2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00039

Two-Stage Attention-Based Model for Code Search with Textual and Structural Features

Ling Xu[1,2], Huanhuan Yang[1,2], Chao Liu[3], Jianhang Shuai[1,2], Meng Yan[1,2,*], Yan Lei[1,2], Zhou Xu[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
[3]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
Email:{xuling, yanghh, shuaijianhang, mengy, yanlei, zhouxu11x}@cqu.edu.cn, liuchaoo@zju.edu.cn

***Abstract*—Searching and reusing existing code from a large scale codebase can largely improve developers' programming efficiency. To support code reuse, early code search models leverage information retrieval (IR) techniques to index a large-scale code corpus and return relevant code according to developers' search query. However, IR-based models fail to capture the semantics in code and query. To tackle this issue, developers applied deep learning (DL) techniques to code search models. However, these models either are too complex to determine an effective method efficiently or learning for semantic correlation between code and query inadequately.**

To bridge the semantic gap between code and query effectively and efficiently, we propose a code search model TabCS (Two-stage Attention-Based model for Code Search) in this study. TabCS extracts code and query information from the code textual features (i.e., method name, API sequence, and tokens), the code structural feature (i.e., abstract syntax tree), and the query feature (i.e., tokens). TabCS performs a two-stage attention network structure. The first stage leverages attention mechanisms 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. We evaluate the performance of TabCS on two existing large-scale datasets with 485k and 542k code snippets, respectively. Experimental results show that TabCS achieves an MRR of 0.57 on Hu et al.'s dataset, outperforming three state-of-the-art models CARLCS-CNN, DeepCS, and UNIF by 18%, 70%, 12%, respectively. Meanwhile, TabCS gains an MRR of 0.54 on Husain et al.'s, outperforming CARLCS-CNN, DeepCS, and UNIF by 32%, 76%, 29%, respectively.

***Index Terms*—code search, attention mechanism, representation learning, code structural feature**

I. Introduction

Open source communities, such as GitHub and SourceForge, present millions of source code in public. Searching and reusing existing code from existing large-scale codebase can substantially help developers improve their software development efficiency. To support the code search task, early code search models leverage information retrieval (IR) technique to return a list of code snippets that match the intention of a search query [1]–[9]. CodeHow [10] is a state-of-the-art IR-based model that indexes a large-scale codebase by Lucene, a text search engine [11], and searches a set of code from codebase according to a search query. To improve the search effectiveness, CodeHow extends the query with related APIs and re-ranks the searched code by an extended Boolean model [12].

However, IR-based models are difficult to match the query intention to the semantics of code [13]. To better bridge the semantic gap between query (natural language) and code (programming language), Gu et al. [13] proposed a deep learning (DL) based model DeepCS. It embeds query and code into representative vectors by using the LSTM (long and short-term memory) model [14]. The code search can be performed by measuring the cosine similarity between the vectors of query and code. Shuai et al. [15] proposed a model CARLCS-CNN that leverages convolutional neural network (CNN) and co-attention mechanism to learn the correlation between code and query. However, these two models still suffer from two limitations: (1) The networks are complex. DeepCS and CARLCS-CNN are both based on deep and complex neural networks, and training them costs enormous computation resources. (2) The structural features, such as abstract syntax trees (ASTs) of code, are often ignored. These models capture the code textural features, including code tokens, method name and API sequence, but not capturing the rich structural semantics of source code.

Recently, Cambronero et al. [16] proposed a simple model UNIF that leverages an attention layer for code/query embedding. Experimental results show that UNIF outperforms DeepCS which has more complex network designs. However, similar to DeepCS, UNIF did not consider the correlation of query and code and the structural features of code. Motivated by these observations, we explore the idea of combining the advantages of the UNIF and CARLCS-CNN to build an effective and efficient code search model with textural and structural features.

```
\\ Query: concatenate two arrays
\\ Code:
public static int[] concatenate (int[] v1, int[] v2)
{
   int[] values = new int[v1.length + v2.length];
   System.arraycopy(v1, 0, values, 0, v1.length);
   System.arraycopy(v2, 0, values, v1.length, v2.length);
   return values;
}
```

Fig. 1. Example of code search with a natural language query.

*Corresponding author.

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00039

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

In this paper, we propose a code search model TabCS (Two-stage Attention-Based model for Code Search). The model aims to learn the semantic relationship between query and code. To capture the semantic correlation in query/code, we build a two-stage attention network structure. The first stage leverages attention mechanism to assign higher weights on words that represents the code functionality and the intention of the developer. As illustrated in Fig. 1, the model would assign less weights on words that frequently appeared in code and query (e.g., "new", and "return"). Besides, to further align the semantics of words in code and query, the second stage leverages the co-attention mechanism to assign greater weights on the words that have semantic correlations in query and code. For example, the query words ("concatenate" and "arrays") are strongly correlated with the words ("concatenate" and "arraycopy") in code.

To verify the model validity, we evaluated TabCS on two existing large-scale datasets, Hu et al.'s dataset [17] with 485k Java methods and Husain et al.'s dataset [18] with 542k Java methods. We compared TabCS with three state-of-the-art DL-based models CARLCS-CNN [15], DeepCS [13], and UNIF [16]. Experimental results show that TabCS achieves an MRR (Mean Reciprocal Rank, a widely used performance metric for code search) of 0.571 on Hu et al.'s dataset, outperforming CARLCS-CNN, DeepCS, and UNIF by 17.73%, 70.45%, and 10.66% respectively. Meanwhile, TabCS gains an MRR of 0.497 on Husain et al.'s dataset [18], outperforming CARLCS-CNN, DeepCS, and UNIF by 31.78%, 75.57%, and 28.64% respectively.

The main contributions of this study are:

- Proposing an attention-based code search model TabCS, which performs a two-stage attention network structure on both textural and structural features of code.
- Evaluating the effectiveness of TabCS on two existing large-scale datasets, where TabCS shows substantial advantages over the state-of-the-art models CARLCS-CNN, DeepCS, and UNIF.
- We open source our replication package[1], including the dataset and the source code for follow-up study.

The remainder of this paper is organized as follows. Section II introduces the background of code search. Section III presents our proposed model TabCS. Section IV describes the experiment setup. Section V and Section VI show the experimental results and discussion respectively. Section VII presents the related works, and Section VIII concludes the work and presents future works.

II. Background

This section briefly describes the background of code search task. Subsection II-A and II-B present code and query embedding, respectively. Subsection II-C describes the deep learning technique in code search.

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

A. Code Embedding

For the state-of-art deep learning models, like DeepCS and CARLCS-CNN, three features are extracted from a method: method name (a list of camel split tokens), API sequence (a list of API words in method body), and tokens (a bag of words in method body).

