A Code Search Method Incorporating Code Annotations

Qi Li[1,2], Jianxun Liu[1,2(✉)], and Xiangping Zhang[1,2]

[1] School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411201, Hunan, China
37323474@qq.com

[2] Hunan Key Lab for Services Computing and Novel Software Technology, Hunan University of Science and Technology, Xiangtan 411201, Hunan, China

Abstract. Code search is a technique for users to retrieve code snippets from the Code base using natural language, which is dedicated to retrieve the target code accurately and quickly to improve the efficiency of software development. The deep learning based code search technique greatly improves the accuracy of search by learning the relationship between code and query statements. Since it relies on the extracted code features, acquiring more code features is the key to quickly improve the search performance. However, most of the previous works have not taken code annotations into consideration. In this paper, we take code annotations as code features and apply them to code search, which is named ICA-CS (Code Search that Incorporates Code Annotations). In the method, firstly, the code features are embedded to get the corresponding vector representation. It is then processed by bidirectional LSTM (Long Short-Term Memory) network or multi-head attention respectively, followed by features fusion. And finally, the model is trained by joint embedding and using the minimised ranking loss function. As the experimental results show, on the evaluation metric MRR (mean reciprocal rank) compared to the state-of-the-art models DeepCS, SAN-CS, CARLCS-CNN and SelfAtt, the proposed model improves 48.96%, 17.11%, 41.01% and 13.07%, respectively.

Keywords: Code search · Code features · Code annotations · Bidirectional LSTM · Multi-head attention

1 Introduction

Code search has received a lot of attention in recent years [1–5]. The goal of code search is to perform natural language queries from large code corpora to retrieve code snippets that meet developers' needs [6]. Studies have shown that developers spend an average of 19% of their total development time searching for code online [7], and code search plays an important role in the software development process, helping to improve developer productivity and shorten product development cycles [7, 8].

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 561, pp. 323–342, 2024.
https://doi.org/10.1007/978-3-031-54521-4_18

Nowadays, there are more and more code resources available. On the popular platform GitHub alone, there will be 413 million open source contributions in 2022 [9]. It follows that most of the code written by developers has already been written by others [10, 11] and that this code is easily accessible on open source platforms. However, developing accurate code search engines faces considerable challenges [10, 12, 13]. Developers often have difficulties in finding satisfactory code in a short period of time, which may affect the development progress of the project. Therefore, how to find the target code efficiently and quickly becomes a major challenge. For this reason, researchers have been trying to explore various code search methods to help developers.

Early code search methods were mainly based on information retrieval (IR) techniques, especially keyword matching mechanisms. For example, Lv et al. proposed CodeHow [14], which combines text similarity and API matching through an extended Boolean model. McMillan et al. proposed the Portfolio [15], which returns a series of functions through keyword matching and the PageRank algorithm [16].Lu et al. used WordNet [17] to expand the synonyms of a query and perform keyword matching of method names. However, these approaches only treat codes and queries as plain text and do not fully utilize the structural or semantic information in the source code, which leads to unsatisfactory code search [1, 18, 19].

To address the above problems, researchers have started to widely apply deep learning techniques in code retrieval tasks to learn deeper semantic information between the query statements and the source code [6]. DeepCS [20] proposed by Gu et al. is one of the pioneers, which is a model that applies deep learning techniques to code search for the first time. DeepCS utilizes recurrent neural network (RNN) and multilayer perceptron (MLP) to represent code snippets and queries as feature vectors, and trained the model by computing the similarity function of code and query statements with a minimized ranking loss function. The results of the study showed that the deep learning approach achieved better performance in code search. Subsequently, many research works followed the framework of DeepCS, focusing on exploring in mining more code features. Shuai et al. proposed a model CARLCS-CNN [4], which utilizes convolutional neural networks (CNNs) and the joint attention mechanism to learn the correlation between code and query. Cambronero et al. proposed a simple model UNIF [2], which utilizes an attention layer for code and query embedding. To mine more code features, Wan et al. proposed MMAN [5], which combines multiple semantic information of code, including source code token sequences, Abstract Syntax Trees (ASTs), and Control Flow Graphs (CFGs). Zeng et al. proposed deGraphCS [1], which proposes data flow and control flow from intermediate representations of code. Liu et al. proposed GraphSearchNet [21] to represent codes and queries using a unified graph structure. These works start from the properties of the code itself and focus on exploring more features of the code to improve the accuracy of the search. However, none of these works considered the impact of code annotations information on the code retrieval task, which may have a significant impact on the accuracy of the search.

Code annotations describe the main functions of the source code through natural language forms and can help developers to understand the code quickly [22]. High-quality code annotations improve the readability and comprehensibility of the code and play an important role in the development and maintenance of the program [23], thus it is an

integral part of the code. In the code snippet shown in Fig. 1 and the process of extracting code features, DeepCS extracts method names, application programming interface (API) sequences, and code token sequences from the code [20], and even though it extracts relatively comprehensive information, for code annotations information is not fully captured. These annotations contain a lot of useful information, such as descriptions of parameters, explanations of return values, handling of exceptions, and version information. In addition, users also enter information related to these aspects, such as restrictions on input or return values, when performing code searches [24]. Therefore, code annotations also contain important information that can be helpful for code search tasks.

