UTS UNIVERSITY OF TECHNOLOGY SYDNEY

A Study on Neural-based Code Summarization in Low-resource Settings

by Yang He

Thesis submitted in fulfilment of the requirements for the degree of

Master in Analytics (Research)

under the supervision of Guandong Xu

University of Technology Sydney
Faculty of Engineering and Information Technology

06/2022

C03051: Master in Analytics (Research)
13696152 Master Thesis: Analytics
June 2022

A Study on Neural-based Code Summarization in Low-resource Settings

Yang He

School of Computer Science
Faculty of Engineering & IT
University of Technology Sydney
NSW - 2008, Australia

A Study on
Neural-based Code Summarization in Low-resource Settings

A thesis submitted in partial fulfilment of the requirements
for the degree of

Master
in
Analytics (Research)

by

Yang He

to

School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney
NSW - 2008, Australia

June 2022

Certificate of Original Authorship Template

Graduate research students are required to make a declaration of original authorship when they submit the thesis for examination and in the final bound copies. Please note, the Research Training Program (RTP) statement is for all students. The Certificate of Original Authorship must be placed within the thesis, immediately after the thesis title page.

Required wording for the certificate of original authorship

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Yang He declares that this thesis, is submitted in fulfilment of the requirements for the award of master of analytics (research), in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.
**If applicable, the above statement must be replaced with the collaborative doctoral degree statement (see below).*

**If applicable, the Indigenous Cultural and Intellectual Property (ICIP) statement must be added (see below).*

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 23/6/2021

ABSTRACT

Automated software engineering with deep learning techniques has been comprehensively explored because of breakthroughs in code representation learning. Many code intelligence approaches have been proposed for the downstream tasks of this field in the past years, contributing to significant performance progress. Code summarization has been the central research topic among these downstream tasks because of its contributions to practical applications, e.g., software development and maintenance. It remains challenging to represent code snippets and generate more accurate descriptions to summarize the functionality and semantics of programs.

Existing methods of the code summarization task have been devised to tackle real-world problems and have been successfully proven effective. However, there is little attention to its application in novel programming languages where only a few well-documented programs in these low-resource languages are available for training. According to our observation, existing approaches can only acquire poor performances in such settings, and we attribute the problem to *data-hungry* and *programming language gaps*.

Enlightened by recent pre-training methods, we propose METASUM, a meta-learning-based code summarization model, to extract prior and shared knowledge from high-resource programming language where high-quality code snippets are easily accessible and then adapt it to low-resource settings. The critical contribution of this dissertation is that we (1) give a comprehensive illustration of the development of machine-learning-based code summarization task, (2) identify a new problem of low-resource code summarization and propose a meta-learning-based model to improve over other methods by 3.18 and 1.79 BLEU points over state-of-the-art pre-trained models on Nix and Ruby datasets, respectively, and (3) introduce a machine-learning-based toolkit, NATURALCC, for fair comparison of models for the automated software engineering community.

Keywords Automated Software Engineering, Code Summarization, Low-resource Setting, Meta-Learning, Code Intelligence, NATURALCC

AUTHOR'S DECLARATION

I, *Yang He*, declare that this dissertation, submitted in partial fulfillment of the requirements for the award of Master by Research, in the *School of Computer Science*, *Faculty of Engineering and Information Technology* at the University of Technology Sydney, Australia, is wholly my work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis. This document has not been submitted for qualifications at any other academic institution.

I acknowledge that the research work in this dissertation is done under the supervision of Prof. Guandong Xu and Dr. Yao Wan at Huazhong University of Science and Technology. This dissertation is composed of my previous work in cooperation with Dr. Yao Wan and contains no material published except where the open-source NATURALCC toolkit has been released on GitHub, and some figures are adapted from submitted papers.

I have clearly stated the contribution of others to my thesis as a whole. The content of my thesis comes from my research works since the commencement of my Master by Research degree. I acknowledge that an electronic copy of my thesis must be lodged with the University Library and subject to the policy and procedures of the University of Technology Sydney.

SIGNATURE: Production Note: Signature removed prior to publication.

DATE: 23rd June, 2022

PLACE: Sydney, Australia

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my supervisor, Prof. Guandong Xu, and co-supervisor, Dr. Yulei Sui, for presenting me with a chance to study at the DSMI group at the University of Technology Sydney. They suggested that I make research plans and balance research work and daily life. Their guidance considerably helped me when I began my study in Australia.

Besides my advisors, I would like to thank my long-term cooperator Dr. Yao Wan who has been working with me. I cannot remember how often we stayed up all night to submit conference papers via remote cooperation. Under his cooperation, my programming ability is comprehensively developed via maintaining and extending the repositories.

Moreover, I would like to thank my colleagues in the research group. When I met mathematical problems, Dr. Yangyang Shu gave detailed interpretations of machine learning theories. I can acquire many experiences, including adjusting for a research position since it is easy to get depressed, especially for an international student in Australia, while facing Covid-19. On the other hand, thank Dr. Jun Yin and Dr. Qian Li for daily affairs and sharing facial masks when the pandemic is severe in Sydney.

My greatest thankfulness is owed to my parents, who covered my expense for the entire master phase and always encouraged me during the past two years when I was depressed by research work. Without their support, I would never pursue a master's degree at UTS and met friendly people in UTS. Although there are some regrets in my two-year's study, such as no full-paper has been formally accepted, your support and encouragement teach me never to give up pursuing research.

Lastly, thanks to Dr. Chandranath Adak for providing this thesis template and staff in UTS for offering me a quiet environment for study.

List of Publications

Related to the Thesis :

1. Yao Wan*, **Yang He***, Yulei Sui, Jian-Guo Zhang, Zhou Zhao, Lin Li, Guandong Xu, Philip S. Yu, *Cross-Language Knowledge Transfer for Low-Resource Code Summarization*. (*Equal contribution. Under revision for IEEE Transactions on Software Engineering)

2. Yao Wan, **Yang He**, Jian-Guo Zhang, Yulei Sui, Hai Jin, Guandong Xu, Caiming Xiong, Philip S. Yu, *NaturalCC: A Toolkit to Naturalize the Source Code Corpus*. In 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 149-153). IEEE.

Table of Contents

List of Figures

Figure **Page**

List of Tables

CHAPTER 1

INTRODUCTION

In this chapter, we give a brief introduction of the research topic in this dissertation, including the background, contributions, and organization of the dissertation.

1.1 Background

Automated software engineering aims to automatically complete software engineering tasks to meet user-given requirements, which can alleviate the pressure on developers and maintainers in software development. With breakthroughs in machine learning, automated software engineering comprehensively flourished in many downstream tasks, e.g., code summarization [1, 27, 35, 38, 46], code search [34, 45, 65, 83] and code completion [7, 13, 51, 59, 78]. Among these downstream tasks, code summarization has gained the most attention because of its high practical value and its similarity to the research work in Natural Language Processing (NLP), which is defined as the task of automatically generating a comment or docstring in natural language to describe the functionality of a given code fragment. As shown in Figure 1.1, when the yellow code is fed into a code summarization system, it is supposed to generate a docstring to reveal the code's semantics.

Although code summarization intensively follows the progress of neural machine translation, code modalities distinguish it from text generation and neural machine translation tasks. Since programming complies with strict grammar, we can explain a code snippet into various modalities representing different lexical, semantic, and syn-

```ruby
# Replace the words in the word list with the word n places after it
def n_plus(places, word_list)
  analysis = Analysis.new(self, :nouns => word_list)
  substitutor = Substitutor.new(analysis)
  substitutor.replace(:nouns).increment(places)
end
```

Figure 1.1: An example of code summarization. (Data source: CodeSearchNet dataset [45])

tactic information. As shown in Figure 1.2, for instance, we can segment a given program (the Fibonacci function) into a sequence of tokens with tokenizers, programming language lexers, or byte-pair-encoding [76], or parse it into Abstract Syntax Tree (AST) and its variants, e.g., SBT [35], or convert it into Intermediate Representation (IR) or Control Flow Graph (CFG) with LLVM [1].

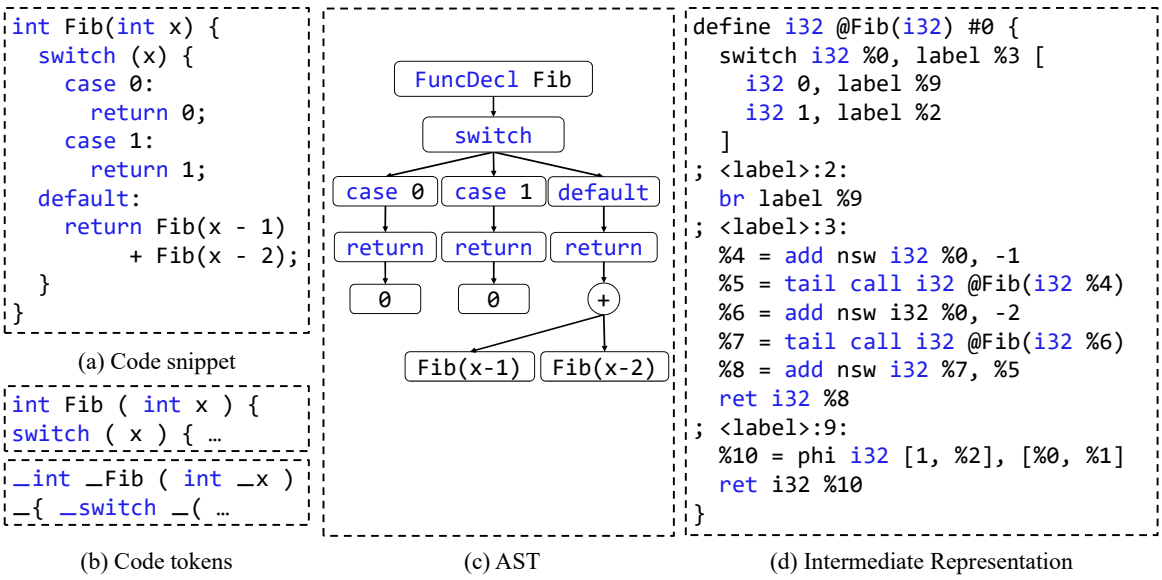


Figure 1.2: A program and its modalities. (Image adapted from [20])

Those morphologically diverse representations interpret programs from different perspectives and serve fundamental roles in code comprehension, contributing to the evolution of code summarization. At the early stage of the task, researchers [5, 46] used to face the Out-of-Vocabulary (OoV) problem in tokenization that too many rare tokens are exempted from the fixed vocabulary and, therefore, will be replaced with a unique unknown token, e.g., <*UNK*>, hindering code understanding. Unlike plain text in NLP, this problem widely exists in machine-learning-based models, and the code summarization task heavily suffers from OoV tokens because it is not uncommon in programming

[1] https://llvm.org/

languages that users-defined identifiers always follow the rules of camel-case or under-score-case, e.g., *funcName* and *func_name*. Except for an extensive vocabulary, copy mechanism [37] is an alternative method to mitigate such a problem which introduces a pointer-generator network to copy unknown words from the encoder input. Although this technique shows substantial effectiveness in some cases [2, 84], it still suffers from unseen words that only exist in ground truths, leading to an intractable performance decrease. Sennrich et al. [76] suggested segmenting (rare) words into sub-word units, enabling us to further separate user-defined tokens with a fixed-size vocabulary. Moreover, AST is another widely applied code representation in the code summarization task. Tai et al. [80] creatively proposed N-ary and Child-Sum Tree-LSTM cells, which encode structural representations of programs via merging a node embedding and that of its children. However, since structural RNNs are notorious for computational complexity, Hu et al. [43] first proposed linearizing ASTs by a structural-based traversal method, e.g., depth-first search, balancing the trade-off between structural information and high computational efficiency. On the other hand, Alon et al. [6] find code snippets sharing the same functionality but with different implementations that contain identical paths and encode programs as path modalities. Furthermore, combining code tokens with the corresponding AST [84] or CFG [38] as a hybrid representation demonstrates efficacy and better performance in program comprehension.

Categories of code summarization neural-based models Despite the diversity of code representations, existing methods' main architectures and training strategies remain consistent. According to model architectures and learning policies, we roughly separate them into three categories, i.e., RNN-based, Transformer-based, and Pre-trained. Note that in this dissertation, we concentrate more on these models can be implemented in all programming languages, including Nix.

- **RNN-based models** Early applications of code summarization models mainly are comprised of RNN-based encoders and decoders. Researchers in this phase concentrate on (1) designing code modalities for code properties and (2) incorporating machine learning techniques to improve performance. CodeNN [46] is a simple yet effective model for SQL and C# code snippets, while Tree2Seq [80] employs two Tree-LSTM cells, i.e., N-ary and Child-Sum Tree-LSTM cells, to represent the AST of a given code. DeepCom [35] leverages the SBT modality to learn structural information of code with sequential RNN cells and replace OoV tokens with their type name in AST to mitigate the OoV problem. Code2seq [6] encodes path

as code representations and generate summaries with the global attention mechanism. Unlike previous works which pay much attention to the encoder architecture, Wan et al. [84] treats decoding as a reinforcement learning problem to find optimal decoding results. Hu et al. [44] introduces an API summarization model to encode APIs of programs and initializes decoder hidden states with knowledge from source codes.

- **Transformer-based models** Transformer [81] architecture pushes code summarization into the next generation. CodeTransformer [2] evaluates three positional encodings in code summarization, i.e., absolute, learned, and relative. Their experimental results prove the hypothesis that mutually swapping operands might not influence the semantic meaning of expressions, and relative positional encoding is feasible for learning code representation.

- **Pre-trained models** CodeBERT [27] is a bi-modal pre-train model for programming language and natural language via masked language modeling and replaced token detection, enabling it to search code snippets with natural language queries. GraphCodeBERT [38] first incorporates structural code information to learn general code representations via graph-guided masked attention, focusing on relevant signals in computing self-attention weights. Furthermore, GraphCodeBERT is pre-trained with masked language modeling, edge prediction, and node alignment to use data flow. Unlike the aforementioned BERT-based models, PLBART [1] utilizes the whole Transformer architecture to learn, which uses a noising scheme to corrupt source codes and restore them by decoding. Such a pre-training strategy enables PLBART to perform better in conditional generation tasks, e.g., code translation and summarization. To our best knowledge, PLBART is a state-of-the-art general code representation learning model, and it achieved the best performance in partial downstream tasks.

1.2 Challenges

Code summarization in low-resource settings Although the code summarization task has been comprehensively discussed with various techniques and in different programming languages, its applicational setting remains rarely explored. In GitHub, there are about 297 identified programming languages. High-resource popular languages cover almost all repositories with extensive high-quality programs, while the novel and low-

resource languages are paired with little well-documented code snippets. For instance, there are 1,536,996 Java and 870,214 Python repositories on GitHub, whereas only 434 and 4,125 repositories are written in the minority languages Nix[2] and Haxe[3], respectively. To our best knowledge, these minority and novel languages catch no attention in the code summarization task, which motivates us to perform two toy experiments to interpret the facts.

Our first experiment evaluates model performances with different numbers of parallel data pairs. From Figure 1.3, we observe that a Seq2Seq model can achieve high performance in a large-scale corpus, but when the number of training samples decreases, their performance shrinks correspondingly. Moreover, this phenomenon widely appears in the neural-based models initialized from scratch according to our experimental results in Chapter 4. We attribute the problem to *data-hungry*, which presents the performance of neural-based approaches initialized from scratch and heavily relies on a large amount of well-labeled training samples, which tend to be expensive and time-consuming to collect and therefore hinder researchers from focusing their attention on these minority languages.

