Hierarchical Embedding for Code Search in Software Q&A Sites

Ruitong Li* , Gang Hu† and Min Peng‡
National Engineering Research Center for Multimedia Software, School of Computer Science, Wuhan University
Hubei, China
{liruitong*, hoogang†, pengm‡}@whu.edu.cn

***Abstract*—In recent years, code search techniques on software Q&A sites have become increasingly attractive due to the need for software development. Most of the existing work treats code snippets as text fragments, ignoring the effect of the structured information (i.e. sequential information) of the code. Meanwhile, much of the existing work does not take into account the interactive between code snippets and queries.**

In this paper, we propose a novel deep neural network named HECS[1] (Hierarchical embedding for code search) to solve the problems mentioned above. Our method divides the embedding process of code and query into two hierarchies, that is, the potential information is captured by two modules (the Intra-language encoding module and the Cross-language encoding module). In particular, our approach uses special LSTM (Long Short-Term Memory) variants, which is ON-LSTM (ordered neurons LSTM) to capture the keyword order structure of the code. The Intra-language encoding module is implemented by the LSTM variant and the Cross-language encoding module is an interactive information calculation module implemented by the attention mechanism. In this way, the similarity between the query and the corresponding code snippets in the vector space could be better captured. HECS can understand the difference between positive and negative samples more accurately.

We empirically evaluate HECS, using a large scale codebase collected from StackOverflow. The experimental results show that our approach achieves state-of-the-art performance.

***Index Terms*—code search, hierarchical embedding, structured sequence information**

I. Introduction

As data grows rapidly in various software Q&A sites, code search is becoming more and more important. Developers often search for examples of how to accomplish a certain coding task in software Q&A sites. Code search increases project developer productivity and reduces duplication of effort.

Many of the previous code search approaches have been proposed [1]–[3]. Most of them are base on information retrieval (IR) techniques. For example, Joel et al. [1] proposed that embedding a task-specific search engine in the development environment can significantly reduce the cost of finding information and enable programmers to write greater code more easily. Haiduc et al. [2] proposed a code recommender approach (called Refoqus) based on machine learning, which is trained with a sample of queries and relevant results. Linstead et al. [3] proposed Sourcerer, an information retrieval based code search tool that combines the textual content of a program with structural information. However, these code search methods have limitations because they do not utilize deep learning techniques. For example, an IR-based approach to one program language is often difficult to apply to another program language. Recent work [4], [5] has taken steps towards enabling more advanced code search using deep learning. We call such methods neural code search. Gu et al. [4] proposed CODEnn, which jointly embeds code snippets and natural language descriptions into a high-dimensional vector space. Srinivasan et al. [5] proposed CODE-NN, that uses LSTM networks with attention to producing natural sentences.

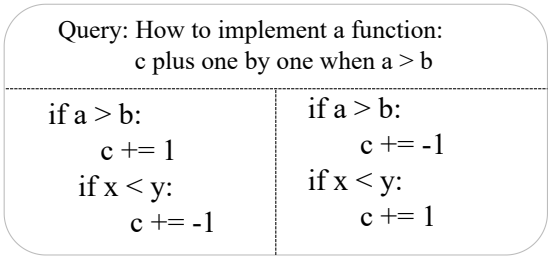


Fig. 1. Examples of different keyword sequences.

A fundamental problem of the previous code search methods is that none of them takes into account the effect of the keyword order of the code on the results. But the same code keywords can make up a completely different code. For example, the different sequences result in completely different results as shown in Fig 1. Where, the level represents the level of the logic of code execution, and the order represents the order in which the code statements are executed. The two pieces of code have the same keywords, but their keywords are of different levels and order, and the results are completely different.

The majority of the existing methods [4]–[11] apply word embedding techniques to train embedding for code. But in common neural networks, neurons are usually out of order. For example, the forget gate in LSTM is a vector, and there is no regularity in the position of the elements of the vector. If the position of all vectors involved in the LSTM operation is re-disrupted in the same way, the order of the weights is also disrupted accordingly. But the output can be just a reorder of the original vector (considering multiple layers,

‡Corresponding author.

[1]Replication package: https://github.com/lrt366/HECS

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

or even completely unchanged), the amount of information remains unchanged, without affecting the subsequent network use of it. In other words, neither LSTM nor ordinary neural networks use neuronal sequence information. Therefore none of the existing methods can capture the sequence information of the code. Namely, existing work cannot sequence these neurons and use this sequence to represent specific structures and effectively using neuronal sequence information.

Furthermore, most of the code search methods [4]–[10], [12], [13] can be divided into two categories, one modeling the question and answer vectors separately, and the other using the correspondence between the word levels of queries and code snippets. Nevertheless, there is no analysis of the interactive information between queries and code snippets.

In this paper, we propose a novel deep neural network named HECS (Hierarchical embedding for code search) for existing problems in software Q&A sites. We define a code search task for software Q&A sites, which can be formalized as a ranking problem searching for the standalone code solution in candidate list. At the same time, to achieve better training results, we use Noise-Contrastive Estimation [4] to learn hierarchical representations of the multiple inputs (query, positive code, and some negative code) directly, using a multiple ranking loss function to learn nonlinear feature correlations from the hierarchical representations.

To better embed code snippets and queries, we use the hierarchical model to extract features, divided into the Intra-language encoding module and the Cross-language encoding module. Through this coding method, HECS successfully captures the intrinsic association between them by mapping code snippets and queries to high-dimensional spaces. In summary, our contributions are as following:

- We introduce the Intra-language encoding module to capture the sequence information of the code. A snippet of code can often be expressed as a hierarchy that, if artificially abstracted, is what we call structural information, and HECS is able to model the natural representation of this hierarchy during training.
- We use the Cross-language encoding module to calculate the interactive between queries and code snippets. The attention to the corresponding code snippets on queries and the corresponding queries on code snippets is calculated, respectively. This approach makes the input layer of the model more interactive at the word level of queries and code snippets.
- We use the codebase of general-purpose imperative and interpreted programming languages, such as Python and C#, to demonstrate its effectiveness in code search for software Q&A sites. The results show that our approach achieves state-of-the-art performance.

