Towards Retrieval-Based Neural Code Summarization: A Meta-Learning Approach

Ziyi Zhou, Huiqun Yu, *Senior Member, IEEE*, Guisheng Fan, Zijie Huang, and Kang Yang

***Abstract*—Code summarization aims to generate code summaries automatically, and has attracted a lot of research interest lately. Recent approaches to it commonly adopt neural machine translation techniques, which train a Seq2Seq model on a large corpus and assume it could work on various new code snippets. However, codes are highly varied in practice due to different domains, businesses or programming styles. Therefore, it is challenging to learn such a variety of patterns into a single model. In this paper, we propose a brand-new framework for code summarization based on meta-learning and code retrieval, named MLCS to tackle this issue. In this framework, the summarization of each target code is formalized as a few-shot learning task, where its similar examples are used as training data and the testing example is itself. We retrieve examples similar to the target code in a rank-and-filter manner. Given a neural code summarizer, we optimize it into a meta-learner via Model-Agnostic Meta-Learning (MAML). During inference, the meta-learner first adapts to the retrieved examples and yields an exclusive model for the target code, and then generates its summary. Extensive experiments on real-world datasets show: (1) Utilizing MLCS, a standard Seq2Seq model is able to outperform previous state-of-the-art approaches, including both neural models and retrieval-based neural models; (2) MLCS can flexibly adapt to existing neural code summarizers without modifying their architecture, and could significantly improve their performance with the relative gain of up to 112.7% on BLEU-4, 23.2% on ROUGE-L, and 31.5% on METEOR; (3) Compared to the existing retrieval-based neural approaches, MLCS can better leverage multiple similar examples, and shows better generalization ability on different retrievers, unseen retrieval corpus and low-frequency words.**

***Index Terms*—Code summarization, deep learning, information retrieval, meta-learning.**

I. Introduction

CODE summaries are high-level natural language descriptions of source code in the forms of comments, documentation, etc. Since developers always spend more than half of their time on program comprehension activities [1], high-quality code summaries are essential to help them understand programs more easily during software development and maintenance. Concise code summaries are also proved useful for various applications such as code search [2], [3], [4], [5] and code categorization [6]. However, manually commenting or documenting source code is time-consuming and labour-intensive [7]. In fact, code summaries are often absent, out dated or even mismatched due to increasing scale and complexity of software as well as limited energy of developers. Code summarization aims to generate code summaries automatically, and has become a rapidly expanding research area in the past decade.

Early approaches to code summarization are mainly based on information retrieval (IR) [2], [8], [9], [10] and templates [11], [12], [13], [14]. The retrieval-based approaches use IR algorithms such as VSM and LSI [2], [8] to mine code text within a database, and find out keywords in the code to construct summaries, while the template-based ones define heuristic rules to extract key information to generate summaries. Due to common program reuse, the comments of similar codes could also be used for summarization via clone detection techniques [15], [16]. Inspired by the advance of deep learning in natural language processing, data-driven models based on neural machine translation (NMT) have proliferated in recent few years and achieved the state-of-the-art performance on this task [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27]. These models follow the encoder-decoder framework (also known as Seq2Seq) where the encoder works as a component for program comprehension and the decoder uses the encoding results for summary generation. Currently, most researches in neural code summarization focus on extracting better code representation via various input forms and advanced neural networks. A common way is to treat source code as token sequence, and then apply recurrent neural network (RNN) [20], [21], [22], [23] or convolutional neural network (CNN) [18]. For structural inputs like abstract syntax tree (AST) and control flow graph (CFG), recursive neural network (Tree-RNN) [20] and graph neural network (GNN) [25], [26], [27] can be used.

Commonly, the learning-based and retrieval-based approaches require a large corpus of codes and comments to build a code summarization model, and assume such a large corpus could enable the model to generalize to diverse scenarios. In this way, each new code snippet is treated similarly using a one-size-fits-all model. However, codes and their summaries are highly varied in practice due to different domains, businesses or programming styles. Therefore, it is challenging to learn such a variety of patterns into a monolithic model. For example, NMT-based models prefer to generate high-frequency words in the corpus according to the literature [28], [29], which is not

Manuscript received 1 April 2022; revised 5 January 2022; accepted 17 January 2023. Date of publication 19 January 2023; date of current version 18 April 2023. This work was supported by the National Natural Science Foundation of China under Grant 61772200, and in part by the Shanghai Natural Science Foundation under Grant 21ZR1416300. Recommended for acceptance by Z. Jin. *(Corresponding Authors: Huiqun Yu, Guisheng Fan.)*

The authors are with the Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China (e-mail: zhouziyi@mail.ecust.edu.cn; yhq@ecust.edu.cn; gsfan@ecust.edu.cn; hzj@mail.ecust.edu.cn; 15921709583@163.com).

Digital Object Identifier 10.1109/TSE.2023.3238161

a good phenomenon when adapted to code snippets that are less common in functionality and style. An alternative approach could be training a unique summarizer for each category of code so that they can perform much better in their own domains. Unfortunately, it is not easy to find general criteria to classify codes to make the trained summarizers optimal on each category in practice. On the other hand, the data allocated for each category may not be sufficient to well-train the data-hungry neural models.

Motivated by the IR-based methods and few-shot learning techniques [30], [31], [32], we propose a totally different framework for code summarization based on meta-learning and code retrieval to tackle above issues. In this framework, the summarization of each target code is formalized as a few-shot learning task, where its retrieved similar examples are used as training data and the testing example is the target code itself. In this way, we bypass the category problem by searching for small clusters of similar codes centered on each target code snippet. This is based on two basic observations: 1) the summaries of highly relevant codes are also similar and thus useful; 2) machine learning models can usually well generalize to examples similar to the training data due to the learned similar patterns. To effectively perform few-shot learning, our framework takes Model-Agnostic Meta-Learning (MAML) [33] as backbone. The purpose of meta-learning is to train a meta-learner that can quickly adapt to a new task from a small amount of new data [33]. Specifically, we obtain <code, comment> pairs similar to the target code snippet using a code-to-code retriever in a rank-and-filter manner. Given a neural code summarizer, we optimize it into a meta-learner via MAML. During inference, the meta-learner first adapts to the retrieved examples of the target code, i.e., yields an exclusive model, and then takes it as input to generate its summary. Considering that the retrieved examples can have different importance, we design a gradient aggregation mechanism for the adaptation process so that the meta-learner can utilize these examples according to their similarities.

Although with different motivation and framework, the idea of combining IR and deep learning for code summarization coincides with several recent work [29], [34], [35]. These efforts aim to incorporate the advantages of classic IR approaches, i.e., their better performance on low-frequency words [29] and the use of exemplars [34], into NMT-based models. However, they have some limitations compared with this work: 1) In order to make use of the retrieved examples, these approaches require either special model design [34], [35] or special retrieval method [29], which means an existing code summarizer have to be modified and retrained or utilize extra tricks for retrieval to obtain maximal improvements. By contrast, our framework is supposed to be model-agnostic for both the summarizer and retriever; 2) All of these methods handle the target codes and the retrieved ones by training a single model, and do not involve targeted learning process for diverse individuals, so their adaptability to the retrieved examples are limited; 3) Only the most similar example is considered by these models, while the proposed approach is able to utilize arbitrary number of similar examples. When taking only the top-1 retrieved result, the useful information in other candidates are ignored and the summarizers are more likely to be affected by the ranking performance of the retriever.

We evaluate our approach on two large real-world datasets: A Java dataset from [21] and a Python dataset from [36]. We spend extra effort to filter and preprocess the datasets in order to obtain reasonable evaluation results. We use common automatic metrics (BLEU [37], ROUGE-L [38], METEOR [39]) to evaluate the quality of the generated summaries, and also perform a human evaluation. Experimental results demonstrate that our approach obviously outperforms the IR-based approaches, state-of-the-art neural approaches as well as recent IR-based neural models by taking merely a vanilla Seq2Seq as basic code summarizer. Our approach shows better generalization ability in various aspects, and is able to well adapt to different neural code summarizers and retrieval methods. We highlight the main contributions of this paper as follows:

- We propose a novel meta-learning paradigm for code summarization, where the code summarizer learns to extract knowledge from similar examples through few-shot learning. To the best of our knowledge, this is the first work that introduces meta-learning into the task of code summarization.
- Extensive experiments conducted on two real-world datasets show that the proposed approach could significantly improve neural code summarizers and achieves state-of-the-art results with various task settings.

The rest of this article is organized as follows. Section II provides some preliminary knowledge. Section III elaborates the proposed approach. Section IV gives the detailed experimental setup. Section V illustrates the results and analysis. Section VI further discusses the effectiveness of our approach and the threats to validity. Related studies are reviewed in Section VII. Finally, Section VIII concludes the whole paper.

II. Background

A. Attentional Encoder-Decoder Framework

To translate source code to natural language, almost all existing neural code summarization models employ the attentional encoder-decoder architecture (also known as Seq2Seq) which is designed for Neural Machine Translation (NMT) [40], [41], [42], [43]. In general, the encoder maps a sequence of input vectors $x = (x_1, \ldots, x_n)$ to a continues representation $h = (h_1, \ldots, h_n)$. Typically, an RNN such as long short-term memory (LSTM) [44] and gated recurrent units (GRU) [45] can be used as the encoder:

$$h_t = f_{enc}(x_t, h_{t-1}) \tag{1}$$

where f_{enc} denotes an RNN function. Tree-RNNs or GNNs can also be applied [20], [25], [26], [27] in order to capture structural information of codes. The decoder is a unidirectional RNN that uses h and its previous outputs to predict a desired target sequence $y = (y_1, \ldots, y_m)$ one element at a time. The conditional probability of generating next token y_t is

$$\begin{aligned} p(y_t|y_{<t}, x) &= \text{softmax}(g(s_t, c_t)) \\ s_t &= f_{dec}(s_{t-1}, y_{t-1}, c_{t-1}) \end{aligned} \tag{2}$$

where g is a parametric function for probability estimation, f_{dec} is a decoder function, s_t is a decoding state and c_t is a context vector computed by the attention mechanism [42]. Attention is proved to be able to significantly improve the performance of NMT models. It selects important parts from input by assigning different weights to the elements in h at each decoding time step t

$$\begin{aligned} c_t &= \sum_{j=1}^{n} \alpha_{tj} h_j \\ \alpha_{tj} &= \frac{\exp(a(s_t, h_j))}{\sum_{k=1}^{n} \exp(a(s_t, h_k))} \end{aligned} \tag{3}$$

where a is an alignment score function that shows how well two states match.

The encoder-decoder network is traditionally trained via maximum likelihood estimation (MLE). Namely, the goal is to minimize the following loss function

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log p\left(y_t^i | y_{<t}^i, x^i\right) \tag{4}$$

where N is the number of training pairs, (x^i, y^i) is the i-th pair and T_i is the length of the i-th target sequence y^i.

B. Meta-Learning

Generally speaking, meta-learning [46], [47] is a technique that endows an agent with the ability of learning to learn. It intends to design models that can adapt to new concepts rapidly without numerous training examples. This is different from machine learning, which usually focuses on one task and requires a lot of training data. A good meta-learning model should be capable of well generalizing to new environments or tasks that do not appear in the training phase. A typical problem setting for meta-learning is few-shot learning [30], [31], [32], [33], such as N-way, k-shot classification. In the k-shot learning scenario, the model is trained to learn a new task from only k (a small value) observations. Recent meta-learning methods emerged from its natural ability to multiple tasks, reaching state-of-the-art performance on vision [31], [32], language [48], and robotic domains [49].

Meta-learning can be divided into three categories [50]: metric-based, model-based and optimization-based techniques. In this paper, the proposed approach is based on an impactful optimization-based algorithm: Model-Agnostic Meta-Learning (MAML) [33]. MAML uses a gradient-based optimization procedure, and is compatible with any model that trained via gradient descent. The key idea of MAML is to explicitly optimize for fast adaptation to new tasks by learning a good set of initialization parameters θ. Formally, consider a model (or learner) represented by a parametrized function f_θ with parameters θ. When the model adapts to a new task $\mathcal{T}_i$, the model updates its parameter θ to θ_i' using one (or more) gradient descent update,

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}\left(f_\theta\right) \tag{5}$$

where α is the step size, and $\mathcal{L}_{\mathcal{T}_i}$ is the loss function of task $\mathcal{T}_i$. This stage is called meta-training. Then the parameters θ are trained to optimize the performance of $f_{\theta_i'}$ on the unseen test examples across the tasks of current batch, which is called meta-testing. Namely, the meta-objective is

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta_i'}\right) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}\right) \tag{6}$$

where $p(\mathcal{T})$ is the distribution over tasks. It can also be optimized using gradient descent. Namely, the model parameters θ are updated as

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta_i'}\right) \tag{7}$$

where β is the meta step size. In effect, MAML aims to obtain a meta-learner such that a small number of gradient steps on a new task will produce maximally effective behavior on that task.

In most cases, only a handful of examples in the code corpus are highly similar to a given code snippet, e.g., contain similar identifiers and literals or be very close in their functionalities. Therefore, suitable retrieved examples are far from enough to train deep learning models, which always carry tens of millions of learnable parameters. This is the reason why we turn to the meta-learning technique. We choose MAML since it is one of the most popular, successful and widely-used meta-learning algorithms. With the help of MAML, we can build high quality internal representations of codes and summaries that are broadly suitable for various examples, so that slightly fine-tuning the model parameters can produce good results.

III. Meta-Learning for Code Summarization

In this section, we elaborate the detailed workflow of our proposed approach: Meta-Learning for Code Summarization (MLCS).

A. Overall Framework

First of all, we give the formal task definition of MLCS: Given a target code snippet x^i to summarize, the goal is to learn a unique summarizer for it using its similar example pairs $\mathcal{S}^i = \{(x^{ij}, y^{ij}) | 1 \le j \le k_i\}$ as training data in a few-shot learning manner, where y^{ij} is the comment of the j-th similar code snippet x^{ij} and k_i is determined by the code-to-code retrieval process.