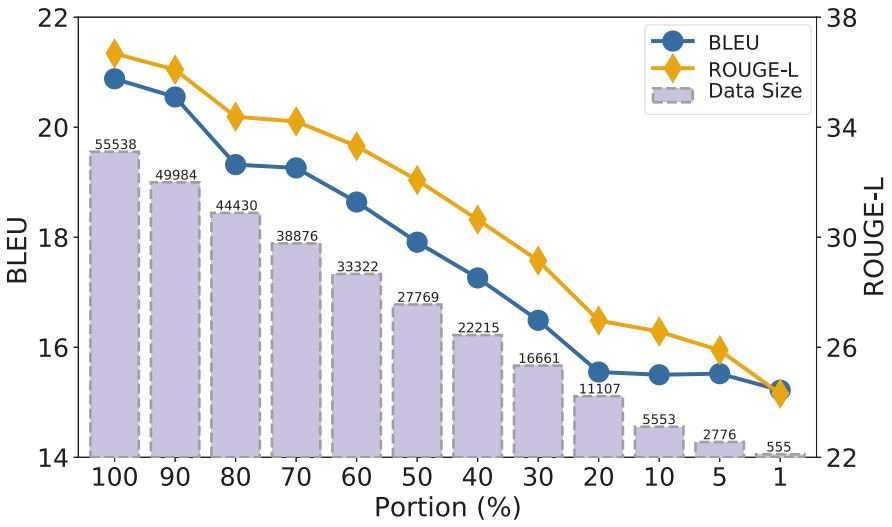


Figure 1.3: The performance of a Seq2Seq model in code summarization when varying the portion of training samples. The Seq2Seq model utilizes 2-layer Bi-LSTMs for its encoder and decoder with an attention mechanism, and the data are collected from [84].

Furthermore, since high-quality datasets of minority languages are unavailable, it brings us a question: *"Can we directly summarize a code written in a low-resource lan-*

[2]https://nix.dev/anti-patterns/language.html
[3]https://haxe.org/

guage with a pre-trained model from a large corpus?". From Table 1.2, we observe that there lies a *language gap* between programming languages (e.g., Ruby and Nix in our experiments) that guarantee high performances for models pre-trained and evaluated on the same programming language but worse performances for models pre-trained and tested on the different datasets. It might be because programmers prefer imperative features in Ruby while Nix only supports functional features, leading to syntactically in Ruby and Nix code snippets. Table 1.1, for instance, shows two implementations in Ruby and Nix programming languages. The Ruby implementation follows imperative programming, while the Nix code employs an accumulator (a fundamental property in functional programming languages) to recursively call the Fibonacci function. Last but not least, it is widely recognized that transferring prior knowledge significantly improves model performances on low-resource datasets, but extracting knowledge for syntactically different programming languages is challenging.

Table 1.1: The Fibonacci function in Ruby and Nix Implementations.

Ruby implementation

```ruby
def fibonacci(n)
        if n == 1
                1
        elseif n == 2
                1
        else
                fibonacci(n - 1) + fibonacci(n - 2)
        end
end
```

Nix implementation

```nix
fibonacci' = i: n: m:  if i == 0 then n
                 else fibonacci' (i - 1) m (n + m);
fibonacci = n: fibonacci' n 1 1;
```

Table 1.2: Experimental results of evaluating pre-trained models on different datasets. $A \rightarrow B$ denotes inference operation where a model is first pre-trained on the A dataset and tested on the B dataset. The base model is Transformer.

Method	**BLEU**	**ROUGE-L**	**METEOR**
Ruby → Ruby	15.50	17.70	6.59
Ruby → Nix	3.87	0.06	0.14
Nix → Nix	13.56	6.70	3.03

Lack of fair comparison It is universally recognized that deep-learning-based models are susceptible to many elements, e.g., random seed, implementation platform, and datasets, making it difficult to compare or seek out factors that ensure performance. To our best knowledge, there are no such toolkits or platforms for researchers in programming language and software engineering, while similar repositories in natural language processing cannot meet our specifications. For instance, there is no implementation to fast generate, read and write the modalities mentioned above.

To this end, we implemented and released NATURALCC, an open-source machine-learning-based toolkit for automated software engineering downstream tasks, enabling users to fast prototype, reproduce state-of-the-art models, or verify the correctness of algorithms.

1.3 Contributions

In summary, the contributions made in this dissertation are shown as follows:

- **The roadmap for the code summarization task** The main contribution of this dissertation is a comprehensive illustration of the code summarization task with machine learning techniques. We first list popular code modalities proposed in previous works and elaborate on the development of code summarization models.
- **METASUM** is a meta-learning-based summarization model that works in enormous corpora and low-resource settings. The experimental results reveal that our method excels baselines on two low-resource datasets, and even the amount of dataset decreases, our model is more stable than other works.
- **NATURALCC benchmark**: a comprehensive open-source toolkit for automated software engineering. With the toolkit, researchers can reproduce state-of-the-art models and explore the development of downstream tasks in related fields. This dissertation only illustrates the source codes related to the code summarization task. Complete detail of this work is given in Chapter 6.

1.4 Organization

The organization of this dissertation is shown as follows:

- A literature review of code summarization is found in Chapter 2.

- Chapter 3 illustrates the roadmap of code summarization in detail.
- Chapter 4 demonstrates our work of code summarization in low-resource settings.
- Chapter 5 presents an open-source toolkit, NATURALCC, to reproduce experiments and for a fair comparison.
- Finally, Chapter 6 shows conclusions and a summary of the results obtained in the dissertation.

CHAPTER 2

LITERATURE REVIEW

This chapter gives a brief overview of related works in the field of code summarization. The novelty of our work is not discussed in this literature review chapter.

2.1 Code Intelligence

Code intelligence automatically completes downstream tasks of automated software engineering using artificial-intelligence-based methods. Benefiting from the recent progress in deep-learning-based NLP, there are has been a resurgence of interest in code intelligence tasks, e.g., code retrieval [15, 34, 45, 65, 83], code translation [17, 36, 54], code summarization [2, 43, 46, 84], and code completion [7, 13, 51, 59, 78]. Code retrieval is to search relevant code based on the semantics meaning of natural language query or source code inputs, also known as semantics-based code search [15, 34, 45] and code-to-code search [50]. As for code translation and code summarization, although literately designed for different purposes, they utilize the encode-decoder framework to encode source programs as contextual information and decode them in another programming language or natural language. Thanks to the pre-trained model [1, 27, 38] in programming language, models of these tasks gain performance enhancement because of general code representation. More importantly, code completion automatically completes code via predicting the candidate tokens to meet user specifications. Structural representations [7, 51] have been prove effective for the task.

This dissertation concentrates on the code summarization task and its real-world

applications in low-resource programming languages. Additionally, in Chapter 5, we introduce a machine-learning-based toolkit, named NATURALCC, for code intelligence and demonstrate the necessity for the software community. We have implemented six downstream tasks and state-of-the-art models on the toolkit, including the experiments and the corresponding source codes.

2.2 Code Summarization

A key challenge of code summarization lies in code representation learning. Existing approaches learn code semantics via encoding different code modalities, e.g., sequential code tokens and AST. Allamanis et al. [5] design a convolution neural network with attention mechanism for code summarization. Iyer et al. [46] propose CodeNN by combining an LSTM-based decoder and attention mechanism to generate descriptions for SQL queries and C# code snippets as well as to search codes. Hu et al. [43] propose linearizing the ASTs for computational efficiency and replacing out-of-vocabulary words with type information for programs' structural representations. Alon et al. [6] propose to represent code by uniformly sampling paths between terminals of a given AST and have achieved a promising performance in code summarization and code caption tasks. In addition, Wan et al. [84] propose a hybrid representation approach via fusing the sequential token representation and structural AST representation. They also consider summary generation as a reinforcement learning problem to mitigate the exposure bias while decoding. Ahmad et al. [2] propose an enhanced Transformer and a relative position encoding to represent the AST of code better and copy attention to handling the out-of-vocabulary problem. Furthermore, Alexander et al. [57] and Liu et al. [62] propose to present codes as graphs (e.g., control-flow graphs and data-flow graphs) and apply Graph Neural Networks (GNNs) to encode the graphs. Although these works perform well in large-scale corpora, they still suffer from the data-hungry problem.

On the other hand, several pre-trained models are recently introduced for the software engineering community, such as CodeBERT [27], GraphCodeBERT [38] and PLBART [1]. They are pre-trained on large-scale corpora for code representation learning and can be directly applied to low-resource settings. Since CodeBERT and GraphCodeBERT are Transformer-encoder-only models, while applied to the code summarization task, they require a different Transformer decoder randomly initialized for comment generation, which makes it challenging to learn in low-resource settings. Furthermore, PLBART is based on a complete Transformer architecture and, therefore, may not suffer from

the *data-hungry* problem, but its input excludes structural information, which can be confused by identifiers with the same name at source codes.

2.3 Few-shot Learning

Another related task is low-resource neural machine translation in NLP [25, 32, 48, 52, 69, 87], focusing on translation with small or no parallel corpora. To this end, some works [32, 87] have been proposed to improve model performance via transfer learning, which leverages external knowledge from high-resource datasets to tackle problems in low-resource language corpora. Zoph et al. [87] transfer the model parameters learned from high-resource language pairs to initialize and constrain training, significantly improving model performances in low-resource languages. Instead of directly transferring the pre-trained parameters learned from a high-resource parallel dataset, Gu et al. [32] use meta-learning to learn optimal parameters as initialization, enabling fast adaptation towards low-resource language data pairs. Neubig et al. [69] propose jointly training a model on a low-resource language and a similar right-resource language to prevent over-fitting. In an extreme context where there are no parallel data pairs for training, some works [48, 52] introduce a third high-resource language as a pivot to bride them.

On the other hand, data augmentation [25, 26, 86] is an alternative method for the low-resource translation task. Inspired by work in computer vision, Fadaa et al. [25] propose a novel data augmentation approach to generate sentence pairs containing rare words. Xia et al. [86] propose a general framework for the low-resource machine translation task, injecting low-resource language words into a high-resource dataset by an induced bilingual dictionary and then editing these modified sentences with an unsupervised machine translation framework. However, data augmentation is not always suitable for NLP tasks according to [47]. Jha et al. [47] find that augmentation often hurts performance before it helps and is less effective for preferred features, which is more challenging for the model to extract from inputs.

Similar to some works above, instead of augmenting training samples with self-defined rules which may conflict with programming language grammars, our method leverage a more stable method, meta-learning, to extract and transfer external knowledge for low-resource code summarization task.

2.4 Pre-trained Models

Previous related works widely acknowledged that using pre-trained models, namely external knowledge, can considerably improve model performance, accelerate training, and tackle tasks in low-resource settings. Enlightened by this idea, Devlin et al. [22] creatively propose BERT, a Transformer [81] encoder only model, which learns textual representation via masked language modeling and next sentence prediction, and start a new pre-training scheme, i.e., pre-training over-parameterized models on extensive corpus and then fine-tuning it on downstream tasks. Based on BERT, Liu et al. [63] introduce using dynamic masking language modeling to learn representation for better performance on downstream tasks. Lewis et al. [58] present BART, a transformer-based language model, pre-trained with noising schemes and achieves high performance in generation tasks. Liu et al. [61] extend BART in multilingual scenarios and propose mBART, which can translate sentences between natural languages.

Due to the immense success of pre-training methods in NLP, pre-trained models for software engineering communities have also benefited from such an idea. Lachaux and Roziere etal. [54] proposed CodeTrans, which is pre-trained in unsupervised learning and translates programs between Java, Python, and C++ languages. Feng et al. [27] propose CodeBERT, the first pre-trained model for natural language and programming language, to learn representation via masked language modeling and replace token detection objectives. With CodeBERT as an initial model, Jung [49] proposes CommitBERT on code modification for commit message generation. Unlike CodeBERT, Guo et al. [38] propose GraphCodeBERT by creatively devising graph-guided masked attention to leverage code structural information for better code understanding. In contrast to pre-train BERT, Ahmad et al. [1] recently proposed PLBART, which pre-trains the mBART for code representation learning and outperforms other BERT-based methods in the code generation task. In this dissertation, we employed those pre-trained models as initialization for our work and then continuously fine-tuned parameters with meta-learning techniques and adapted them to target languages for code summarization evaluation.

2.5 Meta-Learning

Meta-learning, also known as learning to learn, is a task of designing models that can rapidly learn knowledge or adapt to new application scenarios with limited data

samples. It has been widely applied in computer vision [28, 53, 75], natural language processing [33, 68] and manipulation [23]. This field's research mainly includes metric-based, model-based, and optimization-based meta-learning. Metric-based meta-learning learns a distance function between data points to classify instances via comparing them to the K-labeled examples. Koch et al. [53] employ the Siamese neural network to verify whether two images belong to the same class. Vinyals et al. [82] propose the matching networks functioning as a weighted nearest-neighbor classifier to tackle the one-shot problem. Model-based meta-learning models, e.g., Neural Turing Machines, use external memory to store knowledge. Santoro et al. [75] use proposed addressing mechanisms to assign attention weights to memory vectors. Optimization-based methods learn via optimizing network parameters for fast adaption. Ravi et al. use an LSTM-based meta-learner model to learn the exact optimization algorithm. Finn et al. [28] propose Model-Agnostic Meta-Learning (MAML) model to learn the optimal parameters as initialization where few gradient updates will lead to fast learning on a new task. For simplified and cheaper implementation, First Order MAML (FOMAML) [12] is proposed via omitting second derivatives. Nichol et al. [70] propose Reptile, a relatively simple meta-learning optimization algorithm, which updates parameters with accumulated gradients from all tasks. As for meta-learning in NLP, Gu et al. [33] propose MetaNMT, a MAML-based translation model, which translates text between eighteen languages. Madotto et al. [68] extend MAML to personalized dialogue learning without using persona descriptions. In this dissertation, we apply MAML and FOMAML to extract common knowledge from high-resource programming languages and then adapt it to low-resource programming languages datasets for the code summarization task.

Unlike transfer learning, which extracts prior knowledge by direct gradient-based computation, meta-learning, or more specifically, in our experiments, model-agnostic meta-learning extracts knowledge by computing expectation on various domains, enabling it to consider how to acquire sensitive parameters for all programming languages. Therefore, once it is adapted to an unseen target domain, the distilled knowledge from the meta-learning strategy still works even with a vast syntactical gap between programming languages. To our best knowledge, this is the first work about meta-learning for low-resource programming language in the code summarization task.

CHAPTER 3

Roadmap of Code Summarization with Machine Learning

With the progress of code intelligence and natural language processing, code summarization, which aims to generate natural language texts to describe the functionality or semantics of a code fragment, has become increasingly crucial for program comprehension and software development. Since machine-learning-based code summarization has flourished for years, a detailed illustration of the roadmap of the task is beneficial for us to understand. This chapter will first elaborate on how to represent source code into different modalities and their advantages and disadvantages. Then, we will learn some preliminaries about neural-based models and milestone models in the code summarization task with mainstream evaluation metrics.

3.1 Preliminaries

3.1.1 Tokenization

There is no ambiguity in code tokenization because programming languages follow strict grammar. Employing programming language grammar or lexer for source code segmentation has proven effective in the code summarization task.

Out-of-Vocabulary (OoV) Problem Given that the size of a vocabulary is limited, out-of-vocabulary tokens are replaced by a unique unknown token, e.g., <*UNK*>. It may work effectively in NLP because these tokens are scarce. However, it is not uncommon in source code that programmers tend to define identifiers with camel-case or under-score-case, e.g., *funcName* and *func_name*, which makes it challenging to build a fixed vocabulary that can cover almost tokens. As a result, a source input with various <UNK> will be fed into models and predict meaningless comments.

