GNN-Coder: Boosting Semantic Code Retrieval with Combined GNN and Transformer

Anonymous ACL submission

Abstract

Code retrieval is a crucial component in modern software development, particularly in large-scale projects. However, existing approaches relying on sequence-based models often fail to fully exploit the structural dependencies inherent in code, leading to suboptimal retrieval performance, particularly with structurally complex code fragments. In this paper, we introduce GNN-Coder, a novel framework based on Graph Neural Network (GNN) to utilize Abstract Syntax Tree (AST). We make the first attempt to study how GNN-integrated Transformer can promote the development of semantic retrieval tasks by capturing the structural and semantic features of code. We further propose an innovative graph pooling method tailored for AST, utilizing the number of child nodes as a key feature to highlight the intrinsic topological relationships within the AST. This design effectively integrates both sequential and hierarchical representations, enhancing the model's ability to capture code structure and semantics. Additionally, we introduce the Mean Angular Margin (MAM), a novel metric for quantifying the uniformity of code embedding distributions, providing a standardized measure of feature separability. The proposed method achieves a lower MAM, indicating a more discriminative feature representation. This underscores GNN-Coder's superior ability to distinguish between code snippets, thereby enhancing retrieval accuracy. Experimental results show that GNN-Coder significantly boosts retrieval performance, with a 1%-10% improvement in MRR on the CSN dataset, and a notable 20% gain in zero-shot performance on the CosQA dataset.

1 Introduction

Code retrieval, as a technology that takes natural language queries as input and outputs code snippets that match the query intent, plays a facilitating role in program reuse during the software development process (Li et al., 2022; Liu et al., 2024). Meanwhile, it also drives the latest research in the field of retrieval-augmented generation (Zhou et al., 2022; Wang et al., 2024). The main challenge faced by effective code retrieval lies in the semantic gap between natural language descriptions and source code. This is because natural language descriptions and source code are heterogeneous resources that share very few lexical tokens, synonyms, and language structures (Gu et al., 2018; Zhu et al., 2022).

With the Transformer achieving success in the field of NLP(Achiam et al., 2023; Grattafiori et al., 2024), CodeBERT treats code as a special form of natural language. Specifically, it processes code snippets and their corresponding texts as pairs of natural language sentences in the BERT architecture (Devlin, 2018). However, code has unique syntax and structure different from natural languages, such as AST (Zhang et al., 2019; Tang et al., 2021), which reflects the hierarchical organization of code. To improve the performance of the model, researchers have made efforts (Guo et al., 2020; Wang et al., 2021b, 2023) to integrate AST into the transformer training process. For example, UniXcoder (Guo et al., 2022) introduces a one-to-one mapping function to convert the AST into a sequence. However, these methods directly flatten the components of AST to be a linear sequence, ignoring its inherent structural potential and impairing performance in tasks that require deep syntactic understanding.

In this paper, we propose a novel GNN-based framework, investigating the first attempt to explore how GNN-integrated Transformer make full use of the complete syntactic information of the AST for the code retrieval task. The inherent sparsity of GNNs makes them particularly suitable for handling tree-structured AST. Specifically, we begin by encoding the content of AST nodes using a frozen Transformer model, while representing their types through one-hot encoding. Next, we employ a GNN model to integrate information across all

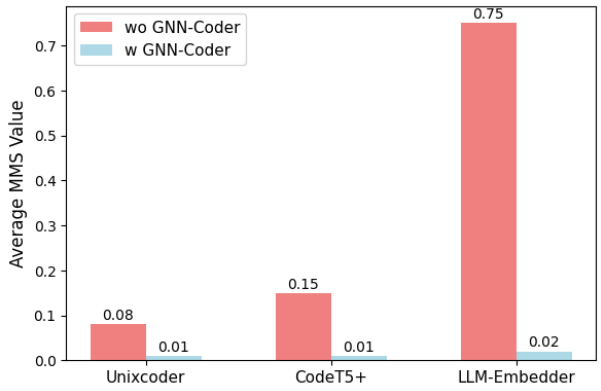

Figure 1: The average MAM for six PLs in CSN dataset. A value close to 0 indicates thorough feature separation.

AST nodes and generate a unified code embedding. To align the code and text embeddings, we apply contrastive loss during the training of the GNN model. Considering the tree-structured nature of the AST and the encoding requirements, we propose a hierarchical GNN model incorporating a graph pooling layer. Additionally, we introduce a novel graph pooling method, ASTGPool, designed specifically for AST, which emphasizes intrinsic topological relationships within the AST and accelerates information propagation to the root node, thereby improving retrieval accuracy.

To further evaluate the discriminative power of learned code features, we introduce the Mean Angular Margin (MAM) metric. MAM calculates the cosine similarity between a given text embedding and all code embeddings, offering a robust measure of feature differentiation—an aspect often neglected in prior research, yet essential for effective code retrieval. The ideal scenario, as indicated by MAM, occurs when the cosine similarities between distinct code embeddings approach zero, reflecting thorough separation of code embeddings. As shown in Figure 1, GNN-Coder achieves a lower MAM value, demonstrating improved feature separation. Experimental results show that GNN-Coder significantly boosts retrieval performance, with a 1%-10% improvement in MRR on the CSN dataset, and a notable 20% gain in zero-shot performance on the CosQA dataset.