Fig. 1 illustrates the training and inference procedure of MLCS. During the training loop, an existing code summarization model will go through meta-training and meta-testing repeatedly, and will finally be optimized into a meta-learner. The training corpus is also used as retrieval database in this stage. During testing, the optimized meta-learner will adapt to the retrieved examples for the target code snippet and then make prediction. For meta-train step, similar examples are obtained by using a code retriever followed by certain rules for filtering, and a gradient aggregation mechanism is designed to make use of them according to their similarities.

B. Basic Code Summarizer

Since this work does not focus on the structural or unstructured representation of code, we simply use a standard Seq2Seq model with attention as the basic code summarizer, without loss of generality (we also take another three state-of-the-art code summarization models into account as shown in Section V). In

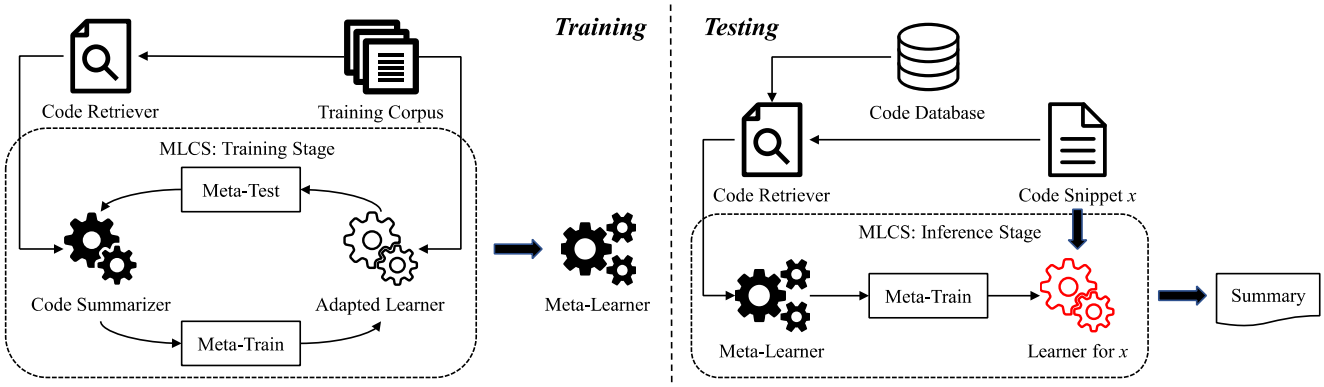


Fig. 1 Overall framework of our approach.

detail, we implement a 2-layer encoder-decoder model based on LSTM, following [29].

Similar to many data-driven code summarization models, the encoder takes token sequence of code as input. After preprocessing and tokenization for the code, a token sequence $x = (x_1, \ldots, x_n)$ will first be mapped to a sequence of embedding vectors using a learnable embedding lookup matrix W_e:

$$e_i = W_e^T x_i \tag{8}$$

Then, to better leverage the long-range contextual information, a bi-directional LSTM (biLSTM) is applied to encode it into a continuous representation h. Specifically, the i-th element of h is the concatenation of the forward LSTM state and the backward one, i.e., $h_i = [\vec{h}_i; \overleftarrow{h}_i]$.

During decoding, we employ multiplicative attention [42] for the decoder. Given an encoder state h_j and a decoder state s_t, the score function a in (3) is defined as

$$a(s_t, h_j) = s_j^T W_a h_j \tag{9}$$

where W_a is a parameter matrix and s_t is calculated by another LSTM. We initialize the decoder LSTM using the backward final state of the encoder, i.e., $s_0 = \overleftarrow{h}_n$.

C. Two-Stage Code Retrieval

1) Basic Retriever: A code-to-code retriever is needed to calculate the similarities between two pieces of codes. Many techniques from information retrieval (IR) could be applied here, e.g., represent code snippets via bag-of-words (BOW) model to estimate the relevance [2], [8]. Here we choose a semantic-based approach used in [29] (we also evaluate another three retrievers in Section V). It uses the state vectors $h = (h_1, \ldots h_n)$ computed by the biLSTM encoder to represent a code snippet, in that the sequential information and semantics of code are embedded into h through end-to-end training. In detail, after encoding a code snippet to $h \in \mathbb{R}^{n \times d}$, a global max pooling is applied to obtain a code vector $r \in \mathbb{R}^d$:

$$r = maxpooling([h_1, \ldots h_n]) \tag{10}$$

Then the similarity between code x^i and x^j can be computed using cosine similarity

$$\cos(r^i, r^j) = \frac{r^i \cdot r^j}{\|r^i\| \|r^j\|} \tag{11}$$

In most of our experiments, the training data is used as code database to retrieve from, following [29], [34], [35]. Since this cause data leakage of the training set during predicting, we further discuss the effect of unseen retrieval corpus in Section VI.

2) Filtering the Retrieved Examples: A drawback of previous retrieval-based neural approaches is that they only consider the most similar example and thus the information in other highly relevant examples is ignored. With MLCS, a code summarizer can naturally make use of arbitrary number of similar examples.

A simple way to get multiple similar examples is to retrieve top-K results from the database. However, this method is not reasonable when K is a fixed number larger than 1. The reason is intuitive: for a fixed K, the results may include noisy examples with low relevance, and thus mislead the model. This method is also heavily dependent on the ranking performance of retrieval module. Another way is to set a similarity threshold ε rather than using the number K. Since we do not make any assumptions about the retrieval method, this also leads to a potential issue: a specific similarity value may have different meanings for different retriever, especially when their similarities are within different ranges (e.g., for similarities calculated via cosine and BM25 [51]). Based on above concerns, we retrieve similar codes following two steps. First, we collect top-k results using the retriever, with a relatively large k, as candidates. Then, we calculate a general similarity metric based on text edit distance (or Levenshtein distance) for the threshold ε to filter the candidates:

$$sim(x^i, x^j) = 1 - \frac{dis(x^i, x^j)}{\max(|x^i|, |x^j|)} \tag{12}$$

where *dis* denotes text edit distance. If none of the candidates could satisfy the threshold, the top-1 result will be selected.

D. Training and Inference

To achieve more efficient model learning and inference via MAML in our scenario, we modify vanilla MAML algorithm by introducing two mechanisms, namely gradient aggregation and alternative inference procedure.

Algorithm 1: Training Stage of MLCS.

Require: Hyper-parameters α, β, k, ε
Require: Code summarization model f_θ
Require: Training dataset $\mathcal{D}$ consisting of code-summary pairs
1: Initialize θ
2: **while** not done **do**
3: Sample batch of training pairs $\{(x^i, y^i)\}$ from $\mathcal{D}$
4: **for all** (x^i, y^i) **do**
5: Retrieve top-k examples similar to code x^i and filter them by ε according to (12), build training set $\mathcal{S}^i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{S}^i}(f_\theta)$ according to (15)
7: Compute $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{S}^i}(f_\theta)$
8: **end for**
9: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_i \mathcal{L}_{(x^i,y^i)}(f_{\theta_i'})$
10: **end while**

Algorithm 2: Inference Stage of MLCS.

Require: Hyper-parameters α, n, k, ε
Require: Meta-learner f_θ' obtained via Algorithm 1
Require: Target code snippet x^i
1: $\gamma = \min(\frac{2}{3}\alpha, \frac{\alpha}{n-1})$
2: Retrieve top-k examples similar to code x^i and filter them by ε according to (12), build training set $\mathcal{S}^i$
3: **for** *step* **in** n
4: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{S}^i}(f_\theta')$ using $\mathcal{S}^i$ according to (15)
5: Compute $\theta \leftarrow \theta - \gamma \nabla_\theta \mathcal{L}_{\mathcal{S}^i}(f_\theta')$
6: **end for**
7: **output** Target summary $\hat{y}^i = f_\theta'(x^i)$

1) Aggregating the Gradients of Retrieved Examples: Since the retrieved examples in $\mathcal{S}^i$ have different similarities to x^i, they should contribute differently when optimizing the meta-learner. The key idea here is to consider the meta-train stage of MAML: the more relevant an example is to the target code, the further the model should step in the direction of this example during optimization. To achieve this, our approach computes a weighted sum of their gradients according to their similarities for back propagation, rather than simply takes the average as conventional. However, the variable size of $\mathcal{S}^i$ makes this a bit trickier. We overcome this problem by calculating a normalized weight u_j for each example in $\mathcal{S}^i$ via softmax function

$$u_j = \text{softmax}^\mu\left(sim\left(x^i, x^{ij}\right)\right) = \frac{\exp\left(\mu \cdot sim\left(x^i, x^{ij}\right)\right)}{\sum_k \exp\left(\mu \cdot sim\left(x^i, x^{ik}\right)\right)} \tag{13}$$

where μ is a base value to control the compact of the input component (increasing μ will make a large input has an even bigger output weight), and *sim* is the general similarity metric described in (12). The reason to introduce μ is that when $\mathcal{S}^i$ increases, the normalized weights of highly similar examples will be pulled down by the less relevant ones and the former tend to be underestimated. Thus, we set $\mu = k_i$ to alleviate such negative impact, where k_i is the size of $\mathcal{S}^i$. After that, a modified similarity is computed as

$$w_j = u_j \cdot \left(sim\left(x^i, x^{ij}\right) + 0.5\right) \tag{14}$$

Here an offset of 0.5 is added to the original similarity. Since the similarity is below 1, the obtained gradients for adaptation process will always be scaled down throughout training without the offset, which will slow the optimization process. On the other hand, we find that the average similarity of the top-1 retrieved examples is around 0.5. Therefore, we introduce the offset 0.5 as compensation instead of tuning the step size α. Finally, we aggregate the gradients of the examples in $\mathcal{S}^i$ as follows

$$\nabla_\theta \mathcal{L}_{\mathcal{S}^i}(f_\theta) = \sum_{j=1}^{k_i} w_j \nabla_\theta \mathcal{L}_{(x^{ij}, y^{ij})}(f_\theta) \tag{15}$$

The training process of MLCS is summarized in Algorithm 1. During meta-testing (step 9), the target example itself is used to evaluate and update the model. In effect, MLCS aims to learn a parameter initialization that can be rapidly tuned to a new code snippet. Therefore, a "general" model trained via (4) with a conventional task setting is quite suitable to initialize MLCS (since it already has certain capability to generalize to new codes). In other words, an existing code summarizer is possible to benefit from MLCS without retraining.

2) Alternative Inference Procedure: Changing the number of gradient steps n of meta-training could affect the final model performance, since more steps will lead to more frequent adjustment of descent direction and is more likely to achieve a good stop position. A large n will badly slow down the overall training process while MAML could achieve good results with n set to 1 [33]. In our scenario, we argue that it is important to take multiple steps during meta-training because the few-shot training set, i.e., the retrieved examples, is highly related to the target example and contains more useful information compared to those few-shot learning benchmarks [52], [53]. Concerning that it is always time-consuming to train neural code summarization models over a large corpus, we use different n during training and inference for efficiency. In detail, we still keep $n = 1$ during the training stage of MLCS, but employ an alternative inference procedure which takes a larger n. During inference, a decayed step size γ is used to fit the new n. Specifically, γ is set to be slightly larger than α / n considering the triangle inequality

$$\gamma = \begin{cases} \frac{2}{3}\alpha & n = 2 \\ \frac{\alpha}{n-1} & n > 2 \end{cases} \tag{16}$$

where α is the adaptation step size used during training. In our experiments, this empirical setting significantly improves the performance of the adapted code summarizer, with the training time unchanged. The inference procedure of MLCS is detailed in Algorithm 2.

TABLE I
THE STATISTICS OF TWO DATASETS BEFORE AND AFTER PREPROCESSING

State	Dataset	Source Codes				Comments			
		MaxL	AvgL	MedL	UniqTok	MaxL	AvgL	MedL	UniqTok
Original	JCSD	6153	103.2	66	208,090	590	16.8	11	47,863
	PCSD	32,256	153.9	89	114,258	4411	46.1	18	218,003
Preprocessed	JCSD	400	111.9	78	26,241	30	10.4	9	14,898
	PCSD	400	131.0	91	55,226	30	9.4	9	22,808

IV. EXPERIMENTAL SETUP

A. Dataset and Preprocessing

Our experiments are conducted upon two large datasets in Java and Python. The Java code summarization dataset (JCSD) is originally collected by Hu et al. [21] from GitHub. It contains 87136 pairs of <method, summary> for code summarization task. The authors take the first sentence of Javadoc as the summary of a method. The dataset is split into training, validation and testing sets in proportion with 8: 1: 1 after shuffling. We keep the same split setting to [21]. The Python code summarization dataset (PCSD) is also extracted from GitHub, and is processed by Barone et.al [36]. It includes 108726 <method, summary> pairs, where the docstrings are used as summary. Following [20], we split it into training, validation and testing set by 6: 2: 2.

We use *javalang* [54] and built-in *tokenize* module of Python to tokenize the source codes. To better train and evaluate the code summarization models, we perform some necessary preprocessing:

1. We remove the duplicate <method, summary> pairs from the datasets, and ensure that the retrieved examples will not contain the same code snippets as the target one during evaluation.
2. We take the first sentence of the summary as target comment since we find that the rest are always supplementary descriptions that are hard to predict based on the method itself. This is important for PCSD because the docstrings always contain example codes. For JCSD, we find that this step is not completely done by the providers so we process both of them.
3. Some redundant elements that can hardly be generated are further removed from the comments, including HTML labels, "TODO: …", "e.g., …", "i.e., …", URLs and special symbols.
4. We replace the numbers, MD5 and hash values in both codes and comments by a special token <num>.
5. The identifiers in the codes and comments are split according to camelCase and snake_case. In this way, the model could benefit from better sequential representation of code and learn to generate subtokens.
6. To better retain the structural information (e.g., blocks) in the token sequence of Python code, we tag indent and dedent as special tokens <t> and <\t> respectively for PCSD.
7. After these steps, the examples whose comments are less than 2 tokens are discarded due to their low amount of information. For efficiency, we truncate the codes to 400 tokens and the comments to 30 tokens according to their distributions.

