Fine-grained Co-Attentive Representation Learning for Semantic Code Search

Zhongyang Deng[1,2], Ling Xu[1,2*], Chao Liu[1,2], Meng Yan[1,2], Zhou Xu[1,2], Yan Lei[1,2]
[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[2]School of Big Data and Software Engineering, Chongqing University, Chongqing, China
Email:{zy.deng, xuling, liu.chao, mengy, zhouxullx, yanlei}@cqu.edu.cn

***Abstract*—Code search aims to find code snippets from large-scale code repositories based on the developer's query intent. A significant challenge for code search is the semantic gap between programming language and natural language. Recent works have indicated that deep learning (DL) techniques can perform well by automatically learning the relationships between query and code. Among these DL-based approaches, the state-of-the-art model is TabCS, a two-stage attention-based model for code search. However, TabCS still has two limitations: semantic loss and semantic confusion. TabCS breaks the structural information of code into token-level words of abstract syntax tree (AST), which loses the sequential semantics between words in programming statements, and it uses a co-attention mechanism to build the semantic correlation of code-query after fusing all features, which may confuse the correlations between individual code features and query.**

In this paper, we propose a code search model named FcarCS (Fine-grained Co-Attentive Representation Learning Model for Semantic Code Search). FcarCS extracts code textual features (i.e., method name, API sequence, and tokens) and structural features that introduce a statement-level code structure. Unlike TabCS, FcarCS splits AST into a series of subtrees corresponding to code statements and treats each subtree as a whole to preserve sequential semantics between words in code statements. FcarCS constructs a new fine-grained co-attention mechanism to learn interdependent representations for each code feature and query, respectively, instead of performing one co-attention process for the fused code features like TabCS. Generally, this mechanism leverages row/column-wise CNN to enable our model to focus on the strongly correlated local information between code feature and query.

We train and evaluate FcarCS on an open Java dataset with 475k and 10k code/query pairs, respectively. Experimental results show that FcarCS achieves an MRR of 0.613, outperforming three state-of-the-art models DeepCS, UNIF, and TabCS, by 117.38%, 16.76%, and 12.68%, respectively. We also performed a user study for each model with 50 real-world queries, and the results show that FcarCS returned code snippets that are more relevant than the baseline models.

***Index Terms*—code search, code structural feature, fine-grained co-attention mechanism, representation learning**

I. Introduction

To improve the software development productivity, developers frequently search and reuse existing source code snippets from a codebase in large-scale software repositories (e.g., GitHub) by leveraging a code search model [1]–[6]. Generally, a code search model aims to find the code snippets whose programming semantics match the developer's intent expressed in natural language queries [7].

Early models leveraged the information retrieval (IR) techniques to complete the code search task [4], [8]–[12]. Sourcerer [8] is a representative model. It regards code snippets as plain text, builds indexes for them using the text search engine Lucene, and finally performs code search by matching keywords between query and the indexed code snippets. However, simple keyword matching can hardly bridge the semantic gap between query and code snippets. To solve this issue, Lv et al. [9] proposed the model CodeHow that extends queries with more programming context in terms of related APIs and improved the keyword matching with the extended Boolean model.

In recent years, researchers have demonstrated that the deep learning (DL) technique is a better choice for code search compared with the traditional IR-based models [13]–[16]. The reason is that DL-based models can directly learn the semantic relationship between code snippets and related natural language descriptions by embedding them into a shared vector space [13]. These models can be trained by pairs of code snippets and corresponding comments in natural language [13]. In this way, the code snippets relevant to a query can be conducted by calculating the cosine similarity between the query vector and the vectors of code snippets in the codebase [13].

DeepCS [13] is a representative DL-based model that represents code snippets by three features (i.e., method name, API sequence, and tokens) and adopted the long and short-term memory (LSTM) [17] and multi-layer perceptron (MLP) [18] to embed query and code snippets. Recently, researchers proposed some models to tackle the disadvantages of DeepCS. UNIF [19] is a simpler model that replaces the complex embedding (i.e., LSTM and MLP) with a shallow neural network model fastText [20] and improves the performance with attention mechanism [21]. Meanwhile, Shuai et al. [15] observed that DeepCS fails to work in many cases because it ignores the complex semantic correlation between query and code snippets. To address this challenge, they proposed an improved model CARLCS-CNN by performing embedding with convolutional neural network (CNN) [22] and leveraging a co-attention mechanism [23]–[27] to learn the semantic correlation. Later, Yang et al. [16] proposed a state-of-the-

[*]Corresponding author.

art model named TabCS. It extends CARLCS-CNN with an abstract syntax tree (AST) feature to better represent the code semantics. Experimental results showed that TabCS outperforms DeepCS, UNIF, and CARLCS-CNN significantly.

```
// Query: How can I convert a stack trace to a string?
// Code:
1:    public static String stackTraceToString(final Exception exception) {
2:        StringWriter sw = new StringWriter();
3:        PrintWriter pw = new PrintWriter(sw);
4:        exception.printStackTrace(pw);
5:        return sw.toString();
      }
```

Fig. 1. An example of the code snippet.

Nevertheless, we observed that the TabCS model has two limitations:

1) Semantic loss in code representation. It breaks the code semantics (e.g., "sw = new StringWriter(); pw = new PrintWriter(sw);" in Fig. 1) into a set of token-level words (e.g., "StringWriter", "PrintWriter", "sw", "pw", and "new") when generating serialized AST feature, which loses the sequential semantics between words in programming statements.

2) Semantic confusion in co-attentive representation. TabCS fuses four code features (i.e., method name, API sequence, tokens, and AST) before learning the co-attentive representation between code and query. However, the fusion of different code features may generate wrong co-attentive semantics with query. As illustrated in Fig. 1, although the query words (e.g., "stack", "trace", "to", "string") are strongly correlated with method name (i.e., "stackTraceToString") and APIs (i.e., "printStackTrace", "toString"), the concatenation of all features (e.g., "stackTraceToString", "public", "static", ..., "toString") containing many irrelevant code words reduces and confuses the semantic correlations between individual code feature and query. Therefore, this semantic confusion will undoubtedly affect the semantic matching of code and query.

To address these issues, we propose a **F**ine-grained **C**o-**A**ttentive **R**epresentation Learning Model for Semantic **C**ode **S**earch (FcarCS). As shown in Fig. 2, this model introduces a statement-level structural feature *StaTree*, named statement tree sequence, which is a better input for code representation. This feature extracts subtrees from AST according to code statements and parses them into many sequences by depth-first traversal. Each sequence is treated as a whole to preserve sequential semantics between words in different code statements (e.g., transforming "S1" to "SubTree1"). Besides, instead of performing one co-attention process for the fused features, we construct a new fine-grained co-attention mechanism to learn the interdependent representations between query and each code feature, respectively. Generally, this mechanism leverages row/column-wise CNN to enable our model to focus on the strongly correlated local information between code feature and query. Moreover, we perform the representation fusion by assigning a larger weight for the representation with a higher correlation.

