OCoR: An Overlapping-Aware Code Retriever

Qihao Zhu
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Zhuqh@pku.edu.cn

Zeyu Sun
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
szy_@pku.edu.cn

Xiran Liang
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
liangxiran11@163.com

Yingfei Xiong*
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
xiongyf@pku.edu.cn

Lu Zhang
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
zhanglucs@pku.edu.cn

ABSTRACT

Code retrieval helps developers reuse code snippets in the open-source projects. Given a natural language description, code retrieval aims to search for the most relevant code relevant among a set of code snippets. Existing state-of-the-art approaches apply neural networks to code retrieval. However, these approaches still fail to capture an important feature: overlaps. The overlaps between different names used by different people indicate that two different names may be potentially related (e.g., "message" and "msg"), and the overlaps between identifiers in code and words in natural language descriptions indicate that the code snippet and the description may potentially be related.

To address this problem, we propose a novel neural architecture named OCoR[1], where we introduce two specifically-designed components to capture overlaps: the first embeds names by characters to capture the overlaps between names, and the second introduces a novel overlap matrix to represent the degrees of overlaps between each natural language word and each identifier.

The evaluation was conducted on two established datasets. The experimental results show that OCoR significantly outperforms the existing state-of-the-art approaches and achieves 13.1% to 22.3% improvements. Moreover, we also conducted several in-depth experiments to help understand the performance of the different components in OCoR.

*Corresponding author
[1]OCoR, is short for An Overlapping-Aware Code Retriever.

ASE'20, September 21–25, 2020, Melbourne, Australia

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Information systems** → **Novelty in information retrieval**; • **Software and its engineering**; • **Computing methodologies** → *Information extraction*; *Neural networks*;

KEYWORDS

Code Retrieval; Neural Network; Overlap

ACM Reference Format:
Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An Overlapping-Aware Code Retriever. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20), September 21–25, 2020, Melbourne, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Code retrieval is an important software engineering problem, which aims to retrieve the most related code snippet among a set of code snippets by a given natural language description. An effective code retriever helps developers reuse the code snippets from the internet. For example, if a SQL programmer gives an instruction "get all the data in table A", a code retriever will help the programmer to search from the large scale of code on the internet and find the target code "select * from A".

With the development of deep learning and the collection of large scale labeled datasets, neural networks have been widely used for various areas [11, 18, 28, 35, 38]. For the task of retrieval, various approaches have been proposed [8, 11, 18, 19, 40, 42], by using neural networks. These approaches mostly embed the question and the answer into a high-dimensional vector space and try to find the most similar one between the vectors of questions and the vectors of answers (e.g., using cosine similarity). When it is applied to code retrieval, it takes the natural language description as the question and the target code as the answer [11, 42].

However, these retrieval approaches fail to effectively handle overlaps, which are important in code retrieval. On one hand, different people may use different names to describe the similar meanings, either in code or in natural languages, and such names often have overlapped substrings. For example, "Sort" and "QuickSort" has the overlapped substring "Sort". On the other hand, identifiers in

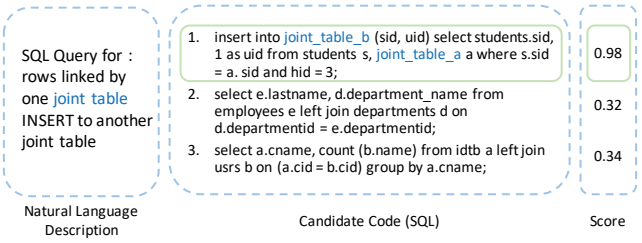
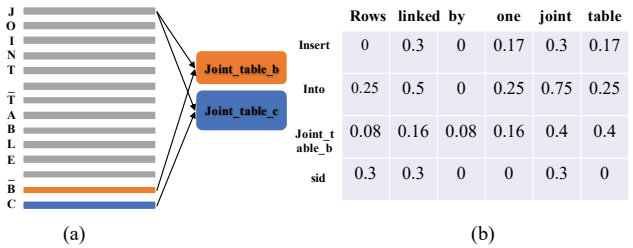


Figure 1: An Example from the StaQC dataset. The code retrieval in our approach ranks candidate code snippets with the scores given by the model.



	Rows	linked	by	one	joint	table
Insert	0	0.3	0	0.17	0.3	0.17
Into	0.25	0.5	0	0.25	0.75	0.25
Joint_table_b	0.08	0.16	0.08	0.16	0.4	0.4
sid	0.3	0.3	0	0	0.3	0

Figure 2: Examples of the computation of the character-level embedding and the overlap matrix.

code are often related to words in the natural language description. Though they may not be fully equal, overlapped sub-strings often exist. For example, in Figure 1 the identifier "joint_table_b" is related to the words "joint" and "table". As far as we are aware, no existing neural architecture is specifically designed for handling overlaps.

To address these problems, we propose a novel neural architecture, OCoR, a code retriever based on the overlap features. We represent each word by combining the representations of the characters within it, namely using the character-level embedding to capture the overlap between the names used by different programmers. Furthermore, we introduce a novel overlap matrix to represent the degrees of overlaps between each word in the natural language description and each identifier in code. Finally, we combine different code retrieval approaches by ensemble to enhance our model.

The experiment was conducted on several established datasets for SQL and C# code retrieval, following Iyer et al., Yao et al. [22, 42]. The experimental results show that our model significantly outperforms existing approaches by 13.1% to 22.3% improvements and achieve the best performance on all the datasets. To better understand our model, we also conducted the experiments focusing on the effectiveness of the components, and the results show that each component contributes to the overall performance.

To summarize, this paper makes the following contributions:

- We propose a novel neural architecture, OCoR, for code retrieval. OCoR uses two novel techniques, namely character-level embedding and the overlap matrix, to capture the overlaps between identifiers in code and words in natural language descriptions.
- We conducted extensive experiments to evaluate the effectiveness of our approach and the components in our approach. The results show that our approach significantly outperforms existing approaches by 13.1% to 22.3% improvements and all components in our approach are effective.

2 MOTIVATION

As mentioned in the introduction, there are two types of overlaps in the code retrieval task. The first type of overlap is that different people may use different names to describe the similar meanings (e.g., the words in natural language and the identifiers in code). For example, "joint_table_a" and "joint_table_b" in the first SQL query in Figure 1 could also be named as "joint_table_1" and "joint_table_2". If a neural network is trained over the code in Figure 1, it is difficult for it to know that the identifiers "joint_table_1" and "joint_table_2" are also related to the same query. To address this challenge, existing approaches [42] for code retrieval replacing variable names and raw strings with the variable types and numbers (e.g., rename the first table variable "joint_table_b" in a SQL with "Table_1"). In this way, the neural network is forced to ignore the identifier names but uses the structure of the code and the identifier types. However, the name of the identifier potentially carries useful information for the code retrieval, and ignoring them is likely to lower the performance. To solve this problem, we propose character-level embedding to encode the names. The character-level embedding first encodes each character within each name via one-hot encoding and combine these relative vectors by a convolutional layer. Figure 2(a) shows the computation of character-level embedding of "joint_table_b" and "joint_table_c". As shown, the combined vectors of these two identifiers are almost computed from the same vectors except the vector of the last character. Thus, the final embedding of these identifiers is closed to each other in the high-dimensional space.

