UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2025-117
Faculty of Science, Technology and Medicine

Dissertation

Presented held on the 29th September 2025 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Weiguo PIAN
Born on 25th October 1996 in Hebei, China

Representation Learning of Software Source Code

Abstract

The growing complexity of modern software systems has made automated tools for code understanding, maintenance, and analysis increasingly essential. As source code exhibits both natural language and formal structural properties, learning effective representations that capture these multi-faceted characteristics is crucial for enabling intelligent software engineering. This thesis focuses on advancing the field of representation learning for software source code by proposing three novel methods across multilingual code modeling, joint localization and editing of source code, and Large Language Models enhancement for cross-lingual code clone detection.

In the first part of this thesis, we address the challenge of multilingual code representation learning by proposing MetaTPTrans, a meta-learning based approach that conditions the parameter generator on the programming language type to dynamically generate encoder parameters. This design allows the model to capture both language-agnostic semantics and language-specific patterns. Through extensive experiments on multilingual code summarization and completion tasks, we demonstrate that MetaTPTrans outperforms existing code representation learning models significantly.

In the second part, we investigate the task of source code editing and propose jLED (Joint Learning to Localize and Edit), a framework that unifies edit location prediction and code editing into a single training objective. Unlike prior approaches that assume the location of edits is known during inference, jLED is trained with supervision for both localization and editing, but is able to automatically predict edits locations and generate corresponding edits code without any location input at inference time. To facilitate this, we construct a large-scale dataset of over 70,000 real-world samples, which provides a new benchmark for learning to localize and edit code jointly. Experiments show that jLED substantially improves both localization accuracy and edit effectiveness, offering a practical solution for automated code editing.

In the third part, we address the task of cross-lingual code clone detection by enhancing the effectiveness of fine-tuning large language models (LLMs). Our method augments the input during fine-tuning with code summaries, providing natural language context that guides the model toward better semantic understanding of code across different programming languages. Extensive experiments across multiple open-source LLMs show that this simple yet effective strategy consistently improves performance compared to both zero-shot and standard fine-tuning baselines. The results demonstrate that incorporating high-level code descriptions can significantly boost the alignment of semantically similar code snippets, achieving new state-of-the-art results on a multilingual code clone detection benchmark.

In summary, the methodologies developed in this thesis advance the field of source code representation learning across multiple dimensions. We first investigate the challenges of multilingual modeling and propose a meta-learning-based approach to improve multilingual code representation learning. After that, we further explore how learned representations can be effectively applied to downstream tasks such as automated code editing and cross-lingual clone detection. These works demonstrate that rich and adaptable code representations can significantly benefit practical software engineering problems. Overall, this thesis contributes new insights and techniques toward more intelligent, language-aware, and scalable code intelligence systems.

Acknowledgements

I would like to express my heartfelt gratitude to everyone who contributed to making this dissertation possible. Many people contributed to my Ph.D. journey deeply, generously sharing their knowledge, advice, experience, and engaging in valuable discussions.

First of all, I would like to express my sincere appreciation to my supervisor, Prof. Tegawendé F. Bissyandé, for giving me the opportunity to pursue my Ph.D. degree, supporting me in exploring every possibility, engaging with me in insightful discussions not only on academic research but also providing invaluable advice for my personal life.

Second, I would also like to express my heartfelt gratitude to my co-supervisor, Prof. Jacques Klein, for his trust, invaluable advice, and unwavering guidance throughout my research journey, as well as for his encouragement and insightful feedback that have greatly enriched my work.

Third, I also want to express my appreciation to the members of my Ph.D. defense committee: Prof. Tegawendé F. Bissyandé, Prof. Jacques Klein, Prof. Anil Koyuncu, Prof. Haoye Tian, and Prof. He Ye. It is a great honor to have had the opportunity to present my research before such a distinguished committee, and I am truly grateful for their valuable time, insightful feedback, and constructive guidance.

I am also deeply grateful to my coauthors — Dr. Tiezhu Sun, Dr. Xunzhu Tang, Prof. Haoye Tian, Dr. Yinghua Li, Yewei Song, and others — for their invaluable discussions and collaborations throughout my research. I would also like to thank all the members of TruX and my friends in the Grand Duchy of Luxembourg for their support and for all the memorable moments we shared during my Ph.D. journey.

Finally, I would like to express my deepest appreciation to my wife, Weiwei Zhao, for her endless support, encouragement, and love, which always comforts me in moments of anxiety and confusion, and gives me the strength and courage to embrace all the uncertainties and challenges in life. Also, I can never overstate how much I owe to my mother, Fengxia Ren, my father, Xuemin Pian, and my sister, Weihua Pian, whose boundless love and selfless devotion have given me everything they could and supported me in every step of my journey.

Weiguo Pian
University of Luxembourg
September 2025

Contents

List of Figures

List of Tables

1 Introduction

In this first chapter, we begin by introducing the motivation behind learning effective representations of software source code, which has become increasingly important for intelligent software engineering. We focus on three core applications: multilingual code modeling, automated code editing, and cross-lingual code clone detection. Next, we summarize the key challenges in each of these tasks and discuss the limitations of existing approaches. Finally, we outline the main contributions of this dissertation and present an roadmap of its organization.

Contents

1.1 Motivation

As software systems become increasingly complex, diverse, and multilingual, understanding and manipulating source code has emerged as a key challenge in modern software engineering. To address this, the research community has devoted significant attention to developing machine learning techniques, particularly deep learning models, that can automatically learn effective representations of source code to support a wide range of downstream tasks, such as code summarization [3], code translation [67, 167], code editing [16, 91, 15], clone detection [100, 101, 44, 103], etc. Unlike natural language, source code poses unique modeling challenges. It is syntactically strict, semantically rich, and structurally complex. For instance, programs involve long-range dependencies, such as variable usage across functions, and are often organized hierarchically in abstract syntax trees (ASTs) or control-flow graphs (CFGs).

Furthermore, the multilingual nature of real-world codebases—spanning languages like Java, Python, C++, and JavaScript—requires models to generalize across different syntax, paradigms, and idioms. While early methods such as token-based embeddings and syntax-aware models [8, 7, 178, 108] made notable progress, they often fall short in capturing both language-agnostic semantics and language-specific nuances. To tackle the multilingual challenge, we introduce MetaTPTrans [112], a meta-learning framework that conditions encoder parameters on the programming language. This design enables the model to dynamically adapt its representations to both language-agnostic semantics and language-specific syntax. Experimental results show that this approach significantly improves cross-lingual code understanding on tasks such as summarization and code completion, demonstrating the importance of conditional parameterization in multilingual code modeling.

Another key application of code representation learning is automated source code editing, which is essential for software maintenance and repair. Recent works often rely on directly fine-tuning pre-trained models such as CodeBERT [27], GraphCodeBERT [34], PLBART [2], etc, which are pre-trained on large corpora to capture high-level code semantics. While these models perform well in many tasks, they are typically applied to editing scenarios by assuming that the location of the code line to be edited is already known, and this information is provided explicitly during inference—often by marking the target line in the input [16]. However, such an assumption does not hold in practice, as real-world systems rarely offer oracle edit locations. This disconnect limits the practicality of existing approaches. Our recent work [111] addresses this gap by introducing a unified framework that jointly learns to localize and edit source code, removing the dependency on ground-truth location inputs and improving both robustness and applicability in real-world editing tasks.

Semantic code clone detection, particularly in the cross-lingual Type-4 setting, presents another major challenge for code representation learning. Type-4 clones refer to functionally equivalent code snippets that may differ substantially in syntax, structure, or even programming language. Detecting such clones is vital for software maintenance, code reuse, and plagiarism detection. However, traditional approaches—including token-based [44], AST-based [103], and graph-based models—often rely heavily on syntactic similarity, making them less effective in cross-lingual scenarios. Recently, researchers have explored using large language models (LLMs) for this task in a zero-shot manner [100, 101], prompting pre-trained models to directly judge functional equivalence. Despite their strong generalization capabilities, such approaches frequently struggle with nuanced semantic reasoning and lack explicit alignment signals across different languages. To address these limitations, our recent work proposes a summarization-based input augmentation strategy that enriches the fine-tuning process of LLMs. By incorporating natural language descriptions of code functionality, we provide an additional semantic bridge that helps the model reason about cross-lingual equivalence beyond surface-level syntax. Extensive experiments across multiple LLMs and ten programming languages demonstrate that this method consistently improves clone detection performance, achieving state-of-the-art results on multilingual clone detection benchmarks.

In summary, this dissertation addresses three fundamental challenges in source code representation learning: (1) the need for adaptable multilingual modeling that captures both shared semantics and language-specific patterns, (2) the lack of realistic end-to-end frameworks for source code editing that do not rely on oracle bug locations, and (3) the difficulty of detecting semantically equivalent code across languages due to syntactic and structural divergence. By proposing novel solutions to each of these problems—MetaTPTrans for multilingual modeling, jLED for joint localization and editing, and summarization-enhanced LLMs for cross-lingual clone detection—this dissertation advances the development of more generalizable, practical, and semantically grounded code intelligence systems.

1.2 Challenges and Limitations of Existing Approaches

In this section, we present the limitations and challenges of existing approaches to code representation learning, with a focus on three key tasks: multilingual modeling, source code editing, and cross-lingual code clone detection. These tasks highlight critical aspects of intelligent code understanding and manipulation that remain insufficiently addressed by current techniques.

1.2.1 Challenges in Multilingual Code Representations Learning

Training effective models on multilingual code data remains a central challenge in multilingual code representation learning. While multilinguality is a growing reality in large-scale codebases—where systems are often composed of modules written in different languages such as Java, Python, C++, and JavaScript—most existing models struggle to balance the shared and language-specific aspects of code semantics. Naïvely combining multilingual data during training can, to some extent, improve monolingual performance by enabling models to capture more language-agnostic semantic patterns [178]. However, such training often overlooks critical language-specific properties, such as language-specific syntax rules, which are essential for accurate understanding and generation within each language.

Several recent models have explored training on multilingual code to improve generalization, including code2seq [7], GREAT [39], CodeTransformer [178], and TP-Trans [108]. These models show that jointly training on multilingual data can indeed lead to improved performance over single-language training by leveraging shared semantic patterns across languages. However, they generally rely on shared encoders and do not explicitly incorporate language-specific modeling mechanisms. As a result, while they successfully capture language-agnostic semantics, they tend to overlook critical language-specific properties such as syntactic rules, etc. This limitation becomes more pronounced when dealing with low-resource languages or languages with vastly different structural paradigms.

Another major challenge arises from data imbalance in multilingual corpora. Certain languages such as Python and Java are overrepresented in public datasets, while others like Ruby or Go are underrepresented. Shared-parameter models tend to bias toward these high-resource languages during training, often at the expense of performance on low-resource languages. This makes multilingual training fragile and difficult to scale, especially in real-world scenarios where language distribution is uneven.

These challenges highlight the limitations of current multilingual training paradigms. There remains a need for approaches that go beyond shared encoders and data scaling, by explicitly incorporating language awareness into model design. Mechanisms such as parameter modulation, language embeddings, or meta-learning strategies offer promising directions to better capture both cross-lingual semantics and per-language intricacies. Our first contribution addresses these needs by proposing such a mechanism, which we discuss in detail in later sections.

1.2.2 Challenges in Source Code Editing

Source code editing is a core task in automated software maintenance, encompassing bug fixing, refactoring, and program repair. A large body of work has approached this task by dividing it into two subproblems: (1) identifying the location of the bug or edit span, and (2) generating the fix. While this decomposition simplifies the modeling process, it introduces unrealistic assumptions and leads to cascading errors. In practice, state-of-the-art method [16] relies on training and inference with oracle line locations. This is typically done by marking the line locations to be edited within the input. While this setting facilitates learning during training, it poses significant limitations during deployment: in real-world scenarios, developers may not annotate buggy or editing lines in advance, and tools must localize the editing lines automatically.

The reliance on oracle edit locations results in two major problems. First, it limits the applicability of these models outside controlled benchmarks, where precise localization annotations are available. Second, it breaks the end-to-end nature of the task, requiring additional models or heuristics to supply the location during inference. These add-ons introduce noise and propagate errors—if localization fails, even a strong edit generator will produce irrelevant patches. Some works attempt to localize and edit separately using pipeline designs, but suffer from cumulative loss in accuracy and efficiency. To avoid the dependency on localization modules, works explore end-to-end editing models that directly generate the edited code without explicitly identifying the editing location [73, 167]. While this simplifies the workflow, it often results in models that lack transparency, interpretability, and fine-grained control, and they still tend to generate edits with less accuracy at the line level. More critically, these models tend to underperform in cases involving subtle or localized changes, as the model receives no focused guidance on which region of the input needs modification.

Furthermore, decoupled modeling overlooks the interaction between localization and editing. In practice, where and how to edit are deeply intertwined: understanding the type of fix may provide clues about the editing location, and vice versa. A unified approach that models both jointly is therefore more natural and potentially more accurate. However, such methods are underexplored, and few benchmarks exist that support joint supervision for location and editing prediction. Developing models that can learn to localize and edit simultaneously—without relying on hard-coded location inputs—remains a crucial open challenge in making automated editing truly usable in real-world systems.

1.2.3 Challenges in Cross-lingual Code Clone Detection

Cross-lingual Type-4 code clone detection is a particularly challenging task that requires identifying semantically equivalent code snippets written in different programming languages and with dissimilar syntax. It is especially useful for software reuse, multilingual maintenance, and detecting plagiarism across diverse codebases. Unlike Type-1 through Type-3 clones, which involve varying degrees of syntactic similarity, Type-4 clones demand deep semantic understanding.

Traditional clone detection techniques, such as token-based [44], AST-based [103], etc, have limited capacity for semantic abstraction and generalization across languages. Their heavy reliance on structural similarity and static code features prevents them from capturing high-level intent when the implementation details differ substantially. Even when augmented with pre-trained embeddings, these models often fall short when confronted with multilingual inputs or abstract algorithmic equivalence.

To overcome these limitations, recent studies have proposed leveraging large language models (LLMs) in a zero-shot setting [100, 101]. These approaches prompt models like CodeLlama, StarCoder, etc, to directly reason about the semantic similarity between code snippets in different languages. While promising in concept, such prompting-based methods remain brittle in practice. They often rely on surface-level patterns, are sensitive to prompt phrasing, and lack consistent performance across different language pairs or code lengths. More importantly, these methods operate without any external semantic grounding, making it difficult for the models to align code snippets that implement the same logic in different ways. For instance, an iterative loop in C++ and a recursive function in Python may express the same functionality, but an LLM prompted only with the raw code may fail to detect their equivalence. Without additional semantic signals, the model lacks sufficient inductive bias to perform robust cross-lingual reasoning.

This highlights the need for techniques that can inject higher-level semantic information, such as natural language descriptions of code behavior, into the representation learning process. By providing models with a concise summary of each snippet's intent, such augmentation may help bridge the gap between differing syntactic structures and reveal deeper equivalence. However, incorporating these auxiliary signals into LLM training in an efficient, consistent, and generalizable way remains an open research question. Solving this challenge is critical to advancing the state of the art in multilingual semantic clone detection.

1.3 The Main Contributions

In this section, we summarize the contributions of this dissertation.

- **MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning** [112]: Representation learning of source code is essential for applying machine learning to software engineering tasks. Learning code representation from a multilingual source code dataset has been shown to be more effective than learning from single-language datasets separately, since more training data from multilingual dataset improves the model's ability to extract language-agnostic information from source code. However, existing multilingual training overlooks the language-specific information which is crucial for modeling source code across different programming languages, while only focusing on learning a unified model with shared parameters among different languages for language-agnostic information modeling. To address this problem, we propose MetaTPTrans, a meta learning approach for multilingual code representation learning. MetaTPTrans generates different parameters for the feature extractor according to the specific programming language type of the input code snippet, enabling the model to learn both language-agnostic and language-specific information with dynamic parameters in the feature extractor. We conduct experiments on the code summarization and code completion tasks to verify the effectiveness of our approach. The results demonstrate the superiority of our approach with significant improvements on state-of-the-art baselines.

- **You Don't Have to Say Where to Edit! jLED – Joint Learning to Localize and Edit Source Code** [111]: Learning to edit code automatically is becoming more and more feasible. Thanks to recent advances in Neural Machine Translation (NMT), various case studies are being investigated where patches are automatically produced and assessed either automatically (using test suites) or by developers themselves. An appealing setting remains when the developer must provide a natural language input of the requirement for the code change. A recent proof of concept in the literature showed that it is indeed feasible to translate these natural language requirements into code changes. A recent advancement, MODIT [16], has shown promising results in code editing by leveraging natural language, code context, and location information as input. However, it struggles when location information is unavailable. While several studies [167, 73] have demonstrated the ability to edit source code without explicitly specifying the edit location, they still tend to generate edits with less accuracy at the line level. In this work, we address the challenge of generating code edits without precise location information, a scenario we consider crucial for the practical adoption of NMT in code development. To that end, we develop a novel joint training approach for both localization and source code editions. Building a benchmark based on over 70k commits (patches and messages), we demonstrate that our jLED (**j**oint **L**ocalize and **ED**it) approach is effective. An ablation study further demonstrates the importance of our design choice in joint training.
- **Enhancing LLMs for Cross-lingual Code Clone Detection with Code Summarization** Cross-lingual Type-4 code clone detection, which aims to identify semantically equivalent code snippets written in different programming languages, poses significant challenges due to syntax divergence and the need for deep semantic reasoning. In this work, we explore the effectiveness of Large Language Models (LLMs) for this task under both zero-shot and fine-tuned settings. We systematically evaluate multiple state-of-the-art LLMs, including both general-purpose and code-specialized models, across ten programming languages. To further enhance fine-tuning performance, we propose a summarization-based input augmentation strategy that provides natural language descriptions of code functionality. These summaries help bridge the semantic gap between syntactically diverse code snippets by offering

a language-agnostic representation of program intent. They are incorporated alongside raw code and prompt templates during fine-tuning. Experimental results on the XLCoST and CodeNet datasets demonstrate that fine-tuning leads to substantial gains over zero-shot performance, and that our augmentation method consistently improves Accuracy, F1-score, Precision, and Recall across all evaluated models. Our study demonstrates the strong potential of LLMs in multilingual code understanding and offers a generalizable framework for semantic clone detection across different programming languages.

1.4 Roadmap

Figure 1.1 illustrates the roadmap of this dissertation. In Chapter 2, we present the necessary background and related work, covering code representation learning, code editing, and code clone detection. In Chapter 3, we introduce MetaTPTrans, a meta-learning-based framework designed to improve multilingual code representation learning by conditioning the encoder on language types. Chapter 4 presents jLED, a unified approach for jointly localizing and editing source code without relying on ground-truth location inputs, enhancing robustness in real-world scenarios. Chapter 5 explores the use of large language models for cross-lingual code clone detection, and proposes augmenting model inputs with natural language summaries to better capture semantic equivalence across programming languages. Finally, Chapter 6 concludes this dissertation and discusses potential directions for future research.

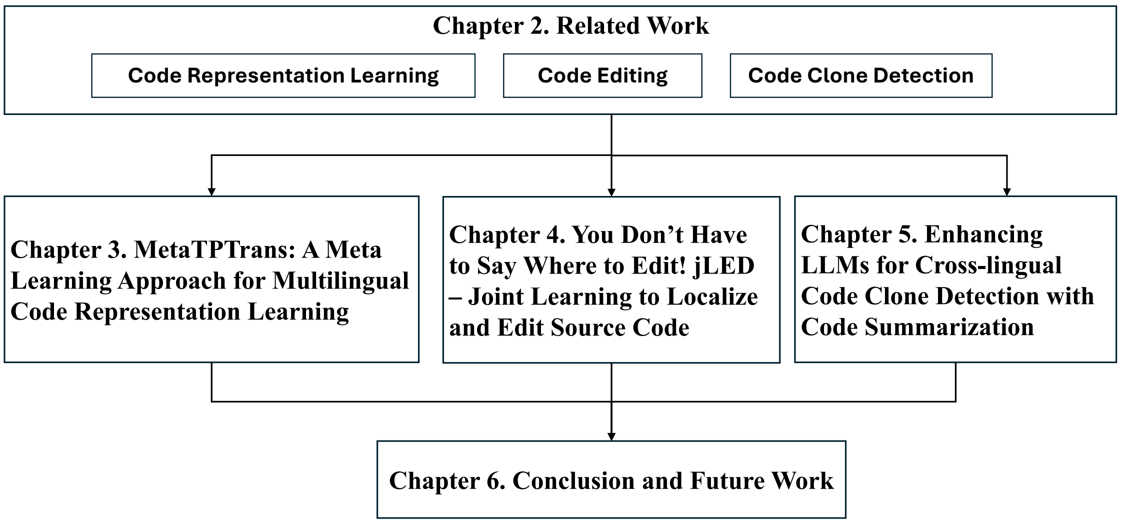

Chapter 2. Related Work
Code Representation Learning
Code Editing
Code Clone Detection
Chapter 3. MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning
Chapter 4. You Don't Have to Say Where to Edit! jLED – Joint Learning to Localize and Edit Source Code
Chapter 5. Enhancing LLMs for Cross-lingual Code Clone Detection with Code Summarization
Chapter 6. Conclusion and Future Work

Figure 1.1: Roadmap of this dissertation

2 Related Work

In this chapter, we review key areas of research that are closely related to our work. We begin by examining the foundations of code representation learning. Next, we discuss existing approaches to automated code editing. Finally, we introduce representative works in the area of code clone detection.

Contents

2.1 Code Representation Learning

2.1.1 Learning Representation of Source Code

Source code representation learning has seen many developments. Early works mainly focus on learning language models from raw token sequences [148, 19, 6, 51]. More recent works explore the effectiveness of leveraging structural information to model source code. Mou *et al.* [99] apply the convolutional operation on ASTs to extract structure features to represent source code. Alon *et al.* [8, 7] extract paths from ASTs and use RNNs to encode them to represent the source code. Some works [5, 28, 172] use Graph Neural Networks to capture the structural information from carefully designed code graphs. On the other hand, Transformer-based models are also used to represent source code by capturing both context and structural information, in which the structural information is integrated into the self-attention module by replacing the position embedding with encoding from AST [39, 178, 108]. Specifically, Hellendoorn *et al.* [39] bias the self-attention process with different types of correlations between nodes in code graph. Zügner *et al.* [178] use several pair-wise distances on ASTs to represent the pair-wise relationships between tokens in code context sequence and find that multilingual training improves the performance of language models compared to single-language models. Peng *et al.* [108] encode AST paths and integrate them into self-attention to learn both context and structural information. Compared to these works, ours is the first to use meta learning to learn multilingual source code models that are capable of learning language-specific in addition to language-agnostic information and improves on several of the aforementioned models yielding state-of-the-art results.

2.1.2 Meta Learning for Parameters Generation

Meta learning is a novel learning paradigm with the concept of learning to learn. There are several types of meta learning such as learning to initialize [29], learning the optimizer [9], and learning hyperparameters [72]. The most related meta learning method with ours is learning to generate model's parameters. Bertinetto *et al.* [11] propose a method called learnet to learn to generate the parameters of the pupil network. Ha *et al.*[37] propose Hypernetworks to generate parameters for large models through layer-wise weight sharing scheme. Chen *et al.* [17] propose a meta learning-based multi-task learning framework in which a meta network generates task-specific weights for different tasks to extract task-specific semantic features. Wang *et al.* [149] use a task-aware meta learner to generate parameters for classification models for different tasks in few-shot learning. Pan *et al.* [105, 106] apply meta learning to generate parameters for models in spatial-temporal data mining to capture spatial and temporal dynamics in urban traffic. Our approach is a form of meta learning to generate the model's parameters according to the programming language type of the input code snippet.

2.2 Code Editing

2.2.1 Automatic Code Change

Thanks to the repetitiveness of code editing, researchers have proposed to automate several code change tasks in the field of software engineering. One research direction aims to refactor existing code without changing its functionality [62, 95, 31]. For example, Meng *et al.* [95] introduced RASE, a highly advanced automated refactoring tool designed to eliminate redundancy in software code through clone removal. The evaluation showed that RASE successfully removes clones in a significant number of method pairs and groups with systematic edits, indicating the increased applicability of automated refactoring based on these edits. Khatchadourian *et al.* [62] transformed legacy Java code to leverage the new enumeration construct, improving type safety, code comprehension, simplicity, and eliminating brittleness issues. It employs an interprocedural type inferencing algorithm and has been evaluated on 17 Java benchmarks. Other research direction addresses the completion or suggestion codes automatically [130, 131, 70, 118]. Svyatkovskiy *et al.* [130] proposed IntelliCode Compose, a versatile code completion tool capable of generating syntactically correct code sequences and entire lines in multiple programming languages. Leveraging a generative transformer model trained on extensive source code, IntelliCode Compose achieves high edit similarity and low perplexity for edit-time completion suggestions in Visual Studio Code IDE and Azure Notebook.