The rest of this article is organized as follows. Section 2 describes the motivation and insight to the code search task. Section 3 describes the background techniques used for code search. Section 4 describes the detailed design of our approach. Section 5 describes the evaluation. Section 6 describes the results of the evaluation. Section 7 discusses our work. Then there is Section 8, which describes the related work. We conclude the paper in Section 9.

II. MOTIVATION AND INSIGHT

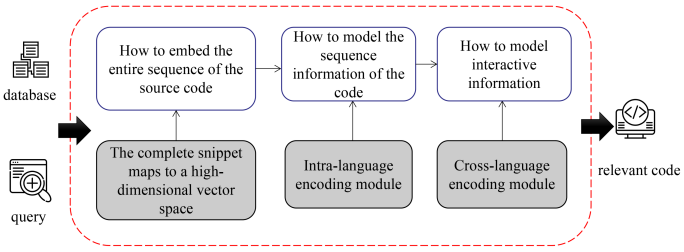


Fig. 2. Conceptual steps for HECS.

Recently, some state-of-the-art methods [4]–[9], [12] focused on using neural network models to perform semantic similarity calculations of code. Nevertheless, these neural modeling methods ignore the semantic modeling of code hierarchical structure information (including sequential information, interactive information). Due to the lack of this potential code information, the existing work will face the following difficulties:

- They do not extract the sequence information of the query and the code snippets, even if the sequence information is critical to the code snippets.
- They do not model semantically matched queries and interactive information represented by code snippets.
- They do not embed the entire sequence of the fabric of the source code, but rather embed the extracted independent elements of methods, APIs, tokens into the vector [4].

Inspired by a summary of the shortcomings of the existing work described above, we first confirmed that the Q&A programming post contained the hierarchical structure information presented. Then, according to the characteristics of the hierarchical information, we propose a method of code search, HECS, based on the idea of hierarchical feature extraction and deeply excavating code information, used to solve these problems. Fig 2 illustrates the steps to solve three problems with existing code search methods in software repositories such as StackOverflow.

HECS captures potential information about queries and code snippets to address these problems with the given queries and code snippets collected from a Q&A post. Specific resolution steps are shown in Fig 2: (1) How to embed the entire sequence of the source code? Embedding code snippets into separate elements can result in loss of information. HECS uses the SGNS-based [14] word2vec model to learn code structure embedding from millions of Q&A programming data. The complete code snippets is mapped to a continuous high-dimensional vector. (2) How to model the sequence information of the code? Ignoring the sequential information of the code and query causes the code search model to match incorrectly. HECS uses the Intra-language encoding modules based on ON-LSTM to embed sequential information in code snippets and queries. (3)

How to model interactive information? Ignoring the interactive information of code snippets and queries is ignoring the correlation analysis. HECS uses the Cross-language encoding modules, the joint training of queries and code snippets by the attention mechanism, to incorporate the importance weights of each other's presentation of learning.

III. BACKGROUND

A. LSTM

LSTM (Long Short-Term Memory) is a time-cycle neural network specifically designed to address the long-term dependence of the general RNN (Recurrent Neural Network). Consider a natural language sentence with a sequence of t words $s = w_1, \ldots, w_t$, LSTM embeds it through the following computations: it reads words in the sentence one by one, maintains long memories through the forget gate and output gate, and updates a hidden state at each time step. Each word w_t is first mapped to a d-dimensional vector $w_t \in R_d$ by an one-hot representation [15] or word embedding [16]. The update formula for hidden states on each time step is as follows:

$$\begin{aligned} f_t &= \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \\ i_t &= \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \\ o_t &= \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) \\ \hat{c}_t &= \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \hat{c}_t \\ h_t &= o_t \circ \tanh\left(c_t\right) \end{aligned} \tag{1}$$

f_t, i_t, o_t are three single-layer full-connection models, input is historical information h_{t-1} and current information x_t, activated with sigmoid, because the results of sigmoid are between 0 and 1, so their meaning can be interpreted as "gate", respectively, named forget gate, input gate, output gate.

B. ON-LSTM

The difference between ON-LSTM and LSTM is that the update formula for hidden states [14]. In order to learn the sequence information of the code, in ON-LSTM, the activation function is defined as:

$$\hat{g} = cumax(\cdots) = cumsum(softmax(\cdots)) \tag{2}$$

Where $cumsum$ denotes the cumulative sum. This formula allows hidden cells to be updated in two segments. Based on the $cumax(\cdot)$ function, ON-LSTM introduce a master forget gate $\tilde{f}_t$ and a master input gate $\tilde{i}_t$.

$$\tilde{f}_t = cumax(W_{\tilde{f}} x_t + U_{\tilde{f}} h_{t-1} + b_{\tilde{f}}) \tag{3}$$

$$\tilde{i}_t = 1 - cumax(W_{\tilde{i}} x_t + U_{\tilde{i}} h_{t-1} + b_{\tilde{i}}) \tag{4}$$

According to the formula introduced above, the update rules for hidden states are as follows:

$$w_t = \tilde{f}_t \circ \tilde{i}_t \tag{5}$$

$$\hat{f}_t = f_t \circ w_t + (\tilde{f}_t - w_t) = \tilde{f}_t \circ (f_t \circ \tilde{i}_t + 1 - \tilde{i}_t) \tag{6}$$

$$\hat{i}_t = i_t \circ w_t + (\tilde{i}_t - w_t) = \tilde{i}_t \circ (\tilde{f}_t \circ i_t + 1 - \tilde{f}_t) \tag{7}$$

$$c_t = \hat{f}_t \circ c_{t-1} + \hat{i}_t \circ \hat{c}_t \tag{8}$$

Shen et al. [14] explains why revised update rules can learn sequential information. The structure of ON-LSTM is shown in the Fig 3.

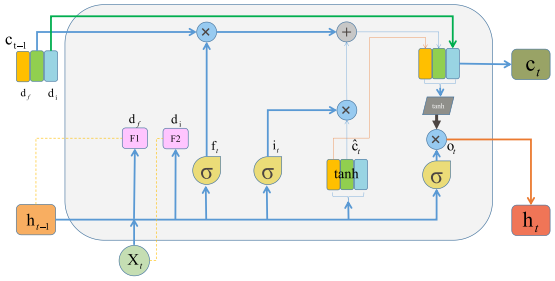


Fig. 3. The structure of ON-LSTM

IV. A DEEP NEURAL NETWORK FOR CODE SEARCH

Since the existing code search methods ignore the interactive and sequential information, we propose a novel deep neural network named HECS. Overall, the method is better able to map a pair of samples (code snippets, queries) to near points in high-dimensional space.

