



Transformer-based code model with compressed hierarchy representation

Kechi Zhang¹ · Jia Li¹ · Zhuo Li¹ · Zhi Jin¹ · Ge Li¹

Accepted: 10 January 2025

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract

Source code representation with deep learning techniques is an important research field. There have been many studies to learn sequential or structural information for code representation. However, existing sequence-based models and non-sequence models both have their limitations. Although researchers attempt to incorporate structural information into sequence-based models, they only mine part of token-level hierarchical structure information. In this paper, we analyze how the complete hierarchical structure influences the tokens in code sequences and abstract this influence as a property of code tokens called hierarchical embedding. This hierarchical structure includes frequent combinations, which represent strong semantics and can help identify unique code structures. We further analyze these hierarchy combinations and propose a novel compression algorithm Hierarchy BPE. Our algorithm can extract frequent hierarchy combinations and reduce the total length of hierarchical embedding. Based on the above compression algorithm, we propose the Byte-Pair Encoded Hierarchy Transformer (BPE-HiT), a simple but effective sequence model that incorporates the compressed hierarchical embeddings of source code into a Transformer model. Given that BPE-HiT significantly reduces computational overhead, we scale up the model training phase and implement a hierarchy-aware pre-training framework. We conduct extensive experiments on 10 datasets for evaluation, including code classification, clone detection, method name prediction and code completion tasks. Results show that our non-pre-trained BPE-HiT outperforms the state-of-the-art baselines by at least 0.94% on average accuracy on code classification tasks with three different program languages. On the method name prediction task, BPE-HiT outperforms baselines by at least 2.04, 1.34 in F1-score on two real-world datasets. Besides, our pre-trained BPE-HiT outperforms other pre-trained baseline models with the same number of parameters over all experiments, demonstrating the robust capability of our approach. Furthermore, we conduct a detailed ablation study, proving the effectiveness of our compression algorithm and the training efficiency of our proposed model.

Keywords Code representation · Code classification · Clone detection · Code summarization

Communicated by: Christoph Treude, Raula Gaikovina Kula, Bonita Sharif

Extended author information available on the last page of the article

1 Introduction

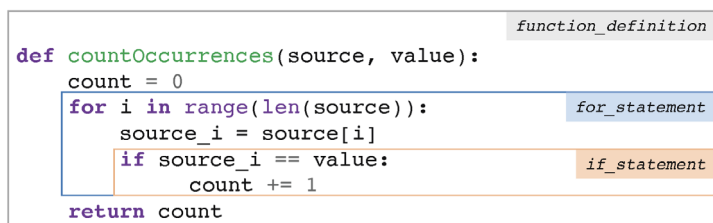
Code representation is a hot research topic in software engineering (SE) and machine learning (ML) fields. Code representation learning with machine learning aims to convert programs of different formats (sequential formats such as token sequences, structural formats such as abstract syntax trees, dependency graphs, *etc.*) into vectorized semantic embeddings. These representation vectors can be applied to many downstream tasks, such as code classification (Mou et al. 2016), type inference (Allamanis et al. 2020), code summarization (Alon et al. 2019a, b; Iyer et al. 2016; Hu et al. 2018), *etc.*

Most existing code representation methods can be divided into two categories including sequence-based models and non-sequence-based models. Sequence-based models (Allamanis et al. 2018; Buratti et al. 2020) are skilled at processing sequence order information with long-term dependency. But they are sub-optimal for capturing structural information (Mou et al. 2016; Zhang et al. 2019). Non-sequence-based models, such as tree-based (ASTNN (Zhang et al. 2019), TBCNN (Mou et al. 2016)) or graph-based models (GGNN (Li et al. 2016)), focus on encoding structural information, but sacrifice the advantages of sequence models or suffer from narrow receptive fields. Both categories of methods predominantly leverage either sequential or structural information of source code, and ignore the combination of the two modal information.

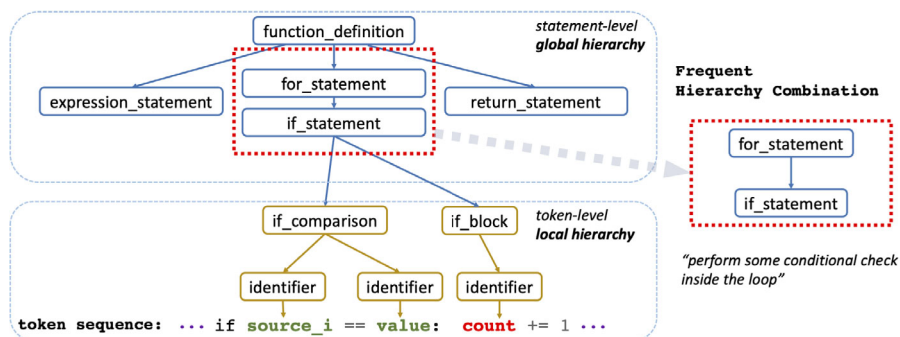
Recently, some studies jointly learned both sequential and structural information for code representation in sequence-based models. Hellendoorn et al. (2020) proposed *GREAT* which encodes the data flow graph and control flow graph into the relative position embedding in Transformer. Zügner et al. (2021) designed *CodeTransformer*, which combines distances computed on the AST in the self-attention operation. However, these methods consider limited structural information of the source code, such as node distances on the program tree or graph, and only focus on relations among tokens with the attention mechanism. They ignore the impact of overall hierarchical structure information, resulting in the sub-optimal program representation.

In this work, we take a step to explore how the complete hierarchical structure influences the tokens in the source code sequence and we abstract this influence as a property of code tokens called hierarchical embedding. We further divide this influence into two aspects: ❶ hierarchical embedding of statements (dubbed **global hierarchy**). The hierarchical embedding will affect the operational semantics of a statement. For example, in Fig. 1, the statement `count += 1` is written in the `for`-block and possibly to be executed more than once, while the statement `count = 0` will be executed only once in each function call to `countOccurrences`. ❷ hierarchical embedding of tokens within a statement (dubbed **local hierarchy**). A statement may contain different components, which will affect the semantics of each token in it. *e.g.*, as shown in Fig. 1(b), the three identifiers, `source_i`, `value`, and `count`, in the same `if`-statement, are references to variables in the function. However, the first two identifiers within `if`-comparison component (marked in green color in the figure) have more similar semantics than `count` because they appear in the same comparison expression. Therefore, hierarchical structure information can strongly affect the token-level and statement-level semantics of source code. Considering them in code representation is indispensable and meaningful.

We also observe that frequent hierarchy combinations can represent strong semantics and help identify unique code structures. In Fig. 1 there is a frequent hierarchy combination `for`-statement \rightarrow `if`-statement. This hierarchy combination is used to perform conditional checks inside the loop, thus often implemented in classical algorithms such



(a) Hierarchical location affects the operational semantics of a statement



(b) Hierarchy information within a statement affects the semantics of a token. Frequent hierarchy combinations can represent unique semantics.

Fig. 1 An illustrative example of the hierarchy information in source code. The hierarchy of a token refers to the hierarchical location of its statement (*global hierarchy*) and the local component of the token in the statement (*local hierarchy*). The token `source_i` and `value` are both `identifier` in `if_comparison` marked in green in (b) and they are more closely related. The framed part in (b) represents a frequent hierarchy combination

as Linear Search or Selection Sort. These frequent hierarchy combinations can represent unique semantics. Mining these common combinations is essential for efficiently representing code semantics. Inspired by the byte-pair encoding algorithm (Sennrich et al. 2016), we innovatively propose a compression algorithm: **Hierarchy BPE**. Our proposed algorithm views the hierarchy node as the basic unit and iteratively merges high frequency combinations to get a uniform and efficient hierarchy representation. Our further experiments show that this compression algorithm can reduce the total length of the hierarchy and efficiently enhance the code representation performance.

To validate our intuition of modeling compressed hierarchies alongside code sequences, we propose unifying the compressed hierarchical structure with the code sequence in a concise Transformer format. We call this method **Byte-Pair Encoded Hierarchy Transformer (BPE-HiT)**, a simple yet effective code representation model that strikes a balance between efficiency and effectiveness. Building on our previous work (Zhang et al. 2023), BPE-HiT provides a more detailed code representation pipeline, implemented in four stages: **①Hierarchy Extraction**. We begin by parsing the source code into a concrete syntax tree and extracting hierarchy information from it. **②Hierarchy BPE**. Next, we employ a compression algorithm to mine frequent hierarchical combinations. **③BPE-HiT and Hierarchy-aware Pre-training**. We introduce the Hierarchy Transformer with the compression algorithm. Given that BPE-HiT significantly reduces computational overhead, we scale up the model training phase and implement a hierarchy-aware pre-training framework. This framework

includes a comprehensive set of novel pre-training objectives designed to help the model learn robust representations that capture the hierarchical structure of source code while establishing the corresponding relationship between code tokens and their hierarchical representations. **④Downstream Modules.** Finally, we use the Hierarchy Transformer to produce a vectorized representation for various downstream tasks. The Hierarchy Transformer consists of two components: a Transformer-based hierarchy encoder that learns the representation of compressed hierarchy information, and a Transformer-based sequence encoder that fuses the hierarchy and token sequence information to generate the final code representation.

We conduct extensive experiments to evaluate the **non-pre-trained** and **pre-trained** versions of our BPE-HiT method. To show the effectiveness and efficiency of our BPE-HiT architecture, we firstly compare the non-pre-trained version of BPE-HiT with baselines: **①** we investigate the impact of our proposed compressed hierarchy by comparing it with the original hierarchy. Experimental results prove the effectiveness of our hierarchical embedding which only costs a small number of additional parameters and achieves significant improvements on sequence-based code representation models. **②** We design a variable scope detection task and verify that our method can acquire the scope information in the global hierarchy and represent the relationship between sequence and structure information of source code. **③** We further apply the code representations obtained from our method to downstream classification and generation tasks on 10 datasets with four different tasks, including code classification, clone detection, method name prediction and real-world code completion. Our method achieves state-of-the-art (SOTA) performances on all tasks. Precisely, on code classification tasks, BPE-HiT outperforms existing models by at least 0.94% on average accuracy on five popular datasets with three different program languages. Our model even significantly outperforms the pre-trained CodeBERT on hard datasets. On the method name prediction task, BPE-HiT outperforms baselines by at least 2.04, 1.34 in F1-score on two real-world datasets. These impressive results indicate the benefits of our approach for aligning the compressed hierarchical embedding with code tokens for source code understanding. **④** For our pre-trained version of BPE-HiT, we show that our pre-trained model outperforms other pre-trained baseline models with the same number of parameters, demonstrating the robust capability of our approach. **⑤** We also measure the impact of different compression strategies for the hierarchy information. Experiments show that our hierarchy BPE algorithm can extract the key structural information from source code and efficiently improve the code representation model. We further conduct ablation studies to explore different settings in our model design. In addition, we also perform experiments to evaluate the time efficiency of our model. Results prove that our BPE-HiT can significantly reduce the total training time and have stable training efficiency.

This paper extends our preliminary study, which appears as a research paper in ICPC (Zhang et al. 2023). In particular, we extend our preliminary work in the following direction:

- **Incorporation of Compressed Hierarchies:** We propose a simple yet effective method to incorporate compressed hierarchical embeddings into a Transformer model, named BPE-HiT. This is an extended version of HiT from our preliminary work (Zhang et al. 2023). HiT integrates original hierarchy information into sequence-based code representation models with minimal overhead. In this paper, we enhance the previous model by employing a BPE algorithm for compressed hierarchies, reducing computational costs. This enables us to scale up the training phase and apply our model to more complex code tasks.

- **Hierarchy-aware Pre-training Framework:** Based on our proposed efficient model architecture, we introduce a complete pre-training framework that allows the code representation model to learn code semantics from both token-level and hierarchy-level information.
- **Strengthened Experiments:** We enhance our experiments by adding a new complex task-real-world code completion-demonstrating the effectiveness and efficiency of our method in real-world software development scenarios. Additionally, we adopt A/B testing to estimate the significance of the improvements achieved by our method.
- **Detailed Performance Analysis:** We further discuss performance variations when adopting different settings and strategies in our model design. We evaluate time efficiency, demonstrating that our BPE-HiT achieves excellent training efficiency and stability. For the pre-training framework, we assess the effectiveness of our proposed new training objectives.

The main contributions of this paper, which form a super-set of those in our preliminary study, are summarized as follows.

- We analyze how the complete hierarchical structure influences tokens in code sequences and abstract this influence as a property of code tokens called hierarchical embedding. We further analyze the semantics of hierarchy combinations in code structure and propose a novel compression algorithm to extract frequent hierarchy combinations. Our algorithm can efficiently reduce the total length of hierarchical embedding.
- We propose an effective code representation model architecture, BPE-HiT, to incorporate the compressed hierarchical embeddings into a Transformer. Based on our model, we introduce a comprehensive pre-training framework that enables the code representation model to learn code semantics from both token-level and hierarchy-level information.
- We evaluate our approach on four source code-related tasks using ten different datasets, including code classification, clone detection, method name prediction, and code completion. Our experimental results demonstrate that the BPE algorithm can effectively mine key hierarchical information. Our method aligns the compressed hierarchical embeddings with code tokens, thereby improving the performance of the Transformer-based code representation model in both the **non-pre-trained** and **pre-trained** versions of BPE-HiT.

2 Related Work

2.1 Sequence-Based Code Representation

Code representation learning is a hot research topic in software engineering and machine learning fields. Among various representation approaches, sequence models are the most mainstream code representation models, based on the concept of “naturalness” (Hindle et al. 2012; Allamanis et al. 2018; Buratti et al. 2020; Yang et al. 2022), which argues that programming languages are usually simple, repetitive, and can be understood through the same approaches used in natural language processing. Sequence models are efficient and effective in processing the code token sequence, and have been applied across many SE tasks (Iyer et al. 2016; Cai et al. 2020; Ahmad et al. 2020; Allamanis et al. 2016; Liu et al. 2019; Nguyen et al. 2020; Wang et al. 2024; Liu et al. 2022). Recently, sequence-based pre-trained models (Feng et al. 2020; Lu et al. 2021; Li et al. 2022), such as CodeBERT, have achieved success in SE tasks, demonstrating the power of sequence-based models. In this paper, our

proposed approach focuses on modifying the Transformer architecture to incorporate hierarchical information, demonstrating that even without pre-training, we can achieve strong results, often surpassing these models that require pre-training.