The statistics of two datasets before and after preprocessing are shown in Table I. It gives the maximum length (MaxL), average length (AvgL), median length (MedL) and number of unique tokens (UniqTok) of codes and comments. As seen, the number of unique tokens is significantly dropped after token splitting and content filtering, for both codes and comments. This directly reduces the vocabulary size and data sparsity. The distributions of code lengths and comment lengths in the filtered datasets are shown in Fig. 2. A certain fraction of examples is concentrated at the tails due to truncation.

B. Training and Inference Settings

We take a conventionally trained Seq2Seq model as our basic code summarizer, as described in Section III. We set the dimensions of its LSTM hidden states and token embeddings to 256. The vocabulary sizes of codes and comments are limited to 40K and 25K, respectively. We clip gradient norm by 5, and apply dropout [55] of 0.3 on embedding vectors and the recurrent units. During conventional training, we initialize the parameters using Glorot initialization [56], and use a batch size of 64. To avoid overfitting, we stop training when the BLEU-4 score [37] recorded on the validation set does not improve within 4 epochs, and then select the best model according to BLEU-4. The model is optimized using Adam [57] with default settings.

The meta-train step of MLCS uses standard gradient descent method, with the step size α set to 0.4. During meta-testing, default Adam is used as meta-optimizer to perform model update, i.e., $\beta = 0.001$. Since the meta-optimization involves a gradient through gradient, we use a first-order approximation for efficiency following [33]. The training with MLCS lasts up to 15 epochs with a batch size of 20, and the model is chosen based on its validating BLEU score. We do not notice significant improvement using more similar examples during training, so we simply let $k = 1$ and $\varepsilon = 0$ in this stage. During inference, they are set to 10 and 0.7 according to the validating scores. The number of adaptation steps n during inference is set to 5. An advanced decoding strategy, beam search [43] is adopted during validating and testing, with the beam size and length penalty set to 5 and 1.0. The source code of our approach is publicly available at https://github.com/zy-zhou/MLCS.

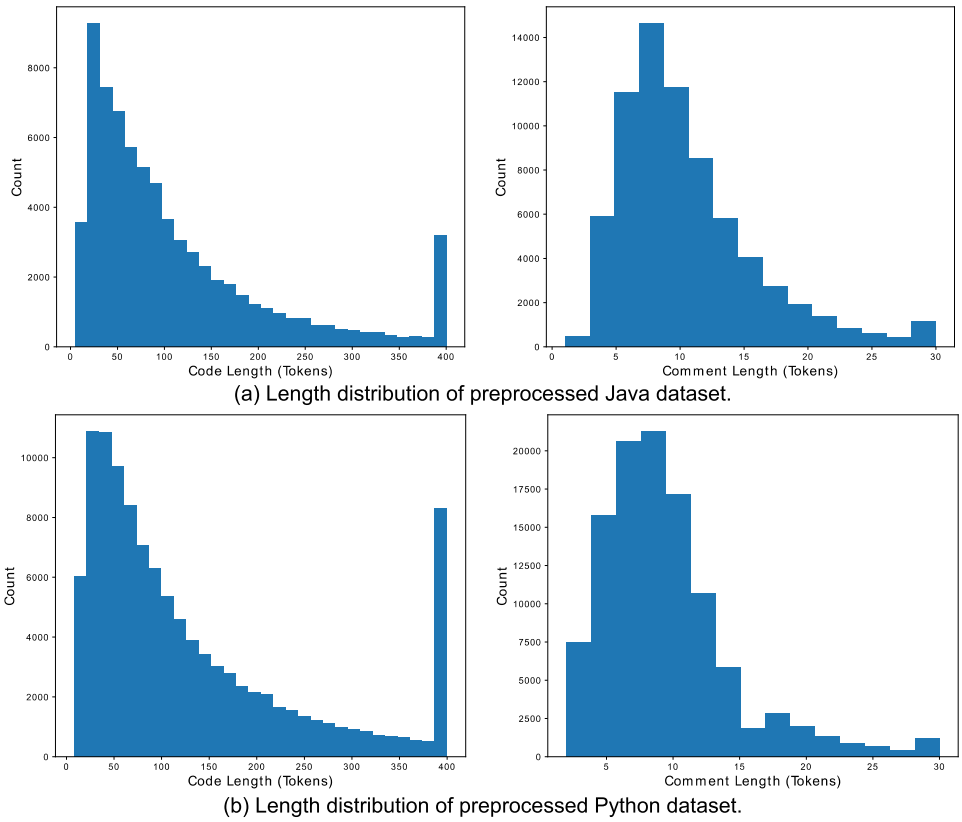


Fig. 2. Distribution of code lengths and comment lengths of the preprocessed datasets.

C. Baselines

We compare our approach with several state-of-the-art neural code summarization models, which exploit various types of information as their inputs:

- **DeepCom** [19] uses a customized traversal method named SBT to linearize AST into node sequence, and then applies an attentional Seq2Seq model to summarize code snippet. This is one of the earliest models that combine NMT technique with structural information of code. To make it more comparable to other baselines, we replace its unidirectional LSTM encoder with a bi-directional one.
- **NMT** is our basic code summarizer. It is representative among the models that take token sequence of code as input [17], [18], [21]. We report its scores to better show the role of MLCS.
- **NMT (Tf)** refers to Transformer [76], a currently state-of-the-art model for sequence modeling and transduction tasks in the field of natural language processing. Different from RNN or CNN, it is solely based on attention mechanism. Transformer is widely used in machine translation, and also has been applied to code summarization recently [77], by taking a stream of code tokens as input.
- **ast-attendgru** [23] is a typical model with multiple inputs, which has one GRU encoder for token sequence of code and another one for its flattened AST, i.e., SBT sequence. During decoding, the attentional contexts of both inputs are fused to predict a target word. Similar strategies are adopted in [20], [22], [27]. For fair comparison, we replace its GRUs with LSTMs (the encoders are also set to be bi-directional, and we find this performs obviously better than the original GRU model), despite that it is still named "gru".
- **Rencos** [29] is the first work that combines retrieval-based and NMT-based method in code summarization. Given a target code snippet, it retrieves two most similar code snippets based on syntax and semantics respectively, and then feeds them into a trained code summarizer. The code summarizer translates the three codes simultaneously and combines their output probabilities according to their similarities. It uses the same code summarizer as ours.
- **Re^{2}Com** [34] is another retrieval-based model proposed recently. Compared to Rencos, it further considers the existing comment of a similar code snippet and uses it as exemplar. In detail, it builds an encoder-decoder model with four biLSTM encoders for the given code snippet, its SBT sequence, its similar code and the exemplar, respectively. The context vectors of the target code and exemplar are fused according to the code similarity.
- **HGNN** [35] is a GNN-based model which also combines retrieval and generation methods. Similar to Re^{2}Com, it takes the top-1 retrieved code snippet and its corresponding summary as input. But differently, it introduces several new mechanisms towards the graph representation of code including the use of code property graph (CPG) and a HybridGNN with customized message passing mechanisms.

These baseline models use similar hyper-parameters as presented in their papers. Specifically, their LSTM hidden sizes are all 256 for both directions (i.e., the concatenated state size is 512). The embedding sizes of DeepCom, NMT, Rencos and HGNN are 256, while those of ast-attendgru and Re^{2}Com are 100 following their original configurations. For NMT (Tf), we apply the base model described in [76], and use the learning rate of 1e-4 for the Adam optimizer. Other training and inference settings of the baselines are all the same as NMT. To obtain the most similar example for Re^{2}Com, we apply our basic code retriever described in Section III.A for consistency.

D. Evaluation Metrics

Following previous work [17], [20], [22], [27], [29], we evaluate the quality of generated summaries using three automatic metrics: BLEU [37], ROUGE-L [38] and METEOR [39]. They are widely used in both NMT and code summarization since they are close to human assessment. These scores are reported in percentage (the bigger the better), and all models are evaluated using the same script.

Given a generated sentence, BLEU calculates the weighted geometric mean of *n*-gram precision on reference sentence for different *n*, multiplied by a brevity penalty to short predictions

$$\begin{aligned} BLEU &= BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \\ BP &= \begin{cases} 1 & c > r \\ \exp\left(1-\frac{r}{c}\right) & c \le r \end{cases} \end{aligned} \tag{17}$$

where p_n is the *n*-gram matching precision of length *n* subsequences, *c* is the length of candidate sentence, *r* is the reference length and *BP* refers to brevity penalty. We set $w_n = 1 \,/\, N$ and compute BLEU for $N = 1, 2, 3$ and 4 (denoted as BLEU-*N*). Since higher order *n*-grams may not overlap, we evaluate both BLEU-4 with and without NIST smoothing [58]. The smoothed BLEU-4 is denoted as BLEU-4(s).

ROUGE-L is a common metric of text summarization. It computes the similarity between generated summary and reference based on the length of longest common subsequence. Giving a reference summary *X* and a candidate *Y* of length *r* and *c* respectively, it is computed as

$$\begin{aligned} P_{lcs} &= \frac{LCS\left(X,Y\right)}{c} \\ R_{lcs} &= \frac{LCS\left(X,Y\right)}{r} \\ F_{lcs} &= \frac{\left(1+\beta^2\right) P_{lcs} R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \end{aligned} \tag{18}$$

where *LCS* calculates the length of the longest common subsequence of two sentences. In our experiments, F_{lcs} is reported and β is set to 1.

On the basis of BLEU, METEOR takes recall into account and applies synonym matching. It computes a parameterized harmonic mean of unigram precision *P* and recall *R* with a penalty

$$\begin{aligned} METEOR &= (1 - Pen) \cdot F_{mean} \\ F_{mean} &= \frac{PR}{\alpha P + (1-\alpha) R} \\ Pen &= \gamma \cdot \left(\frac{ch}{m}\right)^{\beta} \end{aligned} \tag{19}$$

where *m* is the number of unigram matches and *ch* is the number of chunks. We use the optimized parameter settings for English target described in [39]: $\alpha = 0.85$, $\beta = 0.2$ and $\gamma = 0.6$. The implementations of BLEU and METEOR are from NLTK [59].

V. Results and Analysis

We evaluate and analyze our approach by investigating the following research questions:

- *RQ1:* How does the proposed approach perform generally compared to the baselines?
- *RQ2:* To what extent can MLCS improve existing neural code summarizers?
- *RQ3:* What is the effect of using different numbers of retrieved examples, and how can the model benefit from multiple similar examples via MLCS?
- *RQ4:* How well can MLCS adapt to different code retrieval methods?
- *RQ5:* What is the quality of the summaries produced by our approach in practice?
- *RQ6:* What is the impact of cross-project data splitting and near-duplicate removal on retrieval-based code summarizers?
- *RQ7:* How does MLCS perform against existing code category-based method, and how does it perform on different code categories?

A. RQ1: Overall Performance

The overall results of different approaches in terms of the above metrics are shown in Table II. According to the table, DeepCom yields worse results than ast-attendgru and NMT on all metrics, although we enhanced it with a biLSTM encoder. Note that the main difference between ast-attendgru and DeepCom is that ast-attendgru equips an extra token encoder. This suggests that leveraging both structural and sequential information is helpful for this task. The scores of ast-attendgru and NMT are close. In detail, NMT scores higher on BLEU and METEOR on JCSD, while ast-attendgru performs slightly better on PCSD in terms of all metrics except for BLEU-4. The observation is that applying an extra SBT encoder could not achieve sufficient improvement against encoding only token sequence, which is also reported in [23], [34]. We assume one of the reasons is that simultaneously training two encoders over different modalities of input is more difficult and the components within the model may be insufficiently optimized. The effectiveness of SBT could be another cause: most identifier semantics are lost in the SBT sequence and its length is much longer than the token sequence which is not friendly to RNNs. Among the non-retrieval baselines, NMT (Tf) achieves the highest BLEU-4 and BLEU-4(s) scores, while its ROUGE-L scores are obviously lower than ast-attendgru and NMT. We assume the reason is that

TABLE II
OVERALL PERFORMANCE OF PROPOSED APPROACH COMPARED TO BASELINES

Dataset	Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
JCSD	DeepCom	26.03	16.53	11.57	8.91	14.95	32.71	15.95
	ast-attendgru	32.90	22.05	15.26	11.32	18.61	42.04	20.66
	NMT	32.66	21.87	15.54	12.02	18.99	41.69	20.72
	NMT (Tf)	32.02	21.93	17.01	14.33	19.80	37.32	19.88
	Rencos	34.94	24.15	17.55	13.55	20.59	42.85	21.56
	HGNN	31.80	23.54	18.95	15.59	21.11	36.95	19.07
	Re^{2}Com	36.37	26.50	20.89	17.38	23.69	44.40	22.81
	NMT + MLCS	**37.86**	**29.00**	**23.47**	**20.16**	**26.47**	**46.94**	**23.89**
PCSD	DeepCom	24.92	15.86	11.64	8.92	15.07	30.23	14.95
	ast-attendgru	31.32	20.38	14.07	10.26	18.11	38.64	18.99
	NMT	30.58	19.62	13.95	10.61	17.98	37.29	18.51
	NMT (Tf)	30.02	20.26	15.90	13.16	18.91	34.21	18.55
	Rencos	33.70	22.80	16.77	12.85	20.44	39.97	20.12
	HGNN	31.92	22.51	17.42	13.97	20.67	38.04	19.29
	Re^{2}Com	35.17	25.25	19.78	15.97	23.09	41.57	21.37
	NMT + MLCS	**35.89**	**26.43**	**21.16**	**17.38**	**24.70**	**43.06**	**21.98**

our training corpus may be relatively small for such large model, while the RNN-based models are more data-efficient.