In summary, our contributions are as follows:

- We propose a novel framework based on GNN for code retrieval, leveraging the structured sparse relationship captured by AST. To the best of our knowledge, this is the first attempt to explore the AST-guided GNN in Transformer module for code retrieval task.
- We introduce a novel graph pooling method, ASTGPool, tailed for AST. The design evaluates the importance score by utilizing the number of child nodes, which characterize the topological relationship. We show that the proposed pooling method is superior to the existing pooling techniques.
- Experiments validate the effectiveness of the proposed framework. The introduced GNN module enhances the performance of the model which has leveraged part of the AST as a linear sequence. Additionally, we present a new metric, MAM, to assess code representation distribution, demonstrating that the learned features are effectively separated, benefiting code retrieval.

2 Related Work

Code Retrieval. Software developers often rely on the reuse of existing code resources to achieve efficient code. Therefore, code retrieval has emerged as a crucial research area, aiming to explore the implicit connections between natural language queries and code databases, enabling developers to obtain the required code quickly and accurately. Early research mainly represent code and queries as feature vectors and retrieve code based on similarity to the query vectors. Boolean vectors (Salton et al., 1983) characterize the features by indicating the presence or absence of specific features, such as specific types of AST nodes (Luan et al., 2019). Another commonly used method is to map a set of tokens into a Term Frequency-Inverse Document Frequency (TF-IDF) vector, which can not only indicate whether a feature exists but also reflect the importance of this feature (Diamantopoulos et al., 2018; Takuya and Masuhara, 2011).

With the rapid development of deep learning technology, an increasing number of studies have focused on the use of neural networks to achieve efficient code retrieval. Most of the related work adopts end-to-end neural learning methods. The query and the code are embedded into a joint vector space through the model, and the code search problem is then transformed into finding the nearest-neighbor code for a given query in this space (Gu et al., 2018; Sun et al., 2022). The core of code retrieval lies in code encoding, which is reviewed below.

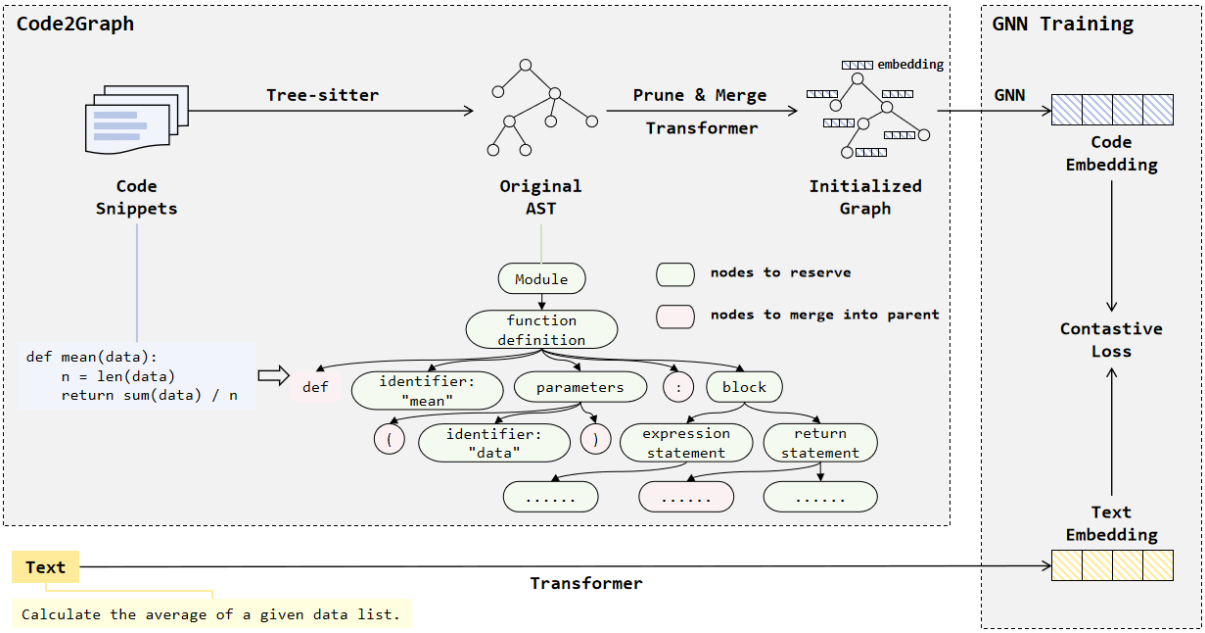

Figure 2: Overall architecture of GNN-Coder. The code is transformed into an AST, which is initialized with a Transformer model, processed by a GNN, and aligned with text embeddings through a contrastive loss function.

Code Encoding With Transformer. Following the significant success of the transformer architecture and pre-training in NLP, researchers have started to explore the potential of the Transformer model in code representation learning. Single encoder models mainly follow the BERT framework. CodeBERT (Feng et al., 2020) utilizes the masked language objective to pre-train on NL-PL pairs, and adds the substitute token detection task (Clark, 2020). GraphCodeBERT (Guo et al., 2020) enhances the pre-training process by incorporating the data flow derived from the AST. SynCoBERT (Wang et al., 2021a) utilizes identifier prediction, AST edge prediction, and multimodal contrastive learning to make full use of AST. In addition, some works adopt the encoder-decoder architecture. CodeT5 (Wang et al., 2021b) extends the T5 (Raffel et al., 2020) model to the code domain and additionally combines the information of the identifier nodes in the AST. TreeBERT (Jiang et al., 2021) utilizes the structural information of the AST by representing it as a set of paths from the root node to the terminal nodes.

The aforementioned methods utilize Transformer models for code encoding. While some integrate AST information, they still treat it as a linear sequence. This sequence-based representation hampers the accurate capture of hierarchical and complex relationships inherent in code structures. Consequently, these approaches fail to fully leverage the syntactic and semantic richness of programming languages, which are defined by intricate nested structures that cannot be effectively represented in a flat sequential format.