Copy Mechanism [37] A leading solution for the OoV problem is a copy mechanism comprising a pointer generator network to predicate tokens conditional on vocabulary and OoV word distributions. The network functions as a switcher to predict a word from vocabulary or copy an OoV word from the input sequence. Let P_{vocab} be the probability distribution over the fixed vocabulary, and P_{gen} denotes the probability distribution over OoV words. We have the final prediction distribution:

$$P(w) = p_{gen} P_{vocab}(w) + (1 - p_{gen}) \sum_{input} a_i^t \tag{3.1}$$

where p_{gen} is the trade-off between two probability distributions, w is the predicated word and a_i^t denotes probability distribution at the time step t over the input sequence. Intuitively, if w is an out-of-vocabulary word, then $P_{vocab}(w)$ is zero, and the model will predicate a word from source input.

Byte-Pair-Encoding (BPE) Although copy mechanism partially addresses the OoV problem by copying words from source input, it still suffers from a situation where ground-truth contains unseen words that do not exist in the vocabulary or source input. Byte pair encoding is proposed to tackle such a problem by segmenting words into smaller units. As an example, consider an OoV word such as *lower* which will be separated into *low* and *er* with BPE tokenizer.

3.1.2 Code Modalities

Unlike plain natural language text, a code can be parsed into many modalities for different purposes. In this sub-section, we will elaborate on some well-known modalities with a code example shown in the Listing 3.1.

Listing 3.1: A Java code example from [34].

```
1 public String extractFor(Integer id){
2     LOG.debug("Extracting method with ID:{}", id);
3     return requests.remove(id);
4 }
```

Code token We treat source code as plain text for this modality and separate it with previously described tokenizers. We usually employ a lexer to tokenize code and remove comments hidden in code snippets to acquire high-quality code tokens. Moreover, since the user-defined identifiers usually follow camel-case or under-score-case, some datasets choose to separate identifiers into sub-tokens by ignoring joint symbols if they have, e.g., "*extractFor*" → "*extract*" and "*For*". Similarly, employing BPE to segment identifiers is an alternative method of code tokenization without worrying about the OoV problem.

AST It is commonly acknowledged that AST includes semantics and syntactic information. Although code token is simple but effective, structural information of codes will be ignored, resulting in low-quality comment generation. Therefore, we parse source code into AST and tokenize its leaf nodes in the same way as code tokens because identifiers are stored in these nodes. While training with ASTs, neural-based models [80] treat it as a directed graph and merge the entire AST information into its root node.

SBT Since structural networks, e.g., TreeLSTM, are notorious for computational complexity, linearizing structural modalities for sequential RNNs is appealing for the code summarization task, which can considerably accelerate training time. Hu et al. [43] proposed to convert AST into specially formatted sequences by traversing it, which guarantees a good balance between structural information loss and computational efficiency. Moreover, the SBT modality replaces the out-of-vocabulary tokens with their type information in ASTs instead of the meaningless word for the OoV problem. Figure 3.1 shows AST and SBT modalities of the code example in Listing 3.1.

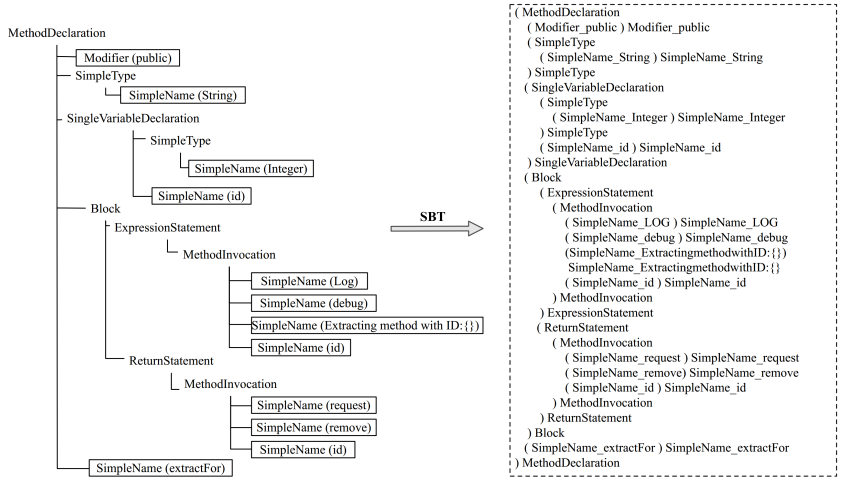


Figure 3.1: The AST and SBT modalities of the code example 3.1. (Image source: [43])

Path Alon et al. [6] argue that a code snippet can be represented as a set of compositional paths over its AST, where each path is compressed to a fixed vector to leverage the syntactic structure of programming languages. In an AST, its leaf nodes are called *terminals* which usually refer to user-defined identifiers, and its non-leaf nodes are called *non-terminals* representing structural information of the language, e.g., declarations, expressions, and statements.

The path modality is defined as a route between two *terminals* of a given AST. Since an AST can contain enormous paths, we only sample k paths as the path representation of a code snippet. Formally, let $\mathbf{v} = (v_1, .., v_{|\mathbf{v}|})$ be a set of terminals in an AST. At each training step, we unifromly sample k paths: $(v_1^1, v_2^1, ..., v_{l_1}^1), ..., (v_1^k, v_2^k, ..., v_{l_k}^k)$, where l_i denotes the length of j-th path.

3.1.3 RNN cells

Recurrent neural networks (RNNs) are widely used to encode contextual information of code input at the early stage of code summarization. According to their application scenarios, the basic unit of RNNs can be separated into two categories, i.e., Sequential RNN cells and Structural RNN cells. Sequential RNN cells mainly encode code modalities that can be represented in a linear format, e.g., code tokens, SBT, and the intermediate part of Path modality, and structural RNN cells handle modalities of structural information, e.g., AST and its variants.

Sequential RNN cells RNNs process linear input sequences of arbitrary length by updating hidden state h_t with a transition function step by step. At each time step t, the

neural network receives a input vector x_t and its previous hidden state h_{t-1} to calculate the new hidden state h_t. A vanilla RNN cell is defined as follows:

$$h_t = tanh(Wx_t + Uh_{t-1} + b) \tag{3.2}$$

where W, U, and b are the network parameters. However, vanilla RNNs meet a problem with exploding or vanishing gradients, making it challenging to learn long-distance correlations because gradients can grow or decay exponentially in a long sequence.

LSTM is devised to address the problem above via introducing a memory cell to preserve states at each time step. Although massive LSTM architecture variants have been proposed, the architecture plays a dominant position in LSTM-based models. The LSTM transition equations are defined as the following:

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}), \tag{3.3}$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}), \tag{3.4}$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}), \tag{3.5}$$

$$u_t = tanh(W^{u}x_t + U^{(o)}h_{t-1} + b^{(o)}), \tag{3.6}$$

$$c_t = i_t \odot u_t + f_t \odot c_{t-1}, \tag{3.7}$$

$$h_t = o_t \odot tanh(c_t), \tag{3.8}$$

where x_t denotes the input at the current time step t, σ denotes the *sigmoid* function and $\odot$ is element-wise multiplication, Intuitively, Equation (3.4) controls the extent of previous memory information and Equation (3.7) updates current memory information c_t upon new time step information i_t and u_t, and the previous one c_{t-1}.

Structural RNN cells Since the above RNNs only receive linear sequence, [80] proposed two Tree LSTM cells, i.e., Child-Sum and N-ary, to merge information from a tree structural input.

Given an AST, let $C(j)$ denote the set of children nodes of node j. The Child-Sum

Tree LSTM transition equations are defined as follows:

$$\tilde{h}_j = \sum_{k \in C(j)} h_j, \tag{3.9}$$

$$i_j = \sigma(W^{(i)} x_j + U^{(i)} \tilde{h}_j + b^{(i)}), \tag{3.10}$$

$$f_{jk} = \sigma(W^{(f)} x_j + U^{(f)} h_k + b^{(f)}), \tag{3.11}$$

$$o_j = \sigma(W^{(o)} x_j + U^{(o)} \tilde{h}_j + b^{(o)}), \tag{3.12}$$

$$u_j = tanh(W^u x_j + U^{(o)} \tilde{h}_j + b^{(u)}), \tag{3.13}$$

$$c_j = i_j \odot u_j + f_{jk} \odot c_k, \tag{3.14}$$

$$h_j = o_j \odot tanh(c_j), \tag{3.15}$$

Intuitively, Equation (3.14) updates memory information c_j of node j upon its new information i_j and u_j, and memory information set $\{c_k\}$ from its children nodes.

For N-ary Tree LSTM cell, it indexes a node's children from 1 to L but only incorporates information from the first N nodes. Its transition equations are defined as follows:

$$i_j = \sigma(W^{(i)} x_j + \sum_{l=1}^{N} U_l^{(i)} h_{jl} + b^{(i)}), \tag{3.16}$$

$$f_{jk} = \sigma(W^{(f)} x_j + \sum_{l=1}^{N} U_{kl}^{(f)} h_{jl} + b^{(f)}), \tag{3.17}$$

$$o_j = \sigma(W^{(o)} x_j + \sum_{l=1}^{N} U_l^{(o)} h_{jl} + b^{(o)}), \tag{3.18}$$

$$u_j = \sigma(W^{(u)} x_j + \sum_{l=1}^{N} U_l^{(u)} h_{jl} + b^{(u)}), \tag{3.19}$$

$$c_j = i_j \odot u_j + \sum_{l=1}^{N} f_{jl} \odot c_{jl}, \tag{3.20}$$

$$h_j = o_j \odot tanh(c_j), \tag{3.21}$$

3.1.4 Attention Mechanism

Global Attention In a generation architecture, we formulate each conditional probability as follows:

$$p(y_t | y_{1:t-1}, \mathbf{x}) = g(y_{t-1}, s_t, c_t) \tag{3.22}$$

where $\mathbf{x} = x_{1:|\mathbf{x}|}$ and $\mathbf{y} = y_{1:|\mathbf{y}|}$ are source sequence and target sequence, respectively, g is a network to compute conditional probability $p(y_t | y_{1:t-1}, \mathbf{x})$ at the decoding time step t, s_t denotes history information and c_t denotes context vector.

The history information s_t denotes accumulated knowledge of the target sub-sequence $y_{1:|\mathbf{y}|}$. In an RNNs based decoder f, the hidden state s_t is computed by

$$s_t = f(s_{t-1}, y_{t-1}, c_t) \tag{3.23}$$

The context vector c_t depends on a sequence of annotations $(h_1, ..., h_{|\mathbf{x}|})$ to which an encoder maps the source sequence. Each annotation h_1 incorporates comprehensive contextual information of the source sequence with a different concentration on the surrounding words of t-th words. The context vector c_t is computed as the following:

$$c_t = \sum_{j=1}^{|\mathbf{x}|} \alpha_{tj} h_j \tag{3.24}$$

where α_{tj} is the weight coefficient of annotation h_j, computed by

$$\alpha_{tj} = \frac{exp(e_{tj})}{\sum_{k=1}^{|\mathbf{x}|} exp(e_{tk})} \tag{3.25}$$

where a is a alignment model [9] to measure the weight matrix $e_{tj} = a(s_{t-1,h_j})$ presenting how well the source tokens around position j and the t-th target token match.

Self Attention [81] In each attention head, the input vectors, $\mathbf{x} = (x_1, ..., x_n)$ where $x_i \in \mathscr{R}^{d_model}$, are transformed into the output vectors $\mathbf{o} = (o_1, ..., o_n)$ as follows:

$$o_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V), \tag{3.26}$$

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{n} exp(e_{ik})}, \tag{3.27}$$

$$e_{ij} = \frac{(x_i W^Q)(x_j W^V)^T}{\sqrt{d_k}}, \tag{3.28}$$

where $W^Q, W^K, W^V \in \mathscr{R}^{d_{model} \times d_k}$ are the network parameters. These parameters are unique per layer and attention head.

3.1.5 Positional Encoding

RNN-based models learn input sequences by time steps, enabling RNN cells to remember the orders between them. However, Transformer-based models contain no recurrence and no convolution and, therefore, Positional Encoding (PE) was proposed to represent sequential information that is added to the input embeddings of encoder and decoder. This dissertation only introduces two PE schemes applied in code summarization

tasks. Absolute positional encoding [81], a.k.a., sinusoid positional encoding, employs sine and cosine functions for even and odd positions, respectively, and is computed as follows:

$$PE_{pos,2i} = sin(pos/10000^{2i/d_{model}}) \tag{3.29}$$

$$PE_{pos,2i+1} = cos(pos/10000^{2i/d_{model}}) \tag{3.30}$$

where pos is the position and i is sequential order. Additionally, learned positional encoding [30] uses a word embedding layer to automatically learn sequential representation and produce nearly identical results as absolute positional encoding.

3.1.6 Encoder-Decoder Framework

Existing code summarization approaches follow the encoder-decoder framework, successfully adopted in the text generation [73] and neural machine translation [67] tasks.

Given a code-comment pair $<\mathbf{x},\mathbf{y}>=<(x_1,x_2,...,x_{|\mathbf{x}|}),(y_1,y_2,...,y_{|\mathbf{y}|})>$, the models following the framework first encode the source code $\mathbf{x}$ into a sequence of continuous representations $\mathbf{z} = (z_1,...,z_{|\mathbf{x}|})$ and decode the target comments conditional on the sequence $\mathbf{z}$. While decoding, the probability of the next target tokencan be factorized as:

$$p(\hat{y}_t|\mathbf{x}) = \prod p(\hat{y}_t|\hat{y}_{<t},\mathbf{x}) \tag{3.31}$$

In attention-based models [19, 42], a contextual vector c_t is computed by attention mechanism [66] and the encoder representations $\mathbf{z}$). The models predict the target token $\hat{y}_t$ conditional on the contextual vector c_t and the decoding state h_t:

$$p(\hat{y}_t|\hat{y}_{<t},\mathbf{x}) = \text{Softmax}(f(c_t,h_t)) \tag{3.32}$$

where f denotes a linear network to map decoder output into vocabulary probability with a *softmax* activation.

The objective of the encoder-decoder-framework, for instance, with cross-entropy criterion, is defined as follows:

$$\mathscr{L} = \text{CrossEntropty}(\mathbf{y},\hat{\mathbf{y}}) = -\frac{1}{|\mathbf{y}|}\sum_{t=1}^{|\mathbf{y}|}\sum_{k=1}^{|\mathscr{V}|}\mathbb{1}\{y_t = k\}\log(p(\hat{y}_t = k)) \tag{3.33}$$

where $|\mathscr{V}|$ is the vocabulary size of the target programming language, y_t is the t-th target token, and $\mathbb{1}\{\cdot\}$ is the indicator function that represents the one-hot label.

3.2 Code Summarization Models

This section will elaborate on some well-known baseline models in code summarization. According to model architecture and training strategies, we separate them into three categories, i.e., RNN-based, Transformer-based, and Pre-training.

3.2.1 RNN-based Models

At the early stage of the code summarization task, models in this category employ RNN cells for encoder and decoder and an extra attention mechanism.

Code-NN is an early neural-based model in the code summarization task. Its structure comprises an embedding layer as an encoder and a one-layer vanilla LSTM as a decoder, and it uses global attention to decode target comments with the encoder representations.

DeepCom proposes Structure-based Traversal (SBT), a new code modality to learn structural code representation, and replaces out-of-vocabulary tokens with their type information in ASTs for the OoV problem. Its architecture follows a classic *Encoder-Decoder-Attention* paradigm.