To verify the model's effectiveness, we evaluate FcarCS on the widely adopted large-scale Java dataset provided by Hu et al. [28]. The training and testing data contain 475k and 10k code-query pairs. The experimental results show that FcarCS achieves an MRR (Mean Reciprocal Rank) of 0.613, significantly outperforming three state-of-the-art models DeepCS [13], UNIF [19], and TabCS [16] by 117.38%, 16.76%, and 12.68%, respectively. Meanwhile, we performed a user study on 50 real-world queries [13]. The result shows that FcarCS achieves more relevant results with an average FRank (the rank position of the first result) of 5.12, reducing the average FRank of DeepCS, UNIF, and TabCS by 36.9%, 23.6%, and 25.1% respectively, and this reduction is significant under the Wilcoxon signed rank test [29].

In summary, the main contributions of this study are:

- We propose a Fine-grained Co-Attentive Representation Learning model for Semantic Code Search (FcarCS), which can capture the sequential semantics between words for different code statements, distinguish the correlated representation between query and each code feature respectively, and enhance representation fusion by considering the contribution of each correlated representation.
- We conduct a code search experiment on a large-scale dataset, and the FcarCS model outperforms three state-of-the-art models (DeepCS, UNIF, and TabCS) significantly.
- We perform a user study on 50 real-world queries, and find that the FcarCS can substantially search code snippets more relevant than DeepCS, UNIF, and TabCS.
- We publicize our replication package[1] with complete source code, trained model, and datasets for reproducing our experiment.

The outline for this article is organized as follows. Section II describes the background of code search. Section III presents our proposed model FcarCS. Section IV describes the experimental setup. Section V and VI show the experimental results and discussion, respectively. Section VII introduces related work. Finally, section VIII summarizes this work and prospects the future work.

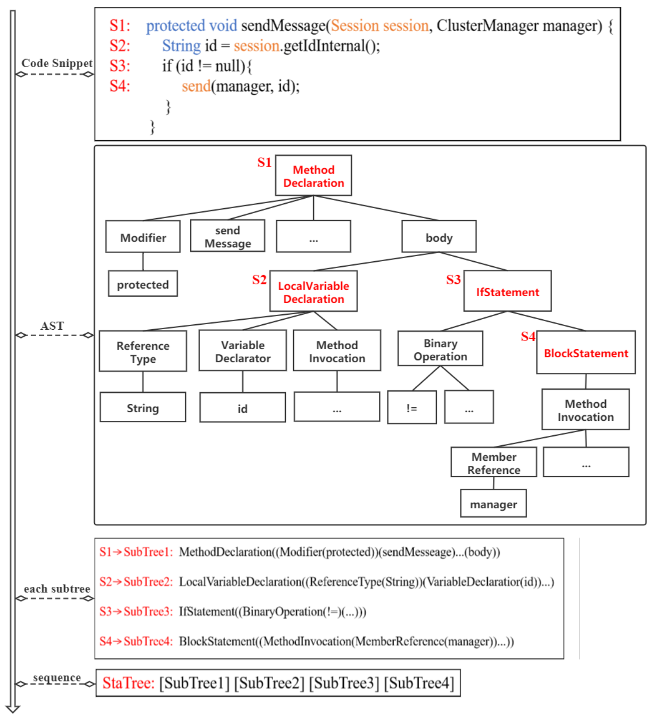
II. Background

This section describes the techniques involved in FcarCS: word embedding for code search and joint embedding.

A. Word Embedding for Code Search

Code Textural Features Embedding. Source code can be treated as texts. Every code snippet like Fig. 1 can be divided into three textual components: method name s_{name} (a list of camel split tokens), API sequence s_{api} (a sequence of API words used in the code snippets), tokens s_{token} (a bag of words used in the code snippets). First, each component is encoded by a vocabulary with the top-n frequently occurred words, respectively. Then, three individual word embedding layers are applied for these components to transform them into vectors with the same dimension. Finally, each component

[1]https://github.com/cqu-isse/FcarCS


```
S1:  protected void sendMessage(Session session, ClusterManager manager) {
S2:      String id = session.getIdInternal();
S3:      if (id != null){
S4:          send(manager, id);
         }
     }
```

```
S1→SubTree1:  MethodDeclaration((Modifier(protected))(sendMesseage)...(body))
S2→SubTree2:  LocalVariableDeclaration((ReferenceType(String))(VariableDeclaratior(id))...)
S3→SubTree3:  IfStatement((BinaryOperation(!=)(...)))
S4→SubTree4:  BlockStatement((MethodInvocation(MemberReference(manager))...))
```

```
StaTree: [SubTree1] [SubTree2] [SubTree3] [SubTree4]
```

Fig. 2. The process of extracting the statement tree sequence.

vector is further embedded by a neural network model. In TabCS, as Eq. (1), it uses an attention mechanism to implement the embedding process for the textual features, so as to capture better semantic information [30]–[32].

$$\begin{aligned} v_{textual} = Att_1(E(s_{name})) + Att_2(E(s_{api})) \\ + Att_3(E(s_{token})) \end{aligned} \tag{1}$$

where E represents a word embedding layer, Att refers to attention mechanism. Note that different code search models may use different embedding networks.

Code Structural Features Embedding. Abstract Syntax Tree (AST) is a structural tree and its nodes correspond to the structure or symbol of the source code, which can capture the code's lexical information and syntactic structure. In TabCS, it uses a new structure-based traversal of AST to obtain a token-level sequence s_{AST} as code structural feature, as Eq. (2), which is used to represent code snippets through an attention mechanism.

$$v_{structural} = Att_4(E(s_{AST})) \tag{2}$$

Query Embedding. Similarly, a query in natural language is treated as a list of words s_{query}. These words are encoded and transformed into the same dimensional query vector by an embedding layer. In TabCS, as Eq. (3), the query is further embedded by an attention mechanism.

$$v_{query} = Att_5\left(E\left(s_{query}\right)\right) \tag{3}$$

B. Joint Embedding

When completing the code and query embedding, the DL-based model learns a joint embedding between v_{code} and v_{query}. Finally, all code snippets are ranked according to their cosine similarities to the corresponding query. The cosine similarity is computed as follows:

$$cosine = \frac{v_{code} \cdot v_{query}}{\|v_{code}\| \cdot \|v_{query}\|} \tag{4}$$

Based on the value of cosine similarity, the model recommends a list of codes for users to choose from.

III. Our Model

In this section, we present the overall structure and details of the proposed model FcarCS. Fig. 3 shows the overall framework of FcarCS, which implements code search by three phases: feature representation, fine-grained co-attention, and feature fusion. The first phase feeds code features (i.e., method name, API sequence, tokens, and statement tree sequence) and query features (i.e., tokens) into an attention mechanism to obtain code/query feature matrices. Then, the second phase feeds the feature matrices into four fine-grained co-attention mechanisms respectively to get the fine-grained correlations between each code feature and query to enhance code/query feature vectors. In the third phase, the interdependent feature vectors are respectively fused into final code/query vectors. Finally, FcarCS calculates the cosine similarity between two representative vectors.

A. Feature Representation

This phase describes the process of representing features, including code features representation and query features representation.