The second type of overlap is that identifiers in code are often related to some words in natural language description. In a general perspective, this type of overlap is the overlap between question and answer, and is often considered in existing information retrieval approaches. These approaches measure the number of the exactly matched tokens between the questions and the answers. However, in the code retrieval task, identifiers in code and words in natural language descriptions are often not fully equal. For example, as shown in Figure 1, the identifier "joint_table_b" is not fully equal to any word in the natural language description, but it is related to two words "joint" and "table". To address this problem, we not only consider exactly matched words but also measure the degree of overlaps between partially matched words. We design a representation, named overlap matrix, to represent the degree of overlap between each word in natural language description and each identifier in code. In this matrix, each row represents a word in the natural language description, while each column represents an identifier in code. Each cell is the degree of overlap between the word and the identifier. The degree of overlap can be measured by different metrics, and in this paper we use the longest common sub-string, the proportion of the longest consecutive sub-string p that appears in both the natural languages word and the code identifier. Figure 2(b) shows a partial overlap matrix of the question-code pair in Figure 1. We can see that though identifiers "joint_table_b" and

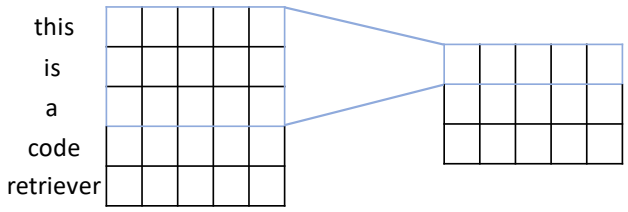


Figure 3: An Example of the computation convolutional layer.

word "joint" are not exact match, their degree of overlap is still higher than most other pairs. Finally, our model takes the overlap matrix as input, utilizing the detailed overlap information for identifying the most related code snippet.

3 BACKGROUND

3.1 Convolutional Layer

The Convolutional layer, which is the main building layer of a convolutional neural network (CNN) [26], has been widely used in various areas [8, 18, 24, 26]. This layer can be regarded as a regularized version of a fully-connected layer. The fully-connected layer usually consists of several neurons, and each neuron in one layer is connected to all neurons in the next layer. Different from such a fully-connected layer, the connection in the convolutional layer is connected from each neuron in one layer to several corresponding neurons in the next layer. Such connections depend on pre-defined convolutional kernels.

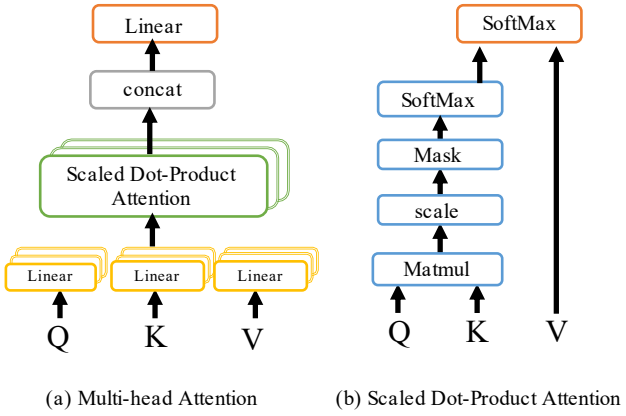
In natural language processing, the convolutional layer is used to extract the features in the contents. Given an input vector, which represents the words in natural language, the convolutional layer uses kernels to extract the features in each vector and its neighboring vectors and outputs a new vector. For example in Figure 3, for an input vector of a sentence "this is a code retriever" with kernel size 3, the layer outputs a new vector of "is" by a weighted summation of the vectors of "is" with its neighbors "this" and "a". This helps capture the features of the contents in the input natural language description and code of OCoR. Thus, we apply it in our approach. The details of using this layer will be introduced in Section 4.

3.2 Attention

In the basic encoder-decoder framework [37][2], the model always suffers from the the long-dependency problem [5] with long sequences. To alleviate this problem, Bahdanau et al. [4] proposed the attention mechanism, which aims to let the neural model inspect the relevance between each pair of tokens in two long sequences.

Recently, to better alleviate the long-dependency problem, Vaswani et al. [38] proposed a widely used attention mechanism called Multi-Head Attention. The overview of such mechanism is shown in

[2]The encoder-decoder framework, which is also known as sequence-to-sequence framework, is a widely used approach in neural machine translation. In this framework, the neural network is divided into two parts: encoder, decoder. The encoder encodes the sequence into a vector and the decoder decodes the vector to a sequence in target language.



(a) Multi-head Attention (b) Scaled Dot-Product Attention

Figure 4: The detail of Multi-Head Attention.

Figure 4(a). This mechanism takes three vectors (mostly the representation of the words in the input sequence) as inputs and maps a query vector and a set of key-value vector pairs to the output. The main computation of this attention is called Dot-Product Attention layers, where the input of each layer consists of query vectors (Q), key vectors (K) and value vectors (V) as shown in Figure 4(b). The weights of the value vectors are calculated by the query vectors and the corresponding key vectors. Finally, the output is computed as a weighted sum of the values, where the query determines which values to focus on.

Self-attention is an attention mechanism for a single sequence to extract the complex comprehension of itself (formally, $Q = K = V$). This technique is often used to capture the long dependency information in sequences and has good performance in various tasks [20, 38, 41]. The detailed computation of the attention mechanism will be introduced in the following section.

4 PROPOSED MODEL

4.1 Problem Definition

We follow the existing studies [11, 42] and use the same definition for code retrieval. Given natural language description Q and a set of candidate code snippets C, our task is to retrieve a relevant code snippet $C^r \in C$ that specified by Q.

As shown in Figure 1, to retrieve a code snippet, we first compute the relevance score between each code snippet $c \in C$ and the input natural language description Q. Then, we rank the code snippets in the set of candidate code snippets C. Finally, the code C^r with the highest score is selected as the output of our approach, which is computed as

$$C^r = \arg\max_{c \in C} R(Q, c) \tag{1}$$

where R denotes the computation of the relevance score. In our approach, the relevance score is a real number between 0 and 1.

