Abstract Syntax Tree for Programming Language Understanding and Representation

WEISONG SUN, Nanyang Technological University, Singapore and China-Singapore International Joint Research Institute, China
CHUNRONG FANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
YUN MIAO, State Key Laboratory for Novel Software Technology, Nanjing University, China
MENGZHE YUAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENPENG CHEN, Tsinghua University, China
YUCHEN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
QUANJUN ZHANG, Nanjing University of Science and Technology, China
AN GUO, State Key Laboratory for Novel Software Technology, Nanjing University, China
XIANG CHEN, School of Artificial Intelligence and Computer Science, Nantong University, China
ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Programming language understanding and representation (a.k.a code representation learning) has always been a hot and challenging task in the field of software engineering. It aims to apply deep learning techniques to produce numerical representations of the source code features while preserving its semantics. These representations can facilitate subsequent code-related tasks, e.g., code summarization. The abstract syntax tree (AST), a fundamental code feature, illustrates the syntactic information of the source code and has been widely used in code representation learning. It is commonly acknowledged that AST-based code representation is critical to solving code-related tasks. However, there is still a lack of systematic and quantitative evaluation of how well AST-based code representation facilitates subsequent code-related tasks. Additionally, learning an AST-based code representation is an extremely complex endeavor involving three intertwining stages, including AST parsing, AST preprocessing, and AST encoding. The solutions available in each stage are diverse. There is currently a lack of guidance on selecting solutions at each stage to get the most out of AST.

*Chunrong Fang is the corresponding author.

Authors' Contact Information: Weisong Sun, weisong.sun@ntu.edu.sg, Nanyang Technological University, Singapore, 50 Nanyang Avenue, Singapore and China-Singapore International Joint Research Institute, Guangzhou, Guandong, China; Chunrong Fang, fangchunrong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Yun Miao, miaoyun001my@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Mengzhe Yuan, shiroha123321@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Zhenpeng Chen, zpchen@tsinghua.edu.cn, Tsinghua University, Beijing, China; Yuchen Chen, yuc.chen@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Quanjun Zhang, quanjunzhang@njust.edu.cn, Nanjing University of Science and Technology, Nanjing, Jiangsu, China; An Guo, guoan218@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Xiang Chen, xchencs@ntu.edu.cn, School of Artificial Intelligence and Computer Science, Nantong University, Nantong, Jiangsu, China; Zhenyu Chen, zychen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/0-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Just Accepted

In this paper, we first conduct comprehensive experiments to reveal the impact of the choice of AST parsing/preprocessing/encoding methods on AST-based code representation across three popular types of code-related tasks, including code clone detection, code search, and code summarization. The experiments involve four AST parsing methods, six AST preprocessing methods, and four AST encoding methods, all of which are widely utilized in existing AST-based code representation research. The experimental results showcase that the impact of different methods at different stages varies for different code-related tasks. Based on these, we further explore the practical influence of the AST-based code representation in facilitating follow-up code-related tasks. To do so, we compare the performance of models trained with code token sequence (Token for short) based code representation and AST-based code representation on the three code-related tasks. Surprisingly, the overall quantitative statistical results demonstrate that models trained with Token-based code representation consistently perform better across all three tasks compared to models trained with AST-based code representation. Our further quantitative analysis reveals that models trained with AST-based code representation outperform models trained with Token-based code representation in certain subsets of samples across all three tasks. For instance, in the code summarization task, such samples constitute as much as 39% of the total, while in the code search task, they account for 28%. As ASTs are now being used in practice under various contexts (a.k.a., code-related tasks), the results in this paper call for more research on context-specific AST-based code representation learning in the future. Our study provides future researchers with detailed guidance on how to select solutions at each stage to fully exploit AST.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: Abstract Syntax Tree, Programming Understanding, Programming Representation, Code Representation

1 Introduction

The ascendancy of deep learning (DL) in software engineering has engendered a growing interest in code representation learning for program understanding and representation. Code representation learning aims to use DL techniques to learn the distributed vector representation of (program) code features, which supersedes the feature engineering and selection step in classic machine learning methods. The distributed vector representation of code features (code representation/embedding, for short) preserves code semantics and facilitates subsequent code intelligence tasks, such as code clone detection [25, 92], neural code search [30, 64, 73], neural code summarization [9, 37, 74], etc.

The procedure of code representation learning can be decomposed into two sequential steps: extracting code features from program source code and encoding code features using DL techniques, where the code embeddings produced by DL techniques aim to capture the semantics of the code features. The choice of code features has a great impact on code representation learning. In existing code representation learning efforts, typical code features include code token sequences (Token), abstract syntax trees (AST), control flow graphs (CFG), data flow graphs (DFG), etc. Researchers utilize different code features depending on the kind of information that needs to be extracted, such as Token for lexical information, AST for syntactic information, and CFG/DFG for semantic information [65]. Syntactic differences are considered to be the key differences between programming languages and natural languages. Therefore, AST, which expresses syntactic information, has received widespread attention and is widely used in code representation learning research [3, 6, 11, 12, 37, 38, 50, 52, 59, 67–69, 78, 82, 90–92, 95, 98, 99, 103].

It is generally believed that Token contains lexical information, while AST contains lexical information and the syntactic structure of source code [99]. Previous research [22, 38, 99] indicates that the source code comprises rich syntactic information and cannot be directly considered as plain text. It is considered crucial to model the code's syntactic knowledge via AST, which is often done in practice. AST has been widely used to model the source code (i.e, learning AST-based code representation) for many follow-up code-related tasks, such as code classification [86, 99], code clone detection [10, 13, 99], code search [30, 31, 34, 36, 83, 95], code summarization [52, 68], etc. Nevertheless, some other studies have demonstrated that Token outperforms AST

Just Accepted

in facilitating DL models to learn code semantics, or the incorporation of AST leads to reduced DL models' performance [1, 41, 67, 82]. Hence, it necessitates conducting a systematic and quantitative evaluation of the extent to which AST-based code representation facilitates code-related tasks.

Furthermore, learning AST-based code representation and using it to solve code-related tasks is extremely challenging. Fig. 1(a)–(c) shows the procedure of using AST for code representation learning to solve code-related tasks. It is observed that given a piece of source code (usually a method/function), it goes through the AST processing pipeline to obtain an AST-based code representation. Then, the AST-based code representation that preserves code semantics is fed into distinct task models to solve different code-related tasks. As shown in Fig. 1(a), the internal design of the AST processing pipeline is complex and varied, which consists of three core and intertwining stages: AST parsing, AST preprocessing, and AST encoding.

During the AST Parsing stage, different AST parsing methods are utilized to parse the source code into ASTs. Since these AST parsing methods adopt different lexical rules and grammar rules, distinct ASTs would be generated for the same source code. As DL models are highly sensitive to input data, users must consider whether the differences in ASTs generated by different parsers might affect the model's learning of AST-based code representation and subsequent code-related tasks. Utkin et al. [80] find out that ASTs generated by different AST parsers vary in size and abstraction level, and such variation affects the models' method name prediction performance. Similar conclusions may also hold in other code-related tasks.

AST Preprocessing stage is responsible for preprocessing the AST to simplify the complexity of the AST [82, 91], or to adapt to the input requirements of different DL models [78]. Existing preprocessing methods can be divided into two categories according to the form of the AST data they output: sequential AST preprocessing methods (e.g., SBT [37] and AST Path [5]) and structural AST preprocessing methods (e.g., Binary Tree [91] and Split AST [68]). The effectiveness of these preprocessing methods has been verified in corresponding papers. However, horizontal comparisons of these methods are still lacking. Besides, there are significant differences in node and structure information among different sequential/structural AST data. The impact of these differences on the model's ability to learn AST-based code representation and its performance on subsequent code-related tasks is still lacking systematic investigation.

AST encoding models transform preprocessed AST data into numerical vector representations (i.e., AST-based code representation), which can be used for solving various code-related tasks. According to the form of the preprocessed AST data, existing AST encoding methods can be categorized into two types: sequence models (e.g., BiLSTM [66] and Transformer [81]) and tree-structured models (e.g., TreeLSTM [77] and AST-Trans [78]). The sequence models and tree-structure models are designed for encoding sequential AST data and structural AST data, respectively. Just as with AST preprocessing methods, there exist notable differences between sequence models and tree-structured models, as well as among different sequence/tree-structure models. Hence, we conduct a systematic evaluation of the impact of different AST encoding models on the produced AST-based code representation and subsequent code-related tasks.

To address the above gaps, in this paper, we conduct a comprehensive empirical study to: (1) reveal the impact of the choice of AST parsing/preprocessing/encoding methods on AST-based code representation and subsequent code-related tasks; (2) explore the effectiveness of the AST-based code representation in improving subsequent code-related tasks. All experiments are performed on three popular types of code-related tasks, including a code-to-code matching task, i.e., code clone detection [92], a text-to-code matching task, i.e., code search [64], and a code-to-text generating task, i.e., code summarization [1]. We quantitatively evaluate the effectiveness of the involved models on commonly used metrics for different tasks. A total of 11 metrics are used in the three tasks. BigCloneBench [76] and CodeSearchNet (Java) [42] are selected as experimental datasets, which are widely used in existing code-related studies.

Our experimental results demonstrate that the choice of AST parsing, preprocessing, and encoding methods has a non-negligible impact on AST-based code representations and subsequent code-related tasks. Specifically,

Just Accepted

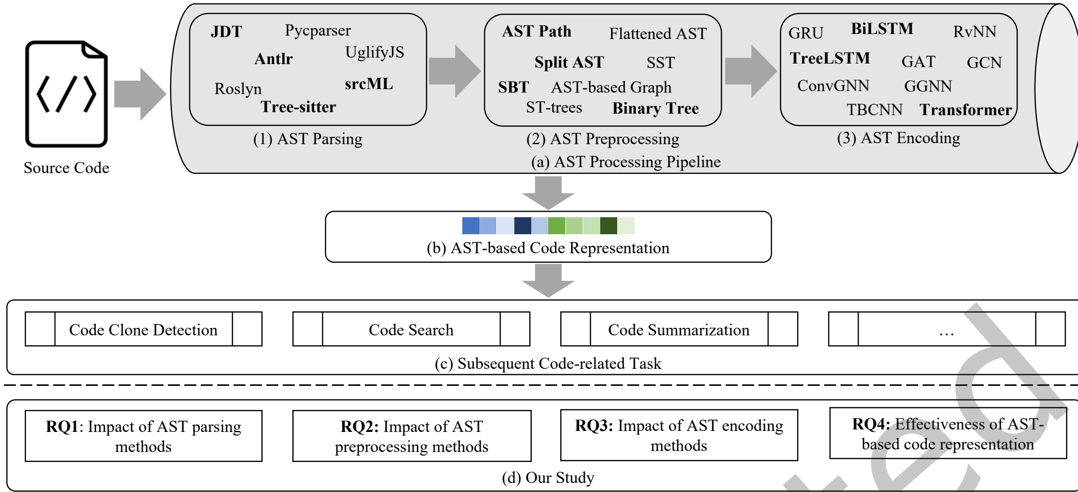


Fig. 1. Overview of our empirical study.

AST parsing methods that generate ASTs with smaller tree sizes, shallower tree depth, and higher abstraction levels yield more favorable outcomes for all three code-related tasks. As for AST preprocessing methods, different code-related tasks have varying requirements for AST node and structure information, leading to differences in performance among different AST preprocessing methods across distinct tasks. Among the AST encoding models we investigate, the Transformer performs best overall. However, there is still room for improvement in designing Transformer-based models suitable for learning AST-based code representation from structural AST data (e.g., Raw AST).

We also verify through experiments that the contribution of AST to the expressiveness of code representation is weaker compared to Token. Nonetheless, it is worth noting that the models trained with ASTs outperform the models trained with Token on certain samples of all three code-related tasks, such as pairs of code snippets that have low token similarity in code clone detection, code snippets that require more high-level abstract summaries in code summarization, and code snippets that semantically match but contain fewer query words in code search. In these samples, the syntactic information contained in the AST plays a vital role in improving the expressiveness of code representation as well as code-related tasks. It inspires us to combine multiple code representation methods, such as Token and AST-based representation, to improve the effectiveness of code representation.

The purpose of this study is to provide a systematic and generalized understanding of AST processing and applications, which could facilitate learning AST-based code representation and solving many code-related tasks. We hope that the conclusions of this paper can provide researchers with guidance on whether they need to choose AST-based code representation and how to use ASTs for code representation. In summary, we make the following contributions in this paper.

- We conduct a comprehensive evaluation of the impact of the choice of AST parsing, preprocessing, and encoding methods on AST-based code representation and subsequent code-related tasks. Our study can provide future researchers with detailed guidance on how to select solutions at each stage to fully exploit AST.
- We conduct a systematic and quantitative evaluation of the effectiveness of AST-based code representations on three popular types of code-related tasks. To better grasp under what circumstances AST-based code representation can facilitate code-related tasks, we conduct a more detailed qualitative analysis of code

Just Accepted

characteristics in samples where models trained using ASTs performed better. The results of quantitative evaluation and qualitative analysis demonstrate that the current application of AST in code-related tasks is still insufficient, and the value of AST is not maximized. As ASTs are now being used in practice under various contexts (many code-related tasks), the results in this paper call for more research on context-specific AST-based code representation learning in the future.
- To facilitate replication and further research in this area, we release the source code, the dataset and results of our experiments to help other researchers replicate and extend our study [75].

The remainder of this paper is organized as follows. Section 2 provides the research background. Section 3 describes our study design. Section 4 presents experimental results and concludes our findings. Section 5 discusses some threats to validity. Section 6 introduces the related work. We conclude our study in Section 7.

2 Background

2.1 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is one of the most commonly used code features in code representation learning [65]. It uniquely represents a source code snippet in a given language and grammar. As the program is highly structured data compared to plain text, many code-related works attempt to extract the structure information behind the source code to capture syntactic information of the source code [71]. Currently, the specific definitions of AST vary slightly across different sources. The representative one is the definition given by Alon et al. [4]. They formalize the AST of a method/function code snippet as follows:

Definition 2.1 (Abstract Syntax Tree). An Abstract Syntax Tree (AST) for a method is a tuple $\langle N, T, X, s, \delta, \phi \rangle$ where N is a set of non-terminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps a non-terminal node to a list of its children, * represents closure operation, and $\phi : T \rightarrow X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.

In the above definition, terminals are considered as leaf nodes of the AST. In fact, many researchers [11, 17, 49, 80, 96] also regard the values of the terminals as the leaf nodes. In this paper, we also follow the latter. We show an example of AST in the supplemental material and our repository [75].

2.2 Code Representation Learning

Code representation learning aims to apply DL techniques to convert source code features (e.g., Token and AST) into distributed, real-valued vector representations (also known as code representations/embeddings). Code representations condense the semantics of code features, and two pieces of code with similar semantics will be located close to one another in the vector space. Code representation enables the application of DL techniques in various code-related tasks.

Code representation learning can be divided into two sequential processes: code feature extraction and code feature representation. Code feature extraction is responsible for extracting code features, such as method name [32, 64, 95], Token [15, 26, 30, 32, 41, 95], API sequences [14, 24, 32, 39], CFG [31, 82, 101], etc. AST that we investigate in this paper is also a fundamental code feature, illustrating the syntactic structure of the source code. More details on other code features are discussed in Section 6.1. Code feature representation is responsible for applying DL techniques to convert source code features into code representations. Typically, the code representation generated for the Token is called the Token-based code representation. In this paper, we focus on code representations derived from ASTs, referred to as AST-based code representations. More details on code representations for other code features are discussed in Section 6.2. To generate high-quality and semantic-preserving AST-based code representation, researchers have successfully proposed many AST

Just Accepted

preprocessing methods and introduced many neural network models from the fields of CV and NLP as AST encoding methods. AST preprocessing methods aim to transform the raw AST generated by AST parsing methods into a data form that is streamlined and easy for AST encoding models to learn. As mentioned in Section 1, common data forms encompass sequential AST data (e.g., SBT [37]) and structural AST data (e.g., Binary Tree [91]). Correspondingly, to process different forms of AST data, existing research has introduced various sequence models (e.g., BiLSTM [66]) and tree-structured models (e.g., TreeLSTM [77]). In this paper, we focus on evaluating the impact of various choices of AST preprocessing and encoding methods, aiming to guide future researchers on how to effectively exploit these methods to enhance AST-based code representation and their code-related tasks.

3 Study Design

3.1 Research Questions

The evaluation presented in this work aims to answer the following research questions:

RQ1: How do AST parsing methods affect the performance of AST-based code representation on subsequent code-related tasks?

This research question systematically compares ASTs generated by different AST parsing methods. Their differences are revealed through multiple quantitative measurements, including tree size, tree depth, branch factor, unique types, and unique tokens, applicable programming languages (detailed in Section 4.1). These differences may have implications for code representation and the models used for subsequent code-related tasks. This research question aims to expose these implications. In this paper, we investigate the four commonly used AST parsing tools, including JDT [45], srcML [21], ANTLR [62], and Tree-sitter [57] (detailed in Section 3.2). We compare the performance differences of these four AST parsing tools and analyze the correlation between tree size/tree depth/branch factor/unique types/unique tokens and evaluation results.

RQ2: How do AST preprocessing methods affect the performance of AST-based code representation on subsequent code-related tasks?

Considering that the original AST generated by the AST parsing method may be too large and complex to be learned by the code representation model, some researchers have proposed some AST preprocessing methods to alleviate this. These preprocessing methods change (or even break) the structure and content of the original AST. These changes may have implications for AST-based code representation and subsequent code-related task models. This research question aims to expose these implications. In this paper, we investigate six commonly used AST preprocessing methods, including Breadth-first Search (BFS), SBT [37], AST Path [5], Binary Tree [41, 82, 91], and Split AST [52] (detailed in Section 3.3).

RQ3: How do AST encoding methods affect the performance of AST-based code representation on subsequent code-related tasks?

Different neural network architectures/models (e.g., LSTM and Transformer) differ in capturing data features. For the data characteristics of different AST preprocessing results (e.g., sequential data or structured data), existing research has also tried a variety of neural network models to capture the features of these AST data. This research question seeks to expose the impact of AST encoding methods on AST-based code representation and subsequent code-related task models. In this paper, we investigated four AST encoding methods, including BiLSTM [66] and Transformer [81] for sequential AST data, TreeLSTM [77] and AST-Trans [78] for structured AST data (detailed in Section 3.4).

RQ4: Does AST improve the expressiveness of code representation and facilitate subsequent code-related tasks?

This research question seeks to figure out whether code representations fused AST perform better on subsequent code-related tasks than unfused ones. Some existing works [22, 38, 99] claim that AST can facilitate code representation, while other works hold the opposite attitude [1, 41, 67, 82]. We conduct a comparative analysis of

Just Accepted

the performance of Transformer trained with four distinct types of inputs, including Token, SBT, SBT without (w/o) Token, and Token + SBT w/o Token, on all three code-related tasks. Besides, we conduct a deeper analysis to explore when to use AST-based code representation.