Code features representation. In our model, we treat code snippet as a four-tuple <method name, API sequence, tokens, statement tree>. The first three elements reflect the textual semantics of code, which are also commonly used in other works [13], [15]. The last element is the structural feature we extracted from AST according to code statements, which reflects the statement-level structural knowledge of code. Fig. 2 shows the process of extracting the sequence of statement trees. First, we build an AST and mark the code statement as the corresponding statement tree. The root node of the AST subtree in the figure is marked in red. For example, "S1" of the code in Fig. 2 represents the statement tree with "*MethodDeclaration*" as the root node.

Then, these statement trees are spliced into a sequence "StaTree: [SubTree1][SubTree2][SubTree3][SubTree4]" and treated as four words in the sequence for subsequent word embedding. Note that we also considered other levels of embedded granularities, such as token-level and sequence-level, and we will discuss these experiments in Section V.

Consider the four-tuple of code snippet, where $MethodName = (m_1, m_2, ..., m_p)$ is the method name expressed as a sequence of p tokens split according to Camel-Case, $Api = (a_1, a_2, ..., a_q)$ is an API sequence of continuous API method invocations, $Token = (t_1, t_2, ..., t_n)$ is a set of tokens in the code snippet, and $StatementTree = (staTree_1, staTree_2, ..., staTree_s)$

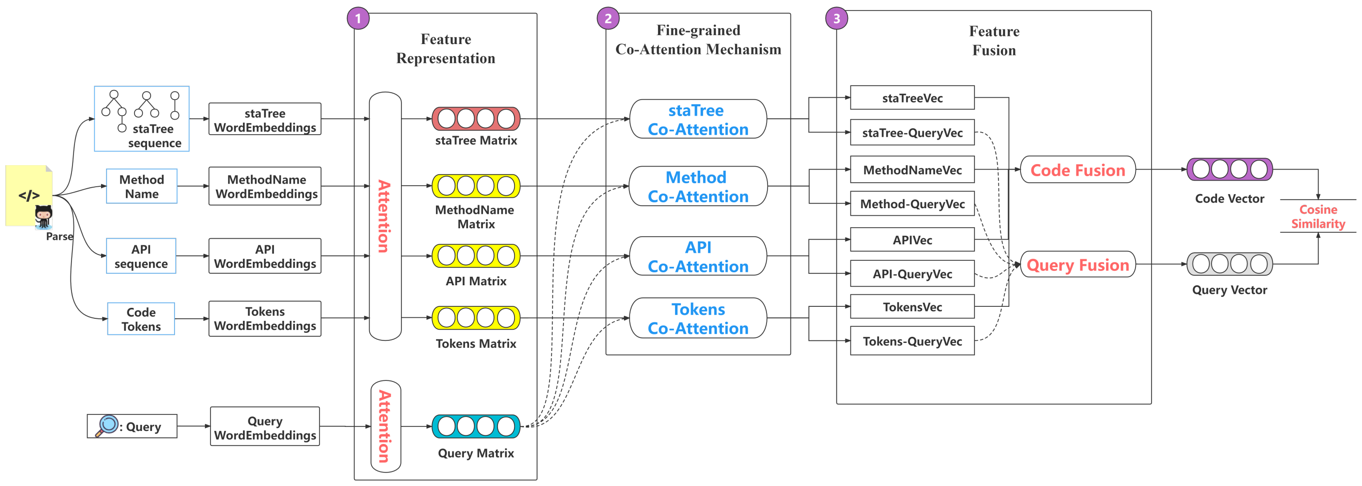


Fig. 3. Overview of FcarCS.

is a sequence of statement trees. For each feature, we convert each word into a vector by constructing a vocabulary and an embedding matrix $E \in R^{g*k}$, where g is the size of the vocabulary and k is the dimension of the word embedding. Thus each feature can be represented as an initial feature matrix composed of a list of word vectors.

Since the attention mechanism can capture the key semantic information in the sequence [21] and faster than the recurrent neural network and the convolutional neural network [16], we perform the attention mechanism on each feature to extract the corresponding embedded representation. Take the method name for example, and suppose $m_i \in R^k$ is a k-dimensional initial word vector corresponding to the i-th word in the method name, the attention weight α_{m_i} of each m_i is calculated as follows:

$$\alpha_{m_i} = SoftMax\left(tanh\left(Wm_i + b\right)\right) \tag{5}$$

where W is a trainable parameter matrix, and b is a trainable bias.

Then, we concatenate the weighted word vectors as the feature matrix of the method name. The calculation formula is as follows:

$$M = \alpha_{m_1} m_1 \bigoplus \alpha_{m_2} m_2 \bigoplus ... \bigoplus \alpha_{m_p} m_p \tag{6}$$

where $M \in R^{k*p}$ is the feature matrix of the method name and $\bigoplus$ is the concatenation operator.

Since the other three features are similar to the method name and are all serialized features, we also perform the same approach to extract their feature matrices. So that we can obtain API feature matrix $A \in R^{k*q}$, tokens feature matrix $T \in R^{k*n}$, and statement tree feature matrix $staTree \in R^{k*s}$.

Query features representation. Query describes a code function and is usually composed of keywords containing the developer's intention. Similar to the tokens sequence, we construct a vocabulary and embedding matrix to convert the query's words into vectors of the same dimension. Then, we use an attention mechanism to extract the semantic information of the query.

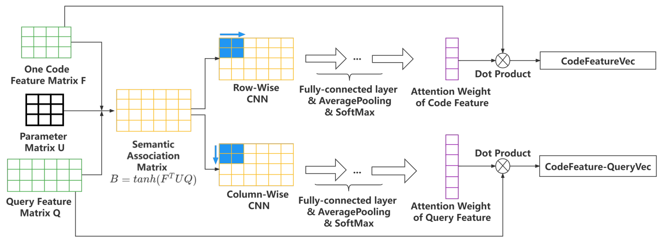
Suppose query is $Q = \{q_1, q_2, ..., q_o\}$, where $q_i \in R^k$ is the k-dimensional initial word vector corresponding to the i-th word in the query. The feature extraction process is as follows:

$$\alpha_{q_i} = SoftMax\left(tanh\left(Wq_i + b\right)\right) \tag{7}$$

$$Q = \alpha_{q_1} q_1 \bigoplus \alpha_{q_2} q_2 \bigoplus ... \bigoplus \alpha_{q_o} q_o \tag{8}$$

$Q \in R^{k*o}$ is the final query feature matrix.

B. *Fine-grained Co-Attention Mechanism*



One Code Feature Matrix F
Parameter Matrix U
Query Feature Matrix Q
Semantic Association Matrix
$B = tanh(F^TUQ)$
Row-Wise CNN
Column-Wise CNN
Fully-connected layer & AveragePooling & SoftMax
Fully-connected layer & AveragePooling & SoftMax
Attention Weight of Code Feature
Attention Weight of Query Feature
Dot Product
Dot Product
CodeFeatureVec
CodeFeature-QueryVec

Fig. 4. Workflow of the fine-grained co-attention mechanism

In the feature representation phase, we get the code feature matrix M, A, T, $staTree$ and query feature matrix Q. Afterwards, we propose a **fine-grained co-attention mechanism** for comparing and aligning each code feature embedding against the query embedding. The key idea behind this mechanism is to explore more fine-grained semantic correlations between each code feature and query, and enrich their representations, respectively. In this way, the model clearly identifies different semantics between code features and query, and avoids semantic confusion caused by feature fusion. Fig. 4 presents the workflow of the fine-grained co-attention mechanism, and the detailed steps are divided into three parts: computing semantic association, extracting semantic information, and calculating semantic vector.