Code Encoding With GNN. Allamanis et al. (2017) first use GNN to represent code. To convert the code into a graph, they design complex edges based on the AST. Devign (Zhou et al., 2019) and Reveal (Chakraborty et al., 2021) adopt the Code Property Graph (CPG), which is composed of the AST, the control flow graph (CFG) and the data flow graph (DFG), and used the Gated Graph Neural Network (GGNN) to encode the graph. Graphsearchnet (Liu et al., 2023) introduces Bidirectional GGNN (BiGGNN) to create graphs for code and text, capture local structural details, and uses a multi-head attention module to enhance BiGGNN.

These methods mainly use GNN as a tool for encoding code for vulnerability identification. In our research work, we focus on the multimodal retrieval task and integrate Transformer with GNN. To ensure that our model is applicable to various PLs, we use only the basic AST without adding complex edges or nodes.

3 Methodology

3.1 Overall Architecture

The architecture of GNN-Coder is depicted in Figure 2. In the Code2Graph stage, we utilize tools to convert the code into an AST, which is then ini-

tialized using a Transformer model. The initialized graph is processed by a GNN model. Finally, a contrastive loss is applied to align the embedding representations of both code and text.

3.2 Code2Graph

Code2AST. We adopt the same method as in the code Transformer series of works and use Tree-sitter[1] to convert various PLs into ASTs.

Refine AST. The ASTs generated by Tree-sitter are usually more redundant. Therefore, we need to further simplify the AST. Specifically, we delete the "shadow nodes" with the same type and content. However, directly deleting these nodes may lead to the loss of code syntax information. To solve this problem, we merge these nodes into their parent nodes. The specific operation is as follows: we reconstruct the content of the parent node by concatenating the contents of all child nodes, except for the "block" node. The "block" node, as a container, has its content sourced from the contents of all child nodes. For example, the "function_definition" node in Figure 2 needs to be reconstructed because it has child nodes to be deleted (i.e., "def" and ":"). First, we replace the content of the "function_definition" node with the concatenation of the contents of all child nodes except the "block" node. Then, we delete the "def" and ":" nodes. Eventually, the content of the "function_definition" node becomes "def mean (data):", which includes the information of the deleted nodes.

AST2Graph. To meet the input requirements of GNN, we need to represent the types and contents of AST nodes as vectors. Previous methods (Devlin, 2018; Chakraborty et al., 2021; Li et al., 2021) used one-hot encoding and a pre-trained word2vec model to encode the types and contents of AST nodes respectively, and then concatenated these encodings to form the initial node embeddings. In this research work, we replace the word2vec model with a pre-trained Transformer model to fully utilize the powerful ability of the Transformer model in encoding context information. In addition, referring to methods such as Devign, we reverse the direction of the edges in the original AST to ensure that the information of the leaf nodes can be effectively propagated to the root node.

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

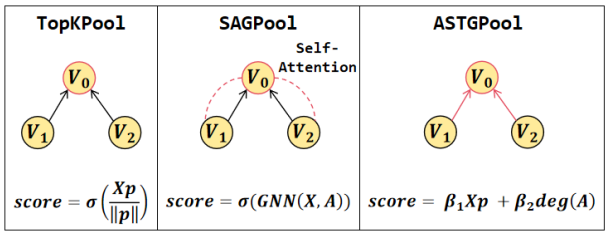

Figure 3: Illustrating importance score calculation for different pooling methods. "deg" represents in-degree.

3.3 GNN Training

3.3.1 GNN Architecture

Since the root node in the tree structure contains much richer information, conventional GNN models struggle to handle it effectively. In view of this, we meticulously design a hierarchical GNN model that incorporates a novel graph pooling layer, adapting to the AST and its coding requirements.

Conv Layer. The GNN model we construct employs the FAConv layer (Bo et al., 2021) as the convolutional layer. The FAConv layer can adaptively adjust the coefficients of low-frequency and high-frequency signals without prior knowledge of the network type. This remarkable flexibility makes it suitable for processing graphs initialized in different ways.

Graph Pooling. We propose a hierarchical architecture that incorporates a graph pooling layer into the original FAGCN model. This design enables the capture of information at multiple granularity levels. The graph pooling layer plays a key role by accelerating information aggregation from leaf nodes to the root node and effectively filtering noise in the leaf nodes.

To explore the most suitable graph pooling method for AST, we experimentally study TopKPool (Gao and Ji, 2019; Cangea et al., 2018) and SAGPool (Lee et al., 2019; Knyazev et al., 2019). However, our findings indicate that these methods are not fully appropriate for AST. Specifically, while SAGPool computes importance scores based on the features of neighboring nodes, its performance is sometimes inferior to TopKPool, which relies solely on node features. This discrepancy arises because the inversion of edges repositions child nodes as neighbors, introducing noise into the parent node's importance calculation. Based on the above situation, we propose a novel graph pooling method specifically designed for AST, called ASTGPool. It evaluates the importance of nodes based on node features and the number of adja-