3.2 AST Parsing Method

AST parsing methods/tools (parsers, for short) are used to convert source code into corresponding abstract syntax trees. There are many AST parsers available for different scenarios. In this paper, we conduct experiments to investigate four commonly used AST parsers as follows.

Eclipse Java Development tools (JDT) [45]: JDT contains a set of plugins to support Java in the Eclipse IDE, including code highlighting, code refactoring, and AST parsing. It has been widely used by researchers in solving various software engineering tasks, such as code comment generation [37], code clone detection [13], and code understanding [85].

srcML [21]: srcML is a lightweight and highly scalable AST parser, which converts source code into an XML representation, where the markup tags identify elements of the AST. Currently, it supports C/C++, C#, and Java. It has been used in multiple works. For example, LeClair et al. [50], Wei et al. [90], and Shahbazi et al. [67] leverage srcML to parse AST in their code summarization techniques.

ANTLR [62]: ANTLR takes the grammar of the target programming language as input and generates a corresponding AST parser. Currently, ANTLR supports many programming languages, including C++, Java, and Python. In this paper, we use the open-source ANTLR Java grammar [1] to generate the Java parser. Shi et al.[68] and Hua et al.[41] employ ANTLR as the AST parser in their code summarizer CAST and clone detector FCCA, respectively.

Tree-sitter [57]: Like ANTLR, Tree-sitter can generate a parser for a target programming language using its grammar. Tree-sitter focuses on real-time usage in text editors or IDEs, and thus, it can parse code incrementally and is robust enough to parse code with syntax errors. In this paper, we use the most popular Tree-sitter Java grammar [2] to generate the Java parser. Gu et al. [30] and Guo et al. [33] adopt Tree-sitter to parse source code into AST in the code search tool and the pre-training task, respectively.

3.3 AST Preprocessing Method

In this section, we introduce several typical AST preprocessing methods that are investigated, including structure-based traversal (SBT) [37], AST path [5], binary tree [41, 82, 91], and split AST [52].

Breadth-first Search (BFS). BFS is a traditional AST traversal method, which starts at the root node and explores all nodes at the present depth before moving on to the nodes at the next depth level. Through BFS traversal, users can get the node sequence of an AST.

Structure-based Traversal (SBT). SBT proposed by Hu et al. [37] is a popular AST traversal method and has been widely used in the field of code representation [50, 90]. SBT converts a tree into a sequence consisting of tree nodes and brackets, from which we can restore the original tree. SBT representation is produced via the tree preorder traversal algorithm. The detailed procedure is as follows: 1) from the root node, SBT first uses a pair of brackets to represent the tree structure and puts the root node itself behind the right bracket. 2) Then, it traverses the subtrees of the root node and puts all root nodes of subtrees into the brackets. 3) Finally, it traverses each subtree recursively until all nodes are traversed and the final sequence is produced.

AST Path. AST path proposed by Alon et al. [5] is a common AST traversal method, which represents the AST using a set of paths extracted from the AST. An AST path is a path between two terminals in the AST, which is formally defined as follows.

[1] https://github.com/antlr/grammars-v4/tree/master/java/java
[2] https://github.com/tree-sitter/tree-sitter-java

Just Accepted

Definition 3.1 (AST Path). Given an $AST := \langle N, T, X, s, \delta, \phi \rangle$ (see Definition 2.1), an AST-path of length k is a sequence of the form: $n_1 d_1 \cdots n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals, for $i \in [2 \cdots k] : n_i \in N$ are non-terminals and for $i \in [1 \cdots k] : d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree). If $d_i = \uparrow$, then: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. For an AST-path p, we use $start(p)$ to denote n_1 - the starting terminal of p; and $end(p)$ to denote n_{k+1} - its final terminal.

We use path-context as the final input of the encoding models. A path-context is a tuple of an AST path and the values of two terminals, which are formally defined as follows.

Definition 3.2 (Path-context). Given an AST Path p, its path-context is a triplet $\langle x_{start}, p, x_{end} \rangle$ where $x_{start} = \phi(start(p))$ and $x_{end} = \phi(end(p))$ are the values associated with the start and end terminals of p.

Following previous work [6], we use maximum *length* – the maximal value of k – to limit the length of an AST Path, and the maximum *width* – the maximal difference in child index between two child nodes of the same intermediate node – to limit the horizontal distance between nodes in an AST Path, thereby limiting the size of the training data and reduce sparsity. These values are determined empirically as hyperparameters.

Binary Tree. Considering the computational cost, some existing code representation works [41, 82, 91] transform an otherwise complex AST into a structurally simplified binary tree. Unlike a binary search tree (BST) designed for ordered data retrieval, this binary tree is a purely topological variant where each non-leaf node has at most two children. In an AST, different kinds of nodes may have different numbers of children, which can cause problems in parameter-sharing [91]. Usually, to avoid this problem, ASTs are transformed into binary trees whose nodes only have two or zero children. The process contains the following two steps: 1) split nodes with more than two children, generate a new right child together with the old left child as its children, and then put all children except the leftmost as the children of this new node; repeat this operation in a top-down way until only nodes with zero, one, and two children are left; 2) combine nodes with one child with its child. Now only nodes with zero or two children remain, and the AST is transformed into a binary tree. Fig. 2 shows a simplified version of AST-to-Binary-Tree conversion process using fundamental tree-binarization algorithms.

Split AST. Due to the complexity of programs, the ASTs of the program source code are usually large and deep, leading to long training time and gradient vanishing problems for DL models [68]. To overcome this problem, some researchers propose to split the large and deep AST into a set of small and shallow subtrees. For example, Zhang et al. [99] split an AST into small statement trees, each of which represents a statement in the source code. Shi et al. [68] split an AST into a set of subtrees that are reconstructible. After learning the representation of subtrees, they reconstruct the split ASTs and learn the AST's representation from all subtrees' representation by a tree-based neural model. Lin et al. [52] first split the code of a method based on the blocks in the dominator tree of the control flow graph (CFG), and then generate a split AST for each split code.

In this paper, we mainly investigate the state-of-the-art AST split solution proposed by Lin et al. [52]. The reason why we investigate the solution by Lin et al. is detailed in Section 5.1.

We show examples of data produced by AST preprocessing methods in the supplemental material and our repository [75].

Just Accepted

3.4 AST Encoding Method

In this section, we will introduce two types of AST encoding methods: sequence models and tree-structured models. More technical details about the AST encoding methods are provided in the supplemental material.

3.4.1 *Sequence models.*

(i) Bidirectional Long Short Term Memory (BiLSTM).

The LSTM architecture [35] addresses the problem of learning long-term dependencies of RNN by introducing a memory cell that can preserve state over long periods of time [77]. Bidirectional LSTM (BiLSTM) [66] uses two

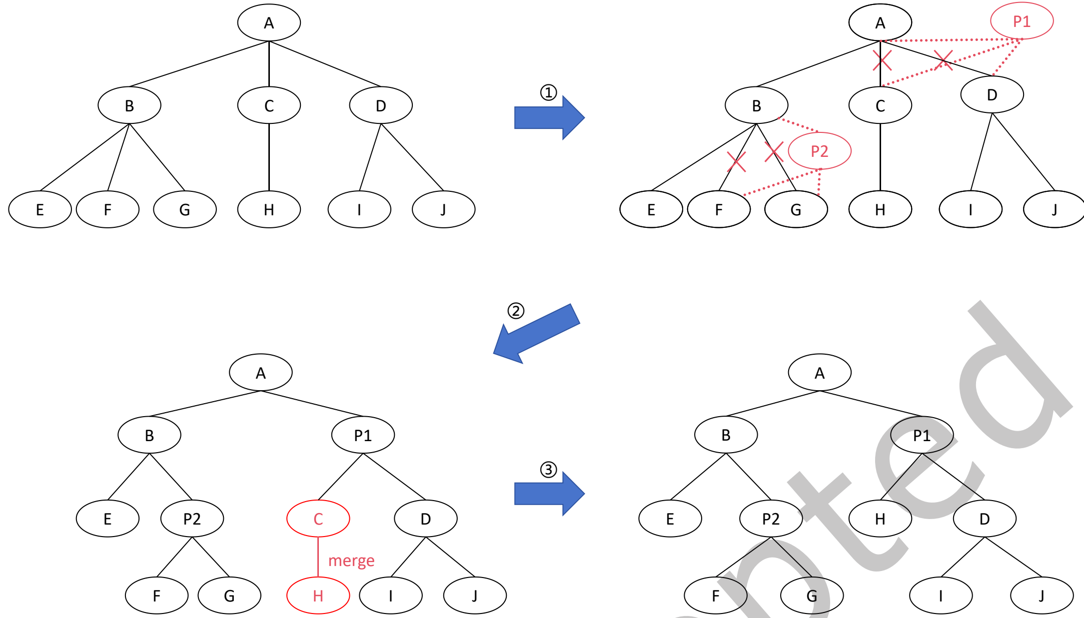


Fig. 2. Simplified AST-to-Binary-Tree conversion process.

Just Accepted

LSTMs at each layer. One LSTM takes the original sequence as input, and the other takes the reversed sequence as input. So it is capable of modeling the sequential dependencies between words and phrases in both directions of the sequence.

(ii) Transformer.

Transformer [81] has demonstrated strong feature learning ability on NLP tasks [46] and also proved to be effective in software engineering studies [1, 33, 67]. The transformer follows an encoder-decoder structure and utilizes a multi-headed self-attention mechanism and positional encoding to draw global dependencies between input and output. For the code clone detection and the code search task, the decoder part of the Transformer is not used since a decoder is not needed. For the code summarization task, both the encoder and decoder are used.

3.4.2 Tree-structured models.

(i) Tree-Structured Long Short-Term Memory Networks (TreeLSTM)

TreeLSTM was first proposed by Tai [77] to capture the syntactic properties of natural language. TreeLSTM is a generalization of LSTMs to model tree-structured topologies. Given a tree, let $C(j)$ denote the set of children of node j. The TreeLSTM unit at each node j is defined to be a collection of vectors in $\mathbb{R}^d$, where $\mathbb{R}$ is the set of real numbers, d is the memory dimension of the TreeLSTM: an input gate i_j, forget gates f_{jk} where $k \in C(j)$, an output gate o_j, a memory cell c_j and a hidden state h_j. The entries of the gating vectors i_j, f_j and o_j are in $[0, 1]$. There are two types of TreeLSTM: Child-Sum TreeLSTM and N-ary TreeLSTM. The Child-Sum TreeLSTM is used on tree structures where the number of children is arbitrary, and children's orders are not considered. Contrastingly, the N-ary TreeLSTM is used on tree structures where the number of children is at most N and children are ordered, i.e., they can be indexed from 1 to N. TreeLSTM has been widely adopted to capture the syntactic features in ASTs. CDLH [91] uses TreeLSTM to learn representations of code fragments for clone detection, where code

fragments are parsed to ASTs. Wang et al. [89] apply TreeLSTM to extract the important information from ASTs for the task of code summarization.

(ii) AST-Trans.

AST-Trans [78] is a simple variant of the Transformer model to efficiently handle the tree-structured AST. AST-Trans exploits ancestor-descendant and sibling relationship matrices to represent the tree structure, and uses these matrices to dynamically exclude irrelevant nodes.

AST-Trans has the same encoder and decoder structure as the Transformer, while replacing the single-head self-attention with tree-structured attention. The absolute position embedding from the original Transformer is replaced with relative position embeddings defined by the two relationship matrices to better model the dependency.

For an AST, it will first be linearized into a sequence, which means being transformed into SBT [37] in our experiment. Then the ancestor-descendant and sibling relationships among its nodes will be denoted through two specific matrices. Based on the matrices, tree-structured attention is adopted to better model these two relationships. In the following part, we will introduce the construction of relationship matrices and tree-structured attention.

3.5 Confounding Factors Separation

To measure the impact of four AST parsers on the downstream tasks, many other steps are required, like processing and encoding. To separate the effect of confounding factors from these steps, we utilize strategies as follows:

- **RQ1: Controlling preprocessing and encoding to isolate the parser's effect.** When comparing the four AST parsing methods, we fix both the preprocessing techniques (SBT or Raw AST) and the encoding models (BiLSTM or TreeLSTM). We select SBT/Raw AST because these methods preserve rich node-level and structural information, enabling a more faithful examination of the parser's impact on downstream tasks. Likewise, BiLSTM and TreeLSTM are chosen due to their simplicity, widespread adoption, and minimal confounding complexity. With these components held constant, the performance differences observed in RQ1 can be attributed primarily to the AST parsing method itself.
- **RQ2: Reducing parser-specific bias with representative parsers.** In RQ2, we select three representative parsers with different AST granularities: JDT and Tree-sitter, which generate compact ASTs, and ANTLR, which produces substantially larger ASTs. (Note that JDT cannot process syntactically incorrect fragments under the Split-AST setting, so it is excluded for that specific scenario.) We again adopt the same BiLSTM/TreeLSTM encoders as in RQ1 to ensure that differences across experiments arise from the parsers, not downstream components.
- **RQ3: Using the best-performing configuration from RQ1 and RQ2.** To further mitigate unrelated variability, in RQ3 we use the best-performing parser identified in RQ1 (JDT) together with the best-performing preprocessing method from RQ2 (SBT/Raw AST). This allows us to focus on the behavior of AST representations under more challenging or realistic settings, while minimizing the influence of external factors.
- **RQ4: Evaluating token vs. AST representations under an optimal setup.** In RQ4, we adopt the optimal configuration identified in the earlier research questions, namely employing the JDT-generated ASTs together with the SBT-based preprocessing strategy and a Transformer-based encoder. This design ensures that the comparison between token-based and AST-based representations is conducted under a robust and consistent experimental setup, eliminating potential artifacts introduced by suboptimal parser or preprocessing choices.

In summary, across all research questions, we systematically fix preprocessing techniques, encoding models, and parser selections whenever these components are not the primary variables under investigation. This experimental

Just Accepted

design allows us to confidently separate the true effects of AST parsing methods from other potential confounders.

3.6 Experimental Setup

3.6.1 Code-related Tasks.

(1) Code Clone Detection. This task is a binary classification problem to predict whether a given pair of codes has the same semantics, with the F_1-score used as the evaluation metric. Given n code fragments $\{C_1, ..., C_n\}$ where C_i is the i-th raw code fragment, and pairwise labels to indicate whether two code fragments belong to a clone pair or not: y_{ij}=1 if (C_i, C_j) is a clone pair, y_{ij}=0 if (C_i, C_j) is not a clone pair, then the training set is represented by a set of triplets $\mathcal{D} = \{(C_i, C_j, y_{ij}) | i, j \in [n], i < j\}$, where $[n] = 1, 2, ..., n$ [91]. Given a pair of code fragments, we first convert them into ASTs using AST parsers and preprocess the ASTs if necessary. Then we use different encoding methods to encode the input as vectors z_i and z_j. Given z_i and z_j, their distance is measured by $z_{ij} = |z_i - z_j|$ for semantic relatedness [77]. We use a linear layer and a softmax function to map z_{ij} to $r_{ij} \in R^2$, where $r_{ij}[0]$ and $r_{ij}[1]$ represent the probability of $y_{ij} = 0$ or $y_{ij} = 1$ separately. We use cross-entropy as the loss function, which is defined as

$$J(\Theta, \hat{y}, y) = \sum \left(-(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y}))\right) \tag{1}$$

Our goal is to minimize the loss, and we use Adam as the optimizer. The training will stop if the model's F_1-score on the evaluation dataset doesn't exceed the previous best F_1-score for more than 3 continuous epochs. After training, the model with the best F_1-score is stored. For new code pairs (C_i, C_j), they will be fed into the reloaded model and get r_{ij} and $\hat{y} = r_{ij}[1]$. We get the prediction of whether (C_i, C_j) is a clone pair by

$$Prediction = \begin{cases} True & \hat{y} > \delta \\ False & \hat{y} \le \delta \end{cases} \tag{2}$$

where δ is the threshold. We enumerate δ from 0.01 to 0.99 with a step of 0.01 and evaluate the model on the evaluation dataset. The δ with the best model F_1-score is chosen as the final δ.

(2) Code Summarization Code summarization is the task of generating natural language descriptions for the given code snippets. It can help developers quickly understand and maintain source code [74]. With the availability of large-scale data, deep learning methods are widely used to improve the performance of code summarization tools. The pioneers find that some Seq2Seq models, such as RNN [7, 20] and LSTM [43], are capable of modeling the semantic relations between code snippets and summaries. To capture the structural information, some researchers use TreeLSTM to process structural representations of code [69, 77]. However, RNN-based methods have poor long-term dependence. To break through the bottleneck that may be encountered when modeling long sequences, some Transformer-based methods [1, 94] are proposed, which show superior performance for the task.

Generally, the code summary model consists of two parts: an encoder and a decoder. The encoder uses the required input data to generate the context vectors e^{En}. Then, the decoder takes in e^{En} and unfolds it into the target sequence. The words in the target sequence are predicted one by one, which means the word with the highest probability is selected from the vocabulary each time.

Specifically, when both encoder and decoder are based on LSTM, the initial hidden state and cell state of the decoder are initialized by e^{En}. The dynamic model is as follows:

$$\begin{aligned} h_t, c_t &= f(y_{t-1}, h_{t-1}, c_{t-1}) \\ p(y_t | Y_{<t}, X) &= g(y_{t-1}, h_{t-1}) \end{aligned} \tag{3}$$

where $f(\cdot)$ and $g(\cdot)$ are activation functions. h_t and c_t are the hidden state and the cell state at time t. y_t is the predicted target word at t. $Y_{<t}$ denotes the history $y_1, y_2, \dots, y_{t-1}$.

Just Accepted

When both encoder and decoder are based on Transformer, the memory of the decoder is initialized by e^{En}. The dynamic model is as follows:

$$p(y_t|Y_{<t}, X) = k(Y_{<t}, e^{En}) \tag{4}$$

where $k(\cdot)$ is activation function.

In our experiment, when the encoder is BiLSTM, Transformer, or AST-Trans, the decoder is the same neural network architecture, which means BiLSTM, Transformer, or AST-Trans. When the encoder is TreeLSTM, the decoder is BiLSTM.