4.2 Overview

Figure 5 shows the overview of OCoR. We adopt the traditional overall architecture used in information retrieval Fu et al. [8], where we encode the question (the natural language description) and the

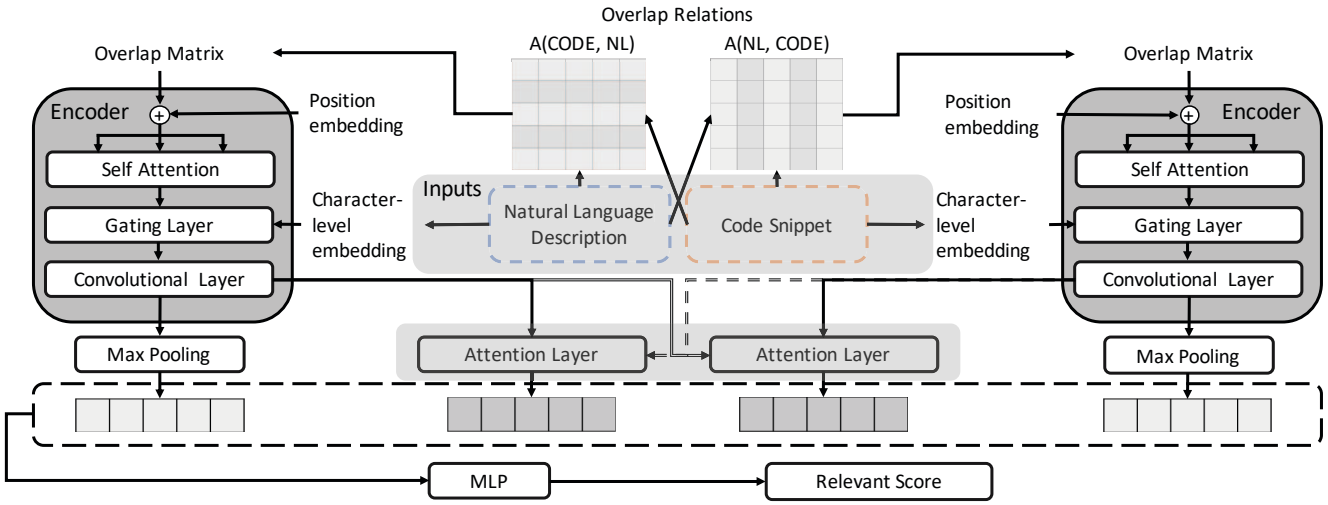


Figure 5: The overview of the OCoR. The $A(CODE, NL)$ and $A(NL, CODE)$ are the Overlap Matrices.

answer (the code snippet) respectively and combine the outputs via attention layers for further predicting the target relevant score. Based on this architecture, we design two encoders with the same structure for the question and the answer. Each encoder takes the overlap matrix and the question / answer as input and turns this input into a set of vectors.

Furthermore, as the evaluation will show later, our model complements existing approaches on code retrieval. To achieve even better performance, we use an an additional ensemble component to combine previous code retrieval models with OCoR.

In the rest of this section, we will describe the components of our architecture one by one. Besides,

4.3 Input of Model

The inputs of OCoR are divided into three parts: 1) the natural language description; 2) the code snippet; 3) the overlaps between the natural language description and the candidate code snippet, where the last one is computed from the first two.

4.3.1 Preprocessing. For the first two parts, we first process these inputs to make them suitable to be fed to the neural network. For the input natural language description, we first tokenize this input by the tool in the NLTK toolkit [27] and convert its characters within the tokens to lowercase. As for the input code snippet, we keep the original variable names and the raw strings, to keep the semantic information in the names. Then, we tokenize the code and also convert the characters within the names to lowercase. Thus, we get the preprocessed inputs for the neural network.

4.3.2 Overlap Matrix. As mentioned before, we use the overlap matrix to represent the degrees of overlaps between natural language words and code identifiers. The overlap matrix is a real-valued matrix $A(T_1, T_2) \in \mathbb{R}^{L^{(T_1)} \times L^{(T_2)}}$, which contains the overlap scores between a token sequence T_1 and another token sequence T_2. Each cell $A_{ij}(T_1, T_2)$ in this matrix denotes the overlap score between the i-th token $T_1(i)$ in the T_1 sequence and the j-th token $T_2(j)$ in the T_2. Such score in OCoR is computed by

$$A_{ij}(T_1, T_2) = \text{len}(S(T_1(i), T_2(j))/\text{len}(T_2(j)) \quad (2)$$

where $\text{len}(T_2(i))$ denotes the length of the word $T_1(i)$, $S(T_1(i), T_2(j)$ denotes the longest common sub-string of T_1, T_2. In particular, the computation of the overlap matrix A is not commutative, i.e., $A(T_1, T_2) \neq A(T_2, T_1)$. In our approach, we consider both the overlap score between the natural language description NL and the code $CODE$ as well as the overlap score between the code and the natural language description, namely both $A(\boldsymbol{n}, \boldsymbol{c})$ and $A(\boldsymbol{c}, \boldsymbol{n})$ respectively. These two metrics are further fed to the encoder layer to extract features for code retrieval.

4.4 Encoder

In OCoR, there are two encoders for both the natural language description and the code. Each encoder takes the overlap matrix and the natural language description / code as inputs. These inputs are encoded into vectors in a high-dimensional space for further similarity computations.

To better encode the input information, inspired by Vaswani et al. [38], we design the encoder with a stack of N mechanisms. Each mechanism contains three sub-layers: 1) a self attention layer; 2) a gating layer; 3) a convolutional layer. After each mechanism, the ResNet [13][3] and the layer normalization [3][4] are used. For the first mechanism, it takes the overlap matrix, namely $A(NL, CODE)$ / $A(CODE, NL)$, as input and further combines the natural language description / the code. For the rest of $N-1$ mechanisms, they take the output of the previous mechanism as input and also combine the features of hidden layers in the natural language description / the code. We will first describe how we feed the overlap matrix to the first mechanism.

[3]ResNet is residual learning framework to ease the training of networks.
[4]Layer normalization normalizes the values of the neurons into a suitable distribution, which eases the training.

Input Overlap Matrix. The Overlap Matrix[5] for our approach is a real-valued matrix, where each cell denotes the overlap scores between the natural language words and the identifiers in code. To take this matrix as input, we first reduce the matrix $A(NL, CODE)$ into an overlap vector $\boldsymbol{a}(NL)$, where the i-th element $\boldsymbol{a}_i(NL)$ in this vector denotes a new overlap score computed from the i-th row in the $A(NL, CODE)$ (the cells representing the i-th token in the natural language and each identifier in the code) via max-pooling. Max-pooling [24] has been shown to be an effective way to reduce the matrix into a vector in various areas [10, 42, 44]. Following Max-pooling, we select the maximum value for each column as the element of $\boldsymbol{a}(NL)$, which is computed by

$$\boldsymbol{a}_i(\boldsymbol{n}) = \max_{j=1}^{\text{len}(\boldsymbol{c})} A_{ij}(\boldsymbol{n}, \boldsymbol{c}) \tag{3}$$

This vector contains the maximum overlap scores. For example, the fixed-size vector of the overlap matrix in Figure 2(b) is $[0.3, 0.75, 0.4, 0.3]$. To ease neural processing of the overlap scores, we further embed the scores with one-hot encoding. Please note that the scores are the real-values between 0 and 1. We partition the scores with the interval 0.01 and use a one-hot vector of length 100 to encode the scores.