There are also researchers trying to encode structural information with sequence models (Alon et al. 2019a, b; Hu et al. 2018; Niu et al. 2022; Li et al. 2023). Some leverage the program AST to model the structure in source code and represent source code as a set of leaf-to-leaf paths over ASTs (Alon et al. 2019a). Others use the flattened (AST) node sequence as input to model the structure in the source code. However, leaf-to-leaf paths would scrap the code sequence information. Using the flattened node sequences to encode tree structures makes the entire sequence representation significantly longer, and code tokens are interspersed with other non-terminal tree nodes. These approaches weaken the “naturalness” of the source code context. We declare that combining the “naturalness” and hierarchical structure information in the code sequence is essential. In this paper, we follow the research line of incorporating hierarchical structure into the code sequence representation model.

2.2 Non-Sequence-Based Code Representation

Programs contain extensive structure information. Therefore, recent studies explore using tree-based (Mou et al. 2016; Zhang et al. 2019; Bui et al. 2021; Alon et al. 2020; Wang et al. 2022) and graph-based (Allamanis et al. 2018; Fernandes et al. 2019; Yin et al. 2019; Wang et al. 2020, ?; Zhang et al. 2022) models for code representation models. Although tree-based and graph-based models can directly capture structural information of source code, they are generally less efficient than sequence models. They require complex data preprocessing designed for particular languages. In the input tree or graph, the number of nodes in the receptive field of each node grows exponentially, which leads to models not being able to understand the complete sequence information well (Alon and Yahav 2021). Figure 2 shows an example of the exponential number of nodes in the code tree or graph compared with code sequences. In our experiments, we include graph-based and tree-based models as our baselines.

Recently, some studies have also integrated structure information of source code into sequence-based models (Hellendoorn et al. 2020; LeClair et al. 2019; Zügner et al. 2021; Kim et al. 2021; Guo et al. 2021). Some of these studies are designed for code representation tasks: (Hellendoorn et al. 2020) proposed Graph Relation Embedding Attention Transformer (GREAT), which biases Transformer with relational information from graph edge types, and achieves good performances on variable misuse task. Zügner et al. (2021) proposed Code-Transformer, which computes pairwise distances on AST and integrates multiple relations into the attention module. However, these methods jointly learn the source code’s sequential and structural information with a simplified token-level hierarchical structure, such as node distances on the program tree or graph. They overlook the full impact of hierarchical structure information on the code sequence, and we will give a deep analysis of the hierarchy information contained in the code structure.

3 Analysis of Hierarchy Information

In this section, we analyze how the complete hierarchy information influences the tokens in the source code sequence and motivating examples. We further analyze the semantic of

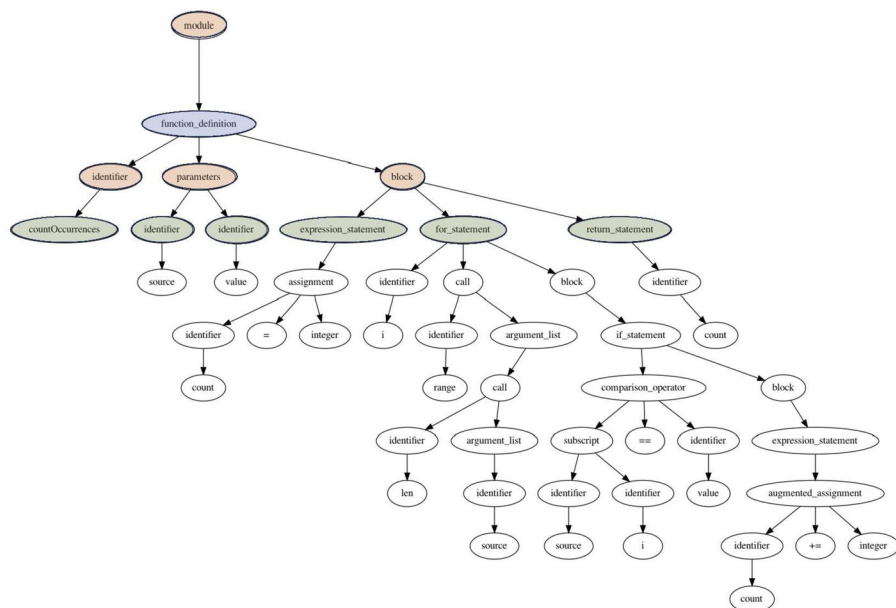


Fig. 2 The abstract syntax tree for the code snippet in Fig. 1. The number of AST nodes from the root to the leaf is approximately twice the length of the code tokens and grows exponentially with the number of non-sequence-base model (tree or graph model) steps. This is illustrated in the figure, where the progression from blue nodes to orange nodes to green nodes highlights this exponential growth

hierarchy combination in source code and explore the feasibility of compressing hierarchy information.

Generally, given a program, we tokenize the source code to get token sequences and encode them with a sequence model with hierarchical embeddings. Hierarchical embedding can be regarded as a semantic property of each token in the sequence. We further divide the semantics in the hierarchical embedding into two levels: hierarchical embedding of tokens within a statement (named as **local hierarchy**) and hierarchical embedding of statements (named as **global hierarchy**). We analyze the token-level and statement-level semantics introduced by each part of the hierarchical embedding.

3.1 Understanding Token-Level Semantics with Local Hierarchy

The semantics of the token is related to the local hierarchy. Simply encoding a token with only the token embedding will lose local structural information. Specifically, the same tokens may have different semantics but are given the same token embedding. On the contrary, different tokens that appear in similar contexts may have similar semantics.

For example, in Fig. 3, the variable `i` in the first program is used as a loop variable. In contrast, the variable `i` in the quick sort program represents the partition index. Although both programs define and use the variable `i`, they express different meanings with it. Furthermore, in the first program in Fig. 3, there is another variable `j` used as a loop variable. We can discover that the variable `i` and `j` in the first program have similar semantics, which is different from the meaning of variable `i` in the second program. However, the embedding layer gives the same representation to `i` and a different representation to `j`. It's important to

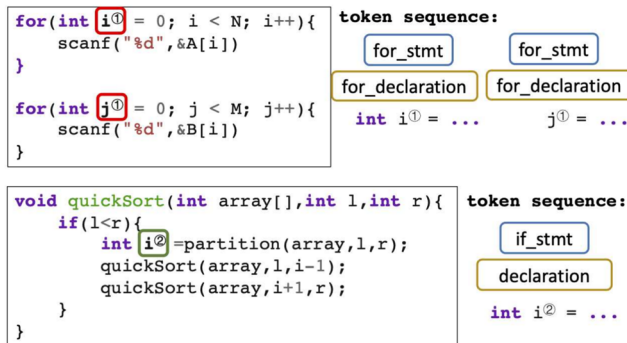


Fig. 3 An illustrative example of these tokens in different contexts. The same token may have different semantics ($i^{\textcircled{1}}$ and $i^{\textcircled{2}}$ in two programs), and different tokens with similar context may have similar semantics ($i^{\textcircled{1}}$ and $j^{\textcircled{1}}$ in the first program)

note that while some sequence model architectures like Transformer (Vaswani et al. 2017) and GREAT (Hellendoorn et al. 2020) can implicitly learn broader semantic information, mitigating errors from basic embedding layers to some extent, they often struggle to effectively and directly model high-level relationships. For example, when using a Transformer model trained on the Python800 dataset, we found that the cosine similarity between the variable i and j in the first program was 0.37, while the similarity between the variable i across the two programs was 0.32, showing no significant difference. In contrast, when hierarchical information is incorporated, the cosine similarity between i and j increases to 0.71. This demonstrates the necessity of local hierarchy for understanding token-level semantics.

Another example is shown in Fig. 4. In these two lines of Python code, the token appears three times, indicating a module import, a module reference, and a module attribute, respectively. It is hard to exploit this implicit semantic difference for a traditional sequence model.

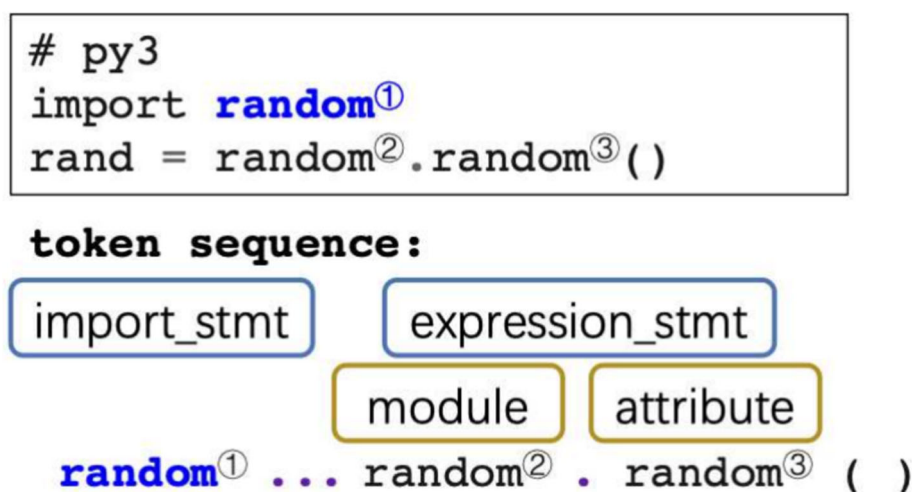


Fig. 4 An illustrative example of the repeated tokens in different statements. The repeated token `random` represents different semantics

Thus we believe that it is essential to model the structure of the local hierarchy to understand the meaning of tokens.

We revisit existing joint learning models for code representation (Hellendoorn et al. 2020; Zügner et al. 2021) and find that most of them focus on encoding relations between tokens in code sequences, such as node distances on the program tree/graph. They are skilled at encoding the token-level semantics and yield good performance. However our further analysis shows that hierarchical structural information contains more than token-level information.

3.2 Understanding Statement-Level Semantics with Global Hierarchy

To illustrate the statement-level semantics contained in the global hierarchy, we show two classical examples: ❶ The semantics of statements is related to the global hierarchy. Most programming languages permit the creation of blocks and nested blocks. The block structure is fundamental for creating the control flow and defining the scopes of variables. Thus modeling block structure helps understand control flow and variable scope. The control flow will influence the effects of a statement. Figure 5 shows an example of the semantics of the statement in the block structure. The statement `print(sum)` can be placed at any of the three marked positions: in the inner `for` loop, in the outer `for` loop, and outside of the outer `for` loop. A small change in the statement's position will affect the program's output. Locating the statement in the block structure will help the model better determine the function of the source code. ❷ We also observe that the functionality of the program is closely related to the global hierarchy. Source code with similar functionalities tends to have similar global hierarchies. This unique global hierarchy can help the model distinguish the functionality and semantics of the program. To better confirm our observation, we conducted a simple statistical experiment on the code classification task on **Python800** dataset from the CodeNet project (Puri et al. 2021). Figure 6 gives an example solution with problem id *p02412* in CodeNet. This program counts the number of triplets of numbers satisfying two requirements: each number is less than n and they sum up to k . There is a `if` statement in four levels of nesting while/for loop. We traverse the dataset and find that 121 programs have a `while-for-for-for-if` hierarchical structure. We surprisingly find that 111 of the 121 (about 91.7%) programs are written to solve problem *p02412*. There are 300 programs in total for solving this problem, which means about 37% of the programs solving *p02412* use the special structure mentioned above. Through these statistics, we claim that the global hierarchy of source code is strongly related to the functionality and semantics of the program.

Through our analysis, we show that global and local hierarchical structures are essential for code representation models while existing joint learning models ignore the former. We will conduct an empirical study to prove our point experimentally in Section 6.1.

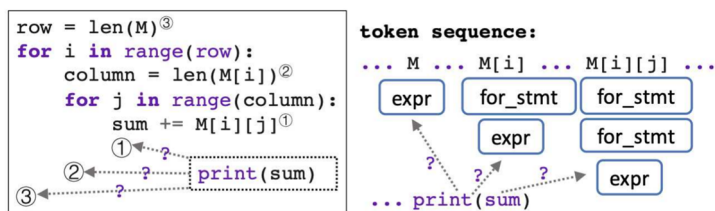


Fig. 5 An illustrative example of the implicit block structure ignored in token sequences. We can place the statement `print(sum)` in three places, where the token sequences are almost the same, but the semantics are different

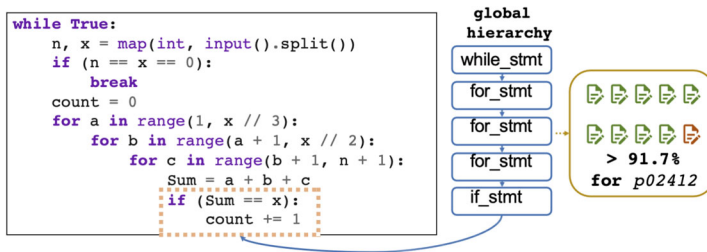


Fig. 6 An illustrative solution example from problem id *p02412* in CodeNet Python800. There are 121 programs that have such a hierarchical position in the dataset. 111 of the 121 programs (more than 91.7%) are written for the problem *p02412*

3.3 Compressing Hierarchy into Frequent Combinations

The hierarchical structure of source code plays a significant role in shaping both token-level and statement-level semantics. The hierarchy information within source code can be further effectively compressed into a variety of combinations. An illustrative example is provided in Fig. 6, where the frequently occurring hierarchy combination *while-for-for-for-if* can serve as discriminators for unique code structures. This intriguing observation motivates us to delve deeper into the realm of high-frequency combinations, which hold promise for enhancing the efficiency of code representation.

To further explore this observation, we conduct a preliminary experiment aimed at extracting frequent hierarchy combinations from **Python800** dataset. Table 1 shows a collection of example frequent combinations. These combinations exhibit a wide coverage, encompassing diverse aspects such as function definition structures, loop conditional statements, and more, representing various distinct semantics.

Table 1 Example Hierarchy Combinations on Python800 dataset

Example Combinations		
function-definition		
function-definition	→	return-statement
...		
expression-statement		
expression-statement	→	assignment
expression-statement	→	call
attribute	→	identifier
expression-statement	→	call
argument-list		
...		
for-statement		
for-statement	→	while-statement
for-statement	→	for-statement
for-statement	→	if-statement
...		

We can also observe that these combinations exhibit clear semantic patterns, incorporating both global hierarchy and local hierarchy. For instance, the combination `function_definition-return_statement` is a **compressed global hierarchy** structure frequently appearing at the end of functions to indicate return statements, emphasizing the function's return value. Our compression algorithm effectively merges the nodes `function-definition` and `return-statement`, enhancing this semantic relationship.