Computing Semantic Association. In order to enrich the representations of code feature and query feature, we first calculate the mutually associated semantic matrix between each code feature and query. Taking the method name matrix M and query matrix Q as examples, we calculate the semantic association matrix $B \in R^{p*o}$, where p is the number of method name tokens, and o is the number of query words. The calculation formula is as follows:

$$B = tanh\left(M^TUQ\right) \tag{9}$$

where $U \in R^{k*k}$ is a trainable parameter matrix, the semantic association matrix B represents the correlation between the words in the method name and the words in the query. We use the $tanh$ activation function to constrain the value of each element in the matrix to be between -1 and 1.

The correlation between words is reflected on each element $b_{i,j}$ in matrix B, which represents the correlation between the i-th code word and the j-th query word. Specifically, row i in the B matrix represents the correlation between each query word and the i-th code word. Similarly, column j in the B matrix represents the correlation between each code word and the j-th query word.

Extracting Semantic Information. CNN can perceive the matrix's local information and is good at extracting abstract features [33]. So we extract the method name feature matrix B^M related with the query feature matrix Q, and the query feature matrix B^Q related with the method name feature matrix M through CNN $f(\cdot)$ along with the row and column directions of matrix$B$, respectively:

$$b_i^M = f\left(W * b_{i:i+h-1} + b\right) \tag{10}$$

$$B^M = ave-pooling\left(FC\left[b_1^M, b_2^M, ..., b_{p-h+1}^M\right]\right) \tag{11}$$

$$b_i^Q = f\left(W * b_{i+h-1:i} + b\right) \tag{12}$$

$$B^Q = ave-pooling\left(FC\left[b_1^Q, b_2^Q, ..., b_{o-h+1}^Q\right]\right) \tag{13}$$

where $W \in R^{k*h}$ is the filter, the filter window size h is set to 2, and $b_{i,j}$ is the element in the semantic association matrix B. FC denotes a fully connected layer, which aggregates local information b_i^M and b_i^Q obtained by convolution. Since the max-pooling strategy may ignore some important features, we use the average pooling strategy $ave-pooling$ to extract more information after aggregating information at the full connection layer.

Calculating Semantic Vector. After obtaining the semantic feature matrix B^M and B^Q, we use a softmax function to convert them into attention weights $a^M \in R^p$ and $a^Q \in R^o$ to enrich each corresponding feature representation. The process is as follows:

$$a_i^M = \frac{exp\left(B_i^M\right)}{\sum_{j=1}^{p} exp\left(B_j^M\right)} \tag{14}$$

$$a_i^Q = \frac{exp\left(B_i^Q\right)}{\sum_{j=1}^{o} exp\left(B_j^Q\right)} \tag{15}$$

$$a^M = \left[a_1^M, ..., a_p^M\right]^T \tag{16}$$

$$a^Q = \left[a_1^Q, ..., a_o^Q\right]^T \tag{17}$$

Finally, the semantic feature vector is obtained by dot product operation of the feature matrix and attention weight. The calculation process is as follows:

$$V_M = M \cdot a^M \tag{18}$$

$$V_{QM} = Q \cdot a^Q \tag{19}$$

V_M represents the method name feature vector related with the query feature matrix, and V_{QM} represents the query feature vector related with the method name feature matrix.

Similar to the method name, after performing the above fine-grained co-attentive operations, we can also obtain API sequence feature vector V_A, tokens feature vector V_T, statement tree sequence feature vector $V_{staTree}$, and corresponding query vectors $\{V_{QA}, V_{QT}, V_{QstaTree}\}$.

C. Feature Fusion

Through the fine-grained co-attention mechanism, we obtain the semantic code feature vectors $Code = \{V_M, V_A, V_T, V_{staTree}\}$ and the semantic query feature vectors $Query = \{V_{QM}, V_{QA}, V_{QT}, V_{QstaTree}\}$. Considering the different importance of feature vectors, we leverage the attention mechanism between feature vectors to learn the correlation weights α_{v_c} and α_{v_q}, where $v_c \in Code$ and $v_q \in Query$. And then, we perform weighted fusion to obtain the final code vector $V_C \in R^{k*c}$ and query vector $V_Q \in R^{k*r}$, where c is the sum of the words of all the code features, and r is four times the number of query words. The calculation process is shown in Eq. (20- 23):

$$\alpha_{v_c} = SoftMax\left(tanh\left(Wv_c + b\right)\right) \tag{20}$$

$$V_C = \sum_{v_c \in Code} v_c \cdot \alpha_{v_c} \tag{21}$$

$$\alpha_{v_q} = SoftMax\left(tanh\left(Wv_q + b\right)\right) \tag{22}$$

$$V_Q = \sum_{v_q \in Query} v_q \cdot \alpha_{v_q} \tag{23}$$

where W is a trainable parameter matrix, b is a trainable bias.

D. Model Optimization

In the experiment, each training instance is a triple $\langle c, q^+, q^- \rangle$, which is constructed as follows: there are relevant query q^+(the correct query of c) and irrelevant query q^-(the incorrect query of c) for each code. Our model expects that when a code and a query have similar semantics, their representation vectors should be close to each other in the vector space. Specifically, our model would predict a higher similarity for $sim(c, q^+)$ over $sim(c, q^-)$. The model is optimized by the margin ranking loss [34], [35] as follows:

$$L(\theta) = \sum_{\langle c,q^+,q^- \rangle \in G} max\left(0, \beta - sim\left(c, q^+\right) + sim\left(c, q^-\right)\right) \tag{24}$$

where θ represents all parameters in the model, β is the margin constraint parameter, which ensures that the distance between the code vector and the correct query vector is closer to at least β than that between the code vector and the wrong query vector. β is set to the default value of 0.05. sim is the cosine similarity between the code vector and the query vector.

We use the Adam algorithm [36] to minimize the loss function. In the training process, the loss function realizes gradient descent through Adam [37], updates the model parameters iteratively, and learns the final code representation vector and query representation vector simultaneously.

E. Implementation Details

The necessary implementation of FcarCS is as follows: the batch size is set to 128, the word embedding size is set to 100, the CNN output dimension in the fine-grained co-attention mechanism is set to 100, and the number of training epoch is set to 400. All experiments were implemented by using the Keras framework and python 3.6. The experiments were conducted on a server (Ubuntu 18.04) with an NVIDIA Titan V GPU and 256GB of memory.

IV. Experiment Setup

This section describes the used dataset, the baseline models, the evaluation metrics, and five investigated research questions (RQs).

A. Dataset

To evaluate the effectiveness of the model, we conduct our experiments on an existing large-scale Java dataset collected by Hu. et al. [28]. This dataset contains over 475k code-query pairs for training and 10k code-query pairs for testing. And the content of this dataset was extracted from Java repositories created on GitHub between 2015 and 2016 with at least 10 stars. We also perform a user study to test the performance of our model on 50 real-world queries collected from Stack Overflow by Gu et al. [13].

