Fine-Grained Features-based Code Search for Precise Query-Code Matching

Xinting Zhang[3,*], Mengqiu Cheng[4,*], Mengzhen Wang[2], Songwen Gong[2], Yi Cai[2], Jiayuan Xie[1,#], Qing Li[1]

[1]Department of Computing, The Hong Kong Polytechnic University
[2]School of Software Engineering, South China University of Technology
[3]Department of Mathematics, The University of Hong Kong
[4]Guangdong Neusoft University

***represents equal contribution**

#Correspondence:jiayuan.xie@polyu.edu.hk

Abstract

Code search aims to quickly locate target code snippets from databases using natural language queries, which promotes code reusability. Existing methods can effectively obtain aligned token-level and query word-level features. However, these studies usually represent the semantics of code and query by averaging the features of each token and word respectively, which makes it difficult to accurately capture the code details that are closely related to the query. To address this issue, we propose a fine-grained code search model that consists of a cross-modal encoder, a mapping layer, and a classification layer. Specifically, we utilize a pre-trained model, GraphCodeBERT, in the cross-modal encoder to align features. In the mapping layer, we introduce a co-attention network to capture the fine-grained interactions between code and query, ensuring a model can precisely identify key code segments relevant to the query. Finally, in the classification layer, we incorporate instruction learning techniques that leverage contextual reasoning to improve the accuracy of query-code matching. Experimental results show that our proposed model significantly outperforms existing methods across multiple programming language datasets.

1 Introduction

Code search aims to swiftly locating specific code snippets within extensive repositories, which is a pivotal task within software engineering (Gu et al., 2018). With a well-developed code search system, developers can harness the power of natural language to search for code snippets and reuse code written by others or previously written. This not only speeds up the software development cycle, but also helps developers understand various specific functions of the code, thereby speeding up debugging and problem solving (Sachdev et al., 2018).

Existing research on the code search task can be categorized into retrieval-based methods and deep learning-based methods. Retrieval-based methods, exemplified by Schütze et al. (2008) and Robertson et al. (1976), which primarily leverage structured information and keyword matching for code search. While adept at handling precise keyword queries, these methods are limited by their superficial understanding of code semantics and reliance on user-provided keywords or structural patterns. Conversely, deep learning-based approaches, such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021), utilize advanced models to grasp the relationship between code and natural language queries. These approaches aim to obtain appropriate fine-grained features for both the token level of the code and the word level of the query, effectively generating aligned semantic representations. However, they often rely on mean pooling or multi-layer perceptron techniques (Jian et al., 2021) based on fine-grained features to generate a vector representation of the entire code and the entire query respectively. However, the code search task requires models not only to understand the overall semantics of the code and query but also to capture the detailed information within both, in order to more accurately match and retrieve relevant code snippets. As illustrated in Figure 1, for the query "implement a system that supports user registration, password encryption, and password login", existing approaches typically generate a single representation vector for the entire query, treating "user registration", "password encryption", and "password login" as a whole. While these approaches capture the overall semantics of the query, they tend to miss finer-grained functional details. Specifically, a CodeBERT-based model selects **Option D**, which lacks the critical detail of password encryption. To address this issue, it is essential for models to conduct more fine-grained semantic analysis across both the query and the code, rather than relying solely on overall representation matching.

In this paper, we propose a fine-grained code

Proceedings of the 31st International Conference on Computational Linguistics, pages 7229–7238
January 19–24, 2025. ©2025 Association for Computational Linguistics

Query: implement a system that supports user registration, password encryption, and password login

Code, Option A:
```
import hashlib
class UserSystem:
    def __init__(self):
        self.users = {}
    def encrypt_password(self, password):
        return hashlib.sha256(password.encode()).hexdigest()
    def register(self, username, password):
        self.users[username] = self.encrypt_password(password)
    def login(self, username, password):
        encrypted_password = self.encrypt_password(password)
        if username in self.users and self.users[username] == encrypted_password:
            return True
        return False
```
... ...

