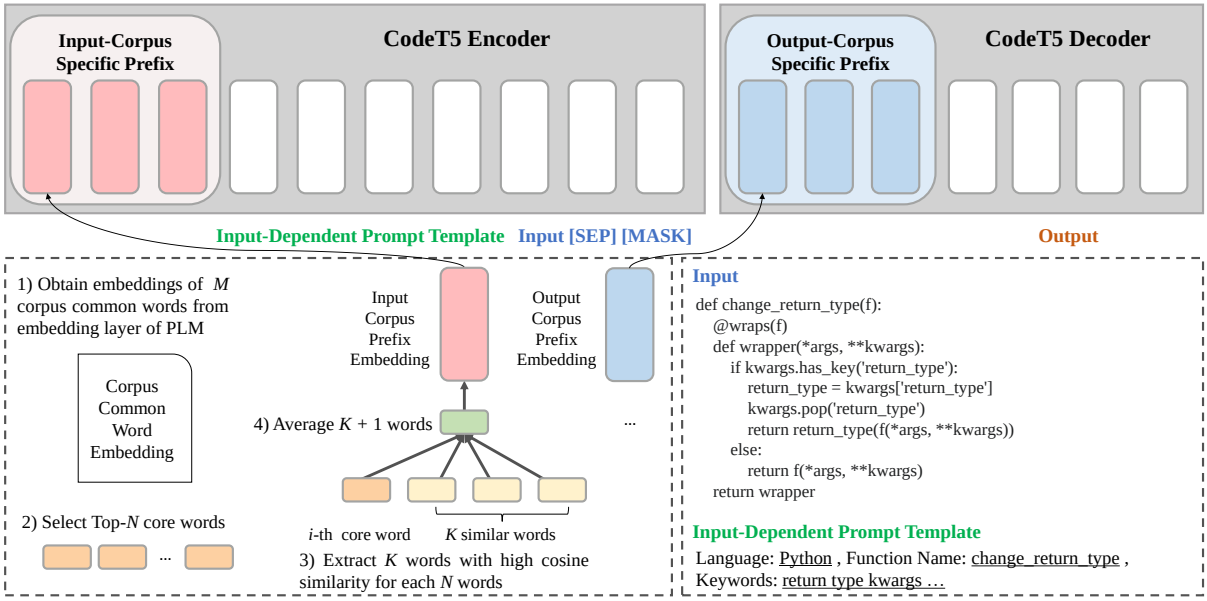

Figure 2: Overview of our Task-Agnostic Prompt Tuning, CodePrompt.

have recently been proposed in PLG tasks. Feng et al. (2020) proposed CodeBERT, a pretrained language model, based on BERT (Devlin et al., 2019). The model learns cross-modal representation both program language and natural language in the pre-training stage. Guo et al. (2020) proposed GraphCodeBERT to incorporate the code structure into CodeBERT. However, such models are vulnerable to the PLG task because they learn the PL-NL representation with the transformer encoder only. Ahmad et al. (2021) proposed PLBART to support both code understanding and generation tasks using encoder-decoder model BART (Lewis et al., 2020). Also, Wang et al. (2021) proposed CodeT5, a pre-trained sequence-to-sequence model based on T5 (Raffel et al., 2022), to facilitate generation tasks for source code. We utilize the framework of the CodeT5, the state-of-the-art pre-trained model, in the PLG tasks for an effective prompt tuning method of PLM for program and language.

Prompt tuning for Generation In NLG tasks, the concept of prompt-tuning originated from in-context learning, which was first introduced in GPT-3 (Brown et al., 2020). Schick and Schütze (2021) explored the use of fixed-prompt language model (LM) tuning for few-shot text summarization using manually created templates. Li and Liang (2021) investigated prefix tuning, fixed-LM prompt tuning, where learnable prefix tokens are prepended to the input while the parameters in pre-trained models are frozen. Lester et al. (2021) proposed soft prompt as a simplification of prefix tuning. Several prompt tuning methods have been proposed for NLG tasks, but they are not applicable to PLG tasks because they require additional data information for specific natural language tasks. Recently, Wang et al. (2022) evaluated the effect of prompt tuning in Program and Language Understanding and Generation tasks. However, they used the prompt tuning method with updating all parameters of the pre-trained language model, not freezing the parameters. Moreover, they did not consider the gap between pre-training and fine-tuning of PLMs for program and language.

3 CodePrompt

In this section, we explain our prompt tuning method, CodePrompt, in detail. Our prompt tuning method aims to consider the traits of PLMs for program and language. Figure 2 shows the architecture of our method, which is on the basis of the CodeT5 framework, including input-dependent prompt template, corpus-specific prefix tuning, and multi-word prefix initialization.