4.4.1 Self Attention Layer. The first sub-layer in the encoder mechanism is the self attention layer. As we know, sequential information is important in both the natural language description and the code. To handle such information effectively, we apply the self attention mechanism, which is proposed by Vaswani et al. [38] and has shown to be an effective way to encode this information [38, 41], as the first sub-layer in the encoder.

The self attention layer takes the previous output vectors $\boldsymbol{o}_1, \boldsymbol{o}_2, \cdots \boldsymbol{o}_L$ as input, where L denotes the length of the input natural language description / code. This layer consists of two parts: 1) the position embedding layer; 2) multi-head attention layer.

Position Embedding Layer. A position embedding layer is a standard layer in transformer architecture [38] to provide the indexes of the words in the input matrix. For example, the layer should know the word "joint" is the 5-th token in the natural language description in Figure 1. If we directly use an attention layer for the input vectors, the position information will not be considered, and this is why we need the position embedding layer.

In this layer, the vector of the i-th position is represented as a real-valued vector, which is computed by

$$\begin{aligned} p_{(i,2j)} &= \sin(pos/(10000^{2j/d})) \\ p_{(i,2j+1)} &= \cos(pos/(10000^{2j/d})) \end{aligned} \tag{4}$$

where $pos = i + step$, j denotes the element of the input vector and $step$ denotes the embedding size. After we get the vector of each position, we directly add this vector to the corresponding input vector, where $\boldsymbol{e}_i = \boldsymbol{o}_i + \boldsymbol{p}_i$.

Multi-head Attention Layer. The second part of the self attention layer is the multi-head[6] attention layer. As introduced in the background section, an attention mechanism maps a query, a key and a value to an output. In this layer, the query, the key, the value, and output are all vectors.

Following the definition of Vaswani et al. [38], we divide the multi-head mechanism into H heads. Each head is an attention layer, which maps the query Q, the key K and the value V to an output, namely the output of each head $head$. The computation of the s-th head is represented as

$$head_s = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{5}$$

where $d_k = d/H$ denotes the length of each extracted feature vector and Q, K and V are computed by a fully-connected layer from Q, K, V. In the encoder, the vectors Q, K and V are all the outputs of the position embedding layer $\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_L$. The outputs of these heads are further jointed together with a fully-connected layer, which is computed by

$$Att = [head_1; \cdots; head_H] \cdot W_h \tag{6}$$

where W_h denotes the weights in fully-connected layer and the output vectors $Att = [\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_L]$ are the high-level vectors, which combine the sequential information and the original information together. However, these vectors still fail to encode the semantic information of each word effectively at least in the first mechanism in the encoder. Thus, we will then describe how we address this issue via a gating layer .

4.4.2 Gating Layer. The second sub-layer in the encoder mechanism is a gating layer. This layer takes the outputs of the previous layer and the input natural language description / code as input. existing state-of-the-art approaches [42] use the word2vec [30] mechanism to utilize the semantic information of the input. However, it may be not suitable for code retrieval, where the similar identifiers can be named differently by different programmers but with overlapped characters (e.g., it may have a very similar meaning for two words "dataId" and "data_id"), which may lead to a large number of vocabulary for neural networks to learn. To address this issue, we propose to use the character-level semantics to catch the overlaps between identifiers in code retrieval. We combine the outputs of the previous layer with the character-level embedding approach for each word.

Character Embedding. To implement the character embedding, we first pad each token (both the word in the natural language description and the identifier in code) to a fixed length CL with a special character. In particular, if the length of the token is more than CL, we truncate the end of this token and make it a CL-length token. Then, we represent each character in the token as a real-value vector, namely *embedding*. As we know, a token consists of several characters. To catch the semantic information of each token, we adopt a set of convolutional layers to integrate the vectors of the characters within the token. The extracted semantic vector for the i-th token $\boldsymbol{t}_i$ is computed by

$$\boldsymbol{t}_{(i,n)} = W_{(c,n)}[\boldsymbol{t}_{(1,n-1)}; \boldsymbol{t}_{(2,n-1)}; \cdots; t_{(CL,n-1)}] \tag{7}$$

where W_c are the convolutional weights and n denotes the n-th layer of the convolutional layers. In particular, $\boldsymbol{t}_{(k,0)} = \boldsymbol{c}_k$, where $\boldsymbol{c}_k$ denotes the character embedding vector of the k-th character within the i-th token. In our approach, we have three convolutional layers

[5] We take $A(NL, CODE)$ as example in this section.

[6] The multi-head mechanism consists of several heads, each of which is an attention layer, separately. The outputs of these heads are further jointed together by a fully-connected layer.

for this character embedding layer. For the first two convolutional layers, we use the zero padding and the sizes of the convolutional kernel are set to 3 and 5, respectively.

Gating Mechanism. To incorporate the semantic information of each token with the previous outputs, we use a mechanism named Gating Mechanism [36]. This mechanism incorporates an input semantic vector $\boldsymbol{t}_i$ with a given control vector[7] with the multi-head mechanism. In this paper, we use the previous output vectors, namely $\boldsymbol{a}_i$, as the control vector. The computation of the gating layer in our model can be represented as

$$\alpha_i^o = exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^o)/\sqrt{d} \tag{8}$$

$$\alpha_i^c = exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^c)/\sqrt{d} \tag{9}$$

$$\boldsymbol{h}_i = (\alpha_i^o \cdot \boldsymbol{v}_i^o + \alpha_i^c \cdot \boldsymbol{v}_i^c)/(\alpha_i^o + \alpha_i^c) \tag{10}$$

$$head_s = [\boldsymbol{h}_i; \cdots; \boldsymbol{h}_H] \tag{11}$$

where $\boldsymbol{q}_i, \boldsymbol{k}_i^o, \boldsymbol{v}_i^o$ are all computed by a fully-connected layer over the control vector $\boldsymbol{a}_i$; $\boldsymbol{k}_i^c, \boldsymbol{v}_i^c$ are computed by another fully-connected layer over the semantic vector $\boldsymbol{t}_i$. After this computation, we enhance the vectors with the semantic information, and the extracted new features are denoted as $\boldsymbol{c}_1^{(com)}, \boldsymbol{c}_2^{(com)}, \cdots, \boldsymbol{c}_L^{(com)}$.

