Improve Code Summarization via Prompt-Tuning CodeT5

LI Huanzhen
College of Information Engineering, Jiangxi University of Technology, Nanchang 330022, Jiangxi, China
© Wuhan University 2023

Abstract: Code comments are crucial in software engineering, aiding in program maintenance and code reuse. The process of generating clear and descriptive code comments, outlining code functionality, is called code summarization. Existing code summarization methods are typically trained using transformer-based models. However, these trained models often possess limited parameters and lack specific training tasks, hindering their ability to capture code semantics effectively. This paper uses a high-capacity pre-trained model, CodeT5, for code summarization. CodeT5 is designed with an encoder-decoder architecture that excels in code summarization tasks. Furthermore, we adopt a novel paradigm, "pre-train, prompt, predict", to unlock the knowledge embedded within CodeT5. We devise a prompt template to convert input code into code prompts and fine-tune CodeT5 with these prompts—a process we term prompt tuning. Our effectiveness experiments demonstrate that prompt tuning CodeT5 with only 40% of the dataset can achieve comparable performance to fine-tuning CodeT5 with 100% of the dataset. This means our approach is applicable in few-shot learning scenarios. Additionally, our prompt learning method is not sensitive to the size of the tuning dataset. Our practicality experiments show that the performance of prompt-tuned CodeT5 far surpasses that of transformer-based models trained on code-comment datasets collected from Stack Overflow.

Key words: code summarization; transformer-based model; prompt learning; CodeT5; few-shot learning

CLC number: TP311

0 Introduction

Code comments play a crucial role in software development and maintenance, aiding developers to comprehend and reuse code without meticulous examination. For example, in the process of transforming program specifications into code, code summarization can significantly enhance the readability and comprehensibility of the code. However, manually composing code comments for extensive programming projects can be labor-intensive and time-consuming. Furthermore, as code evolves to meet the developer's requirements, maintaining consistency between the code and comments demands substantial effort. Consequently, the automated generation of high-quality code comments, called "code summarization", holds significant importance in software engineering. Code summarization pertains to producing a natural language description for a provided code snippet.

Recently, many studies have concentrated on utilizing neural-based methods to generate code summarization. For example, Iyer et al. trained a Recurrent Neural Network (RNN) with an attention mechanism to generate code summarization. Liang et al. adopted the traditional RNN-based sequence-to-sequence network with attention mechanism on the different abstraction of code. However, the RNN-based approach fails to capture long-term dependencies be-
between code tokens. To address this problem, CODE-NN\textsuperscript{[7]} uses the Long Short-Term Memory (LSTM)\textsuperscript{[2]} network combined with global attention \textsuperscript{[15]}, and HYBRID-DRL\textsuperscript{[13]} applies Reinforcement Learning (RL)\textsuperscript{[14]} to incorporate the AST structure and sequential content of snippet by using an actor-critic network. Furthermore, since the information retrieval methods perform well in code summarization, Refs.\textsuperscript{[15, 16]} combined it with the neural-based method to augment the generated code summarization. Considering the success of the Transformer\textsuperscript{[17]} in natural language generation tasks, Refs.\textsuperscript{[18, 19]} enhanced its code summarization performance by employing copy attention\textsuperscript{[20]} and relative position encoding\textsuperscript{[21]}. However, it merely treats code as the text and captures the knowledge from the sequence of code tokens. At this point, SIT\textsuperscript{[22]} introduced the structure-induced attention mechanism to capture information from syntax structure, data flow, and data dependency.

With the development of deep learning (DL), a pre-trained high-capacity code language model (e.g., CodeBERT\textsuperscript{[23]} with 125 million parameters, CodeT5\textsuperscript{[24]} with 220 million parameters) emerged. Compared with previous approaches employing traditional models for code summarization, pre-trained language models (PLMs) offer a wealth of prior knowledge acquired during the extensive pre-training phase and embedded within their parameters. In this study, we employ CodeT5 for code summarization, as it boasts an encoder-decoder architecture and excels in generation tasks.

