The Journal of Systems & Software 170 (2020) 110773

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Neural joint attention code search over structure embeddings for software Q&A sites

Gang Hu [a], Min Peng [a,*], Yihan Zhang [b], Qianqian Xie [a], Mengting Yuan [a,*]

[a] Department of National Engineering Research Center for Multimedia Software, School of Computer Science, Wuhan University, Wuhan, China
[b] School of Computing, National University of Singapore, Singapore

ARTICLE INFO

Article history:
Received 15 September 2019
Received in revised form 28 July 2020
Accepted 1 August 2020
Available online 5 August 2020

Keywords:
Code search
Software Q&A sites
Joint attention
Structure embeddings

ABSTRACT

Code search is frequently needed in software Q&A sites for software development. Over the years, various code search engines and techniques have been explored to support user query. Early approaches often utilize text retrieval models to match textual code fragments for natural query, but fail to build sufficient semantic correlations. Some recent advanced neural methods focus on restructuring bi-modal networks to measure the semantic similarity. However, they ignore potential structure information of source codes and the joint attention information from natural queries. In addition, they mostly focus on specific code structures, rather than general code fragments in software Q&A sites.

In this paper, we propose NJACS, a novel two-way attention-based neural network for retrieving code fragments in software Q&A sites, which aligns and focuses the more structure informative parts of source codes to natural query. Instead of directly learning bi-modal unified vector representations, NJACS first embeds the queries and codes using a bidirectional LSTM with pre-trained structure embeddings separately, then learns an aligned joint attention matrix for query-code mappings, and finally derives the pooling-based projection vectors in different directions to guide the attention-based representations. On different benchmark search codebase collected from StackOverflow, NJACS outperforms state-of-art baselines with 7.5% to 6% higher Recall@1 and MRR, respectively. Moreover, our designed structure embeddings can be leveraged for other deep-learning-based software tasks.

1. Introduction

In software development activities, source code examples are critical for concept understanding, applying fixes, extending software functionalities, etc. Code search increases developers' productivity and reduces duplication of effort. The goal of code search is to retrieve code fragments from a large code corpus that most closely match a developer's intent, which is expressed in natural language. Previous studies have revealed that more than 60% of developers search code in Q&A sites every day (Hoffmann et al., 2007). As online software Q&A sites (e.g., StackOverflow, 0000; Github, 0000; Krugle, 0000) contain millions of open source projects with code fragments, many search engines are designed to retrieve code solutions for natural language (NL) queries issued by developers. Unfortunately, these search engines often return with unrelated or other language codes in search results (Sadowski et al., 2015), even when reformulated queries are provided (Carpineto and Romano, 2012).

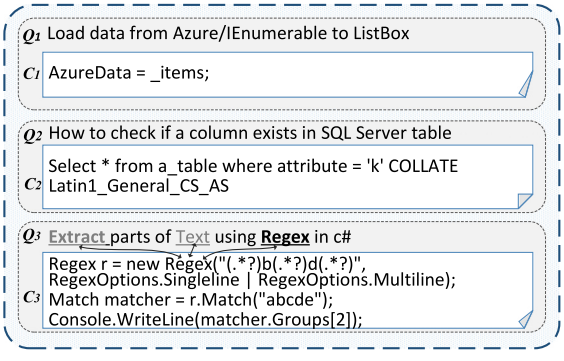

Fig. 1. Example of some user queries $Q_i (i = 1, 2, 3)$ in natural language and their code solutions $C_i (i = 1, 2, 3)$ from StackOverflow. Our goal is to automatically search these code solutions.