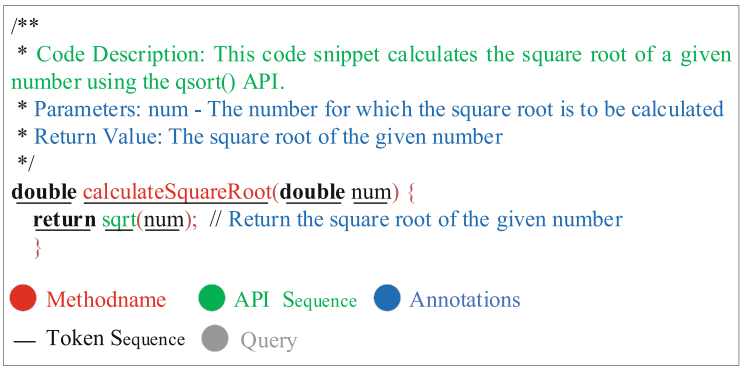

Fig. 1. Example of code features extracted

Mainstream code search models have made significant progress in exploring more features, however, code annotations are rarely addressed in existing research and are not explored in depth. For example, literature [25] simply incorporates code annotations into token sequences. Currently, there are fewer works that investigate code annotations in depth in the direction of code search. To address this point, we introduce code annotations into code search and proposes a new model called ICA-CS. Specifically, ICA-CS extracts four features from code: method names, API sequences, code annotations, and code token sequences, and it learns the relationship between code and query statements by using a bidirectional LSTM to process method names and query statements, and a multi-head attention mechanism to process API sequences, code annotations, and code token sequences. The results show that introducing code annotations into code search with deep exploration can significantly improve the performance of code search models.

The contributions of this paper are as follows:

1. We propose the model ICA-CS, which fuses method names, API sequences, code annotations and code token sequences to extract more code features information and significantly improve the accuracy of code search.
2. In this paper, code annotations are introduced into code search as one of the features of code for the first time, and are validated using three models (DeepCS, SAN-CS, and the proposed model ICA-CS). The experiments demonstrate that the search performance of all models is improved after the introduction of code annotations.

2 Technical Background

2.1 Joint Embedding

Joint embedding is a technique for jointly embedding heterogeneous data into a unified vector space [26], aiming to bring semantically similar concepts of two modalities close to each other in the vector space [27].

Since codes and queries are heterogeneous [20], correlations between them are difficult to detect directly. Therefore, after completing the representation of codes and queries, we jointly embed them. Specifically, the joint embedding is realized by calculating the cosine similarity between code vectors V_c and query vectors V_q.

$$sim(V_c, V_q) = cos(V_c, V_q) = \frac{V_c^T \cdot V_q}{||V_c|| \cdot ||V_q||} \tag{1}$$

where V_c denotes the vector representation of the code and V_q denotes the vector representation of the query. The larger the cosine similarity, the higher the correlation between the corresponding code snippet and the query. Finally, the recommendation list is obtained by ranking the query vectors and the vectors of candidate code fragments in descending order based on the cosine similarity between them [19].

2.2 Word Embedding and Sequence Embedding

Embedding is a technique for representing entities as vectors [28, 29], which can be used to transform various entities, such as words, utterances, books, etc., into dense vector representations of fixed length so that machines can understand and process them [30]. Among them, word embedding is a typical embedding technique [28, 29], which represents words as fixed-length vectors such that semantically similar words are close together and semantically dissimilar words are farther apart in the vector space [20]. Word embeddings are usually implemented using models such as continuous bag-of-words (CBOW) or Skip-Gram [29], which are trained using a text corpus by building a neural network to capture the relationship between a word and its context [30].

Sequence embedding treats a sentence as a bag of words, and then embeds the words one by one, and finally integrates them into a tensor to represent the embedding vector of the whole sequence. In the process of sequence embedding, since each word is independent of each other, the contextual logical relationship is still preserved after the sequence embedding is completed.

2.3 Bidirectional LSTM

LSTM is an RNN architecture designed to solve the gradient vanishing and gradient explosion problems in traditional RNNs [31]. Bidirectional LSTM consists of forward LSTM and backward LSTM. Taking “open an xml file” as an example, the vectors of each word are obtained after sequence embedding, and then these vectors are fed into the forward LSTM and the backward LSTM for computation to obtain their outputs respectively. Finally, the vectors obtained from the forward LSTM and the backward

LSTM are concatenated to obtain the final output vectors. The ability of the bidirectional LSTM to utilize both the previous and subsequent information in the sequence makes it more effective in dealing with code features sequence data, which greatly improves the accuracy of the search.

2.4 Multi-head Attention

Multi-head Attention is the key idea in Transformer [32], which is widely used in the field of natural language processing (NLP). An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are vectors. In practice, a set of queries are computed simultaneously using the scaled dot product attention [32] function, which packs them into a matrix Q. The keys and values are also packed together into matrices K and V. The output matrix is computed as follows:

$$Attention(Q, K, V) = SoftMax(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

where d_k is the dimension length and Q, K and V represent the query, key and value matrices.

To further improve the expressiveness of self-attention, the query, keys and values are projected h times with different linear projections and then the projections are concatenated to obtain the final output, a computational process known as multi-head attention [21, 32]:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O$$

$$\text{where head}_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

where the projections are parameter matrices $W_i^Q \in R^{d_{model} \times d_k}$, $W_i^K \in R^{d_{model} \times d_k}$, $W_i^V \in R^{d_{model} \times d_v}$ and $W^O \in R^{hd_v \times d_{model}}$, h is the number of heads.

On the basis of multi-attention, in order to better extract the values of the feature vectors, we employ an average pooling strategy to extract the key information. Average pooling is a feature extraction method that performs an averaging operation on each feature dimension in the input data to obtain a single value representing the feature. Similar to the average pooling strategy is the maximum pooling strategy, but it is not used, on the one hand due to the fact that it only focuses on the highest feature values, which may ignore some important features [19]. On the other hand based on the experimental results, it is found that the average pooling strategy can obtain better performance.

3 The Proposed Model

Figure 2 illustrates the general framework of the proposed model, which is divided into three main parts: model representation, feature extraction, and model training. In the model characterization, the extracted codes and queries are transformed into vector

representations using embedding techniques to characterize the features information in the form of vectors. The code includes four features information: method name, API sequence, code annotations and code token sequence. The feature extraction part mainly extracts the feature vectors after the model characterization. For method name and query, the feature information is extracted by using bidirectional LSTM network, while for API sequence, code annotations and code token sequence, the feature information is extracted by using multi-attention and average pooling strategies. After the feature extraction is completed, the four features information of the code is sent to the fusion layer for features fusion, and the vector representation of the code is obtained by representing the code segments with the fused features. The model training part utilizes the obtained vector representations of the code and the query, calculates the similarity between them through the cosine similarity function, and ranks them. Then, the model is trained using a minimized ranking loss function to optimize the performance of the model. Through the collaboration of these three parts, the proposed model ICA-CS is able to perform code search more accurately, thus improving the efficiency and accuracy of code retrieval.