Firstly, the words in these features are encoded by the ranking of occurrence frequency in the context. Then they are transformed into vectors of the same dimension. Next, a feature containing several words is represented by a matrix (i.e., consisting of a list of word vectors). Finally, each matrix is processed by a neural network model for embedding. In DeepCS, as Eq.(1), method name and API sequence are embedded by an LSTM model to catch the sequential relationship among words. The tokens are embedded by a common multilayer perceptron (MLP) [19]. In CRALCS-CNN, AS Eq.(2), method name and tokens are embedded by a CNN model, and the API sequence is embedded by a LSTM network.

$$\boldsymbol{v}_c = LSTM_1(V_{name}) + LSTM_2(V_{API}) + MLP(V_{tokens}) \quad (1)$$

$$\boldsymbol{v}_c = CNN_1(V_{name}) + CNN_2(V_{tokens}) + LSTM(V_{API}) \quad (2)$$

B. Query Embedding

Similarly, the query in natural language is treated as a bag of tokens. The tokens are encoded and transformed into vectors of the same dimension. Then a query is represented by a matrix. Finally, the matrix is embedded by a neural network. DeepCS performs a LSTM model for embedding as Eq. (3), and CARLCS-CNN performs a CNN model as Eq. (4).

$$\boldsymbol{v}_q = LSTM_3(V_{query}) \quad (3)$$

$$\boldsymbol{v}_q = CNN_3(V_{query}) \quad (4)$$

C. Deep Learning for Code Search

For a query and a candidate code, DL-based models firstly learn a code representative vector v_c and a query representative vector v_q. Then, models compute their cosine similarity. Finally, all the candidate code snippets are ranked from the best to the worst according to their cosine similarity to the query. The cosine similarity is computed as Eq. (5).

$$cos = \frac{\boldsymbol{v}_c \cdot \boldsymbol{v}_q}{||\boldsymbol{v}_c|| \cdot ||\boldsymbol{v}_q||} \quad (5)$$

In this way, models recommends a code list in which the k_{th} code is the k_{th} possibly corresponding code for the query.

III. Proposed Approach

This section presents the overall framework and details of our proposed model TabCS.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

A. Overall Framework

Fig. 2 shows the overall framework of TabCS that implements code search by two stages. The first stage feeds the code features (i.e., method name, API sequence, tokens, and AST) and the query feature (i.e., tokens) into attention mechanisms to obtain code/query feature matrices. Then, the second stage feeds the feature matrices into a co-attention mechanism to obtain code/query representative vectors. Finally, the model recommends a code list according to the cosine similarity of the two representative vectors.

The following subsections present the model details. Specifically, Section III-B and III-C respectively describe the two stages of the proposed two-stage attention network structure. Section III-D describes the cooperation of the two stages. Section III-E describes model optimization, Section III-F describes model prediction for code search, and Section III-G describes model implementation details.

B. The First Stage

1) Attention-Based Code Embedding: We extract four features from code, including three textural features and a structural feature. We implement the code feature embedding according to the following three steps:

Step-1: Code Textural Feature Embedding. We represent a code snippet by three code textural features: *1) method name,* a list of camel split words; *2) API sequence,* a list of API words in method body; *3) tokens,* a set of words in method body. For a textural feature (e.g., tokens), we transform each word into a vector by building a vocabulary and an embedding matrix $E \in \mathbb{R}^{o \times k}$, where o is the size of the vocabulary, k is the dimension of word embedding. The embedding matrix is initialized randomly and learned in the training process. Then the feature can be represented as a matrix (i.e., consisting of a list of word vectors), namely initial feature matrix. To capture the semantics of a code, we perform attention mechanism on three initial feature matrices respectively. The attention mechanism assigns weights on each word, where the words much frequently used in programming language are assigned with less weights. As illustrated in Fig. 2, code words "is", "if", "return", "false", and "true" frequently exist in many code snippets. Therefore, in the training process, the attention mechanism learns that these words have nothing to do with the method's semantics. Compared with these words, the words not frequently used are assigned with greater weights since they are more likely to reflect method's functionality. Finally, we obtain three weighted feature matrices that respectively represent the feature's semantics.

Let $\boldsymbol{m}_i \in \mathbb{R}^k$ be a k-dimensional word initial vector corresponding to the i-th word in a method name. Given a sequence of length n $\{\boldsymbol{m}_1, ..., \boldsymbol{m}_n\}$, the attention weight α_{m_i} for each $\boldsymbol{m}_i$ is computed as follows:

$$\alpha_{m_i} = \frac{exp(\boldsymbol{a}_{m_i} \cdot \boldsymbol{m}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{m_i} \cdot \boldsymbol{m}_i^T)} \tag{6}$$

Where the attention vector $\boldsymbol{a}_{m_i} \in \mathbb{R}^k$ is a k-dimensional vector and optimized during model training. We calculate the attention weight for each initial vector by using the softmax function over the product of the initial vectors and attention vectors.

Then we compute each product of initial vectors and corresponding attention weights, and then concatenate the weighted vectors. The final concatenation $M \in \mathbb{R}^{k \times n}$ is the feature matrix of the method name, which is formulated as Eq. (7), Where $\oplus$ is the concatenation operator.

$$M = \alpha_{m_1}\boldsymbol{m}_1 \oplus \alpha_{m_2}\boldsymbol{m}_2 \oplus ... \oplus \alpha_{m_n}\boldsymbol{m}_n \tag{7}$$

Given an API sequence of length n $\{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$, the embedding process is shown as Eq. (8-9). The final concatenation $P \in \mathbb{R}^{k \times n}$ is the feature matrix of the API sequence.

$$\alpha_{p_i} = \frac{exp(\boldsymbol{a}_{p_i} \cdot \boldsymbol{p}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{p_i} \cdot \boldsymbol{p}_i^T)} \tag{8}$$

$$P = \alpha_{p_1}\boldsymbol{p}_1 \oplus \alpha_{p_2}\boldsymbol{p}_2 \oplus ... \oplus \alpha_{p_n}\boldsymbol{p}_n \tag{9}$$

Given n tokens $\{\boldsymbol{t}_1, ..., \boldsymbol{t}_n\}$, the embedding process is shown as Eq. (10-11). The final concatenation $T \in \mathbb{R}^{k \times n}$ is the feature matrix of the tokens.

$$\alpha_{t_i} = \frac{exp(\boldsymbol{a}_{t_i} \cdot \boldsymbol{t}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{t_i} \cdot \boldsymbol{t}_i^T)} \tag{10}$$

$$T = \alpha_{t_1}\boldsymbol{t}_1 \oplus \alpha_{t_2}\boldsymbol{t}_2 \oplus ... \oplus \alpha_{t_n}\boldsymbol{t}_n \tag{11}$$

Step-2: Code Structural Feature Embedding. We extract AST from the method body as the structural feature. It parses a code snippet into a syntax tree. The nodes in the tree describe the type of corresponding code, such as loop structure, conditional judgment structure, method call and variable declaration. By traversing an AST in breadth-first strategy, we get all the AST nodes. Like the textural features, part of the nodes reflects the method's function, and the rest can not. For instance, in Fig. 2, the node "IfStatement" is frequently used in many code snippets and hardly reflects the method's functionality. But the node "MethodDeclaration" is related to the method's functionality. Therefore, to catch the method's functionality, we perform an attention mechanism on the AST nodes.