3.1 Input-dependent Prompt Template

In NLG tasks, it is common to manually craft templates specific to tasks. However, in PLG tasks, it is hard to select task-specific templates because they are rarely seen in the pre-training stage. Instead of providing task-related templates, we will provide input-dependent templates. Input-dependent templates, that are agnostic to the task, can help in un-

derstanding the input by reducing the gap between pre-training and fine-tuning. PLMs for program and language are pre-trained using bi-modal data, pairs of code and its comment, but provided with unimodal data, either code or comment, in the fine-tuning stage. If we provide additional information that seems like comment or code, it can bridge the gap between inputs of pre-training and fine-tuning stages. It will help better transfer the knowledge gained in pre-training stage to fine-tuned models.

There is a lot of information about code such as repository (owner), path, and library information, but we choose three easily extractable information from code: language, function name, and keywords. The backbone of the template is "Language _, Function Name _, Keywords _". The backbone words are task-agnostic, fixed and unified template, and "_" is dependent to the input. The information of language and function name can be easily obtained, and keywords can be extracted by various keyword extraction methods. To prove the efficiency of our template, we use a simple but effective keyword extraction method, TextRank (Mihalcea and Tarau, 2004).

For example, if a code snippet is given in the code summarization task, the following prompt for the code will be added: "Language: Python, Function Name: simulate_request, Keywords: simulate request wsgi ...", as shown in Figure 2. The actual ground truth comment for the code is "simulate a request to a wsgi application". This prompt template can act not only as the comment for the code in natural language, but also as a hint for summarization. The pair of the template and the input will act like a bimodal pair seen in the pre-training stage. In the code generation task where an NL comment is given, we also use the template without the backbone word "Function Name". In this case, the keywords from the comment will act like a simple version of the corresponding code, because most of keywords from the comment may also appear in the code.

3.2 Corpus-Specific Prefix Tuning

In prefix-tuning, prefix embedding initialization is important. Initialization with task-specific prefix words has been proven to be effective, especially in low-resource settings, because it can easily transfer task-specific knowledge to the pre-trained model. However, as mentioned above, in the pre-training stage of PLMs for program and language, the task-specific words were rarely seen. Instead of task-specific words, we try to transfer task-related knowledge to the fine-tuned model by providing frequent words in input and output corpora. We initialize the encoder prefix with common words in the input corpus and the decoder prefix with common words in the output corpus. By providing common words of each input and output corpus, we can indirectly provide what the model is required to do for the given task.

For the transfer of corpus-specific information to each encoder and decoder, we initialize using corpus-specific prefix words corresponding to each input and output corpus. Corpus-specific prefix embeddings are initialized with input and output corpus's common words obtained from the train data. Our encoder combines the input embedding and input corpus prefix embedding with a bidirectional language model to learn the context. If the input corpus prefix embeddings of prefix words are composed of words representing the input corpus, the encoder can learn the global features of the corpus of the prefix embeddings and the individual feature of the input. Our decoder generates output words with a left-to-right language model through output corpus prefix embeddings. If the output corpus prefix words are the output corpus representative words, the output sequence is generated by considering frequently occurring words such as frequently occurring words "return", "get", "method", and "file".

3.3 Multi-Word Prefix Initialization

Prefix tuning has shown effective performance in both full data and low resource settings, but only one word is used to initialize each prefix embedding. If we provide as many words as possible, it will help to transfer more knowledge of the pre-trained model to the fine-tuned model.

We propose a multi-word prefix initialization method. Each prefix embedding is initialized with multiple words within the limited prefix length. Let N be the prefix length and M be the corpus common words ($N \ll M$). First, we obtain the embedding of the corpus common words M from the embedding layer of a pre-trained language model. Then, we select the top-N of the corpus common words as the core words. For each N core word, K words with high cosine similarity among M words are extracted. We combine one core word and its similar words using a feed-forward neural

network (FFN). $K + 1$ word embeddings are averaged through mean pooling in the hidden layer and then obtained the prefix embedding of all layers, as shown in Figure 2. Multi-word prefix initialization can provide a rich set of prefix words while maintaining the same number of parameters as the FFN used for prefix tuning for stable optimization.

3.4 CodePrompt-based CodeT5 Architecture

As shown in Figure 2, we utilize our CodePrompt to apply to the framework of CodeT5. First, we extract common words from the input language and output language to obtain corpus-specific prefix words. Then, the prefix embeddings of our encoder and decoder are initialized through the multi-word prefix initialization method. When a code or comment is given as input, we generate its input-dependent prompt template and combine the template with the input. Our encoder and decoder are frozen and only the prepended prefix embedding is trained.