Code, Option D:
```
class UserSystem:
    def __init__(self):
        self.users = {}
    def register(self, username, password):
        self.users[username] = password
    def login(self, username, password):
        if username in self.users and self.users[username] == password:
            return True
        return False
```

Figure 1: A sample of a code search task.

search (FCS) model, which is based on token-level code features and word-level query features to calculate the relevance between each other. In detail, our model consists of three modules: cross-modal encoder, mapping layer, and classification layer. Since code tokens and query words belong to two modalities, these features usually have different distributions and representations. Thus, the GraphCodeBERT (Guo et al., 2021) is introduced into the cross-modal encoder for feature alignment. To obtain suitable code representation, a co-attention network (CAN) (Xie et al., 2021) is introduced in our alignment mapping layer to learn the fine-grained relationships among code tokens and query words. It includes the inter-modal correlation between each token and each word, as well as the intra-modal connections among tokens and those among words. Specifically, the CAN model uses the self-attention of tokens or words, as well as the guided attention between tokens and words. With the development of large pre-trained models, we fully consider these technologies in the prediction process. In the classification layer, we utilize an instructional learning method to replace multi-layer perceptron techniques or vector similarity calculations (e.g., cosine) for classification. Specifically, we innovatively combine the code representation and original query text into a cohesive prompt. This synthetic prompt is then used as input to a large pre-trained model, i.e., RoBERTa (Liu et al., 2019), to guide the model to select the code snippet that is most consistent with the user query. More importantly, by understanding the instructions, our model can be more user-friendly, simplifying the interaction between the user and the technology.

The main contributions can be summarized:

• To the best of our knowledge, our work is the first to propose a code search model that simultaneously focuses on multiple key code tokens related to multiple query words, rather than treating all tokens as equally important. This innovation addresses the limitation in existing methods that fail to recognize the varying significance of tokens, enabling the model to more accurately capture the fine-grained semantic relationships between the given query and candidate codes.

• We introduce a co-attention network (CAN) combined with instruction learning to enhance the performance of the model. The CAN models the bidirectional interaction between code tokens and query words while instruction learning helps to better parse user queries and improve the model's interpretability. This design not only strengthens the model's understanding of queries but also simplifies user-system interaction, making it particularly suitable for complex code search tasks.

• We conduct experiments on the CodeSearchNet dataset (Husain et al., 2019), which includes six different programming languages. The results show that our proposed model significantly outperforms existing models in both automated and manual evaluations across all languages, particularly in handling fine-grained semantics and improving the precision of query-code matching.

2 Related Work

Early explorations on code search predominantly utilized direct information retrieval (IR) techniques (Hill et al., 2011; Huang et al., 2021), which regarded code search as the text matching task. These methods mainly utilize such as Bag of Words (BOW) (Schütze et al., 2008), TF-IDF (Robertson and Jones, 1976), and the extended boolean model (Lv et al., 2015). However, these approaches often fail to capture semantic information.

To address the limitations of IR-based approaches, deep learning-based approaches are used to capture semantic representations of queries and codes. Early models leverage MLP (Gu et al., 2018), LSTM (Wan et al., 2019), and Graph Neural Networks (GNN) (Ling et al., 2021) to obtain embeddings for queries and codes, which are then trained by computing similarity scores. Motivated

by the huge success of pre-trained language models (PLMs) in the NLP area and other application (Kuang et al., 2024; Qu et al., 2023), many researchers have introduced PLMs into the field of software engineering (Dong et al., 2023; Shi et al., 2023b; Hu et al., 2023). Specifically, CodeBERT (Feng et al., 2020) is a model pre-trained on unlabeled source code and comments. GraphCodeBERT (Guo et al., 2021) introduces the information of dataflow. Unixcoder (Guo et al., 2022) enhances code representation through cross-modal content like AST and code comments. CodeRetriever (Li et al., 2022) employs both unimodal and bimodal contrastive learning for pre-training in code search. Following pre-training, code PLMs can be fine-tuned on code search, significantly outperforming previous models. Additionally, CoCoSoDa (Shi et al., 2023a) effectively utilizes contrastive learning for code search by focusing on two key factors in contrastive learning: data augmentation and negative samples. Most of these methods directly average the vectors of each code token or query word to obtain their respective representations. However, not all code tokens or query words contribute equally to the matching process. Therefore, this paper focuses on fine-grained similarity calculations between query words and code tokens, aiming to better capture the key segments within the code.

3 Model

In the code search task, given a query Q and a set of candidate codes $C = \{c_1, ..., c_N\}$, our goal is to select the target code that matches the query. The overview of our model is shown in Figure 2, which consists of three modules: i) Cross-modal encoder, which aims to obtain the token-level of code and the word-level of query representations in the same vector space based on pre-trained models. ii) Mapping layer, which aims to obtain an entire code representation containing query information based on the attention mechanism. iii) Classification layer, which aims to input query and code representation into the language model for contextual reasoning and then classification through instructional learning. Details of each part of our framework are presented in the following sections.