Like textural features, we convert nodes into initial vector embeddings by building vocabularies.Then, we perform an attention mechanism and concatenate the weighted vectors into a feature matrix. The matrix extracts the important nodes. Given a node sequence $\{\boldsymbol{ast}_1, ..., \boldsymbol{ast}_n\}$, the embedding process is shown as Eq. (12-13). The final concatenation $AST \in \mathbb{R}^{k \times n}$ is the feature matrix of the AST.

$$\alpha_{ast_i} = \frac{exp(\boldsymbol{a}_{ast_i} \cdot \boldsymbol{ast}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{ast_i} \cdot \boldsymbol{ast}_i^T)} \tag{12}$$

$$AST = \alpha_{ast_1}\boldsymbol{ast}_1 \oplus \alpha_{ast_2}\boldsymbol{ast}_2 \oplus ... \oplus \alpha_{ast_n}\boldsymbol{ast}_n \tag{13}$$

Step-3: Code Features Fusion. After embedding four code features to four matrices, we eventually concatenate them into a matrix C as the final code feature matrix:

$$C \in \mathbb{R}^{k \times p} = M \oplus P \oplus T \oplus AST \tag{14}$$

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

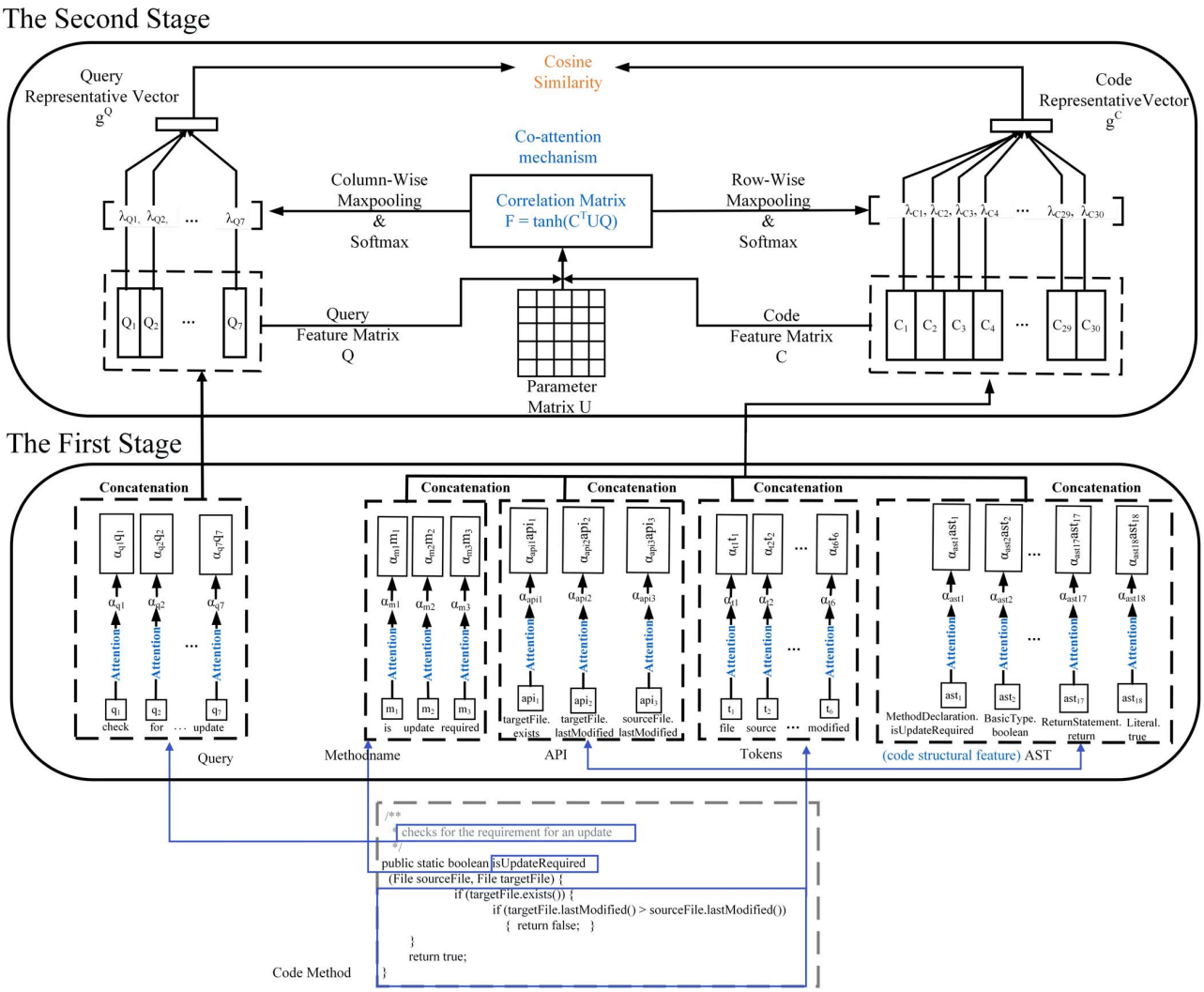

Fig. 2. Overall Framework of TabCS

Were p is the total number of the embedded words in four features. Actually, C is formed by concatenating all the weighted vectors in textural and structural features. Compared with compressing weighted vectors into one vector, the simple concatenation operation can effectively avoid losing original information in the fusion process.

2) Attention-Based Query Embedding: The query always contains informative keywords of developers' intention. We extract tokens from the query. Like code tokens, some words hardly reflect the query's semantics since they are frequently used. After embedding words, to extract query's semantics, we perform an attention mechanism to assign less attention weights on words that frequently appeared in query and assign greater weights on words that represents the intention of the developers. Finally, the weighted word vectors are concatenated into the query feature matrix. Given n tokens $\{\boldsymbol{q}_1, ..., \boldsymbol{q}_n\}$, the embedding process is shown as Eq. (15-16). The concatenation $Q \in \mathbb{R}^{k \times n}$ is the final query feature matrix.

$$\alpha_{q_i} = \frac{exp(\boldsymbol{a}_{q_i} \cdot \boldsymbol{q}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{q_i} \cdot \boldsymbol{q}_i^T)} \tag{15}$$

$$Q = \alpha_{q_1}\boldsymbol{q}_1 \oplus \alpha_{q_2}\boldsymbol{q}_2 \oplus ... \oplus \alpha_n \boldsymbol{q}_n \tag{16}$$

C. The Second Stage

In the first stage, we obtain the code feature matrix $C \in \mathbb{R}^{k \times p}$ and the query feature matrix $Q \in \mathbb{R}^{k \times q}$ (here we replace n in Eq. (16) with q). In the second stage, we feed the two matrices into a co-attention mechanism. Firstly, we compute the correlation matrix $F \in \mathbb{R}^{p \times q}$ as follows:

$$F = tanh(C^T U Q) \tag{17}$$

Where the parameter matrix U $\in \mathbb{R}^{k \times k}$ is to be learned in training process. The correlation matrix F represents semantic correlation between code words and query words. We use the tanh activation function to limit the values of each element of the matrix between -1 and 1.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

Then, we perform max-pooling operations along rows and columns over F as Eq. (18-20).

$$\boldsymbol{w}_i^C = maxpooling(F_{i,1}, ..., F_{i,q}) \tag{18}$$

$$\boldsymbol{w}_j^Q = maxpooling(F_{1,j}, ..., F_{p,j}) \tag{19}$$

$$\boldsymbol{w}^C = \left[\boldsymbol{w}_1^C, ..., \boldsymbol{w}_p^C\right], \quad \boldsymbol{w}^Q = \left[\boldsymbol{w}_1^Q, ..., \boldsymbol{w}_q^Q\right] \tag{20}$$

Where $\boldsymbol{w}^C \in \mathbb{R}^p$ and $\boldsymbol{w}^Q \in \mathbb{R}^q$ represent the correlation between code and query. Afterward, $\boldsymbol{w}^C$ and $\boldsymbol{w}^Q$ are transformed into $\boldsymbol{v}^C$ and $\boldsymbol{v}^Q$ by using the softmax function as Eq. (21-22). We take $\boldsymbol{v}^C \in \mathbb{R}^p$ and $\boldsymbol{v}^Q \in \mathbb{R}^q$ as the weight vectors for code feature matrix C and query feature matrix Q.

