Prompt Tuning in Code Intelligence: An Experimental Evaluation

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu

***Abstract*—Pre-trained models have been shown effective in many code intelligence tasks, such as automatic code summarization and defect prediction. These models are pre-trained on large-scale unlabeled corpus and then fine-tuned in downstream tasks. However, as the inputs to pre-training and downstream tasks are in different forms, it is hard to fully explore the knowledge of pre-trained models. Besides, the performance of fine-tuning strongly relies on the amount of downstream task data, while in practice, the data scarcity scenarios are common. Recent studies in the natural language processing (NLP) field show that prompt tuning, a new paradigm for tuning, alleviates the above issues and achieves promising results in various NLP tasks. In prompt tuning, the prompts inserted during tuning provide task-specific knowledge, which is especially beneficial for tasks with relatively scarce data. In this article, we empirically evaluate the usage and effect of prompt tuning in code intelligence tasks. We conduct prompt tuning on popular pre-trained models CodeBERT and CodeT5 and experiment with four code intelligence tasks including defect prediction, code search, code summarization, and code translation. Our experimental results show that prompt tuning consistently outperforms fine-tuning in all four tasks. In addition, prompt tuning shows great potential in low-resource scenarios, e.g., improving the BLEU scores of fine-tuning by more than 26% on average for code summarization. Our results suggest that instead of fine-tuning, we could adapt prompt tuning for code intelligence tasks to achieve better performance, especially when lacking task-specific data. We also discuss the implications for adapting prompt tuning in code intelligence tasks.**

***Index Terms*—Code intelligence, prompt tuning, empirical study.**

Manuscript received 5 December 2022; revised 16 August 2023; accepted 28 August 2023. Date of publication 2 October 2023; date of current version 14 November 2023. This work was supported in part by the National Natural Science Foundation of China under Project 62002084, in part by the Natural Science Foundation of Guangdong Province under Project 2023A1515011959, in part by Shenzhen International Cooperation under Project GJHZ20220913143008015, in part by Shenzhen Basic Research under General Projects JCYJ20220531095214031 and JCYJ20200109113403826, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B010165002, and in part by the General Research Fund of the Research Grants Council of the Hong Kong Special Administrative Region, China, under Grant CUHK 14206921. Recommended for acceptance by T. Menzies. *(Corresponding author: Cuiyun Gao.)*

Chaozheng Wang, Yuanhang Yang, and Cuiyun Gao are with the Harbin Institute of Technology, Shenzhen 518055, China (e-mail: wangchaozheng@stu.hit.edu.cn; ysngkil@gmail.com; gaocuiyun@hit.edu.cn).

Yun Peng and Michael R. Lyu are with The Chinese University of Hong Kong, Hong Kong 999077, China (e-mail: ypeng@cse.cuhk.edu.hk; lyu@cse.cuhk. edu.hk).

Hongyu Zhang was with The University of Newcastle, Newcastle, Australia. He is now with Chongqing University, Chongqing 400044, China (e-mail: hyzhang@cqu.edu.cn).

Digital Object Identifier 10.1109/TSE.2023.3313881

I. Introduction

Code intelligence leverages machine learning, especially deep learning (DL) techniques to intelligently assist developers programming, aiming at improving their developing efficiency. The state-of-the-art DL-based approaches to code intelligence exploit the *pre-training and fine-tuning* paradigm [1], [2], [3], [4], [5], in which language models are first pre-trained on a large unlabeled text corpora and then fine tuned on downstream tasks. For instance, Feng et al. [1] proposed CodeBERT, a pre-trained language model for source code, which leverages both texts and code in the pre-training process. To facilitate generation tasks for source code, Wang et al. [2] proposed a pre-trained sequence-to-sequence model named CodeT5. These pre-trained source code models achieve significant improvement over previous approaches.

However, there exist gaps between the pre-training and the fine-tuning process of these pre-trained models. As shown in Fig. 1(a), pre-training models such as CodeBERT [1] and CodeT5 [2] are generally pre-trained using the Masked Language Modeling (MLM) objective [6], [7]. The input to MLM is the representation of the randomly masked tokens in a mixture of code snippets and natural language texts, and the models are trained to predict the masked tokens via the MLM head. However, when models are fine-tuned into the downstream tasks, e.g. defect detection, the input involves only source code, and the training objective changes to a classification problem. As shown in Fig. 1(b), the pre-trained model calculates the representation of each input code snippet [X] and predicts the label via a classifier which is also called CLS head [6]. The inconsistent inputs and objectives between pre-training and fine-tuning render the knowledge of pre-trained models hard to fully exploit, leading to sub-optimal results for downstream tasks. Besides, the performance of fine-tuning largely depends on the scale of downstream task data [8], [9], [10], [11]. For instance, the work [11] points out that utilizing limited data to fine-tune the large amount of parameters in pre-trained models is prone to cause overfitting, leading to a sub-optimal performance in downstream tasks.

Recently, prompt tuning [8], [9], [12], [13], [14] is proposed to mitigate the above issues of fine-tuning. Fig. 1(c) illustrates the concept of prompt tuning. Instead of only involving source code as input, prompt tuning firstly rewrites the input by adding a natural language prompt such as "*The code is [MASK]*" at the end of the code snippet, and then let the model predict the masked token *[MASK]*. The added prompt to rewrite the model input is also called the prompt template. There is also a verbalizer [9], [13] that maps the tokens predicted by the model to a class. By adding a prompt and verbalizer, prompt tuning reformulates the classification problem into an MLM problem, aligning the objective with the pre-training stage. This alignment unleashes the hidden power stored in the pre-trained models. Besides, the inserted prompt can involve task-specific knowledge to facilitate the adaption to downstream tasks [12], [13], [15], [16]. Equipped with the knowledge provided by prompt tuning, pre-trained models can handle low resource scenarios where a limited amount of data are available. Additionally, another training paradigm of prompt tuning called parameter efficient prompt tuning [12], [17] is also widely utilized in low resource scenarios. The parameter efficient prompt tuning tunes only partial parameters in the prompt with other parameters fixed for ameliorating overfitting and reducing the training cost.

Although prompt tuning has been proven useful in type prediction [18], the effectiveness of prompt tuning in popular code intelligence tasks still remains unexplored. The code intelligence tasks refer to the tasks about functionalities that are closely tied to source code and help to improve programmer productivity and software quality Therefore, in this article, we aim at investigating if prompt tuning is effective for code intelligence. We conduct an experimental evaluation of the effectiveness of prompt tuning on four popular code intelligence tasks: defect detection, code search, code summarization, and code translation. We mainly investigate the following four research questions (RQs):

RQ1: How effectively does prompt tuning solve four canonical code intelligence tasks?
RQ2: How well does prompt tuning handle data scarcity?
RQ3: How sensitive is prompt tuning to prompt templates?
RQ4: How effective is parameter efficient prompt tuning?

To answer the first RQ, we apply prompt tuning to the four code intelligence tasks. To answer the second RQ, we evaluate prompt tuning in data scarcity scenarios from two aspects, including low-resource settings and cross-domain settings. To answer the third RQ, we comprehensively study the influence of different prompt templates and verbalizers on model performance. In the fourth RQ, we explore the effectiveness of parameter efficient prompt tuning.

Key Findings. Based on the extensive experiments, we obtain some key findings:

- Prompt tuning brings non-trivial improvement to the performance of downstream code intelligence tasks, including classification, retrieval, and generation tasks.
- When the training data are scarce, prompt tuning can significantly outperform conventional fine-tuning. The less training data, the greater the performance improvement.
- Involving domain knowledge into the prompt template and verbalizer design is helpful to obtain further improvement.
- Parameter efficient prompt tuning can achieve comparable and even better performance than fine-tuning with significantly fewer parameters tuned in data scarcity scenarios.

Contributions. The major contributions of this article are listed as follows:

1. This article explores the performance of prompt tuning for code intelligence tasks.
2. We explore how different prompts can affect the performance of prompt tuning on code intelligence tasks.
3. We discuss the implications of our findings for adapting prompt tuning in code intelligence tasks.

Paper structure. The remainder of the paper is organized as follows. In Section II, we briefly introduce the background of prompt tuning. In Section III we present four code-related tasks and the corresponding prompts we use in different tasks. In addition, we describe the datasets, baselines, and evaluation metrics we used in experiments. Section IV discusses the experiment results and findings in detail. In Section V we summarize the implications of this article, provide case studies, and discuss the threats to validity. Sections VI and VII present the related work and conclusion.

This work is an extension of our ESEC/FSE 2022 paper [19]. Compared to the preliminary version, we explore mixed prompts in Section II and the code search downstream task in Section III. Additional discussions and experimental results of mixed prompts and code search tasks are included in Section IV. We also answer an additional research question (RQ4) about parameter efficient prompt tuning in Section IV.

II. Background

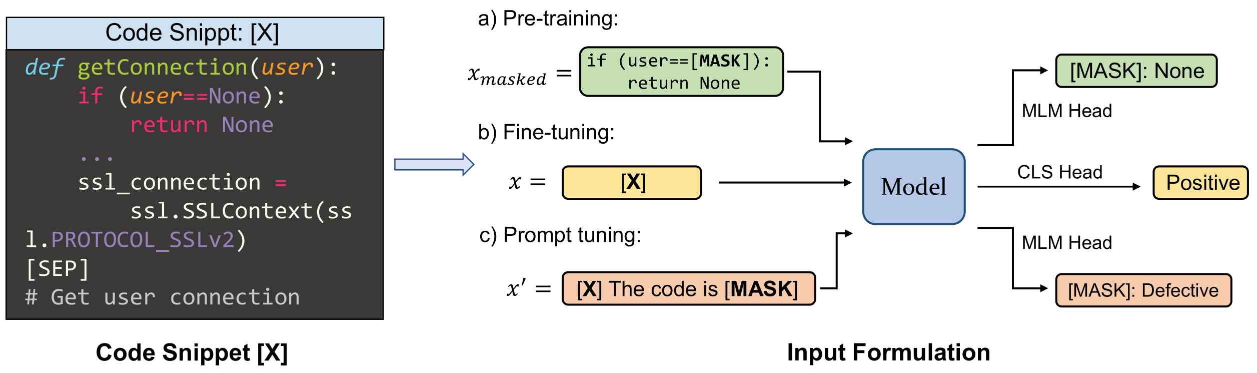
A. Fine-Tuning

Fine-tuning a pre-trained model for downstream tasks [6], [20], [21] is a prevalent paradigm in the NLP field. Fine-tuning aims at exploiting the knowledge learned by pre-trained models without learning from scratch and can be regarded as a way of applying transfer learning [22]. To adapt pre-trained models into downstream tasks, fine-tuning trains the model in a supervised way. Specifically, given a dataset that consists of task-specific samples X and corresponding labels Y, fine-tuning aims to find a set of parameters θ for the pre-trained model, that $\theta = \arg\min_{\theta} P(Y|X;\theta)$.

B. Prompt Tuning

The intuition of prompt tuning is to convert the training objective of downstream tasks into a similar form as the pre-training stage, i.e., the MLM objective [1], [6], [7]. As shown in Fig. 1(c), prompt tuning aims at predicting masked tokens in the input. It also modifies the model input by adding a natural language prompt, enabling the input format identical to the pre-training stage.

