The Journal of Systems and Software 216 (2024) 112149

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Project-specific code summarization with in-context learning ☆

Shangbo Yun [a], Shuhuai Lin [b], Xiaodong Gu [a], Beijun Shen [a,*]

[a] School of Software, Shanghai Jiao Tong University, Shanghai, China
[b] Department of Electrical and Computer Engineering, Carnegie Mellon University, Mountain View, United States

ARTICLE INFO

Keywords:
Prompt generation
Project-specific code summarization
Large language model
In-context learning

ABSTRACT

Automatically generating summaries for source code has emerged as a valuable task in software development. While state-of-the-art (SOTA) approaches have demonstrated significant efficacy in summarizing general code, they seldom concern code summarization for a specific project. Project-specific code summarization (PCS) poses special challenges due to the scarce availability of training data and the unique styles of different projects. In this paper, we empirically analyze the performance of Large Language Models (LLMs) on PCS tasks. Our study reveals that using appropriate prompts is an effective way to solicit LLMs for generating project-specific code summaries. Based on these findings, we propose a novel project-specific code summarization approach called P-CodeSum. P-CodeSum gathers a repository-level pool of (code, summary) examples to characterize the project-specific features. Then, it trains a neural prompt selector on a high-quality dataset crafted by LLMs using the example pool. The prompt selector offers relevant and high-quality prompts for LLMs to generate project-specific summaries. We evaluate against a variety of baseline approaches on six PCS datasets. Experimental results show that the P-CodeSum improves the performance by 5.9% (RLPG) to 101.51% (CodeBERT) on BLEU-4 compared to the state-of-the-art approaches in project-specific code summarization.

1. Introduction

Code summarization, which generates a brief natural language description for source code, can greatly facilitate developers in program understanding (Xie et al., 2022). In recent years, code summarization has been significantly advanced by large language models (LLMs). By pre-training on a large-scale monolingual code corpus, pre-trained language models (PLMs) learn the generic knowledge of multiple natural languages and programming languages. This enables a more efficient adaptation to the downstream code summarization task through fine-tuning on parallel code corpus (i.e., code-summary pairs).

While demonstrating a remarkable advance, state-of-the-art (SOTA) code summarization models (Wu et al., 2020) are typically general-purpose; that is, they are trained and evaluated on large independent (code, summary) pairs collected from a variety of software projects. In practice, however, developers are more concerned with code summaries for their working projects, i.e., project-specific code summarization (PCS) (Xie et al., 2022).

A good general-purpose code summarization model is not guaranteed to be a good project-specific one. This is because the former typically focuses on capturing common semantic features across a variety of projects. Thus the generated summaries could fail to capture project-specific characteristics. In contrast, a software project is usually designed to meet needs in specific domains (e.g., scientific computing, business prediction, government information management) featured by their conventions that prescribe specific coding concepts, colloquialisms, and idioms. In addition, there are algorithms and data structures that are specific to projects and domains, exhibiting coding patterns that only developers involved in that project can recognize.

The sticking point here, however, is that project-specific data, especially in the early evolution of software, are severely insufficient. LLMs depend on a large number of samples for fine-tuning. Unfortunately, gathering high-quality data for a specific project is expensive or infeasible. This leads to a data scarcity problem in the project-specific code summarization task (Xie et al., 2022; Ahmed and Devanbu, 2022).

One promising solution is in-context learning with LLMs (Brown et al., 2020), where we prepend a few demonstrations (i.e., input–output examples) into the model's context window and let the model generate the desired output. By leveraging this fragmented example data, we can extend the model's capabilities. In the PCS task, we can provide the model with some code snippets and corresponding summary examples. By learning from these examples, the model can gradually understand how to generate project-specific code summaries.

☆ Editor: Dr. Aldeida Aleti.

* Corresponding author.
E-mail addresses: sjtu_yun@sjtu.edu.cn (S. Yun), shuhuail@andrew.cmu.edu (S. Lin), xiaodong.gu@sjtu.edu.cn (X. Gu), bjshen@sjtu.edu.cn (B. Shen).

https://doi.org/10.1016/j.jss.2024.112149
Received 19 January 2024; Received in revised form 24 April 2024; Accepted 24 June 2024
Available online 28 June 2024

However, direct in-context learning on LLMs usually requires a set of carefully curated prompts by humans, which is costly and inefficient. Crafting effective prompts often demands expertise in both the domain and the model itself. This is not only time-consuming and labor-intensive but also relies heavily on human resources.

To tackle these challenges, we conduct an empirical study on how to design good prompts for PCS tasks. Specifically, we utilize ChatGPT (OpenAI, 2022) on two project-specific datasets, h2oai/h2o-3 and RestComm/jain-sle, and explore the performance under different formulations of prompts. Experimental results indicate that the content and position of instruction description play a significant role in LLM's effectiveness on project-specific code summarization. Moreover, choosing the appropriate number and instances of in-context examples also leads to good prompts.

Based on the empirical findings, we propose P-CodeSum, which focuses on automatically generating high-quality prompts to guide LLMs in generating code summaries tailored to specific projects in few-shot scenarios. Specifically, P-CodeSum gathers a repository-level pool of (code, summary) examples to characterize the project-specific features. P-CodeSum leverages LLMs themselves to craft a dataset of in-context examples using the example pool. Then, it trains a prompt selector (i.e., a classifier based on CodeBERT Feng et al., 2020) for each project using the collected dataset. The selector is then used to select appropriate in-context learning examples from the example pool. The selected in-context examples are accompanied by the code query to guide LLMs in generating high-quality code summaries tailored to specific projects.

We compare P-CodeSum with SOTA baseline methods on the project-specific code summarization datasets from six projects from CodeXGLUE. The experimental results indicate that P-CodeSum outperforms all baseline methods in the task of generating project-specific code summaries in few-shot scenarios. Ablation studies confirm the effectiveness of the proposed prompt selector and the robustness of P-CodeSum under different hyperparameters.

In summary, the contributions of this work are as follows:

- We empirically study the performance of LLMs on project-specific code summarization tasks and explore key factors to high-quality prompts.
- We propose a neural prompt selector trained on data generated by LLM, obtaining project-specific examples for in-context learning in the few-shot situation.
- We conduct comprehensive experiments to evaluate P-CodeSum. Results demonstrate the high quality of prompts generated by our selector and the significant improvements that P-CodeSum brings to PCS tasks.

The rest of the paper is organized as follows: Section 2 provides background knowledge. Section 3 presents an empirical study on project-specific code summarization with LLM. Section 4 describes the technical details of P-CodeSum. The experimental results are analyzed in Section 5. We compare P-CodeSum against related work in Section 6. Finally, Section 7 concludes this paper.

2. Background

2.1. Code summarization with PLMs

Code summarization aims to generate human-readable, concise, and informative natural language summaries for code snippets. This can be formulated as a machine translation task where an encoder is trained to represent a given code snippet and a decoder is trained to produce a summary that accurately captures the functionality, purpose, and critical details of the code. Code summarization has been an indefensible task in helping developers understand and use the code more effectively.

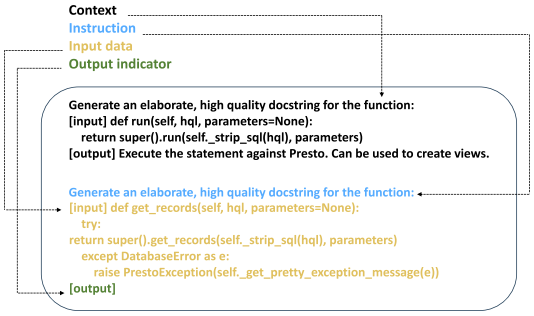

Fig. 1. An example of prompting LLM for code summarization.