$$\boldsymbol{v}_i^C = \frac{exp(\boldsymbol{w}_i^C)}{\sum_{k=1}^{p} exp(\boldsymbol{w}_k^C)}, \quad \boldsymbol{v}_i^Q = \frac{exp(\boldsymbol{w}_i^Q)}{\sum_{k=1}^{q} exp(\boldsymbol{w}_k^Q)} \tag{21}$$

$$\boldsymbol{v}^C = \left[\boldsymbol{v}_1^C, ..., \boldsymbol{v}_p^C\right], \quad \boldsymbol{v}^Q = \left[\boldsymbol{v}_1^Q, ..., \boldsymbol{v}_q^Q\right] \tag{22}$$

Additionally, to obtain the final code representative vector $\boldsymbol{g}^C$ and the final query representative vector $\boldsymbol{g}^Q$, we perform dot product on the feature matrices C, Q, and weight vectors $\boldsymbol{v}^C$, $\boldsymbol{v}^Q$ as Eq. (23).

$$g^C = C\boldsymbol{v}^C, \quad g^Q = Q\boldsymbol{v}^Q \tag{23}$$

D. Cooperation of two stages

Our model bridges the semantics gap between code and query by two stages. In the first stage, to extract code and query's semantics, we use the attention mechanism to assign less weights on words that frequently appeared in code and query. At the same time, assigns higher weights on words that represents code's functionality and the intention of the developer. In the second stage, to learn the semantic correlation of code and query, we use co-attention mechanism to assign higher weights on the code words and query words that contain semantic correlations.

The first stage weights an embedding vector $\boldsymbol{c}_i$ as follow:

$$C_i = \alpha_i \boldsymbol{c}_i \tag{24}$$

Here α_i is the weight assigned to the i_{th} word in the code feature. In the model training, the first stage finds that some words frequently exist in many code snippets which means that these words hardly reflect the method's functionality. Then, to extract semantics from code, the first stage assigns them with less attention weights and assigns greater weights on other words that may reflect the method's functionality, which is the same with query.

Although the code words that are given greater weights by the first stage contains semantics, they may have nothing to do with the query, as do the query words. To address this issue, the second stage performs a co-attention mechanism to learn the semantic correlation of code and query.

The second stage compute their semantics correlation as Eq. (17), a single element $F_{i,j}$ of F in Eq. (17) is computed as follow,

$$F_{i,j} = tanh(C_i^T U Q_j) \tag{25}$$

$C_i \in \mathbb{R}^k$ and $Q_j \in \mathbb{R}^k$ respectively represent the i_{th} word vector in code and j_{th} word vector in query output by the first stage. Element $F_{i,j}$ represents the their semantic correlation. After the max-pooling operation, $\boldsymbol{w}_i^C$ in Eq. (20) represents the correlation of the query word most related to the i_{th} code words. If no words in query is related to the i_{th} code word, $\boldsymbol{w}_i^C$ is much less, it means i_{th} code word is unrelated to the query. Conversely, if $\boldsymbol{w}_i^C$ is much greater, it means i_{th} code word is highly related to the query, which is the same with query. By using $\boldsymbol{w}^C$ and $\boldsymbol{w}^Q$ to weight words as Eq. (18-23), the second stage is able to find the words of code and query that contain semantic correlation. By the cooperation of two weight assignments in two stages, our model learns better code/query representative vectors.

E. Model Optimization

We build five vocabularies respectively for method name, API sequence, AST sequences, code tokens, and query tokens. A vocabulary contains all the words that appear in the feature. In this way, each word in a feature can be uniquely identified by a word vector, and a feature containing a bag of words can be represented by a matrix (i.e., a concatenation of word vectors).

We hope that when a code snippet and a query have similar semantics, their representative vectors should be close to each other. When the semantics of the code and query are different, their representative vectors are far from each other. So in practice, we construct each training instance as a triple $<c, q^+, q^->$: for each code c there is a positive query q^+ (a ground-truth query of c) and a negative query q^- (an incorrect query of c). The incorrect query q^- is selected randomly from the collection of correct query q^+. The loss function are set as follow:

$$L(\theta) = \sum_{(c,q^+,q^-)\in G} max(0, \beta - sim(c, q^+) + sim(c, q^-)) \tag{26}$$

Where sim is the cosine similarity code and query's representative vectors; θ represents all the parameters in the network. β is a small margin constraint and is set to 0.05 according to the default setting. The value of $L(\theta)$ ranges from 0 to β. This loss makes sure that, given a code c, the representative vector of a correct query q^+ is closer to the code's representative vector c than that of an incorrect query q^- by at least a margin β.

We use the Adam algorithm [20] to minimize the loss function. In the training process, in the first stage, the attention mechanism learns the weights of feature words to produce feature matrices. In the second stage, the co-attention mechanism learns the co-attention matrix. This matrix, through column-wise max-pooling and row-wise max-pooling, gets two attention vectors for query and code respectively to weight attention matrices. In the training process, the loss function realizes gradient descent [21] through Adam, the model parameters θ are updated iteratively, and the final representative vectors for query and code are learned simultaneously.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

F. Model Prediction for Code Search

The proposed model performs code search in the following steps: given a query, TabCS first matches all code snippets with the query. For a repository with n code snippets, it generates n query-code pairs. Next, TabCS computes their representative vectors and their cosine similarities for all the query-code pairs. Then, they are ranked according to their cosine similarity values. Finally, TabCS recommends the query-code pairs with top-k values in the list for a search query.

G. Implementation Details

The detailed implementation of the TabCS is as follows: batch size (i.e., the number of instances per batch) is set as 256. The word embedding size is set to 100 following Shuai et al. [22]. All the experiments are implemented using the Keras framework with Python 3.5, and the experiments were conducted on a server (Ubuntu 18.04) with one NVIDIA Titan V GPU and 256 GB memory.

IV. Experiment Setup

This section presents the investigated research questions (RQs), experimental setup, the compared baseline models, and evaluation measures.

A. Research Questions

To verify the validity of the proposed model, this study investigates the following four research questions.

RQ1. Can TabCS outperforms the state-of-the-art models?

The first RQ investigates whether the proposed model TabCS outperforms three state-of-the-art DL-based code search models CARLCS-CNN [15], DeepCS [13] and UNIF [16].

RQ2. Does TabCS run faster than the state-of-the-art models?

RQ2 compares the training and testing time between our TabCS and the baseline models, and tests if the proposed model can save computation resources than the baseline models substantially. Faster models indicate more valuable application in practices.

RQ3. How do textual and structural features affect the model performance?

In TabCS, a code method is respectively represented by four features (i.e., method name, API sequence, tokens and AST), which can capture the structural and semantic information of source code. To analyze their impacts on model effectiveness, we run TabCS by removing one feature at a time, and investigate whether using the four features together is the best choice.

RQ4. How does the two-stage attention network structure improve the model performance?

The two-stage attention network structure of TabCS aims to capture the correlation between code and query in two steps. To analyze its impacts on model effectiveness, we run CARLCS-CNN and UNIF with the textural and structural features of code to investigate whether two-stage attention network structure outperforms the network of CARLCS-CNN and UNIF or not.

B. Datasets

To evaluate the model effectiveness, we tested our model on two existing datasets. One is the Hu's [17] dataset[2] that contains 485,812 code-query pairs. Hu's dataset was collected from GitHub's Java repositories created from 2015 to 2016. To filter out low-quality projects, Hu et al. [17] only considered the projects with more than ten stars. Then, they extracted Java methods and their corresponding Javadoc from these Java projects. The first sentence of the Javadoc is considered as the query. The other is the Husain's [18] dataset[3]. The corpus contains 542,991 Java code with queries written in natural language collected from GitHub repositories.