Specifically, prompt tuning employs a prompt template $f_p(x)$ to reconstruct the original input $\boldsymbol{x}$, producing new input $\boldsymbol{x}'$. As illustrated in Fig. 6, the prompt template can involve two types of reserved slots in, i.e., input slot $[X]$ and answer slot $[Z]$. The input slot $[X]$ is reserved to be filled with original input text, and the answer slot is to be filled by predicted labels such as *defective*. For the example shown in Fig. 1, prompt tuning outputs the final predicted class by a verbalizer [9], [13]. The verbalizer, denoted as $\mathcal{V}$, is an injective function which



```
def getConnection(user):
    if (user==None):
        return None
    ...
    ssl_connection =
        ssl.SSLContext(ss
l.PROTOCOL_SSLv2)
```

```
# Get user connection
```

```
if (user==[MASK]):
  return None
```

Fig. 1. Illustration on the process of pre-training, fine-tuning, and prompt tuning on defect detection task. $[SEP]$ denotes a special token in pre-trained models.

maps each predicted label word to a class in the target class set Y:

$$\mathcal{V} : W \rightarrow Y \tag{1}$$

where W indicates the label word set. For the example in Fig. 1(c), the label word set W includes "*[bad, defective]*" for buggy code snippets and "*[perfect, clean]*" for the others. The class set Y contains "+" and "−" for indicating defective and clean code, respectively. In the example, the verbalizer maps the label with the highest probability "*defective*" into the target class "+" in the class set.

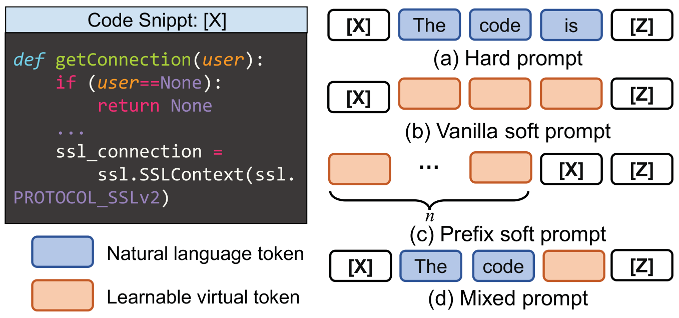
According to the flexibility of the inserted prompt, prompt tuning techniques can be categorized into two types: hard prompt and soft prompt. We elaborated on the details of each prompt type in the following.

1) Hard Prompts: The *hard prompt* [9], [10], [13] is a technique that modifies the model input by adding fixed natural language instruction (prompts). It aims to elicit task-specific knowledge learned during pre-training for the tuning stage. The hard prompt is also known as *discrete prompt* since each token in the prompts is meaningful and understandable [10], [23]. For instance, in the defect detection task, by appending "*The code is* $[Z]$." to the input code, the task objective becomes predicting the label word at the answer slot $[Z]$, such as "*defective*" or "*clean*". The designed prompt template for the defect prediction task can be formulated as:

$$f_p([X], [Z]) = \text{“}[X]\ \mathit{The\ code\ is}\ [Z]\text{”} \tag{2}$$

where $[X]$ denotes the input code. Although hard prompts have shown promising performance in previous work, the template design and the verbalizer choices are challenging. For example, the prompt template $f_p([X], [Z])$ can also be designed as "$[X]$ *It is* $[Z]$", where the label words in the verbalizer involve "*bad*" and "*perfect*".

2) Soft Prompts: The *Soft prompt* [9], [12], [24], as the name implies, is an alternative to the hard prompt. Different from hard prompts, the tokens in the soft prompt templates are not fixed discrete words of a natural language. Instead, these tokens are continuous vectors that can be learned during the tuning stage. They are also called *virtual tokens* because they are not human-interpretable. Soft prompts are proposed to alleviate the burden of manually selecting prompt templates in hard prompts.



```
def getConnection(user):
    if (user==None):
        return None
    ...
    ssl_connection =
        ssl.SSLContext(ssl.
PROTOCOL_SSLv2)
```

Fig. 2. Illustration on the different types of prompt, where $[X]$ and $[Z]$ indicate the input slot and answer slot, respectively. Both vanilla soft prompt (b) and prefix soft prompt (c) belong to the soft prompt.

There are two kinds of soft prompts, denoted as *vanilla soft prompts* and *prefix soft prompts*, respectively.

Vanilla soft prompts, as depicted in Fig. 2(b), can be obtained by simply replacing the hard prompt token with a virtual one, denoted as $[SOFT]$, such as:

$$f_p([X], [Z]) = \text{“}[X]\ [SOFT]\ [SOFT]\ [SOFT]\ [Z]\text{”} \tag{3}$$

The embedding of virtual tokens is optimized during the tuning stage.

Prefix soft prompts prepend several virtual tokens to the original input, as shown in Fig. 2(c). It can generate comparable performance with the vanilla soft prompts and hard prompts.

$$f_p([X], [Z]) = \text{“}[SOFT] * n\ [X]\ [Z]\text{”} \tag{4}$$

where n indicates the number of virtual tokens.

3) Parameter Efficient Prompt Tuning: As aforementioned in Section II-B-2 that the parameters in the soft prompt are learnable, therefore, a training paradigm for soft prompts called parameter efficient prompt tuning is proposed for reducing training cost [12], [17]. For parameter efficient prompt tuning, in the tuning stage, the pre-trained models are fixed and only the parameters in the soft prompts are tuned. In this training paradigm, the training cost can be significantly reduced. For instance, if the soft prompts have n learnable tokens and the dimension of the model embedding is d, and the number of tuned parameters is only $n \times d$ instead of the whole model.

4) Mixed Prompts: *Mixed prompts* [9], [14], also called hybrid prompts, are the combination of hard prompts and soft

TABLE I
STATISTICS OF THE DATASETS USED IN THIS ARTICLE

Tasks	Datasets	Training Set	Val. Set	Test Set
Defect Detection	Defect	21,854	2,732	2,732
Code Summarization & Code Search	Ruby	48,791	2,209	2,279
	JavaScript	123,889	8,253	6,483
	Go	317,832	14,242	14,291
	Python	409,230	22,906	22,104
	Java	454,451	15,053	26,717
	PHP	523,712	26,015	28,391
	C	104,267	13,033	13,034
Code Translation	Translation	10,300	500	1,000

prompts, making prompt templates involve both delicately designed domain knowledge and learnable parameters. In this article, we study two popular kinds of approaches to mix hard and soft prompts including *replacement* and *insertion*, respectively.

For *replacement* [14], we randomly replace some natural language tokens in the hard prompt template with soft ones. For instance, given the hard prompt shown in Equation (2), the mixed prompt version of *replacement* can be:

$$f_p([X],[Z]) = \text{“}[X]\ \mathit{The\ code}\ [SOFT]\ [Z]\text{”} \tag{5}$$

For *insertion*, following [9], we add additional learnable tokens at the beginning and the end of the hard prompts. For the aforementioned hard prompts, the mixed prompt templates generated by *insertion* can be formulated as:

$$f_p([X],[Z]) = \text{“}[SOFT]\ [X]\ \mathit{The\ code\ is}\ [Z]\ [SOFT]\text{”} \tag{6}$$

III. EXPERIMENTAL EVALUATION

A. Research Questions

We aim at answering the following research questions through an extensive experimental evaluation:

RQ1: How effectively does prompt tuning solve four canonical code intelligence tasks?
RQ2: How well does prompt tuning handle data scarcity?
RQ3: How sensitive is prompt tuning to prompt templates?
RQ4: How effective is parameter efficient prompt tuning?

We design RQ1 to verify our hypothesis that prompt tuning, which aligns the training objectives with the pre-training stage, is more effective than fine-tuning for the downstream code intelligence tasks. RQ2 aims at investigating whether prompt tuning embodies an advantage in data scarcity scenarios including low-resource and cross-domain settings. In RQ3, we aim at exploring the impact of different prompt templates, such as varying prompt types and selection of label words, on the performance of downstream tasks. In RQ4, we study the effectiveness of parameter efficient prompt tuning in low resource scenarios. In particular, we fix the weights of pre-trained models and only tune the parameters in the soft prompt.

B. Code Intelligence Tasks With Prompt Tuning

To evaluate the prompt tuning technique on source code, we adopt four downstream code intelligence tasks, namely defect detection, code search, code summarization, and code translation. We describe the details of pre-trained models and prompt templates of each task in the following.

1) Pre-trained Models: We choose CodeBERT [1] and CodeT5 [2] as the studied pre-trained models, since they are the most widely-used model and state-of-the-art model for source code, respectively.

CodeBERT [1] is an encoder-only model which is realized based on RoBERTa [7]. CodeBERT is pre-trained on CodeSearchNet [25]. It is able to encode both source code and natural language text. CodeBERT has 125 million parameters.

CodeT5 [2], a variant of text to text transfer Transformer [26], is the state-of-the-art model for code intelligence tasks. It regards all the tasks as a sequence-to-sequence paradigm with different task specific prefixes. It can solve both code understanding and code generation tasks. Code-T5 is pre-trained on a larger dataset including CodeSearchNet [25] and an additional C/C# language corpus collected by the authors. CodeT5 is classified into two versions: CodeT5-small and CodeT5-base, according to their sizes. The numbers of parameters in CodeT5-small and CodeT5-base are 60 million and 220 million, respectively.

2) Defect Detection: Given a code snippet, defect detection [27], [28] aims to identify whether it is defect prone, such as memory leakage and DoS attack. The task is defined as a binary classification task in training CodeBERT and a generation task in training CodeT5 [2], [26].

For *hard prompts*: As shown in Fig. 1(c), with prompt tuning, models predict the probability distribution over the label words. A verbalizer $\mathcal{V}$ maps the label word with the highest probability to the predicted class. One cloze-style template $f_p(\cdot)$ with an input slot $[X]$ and an answer slot $[Z]$ is designed as below:

$$f_p([X],[Z]) = \text{“}\mathit{The\ code}\ [X]\ \mathit{is}\ [Z]\text{”}$$
$$\mathcal{V} = \begin{cases} + : & [\mathit{defective}, \mathit{bad}] \\ - : & [\mathit{clean}, \mathit{perfect}] \end{cases} \tag{7}$$

where the left and right sides of : indicate the predicted class and corresponding label words. To study the impact of different prompts, we also design other prompt templates including “$[X]$ *It is* $[Z]$”, “$[X]$ *The code is* $[Z]$”, “$[X]$ *The code is defective* $[Z]$” and “*A* $[Z]$ *code* $[X]$”.

For *vanilla soft prompts*: For facilitating the comparison of hard prompts and vanilla soft prompts, we simply replace the natural language tokens in the hard prompt templates with virtual tokens for generating vanilla soft prompts. For example, “$[X][SOFT][SOFT][Z]$” is the vanilla soft prompt version of “$[X]$ *It is* $[Z]$”.

For *prefix soft prompts*: We design the prefix soft prompts by prepending a learnable prefix prompt according to Equation (4).

For *mixed prompts*: To study the effectiveness of mixed prompts, we design mixed prompt templates based on the experimented hard prompts via the approaches including *replacement* and *insertion* as described in Section II-B-4.

3) Code Search: Code search is to retrieve relevant code snippets based on a natural language query. Following previous work [1], [5], [29], we only use CodeBERT in this task because CodeT5 has not been utilized for code search in the original article [2] and a recent work [30] demonstrates that CodeT5 is not ideal for this task.