To apply a pre-trained language model for solving downstream tasks, previous research\textsuperscript{[10, 11, 12, 13]} adheres to the "pre-train, fine-tune" paradigm for model tuning. However, this method of fine-tuning exhibits heterogeneity, often resulting in suboptimal results in downstream tasks. Our paper embraces the "pre-train, prompt, and predict" paradigm defined in Ref.\textsuperscript{[25]}, introducing a prompt to fine-tune CodeT5 homogeneously. Prompt tuning proves to be an effective approach for facilitating knowledge transfer from pre-trained language models into the downstream task. Moreover, numerous studies\textsuperscript{[26, 27]} have demonstrated the applicability of pre-trained language models to low-resource downstream tasks using prompt learning methods. Specifically, prompt learning necessitates the transformation of the downstream task into a format consistent with the pre-training task through prompts before fine-tuning the model. This approach is in contrast to simply fine-tuning the PLM with the raw code form, as illustrated in Fig. 1.

The prompt incorporates contextual information, enabling CodeT5 to discern the nature of the downstream task. In this paper, we leverage the pre-trained large code language model, CodeT5, and employ the prompt-learning method to enhance its code summarization capability.

![Fig. 1 Fine-tuning and prompt-learning](image)

Note: $X$ is the code slot and $Z$ is the comment slot.

1 Approach

This section outlines the specifics of our code summarization approach, as shown in Fig. 2. In the offline learning phase, we select CodeT5, which is pre-trained on a substantial code and comment pairs dataset. Subsequently, we devise a prompt-template to transform the input code and comment pair into a code-comment prompt for fine-tuning CodeT5, resulting in a prompt-tuned CodeT5. During the online phase, we employ the prompt-tuned CodeT5 to generate comments based on the provided code prompt.

1.1 CodeT5 Pre-Training

In this section, we introduce the framework and training mechanism of CodeT5.

1.1.1 CodeT5 framework

CodeT5 is an encoder-decoder framework sharing the same architecture as T5\textsuperscript{[37]}. It acquires general representations for a programming language (PL) and natural language (NL) by employing identifier tagging and prediction tasks to capture token-type information from PL. Additionally, CodeT5 utilizes a bimodal dual learning objective to align NL and PL. The total count of pre-trained CodeT5 parameters stands at 220 million.

1.1.2 Training mechanism

The training code data’s composition depends on whether it contains an NL description alongside the code snippets, resulting in two possible forms: PL-only unimodal or NL-PL bimodal data. The NL-PL bimodal data is concatenated into a sequence and subsequently tokenized. The tokenized words are joined by a delimiter token during training [SEP] and represented as $X = ([CLS], w_1, \ldots, w_n, [SEP], c_1, \ldots, c_m, [SEP])$, where $n$ and
are the number of NL and PL tokens, respectively. When \( m = 0 \), it takes on the PL-only unimodal data input form. Additionally, CodeT5 learns the identifier types to capture more code-specific features. Figure 3 shows the pre-training tasks of CodeT5. In the following section, we delve into the specifics of CodeT5’s training mechanism.

Identifier-Aware Denoising Pre-Training. Given \( X \), the random length of tokens is masked. The decoder is asked to predict these masked tokens. The mask rate is 15\%, and the number of masked tokens is 1 to 5. This task is referred to Masked Span Prediction (MSP), as shown in Fig. 3 (a). The masked span prediction loss is as follows:

\[
L_{MSP}(\theta) = \sum_{t=1}^{k} - \log P_\theta(X_{\text{mask}}|x_{\text{mask}}, x_{<t})
\]  

where \( \theta \) is the model parameter, \( x_{\text{mask}} \) is the masked input, \( X_{\text{mask}} \) is the number of tokens predicted by the decoder, \( k \) is the total number of tokens that need to be predicted, and \( x_{<t} \) is the span sequence generated so far.

Furthermore, CodeT5 uses two additional tasks: Identifier Tagging (IT) and Masked Identifier Prediction (MIP), to learn more code-specific syntactic and semantic information.