Pre-trained language models such as CodeT5 have been the cornerstone technology for code summarization (Xiao et al., 2023). CodeT5 is built upon the Text-to-Text Transfer Transformer (T5), a pre-trained language model with the sequence-to-sequence architecture. CodeT5 undergoes a pre-training phase where the model is pre-trained on a large corpus of code snippets, aiming to represent the syntactic and semantic structures of both natural and programming languages. Pre-training enables the language model to capture the nuances of code syntax and semantics. The pre-trained model is fine-tuned on a parallel dataset of (code, comment) pairs for code summarization.

Pre-training models like CodeT5 capture a broad understanding of code and language, making them versatile for various code summarization tasks. The pre-training phase ensures that the model has learned from a large and diverse codebase, which can lead to better generalization and improved performance when generating code summaries.

2.2. In-context learning and prompts

In-context learning, also known as prompt engineering, is a widely used paradigm for leveraging large language models (Abdali et al., 2023). It is well recognized that fine-tuning LLMs is computationally expensive and can be impractical when the training data for a certain task is scarcely available (Cortes et al., 2015). In-context learning offers a new alternative for leveraging LLMs. In this technology, users provide specific instructions or examples, known as prompts, to guide LLMs in generating desired responses or completing specific tasks (Dong et al., 2022). This allows developers and researchers to harness the power of pre-trained language models without the need for extensive fine-tuning (Brown et al., 2020).

A prompt is a piece of text or instruction inserted into the input sample to explicitly guide the model in completing downstream tasks. It is typically short, human-generated pieces of text that serve as a directive to the model, helping it understand the context or task at hand. A typical prompt, as shown in Fig. 1, consists of instruction, context, input data, and output indicator (DAIR.AI, 2023).

In-context learning has gained impressive results in various software engineering tasks such as code generation (Li et al., 2023) and code summarization (Ahmed and Devanbu, 2022), offering a versatile and efficient way to leverage the capabilities of large language models for specific tasks without the need for extensive model modifications.

3. Empirical analysis

3.1. Study design

In this section, we describe our study methodology and experimental setup. We begin by investigating the performance of state-of-the-art LLMs on project-specific summarization, followed by exploring how to construct efficacy prompts for LLMs. In summary, we design our study methodology by addressing the following research questions:

```
"repo": "h2oai/h2o-3",
"path": "py2/h2o_ray.py",
"func_name": "model_metrics",
"original_string": "def model_metrics(self, timeoutSecs=60, **kwargs):\n '''\n
ModelMetrics list. \n '''\n result = self.do_json_request('/3/ModelMetrics.json',
cmd='get', timeout=timeoutSecs)\n h2o_sandbox.check_sandbox_for_errors()\n return
result",
"language": "python",
"code": "def model_metrics(self, timeoutSecs=60, **kwargs):\n '''\n ModelMetrics list.
\n '''\n result = self.do_json_request('/3/ModelMetrics.json', cmd='get',
timeout=timeoutSecs)\n h2o_sandbox.check_sandbox_for_errors()\n return result",
"code_tokens": ["def", "model_metrics", "(", "self", ",", "timeoutSecs", "=", "60",
",", "*", "*", "kwargs", ")", ":", "result", "=", "self", ".", "do_json_request", "(",
"'/3/ModelMetrics.json'", ",", "cmd", "=", "'get'", ",", "timeout", "=", "timeoutSecs",
")", "h2o_sandbox", ".", "check_sandbox_for_errors", "(", ")", "return", "result"],
"docstring": "ModelMetrics list.",
"docstring_tokens": ["ModelMetrics", "list", "."],
"url": "https://github.com/h2oai/h2o-
3/blob/dd62aaa1e7f680a8b16ee14bc66b0fb5195c2ad8/py2/h2o_ray.py#L592-L598",
"index": 1332,
"time": "2014-11-09 00:05:08"
```


Fig. 2. An example of data for the `h2oai/h2o-3` project.

- **RQ1. How effective are LLMs on project-specific code summarization?** We investigate the performance of ChatGPT (`text-davinci-003`) (OpenAI, 2022), a representative LLM, on the code summarization task. We compare the performance of ChatGPT with an open-source LLM, CodeLlama (Roziere et al., 2023), and two state-of-the-art PLMs, CodeBERT and CodeT5, which are fine-tuned on a general code summarization dataset extracted from CodeXGLUE (Lu et al., 2021).
- **RQ2. How to write prompts that better elicit LLMs for project-specific code summarization?** We further investigate the key factors of prompts, namely, instruction description, instruction position, and the number and selection of in-context examples. We perform experiments using the text-davinci-003 model on PCS datasets (i.e., h2oai/h2o-3, RestComm/jain-slee) and examine their influence on the performance of project-specific code summarization.

Studied models. We take ChatGPT (`text-davinci-003`) as a representative LLM and investigate its performance on project-specific code summarization. We compare the performance of ChatGPT with CodeBERT, CodeT5, and CodeLlama (13b), which stand for LLMs with encode-only, encode-decode, and decode-only architectures, respectively. All selected models have shown superior results in code summarization (Xie et al., 2022; Bhattacharya et al., 2023).

Datasets. We construct a project-specific code summarization dataset from CodeXGLUE. CodeXGLUE is a commonly used benchmark for code-intelligent tasks. The original benchmark involves 14 datasets across 10 tasks. We select six projects from the code summarization task. These projects are selected to ensure a high diversity of programming languages while keeping the volume of (code, summary) examples in each project within a reasonable range. The statistics of the train and test datasets used in this study are presented in Table 1. Each example consists of 12 fields, including the repository name, file path, function name, original string, programming language, code snippet, array of code words, summary, array of summary words, repository URL, code snippet index, and creation time. Fig. 2 illustrates a data sample in the `h2oai/h2o-3` project.

For each project-specific dataset, we randomly sample 10 examples and use them to build an in-context example pool. We also sample 30 examples for training and the remaining examples are reserved as the test set for evaluation. The numbers of samples are chosen through extensive experimentation with different sizes.

Evaluation metrics. We evaluate the performance of all models using BLEU-4 (Papineni et al., 2002) and ROUGE-L (Lin, 2004), two widely used metrics for evaluating text generation tasks. BLEU-4 calculates the similarity between a generated sequence and a reference sequence by counting the occurrences of n-grams in both sequences. ROUGE-L is a metric that considers the structural similarity at the sentence level and automatically identifies the longest common subsequence in the sequences.

Table 1
Statistics for datasets.

Language	Project	# of examples	# of in-context examples	# of test examples
Python	h2oai/h2o-3	268	10	228
Java	RestComm/jain-slee	234	10	194
Go	dgraph-io/badger	225	10	185
Javascript	disnet/contracts.js	289	10	249
PHP	Payum/Payum	277	10	237
Ruby	chef/omnibus	199	10	159

3.2. Performance of LLMs on project-specific code summarization (RQ1)

We test the performance of ChatGPT with and without in-context examples, respectively. For the former setting, we select five in-context examples from the project's in-context example pool. We construct the prompt using a sequence of k in-context examples followed by the task instruction, namely, using the following template:

$$\{\langle code\ example\rangle \mid\mid \langle instruction\rangle \mid\mid \langle summary\ example\rangle\}_0^k\ \langle code\ query\rangle \mid\mid \langle instruction\rangle$$

where k stands for the number of in-context examples. In each in-context example, <*code example*> stands for a code snippet with a certain function, while <*summary example*> is the corresponding natural language description of its functionality. The <*instruction*> within each in-context example is a constant text: "Generate an elaborate, high-quality docstring for the above function". The <*code query*> is the target code snippet that needs summarizing.