C. Baselines

DeepCS. One of the state-of-the-art models is the DeepCS proposed by Gu et al. [13]. DeepCS performs RNN and MLP networks on method name. API sequences, and tokens of code to obtain code representative vector, and performs RNN to obtain query representative vector. We re-ran the DeepCS by using the source code shared on the GitHub[4].

CARLCS-CNN. A deep learning-based code search model is CARLCS-CNN proposed by Shuai et al. [22]. It leverages CNN and LSTM associated with a co-attention mechanism to learn interdependent representations for code and query after the individual embedding[5].

UNIF. A state-of-the-art supervised code search model proposed by Cambronero et al. [16]. Specifically, UNIF uses a learned attention-based weighing scheme to combine per-token embeddings and produces the embedded code sentence vector. The description sentence embedding is produced by averaging the bag of query embeddings.

CARLCS-TS. The proposed model incorporates structural code feature (i.e., abstract syntax tree) to CARLCS-CNN. CARLCS-TS takes four code features (i.e., method name, API sequence, tokens and AST sequence) as input. Meanwhile, the added feature AST sequence is embedded by in individual CNN network and merged into the code feature matrix like other feature.

[2]https://github.com/xing-hu/EMSE-DeepCom
[3]https://github.com/github/CodeSearchNet
[4]https://github.com/guxd/deep-code-searh
[5]https://github.com/cqu-isse/CARLCS-CNN

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

UNIF-TS. An extension of the base UNIF of our own creation. UNIF-TS incorporates structural code feature, i.e., abstract syntax tree to UNIF in order to investigate whether structural features can improve code search effectiveness.

D. Evaluation Metrics

To evaluate the model performance, 10k code-query pairs from a dataset were randomly selected as testing while the rest were used as model training. For each query, a model returns the top-10 candidate code and search performance is measured by two widely used metrics SuccessRate and MRR (mean reciprocal rank) following Shuai et al [15]. To suppress the effect of randomness, we evaluate the model ten times with different randomly selected training/testing data at each time.

SuccessRate@k (SR@k), the proportion of queries that the relevant code method could be found in the top-k ranked lists. In specific, SuccessRate@k is calculated as $SR@k = |Q|^{-1} \sum_{i=1}^{|Q|} \sigma\left(Q_i \leq k\right)$, where Q is the 10k queries in our automatic evaluation, as referred to in Section IV-B; σ is an indicator function that returns 1 if the i-th query (Q_i) could be found in the top-k ranked list, otherwise it returns 0. Following Gu et al. [23], we evaluate SR with k at 1, 5, 10 respectively.

MRR, the average of the reciprocal ranks of all queries. The computation process of MRR is $|Q|^{-1} \sum_{i=1}^{|Q|} Rank_{Q_i}^{-1}$, Where Q is the 10k queries in the automatic evaluation; $Rank_{Q_i}$ is the rank of the ground-truth code related to the i-th query (Q_i) in the ranked list. Different from SR, MRR uses the reciprocal rank as the weight of measurement. Meanwhile, as developers prefer to find the expected code method with short code inspection, we only test MRR on the top-10 ranked list following Gu et al. [23]. In other words, when the rank of Q_i is out of 10, then $1/Rank_{Q_i}$ equals to 0.

V. Results

This section investigates the four research questions (RQs) described in Section IV-A respectively.

A. RQ1: Can TabCS Outperform the State-of-the-Art Models?

We compare code search effectiveness between the state-of-the-art models DeepCS, CARLCS-CNN, CARLCS-TS, UNIF and our TabCS model described in Section III. Results show that TabCS outperforms three DL-based models (i.e., DeepCS, CARLCS-CNN, and CARLCS-TS) and UNIF.

For Hu et al.'s dataset, as shown in Table I, TabCS achieves an MRR of 0.571, and SR@1/5/10 with 0.585/0.746/0.813. TabCS outperforms the baseline models DeepCS, CARLCS-CNN, CALRCS-TS, and UNIF by 70.45%, 17.73%, 11.74%, and 10.66% in terms of MRR; by 76.74%/48.31%/37.10%, 18.66%/14.59%/12.14%, 12.50%/10.03%/8.40%, and 10.80%/10.03%/7.82% in terms of SR@1/5/10 respectively.

For Husain et al.'s dataset, as shown in Table II, TabCS achieves an MRR of 0.539, and SR@1/5/10 with 0.547/0.683/0.748. TabCS outperforms the baseline models DeepCS, CARLCS-CNN, CALRCS-TS, and UNIF by 75.57%, 31.78%, 26.53%, and 28.64% in terms of MRR; by 86.05%/55.23%/41.40%, 32.45%/22.62%/18.17%, 24.60%/17.76%/14.02%, and 30.24%/22.84%/19.87% in terms of SR@1/5/10 respectively.

TABLE I
Effectiveness comparison of models DeepCS, CARLCS-CNN, CARLCS-TS, UNIF and TabCS in terms of SR@1/5/10 and MRR on Hu et al.'s dataset.

Model	SR@1	SR@5	SR@10	MRR
DeepCS	0.331	0.503	0.593	0.335
CARLCS-CNN	0.493	0.651	0.725	0.485
CARLCS-TS	0.520	0.678	0.750	0.511
UNIF	0.528	0.678	0.754	0.516
TabCS	**0.585**	**0.746**	**0.813**	**0.571**

TABLE II
Effectiveness comparison of models DeepCS, CARLCS-CNN, CARLCS-TS, UNIF and TabCS in terms of SR@1/5/10 and MRR on Husain et al.'s dataset.

Model	SR@1	SR@5	SR@10	MRR
DeepCS	0.294	0.440	0.529	0.307
CARLCS-CNN	0.413	0.557	0.633	0.409
CARLCS-TS	0.439	0.580	0.656	0.426
UNIF	0.42	0.556	0.624	0.419
TabCS	**0.547**	**0.683**	**0.748**	**0.539**

Furthermore, to analyze the statistical difference between TabCS and these baselines, we apply the Wilcoxon signed-rank test [24] on MRR between them at a 5% significance level. The p-value is less than 0.01, indicating the improvements of TabCS over these baselines are substantial in statistical significance.

> **Result 1:** *The proposed model TabCS outperforms the state-of-the-art baselines DeepCS, CARLCS-CNN, CARLCS-TS, and UNIF substantially on effectiveness.*

B. RQ2. Does TabCS Run Faster Than the State-of-the-Art Models?

We compare our proposed model with baselines on two datasets. All the experiments are implemented on a server with one Nvidia Titan V GPU with 256 GB memory. Table III compares the training and testing time on Hu's dataset. The efficiency comparison is conducted under the same experimental setup. Results show that DeepCS, CARLCS-CNN, CARLCS-TS, UNIF, and TabCS take 36.6, 12.2, 16.6, 2.8, and 5.1 hours for optimization, respectively, and spend about 1.1, 0.4, 0.7, 0.2, and 0.4 seconds for each code search query, respectively. Comparing with DeepCS, CARLCS-CNN, and CARLCS-TS, TabCS is 7 times, 2 times and 3 times faster in model training, respectively. Comparing with DeepCS and CARLCS-TS, TabCS is 3 times and 2 times faster in model testing, respectively. CARLCS-CNN spends the same time with TabCS in model testing. UNIF is slightly faster than TabCS in model optimization and code search. This is because UNIF only use one textual feature (i.e., tokens) to represent the code while TabCS extracts three textual feature and one structural feature.

Table IV shows the training and testing time on Husain et al.'s dataset. Results show that DeepCS, CARLCS-CNN,

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

TABLE III
TIME COST FOR MODEL TRAINING AND TESTING OF DEEPCS, CARLCS-CNN, CARLCS-TS, UNIF AND TABCS ON HU ET AL.'S DATASET.