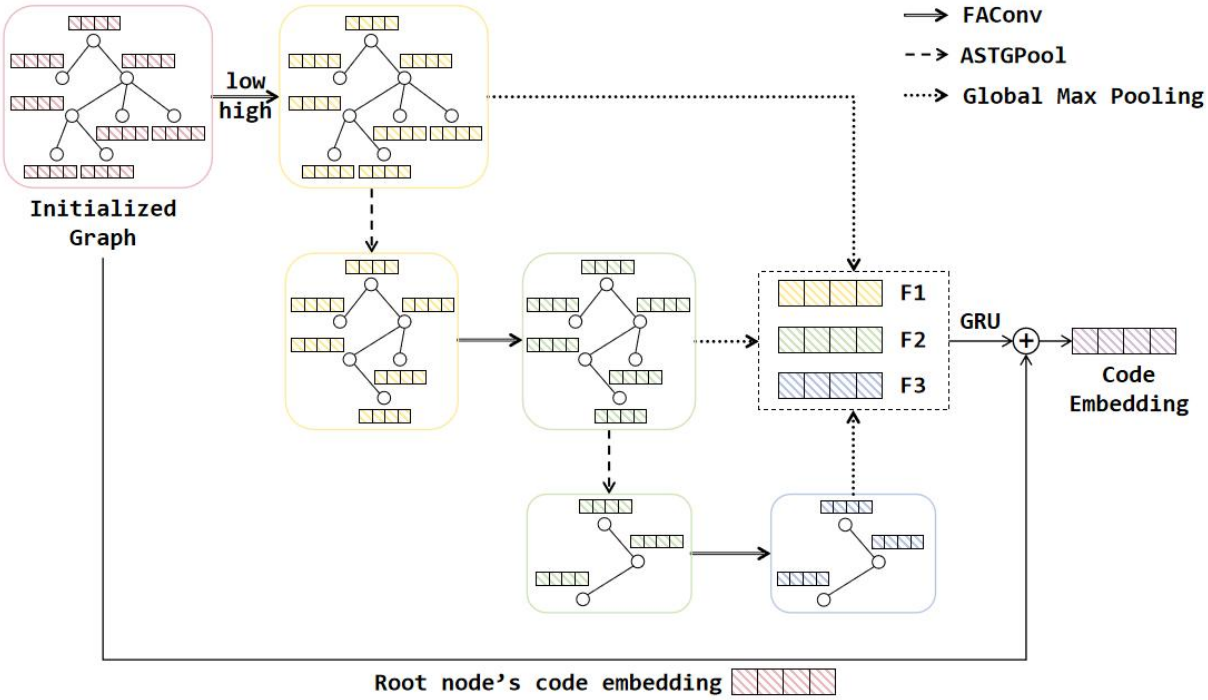

Figure 4: Illustrating the GNN model, which is a hierarchical architecture incorporated with ASTGPool layer. Here we show a hierarchical depth of 3 and $F1, F2, F3$ represent the features extracted at each corresponding depth.

cent nodes, which emphasizes intrinsic topological relationships. The formula for calculating the importance score is as follows:

$$score = \beta_1 Xp + \beta_2 deg(A), \tag{1}$$

where X denotes the node feature matrix, A is the adjacency matrix, and β_1, β_2 are learnable parameters that balance the contribution of each factor. Figure 3 illustrates the importance score calculation for different pooling methods.

Overall Architecture. The overall architecture consists of stacking the FAConv and ASTGPool layers L times, with a global maximum pooling layer added after each FAConv layer to capture embeddings at different granularities. To effectively fuse these multi-level embeddings, we concatenate them and then generate the final code embeddings through a Gated Recurrent Unit (GRU) (Cho et al., 2014). The root node contains the source code in text format. To preserve the Transformer model's context encoding capabilities and expedite training, we incorporate a residual connection between the root node's code embeddings and the final code embeddings, following the design principles of the CLIP-Adapter (Gao et al., 2024). The GNN model architecture is detailed in Figure 4.

3.3.2 Training objective

We draw on the contrastive representation learning method employed in CLIP (Radford et al., 2021) to align the code embeddings generated by the GNN model with the text embeddings generated by the Transformer model. This method is concise and efficient, enabling the learning of aligned embeddings across different modalities. The formula for the loss function used to align the code embeddings and the text embeddings is as follows:

$$\mathcal{L}_{\text{Code}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp\left(\text{sim}(\mathbf{c}_i, \mathbf{t}_i)/\tau\right)}{\sum_{j=1}^{N}\exp\left(\text{sim}(\mathbf{c}_i, \mathbf{t}_j)/\tau\right)},$$

$$\mathcal{L}_{\text{Text}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp\left(\text{sim}(\mathbf{c}_i, \mathbf{t}_i)/\tau\right)}{\sum_{j=1}^{N}\exp\left(\text{sim}(\mathbf{c}_j, \mathbf{t}_i)/\tau\right)},$$

$$\mathcal{L} = \frac{1}{2}\left(\mathcal{L}_{\text{Code}} + \mathcal{L}_{\text{Text}}\right). \tag{2}$$

Here, $\mathbf{c}_i$ and $\mathbf{t}_i$ represent the i-th pair of code embedding and text embedding, respectively. The function $\text{sim}(\cdot, \cdot)$ represents the cosine similarity between two vectors. The parameter τ is a learnable temperature parameter.