On the other hand, `expression_stmt-call-attribute-identifier` is a typical **compressed local hierarchy** feature, commonly used in the “`objective.method(...)`” structure. Our compression algorithm successfully merges this semantically rich pattern, demonstrating the effectiveness of the approach in capturing essential code semantics.

Subsequently, we integrate these identified combinations into the original hierarchical structure of the source code. This process is visually demonstrated in Fig. 7, showcasing a complex nested program structure. The variable `res` appears five times in the provided function. Each occurrence corresponds to a unique hierarchy. We show the original tree structure of this code, which is very complex and full of nested structures. The frequent hierarchy combinations are boxed in red in the figure. These combinations appear widely in the whole dataset as shown in Table 1. We further compress frequent combinations and convert the original structure into a very concise tree structure (the bottom half part of Fig. 7). From this simplified tree, we can see that the variable `res` at the second, third, and fourth positions have the same hierarchy combination `expression_stmt-call-attribute-identifier`. The semantics of these three positions are very similar. They are all used to save the obtained factor in the provided function. Upon application of the compressed hierarchy, the syntax structure transforms into an intuitive and concise representation, making complex code structures easier to understand. This compressed method accentuates key structural features while reducing the redundant length. Consequently, the integration of hierarchy information with compressed combinations emerges as a potent strategy for efficiently representing code

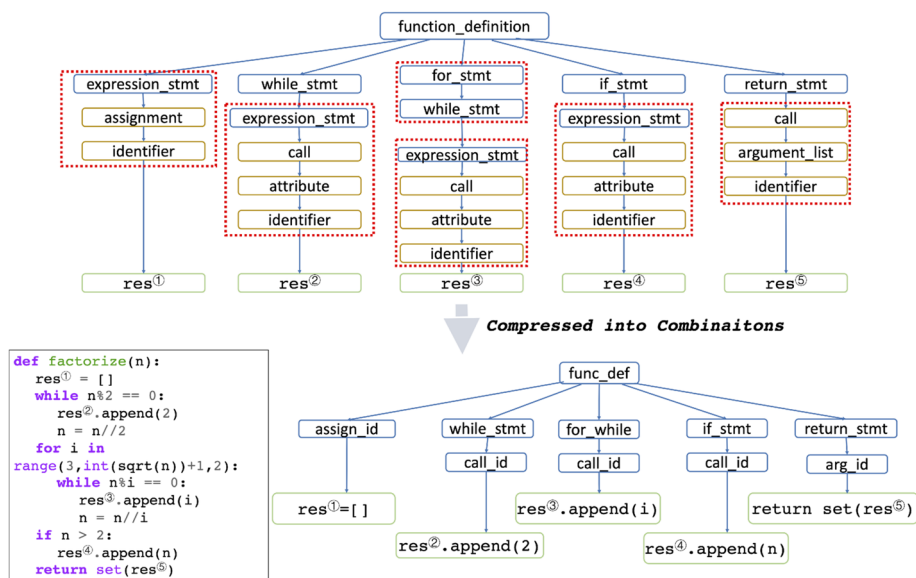


Fig. 7 An example of the compression process for the hierarchy information

structure details. It prompts us to investigate deeper into the process of extracting these frequently occurring hierarchy combinations. Our subsequent investigations serve to underscore its efficiency in Section 4.3.

4 Proposed Model

4.1 Overview

In this work, we propose a source code representation method dubbed BPE-HiT, to encode the code sequence and compressed hierarchical embedding simultaneously. The entire pipeline of our approach is shown in Fig. 8:

- **Hierarchy Extraction.** To get the hierarchical embedding of the source code, we first parse the source code to a concrete syntax tree and extract root-to-leaf paths from it.
- **Hierarchy BPE.** We then use the compression algorithm to extract frequent hierarchy combinations and convert them into fundamental units. We obtain these compressed hierarchical embeddings and use them as the model input.
- **Hierarchy Transformer and Hierarchy-aware Pre-training** We concatenate the token embedding with its compressed hierarchical embedding representation and use another Transformer-based sequence encoder to learn the final code representation. Based on

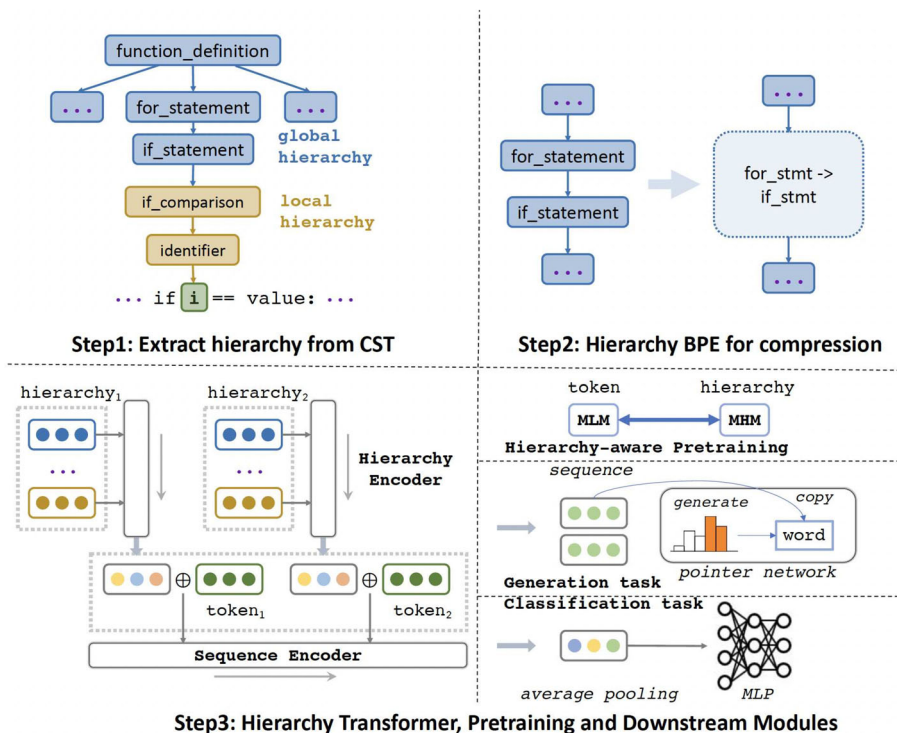


Fig. 8 The pipeline of our approach

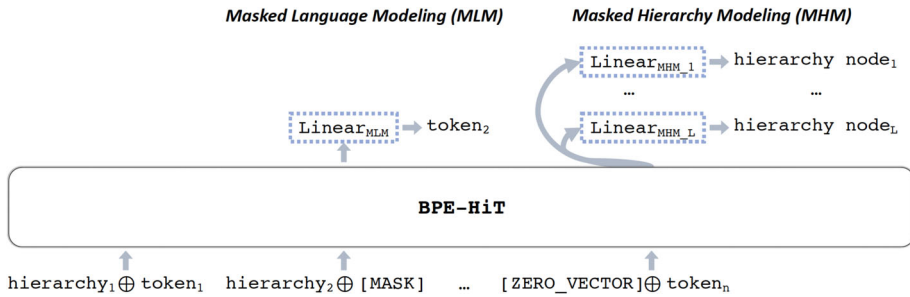


Fig. 9 An illustration about our proposed Hierarchy-aware Pre-training Tasks. The Masked Language Modeling (MLM) task replaces a portion of the token embeddings with the *[MASK]* token. The Masked Hierarchy Modeling (MHM) task replaces a portion of the hierarchical embeddings with zero vectors. Subsequently, we compute the losses for each task using (7), (8) for MLM and (9), (10) for MHM, respectively. These losses are then weighted and summed to facilitate the training process using (11)

this new model architecture, we propose a hierarchy-aware pre-training framework that enables the hierarchy information understanding (Fig. 9)

- **Downstream Modules for Representation and Generation Tasks** We can further use the representation vector to feed into a linear classifier or a decoder to support various downstream classification or generation tasks.

4.2 Hierarchy Extraction

As demonstrated in previous works (Mou et al. 2016; Zhang et al. 2019), parsing trees of source code contain rich structural information. Thus, we use the parsing trees to extract hierarchical embeddings of the source code. In software engineering, concrete syntax tree (CST) and abstract syntax tree (AST) are the two prevailing trees. CST reflects the exact syntax of the source code and each leaf node in CST corresponds to a source code token. In theory, a CST can be converted to an AST equivalently. On the contrary, AST can not represent every detail appearing in the real syntax. For instance, the braces, semicolons, and parentheses are discarded in ASTs, making it hard to align the structural information with these code tokens. In this paper, we apply CST to extract the hierarchical embedding of each source code token.

As stated in Section 3, we expect to model the hierarchy information of source code to better understand the semantics of its statements and code tokens. To get the hierarchical embedding, we extract all root-to-leaf paths from the CST. Considering that each leaf node in the CST corresponds to a source code token, the extracted root-to-leaf paths can be aligned to each token and thus these paths express the hierarchy of the program. Intuitively, we can further divide a path into two parts: the root-to-statement path and the statement-to-leaf path. The root-to-statement path represents the surrounding block structure of a statement, which reflects the position of a statement in the program. We name the structure *global hierarchy*. The statement-to-leaf path indicates the structure of the local context of a source code token. The structure represents the position of a token within the statement and is dubbed *local hierarchy*. Step 1 in Fig. 8 gives an illustration of different parts of the tree path. The *global hierarchy* and *local hierarchy* constitute a root-to-leaf path.

4.3 Hierarchy BPE

Considering that all nodes along the root-to-leaf paths in CST encompass the extracted hierarchy information of the source code, the length of the hierarchy directly correlates with the complexity of the nested program structure. However, directly encoding the complete hierarchy for complex programs can impose a significant computational burden and potentially compromise final performance. In our preliminary experiments (Section 3.3), we observed that the original hierarchy of source code can be restructured into various combinations. These frequent hierarchy combinations often signify distinct semantics within the source code. If we could encode these combinations into fundamental units, we could effectively reduce the length of the original hierarchy sequence. As a result, it is essential to further extract and compress these frequent combinations for the originally extracted hierarchy.

Algorithm 1 Compress the hierarchy of source code.

```

1: procedure HIERARCHYBPE(code_corpus, vocab_size)
2:   Extract hierarchy (Section 4.2) from code_corpus
3:   Initialize vocabulary with each path node
4:   Initialize merge operations as an empty list
5:   while size of vocabulary < vocab_size do
6:     Find the most frequent pair of adjacent path node or combination ( $n_1, n_2$ )
7:     Create a new combination  $comb_1$  to represent ( $n_1, n_2$ )
8:     Update vocabulary and frequencies accordingly
9:     Add merge operation ( $n_1, n_2 \rightarrow comb_1$ ) to merge operations list
10:  end while
11:  return hierarchy combination vocabulary, merge operations
12: end procedure

```

Drawing inspiration from the byte-pair encoding algorithm (Sennrich et al. 2016), we introduce a novel compression method named Hierarchy BPE. The compression process is shown in Algorithm 1. Beginning with the extraction of hierarchy from the code corpus (outlined in Section 4.2), we initially treat hierarchy nodes as the initial vocabulary components. At each iteration, we identify the most frequently occurring adjacent pairs. These pairs are then merged into new combination symbols and replaced in the original hierarchy sequence. The merging process continues until the total vocabulary size aligns with the predefined target. Meanwhile, this process results in the inclusion of newly mined, high-frequency combinations in the vocabulary.

In our experimental setup, the final vocabulary size is automatically selected by our proposed BPE compression algorithm, which varies between 300 and 600 depending on the programming language. This selection is based on the compression length, aiming to maintain the average length of compressed paths around *three*. We determine the length of compressed paths by observing whether the mined compressed paths have practical significance and effectiveness. The compression algorithm is applied to the complete hierarchy, and we also conduct an extensive empirical study to compress global and local hierarchies separately (Section 6.1).

To straightforwardly verify the efficacy of our compressed hierarchy, we conducted a statistical analysis of the length distribution within the hierarchy. The distribution of the original hierarchy and compressed hierarchy in Python800-train dataset is shown in Fig. 10. Notably, it indicates that approximately 82% of the compressed hierarchies can be succinctly represented using combinations of fewer than four elements. In contrast, the majority of original

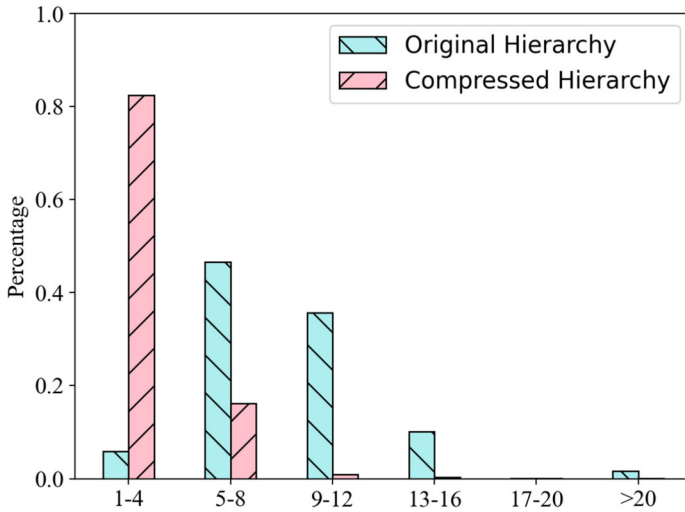


Fig. 10 The distribution of the length of original hierarchy and compressed hierarchy on Python800-train dataset

hierarchies consist of 5-12 nodes, which tend to contain redundant structural information. The empirical evidence underscores the efficacy of the compressed hierarchy, which not only reduces the overall length but also encapsulates structural information more concisely. Further substantiation of these findings will be presented through comparative analyses of hierarchy lengths across various datasets, as expounded upon in Section 5.1.

4.4 Hierarchy Transformer

We propose a new sequence model, Hierarchy Transformer (HiT), and use it to combine the token sequence and compressed hierarchy for code representation tasks. Given a token sequence with compressed hierarchical path $s = [(\mathbf{n}_1, t_1), \dots, (\mathbf{n}_l, t_l)]$, where l is the length of the token sequence, (t_1, \dots, t_l) indicates token sequence of the source code, and $\mathbf{n} = (n_1, \dots, n_{l'})$ is a compressed tree path (sequence of tree nodes or combinations). To process both token sequences and hierarchy information, we use a small Transformer to encode the paths \mathbf{n}_k and acquire the representation of hierarchy information. Lately, we concatenate the hierarchy embeddings and token embeddings, and then feed them into a larger Transformer model. We then concatenate the representation of hierarchy information and token embedding and feed them into a larger Transformer model. We refer to the two transformers as Hierarchy Encoder and Sequence Encoder, respectively.