4.4.3 Convolutional Layer. The final sub-layer in the encoder mechanism is a set of convolutional layers. We follow the design of the encoder proposed by Vaswani et al. [38] and adopt a set of convolutional layers to extract the local features around each token. The computation of the convolution layer can be represented as

$$\boldsymbol{y}_i^l = W_l[\boldsymbol{y}_{i-w}^{l-1}; \cdots; \boldsymbol{y}_{i+w}^{l-1}] \tag{12}$$

where l denotes the l-th convolutional layer in the set, W_l are the convolutional weights, $w = (k-1)/2$ and k denotes the window size. In particular, $\boldsymbol{y}_i^{l-1}$ is the output of the previous gating layer $\boldsymbol{c}_i^{(com)}$. We use two convolutional layers in this sub-layer and add the activation function *GELU* [14] between these convolutional sub-layers. In particular, we use the zero padding in these layers.

4.5 Max Pooling

After all these N mechanisms in the encoder, we adopt an additional convolutional layer as Equation 12, which is padded with a special vector during convolution. Then, we get the final features of each token. The features denote the high-level information of the input natural language description / code. However, these features have the same shape as the input. To facilitate further prediction for code retrieval, we need to aggregate such features into a fixed-size vector, which is regardless of the input size.

Max pooling has shown the power in aggregating features, thus, we apply the max pooling approach over the outputs of the encoder and extract the fixed-size vector for each encoder.

[7] The control vector is the special vectors given in our approach. This vector decides the weights of different vectors.

4.6 Attention Layer

The encoders encode the information of the input natural language description and the input code separately. However, it still lacks the relations between two inputs even if we have used the prior knowledge of the overlaps. To help the neural network learn such relations between the two inputs, we adopt two attention layers after the encoder.

As described in the previous section, the outputs of the encoder combine the overlap information with semantic information (character embedding). Thus, we also apply the multi-head attention layer to the outputs of two encoders to extract the relations. As for the description and the code, we design two separate attention layers for them. The computation of the attention mechanism is similar to the "self attention" with different inputs. One layer treats the encoding of the description as query (Q) and the other treats the encoding of the code as query (Q). This design allows the model to extract the weighted sum of the outputs of two encoders based on each other. After the attention, two convolutional layers and a max pooling layer are followed to integrate the features.

4.7 Prediction

After all the computation of the attention layer, we concatenate all features. They are further fed to a two-layer perceptron followed by a *softmax* activation. The output of these computation is the classification probability of two classes. The first class denotes that the input natural language description and the input code is related, whereas the second class denotes that the input natural language description and the input code is not related. In our approach, the predicted classification probability of the first class is the relevance score between the input natural language description and the code, where the relevance score is computed by

$$R(Q, c) = \frac{\exp\{h_1\}}{\sum_{j=1}^{2} \exp\{h_j\}} \tag{13}$$

where h_i is the input logit of *softmax*.

4.8 Training

Our model is trained by minimizing cross-entropy loss against the ground truth. Specifically, for each training data $< Q, C, A >$ where Q is the description, C is the code, A denotes the ground truth class. The cross-entropy loss is computed by

$$Loss(\theta) = -\sum_{i=1}^{2} g(i) * \log \theta(i) \tag{14}$$

where g denotes the ground truth class, θ is the classification result predicted by the neural network.

4.9 Model Combination

On the basis of our basic model introduced above, we also consider an additional method that combines different models on code retrieval task together by ensemble. Inspired by Yao et al. [42], where the proposed CoaCor combines code retrieval with code annotation for better performance. We consider to combine different models by integrate the relevance scores computed by different models and output the final relevance score for OCoR. The score is computed

by a linear combination as

$$R(Q, c) = \lambda * S_1 + (1 - \lambda) * S_2 \quad (15)$$

where S_1 denotes the relevance score computed by OCoR, S_2 is the score computed by the combined model, λ is a real number between 0 and 1.

5 EXPERIMENT SETUP

In this section, we present the experimented setup. We will first introduce the research questions[8].

5.1 Research Question

Our evaluation aims to answer the following research questions:

- **RQ1 What is the performance of OCoR?**
  To answer this question, we conducted an experiment on several established datasets and compared the performance of OCoR with the existing state-of-the-art approaches.
- **RQ2 What is the contribution of each component in OCoR?**
  To answer RQ2, we start from the full model of OCoR, and in turn removed each component to understand the contribution of it. Then, we also replaced the metrics used in measuring the overlap score with longest common prefix (LCP) and word embedding based similarity to better understand the contribution of the component of the overlap matrix.
- **RQ3 Why does the model combination work?**
  In fact, the result of RQ1 will suggest that the model combination places a significant role in the overall performance. To understand why different models can be combined, we analyzed the distribution of the result predicted by different models on the SQL dataset. More specifically, we selected some examples from these datasets to show the differences of the models.

5.2 Dataset

Our experiment is based on two established benchmarks: the StaQC benchmark[43], and the C# benchmark used in Iyer et al. [22]. The StaQC benchmark contains 119,519 question-code pairs written in the SQL. These pairs are collected from Stack Overflow [31], making itself the largest-to-date in SQL domain. We followed the original train-dev-test split in StaQC, namely StaQC-train, StaQC-val ,and StaQC-test. For better evaluation , we used two additional test dataset namely "DEV" and "EVAL" for SQL, are collected by Iyer et al. [22]. These datasets contain 110 and 100 code written in SQL respectively. For every snippet, they use three different references written by humans as the additional test cases. The second benchmark contains 113,514 question-code pairs written in C# collected from StackOverflow. We split the dataset into C#-train, C#-val ,and C#-test as Iyer et al. [22, 42] The detailed statistics of these datasets are listed in Table 1.

For the StaQC benchmark, we took the training set of StaQC-train as the training set, took the DEV set as the development set, and took other three datasets, StaQC-test, StaQC-val, "EVAL", as the test set. For the C# benchmark, we also followed the same experiment settings during training and the C#-val was treated as the development set in our experiment.

Note that to test the performance of a code retrieval approach, we not only need the desirable question-code pair (the positive answer), but also other code snippets as negative answers. We call such a case containing a question and a set of code snippets as a *retrieval case*. We used the same retrieval cases used in existing works [22, 42], where the counts of these cases are show in the "Number of Cases" row in Table 1. Each retrieval case contains 1 positive code snippet and 49 negative code snippets.

5.3 Metrics

To measure the performance of our approach, we followed Yao et al. [42] and used a standard metrics called Mean Reciprocal Rank (MRR) [7] in this paper.