Model	Training	Testing
DeepCS	36.6 hours	1.1s/query
CARLCS-CNN	12.2 hours	0.4s/query
CARLCS-TS	16.6 hours	0.7s/query
UNIF	2.8 hours	0.2s/query
TabCS	5.1 hours	0.4s/query

TABLE IV
TIME COST FOR MODEL TRAINING AND TESTING OF DEEPCS, CARLCS-CNN, CARLCS-TS, UNIF AND TABCS ON HUSAIN ET AL.'S DATASET.

Model	Training	Testing
DeepCS	34.1 hours	0.9s/query
CARLCS-CNN	10.7 hours	0.4s/query
CARLCS-TS	13.2 hours	0.6s/query
UNIF	1.7 hours	0.2s/query
TabCS	3.9 hours	0.4s/query

CARLCS-TS, UNIF, and TabCS take about 34.1, 10.7, 13.2, 1.7, and 3.9 hours for optimization, respectively, and take about 0.9, 0.4, 0.6, 0.2, and 0.4 seconds for responding each code search query, respectively. Comparing with DeepCS, CARLCS-CNN, and CARLCS-TS, TabCS is 8 times, 2 times and 3times faster in model training, respectively. Comparing with DeepCS and CARLCS-TS, TabCS is 2 times and 1.5 times faster in model testing, respectively. As Hu's dataset, CARLCS-CNN spends the same time with TabCS in model testing, and UNIF is a little bit faster than TabCS in model optimization and model testing.

These results imply that in terms of efficiency, the attention mechanism based search models, especially for our proposed TabCS, is a better choice for practical usage. DeepCS, CARLCS-CNN and CARLCS-TS are slower because they are CNN-based or LSTM-based models with complex network structure and time-consuming optimization process [25]–[28].

Result 2: *The proposed model TabCS outperforms the state-of-the-art baselines DeepCS, CARLCS-CNN, and CARLCS-TS substantially on efficiency.*

C. RQ3. How Do Textual and Structural Features Affect the Model Performance?

To investigate the relative importance of four features (method name, tokens, API sequence, and AST), We remove one feature from our proposed model in turn, and run the model on two datasets. In Table V and Table VI, M, API, T, and AST in TabCS represents the considered code feature method name, API sequence, tokens, and AST sequence, respectively.

From Table V, we can observe that for Hu et al.'s dataset, CARLCS-TS improves CARLCS-CNN by 5.48%/4.15%/3.45%, and 5.36% in terms of SR@1/5/10 and MRR. UNIF-TS improves UNIF by 2.46%/3.39%/2.52%, and 3.29% in terms of SR@1/5/10, and MRR. For TabCS, by removing the features on the method name, API sequence, tokens and AST, the MRR decreases by 10.16%, 9.63%, 15.06% and 8.06% respectively; the SR@1/5/10 decreases by 11.28%/7.37%/5.04%, 10.60%/7.24%/5.17%, 14.87%/ 10.86%/8.12%, and 8.55%/6.17%/4.18% respectively.

Table VI shows the comparison results for Husain et al.'s dataset, CARLCS-TS improves CARLCS-CNN by 6.30%/4.13%/3.63%, and 4.16% in terms of SR@1/5/10, and MRR. UNIF-TS improves UNIF by 8.79%/8.09%/8.01%, and 8.59% in terms of SR@1/5/10, and MRR. For TabCS, by removing the features on the method name, API sequence, tokens and AST, the MRR decreases by 16.51%, 6.49%, 13.91% and 6.49% respectively; the SR@1/5/10 decreases by 17.55%/12.74%/9.49%, 6.03%/3.95%/3.88%, 14.44%/9.81%/7.62% and 7.13%/4.25%/3.21% respectively.

TABLE V
EFFECTIVENESS COMPARISON OF MODELS WITH TEXTUAL AND STRUCTURAL FEATURES SETTINGS IN TERMS OF SR@1/5/10 AND MRR ON HU ET AL.'S DATASET.

Model	SR@1	SR@5	SR@10	MRR
CARLCS-CNN	0.493	0.651	0.725	0.485
CARLCS-TS	**0.520**	**0.678**	**0.750**	**0.511**
UNIF	0.528	0.678	0.754	0.516
UNIF-TS	**0.541**	**0.701**	**0.773**	**0.533**
TabCS (AST+API+T)	0.519	0.691	0.772	0.513
TabCS (AST+M+T)	0.523	0.692	0.771	0.516
TabCS (AST+M+API)	0.489	0.665	0.747	0.485
TabCS (API+M+T)	0.535	0.700	0.779	0.525
TabCS (AST+M+API+T)	**0.585**	**0.746**	**0.813**	**0.571**

TABLE VI
EFFECTIVENESS COMPARISON OF MODELS WITH TEXTUAL AND STRUCTURAL FEATURES SETTINGS IN TERMS OF SR@1/5/10 AND MRR ON HUSAIN ET AL.'S DATASET.

Model	SR@1	SR@5	SR@10	MRR
CARLCS-CNN	0.413	0.557	0.633	0.409
CARLCS-TS	**0.439**	**0.580**	**0.656**	**0.426**
UNIF	0.421	0.556	0.624	0.419
UNIF-TS	**0.458**	**0.601**	**0.674**	**0.455**
TabCS (AST+API+T)	0.451	0.596	0.677	0.450
TabCS (AST+M+T)	0.514	0.656	0.719	0.504
TabCS (AST+M+API)	0.468	0.616	0.691	0.464
TabCS (API+M+T)	0.508	0.654	0.724	0.504
TabCS (AST+M+API+T)	**0.547**	**0.683**	**0.748**	**0.539**

These results show that the four code features(i.e., method name, API sequence, tokens, and AST), works and improves the effectiveness of TabCS. Meanwhile, the code structural feature contributes to the CARLCS-CNN, UNIF and TabCS, and its contribution does not conflict with textural features.

Result 3: *The proposed textural and structural features outperforms textural features substantially. The structural feature is necessary and useful for code search.*

D. RQ4. How Does the Two-Stage Attention Network Structure Improve the Model Performance?

Table VII shows the comparison of CARLCS-TS, UNIF-TS and TabCS on Hu et al.'s dataset. TabCS outperforms

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

the baseline models CARLCS-TS and UNIF-TS by 11.74% and 7.13% in terms of MRR; by 12.50%/10.03%/8.40%, and 8.13%/6.42%/5.17% in terms of SR@1/5/10 respectively.

Meanwhile, Table VIII shows the comparison of CARLCS-TS, UNIF-TS and TabCS on Husain et al.'s dataset. TabCS outperforms CARLCS-TS and UNIF-TS by 26.53% and 18.46% in terms of MRR; by 24.60%/17.76%/14.02%, and 19.43%/13.64%/10.98% in terms of SR@1/5/10 respectively.

These results imply that when considering both code textural and structural features, the two-stage attention network structure outperforms CARLCS-TS and UNIF-TS'networks.