4 Experiment Setup

4.1 Downstream Tasks & Datasets

We evaluate our CodePrompt method on three generation tasks in CodeXGLUE benchmark (Lu et al., 2021): code summarization, code generation and code translation. **Code Summarization** is the task of generating a natural language summary from code. The dataset consists of six programming languages, namely, Ruby, Javascript, Go, PHP, Java, and Python. **Code Generation** is the task of generating code from its natural language description. **Code Translation** is the task of generating a code of target language from the code of source language. Table 1 is detailed statistics of the datasets.

4.2 Evaluation Metrics

BLEU (Papineni et al., 2002) computes the n-gram overlap between a generated sequence and a reference. **CodeBLEU** (Ren et al., 2020) is a metric for measuring the quality of the code. Unlike BLEU, CodeBLEU considers grammatical and logical correctness based on the abstract syntax tree and the data-flow structure. we refer to CodeBLEU as C.BLEU. **Exact Match (EM)** measures whether a generated sequence exactly matches the reference. **#Param** is the number of parameters to be updated. For more details about the evaluation metrics, please refer to Appendix B.

Task	Language	Train	Valid	Test
Summarization	Ruby	24K	1.4K	1.2K
	Javascript	58K	3.8K	3.2K
	Go	167K	7.3K	8.1K
	PHP	241K	12.9K	14K
	Java	164K	5.1K	10.9K
	Python	251K	13.9K	14.9K
Generation	NL to Java	100K	2K	2K
Translation	Java to C#	10.3K	0.5K	1K
	C# to Java	10.3K	0.5K	1K

Table 1: Statistics of three PLG tasks in CodeXGLUE benchmark datasets (Lu et al., 2021).

4.3 Baseline Methods

We compare our method with the state-of-the-art (SOTA) pre-trained models. As encoder-only models, we compare with RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and DOBF (Roziere et al., 2021). For decoder-only models, we compare with GPT2 (Radford et al., 2019) and code-version GPT models, CodeGPT-2 and CodeGPTadapted. As encoder-decoder models, we consider PLBART (Ahmad et al., 2021) and finetuned-CodeT5 (Wang et al., 2021). And we compare our method with CodeT5 which is trained by Prefix-tuning (Li and Liang, 2021).

4.4 Training Details

We implement our prompt method based on the Hugging Face Transformer models[1] (Wolf et al., 2020). We use the AdamW optimizer (Loshchilov and Hutter, 2019) and a linear learning rate scheduler. We follow the implementation details of CodeT5 for all configuration settings. The default prefix length of each task is set to 200, 250, and 100 for code summarization, code generation, and code translation, respectively. We choose a simple but effective keyword extraction method, TextRank (Mihalcea and Tarau, 2004). The number of language common words M is 400, the number of similar words K is 3, and the number of keywords in input-dependent prompt template is 10. For detailed configurations for each task, refer to Appendix A.

[1] https://github.com/huggingface/transformers

Methods	#Param	Ruby	Javascript	Go	PHP	Java	Python	Overall
Fine-Tuning								
RoBERTa	125M	11.17	11.90	17.72	24.02	16.47	18.14	16.57
CodeBERT	172M	12.16	14.90	18.07	25.16	17.65	19.06	17.83
PLBART	139M	14.11	15.56	18.91	23.58	18.45	19.30	18.32
CodeT5-base	222M	15.24	16.16	19.56	26.03	20.31	20.01	19.55
+Input-Dependent (ours)	222M	15.44	16.21	19.66	26.26	20.39	20.27	19.71
Prompt Tuning								
Prefix-tuning	20M	14.91	15.36	19.15	25.03	19.96	19.93	19.06
CodePrompt (ours)	20M	15.71	15.78	19.43	25.43	20.13	20.25	19.46

Table 2: Smoothed BLEU-4 scores on the code summarization task.

Methods	#Param	NL to Java		
		EM	BLEU	C.BLEU
Fine-Tuning				
GPT-2	124M	17.35	25.37	29.69
CodeGPT-2	124M	18.25	28.69	32.71
CodeGPT-adapted	124M	20.10	32.79	35.98
PLBART	139M	18.75	36.69	38.52
CodeT5-base	222M	22.30	40.73	43.20
+Input-Dependent (ours)	222M	23.05	43.13	43.24
Prompt Tuning				
Prefix-tuning	20M	21.30	35.72	36.32
CodePrompt (ours)	20M	21.85	37.51	38.19

Table 3: Results on the code generation task.