It is clear that the retrieval-based methods perform generally better than those using only the target code snippet. This verifies the significance of similar examples. By encoding the similar codes, Rencos is able to improve NMT by more than 1 point of BLEU-4 and ROUGE-L on JCSD, and the gap is even larger on PCSD. Compared with Rencos, Re^{2}Com is obviously better on all metrics, owing to its additional inputs of exemplar and structural information. The results we reproduce for Re^{2}Com are comparable to those presented in [34], which exceed NMT by around 45% of BLEU-4 on JCSD. HGNN outperforms the non-retrieval baselines in terms of all metrics on PCSD, while its ROUGE-L and METEOR scores on JCSD are worse than NMT. This is possibly due to that the graph representation of Java is more complex and more challenging to be learnt than Python. The ROUGE-L and METEOR scores of HGNN are also lower than Rencos, which is inconsistent to the results in [35]. Since HGNN is not open-sourced (we contacted the authors but did not receive any reply), we try our best to implement it according to its paper but fail to reproduce similar comparative results.

As seen, our approach NMT + MLCS achieves the best scores on all metrics and further outperforms the strongest retrieval-based baseline, i.e., Re^{2}Com, by a certain gap despite that NMT itself is much worse than Re^{2}Com. Specifically, it improves Re^{2}Com by more than 2 points of smoothed BLEU-4 and ROUGE-L on JCSD, and around 1.5 points of those on PCSD. Since NMT is merely a standard Seq2Seq model running over code tokens, the improvements are undoubtedly owing to MLCS. Unlike Re^{2}Com and Rencos which embed the knowledge of similar examples via extra model input, MLCS learns to summarize code from them. In other words, this is "learning to edit" versus "learning to learn". Since there is a learning process or fast adapting (step 4 and 5 of Algorithm 2) before generating a summary, the model equipped with MLCS could better use the retrieved examples as well as generalize to new code snippets.

B. RQ2: Improvements of Applying MLCS

Similar to MAML, MLCS is supposed to be model-agnostic. To answer RQ2, we apply MLCS to other non-retrieval-based neural baselines, i.e., DeepCom, ast-attendgru and NMT (Tf) to verify its effectiveness. They use the same hyper-parameters of MLCS as ours, except for the step size α and β. Recall that we use vanilla gradient descent method for meta-training (step 7 of Algorithm 1) which can not adaptively adjust the step size like advanced optimizers such as Adam. Therefore, the optimal value of α seems to be different for different models. It is worth trying other optimizers to check the impact of α for future work. In this study, we roughly tune α based on the validating BLEU-4 scores, and set it to 0.4, 0.2 and 2e-4 for DeepCom, ast-attendgru and NMT (Tf) respectively. The step size β follows their learning rates described in Section IV.C, and we use the trained models from RQ1 to start MLCS training. We report their scores and percentages of relative improvement in Table III.

According to Table III, MLCS is able to improve these models consistently in terms of all metrics on both datasets, and the gap is considerable. Specifically, their scores increase by an average of 64%, 16% and 19% on BLEU-4, ROUGE-L and METEOR respectively after applying MLCS. With MLCS, even the worst baseline, i.e., DeepCom scores higher than Re^{2}Com in terms of BLEU-4. Except for similar code and exemplar, ast-attendgru has the same input as Re^{2}Com. When it uses similar examples in the manner of MLCS, it is able to outperform the latter on

TABLE III
THE IMPROVEMENTS OF THE BASELINES BY UTILIZING MLCS

Dataset	Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
JCSD	DeepCom + MLCS	34.36 **(+32.0%)**	26.40 **(+59.7%)**	21.94 **(+89.6%)**	18.95 **(+112.7%)**	24.37 **(+63.0%)**	40.30 **(+23.2%)**	20.97 **(+31.5%)**
	ast-attendgru + MLCS	**37.88** (+15.1%)	28.60 (+29.7%)	22.74 (+49.0%)	19.17 (+69.3%)	25.76 (+38.4%)	**47.37** (+12.7%)	23.78 (+15.1%)
	NMT + MLCS	37.86 (+15.9%)	**29.00** (+32.6%)	**23.47** (+51.0%)	**20.16** (+67.7%)	**26.47** (+39.4%)	46.94 (+12.6%)	**23.89** (+15.3%)
	NMT (Tf) + MLCS	36.40 (+13.7%)	26.51 (+20.9%)	21.49 (+26.3%)	18.54 (+29.4%)	24.28 (+22.6%)	42.11 (+12.8%)	22.41 (+12.7%)
PCSD	DeepCom + MLCS	32.60 **(+30.8%)**	24.39 **(+53.8%)**	19.93 **(+71.2%)**	16.65 **(+86.7%)**	22.90 **(+52.0%)**	37.81 **(+25.1%)**	19.66 **(+31.5%)**
	ast-attendgru + MLCS	35.75 (+14.1%)	26.07 (+27.9%)	20.31 (+44.3%)	16.33 (+59.2%)	23.97 (+32.4%)	**43.65** (+13.0%)	21.79 (+14.7%)
	NMT + MLCS	**35.89** (+17.4%)	**26.43** (+34.7%)	**21.16** (+51.7%)	**17.38** (+63.8%)	**24.70** (+37.4%)	43.06 (+15.5%)	**21.98** (+18.7%)
	NMT (Tf) + MLCS	33.81 (+12.6%)	24.28 (+19.8%)	19.58 (+23.1%)	16.50 (+25.4%)	22.83 (+20.7%)	38.70 (+13.1%)	20.63 (+11.2%)

all metrics. This again indicates that MLCS is more effective to handle similar examples. Although the exemplars are not exposed to the model via input (of encoder), they are treated as targets to predict during meta-training and thus could directly affect the whole model especially the decoder. It is interesting that DeepCom + MLCS achieves the highest percentage of increase in terms of all metrics, with up to 112% of BLEU-4 on JCSD. This suggests that MLCS is possible to take the performance of a naïve model to a new level.

C. RQ3: Effect of Gradient Aggregation Under Different Numbers of Similar Examples

In this study, the number of retrieved examples to use is controlled by hyper-parameters k and ε. We alter them to check the performance trend of the models. When answering RQ3, we also verify the effectiveness of Gradient Aggregation (GA) described in Section III.D, which is the key component of MLCS that aims to better use similar examples. Therefore, we consider two variants of MLCS:

- **NMT + MLCS (no μ)** removes the base value μ from (13) when aggregating the gradients for meta-training.
- **NMT + MLCS (no GA)** simply averages the gradients of retrieved examples instead of using GA. This is conventional when using MAML in few-shot learning tasks.

Furthermore, it is worth comparing to Rencos because it could also utilize multiple similar examples (although its authors find that it failed to benefit from more) and take NMT as backbone. For comparison, we only consider semantic-level similar examples here for Rencos just like ours. The experiments in this section are conducted on JCSD.

We first check the effect of changing ε with a fixed k of 10, which follows our rank-and-filter retrieval process. In particular, $\varepsilon = 1$ means only one similar example is used and $\varepsilon = 0$ means all of the 10 similar examples are used. Recall that when no candidate can satisfy the threshold ε, the top-1 retrieved example is assigned. This setting is important for both our models and Rencos since that if no similar example is chosen, they will degrade into NMT and produce underestimated results. The curves are shown in Fig. 3. Overall, MLCS with GA performs obviously better than MLCS without GA as well as Rencos. This proves that GA is crucial to MLCS no matter how many retrieved examples are used, and simply averaging over the gradients of low-quality examples will definitely mislead the model. We can find that the scores are stable with a large ε because the input is well-filtered. However, they start to show a downward trend at $\varepsilon = 0.6$ due to more involved noisy examples. Different from other approaches, the scores of NMT + MLCS decline more slowly and begins to recover after $\varepsilon = 0.3$. Compared to NMT + MLCS (no μ), we can conclude that μ plays an important role to mitigate the impact of noisy examples. Since $\mu = k_i$, the computed weights of highly relevant examples will be dominating when their number k_i becomes large, and the noisy ones will be ignored.

However, only presenting Fig. 3 is not enough to show how can the models benefit from multiple similar examples, and there is no obvious improvement when changing ε according to Fig. 3. The underlying reason is that for a certain ε, many target codes cannot receive more than one similar example (especially for a large ε), and compute scores for such targets will lead to underestimated improvements because setting ε has no effect on them. Therefore, we first choose the target codes with more than one similar example for each ε to evaluate, and then calculate the relative gain g_ϵ over them between using multiple similar examples and using only one. We show the results in Fig. 4. MLCS with GA is the only approach that can achieve positive g_ϵ. By contrast, g_ϵ of NMT + MLCS (no GA) and Rencos remains negative and reduce drastically when ε decreases. Rencos combines the decoding results of similar codes from a single model (namely NMT), and directly manipulates their output probabilities, which seems to be easily disturbed by multiple inputs. Although NMT + MLCS (no μ)

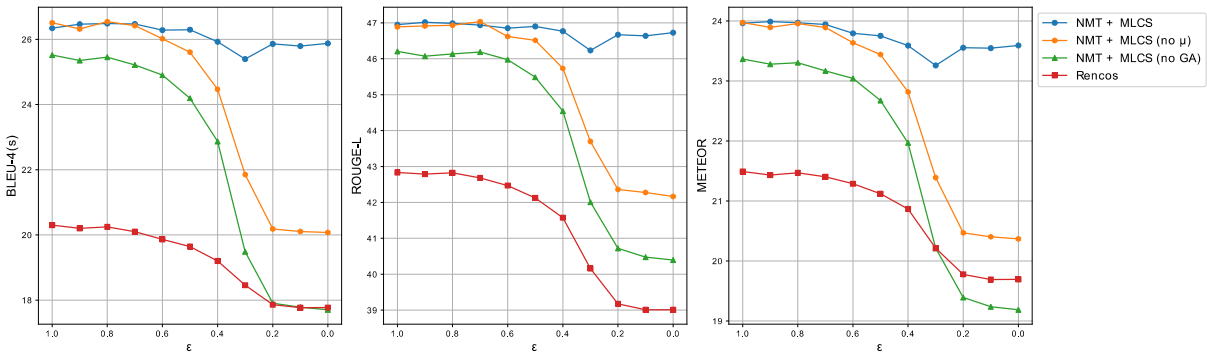


Fig. 3. Effect of changing ε on performance with $k = 10$.

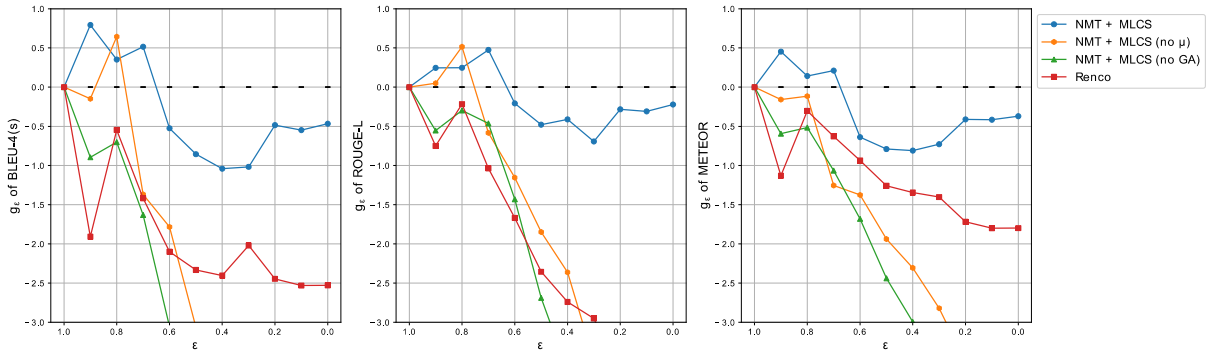


Fig. 4. The relative gain g_ϵ of using different number of retrieved examples.

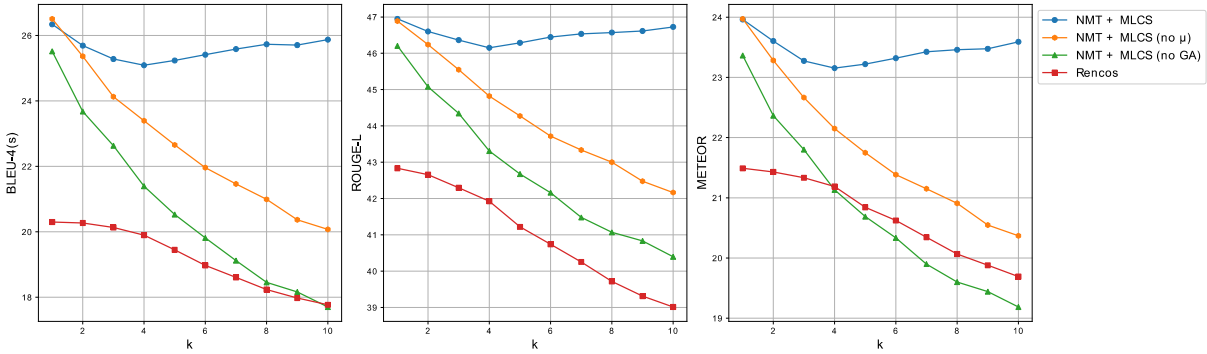


Fig. 5. Effect of changing k on performance with $\varepsilon = 0$.

scores well at a few points, its curves are not as stable as NMT + MLCS whose g_ϵ keeps positive when ε is between 0.7 and 0.9. Generally, multiple well-chosen similar examples could provide more useful information and less biased signal to the model. As shown, MLCS is able to benefit from them with a good ε, and the design of GA is reasonable.

In addition, we report the performance trends when varying k with $\varepsilon = 0$, as shown in Fig. 5. Despite that using a fixed k is not reasonable, this could reflect the tolerance of the models to noisy examples. The performance of the baselines decreases stably when k becomes larger, while the performance of our approach rebounds at $k = 4$ thanks to GA. NMT + MLCS (no GA) does not consider the similarities of retrieved codes, so its performance drops faster than Rencos. Combined with the scores in Fig. 3, it can be concluded that our approach is more robust.