The results are provided in Table 2. As expected, ChatGPT achieves better performance than smaller PLMs (i.e., CodeBERT, CodeT5, and CodeLlama (13b)) when only five in-context examples are provided. This is probably because ChatGPT alleviates the gap between code summarization and pre-training objectives through pre-training with ultra-large code corpora and model parameters. Such extensive training enables ChatGPT to handle various coding styles and conventions in specific projects.

Compared with zero-shot code summarization (i.e., no in-context example), providing ChatGPT with five in-context examples leads to a significant improvement in performance. This indicates that in-context examples enable LLMs to better learn the code-summary mappings.

Table 3 provides an example of a code summary generated by ChatGPT. As can be seen, there are two (code, summary) examples

Table 2
Performance of various models on project-specific code summarization.

Model	# of examples	BLEU-4	ROUGE-L
CodeBERT	–	15.86	15.26
CodeT5	–	17.54	17.05
CodeLlama	–	15.03	15.48
ChatGPT	0	19.65	18.58
ChatGPT	5	**25.80**	**21.51**
p-value[a]		<0.04	<0.04

[a] p-value is calculated with pairwise 2-sample Wilcoxon Signed rank test between ChatGPT (0-shot) and other models.

Table 3
An example of code summary generated by ChatGPT.

In-context Example 1	**code:** def stop_instances(instances, region): if not instances: return ... conn.stop_instances(instances) log("Done") **docstring:** stop all the instances **given by its ids**.
In-context Example 2	**code:** def start_instances(instances, region): if not instances: return ... conn.start_instances(instances) log("Done") **docstring:** start all the instances **given by its ids**.
Code Query	def reboot_instances(instances, region): if not instances: return ... conn.reboot_instances(instances) log("Done")
Reference	reboot all the instances **given by its ids**.
Generated	restart all instances **identified by their respective ids**.

originating from the same project as where the query code originates, which exhibit the same summary style in that project. We observe that ChatGPT's summarization style does not sufficiently match the characteristics of other summaries in the same project. For example, code summaries in the project always end with the phrase "given by its ids". The reference summary for the given code snippet is "reboot all the instances given by its ids". However, the summary generated by ChatGPT ends with the phrase "identified by their respective ids".

Finding 1: LLM surpasses traditional PLMs on project-specific code summarization when provided with a small number of in-context examples. However, the generated summaries do not sufficiently match the style of the same project consistently.

3.3. Key factors to proper prompts (RQ2)

We further explore what kinds of prompts can lead to better performance on project-specific code summarization. We apply ChatGPT on two PCS datasets, h2oai/h2o-3 and RestComm/jain-slee. We vary the prompts in terms of their instructions and in-context examples while using the same prompt template as it is in RQ1.

3.3.1. Impact of instructions

We vary the descriptions and positions of instructions in the prompt and examine their impact on the final summarization.

Descriptions. We compare seven variants of instruction descriptions as provided in Table 4. These descriptions are designed according to what OpenAI suggests for summarization tasks (OpenAI, 2023) (#1-3). Besides, we also design descriptions by ourselves (#4-5) or with the assistance of ChatGPT (#6-7). The results are provided in Table 4. We observe that LLM gains the lowest score without any instruction. This affirms the efficacy that instructions bring to LLMs. We also notice the significant variations in performance when using different instructions. Among all seven variants, *"summarize the above code in the same code project"* achieves the best performance as it gives clear guidance on project-specific summarization.

Fig. 3. Performance of ChatGPT with different numbers of in-context examples.

Positions. We place instructions in three different positions in a prompt, i.e., before code, between code and summary, and after summary (The template in Section 3.2 shows the case "between code and summary"). The results are presented in Table 5. As we can see, placing the instruction *between code and summary* yields the best performance on project-specific code summarization. Compared to *before code*, it achieves an improvement of 21.65% and 21.16% on average in terms of BLEU-4 and ROUGE-L. Similarly, compared to *after summary*, it achieves 20% better scores on the two metrics. We hypothesize that putting instructions between code and summary provides LLMs with a clear sign about where the code ends and where the summary begins, which facilitates LLMs in understanding the task.

Finding 2: Instructions in prompts, particularly their descriptions, play a significant role in prompting LLMs for project-specific code summarization. Furthermore, positioning instructions between the code and summary within a prompt is an optimal choice.

3.3.2. Impact of in-context examples

We analyze the impact of in-context examples by varying their numbers and selection strategies while using the same prompt template as in RQ1.

Number of examples. We vary the number of in-context examples from 0 to 10 and inspect the performance of LLMs. As shown in Fig. 3, as the number of examples increases, the performance goes up accordingly until it exceeds five examples, where the effect becomes marginal. This is because too few demonstration examples are insufficient for LLMs to acquire the project-specific coding styles, whereas too many examples could confuse the LLM in grasping the essence of code summary styles.

Selection strategy. We keep the number of in-context examples at the optimal value of five and experiment with three strategies for in-context example selection: (1) random selection; (2) path similarity (i.e., code-path-first); and (3) code similarity (i.e., code-content-first). We calculate the path similarity based on the longest common subsequence (LCS) between two file paths. The top five examples whose file paths have the longest common subsequence against that of the query code are selected as the in-context examples. Content similarity

Table 4
Performance of PCS with different instruction descriptions.

ID	Instruction	Source	Project			
			h2oai/h2o-3		RestComm/jain-slee	
			BLEU-4	ROUGE-L	BLEU-4	ROUGE-L
0	None	–	23.92	18.63	24.40	19.27
1	Generate an elaborate, high quality docstring for the above function:	OpenAI suggestion	25.41	**23.36**	26.18	**24.27**
2	Here's what the above function is doing:	OpenAI suggestion	25.25	19.93	25.97	20.88
3	Explanation of what the code does\n#	OpenAI suggestion	24.40	19.80	25.63	20.72
4	Summarization of the above code\n#	Self-designed	26.49	21.13	26.87	21.45
5	Summarize the above code in a same code project\n#	Self-designed	**29.48**	23.33	**29.72**	23.67
6	Short, project-specific code summarization that accurately captures the essence of above code\n#	Generated by ChatGPT	26.61	21.46	27.53	22.29
7	Overview of the main functionality and purpose of the above code in a specific code project\n#	Generated by ChatGPT	24.01	19.60	24.94	20.46

Table 5
Performance of PCS with the instruction in different positions.

Position	Project			
	h2oai/h2o-3		RestComm/jain-slee	
	BLEU-4	ROUGE-L	BLEU-4	ROUGE-L
Before	20.98	19.14	21.43	20.18
Between	**25.41**	**23.36**	**26.18**	**24.27**
After	20.64	18.53	21.07	19.75

Table 6
Performance of PCS under different example selecting strategies.

Selecting strategy	Project			
	h2oai/h2o-3		RestComm/jain-slee	
	BLEU-4	ROUGE-L	BLEU-4	ROUGE-L
Random	25.41	23.36	26.18	24.27
Path first	26.33	24.31	27.09	25.15
Content first	**26.91**	**24.94**	**27.80**	**25.86**

is calculated using BLEU-4 scores between code snippets. For each query code, we select five snippets from the in-context example pool that have the highest BLEU-4 scores against the query code as the in-context examples. Table 6 shows the results. Compared to random selection, code-path-first and code-content-first lead to higher quality in-context examples. In detail, code-path-first improves the BLEU-4 and ROUGE-L scores by 3.62% and 4.07%, respectively, in the h2oai/h2o-3 project. Moreover, it achieves improvements of 3.48% on BLEU-4 and 3.63% on ROUGE-L, respectively, in the RestComm/jain-slee project. Such improvement could probably be due to the code examples from the same path, which are cohesive and usually work together with the query code to accomplish the same functionality. We notice that code-content-first achieves greater improvements than code-path-first, probably because the examples chosen by code-content-first have much similar content to that of the query code.