A. Architecture

Inspired by the existing work using neural network models [4], [7], [8], [17], we have proposed a new method of code search called HECS. HECS is a supervised embedded model that uses deep neural networks to embed code snippets and natural language queries into neighbor vectors in uniform space and measure their similarity. As introduced in Section 3, the LSTM used in HECS is replaced with ON-LSTM, Fig 4 shows the overall architecture of HECS. The model consists of three modules, each one corresponding to a module of hierarchical embedding.

B. The Intra-language Encoding Module

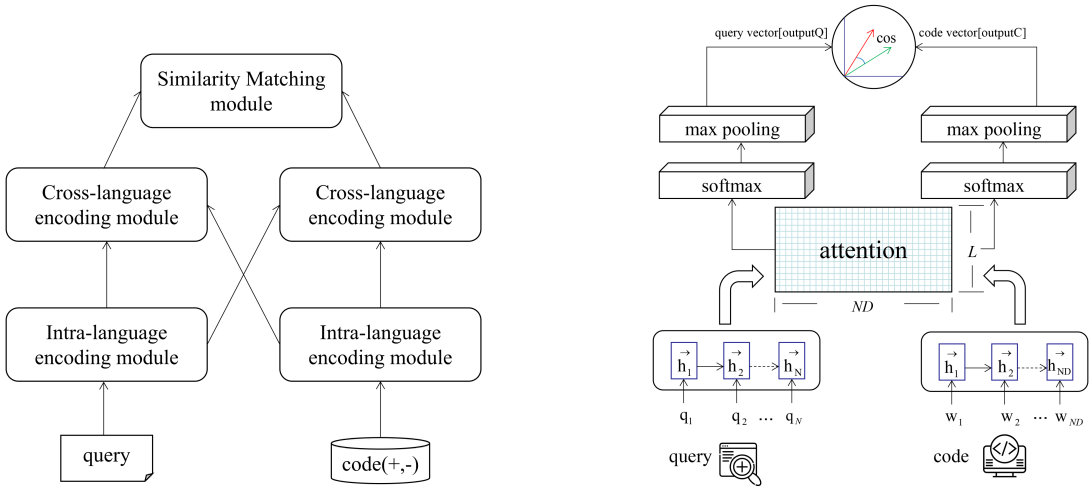
The Intra-language encoding module (ILEM) captures and initially encodes the sequence information of the code snippets and queries. Consider a snippet of code $C = w_1, w_2, ..., w_{ND}$, which contains a sequence of ND words and a query $Q = q_1, q_2, \ldots, q_N$ that contains N words.

$$\begin{aligned} C &= (w_1, \ldots, w_{ND}) \\ Q &= (q_1, \ldots, q_N) \end{aligned} \tag{9}$$

Where w_i or q_i is a vector standing for a d dimentional word embedding for the ith word in the code snippets. The dimension of C is nd-by-d and the dimension of Q is n-by-d. ILEM encodes it into a vector using an ON-LSTM.

$$\vec{h}_C = \overrightarrow{ON - LSTM}(w_t, \vec{h_{t-1}}) \tag{10}$$

$$\vec{h}_Q = \overrightarrow{ON - LSTM}(q_t, \vec{h_{t-1}}) \tag{11}$$



(a) Overall Architecture (b) Detailed Structure

Fig. 4. The structure of the Hierarchical embedding for code search

Let the hidden unit number for each unidirectional ON-LSTM be u. With ON-LSTM encoding, the shape of h_C becomes nd-by-u and the shape of h_Q becomes n-by-u.

Next, we will explain how ON-LSTM embeds a sentence (e.g., get a list) into a vector. Fig 5 shows this process, where the recurrent hidden layer of ON-LSTM expands at each time step. ON-LSTM reads the words in the sentence one by one, captures the additional sequence information, and updates a hidden state at each time step. This is a specific procedure: first it reads the first word "get", maps the word to the vector w_1, and then uses w_1 to calculate the current hidden state h_1. It then reads the second word, "a", maps the word to the vector w_2, and then uses w_2 to update the hidden state h_1 to h_2. This process continues until the ON-LSTM receives the last word "list" and outputs the final state h_3. The last state h_3 can be used as an embedded c for the entire sentence.

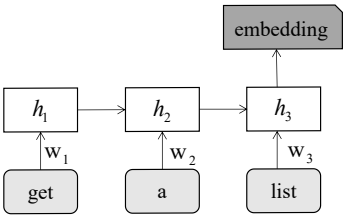


Fig. 5. ON-LSTM for sentence embedding.

C. The Cross-language Encoding Module

The Cross-language encoding module (CLEM) captures and encodes the interactive information of the code snippets and queries. The proposed CLEM model is an attention mechanism, which provides a set of summation weight vectors for the code snippets and queries. For this section, Fig 6 describes our model. First, the vector of the output query is calculated as follows:

$$outputQ = maxpooling(h_Q) \tag{12}$$

Next the attention matrix of the code snippets above the query needs to be calculated.

$$W_{qc} = tanh((h_Q \circ W_q) + (h_C \circ W_c)) \tag{13}$$

Here Wq and Wc is a weight matrix with a shape of u-by-L. Therefore, the shape of Wqc is nd-by-L. Wqc is the attention matrix of the interactive information between the code snippets and the query. The weight that acts on the code is calculated as follows:

$$P = W_{qc} \circ W \tag{14}$$

$$att = matrixdiag(softmax(P)) \tag{15}$$

Here the shape of W is L-by-1. Then the shape of P is reconstructed to nd. att is a diagonal matrix that P builds after the softmax layer. This is the attention with interactive information. Finally, the output code snippets vector required is as follows:

$$outputC = maxpooling(att \circ h_C) \tag{16}$$

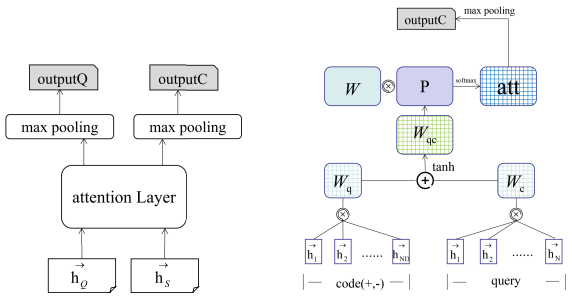
Since att is sized nd-by-nd, the annotation vector $outputC$ will have a size nd-by-u.