- (i) **Hierarchy Encoder** The hierarchy encoder aims to convert the compressed hierarchy into a vector representation. For each compressed hierarchy, we view them into a sequence with tree nodes or combinations and feed them into the Transformer model. Then we perform a mean pooling to get the representation of the whole path.

$$e_1, e_2, \dots, e_{l'} = \text{Embed}(n_1, n_2, \dots, n_{l'}) \quad (1)$$

$$h_1, h_2, \dots, h_{l'} = \text{Transformer}_{hie}(e_1, e_2, \dots, e_{l'}) \quad (2)$$

$$p = \text{MeanPooling}(h_1, h_2, \dots, h_{l'}) \quad (3)$$

- (ii) **Sequence Encoder** The sequence encoder is designed to output the final code representation vector by merging the hierarchy representations and token representations. We first concatenate the hierarchy representations p_i with the token embeddings e_i . Then we use another Transformer as the sequence encoder to get the final code representation.

$$E_1, E_2, \dots, E_l = \text{Embed}(t_1, t_2, \dots, t_l) \quad (4)$$

$$\mathbf{X} = (p_1 || E_1, p_2 || E_2, \dots, p_l || E_l) \quad (5)$$

$$H_1, H_2, \dots, H_l = \text{Transformer}_{seq}(\mathbf{X}) \quad (6)$$

In our experiment, we set the hierarchy dimension (p_i) and token dimension (E_i) the same during concatenating. Through the above process, we can equip the Transformer with the compressed hierarchy and encode the source code into a vectorized representation for various code tasks.

4.5 Hierarchy-Aware Pre-Training Tasks

Our proposed BPE-HiT significantly reduces additional computational overhead, making it feasible to scale up the model training phase. Given that pre-trained models have demonstrated promising performance on various tasks, we extend existing pre-trained code models (Feng et al. 2020; Guo et al. 2021) by introducing a hierarchy-aware pre-training framework based on BPE-HiT. The aim is to enhance the model's understanding of both global and local code hierarchy information before fine-tuning on downstream tasks. We give an illustration in Fig. 9.

4.5.1 Objective

The primary objective of our pre-training framework is to learn robust representations that capture the hierarchical structure of source code while establishing the corresponding relationship between code tokens and their hierarchical representations. This is achieved through two main tasks: Masked Language Modeling (MLM) and Masked Hierarchy Modeling (MHM).

- (i) **Masked Language Modeling (MLM)** Masked Language Modeling (MLM) is inspired by the BERT model (Devlin et al. 2019) and aims to enhance the model's understanding of the token-level semantics in source code. In this task, we randomly mask some of the tokens in the input code sequence and train the model to predict the original tokens based on their surrounding context. Specifically, we sample 15% of the tokens S_m from the input sequence, replacing 80% of them with a [MASK] token, 10% with random tokens from the vocabulary, and leaving the remaining 10% unchanged. These changes are only applied to the token embedding, ensuring the hierarchy encoder module remains unaffected during this pre-training task. This encourages the model to learn meaningful representations of code tokens and their dependencies. The model leverages semantic information from the neighboring code context and hierarchical information from the compressed hierarchy representations to infer masked code tokens, promoting the learning of code representations from diverse sources of knowledge. The MLM prediction probability is given by:

$$p(x_i | X_{\text{mask}}) = \text{softmax}(\mathbf{W}_{\text{mlm}} \mathbf{H} + \mathbf{b}_{\text{mlm}}) \quad (7)$$

where \mathbf{W}_{mlm} and \mathbf{b}_{mlm} are the weights and bias of the linear layer. The MLM loss function is then defined as:

$$\mathcal{L}_{\text{MLM}} = - \sum_{x_i \in S_m} \log p(x_i | X_{\text{mask}}) \quad (8)$$

where S_m denotes the set of masked tokens, x_i represents a masked token, and X_{mask} is the input sequence with masked tokens.

- (ii) **Masked Hierarchy Modeling (MHM)** Inspired by the masked language modeling task in NLP, we introduce Masked Hierarchy Modeling (MHM) to predict nodes in the masked compressed hierarchy. Given an input sequence, we randomly mask 80% of the hierarchical representation, replacing them with zero vectors, and train the model to predict these hierarchical nodes. Since predicting a hierarchical path involves multiple nodes, for each hierarchical path, we use L_H independent linear layers to predict the corresponding nodes in the path. Let \mathbf{H} be the hidden states from the transformer encoder, the prediction for the j -th node $n_{i,j}$ in the compressed hierarchical tree path at time step i is given by:

$$p(n_{i,j} | H_{\text{mask}}) = \text{softmax}(\mathbf{W}_{\text{mhm}_j} \mathbf{H} + \mathbf{b}_{\text{mhm}_j}) \quad (9)$$

where $\mathbf{W}_{\text{mhm}_j}$ and $\mathbf{b}_{\text{mhm}_j}$ are the weights and bias of the j -th linear layer. The MHM loss function is then defined as:

$$\mathcal{L}_{\text{MHM}} = - \sum_{\mathbf{n}_i \in H_m} \sum_{j=1}^{L_H} \log p(n_{i,j} | H_{\text{mask}}) \quad (10)$$

where H_m denotes the set of masked hierarchical paths, $n_{i,j}$ represents the j -th node in the hierarchical path \mathbf{n}_i at time step i , and H_{mask} is the input sequence with masked hierarchy representations.

4.5.2 Training Procedure

The pre-training process involves jointly training the model on both MLM and MHM tasks. We use a multi-task learning approach where the loss function is a weighted sum of the losses from both tasks.

$$\mathcal{L}_{\text{pretrain}} = \lambda_{\text{MLM}} \mathcal{L}_{\text{MLM}} + \lambda_{\text{MHM}} \mathcal{L}_{\text{MHM}} \quad (11)$$

where \mathcal{L}_{MLM} is the loss for the Masked Language Modeling task, \mathcal{L}_{MHM} is the loss for the Masked Hierarchy Modeling task, and $\lambda_{\text{MLM}}, \lambda_{\text{MHM}}$ are their respective weights. The detailed pre-training settings are described in Section 5.3.

4.6 Downstream Modules

In software engineering, most program processing tasks can be categorized into classification tasks and generation tasks. To apply our BPE-HiT on downstream tasks, we use different downstream modules according to the type of task.

- (i) **Classification** For classification tasks, models are required to classify programs based on the functionalities or other properties they implement. We first apply average pooling over the HiT output and get the global representation vector v . After getting v , we apply a multi-layer perceptron (MLP) as the classifier to get the classification result.

The probability of the output label is then calculated with a softmax layer:

$$v = \text{MeanPooling}(H_1, H_2, \dots, H_l) \quad (12)$$

$$o = g(W_2 \cdot f(W_1 \cdot v + b_1) + b_2) \quad (13)$$

$$P_i = \frac{\exp(o_i)}{\sum_i \exp(o_i)} \quad (14)$$

We use the standard cross entropy loss to train our model:

$$\mathcal{L} = - \sum_{i=1}^{|Y|} \mathbb{1}_{y==i} \log P_i, \quad (15)$$

where $\mathbb{1}$ is the indicator function.

- (ii) **Generation** For generation tasks, models are required to generate the target sequence conditioned on the encoder output, such as method name prediction, code summarization and code completion. We pass all encoder output as a sequence to a transformer decoder. To generate out-of-vocabulary (OOV) tokens, we adopt a pointer network (Vinyals et al. 2015) based on the Transformer decoder model. The pointer model first attends to the encoder's output at each timestep and gets a hidden vector h_t^* .

$$e_t^t = W_3^T \tanh(W_1 H_t + W_2 s_t + b) \quad (16)$$

$$a^t = \text{softmax}(e^t) \quad (17)$$

$$h_t^* = \sum_i a_i^t H_i, \quad (18)$$

where W_1, W_2, W_3, b are learnable parameters, s_t represents the output of the transformer decoder at timestep t . After obtaining the context vector, the model produces the vocabulary distribution and the copy probability $p_{copy} \in [0, 1]$ with H_t and s_t at this timestep. The p_c denotes the probability of copying tokens from the input sequence. On the contrary, $p_{gen} = 1 - p_{copy}$ indicates the probability of generating a token from the vocabulary. The probability of predicting the token w is calculated as follows:

$$P_v = \text{softmax}(W_4 h_t^* + b_1) \quad (19)$$

$$P_{copy} = \text{sigmoid}(W_5 h_t^* + b_2) \quad (20)$$

$$P(w) = p_{gen} P_v(w) + p_{copy} \sum_{i:w_i=w} a_i^t. \quad (21)$$

The copying mechanism enables the model to enhance its predictions by pointing at positions in the input sequence. This copying mechanism is inspired by Vinyals et al. (2015) and is adapted into the Transformer architecture in our work. The copy probability in (19) is calculated as a gate mechanism where gate conditions are from the hidden vector h_t^* . During training, the loss for the output sequence is calculated as the average loss over the negative log-likelihood of each target token w_t , so that the weight of the copying mechanism will be also updated during this learning process:

$$\mathcal{L} = \frac{1}{T} \sum_{t=0}^T -\log \sum_{\tilde{w}_t \in vocab} \mathbb{1}_{\tilde{w}_t=w_t} P(\tilde{w}_t) \quad (22)$$

5 Experimental Setup

With the extracted hierarchy information in the code sequence, we adopt BPE-HiT and perform extensive evaluation upon three code understanding tasks involving classification and generation tasks across 8 different datasets. We aim to investigate six research questions:

RQ1. Compressed Hierarchy v.s. Original Hierarchy What is the impact of compressed hierarchy on code representation models compared with directly using the original hierarchy? For different types of hierarchies, how much improvement does compression bring?

RQ2. Performance on Variable Scope Detection Task Can BPE-HiT learn the variable scope information contained in the global hierarchy? Is it important for the code representation model to focus on the global hierarchy?

RQ3. Performance on Classification Tasks How does BPE-HiT perform compared with the SOTA models on the code classification and clone detection tasks?

RQ4. Performance on Generation Tasks How does BPE-HiT perform on generation tasks? Can BPE-HiT produce better results than SOTA models on the method name prediction and real-world code completion tasks?

RQ5. Ablation Study What is the impact of different compression strategies on the design of BPE-HiT? How do different hyperparameters affect the final performance? How do the layer number of the hierarchy encoder and the hierarchy dimension influence the performance of the downstream tasks? What is the impact of settings in our pre-training design?

RQ6. Time Efficiency Does the compressed hierarchy information in BPE-HiT make sequence-based code representation models more efficient? What is the training efficiency of BPE-HiT compared with the vanilla Transformer?

5.1 Subject Tasks and Datasets

Our experiments are conducted upon two representative source code classification tasks (*i.e.*, **code classification** and **clone detection**) and one generation task (*i.e.*, **method name prediction**). We evaluate 8 widely used datasets in total, and the compared baseline models are among the classical models or the SOTA models. The statistics of these popular datasets are summarized in Table 2.

In addition, to investigate the scope information in the global hierarchy learned by BPE-HiT, we also propose a new task called **variable scope detection** in our previous work (Zhang et al. 2023). We will give a detailed description about these tasks.

5.1.1 Variable Scope Detection

The variable scope detection task requires the model to detect whether these two variables are in the same scope in a program.¹ Formally, given the representation vector of the two

¹ The scope of a variable is a block structure in the entire program where the variable is declared, used, and can be modified.

Table 2 Statistics of datasets

(a) On Code Classification Task

	Code Classification				
	Java250	Python800	C++1000	C++1400	POJ-104
Size	Train	144,000	300,000	252,000	36,400
	Valid	15,000	100,000	84,000	5,200
Avg. Length	Test	15,000	100,000	84,000	10,400
	Token Seq	228.02	270.96	334.89	246.96
	Complete Hierarchy	9.26	7.14	7.62	9.15
	+BPE	2.74	2.88	3.11	3.19
	Global Hierarchy	7.15	4.75	5.06	6.22
	+BPE	2.23	2.21	2.98	2.97
	Local Hierarchy	2.11	3.61	2.56	2.92
	+BPE	1.76	1.86	1.94	1.93

(b) On Clone Detection and Method Name Prediction Tasks

	Clone Detection		Method Name Prediction	
	POJ-Clone	Problems	CSN-Ruby	CSN-Python
Size	Examples	64	48,791	412,178
	Train	32,000	2,209	23,107
Avg. Length	Valid	8,000	2,279	22,176
	Test	12,000	79.18	131.59
	Token Seq	246.96	2.23	2.25
	Target Seq	-	6.89	9.12
	Complete Hierarchy	9.15	2.54	2.84
	+BPE	3.19		

Table 2 continued

(a) On Code Classification Task				
	Global Hierarchy	6.22	5.62	5.69
	+BPE	2.97	2.26	2.45
	Local Hierarchy	2.92	1.26	3.42
	+BPE	1.93	1.17	1.96
(c) On Code Completion Task				
		Python150k		JavaScript150k
Size	Train	Examples	# User [†]	Examples
	Valid	76467	3303	69038
	Test	8004	367	8665
	Token Seq	38694	1925	41542
Avg. Length	Complete Hierarchy	596.70		631.41
	+BPE	8.93		11.32
	Global Hierarchy	2.91		3.17
	+BPE	5.95		8.39
	Local Hierarchy	2.47		2.55
	+BPE	2.98		2.93
		1.92		1.89

[†] “# Users” reflects code style or domain differences. Data is divided by different users to ensure non-overlapping datasets, thus maintaining test data accuracy

variables h_A , h_B , the probability $p_{scope_{A,B}}$ of variable A and B in the same scope is calculated by dot product following a sigmoid function:

$$p_{scope_{(A,B)}} = \text{sigmoid}(h_A W_s h_B) \quad (23)$$

where W_s is a learnable parameter. We provide examples of the task in Table 3. Considering that the variable scope is a mapping of the hierarchical structure information on variable tokens, this task requires the model to learn the accurate global hierarchical block structure for sequence tokens.

In our experiment, we adopt Python800 and C++1400 datasets in Project CodeNet. We sample variable pairs and extract about 7 million pairs for Python and 65 million pairs for C++ with balanced labels. The division of the dataset follows the settings suggested by CodeNet in the classification task in Table 2.