BiLSTM In our experiments, BiLSTM employs two-layer bi-directional LSTMs for encoder and decoder and global attention to aligning prediction with source input. The bidirectional LSTM is defined as follows:

$$\overrightarrow{h_t} = \text{LSTM}(\mathbf{e}(x_t), \overrightarrow{h_{t-1}}) \tag{3.34}$$

$$\overleftarrow{h_t} = \text{LSTM}(\mathbf{e}(x_t), \overleftarrow{h_{t-1}}) \tag{3.35}$$

where $t = 1, \ldots, |x|$, and $e(x_t)$ denotes representation from the word embedding layer $\mathbf{e}$. We concatenate the last hidden state of vectors as the final representation of a bidirectional LSTM, i.e., $h_t = \overrightarrow{h_{|\mathbf{x}|}} \oplus \overleftarrow{h_1}$, where $\oplus$ denotes the concatenation operation.

To our best knowledge, the layer number of LSTM-based encoder and decoder can bring better performance, and bi-directional LSTMs excel those models with single directional LSTMs.

Tree2Seq [80] There are two Tree-LSTMs encoders, i.e., Child-Sum Tree-LSTM encoder and N-ary Tree-LSTM encoder, encoding the structural information of a given

AST. Additionally, they use the same global attention mechanism and decoder structure to predicate target tokens.

RL+Hybrid2Seq[84] It employs a vanilla LSTM and a Tree-LSTM to encode token representations and code structural representations, respectively but decodes with reinforcement learning. Unlike the conventional encoder-decoder framework, which learns via maximizing the likelihood of predicted words with ground-truth sequences, reinforcement learning optimizes networks based on evaluation metrics, e.g., BLEU for code summarization. RL+Hybrid2Seq introduces a critic network to approximate the value of generated action at time step t. Given a policy π sampled from probability distribution during decoding, the value function V^{π} is defined as the prediction of total reward from the state s_t under the policy π. Formally, the reward function is defined as follows:

$$V^{\pi}(s_t) = \mathbb{E}_{s_{t+1:T}, \mathbf{y}_{t:T}}[\sum_{l=0}^{T-t} r_{t+l} | \mathbf{y}_{t+1:T}, \mathbf{h}], \tag{3.36}$$

where $T = |\mathbf{y}|$ is the max step of decoding; $\mathbf{h}$ is the representation of code snippet. Additionally, the reward r_t is defined as follows:

$$r_t = \begin{cases} 0, & \text{if } t < T \\ \text{BLEU}, & \text{otherwise} t = T \text{ or } EOS \end{cases} \tag{3.37}$$

Mathematically, Wan et al. [84] optimize the critic network via the following loss function:

$$\mathscr{L}(\phi) = \frac{1}{2} ||V^{\pi}(s_t) - V^{\pi}_{\phi}(s_t)||^2, \tag{3.38}$$

where ϕ denotes the parameters of critic network, $V^{\pi}(s_t)$ denotes the ground-truth value and $V^{\pi}_{\phi}(s_t)$ denotes the predication value of critic network.

Code2Seq [6] encodes each path separately using a bi-directional LSTM layer and captures the compositional nature of terminal tokens with a sub-token embedding layer. Given a set of AST paths $\{x_1, ..., x_k\}$ where $x_i = v^i_1 v^i_2 ... v^i_{l_i}$, Code2Seq aims to encode the path set into a sequence of combined representations $(z_1, ..., z_k)$.

For path representation, Alon et al. [6] represent each node using a word embedding layer E^{nodes} and then encode the entire sequence with a bi-directional LSTM:

$$\overrightarrow{\mathbf{h}} = \text{LSTM}(E^{nodes}(v_1), ..., E^{nodes}(v_l)), \tag{3.39}$$

$$\overleftarrow{\mathbf{h}} = \text{LSTM}(E^{nodes}(v_l), ..., E^{nodes}(v_1)), \tag{3.40}$$

$$\mathbf{e}_{path}(v_1 ... v_l) = \overrightarrow{\mathbf{h}}_l \oplus \overleftarrow{\mathbf{h}}_1, \tag{3.41}$$

Token representation is defined as the sum of sub-token representations using the embedding layer $E^{subtokens}$:

$$\mathbf{e}_{token}(w) = \sum_{s \in split(w)} E^{subtokens}(s) \tag{3.42}$$

where $split$ denotes a tokenizer for user-defined identifiers.

To represent a given path $x_i = v_1^i v_2^i ... v_{l_i}^i$, Alon et al. [6] concatenate the path's representation with the terminals representations followed by a linear network W paired with $tanh$ function:

$$z_i = \tanh(W(\mathbf{e}_{path} \oplus \mathbf{e}_{token}(v_1) \oplus \mathbf{e}_{token}(v_l))) \tag{3.43}$$

To provide the decoder with an initial state, they average the combined representations $(z_1, ..., z_k)$ of a given AST:

$$h_0 = \frac{1}{k} m_{i=1}^{k} z_i \tag{3.44}$$

3.2.2 Transformer-based Models

Transformer[81] It employs an encoder-decoder framework consisting of stacked encode and decoder layers. Encoder layers comprise two sub-layers: a multi-head self-attention layer followed by a position-wise fully connected feed-forward layer. Decoder layers consist of three sub-layers: a multi-head self-attention layer followed by an encoder-decoder self-attention layer and a position-wise fully connected feed-forward layer. It employs residual connections [39] around each of the sub-layers, followed by layer normalization [8].

CodeTransformer[2] employs the relation-aware self-attention [77] mechanism to acquire relative position representations because program semantics does not rely on the absolute order of its code tokens. Instead, for some operators meeting commutative law, swapping the order of operands does not affect the correctness of an algorithm. For instance, the expressions $a + b$ and $b + a$ share the same semantic meaning. Given the pairwise relationships between input elements, Shaw et al. [77] modify the self-attention mechanism as follows:

$$o_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V), \tag{3.45}$$

$$e_{ij} = \frac{(x_i W^Q)(x_j W^V)^T + a_{ij}^K}{\sqrt{d_k}}, \tag{3.46}$$

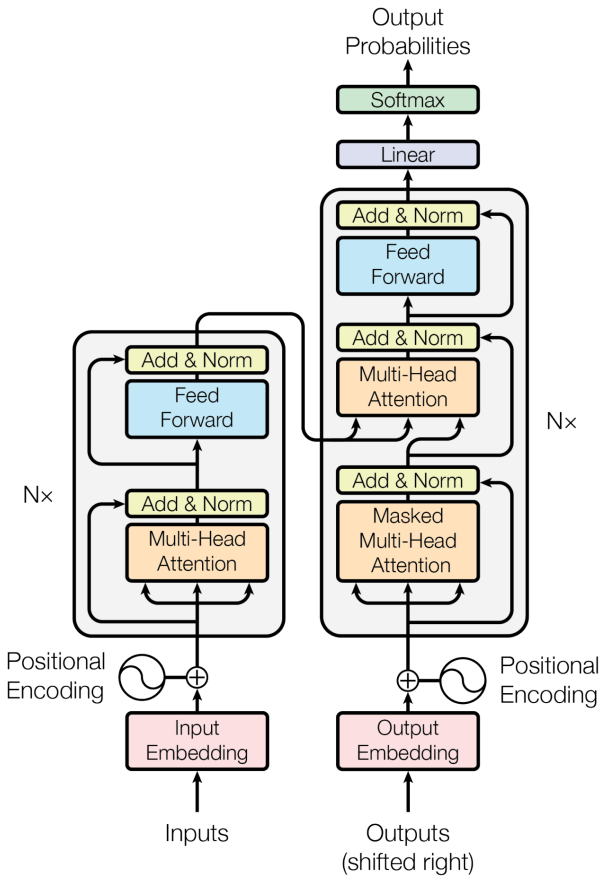


Figure 3.2: The Transformer architecture. (Image source: [81])

where a_{ij}^V and a_{ij}^K are relative positional representations between i-th and j-th positions. Furthermore, Shaw et al. [77] hypothesize that useful relative position representations are not beyond a certain distance and, therefore, the adapted mechanism clips the maximum relative position to a maximum width k.

$$a_{ij}^K = w_{j-i,k}^K \tag{3.47}$$

$$a_{ij}^V = w_{j-i,k}^V \tag{3.48}$$

$$clip(x,k) = min(|x|,k) \tag{3.49}$$

where $w_K = (w_{-k}^K, ..., w_k^K)$ and $w_V = (w_{-k}^V, ..., w_k^V)$ are word embedding layers denoting relative position representations. Note that Equation 3.49 in CodeTransformer is different from Shaw et al. [77], i.e., $clip(x,k) = max(-k, min(k,x))$, because Ahmad et al. [2] want to ignore the *directional* information [3].

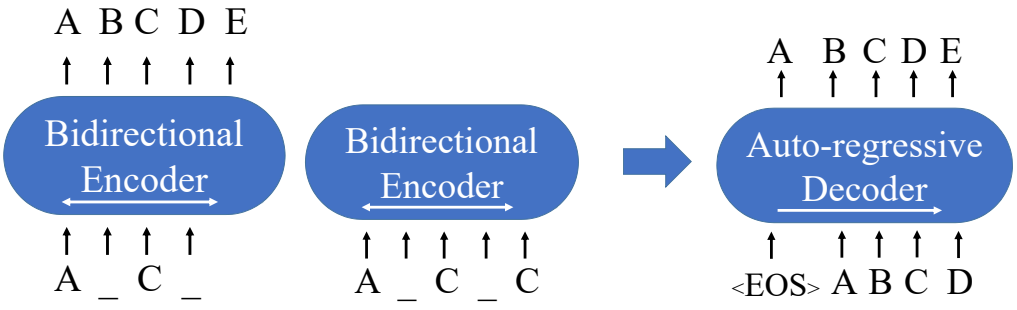
3.2.3 Pre-training Models

CodeBERT [27] is a Transformer encoder model pre-trained on source code paired with natural language comments via masked language modeling and replaced token detection objectives.

GraphCodeBERT is the first pre-trained BERT-based model that leverages program structural information for code presentation learning. The model tasks source code paired with comment and the corresponding data flow as input and is pre-trained using masked language modeling, data flow edge prediction, and variable alignment between code and its data flow.

PLBART is a denoising auto-encoder pre-trained on a large-collection Java and Python corpus and natural language descriptions of codes via denoising schemes, i.e., token masking, token deletion, and token infilling.

Figure 3.3 shows the architectures for BERT-based models and BART-based models. Bert-based models such as CodeBERT and GraphCodeBERT employ a bidirectional encoder to learn contextual representations via masked language modeling and replaced token detection objectives. BART-based models, e.g., simultaneously learn code representations via encoding corrupted input with a directional encoder and learn to predict with an additional auto-regressive decoder in which only previous states are available for the following code generation.



(a) BERT. (b) BART.

Figure 3.3: The architectures of BERT-based (3.3a) and BART-based (3.3b) pre-trained models.

3.2.3.1 Evaluation Metrics

We use three automatic evaluation metrics, i.e., BLEU [72], ROUGE [60] and METEOR [10], to evaluate the quality of generated comments. In this dissertation, we adopt the BLEU, ROUGE-L, and METEOR metrics.

BLEU [72] is a classical evaluation metric that was first designed to assess the quality of translated sentences in machine translation. It measures the n-gram precision on the reference sentences, with a penalty for short sentences. Formally, the BLEU is calculated as follows:

$$\text{BLEU} = \text{BP} * \exp\left(\sum_{n=1}^{N} \alpha_n \log p_n\right), \tag{3.50}$$

where $N = 1,2,3,4$, α_n is the positive coefficient for each gram, and p_n is the percentage of n-gram sub-sequences in both the candidate and reference sentences simultaneously.

Since p_n encourages to predict short sentences, we introduce a brevity penalty (BP) item to penalize those short sentences, which is defined as follows:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{otherwise,} \end{cases} \tag{3.51}$$

where r and c are the reference sentence and candidate sentence length, respectively.

ROUGE [60] is an evaluation metric based on the n-gram precision and recall.

Formally, the ROUGE-N is defined as follows:

$$\text{ROUGE-N} = \frac{2P_nR_n}{R_n + P_n}, \tag{3.52}$$

where P_n and R_n represent the n-gram precision and recall, respectively. Similar to ROUGE-N, ROUGE-L is based on the longest common subsequence between generated sentence and reference sentences, instead of based on the n-gram statistics.

METEOR [10] Like ROUGE, METEOR is also recall-oriented by modifying the computation of precision and recall, with an additional penalty score to penalize those sentences with incorrect word order.

$$\text{METEOR} = F_1(1 - Penalty), \tag{3.53}$$

where F_1 is based on the uni-gram precision (P_1) and recall (R_1), i.e., $F_1 = \frac{10P_1R_1}{R_1+9P_1}$. In addition, the $Penalty$ factor is designed as the ratio of matched chunk number to matched uni-gram number between the candidate and reference sentences, i.e.,

$$Penalty = 0.5\left(\frac{\#\text{chunks}}{\#\text{matched uni-gram}}\right)^3. \tag{3.54}$$

where *chunk* is composed of a group of adjacent uni-grams that are in both the candidate and reference sentences.

CHAPTER 4

CODE SUMMARIZATION IN LOW-RESOURCE SETTINGS

Code documentation in the form of code comments has been an integral component of program development, supporting many sub-fields of software engineering: software maintenance, code categorization, and code retrieval. Unfortunately, only a few real-world software repositories are well-documented with high-quality comments. Due to inconsistent comments or different naming conventions, many projects are inadequately documented, resulting in high maintenance costs. Therefore, code summarization has become increasingly important in automated software engineering and code intelligence. Although the code summarization task has been considerably explored, its application in real-world low-resource settings attracts little attention because of the lack of high-quality corpus and language gap. In this chapter, we propose METASUM, a meta-learning-based summarization model, excels all baselines in two real-world datasets from GitHub, and detailed experiments also prove METASUM has robustness when varying the portion of an enormous corpus.

4.1 Existing Problems

Although the code summarization task [5, 46, 84] benefits from the advances in machine learning and natural language processing communities, it remains challenging for low-resource settings where only a few samples are available for training. Most existing methods will summarize wrong descriptions of the semantics meaning for code snippets and cannot be directly tackled by models in high-resource programming languages. We attribute these problems to the issues of *data-hungry* and *language gap*. From the perspective of *data-hungry*, models without prior knowledge heavily rely on enormous well-documented corpora, which is impossible for low-resource programming languages, and tend to be ineffective in code summarization. Moreover, due to the existence of *language gap*, directly employing models trained on high-resource languages for real-world low-resource applications can only acquire lousy performance. Therefore, the main problem is how to use massive well-documented programs for low-resource programming languages even if there is an ineligible language gap between them.

4.1.1 Challenges

The main challenge of the problem mentioned above is effectively extracting prior knowledge from high-resource programming languages and adapting it to a new programming language in low-resource settings. We assume that although the programming language gap widely exists between programming languages, a shared common knowledge, e.g., the semantic meaning of identifier name, contributes to better comment generation. Moreover, although programming language grammars may differ considerably, e.g., Ruby and Nix, some basic syntactical representations are shared, such as loop and if-else statements, enabling us to learn coarse semantics representations. Furthermore, like meta-learning, which hypothesizes that all tasks are in the same task distribution but represented in different domains, we recognize that programming languages and natural languages follow a language distribution in different domains. Inspired by the points above, we decided to utilize meta-learning to extract knowledge from high-resource datasets and adapt it to a target language.

In this dissertation, we demonstrate that transferring prior knowledge from programming languages to others can boost the model performance of code summarization, especially in a low-resource setting. To summarize, the primary contributions of our proposed method are as follows.