Finding 3: Both the number and selection strategy for the in-context examples affect the performance of LLMs in project-specific code summarization. Incorporating five in-context examples that share similar paths or content with the query code leads to the best performance.

4. Approach

Based on the empirical analysis, we propose a novel method for project-specific code summarization called P-CodeSum.

4.1. Overview

The pipeline of P-CodeSum is presented in Fig. 4. For each specific project, we construct a pool of in-context examples, i.e., (code, summary) pairs. This pool can be utilized throughout the training and prompting stages to characterize the project-specific coding features. Given a query code, our approach randomly selects K examples from the example pool and constructs a prompt candidate following a predefined template as shown in Fig. 5. Based on the empirical study findings outlined in Section 3.3.2, we set $K = 5$ in our experiments.

Next, a neural prompt selector is designed to discriminate whether the candidate prompt can potentially lead to better summarization (Section 4.2). The prompt that is selected by the selector will be fed into an LLM to generate the code summary. Otherwise, a new iteration will begin to re-select other in-context examples until the maximum number of attempts has been reached. We empirically set the maximal number of attempts to 25 through extensive trials and evaluations.

4.2. Prompt selector

The prompt selector aims to score examples in the pool of project-specific (code, summary) pairs and then select the optimal in-context examples that could elicit LLMs to generate high-quality summaries when acting as prompts. This can be formulated as a binary classification task: given a code query x and a set of in-context examples e, the selector aims to predict a categorical label $c \in$ {"good", "bad"} indicating whether e is useful for the LLM.

$$d(c|x, e) = g_\phi(f_\theta(x), f_\theta(e)), \tag{1}$$

where f_θ is a function to encode the input code into a d-dimensional vector and g_ϕ denotes the classifier.

f_θ can be implemented using a pre-trained language model that encodes source code into vectors. In our approach, we select CodeBERT as the code representation learner. Directly fine-tuning f_θ and g_ϕ from scratch could cause overfitting since the project-specific data for training is scarce. To migrate this problem, we propose a parameter-efficient tuning method for training the selector.

4.2.1. Architecture

Fig. 6 illustrates the architecture of the prompt selector. Inspired by prompt tuning (Liu et al., 2023), we optimize CodeBERT by merely adjusting its input sequence. More specifically, we insert a number of trainable prompt tokens into the input sequence of CodeBERT, which casts the project-specific code summarization task to the same form as the pertaining objective. During training, we only adjust the embedding parameters corresponding to the prompt tokens, while keeping most of the prior knowledge learned during pre-training. This mitigates the threat of overfitting with small project-specific data and reduces the computation cost of PLMs significantly. Hence, it allows few-shot learning for pre-trained models with limited labeled samples. More details about the training process will be discussed in the following sections.

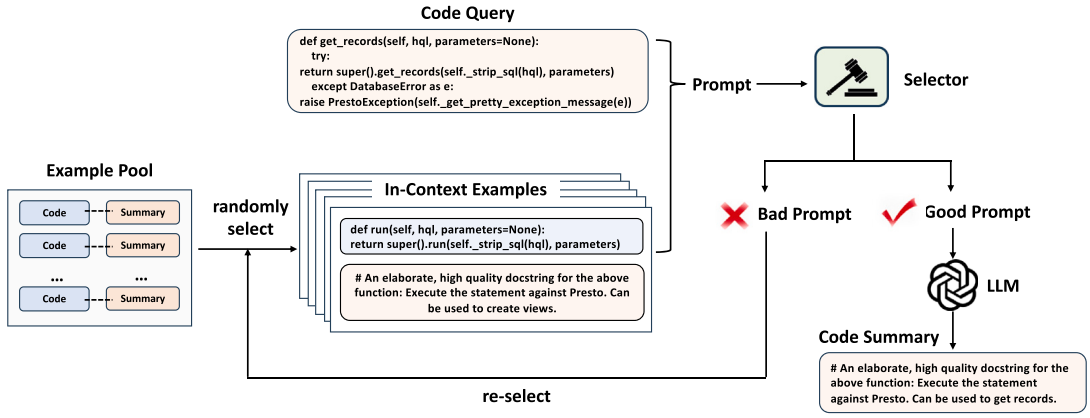

Fig. 4. Overview of P-CodeSum.

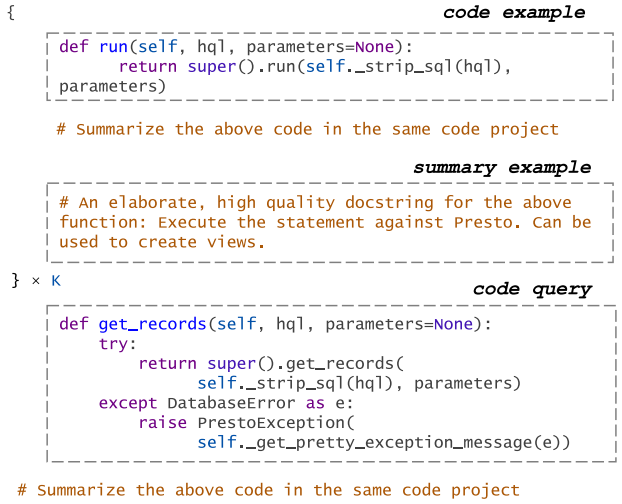

Fig. 5. An example of LLM prompt for project-specific code summarization.

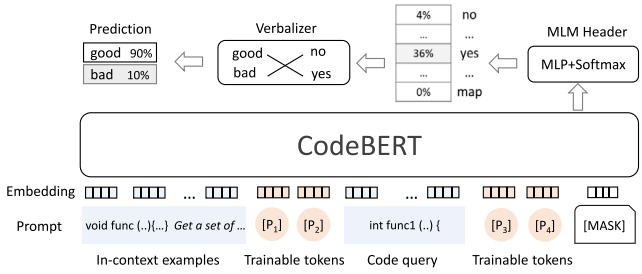

Fig. 6. The architecture of neural selector.

4.2.2. Creating training data

Training the project-specific prompt selector necessitates the collection of positive and negative data samples. Recognizing the expensive time and effort associated with manual annotation of training data, we employ an LLM to automatically craft high-quality training data. More specifically, we select candidate examples from the project-specific example pool (mentioned in the beginning of Section 4.1) and use them to prompt the LLM to generate code summaries. The generated code summaries are categorized into positive and negative groups based on their BLEU-4 and ROUGE-L scores against the ground-truth summary selected from the pool.

The generation consists of three steps, as illustrated in Fig. 7:

(1) Build an in-context example pool. For a specific project, especially the one that is in progress, we first build its in-context example pool by selecting 10 (code, summary) pairs from its historical code summary data.

(2) Generate training samples. We leverage LLMs to generate candidate training samples from the in-context example pool. Besides the initial 10 (code, summary) pairs in the example pool, we select another 30 (x, y) pairs from the project's historical data as the seed of training samples, where y is taken as the summary reference of the code query x. Next, we select 5 (code, summary) examples from the example pool constructed in step 1 as in-context examples for x, and construct the corresponding prompt p following our prompt template. Then, the LLM takes p as input and generates a summary y'. For each code query x from the 30 seeds, we select 25 different sets of in-context examples in turn and generate 25 corresponding summaries.