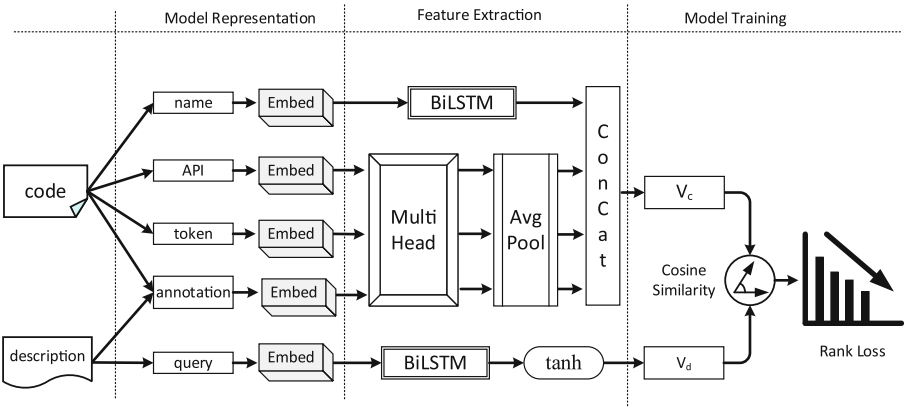

Fig. 2. The framework of the proposed method.

3.1 Code Representation

Code characterization is mainly divided into the following four aspects: method name, API sequence, code annotations and code token sequence. Specifically, the four features information are first characterized as vectors after embedding, then feature extraction is performed by the corresponding model, and finally the four extracted feature vectors are sent to the fusion layer for feature fusion to obtain the final vector representation, i.e., code vector. The following is the detailed operation of the code characterization part.

Method Name. The method name is a brief summary of the function of the whole code, has strong contextual logic and is usually short in length. The bidirectional LSTM network is adopted for feature extraction, using the output of the last time step therein to represent the entire information of a sentence. Suppose a method name is denoted as

$S_m = [m_1, m_2, ..., m_n]$, which after passing through the embedding layer is denoted as $E_m = [em_1, em_2, ..., em_n]$, and is then fed into a bidirectional LSTM network, which outputs its hidden vectors to represent the method names V_m. The whole process can be represented as:

$$V_m = BiLSTM(Embed(S_m)) \tag{4}$$

API Sequences. In code, API sequences usually consist of multiple API calls, which are usually long in length [30]. Although there is a logical relationship between some related APIs, most of them may not have direct correlation with each other, and their semantic information may not be fully captured using bi-directional LSTM networks, so multi-head attention is used to extract the information of API sequences. Suppose an API sequence $S_a = [a_1, a_2, ..., a_n]$, after the embedding layer is denoted as $E_a = [em_1, em_2, ..., em_n]$, which is then fed into the multi-head attention to get M_a, followed by average pooling to obtain the final vector representation of the API sequence V_a. The whole process can be represented as:

$$M_a = AvgPool(MultiHead(Embed(S_a))) \tag{5}$$

Code Annotations. The information in code annotations includes many, such as the description of parameters, the explanation of return values, the handling of exceptions and the description of specific code, etc. In order to fully utilize the various information in code annotations, we adopt the multi-head attention mechanism to extract the features of code annotations. Let a code annotations $S_o = [o_1, o_2, ..., o_n]$, which is represented as $E_o = [eo_1, eo_2, ..., eo_n]$, which is subsequently fed into the multi-head attention to get M_o, after average pooling to get the final vector representation of the code annotations V_o. The whole process can be represented as:

$$M_o = AvgPool(MultiHead(Embed(S_o))) \tag{6}$$

Code Token Sequences. For the code token sequences, which are a collection of all words in code, there is usually no strict order relationship, so it is a reasonable choice to use multi-head attention to extract the information. Let the code token sequence as $S_t = [t_1, t_2, ..., t_n]$, denoted as $E_t = [et_1, et_2, ..., et_n]$, which is subsequently fed into the multi-head attention to obtain M_t, and after average pooling, the final vector representation of the code token sequence V_t is finally obtained. The whole process can be expressed as:

$$M_t = AvgPool(MultiHead(Embed(S_t))) \tag{7}$$

Code Features Fusion. According to the above operation, the vector representation of the four features of the code has been obtained, and now it is ready to fuse these features to get the final code vector representation. We do not use simple summation for fusion, but chooses concatenation, the concatenation operation can connect the information between each feature in an orderly manner, preserving the independent information of each feature. Subsequently, a linear transformation is performed on the concatenated vector, and finally the activation function $tanh()$ is used to obtain the final code vector V_c. The whole process can be expressed as:

$$V_c = tanh(Linear(Concat(V_m, V_a, V_o, V_t))) \tag{8}$$

3.2 Query Representation

The query statement usually contains the code function or problem description that the user wants to find, and has a strong contextual logical relationship, it is a good choice to use a bidirectional LSTM network to extract the information of the query statement. A query statement $S_q = [q_1, q_2, ..., q_n]$, which is represented as $E_q = [eq_1, eq_2, ..., eq_n]$ after the embedding layer, is then fed into a bidirectional LSTM network, which outputs its hidden vector and finally uses an activation function *tanh*() to obtain the final query vector V_q. The whole process can be represented as:

$$V_q = tanh(BiLSTM(Embed(S_q))) \tag{9}$$

3.3 Model Training

Through the above steps, the code vector V_c and the vector representation of the corresponding query V_q can be obtained, and in order to measure the match between the code vector and the query vector, the cosine similarity is used to measure the similarity between them. Specifically, in the training phase, in order for the model to learn the correct code-query matching relationship, the ternary < c, d +, d- > is used for training, where c denotes the code snippet, d + denotes the query statement that matches with code c, and d- denotes the irrelevant query statement. The goal of training is to make the similarity of < c, d + > as high as possible, i.e., to bring the matching code and query vectors closer together, and to minimize the similarity of < c, d- >, i.e., to keep the irrelevant code and query vectors as far away as possible [20]. To achieve this goal, the model is trained by minimizing the ranking loss function [33, 34], which is represented as follows:

$$L(\theta) = \sum\nolimits_{c \in Cd^+, d^- \in D} \max(0, \beta - sim\left(c, d^+\right) + \mathrm{sim}(c, d^-)) \tag{10}$$

where θ represents the parameters of the model, C denotes the code portion of the dataset, D denotes the query statement portion of the dataset, and *sim*() denotes the function that calculates the similarity between the code and the code descriptions, and we use the cosine similarity function. β is a constant and is set to 0.35 in the experiment.

Optimizer. In order to optimize the training process of the model, the AdamW optimizer is used and the learning rate is varied linearly. This adjustment can effectively avoid the instability of the model in the early stage of training and help accelerate the convergence speed of the model. Specifically, the linear variation of the learning rate is carried out during the training process according to the following equation:

$$Lr = Or \cdot Cr \, where Cr = \begin{cases} 1.0/current & , current < warmup \\ 1.0 & , current \geq warmup \end{cases} \tag{11}$$

where *Lr* represents the current learning rate, *Or* represents the set learning rate, the value is 1e–4. *Cr* represents the change rate, the current training times *current* is less than the set number of times *warmup*, *Cr* is the inverse of the number of training times, the current training times *current* is greater than or equal to the set number of times *warmup*, *Cr* value is 1.0. In this experiment, $warmup = epoch * 0.05$, *epoch* represents the number of times the model needs to be trained.

4 Experiments Setup

4.1 Dataset

Date Cleaning: Data cleaning is an important step in data processing to ensure data quality and consistency. We use the original dataset disclosed by degraphCS [1]. It should be noted that we use its original dataset rather than the dataset used in the paper, which is cleaned according to the following cleaning rules.

- Remove duplicate code snippets;
- Remove code snippets without code descriptions;
- Remove code snippets that are less than 3 lines long;
- Remove code snippets where the number of words in the query statement is less than 3;
- Remove code snippets with non-English code descriptions;
- Remove code snippets that have too high a percentage of single characters in query statements

Finally, 43,363 pieces of data were obtained through data cleaning, which were divided into 39,363 training sets and 4,000 test sets. When conducting the test, we use one correct candidate and 3,999 interference items to test the model, i.e., the model will retrieve all the code resources of the codebase, so that the test method can more realistically reflect the effect of the model in the actual application.

Word Split Rules: split the words from the feature information in the code and convert them to lowercase form so that the machine can understand and process them better. Two splitting rules are used to handle different forms of words: the camel case and the special character rule. The camel case splits words based on case, for example, "readXmlFiles" is split into "read", "xml" and "files". And the special character rule splits according to the specific character "_". For example, "read_xml_files" is split into "read", "xml" and "files". By using these splitting rules, the machine can embed different forms of words into the common vector space and understand the same semantics expressed by these words. This can help improve the model's ability to understand and express code features information, thus improving the accuracy and performance of code search.

4.2 Evaluation of Indicators

In order to evaluate the effectiveness of the proposed model ICA-CS, three common evaluation metrics are used, which are $SuccessRate@k$, MRR and $NDCG@k$.

$SuccessRate@k$ is used to measure whether the correct result is included in the top k recommended results, i.e., whether the user requirement item is successfully recommended to the user, as defined below:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(Rank_q) \tag{12}$$

where $|Q|$ is the number of samples, which can be interpreted as the number of user's requirement items. $Rank_q$ is the position of the result of the q_{th} query in the list of items

recommended by the model, and $\delta()$ is used to express whether the correct result of the q_{th} query is included in the list of k results recommended by the model, and its value is 1 if it is in; otherwise, it is 0. The k in *SuccessRate@k* are taken to be 1, 5, and 10, and the higher their scores are, indicating that more users' demanded items have been successfully recommended to the users.

MRR (mean reciprocal rank) is the average of the inverse rank of a set of queries Q in the top k ranked results, *MRR* measures the position of the user's demand in the top k recommended results and is computed as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{Rank_q} \tag{13}$$

where $|Q|$ is the number of samples, which can be interpreted as the number of user's input queries. $Rank_q$ is the position of the result of the q_{th} query in the list of items recommended by the model. The higher *MRR* score indicates, the better performance of the model search.

NDCG@k (normalized discounted cumulative gain at k) is a metric used to evaluate the accuracy of the sorted results. Recommender systems usually return a list of results to the user, and since there may be many results and the user will only view the first part, *NDCG@k* is used to measure the accuracy of the first k recommendations in the sorted results. It is the division of *DCG* (discounted cumulative gain) and *IDCG* (ideal discounted cumulative gain), where *DCG*, *IDCG* and *NDCG* are calculated as follows:

$$DCG@k = \sum_{i=1}^{k}\frac{2^{rel_i}-1}{log_2(i+1)} \qquad IDCG@k = \sum_{i=1}^{k}\frac{2^{rel_i^{ideal}}-1}{log_2(1+i)}$$

$$NDCG@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{DCG@k}{IDCG@k} \tag{14}$$

where $|Q|$ is the number of samples, rel_i is the relevance of the i_{th} item in the recommendation ranking results, rel_i^{ideal} denotes that it is the relevance of the i_{th} item in the ideal ranking results, and k denotes the maximum value that can be given to the query, and the value is taken to be 10, for the purpose of this experiment. A code search model with high *NDCG@k* scores implies that it not only has a high overall search quality, but also ranks the results that the user needs in the top position.