(3) Code Search. The task is to find specific code snippets according to a natural language query. Previous studies mainly relied on the textual similarity between source code and natural language queries. Since the rise of DL technology, today it is more often to jointly embed code snippets and natural language descriptions into a high-dimensional vector space, in such a way that a code snippet and its corresponding description have similar vectors. Using the unified vector representation, code snippets related to a natural language query can be retrieved according to their vectors. Semantically related words can also be recognized, and irrelevant/noisy keywords in queries can be handled. According to related works, the loss function tends to be a triplet loss function, which is defined as:

$$L(a, p, n) = max(d(a, p) - d(a, n) + \alpha, 0) \tag{5}$$

where $d(x, y)$ measures the distance between x and y, and α is the margin that ensures the positive code snippet is closer to the anchor than the negative code snippet. We can consider different distance metrics for d, such as the Cosine distance . To construct a triplet, we first randomly select a description from the dataset as the anchor. Its corresponding code snippet is then used as the positive example, while a random code snippet unrelated to the anchor is chosen as the negative example.

3.6.2 Datasets.

(1) Dataset for Code Clone Detection.

BigCloneBench. BigCloneBench is a well-known dataset of method-level Java code clones provided by Svajlenko et al. [76], which is a widely used large code clone benchmark [56]. It consists of known true and false positive clones from a big data inter-project Java repository. Lu et al. [56] filter the BigCloneBench dataset by discarding code snippets without any tagged true or false clone pairs. In this paper, we directly use the filtered BigCloneBench dataset provided by Lu et al. in CodeXGLUE [3] [56] and remove the data if there exists at least one AST parser that cannot parse the code. Table 1 presents the statistics of the BigCloneBench dataset, including programming language, training set size, validation set size, and test set size.

(2) Dataset for Code Search and Code Summarization.

CodeSearchNet. The CodeSearchNet dataset [42] is a vast collection of methods accompanied by their respective comments, written in Go, Java, JavaScript, PHP, Python, and Ruby. These methods are sourced from open-source projects hosted on GitHub. This dataset is widely used in studying code search [11, 30, 33, 53, 95] and code summarization [33, 52, 67, 103]. Analogously, Lu et al. [56] show that some comments contain content unrelated to the code snippets and perform data cleaning on the CodeSearchNet dataset. Therefore, in this paper, we directly use the clean version of the CodeSearchNet dataset provided by them in CodeXGLUE. In our specific case, we consider Java as the only programming language in our experiments, so we concentrated on utilizing the Java-related data from the CodeSearchNet dataset while disregarding data in other languages. Table 1 presents the statistics of the Java corpus of the CodeSearchNet dataset.

3.6.3 Evaluation Metrics.

(1) Evaluation Metrics for Code Clone Detection

[3]https://github.com/microsoft/CodeXGLUE

Just Accepted

Table 1. Dataset statistics

Dataset	Language	Training	Validation	Test
BigCloneBench	Java	900,713	415,416	415,416
CodeSearchNet	Java	164,923	5,183	10,955

Precision (*P*) measures among all of the clone pairs detected by a clone detection approach, how many of them are true clone pairs:

$$P = \frac{\text{\# of true clone pairs}}{\text{Total \# of detected clone pairs}} \tag{6}$$

Recall (*R*) measures among all known true clone pairs, how many of them are detected by a clone detection approach:

$$R = \frac{\text{\# of true clone pairs detected}}{\text{Total \# of known true clone pairs}} \tag{7}$$

F_1**-score (**F_1**)** is a harmonic mean of precision and recall.

$$F_1 = 2 \times \frac{P \times R}{P + R} \tag{8}$$

Following previous work [33, 91, 99], we use Recall, Precision, and F_1-score as the evaluation metric in the code clone detection task.

(2) Evaluation Metrics for Code Summarization

We use three automatic metrics (BLEU [61], METEOR [8], and ROUGE [19]) to evaluate the quality of the generated comments in the code summarization task. These three metrics are widely used in code summarization [29, 39, 43, 81, 84, 94].

BLEU-4, the abbreviation for BiLingual Evaluation Understudy [61], is widely used for evaluating the quality of generated summaries [39, 43, 84]. It compares *n*-grams in the predicted and target summaries. Typical implementations of BLEU scores set the range of *n* from 1 to 4. An average BLEU score is then computed by combining these individual *n*-gram scores using predetermined weights.

METEOR [8] is introduced to address the concerns of using BLEU [61]. It is also widely used to evaluate the quality of generated summaries [89, 96, 98]. It combines n-gram precision and n-gram recall by taking their harmonic mean to compute a measure of similarity.

ROUGE-L [19] evaluates how much reference text appears in the generated text. It is also widely used to evaluate the quality of generated code summaries [9, 52, 67]. Based on the longest common subsequence (LCS), it uses the F-score, which is the harmonic mean of precision and recall.

BertScore [100] is a metric that reflects the semantic similarity of two summaries. It uses a variant of BERT [23] (we use the default $RoBERTa_{large}$ [54]) to embed every token in the summaries, and computes the pairwise inner product between tokens in the reference summary and generated summary. Then it matches every token in the reference summary and the generated summary to compute the precision, recall, and F_1 measure. We report the F_1 measure of BERTScore.

(3) Evaluation Metrics for Code Search

SuccessRate@k (*SR@k*) [82] measures the percentage of queries for which more than one correct result could exist in the top *k* ranked results, which is calculated as follows:

$$SR@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \delta(FRank_{Q_i} \leqslant k) \tag{9}$$

Just Accepted

where Q is a set of queries, $\delta(\cdot)$ is a function that returns 1 if the input is true and returns 0 otherwise, and $FRank_{Q_i}$ refers to the rank position of the correct result for the i-th query in Q. SuccessRate@k is important because a better code search engine should allow developers to discover the needed code by inspecting fewer returned results. The higher the SuccessRate value is, the better the code search performance is.

Mean Reciprocal Rank (*MRR*) [82] is the average of the reciprocal ranks of results of a set of queries Q. The reciprocal rank of a query is the inverse of the rank of the first hit result. The MRR is defined as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{FRank_{Q_i}} \tag{10}$$

where $|Q|$ is the size of the query set. The higher the MRR value is, the better the code search performance is.

3.6.4 Statistical Tests.

To find out whether or not there is a statistically significant difference between different treatments in our experiments, we conduct statistical tests.

(1) Code Clone Detection: McNemar's Test + Odds Ratio

We use McNemar's test for the significance test. The experimental results are divided into four categories and represented in a 2*2 matrix. The four categories are (a) samples that are misclassified by both treatment-1 and treatment-2, (b) samples that are misclassified by treatment-1 and correctly classified by treatment-2, (c) samples that are correctly classified by treatment-1 and misclassified by treatment-2, and (d) samples that are correctly classified by both treatment-1 and treatment-2. When the $p-value$ is less than 0.05, there is a significant difference.

We use the Odds Ratio (OR) to measure effect size. The experimental results are also divided into four categories and represented in a 2*2 matrix. To ensure that the OR value has meaning, the four categories are different as above. The four categories are (a) samples that are misclassified by treatment-1, (b) samples that are correctly classified by treatment-2, (c) samples that are misclassified by treatment-2, and (d) samples that are correctly classified by treatment-2. Thereby, the Odds Ratio means that the risk of misclassification using treatment-1 is {OR} times that of using treatment-2. When the OR value is 1, it means that the two treatments have no impact on the experimental results. When the OR value is greater than 1, it means that treatment-1 is more likely to result in a misclassification. When the OR value is less than 1, it means that treatment-2 is more likely to result in a misclassification.

(2) Code Summarization & Code Search: Wilcoxon signed-rank test + Cliff's delta

We use the Wilcoxon signed-rank test for the significance test and Cliff's delta to measure effect size. For code summarization, the observation value is the BLEU, METEOR, ROUGE-L, and BertScore for each sample in the test dataset. For code search, the observation value is $1/rank$ where $rank$ is the rank of the ground-truth code snippet. When the $p - value$ is less than 0.05, there is a significant difference. Cliff's delta ranges from -1 to 1. When Cliff's delta is 0, it means the compared groups tend to overlap. The larger the absolute value of Cliff's delta is, the larger the effect size is.

3.6.5 Experimental settings. At the training stage, we employ the AdamW optimizer [55] along with a linear learning rate scheduler. To ensure comprehensive training, we use an early stopping strategy based on the best validation F_1/BLEU/MRR to control the number of training epochs for the model. For specific parameter settings, since different models and code-related tasks use different parameters, please refer to our code published on GitHub [75]. All models are implemented using the PyTorch 1.10.0 framework with Python 3.8. All experiments are conducted on servers equipped with one or two Tesla V100 GPU(s) with 32 GB memory, running on Ubuntu 20.04.4.

Just Accepted

Table 2. Comparison of ASTs generated by different AST parsing methods in the five metrics. Column Parsing lists AST parsing methods for comparison.

Dataset	Parsering	Tree Size		Tree Depth		Branch Factor		Unique Types		Unique Tokens		Language Support
		Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	
BigCloneBench	JDT	312	213	13	12	**2**	**2**	22	22	58	48	Java
	srcML	529	356	22	21	1	1	26	26	67	55	Java/C/C++/C#
	ANTLR	**709**	**480**	**27**	**26**	**2**	**2**	**37**	**37**	**76**	**65**	Any
	Tree-sitter	485	329	14	13	**2**	**2**	28	27	**76**	**65**	Any
CodeSearchNet	JDT	117	83	10	10	**2**	**2**	16	15	29	24	Java
	srcML	193	135	18	17	1	1	23	23	37	29	Java/C/C++/C#
	ANTLR	**262**	**184**	**22**	**21**	**2**	**2**	**31**	**31**	**42**	**36**	Any
	Tree-sitter	181	126	11	11	**2**	**2**	20	19	**42**	**36**	Any
Average	JDT	214.5	148	11.5	11	**2**	**2**	19	18.5	43.5	36	–
	srcML	361	245.5	20	19	1	1	24.5	24.5	52	42	–
	ANTLR	**485.5**	**332**	**24.5**	**23.5**	**2**	**2**	**34**	**34**	**59**	**50.5**	–
	Tree-sitter	333	227.5	12.5	12	**2**	**2**	24	23	**59**	**50.5**	–

4 Results and Findings

4.1 Answer to RQ1: How do AST parsing methods affect the performance of AST-based code representation on subsequent code-related tasks?

As mentioned in Section 1, different AST parsing methods use different lexical and grammatical rules, resulting in ASTs with other structures and node labels. Therefore, before applying AST to subsequent code-related tasks, it is necessary to figure out whether and where the Raw ASTs generated by different AST parsing methods are different. This will help reveal which differences have an impact on AST-based code representation as well as subsequent code-related tasks. We follow the study of Utkin et al. [80] and use five metrics to characterize ASTs generated by AST parsing methods.

Tree size. It is defined as the total number of nodes in an AST.

Tree depth. It is defined as the number of nodes in the path from the root of an AST to its deepest node.

Branching factor: It is defined as the average number of children of non-leaf nodes in an AST.

Unique types: It is defined as the number of unique non-leaf nodes in an AST. Lower values of unique types represent a higher level of abstraction used in an AST parsing method, as it can represent the same code snippet in a more compact way [80].

Unique tokens: It is defined as the number of unique sub-tokens in AST leaf nodes, which often represent code tokens. Lower values of unique tokens also reflect a higher level of abstraction (e.g., whether an AST parsing method keeps binary operators as code tokens or as node types [80].

Tree size, tree depth, and branching factor reflect the difference in the structure of ASTs and can be used to estimate how different the trees are size-wise. Unique types and unique tokens reflect the difference in node labels of ASTs. Lower values in unique types and unique tokens mean a higher level of abstraction.

Specifically, we compare the ASTs generated by the four commonly used AST parsing methods, i.e., JDT, srcML, ANTLR, and Tree-sitter, which are discussed in detail in Section 3.2. We utilize them to generate ASTs for Java code snippets in the BigCloneBench and CodeSearchNet datasets, then compute the five metrics for each AST parsing method.

Table 2 presents the mean and median values of the computed metrics for each AST parsing method. From Table 2, it is observed that in terms of structure (i.e., tree size, tree depth, and branch factor), the ASTs generated by ANTLR achieve the maximum mean and median values. This means that ANTLR tends to generate larger, deeper ASTs than the other three. Compared with ANTLR, JDT is the other extreme. The ASTs generated by JDT achieve

Just Accepted

Table 3. Effect of AST parsing methods on the performance of models trained with AST

Preprocessing	Encoding	Parsing	Code Clone Detection			Code Summarization				Code Search			
			Recall	Precision	F_1	BLEU	METEOR	ROUGE-L	BertScore	SR@1	SR@5	SR@10	MRR
SBT	BiLSTM	JDT	**73.06**	78.68	**75.77**	**9.013**	**4.540**	16.45	**83.19**	0.1720	**0.4052**	**0.5276**	**0.2827**
		srcML	69.77	**80.39**	74.70	8.667	4.019	**17.04**	82.05	0.1395	0.3023	0.3604	0.2254
		ANTLR	71.59	80.11	75.61	7.689	3.353	13.49	82.50	0.1279	0.2093	0.2674	0.1791
		Tree-sitter	72.56	79.00	75.64	8.660	4.332	15.99	82.97	**0.1744**	0.3605	0.4419	0.2732
AST	TreeLSTM	JDT	84.25	**92.18**	**88.04**	**9.779**	**5.317**	**18.54**	83.15	**0.0906**	**0.2076**	**0.2660**	**0.1524**
		srcML	78.13	82.87	80.43	9.578	3.517	18.28	**83.25**	0.0556	0.1784	0.2485	0.1181
		ANTLR	**87.06**	84.04	85.52	9.372	3.425	17.79	83.10	0.0760	0.1520	0.2251	0.1226
		Tree-sitter	84.76	88.36	86.52	9.280	3.521	17.97	82.91	0.0760	0.1784	0.2193	0.1268

Table 4. Statistical tests results for RQ1. The first number is the p-value for McNemar's test / Wilcoxon signed-rank test. The second number (in the parentheses) is Odds Ratio / Cliff's delta.

Model	Parsing	Code Clone Detection	Code Summarization				Code Search
			BLEU	METEOR	ROUGE-L	BertScore	
BiLSTM	JDT vs srcML	**0.071(0.99)**	<0.001(-0.02)	<0.001(0.05)	<0.001(-0.05)	<0.001(0.25)	<0.001(0.87)
	JDT vs ANTLR	**0.082(1.01)**	<0.001(0.08)	<0.001(0.17)	<0.001(0.12)	<0.001(0.15)	<0.001(0.89)
	JDT vs Tree-sitter	**0.975(1.0)**	<0.001(0.02)	<0.001(0.03)	0.004(0.02)	<0.001(0.05)	<0.001(0.9)
	srcML vs ANTLR	<0.001(1.02)	<0.001(0.11)	<0.001(0.14)	<0.001(0.18)	<0.001(-0.1)	<0.001(-0.06)
	srcML vs Tree-sitter	0.049(1.01)	<0.001(0.04)	<0.001(-0.02)	<0.001(0.07)	<0.001(-0.21)	<0.001(0.09)
	ANTLR vs Tree-sitter	**0.081(0.99)**	<0.001(-0.07)	<0.001(-0.14)	<0.001(-0.1)	<0.001(-0.1)	<0.001(0.15)
TreeLSTM	JDT vs srcML	<0.001(0.59)	<0.001(-0.0)	<0.001(0.2)	**0.461(0.01)**	<0.001(-0.05)	<0.001(0.03)
	JDT vs ANTLR	<0.001(0.8)	<0.001(0.02)	<0.001(0.22)	<0.001(0.03)	**0.817(-0.0)**	<0.001(-0.8)
	JDT vs Tree-sitter	<0.001(0.86)	<0.001(0.02)	<0.001(0.21)	0.003(0.02)	<0.001(0.05)	<0.001(0.08)
	srcML vs ANTLR	<0.001(1.36)	<0.001(0.02)	0.005(0.02)	<0.001(0.02)	<0.001(0.04)	<0.001(-0.81)
	srcML vs Tree-sitter	<0.001(1.46)	<0.001(0.02)	**0.421(-0.0)**	0.012(0.01)	<0.001(0.1)	<0.001(0.05)
	ANTLR vs Tree-sitter	<0.001(1.08)	**0.989(0.0)**	0.003(-0.02)	**0.066(-0.02)**	<0.001(0.05)	<0.001(0.83)

the minimum mean and median values in both tree size and tree depth. The same phenomenon that ANTLR and JDT tend to generate larger and smaller ASTs, respectively, can also be observed in the CodeSearchNet dataset. In terms of abstraction level, ANTLR generates ASTs with more unique types and unique tokens, followed by Tree-sitter and srcML, and then JDT.

✎ **Summary** ▸ The ASTs generated by different AST parsing methods differ in structure size (especially in tree size and tree depth), and abstraction level. The AST generated by JDT is the smallest, shallowest, and has the highest level of abstraction, whereas ANTLR is the opposite. Tree-sitter and srcML are both intermediate in structure size and abstraction level between JDT and ANTLR. ◂

Therefore, we further discuss the influence of these differences on AST-based code representation and the performance of models trained with AST on subsequent code-related tasks. Specifically, to obtain more general findings and conclusions, the experiments in this section involve all four AST parsers (i.e., JDT, srcML, ANTLR, and Tree-sitter), two AST preprocessing methods (i.e., SBT and Raw AST), two AST encoding methods (i.e., BiLSTM for encoding SBT and Child-Sum TreeLSTM for encoding Raw AST), and three tasks (i.e., code clone detection, code summarization, and code search).

Table 3 presents the overall performance of BiLSTM-based and TreeLSTM-based models trained with ASTs generated by four AST parsing methods on three subsequent code-related tasks. Table 4 shows the statistical test results. It is observed that on the code clone detection task, when the encoding model is BiLSTM, srcML is significantly worse than ANTLR and Tree-sitter, while the results of other parsers show no significant difference.

Just Accepted

Table 5. Pearson correlation coefficient between Tree Size/Tree Depth/Branch Factor/Unique Types/Unique Tokens and the evaluation result of each sample in the test dataset. Numbers in parentheses are p-values.