4 Experiments

4.1 Settings

Datasets. In code retrieval task, we use the CSN and CosQA datasets to evaluate the performance of

Model	Ruby		JavaScript		Go		Python		Java		PHP		MRR_Avg
	MRR	R@1	MRR	R@1	MRR	R@1	MRR	R@1	MRR	R@1	MRR	R@1	
UniXcoder 110M	45.02	34.26	34.28	25.34	54.64	43.61	33.31	24.42	36.28	26.71	24.94	17.51	38.08
+ GNN-Coder	**50.40**	**40.05**	**40.22**	**30.42**	**67.76**	**58.04**	**44.56**	**34.37**	**46.42**	**35.60**	**35.86**	**26.46**	**47.54**
CodeT5+ 110M	73.55	64.55	65.83	56.24	89.51	84.55	69.75	60.12	69.42	59.74	64.44	54.06	72.08
+ GNN-Coder	**73.85**	**64.71**	**67.20**	**57.70**	**90.71**	**86.17**	**70.37**	**60.77**	**70.80**	**61.46**	**65.93**	**55.47**	**73.14**
LLM-Embedder	63.07	53.29	49.33	39.65	80.94	73.02	55.86	45.52	53.60	42.94	44.70	34.56	57.92
+ GNN-Coder	**65.00**	**55.19**	**52.35**	**42.27**	**87.10**	**81.30**	**61.71**	**51.47**	**60.38**	**49.72**	**54.96**	**44.19**	**63.58**

Table 1: Results comparison in terms of MRR and R@1 on various Transformer architectures on the CSN dataset, showing GNN-Coder consistently enhances the performance of all Transformer-based models.

GNN-Coder. The CSN is derived from the CodeSearchNet dataset (Husain et al., 2019), and it filters out low-quality queries through manually crafted rules (Guo et al., 2020). CosQA (Huang et al., 2021), on the other hand, uses queries from the logs of the Microsoft Bing search engine and the corresponding code snippets. The experimental results of the CSN dataset can demonstrate the basic performance of GNN-Coder in code retrieval, while the results of the CosQA dataset can verify its generalization ability, as the queries in the CosQA dataset are not included in the training set.

Baselines. Our GNN architecture is applicable to various Transformer models. It is crucial to evaluate its compatibility and adaptability with various state-of-the-art Transformer models. Hence, to comprehensively evaluate the performance of GNN-Coder, we select a variety of Transformer models, including UniXcoder, CodeT5+, and LLM-Embedder. UniXcoder and CodeT5+ represent state-of-the-art advancements in code retrieval, while LLM-Embedder is tailored to address the unique retrieval enhancement needs of LLMs.

Evaluation Metrics. We adopt Mean Reciprocal Rank (MRR) and Recall@K (Liu et al., 2021; Di Grazia and Pradel, 2023) as the retrieval metrics. In addition, we utilize the proposed MAM to evaluate the distribution of code embeddings. MAM are calculated through the average cosine similarity between text embeddings and all code embeddings. The specific formula is as follows:

$$\text{MAM}_j = \frac{1}{N} \sum_{i=1}^{N} \text{sim}(\mathbf{c}_i, \mathbf{t}_j). \tag{3}$$

By analyzing the distribution of all MAM, we can evaluate the uniformity of the code embedding distributions. Theoretically, if the code embeddings are uniformly distributed, both their mean and standard deviation (SD) should be close to zero.

4.2 Implementation Details

For UniXcoder and CodeT5+, we use the configurations provided by the authors to encode code and text. For LLM-Embedder, without additional instructions, we use the embedding of the first token in the last hidden layer as the encoded embeddings.

Regarding the configuration of the GNN model, we set it to three layers. The size of the hidden layer of the GNN model is dynamically adjusted according to the output dimension of the Transformer model and the number of AST node types. Meanwhile, the output dimension of the GNN model is kept consistent with that of the corresponding Transformer model. More details related to the training process can be found in Appendix A.

4.3 Results

Results on CSN. Considering that UniXcoder is pre-trained on the CSN dataset, whereas CodeT5+ incorporates a larger volume of code data, we first assess the effectiveness of GNN-Coder in improving Transformer model performance on the observed datasets.

The experimental results, presented in Table 1, provide compelling evidence of the effectiveness of the GNN-Coder in the text-to-code retrieval task. Specifically, GNN-Coder consistently enhances the performance of all Transformer-based models, with the most significant improvements observed in models that exhibit lower initial capabilities in code generation. Notably, despite UniXcoder's partial incorporation of AST information, GNN-Coder still outperforms it. This result can be attributed to the superior ability of GNN-Coder to exploit structured AST information, thereby yielding better performance in the retrieval task.

Furthermore, we report the MAM score in Table 2, which provides insight into the uniformity of code embedding distributions. All Transformer models exhibit varying degrees of non-uniformity across datasets. Notably, UniXcoder demonstrates

Model	Ruby		JavaScript		Go		Python		Java		PHP	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
UniXcoder 110M	0.09	0.02	0.09	0.02	0.07	0.02	0.08	0.02	0.08	0.02	0.08	0.02
+ GNN-Coder	**0.01**	**0.01**	**0.01**	0.02	**0.01**	**0.01**	**0.00**	**0.01**	**0.01**	**0.01**	**0.00**	0.02
CodeT5+ 110M	0.16	0.05	0.17	0.05	0.15	0.03	0.13	0.03	0.14	0.04	0.15	0.04
+ GNN-Coder	**0.01**	**0.01**	**0.02**	**0.01**	**0.00**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**
LLM-Embedder	0.76	0.01	0.77	0.01	0.69	0.02	0.76	0.01	0.76	0.01	0.74	0.01
+ GNN-Coder	**0.02**	0.01	**0.01**	0.01	**0.02**	**0.00**	**0.02**	**0.00**	**0.02**	**0.00**	**0.02**	**0.00**

Table 2: MAM score comparison for different methods on the CSN dataset, providing insight into the uniformity of code embedding distribution. Lower value indicates better distribution.