3.1 Cross-Modal Encoder

Directly computing the relationship between query and code is a challenging task, which is due to the features of the two modalities (code and query text representation) usually have different vector space distributions. Therefore, a pre-trained code-language model GraphCodeBERT (Guo et al., 2021) is introduced into our cross-modal encoder to obtain the joint representation of text and code, i.e.,

$$(h_i^c, h^q) = GraphCodeBERT(c_i, Q), \quad (1)$$

where $h_i^c = \{h_{i,0}^c, ..., h_{i,L}^c\}$ represents the token-level feature of the i-th code content, L and represents the i-th code consisting of L tokens; $h^q = \{h_0^c, ..., h_M^c\}$ represents the word-level feature of the query, M and represents the query consisting of M words;

3.2 Mapping Layer

Since code snippets are typically long, directly averaging the representations of code tokens (i.e., h_i^c) tends to overlook the segments that are most relevant to the query. Therefore, it is essential to identify the relevance between each code token and each query term, allowing the model to accurately capture the key code fragments that are closely related to the query.

A co-attention network (CAN) is introduced to capture both the inter-modal relationships between code tokens and query words, and the intra-modal connections among tokens and among words. The CAN is designed to model self-attention within each token and query word, while also learning guided attention between code tokens and query terms. This process results in a multimodal representation that effectively integrates the knowledge from both the code and query, enhancing the model's ability to understand their relationship.

In detail, our CAN is based on a deep modular attention network (Xie et al., 2021; Chen et al., 2023, 2024a), which includes self-attention (SA) units and guided attention (GA) units. SA and GA are based on the scaled dot-product attention (Vaswani et al., 2017), which includes queries Q, keys K, and values V. For simplicity, we set their dimensions to the same value as the dimension d of the query Q, i.e., $d = d_{query} = d_{key} = d_{value}$. The attention feature F is obtained by weighted summation of all values V with respect to the attention learning form Q and K. The calculation is as follows:

$$F = A(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V. \quad (2)$$

The SA and GA units have the same architecture, which consists of the scaled dot-product attention

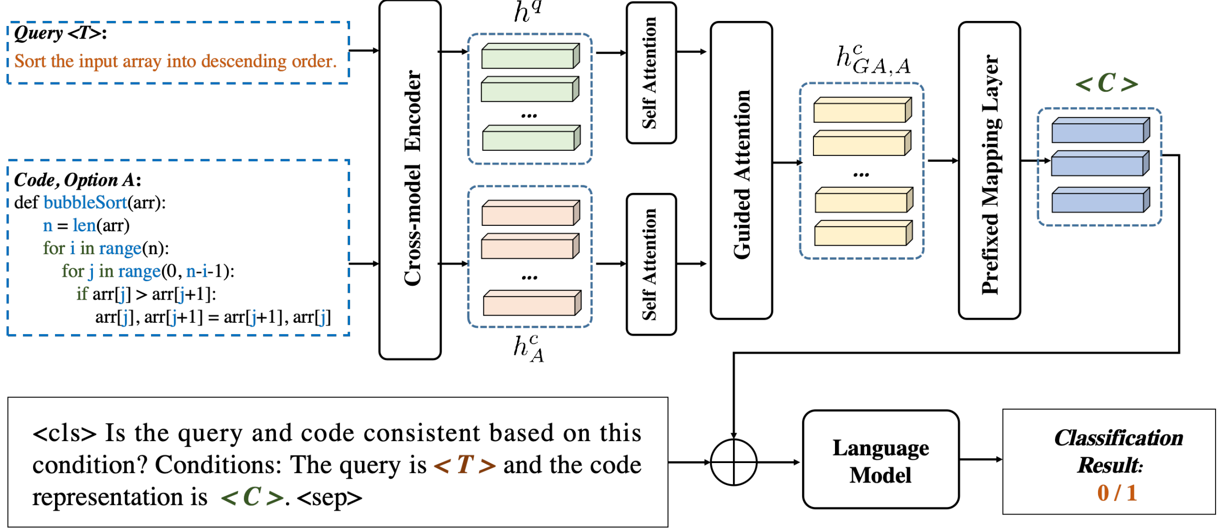

Figure 2: Overview of our fine-grained code search model.