(3) Score training samples. We measure the quality of examples using the BLEU-4 and ROUGE-L scores between the generated summary y' and the ground-truth summary y. We take the average BLEU-4 and ROUGE-L scores of the 25 summaries as the thresholds to judge whether a set of in-context examples are useful for the code query x. Examples that gain higher scores than the threshold will be selected to prompt the LLM.

Finally, we obtain a dataset of 750 samples for building the project-specific selector, with 500 and 250 for training and testing, respectively. Each training sample consists of three fields, including a code query, five in-context examples, and a label indicating whether the sample is useful or not.

4.2.3. Training

We train the prompt selector using the collected project-specific dataset. The model takes the in-context examples and the code query as input and predicts whether the in-context examples are useful for LLMs to generate high-quality summaries. We cast example selection (Eq. (1)) to a masked language modeling (MLM) task. Given a code query x and the candidate examples e, we construct an input sequence for CodeBERT by following our defined prompt template:

$$\tilde{x} = [CLS]; x; [P_{1:i}]; e; [P_{i+1:m}]; [MASK]. \quad (2)$$

where ; denotes the concatenation operation; "[CLS]" is a placeholder for the PLM to predict the classification results. "[MASK]" is a placeholder where the pre-trained model generates the classification result c. $P = [P_1, \ldots, P_m]$ denotes the sequence of prompt tokens that we inject into the input for prompt tuning.

The constructed sequence $\tilde{x}$ is taken as input to the PLM, which yields the hidden states

$$\mathbf{h}_1, \ldots, \mathbf{h}_{|\tilde{x}|} = f_\theta(\tilde{x}) \quad (3)$$

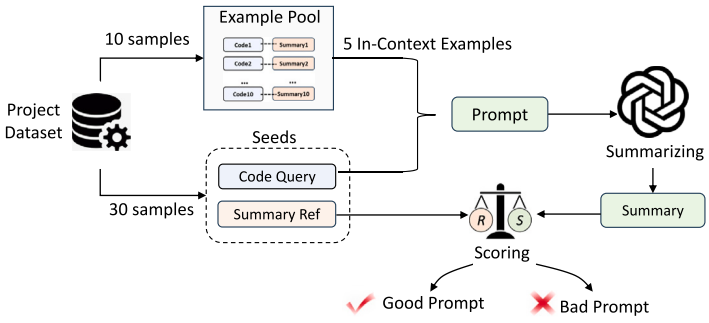

Fig. 7. Generating training data.

for all tokens. The hidden state corresponding to the masked token, namely, $\mathbf{h}_{\_1}$, is fed into an MLM header g_ϕ, which predicts a token indicating the prediction:

$$\hat{\mathbf{c}} = \mathrm{softmax}(g_\phi(\mathbf{h}_{\_1})). \tag{4}$$

The MLM header g_ϕ is a fully connected neural network parameterized by ϕ that is optimized to minimize the cross-entropy loss:

$$L_{MLM}(\phi|\mathbf{c}, \hat{\mathbf{c}}) = -\sum_{i=1}^{|V|} c_i \log(\hat{c}_i), \tag{5}$$

where $\mathbf{c}$ denotes the ground-truth label and $|V|$ is the vocabulary size.

Finally, a verbalizer is employed to cast the predicted word into "good" and "bad" labels. Let $\mathcal{V}$ be the vocabulary of the PLM. The verbalizer is defined as a function v: $\mathcal{C} \rightarrow$ {"good", "bad"} that maps each candidate word in the vocabulary to a classification label. The choice of candidate words is arbitrary, as long as they are sufficiently different. The model will be trained to map candidate words to true predictions.

4.2.4. Prompt simplification

The selected in-context examples cause the prompts to exceed the maximum input length of LLMs. Previous studies (Rodeghero et al., 2014; Zhang et al., 2022) show that developers may skip some structural information such as `For` and `If` conditions while paying special attention to literal cues such as function names, method invocations, and variable names, which exhibit the intention of the code. Inspired by this observation, we propose to simplify code snippets in the prompt when we train the prompt selector. We simplify all code snippets in the prompt by preserving the function signatures, API sequences, and variable names extracted by tree-sitter.[1]

5. Evaluation

To evaluate the effectiveness of P-CodeSum in project-specific code summarization, we conduct experiments by addressing the following research questions.

- **RQ3: How effective is P-CodeSum in project-specific code summarization?** We evaluate the performance of P-CodeSum on six PCS datasets and compare it with baseline models.
- **RQ4: How effective is the prompt selector in selecting in-context examples?** We conduct an ablation study to analyze the effect of the neural prompt selector.
- **RQ5: What is the effect of code simplification on training the prompt selector?** P-CodeSum realizes three code simplification techniques. We conduct an ablation study to identify the effect of each one.
- **RQ6: How do different hyperparameters impact the performance of our approach?** We evaluate the performance of our approach under different hyperparameters. Our goal is to find hyperparameters that lead to optimal performance.

These research questions aim to provide a comprehensive understanding of P-CodeSum's effectiveness, the role of the prompt selector, the influence of code simplification techniques, and the impact of hyperparameters.

5.1. Metrics

We measure the performance of code summarization using BLEU-4 and ROUGE-L. We evaluate the performance of the prompt selector using accuracy (ACC). ACC refers to the ratio of correctly classified points to the total number of predictions.

5.2. Baselines

We compare P-CodeSum against two categories of code summarization techniques, namely, fine-tuned code PLMs and LLMs with in-context learning. For fine-tuned code PLMs, we select CodeBERT and CodeT5, representing encode-only and encode-decode models, respectively. In the realm of LLMs, we choose ChatGPT and Codex. We employ various strategies for selecting in-context examples, including random selection, prompt retrieval (Liu et al., 2022; Ahmed et al., 2023), and prompt learning (Shrivastava et al., 2023). All the baselines demonstrate impressive performance on code summarization tasks (Xie et al., 2022).

(1) **CodeBERT**: a code summarization approach based on CodeBERT. This approach employs CodeBERT as its encoder while training a Transformer decoder from scratch. We use the same tokenizer and token embeddings as in CodeBERT. We fine-tune this model on a general code summarization dataset extracted from CodeXGLUE.

(2) **CodeT5**: a code summarization approach based on CodeT5. CodeT5 employs a sequence-to-sequence architecture, making it a well-suited choice for code summarization tasks. The fine-tuning process for CodeT5 aligns with the hyperparameters and dataset used during our initial prompt learning stage in Section 3.

(3) **Codex+Random** (Chen et al., 2021)+**Random**: an LLM-based code summarization approach with randomly-selection strategy. This approach randomly selects in-context examples from the same project. Then Codex is employed to generate code summaries under the guidance of the examples.

(4) **Codex+ASAP** (Ahmed et al., 2023): an CodeLLM-based code summarization approach with retrieved prompts. ASAP constructs prompts by extracting code attributes such as function name, parameter name, and data flow from the original code snippet using a tree-sitter. The prompt is fed into Codex to generate the code summary.

(5) **ChatGPT+KATE** (Liu et al., 2022): an LLM-based code summarization approach with retrieved prompts. KATE selects the top-k in-context examples by using a kNN (k-nearest neighbors) strategy

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

Table 7
The performance of PCS with different models.