Task	Encoding	Tree Size	Tree Depth	Branch Factor	Unique Types	Unique Tokens
Code Clone Detection	BiLSTM	-0.004(<0.001)	-0.004(<0.001)	0.003(<0.001)	-0.002 (0.003)	-0.006(<0.001)
	TreeLSTM	0.004(<0.001)	-0.18(<0.001)	0.030(<0.001)	-0.007(<0.001)	0.002(0.034)
Code Summarization	BiLSTM	-0.023(<0.001)	-0.038(<0.001)	-0.023(<0.001)	-0.039(<0.001)	-0.012(0.013)
	TreeLSTM	-0.026(<0.001)	-0.013(0.008)	-0.023(<0.001)	-0.012(0.015)	-0.021(<0.001)
Code Search	BiLSTM	0.062(<0.001)	0.059(<0.001)	-0.008(0.087)	0.077(<0.001)	0.079(<0.001)
	TreeLSTM	-0.064(<0.001)	-0.171(<0.001)	-0.086(<0.001)	-0.184(<0.001)	-0.023(<0.001)

Just Accepted

The reason why there is no significant difference between other AST parsers is the insufficient learning ability of BiLSTM. When the encoding model is TreeLSTM, there is a significant difference between the four AST parsers, and JDT achieves the highest F_1-score. On the code summarization task, JDT achieves the highest score across all metrics except for ROUGE-L when the encoding model is BiLSTM and BertScore when the encoding model is TreeLSTM. For ROUGE-L, the results of the statistical test show that there is no significant difference between JDT and the best-performing parser srcML. Hence, on the code summarization task, overall JDT performs best across all metrics. On the code search task, except that TreeLSTM trained with AST generated by Tree-sitter performs best in SR@1, the models trained with the AST generated by JDT perform best in other cases (e.g., in all four metrics and two preprocessing settings). And there is a significant difference between all AST parsers. In most cases, the AST parser will affect the effectiveness of code representation. To sum up, JDT performs best on all three tasks, achieving the highest scores on most metrics.

Recall that the tree size, tree depth, unique types, and unique tokens of the ASTs generated by JDT are the smallest. Therefore, although the ASTs generated by srcML, ANTLR, and Tree-sitter generally contain richer information than that of JDT, this richness could potentially impose a higher learning burden on the model at the same time, which may not be conducive to improving the model's performance on subsequent code-related tasks.

To verify our conjecture, we calculate the Pearson correlation coefficient between Tree Size/Tree Depth/Branch Factor/Unique Types/Unique Tokens and the evaluation result of each sample in the test dataset, shown in Table 5. To represent the evaluation result of each sample, for code clone detection, we use a 0-1 sequence where 0 means the sample is incorrectly classified and 1 means the sample is correctly classified. For code summarization, we use the BLEU score of each sample. For code search, we use $1/rank$, where $rank$ is the rank of the ground-truth code snippet. From Table 5, it is observed that TreeSize, TreeDepth, Branch Factor, Unique Types, and Unique Tokens have a weak negative correlation with the evaluation results in most cases. In the code search task, when the model is BiLSTM, all metrics except Branch Factor are positively correlated with the experimental results (and the four AST parsers have only a small difference in Branch Factor). Theoretically, it should be that the larger the AST parser is in the four metrics, the better the performance is. However, it is not the truth. For example, ANTLR performs worse than srcML. It reveals that these five metrics (especially Unique Types and Unique Tokens) cannot fully describe all the features of ASTs. We need to find more metrics to reflect features of ASTs, such as the node label quality.

✎ **Summary** ▸ JDT, which generates ASTs with the smallest tree size, shallowest tree depth, and highest abstraction level, yields the most favorable outcomes on all three tasks. This suggests that too many nodes in the AST or too deep trees can impede the DL model's ability to learn code semantics from the AST. ◂

4.2 Answer to RQ2: How do AST preprocessing methods affect the performance of AST-based code representation on subsequent code-related tasks?

In this section, we discuss the impact of AST preprocessing methods on AST-based code representation as well as subsequent code-related tasks. AST preprocessing methods we investigate can be classified into two categories: one processes ASTs into sequential data, while the other processes ASTs into structural data. For sequential data, including BFS, SBT, and AST Path, we uniformly leverage BiLSTM, a representative sequence model, to encode them. For structural data encompassing Raw AST and Split AST, we uniformly utilize Child-Sum TreeLSTM to encode them. For the Binary Tree, we use N-ary TreeLSTM as the encoder.

In RQ1, the sizes and depths of ASTs generated by different AST parsing methods are ranked in ascending order as follows: JDT, Tree-sitter, srcML, and ANTLR. In this RQ, we select JDT (which generates the smallest AST) and ANTLR (which generates the largest AST) as representative AST parsing methods to eliminate the influence of AST sizes on the RQ conclusions. Meanwhile, for Raw AST, Binary Tree, and Split AST, Tree-sitter (producing the second-smallest AST) is used instead of JDT. During the processing of Split AST, code snippets are divided into code pieces, which may contain syntactic errors that prevent JDT from generating their ASTs.

Table 6 presents the overall performance of BiLSTM-based and TreeLSTM-based models on three code-related tasks. Table 7 shows the statistical test results. These models are trained using ASTs preprocessed by the above six AST preprocessing methods. Combining the results in Table 6 with the observations in Section 3.3, we can analyze the impact of AST node and structure information on AST-based code representation and different code-related tasks, providing some guidelines for choosing appropriate AST preprocessing methods for each code-related task.

For the code clone detection task, from Table 6, it is observed that for three sequential AST data, the conclusions on JDT and ANTLR are consistent: the BiLSTM-based model trained with SBT achieves the highest F_1-score, outperforming those trained with AST Path and BFS. For the three structural AST data, the best-performing AST preprocessing method varies. Raw AST performs best when preprocessing ASTs generated by JDT, while Binary Tree yields the best results with ANTLR-generated ASTs. Split AST consistently performs the worst. The effectiveness of Binary Tree for ANTLR-generated ASTs may be attributed to the presence of numerous parent nodes with only one child node. During the processing of the Binary Tree, such nodes are merged with their child node, reducing AST depth and potentially improving performance. Table 7 shows that there are significant differences between the experimental results of each preprocessing method.

Recall that, as detailed in Section 3.3, SBT and Raw AST retain complete node and structure information while other preprocessing methods do not. This demonstrates that both node and structure information are crucial to identifying code clones. Therefore, the preprocessing method that preserves complete AST nodes and structure information is optimal, especially for small ASTs.

For the code summarization task, from Table 6, it is observed that for three sequential AST data, the conclusions on JDT and ANTLR are consistent in general: the BiLSTM-based trained with SBT has achieved the best performance on BLEU, METEOR, ROUGE-L, and BertScore (except that ANTLR-generated SBT has lower BertScore than BFS). Although JDT-generated SBT may not significantly outperform BFS in terms of BLEU and ROUGE-L while ANTLR-generated SBT performs relatively better than AST Path and BFS. Notably, the BLEU score of the model trained with SBT has exhibited an improvement of 1.36% and 0.87% on JDT and 2.82% and 4.27% on ANTLR over those of the BiLSTM-based models trained with BFS and AST Path, respectively. It indicates that complete node and structure information (SBT) is the optimal choice when generating code summaries using sequence models. Besides, BFS (complete nodes, little structure information) has exhibited superior performance compared to AST Path (partial nodes, partial structure information), suggesting that nodes play a crucial role in sequence models to generate code summaries.

Among the three structural AST data, the TreeLSTM-based model trained with Binary Tree attains the highest scores on BLEU, METEOR, ROUGE-L and BertScore with significant difference. Therefore, we can find that

Just Accepted

Table 6. Overall performance of models trained with six types of AST preprocessing methods.

Parsing	Encoding	Preprocessing	Code Clone Detection			Code Summarization				Code Search			
			Recall	Precision	F_1	BLEU	METEOR	ROUGE-L	BertScore	SR@1	SR@5	SR@10	MRR
JDT	BiLSTM	BFS	68.24	78.52	72.88	8.347	3.849	16.01	82.07	0.1453	0.3372	0.4244	0.2440
		SBT	**73.06**	**78.68**	**75.77**	**9.013**	**4.540**	**16.45**	**83.19**	**0.1720**	0.4052	**0.5276**	0.2827
		AST Path	72.84	77.45	75.08	7.550	4.118	14.07	82.47	0.1543	**0.4383**	0.5185	**0.2904**
Tree-sitter	TreeLSTM	Raw AST	**84.76**	88.36	**86.52**	9.280	3.521	17.97	82.91	**0.0760**	**0.1784**	0.2193	0.1268
		Binary Tree	80.64	**90.96**	85.49	**9.570**	**3.761**	**18.33**	**83.29**	0.0706	0.1765	0.2353	**0.1279**
		Split AST	66.64	71.57	69.02	9.336	3.634	17.85	83.03	0.0588	0.1647	**0.2588**	0.1179
ANTLR	BiLSTM	BFS	62.92	**82.23**	71.29	5.336	1.845	8.295	80.24	0.0933	0.2391	0.3149	0.1695
		SBT	**71.59**	80.11	**75.61**	**7.689**	**3.353**	**13.49**	**82.50**	**0.1279**	0.2093	0.2674	0.1791
		AST Path	61.17	65.58	63.30	5.041	1.853	7.972	79.12	0.0978	**0.2786**	**0.3524**	**0.1839**
	TreeLSTM	Raw AST	**87.06**	84.04	85.52	9.372	3.425	17.79	83.10	**0.0760**	**0.1520**	0.2251	0.1226
		Binary Tree	85.60	**89.51**	**87.51**	**9.443**	**3.734**	**18.17**	**83.19**	0.0673	0.1813	**0.2456**	**0.1273**
		Split AST	55.87	51.40	53.45	9.403	3.573	18.02	83.18	0.0526	0.1023	0.1637	0.0868

although Raw AST contains complete node and structure information, the TreeLSTM-based model trained with it has the worst code summarization performance. Compared with Raw AST, both Binary Tree and Split AST remove some redundant node/structure information or retain only essential nodes and structures, resulting in improved performance in code summarization. Of course, it is undeniable that both node and structure information is crucial to facilitate AST-based code representation and code summarization tasks.

For the code search task, from Table 6, it is observed among the three sequential AST data, there is a significant discrepancy between the conclusions of JDT and ANTLR. On JDT, the BiLSTM-based model trained with SBT attains the highest values on SR@1 and SR@10 (i.e., 0.1720 and 0.5276, respectively), whereas the BiLSTM-based model trained with AST Path has the highest values on SR@5 and MRR (i.e., 0.4383 and 0.2904, respectively). In comparison to the BiLSTM-based model trained with AST Path, the BiLSTM-based model trained with SBT has exhibited improvements of 11.47% and 1.76% on SR@1 and SR@10, respectively. However, on ANTLR, BiLSTM-based model trained SBT performs best on SR@1 and AST path attains highest values on SR@5, SR@10, and MRR. Conversely, the BiLSTM-based model trained with AST has improved SR@5 and MRR by 8.17% and 2.72% over the BiLSTM-based model trained with JDT-generated SBT and improved SR@1 by 37.08% with ANTLR-generated SBT, respectively.

Among the three structural AST data, the TreeLSTM-based model trained with AST yields the best results on SR@1 and SR@5 (0.0760 and 0.1784, respectively), the TreeLSTM-based model trained with Binary Tree exhibits the best performance on MRR (0.1279), and while the TreeLSTM-based model trained with Split AST exhibits the best performance on SR@10 (0.2588), a similar result can be observed on models trained with ANTLR-generated AST. From Table 7, it is observed that experimental results between different AST preprocessing methods are significant. In terms of overall performance, the TreeLSTM-based model trained with Raw AST emerges as the top performer, followed by the models trained with Binary Tree and Split AST. However, when it comes to larger AST, the TreeLSTM-based model trained with Binary Tree takes the lead.

Based on the performance of BiLSTM-based and TreeLSTM-based models trained with different preprocessed data, we can conclude that, similar to the code clone detection task, capturing complete node and structure information is most advantageous on code search tasks.

Just Accepted

Table 7. Statistical tests results for RQ2. The first number is the p-value for McNemar's test / Wilcoxon signed-rank test. The second number (in the parentheses) is Odds Ratio / Cliff's delta.

Parser	Model	Preprocessing	Code Clone Detection	Code Summarization				Code Search
				BLEU	METEOR	ROUGE-L	BertScore	
JDT	BiLSTM	BFS vs SBT	<0.001(1.09)	0.823(-0.01)	<0.001(-0.07)	0.875(0.01)	<0.001(-0.26)	<0.001(-0.89)
		BFS vs AST Path	<0.001(1.05)	<0.001(0.11)	<0.001(0.04)	<0.001(0.13)	<0.001(-0.09)	<0.001(-0.01)
		SBT vs AST Path	<0.001(0.96)	<0.001(0.11)	<0.001(0.09)	<0.001(0.11)	<0.001(0.16)	<0.001(0.83)
Tree-sitter	TreeLSTM	AST vs Binary Tree	<0.001(0.96)	<0.001(-0.02)	<0.001(-0.03)	0.002(-0.02)	<0.001(-0.1)	<0.001(-0.07)
		AST vs Split AST	<0.001(0.42)	0.013(-0.01)	0.005(-0.01)	0.343(0.00)	<0.001(-0.04)	<0.001(-0.14)
		Binary Tree vs Split AST	<0.001(0.44)	<0.001(0.01)	<0.001(0.02)	<0.001(0.02)	<0.001(0.06)	<0.001(-0.08)
ANTLR	BiLSTM	BFS vs SBT	<0.001(1.1)	<0.001(-0.2)	<0.001(-0.33)	<0.001(-0.3)	<0.001(0.08)	<0.001(0.86)
		BFS vs AST Path	<0.001(0.75)	<0.001(-0.04)	<0.001(-0.2)	<0.001(0.01)	<0.001(0.87)	<0.001(0.86)
		SBT vs AST Path	<0.001(0.68)	<0.001(0.23)	<0.001(0.14)	<0.001(0.34)	<0.001(0.82)	<0.001(0.24)
	TreeLSTM	AST vs Binary Tree	<0.001(1.17)	0.002(-0.01)	<0.001(-0.06)	<0.001(0.04)	0.008(-0.02)	<0.001(0.81)
		AST vs Split AST	<0.001(0.26)	<0.001(0.03)	<0.001(0.04)	0.024(0.02)	0.011(-0.02)	<0.001(0.68)
		Binary Tree vs Split AST	<0.001(0.23)	<0.001(0.05)	<0.001(0.09)	<0.001(-0.01)	0.971(0.00)	<0.001(-0.15)

✎ **Summary ▸** AST preprocessing effectiveness is highly contingent on both the target task and encoder architecture. For code clone detection and search requiring structural comprehension, information-preserving methods (SBT for sequences, Raw AST for trees) perform best. For summarization, BiLSTM favors a complete AST context (SBT), whereas TreeLSTM benefits from structural simplification (Binary Tree). AST scale significantly influences outcomes, with larger ASTs (e.g., from ANTLR) particularly suited to structural compression techniques. Method selection should therefore holistically consider task objectives, model design, and AST properties. ◂

4.3 Answer to RQ3: How do AST encoding methods affect the performance of AST-based code representation on subsequent code-related tasks?

In this section, we conduct experiments to investigate the impact of AST encoding methods on AST-based code representation and follow-up code-related tasks. Specifically, we investigate two types of models, i.e., sequence models and tree-structured models. Both of them are widely used to encode AST in existing code-related studies. For sequence models, we compare two classic models, BiLSTM and Transformer. Such models are not inherently designed to process tree-structured data. Therefore, AST must first be transformed into a sequential format to be fed into these models. For tree-structured models, we compare two representative models, i.e., TreeLSTM and AST-Trans. Such models are specifically designed to handle tree-structured data and can more effectively capture the structural information of AST. From the perspective of model architecture, BiLSTM and TreeLSTM are built on the LSTM architecture, while Transformer and AST-Trans are built on the Transformer architecture. Therefore, we analyze the experimental results from two perspectives: (1) between sequence models and tree-structured models, which one is more effective in modeling AST; and (2) between LSTM and Transformer, which architecture is more suitable for building an AST-based code representation model. For a fair comparison, we uniformly employ JDT to generate ASTs for the BigCloneBench and CodeSearchNet datasets, since JDT achieves the best performance, detailed in Section 4.1. For BiLSTM and Transformer, we convert AST into SBT to adapt to the models. For TreeLSTM and AST-Trans, we feed the Raw AST into the models.

Table 8 shows the overall performance of models built on the four AST encoding methods on three code-related tasks. Table 9 shows the statistical test results. It is observed that compared with the models built on BiLSTM, the models built on Transformer overall perform better on all three tasks. Besides, the model built on TreeLSTM outperforms that built on AST-Trans on the code clone detection tasks, but the reverse is true on the code

Just Accepted

Table 8. Overall performance of models built on four different AST encoding methods

Parsing	Preprocessing	Encoding	Code Clone Detection			Code Summarization				Code Search			
			Recall	Precision	F_1	BLEU	METEOR	ROUGE-L	BertScore	SR@1	SR@5	SR@10	MRR
JDT	SBT	BiLSTM	73.06	78.68	75.77	9.013	4.540	16.45	83.19	0.1720	0.4052	0.5276	0.2827
		Transformer	**92.57**	**91.06**	**91.81**	**15.01**	**8.558**	**29.85**	**87.16**	**0.2216**	**0.4898**	**0.6122**	**0.3490**
	Raw AST	TreeLSTM	84.25	**92.18**	**88.04**	9.779	**5.317**	18.54	83.15	0.0906	0.2076	0.2660	0.1524
		AST-Trans	**87.66**	87.96	87.81	**10.30**	4.710	**20.11**	**84.99**	**0.1241**	**0.2701**	**0.3430**	**0.2015**

Table 9. Statistical test results for RQ3. The first number is the p-value for McNemar's test / Wilcoxon signed-rank test. The second number (in the parentheses) is Odds Ratio / Cliff's delta.

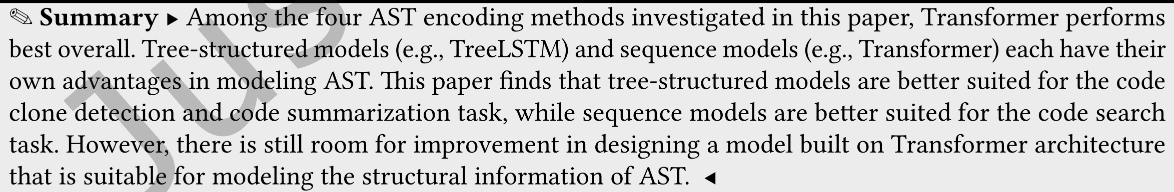
Model	Code Clone Detection	Code Summarization				Code Search
		BLEU	METEOR	ROUGE-L	BertScore	
BiLSTM vs Transformer	<0.001(2.95)	<0.001(-0.29)	<0.001(-0.38)	<0.001(-0.42)	<0.001(-0.67)	<0.001(0.31)
BiLSTM vs TreeLSTM	<0.001(2.11)	<0.001(-0.08)	<0.001(-0.1)	<0.001(-0.1)	**0.825(0.0)**	<0.001(0.87)
BiLSTM vs AST Trans	<0.001(1.98)	<0.001(-0.14)	0.002(-0.06)	<0.001(-0.17)	<0.001(-0.43)	<0.001(0.92)
Transformer vs TreeLSTM	<0.001(0.72)	<0.001(0.23)	<0.001(0.31)	<0.001(0.35)	<0.001(0.68)	<0.001(0.76)
Transformer vs AST Trans	<0.001(0.67)	<0.001(0.2)	<0.001(0.38)	<0.001(0.31)	<0.001(0.41)	<0.001(0.83)
TreeLSTM vs AST Trans	<0.001(0.94)	<0.001(-0.05)	<0.001(0.06)	<0.001(-0.07)	<0.001(-0.44)	<0.001(0.15)

Just Accepted

summarization and code search tasks. Combining the above observations, we can conclude that the Transformer architecture is more advantageous than the LSTM architecture in building the AST-based code representation model.