and feed-forward layer. In the attention layer, SA utilizes one group input features X as queries, keys, and values. And GA has two groups of input features X and Y, where X is input as queries and Y is input as the keys and values. Then, the output feature of the attention layer is transformed through two fully-connected layers with a ReLU activation. In detail, we take the query word representation h^q and the token-level of i-th code representation h_i^c as input features X into independent SA units to capture the inner connection of the tokens or words, respectively. Then, the output feature of the i-th code SA (i.e., $h_{SA,i}^c$) is input to GA as queries, and the output feature of the query SA (i.e., h_{SA}^q) is input to GA as keys and values. The GA unit models the pairwise relationship between each paired code token and query word to obtain the i-th code representation that contains the query information:

$$h_{SA,i}^c = SA(h_i^c, h_i^c, h_i^c), \tag{3}$$

$$h_{SA}^q = SA(h^q, h^q, h^q), \tag{4}$$

$$h_{GA,i}^c = SA(h_{SA,i}^c, h_{SA}^q, h_{SA}^q), \tag{5}$$

where $h_{GA,i}^c = \left\{ h_{GA,i,j}^c \right\}_{j=0}^{L}$, and SA($\cdot$) and GA($\cdot$) correspond to A($\cdot$) in the equation (2).

3.3 Classification Layer

In our classification layer, we aim to leverage popular pre-trained model techniques to replace the existing simple vector similarity measures (e.g., cosine similarity) and improve the accuracy of query-code matching. Inspired by large-scale models' success in instruction learning, we integrate the generated representations of code and original query into a context reasoning module for cross-modal inference. Specifically, we utilize a pre-trained large language model (i.e., RoBERTa) as the context reasoning module. During the instruction learning process, we populate a pre-defined instruction template with the query text and the candidate code representations. The template is formatted as: "<cls> Is the query and code consistent based on this condition? Conditions: The query is <T> and the code representation is <C>. <sep>" Here, <T> represents the original query text, and <C> represents the code representation.

In detail, the current code representations we obtained from the mapping layer are expressed in vector form, but to adapt them to RoBERTa's input format, we need to map them through a prefixed mapping layer (Chen et al., 2024b), i.e., a fully connected layer. We set the prefix length of the code representation to P, i.e., using P input tokens for RoBERTa to represent the code information. Each of these tokens has a vector length that matches the hidden layer dimension D of RoBERTa, ensuring that the code representation can seamlessly integrate into RoBERTa's context encoding process. Thus, the mapped code indicates that the size of <C> is $[P \times D]$. By using this approach, we can harness the contextual learning capabilities of pre-trained language models to more effectively address the code search task.

Then, the sequence representation is fed into the context reasoner to infer the final result. In this way, we can leverage the contextual learning capabilities of pre-trained language models to solve

	Training	Test	Candidates code N
Python	251,820	1200	43,827
PHP	241,241	1200	52,660
Go	167,288	1200	28,120
Java	164,923	1200	40,347
JavaScript	58,025	1200	13,981
Ruby	24,927	1,261	4,360

Table 1: Distribution of the number of different programming languages in the dataset.

multi-modal reasoning problems. We obtain the inference results for each candidate answer by applying a two-layer perceptron with ReLU activation function on the output hidden state $h_{cls,i}$ of the top layer of RoBERTa. The whole training objective of our model can be defined as,

$$p_i = Linear(h_{cls,i}), \tag{6}$$

$$\ell = \sum_{1}^{N} logP_i(p_i = y), \tag{7}$$

where p_i is the output probability on i-th candidate code, y is the label and N represents the number of candidate codes.

3.4 Inference Stage

In the inference stage, the model for the code search task needs to retrieve the target code closest to the query from multiple candidate codes. Thus, we compute the results for each candidate code and query separately, i.e.,

$$score_i = Linear(h_{cls,i}), \tag{8}$$

where i ranges from 1 to N, and N is the number of candidate codes. By sorting the scores, we use the code at the position corresponding to the value mv with the highest score as the target result. The calculation of mv is as follows,

$$mv = Max([score_1, ..., score_N]). \tag{9}$$

4 EXPERIMENTs

4.1 Dataset

In this paper, we evaluate the code search models on the CodeSearchNet dataset (Husain et al., 2019). The dataset originates from public projects on GitHub and covers six popular languages, i.e., Python, Java, JavaScript, PHP, Ruby, and Go. Notably, due to computational resource considerations, except for the Ruby dataset, the test sets of the other five datasets contained a large number of samples. Therefore, we randomly selected 1200 samples from the test sets for each of the five programming languages to conduct our evaluation.