In this article, we use a two-tower architecture [5], [31], [32], [33] to measure the relevance between natural language descriptions and code snippets for code search tasks. More specifically, given the pair of description and code snippet, we obtain their embedding vectors via a shared CodeBERT and calculate the cosine similarity between two vectors as their relevance score.

For *hard prompts*, we prepend a task-specific prompt before the description and code snippet:

$$\begin{aligned} f_p([X_c]) &= \text{``}\textit{The } [LANG] \textit{ code is } [X_c]\text{''} \\ f_p([X_d]) &= \text{``}\textit{The description is } [X_d]\text{''} \end{aligned} \tag{8}$$

where the $[X_c]$ and $[X_d]$ indicate the input code and description, respectively. The $[LANG]$ denotes the name of the corresponding programming language such as Python. It is worth noting that there is no answer slot because the models in two-tower architecture are only used for generating embedding vectors; thus, there is no verbalizer in the code search task.

For the *vanilla soft prompts*, *prefix soft prompts*, and *mixed prompts*: We design the three types of prompt in the same way as the defect detection task. For example, we replace the natural language tokens in the hard prompt templates with virtual tokens for generating vanilla soft prompt templates. The prefix soft prompts are defined according to Equation (4).

4) Code Summarization: Given a code snippet, the code summarization task aims to generate a natural language comment to summarize the functionality of the code. We only utilize CodeT5 in this task because CodeBERT does not have a decoder to generate comments.

For *hard prompts*: We prepend the natural language instruction of the task to the input code, so the template can be:

$$f_p([X],[Z]) = \text{``}\textit{Generate comment for } [LANG]\ [X]\ [Z]\text{''} \tag{9}$$

where $[LANG]$, $[X]$, and $[Z]$ denote the slot of a programming language type, input slot, and the generated answer slot. The natural language instruction "*Generate comment for*" is manually pre-defined for adjusting the generation behavior of CodeT5. We also design other prompt templates for experimentation including *Summarize [LANG]*. Note that there is *no verbalizer* for the generation task.

For the *vanilla soft prompts*, *prefix soft prompts*, and *mixed prompts*: They are designed in the same way as the above two tasks.

5) Code Translation: Code translation aims to migrate legacy software from one programming language to another one. Except *hard prompts*, the other three types of prompts are similar to those for the above three tasks, so here we only describe the *hard prompts* for this task. For *hard prompts*, we design the template by prepending task-specific instructions:

$$f_p([X],[Z]) = \text{``}\textit{Translate } [X] \textit{ to } [LANG]\ [Z]\text{''} \tag{10}$$

The template explains that the model is translating the input code $[X]$ in one programming language to the answer slot in another programming language $[LANG]$.

C. Evaluation Datasets

To empirically evaluate the performance of prompt tuning for source code, we choose the datasets for the four tasks from the popular CodeXGLUE benchmark[1] [29].

Defect Detection The dataset is provided by Zhou et al. [27]. It contains 27K+ C code snippets from two open-source projects QEMU and FFmpeg, and 45.0% of the entries are defective.

Code Search We use the same dataset as the previous work [1], [5]. The dataset is from CodeSearchNet [25], which contains thousands of code snippets and natural language description pairs for six programming languages including Python, Java, JavaScript, Ruby, Go, and PHP. Additionally, we further compare fine-tuning and prompt tuning on the C Code Summarization Dataset (CCSD) dataset [34] which is randomly split into training, valid, and testing set with the ratio of 8:1:1.

Code Summarization We use the same dataset as the CodeT5 work [2]. The datasets used for the code summarization task are also CodeSearchNet and CCSD.

Code Translation The dataset is provided by Lu et al. [29], and is collected from four public repositories (including Lucene, POI, JGit, and Antlr). Given a piece of Java (C#) code, the task is to translate the code into the corresponding C# (Java) version.

D. Evaluation Metrics

1) Defect Detection: For the defect detection task, following [2], we use *Accuracy* as the evaluation metric. The metric is to measure the ability of model to identify insecure source code, defined as:

$$ACC = \frac{\sum_{i=1}^{|D|} 1(y_i == \hat{y}_i)}{|D|} \tag{11}$$

where D is the dataset and $|D|$ denotes its size. The symbol y_i and $\hat{y}_i$ indicate the ground truth label and predicted label, respectively. The $1(x)$ function returns 1 if x is True and otherwise returns 0.

2) Code Search: For code search task, following previous work [1], [5], we utilize Mean Reciprocal Rank (MRR) to evaluate the retrieval performance of prompt tuning and fine-tuning. MRR is the average of the reciprocal ranks of the correct answers of query set Q ordered by matching score, which is another popular evaluation metric for the code retrieval task. The metric MRR is calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{Rank_q}, \tag{12}$$

where $Rank_q$ indicates the rank of the ground truth code snippet in the return sequences ordered by matching score.

[1]https://github.com/microsoft/CodeXGLUE

TABLE II
HYPERPARAMETER SETTINGS

Hyperparameter	Value	Hyperparameter	Value
Optimizer	AdamW [38]	Warm-up steps	10%
Learning rate	5e-5	Training batch size	64
LR scheduler	Linear	Validation batch size	64
Beam size	10	Adam epsilon	1e-8
Max. gradient norm	1.0		

3) Code Summarization: Following previous work [1], [2], we use Bilingual Evaluation Understudy (BLEU) score [35] to evaluate the quality of generated comments. The idea of BLEU is that the closer the generated text is to the result of ground truth text, the higher the generation quality. The metric is defined as below:

$$BP = \begin{cases} 1 & if\ c > r \\ e^{1-r/c} & if\ c \leq r \end{cases} \tag{13}$$

$$BLEU = BP \cdot exp\Big(\sum_{n=1}^{N} w_n \log p_n\Big) \tag{14}$$

where p_n means the modified n-gram precision and w_n is the weight. BP represents the brevity penalty, and c and r indicate the lengths of generated comment and target comment length, respectively. In our experiments, we choose smoothed BLEU-4 score, i.e., $n = 4$, for evaluating the generation tasks following previous work [1], [2], [36].

4) Code Translation: To better measure the quality of generated code snippets, besides the BLEU score, another two metrics including Accuracy and CodeBLEU [37] are used following [2], [29]. The computation of Accuracy is the same as Equ. (11), which is the most strict metric.

CodeBLEU parses the generated code, and takes both the code structure and semantics into account for measuring the similarity between the generated code and the code in ground truth. Its computation consists of four components including n-gram matching score ($BLEU$), weighted n-gram matching score $weighted\_BLEU$, syntactic AST matching score AST_Score and semantic data flow matching score DF_Score:

$$\begin{aligned} CodeBLEU = \alpha * BLEU + \beta * weighted\_BLEU \\ + \gamma * AST\_Score + \delta * DF\_Score \end{aligned} \tag{15}$$

where $\alpha, \beta, \gamma, \delta$ are weights for each component. Following [2], [29], they are all set as 0.25.

E. Implementation Details

1) Experimental Setup: All the pre-trained models and corresponding tokenizer in our experimentation are loaded from the official repository Huggingface[2]. The overall framework is Pytorch[3]. Our implementation of prompt tuning is based on OpenPrompt [39]. We use the generic training strategy and parameter settings following the official implementation of CodeT5 [2], with details shown in Table II.

[2]https://huggingface.co/models
[3]https://pytorch.org/

TABLE III
CLASSIFICATION ACCURACY ON DEFECT DETECTION. "*" DENOTES THE STATISTICAL SIGNIFICANCE IN COMPARISON TO THE BASELINE MODELS WE REPRODUCED (I.E., TWO-SIDED t-TEST WITH p-VALUE < 0.05)

Methods		Accuracy
CodeBERT	Fine-tuning	62.12
	Prompt tuning	**64.55***
CodeT5-small	Fine-tuning	62.96
	Prompt tuning	**63.91***
CodeT5-base	Fine-tuning	65.00
	Prompt tuning	**65.82**

Specifically, for the defect detection task, we train CodeBERT and CodeT5 for 5 and 15 epochs, respectively. For the CodeT5 model, we set the maximum source length and target length as 512 and 3, respectively. For the code summarization task, because CodeBERT is an encoder-only architecture model, we focus on evaluating prompt tuning on CodeT5. We train CodeT5 for 20 epochs. The maximum lengths of the source text and target text are defined as 256 and 128. For the code translation tasks, we train the CodeT5 models for 50 epochs. The maximum length of the source text and target text is set as 256 and 256, respectively. For the code search task, we train CodeBERT for 50 epochs with an early stopping strategy following previous work [1], [29]. The maximum length of natural language descriptions and source code is 256. The batch size used in code search tasks is 32. For parameter configuration in fine-tuning, we use the configurations provided by the original work [1], [2], which were already well-adjusted. For a fair comparison, we use the same parameter configurations when implementing prompt tuning.

All the experiments are run on a server with 4 * Nvidia Tesla V100 and each one has 32GB graphic memory.

2) Fine-Tuning Baselines: We fine-tune CodeBERT and CodeT5 on the four code intelligence tasks. Specifically, we fine-tune CodeBERT for the defect detection and code search tasks, and CodeT5 for all the tasks except code search. For CodeBERT, we use the first output token (the [CLS] token) as the sentence embedding and feed it into a feed-forward network (FFN) to generate predictions. For CodeT5, all the tasks are treated as generation tasks. It takes either code or natural language sentences as input and generates target texts.

IV. EXPERIMENTAL RESULTS

A. RQ1: Effectiveness of Prompt Tuning

In this section, we study the effectiveness of prompt tuning by comparing it with the standard tuning paradigm – fine-tuning on the four code intelligence tasks: defect detection, code search, code summarization, and code translation. We present the best performance achieved by our experimented prompts. Full results can be found in our project repository[4]. We also discuss the impact of different prompts in Section IV-C.