B. Baselines

DeepCS[2]. A DL-based code search model was proposed by Gu et al. [13]. It uses LSTM and MLP to embed code snippets and queries into a shared vector space and calculates the similarity between the vectors.

UNIF. A simpler code search model was proposed by Cambronero et al. [19]. It uses a fastText-based neural network and performs an attention mechanism for code representation. The cosine distance is used as the similarity metric. We have reproduced the model according to the original paper in our replication package[3].

TabCS[4]. A state-of-the-art model based on a two-stage attention mechanism was proposed by Yang et al. [16]. The first stage leverages the attention mechanism to extract semantics from code and query, considering their semantic gap. The second stage leverages a co-attention mechanism to capture their semantic correlation and learn better code/query representation.

C. Evaluation Metrics

We used two widely used metrics to evaluate the effectiveness of the proposed model: mean reciprocal rank (MRR) and SuccessRate at k (SR@k).

MRR, the average of the reciprocal ranks of all queries, is computed as follow:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q} \tag{25}$$

[2]https://github.com/guxd/deep-code-search

[3]https://github.com/cqu-isse/FcarCS

[4]https://github.com/cqu-isse/TabCS

where $FRank_q$ refers to the rank position of the first result for the q-th query, $|Q|$ is the number of queries in Q. Since developers prefer to check the first few code snippets in the list to find the expected code methods, we only test MRR on the top-10 ranked list following Gu et al. [13].

SR@k, the proportion of queries that relevant code methods can be found in the top-k ranked results. Following Gu et al. [13], we evaluate SR with k at 1, 5, 10, respectively.

$$SR@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta\left(FRank_q \leq k\right)) \tag{26}$$

δ returns 1 if the q-th query could be found in the top-k returned results and returns 0 otherwise.

D. Research Questions

To investigate the effectiveness of the proposed approach, we studied the following five research questions (RQs):

RQ1. How effective is FcarCS compared with the state-of-the-art models?

To verify the validity of the proposed model, the first question investigates whether the FcarCS outperforms the state-of-the-art models, including DeepCS, UNIF, and TabCS, on Hu. et al.'s datasets [28].

RQ2. How do four code features affect the model performance?

In the FcarCS model, a code snippet is represented by four features (method name, API sequence, tokens, and statement tree sequence) that capture textual and structural information of the source code. To analyze the impact of each feature on FcarCS effectiveness, we performed feature ablation experiments and investigated whether using all four features together was the best choice.

RQ3. How do different co-attention settings affect the model effectiveness?

To evaluate the effectiveness of the fine-grained co-attention mechanism between each code feature and query, we constructed the following two model variants: FcarCS-No and FcarCS-$Fusion$, and compared them with FcarCS.

- FcarCS-No: This variant removes the fine-grained co-attention mechanism.
- FcarCS-$Fusion$: This variant performs the fine-grained co-attention mechanism between query and code fused all features.
- FcarCS is the proposed model, which performs the fine-grained co-attention mechanism between each code feature and query.

RQ4. How do different word embedding granularities of code structural sequences affect the model performance?

To explore a good trade-off between the embedding granularities and the richness of syntactical information, we carried out the following three different ways of embedding statement tree sequences on FcarCS and TabCS during the word embedding phase.

- Token-level: each token in the statement tree sequence is treated as a word.
- Statement-level: the part of the statement tree sequence corresponding to a code statement is treated as a word.
- Sequence-level: the entire structural sequence is treated as a word.

RQ5. How effective is FcarCS on real-world queries compared with the state-of-the-art models?

To measure the effectiveness of FcarCS in the real world, we identify whether our model returns code snippet more relevant than DeepCS, UNIF, and TabCS on 50 real-world queries collected by Gu. et al. [13].

V. Experiment Results

This section presents the detailed results of the five RQs described in Section IV-D.

A. RQ1: How effective is FcarCS compared with the state-of-the-art models?

We compare our FcarCS model with the state-of-the-art models DeepCS, UNIF, and TabCS. Table I presents the experimental results of FcarCS compared against three baseline models on Hu et al.'s datasets. Result shows that FcarCS achieves an MRR of 0.613, and SR@1/5/10 with 0.628/ 0.770/ 0.830. Our proposed FcarCS outperforms the models DeepCS, UNIF, and TabCS by 117.38%, 16.76%, and 12.68% in terms of MRR; by 125.90%/ 77.42%/ 60.85%, 18.71%/ 12.90%/ 7.51%, and 14.18%/ 7.09%/ 4.80% in terms of SR@1/5/10, respectively.

TABLE I
EFFECTIVENESS COMPARISON OF MODELS DEEPCS, UNIF, TABCS AND FCARCS IN TERMS OF SR@1/5/10 AND MRR.

Model	SR@1	SR@5	SR@10	MRR
DeepCS	0.278	0.434	0.516	0.282
UNIF	0.529	0.682	0.772	0.525
TabCS	0.550	0.719	0.792	0.544
FcarCS	**0.628**	**0.770**	**0.830**	**0.613**

Answer to RQ1: *The proposed model FcarCS outperforms the state-of-the-art baselines DeepCS, UNIF, and TabCS significantly for code search.*

B. RQ2: How do four code features affect the model performance?

To investigate the relative importance of the code's four features (method name, tokens, API sequence, and statement tree sequence), we remove one feature from FcarCS at a time. In Table II, M, A, T, and ST in FcarCS represent method name, API sequence, tokens, and statement tree sequence, respectively. For the TabCS-$staTree$ model, it replaces the structural sequence used in the original paper with the statement tree sequence extracted from our work. And UNIF-$staTree$ means that we have added statement tree sequence as a code structural feature to the model UNIF inputs.

From Table II, we can observe that TabCS-$staTree$ improves TabCS by 6.36%/ 4.17%/ 2.90%, and 5.33% in terms of SR@1/5/10 and MRR. UNIF-$staTree$ improves UNIF by 6.24%/ 5.72%/ 2.07%, and 4.57%. FcarCS outperforms TabCS-$staTree$ and UNIF-$staTree$ by 6.98% and 11.66% in terms of MRR. For FcarCS, by removing the features on method name, API sequence, tokens, and statement tree sequence, the MRR decreases by 3.59%/ 2.77%/ 7.34%/ 3.26%, respectively.

TABLE II
EFFECTIVENESS COMPARISON OF FIVE DIFFERENT FEATURE SETTINGS IN TERMS OF SR@1/5/10 AND MRR.

Model	SR@1	SR@5	SR@10	MRR
UNIF	0.529	0.682	0.772	0.525
UNIF-$staTree$	0.562	0.721	0.788	0.549
TabCS	0.550	0.719	0.792	0.544
TabCS-$staTree$	0.585	0.749	0.815	0.573
FcarCS (A+T+ST)	0.609	0.753	0.814	0.591
FcarCS (M+T+ST)	0.615	0.755	0.814	0.596
FcarCS (M+A+ST)	0.582	0.732	0.794	0.568
FcarCS (M+A+T)	0.609	0.757	0.823	0.593
FcarCS (M+A+T+ST)	**0.628**	**0.770**	**0.830**	**0.613**

The result shows that all four code features can improve the effectiveness of FcarCS. Moreover, we can observe that statement tree sequences can improve the performance of the UNIF, TabCS, and FcarCS. The results confirm that using four code features together for model inputs can achieve the best performance, while code tokens affect the effectiveness of FcarCS most. This result is also consistent with cognition that code tokens contain more words related to query semantics.