Model	MRR	R@1	Mean	SD
UniXcoder 110M	27.96	17.8	0.10	0.02
+ GNN-Coder	**48.22**	**36.2**	**0.01**	0.02
CodeT5+ 110M	45.44	31.2	0.24	0.03
+ GNN-Coder	**66.50**	**53.8**	**0.05**	**0.01**
LLM-Embedder	46.10	33.4	0.77	0.01
+ GNN-Coder	**67.96**	**56.4**	**0.02**	**0.00**

Table 3: Results comparison in terms of MRR, R@1 and MAM on various Transformer architectures on the CosQA dataset.

the least non-uniformity, while CodeT5+ shows a higher degree. This discrepancy stems from the fact that CodeT5+ is trained on a broader and more diverse set of code data. while improving performance, it introduces greater complexity and irregularities in embedding distributions. In contrast, the general model LLM-Embedder, with limited capacity to process code-specific features, exhibits a significantly higher level of non-uniformity, highlighting the challenges faced by non-specialized models in handling code data effectively. The introduction of the GNN model significantly improves embedding uniformity. Despite initial non-uniformity in the output distribution of the Transformer model, the GNN model effectively minimizes these biases, with the average MAM scores of all embeddings approaching zero. This improvement is attributed to the learnable parameter λ within the GNN framework, which dynamically adjusts the balance between the Transformer and GNN models. Additionally, the reduction in the variance of most MAM scores indicates more symmetric and reliable embeddings, demonstrating that the GNN model not only enhances uniformity but also improves the stability and consistency of the results.

Zero-shot Performance on CosQA. To evaluate the generalization ability of GNN-Coder, we use CosQA dataset, which is not part of the training set for the code Transformer model, and evaluate the performance improvement of GNN-Coder. The results, presented in Table 3, show that the general model, LLM-Embedder, outperforms the code model. This suggests that the code model has limited generalization capability, likely due to the significant differences between the CosQA dataset and the training data of the code Transformer model. Notably, after incorporating the GNN model, all models exhibit substantial performance improvements. Furthermore, the reported MAM score indicates that these improvements may be attributed to a more uniform distribution of code.

4.4 Ablation Study

The Effect of Pooling Layer. We first perform an ablation study on the pooling layers in the GNN model, comparing the performance of TopKPool, SAGPool, and ASTGPool. The results are summarized in Table 4. As described in Section 3.3.1, existing pooling methods struggle with reverse AST processing. In contrast, ASTGPool addresses this issue by accounting for both the number of adjacent nodes and their characteristics, thus overcoming the limitations of previous methods. The experimental results demonstrate that ASTGPool outperforms all other pooling methods.

Furthermore, we conduct experiments on GNN-Coder without a pooling layer, as shown in Table 5. It indicates that the impact of the pooling layer varies across models. Specifically, the pooling layer has a more pronounced effect on improving the performance of LLM-Embedder compared to CodeT5+. This difference arises from the models' distinct code processing capabilities. For LLM-Embedder, which has relatively limited code processing ability, most AST nodes contain noise, and pooling improves model performance. Conversely, for CodeT5+, which has a stronger code processing ability, most AST nodes carry valuable information,

Model	Pooling Layer	Ruby	JavaScript	Go	Python	Java	PHP	Avg
UniXcoder 110M	TopKPool	49.14	39.37	67.24	42.80	46.10	35.23	46.65
	SAGPool	50.08	39.49	66.29	**44.90**	45.83	35.16	46.96
	ASTGPool	**50.40**	**40.22**	**67.76**	44.56	**46.42**	**35.86**	**47.54**
CodeT5+ 110M	TopKPool	73.72	66.93	90.25	**70.61**	70.70	65.73	73.02
	SAGPool	73.66	66.84	90.25	70.44	70.73	**66.00**	72.99
	ASTGPool	**73.85**	**67.20**	**90.71**	70.37	**70.80**	65.93	**73.14**
LLM-Embedder	TopKPool	61.62	49.31	84.65	56.84	54.33	48.95	59.28
	SAGPool	61.35	49.40	84.63	56.70	54.86	48.75	59.28
	ASTGPool	**65.00**	**52.35**	**87.10**	**61.71**	**60.38**	**54.96**	**63.58**

Table 4: Illustrating the effect of the proposed ASTGPool by comparing different pooling layers in terms of MRR with respect to various Transformer architectures on the CSN dataset.

	Ruby		JavaScript		Go		Python		Java		PHP		Avg
	MRR	Mean	MRR	Mean	MRR	Mean	MRR	Mean	MRR	Mean	MRR	Mean	MRR
CodeT5+ 110M	73.55	0.16	65.83	0.17	89.51	0.15	69.75	0.13	69.42	0.14	64.44	0.15	72.08
+ MLP Adapter	73.75	0.07	67.18	0.08	90.35	0.05	70.22	0.06	70.75	0.06	65.76	0.06	73.00
+ GNN wo pooling	73.65	0.02	67.19	0.02	90.37	0.00	**70.50**	0.01	**70.96**	0.01	65.84	0.01	73.09
+ GNN w pooling (best)	**73.85**	0.01	**67.20**	0.02	**90.71**	0.00	70.37	0.01	70.80	0.01	**65.93**	0.01	**73.14**
- no AST node type	73.64	0.02	66.47	0.03	89.69	0.00	69.78	0.02	70.51	0.02	65.88	0.01	72.66
- undirect AST	73.32	0.02	67.07	0.02	90.34	0.00	70.40	0.01	70.86	0.01	66.10	0.01	73.02
LLM-Embedder	63.07	0.76	49.33	0.77	80.94	0.69	55.86	0.76	53.60	0.76	44.70	0.74	57.92
+ MLP Adapter	61.53	0.19	52.05	0.19	85.87	0.27	60.41	0.26	58.16	0.22	51.72	0.18	61.62
+ GNN wo pooling	61.53	-0.01	49.77	0.01	84.77	0.02	61.69	0.02	58.90	0.02	54.21	0.02	61.81
+ GNN w pooling (best)	**65.00**	0.02	**52.35**	0.01	**87.10**	0.02	**61.71**	0.02	**60.38**	0.02	**54.96**	0.02	**63.58**
- no AST node type	64.45	0.02	52.13	0.01	86.51	0.02	59.12	0.01	58.57	0.02	54.09	0.02	62.48
- undirect AST	64.78	0.01	52.25	0.01	86.88	0.02	61.51	0.02	55.00	0.01	54.80	0.03	62.54