Methods	#Param	Java to C#		C# to Java	
		BLEU	EM	BLEU	EM
Fine-Tuning					
RoBERTa (code)	125M	77.46	56.10	71.99	57.90
CodeBERT	172M	79.92	59.00	72.14	58.80
GraphCodeBERT	172M	80.58	59.40	72.64	58.80
PLBART	139M	83.02	64.60	78.35	65.00
CodeT5-base	222M	84.03	65.90	79.87	66.90
+Input-Dependent (ours)	222M	85.23	66.60	81.60	67.20
Prompt Tuning					
Prefix-tuning	20M	80.58	57.60	77.21	61.10
CodePrompt (ours)	20M	81.82	59.80	79.27	63.50

Table 4: Results on the code translation task.

5 Experiment Results

5.1 Full-Data Setting

Code Summarization Table 2 shows the results of code summarization for six programming languages in the full-data setting.

First, to prove the effectiveness of our input-dependent prompt template, we fine-tune the SOTA pre-trained model, CodeT5-base, with only input-dependent prompt template (the language model is not frozen). The model fine-tuned with only input-dependent prompt template shows better performance than the other SOTA models. This shows that our input-dependent prompt template effectively acts as a hint for generating summary and reduces the gap between pre-training and fine-tuning of PLMs for program and language. However, simply combining the template with the input is not effective because it tuned all parameters like fine-tuning.

In prompt tuning methods, prefix-tuning (Li and Liang, 2021) shows a performance that is slightly lower than the fine-tuning methods, but with a very small number of parameters to be updated. Here, our method, CodePrompt, shows great performance comparable to fine-tuning, while updating a fewer number of parameters. The number of parameters to update is about 1/11. In the case of Ruby and Python, the scores are higher than the fine-tuning method, CodeT5-base, by 0.47 and 0.24, respectively.

Code Completion The results of the code generation task in the full-data setting are shown in Table 3. Among the fine-tuning methods, CodeT5 with our input-dependent prompt template has the best performance compared to the other fine-tuning methods. The BLEU score increased by 2.4 compared to CodeT5-base. Additionally, CodePrompt has much better EM, BLEU, and CodeBLEU scores compared to prefix-tuning. In particular, our method has almost the same EM score as CodeT5-base and even better performance than other pre-trained models except for CodeT5 while updating very few parameters. This shows that our CodePrompt is very effective on PLM for programs and languages with a small number of updates.

Code Translation Table 4 shows the results of code translation from Java to C# and from C# to Java in the full-data setting. As with code summarization and code generation tasks, our input-dependent prompt template with fine-tuning shows the best performance and CodePrompt show very effective results compared to other baselines. Espe-

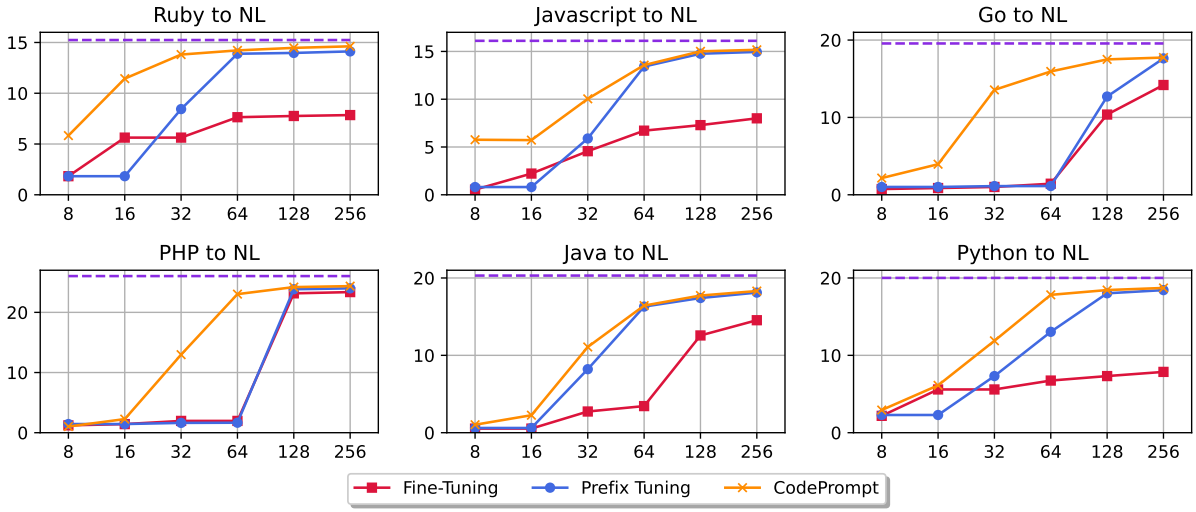

Figure 3: Results on the code summarization tasks in low-resource settings. The x-axis is training size and the y-axis is the evaluation metric score (BLEU). The purple dash line is the score of CodeT5-base in the full data setting.

cially in C# to Java, CodePrompt has BLEU score almost similar to the CodeT5 fine-tuning method and better performance than PLBART. Compared to PLBART (139M) and CodeT5 (222M) in the full-data setting, Codeprompt shows almost comparable performance by updating only 20M parameters, proving its effectiveness in the full-data setting. This shows that CodePrompt is very effective in PLG tasks.