To ensure the simulation of real retrieval scenarios and thoroughly test the performance of our proposed method, we retained all code snippets in the original data as the candidate codes. The distribution of the dataset is shown in Table 1.

4.2 Baseline Methods and Settings

To evaluate the effectiveness of our framework, we compare our model with the following methods.

• **CodeBERT** (Feng et al., 2020) is a bi-encoder Transformer-based model pre-trained to bridge the gap between programming and natural language. It leverages the Transformer architecture to align code and text. We use the official implementation to ensure fidelity.

• **GraphCodeBERT** (Guo et al., 2021), an improved version of CodeBERT, adds dataflow awareness to capture both syntactic structure and data flow in code, improving the model's ability to understand complex programming relationships. We applied the official implementation.

• **UniXcoder** (Guo et al., 2022) is a language model that uses a multi-encoder approach to handle various programming languages simultaneously, excelling at cross-language pattern recognition. We used the official repository's implementation.

• **mAdapter** (Wang et al., 2023) explores the use of adapter fine-tuning to address performance degradation in multi-language fine-tuning, proving effective in cross-language, multi-language, and low-resource scenarios through empirical evidence.

• **CoCoSoDa** (Shi et al., 2023a) leverages multimodal momentum contrastive learning and soft data augmentation for code search.

4.3 Evaluation

In the evaluation of the code search model, we implemented two widely recognized evaluation metrics on the test set: MRR (Mean Reciprocal Rank) and R@K. These metrics have been established as standards in previous code search research (Li et al., 2022). Higher values of MRR and R@K indicate superior code search performance.

In detail, MRR quantifies the effectiveness of a search algorithm by scoring it and then ranking search results in descending order based on their relevance to the search query. For the J test query,

Model	Java			Python			Go		
	R@1	R@5	MRR	R@1	R@5	MRR	R@1	R@5	MRR
CodeBERT	0.515	0.675	0.583	0.505	0.685	0.595	0.825	0.955	0.879
GraphCodeBERT	0.625	0.835	0.722	0.590	0.830	0.705	0.860	0.965	0.907
UniXcoder	0.630	0.840	0.719	0.600	0.845	0.711	0.900	0.975	0.920
mAdapter-CodeBERT	0.580	0.805	0.681	0.585	0.815	0.860	0.686	0.840	0.888
mAdapter-GraphCodeBERT	0.600	0.825	0.701	0.590	0.820	0.885	0.695	0.820	0.885
mAdapter-UniXcoder	0.625	0.835	0.719	0.585	0.850	0.789	0.704	0.870	0.917
CoCoSoDa	0.650	0.875	0.747	0.635	0.880	0.740	0.875	0.995	0.916
Ours	**0.665**	**0.895**	**0.752**	**0.700**	**0.905**	**0.887**	**0.880**	**0.975**	**0.921**

Model	Javascript			PHP			Ruby		
	R@1	R@5	MRR	R@1	R@5	MRR	R@1	R@5	MRR
CodeBERT	0.415	0.645	0.515	0.425	0.630	0.518	0.555	0.739	0.642
GraphCodeBERT	0.515	0.745	0.625	0.510	0.705	0.605	0.610	0.820	0.707
UniXcoder	0.575	0.775	0.669	0.495	0.750	0.610	0.647	0.859	0.739
mAdapter-CodeBERT	0.515	0.775	0.627	0.470	0.680	0.574	0.614	0.835	0.715
mAdapter-GraphCodeBERT	0.510	0.755	0.617	0.530	0.730	0.623	0.636	0.845	0.733
mAdapter-UniXcoder	0.535	0.770	0.644	0.540	0.740	0.634	0.648	0.860	0.745
CoCoSoDa	0.580	0.805	0.678	0.520	**0.760**	0.626	0.649	0.872	0.749
Ours	**0.615**	**0.815**	**0.702**	**0.590**	**0.760**	**0.671**	**0.671**	**0.886**	**0.759**

Table 2: Main automatic metrics results of baselines and our model on six programming languages.

the MRR is calculated as follows:

$$MRR = \frac{1}{J}\sum_{i=1}^{J}\frac{1}{Result_i}. \quad (10)$$

R@K evaluates the relevance of the top K results to the query, reflecting the user's actual expectation to prefer the top most relevant results. We calculate R@K with K = 1, 5, and 10, respectively.