D. Similarity Matching Module

We have described the transformations that encode the natural query Q and the code snippets C, which have been converted to $outputC$ and $outputQ$. Using the existing method [4] [18], we use cosine similarity $cos(VQ, VC)$ to measure the similarity between source code and natural query. The cosine similarity $cos(VQ, VC)$ is defined as:

$$cos(outputQ, outputC) = \frac{outputQ^T outputC}{||outputQ|| \bullet ||outputC||} \tag{17}$$

Where $||outputQ|| \bullet ||outputC||$ represents the multiplication of two matrices via their transpose. In addition, $outputC$



(a) Architecture (b) Attention in (a)

Fig. 6. The Cross-language encoding module

and $outputQ$ are the vectors of code snippets and query respectively. The higher the similarity, the more related the code snippets is to the query. Anyway, we enter a pair of samples (Q, C), which translates into vectors, and finally use this method to predict the similarity between code snippets and query.

E. Loss Function

Our goal is to learn a representation function $f(\cdot)$ such that given some queries q, positive pairs (q, c^+) are assigned larger similarity scores than negative pairs (q, c^-):

$$f(q, c^+) > f(q, c^-), \forall q, c^+, c^- \tag{18}$$

Where q, c^+, c^- denote the query, positive code snippets and negative code snippets, respectively. In order to measure the distance between query and code in high-dimensional space, we use the cosine similarity ($cos(\cdot)$) as function $f(\cdot)$ for the measurement. So our final loss function is defined as follows:

$$\min_{w} \sum_{(q,c^+,c^-)\in N} \max(0, 1 - (f(q, c^+) - f(q, c^-)) + \lambda||W||^2) \tag{19}$$

Where W denotes all the model parameters, $\lambda \in [0, 1]$ is a regularization parameter. We then use a triplet ranking loss, which minimizes the distance between the query q and a positive code snippets c^+, and maximizes the distance between the q and a negative code snippets c^-. According to e.q. (17), our final loss function is as follows:

$$\min_{w} \sum_{(q,c^+,c^-)\in N} \max(0, 1 - (cos(q, c^+) - cos(q, c^-)) + \lambda||W||^2) \tag{20}$$

V. EVALUATION

In this section, we will evaluate the effectiveness of the above methods at different design points. It will also be compared with several existing code search methods.

A. Corpus

Code search tasks require a huge amount of data. StackOverflow is a popular Q&A forum for posting program-related questions, and anonymized versions of posts can be freely downloaded from StackExchange [19]. By using regular expressions (e.g., Tags = ".*Python.*") to match different tags, we can extract multilingual Q&A programming posts to further build the embedded library. So we build the original corpus from posts on StackOverflow, which includes posts in two types of programming languages (Python and C#). In particular, we only keep posts with multiple candidate code snippets. Our model is more concerned with the complete code snippets than the part of the code snippets (method body, API sequence, etc.), so we remove the candidate codes with a length of less than 10 in the corpus to ensure the number of candidates. Finally, we used the CodeMF proposed by Hu et al. [20] to eliminate noisy posts and extract high-quality software repositories from programming forums.

Moreover, we try to obtain more corpora for experiments, but [4], [21] do not share them. In addition, [6], [22] shared their corpora, but did not share the corpora processing method. For each programming language in our corpus, we use 80% of it for training, 10% for validation and 10% for testing. The statistics of the corpus are shown in TABLE I, which will be used to develop our models. The column "label with '1'" represents the ratio of the best answers to the total answers. For Python and C#, the number of samples of not less than 10 lengths is 279695 and 237078. We obtained a labeled corpus, which we call the Multiple Candidate QC Pairs (MucQC) corpus. In the process of collecting posts, we filter out posts with only one answer and keep posts with at least two answers. In the reserved post, we mark the code snippets based on the result of the vote (the code with the highest number of votes is marked positive (the best answer), and the rest is marked as negative.).

TABLE I
THE STATISTICS OF THE MUCQC CORPUS

Data Type	**Python**		**C#**	
Length≥10	279695		237078	
Label Types	*QC pairs*	*Label with "1"*	*QC pairs*	*Label with "1"*
Training data	222524	30.95%	187500	34.71%
Validation data	28717	29.98%	24708	32.92%
Testing data	28449	30.26%	24864	32.72%

Compared to previous work [18], [23], our collected MucQC corpus is better in evaluating the performance of code search methods, which has the following advantages: (1) The data source comes from a real code search Q&A community StackOverflow, not the cleaned GitHub data. (2) The code type is a more complex code-snippet level, not a class or method level.

At the same time, in order to test the validity of the model on the public corpus, we also conducted a comparative experiment on StaQC (Stack Overflow Question-Code pairs), the largest dataset to date of 148K Python and 120K SQL QC pairs. StaQC is a corpus proposed by Yao et al. [23] that was mined based on BiV-HNN. However, the code content in the SQL program in StaQC carries less important structured information when making predictions and lacks intermediate

variable names. Therefore, to better compare with the Python dataset in MucQC, we chose the Python dataset in StaQC for the experiment. StaQC is divided in the same way as MucQC. The statistics of the StaQC corpus are shown in TABLE II. Unlike the data processing of the original work, we filtered the manually annotated QC pairs and retained only Multi-Code Answer Posts and Single-Code Answer Posts. At the same time, we split a pair of data from Multi-Code Answer Posts into multi-pair data such as Single-Code Answer Posts. MucQC uses the same data processing process.