5.2 Low-Resource Setting

We evaluate our prompt method in code summarization in low-resource settings. We randomly selected 8, 16, 32, 64, 128, and 256 training instances from the original data. Figure 3 shows the result of code summarization on six program languages in the low-resource data setting. Our method outperforms the prefix tuning method in all few-shot environments for all languages. In all program languages, prefix-tuning showed poor performance at few shot instances (8 or 16 shots), but CodePrompt can perform well even with 16 shots. In particular, for Go, we can see that prefix tuning cannot be learned up to 64 shots, whereas our method can be learned from 16 shots. This shows that by initializing the prefix embedding for each language from corpus-specific prefix words, the model can learn the global feature of the language with little data. Furthermore, fine-tuning is highly sub-optimal for few-shot data, so it cannot produce general performance. For the few-shot results for generation and translation, please refer to Appendix C.

5.3 Cross Domain Setting

PLMs for PLG tasks should have generalization capability on any unseen program languages. They need to understand and process languages where there is no existing training data in a new language. We study the benefits of our CodePrompt in cross-domain settings (zero-shot settings).

Table 5 presents the results of code summarization in cross-domain settings. Each of the three program languages (Go, Java, and Python) is for training data and other three program languages (Ruby, Javascript, and PHP) are regarded as the unseen target languages. The result shows that CodePrompt is much more effective in cross-domain settings than fine-tuning. Fine-tuning updates all parameters to provide sub-optimal performance for the source language, but CodePrompt based on prefix-tuning only updates prefix embeddings while keeping the language model frozen. For this reason, our CodePrompt based on prefix tuning method shows better performance in cross-domain settings. For the results of other languages, refer to Appendix D.

5.4 Ablation Study

We perform an ablation study on code summarization task in the full data setting. As shown in Table 6, we observe that the removal of input-dependent prompt template significantly decreases performance. The scores drop by 0.57, 0.15, and 0.30 for ruby, java, and python, respectively. The result proves that our input-dependent prompt template is effective to bridge the gap between pre-training and fine-tuning of PLMs for program and

Source	Methods	Target		
		Ruby	Javascript	PHP
Go	Fine-tuning	11.94	12.43	18.61
	CodePrompt	13.05	13.01	19.27
Java	Fine-tuning	14.35	14.21	22.31
	CodePrompt	15.54	14.90	23.42
Python	Fine-tuning	15.13	14.47	21.56
	CodePrompt	15.90	15.59	23.43
CodeT5-base		15.24	16.16	26.03

Table 5: Results on code summarization task in cross-domain setting. The score of CodeT5-base is the result of fine-tuning with the same language as the target.

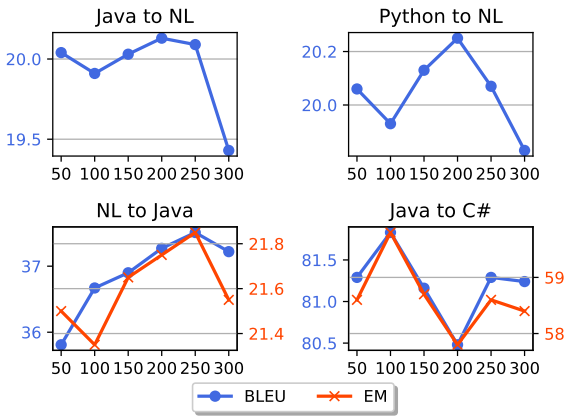

Figure 4: Results on code summarization, code generation, and code translation with different prefix lengths. The x-axis is prefix lengths, the left y-axis (blue) is BLEU, and the right y-asis (red) is EM (Exact Match).

language. Corpus-specific prefix tuning and multi-word prefix initialization are helpful to improve the performance and reduce the parameters of PLMs for update store. For the results of other languages and the effect of prefix module, refer to Appendix E.