Approaches	BLEU-4	ROUGE-L
CodeBERT	15.86	15.63
CodeT5	17.54	17.25
Codex+Random	19.47	18.63
Codex+ASAP	19.71	18.74
ChatGPT+KATE	27.12	23.17
ChatGPT+RLPG	30.19	25.05
DeepSeek-Coder+P-CodeSum	22.12	22.09
- w/o selector	20.02	20.36
ChatGPT+P-CodeSum	**31.96**	**26.19**
- w/o selector	29.60	24.43
p-value[a]	<0.02	<0.02

[a] p-value is calculated with pairwise 2-sample Wilcoxon Signed rank test between ChatGPT+P-CodeSum and other models.

and then employs ChatGPT to generate code summaries with these examples.

(6) **ChatGPT+RLPG** (Shrivastava et al., 2023): an LLM-based code summarization approach with prompt learning. RLPG takes CodeBERT as the backbone model and trains a prompt retriever. The prompt retriever selects several suitable prompts for each code snippet from a project-level prompt candidate pool and inputs them into the ChatGPT model to generate the code summary.

5.3. Implementation details

We implement our prompt selector based on the GitHub repository of CodeBERT[2] using the default hyperparameter settings. The model is trained with the Adam optimizer (Kingma and Ba, 2014) with a learning rate of $1e{-}5$. The batch size and the number of epochs are set to 5 and 100, respectively. We train and evaluate all models on a server with 2 NVIDIA GeForce RTX 4090 GPUs. We perform code summarization on text-davinci-003 (OpenAI, 2022) by leveraging the official API with default settings. For each code query, we select the rank-1 result in the response list as the generated summary.

5.4. Effectiveness of P-CodeSum (RQ3)

Table 7 shows the performance of baseline approaches on a variety of projects. We also evaluate P-CodeSum on two LLMs: DeepSeek-Coder (6.7b) (Guo et al., 2024) and ChatGPT (*text-davinci-003*). Overall, our approach (ChatGPT+P-CodeSum) significantly outperforms all baselines, with 101.51% and 82.21% greater BLEU scores than fine-tuning CodeBLEU and CodeT5, respectively. Our approach also improves CodeBERT by 67.56% and CodeT5 by 51.83% in terms of ROUGE-L.

It can be observed that all LLM-based approaches, such as KATE, ASAP, and RLPG, outperform the fine-tuned pre-trained models. This is because LLM models are guided by a small number of in-context examples specific to the project, which helps them better understand and capture the code structure, semantics, and style of the specific project. In contrast, PLM is fine-tuned on large-scale generic code summary datasets, which may not fully understand and adapt to the characteristics and requirements of specific projects, thus tending to learn generic code syntax and structure.

Our approach demonstrates considerable strength over SOTA LLM-based approaches with different prompt construction strategies. For example, compared to KATE, which selects in-context examples based on code similarity, Our approach achieves improvements of 17.85% in BLEU-4 and 13.03% in ROUGE-L, respectively. The superior performance is attributed to the well-designed prompt template and prompt selector model, which optimize the prompt from the perspectives of instruction and in-context examples, allowing the generated summaries to better align with the style characteristics of the same project.

Our approach also outperforms RLPG, the strongest baseline, which also utilizes the `text-davinci-003` model and prompt learning technique. Similar to our approach, RLPG also employs a prompt retriever that is trained to select project-level in-context examples. However, its prompt retriever is a multi-label binary classifier; furthermore, it retrieves prompts from a set of predefined prompt proposals. These limited prompt proposals restrict its adaptation to specific projects. By contrast, our approach trains a neural prompt selector using a large PLM for a specific project. It also constructs a project codebase from which the selector can identify the best in-context examples.

Notably, Random performs worse than KATE, ASAP, and RLPG, probably due to its random example selecting strategy. This indicates that the prompt construction strategy is a critical part of LLMs to generate high-quality summaries.

We further assess the efficacy of P-CodeSum on other backbone LLMs. The result shows that DeepSeek-Coder consistently works better when equipped with P-CodeSum. We notice that ChatGPT with P-CodeSum exhibits superior performance than DeepSeek-Coder with P-CodeSum, presumably owing to the strong capability of the ChatGPT in generating NL texts (Liu et al., 2024).

Answer to RQ3: Compared to fine-tuning PLMs and state-of-the-art LLM-based approaches, our approach is superior by a remarkable margin due to its well-designed prompt template and prompt selector.

5.5. Effect of neural prompt selector (RQ4)

To assess the impact of the prompt selector, we compare P-CodeSum with a variant that excludes the prompt selector, i.e., randomly selects in-context examples from the example pool. The results are shown in Table 7. We can observe that by using the prompt selector, both ChatGPT and DeepSeek-Coder attain a soar improvement in performance, with an increase in BLEU-4 by 9.23% and Rouge-L by 7.85% in average. This indicates that the prompt selector is an essential component in P-CodeSum to generating high-quality project-specific code summaries. Additionally, it can be seen that P-CodeSum without the prompt selector can still outperform Random in project-specific code summarization due to our well-designed prompt template.

Answer to RQ4: The neural prompt selector is effective in selecting in-context examples for prompting LLMs to generate higher-quality code summaries.

5.6. Effect of code simplification (RQ5)

To study the effect of the code simplification that we propose for training the prompt selector, we experiment with three different code simplification strategies (as illustrated in Table 8). We evaluate their effectiveness in training the prompt selector:

(1) Original Code: Train the selector model using the input of the original code without any simplification.

(2) Function signatures & API sequences: Train the selector model using only function signatures and API sequences extracted from the original code.

(3) Function signatures, API sequences & Var names: Train the selector model using only function signatures, API sequences, and variable names extracted from the original code.

Table 9 presents the accuracy and relative improvement rate of different simplification strategies. It can be observed that retaining only the function signatures, API sequences, and variable names achieves the best performance in training the selector model. Retaining the function signatures and API sequences also outperforms the strategy of using the original code directly.

[2] https://github.com/microsoft/CodeBERT.

Table 8
Examples of code simplification.

Original code	def dedent(ind, text):
	text2 = textwrap.dedent(text)
	if ind == 0:
	return text2
	indent_str = “ ” * ind
	return “ ”.join(indent_str + line for line in text2.split(“\n”)

Func signatures & API sequences	dedent(ind, text) textwrap.dedent “\n”.join text2.split
Func signatures, API sequences & Var names	dedent(ind, text) textwrap.dedent “\n”.join text2.split dedent ind text text2 textwrap dedent text ind text2 indent_str ind join indent_str line line text2 split

Table 9
The performance of code summarization with different code simplifying methods.

Method	ACC	Improvement
Original code	0.537	–
Func signatures & API seq.	0.543	1.12%
Func signatures, API seq. & Var names	**0.577**	**7.45%**

Table 10
Performance of P-CodeSum under different number of in-context examples.

# of examples	ACC
3	0.526
5	**0.577**
8	0.512
10	0.500

The improvement could be attributed to the restriction of input length by CodeBERT, where the original code may exceed the maximum length, with partial code being discarded by CodeBERT. This hinders the model from learning the relationship between the test code and the in-context examples. By utilizing the latter two code simplification methods, the input code does not exceed the length limit of CodeBERT. Therefore, the selector can fully extract the information from the test code and the in-context examples. Additionally, the third preprocessing method, which retains variable names, provides richer code semantic information for the selector compared to the second one.

Answer to RQ5: The code simplification method that retains only the function signatures, API sequences, and variable names performs the best for training the prompt selector.

5.7. Ablation study (RQ6)

To gain more insights on the results, we conduct an ablation study of P-CodeSum under different parameter settings. We inspect two hyperparameters: the number of in-context learning examples and the data size for training the prompt selector. Our goal is to find the optimal hyperparameter settings.