5.1.2 Code Classification

The code classification task requires the model to predict the category of the given code snippet based on the semantics. In particular, for selected datasets, solutions under each question correspond to a category. We consider using Project CodeNet (Puri et al. 2021) and POJ-104 (Mou et al. 2016). Project CodeNet contains over 14M code samples from two open judge platforms AIZU and AtCoder. It provides four large and challenging datasets for the code classification task, including **Java250**, **Python800**, **C++1000**, and **C++1400**. **POJ-104** is collected from another pedagogical online judge system with 104 programming problems. It has been used by many previous studies in code classification and clone detection tasks.

5.1.3 Clone Detection

The clone detection task requires the model to detect whether two pieces of code implement the same functionality. We adopt the **POJ-Clone** and follow the previous task settings (Lu et al. 2021). **POJ-Clone** is one of the most popular clone detection due to its high quality and difficulty. It contains 52,000 programs written in C language, and is split into 104 different algorithm problems. It aims to retrieve other programs that solve the same problem given a program. To test the generalization ability of different approaches, the training/validation/test is split based on the problems.

5.1.4 Method Name Prediction

In method name prediction, a method with its name masked is fed into the model, and the model needs to predict the original method name based on the given method body. Following existing work (Zügner et al. 2021), the method name prediction task requires the model to

Table 3 Examples for Variable Scope Detection in C++ dataset

Example Program	Variable Pair
<code>if (i % 2 == 0)</code>	(C, B)
<code>C -= B;</code>	<i>Same scope</i>
<code>else</code>	(C, A)
<code>A -= D;</code>	<i>Different scope</i>

summarize the functionality of the code and generate a proper method name to cover it, and this task can be regarded as the code summarization task. We experiment on two datasets introduced in the CodeSearchNet (CSN) Challenge (Husain et al. 2019; Zügner et al. 2021): **CSN-Python** and **CSN-Ruby**. These real-world datasets are obtained by scraping from public repositories across the most popular projects on GitHub. **CSN-Python** contains about 450k programs and **CSN-Ruby** contains about 50k programs. The dataset split is shown in Table 2. We follow the work of Husain et al. (2019) which splits the data based on the source repositories to avoid data leakage.

5.1.5 Code Completion

In the code completion task, a long code context is given to the model as the input, and the model needs to predict each code token. Considering that tokens in source code are different from those in natural language, code Tokens are always aligned with the specific type attribution corresponding to the abstract syntax tree. So we also require the model to predict the type of each code token at the same time and follow existing code completion studies (Sun et al. 2020; Kim et al. 2021). The type of each code token is extracted with the static analysis tool *Tree-sitter* (Brunsfeld et al. 2024). We use the **Python150k** and **JavaScript150k** datasets processed by Chirkova and Troshin (2021). Both datasets are constructed from the real-world programs from Github² and are widely used to evaluate code representation models. To avoid biased results, duplicate files are removed by the duplication list, and identical code files and functions are further filtered out. Both datasets are split into training/validation/testing sets with 60%/6.7%/33.3% based on GitHub usernames. We employ such a meticulous data cleaning approach to ensure that the results accurately reflect the performance of our method on real-world code completion tasks.

5.2 Evaluation Metrics

For variable scope detection task, we use the accuracy for this binary classification task for each variable pair in our constructed dataset.

For code classification task, we adopt the measure in Puri et al. (2021) and use the accuracy for this multiclass classification task. Accuracy is calculated as the ratio of correctly classified samples to the total number of samples:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \quad (24)$$

For clone detection task, we follow (Lu et al. 2021) and use the $MAP@R$ (Musgrave et al. 2020) score for evaluation. $MAP@R$ is defined as the mean of the average precision scores, each of which is evaluated to retrieve the most similar R samples given a query. For a code (query), R is the number of other codes in the same class and $R = 499$ in the POJ-Clone dataset:

$$MAP@R = \frac{1}{N} \sum_{i=1}^N \frac{\sum_{k=1}^R P(k) \times \text{rel}(k)}{R} \quad (25)$$

² <https://github.com/>

where N is the number of queries, $P(k)$ is the precision at rank k , and $\text{rel}(k)$ is an indicator function that equals 1 if the item at rank k is relevant, and 0 otherwise.

For method name prediction task, we adopted metrics in previous studies (Alon et al. 2019a, b; Zügner et al. 2021), which measure *precision*, *recall*, and *F1* on subtokens of generated method names. Specifically, for the pair of the target method name t and the predicted name p , the $\text{precision}(t, p)$, $\text{recall}(t, p)$, and $F1(t, p)$ score are computed as:

$$\begin{aligned}\text{precision}(t, p) &= \frac{|\text{subtoken}(t) \cap \text{subtoken}(p)|}{|\text{subtoken}(p)|} \\ \text{recall}(t, p) &= \frac{|\text{subtoken}(t) \cap \text{subtoken}(p)|}{|\text{subtoken}(t)|} \\ F1(t, p) &= \frac{2 \times \text{precision}(t, p) \times \text{recall}(t, p)}{\text{precision}(t, p) + \text{recall}(t, p)}\end{aligned}\quad (26)$$

For code completion task, we adopted metrics in previous studies (Li et al. 2018), which measure *token accuracy* and *type accuracy* for each generated token. Token accuracy is calculated as the ratio of correctly predicted tokens to the total number of tokens:

$$\text{Token Accuracy} = \frac{\text{Number of Correct Tokens}}{\text{Total Number of Tokens}} \quad (27)$$

Type accuracy is calculated as the ratio of correctly predicted token types to the total number of token types:

$$\text{Type Accuracy} = \frac{\text{Number of Correct Token Types}}{\text{Total Number of Token Types}} \quad (28)$$

5.3 Implementation Details

Hierarchy Extraction Parser and Compression Algorithm We extract the hierarchy information from the concrete syntax tree with *Tree-sitter* (Brunsfield et al. 2024), a parser generator tool. We adopt the hierarchy BPE algorithm on the training set for each dataset and task to extract frequent combinations and compress the original hierarchy. Based on the tool and algorithm, our approach is general and dependency-free enough to parse any programming language and extract the compressed hierarchy information. In our experiments, we have selected datasets in *C++*, *Java*, *Python*, *Ruby* for evaluation, showing the generality and effectiveness.

Model Implementation Our model is implemented based on the Pytorch framework. We conduct all experiments on a Tesla V100S GPU with 32GB of memory. We set the maximum epoch number to 50 and set up an early stopping strategy if results on the validation set do not improve in ten consecutive epochs. Each experiment is run five times with random seeds and then averaged for final results. For the non-pre-trained BPE-HiT, we set the embedding size and the hidden size to 256, and employ 8 heads in each transformer layer. We set the hierarchy dimension and token dimension the same in our experiments so their dimensions are each converted to 128 before concatenation. For the code classification task and the clone

detection task, the hierarchy encoder consists of 2 layers, and the sequence encoder consists of 6 layers. For the method name prediction task, the number of sequence encoder layers and decoder layers are set to 4 and 2. We use AdamW with a learning rate of $1e^{-4}$ and weight decay. We use spaces as the separator for the tokenizer and set the vocabulary size between 5000-8000 according to different tasks. For all baseline models, we retrain the given datasets to get more reliable results. We try to keep the hyper-parameters the same as baseline models for a fair comparison.

Pre-training Settings For the pre-trained model, We follow CodeBERT (Feng et al. 2020) and use 12 layers bidirectional Transformer as the base Transformer. The hierarchy encoder consists of 2 layers and we set the hierarchy dimension and token dimension the same. The total number of parameters of our pre-trained BPE-HiT is 129M, which is quite similar with CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2021). The pre-training code dataset we use includes 2.3M code functions from the dataset used in CodeBERT, which covers six programming languages (i.e. ruby, java, python, php, go and javascript).

6 Experimental Results

6.1 RQ1: Compressed Hierarchy vs. Original Hierarchy

To investigate the impact of the compressed algorithm we employed on the hierarchy information, we conduct an empirical study and feed our method with the compressed hierarchy and original hierarchy. In addition to experimenting with the full hierarchy, we also explore the global hierarchy and local hierarchy separately in both compressed and original settings. We perform the analysis on all datasets of three tasks. The experimental results are listed in Table 4. The row *BPE-HiT* and *HiT* refer to the compressed hierarchy and original hierarchy respectively. The row *global* and *local* under each setting represent that we only use the corresponding type of hierarchy for experiments. The column *Para* refers to the total number of parameters for different models. In the following experiments, we will continue to use the same amount of parameters.

On the basic program understanding task of the code classification task, we observe that BPE-HiT outperforms HiT on the five datasets with three different program languages. It shows that the compressed hierarchy can extract key structure information and enhance the code representation model to distinguish code semantics. The *MAP@R* of BPE-HiT with the compressed hierarchy on the clone detection task is improved by 1.91, 15.22 compared with HiT and Transformer. For the method name prediction task, BPE-HiT outperforms the original hierarchy by 0.53 and 0.92 in the F1-score on CSN-Ruby and CSN-Python datasets. Our further experiments on global and local hierarchy also show that the compressed hierarchy can both improve the two types of hierarchy information. The local hierarchy is comparable and sometimes more effective than the global hierarchy for code classification.

Experiments show that BPE-HiT outperforms the original HiT with different types of hierarchy, indicating that our compression algorithm is helpful and efficient for the sequence model to distinguish the semantics and functionalities of the programs. It is worth noting that the hierarchy BPE algorithm can not only improve the performance of code representation tasks but also reduce the computing overhead. In Fig. 10 and Table 2 we have shown that the compression algorithm can reduce the length of the hierarchy, which reduces the time and

Table 4 Performance of BPE-HiT, HiT and Transformer with different types of hierarchy for all three tasks

(a) On Code Classification and Clone Detection Tasks

	Para	Code Classification (Accuracy)					POJ-Clone (MAP@R)
		Java250	Python800	C++1000	C++1400	POJ-104	
Transformer	4.50M	93.49	93.99	89.93	67.87	88.13	67.15
BPE-HiT	4.55M	94.85	96.04	95.16	93.58	97.31	82.37
<i>global</i>	4.55M	93.94	95.82	92.81	88.04	94.80	74.87
<i>local</i>	4.55M	94.12	95.90	93.14	89.18	97.06	69.62
HiT	4.55M	94.81	95.97	95.05	93.27	97.08	80.46
<i>global</i>	4.55M	93.79	94.84	91.35	83.90	94.56	74.85
<i>local</i>	4.55M	93.95	95.42	92.64	90.45	96.37	75.88

(b) On Method Name Prediction Task

	Para	CSN-Ruby			CSN-Python		
		P	R	F1	P	R	F1
Transformer	34.89M	24.26	19.66	21.71	32.71	27.63	29.96
BPE-HiT	36.75M	30.97	28.33	29.59	38.31	34.55	36.33
<i>global</i>	36.75M	24.96	23.01	23.95	34.47	28.99	31.49
<i>local</i>	36.75M	29.26	25.79	27.42	35.42	30.70	32.89
HiT	36.75M	30.70	27.58	29.06	37.25	33.75	35.41
<i>global</i>	36.75M	24.90	22.89	23.87	34.26	29.29	31.58
<i>local</i>	36.75M	28.69	25.58	27.05	35.34	30.50	32.74

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

computing resources required for the model to process hierarchy information. We will give a detailed analysis of time efficiency in Section 6.6.

Through compression, the hierarchical structure becomes more streamlined by encoding frequent combinations into fundamental units. Each of these combinations within the hierarchy represents a distinct semantic module. These combinations are arranged with each other to form a variety of unique code structures with different functionalities. Simultaneously, these combinations also establish connections, linking different code structures due to their shared hierarchy combinations. Experimental results show that the compressed hierarchy plays significant roles in different downstream tasks, which also confirms that extracting key structure information of source code is essential for code representation.

Furthermore, We also notice that the performance of BPE-HiT to using global hierarchy and local hierarchy separately is worse than when using the complete compressed hierarchy. Therefore, we can show that both global and local compressed hierarchical embeddings are essential for code representation. We use the complete compressed hierarchy to feed BPE-HiT in our follow-up experiments.

Answer to RQ1: The empirical study of classification and generation tasks demonstrates the impact of our compression algorithm. The compressed hierarchy focuses on the key structural information of source code and plays a significant role in the code representation.

6.2 RQ2: On Variable Scope Detection Task

We use our trained model on Python800 and C++1400 datasets for further analysis. We compare BPE-HiT, HiT, and Transformer models and evaluate their performance on the variable scope detection task. Our compared baselines also include GGNN and GREAT as they are two classical code representation models that are specially designed to encode structural information. The selection details of baselines are discussed in Section 7.1. We extract the variable representation vector from the encoder output of trained models in the classification task. For sequence models, we extract the corresponding token representation. For graph-based models, we extract from node representations. Experiment results are shown in Table 5.

In this experiment, we select Python and C++ as they represent two highly characteristic programming languages. Python's structured definitions are not as explicit as those in Java or C, where scope is often delineated by clear braces. This makes Python a suitable candidate for testing variable scope detection in a language with less obvious structural markers. On the other hand, C++1400 is one of the most complex datasets in CodeNet, featuring intricate code structures. This complexity makes it an excellent choice for evaluating variable scope detection in a challenging environment. By using these two datasets, we aim to cover a broad spectrum of code complexity and structure, thereby demonstrating the robustness and versatility of our proposed models in handling variable scope detection tasks.

Results show that with the hierarchy information, our model can understand the hierarchical block structure better and achieve the best performance on the variable scope detection task. We can observe that BPE-HiT and HiT perform even better on the C++ dataset by at least 6.7% compared with other baselines. Our in-depth investigation of the dataset reveals that the programs in the C++ dataset are much longer and the relationship between variables is more complex compared with the Python dataset. The program graphs in the C++ dataset for tree-based and graph-based models are large. The number of nodes in the receptive field of each node grows exponentially, which leads to these models not being able to understand the complete sequence information well. Our model retains the advantage of the sequence models to capture long-term semantic dependency and shows the importance of learning the scope information in global hierarchy.

Notably, results also reveal that employing compressed hierarchy information enhances the original HiT model's performance, particularly on the Python800 dataset. We explore the Python dataset and find that there are many shared hierarchy combinations. The application of compressed hierarchy effectively represents these shared combinations into unified units, thereby enhancing the model to learn the accurate hierarchical block structure for sequence tokens.