Recent work from both academia and industry has enabled more advanced code search using different techniques. Consider the examples in Fig. 1, we collect three different NL queries and their code solutions from Post#133031 (StackOverflow, 2009a), Post#19370921 (StackOverflow, 2012b) and Post#11621614

* Corresponding authors.
E-mail addresses: hoogang@whu.edu.cn (G. Hu), pengm@whu.edu.cn (M. Peng), e0261914@u.nus.edu (Y. Zhang), xieq@whu.edu.cn (Q. Xie), ymt@whu.edu.cn (M. Yuan).

https://doi.org/10.1016/j.jss.2020.110773
0164-1212/

(StackOverflow, 2012a) in StackOverflow. There are still several challenges for previous work as follow:

(1) **Common features mismatch using IR-based techniques**. Previous methods for code search applied with information retrieval (IR) techniques, but most of them depend greatly on the quality of matching terms contained in both NL queries and source codes (Haiduc et al., 2013). As NL query and source code in the $\langle Q_1, C_1 \rangle$ (Shown in Fig. 1) pair is heterogeneous, they may not share enough common lexical tokens, synonyms, or language structures, especially in short queries. Although most approaches provide effective ways for query reformulation (Paik et al., 2014; Lu et al., 2018; Sirres et al., 2018) (e.g., query expansion, text paraphrase), over specific queries return no effective results (Grechanik et al., 2010). Moreover, these extractive methods cannot handle irrelevant/noisy keywords in NL queries effectively. In fact, most NL queries and source codes may only be semantically related, as similar in cross-language text retrieval or machine translation. Thus, deep-learning-based methods (Iyer et al., 2016; Huo et al., 2016; Chen and Zhou, 2018) are further proposed to mine the semantic relevance between NL query and source code. Similarly, we designed NJACS for code search in software Q&A sites using a two-way attention-based neural network, which can perform NL semantic query for various programming languages.

(2) **Lack of integral code representation via structure embeddings**. More recently, various neural models are applied to learning the unified representation of both source code and NL query, such as code annotation (Huo et al., 2016), bug localization (Chen and Zhou, 2018) and software repair (White et al., 2016), etc. Similarly with code search, Gu et al. (2018) propose a bi-modal neural network named CODEnn, but limit its application to Java code fragments. CODEnn relies on extracting subelements (including method names, tokens, and API sequences Gu et al., 2016), thus cannot generate the overall semantic representation of the code structure. Unfortunately, like the $\langle Q_2, C_2 \rangle$ (Shown in Fig. 1) pair belongs to SQL domain with no method types, these code-split-based embedding methods (Paik et al., 2014; Grechanik et al., 2010) may not be suitable for code fragments of other structures and program types. Thus, compared to CODEnn, Sachdev et al. (2018), Yao et al. (2019) and Cambronero et al. (2019) present the different supervised neural models, labeled NCS, CoaCor and UNIF respectively that successfully learned code embeddings and integral code representation using corpora of query-code mappings. Nevertheless, these methods use one-hot representation (Turian et al., 2010), pre-trained word2vec embeddings (Mikolov et al., 2013a) or randomly initialized word embeddings (Gu et al., 2018; Yao et al., 2019) for pre-training neural networks, which lack of introducing external pre-training information to improve model performance similar to BERT (Devlin et al., 2018) used in NL processing (NLP). Unlike them, NJACS introduces pre-trained structure embeddings to further capture more structure information to enhance the integral code semantic representation.

(3) **Lack of modeling attention focus and structure information**: Most existing neural approaches (Huo et al., 2016; Chen and Zhou, 2018; Gu et al., 2018; Yao et al., 2019; Cambronero et al., 2019; Hu et al., 2018) utilize RNNs (Hochreiter and Schmidhuber, 1997) (Recurrent Neural Networks), CNNs (Pattanayak, 2017) (Convolutional Neural Networks) or DQNs (Mnih et al., 2013) (Deep Q-learning Networks) based bi-modal network for code search. They generally ignore the joint attention information between NL query and source code, thus cannot effectively capture deeper semantic matching signals. As for the $\langle Q_3, C_3 \rangle$ (Shown in Fig. 1) pair, understanding the focus of NL queries (e.g., important terms "extract", "regex" and "text") is helpful for retrieving more relevant codes that talk about "text extraction" problem. However, such methods do not explicitly model query focus (attention links are shown in Fig. 1). Recent studies (Iyer et al., 2016; Devlin et al., 2018; Wan et al., 2018) have also proved that attention mechanism can be successfully applied in code summarization (Iyer et al., 2016; Wan et al., 2018) and various NLP tasks like Machine Reading Comprehension (MRC) (Devlin et al., 2018). Besides, many other approaches (Huo et al., 2016; Gu et al., 2018; Sachdev et al., 2018; Cambronero et al., 2019) fail to consider the potential structure information of source code, which carries additional semantics to the program functionality besides the lexical terms. For NJACS, its two-way attention mechanism can capture sufficiently matching terms to align and focus the more structural informative parts of source code to NL query.