[4]https://github.com/adf1178/PT4Code

TABLE IV
RESULTS (MRR) OF THE CODEBERT MODEL ON CODE SEARCH TASK. "*" DENOTES THE STATISTICAL SIGNIFICANCE IN COMPARISON TO THE BASELINE MODELS WE REPRODUCED (I.E., TWO-SIDED T-TEST WITH p-VALUE < 0.05)

Methods		Ruby	JavaScript	Go	Python	Java	PHP	C	Overall
CodeBERT	Fine-tuning	73.12	62.04	81.47	61.19	62.23	67.34	57.78	66.44
	Prompt tuning	**75.76****	**63.12***	**84.55***	**65.77***	**63.11***	**69.99***	**59.28***	**68.79***

TABLE V
RESULTS (BLEU-4 SCORES) OF THE CODET5 MODEL ON CODE SUMMARIZATION TASK. "*" DENOTES THE STATISTICAL SIGNIFICANCE IN COMPARISON TO THE BASELINE MODELS WE REPRODUCED (I.E., TWO-SIDED T-TEST WITH p-VALUE < 0.05)

Methods		Ruby	JavaScript	Go	Python	Java	PHP	C	Overall
CodeT5-small	Fine-tuning	13.38	14.94	21.27	17.88	18.38	24.70	24.37	19.27
	Prompt tuning	**13.60**	**15.91***	**22.33***	**18.34**	**20.60***	**26.95***	**25.86***	**20.51***
CodeT5-base	Fine-tuning	13.70	15.80	22.60	17.97	19.56	25.77	25.50	20.12
	Prompt tuning	**14.29***	**16.04**	**23.11**	**18.52**	**19.72**	**27.06***	**26.41***	**20.74***

TABLE VI
EXPERIMENTAL RESULTS ON CODE TRANSLATION TASKS: JAVA-C# AND C#-JAVA. "*" DENOTES THE STATISTICAL SIGNIFICANCE IN COMPARISON TO THE BASELINE MODELS WE REPRODUCED (I.E., TWO-SIDED T-TEST WITH p-VALUE < 0.05)

Methods		C# to Java			Java to C#		
		BLEU	Accuracy	CodeBLEU	BLEU	Accuracy	CodeBLEU
CodeT5-small	Fine-tuning	78.67	65.40	82.55	82.29	63.80	87.01
	Prompt tuning	**79.59***	**66.00**	**83.45***	**83.33***	**64.30**	**87.99***
CodeT5-base	Fine-tuning	79.45	**66.10**	83.96	83.61	65.30	88.32
	Prompt tuning	**79.76**	**66.10**	**84.39**	**83.99**	**65.40**	**88.74**

Defect Detection. Table III shows the comparison results for defect detection, in which CodeBERT and CodeT5 serve as pre-trained models. We can observe that prompt tuning always outperforms fine-tuning across different pre-trained models. For example, prompt tuning obtains an improvement of 3.30% over fine-tuning on CodeBERT. For CodeT5-small and CodeT5-base, the improvements are 1.51% and 1.26%, respectively. We also perform a statistical significance test (t-test) on the defect detection task, and the results show that prompt tuning outperforms fine-tuning at the significance level of 0.05 on both CodeBERT and CodeT5-small. The results indicate that prompt tuning is more effective than fine-tuning for pre-trained models with different architecture or different sizes on the defect detection task.

Code Search. For code search task, we follow previous work [1], [5] and use CodeBERT to compare the performance of prompt tuning and fine-tuning. In Table IV, we show the results of prompt tuning and fine-tuning of the CodeBERT model on seven different programming language datasets. From the table, we can find that prompt tuning improves the MRR consistently over fine-tuning on all of the experimented datasets. On average, prompt tuning increases the performance by 3.54%. For the statistical results, we observe that prompt tuning achieves statistically significant improvement (p-value less than 0.05) over fine-tuning on all of the seven programming languages. Such improvement demonstrates the effectiveness of prompt tuning in the code search task.

Code Summarization. Since CodeBERT is an encoder-only model, we only involve CodeT5 as the pre-trained model on the code summarization task. Table V presents the BLEU-4 scores achieved by prompt tuning and fine-tuning for different programming languages. We can observe consistent improvement on overall performance as in the code search task: compared with fine-tuning, prompt tuning obtains an improvement of 6.43% and 3.08% when using CodeT5-small and CodeT5-base as pre-trained models, respectively. Looking into a specific programming language, prompt tuning also always achieves better summarization performance than fine-tuning. It shows the largest advancement for the code written in PHP, with an increasing rate of 9.11% and 5.01% on CodeT5-small and CodeT5-base, respectively. Overall, prompt tuning can perform statistically better than fine-tuning at the significance level of 0.05 on both CodeT5-small and CodeT5-base. Specifically, prompt tuning on CodeT5-small achieves statistically significant improvement for the five programming languages. The results indicate the effectiveness of prompt tuning in the code summarization task.

Code Translation. For the task, we only involve the pre-trained CodeT5 model for evaluating the performance of prompt tuning. The results of prompt tuning and fine-tuning based on CodeT5 are depicted in Table VI. From the table, we can see that prompt tuning outperforms fine-tuning in both directions of translation. Compared with fine-tuning, prompt tuning achieves an average improvement of 1.22%, 0.85%,

```
public virtual int size(){
  lock (mutex){
    return c.size();
  }
}
```

(a) Original C# code

```
@Override public int size() {
  synchronized (mutex) {
    return c.size();
  }
}
```

(b) Ground truth Java code

```
public int size() {
  return c.size();
}
```

(c) Generated Java code by fine-tuning

```
public int size() {
  synchronized (mutex) {
    return c.size();
  }
}
```

(d) Generated Java code by prompt tuning

Fig. 3. An example for illustrating the quality of code snippets translated by fine-tuning and prompt tuning in the code translation task, respectively, where the pre-trained model is CodeT5-small.

and 0.87% on both directions for BLEU, Accuracy, and CodeBLEU, respectively. We also conduct statistical tests in the code translation task, and the results show that prompt tuning is statistically better than fine-tuning in terms of the metrics BLEU score and CodeBLEU on CodeT5-small. The improvement demonstrates the effectiveness of prompt tuning on this task. To better illustrate how prompt tuning improves the quality of code translation, we give an example in Fig. 3. From the example, we can see that using fine-tuning methods, CodeT5-small does not accurately translate the C# code into the corresponding Java version by missing an important synchronized lock statement *"synchronized (mutex)"*, while it can output more accurately with prompt tuning. We attribute this improvement to the learned prior knowledge carried by the prefix soft prompts. Through powerful prior knowledge, CodeT5 can quickly adapt to the translation of the code in C# to Java, and pay more attention to language-specific grammar details. But fine-tuning methods can only make the model learn the translation direction after multiple iterations of training, the model may fail to focus on the important part such as "*lock*" in the input.

Based on the performance of four tasks, we find that prompt tuning is more effective than fine-tuning. Another interesting observation is that the improvement of prompt tuning on CodeT5-small is 1.51%, 6.46%, and 1.22%, respectively, which is higher than that on CodeT5-base, with the increasing rate at 1.26%, 2.91%, and 0.43%, respectively. This may be attributed to that CodeT5-base is a significantly larger model than CodeT5-small (220 million v.s. 60 million parameters), and prompt tuning (768 parameters per token). The observation suggests that prompt tuning shows more obvious improvement than fine-tuning for smaller pre-trained models.

Finding 1: On average, prompt tuning outperforms fine-tuning by 2.03%, 3.68%, 4.68%, and 1.01% on the studied four code intelligence tasks, respectively Besides, compared to the improvement on CodeT5-small and CodeT5-base, the advantage of prompt tuning on smaller pre-trained models is 60% larger than that on larger models.

B. RQ2: Capability of Prompt Tuning in Different Data Scarcity Scenarios

Considering that the performance of fine-tuning is known to strongly rely on the amount of downstream data [16], [40], [41], while scenarios with scarce data in source code are common [42], [43], [44]. In this section, we study how well prompt tuning can handle data scarcity scenarios. We focus on two kinds of data scarcity settings: 1) low-resource scenario, in which there are significantly few training instances, and 2) cross-domain scenario, in which the model is trained on a similar data-sufficient domain and tested on a target domain.

Performance in low-resource scenario. We study the performance of prompt tuning in low-resource setting on the classification task, i.e., defect detection, one retrieval task, i.e., code search, and one generation task, i.e., code summarization. We simulate this setting by randomly selecting a small subset of training instances (also called *shots*) in the original dataset. To avoid randomness in data selection, we produce each subset five times with different seeds and run four times on each dataset. The average results are reported.

For the defect detection task, we choose 0, 16, 32, 64, and 128 training shots per class to create five small training subsets. Table VII presents the accuracy achieved by prompt tuning and fine-tuning regarding the five settings. Note that in zero-shot settings, no tuning data are involved. Given test data, the fine-tuning model directly generates target labels (defective or clean); while the prompt tuning model predicts the label words. By comparing the results with those in the full-data setting in Table III, we can find that the model performance shows severe drop. For the CodeT5 model, it even does not converge under the zero-shot and 16-shot settings due to the extreme lack of training data. The low results are reasonable since pre-trained models require task-specific data for better adapting to downstream tasks. However, we observe that with prompt tuning, the pre-trained models achieve significantly better performance than the models using fine-tuning. On average, prompt tuning outperforms fine-tuning by 2.59%, 2.16%, and 2.08% on CodeBERT, CodeT5-small and CodeT5-base, respectively. Note that prompt tuning under zero shot setting even outperforms prompt tuning with 32 shots and fine-tuning with 64 shots. It indicates that the knowledge of pre-trained model can be elicited by the prompt without tuning the parameters.

For code search task, we randomly select 100, 200, 300, 500, and 1000 training instances from the original datasets of six programming languages, and the results can be accessed in Fig. 4. We can observe from the figure that prompt tuning achieves obvious and consistent performance improvement over fine-tuning. For instance, when 100 training instances are available, prompt tuning increases the MRR by 20.39% on average compared with fine-tuning. We also observe that the improvement becomes less stark with the growth of training shots. The results demonstrate that prompt tuning is more advantageous on a few training data than fine-tuning.

For the code summarization task, we choose 100, 200, 300, 500, and 1000 training shots as subsets. Fig. 5 shows comparison on BLEU-4 scores of prompt tuning and fine-tuning CodeT5 models on different programming languages. We can find that although the model performance drops significantly on the subsets, prompt tuning consistently outperforms fine-tuning, showing an average improvement at 28.08% and 26.86% for CodeT5-small and CodeT5-base, respectively. The results confirm the prompt tuning's capability in low resource scenarios.

TABLE VII
CLASSIFICATION ACCURACY (%) ON DEFECT DETECTION IN LOW RESOURCE SCENARIOS. '-' DENOTES THE MODEL FAILS TO CONVERGE DUE TO EXTREME LACK OF TRAINING DATA

Methods		Zero Shot	16 Shots	32 Shots	64 Shots	128 Shots
CodeBERT	Fine-tuning	50.52	52.15	53.01	53.61	55.28
	Prompt tuning	**53.99**	**52.98**	**53.83**	**54.28**	**56.19**
CodeT5-small	Fine-tuning	-	-	51.22	52.10	54.28
	Prompt tuning	-	-	**52.36**	**53.59**	**55.04**
CodeT5-base	Fine-tuning	-	-	51.25	52.64	54.52
	Prompt tuning	-	-	**52.44**	**53.82**	**55.47**

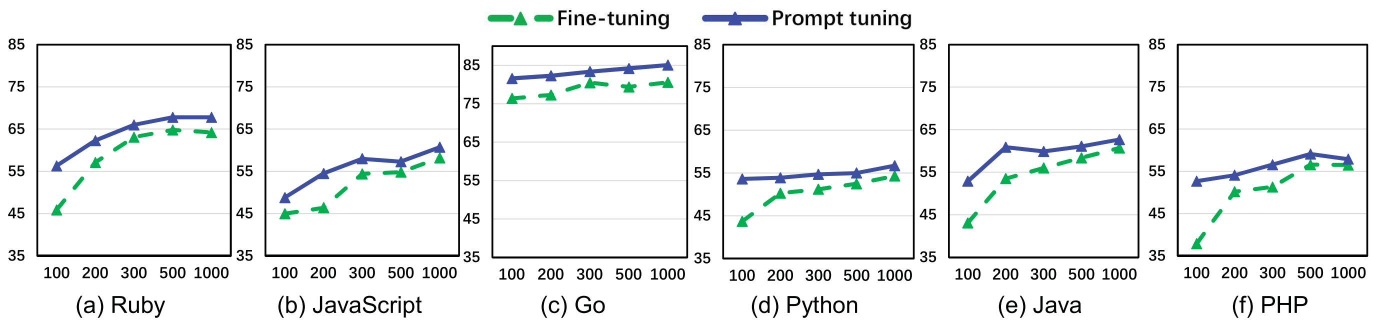


Fig. 4. Results of fine-tuning and prompt tuning on code search task in low resource scenarios. The horizontal axis indicates the number of training instances while the vertical axis means the MRR. The experimented model is CodeBERT.