Table 5 Performance of models on variable scope detection (Accuracy)

Model	Python	C++
Transformer	77.18	63.77
GGNN	79.81	68.94
GREAT	79.39	69.38
BPE-HiT	82.08	77.91
HiT	80.27	76.12

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

Answer to RQ2: We design the variable scope detection task to explore the scope information learned by code representation models. Our BPE-HiT and HiT broadly gain better performance compared with the vanilla Transformer and the graph-based model. Experiments show that BPE-HiT can retain the advantage of sequence models to capture long-term semantic dependency and enhance sequence models with the scope information in the global hierarchy.

6.3 RQ3: On Classification Task

6.3.1 RQ3.1: Code Classification

We compare with the following state-of-the-art code representation models:

- (1) *Graph Neural Networks* We include RGCN (Schlichtkrull et al. 2018) and GGNN (Li et al. 2016) as baseline models.
- (2) *Tree-structured Neural Networks* We include TBCNN (Mou et al. 2016), ASTNN (Zhang et al. 2019) and TreeCaps (Bui et al. 2021) as baseline models.
- (3) *Traversal Sequences of ASTs* We compare our model with SBT (Hu et al. 2018) and XSBT (Niu et al. 2022). SBT (Hu et al. 2018) represents the tree nodes into a token sequence with brackets to denote hierarchies. X-SBT (Niu et al. 2022) simplifies SBT by using an XML-like form.
- (4) *Transformer-based Models* In addition to the vanilla transformer, we also compare our model with GREAT (Hellendoorn et al. 2020). For comparison, we also include results for CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2021), both of which are widely used pre-trained code models. Specifically, our pre-trained BPE-HiT model has a comparable parameter scale to these two models.

The results in Table 6 show that our non-pre-trained BPE-HiT achieves the best performance on the code classification task over all non-pre-trained baseline models across different program languages. The overall average accuracy on four datasets in Project CodeNet is at least 1.06% higher than other baseline models, and the accuracy on the POJ-104 dataset is increased by at least 0.51% compared with the SOTA models. Compared with the original HiT, our proposed BPE-HiT achieves improvement by 0.04%, 0.07%, 0.11%, 0.31%, and 0.23% on five datasets, indicating the effectiveness of the compressed hierarchy on code classification tasks.

We observe that the graph-based or tree-based models are more effective than the vanilla Transformer. It proves that using only the token sequence, the sequence model cannot learn code representation well. Our proposed BPE-HiT addresses this issue for sequential models. With the enhancement of hierarchy information, BPE-HiT outperforms those graph/tree-based models. We also notice that directly feeding the traversal sequences of ASTs in the Transformer performs poorly. The flattened AST node sequence impairs the sequential information of the context in the source code. In comparison, our approach is much more efficient and effective in combining naturalness and hierarchy for code representation.

For pre-trained settings, our BPE-HiT-pretrain model achieves superior performance compared to other pre-trained models such as CodeBERT and GraphCodeBERT. Specifically, the BPE-HiT-pretrain model shows significant improvements in accuracy across all evaluated datasets. These results highlight the efficacy of incorporating compressed hierarchy in the pre-training phase. By leveraging both token-level and hierarchy-level information,

Table 6 Performance of models on code classification task

Model	Java250	Python800	C++1000	C++1400	POJ-104
<i>Non-pre-trained Model</i>					
RGCN	91.93	91.60	92.73	92.34	95.57
GGNN	93.64	92.23	91.72	92.48	94.80
TBCNN	92.84	93.17	94.77	88.29	96.20
ASTNN	92.86	93.80	94.61	90.17	96.79
TreeCaps	93.07	94.35	94.92	90.16	96.81
SBT	65.64	71.69	65.05	56.33	89.58
X-SBT	83.31	89.10	66.79	67.00	94.58
Transformer	93.49	93.99	89.93	67.87	88.13
GREAT	93.36	93.27	92.76	92.50	90.33
BPE-HiT	94.85	96.04	95.16	93.58	97.31
HiT	94.81	95.97	95.05	93.27	97.08
<i>Pre-trained Model</i>					
CodeBERT	96.47	97.41	86.13	83.05	98.40
GraphCodeBERT	97.31	97.09	93.35	91.63	98.41
BPE-HiT-pretrain	97.59	97.93	96.45	94.86	98.45

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

BPE-HiT-pretrain can better capture the semantics of source code, leading to more accurate and reliable code representations. The comparable parameter scale of BPE-HiT-pretrain to CodeBERT and GraphCodeBERT ensures that the improvements are due to the enhanced model architecture and training strategies rather than an increase in model size. We also notice sequence models often struggle on C++1000 and C++1400 datasets, including pre-trained models. Further investigations are conducted in Section 7.2.

6.3.2 RQ3.2: Clone Detection

To further verify the generalization ability of the model in distinguishing program semantics, we choose to evaluate our model on a clone detection dataset (POJ-Clone) for further evaluation. As we mentioned, the clone detection dataset is partitioned into training, validation, and test sets with different OJ problems. We compare our model with several state-of-the-art methods specially designed for code clone detection tasks.

- (1) *Code2vec/Code2seq* (Alon et al. 2019a,b) uses the attention-based method with leaf-to-leaf paths of AST to learn embeddings of codes.
- (2) *NCC* (Ben-Nun et al. 2018) encodes programs by leveraging both the underlying data flow and control flow of the programs with LSTM to build a code similarity system.
- (3) *Aroma* (Luan et al. 2019) is a code recommendation engine with the simplified parse tree (SPT).
- (4) *TBCCD* (Yu et al. 2019) is a clone detection model with tree-based convolution networks. It achieves state-of-the-art performance on a simplified version dataset of clone detection.³

³ In the authors' paper, they evaluate their model on a clone detection dataset that is not partitioned based on OJ problems. The program semantics in the training and test sets are the same. However, this setting weakens the generalization ability of the model.

- (5) *MISIM* (Ye et al. 2020) is a code clone detection system that incorporates the context-aware semantics structure (CASS) in its design. This structure has been carefully tailored to support the analysis of specific programming languages.

We list the results in Fig. 11. In most cases, our non-pre-trained BPE-HiT outperforms baseline models, which improves by at least 27.25 in *MAP@R* except MISIM models. The compressed hierarchy also improves the original HiT by 1.91, which is very close to the performance of the extremely large pre-trained model CodeBERT. The MISIM models need to be evaluated on every possible combination of manually designed configurations (Ye et al. 2020), thus, the preprocessing process is complex. Our method is simple and easy to use, with comparable results with the best MISIM-GNN among those models. When conducted on sequence models, our performance is even better than MISIM-RNN.

We observe that in our challenging experimental setting, TBCCD performs poorly. We also notice that NCC, Aroma, and MISIM both require complex preprocessing designed for particular languages. Our hierarchy extraction process is based on CSTs and the hierarchy compression algorithm is language-agnostic, showing that BPE-HiT is more generalizable and practical compared with existing baselines.

We further compare our pre-trained BPE-HiT with pre-trained baselines on the clone detection task. Experiments show that our pre-trained model significantly outperforms these popular baseline models, demonstrating the effectiveness and efficiency of our hierarchy-aware pre-training framework. These results indicate that our proposed BPE-HiT architecture is reliable and robust across various code representation tasks.

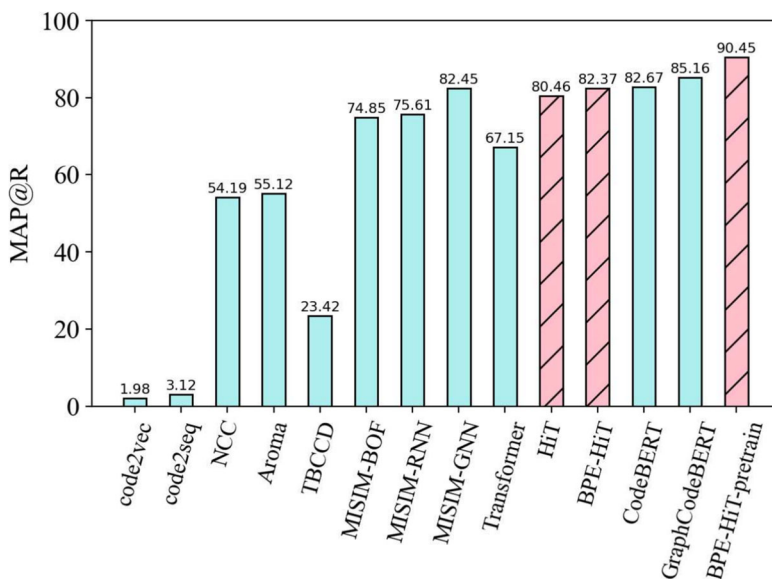


Fig. 11 Performance of models on POJ-Clone (The results of baselines are from the CodeXGLUE benchmark)

Answer to RQ3: Our proposed model BPE-HiT generally outperforms the current SOTA baseline models upon code classification and clone detection tasks. The compressed hierarchy can enhance the code representation model to understand program semantics and handle the classification tasks well. Experiments also demonstrate that our proposed BPE-HiT architecture is reliable and robust in both non-pre-trained and pre-trained settings across various code representation tasks.

6.4 RQ4: On Generation Task

6.4.1 RQ4.1: Method Name Prediction

To answer this RQ, in addition to the baselines mentioned, we also compare with the SOTA models on the method name prediction task, including CodeTransformer (Zügner et al. 2021) and GTNM (Liu et al. 2022):

- (1) *CodeTransformer* learns structure and context jointly, and achieves state-of-the-art performance on CSN datasets.
- (2) *GTNM* is a latest transformer-based model for method name prediction which extracts local contexts and project-level contexts and incorporates them into the sequence model. The original paper of GTNM uses contexts designed for Java that are not available for CSN-Ruby and CSN-Python. We use a variant of GTNM that only considers the local context for fairly comparison. In this experimental setup, we did not cover the tree models included in Table 6, as we found that they did not perform well on this task. We use the same copy mechanism of BPE-HiT and HiT for all baseline models.

We list the results of our BPE-HiT with the baselines upon CSN-Ruby and CSN-Python in Table 7. The experimental results show that our BPE-HiT outperforms the baseline models significantly upon CSN-Ruby and CSN-Python. Specifically, our non-pre-trained model outperforms CodeTransformer by 2.04, and 1.34 in F1-score, respectively. Our non-pre-trained model also significantly outperforms GTNM by 7.46, 6.20. Our pre-trained BPE-HiT achieves the best performance across all baseline models.

6.4.2 RQ4.2: Real-world Code Completion

To demonstrate the generalization and effectiveness of our proposed model architecture on generation tasks, we further evaluate our model on a real-world complex generation scenario — the code completion task. Considering that this task is a complex code sequence generation task which is challenging for tree-based or graph-based methods, we primarily compare against powerful Transformer-based baselines, including GREAT (Hellendoorn et al. 2020) and CodeTransformer (Zügner et al. 2021).

In this real-world experiment settings, we can observe that the compression method helps a lot to combine those frequent patterns in the code hierarchy. In Table 2 we can observe that for both the real-world Python150k and JavaScript150k datasets the compression method can significantly reduce the length of the hierarchy. We further list the results of our BPE-HiT alongside the baselines on Python150k and JavaScript150k in Table 8 to show the improvements of our architecture and compressed method. The experimental results show that

Table 7 Results of models on method name prediction task

Model	CSN-Ruby			CSN-Python		
	P	R	F1	P	R	F1
<i>Non-pre-trained Model</i>						
Code2seq	23.23	10.31	14.28	35.79	24.85	29.34
GGNN	19.15	14.11	16.24	24.07	19.09	21.29
SBT	19.84	12.22	15.12	30.92	18.32	23.01
X-SBT	22.82	13.04	16.60	34.58	20.69	25.89
Transformer	24.26	19.66	21.71	32.71	27.63	29.96
GREAT	24.66	22.25	23.39	35.09	31.62	33.26
CodeTrans	31.46	24.50	27.55	36.41	33.68	34.99
GTNM	24.59	20.11	22.13	32.98	27.73	30.13
BPE-HiT	30.97	28.33	29.59	38.31	34.55	36.33
HiT	30.70	27.58	29.06	37.25	33.75	35.41
<i>Pre-trained Model</i>						
CodeBERT	32.65	26.52	29.27	38.51	30.14	33.81
GraphCodeBERT	34.54	27.81	30.81	38.85	33.22	35.82
BPE-HiT-pretrain	35.95	29.46	32.38	39.87	35.39	37.50

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

our BPE-HiT outperforms the baseline models significantly on both datasets. Specifically, our model achieves higher type and token accuracy compared to CodeTransformer and GREAT, demonstrating its superior ability to handle the complexities of code completion tasks. In particular, the non-pre-trained BPE-HiT outperforms CodeTransformer by 2.74% in type accuracy and 3.23% in token accuracy on the Python dataset, and by 2.10% in type accuracy and 1.73% in token accuracy on the JavaScript dataset. The pre-trained BPE-HiT also achieves the best performance compared with all baselines. This significant improvement highlights the effectiveness of incorporating hierarchical representations in our model architecture.

Table 8 Results of models on code completion task

Model	Python		JavaScript	
	Type Acc	Token Acc	Type Acc	Token Acc
<i>Non-pre-trained Model</i>				
Transformer	81.91	49.56	82.73	57.40
GREAT	81.95	49.74	83.08	57.98
CodeTransformer	83.41	50.48	83.81	58.24
BPE-HiT	86.15	53.71	85.91	59.97
HiT	85.64	52.04	84.32	59.51
<i>Pre-trained Model</i>				
CodeBERT	85.62	55.04	84.68	58.99
GraphCodeBERT	86.51	56.84	85.93	59.75
BPE-HiT-pretrain	87.87	55.19	86.05	60.52

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

Answer to RQ4: Our proposed approach BPE-HiT outperforms all classical code representation models and current SOTA baselines on both CSN-Ruby and CSN-Python method name prediction tasks, as well as on the Python150k and JavaScript150k code completion tasks. This suggests that BPE-HiT is capable of producing superior code representations for generation tasks across different programming languages and scenarios. Specifically, BPE-HiT demonstrates significant improvements in both type and token accuracy for code completion, highlighting its effectiveness in handling complex code sequence generation tasks. These results underscore the potential of BPE-HiT to enhance the performance of various code understanding and generation applications.

6.5 RQ5: Ablation Study

To answer RQ5, we investigate the impact of different compression strategies on the hierarchy information. We also conduct ablation studies to explore the influence of hyper-parameters, including the layer number of the hierarchy encoder and the hierarchy dimension. For these analyses, we primarily use the Python800 and C++1400 datasets. Their substantial size provides a robust basis for evaluating code classification and representation tasks, ensuring our models are tested across a broad spectrum of code complexities.