D. RQ4: Performance of Using Different Code Retrievers

In practical scenarios, a code summarizer is likely to be a tool or plugin of a more complex system. Thus, a good retrieval-based approach should be flexible to adapt to various retrieval methods provided elsewhere, which is the motivation of our RQ4. We consider another three widely-used IR approaches to verify the effectiveness of MLCS:

TABLE IV
THE PERFORMANCE OF USING DIFFERENT CODE RETRIEVERS

Model	Retriever	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
Retrieval Only	VSM	34.71	26.31	22.51	19.85	24.39	38.22	21.27
	LSI	33.91	25.45	21.86	19.30	23.73	37.30	20.75
	FastText	32.23	24.38	20.92	18.47	22.75	35.25	19.63
	biLSTM	35.68	27.40	23.33	**20.43**	25.10	39.24	21.87
Re^{2}Com	VSM	34.45	24.47	18.88	15.63	21.83	42.34	21.62
	LSI	34.51	24.48	18.86	15.61	21.87	42.43	21.61
	FastText	33.57	23.70	18.15	14.92	21.17	41.48	21.13
	biLSTM	36.37	26.50	20.89	17.38	23.69	44.40	22.81
NMT + MLCS	VSM	36.95	28.00	22.42	19.17	25.56	46.40	23.43
	LSI	36.94	27.97	22.46	19.22	25.59	46.22	23.36
	FastText	36.48	27.47	21.83	18.54	25.07	45.78	23.14
	biLSTM	**37.86**	**29.00**	**23.47**	20.16	**26.47**	**46.94**	**23.89**

- **Vector Space Model (VSM)** [60] is a common IR technique based on bag-of-words model. When applying it for code retrieval [2], a code snippet is usually represented as a vector via Term Frequency-Inverse Document Frequency (TF-IDF). After that, cosine similarity is used for retrieval.
- **Latent Semantic Indexing (LSI)** recognizes term relevance between terms and concepts, and analyzes the latent meaning of documents based on singular value decomposition. Haiduc et al. [8] employ LSI to extract important terms of source codes for summarization. After representing code snippets as document matrix, the reduced vectors are used to compute similarity via cosine. We set the dimension of LSI vectors, i.e., number of topics to 500 following [29].
- **FastText** [61] is a well-known library for learning continuous word representations, or embeddings on unlabeled corpora. It considers the morphology of words by representing each word as a bag of character *n*-grams. Different from VSM and LSI, its trained vectors could indicate the level of semantic similarity between words. For its high efficiency, it is widely used for text classification and matching. We first use training set to get the token vectors of codes, and then represent a code snippet by taking the average of its token vectors. Accordingly, the cosine similarity is calculated for retrieval.

We denote our basic retriever as biLSTM. As mentioned before, the IR approaches themselves are typical ways to provide code summaries [2], [8], [10]. Thus, we also examine the performance of only use these approaches to verify the effectiveness of MLCS. In detail, we first retrieve the top 10 similar code snippets and then choose the summary of the code with highest *sim* value based on text edit distances (12), which follows our retrieval process described in Section III.C. To better answer RQ4, we take Re^{2}Com as baseline when applying above retrievers in that it does not require specific retrieval approach compared to Rencos. The experiments are conducted on JCSD and we present the results in Table IV.

At first glance, the pure IR-based approaches achieve higher BLEU scores than most of their neural counterparts. This phenomenon also appears in existing work [29], [34], which is due to the IR methods directly use the reference summaries of retrieved codes. In this way, the provided summaries are always fluent and grammatically accurate compared to those generated by neural models. Therefore, the precision of continuous words in their summaries are better, i.e., they are able to score better on high order BLEU. However, they perform worse on BLEU-1, ROUGE-L and METEOR, indicating that the keywords in their results could be less relevant to the targets. It can be seen that biLSTM has the best retrieval accuracy among IR methods. In effect, it is the only retriever obtained via supervised training since it is optimized end-to-end with the code summarizer via (4). Its task-specific nature makes it performs better than others. LSI shows slightly worse scores than VSM because the code vectors computed by LSI is reduced and could be less informative than those from VSM. Although FastText is able to model the semantic relationship between tokens, it is not as good as LSI and VSM, showing that it is not capable of handling long token sequences.

Clearly, the performance of retrieval-based neural models is affected by the performance of corresponding retriever. We can find that Re^{2}Com is more susceptible to retrieval methods than NMT + MLCS. To be specific, when changing the retriever from biLSTM to any of the above three retrievers, it loses more than 1.9 points of ROUGE-L, 1.1 points of METEOR and up to 2.5 points of BLEU-4(s). By contrast, the decreases in these metrics are controlled below 1.2, 0.75 and 1.4 points respectively by our approach. This indicates that MLCS is more robust for various retrievers as well. Note that Re^{2}Com uses only the top-1 result from the retriever, so it is heavily dependent on the ranking performance of the retriever. On contrary, MLCS allows the retriever to return top-k examples and then filter them with a general metric, which is more stable. Moreover, MLCS has the ability to "choose" from multiple similar examples and thus the

noisy examples will have less impact on it, as shown before. Besides, our approach obviously outperforms all IR methods on all metrics except for BLEU-4, indicating that MLCS effectively combines the advantages of IR methods and neural networks.

E. RQ5: Human Evaluation

In above experiments, we use automatic metrics to evaluate different approaches quickly and objectively, which measure the similarity between their generated summaries and references. However, they do not always agree with the actual quality of the results. Since these metrics are not on semantic level and do not consider the corresponding source code, the scores can be biased, especially when the quality of reference summary is low. Therefore, we conduct human study following previous studies [17], [22], [34] to evaluate the generated results from our approach and the baselines, as a supplement.

Similar to [17] and [34], we consider two uncorrelated aspects of the summary: *naturalness* and *informativeness*. Their scores are both integers and range from 1 to 5. To better guide the participants and make the evaluation results more objective, we provide clear definition of each score to them. Naturalness refers to the grammaticality and fluency of the summary. We explain each score to the participants as follows:

- Score 1: The result is totally not readable
- Score 2: The result is hard to read but part of its meaning can be guessed.
- Score 3: The result contains several grammatical errors or repeated phrases but they do not affect understanding.
- Score 4: The result contains only one repeated phrase or minor grammatical error and is easy to read.
- Score 5: There is no grammatical error or repeated phrase in the result and it is as smooth as from human.

Informativeness refers to the amount of key content carried over from the input code to the generated comments, ignoring fluency of the text. We explain its scores to the participants as follows:

- Score 1: The result cannot reflect any functionality of the code snippet and is totally misleading.
- Score 2: The result shares some information with the code but the rest part is inconsistent with the actual functionality of the code.
- Score 3: The result covers the very basic functionality of the code (such as actions) but loses specific information and seems indifferent.
- Score 4: The result provides a specific description of the code, and only misses the minor details compared to the reference.
- Score 5: The result is able to summarize the code precisely and no worse than the reference.

We invite 12 participants with excellent English level to rate the generated summaries. They are either developers or Ph.D. students of computer science, with 2-5 years of Java and Python development experience and at least 5 years of total programming experience. We first randomly choose 150 code snippets from the testing data (75 from JCSD and the other 75 from PCSD), and equally divide them into 6 groups, where each group is evaluated by two participants in the form of questionnaire. For each code snippet in the questionnaire, we first show the code itself together with the human-written summary for reference, and then randomly list the summaries generated by using retrieval only (biLSTM in Section V.D), NMT, Rencos, Re^{2}Com and our approach. The model names are hidden so that the participants do not know which model a certain result is generated from. The participants are asked to rate the generated results according to the corresponding code and ground truth summary in terms of above aspects. The human-written summaries are also rated for comparison. For each summary, we use the average score of the two raters. Each questionnaire is finished within 75 minutes.

TABLE V
THE RESULTS OF HUMAN EVALUATION

Model	Naturalness	Informativeness
Reference	4.64	4.48
Retrieval Only	**4.53**	2.45
NMT	4.37	2.76
Rencos	4.47	2.95
Re^{2}Com	4.35	2.88
NMT + MLCS	4.48	**3.20**

The average scores of different approaches are shown in Table V. There is a certain gap between the scores of human-written summaries and the full score of 5. This suggests that there exists low-quality summaries in the dataset, which is a potential threat to validity. All neural models get high scores on naturalness, showing that they are able to generate fluent summaries (they use the same beam search setting). The highest naturalness score is achieved by Retrieval Only. As mentioned in Section V.D, the direct use of human-written summaries by IR methods ensures their results are usually more fluent than the generated ones. In spite of its good naturalness, Retrieval Only shows the worst informativeness score, indicating that the retrieved summaries are not always helpful. We can find that NMT + MLCS performs the best in terms of informativeness. To further verify the effectiveness of our approach, we conduct Wilcoxon signed-rank tests [62] on the informativeness scores. When comparing our approach with Retrieval Only, NMT, Rencos and Re^{2}Com, we get the p-values of 1.79e-06, 0.0018, 0.0035 and 0.0006 respectively, which denotes that the improvements are statistically significant. We also use Krippendorff's alpha value [79] to measure the agreement among two raters in each group, and all the values are above 0.7, denoting good agreement.

F. RQ6: Cross-project Evaluation With Near-Duplicates Removal

We noticed that there are massive duplicate code snippets in our datasets, which could lead to biased evaluation results, so we made our training data and testing data disjoint by removing exactly duplicate codes. Nevertheless, a recent study [78] reveals that large amount of near-duplicate code in the dataset can also result in inflated performance of machine learning models.

TABLE VI
PERFORMANCE OF DIFFERENT APPROACHES WITH NEAR-DUPLICATES REMOVAL

Type	Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
Non-retrieval-based	DeepCom	22.51	13.38	9.35	6.89	12.88	27.72	13.56
	ast-attendgru	29.15	17.98	11.78	8.22	15.99	36.46	17.75
	NMT	28.31	17.09	11.53	8.44	15.74	34.98	17.19
	NMT (Tf)	26.87	16.83	12.60	10.12	15.78	31.07	16.68
Retrieval-based	Rencos	30.87	19.86	14.02	10.33	17.85	37.27	18.52
	HGNN	29.05	19.46	14.53	11.37	17.96	35.26	17.65
	Re²Com	32.30	22.11	16.70	13.17	20.20	38.88	19.72
	NMT + MLCS	**32.98**	**23.16**	**17.82**	**14.22**	**21.45**	**40.05**	**20.15**

TABLE VII
CROSS-PROJECT PERFORMANCE OF DIFFERENT APPROACHES WITH NEAR-DUPLICATES REMOVAL

Type	Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
Non-retrieval-based	DeepCom	10.75	3.07	1.13	0.48	4.35	14.56	6.80
	ast-attendgru	18.06	6.59	2.71	1.32	7.32	24.23	11.60
	NMT	17.79	6.24	2.51	1.16	7.21	23.96	11.44
	NMT (Tf)	12.94	3.91	1.95	1.33	5.32	16.22	8.41
Retrieval-based	Retrieval Only	10.44	3.82	2.72	2.34	5.19	12.89	6.93
	Rencos	14.75	5.03	2.28	1.29	6.03	18.99	9.24
	HGNN	13.06	4.41	2.22	1.43	5.97	18.41	8.56
	Re²Com	16.80	6.23	2.92	1.80	7.26	22.93	10.92
	ast-attendgru + MLCS	**18.14**	**7.42**	**3.80**	**2.38**	**8.45**	**25.33**	**11.86**

Since the retrieval-based code summarizers are dependent on the similar examples retrieved from codebase, we wonder how will they perform if there are few highly relevant examples (i.e., near-duplicates) available. To achieve this, we first remove all near-duplicate codes that appear both in training and testing set of PCSD using the duplication index provided by [78], and then recalculate the scores of different approaches, as shown in Table VI. Technically, the near-duplicates are identified by a code clone detection tool according to [78], where two code snippets are considered to be duplicates if the Jaccard similarities of their identifiers and literals exceed certain thresholds. Compared to the results in Table II, we can find that the scores of all models drop for a few points. This verifies that the near-duplicate codes appear both in training and testing set will inflate the evaluation results of code summarization models, no matter whether they are based on retrieval or not. The retrieval-based approaches still perform generally better than non-retrieval-based models in Table VI, and the scores of our approach, i.e., NMT + MLCS are still the highest.

In addition, we perform cross-project evaluation on the de-duplicated corpus. In this setting, retrieving from the project where the target code located is not allowed, which is more challenging to the retrieval-based approaches. Specifically, after removing all near-duplicates, we randomly split PCSD by project, and then retrain and reevaluate the models on it. The new results are shown in Table VII. As seen, the performance of all models drastically reduces in this setting. Since different projects can have totally different code patterns and vocabularies, the models may have difficulty to generalize to new projects, e.g., using the source code of Django (a web framework) for training and then testing on that of Scipy (a scientific computing library). Different from Tables II and VI, we can observe from Table VII that the retrieval-based baselines do not show consistently better scores than non-retrieval-based methods, denoting that they are adversely affected by the dissimilar retrieved examples. In particular, Retrieval-Only, Rencos, HGNN and Re^{2}Com show lower scores than ast-attendgru in terms of BLEU-4(s), ROUGE-L and METEOR. It can be concluded that the retrieval-based approaches may not work ideally when there is no highly similar example (e.g., the ones come from the same project as the target code) available. By contrast, utilizing MLCS, ast-attendgru achieves the best scores on all metrics, and outperforms Rencos and Re^{2}Com by a certain gap. This suggests that although the retrieved examples are not relevant enough to the target code, MLCS is still able to effectively learn from them.