Another direction that is widely recognized as significant is the automation of bug detection and correction. With recent advances in deep learning, researchers proposed to use of NMT architecture or pre-trained encoder or decoder for program repair [16, 15, 137, 56, 36, 127, 163]. Our study is related to the last thrust. The evaluations conducted on the tasks showed promising results in automatic code change. Nevertheless, we argue that previous studies often assume the availability of a perfect edit location, such as fault location in program repair, which is not typically the case in real-world scenarios. Our objective is to fill this gap by jointly learning the localization and editing of source code.

2.2.2 Code Change Modeling

The code change modeling plays a crucial role in code-related tasks [27, 25, 41, 20, 22, 132]. To learn distributed representations, Hoang *et al.* [41] utilized deep learning models to create CC2Vec, an approach of representing patches through sequence learning on code change. CC2Vec was evaluated as effective as the state of the arts on three patch tasks: generating patch descriptions, identifying bug-fixing patches, and just-in-time defect prediction. Similarly, CoDiSum [157] is a token-based approach to patch representation, particularly useful in generating patch descriptions. Moreover, Tufano *et al.* [141, 142] investigated the usage of NMT for general-purpose code changes learning. Recently, large pre-trained code models have been further re-pre-trained on code change datasets to obtain large pre-trained code change models [171, 84, 75].

The utility and adaptability of large-scale language models (LLMs) extend beyond natural language processing tasks. The BERT architecture [21], for instance, encodes both the left and right context of a token, making it adapted for the tasks of interpretation and generation of code that often relies heavily on the surrounding context. Several studies have revealed the efficacy of these models in handling source code. CodeBERT [27] and GraphCodeBERT [34], both derivatives of the BERT architecture, have shown considerable promise in understanding code semantics. CodeBERT is designed to tackle programming tasks by learning from bilingual data, including natural language and code, while GraphCodeBERT incorporates structural information from code into the pre-training process for a better understanding of the dependencies within the code. Furthermore, PLBART [2] and CodeT5 [150] have demonstrated substantial effectiveness in code translation and summarization tasks. PLBART, an encoder-decoder model, is specifically designed for programming language tasks. It leverages a large-scale bilingual corpus to learn a unified representation that captures the semantics of code. CodeT5, yet another encoder-decoder model, extend the T5 model by incorporating a code tokenizer and a new pre-training objective, making it successful in the task of code defect detection and clone detection.

Because of the outstanding representation ability of these pre-trained models on source code, they have also been widely adapted to various code change tasks such as program repair [138, 133, 134, 165], commit message generation [139, 124, 57] or code recommendation [170, 143]. In our proposed methodology, we initially harness the power of pre-trained LLMs as a fundamental architecture to represent both code and natural language. Subsequently, the weights of the LLMs are jointly optimized to learn to localize and repair bugs effectively. This approach allows the model to continually evolve and adapt, thereby enhancing its capability to perform code change tasks.

2.3 Code Clone Detection

2.3.1 Advances in Code Clone Detection

Code clone detection has evolved significantly, transitioning from traditional syntactic approaches to sophisticated methods incorporating deep learning and large language models. Early tools like CCFinder et al. [58] and Deckard et al. [54] laid the groundwork by employing token-based and AST-based analyses, respectively. Building upon these foundations, Hu et al. [44] introduced a fine-grained tree-based syntactic clone detector that transforms complex ASTs into simplified subtrees, enhancing detection accuracy and scalability.

The integration of graph-based representations has further improved clone detection. Gitor [122] constructs a global sample graph to capture underlying connections among code samples, leveraging node embedding techniques to enhance detection performance. Similarly, CloneXformer [103] employs a transformer-based framework that collaborates multiple code representations, achieving interpretable and accurate clone detection. Incorporating deep learning, CCDive [32] utilizes local context-aware embeddings to detect a wide range of clones, demonstrating the effectiveness of contextual information in clone detection tasks. Additionally, the application of unsupervised similarity measures [92] offers an alternative approach by assessing code similarity without labeled data, providing flexibility in various scenarios.

The advent of LLMs has introduced new dimensions to clone detection. Moumoula et al. [100] evaluated the performance of five LLMs and eight prompts for cross-lingual code clone detection, revealing that while LLMs achieve high F-1 scores on straightforward examples, they struggle with complex programming challenges and do not necessarily understand the meaning of “code clones” in a cross-lingual setting. Zhang and Saber [169] assessed GPT-3.5 and GPT-4, finding that GPT-4 consistently surpasses GPT-3.5 across all clone types but exhibits low effectiveness in detecting complex Type-4 code clones.

2.3.2 Large Language Models for Code-Ralated Tasks

Large Language Models have significantly advanced code summarization by generating human-readable descriptions of code snippets, aiding in code comprehension and documentation [128, 4, 153]. Sun et al.[128] conducted a comprehensive study on LLM-based code summarization, evaluating various automated methods and highlighting that GPT-4's evaluations closely align with human judgments. They also explored different prompting techniques, finding that simple zero-shot prompting often outperforms more complex methods. Ahmed et al.[4] proposed the Automatic Semantic Augmentation of Prompts approach, enhancing LLM prompts with semantic facts extracted from code, such as identifier roles and control flow information, achieving significant improvements in BLEU scores across multiple programming languages. Wu et al. [153] introduced CODERPE, a novel evaluation method leveraging role-playing prompts to assess the quality of generated summaries, showing a high correlation with human evaluations and outperforming traditional metrics like BERTScore.

LLMs have become effective tools for automated program repair by generating patches for buggy code, reducing manual debugging effort [160, 126, 89, 88]. MORepair [160] introduces a multi-objective fine-tuning framework that optimizes both syntactic correctness and the logic of code changes. It achieved significant improvement in Top-10 repair suggestions across C++ and Java, outperforming baselines like Fine-tune-CoT and RepairLLaMA [126]. To improve repair in low-resource languages, LANTERN [89], as a cross-language repair approach, uses LLMs' stronger performance in high-resource languages via multi-agent iterative refinement. This strategy led to a 22% Pass@10 gain for Rust. To address privacy concerns in fine-tuning with proprietary code, Luo et al. [88] applied federated learning, showing that it enhances repair performance without exposing sensitive data, and is robust to heterogeneous code distributions. These works expand LLM repair capabilities through fine-tuning, cross-language transfer, and privacy-aware learning.

In the realm of code generation, LLMs have also demonstrated the ability to translate natural language descriptions into executable code [53, 80, 74]. A survey [53] delved into how LLMs empower users to generate code, discussing limitations and challenges in automated code generation. Liu et al.[80] introduced EvalPlus, an enhanced evaluation framework that extends existing benchmarks with extra test cases to better assess functional correctness, revealing that previous evaluations often overestimate model performance. Li et al.[74] addressed the robustness of generated code, identifying issues like missing input checks and proposing RobGen, a framework to improve robustness without retraining.

3 MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning

In this chapter, we introduce the MetaTPTrans approach, a meta learning approach for multilingual code representation learning, which generates different parameters for the feature extractor according to the specific programming language type of the input code snippet, enabling the model to learn both language-agnostic and language-specific information with dynamic parameters in the feature extractor.

This chapter is based on the work published in the following research paper:

- Pian, W., Peng, H., Tang, X., Sun, T., Tian, H., Habib, A., Klein, J. and Bissyandé, T.F., 2023, June. MetaTPTrans: A meta learning approach for multilingual code representation learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 37, No. 4, pp. 5239-5247).

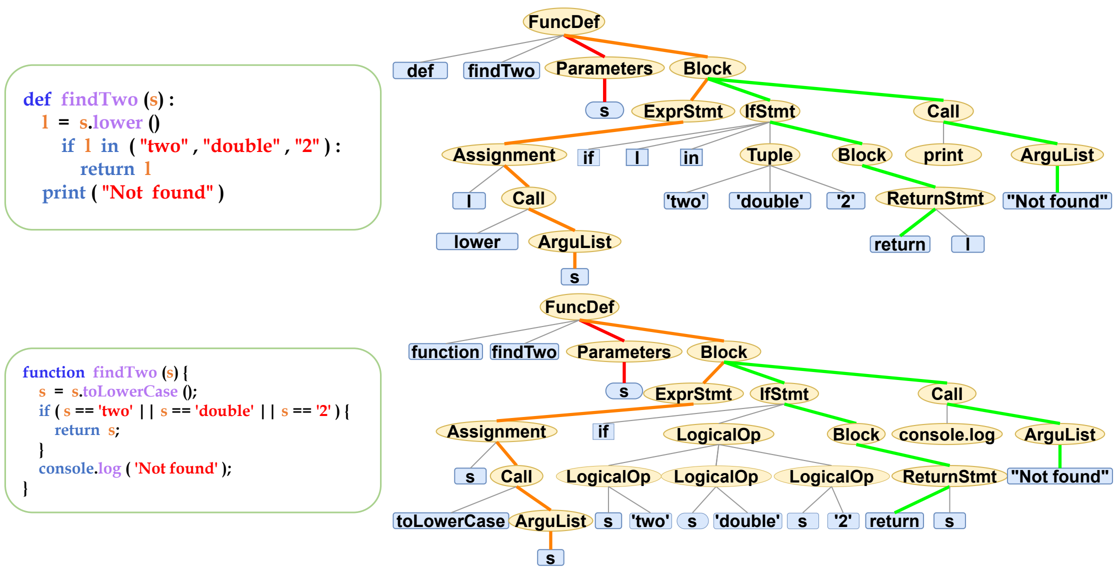
Contents

3.1 Introduction

Modeling source code aims at capturing both its syntax and semantics to enable applying machine learning to software engineering tasks such as code summarization [178, 108], code completion [76, 78], bug finding [148, 113], patch correctness assessment [133, 135], etc. Inspired by the success of deep learning in the field of natural language processing (NLP) [42, 145], modeling source code with deep learning techniques has attracted increasing attention from researchers in recent years [7, 8, 39, 63]. Although programs are more repetitive and predictable (i.e. "natural" [40]), unlike natural language text, they also contain rich structural information, e.g., ASTs, and data- and control-flow information. Therefore, a straightforward adoption of NLP models to source code struggles to capture structural information from the source code context (sequence of tokens).

To alleviate the problem above, Alon *et al.* [7, 8] proposed to incorporate the structural information by encoding pairwise paths on AST via LSTMs [42] to represent the source code. Inspired by the success of Graph Neural Networks (GNN) [65, 38, 146, 30] in modeling structural data, several works leverage GNNs on ASTs to capture the program structure. For instance, Allamanis *et al.* [5] used Gated Graph Neural Networks (GGNN) to learn program representation over ASTs and data flow graphs. Based on GGNN and programs graphs, Fernande *et al.* [28] proposed Sequence GNN for code summarization and Zhou *et al.* [172] proposed a GNN-based approach to detect software vulnerabilities. However, GNN-based methods struggle to extract global structural information since GNNs focus on local message passing when aggregating information across nodes. Besides, the aforementioned approaches focus on structural information and ignore the contextual information when extracting features from source code. To model both structural and contextual information, Hellendoorn *et al.* [39] combined the two kinds of information through Transformers [145] by biasing the self-attention process with relation information extracted from graph edge types. Zügner *et al.* [178] proposed to bias the computation of self-attention with multiple kinds of pairwise distances between AST nodes and integrate them into the XLNet model [162]. Lastly, Peng *et al.* [108] introduced TPTrans, which biases the attention processes in Transformers with the encoding of relative and absolute AST paths.

In a different direction, Zügner *et al.* [178] introduced their novel insight that multilingual training improves the performance of the model compared to training models on single-language datasets separately, since training the model with multilingual source code data enhances the model's ability to learn language-agnostic information [178]. The language-agnostic information is introduced in [178], which means that the information can be directly extracted from the source code or AST by a unified model across different languages, without relying on any language-specific features [178]. Such as structural information from the AST of the code, since these ASTs have a unified structural (tree) form across different programming languages (different programming languages have a unified rule of constructing an AST from the source code, and the structural modeling of ASTs is essentially to make the model understand such unified structural rule of them). In addition to language-agnostic information, other information is defined as language-specific information, e.g., the language-specific underlying syntactic rules, the language-specific API names, etc. As an example of language-agnostic information, Figure 3.1 shows two code snippets of



```
def findTwo (s) :
  l = s.lower ()
    if l in ("two","double","2") :
      return l
  print ( "Not found" )
```

```
function findTwo (s) {
  s = s.toLowerCase ();
  if ( s == 'two' | | s == 'double' | | s == '2' ) {
    return s;
  }
  console.log ( 'Not found' );
}
```

Figure 3.1: The same code snipped and its associated AST in Python (top) and JavaScript (bottom).

the same function in Python (top of the figure) and JavaScript (bottom of the figure). The two ASTs of the code snippets are shown on the right hand side of the respective figures. We can obviously see that the two ASTs have a unified structural (tree) form, besides, we also observe that they have some shared paths, e.g., the highlighted paths in red, orange, and green. Because of such language-agnostic information which can be extracted by a unified model directly across different languages, multilingual training shows superior performance compared to training models on single-language datasets separately.

Although the multilingual training strategy improves the model's performance significantly, it ignores the language-specific information hidden in the characteristics of each specific language. Considering the same example shown in Figure 3.1, we see that the context of the code snippets (the sequence of source code tokens) in different languages carries distinctive language-specific features, e.g., the different coding rules of the code context in different languages (language-specific underlying syntactic rules), the language-specific API names in different languages: `lower()` and `print()` in Python vs. `toLowerCase()` and `console.log()` in JavaScript. For the existing multilingual training, it struggles to learn such language-specific syntactic rules and language-specific token representation/projection with a unified model, *since a unified model is more inclined to learn the common characteristics of the input data*, i.e., the language-agnostic information under the scenario of multilingual code modeling.

To address this problem, in this paper, we propose MetaTPTrans, a meta learning based approach for multilingual code representation learning. Meta learning is a novel learning paradigm with the concept of learning to learn. There are several forms of learning to learn in meta learning, such as learning to initialize [29], learning the optimizer [9], learning hyperparameters [72], learning to generate parameters of the feature extractor by another network [11, 37, 17, 149, 105, 106]. *The meta learning form we applied is the last one, i.e., learning to generate the model's parameters, in which the feature extractor can be named as Base Learner while the network for generating parameters is named as Meta Learner* [149, 105, 106]. With this form of meta learning, our model can adjust its parameters dynamically regarding the programming language type of the input code snippet. This enables MetaTPTrans to not only extract language-agnostic information from multilingual source code data, but also capture the language-specific information, which is overlooked by standard multilingual training. Specifically, our approach consists of a language-aware Meta Learner and a feature extractor (Base Learner). The Meta Learner takes the programming language type (e.g., `Python`) as input and outputs a specific group of parameters for the feature extractor based on the associated language type. The language type is like an indicator to guide the Meta Learner to output specific parameters for a specific programming language. For the Base Learner, we apply TPTrans [108], a state-of-the-art code representation model, as the architecture of it. To the best of our knowledge, we are the first to leverage meta learning to learn to generate language-aware dynamic parameters for the feature extractor to learn both language-specific and language-agnostic information from multilingual source code datasets. We evaluate MetaTPTrans on two common software engineering tasks: code summarization and code completion. The experimental results show that our approach outperforms state-of-the-art methods significantly on both tasks.

In summary, this paper contributes the following: (1) MetaTPTrans, a meta learning based approach for multilingual code representation learning, which learns both language-agnostic and language-specific information from multilingual source code data, (2) Three different schemes for generating parameters for different kinds of weights of the Base Learner in MetaTPTrans, and (3) Experimental evaluation on two important and widely-used software engineering tasks: code summarization and code completion, and the results show that our approach outperforms state-of-the-art baselines significantly. Our code is available at: `https://github.com/weiguoPian/MetaTPTrans`.

3.2 Technical Preliminaries

3.2.1 Self-Attention with Absolute and Relative Position Embedding

Self-attention (SA) is the basic module in Transformers [145]. It maintains three projected matrices $\boldsymbol{Q} \in \mathbb{R}^{d_q \times d_q}$, $\boldsymbol{K} \in \mathbb{R}^{d_k \times d_k}$, and $\boldsymbol{V} \in \mathbb{R}^{d_v \times d_v}$ to compute an output that is the weighted sum of the input by attention score:

$$\begin{aligned} SA(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= softmax(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d}})\boldsymbol{V} \\ s.t. \quad \begin{bmatrix} \boldsymbol{Q} \\ \boldsymbol{K} \\ \boldsymbol{V} \end{bmatrix} &= \boldsymbol{X} \begin{bmatrix} \boldsymbol{W}^Q \\ \boldsymbol{W}^K \\ \boldsymbol{W}^V \end{bmatrix} \end{aligned} \tag{3.1}$$

where $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n)$ is the input sequence of the self-attention module, $\boldsymbol{x}_i \in \mathbb{R}^d$, d is the dimension of the hidden state, and $\boldsymbol{W}^Q \in \mathbb{R}^{d \times d_q}$, $\boldsymbol{W}^K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}^V \in \mathbb{R}^{d \times d_v}$ are the learnable parameters matrices of the self-attention component. Here, we follow the previous works [145, 178, 108] and set $d_q = d_k = d_v = d$. More specifically, the above equation can be reformulated as:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n}\exp(\alpha_{ik})}(\boldsymbol{x}_j \boldsymbol{W}^V) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K)^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.2}$$

where $\boldsymbol{z}_i$ is the output of $\boldsymbol{x}_i$ calculated by self-attention operation. In Vanilla Transformer, Vaswani *et al.* [145] used a non-parameteric absolute position encoding, which is added to the word vectors directly. Ke *et al.* [60] proposed a learnable projection for absolute position for computing the attention score among words:

$$\alpha_{ij} = \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K)^{\mathrm{T}}}{\sqrt{2d}} + \frac{(\boldsymbol{p}_i \boldsymbol{U}^Q)(\boldsymbol{p}_j \boldsymbol{U}^K)^{\mathrm{T}}}{\sqrt{2d}} \tag{3.3}$$

where $\boldsymbol{p}_i$ denotes the learnable real-valued vector of position i, and $\boldsymbol{U}^Q, \boldsymbol{U}^K \in \mathbb{R}^{d\times d}$ are the projection matrices of the position vectors $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ respectively. To capture the relative position relationship between words, Shaw *et al.* [123] proposed to use the relative position embedding between each two words:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n}\exp(\alpha_{ik})}(\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij}^V) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij}^K)^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.4}$$

where $\boldsymbol{r}_{ij}^K, \boldsymbol{r}_{ij}^V$ denote the learnable relative position embedding between positions i and j.

3.2.2 TPTrans

We briefly introduced how absolute and relative position embeddings are integrated into the self-attention module of Transformers. In this subsection, we describe the TPTrans [108] which is based on the aforementioned position embedding concepts.

TPTrans modifies the relative and absolute position embedding in self-attention with AST paths encodings so that AST paths could be integrated into Transformers. Specifically, they first encode the relative and absolute path via a bi-directional GRU [18]:

$$\begin{aligned} \boldsymbol{r}_{ij} &= GRU(\boldsymbol{Path}_{\boldsymbol{x}_i \to \boldsymbol{x}_j}) \\ \boldsymbol{a}_i &= GRU(\boldsymbol{Path}_{\boldsymbol{root} \to \boldsymbol{x}_i}) \end{aligned} \tag{3.5}$$

where $\boldsymbol{Path}_{\boldsymbol{x}_i \to \boldsymbol{x}_j}$ denotes the AST path from node $\boldsymbol{x}_i$ to node $\boldsymbol{x}_j$, $\boldsymbol{r}_{ij}$ is the relative path encoding between positions i and j, and $\boldsymbol{a}_i$ is the absolute path (from the $\boldsymbol{root}$ node to node $\boldsymbol{x}_i$) encoding of position i. Then, the two types of path encodings are integrated into Eq. (3.3) - (3.4) by replacing the absolute and relative position embeddings:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n}\exp(\alpha_{ik})}(\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij}\boldsymbol{W}^{r,V}) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij}\boldsymbol{W}^{r,K})^{\mathrm{T}}}{\sqrt{d}} + \frac{(\boldsymbol{a}_i \boldsymbol{W}^{a,Q})(\boldsymbol{a}_j \boldsymbol{W}^{a,K})^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.6}$$

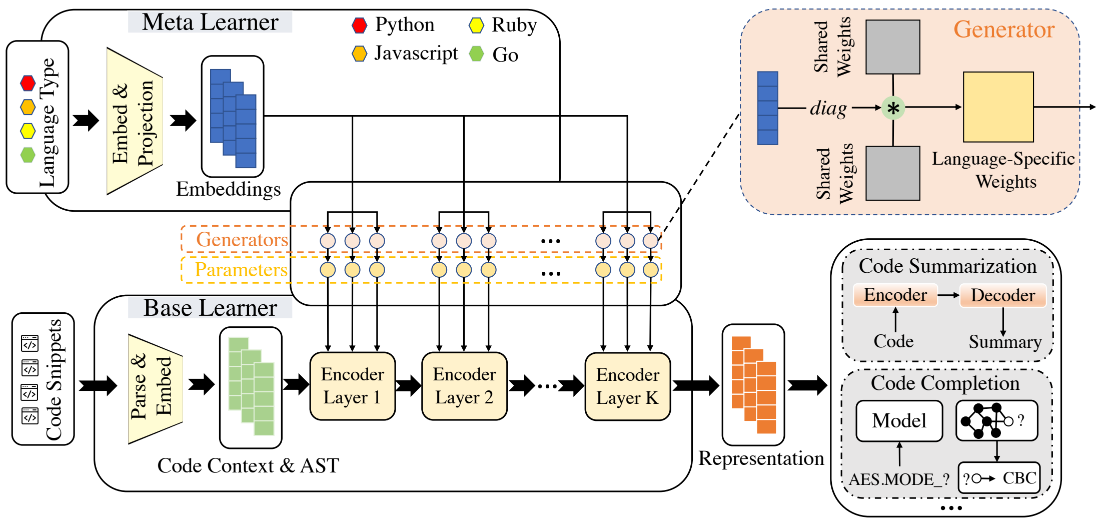


Figure 3.2: Architecture of MetaTPTrans.

where $\boldsymbol{W}^{r,K}, \boldsymbol{W}^{r,V} \in \mathbb{R}^{d\times d}$ are the key and value projection matrices of the relative path encoding and $\boldsymbol{W}^{a,Q}, \boldsymbol{W}^{a,K}$ denote the query and key projection matrices of the absolute path encoding.

3.3 Approach

We introduce MetaTPTrans which consists of two components: a Meta Learner and a Base Learner. Specifically, the Meta Learner takes the language type as input and generates language-specific parameters for the Base Learner. We apply the TPTrans as the architecture of the Base Learner, which takes the source code as input, and uses the language-specific parameters generated from the Meta Learner to calculate the representation of the input source code snippet. The overview of our approach is shown in Figure 3.2.

3.3.1 Meta Learner

The Meta Learner generates parameters for the Base Learner according to the language type. Specifically, for a given source code snippet $\boldsymbol{X}_i$ and its corresponding language type t_i, the *Language Embedding Layer* $\mathcal{T}$ embeds the language type t_i into an embedding $\boldsymbol{T}_i \in \mathbb{R}^{d_T}$. Then, a projection layer scales the dimension of $\boldsymbol{T}_i$. The overall process can be presented as:

$$\begin{aligned} \boldsymbol{T}_i &= \mathcal{T}(t_i) \\ \boldsymbol{P}_i &= Projection(\boldsymbol{T}_i) \end{aligned} \tag{3.7}$$

where $Projection(\cdot)$ and $\boldsymbol{P}_i \in \mathbb{R}^{d_P}$ denote the projection layer and the projected language type embedding, respectively. After producing the projected language type embedding $\boldsymbol{P}_i$, we now present the parameter generation scheme for the Base Learner. For a weight matrix $\boldsymbol{W}^{\lambda} \in \mathbb{R}^{d\times d}$ in the parameters of Base Learner, we conduct it via a generator $\mathcal{G}_{\lambda}$, which can be denoted as:

$$\boldsymbol{W}^{\lambda} = \mathcal{G}_{\lambda}(\boldsymbol{P}_i) \tag{3.8}$$

As noted in [11, 149], using a linear projection layer to scale a d_P-dimension vector to a matrix of dimension $d \times d$ exhibits a large memory footprint, especially that there are many such weight matrices in the parameters of the Base Learner. To reduce the computational and memory cost, we adopt the factorized scheme for the weight generation procedure by factorizing the representation of the weights, which is analogous to the *Singular Value Decomposition* [11]. In this way, the original vector can be projected into the target matrix dimension space with fewer parameters in the generator. More specifically, we first apply the diagonal operation to transform the vector $\boldsymbol{P}_i$ to a diagonal matrix, then two projection matrices $\boldsymbol{M}_\lambda \in \mathbb{R}^{d_P \times d}$ and $\boldsymbol{M'}_\lambda \in \mathbb{R}^{d \times d_P}$ are defined to project the diagonal projected language type embedding matrix into the space of the target weight matrix. This operation can be formulated as:

$$\boldsymbol{W}^\lambda = \mathcal{G}_\lambda(\boldsymbol{P}_i) = \boldsymbol{M'}_\lambda diag(\boldsymbol{P}_i)\boldsymbol{M}_\lambda \tag{3.9}$$

where $diag(\cdot)$ is the non-parametric diagonal operation, and $\boldsymbol{M}_\lambda$ and $\boldsymbol{M'}_\lambda$ are the learnable parameters in generator $\mathcal{G}_\lambda$.