Comparing the performance of BiLSTM and TreeLSTM, it is observed from Table 8 that the models built on TreeLSTM achieve better performance in the code clone detection and code summarization tasks, while the model built on BiLSTM achieves better performance in the code search task. It indicates that tree-structured models are better suited for the code clone detection and code summarization tasks, while sequence models are better suited for the code search task. However, a similar phenomenon does not appear in the comparison between Transformer and AST-Trans, as shown in Table 8, where Transformer outperforms AST-Trans in all metrics. Unlike Transformer, AST-Trans takes the relationship matrix as input in addition to SBT, and uses tree-structured attention to model the structure information of AST. Nevertheless, we believe that there is still room for improvement in enhancing the Transformer architecture to better model AST structures.

The statistical test results shown in Table 9 demonstrate a significant difference between encoding models, except for BiLSTM and TreeLSTM on BertScore, which does not threaten our conclusion.



✎ **Summary** ▸ Among the four AST encoding methods investigated in this paper, Transformer performs best overall. Tree-structured models (e.g., TreeLSTM) and sequence models (e.g., Transformer) each have their own advantages in modeling AST. This paper finds that tree-structured models are better suited for the code clone detection and code summarization task, while sequence models are better suited for the code search task. However, there is still room for improvement in designing a model built on Transformer architecture that is suitable for modeling the structural information of AST. ◂

4.4 Answer to RQ4: Does AST improve the expressiveness of code representation and facilitate subsequent code-related tasks?

Previous works [28, 69, 79, 99] hold on to the point that AST contains syntactic knowledge, which contributes more to modeling source code than token information, but there is no solid evidence. To figure out whether AST can improve the effectiveness of code representation, we conduct a comparative analysis of models trained with four distinct types of inputs, including Token, SBT, SBT w/o Token, and Token + SBT w/o Token, on three

Table 10. Overall performance of models trained with four types of inputs on three code-related tasks

Model	Input	Code Clone Detection			Code Summarization				Code Search			
		Recall	Precision	F_1	BLEU	METEOR	ROUGE-L	BertScore	SR@1	SR@5	SR@10	MRR
Transformer	Token	**93.80**	93.15	**93.74**	**15.14**	**8.975**	**30.75**	**87.19**	**0.3615**	0.6122	0.6909	0.4774
	SBT	92.57	91.06	91.81	15.01	8.558	29.85	87.16	0.2216	0.4898	0.6122	0.3490
	SBT w/o Token	68.47	58.40	63.03	11.56	5.204	22.22	85.37	0.0421	0.0146	0.0671	0.0962
	Token + SBT w/o Token	91.39	**93.75**	92.55	14.72	8.658	29.87	87.06	0.3586	**0.6327**	**0.7114**	**0.4842**

code-related tasks, i.e., code clone detection, code search, and code summarization. The specifics of the four inputs are outlined below:

Token: Use the complete token sequence of the code snippet as input.

SBT: Use SBT sequences as input. Here, code snippets are parsed into AST using JDT. SBT sequences are generated using the AST preprocessing method proposed by Hu et al. [37], as detailed in Section 3.3.

SBT w/o Token: The difference between SBT and SBT w/o Token is that we change every leaf node label to <mask> in SBT w/o Token. Intuitively, we can regard SBT w/o Token as only retaining the structural information of the AST. Fig. 3 shows an example of Token, SBT, and SBT w/o Token.

Token + SBT w/o Token: We feed the model with Token and SBT w/o Token, and let it learn the embedding vectors for these two inputs, respectively. Then, we concatenate these two vectors on the second dimension and use a linear layer to compress the concatenated vector to get the final embedding vector, which is the same size as the Token embedding vector and SBT w/o Token embedding vector. In fact, SBT and Token + SBT w/o Token can be regarded as two combinations of token information and syntactic information.

Code
`int a=1;`
Token
['int', 'a', '=', '1', ';']
SBT
(VariableDeclarationStatement(PrimitiveType(int)int)PrimitiveType(VariableDeclarationFragment(SimpleName(a)a)SimpleName(NumberLiteral(1)1)NumberLiteral)VariableDeclarationFragment)VariableDeclarationStatement
SBT w/o Token
(VariableDeclarationStatement(PrimitiveType(<mask>)<mask>)PrimitiveType(VariableDeclarationFragment(SimpleName(<mask>)<mask>)SimpleName(NumberLiteral(<mask>)<mask>)NumberLiteral)VariableDeclarationFragment)VariableDeclarationStatement

Fig. 3. An example of Token, SBT, and SBT w/o Token.

Just Accepted

In this RQ, we uniformly use JDT as the AST parsing method. This is because JDT is the best-performing AST parser in RQ1, and Transformer is the best-performing encoding model in RQ3.

Table 10 presents the overall performance of Transformer-based models trained with four types of inputs on the three code-related tasks. Table 11 shows the statistical test results. The experimental results of Token are significantly different from those of others according to Table 11. The exception is that for the code summarization task, Token and SBT achieve results with no significant difference in BertScore. Additionally, on the code search task, the experimental results of Token and Token + SBT w/o Token show no significant difference. Overall, the

Table 11. Statistical test results for RQ4. The first number is the p-value for McNemar's test / Wilcoxon signed-rank test. The second number (in the parentheses) is Odds Ratio / Cliff's delta.

Model	Input	Code Clone Detection	Code Summarization				Code Search
			BLEU	METEOR	ROUGE-L	BertScore	
Transformer	Token vs SBT	<0.001(0.79)	<0.001(0.02)	<0.001(0.04)	<0.001(0.03)	**0.164(0.0)**	<0.001(0.32)
	Token vs SBT w/o Token	<0.001(0.15)	<0.001(0.18)	<0.001(0.38)	<0.001(0.28)	<0.001(0.34)	<0.001(0.73)
	Token vs Token + SBT w/o Token	<0.001(0.89)	<0.001(0.02)	<0.001(0.03)	<0.001(0.02)	<0.001(0.02)	**0.674(-0.0)**
	SBT vs SBT w/o Token	<0.001(0.19)	<0.001(0.15)	<0.001(0.34)	<0.001(0.24)	<0.001(0.34)	<0.001(0.49)
	SBT vs Token + SBT w/o Token	<0.001(1.13)	**0.298(0.0)**	**0.194(-0.01)**	**0.169(-0.01)**	<0.001(0.02)	<0.001(-0.32)
	SBT w/o Token vs Token + SBT w/o Token	<0.001(6.01)	<0.001(-0.16)	<0.001(-0.36)	<0.001(-0.27)	<0.001(-0.33)	<0.001(-0.74)

models trained with Token perform significantly better on all three tasks. Besides, models trained with SBT w/o Token significantly perform worse than models trained with the other three types of input. Therefore, we can come to the conclusion that lexical information (Token) plays a major role in AST-based code representation and solving code-related tasks.

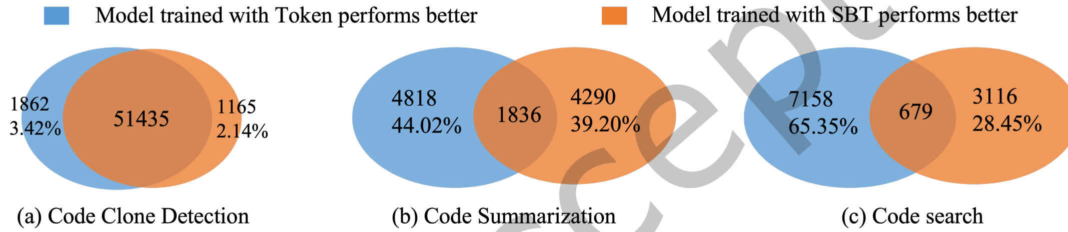


Fig. 4. Venn diagram of samples on which models trained with Token/SBT perform better.

Tufano et al. [79] suggest that different code features (e.g., Token, AST, CFG, and DFG) can provide an orthogonal view of the same code snippet. To figure out under what circumstances SBT (or syntactic information) can perform better than Token, we further drill down to analyze samples where models trained with only Token or SBT perform better. Fig. 4 shows the Venn diagram of samples on which models trained with Token/SBT perform better on code clone detection, code summarization, and code search. The percentage of the test samples might benefit from the AST-based code representation is 2.14%, 28.45%, and 39.20% on code clone, code summarization, and code search, respectively. The non-overlapping parts of the Venn diagram indicate that Token is more effective in some cases, while SBT is more effective in other cases. If we can find a way to distinguish these two types of cases and combine token-based and AST-based code representation, we may be able to improve the effect of code representation.

Hence, we undertake a more detailed excavation of the characteristics of the code within samples where SBT-based models perform better. **For the code clone detection task**, intuitively, the model trained with Token should exhibit greater accuracy in identifying true clone pairs with similar tokens. In this paper, we leverage the Jaccard index [60] which is widely used in code clone detection works [40, 44, 47] to compute the similarity between tokens of two code snippets in each clone pair. The Jaccard index $S_{Jaccard}$ between two code snippets can be computed as follows:

$$S_{Jaccard} = JaccardIndex(code_1, code_2) = \frac{|Tok(code_1) \cap Tok(code_2)|}{|Tok(code_1) \cup Tok(code_2)|} \quad (11)$$

Just Accepted

Table 12. Number of samples that Token/SBT is better counted by different $S_{Jaccard}$ intervals.

Input	≤ 0.1	(0.10, 0.15]	(0.15, 0.20]	(0.20, 0.25]	(0.25, 0.30]	(0.30, 0.35]	>0.35	Average Score
Token	33	363	627	429	282	94	34	0.2021
SBT	84	553	154	145	136	59	34	0.1763
Ratio	71.79%	60.37%	19.72%	25.26%	32.54%	38.56%	50%	/

Table 13. Number of samples that Token/SBT is better counted by different e intervals

Input	≤0.05	(0.05, 0.10]	(0.10, 0.15]	(0.15, 0.20]	>0.20	Average Score
Token	3110	1061	429	153	65	0.0473
SBT	2822	922	363	124	59	0.0453
Ratio	47.57%	46.5%	45.83%	44.77%	47.58%	/

Table 14. Number of samples that Token/SBT is better counted by different r intervals

Input	≤ 0.1	(0.1, 0.12]	(0.12, 0.14]	(0.14, 0.16]	>0.16	Average Score
Token	6209	298	212	181	258	0.046
SBT	2678	133	93	66	146	0.050
Ratio	30.13%	30.86%	30.49%	26.72%	36.14%	/

where $Tok(\cdot)$ is a function used to tokenize code snippets; $|\cdot|$ represents the size of the set. As for the tokenizer, we use the pre-trained tokenizer provided by CodeBERT [27].

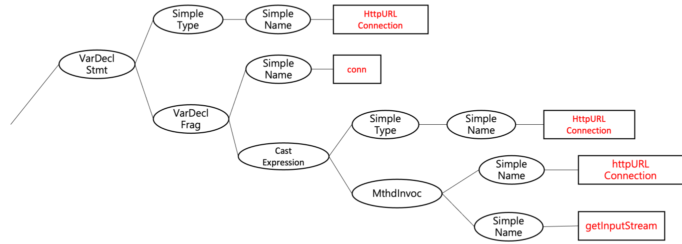
Table 12 divides the true clone pairs in the test dataset into 7 intervals according to the Jaccard index, and counts the number of clone pairs only detected by the model trained with Token and the number of clone pairs only detected by the model trained with SBT. *Ratio* in Table 12, Table 13 and Table 14 refers to the proportion of samples on which SBT performs better to the total samples, that is to say $Ratio = NUM(SBT)/(NUM(SBT) + NUM(Token))$, where $NUM(Token)$ and $NUM(SBT)$ refers to the number of samples on which models trained with Token/SBT perform better respectively. No matter which interval, some clone pairs are only correctly detected by the model trained with Token/SBT. When the Jaccard index is less than 0.15, SBT detects more clone pairs. It reveals that, when the similarity between the tokens of two code snippets in a clone pair is low, AST-based code representation is a better alternative than token-based code representation. As the Jaccard index increases, *Ratio* first decreases and then increases. It indicates that when the textual similarity between code and code is relatively high, Token is a more effective way to represent code. However, when the textual similarity reaches a certain level, the effectiveness of SBT also increases. This is because the leaf nodes in AST also tend to be similar, not because of the syntactic information in the AST. Considering the complexity difference between using SBT and using Token, as well as the input length limit of DL models, it is recommended to use Token as model input when the Jaccard similarity is high.

Fig. 5 visually shows an example from BigCloneBench of a true cloned code pair $\langle s_3, s_4 \rangle$ which is only detected by Transformer-based models trained with Token. Due to the page limit, we only display the ASTs of partial code statements highlighted in red font. Fig. 5(b) showcases the AST of the 4-*th* statement in s_3. Fig. 5(d) showcases the AST of the 3-*th* and 5-*th* statements in s_4. It is observed that although the code statements highlighted in red font in s_3 and s_4 share many similar tokens and implement the same semantics, that is, using the *url* to create an

Just Accepted

idx	4562786
1	`private String GetResponse(URL url) {`
2	`    String content = null;`
3	`        try {`
4	`            HttpURLConnection conn = (HttpURLConnection) url.openConnection();`
5	`            conn.setDoOutput(false);`
6	`            conn.setRequestMethod("GET");`
7	`            ......`
8	`        } catch (IOException e) {`
9	`            e.getStackTrace();`
10	`        }`
11	`        return content;`
12	`}`

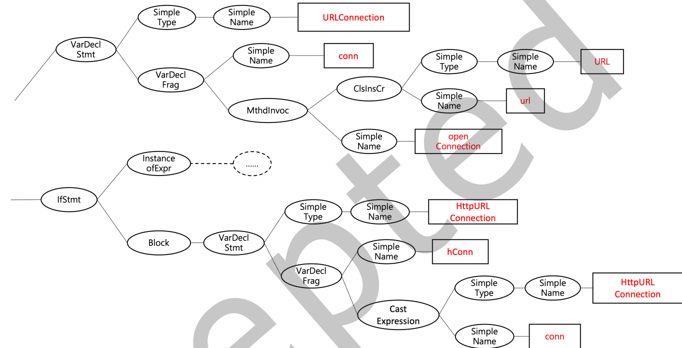
(a) Code Snippet s_3



(b) AST of the 4-*th* statement in s_3

idx	5989666
1	`private String postXml(String url, String soapAction, String xml) {`
2	`    try {`
3	`        URLConnection conn = new URL(url).openConnection();`
4	`        if (conn instanceof HttpURLConnection) {`
5	`            HttpURLConnection hConn = (HttpURLConnection) conn;`
6	`            hConn.setRequestMethod("POST");`
7	`        }`
8	`        ......`
9	`    } catch (IOException e) {`
10	`        throw new RuntimeException(e);`
11	`    }`
12	`}`

(c) Code Snippet s_4



(d) AST of the 3-*th* and 5-*th* statements in s_4

Fig. 5. An example of a clone pair detected by the Transformer-based model trained with Token.

HTTP connection, their ASTs are significantly different. In this case, feeding syntax information into the DL model is not conducive to identifying clone pairs.

Fig. 6 shows an example where syntactic information is conducive to code clone detection. The example is also a true clone pair from BigCloneBench, which is only detected by Transformer-based models trained with SBT. Contrary to Fig. 5, the code statements highlighted in red font in s_5 and s_6 share a few similar tokens, but their ASTs are very similar. In this case, syntactic information helps Transformer identify this clone pair.

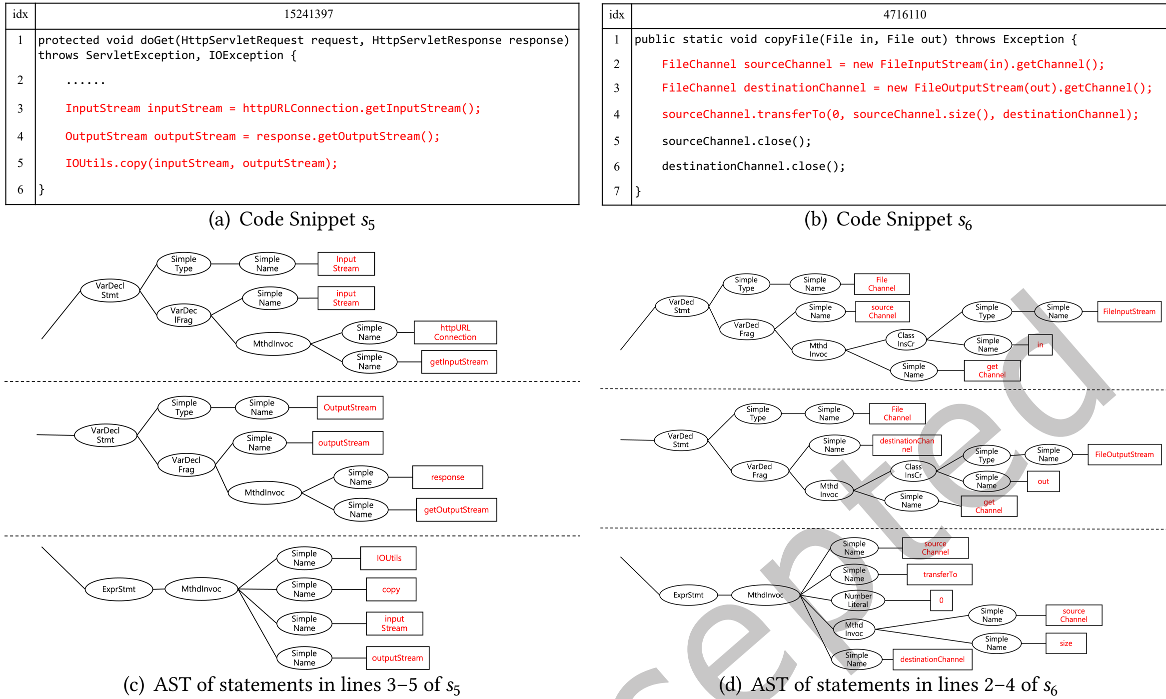
For the code summarization task, Intuitively, code summarization is easier if most of the words in the summary can be found in the code. In this case, it may be sufficient to train the model with only the Token. Therefore, we measure the ease of code summarization by the proportion of words in the summary that appear in the code snippet, which is represented by *e*. It can be regarded as a conditional probability distribution in set theory, which is the proportion of the intersection of two sets A and B relative to the size of either set A or B. In our setting, A and B are summaries and the corresponding code snippets. Therefore, e can be computed as follows:

$$e = P(code|summary) = \frac{|Tok(summary) \cap Tok(code)|}{|Tok(summary)|} \tag{12}$$

where $Tok(\cdot)$ is a function used to tokenize summaries and code snippets; $|\cdot|$ represents the size of the set. As for the tokenizer, we use the pre-trained tokenizer provided by CodeBERT[27].