Answer to RQ2: *The proposed structural feature is necessary and useful for code search. Meanwhile, all the four code features can contribute to the effectiveness of FcarCS.*

C. RQ3: How do different co-attention settings affect the model effectiveness?

Table III shows the effectiveness comparison of the following variants using the same features: TabCS-$staTree$ is a representative of co-attention mechanism; FcarCS-No concatenates all code feature matrices directly to obtain code matrix after attention mechanism, and then the average-pooling is performed for code/query matrix to get code/query vector, respectively; FcarCS-$Fusion$ uses the fine-grained co-attention mechanism to get code/query vector directly after fusing all the code feature matrices; FcarCS is our proposed model.

The core idea of models TabCS-$staTree$ and FcarCS-$Fusion$ is to fuse all code feature embeds firstly and then calculate the co-attentive representations with query. However, as shown in Table III, we can see that FcarCS-$Fusion$

outperforms TabCS-$staTree$ in terms of MRR and SR. This result indicates that the fine-grained co-attention mechanism is more effective.

Moreover, FcarCS outperforms TabCS-$staTree$, FcarCS-No and FcarCS-$Fusion$ by 6.98%, 5.51% and 1.32% in terms of MRR. This result implies that capturing semantic correlations between code features and queries to enrich representations can improve performance and the fine-grained co-attention mechanism contributes to the effectiveness of FcarCS.

TABLE III
EFFECTIVENESS COMPARISON OF THREE DIFFERENT CO-ATTENTION SETTINGS IN TERMS OF SR@1/5/10 AND MRR.

Model	SR@1	SR@5	SR@10	MRR
TabCS-$staTree$	0.585	0.749	0.815	0.573
FcarCS-No	0.594	0.745	0.811	0.581
FcarCS-$Fusion$	0.617	0.765	0.823	0.605
FcarCS	**0.628**	**0.770**	**0.830**	**0.613**

Answer to RQ3: *The proposed fine-grained co-attention mechanism is the best setting for code search, compared with the other representative co-attention settings.*

D. RQ4: How do different word embedding granularities of code structural sequences affect the model performance?

For a statement tree sequence, there are different ways of embedding the sequence. In addition to statement-level embedding granularity, we also consider two other granularities: token-level and sequence-level. The token-level treats each token of a sequence as a word, and the sequence-level treats the entire sequence as a word. We compare token-level granularity, statement-level granularity, and sequence-level granularity on TabCS and FcarCS, respectively.

From Table IV, we can observe that statement-level granularity is the best choice for the embedding of statement tree sequence on both TabCS and FcarCS. For these two models, the statement-level granularity shows better performance, achieving the MRR of 0.573 and 0.613, which outperforms token-level and sequence-level. The results demonstrate that statement-level representation is a good trade-off between the token-level and sequence-level representation.

TABLE IV
EFFECTIVENESS COMPARISON OF DIFFERENT WAYS OF EMBEDDING CODE STRUCTURAL SEQUENCES IN TERMS OF SR@1/5/10 AND MRR.

Model (TabCS)	SR@1	SR@5	SR@10	MRR
Token-level	0.584	0.746	0.815	0.568
Statement-level	**0.585**	**0.749**	**0.815**	**0.573**
Sequence-level	0.577	0.741	0.813	0.566
Model (FcarCS)	SR@1	SR@5	SR@10	MRR
Token-level	0.618	0.762	0.820	0.599
Statement-level	**0.628**	**0.770**	**0.830**	**0.613**
Sequence-level	0.621	0.767	0.828	0.600

TABLE V
BENCHMARK QUERIES AND EVALUATION RESULTS (NF: NOT FOUND WITHIN THE TOP 10 RETURNED RESULTS D:DEEPCS U:UNIF T:TABCS F:FCARCS)

No.	Question ID	Query	D	U	T	F
1	309424	convert an inputstream to a string	NF	2	1	2
2	157944	create arraylist from array	NF	4	5	1
3	1066589	iterate through a hashmap	4	1	NF	8
4	363681	generating random integers in a specific range	4	NF	NF	1
5	5585779	converting string to int in java	1	NF	1	3
6	1005073	initialization of an array in one line	NF	NF	3	5
7	1128723	how can I test if an array contains a certain value	NF	NF	NF	2
8	604424	lookup enum by string value	2	7	NF	9
9	886955	breaking out of nested loops in java	NF	NF	NF	NF
10	1200621	how to declare an array	NF	3	2	6
11	41107	how to generate a random alpha-numeric string	5	2	5	2
12	409784	what is the simplest way to print a java array	3	4	9	5
13	109383	sort a map by values	NF	NF	NF	NF
14	295579	fastest way to determine if an integer's square root is an integer	NF	NF	NF	NF
15	80476	how can I concatenate two arrays in java	NF	7	1	7
16	326369	how do I create a java string from the contents of a file	5	1	1	1
17	1149703	how can I convert a stack trace to a string	1	1	1	1
18	513832	how do I compare strings in java	7	3	3	1
19	3481828	how to split a string in java	NF	1	5	2
20	2885173	how to create a file and write to a file in java	1	1	1	1
21	507602	how can I initialise a static map	NF	NF	NF	2
22	223918	iterating through a collection, avoiding concurrentmodification -exception when removing in loop	NF	NF	NF	NF
23	415953	how can I generate an md5 hash	NF	1	NF	1
24	1069066	get current stack trace in java	1	2	5	1
25	2784514	sort arraylist of custom objects by property	NF	NF	NF	NF
26	153724	how to round a number to n decimal places in java	NF	NF	NF	NF
27	473282	how can I pad an integers with zeros on the left	NF	NF	NF	NF
28	529085	how to create a generic array in java	NF	4	NF	1
29	4716503	reading a plain text file in java	1	1	1	1
30	1104975	a for loop to iterate over enum in java	1	10	5	7
31	3076078	check if at least two out of three booleans are true	NF	NF	NF	NF
32	4105331	how do I convert from int to string	NF	2	7	2
33	8172420	how to convert a char to a string in java	NF	NF	NF	NF
34	1816673	how do I check if a file exists in java	NF	NF	NF	NF
35	4216745	java string to date conversion	1	10	5	2
36	1264709	convert inputstream to byte array in java	NF	1	1	1
37	1102891	how to check if a string is numeric in java	1	1	1	1
38	869033	how do I copy an object in java	NF	NF	NF	NF
39	180158	how do I time a method's execution in java	NF	NF	10	1
40	5868369	how to read a large text file line by line using java	NF	8	7	1
41	858572	how to make a new list in java	NF	NF	2	1
42	1625234	how to append text to an existing file in java	7	9	NF	NF
43	2201925	converting iso 8601-compliant string to date	3	8	6	2
44	122105	what is the best way to filter a java collection	NF	5	NF	NF
45	5455794	removing whitespace from strings in java	NF	NF	NF	NF
46	225337	how do I split a string with any whitespace chars as delimiters	NF	2	2	1
47	52353	in java, what is the best way to determine the size of an object	NF	4	8	7
48	160970	how do I invoke a java method when given the method name as a string	NF	1	1	1
49	207947	how do I get a platform dependent new line character	NF	NF	NF	NF
50	1026723	how to convert a map to list in java	6	9	1	1