- **Identifier Tagging (IT)** As shown in Fig. 3 (b), the PL segment of \( X \) is mapped into a sequence of probabilities \( p=(p_1, \ldots, p_m) \) in the encoder, and computes a binary cross entropy loss for sequence labeling, and \( \theta_e \) is the encoder parameter.

\[
L_{IT}(\theta_e) = \sum_{i=1}^{m} - [y_i \log p_i + (1 - y_i) \log (1 - p_i)]
\]

- **Masked Identifier Prediction (MIP)** As shown in

---

**Fig. 2** Approach overview

**Fig. 3** Pre-training tasks of CodeT5
Fig. 3 (c), unlike the random span masking in MSP, MIP only focuses on the identifier tokens in the PL segment to mask and then predict it in an auto-regressive manner, \( I \) is the target sequence and \( x^i \) is the masked input.

\[
L_{\text{mip}}(\theta) = \sum_{i=1}^{N} \log P_e(I|x^i, I, y)
\]  

(3)

The above task is the identifier-aware denoising task to pre-training CodeT5, and three loss functions are optimized with equal probability.

**Bimodal Dual Generation.** The PL-NL bimodal data is employed for pre-training CodeT5, facilitating bi-directional conversion as depicted in Fig. 3 (d). This bi-directional conversion encompasses NL to PL generation and PL to NL generation, commonly called dual tasks. It is worth noting that the dual task can be regarded as a specialized span masking approach, where the NL or PL segment is masked from the input. CodeT5 leverages the dual task to enhance the alignment between the NL and PL components.

### 1.2 Prompt-Based Learning

To fully explore CodeT5’s adaptability to the downstream task, code summarization in this case, we employ the prompt learning method to fine-tune CodeT5. Specifically, for a given tuning dataset \( D \), each piece of data comprises code and its corresponding comment. Formally, it can be expressed as \( D = \{(x_i, y_i)\}_{i=1}^{N} \), where \( x_i \) represents the \( i \)-th code snippet out of the total \( N \) samples, and \( y_i \) represents the code comment for \( x_i \). Rather than directly utilizing \( D \) to fine-tune CodeT5, we create a prompt template to transform the input into a prompt, which serves as a natural language instruction. The aim is to extract task-specific knowledge acquired during pre-training for the downstream task. The defined prompt template is as follows:

\[
T() = "\text{The < language > LAN < /language > code < code > X < /code > comment is < comment > Z < /comment > }" \]

Within the prompt template, \( X \) represents the input code snippet slot, \( Z \) signifies the comment slot generated by CodeT5, and LAN denotes the type of programming language slot (e.g., Java, Python). The natural language present in the prompt serves to instruct CodeT5 regarding the specific task it needs to complete, and the tag (e.g., \(<\text{code}>>/<\text{code}>\) designates the specific content that needs to be generated.

For prompt learning, with the defined template \( T() \), we convert the input data \( x \) as the \( T(x) \). The learning objective is to minimize the prediction loss:

\[
L(\theta') = \sum_{i=1}^{N} -\log P_e(y_i|T(x_i), y_i)
\]

(4)

where \( \theta \) is the CodeT5 parameter, and \( \theta' \) is the tuned CodeT5 parameter.

### 1.3 Comment Generation

When a code snippet \( X_i \) is provided, we employ the prompt-tuned CodeT5 to generate the associated comment, which pertains to code summarization in this context. To accomplish this, we convert \( X_i \) into \( T(X_i) \) using the transformation \( T() \), and then feed it into the prompt-tuned CodeT5 to generate the comment, thereby filling the slot \( Z \) within the \( T(x) \).

### 2 Evaluation

This section will evaluate our code summary approach from two perspectives: effectiveness and usefulness. To assess the efficacy (RQ1), we scrutinize the impact of prompt-tuning data size on our approach and compare its performance to fine-tuning. To evaluate the usefulness (RQ2), we examine how well our approach generates code summarization for code snippets sourced from Stack Overflow, compared with the transformer-based approach\(^{14}\).

All experiments are executed using Python 3.7 on an NVIDIA GeForce RTX 3090. The operating system is Ubuntu 20.04.3 LTS. The hyperparameters employed for prompt-tuning CodeT5 are detailed in Table 1. It is worth noting that the source length represents the maximum length of the code tokens, while the target length signifies the maximum length of the comment tokens.