G. RQ7: Comparison With Code Category-Based Method

As mentioned in Section I, it remains unknown what classification criteria of code is optimal for code summarization task, and what kind of code summarizer is the most suitable for each code category. We bypass these issues by introducing MLCS,

TABLE VIII
SMOOTHED BLEU-4 SCORES OF DIFFERENT APPROACHES IN EACH COMMENT CATEGORY

Model	What	Why	How-to-use	How-it-is-done	Property	Others	All
DeepCom	13.53	7.32	8.57	8.23	17.66	5.60	9.62
ast-attendgru	20.24	8.24	12.60	11.48	19.68	7.02	13.78
NMT	20.38	8.40	12.23	10.86	20.48	6.60	13.56
NMT(Tf)	16.91	6.46	9.44	9.25	20.14	3.34	10.88
Rencos	16.31	9.80	11.10	10.42	17.35	7.53	12.11
HGNN	13.78	7.56	8.86	7.77	20.97	5.90	9.77
Re^{2}Com	19.75	10.89	12.91	12.14	22.96	**8.92**	14.25
ast-attendgru + MLCS	**21.56**	**11.97**	**13.76**	**13.13**	**25.09**	8.81	**15.37**

which trains exclusive model on a small cluster of examples similar to the target code snippet. In this Section, we compare MLCS to an existing code summarization approach based on comment category [67], despite that our work does not aim to solve the category problem directly. The authors of [67] try to improve code summarization performance by predicting the comment category. They first manually label 20000 <code, comment> pairs into six categories, including "what", "why", "how-to-use", "how-it-is-done", "property" and "others", and then train a classifier to infer the comment category. Based on varied performance of the code summarizers on different comment categories, they propose to pick the best model for each inferred category and generate the composite result. According to the performances on the labeled dataset, their composite approach finally performs model selection over three code summarizers ([16], [19] and [24]) trained on the same large code corpus.

To reproduce the composite approach, we test all baselines in Section IV.C on their labeled dataset, and determine which of them will be picked eventually (the approach of [67] is model-agnostic, so taking previous state-of-the-art models as candidates is reasonable). In detail, we first remove the duplicate examples that appear both in JCSD and the labeled dataset, and then apply these models, which are trained and validated on JCSD, to the latter. Afterwards, their smoothed BLEU-4 scores on each comment category are computed. We also get the results of ast-attendgru + MLCS accordingly, as shown in Table VIII. We can find that the scores of Re^{2}Com dominate other baselines in all categories except for "what" category, which is exceeded by NMT. This phenomenon is different from the results reported by Chen et al. [67]. In their experiments, DeepCom performs better on "what", "how-it-is-done" and "property" category, while Table VIII shows that DeepCom is not a strong model compared to others. The main reason could be that the baselines we choose are more recent and robust than those of [67] (e.g., ast-attendgru, Rencos and Re^{2}Com). Since the distribution of the new labeled dataset is different from JCSD, the retrieval-based baselines do not show large improvements against traditional models due to less similar retrieved examples (recall the results in Section V.F). As seen, our approach consistently outperforms the baselines in all comment categories except for "other" category. This demonstrates that MLCS is effective on different subsets of data, and could stably improve the code summarizers. According to Table VIII, we choose NMT and Re^{2}Com to produce the composite results, i.e., use NMT to generate the summary in "what" category and Re^{2}Com to generate the others.

The overall results of the two selected baselines, the composite approach and MLCS on the labeled dataset are shown in Table IX. It should be noted that for the composite approach, we skip the "category prediction" stage and directly use the ground truth label for simplicity, which assumes there is an oracle model selector and thus its scores are inflated here. We can find that the composite approach performs slightly better on BLEU metrics than NMT and Re^{2}Com, but achieves similar scores to them on ROUGE-L and METEOR. These improvements are not as large as the ones presented in [67], which show that the composite approach improves the best baseline by 2.8 points on BLEU-4. In essence, the condition for the composite approach to work best is that the predictions of the candidate models are orthogonal in different code categories, i.e., the highest scores in each category are achieved by different models, so that the advantages of various models can be combined. Unfortunately, when a candidate model dominates almost all categories, e.g., Re^{2}Com in our case, the improvements provided by the composite approach will be limited, because most predictions will come from that model. This reveals the need of more in-depth research on code classification criteria and model selection strategy. As for our approach, ast-attendgru + MLCS scores the highest in terms of all metrics in Table IX, and its performance gain is more significant than the composite approach. From Tables VIII and IX, we can find that MLCS is a flexible method for learning to summarize different kinds of codes, which relieves the burden of designing specific code classification and model selection criteria.

VI. DISCUSSION

A. Generalizing to Unseen Retrieval Corpus

As far as we know, the offline training dataset for the neural model is always used as retrieval corpus in the existing studies of retrieval-based neural code summarization [29], [34], [35]. Despite that we also followed this in above experiments, it has some potential issues. First, the generalization ability of the model may be overestimated because it has already seen and

TABLE IX
PERFORMANCE OF MLCS COMPARED TO COMPOSITE APPROACH

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
NMT	28.58	16.29	9.38	4.92	13.56	37.03	17.90
Re^{2}Com	27.84	16.82	10.30	5.94	14.25	36.56	17.72
Composite Approach	28.32	16.90	10.47	6.05	14.39	36.96	17.92
ast-attendgru + MLCS	**29.22**	**18.12**	**11.46**	**6.72**	**15.37**	**38.76**	**18.41**

TABLE X
PERFORMANCE ON UNSEEN RETRIEVAL CORPUS

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	BLEU-4(s)	ROUGE-L	METEOR
Retrieval Only	29.53 (-13.6%)	21.42 (-17.5%)	17.66 (-19.0%)	**15.01** (-19.0%)	20.23 (-16.4%)	33.01 (-12.7%)	18.15 (-13.1%)
Rencos	30.16 (-10.5%)	19.02 (-16.6%)	13.32 (-20.6%)	9.93 (-22.7%)	17.28 (-15.5%)	36.53 (-8.6%)	18.17 (-9.7%)
Re^{2}Com	31.46 (-10.5%)	21.11 (-16.4%)	15.67 (-20.8%)	12.12 (-24.1%)	19.32 (-16.3%)	38.15 (-8.2%)	19.17 (-10.3%)
NMT + MLCS	**33.39 (-7.0%)**	**23.71 (-10.3%)**	**18.25 (-13.8%)**	14.47 **(-16.7%)**	**22.01 (-10.9%)**	**40.93 (-4.9%)**	**20.33 (-7.5%)**

learned from all of the similar examples. On the other hand, this may not be in line with practical application since the database for retrieval is usually non-local and changing over time.

Based on these considerations, we construct unseen retrieval corpus that is not exposed during training for each target code to further evaluate our approach. In detail, the unseen corpus is the union of validating and testing data excluding the target example itself. It should be noted that although the testing data will pass through the model by doing so, there will be no actual model update (the meta-train phase of MLCS is performed on a copy of the meta-learner f'_θ but not on itself) so that the model will not really learn from them. The experiments are performed on PCSD for its larger validating and testing set. We compare our model to retrieval only (biLSTM in Section V.D), Rencos and Re^{2}Com. The obtained scores and their changes from the ones in Table II are presented in Table X.

The performance of all models declines on unseen retrieval corpus, including the retrieval-only approach. This is partially due to the unseen corpus is not large as the training data (44075 v.s. 66114). However, the scores and their percentages of decrease could still reflect the generalization ability of the models. Although the score decreases of Rencos are less than Retrieval Only, it performs even worse than NMT (Table II) on BLEU-4(s), ROUGE-L and METEOR. Such degradation suggests that Rencos cannot well generalize to unseen corpus. Recall that Rencos apply NMT on both target code snippet and the retrieved two. When using training data as retrieval corpus, two of its three inputs have already been well-learned, and thus it will produce overly optimistic results. Re^{2}Com performs better for its use of exemplar, but the unseen exemplars also make its scores drop a lot. NMT + MLCS still shows the best scores among the neural models. Moreover, the decreases in its scores are significantly less than others, indicating that our approach could better generalize to the examples never seen and is affected less by the quality of retrieval corpus. The main reason is that MLCS forces the model to learn the patterns of the retrieved examples during inference instead of simply encoding them.

B. Prediction of Low-frequency Words

One of the motivations to combine neural code summarization with IR is to better predict low-frequency words within the comments [29], [34], in that such words may come from the similar code snippets instead of the target code. To find out how well could MLCS tackle the low-frequency word problem, we perform a statistical analysis following [29], [34]: we first collect all correctly generated words in testing set according to the references, and then count the number of low-frequency words (with frequency of no more than 1, 2, 5, 10, 20, 50 and 100) among them using the frequencies calculated on training data. We compare our approach with Rencos, Re^{2}Com, as well as NMT, as shown in Table XI. We also count the low-frequency words in the ground-truth summaries for reference. Due to naming conventions, many identifiers in PCSD are consist of abbreviations and cannot be further split via camelCase or snack_case, so the number of low-frequency tokens in PCSD is much more than JCSD.

It seems strange that Re^{2}Com generates fewer correct low frequency tokens than NMT, e.g., frequency ≤ 20 on JCSD and ≤ 5 on PCSD, despite that it leverages the exemplars and also achieves much higher BLEU-1 scores (Table II). By contrast, Rencos performs the best among them. This is likely due to their different ways to integrate the retrieved knowledge: Re^{2}Com sums the hidden states and context vectors from different encoders according to the similarities while Rencos sums the output probabilities from the decoders. Therefore, the retrieved example could have more direct effect on generating a certain token for Rencos than Re^{2}Com. Even if MLCS does not aim at

TABLE XI
PERFORMANCE ON LOW-FREQUENCY WORDS

Dataset	Model	≤ 1	≤ 2	≤ 5	≤ 10	≤ 20	≤ 50	≤ 100
JCSD	Reference	886	1185	1797	2434	3289	4213	4747
	NMT	24	53	153	301	586	1068	1453
	Rencos	30	69	186	367	684	1185	1585
	Re²Com	17	48	129	276	564	1061	1488
	NMT + MLCS	**40**	**100**	**235**	**444**	**796**	**1354**	**1800**
PCSD	Reference	3549	4563	6430	8014	9329	10556	11,128
	NMT	142	310	716	1265	1931	2862	3408
	Rencos	209	443	942	1604	2366	3368	3921
	Re²Com	105	286	699	1318	2088	3133	3698
	NMT + MLCS	**243**	**493**	**1051**	**1746**	**2561**	**3632**	**4196**

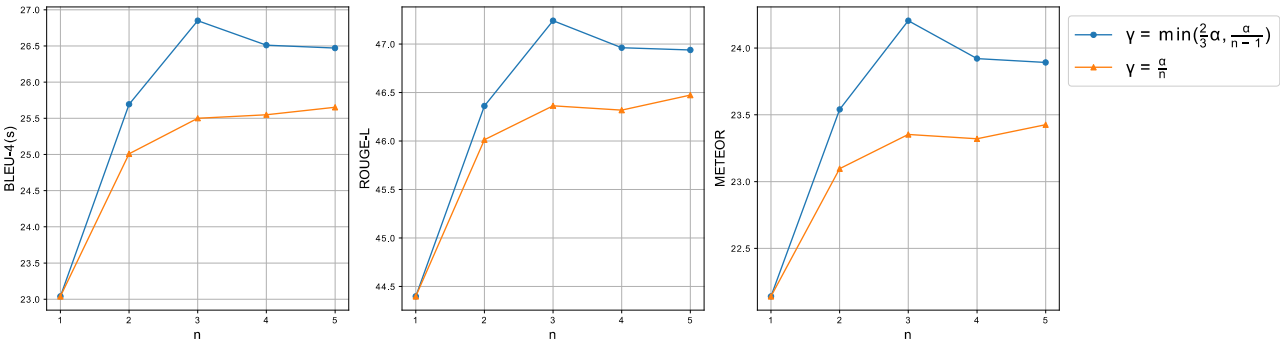


Fig. 6. Effect of adaptation step number n and the calculation of γ on performance.

the low-frequency word issue, our approach generates the most low-frequency words correctly on both datasets, with obvious gaps. This shows that by learning the mapping between similar codes and its summaries, MLCS is able to more efficiently tackle this problem without taking additional model inputs.

C. Effect of Parameter n and γ on Inference

In this section, we examine the effect of adaptation step number n and the empirical calculation of step size γ during inference. We compute the scores of NMT + MLCS on testing set with n ranges from 1 to 5 and remain other hyper-parameters unchanged. To check the effectiveness of (16), we compare it with setting $\gamma = \alpha$ / n. Fig. 6 shows the evaluation results on JCSD.

It is clear that the model performs significantly better when taking more than one adaptation step, which proves our statements in Section III.D.2: multiple steps with reduced step size could lead to better optimization. When $n > 1$, using (16) to compute γ achieves obviously higher scores, showing that letting γ slightly larger than α / n did the trick. Using either way to obtain γ, the scores can hardly be improved when n is larger than 3, so there is actually a bottleneck. In Fig. 6, the best performance is reached at $n = 3$ with (16), which is different from the value 5 in our experiments. This is because the hyper-parameters are chosen according to the BLEU-4 scores on validating set, and we get slightly higher score (19.57) with $n = 5$ than $n = 3$ (19.53). This suggests that validating via only BLEU-4 is a potential threat to validity, which will be discussed in Section VI.F. Considering the testing performance, it seems more efficient to set $n = 3$ instead.

D. Qualitative Analysis

In this section, we show output examples from different approaches on the testing sets of both JCSD and PCSD for qualitative analysis, as presented in Table XII. For each of these target codes, we show the retrieved example(s) used by MLCS for meta-training (i.e., $\mathcal{S}^i$) together with the similarity value(s) calculated via (12). Due to limited space, we only give the method signatures of the retrieved codes, and omit part of the codes in Example 4. In Table XII, the first retrieved examples are also the top-ranked ones from the retriever, so they are shared among MLCS, Rencos and Re^{2}Com.