After collecting the data, we used different bilingual tokenizers (ANTLR parser for C# [5], python built-in parser for Python [23], WordNet [24] and Lemmatizer [25] for NL queries) to obtain a structured sequence of code snippets and NL queries, respectively. In addition, we have unified the same type of characters. (e.g. "c#", "C#" and "C#.Net" replaced with "csharp")

TABLE II
THE STATISTICS OF THE STAQC CORPUS

Data Type	**Python**	
Size	185906	
Label Types	*QC pairs*	*Label with "1"*
Training data	147851	63.05%
Validation data	18862	63.81%
Testing data	19193	64.42%

B. Evaluation Methodology

We used four evaluation metrics, FRank, Precision@k, Recall@k and Mean Reciprocal Rank (MRR), that were widely used in information retrieval tasks as well as code search tasks [26]–[29].

FRank: The FRank (also called hit rank [26]) is the rank of the first hit result in the response list. This measurement is a very classic and important metrics in code search tasks. It represents the ability of the method to sort responses. The smaller Frank, the faster the user can find the right response.

Precision@k: The Precision@k [27] measures the percentage of relevant results in the top k returned responses for each query. It represents the amount of noise that the code search method returns the result. Precision@k is calculated as follows:

$$Precision@k = \frac{\text{relevant responses in the top } k \text{ responses}}{k} \tag{21}$$

The higher the metrics value, the less noise, and the better the performance of the code search approach.

MRR: The calculation process of MRR is to count the standard answer in the order of the results given by the evaluation system as its accuracy, and then average all the questions. MRR is calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q} \tag{22}$$

The higher the value of MRR, the better the code search capability of the model.

Recall@k: Recall@k indicates that the positive sample is predicted to be the number of positive samples for all samples. Recall@k is calculated as follows:

$$\text{Re}call@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\lambda(FRank_q \leq k) \tag{23}$$

The higher the value of Recall@k, the better the model.

C. Comparison Methods

To prove the effectiveness of our model, we selected the most advanced or widely used methods to compare. This section describes the models used for comparison and the specific implementations of the models we propose. All the experiments were carried out on the MucQC/StaQC corpus.

CodeLSTM: LSTM is a neural network widely used in natural language processing tasks. The model consists of an embedding layer and a bidirectional LSTM layer, with the hidden unit of LSTM set to 256. Based on experience, we set the dimension of word embedding to 300.

CodeCNN: CNN (convolutional neural networks) is the representative algorithm for machine learning. The model consists of an embedding layer and a CNN layer, with the number of filters set to 150 and the size of filters set to [1, 2, 3, 5]. Based on experience, we set the dimension of word embedding to 300.

CodeRCNN: The model includes an embedding layer, an LSTM layer, and a CNN layer. The CNN layer and the LSTM layer are set up the same as above. Based on experience, we set the dimension of word embedding to 300.

DeepCS: DeepCS is a state-of-the-art code search engine proposed recently [4]. It is a code search model based on neural networks that combine several pooled layers and several RNN (Recurrent Neural Network) layers. For the consistency of the experiment, we did not use the method of dividing the code into three aspects. We encode the entire code snippets into a vector. The model consists of an embedding layer, a bidirectional LSTM layer, and a max-pooling layer, with the hidden unit of LSTM set to 256. Based on experience, we set the dimension of word embedding to 300.

UNIF: UNIF is an improved method of supervised nerves in NCS [21]. NCS uses the Bag-of-words model and FastText model [30] to joint training code snippets and natural language queries, while UNIF primarily improves the use of attention mechanisms to calculate the weights of word embedding. Based on experience, we set the dimension of word embedding to 300.

CodeATT: CodeATT is a recently proposed baseline, which similar to attention-based QA-LSTM [31]. CodeATT only involves an embedding layer and an interactive attention layer, where CodeATT and HECS have different attention mechanisms. Based on experience, we set the dimension of word embedding to 300.

CodeONLSTM: To verify the effect of the Cross-language encoding module, we replace the normal LSTM network with

TABLE III
RESULTS FOR RQ1, RQ2 AND RQ3

Model	MucQC				StaQC	
	Python		C#		Python	
	Recall@1	*MRR*	*Recall@1*	*MRR*	*Recall@1*	*MRR*
CodeLSTM	0.5269	0.7247	0.4972	0.7104	0.5787	0.7669
CodeCNN	0.5175	0.7184	0.4850	0.7036	0.5687	0.7603
CodeRCNN	0.5241	0.7233	0.5079	0.7168	0.5790	0.7662
SeltATT	0.5307	0.7273	0.5130	0.7197	0.5777	0.7667
UNIF	0.3696	0.6245	0.3739	0.6338	0.4401	0.6825
DeepCS	0.5306	0.7268	0.5097	0.7177	0.5782	0.7713
CodeONLSTM	0.5334	0.7286	0.5156	0.7205	0.5741	0.7646
CodeATTENTION	0.4768	0.6929	0.4860	0.7036	0.5223	0.7333
HECS	**0.5419**	**0.7338**	**0.5173**	**0.7226**	**0.5900**	**0.7735**

an ON-LSTM network and the remaining parameters are set unchanged.

CodeATTENTION: To verify the effect of the Intra-language encoding module, we designed a model with only an interactive mechanism and the remaining parameters are set unchanged.

HECS: The detailed implementation of the HECS model is as follows: Based on experience, we used an ON-LSTM layer with a hidden unit set to 512, and we set the dimension of word embedding to 300. The shape of matrix W is set to (200, 1), W_c and W_q to (512, 200), and W_{qc} to (200, 200). For the training optimization algorithm, we select the mini-batch Adam algorithm [37, 40]. Based on past experience, the learning rate of the model is set to 0.0002 and the batch size is set to 128. Finally, we built the model using the open-source framework of TensorFlow, which lasted 50 epochs.

VI. EXPERIMENTAL RESULTS

In this section, we evaluate HECS through experiments. Specifically, our experiments aim to address the following research questions:

- RQ1: Whether HECS (Hierarchical embedding for the code search method) achieves state-of-the-art performance in all the benchmarks?
- RQ2: Whether CLEM (Cross-language encoding module) introduced with attention information can improve the performance than Non-cross-language encoding based code search?
- RQ3: Whether ILEM (Intra-language encoding module) introduced can improve the performance than unordered encoding based code search?