Table 5: Retrieval results under different settings on the CSN dataset. For experiments removing parts of AST information, we set the pooling ratio to 0.1, which is generally best.

and pooling may result in information loss.

The Effect of GNN. We conduct an ablation study on GNN-Coder with various configurations. First, we replace the GNN model with a basic Multi-Layer Perceptron (MLP) Adapter, which remaps Transformer embeddings without AST information. Next, we examine the role of AST information by evaluating two variations: excluding AST node types and converting directed edges to undirected edges. The experimental results in terms of MRR and Mean MAM are summarized in Table 5.

The study reveals that the MLP Adapter mitigates the non-uniformity of the Transformer model to some extent, leading to performance improvements across all models. However, the enhancement is modest compared to the GNN model, underscoring the critical role of integrating AST information. Additionally, removing the AST information significantly degrades model performance, even below the baseline performance with the MLP Adapter in some cases. This suggests that performance gains are primarily driven by the inclusion of AST data, rather than an increase in model parameters. Overall, the proposed GNN-Coder achieves the best performance by incorporating AST semantic information with a GNN model.

5 Conclusion

we propose GNN-Coder, a novel framework for code retrieval that leverages GNNs and ASTs to overcome the limitations of traditional sequence-based models. By integrating GNNs with Transformers, GNN-Coder effectively captures both structural and semantic features of code, addressing the challenges posed by structurally complex code fragments. The introduction of a tailored graph pooling method further enhances the model's ability to retrieval accuracy through better feature separation. Our experiments demonstrate that GNN-Coder outperforms existing methods, achieving significant improvements in retrieval performance across multiple datasets. The results highlight the potential of GNN-Coder to advance the field of code retrieval, offering a promising solution for handling complex code structures and enhancing the effectiveness of software development tools.

6 Limitation

The GNN-Coder framework presented in this work is primarily designed for the code retrieval task, demonstrating its potential within this domain. However, the framework's applicability could extend to other related tasks, such as code clone detection and code translation. Future research should explore a broader range of tasks to better assess the framework's effectiveness across various code-related applications.

A limitation in the current approach lies in the handling of text embeddings. As detailed in Appendix C, the uniform distribution of code embeddings does not entirely eliminate slight non-uniformity in text embeddings, caused by the dynamic nature of the distribution metrics. Given that GNN-Coder is primarily tailored for code modality, further investigation into the text embedding non-uniformity is necessary. Future work will focus on addressing this issue and evaluating the framework's generalization capabilities, particularly in combination with LLMs.

Additionally, the experiments conducted so far have been limited to models with a relatively small parameter scale due to hardware constraints. To fully understand the scalability of the GNN-Coder framework, future work should explore its performance and applicability on larger models and more diverse types of model architectures.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

Deyu Bo, Xiao Wang, Chuan Shi, and Huawei Shen. 2021. Beyond low-frequency information in graph convolutional networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 3950–3957.

Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. 2018. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287*.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

K Clark. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Luca Di Grazia and Michael Pradel. 2023. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31.

Themistoklis Diamantopoulos, Georgios Karagiannopoulos, and Andreas L Symeonidis. 2018. Codecatch: Extracting source code snippets from online sources. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 21–27.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Hongyang Gao and Shuiwang Ji. 2019. Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR.

Peng Gao, Shijie Geng, Renrui Zhang, Teli Ma, Rongyao Fang, Yongfeng Zhang, Hongsheng Li, and Yu Qiao. 2024. Clip-adapter: Better vision-language models with feature adapters. *International Journal of Computer Vision*, 132(2):581–595.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR.

Boris Knyazev, Graham W Taylor, and Mohamed Amer. 2019. Understanding attention and generalization in graph neural networks. *Advances in neural information processing systems*, 32.

Junhyun Lee, Inyeop Lee, and Jaewoo Kang. 2019. Self-attention graph pooling. In *International conference on machine learning*, pages 3734–3743. pmlr.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2898–2910.

Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)*, 54(9):1–40.

Chao Liu, Xindong Zhang, Hongyu Zhang, Zhiyuan Wan, Zhan Huang, and Meng Yan. 2024. An empirical study of code search in intelligent coding assistant: Perceptions, expectations, and directions. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 283–293.

Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Transactions on Software Engineering*, 49(4):2839–2855.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Gerard Salton, Edward A Fox, and Harry Wu. 1983. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036.

Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code search based on context-aware code translation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 388–400.

Watanabe Takuya and Hidehiko Masuhara. 2011. A spontaneous code recommendation tool based on associative search. In *Proceedings of the 3rd International Workshop on search-driven development: Users, infrastructure, tools, and evaluation*, pages 17–20.

Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. 2021. Ast-transformer: Encoding abstract syntax trees efficiently for code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1193–1195. IEEE.