3.3.2 Base Learner

The Base Learner is the module that actually learns the representation of code snippets, the parameters of which are generated from the Meta Learner. In our approach, we apply the TPTrans model as the architecture of the Base Learner. Recall from Eq. (3.6), the learnable parameters of TPTrans consist of the projection matrices of tokens ($\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$) and the projection matrices of path encodings ($\boldsymbol{W}^{r,K}, \boldsymbol{W}^{r,V}, \boldsymbol{W}^{a,Q}, \boldsymbol{W}^{a,K}$).

The Meta Learner generates the parameters of the Base Learner. First, we consider that the most obvious language-specific information is the language-specific underlying syntax rule in the context of the code snippet. Such contextual information is extracted from the input token sequences. Therefore, we first generate the projection matrices for token sequences. This procedure can be formulated as:

$$\begin{aligned} \boldsymbol{W}^\lambda_{t_i} &= \mathcal{G}_\lambda(\boldsymbol{P}_i) = \boldsymbol{M'}_\lambda diag(\boldsymbol{P}_i)\boldsymbol{M}_\lambda \\ & s.t. \;\; \lambda \in \{Q, K, V\} \end{aligned} \tag{3.10}$$

where t_i denotes the corresponding language type of code snippet $\boldsymbol{X}_i$, and $\boldsymbol{W}^Q_{t_i}, \boldsymbol{W}^K_{t_i}, \boldsymbol{W}^V_{t_i}$ are the learnable weights of $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$ associated with language type t_i, and $\boldsymbol{P}_i = Projection(\mathcal{T}(t_i))$ denotes the projected language type embedding of t_i. After that, the generated weights matrices are assigned to the related parameters by replacing the unified related weights matrices in Eq. (3.6):

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n}\exp(\alpha_{ik})}(\boldsymbol{x}_j\mathcal{G}_V(\boldsymbol{P}_i) + \boldsymbol{r}_{ij}\boldsymbol{W}^{r,V}) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i\mathcal{G}_Q(\boldsymbol{P}_i))(\boldsymbol{x}_j\mathcal{G}_K(\boldsymbol{P}_i) + \boldsymbol{r}_{ij}\boldsymbol{W}^{r,K})^{\mathrm{T}}}{\sqrt{d}} \\ &+ \frac{(\boldsymbol{a}_i\boldsymbol{W}^{a,Q})(\boldsymbol{a}_j\boldsymbol{W}^{a,K})^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.11}$$

Further, the weights matrices for path encoding projection can also be generated by the Meta Learner for different language types. This process aims to integrate the language-agnostic structural information (path encodings) into the contextual information dynamically according to the associated language type, which we believe is meaningful to investigate since the combination of these two kinds of information may also be influenced by the underlying language type. Similar to Eq. (3.10), this procedure can be expressed as:

$$\begin{aligned} \boldsymbol{W}_{t_i}^{\lambda} &= \mathcal{G}_{\lambda}(\boldsymbol{P}_i) = \boldsymbol{M'}_{\lambda} diag(\boldsymbol{P}_i) \boldsymbol{M}_{\lambda} \\ s.t.\ \ &\lambda \in \{\{r,K\},\{r,V\},\{a,Q\},\{a,K\}\} \end{aligned} \tag{3.12}$$

where $\boldsymbol{W}_{t_i}^{r,K}$, $\boldsymbol{W}_{t_i}^{r,V}$, $\boldsymbol{W}_{t_i}^{a,Q}$, $\boldsymbol{W}_{t_i}^{a,K}$ are the generated weights of $\boldsymbol{W}^{r,K}$, $\boldsymbol{W}^{r,V}$, $\boldsymbol{W}^{a,Q}$, $\boldsymbol{W}^{a,K}$ associated with language type t_i. After that, the generated weights matrices in Eq. (3.12) can be integrated into Eq. (3.6) by replacing the related weights matrices:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij} \mathcal{G}_{r,V}(\boldsymbol{P}_i)) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij} \mathcal{G}_{r,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \\ &+ \frac{(\boldsymbol{a}_i \mathcal{G}_{a,Q}(\boldsymbol{P}_i))(\boldsymbol{a}_j \mathcal{G}_{a,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.13}$$

Finally, we combine the above two kinds of weights generation schemes in which both context token projection and path encoding projection are generated by the Meta Learner:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \mathcal{G}_V(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \mathcal{G}_{r,V}(\boldsymbol{P}_i)) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \mathcal{G}_Q(\boldsymbol{P}_i))(\boldsymbol{x}_j \mathcal{G}_K(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \mathcal{G}_{r,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \\ &+ \frac{(\boldsymbol{a}_i \mathcal{G}_{a,Q}(\boldsymbol{P}_i))(\boldsymbol{a}_j \mathcal{G}_{a,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{3.14}$$

In the above, we generate weights matrices from three perspectives: (i) For context token projection (Eq. (3.11)), (ii) For path encoding projection (Eq. (3.13)), and (iii) For both context token projection and path encoding projection (Eq. (3.14)). We name those three schemes of weights matrices generation: *MetaTPTrans-*α, *MetaTPTrans-*β, and *MetaTPTrans-*γ, respectively.

3.4 Experimental Setup and Results

In this section, we present our experimental setup and results. We evaluate MetaTPTrans on two common and challenging tasks: code summarization and code completion. For the number of parameters and training time cost compared with the baseline (TPTrans), we present them in the Appendix.

3.4.0.0.1 Code Summarization Code summarization aims at describing the functionality of a piece of code in natural language and it demonstrates the capability of the models in capturing the semantics of source code [8, 178, 108]. Similar to previous work [178, 108], we consider a complete method body as the source code input and the method name as the target prediction (i.e. the NL summary) while predicting the method name as a sequence of subtokens. Following [178, 108], we evaluate the performance of our approach and baselines on the code summarization task using the metrics of precision, recall, and F1 scores over the target sequence.

3.4.0.0.2 Code Completion Code completion is another challenging downstream task in source code modeling. As the settings in [76, 78], the model aims to predict a missing token by taking the incomplete code snippet as input. To generate data for code completion task, we randomly replace a token with a special token `<MASK>` in a given code snippet. And then, the new code snippet with the special token `<MASK>` is used to predict the replaced missing token. We report Top-1 and Top-5 prediction accuracy as the evaluation metrics for the code completion task.

3.4.0.0.3 Dataset We conduct our experiments on the CodeSearchNet [50] dataset. Following [178, 108], we consider four programming languages in the dataset: Python, Ruby, JavaScript, and Go. Please see the Appendix for the details and the pre-processing of the dataset.

3.4.0.0.4 Baselines For code summarization, we compare MetaTPTrans against code2seq [7], GREAT [39], CodeTransformer [178] and TPTrans [108]. For code completion, we compare MetaTPTrans against Transformer [145], CodeTransformer [172] and TPTrans [108]. Please see the Appendix for more details.

Table 3.1: Precision, recall, and F1 for the code summarization task. The bold part denotes the overall best results, and the underlined part denotes the best results of baselines.

Model	Python			Ruby			JavaScript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
code2seq (Single-language)	35.79	24.85	29.34	23.23	10.31	14.28	30.18	19.88	23.97	52.30	43.43	47.45
GREAT (Single-language)	35.07	31.59	33.24	24.64	22.23	23.38	31.20	26.84	28.86	50.01	46.51	48.20
CodeTransformer (Single.)	36.40	33.66	34.97	31.42	24.46	27.50	35.06	29.61	32.11	55.10	48.05	51.34
TPTrans (Single-language)	38.39	34.70	36.45	33.07	28.34	30.52	33.68	28.95	31.14	55.67	51.31	53.39
code2seq (Multilingual)	34.49	25.49	29.32	23.97	17.06	19.93	31.62	22.16	26.06	52.70	44.36	48.17
GREAT (Multilingual)	36.75	31.54	33.94	30.05	24.33	26.89	33.58	27.78	30.41	52.65	48.30	50.38
CodeTransformer (Multi.)	38.89	33.82	36.18	33.93	28.94	31.24	36.95	29.98	33.10	56.00	50.44	53.07
TPTrans (Multilingual)	39.71	34.66	37.01	39.51	32.31	35.55	34.92	30.01	32.33	56.48	52.02	54.16
MetaTPTrans-α	40.22	**36.22**	**38.12**	**40.62**	**34.01**	**37.02**	37.87	31.92	34.64	58.12	53.82	55.89
MetaTPTrans-β	39.97	36.12	37.94	40.44	33.69	36.76	**38.87**	**32.66**	**35.50**	**58.86**	**54.24**	**56.45**
MetaTPTrans-γ	**40.47**	35.19	37.65	40.58	32.04	35.81	37.90	30.11	33.56	58.20	53.38	55.68

3.4.0.0.5 Implementation Details For both tasks, following [108], we set the embedding sizes of the word, path node, and hidden size of the Transformer to 512, 64, and 1024, respectively. A linear layer projects the word embedding into the size of the hidden layer of the Transformer. We use one bidirectional-GRU [18] layer of size 64 to encode the paths, and concatenate the final states of both directions as output. We use the Adam [64] optimizer with a learning rate of $1e^{-4}$. We train our models for 10 and 40 epochs for the code summarization and code completion tasks, respectively on 4 Tesla V100 GPUs with batch size of 128 and dropout of 0.2. For the Base Learner in the code summarization task, we use the same hyperparameters setting of TPTrans [108] for a fair comparison. Specifically, we set the number of encoder and decoder layers to 3 and 8, the number of attention heads to 3, and the dimension of the feed-forward layer to 4096. In the Meta Learner, the dimension of the language type embedding (d_T) and its projection (d_P) are set to 1024 and 2048, respectively. Following [178, 108], we add the pointer network [147] to the decoder. For the code completion task, we set the number of encoder layers, number of heads, and the dimension of feed-forward layers to 5, 8 and 2048 respectively for all the baselines and our approaches. As we mentioned above, we follow the code completion settings in [76, 78] that predict a missing token amid a incomplete code

Table 3.2: Top-1 and Top-5 accuracy for the code completion task. Underlined, bold values denote the best results in the baselines and the overall best results respectively.

Model	Python		Ruby		JavaScript		Go	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Transformer (Single-language)	47.57	69.86	44.39	62.24	37.57	53.15	40.21	59.65
CodeTransformer (Single.)	62.45	76.73	51.63	69.96	47.56	68.88	47.71	61.35
TPTrans (Single-language)	63.71	77.99	64.42	72.50	64.67	73.42	57.15	67.81
Transformer (Multilingual)	47.02	78.82	47.16	77.32	38.77	70.84	42.01	72.95
CodeTransformer (Multi.)	68.19	82.98	67.67	<u>83.47</u>	59.32	80.07	57.12	77.67
TPTrans (Multilingual)	<u>69.81</u>	<u>84.10</u>	<u>72.14</u>	82.27	<u>67.45</u>	<u>81.17</u>	<u>60.45</u>	<u>79.03</u>
MetaTPTrans-α	**77.13**	**94.28**	**78.05**	**95.42**	**73.52**	**92.88**	**67.47**	**91.15**
MetaTPTrans-β	71.75	86.26	73.82	86.85	72.71	86.90	66.74	85.21
MetaTPTrans-γ	67.12	90.72	71.89	93.61	69.55	90.97	61.60	88.99

snippet, which is not a sequence-to-sequence task. Thus, we apply a fully connected layer after the encoder rather than using a decoder to generate the predicted token. In the Meta Learner, we set both the dimension of the language type embedding (d_T) and its projection (d_P) to 512.

3.4.1 Code Summarization

Table 3.1 shows the results of the code summarization task. The top part of the table shows the results of the baselines trained on single-language datasets. The middle and bottom parts of the table show the results of the baselines trained on multilingual dataset and our MetaTPTrans respectively. MetaTPTrans-α (Eq. 3.11), MetaTPTrans-β (Eq. 3.13) and MetaTPTrans-γ (Eq. 3.14) outperform all the baseline methods significantly. Specifically, compared with the state-of-the-art results on the Python, Ruby, JavaScript, and Go datasets, our approach improves the F1 score by 1.11, 1.47, 2.40 and 2.29 respectively. Overall, MetaTPTrans improves precision by 0.76–2.38, recall by 1.52–2.65, and F1 by 1.11–2.40 across the four programming languages. For the experiment results without pointer networks, please see Appendix for details.

3.4.2 Code Completion

Table 3.2 shows the results for the code completion task where the top and middle parts denote the testing results of the baselines trained on single-language datasets and multilingual dataset respectively. And bottom part of the table demonstrates the results of our MetaTPTrans. MetaTPTrans-α improves the Top-1 (Top-5) prediction accuracy over the best baseline, by 7.32 (10.18), 5.91 (11.95), 6.07 (11.71) and 7.02 (12.12) for Python, Ruby, JavaScript, and Go, respectively. Among the three variants of MetaTPTrans, the MetaTPTrans-α consistently achieves the best results over the four programming languages. As described before, the language-specific weights matrices generated by the Meta Learner in MetaTPTrans-α are only assigned to the parameters of the code context token projection in the Base Learner (Eq. 3.11) while MetaTPTrans-β and MetaTPTrans-γ assign the generated weights to AST path encodings. This means that for code completion, compared with generating language-specific weights for the projection of structural information, generating weights for context token projection is more conducive for the extraction of language-specific information, which leads to a significant performance improvement.

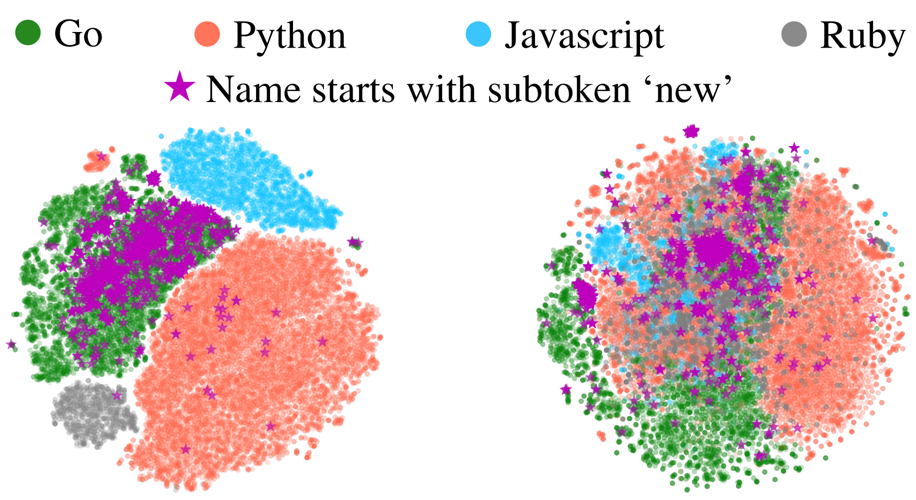


Figure 3.3: t-SNE visualization of the representation learned by MetaTPTrans-α (left) and TPTrans (right).

3.4.3 Visualization of Learned Representation

In figure 3.3, we show the t-SNE [144] visualization of the learned representations of all the code snippets from the validation set of the code summarization task. The left and right parts of Figure 3.3 show the code representation generated by the encoder of MetaTPTrans-α and the best baseline TPTrans respectively. We see that MetaTPTrans-α learns a distributed code representation that also respects the type of programming language of the code snippet as data points from the same language group together. This demonstrates that our model learns languages-specific features much better than the baseline model, where code snippets from the same language do not necessarily group together. Moreover, as an example, we mark all the code snippets whose names start with the subtoken `new` by the symbol ★. We see that MetaTPTrans-α achieves a much better grouping of those code snippets compared to the baseline model emphasizing our model ability to learn a better semantic representation of source code, while also respecting the language-specific features.

3.5 Conclusion

We propose MetaTPTrans, a meta learning approach for multilingual code representation learning. Instead of keeping the feature extractor with a fixed set of parameters across different languages, we adopt meta learning to generate different sets of parameters for the feature extractor according to the language type of the input code snippet. This enables MetaTPTrans to not only extract language-agnostic information, but to also capture language-specific features of source code. Experimental results show that our approach outperforms the state-of-the-art baselines significantly. Our work provides a novel direction for multilingual code representation learning.

4 You Don't Have to Say Where to Edit! jLED – Joint Learning to Localize and Edit Source Code

In this chapter, we present jLED, a joint learning approach for localizing and editing source code based on natural language descriptions. Without requiring precise location information, jLED jointly learns to localize edit regions and generate code edits, achieving improved performance on a benchmark of over 70k commits.

This chapter is based on the work published in the following research paper:

- Pian, W., Li, Y., Tian, H., Sun, T., Song, Y., Tang, X., Habib, A., Klein, J. and Bissyandé, T.F., 2025. You Don't Have to Say Where to Edit! jLED–Joint Learning to Localize and Edit Source Code. ACM Transactions on Software Engineering and Methodology.

Contents

4.1 Introduction

Code editing [97, 116] is a critical and continuous activity in the realm of software development. As software systems expand in size and complexity, developers undertake a multitude of edits to maintain and enhance their functionality. These edits may include bug fixes [161, 33, 81, 134], feature additions [98, 95, 96], or performance improvements [91]. Nevertheless, a significant portion of code editing across various projects is repetitive or similar in practice, which results in decreased efficiency in software development [117]. Therefore, researchers are motivated to devise automated approaches that can facilitate code editing by learning from historical examples [98, 97].

Approaches based on Neural Machine Translation (NMT) have demonstrated remarkable success in the realm of automated code editing. NMT is a type of machine learning approach that uses the sequence-to-sequence [129] architecture to predict the target sequence based on the source sequence of words or tokens, which is commonly used to translate sentences from one language to another, or to generate answers from questions. In the context of code editing, the process refers to the translation of source code to targeted code. Contrasting to traditional machine translation methods that translate words or phrases in isolation, NMT models the entire context of a sentence or even a paragraph to produce a nuanced translation [154]. Leveraging this capability, literatures [16, 163, 90] have successfully adapted NMT models to comprehend code semantics and generate more accurate and contextually relevant edits by leveraging multiple modalities of information relevant to code editing, such as the context, natural language guidance, test cases, etc.

Despite the achievements, NMT-based approaches face a substantial obstacle in real-world applications: Several studies [167, 73] show the capability to edit source code without explicitly specifying the edit location, they however tend to generate edits at locations with less accuracy at the line level. Consequently, NMT struggles to edit code effectively within a broad context without the knowledge of the exact edit location. Chakraborty*et al.* [16] investigated the contribution of different input modalities to the performance of their proposed NMT model. The findings indicate a considerable decline in performance when the edit location remains unknown to the NMT model. This however hinders the adoption of the NMT-based code edit approaches to practical scenarios.

Motivation example. To further illustrate our motivation, we present a motivation example in Figure 4.1 generated by fine-tuned CodeReviewer models [73]. In this example, we show that when the model is trained and inferred without line-level location information, it generates the edit that corresponds to the inaccurate location. Conversely, when the location information is explicitly provided, the model is able to generate a more accurate edit corresponding to the precise line location.

This example highlights the importance of explicitly providing precise location information to the model during training and inference. However, in real-world code editing scenarios, the exact edit location is often unavailable, making it impractical to rely on explicitly provided edit locations. To address this challenge, leveraging a localization model to predict the edit location could improve the editing accuracy of the code edit model. Nevertheless, imperfect location predictions can harm the edit generation process—potentially leading the model to edit irrelevant lines, for which we will discuss in Section 4.5.5 and 4.5.6, where we show that using the

```
# Don't open code task_pid in kvm_vcpu_ioctl.

struct pid *oldpid;
r = -EINVAL;
if (arg) goto out;
oldpid = rcu_access_pointer(vcpu->pid);
# Line to edited
- if (unlikely(oldpid != current->pids[PIDTYPE_PID].pid)) {
# Ground truth
+ if (unlikely(oldpid != task_pid(current))) {
# Edit result generated by the model trained and inferred with location information.
+ if (unlikely(oldpid != get_task_pid(current, PIDTYPE_PID))) {
# Edit result generated by the model trained and inferred without location information.
+ newpid = get_task_pid_noauto(current, PIDTYPE_PID);
    struct pid *newpid;
    r = kvm_arch_vcpu_run_pid_change(vcpu);
    if (r) break;
# Inaccurate line location
    newpid = get_task_pid(current, PIDTYPE_PID);
    rcu_assign_pointer(vcpu->pid, newpid);
    if (oldpid) synchronize_rcu();
    put_pid(oldpid);
}
r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
```

Figure 4.1: Motivation example.

localization model's predicted results as direct input to the editing model does not yield satisfactory performance. A potential approach to tackle this problem is the recent multi-task frameworks, which have been proven effective in software engineering tasks by using one task as a "soft constraint" for another [77, 78, 76]. By leveraging the parameter-sharing mechanism of multi-task learning, these approaches facilitate knowledge transfer between tasks [121, 13] through learning more generalizable features that are relevant to all tasks. Additionally, different optimization targets can perform as regularization terms for one another [82], further improving the generalization of the model on each task. Moreover, a recent work [85] also shows the insight of leveraging the program repair/refinement task to improve a model's fault localization capabilities, further motivating us to explore the latent semantic relationship between code editing and localization tasks, enabling the model to generate more accurate code edits without explicitly relying on precise location information. More specifically, in our code editing scenario, the training process of the edit target and the localization target are *related but not strictly dependent*, creating an opportunity to use one as a guiding signal to enhance the performance of the other. This approach avoids imposing strict dependencies while promoting mutual benefits in training both the editing and localization tasks.

This paper. We propose to jointly optimize the two loss functions of edit location and code edition in NMT models, towards producing an integrated approach to enable precise localization and edition of source code without the knowledge of exact edit locations. The main contributions are as follows:

❶ Our paper introduces jLED (**j**ointly **L**ocalize and **ED**it), a novel supervised learning approach designed to enable the practical application of code editing without the need for edit location. jLED leverages large-scale language models to uniformly localize and edit source code.

❷ We conduct comprehensive experiments to evaluate the performance changes by employing different modalities for sequence-to-sequence editing baselines.

❸ After collecting a large dataset of 77,044 edited code samples from two famous

GitHub projects – Linux and Wireshark, we extensively evaluate the effectiveness of jLED to localize and edit source code. The results demonstrate our tool jLED outperforms or achieves competitive performance when compared against the localizing and editing baselines, using five different large pre-trained code models trained with our pipeline.

❹ To further evaluate the effectiveness of our proposed joint learning pipeline, we construct a two-stage localization-editing pipeline, in which a localization model and an editing model are trained separately. We conduct experiments for ablation study with a two-stage pipeline, and experimental results further demonstrate the superiority of our proposed joint learning pipeline.

Availability. Our artifact, code, and dataset are publicly available at: `https://github.com/weiguoPian/Code_Edit_Joint_Learning`.

The remainder of this paper is presented as follows. Section 4.2 introduces the background of this work. In Section 4.3, we present our methodology with detailed explanations. Section 4.4 and 4.5 cover the experimental design and results. We provide discussions in Section 4.6. Section 4.7 concludes this work.

4.2 BACKGROUND

4.2.1 Neural Machine Translation

Neural Machine Translation (NMT) has emerged as a promising approach in the field of machine translation, exhibiting well performance in automating language translation tasks [16]. By leveraging deep neural networks, NMT models are capable of learning and generating translations in an end-to-end manner, thereby overcoming the limitations of traditional statistical machine translation methods. At its core, NMT comprises two fundamental components: the encoder and the decoder, which work synergistically to facilitate the translation process. The encoder component plays a crucial role in comprehending and processing the input sentence, utilizing sophisticated neural architectures to generate a vector representation that encapsulates the underlying semantic meaning of the source text. This vector representation serves as a rich and comprehensive representation of the source sentence, enabling the subsequent translation process to leverage the encoded information effectively. The decoder component, on the other hand, capitalizes on the encoded input representation to sequentially generate the target sentence through a process of logical reasoning.

In recent years, the field of Software Engineering (SE) has witnessed a broad range of applications for NMT. Notably, NMT has found utility in areas such as Automatic Program Repair [56, 90, 159], Program Synthesis [164], and Code Edit Generation [142, 141]. These applications capitalize on NMT's ability to comprehend and generate intricate patterns, making it a valuable tool for SE-related tasks. The integration of NMT in software engineering research highlights its potential to enhance various aspects of software development, offering new avenues for improving program understanding, code generation, and automated repair.