4.4 Implementation Details

We implement our approach based on PyTorch. During the training stage, we use the Adam optimizer to train our model and baselines on a single 3090 GPU. Its basic learning rate is 1e-6 and the weight decay is 0.01. The hidden layer dimension D is 768, and the dropout rate is 0.1. The prefix length of the code P is set to 5.

4.5 Results and Analysis

4.5.1 Automatic Evaluation Result

Table 2 presents the results in six programming languages. We find that: (i) Across all six programming languages, our model consistently outperforms baseline models. For example, in Java, our model achieves an R@1 of 0.665 and an R@5 of 0.895, which are both higher than the top-performing baseline, CoCoSoDa (0.650 and 0.875 respectively). This demonstrates the overall superiority of our approach in capturing relevant code snippets in multi-language settings.

Method	R@1	R@5	R@10	MRR
CodeBERT	0.555	0.739	0.805	0.642
Ours w CB	0.663	0.851	0.890	0.747
GraphCodeBERT	0.610	0.820	0.872	0.707
Ours w GCB	0.671	0.866	0.904	0.759

Table 3: Compare the performance of using different code pre-trained models (i.e., CodeBERT and GraphCodeBERT) as encoders in our model. "Ours w CB" represents "Ours model with CodeBERT"; and "Ours w GCB" represents "Ours model with GraphCodeBERT".

(ii) Results for mAdapter variants (e.g., mAdapter-CodeBERT, mAdapter-UniXcoder) show improved performance when fine-tuning on multiple languages. For example, mAdapter-GraphCodeBERT in Ruby achieves better R@1 (0.636) compared to GraphCodeBERT (0.610), showing that adapter tuning helps in cross-language fine-tuning scenarios. This suggests that adapter-based fine-tuning can be effective in reducing the negative impact of multi-language fine-tuning.

(iii) While models like UniXcoder perform reasonably well in individual languages (e.g., in

Model	R@1	R@5	R@10	MRR
CS_rCAN	0.546	0.751	0.839	0.706
CS_T	0.632	0.827	0.891	0.724
CS_rIn	0.649	0.860	0.900	0.746
Ours	**0.671**	**0.866**	**0.904**	**0.759**

Table 4: Comparison results of ablation studies in Ruby. **Bold**: the best performance in the column.

Length	**R@1**	**R@5**	**R@10**	**MRR**
1	0.662	0.847	0.896	0.750
2	0.660	0.843	0.895	0.749
3	0.668	0.857	0.902	0.758
4	0.670	0.863	**0.904**	**0.760**
5	**0.671**	**0.866**	**0.904**	0.759
6	0.664	0.862	0.901	0.756

Table 5: Comparison results of different lengths in Ruby. **Bold**: the best performance in the column.

Python with an R@5 of 0.845), they struggle to maintain consistent performance across all languages. This highlights the importance of advanced fine-tuning strategies, such as adapter-based tuning, to handle multi-language tasks where resource availability and language characteristics vary.

4.5.2 Comparison of Different Encoders

To evaluate the applicability of our proposed method, we used two pre-trained code models (i.e., CodeBERT and GraphCodeBERT) as cross-modal encoders for our model and conducted comprehensive experiments on the Ruby dataset, with the results shown in Table 3.

Specifically, our model with CodeBERT shows notable improvements across all metrics, with R@1 increasing from 0.555 to 0.663, and MRR improving from 0.642 to 0.747. Similarly, in experiments with GraphCodeBERT, our model demonstrates excellent performance, with R@1 rising from 0.610 to 0.671, and MRR increasing from 0.707 to 0.759. These results indicate that our approach not only enhances the performance of models based on GraphCodeBERT but also improves those based on CodeBERT. Thus, the experimental results fully demonstrate the broad applicability of our method across various pre-trained code models and its strong generalizability in cross-modal reasoning and code search tasks.

4.6 Ablation Studies

We conduct a comprehensive ablation study to determine the impact of different components in our method, and the results are detailed in Table 4. Specifically, CS_rCAN means removing the CAN mechanism and using a direct averaging method to obtain code representation; CS_T means that the query text is also used in the form of an average vector for instructions. CS_rIn involves eliminating instruction templates and directly concatenating all information. We find that:

(i) In the CS_rCAN experiment, we remove the CAN mechanism and use a direct averaging method to obtain the code representation. The results show that R@1 is only 0.546, R@5 is 0.751, R@10 is 0.839, and MRR is 0.706. Compared to the full model, the performance dropped significantly, indicating the importance of the CAN mechanism in capturing code representations and effectively aligning the code with the query.