- **The first application of meta-learning in code summarization.** To our best

knowledge, METASUM is the first work about meta-learning in code summarization. To verify the generalization of our proposed METASUM, we conduct our experiments on high-/low-resource programming languages.

- **Evaluation and analysis.** We perform comprehensive experiments on real-world datasets collected from GitHub. The experimental results demonstrate the effectiveness of transferring the prior/common knowledge from high-resource languages to low-resource settings in the code summarization task.

4.2 Prelimiaries

In this section, we first formulate the problem of code summarization in a low-resource setting and then introduce some primitives about transfer learning and meta-learning. Finally, we illustrate the difference between these two learning methods.

4.2.1 Problem Formulation

We aim to summarize a program corpus of low-resource programming languages with limited training samples via transferring the learned knowledge from other programming languages with massive well-labeled training data.

Given the labeled data from a set of source programming language $S \in \{\text{Java}, \text{Python}, \text{JavaScript}, \text{PHP}, \text{Go}\}$, we have the training data $\mathscr{D}^S_{train} = \{(\mathbf{x}^S_n, \mathbf{y}^S_n) \mid n = 1, \ldots, N_S\}$, where each language has N_S labeled data pairs. Similarly, $T \in \{\text{Nix}, \text{Ruby}\}$ denotes the set of target languages with low-resource training data $\mathscr{D}^T_{train} = \{(\mathbf{x}^T_n, \mathbf{y}^T_n) \mid n = 1, \ldots, N_T\}$, where the dataset in the target language T is much smaller than any source language S, i.e., $N_T \ll N_S$. Our solution is to leverage prior knowledge learned by transfer learning or meta-learning from the source datasets $\mathscr{D}_S$ to improve the performance on the target datasets $\mathscr{D}_T$.

While extracting prior knowledge, a model is pre-trained across different high-resource languages

$$f^{source}_\theta : \mathscr{X} \times S \to \mathscr{C}, \tag{4.1}$$

where $\mathscr{X}$ denotes code snippets, $\mathscr{C}$ denotes the corresponding comments, and $\times$ is the cartesian product operation. θ is the model parameters that can serve as initial parameters for the target datasets.

During the adaptation phase, we aim to learn a model that could perform well in an unseen target domain. We fine-tune the model f_θ^{source} with the training data $\mathscr{D}_{train}^T$ of the target domain and obtain a new model f_θ^{target}.

$$f_\theta^{target} : \mathscr{X}^{target} \times T \rightarrow \mathscr{C}^{target}, \tag{4.2}$$

The goal is to maximize the likelihood of generating code summaries on the target domain.

4.2.2 Transfer Learning

Although existing neural-based methods have achieved decent performances in the code summarization task, they are based on the assumption that the data distribution of *source domain* (training data) and *target domain* (testing data) are identical or at least highly similar. However, in practice, the data distribution can differ when adapting to a new scenario, e.g., programming languages following distinct grammars, preventing learned knowledge from being directly applied to new domains.

Transfer learning is a learning strategy designed to transfer the knowledge from source domains and adapt it to target domains to alleviate the *data hungry* issue in the target domains. Formally, given a source domain $\mathscr{D}^S$ and a target domain $\mathscr{D}^T$, transfer learning aims to improve the model f_θ^{target} in $\mathscr{D}^T$ using the knowledge in $\mathscr{D}^S$ and $\mathscr{D}^T$. Moreover, when the target data is sufficient, transfer learning can boost training efficiency, as in our scenario.

Fine-tuning is a simple yet effective approach to transfer learning via initializing parameters for target domains, which comprises a pre-training phase and adaptation. Compared with training from scratch, fine-tuning can achieve better performance on related tasks and reduce the number of training samples. Furthermore, to ensure the effectiveness of fine-tuning, pre-training and adaptation share the same objective function. Besides fine-tuning, **pre-trained models** learn general representation via masked language modeling, which has been proven effective for downstream tasks.

This dissertation takes several fine-tuning policies and pre-trained models as baselines for comparison.

4.2.3 Meta-Learning

Unlike transfer learning, which completes tasks directly using past experiences, meta-learning considers how to extract common knowledge for future unseen tasks. In this

dissertation, we comprehensively explore two meta-learning methods for the low-resource code summarization task: Model-Agnostic Meta-Learning (MAML) and its variant First-Order MAML (FOMAML).

MAML-based METASUM Formally, let f_θ be a neural-based model with parameters θ. For each programming language S_k, MAML-based METASUM compute language-specific gradients as follows:

$$g_k = \nabla_\theta \mathscr{L}_{S_k}(f_\theta), \tag{4.3}$$

where $\mathscr{L}_{S_k}(f_\theta)$ is the task-specific adaption loss function that evaluates the difference between the generated comments and golds. After gradient updating, we have new model parameters $\theta_k^{'} \leftarrow \theta - \eta g_k$ where η is the learning rate for language-specific gradients. Then the model parameters θ are trained to optimize the performance on unseen samples from programming language S_k. Formally, to measure a model's adaptation ability, the meta-objective is defined as follows:

$$\min_\theta \sum_{S_k \sim S} \mathscr{L}_{S_k}(f_{\theta_k^{'}}) = \sum_{S_k \sim S} \mathscr{L}_{S_k}(f_{\theta - \eta \nabla_\theta \mathscr{L}_{S_k}(f_\theta)}). \tag{4.4}$$

Furthermore, to acquire common knowledge, MAML-based METASUM is updated by the average language-specific gradients:

$$\theta \leftarrow \theta - \varphi \nabla_\theta \sum_{S_k} \mathscr{L}_{S_k}(f_{\theta_k^{'}}), \tag{4.5}$$

where φ denotes the meta learning rate for adaptation. We describe the training procedure of MAML-based METASUM in Algorithm 1.

FOMAML-based METASUM This learning policy computes gradients without considering the initialization. It continuously updates model parameters at each language-specific task and final gradient computation. Formally, the gradient update of FOMAML-based METASUM is defined as follows,

$$\theta \leftarrow \theta - \varphi \nabla_\theta \sum_{S_k} \mathscr{L}_{S_k}(f_\theta), \tag{4.6}$$

Workflow of METASUM The workflow of (MAML-/FOMAML-based) METASUM for code summarization task is shown in Figure 4.1 and consists of three steps, i.e., meta pre-training, adaption, and model application. It is believed that meta pre-training is to

Algorithm 1 MAML-based METASUM for low-resource code summarization.

Require: $p(\mathcal{T})$: programming language distribution
Require: η, φ: learning rates
Initialize the model parameters θ with a pre-trained model
while *not done* **do**
  Sample a batch of programming language i.e., $S_k \sim p(\mathcal{T})$
  for *all* S_k **do**
    Evaluate $\nabla_\theta \mathcal{L}_{S_k}(f_\theta)$ according to Eq. 3.33
    Update parameters θ with gradient decent: $\theta_k^{'} \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{S_k}(f_\theta)$
  end
  Update $\theta \leftarrow \theta - \varphi \nabla_\theta \sum_{S_k \sim p(\mathcal{T})} \mathcal{L}_{S_k}(f_{\theta_k^{'}})$
end

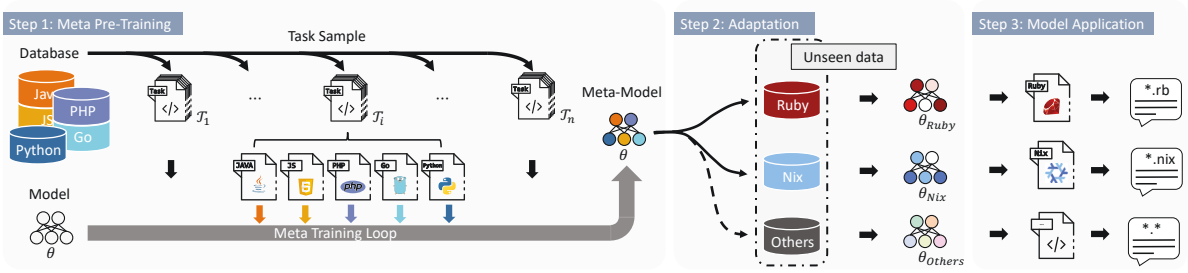


Figure 4.1: The workflow of METASUM for code summarization task.

extract common (prior) knowledge cross programming languages. At each meta-training loop, we compute language-specific gradients based on the initial parameters and sample data batches from programming languages and then update METASUM via gradients. At the adaptation step, the meta-model is fine-tuned on unseen target datasets for real-world applications.

4.2.4 Transfer Learning v.s. Meta-Learning

Conventionally, transfer learning refers to directly applying prior knowledge for target tasks without thinking about future application scenarios, while meta-learning computes gradients via considering the optimal step towards all tasks based on current parameters. It only requires a few gradient updates in target zero-/few-shot problems. Theoretically speaking, meta-learning assumes all tasks, including unseen target tasks, are in the same task distribution but represented in different domains, and, therefore, it works on unseen tasks during adaptation. However, transfer learning does not need to follow such a strict assumption, and it can help by pre-trained on any source tasks.

4.3 Experimental Setup

4.3.1 Dataset

We collect code snippets and corresponding comments from high-resource and low-resource programming languages to verify our proposed method. In this dissertation, we adopt a public dataset CodeSearchNet [45], which is proposed for the code search challenge. The dataset is collected from GitHub, composed of publicly available repositories written in multiple programming languages, e.g., Java, JavaScript, Python, PHP, Go, and Ruby. Furthermore, we also crawled repositories in the Nix programming language from GitHub.

Pre-processing We analyze the qualities of samples in the original dataset for good performance scores, trying to filter out those low-quality code snippets. Our pre-processing rules are defined as follows:

- Removing code snippets containing non-ASCII characters, e.g., Chinese and Russian comments.
- The length of a comment separated by space tokenizer should be greater than 3 and lower than 128.
- Removing meaningless symbols in comment, e.g., "{" and "@".
- Removing codes that cannot be correctly parsed by Tree-Sitter. [1]
- The length of a code tokens should be greater than 3 and lower than 512.
- Removing comments in code snippets.
- Given that Nix samples are too scarce, we refine their comments.
- Remove duplicate code snippets because of [4].

Finally, the statistics of the dataset after data pre-processing are summarized in Table 4.1.

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

Table 4.1: The statistics of dataset in our experiments.

	Java	**Python**	**JavaScript**	**PHP**	**Go**	**Ruby**	**Nix**
Train	164,923	251,820	58,025	241,242	167,288	24,927	75
Validation	5,183	13,914	3,885	12,982	7,325	1,400	188
Test	10,955	14,918	3,291	14,014	8,122	1,261	114
# AVG code length	98.34	96.84	112.31	106.00	95.34	64.19	47.23
# AVG comment length	13.22	11.36	11.79	9.04	15.48	12.75	7.90

4.3.2 Implementation Details

In our experiments, we tokenize each code snippet with the BPE from PLBART to get the sub-word vocabulary of size 50K. For RNN-based and Transformer-based methods, we only follow the hyper-parameters revealed in the original papers. Those models are trained with a learning rate of 1e-4 and a batch size of 32. Since CodeBERT and GraphCodeBERT are Transformer encoders, we build Transformer decoders with uniform initialization. Moreover, We fine-tune pre-trained models (such as CodeBERT, GraphCodeBERT, and PLBART) on datasets with a learning rate of $5e-5$ and a batch size of 4. Since PLBART outperforms the resting pre-trained models, we utilize it as the initialization for fine-tuning and meta-learning methods. For those two methods, we first pre-train PLBART on high-resource programming languages with four V100 GPUs until we acquire minimized loss value on validation datasets and then adapt it on low-resource datasets with the learning rate of 1e-5 and a batch size of 2.

In this dissertation, we conducted all experiments on a 64-code 2.00 GHz Intel Xeon(R) Gold 5117 CPU with 256GB of RAM and four Tesla V100-SXM2-32GB GPUs. We use Python 1.8, PYTORCH 1.8.1, and Cuda 10.2 as the running environment. All our experimental data and source code are available at: `https://github.com/CGCL-codes/naturalcc`.

4.3.3 Research Questions

To evaluate the performance of our proposed METASUM, we have conducted experiments to answer the following research questions:

RQ1. Can our proposed METASUM transfer prior knowledge to improve model performance?

RQ2. What is the impact of parameters initialization?

RQ3. How does METASUM work on different scales of datasets?

RQ4. What is the sensitivity of METASUM under different numbers of code tokens and different comment lengths?

4.3.4 Experimental Results

RQ1: Evaluation of METASUM. To investigate the effectiveness of our proposed meta-learning-based model, METASUM, which considers transferring common knowledge for target datasets, we compared it with state-of-the-art models and a transfer-learning-based method on the Nix dataset. The reported results are determined based on the models best BLEU score on validation datasets.

Table 4.2: Experimental results on **Nix** dataset. (Best scores are in **boldface**)

Method	**BLEU**	**ROUGE-L**	**METEOR**
CodeNN [46]	9.06	5.77	2.78
DeepCom [43]	12.56	4.52	1.94
Bi-LSTM [79]	13.46	5.96	2.11
N-ary-Tree2Seq [85]	14.27	6.42	2.91
Child-Sum-Tree2Seq [85]	13.48	5.78	2.58
RL+Hybrid2Seq [84]	13.34	6.18	2.24
Code2Seq [6]	10.42	4.65	2.25
Transformer [81]	13.56	6.70	3.03
CodeTransformer [2]	13.63	6.87	2.89
CodeBERT [27]	15.01	6.13	2.98
GraphCodeBERT [38]	15.49	7.09	3.61
PLBART [1]	17.48	16.65	7.97
Fine-tuning	17.43	15.18	7.34
MAML-based METASUM	12.69	5.60	2.80
FOMAML-based METASUM	**17.78**	**18.43**	**8.56**

Table 4.2 confirms that common knowledge exists across programming languages and that using prior knowledge for low-resource settings is promising. When comparing models with prior knowledge (such as PLBART, Fine-tuning, and METASUM) with those models from scratch, they acquire at least 1.5 absolute points in terms of BLEU score. For those pre-trained models which only employed a bidirectional encoder, i.e., CodeBERT and GraphCodeBERT, they only outperform models from scratch because it requires a new auto-regressive decoder during adaptation. Furthermore, since PLBART

is pre-trained with an encoder-decoder framework, it is more effective than other pre-trained models in generation tasks. Most importantly, FOMAML-based METASUM acquires 0.35 BLEU and 3.35 ROUGE-L points over fine-tuning and at least 0.3 BLEU and 1.78 ROUGE-L points over state-of-the-art pre-trained models.

Table 4.3: Experimental results on **Ruby** dataset. (Best scores are in **boldface**)

Method	**BLEU**	**ROUGE-L**	**METEOR**
Bi-LSTM [79]	10.58	17.07	6.02
Transformer [81]	11.35	17.30	5.68
CodeBERT [27]	12.98	18.96	7.06
GraphCodeBERT [38]	13.83	20.70	8.09
PLBART [1]	**14.96**	**24.67**	**9.92**
Fine-tuning	14.54	23.88	9.27
MAML-based METASUM	12.61	18.89	6.39
FOMAML-based METASUM	14.29	23.81	8.71

Bedside the evaluation on a low-resource dataset, we also explore the effectiveness of prior knowledge in a high-resource setting closer to vanilla applications. Table 4.3 shows some classical methods of code summarization task on a Ruby dataset. When it comes to a high-resource scenario, the pre-trained models still excel models from scratch, and PLBART achieves the best performance. However, FOMAML-based METASUM cannot defeat PLBART on all evaluation metrics.