TABLE VII
EFFECTIVENESS COMPARISON OF MODELS CARLCS-TS, UNIF-TS AND TABCS FOR ATTENTION MECHANISM IN TERMS OF SR@1/5/10 AND MRR ON HU ET AL.'S DATASET

Model	SR@1	SR@5	SR@10	MRR
CARLCS-TS	0.520	0.678	0.750	0.511
UNIF-TS	0.541	0.701	0.773	0.533
TabCS	**0.585**	**0.746**	**0.813**	**0.571**

TABLE VIII
EFFECTIVENESS COMPARISON OF MODELS CARLCS-TS, UNIF-TS AND TABCS FOR ATTENTION MECHANISM IN TERMS OF SR@1/5/10 AND MRR ON HUSAIN ET AL.'S DATASET

Model	SR@1	SR@5	SR@10	MRR
CARLCS-TS	0.439	0.580	0.656	0.426
UNIF-TS	0.458	0.601	0.674	0.455
TabCS	**0.547**	**0.683**	**0.748**	**0.539**

Result 4: *The proposed two-stage attention network structure substantially improves model's performance.*

VI. DISCUSSION

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

A. Why Does TabCS Work?

Above experimental results imply that the proposed model shows substantial advantages over the state-of-the-art models due to two reasons – the code structural feature and the two-stage attention network structure.

Code Structural Feature. TabCS takes the AST sequence as the structural feature. Like the example in Fig.3(a), the AST of the method body includes some methods call statements that reflect the method's function. TabCS finds the potential correlation between the method call structures and query. Therefore, TabCS predicts that this method may correspond to the given query.

Two-stage attention network structure. Fig. 3(a) shows the first retrieved results of TabCS, CARLCS-CNN, and UNIF for the query "transform date to string". We can notice that CARLCS-CNN and UNIF return irrelevant code snippets. TabCS returns the correct code snippet. For the code A in Fig.3(a), TabCS assigns less attention weights on query words (i.e., "to") and code words (i.e., "if", "null", "return", "try",

```
//Query:  transform date to string
//Code A:
public String dateToString(Date date)
{
   if (date == null) return "";
   try {
       return DateFormat.getDateTimeInstance(DateFormat.SHORT,
                                             DateFormat.LONG).format(date);    }
   catch (Exception ex) { }
   return "";
}
```

(a) TabCS's first retrieved result.

```
//Query: transform date to string
//Code B:
private static Date toDateWithFormatString(String date, String format)
{
   if (date == null) { return null; }
   try {
      return strategy.formatFor(format).parse(date);  }
   catch (ParseException e) {
     UTILS_LOGGER.trace("Unable to parse date \'{}\' using format string \'{}\': {}",
                        date, format, e);
      return null;  }
}
```

(b) CARLCS-CNN's first retrieved result.

```
//Query: transform date to string
//Code C:
private String[] getDate( String date )
{
   String dateStr = "20" + date.substring(4) + ":" + date.substring(2, 4)
                    + ":" + date.substring(0, 2);
   String[] dateArray = new String[11];
   for( int i = 0; i < dateStr.length(); i++ )
      dateArray[i] = dateStr.substring(i, i + 1);
   dateArray[10] = "";
   return dateArray;
}
```

(c) UNIF's first retrieved result.

Fig. 3. The retrieved first results for the query "transform date to string".

"catch" and "Exception") in the first stage. The second stage assigns greater weights on query words (i.e., "transform", "date", and "string") and code words (i.e., "String", "date", "to", "DateFormat", and "getDateTimeInstance"). In this way, TabCS learns the code/query representative vectors and compute their cosine similarity. Compared to code B in Fig. 3(b) and Code C in Fig. 3(c), Code A get the highest similarity. This result implies that the two-stage attention network structure is advantageous over CARLCS-CNN and UNIF.

B. Why Is TabCS Fast?

TabCS's network structure has significantly lower complexity compared to the state-of-the-art model DeepCS and CARLCS-CNN. For example, CNN needs to do convolution operations, and LSTM model has many parameters to be trained. These factors greatly increase time consumption. But in TabCS, the first stage just need to perform a attention mechanism. It's simple and has less parameters. In the second stage, co-attention mechanism only needs to train an attention parameter matrix U and its network is simple. On the whole, the construction consists of two types of attention mechanisms are much simpler and the number of parameters is much less. Thus, the training and practice time-costing of TabCS is much shorter than that of those baselines.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

C. Threats To Model Validity

The query statements in the datasets are written in English. Therefore, the model may not work as well for other languages. In addition, we only perform experiments on method-level Java code repository. We plan to extend the datasets in the near future. Our implementation of a multi-feature combination possesses some threats. We combine four structural and semantic features by concatenation directly. This combination method may ignore the connection and difference between the four features. Additionally, some parameters like the length of code features and word vector dimensions are set in default. Thus, such model setting may not be generalizable for other datasets.

VII. Related Work

This section provides the related works in three aspects: code search, AST, and attention mechanism.

A. Code Search

Since the development of software engineering, code search has been a hot topic. At the beginning, code search can be realized by search engines [29]–[31]. Mica [32] and Assieme [33] use search engines to get the results and then, extract code snippet from these results. Except for search engine, some code search tools based on IR technique has been proposed [2], [10], [34]. They take the same words of code and query as their relevance. For example, CodeHow [10] performed the code search by recognizing a user query as relevant APIs and using an Extended Boolean model. Chan et al. [2] proposed a model which returns the API sequences according to the textual similarity of query and API. However, above IR-based code search engines/tools only care about the text features, not the semantic features. [35]. With the development of natural language processing and deep learning, many tools have been proposed [16], [22], [23]. They use natural language processing technique to preprocess the code and query statements, and use DL technique to learn their semantic correlation. One of the representative models is DeepCS [23]. It embeds code and query into vector spaces by two LSTM models. CARLCS-CNN [22] uses CNN and LSTM to embed code and query, and performs a co-attention mechanism to learn their correlation. UNIF [16] perform an attention mechanism on code and query to find their relation.

B. Abstract Syntax Tree

Abstract syntax tree [36] represents the syntactic structure of programming language in the form of tree. Each node in the tree represents a structure in the source code. AST are widely used in software engineering [37]–[41]. Wang et al. [42] proposed a bug localization tool which use AST of code to find bug information. Zhang et al. [43] parse code snippets into ASTs and calculate their similarities based on ASTs to find the relation between two code snippets. Li et al. [44] proposed a source code plagiarism detection tool to calculate the similarity between programs based on AST. Wang et al. [45] presents an approach for recovering the UML class diagram from the Java source code using AST.

C. Attention Mechanism

Attention mechanism is widely used in natural language processing and image semantic extraction [46]–[48]. Results show that attention mechanism has a excellent performance on learning semantics [49]–[51]. Ueda et al. [52] proposed a request estimation method using LSTM with four Self-Attention mechanisms to represent the sentences from multiple perspectives and get excellent results. Chowdhury et al. [53] proved that attention-based models can improves the Equal Error Rate (EER) of speaker verification system. However, these models are only suitable for a single input. It doesn't work when they need to capture the interactive semantics of two or more data.

Therefore, researchers proposed co-attention mechanism which can learn the interactive semantic information from two input datas [54]–[59]. Zhang et al. [60] present a novel co-attention based network to capture the correlation between aspect and contexts and the results shows good performance. Nguyen et al. [61] proposed an approach for visual question answering using dense symmetric co-attention mechanism and achieves a new state-of-the-art on VQA and VQA 2.0. Ma et al. [62] proposed an improved multi-step multi-classification model based on co-attention mechanism to mitigate the phenomenon of error prediction, label repetition and error accumulation. We apply the traditional attention mechanism to extract semantic of code and query, and apply co-attention mechanism to address the semantic gap between code and query. And our experiment results indicate the combination of two kinds of attention mechanism is valuable and promising for the code search.