5.5 Effect of Prefix Length

We also study the impact of different lengths of prefix prompts. We illustrate the performance under different prefix prompt lengths for the three tasks. As shown in Figure 4, prefix prompts of too short or long lengths can degrade the model performance. For each task and program language, the best performance of prefix length varied slightly. In our work, the prefix lengths are set as 200, 250, and 100 for code summarization, code generation, and code translation tasks, respectively.

Methods	Ruby	Java	Python
CodePrompt	15.71	20.13	20.25
w/o Input.	15.14	19.98	19.95
w/o Corpus.	15.46	20.04	20.06
w/o Multi-Word.	15.50	20.06	20.06

Table 6: Ablation study on code summarization task in the full data setting. "w/o Input." refers to without input-dependent prompt template, "w/o Corpus." means the encoder and decoder initialization with the prefix word of the output language, and "w/o Multi-Word." means the initialization with single word per a prefix embedding.

Methods	# Params	Ruby	Java	Python
Fine-Tuning				
CodeT5-small	60M	14.87	19.92	20.04
CodeT5-base	222M	15.24	20.31	20.01
CodeT5-large	737M	15.58	20.74	20.57
Prompt Tuning				
Prefix-tuning	52M	15.06	20.32	20.09
CodePrompt	52M	15.79	20.61	20.35

Table 7: Results of CodeT5-large on code summarization task. We compare our method with CodeT5-large reported in (Le et al., 2022).

5.6 Expand to Large Pre-trained Model

We applied our CodePrompt to the CodeT5-large by utilizing the very effective benefits of the parameters. Table 7 shows the results of applying CodePrompt to the CodeT5 large model. Our method showed better performance to the CodeT5-base model, as well as comparable performance to fine-tuning for the codeT5-large. Especially when using CodePrompt on the large model, the number of parameters was much less than when fine-tuning CodeT5-base, yet it showed much better performance. Our CodePrompt has shown to be very effective even for models with a large number of parameters. For more results of other languages, please refer to Appendix F.

6 Conclusion

In this work, we proposed CodePrompt, a Task-Agnostic prompt tuning method for Program and Language Generation tasks. Our CodePrompt combined input-dependent prompt template to bridge the gap between pre-training and fine-tuning of PLMs for program and language, and corpus-specific prefix tuning to efficiently update the parameters of PLMs. Additionally, we proposed multi-word prefix initialization method to provide more rich prefix word information for limited pre-

fix lengths. We demonstrated that our method is effective in three PLG tasks, both in full-data and low-resource settings, as well as in cross-domain settings.

Limitations

In this section, we discuss some limitations and potential risks of our work. (1) Our CodePrompt focused on Program and Language Generation tasks, so it is difficult to directly apply our method to Program and Language Understanding tasks. (2) We designed an input-dependent prompt template with fixed backbone words (Language, Function Name, Keywords) for a simple and efficient template. A more effective template can be crafted. (3) We applied only CodeT5, the most state-of-the-art model, as the basis of the framework of our CodePrompt.

Ethics Statement

This paper proposes a task-agnostic prompt tuning method for the PLG tasks to bridge the gap between pre-training and fine-tuning of PLMs for program and language and to efficiently update the parameters of PLMs for program and language, which is beneficial to energy efficient Program and Language applications. The research conducted in this paper will not cause any ethical issues or have any negative social effects. The data used is publicly accessible and is commonly used by researchers as a benchmark for program and language generation tasks. The proposed method does not introduce any ethical or social bias, or worsen any existing bias in the data.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-00421, AI Graduate School Support Program(Sungkyunkwan University)), (No.2022-0-01045, Self-directed Multi-modal Intelligence for solving unknown, open domain problems), and (No.2020-0-00990,Platform Development and Proof of High Trust & Low Latency Processing for Heterogeneous·Atypical·Large Scaled Data in 5G-IoT Environment)

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

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, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2022. PPT: Pre-trained prompt tuning for few-shot learning. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8410–8423, Dublin, Ireland. Association for Computational Linguistics.

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

Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. 2022. Ptr: Prompt tuning with rules for text classification. *AI Open*, 3:182–192.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv preprint*, abs/1909.09436.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven CH Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv preprint*, abs/2207.01780.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for 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)*, pages 4582–4597, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv preprint*, abs/1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv preprint*, abs/2102.04664.

Rada Mihalcea and Paul Tarau. 2004. TextRank: Bringing order into text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 404–411, Barcelona, Spain. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

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):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2022. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv preprint*, abs/2009.10297.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. *ArXiv preprint*, abs/2102.07492.

Timo Schick and Hinrich Schütze. 2021. Few-shot text generation with natural language instructions. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 390–402, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 382–394.