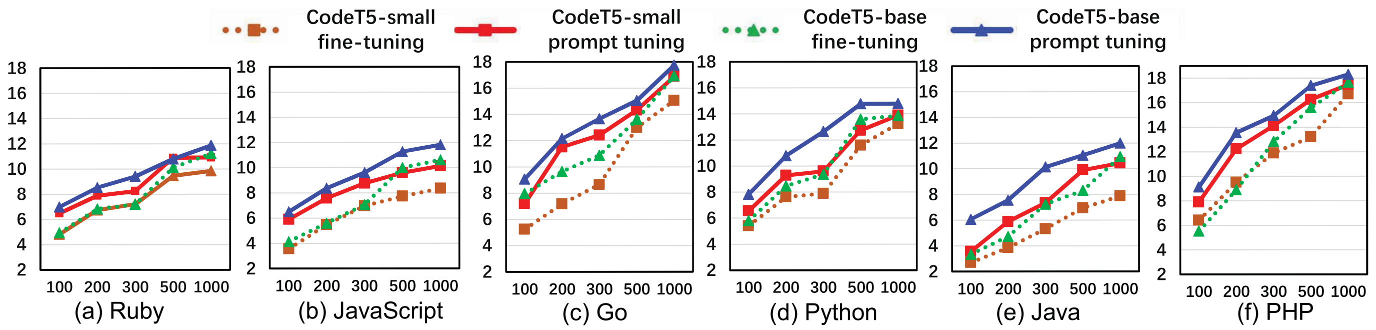


Fig. 5. Results of fine-tuning and prompt tuning on code summarization task in low resource scenarios. The horizontal axis indicates the number of training instances while the vertical axis means the BLEU-4 score.

In addition, akin to the results of code search tasks, a similar trend that prompt tuning performs better than fine-tuning on fewer training data is also observed.

Performance in the cross-domain scenario. For some programming languages, the training data are generally insufficient. As shown in Table I, the data sizes of languages such as Java and Python are greatly larger than those of languages including Java-script and Ruby. Cross-domain learning is a popular solution, i.e., transferring the knowledge of similar domains with sufficient data to the target domains with fewer data. We use the code summarization task for evaluating the performance of prompt tuning under cross-domain settings. Considering the adequacy of training data and domain similarity, we perform training on the programming language Java or Python, and evaluate the language with less data such as Ruby, JavaScript, and Go. Table VIII shows the cross-domain BLEU-4 scores achieved by CodeT5. We can observe that prompt tuning achieves better performance than fine-tuning for most cross-domain settings, except for the adaption from Python to JavaScript. With prompt tuning, the BLEU-4 scores of CodeT5-small and CodeT5-base are increased by 2.53% and 5.18% on average, respectively.

Finding 2: Prompt tuning is more effective in low-resource scenarios than fine-tuning. Specifically, prompt tuning achieves an average of more than 20% improvement on code search and code summarization tasks when 100 training instances are available. In addition, the fewer training instances, the larger the improvement achieved by prompt tuning. Besides, prompt tuning also outperforms fine-tuning by 3.85% on average in cross-domain code intelligence scenarios.

TABLE VIII
EXPERIMENTAL RESULTS (BLEU-4 SCORE) ON CROSS-LANGUAGE CODE SUMMARIZATION. THE MODELS ARE TRAINED ON PYTHON OR JAVA DATASETS, AND TESTED ON RUBY, JAVASCRIPT AND GO, RESPECTIVELY

Training	Methods	Ruby	JavaScript	Go
CodeT5-small				
Python	Fine-tuning	12.75	**12.37**	11.57
	Prompt tuning	**13.01**	12.35	**12.15**
Java	Fine-tuning	12.20	11.45	10.96
	Prompt tuning	**12.59**	**11.84**	**11.15**
CodeT5-base				
Python	Fine-tuning	13.06	12.81	12.89
	Prompt tuning	**13.37**	**13.11**	**14.27**
Java	Fine-tuning	12.67	11.50	11.88
	Prompt tuning	**13.13**	**11.99**	**12.96**

TABLE IX
CLASSIFICATION ACCURACY (%) OF COMPARING THE PERFORMANCE OF CODEBERT MODEL ON DEFECT DETECTION TASK VIA DIFFERENT PROMPT TEMPLATES. THE VERBALIZER IS FIXED AS $+$: "*BAD*", "*DEFECTIVE*"; $-$: "*PERFECT*", "*CLEAN*". THE UNDERLINED TEXTS ARE REPLACED BY VIRTUAL TOKENS IN THE CORRESPONDING VANILLA SOFT PROMPT

Hard Prompt	Vanilla Soft Prompt	Accuracy	
		Hard	Soft
$[X]$ The code is $[Z]$	$[X]$ $[SOFT]*3$ $[Z]$	63.68	63.15
A $[Z]$ code $[X]$	$[SOFT]$ $[X]$ $[SOFT]$ $[Z]$	63.36	62.95
$[X]$ It is $[Z]$	$[X]$ $[SOFT][SOFT]$ $[Z]$	63.92	63.39
The code $[X]$ is $[Z]$	$[SOFT]*2$ $[X]$ $[SOFT]$ $[Z]$	**64.17**	63.34

C. RQ3: Impact of Different Prompts

In this RQ, we explore the impact of different prompts on the performance of prompt tuning. We focus on the following four aspects: 1) hard prompt template; 2) hard prompt v.s. vanilla soft prompt; 3) mixed prompt templates, and 4) length of prefix soft prompt.

1) Different Hard Prompt Templates: There are two factors that can impact the performance of hard prompts, including the template design and verbalizer. Due to the space limit, we present the evaluation results on the classification task, i.e., defect detection, and one generation task, i.e., the code summarization. Note that we have the same observation for the code translation task.

Template Design. The natural language tokens in hard prompt templates are manually defined. To study the impact of different tokens in the template, we conduct experiments with fixed verbalizers. Tables IX and XII show the results of the defect detection task and code summarization task, respectively. Comparing the rows 2-5 in Table IX, we can find that the template design impacts the model performance. For example, when using the hard prompt "$The\ code\ [X]\ is\ [Z]$", CodeBERT outperforms the case when using "$A\ [Z]\ code\ [X]$" by 1.39%. In addition, by changing "$The\ code\ [X]\ is\ [Z]$" to "$[X]\ The\ code\ is\ [Z]$" in which only the token order is different, a drop in performance by 0.8% is observed. However, comparing row 2 and 4 in Table XII, we can find that the model performance is less affected by the template design for the code summarization task. This may be attributed to that only few prompt tokens in the templates can hardly provide helpful guidance for the large solution space in the code summarization task. Thus, we achieve that the template design for hard prompts is more important for the classification task than the generation task.

TABLE X
CLASSIFICATION ACCURACY (%) OF DIFFERENT VERBALIZERS ON THE DEFECT DETECT TASK, WHERE THE PRE-TRAINED MODEL IS CODEBERT. THE TEMPLATE IS "THE CODE $[X]$ IS $[Z]$"

Verbalizer	Accuracy
$+$: "Yes" $-$: "No"	63.08
$+$: "bad" $-$: "perfect"	63.71
$+$: "bad", "defective" $-$: "clean", "perfect"	64.17
$+$: "bad", "defective", "insecure" $-$: "clean", "perfect", "secure"	63.26
$+$: "bad", "defective", "insecure", "vulnerable" $-$: "clean", "perfect", "secure", "invulnerable"	63.10

Different Verbalizers: We fix the hard prompt template as "$The\ code\ [X]\ is\ [Z]$" and analyze the impact of different verbalizers on the model performance. Specifically, we choose task-relevant label words for the verbalizers, with the results of the defect prediction task shown in Table X. We can observe that different verbalizers influence the performance of prompt tuning. When choosing label words such as *"yes"* and *"no"* (row 2) rather than adjectives to fill the answer slot $[Z]$, the result is 0.99% lower than that of using adjectives in the verbalizer (row 3). It indicates that constructing a verbalizer with correct grammar is helpful for prediction. Comparing rows 3-6, we can also find that increasing the number of label words is not always beneficial for the model performance, which may be because more label words could introduce bias to the prediction results. When using two label words for indicating each class, the model presents the highest performance.

2) Hard Prompts vs. Vanilla Soft Prompts: As introduced in Section II-B-2, the vanilla soft prompt replaces the natural language tokens in the hard prompt with virtual tokens. The comparison results on the defect detection task are illustrated in Table IX. We experiment with different hard prompts, shown in the first column, with the corresponding vanilla soft prompts in the second column. From the results listed in the last two columns, we can find that hard prompts present better prediction accuracy than the corresponding vanilla soft prompts. For the code search task, we append a natural language task prompt to both the code snippet and description to construct a hard prompt template, and we replace the hard tokens with learnable ones to form vanilla soft prompts. The experiment results are presented in Table XI. We can find that the average performance of hard prompts (67.26) and vanilla soft prompts (67.94) is overall close to that of fine-tuning (67.89). This may be attributed to that several prompt tokens can only bring limited influence on

TABLE XI
RESULTS (MRR) OF PROMPT TUNING WITH DIFFERENT PROMPT TEMPLATES ON THE CODE SEARCH TASK. THE FIRST TWO ROWS DENOTE HARD PROMPTS, AND THE LAST TWO ROWS ARE CORRESPONDING TO VANILLA SOFT PROMPTS. THERE IS NO ANSWER SLOT OR VERBALIZER FOR THE PROMPTS OF RETRIEVAL TASKS

	$f_p(\cdot)$	Ruby	JavaScript	Go	Python	Java	PHP	Overall
CodeBERT	The $[LANG]$ code is $[X_{code}]$ The description is $[X_{description}]$	73.02	61.58	82.61	58.19	62.61	65.56	67.26
	$[SOFT]*4$ $[X_{code}]$ $[SOFT]*3$ $[X_{description}]$	71.89	62.32	83.17	60.60	63.64	66.01	67.94

TABLE XII
RESULTS (BLEU-4 SCORES) OF PROMPT TUNING WITH DIFFERENT PROMPT TEMPLATES ON THE CODE SUMMARIZATION TASK. THERE IS NO VERBALIZER FOR THE PROMPTS OF GENERATION TASKS

	$f_p(\cdot)$	Ruby	JavaScript	Go	Python	Java	PHP	Overall
CodeT5-small	Summarize $[LANG]$ $[X]$ $[Z]$	13.45	15.01	21.20	17.82	18.43	24.52	18.41
	$[SOFT]*2$ $[X]$ $[Z]$	13.33	14.96	21.17	17.93	18.29	24.61	18.38
	Generate comments for $[LANG]$ $[X]$ $[Z]$	13.44	14.96	21.24	17.90	18.52	24.46	18.42
	$[SOFT]*4$ $[X]$ $[Z]$	13.49	14.87	21.29	17.92	18.34	24.68	18.44
CodeT5-base	Summarize $[LANG]$ $[X]$ $[Z]$	13.67	15.91	22.51	18.00	19.63	25.76	19.25
	$[SOFT]*2$ $[X]$ $[Z]$	13.86	15.75	22.48	18.12	19.52	25.91	19.27
	Generate comments for $[LANG]$ $[X]$ $[Z]$	13.68	15.84	22.49	18.03	19.59	25.88	19.25
	$[SOFT]*4$ $[X]$ $[Z]$	13.74	15.82	22.63	18.06	19.60	25.83	19.28