<table>
<thead>
<tr>
<th>Table 1 Hyperparameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperparameter</td>
</tr>
<tr>
<td>Optimizer</td>
</tr>
<tr>
<td>Learning rate</td>
</tr>
<tr>
<td>LR scheduler</td>
</tr>
<tr>
<td>Gradient norm</td>
</tr>
<tr>
<td>Source length</td>
</tr>
</tbody>
</table>

### 2.1 Evaluation Metrics

We use the BELU-4\(^{30}\) score and METEOR\(^{30}\) metrics to measure the performance of the code summarization approach.
2.1.1 BELU-4

BLEU (Bilingual Evaluation Understudy) is a metric for assessing the similarity between the reference and generated text through N-gram analysis. This metric takes into account the coherence and fluency of the generated text. However, the BLEU score may register as exceptionally high when the generated text comprises only a portion of the reference text and exhibits a high level of overlap. To mitigate this issue, we introduce a short penalty (BP) to counteract the potential bias stemming from the length of the generated text, thereby enhancing the rigor of the measurement. The calculation formula for BP is as follows:

$$ BP = \begin{cases} 1 & \text{if } I_s > I, \\ \exp \left( -\frac{I_s}{I} \right) & \text{if } I_s \leq I, \end{cases} $$

where $I_s$ is the reference text length, $I$ is the translated text length. When the length of the translated text is greater than the length of the reference text, the penalty coefficient is 1, meaning no punishment.

As the accuracy of each N-gram statistic diminishes exponentially with the order’s increase, a balanced approach is needed to account for each order statistic’s effect. To achieve this balance, the geometric average form is employed for averaging, followed by weighting and multiplication by the length penalty factor. The ultimate BLEU calculation formula is as follows:

$$ \text{BLEU} = \text{BP} \times \exp \left( \sum_{n=1}^{N} \frac{1}{n} \log \left( P_n \right) \right) $$

where $P_n$ is the geometric average precision of n-gram, and $N$ is the number of consecutive occurrences. Here, we use BELU-4 (i.e., $N = 4$) to evaluate the quality of the generated code comment.

2.1.2 METEOR

The METEOR score is a tool for computing the similarity between reference text and generated text. METEOR incorporates various factors, including word choice, stem matching, and phrasal alignment. METEOR stands out compared with BLEU-4 due to its heightened sensitivity to language nuances and ability to effectively capture the overall meaning and quality of the translated text. The METEOR calculation formula is as follows:

$$ \text{METEOR} = \left( 1 - \gamma \cdot \frac{\text{frag}^\beta}{{\alpha \cdot P + (1 - \alpha) \cdot R}} \right) $$

where $\gamma$, $\alpha$, and $\beta$ are the coefficients. frag is calculated with the ch/m, where ch is the number of chunks, and m is the number of mapped unigrams found between the two strings. $P$ and $R$ are the precision and recall, respectively. $P = \frac{m}{t}$ and $R = \frac{m}{r}$, where $t$ is the generated text length, and $r$ is the reference text length.

In our experiment, we calculate BLEU-4 and METEOR to measure the quality of the generated code comments, which correspond to code summarization. These metrics fall within a range of [0, 1], where a higher value indicates a more favorable outcome.

2.2 Effectiveness Evaluation (RQ1)

In this section, we introduce the details of effectiveness evaluation.

2.2.1 Motivation

We adopt a novel paradigm, "pre-train, prompt, and predict", to fine-tune CodeT5 for code summarization. In the process of model tuning, the size of the tuning dataset can significantly impact the performance of the fine-tuned model. In this context, we initially employ varying data sizes to assess the effectiveness of our approach, specifically tuning CodeT5 for the code summarization task using different data sizes. Additionally, we evaluate the effectiveness of our approach by making comparisons with the "pre-training fine-tuning" paradigm.

2.2.2 Dataset

To ensure the reliability of our results, we utilize a substantial code corpus for fine-tuning and testing CodeT5, specifically CodeSearchNet\cite{29}. This resource comprises thousands of code-comment pairs for six programming languages: Python, Java, JavaScript, Ruby, Go, and PHP. It is important to note that the comments are considered as the ground truth. Initially, we transform all code-comment pair data into the Code-Prompt format, based on the template defined in Section 1.2. The specifics of the converted data are presented in Table 2.