The MRR metrics is computed over the entire dataset $D = \{(Q_1, C_1), (Q_2, C_2), \cdots, (Q_n, C_n)\}$:

$$MRR = \frac{\sum_{i=1}^{n} 1/r_i}{|D|} \quad (16)$$

where r_i denotes the ranking of C_i in the i-th query Q_i. In this metrics, the higher value denotes the better performance of code retrieval.

5.4 Implementation Details

We implemented our approach based on Tensorflow [1]. We set the $N = 3$, which denotes that each encoder in our experiment contains a stack of 3 mechanisms. We set the embedding size for both the characters and the overlap scores to 256. All hidden sizes were set to 256 except that 1024 was used for both the first layer of the convolutional layer and the first layer in the MLP. During training, the dropout [16] was used to avoid overfitting, where the droprate was set to 0.2. Our model was optimized by Adam optimizer [25] with learning rate 0.0001. For the model combination, we set the hyper-parameter λ to 0.1. These hyper-parameters and parameters for our model were chosen based on the development set (DEV is used), which followed the existing state-of-the-art work [42]. Specifically, for each query natural language description in the training corpus, we randomly sampled 5 code snippets as the negative examples for each training epoch. In OCoR, the natural language description and the code snippet shared the same embedding weights.

5.5 Baselines

In our experiment, we used the existing state-of-art code retrieval approaches as the baselines for comparison.

- Deep Code Search (DCS) [11]. DCS jointly embeds the input code snippets and the input natural language description into a high-dimensional vector space with an RNN based neural network. In this way, a code snippet and its corresponding natural language description have similar vectors, which are then used for computing the similarity between two inputs by cosine similarity.
- CODE-NN [22]. The core component of CODE-NN is an LSTM-based RNN [17] with attention. This attention mechanism computes the probability of an natural language description, given a code snippet. For code retrieval, given an

[8]The code of our experiment is available at https://github.com/anyone546/OCoR.

Statistics	StaQC-train	StaQC-val	StaQC-test	DEV	EVAL	C#-train	C#-dev	C#-test
Number of QC-pairs	89,688	11,932	17,899	330	300	77,816	17,849	17,849
Number of Cases	-	11,900	17,850	6,600	6,000	-	17,800	17,800
Avg. tokens in description	9	9	9	10	15	12	12	12
Max. tokens in description	32	35	45	45	35		37	34
Avg. tokens in code	59	62	60	47	47	38	38	38
Max. tokens in code	3,367	2,774	2,672	291	291	290	300	310

Table 1: Statistics of the datasets we used. "Number of QC-pairs" denotes the total amount of question-code pairs in the specific dataset. "Number of Cases" is the number of the retrieve cases for evaluation.

input natural language description, CODE-NN computes the probability of the input for each code. After the computation, CODE-NN ranks the given code snippets based on the probability.

- CoaCor [42]. CoaCor used a reinforcement learning-based framework to combine code retrieval and code annotation together for enhancing the code retrieval. They also combine the code retrieval approaches together by ensemble to improve the performance. In particular, the basic model of CoaCor is denoted as QN-RL^{MRR}, whereas the combined models (the best performance) are denoted as QN-RL^{MRR} + CODE-NN.

6 RESULTS

In this section, we report the results of our experiment and answer the research questions.

6.1 Performance of OCoR (RQ1)

The results of RQ1 are presented in Table 2.

We first compare the original OCoR with the original models (Ori. in Table 2). As shown, among the three state-of-the-art models, OCoR achieves the best performance on all datasets. OCoR is higher than the existing best results, by 9.1% to 28.2% improvements.

For the model combination, we first combine OCoR with the original model QN-RL^{MRR} (denoted as OCoR + QN-RL^{MRR}). We select the state-of-the-art models proposed by Yao et al. [42] (QN-RL^{MRR} + CODE-NN) as the baselines. As shown in Table 2, our approach significantly outperforms existing state-of-the-art models by 10 points improvement on average. In particular, we achieved 13.1% to 22.3% improvements on all datasets, which shows effectiveness of our approach. Then, we combine OCoR with the model combined by Yao et al. (QN-RL^{MRR} + CODE-NN), where we also achieved the best results (OCoR + (QN-RL^{MRR} + CODE-NN)) among all datasets.

The results suggests that our approach is more effective than existing state-of-the-art models on different datasets and different program languages.

> Answer to **RQ1**: OCoR has a good performance (13.1% to 22.3% improvements) compared with the existing state-of-the-art approaches on all datasets covering two programming languages.

6.2 The contribution of each component (RQ2)

To answer RQ2, we first conducted the ablation test on the SQL dataset to figure out the contribution of each component. In this subsection, we only conducted the experiment based on the original OCoR, which aims to understand the contribution of each component more clearly.

The model in the ablation test had the same setting with the original OCoR except we removed each component in turn. The results are presented in Table 3. We first removed the input overlap scores from OCoR. To remove this, instead of using overlap matrices as the inputs of the first mechanism in the encoder, we replaced it with the input tokens in natural language description / code. Furthermore, we followed the previous joint model [6, 11] and the model was trained by minimizing the cosine similarity between the natural language description and code as the . By applying such settings, the performance was closed to the previous approach based on word2vec which shows the effect of the overlap matrix.

To better understand the contribution of the metrics for computing overlap scores (namely Equation 2), we further conducted the experiment on the same datasets but with a different metrics of overlap scores. The baseline metric compared with the original metric is word similarity based on word embedding. We first used the GloVe [32] to pretrain the word embedding vector based on the training set with BPE [34]. Then we use the cosine similarity of $\boldsymbol{w}_1, \boldsymbol{w}_2$ as the overlap score. The results are presented in Table 3. The character-level metric achieves better performance on all datasets. This result shows our metrics are more suitable than the traditional similarity matrix used in document retrieval for code retrieval.

We also replaced the original overlap metric with Longest Common Prefix (LCP). The longest common prefix of a pair of strings a and b is the longest string p which is the prefix of both strings. We use $len(p)/len(a)$, $len(p)/len(b)$ as the overlap scores of two strings, respectively. In particular, we also compute the score based on the longest common suffix, and finally select the bigger one as the overlap score. The performance of this metric is slightly lower. The ablation test also indicates that the character-level information is important to the code retrieval task.

> Answer to **RQ2**: Each component contributes to the overall performance of OCoR.