TABLE XIII
CLASSIFICATION ACCURACY (%) OF COMPARING THE PERFORMANCE OF CODEBERT MODEL ON DEFECT DETECTION TASK VIA DIFFERENT MIXED PROMPT TEMPLATES. THE VERBALIZER IS FIXED AS +: "*BAD*", "*DEFECTIVE*"; −: "*PERFECT*", "*CLEAN*". THE "TYPE" COLUMN DENOTES THE APPROACH TO CONSTRUCT MIX TEMPLATE

Original Hard Prompt	Mixed Prompt	Type	Accuracy	
			Hard	Mix
$[X]$ The code is $[Z]$	$[X]$ The code $[SOFT]$ $[Z]$	*replacement*	63.68	64.19
	$[X]$ The $[SOFT]$ $[SOFT]$ $[Z]$	*replacement*		63.96
	$[SOFT]$ $[X]$ The code is $[Z]$ $[SOFT]$	*insertion*		**64.55**
A $[Z]$ code $[X]$	A $[Z]$ $[SOFT]$ $[X]$	*replacement*	63.36	63.45
	$[SOFT]$ A $[Z]$ code $[X]$ $[SOFT]$	*insertion*		**63.67**
$[X]$ It is $[Z]$	$[X]$ It $[SOFT]$ $[Z]$	*replacement*	63.92	63.89
	$[SOFT]$ $[X]$ It is $[Z]$ $[SOFT]$	*insertion*		**64.33**
The code $[X]$ is $[Z]$	$[X]$ The code $[X]$ $[SOFT]$ $[Z]$	*replacement*	64.17	63.40
	$[X]$ The $[SOFT]$ $[X]$ $[SOFT]$ $[Z]$	*replacement*		63.30
	$[SOFT]$ The code $[X]$ is $[Z]$ $[SOFT]$	*insertion*		**63.97**

computing embedding vectors of code snippets and corresponding descriptions. In addition, we also compare hard prompts and vanilla soft prompts in the code summarization task, and the results are shown in Table XII. Comparing the performance of hard prompts such as "$Summarize$ $[LANG]$ $[X][Z]$" and "*Generate comments for* $[LANG][X][Z]$" with the corresponding vanilla soft version, we can observe that the difference is marginal, which may be due to the large generation space of the task. Thus, we summarize that hard prompts may be more effective than the corresponding vanilla soft prompts for classification tasks, and the advantage tends to be weakened for generation tasks.

Finding 3: Prompt templates have large impact on the performance of prompt tuning. Specifically, the improvement on the defect detection task of prompt tuning varies from 1.33% to 3.30% in term of different prompt tamplates. It is crucial to construct prompt templates with suitable template design and verbalizers based on domain knowledge.

3) Different Mixed Prompt Templates: In this section, we explore the influence of different mixed prompts by constructing mixed prompt templates via *replacement* and *insertion* approaches as introduced in Section II-B-4. We construct

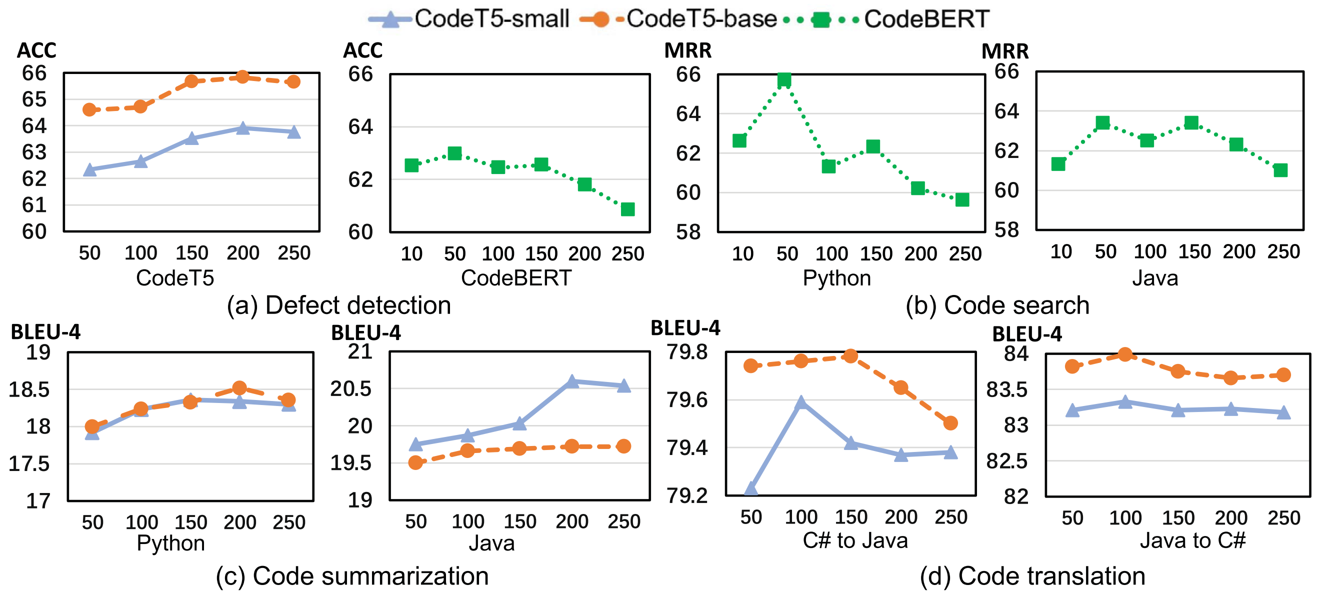


Fig. 6. Comparing the performance of CodeBERT and CodeT5 on four tasks with different prefix lengths. The horizontal axis indicates the length of prefix.

mix template prompts based on experimented hard prompts in Table IX on the defect detection task and the classification accuracy is shown in Table XIII. From the results, we can observe that the best performance of the mixed prompt is 64.55, which is higher than the best hard prompt (64.17). The results indicate that compared with hard prompts, delicately constructing mixed prompt templates can obtain further improvement. In addition, the *insertion* approach outperforms *replacement* consistently on four hard prompt templates. Specifically, using *insertion* is 0.75% better than using *replacement* on average.

Finding 4: For mixed prompts, using the *insertion* approach to construct mixed prompt templates can achieve 0.75% higher accuracy than using the *replacement* approach.

4) Different Lengths of Prefix Soft Prompts: We also study the impact of different lengths of prefix soft prompts. We illustrate the performance under different prefix prompt lengths for the four tasks in Fig. 6. As can be seen, too short or long lengths of prefix prompts can degrade the model performance. For all the tasks that CodeT5 models are applied to, prompt tuning achieves the best or nearly the best performance when the length of the prefix prompt is set to a value between 100 and 200. In our work, the prefix lengths are set as 200, 50, 200, and 100 for defect detection, code search, code summarization, and code translation tasks, respectively. We notice that in Fig. 6(c) CodeT5-small outperforms CodeT5-base on Java in the code summarization task, which may be attributed to the noise when observing the overall experiment results. For the CodeBERT model on defect detection task, the highest classification accuracy is obtained when the prefix length is 50. It is worth noting that the best accuracy is about 63, which is obviously lower than those of hard and mixed prompts, indicating that prefix prompt is not a good choice for CodeBERT on the classification tasks. For code search tasks, a prefix with a length of 50 can also bring the overall largest improvement.

Finding 5: When using the prefix prompts, the length of prompts has an impact on the model performance. The optimal length of prefix prompt varies among different code intelligence tasks and pre-trained models. Utilizing the prefix length of 200, 50, 200, and 100 for defect detection, code search, code summarization, and code translation can achieve the highest empirical results.

D. RQ4: Parameter Efficient Prompt Tuning

In this RQ, we investigate the performance of parameter efficient prompt tuning in low resource scenarios. Following previous work [12], we focus on generation tasks (i.e., code summarization in this article) to evaluate the performance of prefix prompt by fixing the weights of pre-trained models and only tuning the parameter of the learnable tokens in the soft prompt. Similar to RQ2, we randomly select 100, 200, and 300 training instances from the training set. The prompt templates used in this RQ are the same as those in RQ1. The results are shown in Table XIV.

From the table, we can observe that parameter efficient prompt tuning can achieve comparable performance as fine-tuning. For instance, when the number of training instances is 100, 200, and 300, the average BLEU-4 score of parameter efficient prompt tuning is 17%, 9%, and 10% lower than that of fine-tuning on CodeT5-small, respectively. Despite the relatively lower performance, the parameter efficient prompt tuning only needs to tune 0.25% of the parameters in fine-tuning (150K v.s. 60M). However, when training on a larger model, i.e., CodeT5-base, the overall performance of parameter efficient prompt tuning is generally better than fine-tuning.

TABLE XIV
RESULTS (BLEU-4 SCORES) OF COMPARING PARAMETER EFFICIENT PROMPT TUNING AND FINE-TUNING ON THE CODE SUMMARIZATION TASK IN LOW RESOURCE SCENARIOS. THE NUMBER OF TUNED PARAMETERS OF PROMPT TUNING IS OBTAINED BY 200 SOFT PROMPT TOKENS WITH 768 EMBEDDING DIMENSIONS

Methods		#Tuned Parameters	Ruby			JavaScript			Go		
Training instances			100	200	300	100	200	300	100	200	300
CodeT5-small	Fine-tuning	60M	**4.82**	**6.75**	**7.22**	**3.56**	**5.48**	**6.97**	**5.24**	7.18	**8.65**
	Prompt tuning	150K	3.91	5.95	6.23	2.63	4.59	5.48	4.93	**7.31**	7.95
CodeT5-base	Fine-tuning	220M	4.93	6.83	7.19	4.14	5.60	7.07	7.96	9.64	10.88
	Prompt tuning	150K	**5.22**	**7.18**	**7.87**	**5.13**	**6.19**	**7.30**	**8.81**	**10.61**	**12.10**
Methods		#Tuned parameters	Python			Java			PHP		
Training instances			100	200	300	100	200	300	100	200	300
CodeT5-small	Fine-tuning	60M	**5.42**	**7.62**	7.89	2.70	**3.86**	5.33	**6.41**	**9.50**	**11.89**
	Prompt tuning	150K	4.98	7.58	**8.24**	**2.99**	3.82	**5.45**	5.76	8.81	11.45
CodeT5-base	Fine-tuning	220M	5.80	**8.46**	9.36	3.35	4.73	**7.24**	5.52	8.90	12.83
	Prompt tuning	150K	**5.87**	7.99	**9.63**	**3.72**	**4.78**	6.97	**5.58**	**9.46**	**13.27**

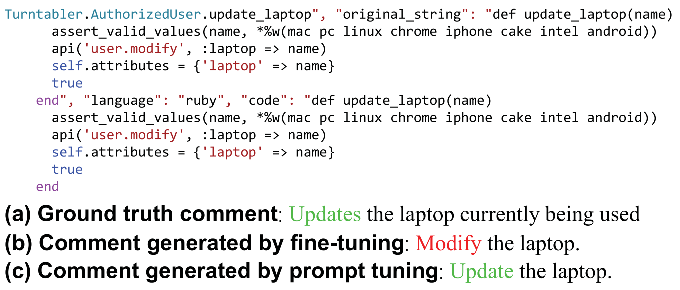

```
Turntabler.AuthorizedUser.update_laptop", "original_string": "def update_laptop(name)
      assert_valid_values(name, *%w(mac pc linux chrome iphone cake intel android))
      api('user.modify', :laptop => name)
      self.attributes = {'laptop' => name}
      true
    end", "language": "ruby", "code": "def update_laptop(name)
      assert_valid_values(name, *%w(mac pc linux chrome iphone cake intel android))
      api('user.modify', :laptop => name)
      self.attributes = {'laptop' => name}
      true
    end
```
(a) Ground truth comment: Updates the laptop currently being used
(b) Comment generated by fine-tuning: Modify the laptop.
(c) Comment generated by prompt tuning: Update the laptop.