Answer to RQ4: *Statement-level granularity is the best choice for embedding structural sequence in feature representation to code search effectiveness.*

E. RQ5: How effective is FcarCS on real-world queries compared with the state-of-the-art models?

Table V shows the user study of FcarCS and baselines on 50 real-world queries. The column Question ID shows the original ID of the question in Stack Overflow where the query comes from. The last four columns show the FRank result of each model, which refers to the rank position of the first relevant result. Following Gu et al. [13], the relevancy is manually identified by two independent developers, and the disagreements are resolved by open discussions. The agreement of the manual assessment was measured using pairwise inter-rater reliability with Cohen's Kappa statistic [38]. The agreement rate in the pilot study was "substantial" (0.83). The symbol 'NF' means that Not Found relevant result within the top K results (K=10), and we conservatively treat the FRank as 11 for queries signed 'NF'. We observe that FcarCS achieves more relevant results with an average FRank of 5.12, reducing the average FRank of DeepCS (8.12), UNIF (6.70), and TabCS (6.84) by 36.9%, 23.6%, and 25.1%, respectively. The results show that FcarCS generally returns code more relevant than DeepCS, UNIF, and TabCS.

Furthermore, to analyze the statistical difference between FcarCS and these baselines, we apply the Wilcoxon signed-rank test [29] on FRank between them at a 5% significance level. The p-value is less than 0.01, indicating the improvements of FcarCS over these baselines are substantial in statistical significance.

> **Answer to RQ5:** *For real-world queries, FcarCS outperforms the state-of-the-art models substantially in statistical significance.*

VI. Discussion

This section first discusses the advantages of the proposed model FcarCS in Section VI-A. Then, we discuss the threats to validity in Section VI-B.

A. Why Does FcarCS Work Well?

Although TabCS considers both textual and structural features in code representation [16], it may cause semantic loss by treating a set of token-level AST's words as code structure described in Section I. As for FcarCS, it breaks the structural semantics into a set of statement-level subtrees of AST corresponding to code statements and converts them into a sequence, which allows the model to preserve sequential semantics between words in code statements while capturing structural semantics. The result in Table II proves that the statement-level structure can significantly improve the performance of both TabCS and FcarCS.

Moreover, FcarCS learns the semantic correlations between query and each code feature respectively through a fine-grained co-attention mechanism, which is higher conducive to learning interdependent code/query representation than performing one co-attention process for the fused feature of code in TabCS. Fig. 5 and Fig. 6 show the first result of TabCS vs. our FcarCS for query "how to read text file line by line". From Fig. 5, we can observe that TabCS returned an irrelevant code, although method name and API contain the query keywords (e.g., "read" and "line"). Because TabCS fuses all code features to compute co-attentive representation with query, in which a lot of irrelevant code words (e.g., "InputStreamReader", "getLineNumber") will confuse semantic relevance between code and query. In contrast, FcarCS can retrieve the expected code snippet in Fig. 6, which involves a lot of query-related keywords, such as "readLines", "fileName", "reader", "readLine" and etc. An important reason is that FcarCS obtains the correlations between each code feature and query respectively through a fine-grained co-attention mechanism. In this way, FcarCS clearly captures the interdependent semantics between code and query, and mitigates semantic confusion.

```
static private String readFromStdin() throws IOException {
  final LineNumberReader r =
      new LineNumberReader(new InputStreamReader(System.in));
  try {
    final StringBuilder sb = new StringBuilder();
    String s;
    while ((s = r.readLine()) != null) {
      if (r.getLineNumber() > NUM_) sb.append(STR_);
      sb.append(s);
    }}
}
```

Fig. 5. TabCS's first retrieved result for "how to read text file line by line".

```
public static String[] readLines(String fileName) {
    List<String> lines = new ArrayList<String>();
    if (reader == null) createReader(fileName);
    String line = null;
    try {
      while ((line = reader.readLine()) != null) {
        if (line.trim().length() > NUM_) {
          lines.add(line.trim());
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    closeReader();
    return lines.toArray(new String[lines.size()]);
  }
```

Fig. 6. FcarCS's first retrieved result for "how to read text file line by line".

B. Threats To Validity

There are many threats to validity for the proposed model FcarCS. First, the model works well on the Java dataset, which does not mean that it will perform well in other programming language datasets (e.g., c#, python). We plan to extend the types of our dataset on the same model. Additionally, the relevancy of returned code methods to the 50 real-world queries was manually identified by two independent developers, which could suffer from subjectivity bias. In the future, we will mitigate this threat by inviting more developers. In addition, many parameters used by our model are set to default values, such as the length of code features, the dimension of word embedding, and the size of CNN's convolution kernel. Thus, such a setting may not represent the best model performance.

VII. Related Work

This section provides the related works in two aspects: code search and abstract syntax tree.

A. Code Search

Over the years, many IR-based code search methods have been proposed [4], [8]–[12]. Krugle [39] and Koders [40] are early code search engines that return code snippets containing the keywords specified in the query. Later, many researchers put forward a lot of work focusing on query extension in order to understand better natural language queries [41]–[43]. For example, Lu et al. [43] extended a query with synonyms generated from WordNet [44]. However, source code is more than just plain text, and it contains a wealth of programming knowledge [45]. In order to further improve search performance, Bajracharya et al. [8] proposed a Lucene-based (traditional text search engine) tool called Sourcerer, which searches code based on the similarity between text attributes and code features. Lv et al. [9] proposed the CodeHow, which uses the extended Boolean model to extract the relations between queries and related APIs.

With the development of deep learning (DL), Gu et al. [13] applied DL techniques to code search model for the first time and named it DeepCS (Deep Code Search). It uses two independent LSTM embedded code snippets and their corresponding queries into vector spaces, so as to search code by comparing the similarity between these vectors. Their experiments show that DeepCS can significantly outperform two representative models, Sourcerer [8] and CodeHow [9]. The following other work proposed many improved DL-based models. Cambronero et al. [19] proposed a simpler model UNIF than DeepCS, which is based on the neural network model fastText [20] looks for the relationship between code and query through the attention mechanism. Shuai et al. [15] improved the performance of code search by computing the co-attentive representation of code-query, and they proposed CARLCS-CNN, a code search model based on convolutional neural network (CNN), to establish the semantic relationship between code and query through the co-attention mechanism. Recently, Yang et al. [16] combined the advantages of attention mechanism and co-attentive representation to build an effective model TabCS with a structural feature AST. As described in Section III, our proposed FcarCS improved the TabCS by introducing a statement-level structure and fine-grained co-attention mechanism to address the semantic loss and semantic confusion in TabCS described in Section I. Experimental results showed that FcarCS substantially outperforms these representative models (i.e., DeepCS, UNIF, TabCS).