4.2.2 Transformer Model for Sequence Processing

Transformer model [145] has emerged as a highly influential and prominent model for sequence processing tasks within the field of natural language processing (NLP), leading to numerous state-of-the-art achievements [152, 10]. Prior to the advent of the

Transformer, recurrent neural networks (RNNs) and their variants, such as long short-term memory (LSTM) and gated recurrent units (GRUs), were conventionally utilized for sequence processing tasks. While RNNs showcased commendable performance, they encountered challenges in parallelization and capturing long-range dependencies adequately. To overcome these limitations, the Transformer model introduced a novel self-attention mechanism, enabling selective attention to different parts of the input sequence through adaptive weighting. This mechanism played a pivotal role in capturing dependencies between distinct positions within the sequence, facilitating parallel processing of the input. During the token representation learning phase, the Transformer model learned to attend to all input tokens, transforming the sequence into a comprehensive graph where each token represented a node. The edge weights in this graph denoted the attention weights between tokens, which were learned based on the specific task at hand. Additionally, positional encoding was incorporated into the Transformer model to encode the position of each token in the sequence, facilitating the learning of long-range dependencies. The Transformer's ability to reason about long-range dependencies has proven to be highly advantageous for various source code processing tasks, including code generation [130] and code summarization [3].

4.2.3 Transfer Learning for Source Code

Transfer learning [151] has emerged as a prominent research direction in the domain of software engineering (SE) due to its potential to address various SE tasks effectively. In SE, transfer learning involves the creation of task-agnostic representations of source code, which can be leveraged and repurposed across different tasks. One prevalent approach to obtain such task-agnostic representations entails pre-training models using a large corpus of source code. During the pre-training phase, the primary objective is to enhance the model's understanding of code or its ability to generate accurate code. By leveraging a substantial collection of source code, a pre-trained model is expected to encapsulate valuable code-related knowledge within its learnable parameters. Subsequently, these pre-trained models are fine-tuned to adapt to specific task objectives.

Several transformer-based encoder models have been developed to facilitate pre-training for comprehending source code. Notable examples include CodeBERT [27] and GraphCodeBERT [34]. CodeBERT [27] focuses on learning continuous representations of code snippets, enabling a deeper understanding of their structural and contextual aspects. GraphCodeBERT [34] incorporates the data-flow graph modeling into the masked token pre-training process of the BERT model to capture code dependencies and interactions, facilitating higher-level reasoning and tasks such as code completion and refactoring. In the context of code generation, two prominent models are CodeGPT [87] and PLBART [2]. CodeGPT pre-trains a transformer-based model specifically designed for sequentially generating general-purpose code. More recently, PLBART introduced a joint pre-training approach that encompasses both code understanding and code generation, utilizing denoising auto-encoding. PLBART consists of an encoder and a decoder, where the encoder is exposed to slightly perturbed code, while the decoder is responsible for producing code without such perturbations.

```
# Don't open code task_pid in kvm_vcpu_ioctl.

struct pid *oldpid;
r = -EINVAL;
if (arg) goto out;
oldpid = rcu_access_pointer(vcpu->pid);
- if (unlikely(oldpid != current->pids[PIDTYPE_PID].pid)) {
+ if (unlikely(oldpid != task_pid(current))) {
    struct pid *newpid;
    r = kvm_arch_vcpu_run_pid_change(vcpu);
    if (r) break;
    newpid = get_task_pid(current, PIDTYPE_PID);
    rcu_assign_pointer(vcpu->pid, newpid);
    if (oldpid) synchronize_rcu();
    put_pid(oldpid);
}
r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
```

```
# make buddy table static. Idea is to reduce false cacheline sharing and stuff.

static int cntlz(u32 value);
static int cnttz(u32 word);
static int dbAllocDmapBU(struct bmap * bmp, struct dmap * dp, s64 blkno, int nblocks);
static int dbInitDmap(struct dmap * dp, s64 blkno, int nblocks);
static int dbInitDmapTree(struct dmap * dp);
static int dbInitTree(struct dmaptree * dtp);
static int dbInitDmapCtl(struct dmapctl * dcp, int level, int i);
static int dbGetL2AGSize(s64 nblocks);
- static s8 budtab[256] = {
+ static const s8 budtab[256] = {
    3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
}
```

Figure 4.2: Two illustrative examples of our collected dataset.

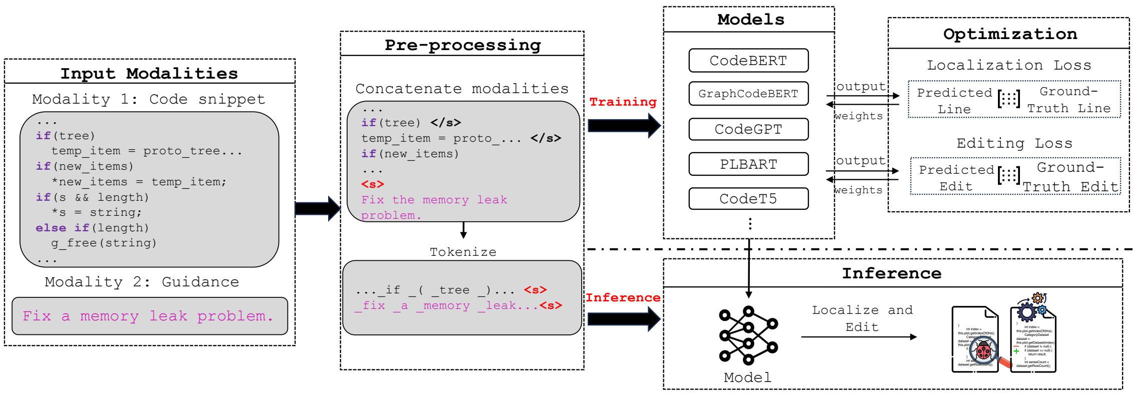
4.3 Approach

In this section, we delve into a detailed presentation of our proposed approach. Figure 4.3 illustrates the pipeline of our jLED. The overall pipeline is composed of three primary components: input pre-processing, model architecture, and the optimization objective. Note that, during the inference phase, the optimization objective is replaced by an output generation module. In the pre-processing stage, we concatenate different parts/modalities, converting them into a token sequence for input.

4.3.1 Input Modalities, Data Collection and Pre-processing

As we mentioned before, since we may not know the exact line-level location in which the line content should be edited, it is not in practice to provide the exact line-level location information as a part of the input of the model. Therefore, in our setting, the model can only take natural language guidance and code context (a multiple lines code fragment) as input. We use $\mathcal{G}$ and $\mathcal{C}$ to denote the natural language guidance and the code context respectively, and we follow the setting in [16] to use a special token `<s>` to split the two modalities $\mathcal{G}$ and $\mathcal{C}$.

For the data collection process, we construct the dataset using a series of git commands: git show, git diff, and git checkout, targeting the retrieval of code changes across different versions. Specifically, we focus on extracting the differences that



```
...
if(tree)
  temp_item = proto_tree...
if(new_items)
  *new_items = temp_item;
if(s && length)
  *s = string;
else if(length)
  g_free(string)
...
```

```
Fix a memory leak problem.
```

```
...
if(tree) </s>
temp_item = proto_... </s>
if(new_items)
...
<s>
Fix the memory leak
problem.
```

```
..._if _( _tree _)... <s>
_fix _a _memory _leak...<s>
```

Figure 4.3: Overview of our jLED pipeline.

represent single-line edits and their associated commit messages, constituting a single dataset sample. This approach ensures that each sample captures a specific code modification scenario, accompanied by the developer's intent as expressed in the commit message. Afterward, to maintain the quality and relevance of our dataset, we apply stringent filtering criteria, in which samples are excluded if the single-line change pertains solely to comments rather than code content. This decision is based on our goal which focuses on code editing that potentially affects software behavior and performance. By filtering out comment-only changes, we aim to enhance the model's learning of code syntax and semantics. For each sample, it includes not only the line to be edited but also contextual code lines surrounding the edit. This context, encompassing lines before and after the target line, is crucial to create a scenario for localizing the code line to be edited before generating edits. Finally, we remove the blank lines in each sample to get the clean ones, and get the final location line number to be edited.

In the Pre-processing step, we then apply the tokenizer to tokenize the input into sequences of tokens. For the tokenizer, we follow Chakraborty et al. [16] and use the sentence-piece tokenizer [66] in all our experiments, which divides each token into sequences of subtokens. In our implementation, we apply their original pre-trained sentence-piece tokenizer to different pre-trained models used in our training pipeline. Finally, the tokenized sequence is generated after the data pre-processing as the input of the model. Figure 4.2 presents two illustrative examples of the collected dataset.

4.3.2 Models

Recently, there has been a significant surge in interest of Transformer-based [145] code models [178, 112, 108] in fields of code representation learning and software engineering. These models, especially pre-trained large code models [27, 34, 87, 2, 150], attract lots of attention due to their superior performance and generalizability, proving to be highly advantageous for various research topics in software engineering [138, 133, 45, 14, 93, 52, 136, 55]. Inspired by this, we also apply the pre-trained large code models in our approach, which can be divided into three categories: encoder pre-trained code models, decoder pre-trained code models, and encoder-decoder pre-trained code models. Specifically, the representative encoder pre-trained code models include CodeBERT [27] and GraphCodeBERT [34], while one of the most representative decoder pre-trained models is CodeGPT [87]. Recently, researchers also explored the ability of pre-trained encoder-decoder models in the field of code representation learning and proposed the pre-trained encoder-decoder code models PLBART [2] and CodeT5 [150]. In the rest of this subsection, we introduce the details of these pre-trained models' architecture.

The basic component of the large code models is the encoder-decoder architecture, a powerful sequence-to-sequence deep learning architecture, which has been widely used in text-to-text task, text-to-code task, code-to-text, and code-to-code (our task) tasks. Note that, in our experiments, all the models used in our proposed jLED are based on the encoder-decoder architecture, except CodeGPT which is a decoder-only model. We will discuss it later.

4.3.2.0.1 Encoder. Encoder is the first part of the encoder-decoder architecture, which is used for encoding the pre-processed input subtokens sequence (see last subsection for details) into the semantic feature space to obtain the representation of the input sequence. For an L^e-layers encoder model, the *l-th* layer's output feature can be denoted as:

$$\begin{aligned} \boldsymbol{X}_e^l &= \mathcal{F}_e^l(\boldsymbol{X}_e^{l-1}; \boldsymbol{W}_e^l), \\ s.t. \;\; \boldsymbol{X}_e^{l-1} &= \{\boldsymbol{x}_1^{l-1}, \boldsymbol{x}_2^{l-1}, ..., \boldsymbol{x}_n^{l-1}\} \end{aligned} \tag{4.1}$$

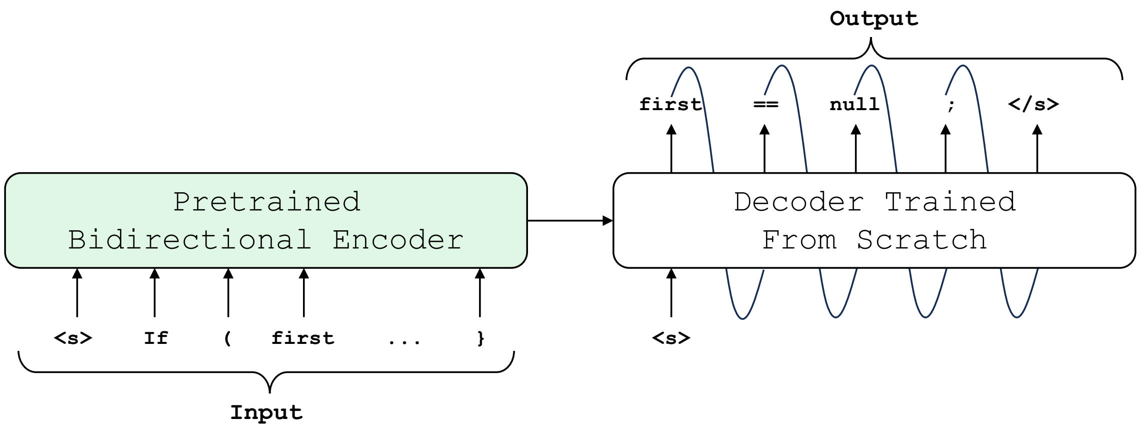
where $\boldsymbol{X}_e^{l-1} = \{\boldsymbol{x}_1^{l-1}, \boldsymbol{x}_2^{l-1}, ..., \boldsymbol{x}_n^{l-1}\}$ denotes the intermediate feature of the input subtokens sequence, and $\boldsymbol{x}_i^{l-1} \in \mathbb{R}^d$ and d are the intermediate representation of the *i-th* subtoken generated by the *l-th* encoder layer and the length of the intermediate representation of each subtoken, respectively. Note that, we use $\boldsymbol{X}_e^0 = \{\boldsymbol{x}_1^0, \boldsymbol{x}_2^0, ..., \boldsymbol{x}_n^0\}$ to represent the original input subtokens sequence. $\mathcal{F}_e^l$, parameterized by the learnable weights $\boldsymbol{W}_e^l$, is the *l-th* layer of the encoder model. For an L-layers encoder model, the final output of the encoder model can be denoted as $\boldsymbol{X}_e^L \in \mathbb{R}^{n \times d}$, which will be used as the input of the decoder model to generate the prediction (target sequence, code edits in our task), as well as the input of the localization branch of our proposed pipeline. In the rest of this paper, we simply use $\boldsymbol{Z}$ and $\boldsymbol{X}$ to denote $\boldsymbol{X}_e^L$ and $\boldsymbol{X}_e^0$ respectively, and use $\mathcal{F}_e$ and $\boldsymbol{W}_e$ to denote the entire encoder model and its trainable parameters.

4.3.2.0.2 Decoder. Decoder is another part of the encoder-decoder architecture, which aims to decode the representation generated by the encoder, to the target output subtokens sequence. In sequence-to-sequence tasks, the decoder generates subtokens sequentially using the encoder generated global representation and previous decoder generated subtokens. For the n-th subtoken's generation, the decoder takes previously generated subtokens and the encoder generated representation/hidden states as the input, to predict the next subtoken. This process can be denoted as:

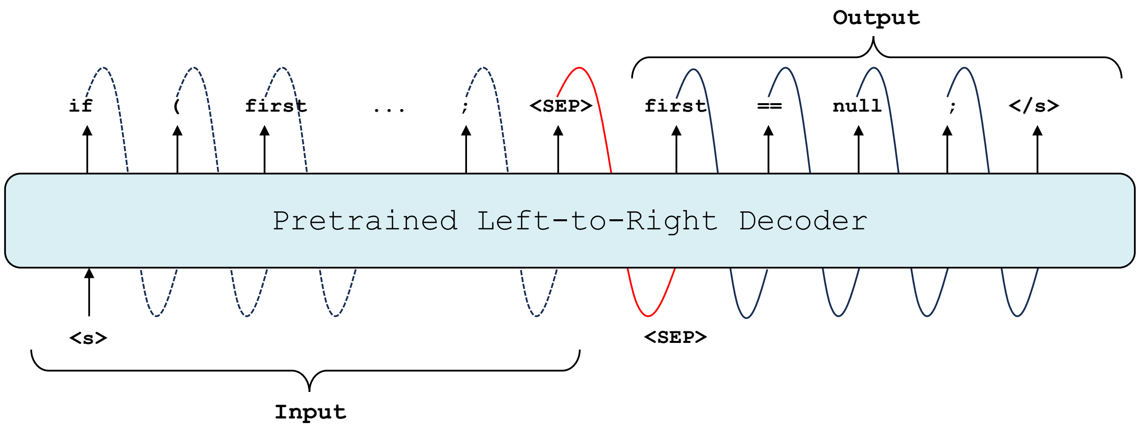
$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}, \boldsymbol{Z}; \boldsymbol{W}_d), \\ s.t. \;\; \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{4.2}$$

where $\mathcal{F}_d$ denotes the decoder model with learnable parameters $\boldsymbol{W}_d$. $\boldsymbol{u}_i$ denotes the i-th output subtoken. Please note that, $\boldsymbol{u}_0$ is a special token that indicates the start of output generation.

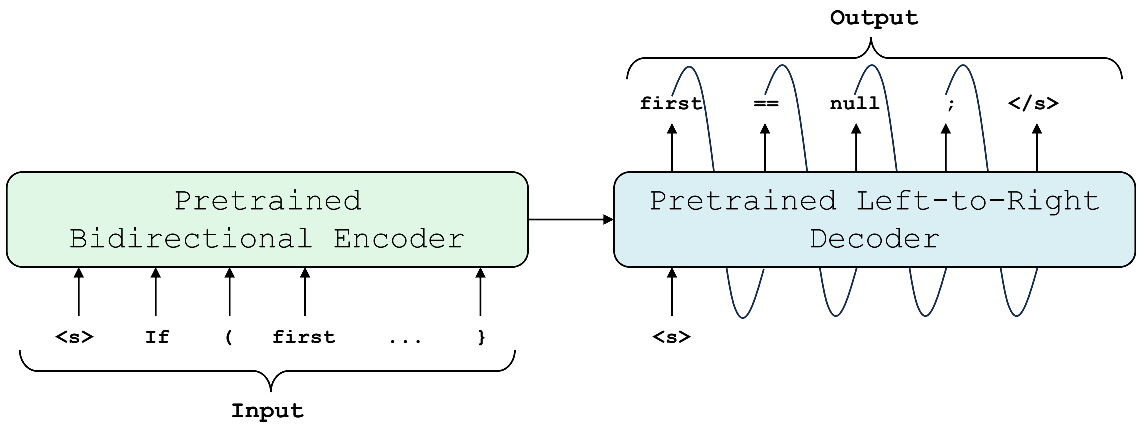
Note that, for decoder-only models, *e.g.* CodeGPT [87], due to the lacking of encoder part, when generate the n-th subtoken $\boldsymbol{u}_n$, the model only takes $\boldsymbol{U}_{n-1}$ as input without the intermediate representation $\boldsymbol{Z}$ in the modeling process, which can


Output
first
==
null
;
</s>
Pretrained Bidirectional Encoder
Decoder Trained From Scratch
<s>
If
(
first
...
}
<s>
Input

(a) Encoder pre-trained encoder-decoder models' architecture — Consists of bidirectional pre-trained encoder and a decoder trained from scratch.



(b) Decoder-only pre-trained models' architecture — One pre-trained single decoder processes the input and output sequentially from left to right.


Output
first
==
null
;
</s>
Pretrained Bidirectional Encoder
Pretrained Left-to-Right Decoder
<s>
If
(
first
...
}
<s>
Input

(c) Joint encoder-decoder pre-trained models' architecture — Consists of pre-trained bidirectional encoder and pre-trained left to right decoder.

Figure 4.4: Schematic diagram [16] of the three types of pre-trained models: (a) Encoder pre-trained encoder-decoder models, (b) Decoder-only pre-trained models, and (c) Joint encoder-decoder pre-trained models.

be represented as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{4.3}$$

Based on above encoding and decoding processes, models inference with encoder-decoder architecture (encoder pre-trained models and encoder-decoder pre-trained models), *e.g.* CodeBERT [27], GraphCodeBERT [34], PLBART [2], and CodeT5 [150], in code editing task can be denoted as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}, \boldsymbol{Z}; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\},\ \ \boldsymbol{Z} = \mathcal{F}_e(\boldsymbol{X}; \boldsymbol{W}_e), \end{aligned} \tag{4.4}$$

Similarly, the code editing task in decoder-only pre-trained model, *e.g.* CodeGPT [87], can be expressed as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d([\boldsymbol{X}, \boldsymbol{U}_{n-1}]; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{4.5}$$

We use the pre-processed subtokens sequence (see last subsection for details) as the input of the model. In this subsection, we introduce the details of the models that can be used in our approach.

In summary, Equation 4.4 and 4.5 present the edits generation process in the code editing task. However, as we mentioned before, in real-world code editing, given a code chunk/snippet, we usually don't know the exact line location where we should edit. That is, the modality of *the content of the code line that to be edited*, is unknown. In this situation, since the model does not learn *where* to be edited in the given code chunk/snippet, as well as the content to be edited, existing standard end-to-end sequence-to-sequence training/fine-tuning strategy is inadequate for enabling the model to generate precise code edits. To tackle this problem, we propose jLED, a joint training strategy to allow the model to learn to localize and edit simultaneously, in which the localization task can be used to facilitate the editing generation process for unknown edit locations. In the following subsection, we introduce the details of our proposed jLED.

4.3.3 Optimization

In this subsection, we introduce the details of our proposed joint training loss function for our jLED.

4.3.3.1 Editing Loss

The first component of our final loss function is the editing loss, serving as the optimization target to minimize the distance (loss score) between the generated edits (sequence of output subtokens) and the ground truth edits. The editing loss can be denoted as:

$$\mathcal{L}_{edit} = \mathbb{E}_{\boldsymbol{X} \sim \mathcal{D}} \Big[\sum_{i=1}^{L} \mathcal{L}_{CE}(\boldsymbol{u}_i, \boldsymbol{y}_i) \Big], \tag{4.6}$$

where $\mathcal{D}$ denotes the training set, $\boldsymbol{y}_i$ is the i-th subtoken in the ground-truth subtoken sequence (ground truth edits) $\mathcal{Y}$, L is the maximum length of the output sequence, and $\mathcal{L}_{CE}$ is the cross-entropy loss function.

4.3.3.2 Localization Loss

Localization loss aims to guide the model to learn the exact line-level location that to be edited given a natural language description and associated code context. To predict the location line number, a predictor $\mathcal{P}(\cdot; \boldsymbol{\theta}_p)$ with learnable parameters $\boldsymbol{\theta}_p$ is added on the top layer of the model. The predictor takes the hidden state generated by the model from the input sequence as its input, to output the predicted location line number.

Specifically, in the encoder-decoder models, *e.g.* CodeBERT [27], GraphCodeBERT [34], PLBART [2], and CodeT5 [150], the predictor takes the encoder generated representation $\boldsymbol{Z}$ as the input and output the predicted location line number, which is used to calculate the localization loss with the ground truth location line number. This process can be denoted as:

$$\mathcal{L}_{loc.} = \mathbb{E}_{\boldsymbol{X} \sim \mathcal{D}} \Big[\mathcal{L}_{CE}(\mathcal{P}(\boldsymbol{Z}; \boldsymbol{\theta}_p), \boldsymbol{l}) \Big], \tag{4.7}$$

where $\boldsymbol{l}$ denotes the ground location line number where to be edited.

Similarly, for decoder-only models, *e.g.* CodeGPT [87], the predictor takes the decoder generated hidden states of the input sequence as its input to generate the line number prediction, which can be presented as:

$$\mathcal{L}_{loc.} = \mathbb{E}_{\boldsymbol{X} \sim \mathcal{D}} \Big[\mathcal{L}_{CE}(\mathcal{P}(\mathcal{F}_d(\boldsymbol{X}; \boldsymbol{W}_d); \boldsymbol{\theta}_p), \boldsymbol{l}) \Big], \tag{4.8}$$

4.3.3.3 Joint Loss Function

Finally, the editing loss and the localization loss are combined as the joint loss function, which is denoted as:

$$\mathcal{L}_{joint} = \mathcal{L}_{edit} + \lambda \mathcal{L}_{loc.}, \tag{4.9}$$

where λ is the hyperparameter to balance the values of the editing loss and localization loss. Incorporating a joint loss function, the model benefits from two optimization targets: localization and editing. The localization target enables the model to pinpoint the specific line number requiring edits, thus refining its focus during the edit generation process. Simultaneously, the editing training target further guides the model to accurately identify these line numbers. Owing to this synergistic training framework, the model is capable of generating highly accurate edits even when the exact line locations within the source code are unknown.

4.3.4 Inference

After the training process described in the previous sections, the trained model will be utilized to predict edits based on the pre-processed and tokenized subtoken sequence. Following a similar approach to the training process mentioned earlier, the model will generate another sequence of subtokens representing the predicted edits. These predicted edits will be used to calculate evaluation metrics and generate the post-editing code context. This process can also be expressed by Equations 4.4 and 4.5.

4.4 Experimental Design

4.4.1 Dataset

Considering that existing fine-tuning datasets [143, 73] for code editing contain a significant proportion of input data samples that are small (no more than five lines), a model fine-tuned on such datasets may not be able to effectively cope with the challenge of editing source code with multiple lines input. Therefore, we construct a new large-scale dataset, in which each sample contains a number of lines from 10 to 20. Our collected datasets contains (1) source code context before edited, (2) natural language description, (3) line-level edits location (line number), and (4) the ground-truth edits. Our dataset is collected from two famous GitHub projects – Linux and Wireshark. There are a total of 77,044 samples in the dataset. More specifically, 66,044, 5,500, and 5,500 samples for training, evaluation, and testing, respectively. For each sample in the dataset, it contains (1) a source code snippet with several lines as the code context, (2) a natural language description of the editing purpose as the natural language guidance, (3) a line number as the editing location, and (4) the ground-truth edits. Each of the sample is generated by using the `git` command to extract the code change, commit message, and the original code file before editing. Table 4.1 presents the statistics of the dataset.