Fig. 7. Case study on the code summarization task, where the pre-trained model is CodeT5-small.

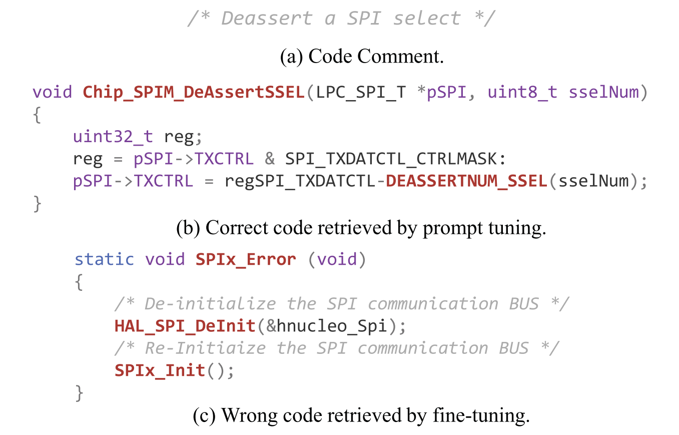

```
/* Deassert a SPI select */
```
(a) Code Comment.
```
void Chip_SPIM_DeAssertSSEL(LPC_SPI_T *pSPI, uint8_t sselNum)
{
    uint32_t reg;
    reg = pSPI->TXCTRL & SPI_TXDATCTL_CTRLMASK:
    pSPI->TXCTRL = regSPI_TXDATCTL-DEASSERTNUM_SSEL(sselNum);
}
```
(b) Correct code retrieved by prompt tuning.
```
static void SPIx_Error (void)
{
    /* De-initialize the SPI communication BUS */
    HAL_SPI_DeInit(&hnucleo_Spi);
    /* Re-Initiaize the SPI communication BUS */
    SPIx_Init();
}
```
(c) Wrong code retrieved by fine-tuning.

Fig. 8. An example for illustrating the quality of code snippets retrieved by fine-tuning and prompt tuning in the code search task, respectively, where the pre-trained model is CodeBERT.

On average, parameter efficient prompt tuning (150K) outperforms fine-tuning (220M) by 14%, 8%, and 7% with 100, 200, and 300 training instances available, respectively. The results could be explained by the fact that the pre-trained models tend to overfit on scarce data in low-resource scenarios, and tuning partial parameters in the soft prompt is already enough for the models to converge on downstream tasks.

Finding 6: In low resource scenarios, parameter efficient prompt tuning is able to achieve comparable and even better performance than fine-tuning when tuning significantly fewer parameters. When 100 instances are used for training, parameter efficient prompt tuning outperforms fine-tuning by 14% on the CodeT5-base model. In addition, parameter efficient prompt tuning performs better on larger pre-trained models such as CodeT5-base.

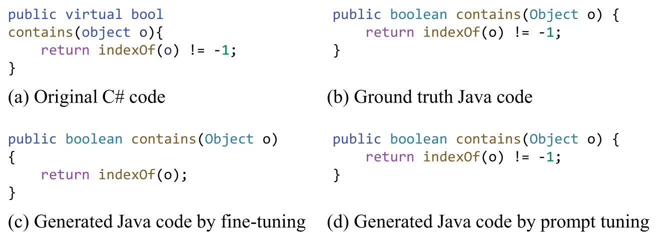
V. Discussion

A. Case Study

In this section, we provide additional case studies to qualitatively compare prompt tuning with fine-tuning. The case in Fig. 7 shows a Ruby code snippet with comments generated by fine-tuning and prompt tuning models. From the case we can observe that the fine-tuning model is misled by the word "*modify*" in the code snippet and fails to capture the main functionality "*update*". Quite the opposite, the prompt tuning model accurately summarizes the code snippet.

Furthermore, we illustrate an example in the code search task in Fig. 8. Given the code comment "*Deassert a SPI select*" (Fig. 8a), fine-tuning only focuses on the keyword "*SPI*" and ignores the most important action "*deassert*". Thus, fine-tuning retrieves a code snippet about SPI error as Fig. 8 shows and ranks the correct code at 15. However, prompt tuning successfully captures the semantics in the given code comment and retrieves the correct code snippet related to deasserting a SPI select (in Fig. 8b).

We also give another case in code translation task in Fig. 9. The original C# code (a) is to check whether object o is contained. The code translated by fine-tuning model (c) only returns the index of o but does not compare it with -1, where the code semantic changes. However, the prompt tuning model generates the identical Java code (d) with the ground truth one (b).



```
public virtual bool
contains(object o){
    return indexOf(o) != -1;
}
```

(a) Original C# code

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

(b) Ground truth Java code

```
public boolean contains(Object o)
{
    return indexOf(o);
}
```

(c) Generated Java code by fine-tuning

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

(d) Generated Java code by prompt tuning

Fig. 9. Case study on the code translation task, where the pre-trained model is CodeT5-small.

B. Implications of Findings

1) Implication on the Utilization of Pre-Trained Models: Prompt tuning performs well in adapting pre-trained models on code intelligence tasks. We observe that prompt tuning can consistently outperform fine-tuning in our experiments under full-data settings, data scarcity settings, and cross-domain settings. The advantage of prompt tuning is especially outstanding in data scarcity settings. In addition, parameter efficient prompt tuning can achieve comparable performance with fine-tuning with significantly fewer parameters tuned, greatly reducing the training cost. These advantages suggest that prompt tuning is a superior solution when there is a lack of task-specific data.

2) Implication on the Utilization of Prompts: Our experiments demonstrate that different templates and verbalizers influence the performance of code intelligence tasks. The templates that have the same semantics but different prompt tokens can lead to different performance results. Researchers could try different combinations of the words in their templates and evaluate the effectiveness through experiments. Besides, although the vanilla soft prompt is helpful to reduce the manual cost of prompt template designing, a better performance can be achieved by a well-designed hard prompt. To obtain further improvement, adding learnable tokens in existing hard prompt templates to construct mix prompts is also worth adopting. Furthermore, we find that the performance of prefix soft prompt varies with its length. Determining the best length of the prompt for a downstream task is difficult. Based on our experiments, in general, promising results can be achieved by soft prompt when the length is between 100 and 200 for generation tasks. For CodeBERT model on code search task, a prefix with length 50 is a good choice. In addition, in low resource scenarios where limited training data are available, parameter efficient prompt tuning, which fixes the pre-trained model and merely tunes the significantly less parameters in the soft prompt, can achieve comparable performance with fine-tuning.

In addition, We suggest future research to consider more characteristics of source code, like syntactic structures, in the design of the template and the choices of verbalizer. Experiment results demonstrate that domain knowledge plays an important role in the design of prompts. As code structure information has been demonstrated on the design of regular DL models for code-related tasks [5], [34], [45], [46], [47], we believe that the domain knowledge carried by them can also help the design of prompts.

C. Prompt Template Selection

In this work, we empirically select the different prompt templates based on the principles that the prompt templates are supposed to be highly related to the corresponding task and concise following previous work [2], [18]. For instance, we use the template "*Summarize [LANG] [X]*" for the code summarization task. For the classification tasks such as defect detection, we use the prompt templates with SVP structure where the input is the subject and the label word are predicative, e.g., "*The code [X] is [Z]*". In addition, the label words used in the verbalizer are adjectives. This prompt design can achieve the highest performance in our experiments.

D. Discussion on Catastrophic Forgetting

The previous work [48], [49] analyze the catastrophic forgetting problem of fine-tuning, revealing that fine-tuning pre-trained models via a large amount of data leads to forgetting the knowledge learned in the pre-training stage and reducing the generalization of the models. However, our paper is motivated by the *inconsistency* of pre-trained models between the pre-training stage and the downstream fine-tuning stage. Prompt tuning is utilized to ameliorate the inconsistency and focuses on improving the performance of pre-trained models in downstream tasks. Therefore, mitigating the catastrophic forgetting issue is not within our paper's scope.

In recent work, instruction tuning [50], [51] (tuning models with well-designed task prompts) on large language models (LLM) has emphasized the importance of prompting. High-quality downstream data with detailed task prompts are able to not only avoid LLMs from catastrophic forgetting but also generalize the model to unseen tasks. Therefore, utilizing prompt tuning to relieve catastrophic forgetting and improve the generalization of LLMs is a valuable future direction.

E. Threats to Validity

We have identified the following major threats to validity:

Limited datasets. The experiment results are based on a limited number of datasets for each code intelligence task. The selection of data and datasets may bring bias to the results. To mitigate this issue, we choose the most widely-used datasets for each code-related task, modify the seeds and run the sampling multiple times. We also plan to collect more datasets in the future to better evaluate the effectiveness of prompt tuning.

Limited downstream tasks. Our experiments are conducted on four code intelligence tasks, including one classification task, one retrieval task, and two generation tasks. Although these tasks are the representative ones in code intelligence, there are many other tasks, such as commit message generation [46] and bug fixing [52], [53]. We believe that we could obtain similar observations on these tasks since they can all be formulated as either classification tasks or generation tasks for source code. We will evaluate more tasks with prompt tuning in our future work.

Suboptimal prompt design. We demonstrate that prompt tuning can improve the performance of pre-trained models.

However, the prompts we use in this article may not be the best ones. It is challenging to design the best prompt templates and verbalizers, which will be an interesting future work.

VI. Related Work

A. Pre-training on Programming Languages

Code intelligence aims at learning the semantics of programs to facilitate various program comprehension tasks, such as code search, code summarization, and bug detection [31], [45], [54], [55], [56], [57], [58], [59]. Recently, inspired by the huge success of pre-trained models in NLP, a boom of pre-training models on programming languages arises. CuBERT [4] and CodeBERT [1] are two pioneer works. CuBERT utilizes the MLM pre-training objective in BERT [6] to obtain better representation of source codes. CodeBERT is able to learn NL-PL representation via replaced token detection task [60]. Svyatkovskiy et al. [61] and Kim et al. [62] train GPT-2 [63] on large scale programming languages for solving code completion task. The work GraphCodeBERT [5] leverages data flow graph (DFG) in model pre-training stage, making model better understand the code structure.