On the other hand, MAML-based METASUM only achieves lousy performance in both Table 4.2 and Table 4.3 among those models with prior knowledge, even worse than most RNN-based methods. We attribute the problem to the low convergence rate of MAML, which will be further explored and illustrated in RQ2.

RQ2: Impact of parameter Initialization Since pre-trained models function as initialization for better performance, we initialize METASUM with or without PLBART parameters for meta pre-training. Figure 4.2 shows the meta pre-training loss of METASUM within 30,000 iterations. When there is no parameter initialization, the FOMAML-based model has a faster convergence rate than the MAML-based METASUM. With PLBART parameters as initialization, FOMAML-based METASUM still converges while the MAML-based does not. Furthermore, the parameters enable MAML-based and FOMAML-based METASUM to share a lower initial loss, contributing to better prior knowledge.

For the faithfulness of parameter initialization, we conduct experiments with three methods, i.e., fine-tuning and MAML-/FOMAML-based METASUM. From Table 4.4 and

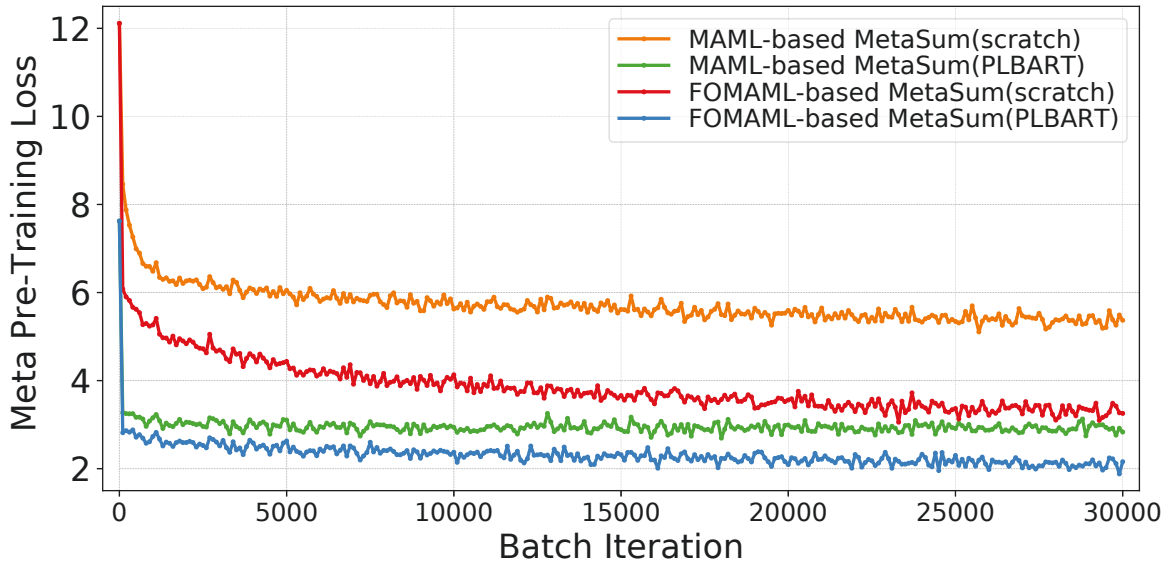


Figure 4.2: Meta pre-training loss of METASUMwith different parameter initialization.

Table 4.5, we can observe that all methods benefit from parameter initialization and excels PLBART. Compared with fine-tuning, MAML-based METASUM acquires better improvement from initialization on the Nix and Ruby datasets. Compared with PLBART, FOMAML-based METASUM acquires 3.18 points and 5.41 in BLEU and ROUGE-L scores on the Nix dataset. It also achieves the best performance on the Ruby dataset with parameter initialization.

Table 4.4: Experimental results of different parameter initialization on **Nix** dataset. (Best scores are in **boldface**)

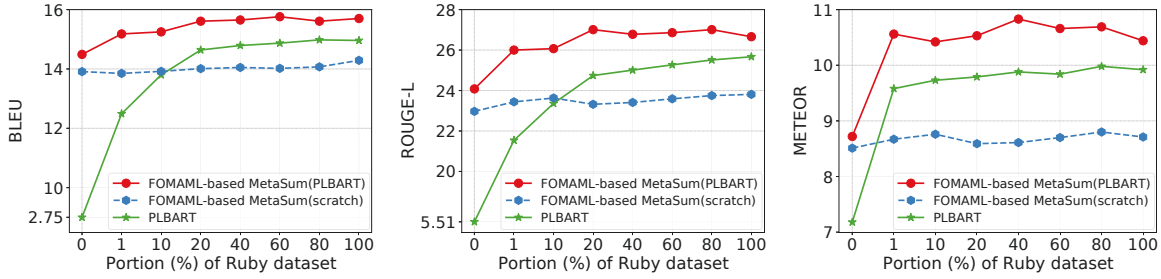
Method	**BLEU**	**ROUGE-L**	**METEOR**
PLBART [1]	17.48	16.65	7.97
Fine-tuning(scratch)	17.43	15.18	7.34
Fine-tuning(PLBART)	18.52	17.72	8.33
MAML-based METASUM(scratch)	12.69	5.60	2.80
MAML-based METASUM(PLBART)	19.81	19.42	8.61
FOMAML-based METASUM(scratch)	17.78	18.43	8.56
FOMAML-based METASUM(PLBART)	**20.66**	**22.06**	**9.71**

RQ3: Robustness of METASUM As we mentioned before, when the well-documented training data is sufficient, transferring knowledge across programming languages can also benefit model performance. To demonstrate the effectiveness of METASUM in low-resource and high-resource scenarios, we compare METASUM and some baselines on a

Table 4.5: Experimental results of different parameter initialization on **Ruby** dataset. (Best scores are in **boldface**)

Method	**BLEU**	**ROUGE-L**	**METEOR**
PLBART [1]	14.96	24.67	9.92
Fine-tuning(scratch)	14.54	23.88	9.27
Fine-tuning(PLBART)	15.26	26.21	10.25
MAML-based METASUM(scratch)	12.61	18.89	6.39
MAML-based METASUM(PLBART)	15.29	26.34	10.36
FOMAML-based METASUM(scratch)	14.29	23.81	8.71
FOMAML-based METASUM(PLBART)	**15.70**	**26.66**	**10.44**

decremental dataset in Ruby. Figure 4.3 shows the experimental results when we set the portion of training samples as 0%, 1%, 10%, 20%, 40%, 80% and 100%. From this figure, it is observed that the performance of all models decreases corresponding to the portions. When there are no training samples, PLBART acquires low performance on all evaluation metrics because of random parameters initialization, whereas PLBART and METASUM still work because they have been pre-trained on a large corpus. Furthermore, our proposed METASUM with or without parameter initialization demonstrates strong robustness among those methods. Most importantly, with PLBART parameter initialization, METASUM excels other baselines in all cases.

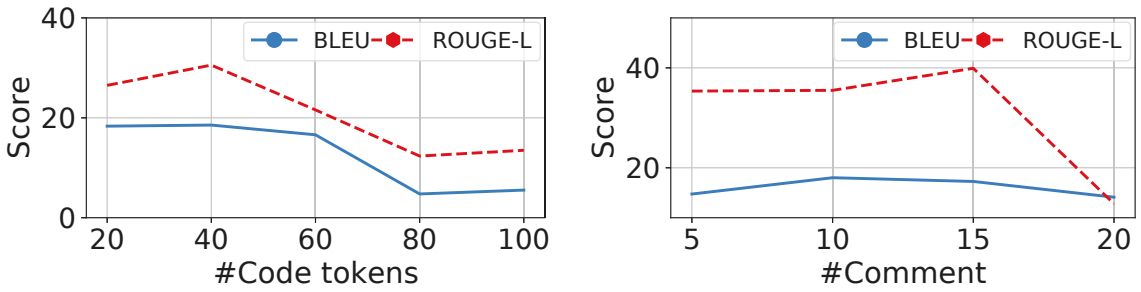


(a) BLEU. (b) ROUGE-L. (c) METEOR.

Figure 4.3: The performance of METASUM and PLBART when varying the portions of Ruby training dataset.

RQ3: Sensitivity Analysis In order to analyze the sensitivity of METASUM, we conduct experiments when varying the length of code and comment and test the model performance on the Nix and Ruby datasets. Here, we investigate two parameters (i.e., the number of code tokens and comment length) that may affect code representation learning and generation. Figure 4.4 and 4.5 show the evaluation metrics of METASUM

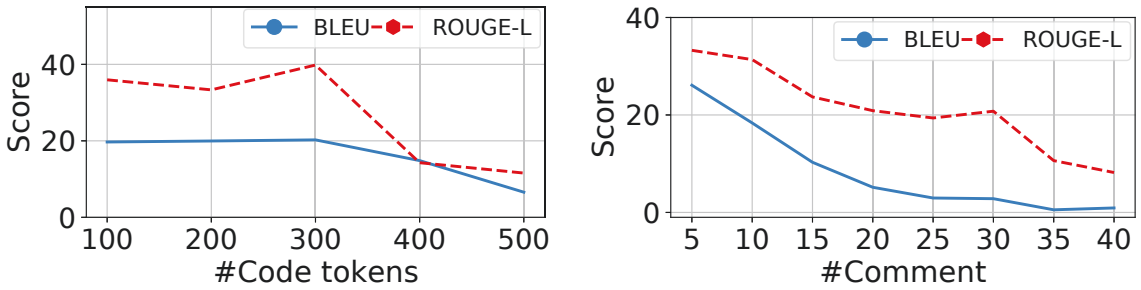
on the Nix and Ruby dataset, respectively, when varying the number of code tokens and the comment length.



(a) #code tokens. (b) Comment length.

Figure 4.4: The performance of our METASUM on the Nix dataset, w.r.t. varying the numbers of code tokens and the comment lengths.

From Figure 4.4a, we observe that performance decrease with the growth of the code tokens, showing it is challenging to present long code tokens. From Figure 4.4b, the ROUGE-L performance of our proposed METASUM dramatically decreases in generating long comments which are longer than 15. It implies that there remains challenging to transfer knowledge-based methods to generate long comments.



(a) #code tokens. (b) Comment length.

Figure 4.5: The performance of our METASUM on the Ruby dataset, w.r.t. varying the numbers of code tokens and the comment lengths.

From Figure 4.5a, we observe that performance decrease with the growth of the code tokens, showing the same challenge in presenting long code tokens. From Figure 4.5b, the performance of our proposed METASUM decreases in generating long comments, suggesting that it remains challenging for long comments generation.

4.3.5 Case Study

To better understand our model, we show two real-world code snippets from Nix and Ruby test datasets with generated comments from METASUM as well as other baselines, as shown in Table 4.6 and 4.7. Thanks to byte-pair-encoding, our generated comments have no <UNK> token.

The code snippet in Table 4.6 is selected from the Nix dataset in which there are only 75 samples available for models to learn or adapt. We observe that the models from scratch can hardly learn from insufficient data and only generate meaningless words from this case. It verifies our assumption that the data-hungry problem lies in existing methods, and these models cannot be directly applied to such settings. Note that Transformer-based, pre-trained models, Fine-tuning, and METASUM have highly similar Transformer architecture. As prior knowledge is introduced, their performance boosts considerably in all evaluation metrics, demonstrating the necessity of prior knowledge for the low-resource application.

Table 4.6: A Nix example of generated comments for case study.

A Nix code snippet	

`system = fold ({ oldDependency, newDependency }: drv:`
`  pkgs.replaceDependency { inherit oldDependency`
`  ↪ newDependency drv; }`
`) baseSystem config.system.replaceRuntimeDependencies;`	

Human	replace runtime dependencies
Bi-LSTM	import a packages
Transformer	enable BGQ packages
CodeBERT	disable dependencies
GraphCodeBERT	disable dependencies
PLBART	replace runtime dependencies
Fine-tuning(scratch)	replace system dependencies with new dependencies
Fine-tuning(PLBART)	fold a system into a new system
MAML-based METASUM(scratch)	python packages
MAML-based METASUM(PLBART)	replace runtime dependencies
FOMAML-based METASUM(scratch)	replace dependency dependencies with new dependencies
FOMAML-based METASUM(PLBART)	replace runtime dependencies

On the other hand, the code snippet in Table 4.7 is collected from the Ruby dataset in which there are over 17K high-quality data pairs for training. In this case, we find that even models without prior knowledge can successfully predict tokens in the ground

truth with a larger dataset. Moreover, it is worth mentioning that METASUM still excels other baselines when data are sufficient, proving that our proposed model is more robust than others.

Table 4.7: A Ruby example of generated comments for case study.

A Ruby code snippet	

`def retrieve_watcher(user, repo, watcher)`
`  repo_bound_item(user, repo, watcher, :watchers,`
`  ["repos/#{user}/#{repo}/stargazers"],`
`  {'repo' => repo, 'owner' => user},`
`  'login', order = :desc)`
`end`	

Human	Retrieve a single watcher for a repository
Bi-LSTM	Shows a single task for the user .
Transformer	Creates a new user with a user s user .
CodeBERT	Gets the watchers for the user .
GraphCodeBERT	Gets the watchers for the user .
PLBART	Retrieve a watcher for a repo
Fine-tuning(scratch)	Retrieve the watcher information for a user
Fine-tuning(PLBART)	Find changesets for a repository
MAML-based METASUM(scratch)	Adds a list of the given file .
MAML-based METASUM(PLBART)	Retrieve a watcher for a repository
FOMAML-based METASUM(scratch)	Retrieve a watcher for a repository .
FOMAML-based METASUM(PLBART)	Retrieve a single watcher for a repository

4.3.6 Error Analysis

We also present a terrible case generated by METASUM and other baselines on the Nix dataset to seek insights for further improvement. As shown in Table 4.8, METASUM does not understand the difference between *create* and *make*, and confuses *conf* and *config*. Even in most cases, *conf* is recognized as the abbreviation of *config*. We acknowledge that a margin still exists for further investigation than the ground truth. Here we point out two future solutions for improvements. The first solution is introducing syntactical information for tokens, e.g., adding a syntax embedding layer. Another solution is that since the high-resource programming language datasets are imbalanced, we can utilize an up/down sampling strategy [1, 55].

Table 4.8: A Nix example of generated comments for error analysis.

A Nix code snippet	

`makeLimitsConf = limits:`
`  pkgs.writeText "limits.conf"`
`  (concatMapStrings ({ domain, type,`
`  ↪ item, value }:`
`  "${domain} ${type} ${item}`
`  ↪ ${toString value}\n")`
`  limits);`	

Human	Retrieve a single watcher for a repository
Bi-LSTM	import a packages
Transformer	shell script with additional plugins
CodeBERT	create all plugins
GraphCodeBERT	create all plugins
PLBART	create limits config
Fine-tuning(scratch)	write config limits
Fine-tuning(PLBART)	make limits config
MAML-based METASUM(scratch)	for a
MAML-based METASUM(PLBART)	make limits configuration
FOMAML-based METASUM(scratch)	create all limits
FOMAML-based METASUM(PLBART)	make limits conf

4.4 Conclusion

This dissertation investigates a new but essential code summarization scenario in which only a few samples are available for training. Experimental results on two real-world datasets show that existing models without prior knowledge heavily suffer from such a setting in which pre-trained models still work. Furthermore, we also explore and interpret different meta-learning-based policies for prior knowledge extraction and with or without parameter initialization. The sensitivity analysis shows it is still challenging for long and complicated code snippets to generate long comments. Moreover, from error analysis, we found that our proposed METASUM with pre-trained knowledge cannot recognize identifiers with the same name and propose a candidate solution for such a problem.