Table 4.1: Statistic of the dataset

Split	Training	Validation	Testing
Sample #	66,044	5,500	5,500
Avg. code snippet tokens #	201.68	201.89	203.49
Avg. edit tokens #	17.86	17.95	18.26

4.4.2 Data Preparation

For our collected dataset described in Section 4.4.1, we follow the pre-processing method described in Section 4.3.1 to pre-process each sample in the dataset by concatenating the source code context $\mathcal{C}$ and the associated natural language guidance $\mathcal{G}$ as an input data sample [$\mathcal{C}$ `<s>` $\mathcal{G}$] where `<s>` denotes the special token for splitting different modalities, and apply the tokenizer to tokenize each input data sample to generate the input subtoken sequence $\boldsymbol{X}$. In the modality of source code context $\mathcal{C}$, we add another special token `</s>` to the end of each code line for splitting different lines of the source code context. In this way, the model is able to know how many lines in the source code context $\mathcal{C}$ as well as the exact start and end position of each line. Then, we extract each sample's ground-truth edit as the editing label $\mathcal{Y}$ and the location line to be edited as the localization label $\boldsymbol{l}$.

4.4.3 Training

After the preparation of each data point in the dataset, *i.e.* concatenating each sample's source code context $\mathcal{C}$ and the associated natural language guidance $\mathcal{G}$ into [$\mathcal{C}$ `<s>` $\mathcal{G}$] as the input, and extracting each sample's ground-truth edit and the location line to be edited as the editing label $\mathcal{Y}$ and the localization label $\boldsymbol{l}$. We apply the processed dataset to train and evaluate the models. To evaluate the performance of our proposed joint training under the scenario of editing source code using natural language guidance but without knowing the exact line location to be edited, we conduct experiments with various well-known and most used pre-trained models fine-tuned on our dataset, including encoder pre-trained encoder-decoder models, such as CodeBERT [27] and GraphCodeBERT [34], joint encoder-decoder pre-trained models, such as PLBART [2] and CodeT5 [150], and pre-trained decoder-only models, such as CodeGPT [87]. We train the model with our joint training pipeline, using the Adam [64] optimizer with the learning rate of 5e-5. As described before, both editing loss and localization loss are implemented by the cross-entropy loss function. We train each model to convergence, and use the beam search to generate output edits during inference (validation and testing). For all models, we set the balance hyperparameter λ to 0.1 during training. We implement the training and inference pipeline with Pytorch [107], and use the pre-trained parameters from Hugging Face.

4.4.4 Evaluation Metric

We use the BLEU score and Top-1 accuracy as the evaluation metric to evaluate the performance of the editing results. For Top-1 accuracy of the editing results, we follow the settings in the previous work [16], in which the beam size is set to 5, and then, *only the generated edits that perfectly match the ground-truth edits are correct.* This setting allows the most stringent metric for evaluation [16]. For the evaluation of the localization results, we also apply the Top-1 accuracy and the Top-5 accuracy as the evaluation metric. For the Top-1 accuracy in localization, the line number with the highest probability score in the line-aware probability distribution is considered as the predicted localization result to match the ground-truth localization label $\boldsymbol{l}$. For the Top-5 accuracy in localization, the prediction is considered as correct if any of our model's Top-5 highest probability prediction matches with the ground-truth localization label $\boldsymbol{l}$.

4.4.5 Research Questions

Different from the previous code editing approach, *i.e.*, MODIT [16], we consider a more practical code editing scenario, in which the exact location to be edited is unknown (the first modality in the input of MODIT [16]). To evaluate the performance of the existing standard sequence-to-sequence training with multi-modalities, *i.e.* training pipeline in the paper of MODIT [16], we firstly conduct experiments by training different large pre-trained code models on this state-of-the-art multi-modalities training pipeline [16] under our considered more practical setting, *i.e.*, only with the modalities of the source code context ($\mathcal{C}$) and the natural language guidance ($\mathcal{G}$). Then, we conduct experiments by training different large pre-trained code models with our proposed joint training pipeline. Our research questions are as follows:

RQ-1. How do the models perform when trained on a standard sequence-to-sequence multi-modalities training pipeline with only modalities of the source code context and the natural language guidance?

RQ-2. How does our joint training pipeline (jLED) perform compared to the baseline?

RQ-3. How does our joint training pipeline perform compared to the two-stage localization-editing pipeline?

4.5 Experimental Results

✍ **RQ-1** ► *How do the models perform when trained on standard sequence-to-sequence multi-modalities training pipeline with only modalities of the source code context and the natural language guidance?* ◄

4.5.1 Experimental Setup for RQ-1

In our first research question, we aim to evaluate the performance of the standard sequence-to-sequence training (*i.e.*, only using the editing loss in Equation 4.6 as the training loss function) with multi-modalities (source code context $\mathcal{S}$ and natural language guidance $\mathcal{G}$). We name these trained models as baselines. To evaluate the performance of the baselines, we carefully choose the most representative and common pre-trained code models, including encoder pre-trained encoder-decoder models, decoder-only pre-trained models, and joint encoder-decoder pre-trained models. For the encoder pre-trained encoder-decoder models, we select CodeBERT [27] and GraphCodeBERT [34] in our pipeline. For the decoder-only pre-trained models, we train a CodeGPT [87] model. And for the joint encoder-decoder pre-trained models, we apply the most used PLBART [2] and CodeT5 [150] , and recent proposed pre-trained code editing/reviewing models CoditT5 [167] and CodeReviewer [73], as the model to be trained in our approach. The description of the chosen models is as follows:

- **CodeBERT:** CodeBERT is a pre-trained code model based on the BERT architecture [21]. CodeBERT is pre-trained on a large-scale source code dataset using the pre-training scheme of RoBERTa [83]. CodeBERT is the first large pre-trained NL-PL model for both natural language and source code modeling.
- **GraphCodeBERT:** GraphCodeBERT [34] is a large pre-trained code model based on the BERT architecture [21]. Different from CodeBERT which uses the pre-training strategy of RoBERTa [83] only on context information (masked language modeling), GraphCodeBERT also applies pre-training approaches on the data-flow graph, *i.e.*, cross code-graph variable-alignment and data flow edge prediction.
- **CodeGPT:** CodeGPT is a decoder-only source code model, which is based on the GPT architecture and pre-trained on the source code context. Similar to the GPT, CodeGPT is also trained with the autoregressive manner.
- **PLBART:** PLBART is a joint encoder-decoder pre-trained model based on the BART [68] architecture, and pre-trained with the denoising sequence-to-sequence strategy.
- **CodeT5:** CodeT5 is a joint encoder-decoder pre-trained model with the same model architecture of T5 [115], and pre-trained using the text-to-text training strategy as described in T5 [115].
- **CoditT5:** CoditT5 is a recent pre-trained model for code editing based the T5 [115] architecture.
- **CodeReviewer:** CodeReviewer is a recent pre-trained model for code reviewing based the T5 [115] architecture.

We train the above described baselines on the training set of our dataset, and evaluate the performance of them. During the training process, we pre-process the data using the same method described in Section 4.3.1 and 4.4.2 and generate the input data as [$\mathcal{C}$ `<s>` $\mathcal{G}$], where `<s>` is the special token for splitting different modalities, $\mathcal{C}$ and $\mathcal{G}$ denote the source code context information and the natural language guidance respectively. For the baselines, as described before, they are the existing standard sequence-to-sequence multi-modalities training pipeline for source code editing, therefore, their training objective only contains an editing loss (Equation 4.6). Further, we also conduct experiments by training models under the setting in previous work of multi-modalities training for source code editing [16], in which the content of the code line to be edited is also included in the input as another modality. In this way, the input becomes [$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$], where $\mathcal{E}$ denotes the content of the code line to be edited. The experimental results of three fully input modalities can be seen as the *upper bound* of the results of with only source code context and natural language guidance modalities.

Table 4.2: Experimental results of different models trained with all three modalities ([$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]) and with only the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$).

Model	Modalities	BLEU	Top-1 Acc.
CodeBERT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	50.32	41.95
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	65.77	53.24
GraphCodeBERT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	52.58	43.38
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	66.00	53.98
CodeGPT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	44.25	38.58
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	64.50	53.00
PLBART	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	52.20	45.45
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	67.89	57.36
CodeT5	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	56.85	50.87
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	68.41	59.75
CoditT5	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	55.57	49.71
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	69.20	60.33
CodeReviewer	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	57.99	52.65
	[$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]	70.86	62.82

4.5.2 Experimental Results for RQ-1

The experimental results for RQ-1 are shown in Table 4.2, where we present the BLEU score and the Top-1 accuracy of different models' results on the testing set with input modalities of [$\mathcal{C}$ `<s>` $\mathcal{G}$] and [$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$] respectively. From the table, we can see that all the models' performance drops significantly after removing the content of the line to be edited (i.e., after removing $\mathcal{E}$). Specifically, compared to training with all three modalities, the CodeBERT drops 15.45 and 11.29 for BLEU score and Top-1 accuracy respectively. For GraphCodeBERT, the results of trained with [$\mathcal{C}$ `<s>` $\mathcal{G}$] decrease by 13.42 and 10.60 for BLEU and Top-1 accuracy respectively, compared to training with the modalities of [$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]. For the decoder-only model CodeGPT, it drops by 20.25 and 14.42 for BLEU score and Top-1 accuracy respectively when trained without the modality of the content of the line to be edited $\mathcal{E}$. For the joint encoder-decoder pre-trained models PLBART and CodeT5, compared to training with all three modalities, when trained with only the modalities of source code context and natural language guidance, the BLEU scores decrease by 15.69 and 11.56 respectively, and the Top-1 accuracy of them drops by 11.91 and 8.88 respectively.

This phenomenon can be explained as follows: (i) The models do not know where to edit in the source code context, therefore, compared to the upper bound, the models trained with [$\mathcal{C}$ `<s>` $\mathcal{G}$] lacks more information that would help generate the exact edits. (ii) During the training process, the models do not learn how to localize the line-level location where it is to be edited.

Based on these results, we can conclude that the models do not perform well when trained with only the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$), since the models cannot capture the information that is associated to the ground-truth edits directly, and the models also do not learn knowledge and abilities to help themselves localize the line-level location that is to be edited from the given source code context ($\mathcal{C}$) and the natural language guidance ($\mathcal{G}$). Therefore, the answer to the RQ-1 appears:

✍ **Answer 1** ▶ *Under a more practical setting, in which only the source code context $\mathcal{C}$ and the natural language guidance $\mathcal{G}$ can be used to generate edits, standard sequence-to-sequence multi-modalities training pipeline is not enough for models to learn to generate accurate edits, since no more additional knowledge and abilities have been learned to help the model localize and edit exact code line.* ◀

To improve the performance of the models under this more practical scenario, in which only the source code context ($\mathcal{C}$) and the natural language guidance ($\mathcal{G}$) are available, we propose to train the models with the joint optimization target to enable the model learning both editing and localization abilities. To evaluate the effectiveness of our proposed approach, we investigate the next research question:

✍ **RQ-2** ▶ *How does our jLED perform compared to the baseline?* ◀

4.5.3 Experimental Setup for RQ-2

In the second research question, we aim to evaluate the performance of our proposed jLED. In the experiments of RQ-2, we select same pre-trained models used in the experiments of RQ-1, that are CodeBERT [27], GraphCodeBERT [34], CodeGPT [87], PLBART [2], and CodeT5 [150]. For the description of these models, please refer to Section 4.5.1 for details. For each model, we train two baselines, which are the editing baseline and localization baseline respectively. More specifically, the editing baseline, *i.e.* the [$\mathcal{C}$ `<s>` $\mathcal{G}$] parts of Table 4.2 in RQ-1, takes the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$) as the input ([$\mathcal{C}$ `<s>` $\mathcal{G}$]), and uses the editing loss in Equation 4.6 as the loss function to train the model. During inference, the editing baseline can generate the predicted edit, therefore, the evaluation metrics BLEU score and Top-1 accuracy can be used to evaluate the performance of the model. For the localization baseline, it also takes [$\mathcal{C}$ `<s>` $\mathcal{G}$] as the input, and uses the localization loss in Equation 4.7 (for CodeBERT, GraphCodeBERT, PLBART, and CodeT5) or Equation 4.8 (for CodeGPT) as the loss function to train the model. During the inference process of localization baselines, models output the predicted line-level position to be edited, *i.e.*, predicted line number. Then, we use the Top-1 and Top-5 accuracy to evaluate the predicted line-level position.

4.5.4 Experimental results for RQ-2

The experimental results for RQ-2 are presented in Table 4.3, where we show the BLEU score and the Top-1 accuracy of models' editing baselines and jLED (ours), and we also show the Top-1 and Top-5 accuracy of different models' localization

Table 4.3: Experimental results of different models trained with only the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$) as input. All the models use only editing loss to train the editing baseline models, use only localization loss to train the localization baseline models, and use our joint loss function to train models for joint localization and editing.

Model	Modality	Training target	Editing results		Localization results	
			BLEU	Top-1 Acc.	Top-1 Acc.	Top-5 Acc.
CodeBERT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	50.32	41.95	-	-
		Localization Baseline	-	-	62.09	**79.78**
		jLED (Ours)	**55.15**	**46.58**	**62.62**	78.76
GraphCodeBERT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	52.58	43.38	-	-
		Localization Baseline	-	-	74.91	88.24
		jLED (Ours)	**55.71**	**47.20**	**74.96**	**89.34**
CodeGPT	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	44.25	38.58	-	-
		Localization Baseline	-	-	53.75	84.80
		jLED (Ours)	**49.90**	**42.71**	**62.49**	**85.28**
PLBART	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	52.20	45.45	-	-
		Localization Baseline	-	-	66.85	83.89
		jLED (Ours)	**54.78**	**48.84**	**70.09**	**86.62**
CodeT5	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	56.85	50.87	-	-
		Localization Baseline	-	-	**77.29**	88.93
		jLED (Ours)	**61.08**	**55.20**	77.07	**89.33**
CoditT5	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	55.57	49.71	-	-
		Localization Baseline	-	-	**74.15**	88.22
		jLED (Ours)	**58.76**	**52.56**	73.16	**89.15**
CodeReviewer	[$\mathcal{C}$ `<s>` $\mathcal{G}$]	Editing Baseline	57.99	52.65	-	-
		Localization Baseline	-	-	**72.75**	89.38
		jLED (Ours)	**61.23**	**55.64**	72.38	**89.93**

baselines and jLED (ours). As described before, in all the experiments for RQ-2, all the models' editing baselines, localization baselines, and jLED take the modalities of [$\mathcal{C}$ `<s>` $\mathcal{G}$] as input.

Our experimental results presented in Table 4.3 offer a wealth of insights into the performance of our approach, jLED, compared to various baselines across multiple metrics. Most notably, jLED consistently outshines all the editing baselines in both BLEU score and Top-1 accuracy, irrespective of the underlying model architecture.

Specifically, when integrated with CodeBERT, jLED yields a significant uplift of **4.83** and **4.63** in BLEU score and Top-1 accuracy, respectively. GraphCodeBERT's performance also receives considerable boosts of **3.13** and **3.82** in BLEU and Top-1 metrics, reinforcing jLED's adaptability across different coding paradigms. The improvements are not merely incremental but substantial, highlighting jLED's generalization capabilities across different models.

Similarly, our jLED improves the editing baseline of CodeGPT by **5.65** and **4.13** for BLEU score and Top-1 accuracy. For PLBART, jLED also boost the editing baseline by **5.65** and **4.13** for both metrics, further signifying its cross-model efficacy. CodeT5's performance also escalates, with gains of **5.23** and **4.33** in BLEU score and Top-1 accuracy, underscoring the versatility of jLED in adapting to varied model architectures and optimization landscapes.

On the other hand, with the assistance of the editing loss, the model can also localize the editing position more precisely compared to the localization baseline which is trained with only the localization loss. Specifically, compared to the localization baselines, the Top-1 accuracy improves by 0.53, 0.05, 8.74, and 3.24 for CodeBERT, GraphCodeBERT, CodeGPT, and PLBART respectively. For the Top-5 accuracy, the GraphCodeBERT, CodeGPT, PLBART, and CodeT5 outperform their localization baselines by 1.10, 0.48, 2.73, and 0.40, respectively. This suggests that jLED doesn't merely generate more precise edits, but also benefits the localization process, which is crucial for practical implementation.

In summary, these results demonstrate that our jLED enables models to acquire greater knowledge and abilities in localizing editing locations within the code context. This acquired knowledge and these abilities help the models focus on more precise potential editing locations when generating edits, leading to the production of more accurate edits. Additionally, learning to generate edits can also help the models learn to localize the editing position line more accurately. This also substantiates jLED's superior performance and adaptability across different model architectures and evaluation metrics. Whether in terms of edit quality or location precision, jLED consistently advances the state-of-the-art, making it a robust solution for automated code editing tasks.

Therefore, based on these experimental results and findings, we can conclude the answer to the RQ-2 as:

✍ **Answer 2** ▶ *Compared to the baselines, our jLED pipeline enables the models to learn additional knowledge and abilities regarding to localizing the editing location. Therefore, given only the modalities of source code context $\mathcal{C}$ and natural language guidance $\mathcal{G}$, the models, which are trained by our jLED pipeline, can pay more attention to the potential editing location, so that yielding more precise and location-related edits.* ◀

✍ **RQ-3** ▶ *How does our joint training pipeline perform compared to the two-stage localization-editing pipeline?* ◀

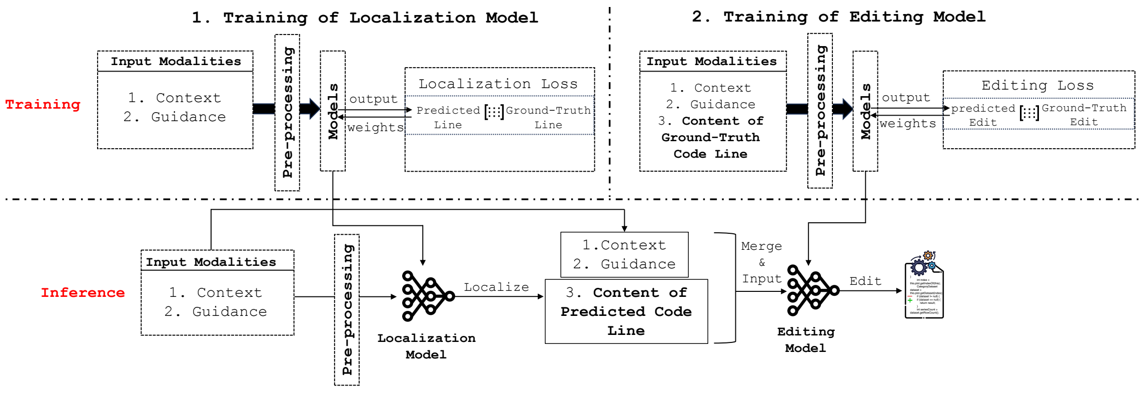


Figure 4.5: Overview of the two-stage localization-editing pipeline.

4.5.5 Experimental Setup for RQ-3

Based on the experiments of RQ-2, we prove that our proposed joint training approach performs better than the baseline, since it enables the models to learn more knowledge and abilities in localizing editing locations with the additional localization loss. This also proves that it is crucial for models to know locations before editing. However, there also is another manner that can provide models with some location information (predicted locations) except the joint training manner, that is, the two-stage localization-editing pipeline (we simply call it the two-stage pipeline in the rest of this paper). The overview of the two-stage pipeline is shown in Figure 4.5. During the training process of the two-stage pipeline, we first train the model with the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$) as input ([$\mathcal{C}$ `<s>` $\mathcal{G}$]), which is trained using only the localization loss (Equation 4.7 or 4.8) as the optimization target (same with the localization baseline in RQ-2). We name it as the localization model in the two-stage pipeline. Then, we train another model with the input of [$\mathcal{E}_{GT}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$], where $\mathcal{E}_{GT}$ denotes the content of the ground-truth code line to be edited. This model is named as an editing model in the two-stage pipeline, optimized using the editing loss (Equation 4.6). During inference, given a sample with the modalities of code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$), the trained localization model is used to predict the line-level location, which can be used to inquire the corresponding line-level content in the code context $\mathcal{C}$. We use $\mathcal{E}_{pred}$ to denote the inquired line-level content. After that, we concatenate the predicted line-level content to be edited ($\mathcal{E}_{pred}$), the source code context ($\mathcal{C}$), and the natural language guidance ($\mathcal{G}$) as the multimodal input [$\mathcal{E}_{pred}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$] of the trained editing model, to generate the predicted edits. Finally, we use the BLEU score and the Top-1 accuracy as the evaluation metrics for the generated edits. For the choice of the localization and editing models, we use the same settings in the experiments of RQ-1 and RQ-2, *i.e.*, CodeBERT [27], GraphCodeBERT [34], CodeGPT [87], PLBART [2], and CodeT5 [150].

Table 4.4: Experimental results of different models trained with our joint training approach and the two-stage approach.

Model	Approach	BLEU	Top-1 Acc.
CodeBERT	Two-stage	47.95	41.65
	jLED (Ours)	**55.15**	**46.58**
GraphCodeBERT	Two-stage	45.62	37.04
	jLED (Ours)	**55.71**	**47.20**
CodeGPT	Two-stage	40.80	36.65
	jLED (Ours)	**49.90**	**42.71**
PLBART	Two-stage	50.59	45.42
	jLED (Ours)	**54.78**	**48.84**
CodeT5	Two-stage	32.96	31.51
	jLED (Ours)	**61.08**	**55.20**

4.5.6 Experimental results for RQ-3

The experiments results of RQ-3 are presented in Table 4.4, where we show the evaluation results of the two-stage training approach and our joint training approach regarding CodeBERT, GraphCodeBERT, CodeGPT, PLBART, and CodeT5 respectively. From the table, we can see that, for all the models, our proposed joint training outperforms the two-stage training approach significantly. Specifically, for CodeBERT, our approach outperforms the two-stage method by **7.20** and **4.93** for BLEU score and Top-1 accuracy respectively. For GraphCodeBERT, our approach improves the two-stage approach by **10.09** and **10.16** for BLEU score and Top-1 accuracy. For CodeGPT, compared to the two-stage approach, our joint training has the improvement of **9.10** and **6.06** for BLEU score and Top-1 accuracy respectively. For PLBART, our approach improves the two-stage method by **4.19** and **3.42** for BLEU and Top-1 accuracy respectively. For CodeT5, compared to the two-stage method, our proposed joint training approach outperforms it by **28.12** and **23.69** respectively. These experimental results demonstrate the superiority and the significant improvement of our proposed joint training approach over the two-stage method, so we can conclude that our joint training is a better choice compared to the two-stage approach. The answer to RQ-3 can be denoted as:

✍ **Answer 3▶** *The two-stage method, the editing performance of which heavily relies on the localization model's predicted line-level location, which means that, when the predicted line-level location is incorrect, the generated edits in the next stage has a high probability of being wrong – even the input content of location to be edited ($\mathcal{E}$) is perfect, the generated edits are not 100% correct, let alone the input location is wrong. Compared to it, our joint training pipeline does not totally rely on the predicted localization results to generate the edits, so that the models could pay some attention to the correct code lines, even if they are not with the highest probability scores. Based on these experimental results, we conclude that our joint training pipeline is a better choice compared to the two-stage method.* ◀

4.6 Discussion

4.6.1 Compare to Another Existing Joint Learning Approach

Table 4.5: Experimental results compared to another existing joint learning method CodeT5-DLR.

Method	Editing results		Localization results	
	BLEU	Top-1 Acc.	Top-1 Acc.	Top-5 Acc.
CodeT5-DLR	26.84	21.38	24.29	57.38
jLED (Ours, using CodeT5)	**61.08**	**55.20**	**77.07**	**89.33**

To compare our proposed jLED with the existing joint learning approach CodeT5-DLR [12], designed for bug detection, localization, and repair without natural language guidance, we conducted an experiment on it. CodeT5-DLR utilizes the pre-trained CodeT5 as its base model, and fine-tune it to process code snippets and simultaneously predict 1) the presence of bugs, 2) the location of the buggy line, and 3) the repair results.

In the context of our natural language-guided code editing, we fine-tune the CodeT5-DLR on our dataset, during which we follow the pre-processing and tokenization process described in their original paper, and take the tokenized code snippet as the input of the model. For the fine-tuning loss function, as our task does *not* include the bug detection objective, we remove the binary classifier for bug detection and only keep the localization and repair training objective.

The experimental results are illustrated in Table 4.5. We can observe that our jLED significantly outperforms the CodeT5-DLR. This indicates that our approach is more suitable for the natural language-guided code editing task compared to CodeT5-DLR. The underperformance of CodeT5-DLR could be due to the lack of natural language guidance, which is essential for understanding the specific purpose of each editing sample. Unlike bug repair, which only requires learning the purpose and underlying patterns for fixing bugs, code editing involves a variety of modification purposes. Therefore, while CodeT5-DLR performs well in bug repair without natural language guidance, it struggles in code editing if the guidance is not provided.

4.6.2 Parameter Study

To investigate the impact of the hyperparameter λ in our joint learning loss function (Eq.4.9) on different models, we conduct a parameter study using different values of λ. We select the models CodeBERT [27], GraphCodeBERT [34], CodeGPT [87], and CodeReviewer [73] for this analysis.