(ii) In the CS_T setting, we treat the query text as an average vector. The results show an improvement, with R@1 reaching 0.632, R@5 at 0.827, R@10 at 0.891, and MRR at 0.724. While this approach improves performance to some extent, it still falls short of the full model, indicating that preserving the context of the query text plays an important role in achieving better performance.

(iii) In the CS_rln experiment, we remove the instruction templates and directly concatenate all information. The results show R@1 reaching 0.649, R@5 at 0.870, R@10 at 0.900, and MRR at 0.746. Although this method performs relatively well, it still underperforms compared to the full model, highlighting the key role that instruction templates play in cross-modal reasoning.

(iv) Compared to other experiments, the full model achieves the best results across all metrics, with R@1 at 0.671, R@5 at 0.866, R@10 at 0.904, and MRR at 0.759. This shows that the combination of all components in our method maximizes the model's performance, especially in code search tasks, where complete context reasoning and instruction learning are crucial.

4.7 Effect of Prefix Length on Model Performance

Table 5 reflects the impact of mapping code vector representations to different prefix lengths on model performance. We find that: (i) It can be observed that as the prefix length increases, the model's scores across all four metrics also rise. When the

Query: Returns true if the inject annotation is on the constructor.

Our Model (✓)
```
private static boolean hasAtInject(Constructor cxtor) {
   return cxtor.isAnnotationPresent(Inject.class)
          || cxtor.isAnnotationPresent(javax.inject.Inject.class);
}
```

CoCoSoDa (✗)
```
private static boolean hasDeprecatedAnnotation(Method method) {
     return method.isAnnotationPresent(Deprecated.class);
  }
```

mAdamAdapter-UniXcoder (✗)
```
private static boolean isFieldTypeString(Field field) {
     return field.getType().equals(String.class);
  }
```

GraphCodeBERT (✗)
```
public void visitOuterClass(final String owner, final String name,
final String descriptor) {
      if (cv != null) {
        cv.visitOuterClass(owner, name, descriptor);
      }
     }
```

Figure 3: Case study of sample outputs in JAVA.

prefix length is set to 4, the model shows improvement in R@10 and MRR, with R@10 reaching 0.904 and MRR reaching 0.760. This suggests that the model is better able to learn and capture contextual information at this length.

(ii) When the prefix length is set to 5, the model achieves its best results in R@1 and R@5, with scores of 0.671 and 0.866, respectively, indicating optimal performance for top-1 and top-5 retrieval results. However, R@10 remains the same as at length 4, and MRR is slightly lower than at length 4. This may be due to the introduction of redundant or inaccurate information, which affects the model's ability to learn from context.

(iii) When the prefix length increases to 6, although the scores for R@1, R@5, and R@10 are close to the highest values, overall performance slightly declines. This suggests that longer prefixes may introduce unnecessary information, negatively impacting model performance. Therefore, an appropriate prefix length can effectively enhance model performance, while overly long prefixes may introduce noise.

4.8 Case Study

Figure 3 presents the experimental results of our model compared to the baseline models CoCoSoDa and mAdamAdapter-UniXcoder on Java programs. We observed the following: (i) Our model demonstrates superior semantic understanding when handling queries, while other baseline models (such as CoCoSoDa and mAdamAdapter-UniXcoder) failed to accurately grasp the core semantics of the query. For instance, CoCoSoDa incorrectly checked for the @Deprecated annotation, which clearly deviates from the query requirements, and mAdamAdapter-UniXcoder focused on field type checks, neglecting the annotation on the constructor, thus missing the query's semantic needs.

(ii) Our model precisely captures every crucial aspect of the query, showing exceptional recognition of fine-grained features, especially in the code search task. In contrast, other models only focus on superficial or irrelevant details, leading to incorrect results.

5 Conclusion

In this paper, we propose a fine-grained code search model that leverages co-attention networks (CAN) to address the limitations of existing methods, which typically average code and query representations, leading to the loss of critical details. Our approach integrates a cross-modal encoder with GraphCodeBERT for feature alignment, a mapping layer with CAN to capture fine-grained interactions between code and queries, and a classification layer enhanced with instruction learning for contextual reasoning. Through extensive experiments on multiple programming language datasets, our model demonstrated improvements over state-of-the-art methods, particularly in its ability to accurately match code snippets with queries by focusing on detailed semantic features.