B. Abstract Syntax Tree

There are many tasks in software engineering related to source code, such as code search [13]–[16], [46], code summarization [28], [47], [48], code representation learning [49]–[52], etc. These tasks face one challenge while learning the source code, that is, structural feature learning. Abstract syntax tree (AST) is a kind of tree that represents the syntactic structure of source code [53]. It has been widely used in programming language and software engineering tasks [53]–[57]. Zhang et al. [49] proposed an AST-based neural network ASTNN to learn vector representation of source code, which can capture the naturalness of statements and works well in source code classification and code clone detection tasks. Hu et al. [28] proposed the Hybrid-DeepCom model to solve the code summary task, and they used a new structure-based traversal (SBT) method to convert the ASTs into specially formatted sequences which can reflect the context structure of code. This model utilized source code and its SBT structure to generate code summaries. Followed Hybrid-DeepCom, Yang et al. [16] proposed a code search model TabCS combining code textual features and SBT structure feature, and it achieved improvement in code search tasks. Lin et al. [58] noted that it was difficult to training over the entire AST and present the Block-wise Abstract Syntax Tree Splitting method (BASTS), which fully utilizes the rich tree-form syntax structure in ASTs, for improving code summarization. Nighi et al. [50] proposed the InferCode model, a self-supervised learning technique for source code learning of unlabeled data. The model's novelty lies in the training of code representations by predicting subtrees automatically identified from the contexts of ASTs.

These works have a similar spirit that extracting the structural feature by transforming AST into different sequences to better represent code in their tasks. Following this spirit, in this paper, we introduce a statement-level structural sequence and explore the trade-off between different embedding granularities and richness of structural information in Section V-D. The experimental results show that statement-level structure is the best choice for our model.

VIII. Conclusion and Future Work

In this paper, we propose FcarCS, a fine-grained co-attentive representation learning model for semantic code search, which not only extracts code textual features, but also extracts structural information by introducing a statement-level code structure. Instead of learning isolated representations, FcarCS constructs a fine-grained co-attention mechanism to learn semantic correlations of each code feature and query, which can directly enhance their respective representations. Based on these semantic feature representations, FcarCS can learn interdependent code/query vectors better. An evaluation of the latest open-source dataset shows that FcarCS outperforms the state-of-the-art models (i.e., DeepCS, UNIF, and TabCS) by 117.38%, 16.76%, and 12.68% in terms of MRR, respectively. Moreover, we perform a user study on 50 real-world queries, and the result shows that FcarCS significantly reduces the average FRank of DeepCS, UNIF, and TabCS by 36.9%, 23.6%, and 25.1%, respectively. Therefore, the statement-level structure and fine-grained co-attention mechanism used by the FcarCS facilitate searching for code snippets that match the developer's intent. In the future, we plan to verify the validity of our model in multiple programming languages and more datasets.

Acknowledgment

This work was supported in part by the General Program of Chongqing Natural Science Foundation(cstc2021jcyj-msxmX0309), the National Key Research and Development Project (No.2018YFB2101200), the Chongqing Science and Technology Plan Projects (No. cstc2018jszx-cyztzxX0037), Key Project of Technology Innovation and Application Development of Chongqing (No. cstc2019jscx-mbdxX0020), the National Natural Science Foundation of China (No. 62102054), and the Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China (CPSDSC202004).

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, 2009.

[2] K. Kevic and T. Fritz, "Automatic search term identification for change tasks," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 468–471.

[3] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[4] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[5] S. P. Reiss, "Semantics-based code search," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 243–253.

[6] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou, "Expanding queries for code search using semantically related api class-names," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1070–1082, 2017.

[7] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, "Deep graph matching and searching for semantic code retrieval," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 5, pp. 1–21, 2021.

[8] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 681–682.

[9] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[10] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[11] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *International Conference on Software Engineering*, 2006.

[12] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[13] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[14] W. Li, H. Qin, S. Yan, B. Shen, and Y. Chen, "Learning code-query interaction for enhancing code searches," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 115–126.

[15] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 196–207.

[16] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "Two-stage attention-based model for code search with textual and structural features," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 342–353.

[17] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[18] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[19] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[20] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[23] J. Yu, Y. Lu, W. Zhang, Z. Qin, Y. Liu, and Y. Hu, "Learning cross-modal correlations by exploring inter-word semantics and stacked co-attention," *Pattern Recognition Letters*, vol. 130, pp. 189–198, 2020.

[24] D.-K. Nguyen and T. Okatani, "Improved fusion of visual and language representations by dense symmetric co-attention for visual question answering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6087–6096.

[25] Y. Tay, A. T. Luu, and S. C. Hui, "Multi-pointer co-attention networks for recommendation," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2309–2318.

[26] L. Li, R. Dong, and L. Chen, "Context-aware co-attention neural network for service recommendations," in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019, pp. 201–208.

[27] B. Li, Z. Sun, Q. Li, Y. Wu, and A. Hu, "Group-wise deep object co-segmentation with co-attention recurrent neural network," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 8519–8528.

[28] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[29] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[30] M. Liu and H. Yin, "Cross attention network for semantic segmentation," in *2019 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2019, pp. 2434–2438.

[31] X. Bai, "Text classification based on lstm and attention," in *2018 Thirteenth International Conference on Digital Information Management (ICDIM)*. IEEE, 2018, pp. 29–32.

[32] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*. PMLR, 2015, pp. 2048–2057.

[33] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of cnn and rnn for natural language processing," *arXiv preprint arXiv:1702.01923*, 2017.

[34] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.

[35] A. Frome, G. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov, "Devise: A deep visual-semantic embedding model," 2013.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approximation*, vol. 26, no. 2, pp. 289–315, 2007.

[38] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[39] S. E. Sim and R. E. Gallardo-Valencia, *Finding source code on the web for remix and reuse*. Springer, 2013.

[40] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4, pp. 424–466, 2012.

[41] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 842–851.

[42] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 524–527.

[43] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[44] C. Leacock and M. Chodorow, "Combining local context and wordnet similarity for word sense identification," *WordNet: An electronic lexical database*, vol. 49, no. 2, pp. 265–283, 1998.

[45] C. Liu, X. Xia, D. Lo, Z. Liu, A. E. Hassan, and S. Li, "Codematcher: Searching code based on sequential semantics of important query words," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–37, 2021.

[46] S. Fang, Y.-S. Tan, T. Zhang, and Y. Liu, "Self-attention networks for code search," *Information and Software Technology*, vol. 134, p. 106542, 2021.

[47] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[48] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[49] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[50] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[51] Y. Li, "Improving bug detection and fixing via code representation learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 137–139.

[52] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[53] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1–5.

[54] Y. Zhang, X. Gao, C. Bian, D. Ma, and B. Cui, "Homologous detection based on text, token and abstract syntax tree comparison," in *2010 IEEE International Conference on Information Theory and Information Security*. IEEE, 2010, pp. 70–75.

[55] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 95–104.

[56] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[57] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 314–325.

[58] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," *arXiv preprint arXiv:2103.07845*, 2021.