Theoretically, very small values of λ may lead to the under-optimization of the localization target, thereby limiting the benefits to the editing target. In contrast, very large values may cause the localization target to dominate the training process, leading to under-optimization of the editing target. Therefore, it is crucial to determine an appropriate value of λ to balance the two optimization objectives and achieve optimal code editing performance.

In this parameter study, for each model, we trained them using λ values of 0.01, 0.05, 0.1, 0.5, 1.0, 5.0, and 10.0. The results are presented in Table 4.6. It can be observed that when the value of λ is set to 0.1, all these selected models achieve their best editing performance, which demonstrates that 0.1 is a suitable selection of the value of λ to balance the value scale of editing and localization loss.

In general, in deep learning model training, having two targets of equal importance does not necessarily require assigning them identical loss weights. This is because gradient scales and convergence speeds can vary significantly across different learning tasks. To some extent, this also explains why a value like 0.1 is appropriate for λ in our scenario.

Moreover, for very large λ values, such as 100, the training becomes unstable due to the amplification of the gradient by a factor of 100, which may even lead to gradient explosion. Therefore, these values are deemed incompatible with our proposed method.

4.6.3 Experiments on Another Existing Code Edit/Refinement Dataset

To evaluate the generalizability of our proposed jLED over other datasets, we conduct experiments on an existing code edit/refinement dataset used in the paper of CodeReviewer [73]. We apply the same pre-processing method as described in Section 4.3.1 on this dataset. The experimental results are presented in Table 4.7, where we show the results of CodeBERT [27], CodeT5 [150], and CodeReviewer [73] with the training targets of editing baseline, localization baseline, our proposed joint training targets (jLED), and the editing upperbound (training and inference with all three modalities, *i.e.*, [$\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}$]), respectively. From the experimental results, we can observe that, compared to the editing baseline and the localization baseline, our proposed jLED improves both of them significantly on all three models, further demonstrating the superiority and generalizability of our proposed method.

Additionally, we observe that the gap between the editing baseline and the editing upper bound is not as significant as that observed in our dataset. This is because our dataset contains more code lines in each individual input sample, increasing the model's training difficulty in generating accurate edits without explicit location

Table 4.6: Parameter study on the value of λ.

Model	λ	Editing results		Localization results	
		BLEU	Top-1 Acc.	Top-1 Acc.	Top-5 Acc.
CodeBERT	0.01	54.81	45.75	56.73	77.98
	0.05	53.58	45.40	52.95	67.09
	0.1	55.15	46.58	62.62	78.76
	0.5	54.75	45.51	60.49	79.05
	1.0	52.20	43.40	62.49	80.36
	5.0	46.79	37.75	67.40	86.36
	10.0	45.09	36.60	60.82	80.44
GraphCodeBERT	0.01	53.97	47.10	69.90	88.89
	0.05	54.79	47.00	71.73	89.73
	0.1	55.71	47.20	74.96	89.34
	0.5	54.83	46.44	76.07	89.31
	1.0	53.21	43.98	77.64	89.69
	5.0	48.67	40.15	74.96	88.90
	10.0	44.50	35.58	69.49	86.96
CodeGPT	0.01	48.84	42.18	46.93	84.07
	0.05	48.11	41.82	58.82	86.04
	0.1	49.90	42.71	62.49	85.28
	0.5	46.93	39.95	59.24	85.33
	1.0	42.18	36.60	63.09	86.00
	5.0	46.92	38.89	60.75	86.15
	10.0	44.64	36.64	60.95	86.07
CodeReviewer	0.01	60.16	55.02	57.62	90.22
	0.05	60.42	55.15	69.24	91.05
	0.1	61.23	55.64	72.38	89.93
	0.5	60.08	54.71	78.33	90.22
	1.0	60.04	54.51	76.78	90.33
	5.0	55.77	49.36	68.55	90.35
	10.0	54.38	47.11	69.62	90.58

information. As a result, our dataset better represents the real-world challenges of code editing tasks, providing a more rigorous benchmark for evaluating model performance.

4.6.4 Threats to Validity

The internal threat to validity lies in the bias of the characteristic of the model in our pipeline. To reduce this threats, we conduct experiments with different models for both baselines and our proposed pipeline. Further, we consider that it is not fair to compare different approaches implemented by different models, *e.g.*, compare our joint training approach implemented by CodeBERT to the CodeT5-based baseline, therefore, for fair comparison, we only compare different approaches within the same model.

Threats of external validity refer to the dataset used for the experiment. To reduce this threat, we construct a well-established dataset with high-quality data samples for training, validating, and testing our approach and the baselines.

Table 4.7: Experimental results on the Code Refinement dataset in the paper of CodeReviewer [73].

Model	Modality	Training target	Editing results		Localization results	
			BLEU	Top-1 Acc.	Top-1 Acc.	Top-5 Acc.
CodeBERT	$[\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Baseline	42.25	24.80	-	-
		Localization Baseline	-	-	88.47	92.79
		jLED (Ours)	**46.81**	**28.30**	**92.98**	**98.05**
	$[\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Upperbound	47.68	30.18	-	-
CodeT5	$[\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Baseline	46.95	33.73	-	-
		Localization Baseline	-	-	93.61	97.80
		jLED (Ours)	**49.78**	**36.38**	**94.10**	**98.29**
	$[\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Upperbound	50.81	37.89	-	-
CodeReviewer	$[\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Baseline	47.88	36.67	-	-
		Localization Baseline	-	-	94.66	97.77
		jLED (Ours)	**49.38**	**38.44**	**94.70**	**99.15**
	$[\mathcal{E}$ `<s>` $\mathcal{C}$ `<s>` $\mathcal{G}]$	Editing Upperbound	50.59	38.49	-	-

4.6.5 Limitations

Our approach mainly focuses on single-line source code editing. However, in the real-world scenario, multi-line editing appears frequently, which limits our proposed approach to fully address the complexities of real-world software development where multi-line editing is a common necessity. This limitation potentially hinders the broad applicability of our method, particularly in situations where changes span several lines of code, as is often the case in bug fixes, feature enhancements, or code refactoring. To mitigate this, one potential solution is to employ binary cross-entropy loss instead of the current standard cross-entropy loss during training, and set a threshold to filter multiple results as the predicted lines. This adjustment would enable our model to predict edits across multiple lines. Such a modification aims to enhance the localization capabilities of our approach, allowing for simultaneous multi-line source code editing. In summary, we acknowledge the complexity of integrating multi-line editing capabilities and recognize this as an area for future work to be rigorously pursued and refined.

4.7 Conclusion

In this paper, we investigate the performance of existing standard sequence-to-sequence multi-modal learning for code editing under a more practical situation, in which the precise line-level location information is unknown or unavailable. Through comprehensive experiments, we have confirmed that the models are not able to generate precise edits after training without exact line-level location information. To tackle this challenge, we proposed jLED (**j**ointly **L**ocalize and **ED**it), a training pipeline to jointly learn to localize the edit buggy code simultaneously, which enables models to learn additional knowledge and abilities regarding the line-level localization while learning to edit. We conduct experiments using our proposed jLED, and the experimental results show that our approach not only generates more precise edits, but also predicts more accurate line-level locations than the considered baselines. Moreover, to evaluate the effectiveness of our joint learning pipeline against other localization and editing alternatives, we construct a two-stage localization-editing pipeline, in which a localization model and an editing model are trained separately. Experimental results demonstrate the superiority of our jLED over this two-stage manner.

5 Enhancing LLMs for Cross-lingual Code Clone Detection with Code Summarization

In this chapter, we investigate the use of large language models (LLMs) for cross-lingual Type-4 code clone detection, which involves identifying semantically equivalent code across different programming languages. We evaluate both zero-shot and fine-tuned settings, and introduce a summarization-based input augmentation strategy that enhances fine-tuning by providing natural language descriptions of code functionality. Experiments on XLCoST and CodeNet show that our method significantly improves performance across multiple metrics and models, highlighting the effectiveness of LLMs for multilingual semantic code understanding.

This chapter is based on the work published in the following research paper, which is current under review:

- Pian, W., Sun, T., Tian, H., Klein, J. and Bissyandé, T.F.. Enhancing LLMs for Cross-lingual Code Clone Detection with Code Summarization. Under review.

Contents

5.1 Introduction

Detecting semantically equivalent code snippets across different programming languages—also known as cross-lingual Type-4 code clone detection [125, 24, 101, 100, 169, 61]—is becoming increasingly critical in software engineering. Such capabilities have significant implications for cross-language code reuse, plagiarism detection, software maintenance, vulnerability propagation analysis, and facilitating collaboration in multilingual development teams [175, 100]. However, due to the significant differences in syntax, naming conventions, and libraries across programming languages, detecting semantically similar code across different programming languages remains highly challenging.

Traditional code clone detection techniques, such as token-based [26, 166, 58], AST-based [155, 46, 156], or graph-based methods [94, 177, 79, 47, 109], often struggle to generalize well across different programming languages due to their reliance on syntactic similarities. Recently, the emergence of Large Language Models (LLMs) has sparked new interest in addressing this cross-lingual challenge. With the remarkable semantic understanding and generative capabilities, LLMs have shown promising results in various code-related tasks [153, 59, 102, 4], including code generation, summarization, translation, and zero-shot classification. However, the direct application of these models to cross-lingual code clone detection has been limited.

Detecting Type-4 code clones—semantically equivalent code fragments that may differ significantly in syntax or even be written in different programming languages—is a critical task in software engineering. Effective cross-lingual clone detection facilitates software maintenance, reuse, and plagiarism detection in increasingly multilingual codebases. However, due to the significant differences in syntax, naming conventions, and libraries across programming languages, detecting semantically similar code across languages remains highly challenging.

Recent attempts [101, 100] have primarily explored leveraging the zero-shot capabilities of LLMs for cross-lingual clone detection. These approaches typically prompt pre-trained LLMs to classify or measure semantic similarity between code snippets without additional training on domain-specific datasets. Nevertheless, such zero-shot pipelines have exhibited limited success in detection performance due to insufficient adaptation to code-specific semantic nuances and inherent differences in programming paradigms across languages.

Motivated by the limitations observed in prior zero-shot approaches, in this paper, we explore whether fine-tuning LLMs specifically on multilingual code data can effectively improve their performance for cross-lingual clone detection tasks. Furthermore, recognizing that cross-lingual code snippets often vary significantly in length, structure, and level of abstraction, we propose to integrate a summarization component into our fine-tuned LLM framework. This summarization module aims to distill the core semantic essence of code snippets, facilitating more effective cross-language comparisons by mitigating language-specific syntactic, structural, and idiomatic biases.

To systematically evaluate our proposed approach, we structure our research in three research questions (RQs):

- **RQ-1**: How effective are LLMs' zero-shot capabilities in cross-lingual code clone detection?

- **RQ-2**: To what extent does fine-tuning LLMs on multilingual code data enhance detection performance?
- **RQ-3**: How does integrating our proposed summarization component further improve the effectiveness of fine-tuned LLMs in detecting cross-lingual Type-4 code clones?

Through extensive experiments conducted across multiple programming languages, we demonstrate substantial improvements our approach achieves over prior zero-shot baselines. Our results highlight that targeted fine-tuning significantly enhances LLMs' semantic understanding and adaptation to code-specific contexts. Additionally, we show that the summarization step provides complementary benefits by effectively reducing noise and extracting high-level semantic information, achieving state-of-the-art performance in cross-lingual code clone detection tasks. This study not only advances our understanding of LLMs' capabilities in handling cross-language semantic challenges but also sets a foundation for further research in multilingual software engineering applications. Our findings offer insights into improving automated tools for code search, software security assessments, plagiarism detection, and fostering efficient cross-language software development practices. Our key contributions are summarized as follows:

❶ We conduct a comprehensive evaluation of state-of-the-art LLMs for cross-lingual Type-4 code clone detection and demonstrate that fine-tuning on multilingual code data substantially improves model performance over zero-shot baselines.

❷ We propose a novel framework that augments input with natural language descriptions generated via a prompt-based summarization module. By jointly leveraging diverse prompt templates and semantically rich summaries, our method enhances the model's ability to reason about functional equivalence across different programming languages.

❸ Through extensive experiments on XLCoST and CodeNet across ten programming languages, we show that our approach consistently improves Accuracy, Recall, and F1-score across multiple LLMs, achieving state-of-the-art performance. Our results validate the effectiveness and generalizability of the proposed framework, and offer a new perspective for addressing cross-lingual code clone detection with LLMs.

The remainder of this paper is presented as follows. In Section 5.2, we introduce the background of this work. In Section 5.3, we present our proposed method. Section 5.4 and 5.5 present the experimental setup and the experimental results, respectively. Then, the discussion are provided in Section 5.6. Finally, we conclude this work in Section 5.7.

To facilitate replication, we have made source code available at `https://osf.io/59xdj/?view_only=f7b3976fbde1456c82f9fc02e32865fa`

5.2 Background

This section provides foundational knowledge essential for understanding our research. Specifically, we present key concepts related to code clones and Large Language Models, which form the central themes of this study.

5.2.1 Code Clones

A code clone refers to two or more code fragments that are similar based on a defined notion of similarity. Such similarities arise naturally in software development due to practices like code reuse, copy-paste programming, or design patterns implementation. Detecting and understanding code clones can assist developers in software maintenance, refactoring, plagiarism detection, and bug identification. Following the widely adopted classification [119], code clones are typically categorized into four primary types:

- **Type-1 (Exact Clones)**: These are identical code fragments except for differences in whitespace, formatting, layout, and comments. They are the simplest form of clones and can typically be identified through straightforward textual comparisons.
- **Type-2 (Lexical Clones)**: This type extends Type-1 clones by allowing variations in variable names, function names, literals, and data types. Despite these lexical differences, the underlying code structure and logic remain identical. Such clones require lexical analysis tools for detection.
- **Type-3 (Syntactic Clones)**: Type-3 clones represent code fragments that are syntactically similar yet differ structurally at the statement level. Differences can include added, removed, or rearranged statements, conditional structures, or loops, in addition to the lexical variations observed in Type-2 clones. Detecting Type-3 clones typically demands more advanced parsing and syntactic analysis techniques.
- **Type-4 (Semantic Clones)**: These code snippets perform the same or highly similar functionality but differ significantly in syntax and structure. Semantic clones are the most challenging to detect because they require an understanding of the functional intent rather than relying solely on syntactic similarity. Detection usually involves sophisticated analysis, including dynamic execution, symbolic execution, or semantic embeddings.

In the context of increasingly global software development and cross-platform applications, another important category is:

- **Cross-lingual Code Clones**: These represent semantic code clones whose code fragments are implemented in different programming languages but exhibit equivalent functionality. Identifying such clones is particularly challenging due to differences in syntax, paradigms, language-specific idioms, and coding conventions across languages. Cross-lingual clone detection can support tasks such as code translation, knowledge transfer between programming communities, and multilingual software maintenance.

5.2.2 Large Language Models

Large Language Models (LLMs) are deep neural networks trained on vast amounts of textual data, enabling them to understand, generate, and reason about natural language text. Over recent years, these models have transformed the fields of natural language processing (NLP) and artificial intelligence due to their remarkable capabilities across a wide array of tasks, such as text classification [168], machine translation [174], summarization/captioning [110], question-answering [35, 176], code generation [80, 23], reasoning [48, 104], etc.

A key feature of LLMs is their ability to encode rich semantic and syntactic representations, which emerge through extensive self-supervised pre-training on large-scale datasets. These models are based on the Transformer [145] architecture, which leverages the self-attention mechanism to enable the model to weigh the significance of different parts of the input dynamically.

Beyond traditional language or code models [108, 178, 112], recent advancements in LLMs, such as CodeLlama [120], StarCoder [71], etc, have demonstrated promising abilities in code-related tasks. These models can effectively capture programming logic, syntax, semantics, and even software-specific conventions. Leveraging this capability, researchers and practitioners increasingly apply LLMs to programming tasks, including code completion, code summarization, bug detection, code repair, etc. In our study, we exploit the unique strengths of LLMs to facilitate the challenging task of cross-lingual code clone detection. Specifically, we investigate whether LLMs can reliably encode semantic knowledge across multiple programming languages and utilize these representations to detect semantic similarities among code snippets, even when they differ considerably at the syntactic level.

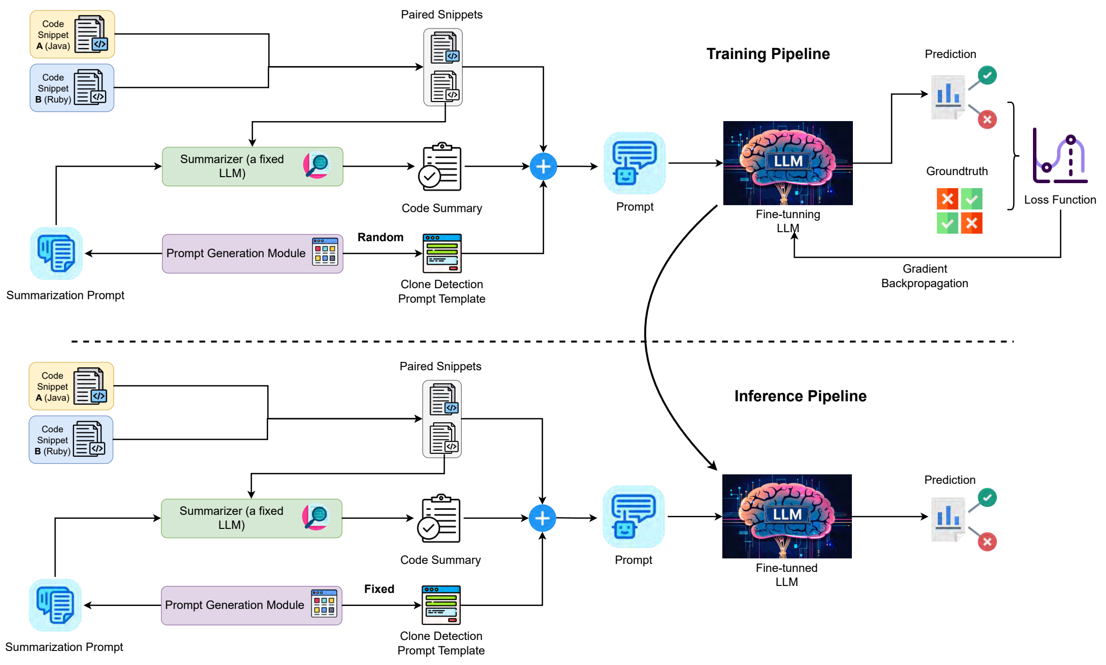


Figure 5.1: The overview of our training and inference pipeline.

5.3 Approach

In this section, we present the details of our proposed approach for cross-lingual code clone detection. The framework primarily consists of a prompt generation module, a pre-processing module, a code summarizer, and a Large Language Model (LLM) backbone. An overview of the framework is illustrated in Figure 5.1. During training, given a pair of code snippets, the Prompt Generation Module randomly selects from a set of predefined templates to generate Clone Detection Prompts that query whether the two cross-lingual snippets are clones. The resulting prompt is then paired with the code snippet pair to construct the initial input for the LLM. Additionally, the Prompt Generation Module can also produce Summarization Prompts, which are used to guide the Summarizer in generating natural language descriptions for each code snippet. These descriptions are then combined with the original code snippets and the Clone Detection Prompt to form the final, comprehensive input to the LLM. Finally, the LLM is trained by calculating the loss between its output prediction and the ground-truth. During inference, the Prompt Generation Module creates a fixed Clone Detection Prompt, which is combined with the generated code summary and the paired code snippets to form the final comprehensive input for the fine-tuned LLM to make predictions.

Specifically, given a pair of code snippets $\boldsymbol{x}_i^1$, $\boldsymbol{x}_i^2$, and their label $\boldsymbol{y}_i$, the Prompt Generation Module firstly generate prompts for them:

$$PGM \rightarrow \boldsymbol{p}_i^d, \boldsymbol{p}_i^s, \tag{5.1}$$

where PGM denotes the Prompt Generation Module, $\boldsymbol{p}_i^d$ and $\boldsymbol{p}_i^s$ present the generated Clone Detection Prompt and Summarization Prompt for sample pair i, respectively. After that, the Summarization Prompt $\boldsymbol{p}_i^s$ is combined with each code snippet as the input of the Summarizer to obtain the detailed code description/summary:

$$\boldsymbol{s}_i^1 = \mathcal{S}([\boldsymbol{p}_i^s, \boldsymbol{x}_i^1], \boldsymbol{\Theta}_s), \quad \boldsymbol{s}_i^2 = \mathcal{S}([\boldsymbol{p}_i^s, \boldsymbol{x}_i^2], \boldsymbol{\Theta}_s), \tag{5.2}$$

where $\mathcal{S}(\cdot, \boldsymbol{\Theta}_s)$ denotes the Summarizer with fixed parameters $\boldsymbol{\Theta}_s$. In our framework, we implement the Summarizer with a frozen pre-trained LLM. After obtaining the description/summary of the code snippets, they are combined with the code snippets and the Clone Detection Prompt to construct the comprehensive input of the LLM component:

$$\hat{\boldsymbol{y}}_i = \mathcal{F}([\boldsymbol{p}_i^s, \boldsymbol{x}_i^1, \boldsymbol{x}_i^2, \boldsymbol{s}_i^1, \boldsymbol{s}_i^2], \boldsymbol{\Theta}_d), \tag{5.3}$$

where $\mathcal{F}(\cdot, \boldsymbol{\Theta}_d)$ denotes the LLM detector of our framework with trainable parameters $\boldsymbol{\Theta}_d$, which is training with LoRA [43] during our training process. Then, we calculate the loss function between the LLM detector's prediction $\hat{\boldsymbol{y}}_i$ and the ground-truth $\boldsymbol{y}_i$:

$$\mathcal{L} = \mathbb{E}_{(\boldsymbol{x}_i^1, \boldsymbol{x}_i^2 \sim \mathcal{D})} \mathcal{L}_{CE}(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i), \tag{5.4}$$

where $\mathcal{D}$ denotes the training set, and $\mathcal{L}_{CE}$ is the cross-entropy loss function. Finally, the overall training process can be formulated as:

$$\boldsymbol{\Theta}_d' = \underset{\boldsymbol{\Theta}_d}{\operatorname{argmin}} \, \mathbb{E}_{(\boldsymbol{x}_i^1, \boldsymbol{x}_i^2 \sim \mathcal{D})} \mathcal{L}_{CE}(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i), \tag{5.5}$$

where $\boldsymbol{\Theta}_d'$ denotes the parameters of the fine-tuned LLM.

5.4 Experimental Setup

In this section, we begin by presenting the Research Questions (RQs) that guide our study. We then provide a detailed overview of the Large Language Models (LLMs) employed in our experiments, along with the baseline methods used for comparison. Following that, we describe the dataset utilized in our experiments, including the specifications of the training, validation, and testing splits. Finally, we outline the key implementation details to facilitate reproducibility and further understanding of our experimental setup.

5.4.1 Research Questions

To systematically investigate the performance and behavior of Large Language Models (LLMs) in the task of cross-lingual code clone detection, we formulate and address three core research questions. These questions are designed to guide our empirical study and provide a structured understanding of the capabilities and limitations of LLMs in this context. They collectively examine the impact of zero-shot inference, fine-tuning with multilingual data, and the integration of auxiliary components to enhance detection quality.

- **RQ-1: How effective are LLMs' zero-shot capabilities in cross-lingual code clone detection?**
  This question evaluates the ability of pre-trained LLMs to detect Type-4 code clones across different programming languages without any task-specific fine-tuning. We investigate whether these models, when prompted with carefully designed instructions, can generalize their understanding of code semantics to the cross-lingual setting. This evaluation serves as a baseline for understanding the intrinsic reasoning power of LLMs in a zero-shot context.
- **RQ-2: To what extent does fine-tuning the LLM on multilingual code data enhance detection performance?**
  While LLMs exhibit strong generalization, this question focuses on whether their performance can be further improved by adapting them to the task using a curated multilingual code clone dataset. We explore the benefits of task-specific supervision, assess improvements over zero-shot baselines.
- **RQ-3: How does integrating our proposed summarization component further improve the effectiveness of fine-tuned LLMs in detecting cross-lingual Type-4 code clones?**
  This question investigates the role of a summarization-based auxiliary module that condenses each code snippet into a concise functional description. We hypothesize that incorporating such summaries alongside code pairs enhances the model's understanding by highlighting core semantics, thereby improving Type-4 clone detection across languages. Our experiments quantify this improvement and analyze its consistency across different LLM model architectures.

5.4.2 Models

We conducted our experiments using four advanced Large Language Models (LLMs) that are known for their effectiveness in programming-related tasks and cross-lingual understanding. The selection of these models was guided by several factors, including open access, strong performance in recent benchmarks, multilingual training, and relevance to software engineering applications. All selected models are instruction-tuned or chat-optimized, making them well-suited for prompt-based code reasoning and generation tasks. Below, we provide a detailed overview of each model used in this study.