A. RQ1

The first experiment is conducted to identify the effectiveness of HECS for code search. TABLE III shows the overall performance of the comparative approaches, measured in terms of Recall@1 and MRR. In contrast to the original work, Table III shows that DeepCS is better than UNIF. This is because DeepCS adopts special tuples of the code fragment for representing program method instead of using overall sequential tokens. However, recent work has demonstrated that sequence-based embedding technology enables state-of-the-art code search performance, such as CoaCor [4] and CODE-NN [5]. As a result, the sequence-based DeepCS network provides a better way to capture the joint representation of the source code and NL queries for code search.

Compared with the state-of-the-art methods (e.g., UNIF [21] and CODEnn [4]) available, we can conclude that HECS has achieved the optimal performance. Because HECS substantially outperforms all baselines. Especially on the results of the public corpus StaQC, we can see that the gain of HECS is relatively higher. The reason for this result is that the StaQC corpus is collected using supervised BiV-HNN model, the quality of the QC pairs obtained was higher than MucQC, and then, HESC can learn embedded representation more efficiently with the better-quality QC pairs. The results show that the embedded vector of HESC is a superior representation of the potential information of code snippets and queries than other methods. Thus given query HESC can generate more relevant code.

The columns Recall@1 show the results of Recall@k when k is 1. For this measurement, we select the most stringent criteria (k=1), which makes the results more representative. HECS achieves a Recall@1 of 1% to 14% higher than the benchmarks. The MRR column represents the MRR values of each comparison model. HECS achieves an MRR of 1% to 11% higher than the benchmarks. This means that HECS can generate code snippets that are more relevant to the query.

The hierarchical embedding of code search method HECS enables state-of-the-art performance across all benchmarks.

B. RQ2

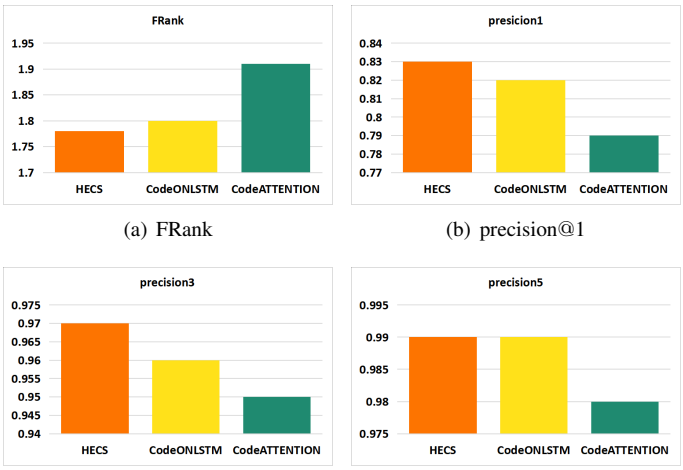
The second experiment is conducted to identify the effectiveness of the Cross-language encoding module for HECS. We designed an ablation experiment on MucQC/StaQC corpus. Model CodeONLSTM is a model that removes the Cross-language encoding module from the HECS, with only one Intra-language encoding module and one Similarity Matching module.

Additionally, TABLE III shows the evaluation results of the ablation experiment. HECS achieves Recall@1 and MRR is 1% higher than CodeONLSTM. Fig 7 is a summary of FRank and Precision@k for the three ablation experiments on MucQC corpus. The FRank value of HECS is 2% lower than CodeONLSTM. The Precision@k values of HECS when k is 1, 3 and 5 are 0% to 1% higher than CodeONLSTM, respectively. Besides, CodeONLSTM achieves Recall@1 and MRR is 1% to 2% higher than CodeLSTM. The results show that ON-LSTM captures the sequence information of the code more successfully than LSTM.

This result shows that the Cross-language encoding module (CLEM) is facilitating the potential connection between the learning code snippets and the corresponding query.

C. RQ3

The third experiment is conducted to identify the effectiveness of the Intra-language encoding module for HECS. Similar to the above, we also designed an ablation experiment on MucQC/StaQC corpus. CodeATTENTION is a model that removes the Intra-language encoding module from the HECS, with only one Cross-language encoding module and one Similarity Matching module.



(a) FRank (b) precision@1

(c) precision@3 (d) precision@5

Fig. 7. The statistical comparison of FRank and Precison@k for three code search approaches.

Furthermore, TABLE III and Fig 7 show the evaluation results of ablation experiment. HECS achieves more 3% to 7% higher Recall@1 and 1% to 4% better MRR than CodeATTENTION in C# and Python datasets. The FRank value of HECS is 13% lower than CodeATTENTION. The Precision@k values of HECS when k is 1, 3 and 5 are 1% to 4% higher than CodeATTENTION, respectively.

In particular, the results of the model CodeONLSTM are better than CodeATTENTION. CodeONLSTM achieves more 3% to 6% higher Recall@1 and 2% to 3% better MRR than CodeATTENTION. This result further demonstrates that the Intra-language encoding module is essential for learning to embed representation. However, we cannot conclude that the Cross-language encoding module is not important, because the task of Cross-language encoding module is to learn the interactive information between the code snippets and the query on the basis of the Intra-language encoding module.

This result indicates that the Intra-language encoding module (ILEM) is more suitable for code search tasks and can capture the sequence information of code snippets.

D. Example of Search Results

To demonstrate the actual effect of HECS, we now provide specific examples of code search results for benchmark searching codebase. In case of Python, we chose a more representative post as a sample. For example, Post# 952914, the query is *"How to make a flat list out of list of lists?"*. We pick the top five from its response. As shown in Fig 8, HECS is able to retrieve a stand-alone code solution (the correct code required).

query	**How to make a flat list out of list of lists**
1st	`flat_list = [item for sublist in l for item in sublist]`
2nd	`flatten = lambda l: [` `  item for sublist in l for item in sublist]`

3rd	`flat_list = []`
	`for sublist in l:`
	`  for item in sublist:`
	`  flat_list.append(item)`

4th	`import itertools`
	`list2d = [[1,2,3],[4,5,6], [7], [8,9]]`
	`merged = list(itertools.chain(*list2d))`

5th	`import itertools`
	`list2d = [[1,2,3],[4,5,6], [7], [8,9]]`
	`merged = list(itertools.chain.from_iterable(list2d))`