	Model	EVAL	StaQC-val	StaQC-test	C#
Ori.	DCS	0.555	0.534	0.529	0.441
	CODE-NN	0.514	0.526	0.522	0.531
	QN-RL^{MRR}	0.512	0.516	0.523	0.528
	OCoR	**0.601**	**0.647**	**0.643**	**0.682**
Com.	QN-RL^{MRR} + CODE-NN	0.571	0.575	0.576	0.629
	OCoR + QN-RL^{MRR}	0.630	0.658	0.677	0.746
	OCoR + (QN-RL^{MRR} + CODE-NN)	**0.646**	**0.665**	**0.685**	**0.764**

Table 2: The results show the MRR of code retrieval among 50 examples. In this table, we divide the existing model into two different categories. The first category (Ori.), where the original model is used, is in the 2 to 5 rows. The second category (Com.), where different models are combined together by ensemble, is in the 6 to 9 rows.

Model	EVAL	StaQC-val	StaQC-test
OCoR	**0.601**	**0.647**	**0.643**
- Overlap Score	0.420	0.545	0.538
Character-level Overlap → WordSimilarity	0.554	0.603	0.605
Overlap → LCP	0.591	0.628	0.632

Table 3: The ablation test on OCoR, where OCoR denotes the original model of our approach.

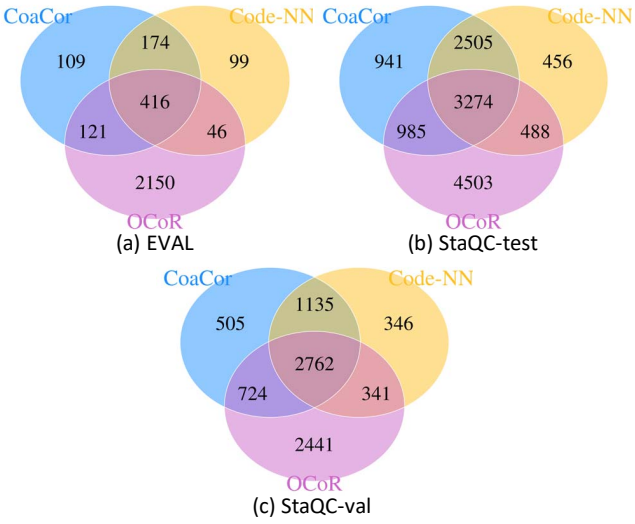


Figure 6: The overlaps of the prefect ranking of different approaches the among three datasets.

6.3 Model Combination Analysis (RQ3)

To answer RQ3, we try to figure out the reason why the model combination works. We first implemented the existing approaches, CoaCor (QN-RL^{MRR}) and CODE-NN. Then, we studied the prediction overlaps of the perfect ranking among the datasets. For a given natural language description and a set of code snippets with one positive code snippet, the "perfect ranking" in this paper means that the positive code snippet is ranked in the top-1.

The results are presented in Figure 6. As shown, on these three datasets, 16.2% perfect ranking cases on average can be solved by all three approaches, whereas 26.1%, 3.7%, 2.3% (32.3% in all) perfect ranking cases on average can only solved by the approach OCoR, CoaCor and CODE-NN respectively. Such 32.3% cases show the potential improvements on model combination, and this is why the model combination works well in our approach.

To help understand the model combination, we also conducted an additional case study. In this case study, we analyze OCoR with the existing state-of-the-art model (CoaCor, QN-RL^{MRR} + CODE-NN).

Case study. Table 4 shows three examples that are ranked perfectly by OCoR but not CoaCor. As shown, there are many overlaps between the input natural language description and the code in SQL (e.g., the word "select" and "data" in the first example, row 2,3; the word "table" and "time" in the second example). In these examples, the information of overlap scores is important, where a human can utilize this information and retrieved the target code easily, and OCoR catches such information successfully. Existing approaches like CoaCor do not utilize the information of overlap scores properly, where the CoaCor approach directly uses the token-level embedding for the neural network and replaces identifier names with numbered placeholder tokens (e.g., the SQL code in the second example is turned to "select col0 (col1) from tab1" in Table 4). Thus, OCoR has a good performance on these examples, while the CoaCor does not work well, which is the strength of OCoR.

To understand the weakness of OCoR, we also conducted another case study on examples where OCoR does not work well compared with the CoaCor. These examples are presented in Table 5. As shown, in these examples there are few overlaps between the input natural language description and the code in SQL. In such a situation, OCoR can hardly measure the overlap scores in the matrix, which makes it difficult to utilize the key information of overlap scores in OCoR. However, CoaCor, which combines the annotation generation and code retrieval together, replaces identifier names with numbered placeholders, and extracts the high-level information of these situations. This is the reason why CoaCor has a good performance on these examples. The cases show the

Type	Example
Description	select a formatted date range from values in a table column
SQL Code	select date_format (startdate, '%m') + date_format (startdate, '%d') + '-' + date_format (enddate, '%d') + ',' + date_format (startdate, '%y') from yourtable;
Description	SQL Insert multiple Values where 1 value comes from a select query
SQL Code	insert into table2 (telnumber , adress) select '12324567890' , applicatieid from applicatie where name = 'piet' ;
Description	Quick way to space fill column 256 chars SQL-Server 2012
SQL Code	select space (256) ;

Table 4: The examples that ranked perfectly by OCoR but not CoaCor.

Type	Example
Description	how to use max and top in sql query in oracle?
SQL Code	select id, item, quantity, date from (select id, item, quantity, date from your_table order by quantity desc, date desc) where rownum = 1;
Description	find 1 level deep hierarchical relationship between columns of a table for one of the top level values
SQL Code	select t2.cat_id, t2.subcat_id, t2.name from test t1 join test t2 on t1.cat_id = t2.cat_id where t1.subcat_id = 42 and t2.subcat_id <> 42 ;

Table 5: The examples that ranked perfectly by CoaCor but not OCoR.

weakness of OCoR, which does not have a good performance when the overlap scores are hard to measure. It is probably a good way to combine different approaches together and utilize the strength of each approach. Thus, we use the model combination to combine the strengths of different approaches.

> Answer to **RQ3**: The three techniques complements each other, allowing the model combinations to produce better results.

7 THREATS TO VALIDITY

Threats to internal validity. A threat to internal validity is the potential faults in the implementation of our experiments. To reduce this threat, for the performance of original models, we directly use their reported performances [42], and, for the performance of combined models, we directly used their published code [42]. Furthermore, the implementation of our model was based on a published model [36] to avoid faults in re-implementation.

Threats to external validity. Our model was evaluated on the StaQC and the C# benchmarks, which are widely used in previous code retrieval approaches [22, 42]. In these two benchmarks, all of the programs are collected from Stack Overflow, which gives a threat to external validity. However, we find that the overlap relations we used also widely exist in datasets collected from other sources such as GitHub [9]. For example, in CodeSearchChallenge Corpus [21] which is another code retrieval benchmark collected from GitHub repositories, 98.6%, 95.4%, 94.18% and 96.8% of the instances have at least one overlap for Python, Java, JavaScript, and Go, respectively, while only 89.12% in the StaQC dataset used in our experiment. Such widely existed relations show the potential value of our approach when applied to the GitHub benchmarks. Meanwhile, since our approach was only tested on SQL and Java programs collected from Stack Overflow, further studies are also needed to apply our model to other programming languages collected from GitHub.