Apart from aforementioned encoder or decoder only models, pre-trained models that utilize both encoder and decoder are also proposed for programming languages. For example, Ahmad et al. propose PLBART [3], which is able to support both understanding and generation tasks. The work [53], [64] utilizes text to text transfer transformer (T5) framework to solve code-related tasks. Wang et al. modify the pre-training and fine-tuning stages of T5 and propose CodeT5 [2]. The authors use identifier-aware denoising pre-training and masked identifier prediction tasks for pre-training and employ multi-task learning for finetuning.

B. Prompt Tuning

The concept of prompt tuning is formed gradually. In the work [65], the authors find that the pre-trained language models have ability to learn the factual knowledge due to the mask-and-predict pre-training approach. Therefore, pre-trained language models can be regarded as a kind of knowledge base. To measure the capability of pre-trained models to capture factual information, they propose a language model analysis dataset (LAMA). Later, Jiang et al. attempt to more accurately estimate the knowledge constrained in the language model [66]. They propose LPAQA to automatically discovery better prompt templates. Schick et al. [13] propose Pattern-Exploiting Training (PET) for few shot text classification. Although the authors do not mention the word ”prompt”, they introduce the concept of patterns (the prompt templates in this article) and verbalizers. Several works focus on exploring good templates. Yuan et al. [67] replace phases in the template via a thesaurus. The work [68] utilizes a neural prompt rewriter to improve the model performance. Aforementioned works explore the manual templates or hard templates (meaning the words in the template are fixed and not learnable). Researchers also attempt to optimize the template in the training process (soft prompt) [12], [24], [69]. For example, Li et al. add an additional learnable matrix in front of the input embedding [12]. Min et al. [70] combine prompt tuning with noisy channel [71], greatly improving the performance of few shot text classification. Zhong et al. propose to initialize these matrices by natural language tokens for more effective optimization [69]. Recently, a series of works also study prompts in pre-training stage. They find that the behavior of language models can be manipulated to predict desired outputs [72], [73], [74], [75], sometimes even require no task specific training. In our work, we adapt prompt tuning in code intelligence tasks to exploit knowledge about both natural language and programming languages captured by pre-trained models.

VII. Conclusion

In this article, we experimentally investigate the effectiveness of prompt tuning on four code intelligence tasks and two pre-trained models. Our study shows that prompt tuning can outperform fine-tuning under full-data settings, data scarcity settings, and cross-domain settings. We summarize our findings and discuss implications that can help researchers exploit prompt tuning effectively in their code intelligence tasks. Our source code and experimental data are publicly available at: https://github.com/adf1178/PT4Code.

References

[1] Z. Feng et al., “CodeBERT: A pre-trained model for programming and natural languages,” in *Proc. Findings Assoc. Comput. Linguistics (EMNLP)*, 2020, pp. 1536–1547.

[2] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2021, pp. 8696–8708.

[3] W. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “Unified pre-training for program understanding and generation,” in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2021, pp. 2655–2668.

[4] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *Proc. Int. Conf. Mach. Learn.*, New York, NY, USA: PMLR, 2020, pp. 5110–5121.

[5] D. Guo et al., “GraphCodeBERT: Pre-training code representations with data flow,” in *Proc. Int. Conf. Learn. Representations*, 2021, pp. 1–18.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol. (NAACL-HLT)*, 2019, pp. 4171–4186.

[7] Y. Liu et al., “Roberta: A robustly optimized bert pretraining approach,” 2019, *arXiv:1907.11692*.

[8] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 3045–3059.

[9] X. Han, W. Zhao, N. Ding, Z. Liu, and M. Sun, “PTR: Prompt tuning with rules for text classification,” *AI Open*, vol. 3, pp. 182–192, Nov. 2022.

[10] Y. Gu, X. Han, Z. Liu, and M. Huang, “PPT: Pre-trained prompt tuning for few-shot learning,” in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (Vol. 1: Long Papers)*, 2022, pp. 8410–8423.

[11] N. Zhang et al., “Differentiable prompt makes pre-trained language models better few-shot learners,” in *Proc. Int. Conf. Learn. Representations*, 2021, pp. 1–19.

[12] X. L. Li and P. Liang, “Prefix-tuning: Optimizing continuous prompts for generation,” in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics 11th Int. Joint Conf. Natural Lang. Process. (Vol. 1: Long Papers)*, 2021, pp. 4582–4597.

[13] T. Schick and H. Schütze, “Exploiting cloze-questions for few-shot text classification and natural language inference,” in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics,* Main Vol., 2021, pp. 255–269.

[14] X. Liu, K. Ji, Y. Fu, Z. Du, Z. Yang, and J. Tang, “P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks,” 2021, *arXiv:2110.07602*.

[15] Z. Shen, Z. Liu, J. Qin, M. Savvides, and K.-T. Cheng, “Partial is better than all: Revisiting fine-tuning strategy for few-shot learning,” in *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 11, pp. 9594–9602, 2021.

[16] Y. Guo, H. Shi, A. Kumar, K. Grauman, T. Rosing, and R. Feris, “SpotTune: Transfer learning through adaptive fine-tuning,” in *Proc. IEEE/CVF Conf. Comput. Vision Pattern Recognit.*, 2019, pp. 4805–4814.

[17] G. Chen, F. Liu, Z. Meng, and S. Liang, “Revisiting parameter-efficient tuning: Are we really there yet?” in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022, pp. 2612–2626.

[18] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, “Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code,” in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–13.

[19] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence,” in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 382–394.

[20] P. Liu, X. Qiu, and X. Huang, “Recurrent neural network for text classification with multi-task learning,” in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2873–2879.

[21] M. E. Peters et al., “Deep contextualized word representations,” in *Proc. NAACL-HLT*, 2018, pp. 2227–2237.

[22] S. J. Pan and Q. Yang, “A survey on transfer learning,” *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.

[23] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” 2021, *arXiv:2107.13586*.

[24] M. Tsimpoukelli, J. Menick, S. Cabi, S. Eslami, O. Vinyals, and F. Hill, “Multimodal few-shot learning with frozen language models,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 200–212.

[25] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” 2019, *arXiv:1909.09436*.

[26] C. Raffel et al., “Exploring the limits of transfer learning with a unified text-to-text transformer,” *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 5485–5551, 2020.

[27] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 10197–10207.

[28] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, “VulDeeLocator: A deep learning-based fine-grained vulnerability detector,” *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2821–2837, Jul./Aug. 2022.

[29] S. Lu et al., “CodeXGLUE: A machine learning benchmark dataset for code understanding and generation,” in *Proc. 35th Conf. Neural Inf. Process. Syst. Datasets Benchmarks Track (Round 1)*, 2021, pp. 1–16.

[30] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, “An extensive study on pre-trained models for program understanding and generation,” in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2022, pp. 39–51.

[31] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 933–944.

[32] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When deep learning met code search,” in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 964–974.

[33] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using siamese bert-networks,” in *Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, 2019, pp. 3982–3992.

[34] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, “Retrieval-augmented generation for code summarization via hybrid GNN,” in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–16.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[36] S. Gao et al., “Code structure-guided transformer for source code summarization,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 23:1–23:32, 2023.

[37] S. Ren et al., “CodeBLEU: A method for automatic evaluation of code synthesis,” 2020, *arXiv:2009.10297*.

[38] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–18.

[39] N. Ding et al., “OpenPrompt: An open-source framework for prompt-learning,” in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL), Syst. Demonstrations*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022, pp. 105–113.

[40] W. Fu and T. Menzies, “Easy over hard: A case study on deep learning,” in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 49–60.

[41] S. Min, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, “Noisy channel language model prompting for few-shot text classification,” in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022, pp. 5316–5330.

[42] Y. Chai, H. Zhang, B. Shen, and X. Gu, “Cross-domain deep code search with meta learning,” in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 487–498.

[43] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, “Unsupervised translation of programming languages,” *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 20601–20611, Dec. 2020.

[44] Z. Sun, L. Li, Y. Liu, X. Du, and L. Li, “On the importance of building high-quality training datasets for neural code search,” in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 1609–1620.

[45] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 783–794.

[46] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, “ATOM: Commit message generation based on abstract syntax tree and hybrid ranking,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1800–1817, May 2020.

[47] W. Gu et al., “CRaDLe: Deep code retrieval based on semantic dependency learning,” *Neural Netw.*, vol. 141, pp. 385–394, Sep. 2021.

[48] S. Gao, H. Zhang, C. Gao, and C. Wang, “Keeping pace with ever-increasing data: Towards continual learning of code intelligence models,” 2023, *arXiv:2302.03482*.

[49] T. He et al., “Analyzing the forgetting problem in pretrain-finetuning of open-domain dialogue response models,” in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics,* Main Vol., 2021, pp. 1121–1133.

[50] H. W. Chung et al., “Scaling instruction-finetuned language models,” 2022, *arXiv:2210.11416*.

[51] J. Wei et al., “Finetuned language models are zero-shot learners,” in *Proc. Int. Conf. Learn. Representations*, 2021, pp. 1–46.

[52] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports,” in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24.

[53] A. Mastropaolo et al., “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 336–347.

[54] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (Vol. 1: Long Papers)*, 2016, pp. 2073–2083.

[55] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, “Modular tree network for source code representation learning,” *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–23, 2020.

[56] A. LeClair, S. Haque, L. Wu, and C. McMillan, “Improved code summarization via a graph neural network,” in *Proc. 28th Int. Conf. Program Comprehension*, 2020, pp. 184–195.

[57] Y. Wan et al., “Improving automatic source code summarization via deep reinforcement learning,” in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 397–407.

[58] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[59] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 218–229.

[60] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–18.

[61] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.

[62] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 150–162.

[63] A. Radford et al., "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[64] A. Elnaggar et al., "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," 2021, *arXiv:2104.02443*.

[65] F. Petroni et al., "Language models as knowledge bases?" in *Proc. Conf. Empirical Methods Natural Lang. Process., 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2019, pp. 2463–2473.

[66] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 423–438, Jul. 2020.

[67] W. Yuan, G. Neubig, and P. Liu, "BARTScore: Evaluating generated text as text generation," *Adv. Neural Inf. Process. Syst.*, vol. 34, Dec. 2021, pp. 27263–27277.

[68] A. Haviv, J. Berant, and A. Globerson, "Bertese: Learning to speak to bert," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics,* Main Vol., 2021, pp. 3618–3623.

[69] Z. Zhong, D. Friedman, and D. Chen, "Factual probing is [mask]: Learning vs. learning to recall," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2021, pp. 5017–5033.

[70] S. Min, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Noisy channel language model prompting for few-shot text classification," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (Vol. 1: Long Papers)*, 2022, pp. 5316–5330.

[71] L. Yu, P. Blunsom, C. Dyer, E. Grefenstette, and T. Kociský, "The neural noisy channel," in *Proc. 5th Int. Conf. Learn. Representations (ICLR)*, Toulon, France, Apr. 24–26, 2017, pp. 1–13.

[72] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[73] F. Petroni et al., "How context affects language models' factual predictions," in *Proc. Automated Knowl. Base Construction*, 2020, pp. 1–15.

[74] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, vol. 33, pp. 1877–1901.

[75] T. Schick and H. Schütze, "It's not just size that matters: Small language models are also few-shot learners," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2021, pp. 2339–2352.