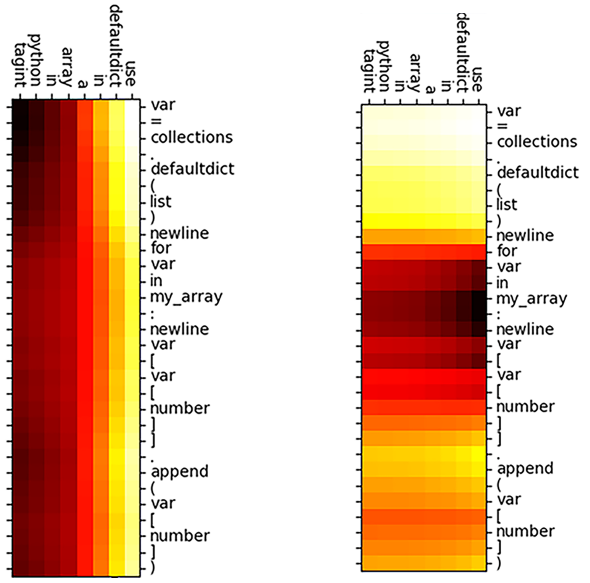
Fig. 8. Example of code search.

As with previous code search methods using neural networks [4], query comprehension enables HECS to perform more powerful code search. Its search results are less affected by irrelevant or noisy keywords. Fig 8 shows that HECS got the correct response to the keyword "flat list".

In addition, we can find two very similar results in the first three responses (1st and 2nd). These two candidate code snippets have the most similar keywords, and traditional code search methods are difficult to rank them. Obviously HECS is able to capture the sequence information of the code snippets and rank them as a way to do so.

HECS also fully understands the semantics of queries and finds responses (1st and 4th) with different keywords but the same meaning through associated searches.

To demonstrate the effect of the interactive mechanism, we show the attention weight above the query *"use defaultdict in a array in Python tagint"* for the different fragments of



(a) Attention weights of the direction of the query

(b) Attention weights of the direction of the code

Fig. 9. Heatmap of attention weights M_{qc} for example Python code fragments

the relevant code. Fig 9 shows the interactive attention weight value between the query and its corresponding code snippets. Fig 9(a) indicates that attention weights of the direction of the query, Fig 9(b) indicates that attention weights of the direction of the code. Darker areas represent stronger weights. We can observe that dark areas are concentrated in critical and important parts.

VII. DISCUSSIONS

A. Why Does HECS Work?

We have identified three advantages of HECS that may explain its effectiveness in code search.

The well-informationed embedded vector of global code information. The field of natural language processing usually embeds a complete sentence into a high-dimensional space. But the code search tasks typically divide the code into several parts to embed and then connect the embedded vector [4]. This can result in the loss of potential information between parts of the code. Our work is to map the complete code snippets to the continuous high-dimensional space to preserve this potential information.

Take full advantage of code snippets structured sequence information. The sequence information of a code snippets is often important compared to a text snippet. Different keyword sequences result in different meanings in the code. However, most of the current neural network code search methods do not deal with this characteristic of code. Our work uses the Intra-language encoding modules based on ON-LSTM to fully embed sequential information, effectively solve this problem.

A deeper understanding of query based on code attention weights. Unlike traditional code search methods, our method uses attention-based Cross-language encoding modules to capture the correlation information between queries and code to better understand the meaning of queries. This value-weighted model is able to give different attention of both the natural language query and code terms in their respective directions, achieving a focus on the natural language query description.

B. Threats to Validity

HECS differs from these existing methods that focus on class-type or method-type code snippets because we are designed to improve the performance of retrieving code snippets through a Q&A site, such as StackOverflow. Threats to the effectiveness of HECS include: (1) **external validity.** The use of large codebases in previous work [4], [18], [21], [23]. However, because online forums such as StackOverflow contain a large number of software libraries, there are very limited code candidates from actual code search scenarios. Thus our corpus is a little bit small. (2) **internal validity.** Since our approach is based on a neural model with supervised learning, the performance of the model is largely limited by the preprocessing of the data and the size of the corpus. (3) **construct validity.** Due to the limitations of experimental conditions, we are short of the experiments of manually evaluating the relevance of the returned code ranking.

VIII. RELATED WORK

Code search tasks have become increasingly popular recently. We survey some of the work that is most closely related to what we are focused on. Most of the traditional code search methods are based on information retrieval technology [32], [33]. For example, Mohammad et al. [33] proposed RACK to retrieve by converting queries to lists of API classes used to collect relevant code. McMillan et al. [32] have proposed portfolios and extracted relevant functions of incentive queries through keyword matching and PageRank. There are also work based on query expansion and reformulation [2], [34]. Haiduc et al. [2] propose machine learning-based queries to reformulate strategies. Zhang et al. [34] proposed an automated approach to using Word2vec [16] to recommend semantically related API class names. As described in Section 4, the HECS we recommend is different from the existing code search techniques described above, because the model is based on a supervised model of a neural network and there is spatial proximity between queries and code snippets. Therefore, it does not need to deal with a term mismatch.

Due to the development of natural language processing tasks, much of the recent work has focused on semantic-based code search [4], [18], [21]. Yao et al. [18] proposed the code annotation method CoaCor, based on intensive learning. Gu et al. [4] proposed the method CODEnn for combining embedding and deep learning to measure the similarity between code snippets and user queries. Hamel et al. [12] proposed a supervised neural code search system that uses multiple sequence-to-sequence networks. Cambronero et al. [21] proposed a method named UNIF for calculating code

sentence embedding using attention mechanisms. As shown in Section 4, there are significant differences between these existing technologies and HECS. None of them considered the importance of interactive information for semantic feature representation and matching between natural language queries and source code. In addition, their embedded representation does not take into account the sequential information of the code snippets, but this potential information is critical to the code search task.

In recent years, most code search tasks have used models based on deep learning techniques [5], [9], [35], such as code summarization, code generation, code recommendation, etc. For the code generation task, Mou et al. [35] proposed a code model based on the user intent Encoder-Decoder model. Srinivasan et al. [5] generate synth snippets and SQL programs through attention mechanisms and LSTM networks. Allamanis et al. [9] use convolutional neural networks (CNN) to summarize code snippets. In our work, we embed the sequence information of the code snippets and the interactive information of the query and the code snippets into the high-dimensional space, and dig the in-depth information of the code search task hierarchically.