Number of in-context learning examples. Table 10 presents the results of P-CodeSum under different numbers of in-context learning examples. It can be seen that the prompt selector achieves the best performance when the number of in-context learning examples is set to 5. The performance gets slightly lower with 3 or 8 examples. Notably, there is no evident improvement when the number is set to 10.

The reason behind these results is apparent: a smaller number of in-context learning examples (e.g., ≤5) is insufficient for the LLM to acquire knowledge about the format and language style of the summaries (Cai et al., 2023). On the other hand, the marginal effect of examples becomes slight when the number exceeds 5.

Training size for the prompt selector. We test P-CodeSum on two different data sizes, 500 and 1000. Table 11 presents the results. We can observe that when the training data size is set to 500, the selector achieves an accuracy of 57.7%. However, when the data size reaches 1000, the accuracy decreases to 53.1%.

Table 11
The performance under different size of training data.

Size of training set	ACC
500	**0.577**
1000	0.531

There could be three reasons for this observation:

(1) Data quality. Increasing the data could introduce more noise, making it difficult for the model to accurately learn the correct method of generating summaries.

(2) Overfitting. As the training data increases, the capacity of the model may also increase, increasing the risk of overfitting. If the model overfits the training data, introducing more data can make it difficult for the model to generalize to new samples, resulting in decreased performance.

(3) Hyperparameter tuning. After increasing the data volume, it may be necessary to re-evaluate and adjust the hyperparameters such as learning rate, batch size, and number of layers.

Answer to RQ6: The performance of P-CodeSum is influenced by the number of in-context examples and the size of the training data for the prompt selector. The best results can be obtained when the number of in-context examples is set to 5. A training dataset size of 500 is sufficient for the selector model to learn the strategy of selecting in-context examples, while increasing the data size results in a decline in performance.

6. Related work

6.1. Code summarization

Code summarization has been a well-studied research area in software (Haiduc et al., 2010; Zhu and Pan, 2019; Lin et al., 2021). A common code summarization method is treating source code as plain text and applying neural machine translation (NMT) techniques (Zhang et al., 2020).

Inspired by NMT, researchers have adopted a neural encoder–decoder framework for code summarization tasks. For example, Hu et al. (2018) proposed a novel encoder–decoder architecture approach that successfully uses API knowledge learned in a different but related task to code summarization. Wei et al. (2019) treated code summarization and code generation as a dual task, training them together using an encoder–decoder architecture. Ahmad et al. (2020) addressed the issue of long-range dependency in code summarization by employing a Transformer model equipped with self-attention mechanisms.

Recently, inspired by the great success of LLMs in NLP, a boom has arisen in LLMs on code summarization. He et al. (2022) presented a new pre-trained language model optimized for abstractive text summarization, Z-Code++, which creates a new SOTA on 9 out of 13 text summarization tasks across five languages. Arakelyan et al. (2023) conducted a systematic evaluation of two large language models – CodeT5 and Codex – on code generation and code summarization tasks to understand the models' performance. Sun et al. (2023) evaluated

ChatGPT on a widely-used Python dataset and compared it with SOTA code summarization models.

One common challenge for project-specific code summarization is the insufficiency of training data. To mitigate this issue, researchers have proposed various solutions. For example, Xie et al. (2022) introduced meta-transfer learning, adding a lightweight fine-tuning mechanism to enable rapid adaptation to few-shot scenarios in code summarization. Ahmed and Devanbu (2022) experimented with Codex, finding that LLM, when combined with contextual learning techniques, outperformed existing models in project-specific code summarization. However, they noted that LLMs rely on manual prompts, which can be time-consuming and inconsistent. To improve project-specific code summarization in LLM, ASAP (Ahmed et al., 2023) suggests using automatically extracted code semantics like parameter names, local variables, method invocations, and data flow in prompts. Despite these efforts, ASAP's overall impact on project-specific code summarization remains limited.

Overall, differing from these existing methods, P-CodeSum makes full use of LLM's sensitivity to in-context examples. It selects appropriate (code, summary) pairs from the in-context example pool for each code snippet to build high-quality prompts and achieves the generation of highly accurate code summaries that conform to the project style in a few-shot scenario.

6.2. Prompt generation

A prompt is a text instruction inserted into the LLM's input, serving as explicit guidance to direct LLMs in downstream tasks (Brown et al., 2020). Existing prompt generation technologies can be subdivided into the following four categories:

(1) Manually-written prompts. Manually written prompts prove to be more useful for template-based prompts than uncomplicated prompts (Shin et al., 2020). Shi et al. (2023) studied the technology of manually constructing prompts on the code-davinci-002 model.

(2) Model generation. Customizing prompts for each text input by using language models can make up for the shortcomings of manually constructed prompts. Automatic Prompt Engineer (APE) (Zhou et al., 2023) uses language models to automatically generate prompts. Experiments show that the prompts automatically generated by the APE framework have reached the human level, regardless of whether there are zero or few samples.

(3) Retrieval-based prompt. The basic idea of this method is to retrieve text fragments relevant to a specific task and then extract key information from them as part of the prompt. Liu et al. proposed Knn-Augmented in-conText Example selection (KATE) (Liu et al., 2022), a non-parametric selection approach that retrieves in-context examples according to their semantic similarity to the test samples. Automated Semantic Augmentation for Prompting (ASAP) (Ahmed et al., 2023) uses the tree-sitter tool to automatically extract key information from the source code as a component of the prompt in the code summarization task.

(4) Prompt learning (Qiao et al., 2023; Li et al., 2024). This method constructs a supervised learning model to generate prompts and update prompts according to their generation and related basic facts. B. Lester et al. proposed Soft Prompt Tuning (Lester et al., 2021), which achieves results comparable to traditional methods of fine-tuning the entire PLM while only training less than one-thousandth of the number of parameters. Prefix-tuning (Li and Liang, 2021) believes that the core concept of prompt learning is to find appropriate background knowledge guiding PLM to adapt its knowledge to specific downstream tasks. Reinforcement Learning-based Prompt Generation (RLPG) (Shrivastava et al., 2023) is a framework for automatic prompt generation and constructing a project-level prompt candidate pool.

Inspired by prompt learning, P-CodeSum leverages LLMs to generate training data for each specific project to train a neural prompt selector based on CodeBERT. The selector can score in-context examples from the example pool and select the most effective ones to construct a high-quality prompt.

7. Conclusion

In this paper, we introduce P-CodeSum, an in-context learning-based method for project-specific code summarization. In order to better harness the contextual understanding capabilities of LLM, we propose a prompt selector, which assesses the efficacy of in-context examples and constructs high-quality prompts for code snippets. Experiments on six PCS datasets extracted from CodeXGLUE show that P-CodeSum outperforms the existing SOTA, generating high-quality code summarization for specific projects in few-shot scenarios. In the future, we will design a universal dataset for training the prompt selector.

CRediT authorship contribution statement

Shangbo Yun: Writing – review & editing, Writing – original draft, Methodology, Investigation, Data curation. **Shuhuai Lin:** Writing – original draft, Methodology, Investigation, Data curation. **Xiaodong Gu:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Conceptualization. **Beijun Shen:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Beijun Shen reports financial support was provided by National Key R&D Program of China and National Natural Science Foundation of China. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This research is supported by National Key R&D Program of China (Grant No. 2023YFB4503802) and National Natural Science Foundation of China (Grant No. 62102244, 62232003).

References

Abdali, S., Parikh, A., Lim, S., Kiciman, E., 2023. Extracting self-consistent causal insights from users feedback with LLMs and in-context learning. arXiv preprint arXiv:2312.06820.