- **Llama-2-Chat-7B**: Llama-2-Chat-7B [140] is a chat-optimized version of the Llama-2 series developed by Meta AI. It is a 7-billion parameter model fine-tuned for dialogue and multi-turn interactions. Llama-2-Chat is trained on a diverse mixture of publicly available text and code data, enabling it to handle a broad range of natural language and programming tasks. The model benefits from alignment tuning using reinforcement learning from human feedback (RLHF), which improves its ability to follow instructions and generate safe, coherent responses. Although it was not specifically fine-tuned on multilingual code, its general-purpose capabilities and strong instruction-following behavior make it a relevant baseline in our cross-lingual code clone detection setup.

- **Llama-3.1-8B-Instruct**: Llama-3.1-8B-Instruct [1] is a more recent version in the Llama family, incorporating architectural and training improvements over its predecessors. Released with enhanced instruction-following ability, Llama-3.1-Instruct demonstrates superior reasoning, language understanding, and multilingual support. It has been trained with a broader mixture of data, including code from multiple programming languages and natural language prompts, which enhances its ability to perform cross-lingual code reasoning. In our experiments, Llama-3.1-Instruct serves as one of the most capable models, particularly in settings that require nuanced semantic understanding of code snippets written in different languages.
- **StarCoder2-7B**: StarCoder2-7B [86] is a 7-billion-parameter multilingual code language model developed as part of the BigCode project. It is trained on The Stack v2.1, a permissively licensed dataset containing source code from over 600 programming languages. StarCoder2-7B is designed specifically for code generation, translation, summarization, and understanding tasks, and it incorporates a mixture of English and multilingual programming content. While smaller than the 15B variant, the 7B version maintains competitive performance across a wide range of code benchmarks due to its efficient architecture and targeted training objectives. Its compact size also makes it more practical for scalable evaluations. In our experiments, StarCoder2-7B serves as a strong code-centric model, providing robust baseline results in detecting cross-lingual Type-4 code clones. Its pretraining on diverse programming languages enables it to reason about code semantics even when syntax and structure differ significantly across languages.
- **Qwen2.5-Coder-7B-Instruct**: It is a 7-billion-parameter instruction-tuned model from the Qwen2.5-Coder [49] series developed by Alibaba DAMO Academy. Unlike the general-purpose Qwen2.5 [158] models, the Qwen2.5-Coder series is specifically optimized for programming tasks, with training focused on high-quality code corpora across multiple programming languages. The Instruct variant is further fine-tuned to follow task-specific instructions, enabling it to perform more accurately in prompt-based scenarios such as code reasoning, generation, and transformation. With its multilingual code training and strong instruction-following ability, Qwen2.5-Coder-7B-Instruct is well-suited for complex tasks like cross-lingual Type-4 code clone detection, where understanding functional equivalence across syntactically diverse code is essential. In our study, we leverage this model to evaluate how instruction tuning and code specialization jointly contribute to its clone detection performance.

Baselines. To fairly evaluate the effectiveness of our proposed method, we establish two baseline configurations for each model: zero-shot and fine-tuning without summarization. The zero-shot setting directly prompts each LLM to perform cross-lingual Type-4 code clone detection without any task-specific parameter updates. This baseline highlights the intrinsic reasoning capabilities of the pre-trained and instruction-tuned models under prompt-based inference. The fine-tuning baseline, on the other hand, involves supervised training on our multilingual code clone dataset using only the original code snippets as input. This setting assesses how much performance improvement can be achieved through task-specific adaptation, excluding any additional modules. Importantly, this baseline does not include our proposed summarization component. As such, it serves as a controlled comparison point to isolate the contribution of summarization in our full method.

5.4.3 Datasets

To ensure a robust and comprehensive evaluation across diverse programming languages, we utilize two cross-lingual benchmarks: XLCoST [173] and CodeNet [114]. These datasets are leveraged for different phases of our experiment: XLCoST is used exclusively for training, while CodeNet is used for both validation and testing. For all datasets, we construct code snippet pairs following the same procedure described in [100], which ensures consistency with prior work in cross-lingual code clone detection.

- **XLCoST**: The XLCoST [173] dataset is collected from GeeksForGeeks[1], a widely used platform for computer science education and coding practice. It contains true cross-lingual Type-4 clone pairs manually curated across seven programming languages: C, C++, C#, Java, JavaScript, PHP, and Python. Since XLCoST includes only positive clone pairs by default, we construct an equal number of negative pairs by randomly sampling non-equivalent snippets from different problem sets, following the strategy proposed in [100]. The final training set consists of 6000 pairs, evenly split between 3000 positive and 3000 negative samples.
- **CodeNet**: The CodeNet [114] dataset comprises a large-scale repository of code submissions covering over 50 programming languages, collected from AIZU Online Judge[2] and AtCoder[3]. Each code sample is accompanied by detailed metadata, including the problem ID, language, and execution status. To align with the languages in XLCoST while adding further diversity, we select a subset of 10 widely used languages from CodeNet: Java, Python, PHP, JavaScript, C#, C, C++, Ruby, Go, and PyPy. Based on problem IDs, we construct clone and non-clone pairs by treating submissions to the same problem as positive candidates and those to different problems as negative ones, again following [100]. This yields a total of 4800 code pairs, from which 2000 pairs are randomly sampled as the validation set (containing 990 positive and 1010 negative pairs, respectively), and the remaining 2800 pairs are used as the test set (with 1410 positive and 1390 negative pairs, respectively).

5.4.4 Evaluation Metrics

To quantitatively evaluate the performance of LLMs on the cross-lingual code clone detection task, we adopt four widely used classification metrics: Accuracy, Precision, Recall, and F1-score. These metrics allow us to assess both overall correctness and the model's ability to distinguish between true and false clone pairs.

5.4.5 Implementation Details

All experiments are implemented using the PyTorch [107] deep learning framework, in combination with the Hugging Face Transformers library for model loading and inference. To streamline model training and logging, we also leverage components from the LAVIS [69] codebase, which provides a flexible and modular training

[1]https://www.geeksforgeeks.org/
[2]https://onlinejudge.u-aizu.ac.jp/
[3]https://atcoder.jp/

framework for Large Language Models. For training the LLMs on the code clone detection task, we adopt the LoRA (Low-Rank Adaptation) [43] technique to enable parameter-efficient fine-tuning. LoRA allows us to update a small subset of parameters, significantly reducing memory and computation costs while maintaining competitive performance. We apply LoRA adapters to the layers of each model and train only the adapter weights during fine-tuning. The training is performed with the linear warmup and the cosine decay learning rate schedule. The initial learning rate is set to 1e-5, with a warmup phase of 1000 steps, during which the learning rate linearly increases from 1e-8. After warmup, the learning rate decays according to a cosine schedule. We apply a weight decay of 5e-2 to regularize the model and prevent overfitting. All experiments are conducted with mixed-precision training enabled to accelerate computation. During training, models are evaluated periodically on the validation set, and the checkpoint with the highest Accuracy is selected for final evaluation on the test set.

5.5 Experimental Results

We now present the experimental findings corresponding to the three research questions introduced in Section 5.4.1. First, to address RQ1, we evaluate the zero-shot performance of various Large Language Models (LLMs) on the cross-lingual code clone detection task. This establishes a baseline for understanding how well pre-trained LLMs generalize without task-specific fine-tuning. Next, in response to RQ2, we examine the improvements brought by fine-tuning these models on multilingual code clone data. Finally, to answer RQ3, we assess the effectiveness of our proposed summarization-based augmentation method and analyze how it enhances the performance of fine-tuned LLMs. For all evaluations, we report Accuracy, Precision, Recall, and F1-score as defined in Section 5.4.4.

✍ **RQ-1** ► *How effective are LLMs' zero-shot capabilities in cross-lingual code clone detection?* ◄

5.5.1 Experimental Setup for RQ-1

To evaluate the zero-shot capabilities of LLMs for cross-lingual Type-4 code clone detection, we directly apply the pre-trained models to the testing set without any task-specific fine-tuning. For each model, we design the Clone Detection Prompt that provides a concise and consistent instruction, formulated in natural language, to guide the model in comparing a pair of code snippets. The input to the model consists of the fixed prompt followed by the two code snippets to be compared. No training or gradient updates are performed during this stage. The models are expected to infer the functional similarity between the code snippets purely based on their pre-trained knowledge of programming languages and semantics. A clone is predicted if the model's response explicitly indicates functional equivalence or high similarity between the code snippets. This setup allows us to assess how well each model generalizes to the task out-of-the-box and serves as a baseline for evaluating the benefits of subsequent fine-tuning and summarization-based augmentation. All predictions are converted into binary labels using rule-based keyword matching to enable metric-based evaluation.

Table 5.1: Zero-shot performance of different LLMs on the cross-lingual code clone detection task. The bold values denote the best results within each column.

Models	Accuracy (%)	Precision (%)	Recall (%)	F1-score
Llama-2-Chat-7B [140]	50.14	**99.57**	50.25	66.79
Llama-3.1-8B-Instruct [1]	**80.71**	65.11	**95.03**	**77.27**
StarCoder2-7B [86]	50.18	99.15	50.27	66.71
Qwen2.5-Coder-7B-Instruct [49]	53.82	99.43	52.18	68.44

Table 5.2: Performance comparison of zero-shot and fine-tuned (without description augmentation) LLMs on the cross-lingual code clone detection task. The bold values denote the best results within each column of each model.

Models		Accuracy (%)	Precision (%)	Recall (%)	F1-score
Llama-2-Chat-7B [140]	Zero-shot	50.14	**99.57**	50.25	**66.79**
	Fine-tuning w/o description	**57.25**	73.83	**55.70**	63.49
Llama-3.1-8B-Instruct [1]	Zero-shot	80.71	65.11	**95.03**	77.27
	Fine-tuning w/o description	**85.46**	**89.57**	82.93	**86.12**
Starcoder2-7B [86]	Zero-shot	50.18	**99.15**	50.27	66.71
	Fine-tuning w/o description	**80.11**	81.70	**79.39**	**80.53**
Qwen2.5-Coder-7B-Instruct [49]	Zero-shot	53.82	**99.43**	52.18	68.44
	Fine-tuning w/o description	**90.46**	84.33	**96.28**	**89.91**

5.5.2 Experimental Results for RQ-1

Table 5.1 summarizes the zero-shot performance of the four selected Large Language Models on the cross-lingual Type-4 code clone detection task.

Among the evaluated models, Llama-3.1-8B-Instruct achieves the best overall performance, with the highest accuracy (80.71%) and F1-score (77.27%). Its relatively balanced precision (65.11%) and high recall (95.03%) indicate that it is effective at identifying true clone pairs while maintaining moderate conservativeness in its predictions. This suggests that Llama-3.1-8B-Instruct has a strong ability to understand functional equivalence across programming languages, even in the absence of task-specific adaptation. In contrast, Llama-2-Chat-7B, StarCoder2-7B, and Qwen2.5-Coder-7B-Instruct exhibit much higher precision (all around 99%) but considerably lower recall (approximately 50%), resulting in lower overall F1-scores. This pattern implies that these models are highly conservative in classifying code snippets as clones, likely predicting something like "not equivalent" unless the semantic similarity is extremely explicit. As a result, while their false positive rates are low, they miss many true positives—an undesirable property in clone detection tasks that prioritize completeness. Interestingly, StarCoder2-7B, despite being pre-trained extensively on multilingual code, does not outperform general-purpose models like Llama-3.1-8B-Instruct in the zero-shot setting. This may be attributed to its lack of instruction tuning, which limits its ability to interpret and act on task-specific prompts. Similarly, Qwen2.5-Coder-7B-Instruct shows moderate performance, outperforming Llama-2-Chat-7B and StarCoder2-7B on F1-score, but still falling behind Llama-3.1-8B-Instruct in all metrics except precision.

Overall, these results highlight the importance of high-quality instruction tuning and broad language modeling capabilities for achieving strong zero-shot performance on cross-lingual, high-level reasoning tasks like Type-4 code clone detection. Fine-tuning may further improve the performance of all models by adapting them to the specific characteristics of the clone detection task, which we explore in the next subsection.

✍ **RQ-2** ▶ *To what extent does fine-tuning the LLM on multilingual code data enhance detection performance?* ◀

5.5.3 Experimental Setup for RQ-2

To examine the impact of fine-tuning on model performance (RQ-2), we adapt each LLM to the code clone detection task using the XLCoST [173] training set. The models are fine-tuned using LoRA (Low-Rank Adaptation) [43], which updates only a small set of adapter parameters while keeping the majority of the model weights frozen. This strategy enables efficient training while preserving the core knowledge embedded in the pre-trained models.

During training, unlike the fixed prompt used in the zero-shot setting, the Clone Detection Prompt is randomly sampled from a predefined set of prompt templates. This is consistent with the training setup used in our proposed framework (see Section 5.3 for details), and introduces variation in prompt phrasing to improve model robustness and generalization. The sampled prompt is then concatenated with the raw pair of code snippets to form the model input. No functional descriptions or summarization components are included at this stage. The model is trained to generate a binary textual prediction (“yes" or “no") for each code pair, indicating whether the two code snippets are semantically equivalent.

All models are fine-tuned under the same hyperparameter configuration as described in Section 5.4.5, and early stopping is applied based on validation Accuracy. The objective is to evaluate how much performance can be improved through task-specific data exposure alone, without changing the input format or augmenting the model with auxiliary information.

5.5.4 Experimental Results for RQ-2

The results of the experiments for RQ-2 are shown in Table 5.2, alongside each model’s corresponding zero-shot baseline. Across the board, fine-tuning significantly improves model performance, particularly in terms of F1-score, indicating better balance between precision and recall after task adaptation. This demonstrates that even powerful pre-trained LLMs benefit from exposure to task-specific cross-lingual code semantics.

The most substantial improvement is observed in StarCoder2-7B [86], whose F1-score rises dramatically from 66.71 to 80.53 after fine-tuning. Similarly, its accuracy improves from 50.18% (close to random guessing) to 80.11%, indicating that fine-tuning enables the model to overcome its conservative zero-shot behavior and better generalize over the positive/negative decision boundary. Qwen2.5-Coder-7B-Instruct [49] also shows strong gains, achieving the highest overall F1-score (89.91) and accuracy (90.46%) among all models post fine-tuning. This demonstrates that its domain-specific pretraining and instruction-tuning make it particularly receptive to further adaptation on task-aligned data. Additionally, recall improves from 52.18% to 96.28%, suggesting that fine-tuning significantly enhances the model’s ability to identify clone pairs without sacrificing precision. Llama-3.1-8B-Instruct [1], which is already the best-performing model in the zero-shot setting, also benefits from fine-tuning, with F1-score rising from 77.27 to 86.12. While its precision increases significantly (from 65.11% to 89.57%), recall slightly drops (from 95.03%

Table 5.3: Performance comparison of LLMs across zero-shot, fine-tuning without description, and our fine-tuning with description on the cross-lingual code clone detection task. The bold values denote the best results within each column of each model.

Models		Accuracy	Precision (%)	Recall (%)	F1-score
Llama-2-Chat-7B [140]	Zero-shot	50.14	**99.57**	50.25	66.79
	Fine-tuning w/o description	57.25	73.83	55.70	63.49
	Fine-tuning w/ description	**80.36**	75.18	**84.13**	**79.40**
Llama-3.1-8B-Instruct [1]	Zero-shot	80.71	65.11	**95.03**	77.27
	Fine-tuning w/o description	85.46	89.57	82.93	86.12
	Fine-tuning w/ description	**89.29**	**90.07**	88.81	**89.44**
Starcoder2-7B [86]	Zero-shot	50.18	**99.15**	50.27	66.71
	Fine-tuning w/o description	80.11	81.70	79.39	80.53
	Fine-tuning w/ description	**88.07**	86.52	**89.44**	**87.96**
Qwen2.5-Coder-7B-Instruct [49]	Zero-shot	53.82	**99.43**	52.18	68.44
	Fine-tuning w/o description	90.46	84.33	96.28	89.91
	Fine-tuning w/ description	**92.96**	88.44	**97.35**	**92.68**

to 82.93%), indicating a trade-off where the model becomes more confident but slightly more conservative. Interestingly, Llama-2-Chat-7B [140] shows only modest improvement after fine-tuning, with F1-score even slightly decreasing (from 66.79 to 63.49), despite gains in accuracy and recall. This suggests that the model's architecture or pretraining objectives may not be optimal for adaptation to clone detection tasks, or that LoRA [43] adaptation alone is insufficient for performance breakthrough in weaker base models. Overall, these results clearly demonstrate that fine-tuning on cross-lingual code clone data improves detection capability across diverse model architectures. The extent of improvement, however, varies with model size, specialization, and pretraining quality, highlighting the value of combining instruction tuning, domain alignment, and architectural capacity in building high-performing LLM-based clone detectors.

✍ **RQ-3** ▶ *How does integrating our proposed summarization component further improve the effectiveness of fine-tuned LLMs in detecting cross-lingual Type-4 code clones?* ◀

5.5.5 Experimental Setup for RQ-3

To evaluate the effectiveness of our proposed summarization-based augmentation strategy (RQ-3), we extend the input to the LLM by incorporating natural language descriptions of the code snippets alongside the Clone Detection Prompt and original code pairs. These descriptions are generated by a dedicated Summarizer, which is prompted using a Summarization Prompt generated by the Prompt Generation Module. This module translates each code snippet into a concise natural language summary that captures its functionality, structure, or intent. As in our overall framework (see Figure 5.1), the final input to the LLM consists of: (1) the Clone Detection Prompt, (2) the original code snippets, and (3) two generated code summaries.

Similar to the experimental setup for RQ-2, all models are fine-tuned under the same hyperparameter configuration as described in Section 5.4.5, and the model is trained to generate a binary textual prediction (“yes" or “no") for each code pair, indicating whether the two code snippets are semantically equivalent. This setting allows us to directly compare performance with and without summarization augmentation, isolating the contribution of the proposed component in our overall architecture.

5.5.6 Experimental Results for RQ-3

To answer RQ-3, we evaluate the performance of LLMs fine-tuned using our proposed framework, which augments the input with natural language code descriptions generated by a Summarizer. Table 5.3 presents the full results for each model in three conditions: zero-shot (no tuning), fine-tuning without description, and fine-tuning with description. The results clearly show that incorporating high-level natural language summaries of code snippets significantly improves model performance across all models. The F1-score, in particular, benefits significantly, reflecting better balance between precision and recall when descriptions are included.

For instance, Qwen2.5-Coder-7B-Instruct [49] achieves the highest F1-score of 92.68 and accuracy of 92.96% after description-based fine-tuning, which outperforms its non-augmented counterpart (with the F1-score of 89.91) and its zero-shot baseline (with the F1-score of 68.44) by large margins. The improvement in Recall (from 96.28% to 97.3%) suggests that the model becomes even more confident in identifying functionally similar code snippets across different languages when given semantic context in natural language. Similarly, StarCoder2-7B [86], which is originally underperforming in the zero-shot setting, reaches 87.96 for F1-score and 88.07% for accuracy respectively with description augmentation—an increase of over 7 points for F1-score compared to the non-augmented fine-tuned version. This demonstrates the utility of external semantic cues, especially for code-specialized models that may not have strong instruction-following abilities by default. For the Llama-3.1-8B-Instruct [1], which is already the best performer in the fine-tuning without description setting, shows a further improvement from 86.12 to 89.44 for F1-score, and from 85.46% to 89.29% for accuracy. Notably, both its Precision and Recall increase with description guidance, indicating a more robust and balanced detection behavior compared to the non-augmented fine-tuned version. For the Llama-2-Chat-7B [140], which had limited gains from fine-tuning alone, benefits meaningfully from our summarization component: its F1-score increases from 63.49 to 79.40, Accuracy improves significantly from 57.25% to 80.36%, and Precision and Recall increase from 73.83% and 55.70% to 75.18% and 84.13%, respectively. This highlights the generalizability of our approach, even for weaker or less instruction-optimized models.

5.5.7 Motivation Example

Figure 5.2 presents a real-world case illustrating the challenge of multilingual code clone detection. The C++ and Java implementations both compute the length of the overlap between two integer intervals. Despite their functional equivalence, they differ significantly in syntax, structure, and language-specific constructs.

The C++ code, shown in the left panel, uses a direct mathematical expression, while the Java code, split across the middle and right panels, employs additional classes, I/O utilities, and nested control structures. Without description guidance, Llama-3.1-8B-Instruct fails to recognize these as functional clones, misled by their surface-level differences.

However, when natural language descriptions are provided (shown below each code snippet), the model correctly identifies their semantic alignment. This demonstrates the effectiveness of using external descriptions to bridge structural and syntactic

```
// C++ Code
#include <iostream>
using namespace std;
int main() {
    int a, b, c, d;
    cin >> a >> b >> c >> d;
    if (min(b, d) - max(a, c) > 0)
        cout << min(b, d) - max(a, c) << endl;
    else
        cout << 0 << endl;
    return 0;


//C++ Code Description
"This code is a C++ program that solves a specific problem involving the calculation of the difference between two values. It uses various preprocessor directives and macros to optimize performance and simplify coding. The main function takes four integer inputs, calculates the minimum value of two pairs and subtracts them, and outputs the result."


// Java Code
import java.io.*;
import java.util.*;
public class Main {
    public static void main(String[] args) {
```

```
        InputStream inStream = System.in;
        OutputStream outStream = System.out;
        FastReader in = new FastReader(inStream);
        PrintWriter out = new PrintWriter(outStream);
        ABC70 solver = new ABC70();
        solver.solve(1, in, out);

        out.close();
    }
}
static class ABC70 {
    public void solve(int testNum, FastReader in, PrintWriter out) {
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();
        int d = in.nextInt();
        if (c < a) { c ^= a; a ^= c; c ^= a; }

        if (b < c) { out.println("0"); return; }
        out.println(Math.min(d, b) - c);
    }
}
static class FastReader {
    BufferedReader br;
    StringTokenizer st;
    public FastReader(InputStream i) {
```

```
        br = new BufferedReader(new InputStreamReader(i));
    }
    public String next() {
        while (st == null || !st.hasMoreElements()) {
            try { st = new StringTokenizer(br.readLine()); }
            catch (Exception e) { e.printStackTrace(); }
        }
        return st.nextToken();
    }
    public int nextInt() {
        return Integer.parseInt(next()); }
}
}

// Java Code Description:
"This Java code is a solution to an ABC70 problem, where it reads input from the console and prints output to the console. The `ABC70` class contains a method `solve` that takes in several integer values and outputs the difference between two integers based on certain conditions."
```

Figure 5.2: Motivation Example: C++ and Java implementations displayed in a three-panel layout.

gaps, improving cross-language clone detection performance.

5.6 Discussion

In this section, we discuss potential factors that may affect the interpretation and generalization of our findings. We highlight both internal and external threats to validity and acknowledge limitations of our current framework that may guide future research directions.

5.6.1 Threats to Validity

Internal validity. One threat to internal validity lies in the generation of negative samples. While positive code clone pairs are directly provided or inferred from dataset annotations, negative pairs are generated via random sampling. Although this follows prior works [101, 100], it may introduce label noise, especially if semantically similar but non-clone pairs are mistakenly labeled as negative. We mitigate this risk by ensuring that sampled pairs belong to different problem IDs where possible, but there is still a chance of subtle semantic overlap.

External Validity. While LoRA [43] enables efficient fine-tuning, its effectiveness can vary across different LLM architectures and parameter sizes. We use a uniform adaptation strategy for fairness, but some models may benefit from architecture-specific tuning. Thus, our conclusions may not fully generalize to other tuning methods or model variants. Moreover, our method assumes that the summarizer can reliably generate informative, semantically faithful descriptions. If the generated summaries are inaccurate or overly generic, they could mislead the model rather than aid understanding. Since we use a fixed, pre-trained summarizer, our results are sensitive to its coverage and quality.

5.6.2 Limitations

While our proposed approach shows strong empirical performance, several limitations remain:

- **Scalability and Compute Cost**: Although LoRA [43] reduces memory and training overhead, fine-tuning large LLMs with augmented inputs still requires significant computational resources. This may pose challenges for deployment in low-resource or real-time settings.
- **Lack of structural modeling**: Our framework treats code as plain text without leveraging explicit program structures such as abstract syntax trees (ASTs) or control/data flow graphs. Future work could explore incorporating such structure-aware representations to improve reasoning.
- **Limited Programming Language Coverage**: Although our experiments span ten programming languages, they represent a subset of those available in CodeNet and real-world repositories. Performance may differ on niche, low-resource, or syntactically distant languages, potentially limiting generalization beyond the selected set.

5.7 Conclusion

This paper investigates cross-lingual Type-4 code clone detection with Large Language Models. We evaluate models in zero-shot and fine-tuned settings and propose a summarization-based input augmentation strategy. Our results show that fine-tuning significantly improves performance, and that adding code descriptions further enhances semantic understanding across languages. The proposed method achieves consistent gains across models and languages, demonstrating strong generalizability. Future work can explore joint training of the summarizer, integrate structural code information, and extend evaluation to more diverse programming settings.