We divide the tuning and validation sets into five

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Tuning set</th>
<th>Validation set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby</td>
<td>48 791</td>
<td>2 209</td>
<td>2 279</td>
</tr>
<tr>
<td>JavaScript</td>
<td>123 889</td>
<td>8 253</td>
<td>6 483</td>
</tr>
<tr>
<td>GO</td>
<td>317 832</td>
<td>14 242</td>
<td>14 291</td>
</tr>
<tr>
<td>Python</td>
<td>409 230</td>
<td>22 906</td>
<td>22 104</td>
</tr>
<tr>
<td>Java</td>
<td>454 451</td>
<td>15 053</td>
<td>26 717</td>
</tr>
<tr>
<td>PHP</td>
<td>523 712</td>
<td>26 015</td>
<td>28 391</td>
</tr>
<tr>
<td>Total</td>
<td>1 877 905</td>
<td>88 678</td>
<td>100 265</td>
</tr>
</tbody>
</table>
equal portions, each accounting for 20% of the total. We use x-pl (x=1, 2, 3, or 4) to represent the number of partitions used for fine-tuning and validating CodeT5. For example, 2-pl indicates that 40% of each language data in the tuning set is used to fine-tune CodeT5, while the remaining 60% is designated for validation. Furthermore, it is important to note that the tuning, validation, and test sets contain distinct data.

2.2.3 Experiment setting

In this experimental study, we aim to assess how our approach’s performance is influenced by two key factors: the size of the tuning dataset and the different tuning ways.

For the different tuning data sizes, we use four sizes of tuning datasets (1-pl, 2-pl, 3-pl, 4-pl) to tune CodeT5. Note that x-pl contains the tuning set from the six programming languages, and each account for x>20% of the dataset. We refer to the prompt-tuned CodeT5 as the CodeT5p for the x-pl portion. CodeT5p is tested separately on test sets for the six programming languages. Furthermore, we use a zero-shot setting to contrast the effectiveness of prompt learning. In the zero-shot setting, the vanilla CodeT5 is used directly on the test set to generate code comments for code snippets.

For the different tuning ways, we explore the prompt learning and fine-tuning impact on our approach. Specifically, we use the 5-pl portion (i.e., all the tuning set) dataset to tune CodeT5 with the Prompt-learning to obtain a prompt-tuned CodeT5p. In addition, we use all the code-comment tuning set (i.e., the code does not convert into the code prompt format) to tune CodeT5 with the fine-tuning to obtain a fine-tuned CodeT5p. CodeT5p and CodeT5p both test on the same test set.

2.2.4 Experiment result

1) Various data size

Table 3 presents the experimental results. The vanilla CodeT5 (without prompt learning using the tuning set) only achieves an average BLEU-4 score of 0.000 6 and a METEOR score of 0.013 0 in the zero-shot setting. This indicates that the task-agnostic CodeT5 does not recognize the specific tasks it needs to accomplish, resulting in generated code comments that exhibit little correlation with the ground truth. Moreover, the vanilla CodeT5 often generates repetitive text, such as function names. For example, the generated comment "observable(); observable(); observable(); observable();;}" repeats the "observable" function in the code snippet, whereas the ground truth comment is "Wraps an ObservableSource into an Observable if not already an Observable." There is no semantic connection between these generated and ground truth comments.

Nevertheless, when we apply 1-pl for prompt learning, denoted as CodeT5p, the BLEU-4 score experiences a significant rise from 0.000 6 to 0.506 3 (a substantial increase of 0.505 7), and the METEOR score also significantly increases from 0.013 0 to 0.729 5 (an increase of 0.716 5). This demonstrates that prompt learning effectively enhances the performance of CodeT5 in code summarization, resulting in more fluent generated comments. Prompt learning accomplishes this by enabling CodeT5 to capture the semantic associations between the code and comments.