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.-W., 2020. A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653.

Ahmed, T., Devanbu, P.T., 2022. Few-shot training LLMs for project-specific code-summarization. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. ACM, pp. 177:1–177:5.

Ahmed, T., Pai, K.S., Devanbu, P., Barr, E.T., 2023. Improving few-shot prompts with relevant static analysis products. CoRR abs/2304.06815.

Arakelyan, S., Das, R.J., Mao, Y., Ren, X., 2023. Exploring distributional shifts in large language models for code analysis. CoRR abs/2303.09128.

Bhattacharya, P., Chakraborty, M., Palepu, K.N.S.N., Pandey, V., Dindorkar, I., Rajpurohit, R., Gupta, R., 2023. Exploring large language models for code explanation. In: IRSE.

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

Cai, C., Wang, Q., Liang, B., Qin, B., Yang, M., Wong, K.-F., Xu, R., 2023. In-context learning for few-shot multimodal named entity recognition. In: The 2023 Conference on Empirical Methods in Natural Language Processing.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. CoRR abs/2107.03374.

Cortes, C., Lawarence, N., Lee, D., Sugiyama, M., Garnett, R., 2015. Advances in neural information processing systems 28. In: Proceedings of the 29th Annual Conference on Neural Information Processing Systems.

DAIR.AI, 2023. Elements of a prompt, URL https://www.promptingguide.ai/introduction/elements.

Dong, Q., Li, L., Dai, D., Zheng, C., Wu, Z., Chang, B., Sun, X., Xu, J., Sui, Z., 2022. A survey for in-context learning. arXiv preprint arXiv:2301.00234.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics. EMNLP, pp. 1536–1547.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y.K., Luo, F., Xiong, Y., Liang, W., 2024. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. arXiv:2401.14196.

Haiduc, S., Aponte, J., Marcus, A., 2010. Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. pp. 223–226.

He, P., Peng, B., Lu, L., Wang, S., Mei, J., Liu, Y., Xu, R., Awadalla, H.H., Shi, Y., Zhu, C., et al., 2022. Z-code++: A pre-trained language model optimized for abstractive summarization. CoRR abs/2208.09770.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018. Summarizing source code with transferred API knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 2269–2275.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. CoRR abs/1412.6980.

Lester, B., Al-Rfou, R., Constant, N., 2021. The power of scale for parameter-efficient prompt tuning. CoRR abs/2104.08691.

Li, J., Li, G., Tao, C., Zhang, H., Liu, F., Jin, Z., 2023. Large language model-aware in-context learning for code generation. arXiv preprint arXiv:2310.09748.

Li, X.L., Liang, P., 2021. Prefix-tuning: Optimizing continuous prompts for generation. CoRR abs/2101.00190.

Li, X., Yuan, S., Gu, X., Chen, Y., Shen, B., 2024. Few-shot code translation via task-adapted prompt learning. J. Syst. Softw. 212, 112002.

Lin, C.-Y., 2004. ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. Association for Computational Linguistics, Barcelona, Spain, pp. 74–81, URL https://aclanthology.org/W04-1013.

Lin, C., Ouyang, Z., Zhuang, J., Chen, J., Li, H., Wu, R., 2021. Improving code summarization with block-wise abstract syntax tree splitting. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. ICPC, IEEE, pp. 184–195.

Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., Zhang, L., 2024. Exploring and evaluating hallucinations in LLM-powered code generation. arXiv:2404.00971.

Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., Chen, W., 2022. What makes good in-context examples for GPT-3? In: Agirre, E., Apidianaki, M., Vulic, I. (Eds.), Proceedings of Deep Learning Inside Out: The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeeLIO@ACL 2022, Dublin, Ireland and Online, May 27, 2022. Association for Computational Linguistics, pp. 100–114.

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., Tang, J., 2023. GPT understands, too. CoRR abs/2103.10385.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., GONG, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., LIU, S., 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks.

OpenAI, 2022. text-davinci-003. URL https://api.openai.com/v1/completions.

OpenAI, 2023. Examples. URL https://platform.openai.com/examples.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. BLEU: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. ACL, pp. 311–318.

Qiao, S., Ou, Y., Zhang, N., Chen, X., Yao, Y., Deng, S., Tan, C., Huang, F., Chen, H., 2023. Reasoning with language model prompting: A survey. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, ACL 2023, Toronto, Canada, July 9-14, 2023. Association for Computational Linguistics, pp. 5368–5393.

Rodeghero, P., McMillan, C., McBurney, P.W., Bosch, N., D'Mello, S., 2014. Improving automated source code summarization via an eye-tracking study of programmers. In: Proceedings of the 36th International Conference on Software Engineering. pp. 390–401.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al., 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E.H., Schärli, N., Zhou, D., 2023. Large language models can be easily distracted by irrelevant context. CoRR abs/2302.00093.

Shin, T., Razeghi, Y., Logan IV, R.L., Wallace, E., Singh, S., 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. CoRR abs/2010.15980.

Shrivastava, D., Larochelle, H., Tarlow, D., 2023. Repository-level prompt generation for large language models of code. In: International Conference on Machine Learning. PMLR, pp. 31693–31715.

Sun, W., Fang, C., You, Y., Miao, Y., Liu, Y., Li, Y., Deng, G., Huang, S., Chen, Y., Zhang, Q., et al., 2023. Automatic code summarization via ChatGPT: How far are we? CoRR abs/2305.12865.

Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z., 2019. Code generation as a dual task of code summarization. Adv. Neural Inf. Process. Syst. 32.

Wu, H., Zhao, H., Zhang, M., 2020. Code summarization with structure-induced transformer. CoRR abs/2012.14710.

Xiao, Y., Zuo, X., Xue, L., Wang, K., Dong, J.S., Beschastnikh, I., 2023. Empirical study on transformer-based techniques for software engineering. arXiv preprint arXiv:2310.00399.

Xie, R., Hu, T., Ye, W., Zhang, S., 2022. Low-resources project-specific code summarization. In: 37th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1–12.

Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X., 2020. Retrieval-based neural source code summarization. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE, pp. 1385–1397.

Zhang, Z., Zhang, H., Shen, B., Gu, X., 2022. Diet code is healthy: simplifying programs for pre-trained models of code. In: ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022. ACM, pp. 1073–1084.

Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J., 2023. Large language models are human-level prompt engineers. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net.

Zhu, Y., Pan, M., 2019. Automatic code summarization: A systematic literature review. CoRR abs/1909.04352.

Shangbo Yun received the bachelor's degree in Information Security from Shanghai Jiao Tong University, China, in 2023. He is currently a master's graduate student in the School of Software at Shanghai Jiao Tong University. His research interests include intelligent software engineering and large language models.

Shuhuai Lin received the bachelor's degree in School of Software from Shanghai Jiao Tong University, China, in 2023. She is currently a master's graduate student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. Her research interests include intelligent software engineering and large language models.

Xiaodong Gu is currently an associate professor in the School of Software, Shanghai Jiao Tong University. He received a Ph.D. degree from the Department of Computer Science and Engineering from The Hong Kong University of Science and Technology, in 2017. His research interests lie in the broad areas of software engineering and deep learning, including large language models (LLMs) for code, code generation, code search, and code translation.

Beijun Shen is an associate professor in the School of Software, Shanghai Jiao Tong University. She received a Ph.D. degree from Institute of Software, Chinese Academy of Sciences, in 2001. Her research interests include LLMs for code and code intelligence. She has published over 200 papers in top conferences and journals in the field of software engineering, and received ICSE Distinguished Paper Award in 2023.