Chengyi Wang, Yu Wu, Shujie Liu, Zhenglu Yang, and Ming Zhou. 2020. Bridging the gap between pretraining and fine-tuning for end-to-end speech translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9161–9168.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu,

Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

A Implementation Details

We set the environment for all experiments as follows: one NVIDIA 3090 GPU with 24GB graphic memory, Ubuntu 20.04, Python 3.8, and CUDA 11.7 version. In the full data settings, the average training time for CodePrompt takes about 2, 4, 15, 18, 14, and 20 hours on ruby, javascript, go, php, java, and python, respectively. The average training time for code generation and code translation takes about 14 and 5 hours, respectively. The description of the hyperparameter for the experiment is shown in the tables below.

Task	Hyper-parameter	Value
Common	Optimizer	AdamW
	beam size	10
	warm-up ratio	0.1
Code Summarization	train batch size	20
	eval batch size	12
	source length	256
	target length	128
	num train epoch	15
	learning rate	5e-5
	patience	3
Code Generation	train batch size	14
	eval batch size	8
	source length	320
	target length	150
	num train epoch	30
	learning rate	1e-4
	patience	3
Code Translation	train batch size	12
	eval batch size	8
	source length	320
	target length	256
	num train epoch	100
	learning rate	5e-5
	patience	5

Table 8: Hyper-parameter settings

B Evaluation Metrics

BLEU(Papineni et al., 2002) is a Bilingual Evaluation Understudy to measure the quality of generated code summaries. The formula for computing BLEU is as follows:

$$\text{BLEU} = \text{BP} \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n$$

where p_n is the geometric average of the modified n-gram precisions, ω_n is uniform weights $1/N$ and BP is the brevity penalty.

CodeBLEU (Ren et al., 2020) is an automatic evaluation of code synthesis considering information from the n-gram, syntactic, and semantic match. The formula for computing CodeBLEU is as follows:

$$\begin{aligned}\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} \\ + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}}\end{aligned} \quad (1)$$

where BLEU is calculated by standard BLEU, $\text{BLEU}_{\text{weight}}$ is the weighted n-gram match, $\text{Match}_{\text{ast}}$ is the syntactic AST match, Match_{df} is the semantic dataflow match. The weighted n-gram match and the syntactic AST match are used to measure grammatical correctness, and the semantic data-flow match is used to calculate logic correctness. The values of $\alpha, \beta, \gamma, \delta$ are all set as 0.25.

Exact Match (EM) evaluates whether a generated sequence exactly matches the reference. If the characters of the sequence generated by the model exactly match the characters of the reference, **EM** = 1, otherwise **EM** = 0.

C Low Resource Settings

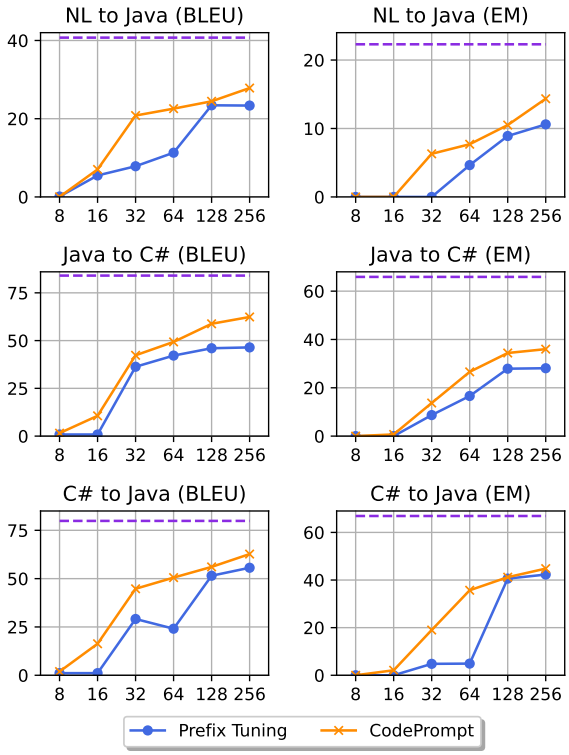

Figure 5: Results on the code generation and code translation tasks in low-resource settings. We randomly selected 8, 16, 32, 64, 128, and 256 training instances from the original data.