We count the number of samples on which models trained with Token/SBT obtain better BLEU scores in different *e* intervals, which is shown in Table 13. *Ratio* is defined similarly as Table 12. From Table 13, it is observed that as *e* increases, *Ratio* tends to first decrease, then increase. The conclusion is similar to that in the code clone

Just Accepted



(a) Code Snippet s_5

(b) Code Snippet s_6

(c) AST of statements in lines 3–5 of s_5

(d) AST of statements in lines 2–4 of s_6

Fig. 6. An example of a clone pair detected by the Transformer-based model trained with SBT.

detection task; syntactic information embedded in AST can facilitate code representation and subsequent code summarization tasks when the similarity between code and summary is low. When e is greater than 0.2, the performance of SBT also increases. We believe it is attributed to the overlap between leaf nodes and summaries, rather than the structural information provided by AST. Therefore, when the similarity between the code and the summary is high, the better code representation method is still Token, considering the usage complexity and model input length limit. In other words, if you want to generate a summary that contains fewer tokens in the code, the AST-based code representation can be taken into account.

Fig. 7 illustrates two code summarization cases, where two code snippets s_7 and s_8 in (a) and (d) are from the CodeSearchNet dataset. From Fig. 7(a) and (b), it is observed that compared with the summary generated by the model trained with SBT, the summary generated by the model trained with Token has a higher BLEU value of 1.0 and is closer to the reference summary. The e between s_7 and its reference summary we compute is 0.8571. From Fig. 7(c) and (d), it can be seen that the summary generated by the model trained with SBT has a BLEU value of 0.4416 and is better than that generated by the model trained with Token (0.1674). The e between s_8 and its reference summary we compute is only 0.2. These two examples intuitively illustrate that SBT is suitable for situations where the code snippet itself lacks tokens that can be used to constitute a summary, such as when the tokens (especially the developer-defined identifiers) in the code snippet are not informative. The developer's expectation that the model should not rely on the identifiers within the code snippet when generating summaries can be regarded as one such case.

For the code search task, intuitively, the results of the code search are more relevant if most of the words in the user queries can be found in the code. In this case, it may be sufficient to train the code search model directly

Just Accepted

```
1  private String addTitleToHtmlFile(String html, String title){
2      if (html == null) {return html;}
3      if (title != null) {
4          getLog().debug("Setting the title in the HTML file
               to: " + title);
5          return html.replaceFirst("titleToken", title);
6      } else {
7          getLog().debug("Title was null, setting the title
               in the HTML file to an empty string");
8          return html.replaceFirst("titleToken", "");
9      }
12 }
```

(a) A Java code snippet s_7

```
1  public <T> T getValue(final String key) {
2      T val = (T) map.get(key);
3      if (val instanceof Map) {
4         return (T) new JsonObject((Map) val);
5      }
6      if (val instanceof List) {
7          return (T) new JsonArray((List) val);
8      }
9      return val;
10 }
```

(c) A Java code snippet s_8

Reference: Adds the title to the html file.	e: 0.8571
Token: Adds the title to the HTML file.	BLEU: 1.0
SBT: Adds the HTML.	BLEU: 0.2671

(b) Summaries generated by models trained with Token/SBT for s_7

Reference: Returns the value with the specified key as an object.	e: 0.2
Token: Gets the value of the JSON property.	BLEU: 0.1674
SBT: Returns the value associated with the specified key.	BLEU: 0.4416

(d) Summaries generated by models trained with Token/SBT for s_8

Fig. 7. Case of SBT having a negative/positive effect on code summarization task.

with Token. Therefore, similar to the calculation of e, we measure the relevance of the query and the code by the proportion of words in the query that appear in the code snippet. The relevance of code search represented by r can be computed as follows:

$$r = P(code|query) = \frac{|Tok(query) \cap Tok(code)|}{|Tok(query)|} \quad (13)$$

where $Tok(\cdot)$ is a function used to tokenize queries and code snippets; $|\cdot|$ represents the size of the token set. As for the tokenizer, we also use the pre-trained tokenizer provided by CodeBERT [27].

We count the number of samples on which the ground-truth results (i.e., code snippets) are ranked higher by models trained with Token/SBT in different r intervals. Table 14 shows the results *Ratio* is defined similarly as in Table 12. From Table 14, it is observed that *Ratio* is bigger when r is quite low or quite high, which is similar to Table 12 and Table 13. Hence, our conclusion is the same as that of the code clone detection and code summarization tasks. When the textual similarity between the query and the code is low, which means the developer expects the matching code to contain fewer tokens in the query, the structural information in the AST has the potential to improve code representation. In other cases, token-based code representation is more recommended.

Fig. 8 illustrates two code search cases, where queries q_1 and q_2 in (a) and (d) are comments from the CodeSearchNet dataset. Fig. 8(b) and (e) shows two code snippets s_9 and s_{10} that are ground-truth results of q_1 and q_2, respectively. Fig. 8(c) and (f) show the rankings of s_9 and s_{10} in corresponding results retrieved for q_1 and q_2 respectively by the models trained with Token/SBT. For example, "Rank (Token as input): 1" in Fig. 8(c) indicates that s_9 ranks first in the results retrieved by the model trained with Token for query q_1, while it ranks 6th in the results retrieved by the model trained with SBT (denoted "Rank (SBT as input): 6" in Fig. 8(c)). Clearly, in this case, the model trained with Token outperforms the one trained with SBT. The r we calculate between s_9 and q_1 has a large value of 0.6333. From Fig. 8(e) and (f), it can be seen that the rankings of s_{10} in the results retrieved for q_2 by the models trained with Token and SBT are 105 and 23, respectively. It indicates that in this case, the model trained with SBT outperforms the one trained with Token. The r between s_{10} and q_2 we compute is only 0.0556. These two examples intuitively illustrate that SBT is suitable for situations where the code snippet itself may not

Just Accepted

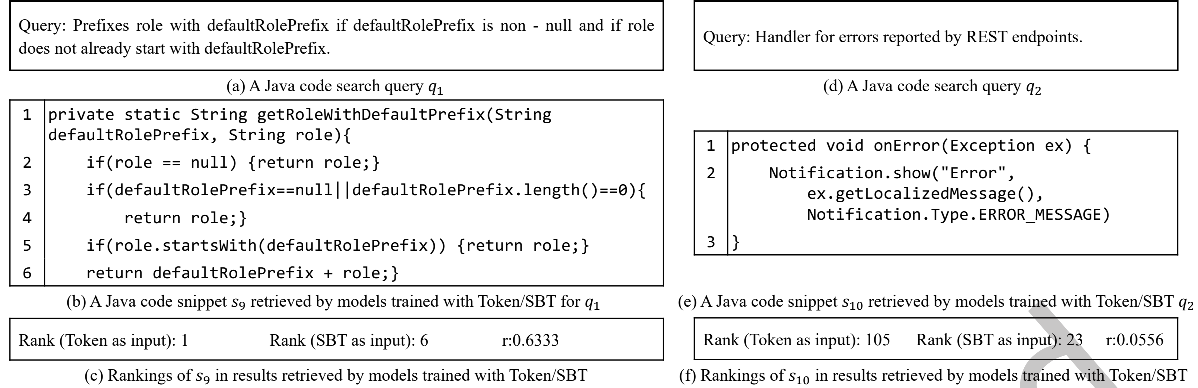


Fig. 8. Case of SBT having a negative/positive effect on code search task.

contain many tokens that may appear in the user queries, such as when the identifiers in the code snippet are not informative, or the user's query is quite abstract and does not contain words that may appear in the code snippet.

These findings further provide practical guidance for developers and researchers on how to select and utilize appropriate code representations in real-world scenarios. Although token-based representations generally achieve better overall performance, syntactic information encoded in ASTs can still offer complementary benefits under specific conditions. Therefore, instead of treating token and AST representations as competing alternatives, practitioners should consider how to adaptively choose or combine them based on the characteristics of their specific tasks.

For code clone detection and other similarity-based tasks, developers can directly compute the Jaccard similarity between two code snippets before model training or inference. This value serves as a concrete and accessible indicator for representation selection. When the Jaccard similarity is relatively high (e.g., above 0.15), token-level overlap between snippets is already strong enough to capture their equivalence, making token-based representations the recommended choice. Conversely, when the Jaccard similarity is low, but the code snippets are expected to be functionally related, AST-based representations can better capture the underlying structural and semantic similarities.

In code search tasks, although developers cannot directly measure variables such as e/r in advance, the experimental findings still provide useful guidance for practical decision-making. When the goal is to retrieve code snippets that share high lexical overlap with natural language queries, such as when query terms resemble function or variable names, token-based models should be preferred, as they preserve detailed lexical information and improve keyword-level matching. However, if the developer aims to find semantically relevant code that may use different identifiers or abstractions, AST-based models can leverage syntactic structure and semantic relations to enhance retrieval performance.

A similar principle applies to code summarization. When the desired summaries should follow the lexical and naming patterns present in the source code, token-based representations are more effective. In contrast, when the goal is to generate more abstract or semantically rich summaries, particularly for code with poor naming conventions or limited comments, AST-based representations can better capture the underlying structure and functional intent of the code.

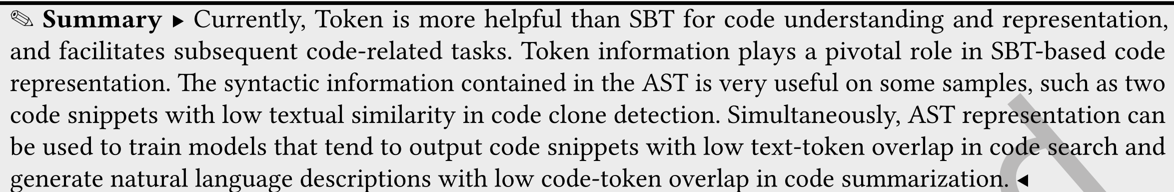
In summary, these results offer actionable guidance for developers:

- using token-based representations when the lexical similarity between code and its counterpart (another snippet, a query, or a summary) is high;

Just Accepted

- preferring AST-based representations when lexical similarity is low but semantic correspondence is important.

For researchers, these observations suggest a promising direction toward hybrid or dynamic code representation approaches that can automatically adjust between token-level and structure-level encoding according to the characteristics of each task instance.


✎ **Summary** ▸ Currently, Token is more helpful than SBT for code understanding and representation, and facilitates subsequent code-related tasks. Token information plays a pivotal role in SBT-based code representation. The syntactic information contained in the AST is very useful on some samples, such as two code snippets with low textual similarity in code clone detection. Simultaneously, AST representation can be used to train models that tend to output code snippets with low text-token overlap in code search and generate natural language descriptions with low code-token overlap in code summarization. ◂

5 Threats to Validity

5.1 Threats to Internal Validity

The threats to internal validity stem from potential inaccuracies in implementing AST preprocessing and encoding methods. While we cannot ensure complete correctness or identical performance to the original claims, we have taken measures to minimize these threats.

Specifically, in terms of AST preprocessing methods, we implement the algorithms to convert source code into SBT [37], AST Path [5], and Binary Tree [91] according to the algorithms described in the corresponding paper. This is necessary because either the original paper does not provide code or the code provided in the original paper is limited to a specific AST parsing tool. As BFS is a widely used and simple algorithm, we implement the algorithm to convert source code into BFS on ourselves as well. As for split AST, there are several preprocessing methods to split the complete AST into smaller components, such as ASTNN [99], CAST [68] and BASTS [52]. The split AST of ASTNN consists of AST nodes of one statement, which is considered to ignore the long-term dependencies of multiple statements and has been proven to be less effective than the split AST of CAST [68] and BASTS [52]. Between CAST and BASTS, considering the need to not be limited to a specific AST parser, we opt for the split AST preprocessing method of BASTS, which is proposed by Lin et al. [52]. We use the implementation code provided by Lin et al. to process source code into split code, and then generate ASTs for the split code using an AST parsing tool.

In terms of AST encoding models, we call the interface provided by PyTorch to implement BiLSTM and Transformer. TreeLSTM is modified based on the code provided by Lin et al. [52]. The original code is implemented using TensorFlow, and we convert it into a PyTorch-based implementation. As to AST-Trans [78], we reuse the model part and data processing part of the code provided in the original paper, and slightly modify the code to adapt to the code framework of CodeXGLUE. We have published the specific implementation code for other researchers to check and use [75].

5.2 Threats to External Validity

The threats to external validity lie in the generalizability of our findings. To reduce these threats, we considered three popular types of code-related tasks, including a code-to-code matching task, i.e., code clone detection, a text-to-code matching task, i.e., code search, and a code-to-text generating task, i.e., code summarization.

Another potential threat to external validity is the variety of AST parsing/preprocessing/encoding methods we investigated, which may affect the generalization of our findings to other methods. To reduce this threat, we consider multiple representative AST parsing/preprocessing/encoding methods in our experiments, which

Just Accepted

are commonly used for various code-related tasks. In the future, we will conduct experiments on more AST parsing/preprocessing/encoding methods to generalize our conclusions.

The selection of the experimental dataset and programming language also poses a threat to external validity. To mitigate this threat, we select two classic and representative datasets, i.e., BigCloneBench and CodeSearchNet, as our experimental datasets. Both of them are widely used to evaluate the effectiveness of code models in the field of software engineering. In addition, we mainly focus on the Java programming language in this paper for the following reasons: (1) Java is one of the most popular and well-studied programming languages [4]; (2) existing AST parsing tools vary in language support, but almost all of them can reliably generate ASTs for Java code, ensuring consistency and comparability across different parsing tools. In summary, investigating the influence of ASTs on Java code representation learning has a potential impact on real-world software development. We acknowledge that programming language characteristics (e.g., syntax regularity and typing system) and dataset sample distributions may influence the effect of AST-based representations. Therefore, the findings, such as τ value for Jaccard, reported in this paper may not directly generalize to other programming languages such as Python or C/C++. We consider this as an explicit threat to external validity and leave cross-language and cross-dataset validation as an important direction for future work.

Another concern regarding the clone detection task is the potential impact of different clone types on the results. In BigCloneBench, clone pairs are categorized into five types: Type-1 (T1), Type-2 (T2), Strongly Type-3 (ST3), Moderately Type-3 (MT3), and Weakly Type-3/Type-4 (WT3/T4) [87]. Among these, WT3/T4 clone pairs constitute the vast majority (98.23%) of the dataset. Our analysis reveals that for T1, T2, ST3, and MT3 clone types, both Token-based and SBT-based BiLSTM models achieve correct detection in most cases, indicating that detecting these types is relatively straightforward and the input representation (Token vs. SBT) has negligible impact. However, for the predominant WT3/T4 clone type, which represents the most challenging and prevalent scenario, we observe significant performance differences between Token-trained and SBT-trained models. Therefore, the experimental conclusions for RQ4 primarily focus on WT3/T4 clones. While aggregating results across all clone types could oversimplify the task and be influenced by type imbalance, our focused approach on WT3/T4 ensures that the findings are representative of the most critical and common clone detection scenario in the dataset.

5.3 Threats to Construct Validity

The threats to construct validity mainly lie in the evaluation metrics we use in our work. In order to ensure the effectiveness of the evaluation metrics, we consider the metrics that are widely used on three code-related tasks. Additionally, to avoid implementation errors, we use public libraries or reuse the implementation code provided by previous researchers to calculate these metrics. For the code clone detection task, we use the interface provided by Scikit-learn to calculate the precision, recall, and F_1-score. For the code summarization task, we calculate BLEU using the code provided in the CodeXGLUE [56]. We calculate METEOR and ROUGE-L using the code provided by Chen et al. [16]. For the code search task, we calculate MRR using the code provided in the CodeXGLUE code framework. We calculate SR@k ($k = 1, 5, 10$) using the code provided by Sun et al. [73].

6 Related Work

6.1 Code Features

Code features refer to a standardized data format that is derived from the source code. Code features usually describe certain aspects of the source code, e.g., the lexical aspect, syntactic aspect, semantic aspect, etc. They are designed to facilitate the processing and analysis of the source code and can be used for a variety of code-related tasks, such as code clone detection [25, 92], code search [30, 64, 73], code summarization [9, 37, 74], etc. The choice of code features is crucial and non-trivial as it requires taking into account the available DL model and the

[4]https://www.tiobe.com/tiobe-index/

Just Accepted

characteristics of the code-related tasks that need to be solved [65]. Due to differences in model architectures, different DL models accept different formats of inputs, and their learning capabilities vary with different code features. Furthermore, different code-related tasks focus on different aspects of the source code.

Code features that are widely used in code-related tasks can be divided into three categories: token-based features, tree-based features, and graph-based features, as detailed below.

Token-based features. Token-based features extract partial or complete tokens from the source code as code features. Tokens are bags of words that are parsed from the source code. Commonly used partial tokens are method names [32, 64, 95], API sequences [14, 24, 32, 39]. Compared with the complete token, partial tokens include user-defined identifiers, are simpler, and are easier for the model to learn. But it should be noted that partial tokens lose contextual syntax and semantics, while complete tokens contain complete code semantics. Therefore, many researchers are prone to use the complete code token sequence [15, 30, 41, 95]. They usually refer to the complete code token sequence directly as "Token". In this paper, we also follow this habit. Since Token contains rich lexical information of the source code, they are frequently used by researchers to solve code-related tasks [15, 26, 30, 32, 34, 41, 95].

Tree-based features. Tree-based features mainly refer to parsing the syntax tree of the source code and using it as code features. The syntax tree can be further divided into two categories: concrete syntax tree (CST) and abstract syntax tree (AST). A CST is an ordered, rooted tree that represents the syntactic structure of the source code according to some context-free grammar [93]. Some studies use CST as code features to learn code representation and assist in solving code-related tasks [63, 88, 97]. For example, Peng et al. [63] propose to learn code representation from CST and apply learned code representation to the code summarization task. Ye et al. [97] propose a context-aware semantic structure called CASS, which is built on CST. They use CASS to learn code representation and solve the code similarity analysis task.