6.5.1 Compression Strategy

We first explore the impact of different compression strategies on the hierarchy information to show the effectiveness of our proposed compression algorithm Hierarchy BPE. We perform ablation experiments on different compression strategies on the Python800 dataset in Table 9. HiT uses the original hierarchy and BPE-HiT uses the compressed hierarchy with our algorithm in Section 4.3. We also compare with the following strategies:

- (1) *Global-Local Compression* We regard each global hierarchy and local hierarchy as a basic unit respectively so the complete hierarchy is transformed into a sequence with $length = 2$. This global-local compression strategy considers the global and local hierarchy as a combination.
- (2) *Total Compression* We regard each complete hierarchy as a basic unit respectively so it is transformed into a sequence with $length = 1$. This total compression strategy considers each complete hierarchy as an independent combination, which can be viewed as an extreme compression method.

Table 9 Performance of different compression strategies on Python800 dataset (Accuracy)

Model	Accuracy	avg. Hierarchy Length
HiT	95.97	7.66
BPE-HiT	96.04	2.51
Global-Local Compression	95.87	2
Total Compression	94.39	1
Transformer	93.99	–

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

We notice that all compression strategies outperform the Transformer model, indicating the importance of the hierarchy information for code representation. Among these strategies, BPE-HiT with our proposed compression algorithm achieves the best performance. We can prove that the improvement of BPE-HiT is not only from the reduction of hierarchy length. The hierarchy length obtained by the global-local compression strategy is similar to that of BPE-HiT, but its performance is significantly reduced compared with the HiT model. Our proposed hierarchy BPE algorithm can extract frequent combinations and highlight key information. These key hierarchy information can effectively enhance the code representation model.

Experiments also show that excessive compression is undesirable. The total compression strategy can be viewed as an extreme compression method, and it achieves the worst performance of these strategies. Excessive compression reduces the common features between different codes, making it difficult for models to learn and understand code semantics.

We observe that our proposed compression algorithm retains semantic information effectively. Compared to the original hierarchy in *HiT*, the *global-local compression* and *total compression* strategies, which reduce the hierarchy length, result in performance degradation. Directly shortening the hierarchy loses important information embedded in the hierarchical structure of the source code. In contrast, our proposed compressed hierarchy in BPE-HiT demonstrates a performance improvement. As shown in Table 1 and Fig. 7, the mined compressed hierarchy reveals strong semantic patterns. These patterns effectively capture the semantics and establish connections within the corresponding code structures.

6.5.2 Layer Number of the Hierarchy Encoder

We further investigate the influence of the layer number of the hierarchy encoder in Section 4.4 in our experiments. In our experiments, we set the layer number to 2. We conduct a detailed ablation experiment with five different settings, varying from 1 to 5. We choose the Python800 dataset and our experiment results are shown in Fig. 12(a).

Experiments show that the model performance tends to rise first and then decline as the number of layers increases. The model achieves the best results when the number of layers is 2. More layers in the hierarchy encoder will result in a slight decrease in the final performance. We attribute it to the fact that the compressed hierarchy is a short sequence, and a too deep hierarchy encoder may lead to overfitting. We also conduct the same hyper-parameter ablation studies on other datasets and observe a similar tendency.

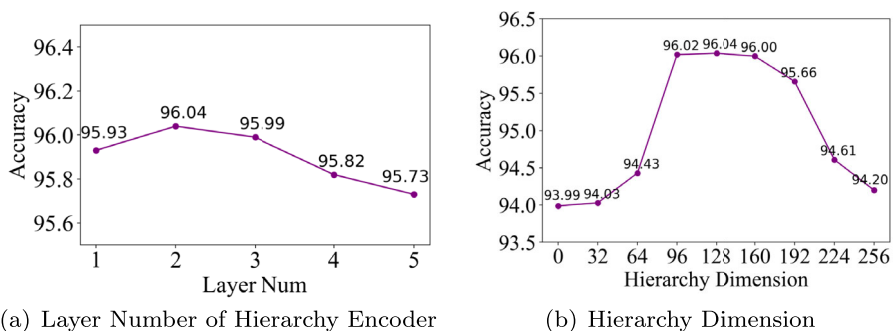


Fig. 12 Ablation studies on Layer Number of Hierarchy Encoder and Hierarchy Dimension on Python800 dataset

It is also worth noting that such a model setting will not significantly increase the computational burden of the original Transformer model. We list the parameter cost to compare our BPE-HiT with the vanilla Transformer in Table 4. The column *Para* refers to the total number of parameters for different models. With the proper setting of the hierarchy encoder, our model significantly enhances the original sequence model on different tasks, at a minimal extra parameter cost of 1%-5%.

6.5.3 Hierarchy Dimension

In our experiments, we set the hierarchy and token dimension the same so their dimensions are each converted to 128 before concatenation in Section 4.4. It leads to a question: What if we change the hierarchy and token dimension to control the proportion of these two kinds of information? We add an ablation experiment to answer this question. We choose the Python800 dataset and increase the hierarchy dimension from 0 to 256 so that the token dimension reduces from 256 to 0 during this process. Our experiment results are shown in Fig. 12(b). When the hierarchy dimension is 0, the model is the vanilla transformer model, which only accepts code tokens as model input. When the hierarchy dimension is 256, the model will only accept the compressed hierarchy as model input.

Experiment results show that when we increase the hierarchy information in the code representation model, the performance will first increase and then decrease. It proves that both the hierarchy information and sequence information are important for code representation tasks. Too much or too little hierarchy information ratio will lead to a significant performance drop. When we control the hierarchy dimension between 96 and 160, the performance of the model reaches the optimal level and the changes are relatively smooth. The key to the improvement of our BPE-HiT is to balance the hierarchy information and sequence information in the model so that the model can comprehensively learn a more accurate code representation.

6.5.4 Hierarchy-Aware Pretraining Objective

To further explore the impact of our hierarchy-aware pretraining framework, we conduct an ablation study on the code classification task using various datasets. The results are shown in Table 10. We compare the performance of the full BPE-HiT-pretrain model with two variations: one without the Masked Hierarchy Modeling (MHM) objective and one without any pre-training. The results indicate that the full BPE-HiT-pretrain model consistently outperforms the variations across all datasets. Specifically, the absence of the MHM objective results in a noticeable performance drop, underscoring the importance of incorporating hierarchical information during pre-training. Additionally, the lack of pre-training leads to a further decrease in performance, highlighting the critical role of pre-training in enhancing model capabilities.

Table 10 Ablation study on Hierarchy-aware Pretraining Objective on Code Classification Task

Model	Java250	Python800	C++1000	C++1400	POJ-104
BPE-HiT-pretrain	97.59	97.93	96.45	94.86	98.45
w/o MHM objective	96.55	97.36	95.25	94.56	98.41
w/o pre-training	94.85	96.04	95.16	93.58	97.31

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

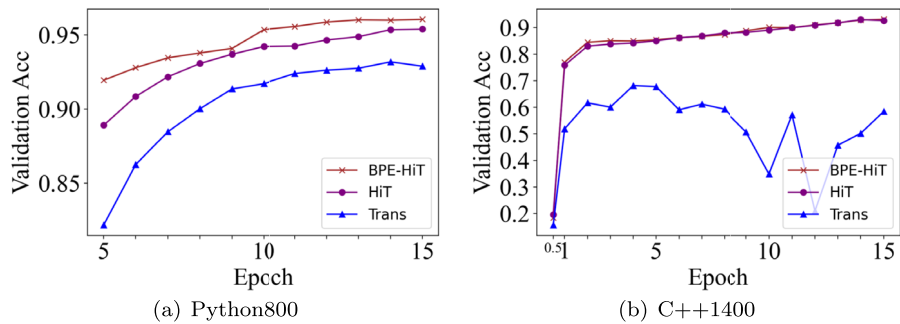


Fig. 13 The validation accuracy during training of *BPE-HiT*, *HiT* and *Transformer* on Python800 and C++1400 datasets. We have selected a part of the records during the training process to show it more clearly

Answer to RQ5: We compare our hierarchy BPE algorithm with different compression strategies. Experiments confirm the importance of extracting key structure information of source code. Our ablation also demonstrates the impact of different hyper-parameters in BPE-HiT. We have conducted hyperparameter search experiments to ensure the effectiveness and efficiency of our model. Additionally, our ablation study on the hierarchy-aware pre-training objective shows that our proposed pre-training framework significantly enhances performance.

6.6 RQ6: Time efficiency

To answer RQ6, we recorded the training process of BPE-HiT, HiT, and Transformer. We selected two representative datasets: Python800 and C++1400. Figure 13 shows the validation accuracy during training. Table 11 shows the required training time and epochs to achieve the best performance for different models on Python800.

Results in Fig. 13 show the training process of our BPE-HiT and HiT is faster and more stable. On Python800 dataset, BPE-HiT can achieve comparable results to the Transformer with fewer training epochs and time. Compared with HiT, our BPE-HiT also improves training efficiency. For difficult datasets such as C++1400, the performance of the Transformer is very fluctuating during training. In comparison, BPE-HiT shows more stable learning ability, making it perform more efficiently on such datasets. We conduct an in-depth analysis for the reason why the sequence model is unstable on C++1400 dataset in Section 7.2.

Table 11 Training time and epochs required for *BPE-HiT*, *HiT*, and *Transformer* to achieve optimal performance on Python800 dataset

Model	Final Accuracy	Total Time(min)	Epoch	Time per Epoch(min)
Transformer	93.99	169	29	5.8
BPE-HiT	96.04	105	15	6.9
HiT	95.97	122	16	7.6

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

We further compare the total training time of these three models in Table 11. Our BPE-HiT can achieve the best performance on Python800 dataset with the least training time and the number of training epochs. Results show that with the hierarchy information enhanced, the code representation model can achieve the best results with fewer training rounds. Compared with Transformer, the BPE-HiT model can save 37% of the total training time and 48% of the number of training epochs. For the training time for each epoch, the additional time required by BPE-HiT can also be controlled within a smaller range. This demonstrates the efficiency of BPE-HiT, especially at training time.

We also notice that when compared with HiT, the BPE-HiT can reduce the training time for each epoch. We attribute it to the fact that with the hierarchy compression algorithm, we can significantly reduce the hierarchy length, thus reducing the computational overhead.

Answer to RQ6: The compressed hierarchy is capable of improving the time efficiency and training stability of sequence-based code representation models. Our BPE-HiT slightly increases the training time required at each epoch but reduces the total time and total number of training epochs. Compared with HiT, our BPE-HiT with compressed hierarchy can significantly reduce the hierarchy length, thus reducing the computational overhead.

7 Discussions

7.1 Comparison with Existing Sequence/Structure-Based Models

Due to the design of the model structure, the existing sequence model (*e.g.*, the vanilla Transformer (Vaswani et al. 2017)) or structural-based model (*e.g.*, ASTNN (Zhang et al. 2019), TBCNN (Mou et al. 2016), GGNN (Li et al. 2016)) is usually better at capturing the information of a certain modal. Some studies jointly learn both sequential and structural information for code representation in sequence-based models such as GREAT (Hellendoorn et al. 2020). They focus on modeling structure as a relation between tokens with attention mechanism and overlook the full impact of hierarchical structure, especially for the global hierarchy. In this paper, we comprehensively investigate the impact of different types of hierarchy information on code representation tasks. In Section 6.2, we design a Variable Scope Detection task to check the ability of different models to capture the global hierarchy information. We choose one of the sequence models, structural-based models and jointly learning models as the baseline. Experiments show that our model can better identify the information brought into the hierarchy. Our model retains the advantage of the sequence models to capture long-term semantic dependency and shows the importance of implanting the full hierarchy information.

7.2 Analysis of C++ Datasets in Project CodeNet

We further investigate the reason why the sequence model does not perform well on the C++ dataset. Analysis of the code sequences in these C++ datasets revealed that they are longer (as shown in Table 2). Upon further investigation, we treat the programs from the same class as a single text snippet by concatenating them and calculate the TF-IDF cosine similarity between every two classes (Wang et al. 2022). The experiment results are shown

in Table 12. The average similarity for C++1000 and C++1400 is 0.787 and 0.754, while the average score for Java250 and Python800 is only 0.722 and 0.420. This suggests that code sequences in C++ datasets are highly similar even in different classes, making it difficult for sequence models to accurately classify them. Although CodeBERT has been pre-trained on a pretty large code corpus and its parameter number is nearly 27 times that of BPE-HiT, it still performs poorly on these harder datasets. Our BPE-HiT can alleviate this problem by fusing sequence information and hierarchy information to get a more accurate vector representation. BPE-HiT shows strong training stability, especially on such difficult datasets.

7.3 Confidence Evaluation for All Experiments

We also conduct A/B (Kohavi and Longbotham 2017) Testing to show the significance of the experimental results. Specifically, we employ A/B testing to determine whether our methods outperform the other baselines with confidence scores.

A/B testing involves comparing the performance of our models against baseline models by randomly splitting the datasets and evaluating the models' performance on these splits. We use statistical significance tests to measure whether the observed performance differences are significant. The key metric for this evaluation is the p-value, which indicates the probability that the observed difference occurred by chance. A p-value less than 0.05 is typically considered statistically significant, suggesting that our model's performance is reliably better than the baseline.

We conduct A/B testing for **all experiments**. And here we show CodeNet dataset as an example. The results are shown in Table 13. We observe that the p-values are less than 0.05 for all experiment results, including code understanding tasks and generation tasks. These results, supported by statistical significance testing, provide strong evidence that our models not only improve accuracy but do so with high confidence, making them reliable choices for various code-related tasks.