For the above evaluation metrics, larger values represent higher search accuracy and better model performance. In order to demonstrate the performance of the model more intuitively, we use the following formula for the evaluation of model performance improvement:

$$improvement = \frac{P_{new}-P_{old}}{P_{old}} \tag{15}$$

P_{new} is the performance of the improved model, and P_{old} is the performance of the original model. The larger the value of *improvement*, the greater the improvement in model performance.

4.3 Baseline Models

- **DeepCS:** DeepCS [20] applies deep learning to code search for the first time, capturing semantic information of code from method names, API sequences, and code token sequences, etc., representing the source code and query statements using uniform vectors and computing the similarity.
- **SAN-CS:** SAN-CS [30] is a code search model based on self-attention networks, which utilizes self-attention networks to performjoint representation of code snippets and their descriptions separately.
- **CARLCS-CNN:** CARLCS-CNN [4] uses a co-attention mechanism to learn the correlation matrix between embedded codes and queries and jointly participate in their semantic relations.
- **SelfAtt:** SelfAtt is one of the four baseline models used by CodeSearchNet [25] and is the best performing model, which uses the BERT [35] encoder to encode the code and the query, and then computes a similarity score between the code and query representations.

4.4 Experimental Parameters

First, the data set is organized by setting the batch size to 64, and constructing separate vocabularies for each code features (method name, API sequence, code annotations, code token sequence) and query statement, and storing the 10,000 most frequent words in the data set in their respective vocabularies. In each batch, code features and query statements are padded to reach the maximum length using a special marker "PAD", which is set to 0. In addition, all features in the dataset are split into sequences according to the specified splitting rules and converted to lower case, e.g. and converted to lowercase, for example, "open_xml_files" will be split into "open xml files" sequence, "readTxtFiles" will be split into "read txt files". Next, we use the AdamW optimizer to update the model parameters, set the initial learning rate to 1e-4, and adopt a variable learning rate for warm-up, with a warm-up ratio of 0.05. The number of training times is set to 300, and the gradient trimming threshold is set to 5 in order to prevent the gradient from exploding. The word embedding size is set to 1024, the dimension of the bidirectional LSTM network is set to 512, and a multi-head attention mechanism with 4 attention heads is also used. To minimize the possibility of overfitting, the dropout is set to 0.1. All experiments are run experimentally on a server with Ubuntu 18.04.5 operating system using dual 2080ti GPUs.

5 Experimental Results

In this section, in order to evaluate the performance of the proposed model in terms of code search, the following questions are posed for investigation:

- **Question 1:** How is the performance of the proposed model compared to some state-of-the-art models? In addition, can code annotations be used as code features and how much does it improve the accuracy of code search?

- **Question 2:** Is the bidirectional LSTM network used in the model better than other recurrent neural networks? Does the multi-head attention mechanism perform better compared to other attention mechanisms? Is the average pooling strategy better than the maximum pooling strategy?
- **Question 3:** Which of the four features (method names, API sequences, code annotations, and code token sequences) is used in the model and which one contributes the most?
- **Question 4:** How much do different parameters affect the model?

Question 1 explores the performance of the model ICA-CS proposed in this paper and the effect of code annotations on code search; Question 2 discusses the effectiveness of the neural network model used in ICA-CS; Question 3 explores the contribution of individual features; and Question 4 discusses the effect of the model's parameters on the search performance.

5.1 Question 1: Model Performance and the Extent to Which Code Annotations Improve the Performance of Code Searching

Table 1 demonstrates the performance evaluation results of the model proposed and the comparison model on the test samples of the dataset, and it can be clearly seen that the proposed model significantly outperforms the comparison model in all evaluation metrics. Compared with the comparison model DeepCS, the proposed model improves 73.65%, 36.92% and 25.99% on SuccessRate@1, 5 and 10, and 48.96% and 43.08% on MRR and NDCG@10, respectively. Compared to the comparison model SAN-CS, the proposed model improves 24.56%, 11.49% and 9.52% on SuccessRate@1, 5 and 10, and 17.11% and 15.23% on MRR and NDCG@10, respectively. Compared with the comparison model CARLCS-CNN, the proposed model improves 61.45%, 29.69% and 20.6% on SuccessRate@1, 5 and 10, and 41.01% and 35.97% on MRR and NDCG@10, respectively. Compared to the comparison model SelfAtt, the proposed model improves 18.74%, 8.63% and 7.01% on SuccessRate@1, 5 and 10, and 13.07% and 11.52% on MRR and NDCG@10, respectively.

The combined results show that the proposed model significantly outperforms all the compared models in code search. The ICA-CS model synthesizes a variety of features to achieve better performance in code search, and also adopts a more appropriate mechanism for extracting information for different features, and thus is able to achieve significant performance improvement.

In order to verify whether code annotations can be used as a feature of the code, code annotations are added to DeepCS, SAN-CS and ICA-CS models and the performance is evaluated. The results are shown in Table 1, where the performance of all models is significantly improved after adding code annotations. Adding code annotations to DeepCS improves 18.13%, 10.7%, and 6.7% at SuccessRate@1, 5, and 10, and 11.94% and 10.54% at MRR and NDCG@10, respectively. Similarly, adding code annotations to SAN-CS improves 10.42%, 5.78% and 4.27% on SuccessRate@1, 5 and 10, and 7.68% and 6.81% on MRR and NDCG@10, respectively. Whereas, in the proposed model, the addition of code comments equally improves the performance by 18.64%, 7.66% and 6.09% on SuccessRate@1, 5 and 10, and 12.09% and 10.48% on MRR and NDCG@10, respectively.