VIII. Conclusion and Future Work

In this paper, we propose a two-stage attention-based model named TabCS, which extracts textural and structural features from code, and performs a two-stage attention network structure on code features and query to learn representative vectors for them. We evaluate the proposed model TabCS on Hu's dataset and Husain et al.'s dataset. The results show that the proposed TabCS outperforms the state-of-the-art models DeepCS, UNIF, and CARLCS-CNN in terms of MRR by 70.45%, 10.66%, 17.73% on Hu's dataset; 75.57%, 28.64%, 31.78% on Husain et al.'s dataset. The experimental results indicate that the AST of code as the structural feature and the two-stage attention network structure are valuable and promising for the code search. In the future, we plan to investigate more features of source code to enhance the code representation, such as the control flow graph of code.

Acknowledgment

This work was supported in part by the National Key Research and Development Project (No.2018YFB2101200), the Fundamental Research Funds for the Central Universities (No.2019CDYGYB014 and No.2020CDCGRJ037), the National Nature Science Foundation of China (No.62002034) and the National Defense Basic Scientific Research Program (No. WDZC20205500308).

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[2] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 10:1–10:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393606

[3] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 842–851.

[4] E. Hill, M. R.Vega, J. A.Fails, and G. Mallet, "Nl-based query refinement and contextualized code search results: A user study," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb 2014, pp. 34–43.

[5] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger, "The end-to-end use of source code examples: An exploratory study," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 555–558.

[6] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pp. 545–549, 04 2015.

[7] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," 01 2011, pp. 111–120.

[8] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: A case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 191–201. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786855

[9] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[10] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," pp. 260–270, 2015.

[11] M. McCandless, E. Hatcher, O. Gospodnetić, and O. Gospodnetić, *Lucene in action*. Manning Greenwich, 2010, vol. 2.

[12] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Communications of the ACM*, vol. 26, no. 11, pp. 1022–1036, 1983.

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

[15] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *28th International Conference on Program Comprehension (ICPC)*, 2020.

[16] 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.

[17] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, pp. 1–39, 2019.

[18] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[19] M. W. Gardner, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric Environment*, vol. 32, 1998.

[20] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Computer Science*, 2014.

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

[22] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning."

[23] G. Xiaodong, Z. Hongyu, and K. Sunghun, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 933–944.

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

[25] W. Yin, H. Schütze, B. Xiang, and B. Zhou, "Abcnn: Attention-based convolutional neural network for modeling sentence pairs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 259–272, 2016.

[26] 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.

[27] X. Yao, B. Van Durme, and P. Clark, "Automatic coupling of answer extraction and information retrieval," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, pp. 159–165. [Online]. Available: https://www.aclweb.org/anthology/P13-2029

[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *computer science*, vol. 1409, 09 2014.

[29] M. Arora, "How google search engine works," *Electronics for You*, 2013.

[30] K. Krugler, *Krugle Code Search Architecture*, 2013.

[31] B. R. Muddu, A. M. Asadullah, J. Vinod, and K. K. Pooloth, "Structural search of source code," 2012.

[32] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK*, 2006.

[33] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: finding and leveraging implicit references in a web search interface for programmers," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 2007, pp. 13–22.

[34] D. Manjula, S. Kulandaiyan, S. Sudarshan, A. Francis, and T. V. Geetha, "Semantics based information retrieval using conceptual indexing of documents," in *Intelligent Data Engineering Automated Learning, International Conference, Ideal, Hong Kong, China, March, Revised Papers*, 2003.

[35] A. Babashzadeh, J. Huang, and M. Daoud, "Exploiting semantics for improving clinical information retrieval," p. 801, 2013.

[36] A. S. Graph, "Abstract syntax tree," 2015.

[37] Yan Zhang, Xinyu Gao, Ce Bian, Ding Ma, and Baojiang Cui, "Homologous detection based on text, token and abstract syntax tree comparison," in *2010 IEEE International Conference on Information Theory and Information Security*, 2010, pp. 70–75.

[38] 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)*, 2019, pp. 95–104.

[39] 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*, 2019.

[40] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, and Y. L. Traon, "ifixr: Bug report driven program repair," 2019.

[41] I. G. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, 2005.

[42] B. Wang, L. Xu, M. Yan, C. Liu, and L. Liu, "Multi-dimension convolutional neural network for bug localization," *IEEE Transactions on Services Computing*, pp. 1–1, 2020.

[43] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," *International ConferenceonSoftwareEngineering*, pp. 1–1, 2020.

[44] X. Li and X. J. Zhong, "The source code plagiarism detection using ast," in *2010 International Symposium on Intelligence Information Processing and Trusted Computing*, 2010, pp. 406–408.

[45] X. Wang and X. Yuan, "Towards an ast-based approach to reverse engineering," in *2006 Canadian Conference on Electrical and Computer Engineering*, 2006, pp. 422–425.

[46] A. P. Parikh, O. Tckstrm, D. Das, and J. Uszkoreit, "A decomposable attention model for natural language inference," 2016.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.

[47] Y. Liu, C. Sun, L. Lin, and X. Wang, "Learning natural language inference using bidirectional lstm model and inner-attention," 2016.
[48] A. Gajbhiye, S. Jaf, N. A. Moubayed, S. Bradley, and A. S. Mcgough, "Cam: A combined attention model for natural language inference," in *IEEE International Conference on Big Data*, 2018.
[49] M. Liu and H. Yin, "Cross attention network for semantic segmentation," in *2019 IEEE International Conference on Image Processing (ICIP)*, 2019, pp. 2434–2438.
[50] X. Bai, "Text classification based on lstm and attention," in *2018 Thirteenth International Conference on Digital Information Management (ICDIM)*, 2018, pp. 29–32.
[51] S. Yadav and A. Rai, "Frequency and temporal convolutional attention for text-independent speaker recognition," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 6794–6798.
[52] T. Ueda, M. Okada, N. Mori, and K. Hashimoto, "A method to estimate request sentences using lstm with self-attention mechanism," in *2019 8th International Congress on Advanced Applied Informatics (IIAI-AAI)*, 2019, pp. 7–10.
[53] F. A. Rezaur rahman Chowdhury, Q. Wang, I. L. Moreno, and L. Wan, "Attention-based models for text-dependent speaker verification," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 5359–5363.
[54] Y. Jing, L. Yuhang, Z. Weifeng, Q. Zengchang, L. Yanbing, and H. Yue, "Learning cross-modal correlations by exploring inter-word semantics and stacked co-attention," *Pattern Recognition Letters*, pp. S0 167 865 518 304 380–, 2018.
[55] D. K. Nguyen and T. Okatani, "Improved fusion of visual and language representations by dense symmetric co-attention for visual question answering," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
[56] Y. Tay, A. T. Luu, and S. C. Hui, "Multi-pointer co-attention networks for recommendation," in *the 24th ACM SIGKDD International Conference*, 2018.
[57] 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)*, 2019.
[58] 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)*, 2019, pp. 201–208.
[59] B. Li, Z. Sun, Q. Li, Y. Wu, and H. Anqi, "Group-wise deep object co-segmentation with co-attention recurrent neural network," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 8518–8527.
[60] P. Zhang, H. Zhu, T. Xiong, and Y. Yang, "Co-attention network and low-rank bilinear pooling for aspect based sentiment analysis," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 6725–6729.
[61] D. Nguyen and T. Okatani, "Improved fusion of visual and language representations by dense symmetric co-attention for visual question answering," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6087–6096.
[62] H. Ma, Y. Li, X. Ji, J. Han, and Z. Li, "Mscoa: Multi-step co-attention model for multi-label classification," *IEEE Access*, vol. 7, pp. 109 635–109 645, 2019.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on July 15,2021 at 10:40:36 UTC from IEEE Xplore. Restrictions apply.