7.4 Threats to Validity

Threats to internal validity relate to the roles of the model architecture and hyper-parameter settings. In our experiments, we performed a small-range grid search on learning rate and batch size settings to determine the optimal configuration. Although this method helps to identify suitable parameters, it may not cover all possible configurations, potentially missing more optimal settings. Additionally, another threat arises from the implementation of GTNM. We did not use additional project and document-level context for method name prediction as described in the original paper (Liu et al. 2022) because such contexts are not available

Table 12 The TF-IDF similarity and sequence model performance on CodeNet datasets

	Java250	Python800	C++1000	C++1400
Similarity	0.722	0.420	0.787	0.754
Accuracy				
Transformer	93.49	93.99	89.93	67.87
CodeBERT	96.47	97.41	86.13	83.05
BPE-HiT	94.85	96.04	95.16	93.58

The bold entries indicate the best performance achieved in the given tasks, highlighting the comparison between our results and other baselines

Table 13 A/B testing for models on CodeNet dataset

Model	Java250 p-value	Python800 p-value	C++1000 p-value	C++1400 p-value
<i>Non-pre-trained Model</i>				
RGCN	< .0001	< .0001	< .0001	< .0001
GGNN	< .0001	< .0001	< .0001	< .0001
TBCNN	< .0001	< .0001	< .0001	< .0001
ASTNN	< .0001	< .0001	< .0001	< .0001
TreeCaps	< .0001	< .0001	< .0001	< .0001
SBT	< .0001	< .0001	< .0001	< .0001
X-SBT	< .0001	< .0001	< .0001	< .0001
Transformer	< .0001	< .0001	< .0001	< .0001
GREAT	< .0001	< .0001	< .0001	< .0001
BPE-HiT	-	-	-	-
HiT	0.0036	0.0021	< .0001	< .0001

P-values are calculated by comparing with the non-pre-trained BPE-HiT

in the CSN-Ruby/Python datasets. This limitation might affect the model's performance in method name prediction, suggesting that the results could be further improved with richer contextual information.

In our experiments, we also conduct a pre-trained version of our BPE-HiT model. Considering that the training cost is exceedingly high, we adopted the existing experiences from works like CodeBERT, choosing the same or similar training data and parameters. We took robustness and safety issues into account as much as possible during the training process, and performed comparisons under the same evaluation conditions in the final experiments.

Threats to external validity mainly relate to the tasks and datasets we chose for evaluation. To mitigate this threat, we evaluated our model on ten different datasets spanning three distinct tasks, including both classification and generation tasks across four programming languages. These datasets provide a broad spectrum of code structures and complexities, enhancing the generalizability of our findings. However, further validation on additional datasets and other programming languages would be beneficial to confirm and extend the applicability of our approach. Furthermore, real-world scenarios might present different challenges that are not fully captured by our selected datasets.

Threats to construct validity include the evaluation metrics we used in this work. We selected metrics that are widely regarded as standard for the corresponding tasks, such as precision, recall, and F1 score for method name prediction, as well as accuracy and MAP@R for other tasks. These metrics have been adopted by many previous studies (Puri et al. 2021; Lu et al. 2021; Musgrave et al. 2020; Alon et al. 2019a,b; Zügner et al. 2021), ensuring that our evaluation measures are robust and widely accepted. Nonetheless, the chosen metrics may not capture all aspects of model performance. For instance, while precision and recall are useful for assessing method name prediction, they might not fully reflect the semantic quality of the predicted names. Additional qualitative evaluations and user studies could provide deeper insights into the practical utility and effectiveness of our models.

8 Conclusion

In this paper, we analyze how the hierarchical structure influences tokens in code sequence representation and put forward the property of hierarchical embedding. We explore the frequent hierarchy combinations contained in these hierarchical structures, and our detailed analysis shows they can represent strong semantics and are essential for efficient code representation. To extract these frequent combinations, we propose a novel compression algorithm Hierarchy BPE. Based on the compression algorithm, we propose the Byte-Pair Encoded Hierarchy Transformer (BPE-HiT), a simple but effective sequence model that incorporates the compressed hierarchical embeddings of source code into a Transformer model. Given that BPE-HiT significantly reduces computational overhead, we scale up the model training phase and implement a hierarchy-aware pre-training framework. Our in-depth evaluations demonstrate that our non-pre-trained BPE-HiT can generate accurate and delicate representations and outperforms the SOTA baselines for classification and generation tasks on 10 challenging datasets. Besides, our pre-trained BPE-HiT outperforms other pre-trained baseline models with the same number of parameters over all experiments, demonstrating the robust capability of our approach. We also strengthen the experiments by adding more ablation studies, proving the effectiveness of our compression algorithm and the training efficiency of our proposed model. To the best of our knowledge, this paper takes the very first step to explore the concept of incorporating the popular byte-pair encoding algorithm into the structural representation of source code, shedding light on this line of future work.

Acknowledgements This research is supported by the National Natural Science Foundation of China under Grant No. 62072007, 62192733, 61832009, 62192730.

Data Availability For code and data availability, we open source our replication package Github: <https://github.com/zkcpku/HiT-hierarchy-transformer>, including the datasets and the source code, to facilitate other researchers and practitioners to repeat our work and verify their studies.

Declarations

Competing Interests Beyond this, the authors have no competing interests to declare that are relevant to the content of this article.

References

- Ahmad WU, Chakraborty S, Ray B, Chang K (2020) A transformer-based approach for source code summarization. In: Proceedings of the annual meeting of the association for computational linguistics
- Allamanis M, Barr ET, Devanbu PT, Sutton C (2018) A survey of machine learning for big code and naturalness. *ACM Comput Surv* 51(4)
- Allamanis M, Barr ET, Ducousso S, Gao Z (2020) Typilus: neural type hints. In: Proceedings of the ACM SIGPLAN Conference on programming language design and implementation
- Allamanis M, Brockschmidt M, Khademi M (2018) Learning to represent programs with graphs. In: International conference on learning representations
- Allamanis M, Peng H, Sutton C (2016) A convolutional attention network for extreme summarization of source code. In: International conference on machine learning
- Alon U, Brody S, Levy O, Yahav E (2019) code2seq: Generating sequences from structured representations of code. In: International conference on learning representations
- Alon U, Sadaka R, Levy O, Yahav E (2020) Structural language models of code. In: Proceedings of the international conference on machine learning
- Alon U, Yahav E (2021) On the bottleneck of graph neural networks and its practical implications. In: The international conference on learning representations

- Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. *Proc ACM Program Lang* 3(POPL)
- Ben-Nun T, Jakobovits AS, Hoefler T (2018) Neural code comprehension: A learnable representation of code semantics. In: *Advances in neural information processing systems*
- Brunsfeld M, Hlynyski A, Qureshi A, Thomson P, Vera J, Turnbull P, dundargoc Clem T, ObserverOfTime Creager D, Helwer A, Rix R, Kavolis D, van Antwerpen H, Davis M, Ika Nguyen TA, Yahyaabadi A, Brunk S, Massicotte M, Hasabnis, N, bfredl Dong M, Moelius S, Kalt S, Lillis W, Kolja Pantelev V, Arnett J (2024) tree-sitter/tree-sitter: v0.22.6. <https://doi.org/10.5281/zenodo.11117307>
- Bui NDQ, Yu Y, Jiang L (2021) Treecaps: Tree-based capsule networks for source code processing. In: 35th AAAI conference on artificial intelligence, AAAI 2021
- Buratti L, Pujar S, Bornea MA, McCarley JS, Zheng Y, Rossiello G, Morari A, Laredo J, Thost V, Zhuang Y, Domeniconi G (2020) Exploring software naturalness through neural language models. *CoRR arXiv:2006.12641*
- Cai R, Liang Z, Xu B, Li Z, Hao Y, Chen Y (2020) TAG : Type auxiliary guiding for code comment generation. In: *Proceedings of the 58th annual meeting of the association for computational linguistics*
- Chirkova N, Troshin S (2021) Empirical study of transformers for source code. In: *ESEC/FSE '21: 29th ACM Joint European software engineering conference and symposium on the foundations of software engineering, 2021*
- Devlin J, Chang M, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, vol 1 (Long and Short Papers)*, pp 4171–4186. Association for Computational Linguistics. <https://doi.org/10.18653/V1/N19-1423>
- Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: A pre-trained model for programming and natural languages. In: *Findings of the association for computational linguistics: EMNLP 2020*
- Fernandes P, Allamanis M, Brockschmidt M (2019) Structured neural summarization. In: *International conference on learning representations*
- Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: Pre-training code representations with data flow. In: *9th international conference on learning representations, ICLR 2021*
- Hellendoorn VJ, Sutton C, Singh R, Maniatis P, Bieber D (2020) Global relational models of source code. In: *International conference on learning representations*
- Hindle A, Barr ET, Su Z, Gabel M, Devanbu PT (2012) On the naturalness of software. In: *34th international conference on software engineering, ICSE 2012*
- Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: *2018 IEEE/ACM 26th international conference on program comprehension (ICPC)*. IEEE
- Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*
- Iyer S, Konstantinos I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: *Proceedings of the 54th annual meeting of the association for computational linguistics, ACL 2016*
- Kim S, Zhao J, Tian Y, Chandra S (2021) Code prediction by feeding trees to transformers. In: *43rd IEEE/ACM international conference on software engineering, ICSE 2021*
- Kohavi R, Longbotham R (2017) Online controlled experiments and a/b testing. *Encycl Mach Learn Data Min* 7(8):922–929
- LeClair A, Jiang S, McMillan C (2019) A neural model for generating natural language summaries of program subroutines. In: *Proceedings of the 41st international conference on software engineering, ICSE 2019*
- Li J, Li Y, Li G, Jin Z, Hao Y, Hu X (2023) Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144*
- Li Z, Lu S, Guo D, Duan N, Jannu S, Jenks G, Majumder D, Green J, Svyatkovskiy A, Fu S, Sundaresan N (2022) Automating code review activities by large-scale pre-training. *ESEC/FSE 2022*
- Li Y, Tarlow D, Brockschmidt M, Zemel RS (2016) Gated graph sequence neural networks. In: *International conference on learning representations*
- Liu K, Kim D, Bissyandé TF, Kim T, Kim K, Koyuncu A, Kim S, Traon YL (2019) Learning to spot and refactor inconsistent method names. In: *ICSE 2019*
- Liu F, Li G, Fu Z, Lu S, Hao Y, Jin Z (2022) Learning to recommend method names with global context. *CoRR arXiv:2201.10705*
- Li J, Wang Y, Lyu MR, King I (2018) Code completion with neural attention and pointer networks. In: *Lang J (ed) Proceedings of the 27th international joint conference on artificial intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pp 4159–4165. <https://doi.org/10.24963/IJCAI.2018/578>


- Luan S, Yang D, Barnaby C, Sen K, Chandra S (2019) Aroma: code recommendation via structural code search (OOPSLA)
- Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement CB, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. In: *NeurIPS Datasets and Benchmarks 2021*
- Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: *Proceedings of the 30th AAAI conference on artificial intelligence*
- Musgrave K, Belongie SJ, Lim S (2020) A metric learning reality check. In: *Computer vision - ECCV 2020 - 16th European conference*
- Nguyen S, Phan H, Le T, Nguyen TN (2020) Suggesting natural method names to check name consistencies. In: *ICSE '20: 42nd international conference on software engineering*
- Niu C, Li C, Ng V, Ge J, Huang L, Luo B (2022) Spt-code: Sequence-to-sequence pre-training for learning source code representations. *CoRR arXiv:2201.01549*
- Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, Dolby J, Chen J, Choudhury M, Decker L, Thost V, Buratti L, Pujar S, Ramji S, Finkler U, Malaika S, Reiss F (2021) Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In: *Neural information processing systems datasets and benchmarks track*
- Schlichtkrull MS, Kipf TN, Bloem P, van den Berg R, Titov I, Welling M (2018) Modeling relational data with graph convolutional networks. In: *ESWC*
- Sennrich R, Haddow B, Birch A (2016) Neural machine translation of rare words with subword units. In: *Proceedings of the 54th annual meeting of the association for computational linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, vol 1: Long Papers. The Association for Computer Linguistics (2016). <https://doi.org/10.18653/v1/p16-1162>*
- Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L (2020) Treegen: A tree-based transformer architecture for code generation. In: *The 34th AAAI conference on artificial intelligence, AAAI 2020, The 32nd innovative applications of artificial intelligence conference, IAAI 2020, The 10th AAAI symposium on educational advances in artificial intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, pp 8984–8991. AAAI Press. <https://doi.org/10.1609/AAAI.V34I05.6430>*
- Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: *Advances in neural information processing systems*
- Vinyals O, Fortunato M, Jaitly N (2015) Pointer networks. In: *Advances in neural information processing systems*
- Wang W, Li G, Ma B, Xia X, Jin Z (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*
- Wang Y, Wang K, Gao F, Wang L (2020) Learning semantic program embeddings with graph interval neural network. *Proc ACM Program Lang 4(OOPSLA)*
- Wang S, Wen M, Lin B, Mao X (2024) Lightweight global and local contexts guided method name recommendation with prior knowledge. In: *ESEC/FSE '21: 29th ACM joint European software engineering conference and symposium on the foundations of software engineering*
- Wang W, Zhang K, Li G, Liu S, Li A, Jin Z, Liu Y (2022) Learning program representations with a tree-structured transformer
- Yang M, Gu B, Duan Z, Jin Z, Zhan N, Dong Y (2022) Intelligent program synthesis framework and key scientific problems for embedded software. *Chin Space Sci Technol 42(4):1*
- Ye F, Zhou S, Venkat A, Marcus R, Tatbul N, Tithi JJ, Petersen P, Mattson TG, Kraska T, Dubey P, Sarkar V, Gottschlich J (2020) MISIM: an end-to-end neural code similarity system. *CoRR arXiv:2006.05265*
- Yin P, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL (2019) Learning to represent edits. In: *International conference on learning representations*
- Yu H, Lam W, Chen L, Li G, Xie T, Wang Q (2019) Neural detection of semantic code clones via tree-based convolution. In: *Proceedings of the 27th international conference on program comprehension, ICPC 2019*
- Zhang K, Li Z, Jin Z, Li G (2023) Implant global and local hierarchy information to sequence based code representation models. In: *31st IEEE/ACM international conference on program comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023, pp 157–168. IEEE. <https://doi.org/10.1109/ICPC58990.2023.00030>*
- Zhang K, Wang W, Zhang H, Li G, Jin Z (2022) Learning to represent programs with heterogeneous graphs. In: *Proceedings of the 30th IEEE/ACM international conference on program comprehension, ICPC '22*
- Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: *Proceedings of the 41st international conference on software engineering, ICSE 2019*

Zügner D, Kirschstein T, Catasta M, Leskovec J, Günnemann S (2021) Language-agnostic representation learning of source code from structure and context. In: 9th international conference on learning representations, ICLR 2021

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Kechi Zhang¹  · Jia Li¹ · Zhuo Li¹ · Zhi Jin¹ · Ge Li¹

✉ Kechi Zhang
zhangkechi@pku.edu.cn

Jia Li
lijiaa@pku.edu.cn

Zhuo Li
lizhmq@pku.edu.cn

Zhi Jin
zhijin@pku.edu.cn

Ge Li
lige@pku.edu.cn

¹ Key Lab of High Confidence Software Technology, MoE, Peking University, Beijing, China