Table 1. Experimental results of the proposed model and the comparison models

Model	SR@1	SR@5	SR@10	MRR	NDCG@10
DeepCS	0.2383	0.4393	0.5298	0.3368	0.3749
SAN-CS	0.3322	0.5395	0.6095	0.4284	0.4655
CARLCS-CNN	0.2563	0.4638	0.5535	0.3558	0.3945
SelfAtt	0.3485	0.5537	0.6238	0.4437	0.4810
ICA-CS	**0.4138**	**0.6015**	**0.6675**	**0.5017**	**0.5364**
DeepCS	0.2383	0.4393	0.5298	0.3368	0.3749
	0.2815	**0.4863**	**0.5653**	**0.3770**	**0.4144**
SAN-CS	0.3322	0.5395	0.6095	0.4284	0.4655
	0.3668	**0.5707**	**0.6355**	**0.4613**	**0.4972**
ICA-CS	0.3488	0.5587	0.6292	0.4476	0.4855
	0.4138	**0.6015**	**0.6675**	**0.5017**	**0.5364**

In summary, code annotations can be an important feature of code and can significantly improve the accuracy of code search. In particular, for SuccessRate@1, the performance improvement brought by the addition of code annotations is most obvious, which indicates that code annotations play a key role in accurately matching the relationship between code and query.

5.2 Question 2: Performance of Individual Components Used in the Model

In order to study the advantages of bidirectional LSTM, RNN, Bidirectional RNN (BiRNN) and LSTM networks are chosen as a comparison and the results are shown in Table 2. The following conclusions can be drawn, (1) LSTM networks perform significantly better than RNN networks. This is because LSTM networks have memory cells compared to traditional RNN networks, which can effectively solve the long-term dependency problem, enabling the model to achieve better performance when dealing with long sequences. (2) The performance of using bi-directional neural networks is better than unidirectional neural networks. By studying the principle of LSTM network, it can be found that LSTM network prefers to learn the information at the end of the sentence and ignores the more distant head information. In concise code, the head information is not less important than the tail information, so using the inverse LSTM in a bidirectional LSTM network can preserve the head information and combine it with the tail information of the forward LSTM, thus comprehensively capturing the contextual logical relationships in the code features, and thus improving the accuracy of the code search.

In order to verify the advantages of the multi-attention mechanism, this experiment uses self-attention, attention mechanism, summation mechanism and averaging mechanism as a comparative experiment. Self-attention utilizes the self-attention network to learn the different weights of the node vectors while attention utilizes the attention

Table 2. Performance of the proposed model using various neural network models

Model	SR@1	SR@5	SR@10	MRR	NDCG@10
RNN	0.1128	0.2678	0.3565	0.1865	0.2209
BiRNN	0.1763	0.3590	0.4468	0.2674	0.3007
LSTM	0.2795	0.4778	0.5573	0.3733	0.4100
ICA-CS	**0.4138**	**0.6015**	**0.6675**	**0.5017**	**0.5364**
self-attention	0.3455	0.5575	0.6290	0.4427	0.4812
attention	0.3065	0.5083	0.5935	0.4038	0.4418
average	0.2667	0.4743	0.5555	0.3653	0.4030
sum	0.2137	0.4160	0.5065	0.3109	0.3485
ICA-CS	**0.4138**	**0.6015**	**0.6675**	**0.5017**	**0.5364**
Max Pooling	0.2988	0.5020	0.5750	0.3947	0.4308
ICA-CS	**0.4138**	**0.6015**	**0.6675**	**0.5017**	**0.5364**

network to learn the different weights of the node vectors. Average averages the vectors across all the nodes, and sum sums the vectors of all nodes.

All these mechanisms aim to extract more critical features information. As can be seen from Table 2, the multi-head attention mechanism achieves the best performance on all evaluation metrics, and also the attention mechanism outperforms the sum and average mechanisms. We argue that, firstly, the attention mechanism can focus more on important information, thus ignoring unimportant information; secondly, the use of the multi-head attention mechanism can extract the critical information of the code features more effectively, thus improving the accuracy of the model.

The maximum pooling operation is compared with the average pooling operation used in ICA-CS, and the results are shown in Table 2. The results show that the average pooling operation is more suitable for ICA-CS, and the maximum pooling operation selects the most prominent features information, but this may not be able to represent all the features information completely. In contrast, the average pooling operation can fuse more features information and express the overall features more accurately, which improves the search performance. Therefore, the average pooling operation is a more effective strategy.

5.3 Question 3: Ablation Experiments

In this experiment, four features of the code (method names, API sequences, code annotations, and code token sequences) are experimented with separate deletions to validate the contribution of each feature to the performance of the model. As can be seen in Table 3, method name has the greatest impact on search performance, followed by code token sequence, followed by code annotations, while API has the least impact on code search. We speculate that this is because the method name best visualizes the specific

function of the code, and therefore has the most significant impact on code search performance. Whereas not every piece of code involves an API sequence, therefore API has less impact on code search performance.

Table 3. Results of ablation experiments

Model	SR@1	SR@5	SR@10	MRR	NDCG@10
-w/o methodname	0.2375	0.4253	0.5028	0.3277	0.3618
-w/o api	0.4012	0.5847	0.6455	0.4863	0.5189
-w/o annotations	0.3488	0.5587	0.6292	0.4476	0.4855
-w/o token	0.2988	0.5038	0.5890	0.3962	0.4353
ICA-CS	**0.4138**	**0.6015**	**0.6675**	**0.5017**	**0.5364**

5.4 Question 4: Sensitivity Experiments

Sensitivity experiments are conducted in various aspects such as number of heads, annotations length, learning rate, dimension size, batch size, and β-value, respectively. As can be seen in Fig. 3, the model is insensitive to the number of heads and code annotations length, and its performance is relatively stable and fluctuates little. For dimension and batch size, the model is slightly sensitive and the search performance improves slowly as the dimension increases. It is interesting to note that the ICA-CS model is very sensitive to the learning rate and the β-value, and there are significant peaks in these two hyperparameters. Specifically, the model obtains the best results when the learning rate is set to 1e-4, while the model achieves the best search performance when the β value is set to 0.35. This suggests that choosing appropriate values when adjusting the learning rate and β-value of the model can significantly improve the performance of the model.