To address aforementioned issues, we propose a novel neural network named NJACS for code search in software Q&A sites, which leverages two attention mechanisms as global attention and attentive pooling. To solve the problem of common features mismatch, NJACS learns an enhanced joint representation of NL query and source code, which captures the semantics in lexicon along with code structure and introduces focus information between NL query and code fragment. In NJACS, the desired codes can be retrieved from most common queries without frequent reformulation.

To the best of our knowledge, we are the first to propose joint-attention-based code search. The main contributions of our work are as follows.

- We propose NJACS, a neural joint attention network that utilizes two-way attention mechanism to improve unified representation learning from both natural language and programming languages for retrieving relevant code solutions in software Q&A sites.
- We design particular code structure embeddings for pre-training operations with respect to different programming languages and code structures, which is able to capture semantics of the program from both lexical and program structural perspectives.
- We construct large-scale software repositories containing almost 10 million pairs and 200 sampled examples with most candidate codes, all extracted from use-case scenarios like StackOverflow, and conduct comprehensive experiments on the repositories.

The rest of this paper is organized as follows. The next section gives the definitions and motivation. Sections 3 and 4 describes the detailed design of the structure embeddings and the NJACS model, respectively. Experiments are shown in Section 5. Sections 6 and 7 gives the evaluation results and discussion. Section 8 presents the related work. Finally, we conclude the paper in Section 9.

2. Definitions & motivation

2.1. Problem formulation

In our work, we focus on the desired code[1] (also named as code solution) which can solve the problem for the questioner. Code search for software Q&A sites can be described as follows: Given a set of common retrieved lists, where each NL query $q_i \in Q$ comes together with its list of candidate code fragments $Codes_i = \{C_{i1}, \ldots, C_{in}\}$. The goal is to build a model $\zeta(\cdot)$ that generates an optimal ranking R, s.t. the code solution appears at the top of the candidate list.

[1] Following Iyer et al. (2016) and Yang et al. (2016), although there can be multiple candidate code solutions in Q&A sites like StackOverflow, we only consider the desired one because of its verified quality by public recognition.

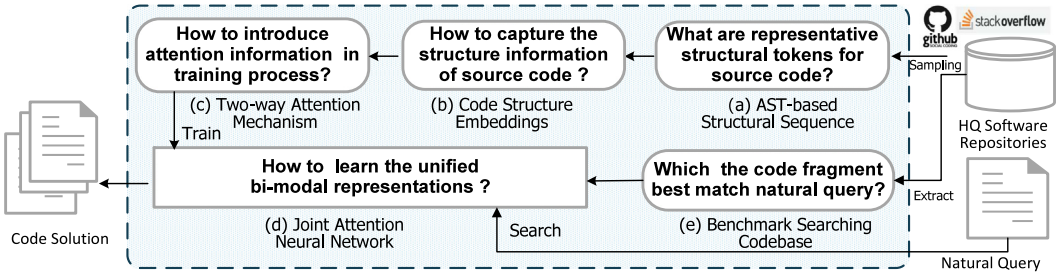

Fig. 2. The overall workflow of the problems and our solutions in NJACS.

More formally, the task is to learn a ranking function:

$$\zeta_w : Q \times Codes_i \rightarrow R \quad (1)$$

The model's parameter w is learned during the training.

2.2. Basic definitions

We use the following definitions for NJACS in our work.

•**Source Code Fragment**: A contiguous set of code lines found in the software Q&A sites. We propose this terminology to differentiate code snippets that are leveraged during the search process by our approach.

•**Source Code Snippet**: A subsequence of different parts in the code fragment. We take the sequence of tokens representing (full or partial) code fragment at different granularities as source code snippet: e.g., a method body, an API sequence or an arbitrary sequence of statements.

•**Code Fragment Sequence**: A sequence of words in code fragment. We extract sequence of tokens in code fragment according to camel-case, separators (e.g., "_") and regular-expressions, and remove the duplicate and stop words. Then we take the sequential tokens as code fragment sequence.