Example 2 and 4 show the situation that the similarity of the retrieved code is low. In other words, in these cases, there is no code highly similar to the targets in the training set, which challenges the generalization ability of both retrieval and non-retrieval-based approaches. As seen, NMT + MLCS performs well compared to the baselines. In particular, it is the only model

TABLE XII
EXAMPLES OF GENERATED SUMMARIES

<table>
<tr><td colspan="4">Example 1</td></tr>
<tr><td>Source Code</td><td colspan="3"><pre>public void add(IMultiPoint pt){
  if (inProgress) {
    throw new ConcurrentModificationException("Can't add point while iterator in progress");
  }
  points.add(pt);
}</pre></td></tr>
<tr><td rowspan="2">Retrieved</td><td>sim = 0.89 (Top-1)</td><td>Signature
Summary</td><td><pre>public void add(TwoDNode dn)</pre>
add subtree rooted at given node to the result set</td></tr>
<tr><td>sim = 0.88</td><td>Signature
Summary</td><td><pre>public void add(DimensionalNode dn)</pre>
add subtree rooted at given node to the result set</td></tr>
<tr><td>Generated</td><td colspan="2">Reference
NMT
Rencos
Re²Com
NMT + MLCS</td><td>add point to the result set
adds a new point at the end of the list
adds a node to the end of this node
add subtree rooted at given node to the result set
add a multi point to the result set</td></tr>
<tr><td colspan="4">Example 2</td></tr>
<tr><td>Source Code</td><td colspan="3"><pre>def remove_duplicates(errors):
    passed = defaultdict(list)
    for error in errors:
        key = (error.linter, error.number)
        if (key in DUPLICATES):
            if (key in passed[error.lnum]):
                continue
            passed[error.lnum] = DUPLICATES[key]
        (yield error)</pre></td></tr>
<tr><td>Retrieved</td><td>sim = 0.27</td><td>Signature
Summary</td><td><pre>def flatten_errors(cfg, res, levels=None, results=None)</pre>
an example function that will turn a nested dictionary of results into a flat list</td></tr>
<tr><td>Generated</td><td colspan="2">Reference
NMT
Rencos
Re²Com
NMT + MLCS</td><td>filter duplicates from given errors list
remove all duplicates from a list of errors
remove all duplicates from a list of errors
an example function that will turn a nested dictionary of exceptions
remove duplicates from the given errors</td></tr>
<tr><td colspan="4">Example 3</td></tr>
<tr><td>Source Code</td><td colspan="3"><pre>private void stop(HttpServletRequest request,HttpServletResponse response)
throws IOException {
    PrintWriter out=response.getWriter();
    String label=request.getParameter("stop");
    Monitor mon=removeMon(label,null,DEFAULT_UNITS);
    if (mon == null) {
      out.println(ERROR + ": stop requires that an existing label is passed in. For example:
curl http://localhost:8080/jamon/Mon?stop=mylabel");
    }
   else {
      mon.stop();
      out.println(OK + ": stop successfully called on - " + mon);
    }
}</pre></td></tr>
<tr><td>Retrieved</td><td>sim = 0.98</td><td>Signature
Summary</td><td><pre>private void skip(HttpServletRequest request,HttpServletResponse response)</pre>
skip a previously started monitor if it exists</td></tr>
<tr><td rowspan="2">Generated</td><td colspan="2">Reference
NMT
Rencos
Re²Com</td><td>stop a previously started monitor if it exists
called when the user stop the user
skips monitor from the request
stops the http request</td></tr>
<tr><td colspan="2">NMT + MLCS</td><td>stops a previously started monitor if it exists</td></tr>
</table>

TABLE XII
CONTINUE

Example 4			

Source Code	`public static void loadAll(final JFrame parent, final IDebugger debugger, final IAddress offset, final int size){`
	`  checkArguments(parent, debugger, offset);`
	`  final CDumpAllWaiter waiter=new CDumpAllWaiter(debugger, offset, size);`
	`  CProgressDialog.showEndless(parent, "Loading memory" + " ...", waiter);`
	`  if (waiter.getException() != null) {`
	`  CUtilityFunctions.logException(waiter.getException());`
	`  final String innerMessage="E00078: " + "Could not load memory section";`
	`  final String innerDescription=CUtilityFunctions.createDescription(String.format(……, offset.toHexString()), new String[]{"There was a problem with the connection to the debug client."}, new String[]{"The memory data was not loaded."});`
	`  NaviErrorDialog.show(parent, innerMessage, innerDescription, waiter.getException());`
	`  }`
	`}`		

Retrieved	*sim* = 0.47	Signature Summary	`public static void showReilGraph(final CGraphWindow parent, ……)` turns a view into reil code and shows a graph that contains that reil code
Generated	Reference NMT Rencos Re²Com NMT + MLCS		loads all data of a memory section creates a new instance of this class creates a new reil for all reil nodes loads all data from a dialog loads a memory section
Example 5			

Source Code	`def get_jids(returner):`
	`  returners = salt.loader.returners(__opts__, __salt__)`
	`  return returners['{0}.get_jids'.format(returner)]()`		

Retrieved	*sim* = 0.89 (Top-1)	Signature Summary	`def get_fun(returner, fun)` return info about last time fun was called on each minion
	sim = 0.92	Signature Summary	`def get_jid(returner, jid)` return the information for a specified job id
Generated	Reference NMT Rencos Re²Com NMT + MLCS		return a list of all job ids return a list of all job id return a list of the last function called for each minion return info about last time job was called on each minion return a list of all job id
Example 6			

Source Code	`def getCraftedTextFromText(gcodeText, liftRepository=None):`
	`  if gcodec.isProcedureDoneOrFileIsEmpty(gcodeText, 'lift'):`
	`  return gcodeText`
	`  if (liftRepository == None):`
	`  liftRepository = settings.getReadRepository(LiftRepository())`
	`  if (not liftRepository.activateLift.value):`
	`  return gcodeText`
	`  return LiftSkein().getCraftedGcode(liftRepository, gcodeText)`		

Retrieved	*sim* = 0.90 (Top-1)	Signature Summary	`def getCraftedTextFromText(gcodeText, exportRepository=None)` export a gcode linear move text
	sim = 0.86	Signature Summary	`def getCraftedTextFromText(gcodeText, repository=None)` outset the preface gcode text
	sim = 0.72	Signature Summary	`def getCraftedTextFromText(gcodeText, repository=None)` inset the preface gcode text
Generated	Reference NMT Rencos Re²Com NMT + MLCS		lift the preface gcode text lift the preface gcode linear move text export a gcode linear move text export a gcode linear move text lift the preface gcode text

that generates correct summary for Example 4, where none of the baselines can extract the keywords "load memory" from line 5 and 11.

The retrieved code snippets are highly related to the target ones in Example 1 and 3, while these examples reveal that Rencos and Re^{2}Com sometimes copy too much from their extra retrieved inputs. Example 1 is a case that fails both Rencos and Re^{2}Com. Here the target code and the retrieved ones are all performing an "add" action but on different object. Their similar logic results in a high *sim* value which confuses Rencos and Re^{2}Com. By contrast, NMT encodes only the target code so it performs obviously better. The result of NMT + MLCS is even more accurate than the reference for its prediction of an extra correct token "multi". In Example 3, the retrieved code merely replaces a few tokens within the target code, but this includes the changing of the action keyword "stop" to "skip". Owing to the large *sim*, Rencos incorrectly predicts "skip". Compared with the baselines, NMT + MLCS accurately rewrites the retrieved summary, again showing that it has the merits of both retrieval and non-retrieval-based methods, as well as a better judgment on the retrieved examples.

Example 5 and 6 illustrate that using multiple retrieved results could be helpful. In Example 5, the second retrieved result is more relevant to the target code than the top-ranked one, and contains the target tokens "job id" in its summary. In Example 6, the keyword "preface" appears in the other two retrieved summaries except the top one. Since Rencos and Re^{2}Com use only the top-1 results, their outputs are biased towards them just like in Example 1, and the information in other retrieved results are overlooked. Utilizing multiple similar examples, our approach is able to produce matched summaries. Here we can also observe that Rencos and Re^{2}Com do not always perform better than the simple NMT, in spite of their much higher scores in Table II. Due to the performance of retrieval method, its top-ranked result is not always the best. For instance, the top-1 result in Example 5, which has a slightly lower *sim* value than the other. Besides, multiple similar examples are possible to provide extra useful information. These cases further explain why our approach is more stable when using various retrievers (RQ4).

In above examples, we can find that the proposed approach could better generalize to different quality of retrieved examples and perform effective information selection. In essence, existing methods such as Rencos and Re^{2}Com merge the use of similar examples and summarizing the target code into a single encoder-decoder model. Differently, MLCS divides it into two stages during inference (Algorithm 2), i.e., adapts to these examples and then generalizes to the target code, which allows the model to better capture their respective patterns.

E. Time Cost

Despite the merits of applying MLCS, we must point out that its major drawback is its high computational cost. This is a common disadvantage of optimization-based meta-learning techniques and remains an open problem [63]. For comparison, we report the average inference speed of NMT + MLCS and the baselines in Table XIII with the settings described in Section IV.B. They are tested on a CentOS server with an RTX 3090 GPU. All models are implemented using PyTorch 1.7 with CUDA 11 and cuDNN 8.0.5. It can be seen from the table that NMT is dozens of times slower after equipping MLCS. For each target code snippet, MLCS will train it an exclusive code summarizer on its similar examples. Compared to standard Seq2Seq models, this introduces two expensive operations: copying the meta-learner, and optimizing it using the similar examples. Due to these operations, the training and inference of MLCS are difficult to parallelize on the examples within the same batch so we currently process them sequentially in our implementation (during inference, this is equivalent to using a batch size of 1). Besides, each time the adaptation steps n is increased by 1, extra 40 ms is needed for more forward and backward passes of the neural network. On contrary, all the baselines can easily benefit from the highly parallel batch process provided by the deep learning library (their batch size is 64), so they are much faster. In future work, it is necessary to find a more efficient way to implement MLCS, e.g., replacing MAML with less expensive meta-learning algorithms such as Reptile [64].

TABLE XIII
INFERENCE SPEED OF DIFFERENT METHODS

Model	Inference Speed (ms / example)
DeepCom	3.0
ast-attendgru	4.3
NMT	3.6
NMT (Tf)	15.8
Rencos	6.4
HGNN	3.5
Re^{2}Com	5.1
NMT + MLCS (n = 1)	63.9
NMT + MLCS (n = 2)	105.8
NMT + MLCS (n = 3)	148.3
NMT + MLCS (n = 4)	193.3
NMT + MLCS (n = 5)	229.7

F. Threats to Validity

The major threat to the deep learning models is the quality of dataset for training and evaluation. Although we spent a lot of effort to clean and filter the datasets from [21], [36], there are still many noisy examples such as meaningless or ambiguous comments, and functions with nearly empty body. Through human evaluation, we also find that many comments contain information from the context of the target code snippet. Such comments are impossible to be generated without the knowledge from the corresponding classes. To better train and evaluate data-driven approaches, building higher quality parallel corpus is necessary in future work.

In order to perform a fair comparison, we try our best to reproduce all baselines according to their papers on our platform, and use the same data processing and inference settings for

them. This could be a threat to validity because there may be some tricks not shown in those papers and thus missed from our implementations.

BLEU-4 is considered an important metric in both NMT [42], [43], [45] and code summarization [17], [19], [20], [21], [22], [23], [24], [25], [27], [29], [34], [35]. In our experiments, we choose the hyper-parameters of MLCS and avoid overfitting according to BLEU-4 scores on validating sets. However, the increase in BLEU-4 does not always guarantee consistent improvements in other metrics so this is the third threat to validity. It remains an open question what automatic metric is most suitable for code summarization. Although the hyper-parameters of MLCS may not be optimal, our approach sill outperforms the baselines. On the other hand, since the baselines use the same settings to ours, this threat could be mitigated.

Another threat to validity is related to the retrieval process. We use a naïve method, i.e., text edit distance to filter the top-k retrieved examples. This metric is on character-level and cannot reflect semantic relevance between the code snippets, so some useful examples could still miss from the retrieved results. It is non-trivial to design a similarity metric for code summarization, and we leave it in future work. On the other hand, the size of the retrieval corpus may not be large enough to thoroughly evaluate the models. We will explore larger external dataset in the future.

VII. Related Work

A. Code Summarization

The research topic of code summarization has received increasingly more attention in recent few years. At first, code summarization relies on heuristic rules and manually-crafted templates. Sridhara et al. [11] propose Software Word Usage Model (SWUM) to represent source codes and define templates to generate short summaries for Java methods. In their following work [12], they further incorporate the parameter comments into the summaries. Also utilizing SWUM, McBurney et al. [13] predict summaries based on context information of the methods by analyzing method calls via PageRank. Moreno et al. [14] use rules to extract key information from codes, and define templates to generate different parts of the comments. Since rules and templates can hardly cover all the cases that appear in source codes, IR-based approaches appear. Using IR algorithms such as VSM and LSI, keywords of desired summaries can also be retrieved from the code corpus [2], [8]. Eddy et al. [9] further improve this by topic modeling. Movshovitz-Attias et al. [10] represent codes using topic models, i.e., Latent Dirichlet Allocation (LDA), and generate summaries via n-gram models. Wong et al. [15], [16] leverage clone detection techniques to find similar codes and use the corresponding comments for code summarization.

Most of the recent studies are data-driven and learning-based and turn to deep learning techniques, utilizing neural networks to represent source codes. Based on their different motivations, they can be broadly divided into three categories: to better learn code representations, to improve performance with the help of related tasks, and to combine the IR-based methods into neural models.

The earliest neural approach is proposed by Iyer et al. [17]. They use an RNN with attention mechanism to produce summaries directly from token embeddings of codes. Allamanis et al. [18] propose a convolutional attention network to extract features from token sequences to generate function name-like summaries. Structural information of source code has also been proved important. Hu et al. [19] customize a traversal method called SBT to serialize ASTs and then apply a standard Seq2Seq model. Alon et al. [24] represent source code as a set of AST paths and encode them via RNN, and use attention to focus on relevant paths during decoding. Wan et al. [20] employ a Tree-RNN to model AST and an RNN to model token sequence of code, and then integrate their context vectors to generate summaries. They also exploit reinforcement learning to cope with exposure bias. LeClair et al. [23] and Hu et al. [22] both combine the contexts of SBT and token sequence of code to generate summaries. Hu et al. [21] apply transfer learning to leverage the knowledge of API, and take API sequence as additional input of a Seq2Seq model. There are also increased research interests in encoding codes via GNN. Graph representation of code can be constructed using AST and lexical information [25], [27], even a combination of AST, CFG and program dependency graph (PDG) [26]. As illustrated in Section V.B, MLCS is able to boost neural code summarizers with various input forms.