D Cross Domain Settings

Source	Methods	Target					
		Ruby	JS	Go	PHP	Java	Python
Ruby	Fine-tuning	-	14.98	15.87	22.59	17.05	17.78
	CodePrompt	-	14.78	15.13	22.53	18.44	18.28
JS	Fine-tuning	14.60	-	15.15	23.17	18.03	18.04
	CodePrompt	15.68	-	15.48	23.32	18.92	18.83
Go	Fine-tuning	11.94	12.43	-	18.61	15.39	13.41
	CodePrompt	13.05	13.01	-	19.27	17.08	14.61
PHP	Fine-tuning	15.09	15.46	14.75	-	17.23	18.35
	CodePrompt	15.91	15.73	15.24	-	19.1	19.2
Java	Fine-tuning	14.35	14.21	15.48	22.31	-	17.51
	CodePrompt	15.54	14.90	15.72	23.42	-	18.54
Python	Fine-tuning	15.13	14.47	15.36	21.56	16.85	-
	CodePrompt	15.90	15.59	15.13	23.98	18.65	-
CodeT5-base		15.24	16.16	19.56	26.03	20.31	20.01

Table 9: Smoothed BLEU-4 scores on code summarization task in cross-domain setting.

E More Ablation Study

We study ablation study on code summarization task in the full data setting.

Methods	Ruby	JS	Go	PHP	Java	Python
CodePrompt	15.71	15.78	19.43	25.43	20.13	20.25
w/o Input.	15.14	15.44	19.19	25.04	19.98	19.95
w/o Corpus.	15.46	15.41	19.30	25.06	20.04	20.06
w/o Multi-Word.	15.50	15.45	19.39	25.10	20.06	20.06

Table 10: Ablation study on code summarization task in the full data setting.

We evaluated the effects of the prefix module in the encoder and decoder on the code summarization task in the full data setting. When the encoder or decoder prefix module was removed, the performance of the model decreased significantly. Additionally, we observed that removing the source prefix module caused a more critical performance degradation than removing the target prefix module.

Methods	Ruby	JS	Go	PHP	Java	Python
CodePrompt	15.71	15.78	19.43	25.43	20.13	20.25
w/o source. prefix	15.32	15.04	18.75	24.25	19.39	19.18
w/o target. prefix	15.34	15.40	19.01	24.61	19.79	20.01

Table 11: Effects of prefix module on code summarization task in the full data setting.

Table 12 and 13 present the performance comparison between task-specific templates and our approach for the summarization and translation tasks, respectively, in the full data setting. In the code summarization task, we re-implemented the method proposed in Wang et al. (2022) using the publicly available dataset used in our work to ensure a fair comparison. Additionally, for the code translation task, we presented the results as reported in the paper by (Wang et al., 2022). Our model demonstrated significantly more effective performance.

Methods	Ruby	JS	Go	PHP	Java	Python
CodeT5-base	15.24	16.16	19.56	26.03	20.31	20.01
w/ Task.	14.70	15.85	19.35	25.79	19.89	19.77
w/ Input. (ours)	15.44	16.21	19.66	26.26	20.39	20.27

Table 12: Comparison with task-specific template on code summarization task in the full data setting. **Task.** refers to task specific prompt template for summarization task proposed by Wang et al. (2022).

Methods	Java to C#		C# to Java	
	BLEU	EM	BLEU	EM
CodeT5-base	84.03	65.90	79.87	66.90
w/ Task.	83.99	65.40	79.76	66.10
w/ Input. (ours)	85.23	66.60	81.60	67.20

Table 13: Comparison with task-specific template on code translation task in the full data setting. **Task.** refers to task-specific prompt template for translation task proposed by Wang et al. (2022).

F Expand to CodeT5-large

Methods	# Param	Ruby	JS	Go	PHP	Java	Python
Fine-Tuning							
CodeT5-small	60M	14.87	15.32	19.25	25.46	19.92	20.04
CodeT5-base	222M	15.24	16.16	19.56	26.03	20.31	20.01
CodeT5-large	737M	15.58	16.17	19.69	26.49	20.74	20.57
Prompt Tuning							
Prefix-tuning	52M	15.06	15.43	19.12	25.45	20.32	20.09
CodePrompt	52M	15.79	15.53	19.36	25.74	20.61	20.35

Table 14: Results of CodeT5-large on code summarization task.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
Limitations

☑ A2. Did you discuss any potential risks of your work?
Limitations

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Abstract and 1. Introduction

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

5. Experiment Results

☑ B1. Did you cite the creators of artifacts you used?
4. Experiment Setup

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
4. Experiment Setup and Appendix A

☐ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Not applicable. Left blank.

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.

☐ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Not applicable. Left blank.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
4. Experiment Setup

C ☑ Did you run computational experiments?

5. Experiment Results and Appendix A

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
5. Experiment Results and Appendix A

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☒ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
We used the same hyperparameter as the previous study.

☒ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
We adopted the median value among the 3 models.

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
4. Experiment Setup and Appendix A, B

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.