6 Related Work

6.1 Code Search

Code search has been a popular field since the development of software engineering. Early works were mainly based on Information Retrieval (IR) techniques and Natural Language Processing (NLP) techniques, focusing on textual features of source code. For example, Lu et al. used some synonyms generated from WordNet to extend the query to improve the query to improve the hit rate [17]. McMillan et al. proposed.

Portfolio [15], which first computes the pairwise similarity between a query and a set of functions, and then uses a propagation activation algorithm to propagate the similarity scores through a pre-computed call graph. Linstead et al. proposed a code retrieval tool based on the Lucene implementation of the code search tool Sourcerer [36], which combines the textual content of a program with structural information to extract fine-grained structural information from the source code. Lv et al. proposed CodeHow [14], a

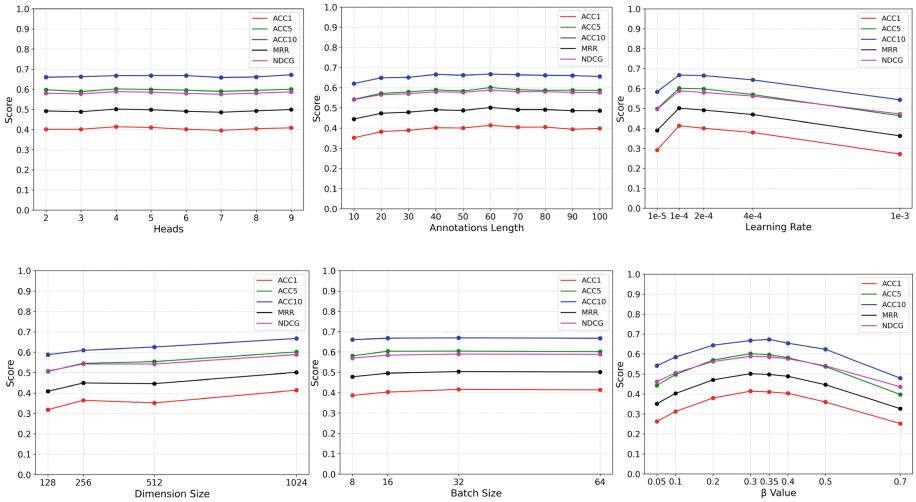

Fig. 3. Results of Sensitivity Experiments

code search tool that combines extended Boolean models with API matching. Although these works were able to produce some accurate results, code search engines based on IR techniques mainly focus on syntactic features and lack attention to semantic features, so the semantic gap between code and query is not well addressed.

With the development of deep learning technology, more and more works have started to apply deep learning to code search. For example, DeepCS [20] introduced by Gu et al. extracts method names, API sequences, and code token sequences as code features, and embeds these features information and queries into a shared space in order to retrieve code snippets through query vectors. Shuai et al. proposed a convolutional neural network (CNN)-based code search model CARLCS-CNN [4], which computes the common attention representation of code queries to improve the performance of code search. Cambronero et al. proposed a simple model UNIF [2] that extends NCS by joint deep learning to design two embedding networks to further tune the code and query embeddings. Husain et al. released the CodeSearchNet [25] challenge, which collects and makes public a dataset that explores some basic encoders for code search. Wan et al. proposed MMAN [5], which utilizes data from multiple modalities (code token sequences, ASTs, and CFGs) to better represent code with good results. Zeng et al. proposed DEGraphCS [1], which integrates code tokens, data flow, and control flow into variable-based graphs, which allows for a more accurate representation of the semantics of the code.

Overall, with the application and development of deep learning techniques, significant progress has been made in the field of code search, and researchers continue to explore new models and methods to improve the accuracy and efficiency of code search.

6.2 Attention Mechanism

Attention mechanism is a technique that simulates the human reading and listening process, which allows the model to learn to pay attention to key parts of the data, and is widely used in NLP and image description generation, and many experimental results have shown that attention mechanism has excellent performance in learning semantics [37–39]. In the field of code search, Cambronero et al. proposed UNIF [2] using unsupervised techniques and attention mechanisms with good results. Ueda et al. proposed a request estimation method using LSTM and four attention mechanisms to represent sentences from multiple perspectives and obtained excellent results [40].

On the basis of attention, Vaswani et al. proposed a self-attention mechanism, as well as a Transformer model based entirely on a self-attention network [32], which achieved significant success in a variety of NLP tasks, thus leading to a trend of using self-attention in the field of Artificial Intelligence. Zhang et al. proposed a self-attentive generative adversarial network which can generate features by using all the feature location cues to generate details [41]. Fang et al. introduced SANCS [30], which introduced self-attention to code search with remarkable success. Liu et al. proposed GraphSearchNet [21], which employs a bi-directional GGNN to capture the local structural information and employs multi-attention global dependencies.

Meanwhile, researchers also proposed a joint attention mechanism for learning interactive semantic information from two input data. Zhang et al. proposed a new joint attention-based network to capture the correlation between aspects and contexts, and the results showed good performance [42]. Xu et al. introduced TabCS [43] in the field of code search applying the traditional attention mechanism to extract the semantics of codes and queries, and applying a joint attention mechanism to address the semantic gap between codes and queries.

These studies demonstrate the important role of attention mechanisms in code search and provide powerful tools and methods to improve the accuracy and efficiency of code search.

7 Conclusion and Future Work

In this paper, a model named ICA-CS is proposed for code search task. The model introduces code annotations as additional features of the code in addition to considering method names, API sequences and code token sequences. For feature extraction, the ICA-CS model employs a bidirectional LSTM and a multi-head attention mechanism to obtain a more comprehensive and accurate representation of the code. In addition, the model maps code vectors and query vectors to a common vector space using joint embedding for code search. The experimental results show that compared to the state-of-the-art models DeepCS, SAN-CS, CARLCS-CNN, and SelfAtt, the proposed model ICA-CS achieves a significant improvement in the evaluation metrics MRR, which are 48.96%, 17.11%, 41.01% and 13.07%, respectively.