CHAPTER 5

NATURALCC TOOLKIT

Despite code intelligence having made gorgeous breakthroughs for years, there is no toolkit for researchers in the field. Therefore, we present NATURALCC (stands for Natural Code Comprehension), an extensible open-source toolkit for machine-learning-based software engineering. Researchers can reproduce state-of-the-art models and verify their algorithms with such a toolkit. NATURALCC is built upon FAIRSEQ and PYTORCH, providing (1) a collection of code corpus with detailed READMEs, (2) a modular and extensible framework to register new downstream tasks and models, and (3) various implementations of network modules and deep learning tricks. However, compared with those toolkits in NLP, FAIRSEQ and TRANSFORMERS, which can only process code snippets as plain text, NATURALCC further explores program properties in detail and has better supports for structural modalities and complex neural networks, e.g., intermediate representation and graph neural networks.

5.1 Introduction

Code intelligence is, informally, the task of applying machine learning methods to analyze programs or complete software engineering tasks. With the breakthrough of machine learning in software engineering [1, 27, 34, 40], compilation [20, 21, 56] and programming language [14, 16, 18, 24] communities, this topic gain wide attention from these areas.

In recent years, many code intelligence approaches have been proposed for automatically completing downstream tasks, such as code summarization, code retrieval, and type inference, to improve developer productivity and alleviate their burden. However, several limitations hinder the development of machine-learning-based code intelligence.

- **Lack of technical supports for code intelligence applications**

  Although many professional toolkits such as FAIRSEQ [71], TRANSFORMERS and *AllenNLP* [29] have been proposed for the field of NLP, it remains challenging to apply them for programming language tasks directly. These toolkits are originally devised for modeling sequence-to-sequence tasks involving natural language and, therefore, ignore the possibility of processing structural data formats and complicated network architectures, which widely occur in code intelligence.

- **Lack of standardized implementation for performance comparison**

  Since current works are implemented on different platforms, it is crucial to build a benchmark platform to understand whether the performance enhancement is from the model design, hyper-parameter, or fine-tuning tricks.

Enlightened by FAIRSEQ [71] and TRANSFORMERS, we present NATURALCC for researchers in automated software engineering to build connections between software engineering, compilation, and programming language communities. The toolkit follows an extensible framework, as shown in Figure 5.1, enabling users to devise an arbitrary model architecture for code intelligence tasks. Additionally, the pipeline of NATURALCCis shown in Figure 5.2 in which data pairs are first batchified via data processing and then fed into models for training. We also demonstrate NATURALCC with a graphical user interface on three software engineering applications, i.e., code summarization, code completion, and type inference.

Compared with FAIRSEQ and TRANSFORMERS, NATURALCC have the following advantages:

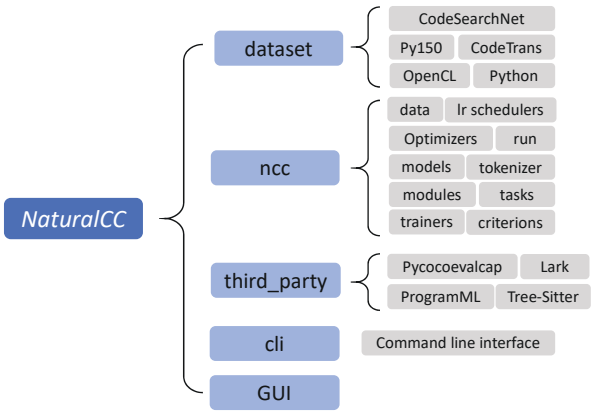


Figure 5.1: The structure of NATURALCC.

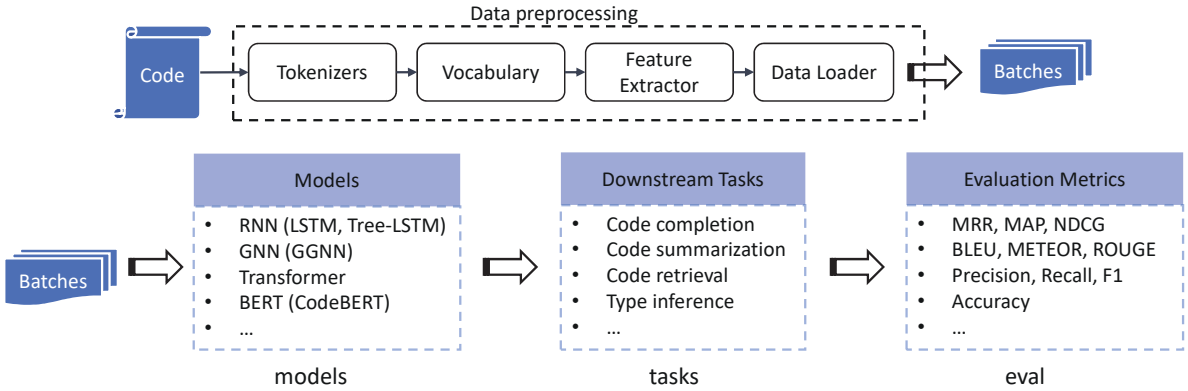


Figure 5.2: The pipeline of NATURALCC.

- **Better code modalities and complex architecture support**

  With different programming language lexers and parers, we can convert a code snippet into various modalities with or without structural information, e.g., code tokens or control flow graphs. NATURALCC supports fast binary reading and writing and complicated neural networks for structural modalities in big corpora.

- **Various implementations**

  For some modules and tricks (e.g., TENSORFLOW gradient clipping and parameter initialization) that PYTORCH does not support, we re-implement them in NATURALCC, which liberates users from platform conflicts.

- **A Collection of code corpus with pre-processing scripts**

  We have cleaned and pre-processed public datasets, i.e., CodeSearchNet [45], PythonDoc [84], Py150 [74] and so on. We also provide data preprocessing scripts and

detailed READMEs for extracting code modalities based on Lark[1], TreeSitter[2], LLVM[3] and other third party repositories.

- **An extensible framework**

  Based on the registry mechanism implemented in FAIRSEQ, our framework is well modularized and easily extended to various downstream tasks. In particular, users only need to implement a new task by instantiating one of our templates and then registering them when implementing a new task. We illustrate this idea with a short tutorial in the appendix A.1.

- **Performance benchmark**

  We have benchmarked six downstream tasks (i.e., code summarization, code retrieval, code completion, code completion, type inference, code translation, and heterogeneous mapping) over the corresponding datasets using NATURALCC, achieving state-of-the-art or competitive performance.

- **More than neural networks**

  Except for neural-based application, NATURALCC also incorporates statistical machine learning methods for code intelligence tasks such as TF-IDF for search tasks and decision tree for classification problems.

5.2 Graphical User Interface

We have also provided a graphical user interface for users to easily access and explore the results of each trained model through a Web browser. The design of our website is based on the open-source demo of ALLENNLP [29]. As shown in Figure 5.3, we have integrated three popular software engineering tasks for demonstration, i.e., code summarization, code retrieval, and code completion. Take code summarization as an example. By default, we have implemented this task based on the Transformer. Given a code snippet of Python, when clicking the *Run* button, NATURALCC will invoke a user-selected trained model and begin inference, and the model output will be displayed at the bottom of the web page.

[1]https://github.com/lark-parser/lark
[2]https://tree-sitter.github.io/tree-sitter/
[3]https://llvm.org/

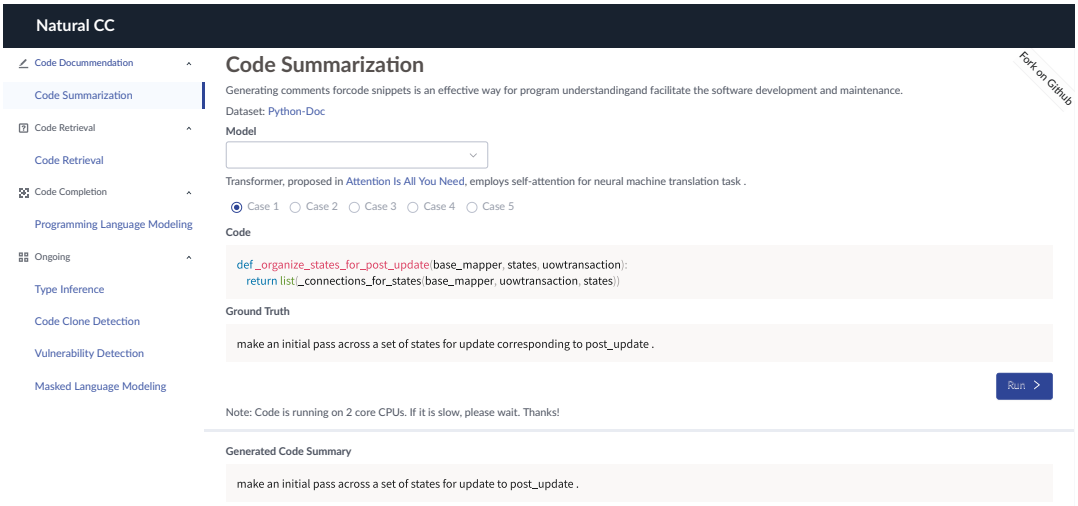


Figure 5.3: A screenshot of NATURALCC GUI.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this dissertation, we have identified and formulated a new problem of low-resource code summarization, a task of summarizing the functionality and semantics of source codes written in a minority or emerging programming language with limited training samples. For a specific programming language, we formulate the code summarization following the encoder-decoder framework and the milestone works of the task. Further, we explore the performance of previous works in a low-resource dataset and observe that existing models without prior or common knowledge indeed suffer from the data-hungry problem. Then, we propose a meta-learning-based approach, METASUM, for better prior knowledge extraction. Experimental results on the real-world datasets, i.e., Ruby and Nix, collected from GitHub confirm the effectiveness of METASUM in generating high-quality code comments in low-/high-resource settings. Compared with state-of-the-art models, even the pre-trained model, our proposed METASUM outperforms all baselines with or without transferring knowledge. Furthermore, case-study and error analysis reveal a future topic to improve METASUM.

6.2 Future Work

METASUM naturally has opportunities for further work: one candidate area is an exploration of further experiments about the combination of METASUMwith CodeBERT and GraphCodeBERT models because we only analyze the experimental results with PLBART initialization. In this dissertation, we evaluated the generalization ability of METASUM where our method has been tested on two real-world datasets and with different portions of training samples. In our future work, it would be interesting to explore the performance of other meta-learning models, e.g., Reptile [70], and other pre-trained models as initialization.

APPENDIX

A.1 A Short Tutorial for NATURALCC Toolkit

In this subsection, We take code summarization as an example to show how to register a new model in NATURALCC.

Register task First of all, one should register a *Task* in which defines the rules to load datasets. As shown in Listing A.1, we register the *DemoTask* class in which it loads datasets with the maximum length of 512 for source and target sentences and appends a special end-of-sentence symbol <EOS> to the target sentences.

Listing A.1: Register a demo task in NATURALCC

```
from ncc.tasks import NccTask, register_task

@register_task('demo')
class DemoTask(NccTask):
    def __init__(self, args, dictionary):
        super(DemoTask, self).__init__(args)
        self.dictionary = dictionary

    def load_dataset(self, mode, src_file, tgt_file, max_len=512):
        # define your loading rules
        # load source tokens
```

```
        # load target tokens and append with <EOS>
        ...
```

Register dataset mainly consists of two elements: a *collation* function to batchify a set of data pairs into a data batch and a *dataset* class to store data and to return i-th data pair.

Listing A.2: Register a demo dataset in NATURALCC

```
import torch
from ncc.data.ncc_dataset import NccDataset

def collate(samples, eos_idx, ):
    # batchify data pairs
    ...

class DemoDataset(NccDataset):
    def __init__(self, dict, src, tgt):
        ...

    def __getitem__(self, index):
        ...

    def collater(self, samples):
        return collate(samples, eos_idx=self.eos)
```

Register model This section defines model construction in the *build_model* function and computation in the *forward* function.

Listing A.3: Register a demo model in NATURALCC

```
from ncc.models import register_model
from ncc.models.ncc_model import NccEncoderDecoderModel

@register_model("demo")
class DemoModel(NccEncoderDecoderModel):
    def __init__(self, encoder, decoder):
        super().__init__(encoder, decoder)

```

```
    @classmethod
    def build_model(cls, dictionary):
        # instantiate encoder and decoder
        ...

    def forward(self, *args, **kwargs):
        # model pipeline
        ...
```

A.2 Performance Benchmark

NATURALCC currently supports six downstream tasks, i.e., code summarization, code retrieval, code completion, type inference, code translation, and heterogeneous mapping, to showcase the effectiveness of the proposed framework. Table A.1 gives a summary of the state-of-the-art models of the above downstream tasks. Note that all methods listed at NATURALCC GitHub homepage (link: `https://github.com/CGCL-codes/naturalcc`) are carefully implemented to ensure the performances are on par with the original papers. To verify the faithfulness and correctness of our implementation, we released all hyper-parameters and training logs of those models.

Table A.1: State-of-the-art models on downstream tasks of automated software engineering and the corresponding datasets.

Task	**Dataset**	**Model**
Code Summarization	Python-Doc [84]	Seq2Seq [46], Transformer [2], PLBART [1]
Code Retrieval	CodeSearchNet [45]	NBow, Conv1D, Bi-RNN, SelfAttn [45]
Code Completion	Py150 [74]	SeqRNN [41], GPT-2, TravTrans [51]
Type Inference	TypeScript [40]	DeepTyper [40], Transformer
Code Translation	CodeTrans [64]	CodeBERT [27], GraphCodeBERT [38], PLBART [1]
Heterogeneous Mapping	OpenCL [31]	DeepTune [21], Inst2Vec [11]

BIBLIOGRAPHY

[1] W. AHMAD, S. CHAKRABORTY, B. RAY, AND K.-W. CHANG, *Unified pre-training for program understanding and generation*, in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2655–2668.

[2] W. U. AHMAD, S. CHAKRABORTY, B. RAY, AND K.-W. CHANG, *A transformer-based approach for source code summarization*, arXiv preprint arXiv:2005.00653, (2020).

[3] W. U. AHMAD, Z. ZHANG, X. MA, E. HOVY, K.-W. CHANG, AND N. PENG, *On difficulties of cross-lingual transfer with order differences: A case study on dependency parsing*, arXiv preprint arXiv:1811.00570, (2018).

[4] M. ALLAMANIS, *The adverse effects of code duplication in machine learning models of code*, in Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2019, pp. 143–153.

[5] M. ALLAMANIS, H. PENG, AND C. SUTTON, *A convolutional attention network for extreme summarization of source code*, in International Conference on Machine Learning, 2016, pp. 2091–2100.

[6] U. ALON, S. BRODY, O. LEVY, AND E. YAHAV, *code2seq: Generating sequences from structured representations of code*, arXiv preprint arXiv:1808.01400, (2018).

[7] U. ALON, R. SADAKA, O. LEVY, AND E. YAHAV, *Structural language models of code*, in International Conference on Machine Learning, PMLR, 2020, pp. 245–256.

[8] J. L. BA, J. R. KIROS, AND G. E. HINTON, *Layer normalization*, arXiv preprint arXiv:1607.06450, (2016).

[9] D. BAHDANAU, K. CHO, AND Y. BENGIO, *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473, (2014).

[10] S. BANERJEE AND A. LAVIE, *Meteor: An automatic metric for mt evaluation with improved correlation with human judgments*, in Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, vol. 29, 2005, pp. 65–72.

[11] T. BEN-NUN, A. S. JAKOBOVITS, AND T. HOEFLER, *Neural code comprehension: A learnable representation of code semantics*, in Advances in Neural Information Processing Systems, 2018, pp. 3585–3597.