Xiaohua Wang, Zhenghua Wang, Xuan Gao, Feiran Zhang, Yixin Wu, Zhibo Xu, Tianyuan Shi, Zhengyuan Wang, Shizheng Li, Qi Qian, et al. 2024. Searching for best practices in retrieval-augmented generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17716–17736.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel

neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

Qingfu Zhu, Xianzhen Luo, Fang Liu, Cuiyun Gao, and Wanxiang Che. 2022. A survey on natural language processing for programming. *arXiv preprint arXiv:2212.05773*.

A Training Process

During the training process, we choose the AdamW optimizer and set the learning rate to 0.004. We warm up the learning rate in the first 10% of the training steps and then adopt a cosine annealing decay strategy. Specifically, for Transformer models with an output dimension of 256, we train the GNN model for 400 iterations with a batch size of 16K. For models with an output dimension of 768, to ensure the training effect, we adjust the batch size to 8K and increase the number of training iterations to 1000.

Notably, for UniXcoder and CodeT5+, different from the original settings, we preserve the original format of the code before tokenization instead of splitting each word. This is because preserving the original code format is more reasonable in real-world application scenarios. Therefore, the experimental results of UniXcoder and CodeT5+ may slightly differ from the original reports.

B Pooling Ratio

We also investigate the performance of different pooling ratios, varying the ratio from 0.1 to 0.9. The results on CSN and CosQA datasets are presented in Table 6. For CodeT5+, which exhibits the best inherent code capability, the performance is relatively insensitive to the pooling ratio, with optimal results observed at ratios of 0.1 and 0.7. This insensitivity may be due to the well-initialized AST, where most nodes are significant. As a result, while reducing computational complexity, the pooling layer may slightly impact performance due to information loss, as detailed in Table 5. For the other two models, which have lower code capabilities, the initialized AST has more noise. Therefore, a lower pooling ratio is more effective in reducing noise and enhancing performance, with the optimal ratio being 0.1. In these cases, the pooling layer plays a more significant role in improving performance.

C Distribution of Text Embeddings

To evaluate the distribution of the text embeddings, we also calculate the standard deviation (SD) of MAM$'$:

$$\text{MAM}'_i = \frac{1}{N}\sum_{j=1}^{N} \text{sim}(\mathbf{c}_i, \mathbf{t}_j) \tag{4}$$

As shown in Table 7, we observe that after incorporating the GNN model, the SD of some MAM$'$ remains the same or even increases. The decrease in the SD of MAM is due to all MAM approaching zero. The increase in the SD of MAM$'$ is more complex. After adding the GNN model, the code embeddings become more dispersed, leading to more diverse observation angles for the text encoded space. However, since the text embeddings are not modified, their SD may slightly increase. Finally, the SD of MAM$'$ are even higher than those of MAM, suggesting that the distribution of text embeddings is less uniform compared with code embeddings. This may be because text embeddings share the encoding space with all texts, which may introduce a certain degree of non-uniformity. Given that GNN-Coder is mainly designed for the code modality, future research should focus on the non-uniformity issues in text embeddings.

model	Pooling Ratio	CSN							CosQA
		Ruby	JavaScript	Go	Python	Java	PHP	CSN_Avg	
UniXcoder 110M	0.1	**50.40**	**40.22**	**67.76**	**44.56**	**46.42**	**35.86**	**47.54**	**48.22**
	0.3	50.18	39.54	67.31	44.28	46.00	35.30	47.21	48.20
	0.5	49.75	39.78	67.07	43.93	46.10	35.25	46.98	47.16
	0.7	49.69	39.57	66.65	44.07	46.11	35.50	46.93	47.22
	0.9	49.73	39.68	65.63	43.57	45.97	35.33	46.65	46.64
CodeT5+ 110M	0.1	73.69	67.15	**90.71**	**70.37**	70.77	65.91	**73.10**	65.51
	0.3	73.72	67.08	90.48	70.36	70.76	65.93	73.06	64.51
	0.5	73.64	67.08	90.46	70.37	70.78	65.90	73.04	64.39
	0.7	**73.85**	**67.20**	90.47	70.36	**70.80**	**65.93**	**73.10**	**66.50**
	0.9	73.70	67.10	90.51	70.36	70.76	65.91	73.06	65.36
LLM-Embedder	0.1	**65.00**	**52.35**	**87.10**	**61.71**	**60.38**	**54.96**	**63.58**	**67.96**
	0.3	62.80	51.05	86.92	61.30	60.32	54.94	62.89	67.50
	0.5	62.63	50.66	86.92	61.59	60.32	54.76	62.81	66.50
	0.7	62.18	50.54	87.07	61.63	60.27	54.66	62.73	67.41
	0.9	61.73	50.35	86.97	61.40	60.35	54.45	62.59	67.60

Table 6: MRR of different pooling ratios with various models and datasets.

Model	CSN						CosQA
	Ruby	JavaScript	Go	Python	Java	PHP	
UniXcoder 110M	0.03	0.03	0.02	0.03	0.03	0.04	0.03
+ GNN-Coder	**0.02**	**0.01**	0.02	**0.02**	**0.02**	**0.02**	**0.02**
CodeT5+ 110M	0.02	0.02	0.02	0.02	0.02	0.02	0.05
+ GNN-Coder	0.02	0.02	0.02	0.02	0.02	0.02	**0.02**
LLM-Embedder	0.01	**0.01**	**0.01**	**0.01**	0.01	**0.01**	0.02
+ GNN-Coder	0.01	0.02	0.02	0.02	0.01	0.02	0.02

Table 7: The distribution of the text embeddings by different models on different datasets. The data in the table represents the standard deviation of MAM′. The mean value of MAM′ is the same as that of MAM.