Different code-related tasks are possible to share their knowledge to improve them jointly. Chen et al. [3] propose a Bimodal Variational Auto Encoder to project natural language and code into a common semantic space, which could be used for both code retrieval and summarization tasks. In [4], the authors investigate a novel perspective of code annotation for code retrieval, where a code annotation model is trained to generate code summaries that can be leveraged by a code retrieval model to better distinguish relevant codes. With dual learning, code summarization and generation model can be trained simultaneously to exploit the duality of them [5], [65]. Xie et al. [66] exploit the task of method name prediction to improve code summarization. Compared to these studies, the auxiliary task we create is code summarization itself which aims to better learn respective patterns of various examples.

Zhang et al. [29] propose a retrieval-based neural code summarization approach where the neural model takes the most similar code snippets retrieved from the training set as extra inputs. Wei et al. [34] also use retrieved examples to improve this task. Except for the similar code snippet, they further encode its paired comment (called exemplar). Similar to [34], Liu et al. [35] add the retrieved code as well as its summary as auxiliary model input, but encode source code in a different way. Specifically, they model the source code with a novel hybrid message passing GNN upon CPG for capturing both local and global structural information. Compared to these studies, we introduce a brand-new way to exploit the similar examples and prove its effectiveness against them.

The motivation of a very recent study [67] is a bit similar to this work, where the authors aim to find out which extent can comment category help to improve code summarization performance. They train a classifier to predict comment categories for source codes, and then select the most suitable models for

the inferred categories. Compared to [67], our approach aims to train exclusive models on similar examples instead of designing specific classification criteria and model selection strategy, and proved to be more flexible and effective when adapting to different types of code. It will be interesting to explore other criteria to construct the few-shot training data for MLCS.

B. Deep Meta-Learning

Meta-learning provides a solution for machine learning model to gain experience over multiple learning episodes to improve its future learning performance. Existing literature tends to categorize meta-learning methods into metric-based, model-based and optimization-based methods [50], [63]. Metric-based methods perform non-parametric learning on new tasks by comparing new inputs to labeled example inputs, and predict the labels according to their similarities. Siamese networks [68] is the first metric-based approach, followed by matching networks [53], prototypical networks [69], relation networks [70] and graph neural networks [30]. Model-based methods, also known as black-box methods, embed the knowledge of a task into an adaptive state, and make predictions based on this state. Typical models are memory-augmented neural networks [71] and meta networks [72]. In optimization-based methods, the inner-level task is solved using optimization strategies. Typical approaches including MAML [33], Meta-SGD [73] and Reptile [64].

The most common application of meta-learning is few-shot learning tasks such as few-shot image classification [30], [31], [33], [52], [53], [73] and object detection [32]. Meta-learning also has succeeded in improving sample efficiency in reinforcement learning, which forms the field of meta reinforcement learning [49], [74], [75]. This enables meta-learning to tackle more real-world problems like navigating [49], [74]. One can refer to recent survey papers [50], [63] for more systematic and comprehensive review of meta-learning. As far as we know, there is still no effort in designing meta-learning framework for code summarization.

VIII. Conclusion

In this paper, we propose the first meta-learning framework for code summarization named MLCS, which transforms the summarization of a code snippet into a few-shot learning task. The few-shot training data for the target code snippet is its similar examples obtained via a two-stage retrieval method. In our framework, a unique code summarizer is optimized for each target code snippet with the knowledge from its similar examples. According to extensive experimental results, the major take-aways from this work are outlined as:

- Utilizing MLCS, a standard Seq2Seq model is able to outperform previous state-of-the-art approaches, including typical deep-learning models and retrieval-based neural models proposed recently.
- MLCS can easily adapt to existing neural code summarizers without modifying their architecture, and could significantly improve their performance with the relative gain of up to 112.7% on BLEU-4, 23.2% on ROUGE-L, and 31.5% on METEOR.
- Compared to existing retrieval-based neural approaches, MLCS can better leverage multiple similar examples, and shows better generalization ability on various retrievers, unseen retrieval corpus and low-frequency words.
- MLCS is proved to be a flexible method for learning to summarize different kinds of codes, which relieves the burden of designing specific code classification and model selection criteria.
- We find that the retrieval-based approaches may not obtain ideal improvements, or even perform worse than the non-retrieval-based counterparts when they cannot retrieve highly similar examples from the codebase.

MLCS has shown its great potential on code summarization, while its time cost is expensive compared with common approaches. For future work, we plan to improve its implementation with faster meta-learning algorithms. The combination of meta-learning and pre-trained language models for code will also be considered in the future.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018.

[2] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in . *Proc. IEEE 17th Work. Conf. Reverse Eng.*, 2010, pp. 35–44.

[3] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *Proc. IEEE/ACM 33rd Int. Conf. Autom. Softw. Eng.*, 2018, pp. 826–831.

[4] Z. Yao, J. R. Peddamail, and H. Sun, "CoaCor: Code annotation for code retrieval with reinforcement learning," in *Proc. World Wide Web Conf.*, 2019, pp. 2203–2214.

[5] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proc. World Wide Web Conf.*, 2020, pp. 2309–2319.

[6] A. T. Nguyen and T. N. Nguyen, "Automatic categorization with deep neural network for open-source Java projects," in *Proc. IEEE 39th Int. Conf. Softw. Eng.*, 2017, pp. 164–166.

[7] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empir. Softw. Eng.*, vol. 10, no. 1, pp. 31–55, 2005.

[8] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. IEEE/ACM 32nd Int. Conf. Softw. Eng.*, 2010, pp. 223–226.

[9] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carve, "Evaluating source code summarization techniques: Replication and expansion," in *Proc. IEEE 21st Int. Conf. Prog. Comprehension*, 2013, pp. 13–22.

[10] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *Proc. 51st Annu. Meeting Assoc. Comput. Linguistics*, 2013, pp. 35–40.

[11] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. IEEE/ACM 25th Int. Conf. Autom. Softw. Eng.*, 2010, pp. 43–52.

[12] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 101–110.

[13] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for Java methods," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 103–119, Feb. 2016.

[14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proc. IEEE 21st Int. Conf. Prog. Comprehension*, 2013, pp. 23–32.

[15] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proc. IEEE/ACM 28th Int. Conf. Autom. Softw. Eng.*, 2013, pp. 562–567.

[16] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, 2015, pp. 380–389.

[17] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 2073–2083.

[18] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. 33nd Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.

[19] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. IEEE 26th Conf. Prog. Comprehension*, 2018, pp. 200–210.

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

[21] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2269–2275.

[22] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 2179–2217, 2020.

[23] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proc. IEEE 41st Int. Conf. Softw. Eng.*, 2019, pp. 795–806.

[24] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. 7th Int. Conf. Learn. Representations*, 2019.

[25] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *Proc. 7th Int. Conf. Learn. Representations*, 2019.

[26] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Automatic code summarization via multi-dimensional semantic fusing in GNN," 2020, *arXiv:2006.05405*.

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

[28] J. Zhang, M. Utiyama, E. Sumita, G. Neubig, and S. Nakamura, "Guiding neural machine translation with retrieved translation pieces," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2018, pp. 1325–1335.

[29] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1385–1397.

[30] V. G. Satorras and J. B. Estrach, "Few-shot learning with graph neural networks," in *Proc. 6th Int. Conf. Learn. Representations*, 2018.

[31] S. Qiao, C. Liu, W. Shen, and A. L. Yuille, "Few-shot image recognition by predicting parameters from activations," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 7229–7238.

[32] J. P´erez-Ru´a, X. Zhu, T. M. Hospedales, and T. Xiang, "Incremental few-shot object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 13843–13852.

[33] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1126–1135.

[34] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. IEEE/ACM 35th Int. Conf. Autom. Softw. Eng.*, 2020, pp. 349–360.

[35] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *Proc. 9th Int. Conf. Learn. Representations*, 2021.

[36] A. V. M. Barone and R. Sennrich, "A parallel corpus of Python functions and documentation strings for automated code documentation and code generation," in *Proc. 8th Int. Joint Conf. Natural Lang. Process.*, 2017, pp. 314–319.

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

[38] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Workshop Text Summarization Branches Out*, 2004, pp. 74–81.

[39] M. J. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proc. 9th Workshop Stat. Mach. Transl.*, 2014, pp. 376–380.

[40] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.

[41] K. Cho et al., "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1724–1734.

[42] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2015, pp. 1412–1421.

[43] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[44] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[45] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," in *Proc. 8th Workshop Syntax Semantics Struct. Stat. Transl.*, 2014, pp. 103–111.

[46] S. Thrun and L. Y. Pratt, "Learning to learn: Introduction and overview," in *Learning to Learn*, S. Thrun and L. Y. Pratt, eds, Berlin, Germany: Springer, 1998, pp. 3–17.

[47] J. Schmidhuber, "Evolutionary principles in self-referential learning. On learning how to learn: The meta-meta-…hook," PhD dissertation, Tech. Univ. München, München, Germany, May 1987.

[48] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, Art. no. 159.

[49] A. Nagabandi et al., "Learning to adapt in dynamic, real-world environments through meta-reinforcement learning," in *Proc. 7th Int. Conf. Learn. Representations*, 2019.

[50] M. Huisman, J. N. van Rijn, and A. Plaat, "A survey of deep meta-learning," *Artif. Intell. Rev.*, vol. 54, no. 6, pp. 4483–4541, 2021.

[51] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, M. Gatford, and A. Payne, "Okapi at TREC-4," in *Proc. 4th Text Retrieval Conf.*, 1998, p. 73.

[52] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," in *Proc. 5th Int. Conf. Learn. Representations*, 2017.

[53] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3630–3638.

[54] "Javalang," 2021. [Online]. Available: https://github.com/c2nes/javalang

[55] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[56] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, 2010, pp. 249–256.

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.

[58] B. Chen and C. Cherry, "A systematic comparison of smoothing techniques for sentence-level BLEU," in *Proc. 9th Workshop Stat. Mach. Transl.*, 2014, pp. 362–367.

[59] "NLTK," 2020. [Online]. Available: http://www.nltk.org/

[60] G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[61] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, 2017.

[62] F. Wilcoxon, S. Katti, and R. WIlcox, "Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test," *Sel. Tables Math Statist.*, vol. 1, pp. 171–259, 1970.

[63] T. M. Hospedales, A. Antoniou, P. Micaelli, and A. J. Storkey, "Meta-learning in neural networks: A survey," 2020, *arXiv:2004.05439*.

[64] A. Nichol, J. Achiam, and J. Schulman, "On first-order meta-learning algorithms," 2018, *arXiv:1803.02999*.

[65] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 6559–6569.

[66] R. Xie, W. Ye, J. Sun, and S. Zhang, "Exploiting method names to improve code summarization: A deliberation multi-task learning approach," in *Proc. IEEE/ACM 29th Int. Conf. Prog. Comprehension*, 2021, pp. 138–148.

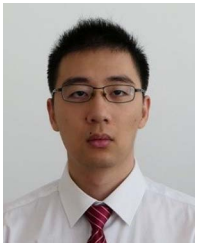
[67] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 25:1–25:29, 2021.

[68] G. Koch et al., "Siamese neural networks for one-shot image recognition," in *Proc. ICML Deep Learn. Workshop*, 2015.

[69] J. Snell, K. Swersky, and R. S. Zemel, "Prototypical networks for few-shot learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4077–4087.

[70] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales, "Learning to compare: Relation network for few-shot learning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 1199–1208.

[71] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. P. Lillicrap, "Meta-learning with memory-augmented neural networks," in *Proc. 33nd Int. Conf. Mach. Learn.*, 2016, pp. 1842–1850.
[72] T. Munkhdalai and H. Yu, "Meta networks," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 2554–2563.
[73] Z. Li, F. Zhou, F. Chen, and H. Li, "Meta-SGD: Learning to learn quickly for few shot learning," 2017, *arXiv:1707.09835*.
[74] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel, "A simple neural attentive meta-learner," in *Proc. 6th Int. Conf. Learn. Representations*, 2018.
[75] A. Gupta, R. Mendonca, Y. Liu, P. Abbeel, and S. Levine, "Meta-reinforcement learning of structured exploration strategies," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5307–5316.
[76] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[77] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4998–5007.
[78] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proc. ACM SIGPLAN Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2019, pp. 143–153.
[79] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. Thousand Oaks, CA, USA: Sage, 2012.



Ziyi Zhou is currently working toward the PhD degree in computer science with the East China University of Science and Technology. His research interests include program comprehension, natural language processing, and deep learning for software engineering.

Huiqun Yu (Senior Member, IEEE) received the BS degree in computer science from Nanjing University, in 1989, the MS degree in computer science from the East China University of Science and Technology (ECUST), in 1992, and the PhD degree in computer science from Shanghai Jiaotong University, in 1995. He is currently a professor of computer science with the Department of Computer Science and Engineering, ECUST. From 2001 to 2004, he was a visiting researcher with the School of Computer Science, Florida International University. His research interests include Big Data, software engineering, and trustworthy computing. He is a member of the ACM, a senior member of the China Computer Federation.

Guisheng Fan received the BS degree in computer science from the Anhui University of Technology, in 2003, the MS degree in computer science from the East China University of Science and Technology (ECUST), in 2006, and the PhD degree in computer science from the East China University of Science and Technology, in 2009. He is presently a research assistant with the Department of Computer Science and Engineering, East China University of Science and Technology. His research interests include formal methods for complex software system, service-oriented computing, and techniques for analysis of software architecture.

Zijie Huang received the BS and ME degrees in computer science from Shanghai Normal University, in 2016 and 2020, respectively. He is currently working toward the PhD degree in computer science with the East China University of Science and Technology. His research interests include code smell, software quality assurance, and empirical software engineering.

Kang Yang received the BS degree in information and computing science from Shanghai Ocean University, in 2017, and the PhD degree in computer science from the East China University of Science and Technology, in 2022. He is currently a postdoctoral with the Shanghai Development Center of Computer Software Technology. His research interests include software engineering and natural language processing.