Future work will continue to extend the experiments of ICA-CS model on different language datasets to verify its effectiveness on different code styles and semantics. Meanwhile, considering the rich structural information embedded in code annotations,

further research directions will involve the introduction of graph neural network techniques to better handle the code search task and improve the performance of the model on structural information such as code annotations. These explorations will further enhance the performance and efficiency in the field of code search.

References

1. Zeng, C., et al.: deGraphCS: embedding variable-based flow graph for neural code search. ACM Trans. Softw. Eng. Methodol. **32**, 34 (2023)
2. Cambronero, J., Li, H., Kim, S., Sen, K., Chandra, S.: 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, pp. 964–974 (2019)
3. Chen, Q., Zhou, M.: A neural framework for retrieval and summarization of source code. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 826–831 , numpages = 6 Association for Computing Machinery (2018)
4. Shuai, J., et al.: Improving code search with co-attentive representation learning. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 196–207 (2020)
5. Wan, Y., et al.: Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 13–25 IEEE (2019)
6. Ling, X., et al.: Deep graph matching and searching for semantic code retrieval. ACM Trans. Knowl. Discov. Data **15**, 88 (2021)
7. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems 1589–1598 Association for Computing Machinery, Boston, MA, USA (2009)
8. Robillard, M.P.: What makes APIs hard to learn? Answers from developers. IEEE Softw. **26**, 27–34 (2009)
9. GitHub. The 2022 State of the Octoverse. https://octoverse.github.com/. Accessed 15 Mar 2023
10. Rahman, M.M. et al.: Evaluating how developers use general-purpose web-search for code retrieval. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 465–475. Association for Computing Machinery, Gothenburg, Sweden (2018)
11. Grazia, L.D., Pradel, M.: Code search: a survey of techniques for finding code. ACM Comput. Surv. **55**, 220 (2023)
12. Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The vocabulary problem in human-system communication. Commun. ACM **30**, 964–971 (1987)
13. Kevic, K., Fritz, T.: Automatic search term identification for change tasks. In: Companion Proceedings of the 36th International Conference on Software Engineering 468–471. Association for Computing Machinery, Hyderabad, India (2014)
14. Lv, F., et al.: 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), pp. 260–270 (2015)
15. McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 111–120. Association for Computing Machinery, Waikiki, Honolulu, HI, USA (2011)

16. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton (2006)
17. Meili, L., Sun, X., Wang, S., Lo, D., Yucong, D.: Query expansion via WordNet for effective code search. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 545–549 (2015)
18. Hu, F., et al.: Revisiting code search in a two-stage paradigm. In: Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining 994–1002 Association for Computing Machinery, Singapore, Singapore (2023)
19. Deng, Z., et al.: Fine-grained co-attentive representation learning for semantic code search. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 396–407 (2022)
20. Gu, X., Zhang, H., Kim, S.: Deep code search. In: Proceedings of the 40th International Conference on Software Engineering, pp. 933–944 (2018)
21. Liu, S., et al.: GraphSearchNet: Enhancing GNNs via capturing global dependency for semantic code search. IEEE Trans. Softw. Eng. **49**, 1–16 (2023)
22. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52. Association for Computing Machinery, Antwerp, Belgium (2010)
23. Chen, X., Yu, C., Yang, G., et al.: Bash code comment generation method based on dual information retrieval. J. Softw. **34**(03), 1310–1329 (2023)
24. Song, Q.W.: Research on Code Search Technology Based on Features of Code and Comment. Southeast University, Nanjing (2020)
25. Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 (2019)
26. Xu, R., Xiong, C., Chen, W. & Corso, J.J. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, pp. 2346–2352 AAAI Press, Austin, Texas (2015)
27. Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3128–3137 (2015)
28. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems, vol. 2, pp. 3111–3119. Curran Associates Inc., Lake Tahoe, Nevada (2013)
29. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
30. Fang, S., Tan, Y.-S., Zhang, T., Liu, Y.: Self-attention networks for code search. Inf. Softw. Technol. **134**, 106542 (2021)
31. Sak, H., Senior, A., Beaufays, F.: Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. arXiv:1402.1128 (2014)
32. Vaswani, A., et al.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 6000–6010. Curran Associates Inc., Long Beach, California, USA (2017)
33. Collobert, R., et al.: Natural language processing (Almost) from Scratch. J. Mach. Learn. Res. **12**, 2493–2537 (2011)
34. Frome, A., et al.: DeViSE: a deep visual-semantic embedding model. In: Proceedings of the 26th International Conference on Neural Information Processing Systems, vol. 2, pp. 2121–2129 Curran Associates Inc., Lake Tahoe, Nevada (2013)

35. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805 (2018)
36. Linstead, E., et al.: Sourcerer: mining and searching internet-scale software repositories. Data Min. Knowl. Disc. **18**, 300–336 (2009)
37. Liu, M., Yin, H.: Cross attention network for semantic segmentation. In: 2019 IEEE International Conference on Image Processing (ICIP), pp. 2434–2438 (2019)
38. Bai, X.: Text classification based on LSTM and attention. In: 2018 Thirteenth International Conference on Digital Information Management (ICDIM), pp. 29–32 (2018)
39. Yadav, S., Rai, A.: Frequency and temporal convolutional attention for text-independent speaker recognition. In: ICASSP 2020–2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6794–6798 (2020)
40. Ueda, T., Okada, M., Mori, N., Hashimoto, K.: A method to estimate request sentences using LSTM with self-attention mechanism. In: 2019 8th International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 7–10 (2019)
41. Zhang, H., Goodfellow, I., Metaxas, D., Odena, A.: Self-attention generative adversarial networks. arXiv:1805.08318 (2018)
42. Zhang, P., Zhu, H., Xiong, T., Yang, Y.: 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), pp. 6725–6729 (2019)
43. Xu, L., et al.: 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), pp. 342–353 (2021)