As the prompt learning data size gradually increases, the average BLEU-4 and METEOR scores exhibit a more incremental growth than the significant improvement observed from the zero-shot setting to CodeT5p. Furthermore, the model’s performance stabilizes at CodeT5p, with an average BLEU-4 score of 0.531 9 and an average METEOR score of 0.751 8, showing only marginal improvements at CodeT5p, with an average BLEU-4 score of 0.543 3 and an average METEOR score of 0.760 3. This suggests that prompt learning, in-

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby</td>
<td>0.001 0</td>
<td>0.010 7</td>
<td>0.449 4</td>
<td>0.684 4</td>
<td>0.469 4</td>
<td>0.724 4</td>
</tr>
<tr>
<td>JavaScript</td>
<td>0.001 0</td>
<td>0.014 6</td>
<td>0.446 9</td>
<td>0.714 9</td>
<td>0.456 9</td>
<td>0.724 8</td>
</tr>
<tr>
<td>GO</td>
<td>0.000 1</td>
<td>0.008 6</td>
<td>0.527 8</td>
<td>0.735 8</td>
<td>0.547 0</td>
<td>0.738 7</td>
</tr>
<tr>
<td>Python</td>
<td>0.000 5</td>
<td>0.020 9</td>
<td>0.496 1</td>
<td>0.695 0</td>
<td>0.516 1</td>
<td>0.706 0</td>
</tr>
<tr>
<td>Java</td>
<td>0.000 2</td>
<td>0.010 4</td>
<td>0.526 9</td>
<td>0.736 6</td>
<td>0.546 9</td>
<td>0.746 5</td>
</tr>
<tr>
<td>PHP</td>
<td>0.001 0</td>
<td>0.012 7</td>
<td>0.590 5</td>
<td>0.810 3</td>
<td>0.600 5</td>
<td>0.819 2</td>
</tr>
<tr>
<td>Average score</td>
<td>0.000 6</td>
<td>0.013 0</td>
<td>0.506 3</td>
<td>0.729 5</td>
<td>0.522 9</td>
<td>0.743 3</td>
</tr>
</tbody>
</table>

Table 3 The effects of different data sizes on Prompt Learning
steal of tuning data size, plays a more fundamental role in enhancing CodeT5. Additionally, CodeT5 performs consistently well across all six languages, indicating that prompt-tuned CodeT5 effectively identifies code written in different programming languages. This ability benefits from its pre-trained task.

2) Prompt learning versus Fine-tuning

Table 4 displays the experimental results. We can see that CodeT5 achieves a BLEU-4 score of 0.5467 and a METEOR score of 0.7856, which is 0.1625 and 0.1860 higher than CodeT5’s BLEU-4 and METEOR scores, respectively. It is worth noting that CodeT5 also attains significantly higher average BLEU-4 and METEOR scores compared with CodeT5’s (i.e., 0.5063 vs. 0.3842 and 0.7295 vs. 0.5996). This underscores the effectiveness of tuning CodeT5 with prompt learning for code summarization. The presence of natural language prompts in the defined template aids CodeT5 in recognizing the content generated in the specific slot during prompt learning or testing. For instance, the LAN slot signals CodeT5 to generate code comments for a particular programming language code. This enables prompt-tuned CodeT5 to be utilized for code summarization across various programming languages and yield significant performance improvements. In contrast, when compared with prompt learning, CodeT5 cannot capture any prompts from the tuning dataset during fine-tuning or testing.

### Table 4 The performance between CodeT5 and CodeT5

<table>
<thead>
<tr>
<th>Programming language</th>
<th>CodeT5</th>
<th>CodeT5&lt;sub&gt;FT&lt;/sub&gt;</th>
<th>CodeT5&lt;sub&gt;FT&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby</td>
<td>0.4907</td>
<td>0.3409</td>
<td>0.5775</td>
</tr>
<tr>
<td>JavaScript</td>
<td>0.4993</td>
<td>0.3336</td>
<td>0.5391</td>
</tr>
<tr>
<td>GO</td>
<td>0.5374</td>
<td>0.3882</td>
<td>0.5931</td>
</tr>
<tr>
<td>Python</td>
<td>0.5439</td>
<td>0.3900</td>
<td>0.6245</td>
</tr>
<tr>
<td>Java</td>
<td>0.5701</td>
<td>0.3795</td>
<td>0.5897</td>
</tr>
<tr>
<td>PHP</td>
<td>0.6385</td>
<td>0.4817</td>
<td>0.6735</td>
</tr>
<tr>
<td><strong>Average score</strong></td>
<td>0.5467</td>
<td>0.3842</td>
<td>0.5996</td>
</tr>
</tbody>
</table>