Despite the strong results, we acknowledge limitations related to scalability and compute cost, structural modeling, and language coverage. Future work may explore joint training of summarization and detection components, integrate code structure representations, and expand evaluation to additional programming languages and real-world software repositories.

6 Conclusion and Future Work

In this chapter, we first revisit the main contributions of this dissertation, and then we present and discuss potential future research directions

Contents

6.1 Conclusion

In this dissertation, we explored novel approaches to learning effective code representations to support essential software engineering tasks across multilingual and practical settings. We focused on three core problems: multilingual code representation learning, automated code editing, and cross-lingual code clone detection.
First, we introduced MetaTPTrans, a meta-learning framework designed to address the challenges of training on multilingual code. Unlike prior models that naively combine code from different languages and often overlook language-specific structures, MetaTPTrans conditions the encoder parameters on programming language identity. This allows the model to better balance language-agnostic semantics with language-specific syntax, leading to improved performance in code understanding tasks such as code summarization and code completion across diverse languages.
Second, we proposed jLED, a unified framework for code editing that jointly learns to localize and edit buggy code segments. While existing approaches often assume the location line of the edit are provided during inference, jLED removes this unrealistic assumption and addresses both localization and editing within a single model. Our method closes the gap between research and real-world software maintenance scenarios, improving robustness and usability in automated code editing and repair tasks.
Third, we explored the use of large language models (LLMs) for cross-lingual code clone detection, a task that requires deep semantic understanding across programming languages. While prior work has attempted zero-shot evaluation using LLMs, their performance remains limited due to the complexity of the task. In this dissertation, we instead focus on fine-tuning LLMs with a summarization-based input augmentation strategy, where language-agnostic natural language descriptions of code are incorporated alongside raw code during training. This augmentation enhances the model's ability to generalize across languages and capture semantic equivalence more effectively. Our method yields consistent improvements in accuracy and robustness for cross-lingual clone detection across diverse LLM architectures.
In summary, this dissertation provides insights and practical solutions for advancing multilingual code understanding and automated reasoning over software source code. Our findings highlight the importance of incorporating language-specific signals, eliminating unrealistic assumptions in model design, and leveraging natural language to enhance model generalization. These contributions pave the way for more robust and general-purpose learning systems for code.

6.2 Future Work

The potential future research directions that are in line with this dissertation include the following:

- **Exploring Prompt-Based and Adapter-Based Approaches for Multilingual Code Modeling.** While MetaTPTrans demonstrates the effectiveness of dynamic parameter generation through meta learning for multilingual code understanding, an alternative line of research lies in leveraging pre-trained models via lightweight adaptation techniques. Future work can explore prompt-based learning, where language-specific prompts guide a frozen or lightly tuned backbone model to adapt to different programming languages. Similarly, adapter modules that inject language-awareness into intermediate layers of pre-trained encoders may offer a more parameter-efficient and scalable alternative to full fine-tuning. Comparing these approaches with meta learning frameworks like MetaTPTrans could provide valuable insights into trade-offs between adaptability, efficiency, and generalization in multilingual code tasks.

- **Code Editing with Multi-turn Interaction.** The current work in Chapter 4 focuses on one-shot code editing, where the model predicts both the edit location and the edit content in a single step. However, in real-world development scenarios, editing is often an iterative process involving human feedback or intermediate corrections. Future research can explore multi-turn interaction frameworks that allow models to refine their edits over multiple steps, possibly guided by intermediate validation signals or user preferences. This may involve integrating reinforcement learning or dialog-based fine-tuning strategies to enhance the controllability and reliability of generated edits.

- **Integrating Code Summaries as Auxiliary Supervision.** In Chapter 5, we demonstrated that incorporating code summaries alongside source code can improve cross-lingual clone detection. A promising direction for future work is to investigate how to better integrate these summaries as auxiliary supervision during model training. Rather than treating summaries as simple concatenated input, future research can explore multi-task learning or contrastive alignment between code and summary embeddings. This may lead to more robust and semantically aligned representations, especially in multilingual or low-resource settings. Furthermore, leveraging external summarization signals, such as documentation or user comments, could enhance generalizability across unseen languages or tasks.

- **Benchmarking and Dataset Expansion.** While our experiments cover a range of datasets and tasks, broader benchmarking is needed to validate model generalizability. In future work, we plan to expand our datasets, especially for underrepresented languages and edit types. Establishing unified benchmarks for multilingual code representation, code editing, and clone detection, along with evaluation protocols for fine-tuning and zero-shot settings, would benefit the community and help standardize future comparisons.

- **Transparency and Safety in LLM-based Code Tasks.** As LLMs continue to evolve, their application in code-related tasks raises concerns about hallucination, unsafe code generation, and unintended memorization. Future research could investigate safety-aware training or decoding strategies, along with tools to interpret LLM behavior in clone detection or editing. Such work will be crucial in ensuring that these models can be reliably deployed in software development pipelines.

List of Papers

Papers included in this dissertation:

- Pian, W., Peng, H., Tang, X., Sun, T., Tian, H., Habib, A., Klein, J. and Bissyandé, T.F., 2023, June. MetaTPTrans: A meta learning approach for multilingual code representation learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 37, No. 4, pp. 5239-5247)

- Pian, W., Li, Y., Tian, H., Sun, T., Song, Y., Tang, X., Habib, A., Klein, J. and Bissyandé, T.F., 2025. You Don't Have to Say Where to Edit! jLED–Joint Learning to Localize and Edit Source Code. ACM Transactions on Software Engineering and Methodology.

- Pian, W., Sun, T., Tian, H., Klein, J. and Bissyandé, T.F.. Enhancing LLMs for Cross-lingual Code Clone Detection with Code Summarization. Under review.

Papers not included in this dissertation

- Tian, H., Li, Y., Pian, W., Kabore, A.K., Liu, K., Habib, A., Klein, J. and Bissyandé, T.F., 2022. Predicting patch correctness based on the similarity of failing test cases. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(4), pp.1-30.

- Li, Y., Dang, X., Pian, W., Habib, A., Klein, J. and Bissyandé, T.F., 2024. Test input prioritization for graph neural networks. IEEE Transactions on Software Engineering, 50(6), pp.1396-1424.

- Tang, X., Tian, H., Chen, Z., Pian, W., Ezzini, S., Kaboré, A.K., Habib, A., Klein, J. and Bissyandé, T.F., 2024, April. Learning to represent patches. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (pp. 396-397).

- Sun, T., Pian, W., Daoudi, N., Allix, K., F. Bissyandé, T. and Klein, J., 2024, June. Laficmil: Rethinking large file classification from the perspective of correlated multiple instance learning. In International Conference on Applications of Natural Language to Information Systems (pp. 62-77). Cham: Springer Nature Switzerland.

- Yang, B., Tian, H., Pian, W., Yu, H., Wang, H., Klein, J., Bissyandé, T.F. and

Jin, S., 2024, September. Cref: An llm-based conversational software repair framework for programming tutors. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 882-894).

- Sun, T., Daoudi, N., Pian, W., Kim, K., Allix, K., Bissyande, T.F. and Klein, J., 2025. Temporal-incremental learning for Android malware detection. ACM Transactions on Software Engineering and Methodology, 34(4), pp.1-30.

Bibliography

[1] Introducing llama 3.1: Our most capable models to date. `https://ai.meta.com/blog/meta-llama-3-1/`, 2024.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei 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*, pages 2655–2668, 2021.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[4] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. Automatic semantic augmentation of language model prompts (for code summarization). in 2024 ieee/acm 46th international conference on software engineering (icse). *IEEE Computer Society*, pages 1004–1004, 2024.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48, pages 2091–2100, 2016.

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[9] Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 3981–3989, 2016.

[10] Wissam Antoun, Fady Baly, and Hazem Hajj. Arabert: Transformer-based model for arabic language understanding. *arXiv preprint arXiv:2003.00104*, 2020.

[11] Luca Bertinetto, João F. Henriques, Jack Valmadre, Philip H. S. Torr, and Andrea Vedaldi. Learning feed-forward one-shot learners. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 523–531, 2016.

[12] Nghi Bui, Yue Wang, and Steven CH Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 812–823, 2022.

[13] Rich Caruana. Multitask learning. *Machine learning*, 28:41–75, 1997.

[14] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 18–30, 2022.

[15] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, 2020.

[16] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–455. IEEE, 2021.

[17] Junkun Chen, Xipeng Qiu, Pengfei Liu, and Xuanjing Huang. Meta multi-task learning for sequence modeling. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 5070–5077. AAAI Press, 2018.

[18] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[19] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code, 2016.

[20] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[22] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. Fira: Fine-grained graph-based code change representation for automated commit message generation. 2022.

[23] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

[24] Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, and Shouling Ji. Adaccd: adaptive semantic contrasts discovery based cross lingual adaptation for code clone detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17942–17950, 2024.

[25] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*, 2021.

[26] Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[28] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[29] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 2017.

[30] Johannes Gasteiger, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[31] Xi Ge and Emerson Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1095–1105, 2014.

[32] Yasir Glani, Luo Ping, Syed Asad Shah, and Lin Ke. Ccdive: A deep dive into code clone detection using local sequence alignment. *Tsinghua Science and Technology*, 30(4):1435–1456, 2025.

[33] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[34] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021.

[35] Jiaxian Guo, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Boyang Li, Dacheng Tao, and Steven Hoi. From images to textual prompts: Zero-shot visual question answering with frozen large language models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10867–10877, 2023.

[36] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, 2017.

[37] David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[38] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[39] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[40] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847. IEEE Computer Society, 2012.

[41] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[43] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[44] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou, and Hai Jin. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 89–100, 2023.

[45] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25:2179–2217, 2020.

[46] Yutao Hu, Yilin Fang, Yifan Sun, Yaru Jia, Yueming Wu, Deqing Zou, and Hai Jin. Code2img: Tree-based image transformation for scalable code clone detection. *IEEE Transactions on Software Engineering*, 49(9):4429–4442, 2023.

[47] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. Treecen: Building tree graph for scalable semantic code clone detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[48] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

[49] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[50] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

[51] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[52] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[53] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[54] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[55] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*, 2023.

[56] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[57] Tae-Hwan Jung. Commitbert: Commit message generation using pre-trained programming language model. *arXiv preprint arXiv:2105.14242*, 2021.

[58] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670, 2002.

[59] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli calling international conference on computing education research*, pages 1–12, 2023.

[60] Guolin Ke, Di He, and Tie-Yan Liu. Rethinking positional encoding in language pre-training. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[61] Mohamad Khajezade, Jie JW Wu, Fatemeh Hendijani Fard, Gema Rodríguez-Pérez, and Mohamed Sami Shehata. Investigating the efficacy of large language models for code clone detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 161–165, 2024.

[62] Raffi Khatchadourian. Automated refactoring of legacy java software to enumerated types. *Automated Software Engineering*, 24(4):757–787, 2017.

[63] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 150–162. IEEE, 2021.

[64] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *International Conference on Learning Representations (ICLR)*, 2015.

[65] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017*, 2017.

[66] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018.

[67] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[68] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020.

[69] Dongxu Li, Junnan Li, Hung Le, Guangsen Wang, Silvio Savarese, and Steven C.H. Hoi. LAVIS: A one-stop library for language-vision intelligence. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 31–41, 2023.

[70] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4159–25, 2018.

[71] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[72] Shuang Li, Kaixiong Gong, Chi Harold Liu, Yulin Wang, Feng Qiao, and Xinjing Cheng. Metasaug: Meta semantic augmentation for long-tailed visual recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 5212–5221, 2021.

[73] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[74] ZiKe Li, MingWei Liu, Anji Li, Kaifeng He, Yanlin Wang, Xin Peng, and Zibin Zheng. Enhancing the robustness of llm-generated code: Empirical study and framework. *arXiv preprint arXiv:2503.20197*, 2025.

[75] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1509–1521, 2023.

[76] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 37–47, 2020.

[77] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A unified multi-task learning model for ast-level and token-level code completion. *Empirical Software Engineering*, 27(4):91, 2022.

[78] Fang Liu, Ge Li, Yunfei Zhao, and Zhi 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, pages 473–485, 2020.

[79] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 345–357. IEEE, 2023.

[80] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[81] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019.

[82] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4487–4496, 2019.

[83] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[84] Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. Ccrep: Learning code change representations via pre-trained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 17–29. IEEE, 2023.

[85] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

[86] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[87] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[88] Wenqiang Luo, Jacky Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawendé F. Bissyandé, Haoye Tian, and Xuan Bach D. Le. When fine-tuning llms meets data privacy: An empirical study of federated learning in llm-based program repair. *ACM Trans. Softw. Eng. Methodol.*, May 2025. Just Accepted.

[89] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Tegawende F Bissyande, Haoye Tian, and Bach Le. Unlocking llm repair capabilities in low-resource programming languages through cross-language translation and multi-agent refinement. *arXiv preprint arXiv:2503.22512*, 2025.

[90] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[91] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

[92] Jorge Martinez-Gil. Source code clone detection using unsupervised similarity measures. In *International Conference on Software Quality*, pages 21–37. Springer, 2024.

[93] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE, 2021.

[94] Nikita Mehrotra, Akash Sharma, Anmol Jindal, and Rahul Purandare. Improving cross-language code clone detection via code representation learning and graph neural networks. *IEEE Transactions on Software Engineering*, 49(11):4846–4868, 2023.

[95] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. Does automated refactoring obviate systematic editing? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 392–402. IEEE, 2015.

[96] Na Meng, Miryung Kim, and Kathryn S McKinley. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 440–443, 2011.

[97] Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[98] Na Meng, Miryung Kim, and Kathryn S McKinley. Lase: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2013.

[99] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press, 2016.

[100] Micheline Bénédicte Moumoula, Abdoul Kader Kabore, Jacques Klein, and Tegawendé Bissyande. Large language models for cross-language code clone detection. *arXiv preprint arXiv:2408.04430*, 2024.

[101] Micheline Benedicte Moumoula, Abdoul Kader Kabore, Jacques Klein, and Tegawende F Bissyande. Cross-lingual code clone detection: When llms fail short against embedding-based classifier. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2474–2475, 2024.

[102] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[103] Mona Nashaat, Reem Amin, Ahmad Hosny Eid, and Rabab F Abdel-Kader. An enhanced transformer-based framework for interpretable code clone detection. *Journal of Systems and Software*, 222:112347, 2025.

[104] Xuefei Ning, Zifu Wang, Shiyao Li, Zinan Lin, Peiran Yao, Tianyu Fu, Matthew Blaschko, Guohao Dai, Huazhong Yang, and Yu Wang. Can llms learn by teaching for better reasoning? a preliminary study. *Advances in Neural Information Processing Systems*, 37:71188–71239, 2024.

[105] Zheyi Pan, Yuxuan Liang, Weifeng Wang, Yong Yu, Yu Zheng, and Junbo Zhang. Urban traffic prediction from spatio-temporal data using deep meta learning. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 1720–1730. ACM, 2019.

[106] Zheyi Pan, Wentao Zhang, Yuxuan Liang, Weinan Zhang, Yong Yu, Junbo Zhang, and Yu Zheng. Spatio-temporal meta learning for urban traffic prediction. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1462–1476, 2022.

[107] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[108] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems*, 34, 2021.

[109] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Complete and accurate clone detection in graph-based

models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 276–286. IEEE, 2009.

[110] Weiguo Pian, Shijian Deng, Shentong Mo, Yunhui Guo, and Yapeng Tian. Modality-inconsistent continual learning of multimodal large language models. *arXiv preprint arXiv:2412.13050*, 2024.

[111] Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. You don't have to say where to edit! jled–joint learning to localize and edit source code. *ACM Transactions on Software Engineering and Methodology*, 2025.

[112] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Metatptrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5239–5247, 2023.

[113] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[114] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[115] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[116] Sarah Rastkar and Gail C Murphy. Why did this code change? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1193–1196. IEEE, 2013.

[117] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The uniqueness of changes: Characteristics and applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 34–44. IEEE, 2015.

[118] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428, 2014.

[119] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[120] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[121] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[122] Junjie Shan, Shihan Dou, Yueming Wu, Hairu Wu, and Yang Liu. Gitor: Scalable code clone detection by building global sample graph. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 784–795, 2023.

[123] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468, 2018.

[124] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Race: Retrieval-augmented commit message generation. *arXiv preprint arXiv:2203.02700*, 2022.

[125] G Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655, 2021.

[126] André Silva, Sen Fang, and Martin Monperrus. Repairllama: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698*, 2023.

[127] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*, 2023.

[128] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024.

[129] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[130] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[131] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2727–2735, 2019.

[132] Xunzhu Tang, Haoye Tian, Zhenghan Chen, Weiguo Pian, Saad Ezzini, Abdoul Kader Kaboré, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Learning to represent patches. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 396–397, 2024.

[133] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM Trans. Softw. Eng. Methodol.*, 31(4):77:1–77:30, 2022.

[134] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 981–992, 2020.

[135] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches, 2022.

[136] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Transactions on Software Engineering and Methodology*, 2023.

[137] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

[138] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and TegawendÉ F BissyandÉ. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[139] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401, 2022.

[140] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[141] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.

[142] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[143] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2291–2302, 2022.

[144] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[145] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems, NeurIPS 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[146] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[147] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2692–2700, 2015.

[148] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 708–719. ACM, 2016.

[149] Xin Wang, Fisher Yu, Ruth Wang, Trevor Darrell, and Joseph E. Gonzalez. Tafe-net: Task-aware feature embeddings for low shot learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 1831–1840. IEEE / Computer Vision Foundation, 2019.

[150] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[151] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[152] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[153] Yang Wu, Yao Wan, Zhaoyang Chu, Wenting Zhao, Ye Liu, Hongyu Zhang, Xuanhua Shi, and Philip S Yu. Can large language models serve as evaluators for code summarization? *arXiv preprint arXiv:2412.01333*, 2024.

[154] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[155] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. Detecting semantic code clones by building ast-based markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[156] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. Scdetector: Software functional clone detection based on semantic tokens analysis. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 821–833, 2020.

[157] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3975–3981, 2019.

[158] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[159] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F Bissyandé, and Shunfu Jin. Cref: An llm-based conversational software repair framework for programming tutors. *arXiv preprint arXiv:2406.13972*, 2024.

[160] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawende Bissyande, Claire Le Goues, and Shunfu Jin. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *ACM Trans. Softw. Eng. Methodol.*, May 2025.

[161] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. Multi-objective fine-tuning for enhanced program repair with llms. *arXiv preprint arXiv:2404.12636*, 2024.

[162] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 5754–5764, 2019.

[163] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1506–1518, 2022.

[164] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[165] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24:33–67, 2019.

[166] Yang Yuan and Yao Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, pages 286–289, 2012.

[167] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[168] Yazhou Zhang, Mengyao Wang, Chenyu Ren, Qiuchi Li, Prayag Tiwari, Benyou Wang, and Jing Qin. Pushing the limit of llm capacity for text classification. *arXiv preprint arXiv:2402.07470*, 2024.

[169] Zixian Zhang and Takfarinas Saber. Assessing the code clone detection capability of large language models. In *2024 4th International Conference on Code Quality (ICCQ)*, pages 75–83. IEEE, 2024.

[170] Shufan Zhou, Beijun Shen, and Hao Zhong. Lancer: Your code tell me what you need. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1202–1205. IEEE, 2019.

[171] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. Ccbert: Self-supervised code change representation learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 182–193. IEEE, 2023.

[172] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207, 2019.

[173] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

[174] Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. Multilingual machine translation with large language models: Empirical results and analysis. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2765–2781. Association for Computational Linguistics, June 2024.

[175] Wenqing Zhu, Norihiro Yoshida, Toshihiro Kamiya, Eunjong Choi, and Hiroaki Takada. Development and benchmarking of multilingual code clone detector. *Journal of Systems and Software*, 219:112215, 2025.

[176] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143, 2023.

[177] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 931–942, 2020.

[178] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

A Appendix of MetaTPTrans

Summary of the CodeSearchNet Dataset

We show the summary of the CodeSearchNet [50] dataset used in our experiments in Table A.1, which contains four programming languages: Python, Ruby, JavaScript, and Go. The dataset has been deduplicated by the creators to avoid data leakage from training set.

Table A.1: Dataset statistics

Language	Samples per partition		
	Train	Valid	Test
Python	412,178	23,107	22,176
Ruby	48,791	2,209	2,279
JavaScript	123,889	8,253	6,483
Go	317,832	14,242	14,291
Total	902,690	47,811	45,229

Table A.2: Dataset Statistics

Language	Samples per partition		
	Train	Valid	Test
Python	69,527	11,112	10,565
Ruby	11,524	1,075	1,048
JavaScript	26,626	4,353	3,504
Go	36,948	3,434	4,563
Total	144,175	19,974	19,680

Preprocessing

We parse code snippets using Tree-Sitter[1], an open-source parser that can parse multiple programming languages into AST. Besides, we follow the token splitting rule in [7, 178, 108] that splits each code token into sub-tokens regarding to the code name convention. For instance, the code token `sendDirectOperateCommandSet` is split into `[send, direct, operate, command, set]`. For the vocabulary of sub-tokens, we limit it with the least occurrence number of 100 in the training set, and we also restrict the max length of the token sequence to 512 after removing all punctuation. For code summarization task, we follow [178, 108] and remove all the anonymous functions in the javaScript dataset which can not be used for this task. For the

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

code completion task, we randomly select a subset from the whole CSN dataset with 144,175, 19,974 and 19,680 code snippets for training, validation and testing respectively. Please see the next section for details of the dataset used in the code completion task. For the paths, we set the max length to 32, and we make padding for the path shorter than max length while sampling nodes with equal intervals to maintain max length following [108].

Summary of the CSN Subset in Code Completion

For the code completion task, we randomly select a subset from the whole CSN dataset with 144,175, 19,974 and 19,680 code snippets for training, validation and testing respectively. We show the summary of it in Table A.2.

Baselines

In our experiments, we set the following methods as baselines:

- **Transformer:** Transformer [145] is a language model based on multi-head attention, which can only model contextual information. Transformer is the basic backbone of GREAT, CodeTransformer and TPTrans.
- **code2seq:** Code2seq [7] is an LSTM-based method that utilizes the pairwise path information in AST to model code snippets.
- **GREAT:** GREAT [39] is a Transformer-based approach that utilizes manually designed edges, such as dataflow, 'computed from', 'next lexical use' edges.
- **CodeTransformer:** CodeTransformer [178] is a Transformer-based model that combines multiple pairwise distance relation between nodes in AST to integrate the structure information into the context sequence of code
- **TPTrans:** TPTrans [108] is a recent state-of-the-art approach for code representation learning based on Transformer and tree paths in AST, which integrate the encoding of tree paths into self-attention module by replacing the relative and absolute position embedding.

Table A.3: Number of Parameters of TPTrans and MetaTPTrans

Model	Task	
	Code Summarization	Code Completion
TPTrans	113.47M	101.74M
MetaTPTrans-α	178.48M	109.87M
MetaTPTrans-β	118.65M	102.73M
MetaTPTrans-γ	181.56M	110.59M

Table A.4: Average One Epoch's Training Time Cost of TPTrans and MetaTPTrans

Model	Task	
	Code Summarization	Code Completion
TPTrans	4h15min	32min
MetaTPTrans-α	6h15min	35min
MetaTPTrans-β	4h28min	32min
MetaTPTrans-γ	6h44min	35min

Table A.5: Experimental results of code summarization task w/o pointer network. The bold part denotes the best results, the underlined part denotes the best results of baselines.

Model	Python			Ruby			JavaScript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
CodeTransformer (Multi.)	38.91	33.12	35.78	34.52	27.31	30.50	37.21	29.75	33.07	56.07	50.76	53.28
TPTrans (Multilingual)	38.78	34.72	36.64	38.05	32.35	34.97	36.35	30.06	32.90	56.49	51.99	54.15
MetaTPTrans-α	**39.26**	36.57	**37.87**	**39.22**	34.55	36.74	37.29	32.50	34.73	**57.14**	54.48	55.78
MetaTPTrans-β	38.34	**37.32**	37.82	38.87	**36.07**	**37.42**	37.35	**34.06**	**35.63**	56.56	**55.14**	**55.84**
MetaTPTrans-γ	38.50	36.96	37.71	38.38	33.99	36.05	**37.72**	32.62	34.98	56.49	54.30	55.38

Number of Parameters and Training time Cost

We show the number of parameters of our approaches and TPTrans in Table A.3. In code summarization task, compared with TPTrans, our MetaTPTrans-α, MetaTPTrans-β and MetaTPTrans-γ have 57.29%, 4.57% and 60.01% more parameters respectively. For code completion task, MetaTPTrans-α, MetaTPTrans-β and MetaTPTrans-γ have 7.99%, 0.97% and 8.70% more parameters than TPTrans respectively. In Table A.4, we show the average one epoch's training time cost of TPTrans and our approaches.

Code Summarization Results w/o Pointer Network

In our experiments on code summarization task, we apply a pointer network [147] in the decoder following [178, 108]. Here, we conduct an ablation study in which we remove the pointer network and the experiment results are shown in Table A.5. We can see that, our approaches still have better performance compared with the baselines.