An AST is a simplified representation of a CST and focuses on the essential elements of the program's structure [2, 93]. Compared to CST, AST provides a more concise and structured representation of the program's semantics, making it easier to analyze and manipulate. Currently, AST is one of the most commonly used code features in code representation learning [65, 71]. AST has been widely used to solve many code-related tasks, such as code classification [86, 99], code clone detection [10, 13, 99], code search [31, 34, 83, 95], code summarization [52, 68], code property (e.g., method and variable names) prediction [5], etc. In many cases, ASTs are preprocessed before being fed to the DL model, to adapt to the model input format or further emphasize certain features in the AST. For example, some existing works traverse an AST using BFS or DFS to get the list of all the AST nodes [11, 67, 95]. Many techniques that chose to convert AST into SBT [37, 37, 38, 50, 90, 96]. Some techniques [4, 6, 12] use a collection of paths on the tree (i.e., AST Path) to represent an AST. The purpose is to extract key information in the AST and reduce the length of the model input. There are also a lot of techniques that convert AST into simple tree structures, such as Binary Trees [41, 82, 84, 91] and Split AST [52, 68, 99], to facilitate model learning.

Graph-based features. Graph-based features refer to extracting the graph information from the source code as code features. Existing code-related works have mined multiple types of graph information to enhance code representation and solve code-related tasks. For example, the control flow graph (CFG) can represent all possible execution paths for the program and is commonly used to capture control dependencies between code elements [31, 82, 101]. The data flow graph (DFG) describes the process of how data flows and is processed. Many works [31, 82, 101] use DFG to capture the data dependencies between code elements. The program dependence graph (PDG) depicts the data dependencies and control dependencies of each operation in a program. Gu et al. [31] utilize PDG to extract code structures. In addition to the well-known graph information mentioned above, some researchers have also mined some new types of graph information. For example, Zhou et al. [102] represent various sub-graphs, including AST, CFG, DFG, and Natural Code Sequence (NCS), into one joint graph to get a code property graph (CPG). Allamanis et al. [3] enhance ASTs with different data flow information by

Just Accepted

constructing diverse types of edges among the nodes, such as connecting variable nodes where the variable is last written.

In this paper, we focus on a tree-based feature, i.e., AST. We conduct a quantitative evaluation and qualitative analysis of the extent to which AST facilitates code representations and subsequent code-related tasks.

6.2 Code Representation Learning

As mentioned in Section 2.2, code representation learning aims to convert source code features into distributed, real-valued vector representation, i.e., code representation. Such numerical representation condenses the semantics of the code, and facilitates the solution of subsequent code-related tasks. In the previous section, we have discussed in detail the first process of code representation learning (i.e., code feature extraction). Next, we mainly introduce the second process, i.e., code feature representation.

Code representation for token-based features. Token-based features are sequential data. Most works transfer sequence models from the NLP field to encode token-based features. For instance, Gu et al. [32] introduce the Recurrent Neural Networks (RNN) [58] to encode method names and API sequences. LeClair et al. [50] utilize a GRU layer to encode Token (i.e., the complete code token sequence). Wei et al. [90] first map the one-hot embedding token sequence into a word embedding sequence and then use a BiLSTM to process word embedding sequences. Cai et al. [14] and Shuai et al. [70] employ BiLSTM to encode method names and API sequences, respectively.

There are also some works that first transform token-based features into other forms before selecting a suitable encoding model. For example, Cheng et al. [18] and Deng et al. [22] convert method name, API sequence, and Token into vector matrices, and then use the Convolutional Neural Network (CNN) [51] to encode them.

Code representation for tree-based features. As mentioned in Section 1, many works preprocess the raw AST before feeding it to the model for learning, with the purpose of simplifying the complexity of the AST. When the raw AST is converted into sequential AST data (e.g., SBT and AST Path) by AST preprocessing methods, most works encode it using a sequence model. For example, Hu et al. [37] use LSTM to encode the SBT of the corresponding AST. LeClair et al. [50] utilize a GRU layer to encode SBT-AO. SBT-AO is a modified version of SBT in which all the code structure remain intact, but all words (except official Java API class names) in the code are replaced with a special <OTHER> token. Alon et al. [4] adopt BiLSTM to encode AST Path. Bertolotti et al. [11] split the AST into sub-trees and then use the pre-order visit to get lists of non-terminal and terminal nodes in the AST. Then they use a multi-head global attention layer to encode the terminal node list and use BiLSTM to encode the non-terminal node list. Shahbazi et al. [67] flatten ASTs into sequences by tree traversal and encode the flattened ASTs with Transformer.

When the raw AST is split into simple structural data while retaining the tree structure, most works encode it using a tree-structured model. For instance, Mou et al. [59] propose a novel tree-based convolutional neural network (TBCNN), in which a set of subtree feature detectors, called the tree-based convolution kernel, slides over the entire AST to extract structural information of a program. Dynamic pooling is applied to gather information over different parts of the AST. Finally, a hidden layer and an output layer are added. Many works [41, 82, 84, 91] adopt TreeLSTM (including N-ary TreeLSTM) to encode binary trees of the corresponding AST. Zhang et al. [99] split each AST into a sequence of statement trees (ST-trees) and encode ST-trees with a Recursive Neural Network (RvNN) [72] to learn vector representations of ST-trees. Then the statement representations are fed into a Bidirectional Gated Recurrent Unit (BiGRU) to learn the representation of the code fragment. Similar to [99], Shi et al. [68] hierarchically split a large AST into a set of subtrees according to a pre-defined AST splitting rule and utilize an RvNN to encode the subtrees. Then, they aggregate the embeddings of subtrees by reconstructing the split ASTs to get the representation of the complete AST. Shido et al. [69] propose Multi-way TreeLSTM to handle ASTs with an arbitrary number of ordered children. Specifically, they add an ordinary chain-like LSTM

Just Accepted

to each gate before linear transformation to flexibly adapt to a node that has an arbitrary number of ordered children. Lin et al. [52] split the code of a method based on the blocks in the dominator tree of the control flow graph, and generate a split AST for each code split. Each split AST is then encoded by a Child-Sum TreeLSTM. Some researchers have tried to modify the architecture of the Transformer to facilitate the learning of structural information. For example, LeClair et al. [94] add additional edges to the AST to further represent control flow and data dependency, and then adopt an adjacency matrix to represent the AST. They propose a structure-induced Transformer (SiT), which contains three structure-induced self-attention network (Si-SAN) layers. Code sequences and corresponding adjacency matrices are passed into the SiT encoder to get code embeddings. Gong et al. [29] propose a StruCtural RelatIve Position guided Transformer, named SCRIPT. SCRIPT leverages ASTs to obtain the structural relative positions between tokens, which are then fed into two types of transformer encoders. One transformer encoder directly adjusts the input based on the structural relative distance, while the other transformer encodes the structural relative positions while computing the self-attention scores. Finally, these two types of transformer encoders are stacked together to learn source code representations. Tang et al. [78] denote the ancestor-descendant and sibling relationship matrices among AST nodes through two relationship matrices. They apply tree-structured attention instead of the standard self-attention to dynamically allocate weights for relevant nodes and exclude irrelevant nodes based on these two relationships.

There are also works that treat ASTs as graphs, thus encoding ASTs with graph neural networks (GNNs). For instance, LeClair et al. [49] encode the AST nodes and edges with ConvGNN and allow the nodes of the AST to learn representations based on their neighboring nodes. Zhou et al. [103] treat the AST as an undirected graph and utilize Graph Attention Networks (GAN) as the encoder. Yang et al. [96] transform the source code into ASTs and SBT sequences, and use a graph convolutional neural network (GCN) and Transformer as the encoder, respectively. These works essentially treat a tree as a special type of graph.

Code representation for graph-based features. The Graph Neural Network (GNN) and its variants, including Graph Convolutional Network (GCN), Graph Attention Network (GAT), and Gated Graph Neural Network (GGNN), are the most commonly used neural network architectures in conjunction with graph-based code features in code-related tasks. For example, Allamanis et al. [3] represent program source code as graphs and use different edge types to model syntactic and semantic relationships between different tokens. Gated Graph Neural Network (GGNN) is applied to encode the constructed program graph. Zhao et al. [101] encode the code semantics represented by control flow and data flow into a single semantic matrix, and train a specially designed feed-forward neural network to learn code representations. Zhou et al. [102] use GNN to learn representations for the CPG. Hua et al. [41] adopt the Graph Convolutional Network (GCN) [48] to encode CFGs. Liu et al. [53] construct directed graphs for programs based on ASTs. They propose a multi-head attention module to further improve the expression of bidirectional GGNN (BiGGNN) and feed the graphs into two BiGGNN encoders to learn the vector representations.

In this paper, we focus on a tree-based code representation, i.e., AST-based code representation. We conduct comprehensive experiments to explore the impact of the choice of popular AST preprocessing and encoding methods on AST-based code representation as well as subsequent code-related tasks.

7 Conclusion

In this paper, we first conduct comprehensive experiments to evaluate and reveal the impact of the choice of various AST parsing, preprocessing, and encoding methods on AST-based code representation and subsequent code-related tasks (i.e., code clone detection, code search, and code summarization). The results demonstrate that the impact of different methods at different stages varies for different code-related tasks. Then we quantitatively evaluate the effectiveness of AST-based code representation compared with Token-based code representation. The results show that the models trained with AST-based code representation consistently perform worse across

Just Accepted

all three tasks compared with the models trained with Token-based code representation. We further conduct a qualitative analysis of scenarios/cases in which AST-based code representation performs better than Token-based code representation in the three code-related tasks. We find that the models trained with AST-based code representation outperform models trained with Token-based code representation in certain subsets of samples across all three tasks, such as clone pairs that have low textual similarity. We believe that the results and findings of this paper can provide useful guidance for subsequent researchers to use AST to solve code-related tasks.

Acknowledgment

The authors would like to thank the editors and the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. The authors at Nanjing University were supported, in part by the National Natural Science Foundation of China (U24A20337 and 62372228). The authors at Nanyang Technological University are supported by the National Key Research and Development Program of China (Grant No. 2024YFF0908000).

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4998–5007.

[2] Alfred V Aho, Ravi Sethi, Jeffrey D Ullman, et al. 2007. *Compilers: Principles, Techniques, and Tools*. Vol. 2. Addison-wesley Reading, Addison Wesley.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–17.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. Code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations-Poster*. OpenReview.net, New Orleans, LA, USA, 1–13.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In *Proceedings of the 39th SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Philadelphia, PA, USA, 404–419.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (jan 2019), 40:1–40:29.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the 3th International Conference on Learning Representations*. OpenReview.net, San Diego, CA, USA, 1–15.

[8] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Association for Computational Linguistics, Ann Arbor, Michigan, USA, 65–72.

[9] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 253–264.

[10] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 6th International Conference on Software Maintenance*. IEEE Computer Society, Bethesda, Maryland, USA, 368–377.

[11] Francesco Bertolotti and Walter Cazzola. 2023. Fold2Vec: Towards a Statement-Based Representation of Code for Code Comprehension. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 6:1–6:31.

[12] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. 2021. Authorship Attribution of Source Code: A Language-agnostic Approach and Applicability in Software Engineering. In *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 932–944.

[13] Lutz Büch and Artur Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, Hangzhou, China, 95–104.

[14] Fuqi Cai, Changjing Wang, Qing Huang, Zhengkang Zuo, and Yunyan Liao. 2021. Search for Compatible Source Code. *International Journal of Software Engineering and Knowledge Engineering* 31, 3 (2021), 477–502.

[15] Qingying Chen and Minghui Zhou. 2018. A Neural Framework for Retrieval and Summarization of Source Code. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, Montpellier, France, 826–831.

Just Accepted

[16] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C. Lawrence Zitnick. 2015. Microsoft COCO Captions: Data Collection and Evaluation Server. *CoRR* abs/1504.00325, 1 (2015), 1–7.

[17] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Montréal, Canada, 2552–2562.

[18] Yi Cheng and Li Kuang. 2022. CSRS: Code Search with Relevance Matching and Semantic Matching. In *Proceedings of the 30th International Conference on Program Comprehension*. ACM, Virtual Event, 533–542.

[19] Lin Chin-Yew. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics – workshop on Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81.

[20] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 19th Conference on Empirical Methods in Natural Language Processing*. ACL, Doha, Qatar, 1724–1734.

[21] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *Proceedings of the 29th International Conference on Software Maintenance*. IEEE Computer Society, Eindhoven, The Netherlands, 516–519.

[22] Zhongyang Deng, Ling Xu, Chao Liu, Meng Yan, Zhou Xu, and Yan Lei. 2022. Fine-grained Co-Attentive Representation Learning for Semantic Code Search. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 396–407.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 23th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Minneapolis, MN, USA, 4171–4186.

[24] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning Approach for Semantic Code Search. In *Proceedings of the 30th International Conference on Information & Knowledge Management*. ACM, Queensland, Australia, 2994–2998.

[25] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th International Symposium on Software Testing and Analysis*. ACM, Virtual Event, USA, 516–527.

[26] Sen Fang, You-Shuai Tan, Tao Zhang, and Yepang Liu. 2021. Self-attention Networks for Code Search. *Information and Software Technology* 134 (2021), 106542.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*. Association for Computational Linguistics, Online Event, 1536–1547.

[28] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-modal Approach based on Transformer for Source Code Summarization. In *Proceedings of the 30th International Conference on Program Comprehension*. ACM, Virtual Event, 24–35.

[29] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 13–24.

[30] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution*. IEEE, Luxembourg, 483–494.

[31] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. CRaDLe: Deep Code Retrieval Based on Semantic Dependency Learning. *Neural Networks* 141 (2021), 385–394.

[32] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 933–944.

[33] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225.

[34] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A Multi-Perspective Architecture for Semantic Code Search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 8563–8568.

[35] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[36] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, and Mengting Yuan. 2020. Neural Joint Attention Code Search Over Structure Embeddings for Software Q&A Sites. *Journal of Systems and Software* 170, 1 (2020), 110773.

[37] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, Gothenburg, Sweden, 200–210.

Just Accepted

[38] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[39] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. ijcai.org, Stockholm, Sweden, 2269–2275.

[40] Yutao Hu, Yilin Fang, Yifan Sun, Yaru Jia, Yueming Wu, Deqing Zou, and Hai Jin. 2023. Code2Img: Tree-Based Image Transformation for Scalable Code Clone Detection. *IEEE Transactions on Software Engineering* 49, 9 (2023), 4429–4442.

[41] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2021. FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks. *IEEE Transactions on Reliability* 70, 1 (2021), 304–318.

[42] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436, 1 (2019), 1–6.

[43] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Berlin, Germany, 2073–2083.

[44] Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, Shoufu Luo, Heejo Lee, and Sven Dietrich. 2021. QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection. In *Proceedings of the 36th International Conference on Systems Security and Privacy Protection*. Springer, Oslo, Norway, 66–82.

[45] J.Arthanareeswaran, K.P.Tatavarthi, M.Palat, N.Gupta, and S.Sinha. 2001. Eclipse Java Development tools. site:https://www.eclipse.org/jdt/. Accessed: 2023-11-03.

[46] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A Tree-based Pre-trained Model for Programming Language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. AUAI Press, Virtual Event, 54–63.

[47] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. 2012. Java bytecode clone detection via relaxation on code fingerprint and Semantic Web reasoning. In *Proceeding of the 6th International Workshop on Software Clones*. IEEE Computer Society, Zurich, Switzerland, 36–42.

[48] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net, Toulon, France, 1–10.

[49] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, Seoul, Republic of Korea, 184–195.

[50] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE / ACM, Montreal, QC, Canada, 795–806.

[51] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural computation* 1, 4 (1989), 541–551.

[52] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 184–195.

[53] Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2839–2855.

[54] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019), 1–13.

[55] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, New Orleans, LA, USA, 1–11.

[56] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. 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. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1*. Openreview.net, virtual, 1–14.

[57] M.Brunsfeld, P.Thomson, A.Hlynskyi, J.Vera, P.Turnbull, T.Clem, D.Creager, A.Helwer, R.Rix, H.van Antwerpen, M.Davis, Ika, T.-A.Nguyen, S.Brunk, N.Hasabnis, bfredl, M.Dong, V.Panteleev, ikrima, S.Kalt, K.Lampe, A.Pinkus, M.Schmitz, M.Krupcale, narpfel, S.Gallegos, V.Martí, Edgar, and G.Fraser. 2020. Tree-sitter. site:https://github.com/tree-sitter/tree-sitter. Accessed: 2023-11-03.

[58] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent Neural Network based Language Model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*. ISCA, Makuhari, Chiba, Japan, 1045–1048.

[59] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the 30th Conference on Artificial Intelligence*. AAAI Press, Phoenix, Arizona, USA, 1287–1293.

[60] Jaccard P. 1901. Étude Comparative de la Distribution Florale dans une Portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37, 1 (1901), 547–579.

Just Accepted

[61] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL, Philadelphia, PA, USA, 311–318.

[62] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[63] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating Tree Path in Transformer for Code Representation. In *Proceedings of the 35th Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Virtual, 9343–9354.

[64] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *Proceedings of the 2nd International Workshop on Machine Learning and Programming Languages*. ACM, Philadelphia, PA, USA, 31–41.

[65] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. 2022. A Systematic Mapping Study of Source Code Representation for Deep Learning in Software Engineering. *IET Software* 16, 4 (2022), 351–385.

[66] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[67] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 411–421.

[68] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 4053–4062.

[69] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. In *Proceedings of the 18th International Joint Conference on Neural Networks*. IEEE, Budapest, Hungary, 1–8.

[70] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. In *Proceedings of the 28th International Conference on Program Comprehension*. Association for Computing Machinery, Seoul, Republic of Korea, 196–207.

[71] Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2022. Learning Program Semantics with Code Representations: An Empirical Study. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 554–565.

[72] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y. Ng, and Christopher D. Manning. 2011. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 28th International Conference on Machine Learning*. Omnipress, Bellevue, Washington, USA, 129–136.

[73] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 388–400.

[74] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, and Zhenyu Chen. 2023. An Extractive-and-Abstractive Framework for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* Just Accepted, 1 (2023), 1–39.

[75] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Zhenpeng Chen, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, and Zhenyu Chen. 2023. Artifacts of Abstract Syntax Tree for Programming Language Understanding and Representation. site:https://github.com/wssun/AST4PLU. Accessed: 2023-11-23.

[76] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Victoria, BC, Canada, 476–480.

[77] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53th Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*. The Association for Computer Linguistics, Beijing, China, 1556–1566.

[78] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zheling Zhu, and Bin Luo. 2022. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 150–162.

[79] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, Gothenburg, Sweden, 542–553.