### 2.3 Practicality Evaluation (RQ2)

2.3.1 Motivation

In this practicality evaluation, our objective is to assess the performance of our prompt-tuned CodeT5 for code summarization on real partial code, compared with an existing transformer-based approach\cite{Ahmad et al.2018}.

2.3.2 Dataset

In this study, we have compiled a test dataset from Stack Overflow, a widely used Q&A website. Due to the manual effort required and the transformer-based approach\cite{Ahmad et al.2018} being limited to Java and Python datasets, we have collected data for two programming languages, namely Java and Python, from Stack Overflow. We have gathered 120 code-comment pairs, with 60 pairs for each programming language. To maintain the quality of the collected code-comment pairs, we enlisted the assistance of three graduate students, each with experience in both Java and Python development. Each student was tasked with collecting 15 code-comment pairs for Java and 15 for Python. They were required to meticulously verify the paired comments and code, considering the code’s logic and the comment’s semantics. Additionally, they cross-examined the collected code-comment pairs dataset to ensure the reliability of the 120 pairs further. It is important to note that the collected comments serve as the ground truth.

2.3.3 Experiment setting

In this phase, we execute the prompt-tuned CodeT5 on the 120 collected code snippets. As a baseline comparison, we also employ the transformer-based model trained by Ahmad et al.\cite{Ahmad et al.2018} for testing on the same set of 120 code snippets.

2.3.4 Experiment result

The experiment results are detailed in Table 5, where it becomes evident that our prompt-tuned CodeT5 outperforms the transformer-based approach significantly. Specifically, our CodeT5 boasts an average BLEU-4 score of 0.6472 and an average METEOR score of 0.8573, surpassing transformer-based BLEU-4 and METEOR scores by 0.2915 and 0.4312, respectively. This underscores the superior practicality of our prompt-tuned CodeT5 in code summarization.

### Table 5 Performance on Stack Overflow dataset

<table>
<thead>
<tr>
<th>Programming language</th>
<th>CodeT5</th>
<th>Transformer-based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLEU-4</td>
<td>METEOR</td>
</tr>
<tr>
<td></td>
<td>BLEU-4</td>
<td>METEOR</td>
</tr>
<tr>
<td>Python</td>
<td>0.6382</td>
<td>0.3691</td>
</tr>
<tr>
<td>Java</td>
<td>0.6561</td>
<td>0.3423</td>
</tr>
<tr>
<td><strong>Average score</strong></td>
<td>0.6472</td>
<td>0.3557</td>
</tr>
</tbody>
</table>

This advantage can be attributed to two key factors. Firstly, Ahmad et al.\cite{Ahmad et al.2018} treated code as a text for training their transformer-based model, resulting in a model that
struggles to comprehend code semantics and generate high-quality comments. In contrast, CodeT5 leverages the vast amount of code semantics knowledge acquired during its pre-training task. However, the most crucial factor is the capacity of CodeT5 as a high-capacity model, featuring 220 million parameters. In contrast, Ahmad et al. trained their model with limited data, comprising only 120,000 code-comment pairs.

3 Conclusion

This paper adheres to a novel paradigm, "Pre-train, prompt, and predict", for fine-tuning CodeT5 to generate high-quality code summarizations. To achieve this, we have introduced a prompt template that facilitates the conversion of input code into a code prompt, subsequently fine-tuning CodeT5 using this code prompt, a process we refer to as prompt tuning. The experiments demonstrate the effectiveness and practicality of our code summarization approach. Prompt learning yields substantial benefits even with just a 20% tuning set. Furthermore, the performance of the prompt-tuned CodeT5 surpasses that of the transformer-based model in the context of code summarization.

References