IX. CONCLUSION

In this paper, a new type of deep neural network for Software Q&A Sites called HECS (Hierarchical embedding for code search) is proposed. HECS uses the interactive structure of code snippets and natural queries, and special LSTM variants (ON-LSTM) to capture the keyword order structure of the code, capturing local and global characteristics, respectively. Experiments show that the hierarchical design of HECS is superior to several baselines, and the two modules are effective. Moreover, the framework we designed in HECS could theoretically improve the performance of other software tasks, such as code summarization and code recommendation.

ACKNOWLEDGMENT

Thanks for the financial support of the National Key RD Program of China under Grant No. 2018YFC1604000 and No. 2018YFC1604003 and Natural Science Foundation of China (NSFC) under Grant No. 61872272 and No. 61772382.

REFERENCES

[1] J. Brandt, M. Dontcheva, and et.al, "Example-centric programming: integrating web search into the development environment," in *28th CHI*, 2010, pp. 513–522.

[2] S. Haiduc, G. Bavota, and et.al, "Automatic query reformulations for text retrieval in software engineering," in *35th ICSE*, 2013, pp. 842–851.

[3] E. Linstead, S. K. Bajracharya, and et.al, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. Knowl. Discov.*, vol. 18, no. 2, pp. 300–336, 2009.

[4] X. Gu, H. Zhang, and et.al, "Deep code search," in *40th ICSE*, 2018, pp. 933–944.

[5] S. Iyer, I. Konstas, and et.al, "Summarizing source code using a neural attention model," in *54th ACL*, 2016.

[6] S. Sachdev, H. Li, and et.al, "Retrieval on source code: a neural code search," in *2nd ACM SIGPLAN MAPL*, 2018, pp. 31–41.

[7] X. Qiu and X. Huang, "Convolutional neural tensor network architecture for community-based question answering," in *24th IJCA*, 2015, pp. 1305–1311.

[8] A. Severyn and A. Moschitti, "Learning to rank short text pairs with convolutional deep neural networks," in *38th ACM SIGIR*, 2015, pp. 373–382.

[9] M. Allamanis, H. Peng, and et.al, "A convolutional attention network for extreme summarization of source code," in *33nd ICML*, 2016, pp. 2091–2100.

[10] L. Mou, G. Li, and et.al, "Convolutional neural networks over tree structures for programming language processing," in *30th AAAI*, 2016, pp. 1287–1293.

[11] N. D. Q. Bui and L. Jiang, "Hierarchical learning of cross-language mappings through distributed vector representations for code," in *40th ICSE (NIER)*, 2018, pp. 33–36.

[12] H. Husain and H.-H. Wu, "How to create natural language semantic search for arbitrary objects with deep learning," 2018.

[13] H. Peng, L. Mou, and et.al, "Building program vector representations for deep learning," in *8th KSEM*, 2015, pp. 547–553.

[14] Y. Shen, S. Tan, and et.al, "Ordered neurons: Integrating tree structures into recurrent neural networks," in *7th ICLR*, 2019.

[15] J. P. Turian, L. Ratinov, and et.al, "Word representations: A simple and general method for semi-supervised learning," in *48th ACL*, 2010, pp. 384–394.

[16] T. Mikolov, I. Sutskever, and et.al, "Distributed representations of words and phrases and their compositionality," in *27th NIPS*, 2013, pp. 3111–3119.

[17] X. Huo, M. Li, and et.al, "Learning unified features from natural and programming languages for locating buggy source code," in *25th IJCAI*, 2016, pp. 1606–1612.

[18] Z. Yao, J. R. Peddamail, and et.al, "Coacor: Code annotation for code retrieval with reinforcement learning," in *2019 WWW*, 2019, pp. 2203–2214.

[19] "Stack exchange." [Online]. Available: https://stackexchange.com/

[20] G. Hu, M. Peng, and et.al, "Unsupervised software repositories mining and its application to code search," *Software: Practice and Experience*, 2019.

[21] J. Cambronero, H. Li, and et.al, "When deep learning met code search," in *ACM ESEC/SIGSOFT FSE*, 2019, pp. 964–974.

[22] P. Yin, B. Deng, and et.al, "Learning to mine aligned code and natural language pairs from stack overflow," in *15th MSR*, 2018, pp. 476–486.

[23] Z. Yao, D. S. Weld, and et.al, "Staqc: A systematically mined question-code dataset from stack overflow," in *2018 WWW*, 2018, pp. 1693–1703.

[24] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[25] D. Cavar, T. Gulan, and et.al, "The scheme natural language toolkit (S-NLTK): NLP library for R6RS and racket," in *4th ELS*, 2011, pp. 58–61.

[26] X. Li, Z. Wang, and et.al, "Relationship-aware code search for javascript frameworks," in *24th ACM SIGSOFT, FSE*, 2016, pp. 690–701.

[27] F. Lv, H. Zhang, and et.al, "Codehow: Effective code search based on API understanding and extended boolean model (E)," in *30th IEEE/ACM, ASE*, 2015, pp. 260–270.

[28] M. Raghothaman, Y. Wei, and et.al, "SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis," in *38th ICSE*, 2016, pp. 357–367.

[29] X. Ye, R. C. Bunescu, and et.al, "Learning to rank relevant files for bug reports using domain knowledge," in *22nd ACM SIGSOFT (FSE-22)*, 2014, pp. 689–699.

[30] E. Grave, T. Mikolov, and et.al, "Bag of tricks for efficient text classification," in *15th EACL*, 2017, pp. 427–431.

[31] H. Husain, H.-H. Wu, and et.al, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[32] C. McMillan, M. Grechanik, and et.al, "Portfolio: finding relevant functions and their usage," in *33rd ICSE*, 2011, pp. 111–120.

[33] M. M. Rahman and C. K. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *2018 IEEE,ICSME*, 2018, pp. 473–484.

[34] F. Zhang, H. Niu, and et.al, "Expanding queries for code search using semantically related API class-names," *IEEE Trans. Software Eng.*, vol. 44, no. 11, pp. 1070–1082, 2018.

[35] L. Mou, R. Men, and et.al, "On end-to-end program generation from user intention by deep neural networks," *CoRR*, vol. abs/1510.07211, 2015.