Just Accepted

[80] Ilya Utkin, Egor Spirin, Egor Bogomolov, and Timofey Bryksin. 2022. Evaluating the Impact of Source Code Parsers on ML4SE Models. *CoRR* abs/2206.08713, 1 (2022), 1–12.
[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Long Beach, CA, USA, 5998–6008.
[82] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 13–25.
[83] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 13–25.
[84] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, Montpellier, France, 397–407.
[85] Deze Wang, Yue Yu, Shanshan Li, Wei Dong, Ji Wang, and Qing Liao. 2021. MulCode: A Multi-task Learning Approach for Source Code Understanding. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 48–59.
[86] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification. In *Proceedings of the 30th International Conference on Program Comprehension*. ACM, Virtual Event, 390–400.
[87] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.
[88] Wenhan Wang, Kechi Zhang, Ge Li, Shangqing Liu, Anran Li, Zhi Jin, and Yang Liu. 2023. Learning Program Representations with a Tree-Structured Transformer. In *Proceedings of the 30th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Taipa, Macao, 248–259.
[89] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020. Reinforcement-Learning-Guided Source Code Summarization using Hierarchical Attention. *IEEE Transactions on Software Engineering* 48, 1 (2020), 102–119.
[90] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 349–360.
[91] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. ijcai.org, Melbourne, Australia, 3034–3040.
[92] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st International Conference on Automated Software Engineering*. ACM, Singapore, 87–98.
[93] David S. Wile. 1997. Abstract Syntax from Concrete Syntax. In *Proceedings of the 19th International Conference on Software Engineering*. ACM, Boston, Massachusetts, USA, 472–480.
[94] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Proceedings of the Findings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, Online Event, 1078–1090.
[95] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 342–353.
[96] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 1–12.
[97] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy G. Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. 2020. MISIM: An End-to-End Neural Code Similarity System. *CoRR* abs/2006.05265, 1 (2020), 1–23.
[98] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, Seoul, South Korea, 1385–1397.
[99] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41th International Conference on Software Engineering*. IEEE / ACM, Montreal, QC, Canada, 783–794.
[100] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *Proceedings of the 8th International Conference on Learning Representations*. OpenReview.net, Addis Ababa, Ethiopia, 1–14.

Just Accepted

[101] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *proceedings of the 17th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista, FL, USA, 141–151.

[102] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Vancouver, BC, Canada, 10197–10207.

[103] Yu Zhou, Juanjuan Shen, Xiaoqing Zhang, Wenhua Yang, Tingting Han, and Taolue Chen. 2022. Automatic Source Code Summarization with Graph Attention Networks. *Journal of Systems and Software* 188 (2022), 111257.

Just Accepted

A Appendix

A.1 Example of AST.

```
1  public int compare (int a,int b) {
2     if (a > b) return a;
3     else return b;
4  }
```

Fig. 9. A Java code snippet s_1.

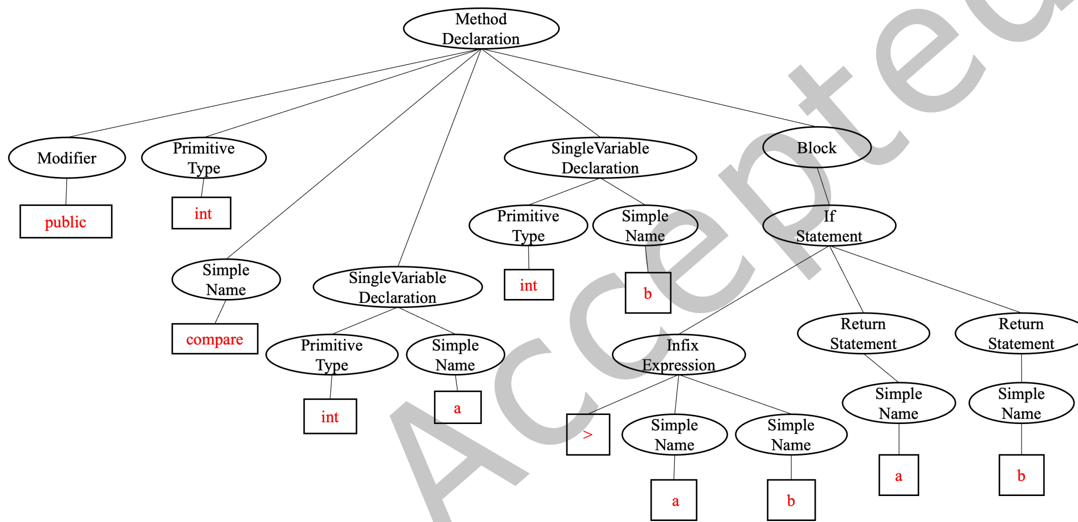


Fig. 10. The AST generated by JDT for code snippet s_1.

Just Accepted

Fig. 10 shows an example of AST, which is generated by JDT (an AST parsing method detailed in Section 3.2) for the Java code snippet s_1 shown in Fig. 9. In the AST shown in Fig. 10, nodes like `MethodDeclaration`, `SingleVariableDeclaration` and `IfStatement` are non-terminals; nodes like `Modifier`, `PrimitiveType` and `SimpleName` are terminals; and leaf nodes like `public`, `int` and `compare` are the corresponding values of the terminals. It is evident that, compared to the source code of s_1, its AST is significantly more complex. In addition, as mentioned in Section 1, different AST parsing methods (e.g., ANTLR and Tree-sitter) would generate distinct ASTs for the same source code due to different lexical rules and grammar rules. Therefore, this paper focuses on investigating the current progress of feature engineering and application of AST, aiming to provide guidance for subsequent researchers on how to effectively leverage complex ASTs to enhance code representation and subsequent code-related tasks.

A.2 Example of data produced by AST preprocessing methods.

Fig. 11 shows a piece of Java code snippet s_2. Fig. 12 shows the BFS, SBT, and AST Path of code snippet s_2 and Fig. 13 shows the Raw AST, Binary Tree, and Split AST of s_2. The complete information contained in an

```
1  private String postXml() {
2      try {
3          URLConnection conn = new URL(url).openConnection();
4      }
5  }
```

Fig. 11. A Java code snippet s_2

AST consists of its nodes and structures. Different AST preprocessing methods differ in retaining the node and structure information of the AST. As illustrated in Fig. 12 and Fig. 13, SBT and Raw AST retain all nodes and complete structure information. BFS also preserves every node of the AST while disregarding much of the structure information. Binary tree, converting the raw AST into a binary tree and merging nodes with only one child node with their child nodes, removes redundant intermediate nodes, and retains complete structure information. AST Path and Split AST divide the complete AST into smaller components, facilitating model learning, but it comes at the cost of losing partial node or structure information of the AST. It is worth noting that AST Path, due to constraints on the input size of the encoding models, is often limited in width, length, and quantity. Consequently, a significant portion of node and structure information tends to be discarded. Similarly, during the process of converting the source code into the Split AST [68], some statements in the source code (e.g., `catch` clause) are not added to the split code set, leading to the loss of node information.

A.3 Technical details of AST encoding methods.

A.3.1 Sequence models.

(i) Bidirectional Long Short Term Memory (BiLSTM).

The LSTM architecture [35] addresses the problem of learning long-term dependencies of RNN by introducing a memory cell that can preserve state over long periods of time [77]. In the work [77], the LSTM unit at each time step t is defined to be a collection of vectors in $\mathbb{R}^d$ ($\mathbb{R}$ is the set of real numbers, and d is the memory dimension of the LSTM): an input gate i_t, a forget gate f_t, an output gate o_t, a memory cell c_t and a hidden state h_t. The LSTM transition equations are the following:

$$
\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right), \quad & f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right), \\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right), \quad & u_t &= tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right), \\
c_t &= i_t \odot u_t + f_t \odot c_{t-1}, & h_t &= o_t \odot tanh(c_t)
\end{aligned}
\tag{14}
$$

where x_t is the input at the current time step; W, U are the weighted metrics; b is the bias vector; σ denotes the logistic sigmoid function; *tanh* denotes the hyperbolic tangent function; $\odot$ denotes element-wise multiplication.

Bidirectional LSTM (BiLSTM) [66] uses two LSTMs at each layer. One LSTM takes the original sequence as input and the other takes the reversed sequence as input. So it is capable of modeling the sequential dependencies between words and phrases in both directions of the sequence. The h_t at the final timestamp can be used as the code representation denoted $\mathbf{z} = h_{t_{final}}$.

(ii) Transformer.

Transformer [81] follows an encoder-decoder structure, as well as utilizing a multi-headed self-attention mechanism and positional encoding to draw global dependencies between input and output. For the code clone detection and the code search task, the decoder part of Transformer is not used since a decoder is not needed. For the code summarization task, both encoder and decoder are used.

Just Accepted

Fig. 12. BFS, SBT, AST Path of the code snippet s_2. Using JDT as the AST parser.

Just Accepted

Encoder [67]: Encoder combines multiple identical layers where each layer consists of two sub-layers. The first sub-layer forms a multi-headed self-attention structure while the other one is a fully connected layer. Both sub-layers are followed by another layer which normalizes the output of each sub-layer. The encoder maps an input sequence of symbol representations $\mathbf{x} = (x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$.

Decoder [67]: Decoder has a similar structure as encoder except that it includes one additional sub-layer. This extra sub-layer conducts multi-head attention on the encoder's output. Moreover, the self-attention sub-layer is reformed to avoid attending to subsequent positions. Given $\mathbf{z}$, the decoder generates an output sequence $\mathbf{y} = (y_1, \ldots, y_m)$ of symbols one element at a time.

Multi-head attention mechanism [67]: Transformer's self-attention aims to map a query (Q) and a collection of key (K) - value (V) pairs to vectors which are calculated as a weighted sum of the values, which is shown in the following formula:

$$Attention(Q, K, V) = softmax(QK^T/\sqrt{d_k})V. \tag{15}$$

where d_k is the dimension of the key vector.

Multi-head attention conducts self-attention process multiple times separately, with different weight matrices. All the results are concatenated and multiplied by an additional weight matrix W, as shown in the following

Fig. 13. Raw AST, Binary Tree, and Split AST of the code snippet s_2. Using Tree-sitter as the AST parser.

Just Accepted

formula:

$$
\begin{aligned}
MultiHead(Q, K, V) &= Concat(head_1, ..., head_h)W^O \\
\text{where} \quad head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V),
\end{aligned}
\tag{16}
$$

where d_{model} is the input dimension, d_k is the dimension of the key vector, d_v is the dimension of the value vector, $h = 8$ is the number of parallel attention layers, $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^v \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Positional Encoding [67]: To retain information regarding relative and absolute tokens' positions, a positional encoding layer is added at the lowest part of the encoder and decoder stacks. There are many choices of positional encodings. In this work, we use sine and cosine functions of different frequencies:

$$
\begin{aligned}
PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}), \\
PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}),
\end{aligned}
\tag{17}
$$

where pos is the position and i is the dimension.

A.3.2 Tree-structured models.

(i) Tree-Structured Long Short-Term Memory Networks (TreeLSTM)

TreeLSTM is first proposed by Tai [77] to capture the syntactic properties of natural language. TreeLSTM is a generalization of LSTMs to model tree-structured topologies. Given a tree, let $C(j)$ denote the set of children of node j. The TreeLSTM unit at each node j is defined to be a collection of vectors in $\mathbb{R}^d$, where $\mathbb{R}$ is the set of real numbers, d is the memory dimension of the TreeLSTM: an input gate i_j, forget gates f_{jk} where $k \in C(j)$, an output gate o_j, a memory cell c_j and a hidden state h_j. The entries of the gating vectors i_j, f_j and o_j are in $[0, 1]$.

Child-Sum TreeLSTMs [77]: The Child-Sum TreeLSTM can be used on tree structures where the number of children is arbitrary and children's orders are not considered. The Child-Sum TreeLSTM transition equations are the following:

$$\begin{aligned} &\tilde{h}_j = \sum_{k\in C(j)} h_k, && i_j = \sigma\left(W^{(i)}x_j + U^{(i)}\tilde{h}_j + b^{(i)}\right), \\ &f_{jk} = \sigma\left(W^{(f)}x_j + U^{(f)}h_k + b^{(f)}\right), && o_j = \sigma\left(W^{(o)}x_j + U^{(o)}\tilde{h}_j + b^{(o)}\right), \\ &u_j = tanh\left(W^{(u)}x_j + U^{(u)}\tilde{h}_j + b^{(u)}\right), && c_j = i_j \odot u_j + \sum_{k\in C(j)} f_{jk} \odot c_k,\ h_j = o_j \odot tanh(c_j), \end{aligned} \tag{18}$$

where x_j is the input at node j, W, U are the weighted metrics, b is the bias vector, σ denotes the logistic sigmoid function, $tanh$ denotes the hyperbolic tangent function, $\odot$ denotes element-wise multiplication. The hidden state h_j at the root node is considered the representation of the source code, that is code representation $\mathbf{z} = h_{root}$.

N-ary TreeLSTMs [77]: The N-ary TreeLSTM can be used on tree structures where the branching factor is at most N and where children are ordered, i.e., they can be indexed from 1 to N. For any node j, write the hidden state and memory cell of its kth child as h_{jk} and c_{jk} respectively. The N-ary TreeLSTM transition equations are the following:

$$\begin{aligned} &i_j = \sigma\left(W^{(i)}x_j + \sum_{\ell=1}^{N} U_\ell^{(i)} h_{j\ell} + b^{(i)}\right), && f_{jk} = \sigma\left(W^{(f)}x_j + \sum_{\ell=1}^{N} U_{k\ell}^{(f)} h_{j\ell} + b^{(f)}\right), \\ &o_j = \sigma\left(W^{(o)}x_j + \sum_{\ell=1}^{N} U_\ell^{(o)} h_{j\ell} + b^{(o)}\right), && u_j = tanh\left(W^{(u)}x_j + \sum_{\ell=1}^{N} U_\ell^{(u)} h_{j\ell} + b^{(u)}\right), \\ &c_j = i_j \odot u_j + \sum_{\ell=1}^{N} f_{j\ell} \odot c_{j\ell}, && h_j = o_j \odot tanh(c_j), \end{aligned} \tag{19}$$

where $k = 1, 2, \ldots, N$, x_j is the input at node j, W, U are the weighted metrics, b is the bias vector, σ denotes the logistic sigmoid function, $tanh$ denotes the hyperbolic tangent function, $\odot$ denotes element-wise multiplication. Similar to Child-Sum TreeLSTM, code representation $\mathbf{z} = h_{root}$.

(ii) AST-Trans.

AST-Trans [78] is a simple variant of the Transformer model to efficiently handle the tree-structured AST. AST-Trans exploits ancestor-descendant and sibling relationship matrices to represent the tree structure, and uses these matrices to dynamically exclude irrelevant nodes.

AST-Trans has the same encoder and decoder structure as the Transformer, while replacing the singe-head self-attention with tree-structured attention. The absolute position embedding from the original Transformer is replaced with relative position embeddings defined by the two relationship matrices to better model the dependency.

For an AST, it will be firstly linearized into a sequence, which means being transformed into SBT [37] in our experiment. Then the ancestor-descendent and sibling relationships among its nodes will be denoted through two specific matrices. Based on the matrices, tree-structured attention is adopted to better model these two

Just Accepted

relationships. In the following part, we will introduce the construction of relationship matrices and tree-structured attention.

Construction of relationship matrices. Two kinds of relationships are defined between AST nodes: ancestor-descendant (A) and sibling (S) relationships. And we use two position matrices $A_{N\times N}$ and $S_{N\times N}$ to represent the ancestor-descendent and sibling relationships respectively, where N is the total number of nodes in AST. The i-th node in the linearized AST is denoted as n_i. A_{ij} is the distance of the shortest path between n_i and n_j in the AST. S_{ij} is horizontal sibling distance between n_i and n_j if they satisfy the sibling relationship. If one relationship is not satisfied, its value in the matrix will be infinity. Note that we consider the relative relationship between two nodes, which means $A_{ij} = -A_{ji}$ and $S_{ij} = -S_{ji}$ if a relationship exists between n_i and n_j.

Formally, we use $SPD(i, j)$ and $SID(i, j)$ to denote the Shorted Path Distance and horizontal Sibling Distance between n_i and n_j. The values in the relationship matrices are defined as:

$$A_{ij} = \begin{cases} SPD(i,j) & \text{if } |SPD(i,j)| \le P \\ \infty & \text{otherwise} \end{cases} \tag{20}$$

$$S_{ij} = \begin{cases} SID(i,j) & \text{if } |SID(i,j)| \le P \\ \infty & \text{otherwise} \end{cases} \tag{21}$$

P is a pre-defined threshold and nodes with relative distance beyond P will be ignored. We set $P = 7$ according to the code provided by the original paper [78].

Tree-structured Attention. Tree-structured attention is built on standard self-attention with relative position embeddings and disentangled attention. Tree-structured attention transforms an input sequence $\mathbf{x} = (x_1, ..., x_n)$ ($x_i \in \mathbb{R}^d$ which stands for the embedding of n_i) into a sequence of output vectors $\mathbf{o} = (o_1, ..., o_n)$ ($o_i \in \mathbb{R}^d$).

The relative distance defined under the linear relationship is replaced with $\delta_R(i, j)$ where R stands for either the ancestor-descendent relationship A or the sibling relationship B in the tree structure. $\delta_R(i, j)$ reflects the pairwise distance between n_i and n_j in relationship R. Denote P as the max relative distance, $\delta_R(i, j)$ is defined as:

$$\delta_R(i,j) = \begin{cases} R_{ij} + P + 1 & \text{if } R_{ij} \in [-P, P] \\ 0 & \text{if } R_{ij} = \infty \end{cases} \tag{22}$$

R_{ij} refers to either A_{ij} defined in Eq 20 or S_{ij} defined in Eq 21.

As there are two kinds of relationships, each head only considers one relationship so that it will not add any additional parameter on top of the standard Transformer. h_A heads will use $\delta_A(i, j)$ and the rest h_S heads will use $\delta_S(i, j)$. Information from the two relationships will be merged together through multi-head attention. The output vector $\mathbf{o} = (o_1, ..., o_n)$ is computed as below:

$$\alpha_{i,j} = Q(x_i)K(x_j)^T + Q(x_i){K^P_{\delta_R(i,j)}}^T + Q^P_{\delta_R(j,i)}K(x_j)^T \tag{23}$$

$$o_i = \sum_{j}^{j\in\{j|\delta_R(i,j)>0\}} \sigma(\frac{\alpha_{i,j}}{\sqrt{3d}})(V(x_j) + V^P_{R_{ij}}) \tag{24}$$

where $Q, K : \mathbb{R}^d \to \mathbb{R}^m$ are query and key functions respectively, $V : \mathbb{R}^d \to \mathbb{R}^d$ is a value function, σ is a scoring function (e.g. softmax or hardmax), $Q^P, K^P \in \mathbb{R}^{(2P+1)\times m}$ represent the query and key projection matrices of relative positions,V^P represents the value project matrix of relative distances, $K^P_{\delta_R(i,j)}$is the $\delta_R(i, j)$-th row of K^P and $Q^P_{\delta_R(i,j)}$is the $\delta_R(i, j)$-th row of Q^P, $V^P_{R_{ij}}$is the R_{ij}-th row of V^P. Note that only the attention weights for node pairs where $\delta_{R(i,j)} > 0$ are computed.

Just Accepted