[12] A. BISWAS AND S. AGRAWAL, *First-order meta-learned initialization for faster adaptation in deep reinforcement learning*, Nips, 2018.

[13] M. BROCKSCHMIDT, M. ALLAMANIS, A. L. GAUNT, AND O. POLOZOV, *Generative code modeling with graphs*, arXiv preprint arXiv:1805.08490, (2018).

[14] R. BUNEL, M. HAUSKNECHT, J. DEVLIN, R. SINGH, AND P. KOHLI, *Leveraging grammar and reinforcement learning for neural program synthesis*, arXiv preprint arXiv:1805.04276, (2018).

[15] J. CAMBRONERO, H. LI, S. KIM, K. SEN, AND S. CHANDRA, *When deep learning met code search*, in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 964–974.

[16] J. CHEN, J. WEI, Y. FENG, O. BASTANI, AND I. DILLIG, *Relational verification using reinforcement learning*, Proceedings of the ACM on Programming Languages, 3 (2019), pp. 1–30.

[17] X. CHEN, C. LIU, AND D. SONG, *Tree-to-tree neural networks for program translation*, arXiv preprint arXiv:1802.03691, (2018).

[18] Y. CHEN, C. WANG, O. BASTANI, I. DILLIG, AND Y. FENG, *Program synthesis using deduction-guided reinforcement learning*, in International Conference on Computer Aided Verification, Springer, 2020, pp. 587–610.

[19] K. CHO, B. VAN MERRIENBOER, D. BAHDANAU, AND Y. BENGIO, *On the properties of neural machine translation: Encoder-decoder approaches*, in Proceedings of

SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014, D. Wu, M. Carpuat, X. Carreras, and E. M. Vecchi, eds., Association for Computational Linguistics, 2014, pp. 103–111.

[20] C. CUMMINS, Z. V. FISCHES, T. BEN-NUN, T. HOEFLER, AND H. LEATHER, *Programl: Graph-based deep learning for program optimization and analysis*, arXiv preprint arXiv:2003.10536, (2020).

[21] C. CUMMINS, P. PETOUMENOS, Z. WANG, AND H. LEATHER, *End-to-end deep learning of optimization heuristics*, in 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2017, pp. 219–232.

[22] J. DEVLIN, M.-W. CHANG, K. LEE, AND K. TOUTANOVA, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019.

[23] Y. DUAN, J. SCHULMAN, X. CHEN, P. L. BARTLETT, I. SUTSKEVER, AND P. ABBEEL, *Rl2: Fast reinforcement learning via slow reinforcement learning. arxiv, 2016*, arXiv preprint arXiv:1611.02779.

[24] K. ELLIS, M. NYE, Y. PU, F. SOSA, J. TENENBAUM, AND A. SOLAR-LEZAMA, *Write, execute, assess: Program synthesis with a repl*, arXiv preprint arXiv:1906.04604, (2019).

[25] M. FADAEE, A. BISAZZA, AND C. MONZ, *Data augmentation for low-resource neural machine translation*, arXiv preprint arXiv:1705.00440, (2017).

[26] S. Y. FENG, V. GANGAL, J. WEI, S. CHANDAR, S. VOSOUGHI, T. MITAMURA, AND E. HOVY, *A survey of data augmentation approaches for nlp*, arXiv preprint arXiv:2105.03075, (2021).

[27] Z. FENG, D. GUO, D. TANG, N. DUAN, X. FENG, M. GONG, L. SHOU, B. QIN, T. LIU, D. JIANG, ET AL., *Codebert: A pre-trained model for programming and natural languages*, arXiv preprint arXiv:2002.08155, (2020).

[28] C. FINN, P. ABBEEL, AND S. LEVINE, *Model-agnostic meta-learning for fast adaptation of deep networks*, in Proceedings of the 34th International Conference on Machine Learning-Volume 70, JMLR. org, 2017, pp. 1126–1135.

[29] M. GARDNER, J. GRUS, M. NEUMANN, O. TAFJORD, P. DASIGI, N. F. LIU, M. PETERS, M. SCHMITZ, AND L. ZETTLEMOYER, *Allennlp: A deep semantic natural language processing platform*, in Proceedings of Workshop for NLP Open Source Software (NLP-OSS), 2018, pp. 1–6.

[30] J. GEHRING, M. AULI, D. GRANGIER, D. YARATS, AND Y. N. DAUPHIN, *Convolutional sequence to sequence learning*, in International Conference on Machine Learning, PMLR, 2017, pp. 1243–1252.

[31] D. GREWE, Z. WANG, AND M. F. O'BOYLE, *Portable mapping of data parallel programs to opencl for heterogeneous systems*, in Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2013, pp. 1–10.

[32] J. GU, H. HASSAN, J. DEVLIN, AND V. O. LI, *Universal neural machine translation for extremely low resource languages*, arXiv preprint arXiv:1802.05368, (2018).

[33] J. GU, Y. WANG, Y. CHEN, K. CHO, AND V. O. LI, *Meta-learning for low-resource neural machine translation*, arXiv preprint arXiv:1808.08437, (2018).

[34] X. GU, H. ZHANG, AND S. KIM, *Deep code search*, in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 933–944.

[35] X. GU, H. ZHANG, D. ZHANG, AND S. KIM, *Deep api learning*, in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 631–642.

[36] X. GU, H. ZHANG, D. ZHANG, AND S. KIM, *Deepam: Migrate apis with multimodal sequence to sequence learning*, in Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17, AAAI Press, 2017, p. 36753681.

[37] C. GULCEHRE, S. AHN, R. NALLAPATI, B. ZHOU, AND Y. BENGIO, *Pointing the unknown words*, arXiv preprint arXiv:1603.08148, (2016).

[38] D. GUO, S. REN, S. LU, Z. FENG, D. TANG, L. SHUJIE, L. ZHOU, N. DUAN, A. SVYATKOVSKIY, S. FU, ET AL., *Graphcodebert: Pre-training code representations with data flow*, in ICLR, 2020.

[39] K. HE, X. ZHANG, S. REN, AND J. SUN, *Deep residual learning for image recognition*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[40] V. J. HELLENDOORN, C. BIRD, E. T. BARR, AND M. ALLAMANIS, *Deep learning type inference*, in Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, 2018, pp. 152–162.

[41] V. J. HELLENDOORN AND P. DEVANBU, *Are deep neural networks the best choice for modeling source code?*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 763–773.

[42] S. HOCHREITER AND J. SCHMIDHUBER, *Long short-term memory*, Neural computation, 9 (1997), pp. 1735–1780.

[43] X. HU, G. LI, X. XIA, D. LO, AND Z. JIN, *Deep code comment generation*, in Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 200–210.

[44] X. HU, G. LI, X. XIA, D. LO, S. LU, AND Z. JIN, *Summarizing source code with transferred api knowledge*, (2018).

[45] H. HUSAIN, H.-H. WU, T. GAZIT, M. ALLAMANIS, AND M. BROCKSCHMIDT, *Codesearchnet challenge: Evaluating the state of semantic code search*, arXiv preprint arXiv:1909.09436, (2019).

[46] S. IYER, I. KONSTAS, A. CHEUNG, AND L. ZETTLEMOYER, *Summarizing source code using a neural attention model.*, in ACL (1), 2016.

[47] R. JHA, C. LOVERING, AND E. PAVLICK, *When does data augmentation help generalization in nlp?*, arXiv e-prints, (2020), pp. arXiv–2004.

[48] M. JOHNSON, M. SCHUSTER, Q. V. LE, M. KRIKUN, Y. WU, Z. CHEN, N. THORAT, F. VIÉGAS, M. WATTENBERG, G. CORRADO, ET AL., *Googles multilingual neural machine translation system: Enabling zero-shot translation*, Transactions of the Association for Computational Linguistics, 5 (2017), pp. 339–351.

[49] T.-H. JUNG, *Commitbert: Commit message generation using pre-trained programming language model*, arXiv preprint arXiv:2105.14242, (2021).

[50] K. KIM, D. KIM, T. F. BISSYANDÉ, E. CHOI, L. LI, J. KLEIN, AND Y. L. TRAON, *Facoy: a code-to-code search engine*, in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946–957.

[51] S. KIM, J. ZHAO, Y. TIAN, AND S. CHANDRA, *Code prediction by feeding trees to transformers*, in ICSE, IEEE, 2021, pp. 150–162.

[52] Y. KIM, P. PETROV, P. PETRUSHKOV, S. KHADIVI, AND H. NEY, *Pivot-based transfer learning for neural machine translation between non-english languages*, arXiv preprint arXiv:1909.09524, (2019).

[53] G. KOCH, R. ZEMEL, R. SALAKHUTDINOV, ET AL., *Siamese neural networks for one-shot image recognition*, in ICML deep learning workshop, vol. 2, Lille, 2015.

[54] M.-A. LACHAUX, B. ROZIERE, L. CHANUSSOT, AND G. LAMPLE, *Unsupervised translation of programming languages*, arXiv preprint arXiv:2006.03511, (2020).

[55] G. LAMPLE AND A. CONNEAU, *Cross-lingual language model pretraining*, arXiv preprint arXiv:1901.07291, (2019).

[56] T. A. LE, A. G. BAYDIN, AND F. WOOD, *Inference compilation and universal probabilistic programming*, in Artificial Intelligence and Statistics, PMLR, 2017, pp. 1338–1348.

[57] A. LECLAIR, S. HAQUE, L. WU, AND C. MCMILLAN, *Improved code summarization via a graph neural network*, arXiv preprint arXiv:2004.02843, (2020).

[58] M. LEWIS, Y. LIU, N. GOYAL, M. GHAZVININEJAD, A. MOHAMED, O. LEVY, V. STOYANOV, AND L. ZETTLEMOYER, *Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension*, in Proceedings of ACL 2020, 2020, pp. 7871–7880.

[59] J. LI, Y. WANG, M. R. LYU, AND I. KING, *Code completion with neural attention and pointer networks*, arXiv preprint arXiv:1711.09573, (2017).

[60] C. Y. LIN, *Rouge: A package for automatic evaluation of summaries*, Text Summarization Branches Out, (2004).

[61] F. LIU, G. LI, Y. ZHAO, AND Z. JIN, *Multi-task learning based pre-trained language model for code completion*, in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 473–485.

[62] S. LIU, Y. CHEN, X. XIE, J. K. SIOW, AND Y. LIU, *Automatic code summarization via multi-dimensional semantic fusing in gnn*, arXiv preprint arXiv:2006.05405, (2020).

[63] Y. LIU, M. OTT, N. GOYAL, J. DU, M. JOSHI, D. CHEN, O. LEVY, M. LEWIS, L. ZETTLEMOYER, AND V. STOYANOV, *Roberta: A robustly optimized bert pretraining approach*, arXiv preprint arXiv:1907.11692, (2019).

[64] S. LU, D. GUO, S. REN, J. HUANG, A. SVYATKOVSKIY, A. BLANCO, C. CLEMENT, D. DRAIN, D. JIANG, D. TANG, ET AL., *Codexglue: A machine learning benchmark dataset for code understanding and generation*, arXiv preprint arXiv:2102.04664, (2021).

[65] S. LUAN, D. YANG, C. BARNABY, K. SEN, AND S. CHANDRA, *Aroma: Code recommendation via structural code search*, Proceedings of the ACM on Programming Languages, 3 (2019), pp. 1–28.

[66] M.-T. LUONG, Q. V. LE, I. SUTSKEVER, O. VINYALS, AND L. KAISER, *Multi-task sequence to sequence learning*, arXiv preprint arXiv:1511.06114, (2015).

[67] M. T. LUONG, H. PHAM, AND C. D. MANNING, *Effective approaches to attention-based neural machine translation*, arXiv preprint arXiv:1508.04025, (2015).

[68] A. MADOTTO, Z. LIN, C.-S. WU, AND P. FUNG, *Personalizing dialogue agents via meta-learning*, in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019, pp. 5454–5459.

[69] G. NEUBIG AND J. HU, *Rapid adaptation of neural machine translation to new languages*, arXiv preprint arXiv:1808.04189, (2018).

[70] A. NICHOL, J. ACHIAM, AND J. SCHULMAN, *On first-order meta-learning algorithms*, arXiv preprint arXiv:1803.02999, (2018).

[71] M. OTT, S. EDUNOV, A. BAEVSKI, A. FAN, S. GROSS, N. NG, D. GRANGIER, AND M. AULI, *fairseq: A fast, extensible toolkit for sequence modeling*, in Proceedings of NAACL-HLT 2019: Demonstrations, 2019.

[72] K. PAPINENI, S. ROUKOS, T. WARD, AND W. J. ZHU, *Bleu: a method for automatic evaluation of machine translation*, in Proceedings of the 40th annual meeting on association for computational linguistics, Association for Computational Linguistics, 2002, pp. 311–318.

[73] R. PAULUS, C. XIONG, AND R. SOCHER, *A deep reinforced model for abstractive summarization*, arXiv preprint arXiv:1705.04304, (2017).

[74] V. RAYCHEV, P. BIELIK, AND M. VECHEV, *Probabilistic model for code with decision trees*, ACM SIGPLAN Notices, 51 (2016), pp. 731–747.

[75] A. SANTORO, S. BARTUNOV, M. BOTVINICK, D. WIERSTRA, AND T. LILLICRAP, *Meta-learning with memory-augmented neural networks*, in International conference on machine learning, 2016, pp. 1842–1850.

[76] R. SENNRICH, B. HADDOW, AND A. BIRCH, *Neural machine translation of rare words with subword units*, arXiv preprint arXiv:1508.07909, (2015).

[77] P. SHAW, J. USZKOREIT, AND A. VASWANI, *Self-attention with relative position representations*, arXiv preprint arXiv:1803.02155, (2018).

[78] D. SHRIVASTAVA, H. LAROCHELLE, AND D. TARLOW, *On-the-fly adaptation of source code models using meta-learning*, arXiv preprint arXiv:2003.11768, (2020).

[79] I. SUTSKEVER, O. VINYALS, AND Q. V. LE, *Sequence to sequence learning with neural networks*, in Advances in neural information processing systems, 2014, pp. 3104–3112.

[80] K. S. TAI, R. SOCHER, AND C. D. MANNING, *Improved semantic representations from tree-structured long short-term memory networks*, arXiv preprint arXiv:1503.00075, (2015).

[81] A. VASWANI, N. SHAZEER, N. PARMAR, J. USZKOREIT, L. JONES, A. N. GOMEZ, Ł. KAISER, AND I. POLOSUKHIN, *Attention is all you need*, in Advances in neural information processing systems, 2017, pp. 5998–6008.

[82] O. VINYALS, C. BLUNDELL, T. LILLICRAP, D. WIERSTRA, ET AL., *Matching networks for one shot learning*, Advances in neural information processing systems, 29 (2016), pp. 3630–3638.

[83] Y. WAN, J. SHU, Y. SUI, G. XU, Z. ZHAO, J. WU, AND P. S. YU, *Multi-modal attention network learning for semantic source code retrieval*, in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 13–25.

[84] Y. WAN, Z. ZHAO, M. YANG, G. XU, H. YING, J. WU, AND P. S. YU, *Improving automatic source code summarization via deep reinforcement learning*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 397–407.

[85] H. H. WEI AND M. LI, *Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code*, (2017).

[86] M. XIA, X. KONG, A. ANASTASOPOULOS, AND G. NEUBIG, *Generalized data augmentation for low-resource translation*, arXiv preprint arXiv:1906.03785, (2019).

[87] B. ZOPH, D. YURET, J. MAY, AND K. KNIGHT, *Transfer learning for low-resource neural machine translation*, arXiv preprint arXiv:1604.02201, (2016).