6 Limitations

While our model has demonstrated improvements across multiple programming language datasets, there are still some limitations. First, the performance of the model relies on large, high-quality datasets, which may be difficult to obtain in certain programming languages or domains. Additionally, although the collaborative attention mechanism enhances the model's ability to capture fine-grained semantics, it also increases computational complexity, particularly when dealing with large-scale codebases. Despite the excellent performance in our experiments, the method has not yet been widely tested in real-world development environments. Future work will focus on optimizing the model's efficiency in low-resource settings and validating its effectiveness in practical applications.

Acknowledgments

This research is supported by the Science and Technology Planning Project of Guangdong Province (2020B0101100002), the National Natural Science Foundation of China (62076100, 62476097), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2240100), Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the China Computer Federation (CCF)-Zhipu AI Large Model Fund, the Hong Kong Research Grants Council through Research Impact Fund (project no. R1015-23).

References

Jiali Chen, Yi Cai, Ruohang Xu, Jiexin Wang, Jiayuan Xie, and Qing Li. 2024a. Deconfounded emotion guidance sticker selection with causal inference. In *Proc. of ACM MM*, pages 3084–3093. ACM.

Jiali Chen, Zhenjun Guo, Jiayuan Xie, Yi Cai, and Qing Li. 2023. Deconfounded visual question generation with causal inference. In *Proc. of ACM MM*, pages 5132–5142. ACM.

Jiali Chen, Xusen Hei, Yuqi Xue, Yuancheng Wei, Jiayuan Xie, Yi Cai, and Qing Li. 2024b. Learning to correction: Explainable feedback generation for visual commonsense reasoning distractor. In *Proc. of ACM MM*, pages 8209–8218. ACM.

Hande Dong, Jiayi Lin, Yichong Leng, Jiawei Chen, and Yutao Xie. 2023. Retriever and ranker framework with probabilistic hard negative sampling for code search. *CoRR*, abs/2305.04508.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Proc. of EMNLP*, pages 1536–1547.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proc. of ICSE*, 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. In *Proc. of ACL*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *Proc. of ICLR*.

Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proc. of ASE*, pages 524–527.

Fan Hu, Yanlin Wang, Lun Du, Xirong Li, Hongyu Zhang, Shi Han, and Dongmei Zhang. 2023. Revisiting code search in a two-stage paradigm. In *Proc. of WSDM*, pages 994–1002.

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. In *Proc. of ACL/IJCNLP*, pages 5690–5700.

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

Junming Jian, Wei Xia, Rui Zhang, Xingyu Zhao, Jiayi Zhang, Xiaodong Wu, Yong'ai Li, Jinwei Qiang, and Xin Gao. 2021. Multiple instance convolutional neural network with modality-based attention and contextual multi-instance learning pooling layer for effective differentiation between borderline and malignant epithelial ovarian tumors. *Artif. Intell. Medicine*, 121:102194.

S Kuang, Y Liu, X Wang, et al. Harnessing multimodal large language models for traffic knowledge graph generation and decision-making.

Senyun Kuang, Yang Liu, Xin Wang, Xinhua Wu, and Yintao Wei. 2024. Harnessing multimodal large language models for traffic knowledge graph generation and decision-making. *Communications in Transportation Research*, 4:100146.

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 *Proc. of EMNLP*, pages 2898–2910.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5):88:1–88:21.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on API understanding

and extended boolean model (E). In *Proc. of ASE*, pages 260–270.

Xiaobo Qu, Hongyi Lin, and Yang Liu. 2023. Envisioning the future of transportation: Inspiration of chatgpt and large models. *Communications in Transportation Research*, 3:100103.

Stephen E Robertson and K Sparck Jones. 1976. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proc. of MAPL@PLDI*, pages 31–41.

Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023a. Cocosoda: Effective contrastive learning for code search. In *Proc. of ICSE*, pages 2198–2210.

Zejian Shi, Yun Xiong, Yao Zhang, Zhijie Jiang, Jinjing Zhao, Lei Wang, and Shanshan Li. 2023b. Improving code search with multi-modal momentum contrastive learning. In *Proc. of ICPC*, pages 280–291.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proc. of NeurIPS*, pages 5998–6008.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *Proc. of ASE*, pages 13–25.

Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One adapter for all programming languages? adapter tuning for code search and summarization. In *Proc. of (ICSE)*, pages 5–16.

Jiayuan Xie, Yi Cai, Qingbao Huang, and Tao Wang. 2021. Multiple objects-aware visual question generation. In *Proc. of ACM MM*, pages 4546–4554.