8 RELATED WORK

Code Retrieval. Code retrieval in software development helps developers reuse the relevance code snippets among a large scale open-source projects. Early studies mostly focus on applying the information retrieval methods to code retrieval task [2, 11, 12, 15, 23, 29, 39]. With the development of deep learning, more and more works try to use neural networks to code retrieval [11, 22, 42]. Gu et al. [11] first proposed an LSTM-based RNN for code retrieval, where they encode the input natural language description and code into a vector space and measure the cosine similarity between them. Based on a code annotation work proposed by Iyer et al. [22], where they use a sequence-to-sequence model to generate the specific annotation by a given code, Yao et al. [42] proposed CoaCor for code retrieval, where they combine the code annotation approach of Iyer et al. and the code retrieval approach of Gu et al. together by a reinforcement learning framework. Different from these approaches, we focus on unitizing the overlap scores between the natural language description and code. Based on this, we proposed a novel neural architecture for code retrieval.

Overlap Information. Many works focus on using neural networks combined with overlap information in sentence pairs matching [8, 18, 19]. Hu et al. [18] first proposed to use a neural network. They adopted a stack of convolution layers to infer the relation between the question and the given answer. Qiu and Huang [33] introduced a transformation layer to use the interaction between

the question and the answer. They tried to utilize hidden units to extract the overlaps based on hidden states. Fu et al. [8] proposed a kind of overlap features and combined it with a convolutional network. Such overlap features compute the similarity between two words via whether they are the same. It cannot utilize the overlap between words (e.g., the overlap between "joint_table_a" and "table"), which is important in encoding identifiers in code. The atomic value of these overlap features cannot represent the relation between identifiers and corresponding words. Thus, we design the overlap matrix based on longest common sub-string to measure the degree of the overlap. We also adopt some special neural components for this representation.

9 CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel overlap-aware neural architecture (OCoR) for code retrieval. Our approach makes use of the overlap score between the natural language description and the code by using the overlap matrix and the character embedding.

We evaluated our approach on several datasets. The experimental results show that OCoR achieves significant improvement compared with existing state-of-the-art approaches. The further evaluation shows that each component in our approach is important.

Future work. Our approach is built mainly on the basis of overlap scores between two inputs, especially for the natural language description and code. The experimental results show that the overlap score can boost the performance of the model. It is interesting to study as the further work to try more metrics to measure the degree of the overlaps between two strings (e.g., Edit Distance and Longest Common Sub-string). Our experiment also shows the potentiality of the ensemble (namely model combination), which may also be an effective way to use such technique to combine different metrics to improve the performance. Furthermore, OCoR, can be directly applied to other programming languages (e.g., Java, Python and C++). Our model is designed for the general code retrieval task and some specific features may be added to it by gating. It is also interesting to study as the further work.

ACKNOWLEDGMENTS

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, and National Natural Science Foundation of China under Grant Nos. 61672045 and 61922003.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Miltos Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Proceedings)*, Francis R. Bach and David M. Blei (Eds.), Vol. 37. Journal of Machine Learning Research: Workshop and Conference Proceedings, 2123–2132.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv:cs.CL/1409.0473

[5] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5, 2 (March 1994), 157–166. https://doi.org/10.1109/72.279181

[6] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. arXiv:cs.SE/1905.03813

[7] Nick Craswell. 2009. Mean reciprocal rank. *Encyclopedia of Database Systems* (2009), 1703–1703.

[8] Jian Fu, Xipeng Qiu, and Xuanjing Huang. 2016. Convolutional deep neural networks for document-based question answering. In *Natural Language Understanding and Intelligent Applications*. Springer, 790–797.

[9] github. 2020. https://github.com/. github.

[10] Alessandro Giusti, Dan C Cireşan, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. 2013. Fast image scanning with deep max-pooling convolutional neural networks. In *2013 IEEE International Conference on Image Processing*. IEEE, 4034–4038.

[11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[12] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 842–851.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:cs.CV/1512.03385

[14] Dan Hendrycks and Kevin Gimpel. 2016. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR* abs/1606.08415 (2016). arXiv:1606.08415 http://arxiv.org/abs/1606.08415

[15] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. 2014. 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)*. IEEE, 34–43.

[16] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. arXiv:cs.NE/1207.0580

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[18] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. 2015. Convolutional Neural Network Architectures for Matching Natural Language Sentences. arXiv:cs.CL/1503.03244

[19] He Hua and Jimmy Lin. 2016. Pairwise Word Interaction Modeling with Deep Neural Networks for Semantic Similarity Measurement. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[20] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. 2019. Music Transformer: Generating Music with Long-Term Structure. In *ICLR*.

[21] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:cs.LG/1909.09436

[22] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[23] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 664–675. https://doi.org/10.1145/2568225.2568292

[24] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. arXiv:cs.CL/1408.5882

[25] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:cs.LG/1412.6980

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[27] Edward Loper and Steven Bird. 2002. NLTK: the natural language toolkit. *arXiv preprint cs/0205028* (2002).

[28] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? *Science China Information Sciences* 61 (05 2018), 056101. https://doi.org/10.1007/s11432-017-9355-3

[29] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 545–549. https://doi.org/10.1109/SANER.2015.7081874

[30] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:cs.CL/1301.3781

[31] Stack Overflow. 2020. https://stackoverflow.com/. Stack Overflow.

[32] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[33] Xipeng Qiu and Xuanjing Huang. 2015. Convolutional Neural Tensor Network Architecture for Community-Based Question Answering.. In *IJCAI*, Qiang Yang and Michael Wooldridge (Eds.). AAAI Press, 1305–1311. http://dblp.uni-trier.de/db/conf/ijcai/ijcai2015.html#QiuH15

[34] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. arXiv:cs.CL/1508.07909

[35] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7055–7062.

[36] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 8984–8991. https://aaai.org/ojs/index.php/AAAI/article/view/6430

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[39] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. 2017. Anne: Improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. 211–220.

[40] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[41] Mingzhou Xu, Derek F Wong, Baosong Yang, Yue Zhang, and Lidia S Chao. 2019. Leveraging local and global patterns for self-attention networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 3069–3075.

[42] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. *The World Wide Web Conference on - WWW '19* (2019). https://doi.org/10.1145/3308558.3313632

[43] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*. 1693–1703.

[44] Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. 2016. Text classification improved by integrating bidirectional LSTM with two-dimensional max pooling. *arXiv preprint arXiv:1611.06639* (2016).