•**Code Structural Sequence**: A sequence of structural tokens in code fragment. We replace the literals (e.g., identifier names, variable names) denoting the code structures and their types with uniform structural tokens based on Abstract Syntax Tree (AST) parser (Iyer et al., 2016). Then we take the sequential structural tokens as code structural sequence.

•**Q&A Programming Posts**: A set of query-code (QC) mappings with context description parsed from software Q&A sites. We use different tags to parse the Q&A programming posts, each of which comprises the short NL queries below with the detailed context descriptions, and several responses.

•**High-quality Q&A Datasets**: A set of Top-λ QC mappings mined from Q&A programming posts. We adopt a CodeMF (Hu et al., 2020) framework to mine Top-λ scored posts and extract their NL queries and accepted answers (using ⟨title⟩ and ⟨accepted⟩ tags) as high-quality Q&A datasets.

•**Single Candidate QC Pairs**: An aligned corpus extracted from high-quality Q&A datasets. In the high-quality Q&A datasets, we extract the desired code fragments from the answers and their NL queries to construct one-to-one QC mappings. Then we pair the query with the desired one and other randomly sampled codes as single candidate QC pairs.

•**Multiple Candidate QC Pairs**: An annotated corpus extracted from high-quality Q&A datasets. In the high-quality Q&A datasets, we collect their query statements and all candidate code fragments of the corresponding answers (using all ⟨code⟩ tags) as multiple candidate QC pairs.

•**StackOverflow Question-Code Pairs**: A kind of systematically mined QC pairs extracted from StackOverflow based on BiV-HNN (Yao et al., 2018). They are for SQL and Python domain and aim at solving "how-to-do-it" NL queries since their answers are more likely to be code solutions.

•**Benchmark Search Codebase**: Benchmark search codebase are the top 200 mappings with most candidate codes collected from the multiple candidate QC pairs, which can be used as the prediction datasets for better evaluating NJACS.

2.3. Challenges and solutions

Recently, most advanced methods (Chen and Zhou, 2018; Gu et al., 2018; Sachdev et al., 2018; Yao et al., 2019; Cambronero et al., 2019) focus on using neural models to successfully perform semantic code search. But in reality, several major problems remain unresolved: (1) They do not extract continuous structural tokens other than lexical terms to represent semantic of code fragments. (2) They do not merge the joint attention information of both query and code representations for semantic matching modeling. (3) They do not learn the pre-trained structure embeddings of code that capture their semantic similarities, combinations, and analogies. (4) They do not embed the integral code representation, while aggregating the independent code subelements into vectors.

Inspired by the summary of the shortcomings of the existing work described above, we first postulate that software Q&A sites constitute multilingual software repositories with a wealth of different code fragments (often find in answers) to match the NL queries. Then, we propose a method of code search, NJACS, based on the idea of joint attention feature extraction and deeply excavating code information, used to solve these problems with a common paradigm.

Fig. 2 illustrates the existing problems and our technical solutions for code search in software Q&A sites. Given the NL query and the search codebase collected from software Q&A sites, NJACS outputs the code solution. Specific resolution steps are shown in Fig. 2: (a) What are representative structural tokens for source code ? NJACS detects special identifiers based on multilingual AST parsers and extracts as structural sequence representing the code structure. (b) How to capture the structure information of source code ? NJACS learns the code structure embeddings from millions of Q&A programming data for pre-training model. (c) How to introduce attention information in training process ? NJACS adopts position-shared weighting scheme for combining different matching signals and incorporates the importance weights of each other's learning representation. (d) How to learn the unified bi-modal representations ? NJACS trains a joint attention neural network on almost 10 million different QC pairs to learn unified features from NL queries and code fragments. (e) Which the code solution best match the NL query ? NJACS ranks the candidate code fragments by measuring how well them match with the given NL query (in comparison with other candidate codes).

3. Learning structure embeddings

In NJACS, we adopt an advanced distributed representation technique from NLP, which learns *structure embeddings* - continuous distributed vectors for grouping similar code libraries or

Table 1
An example of a Q&A programming post (a) which contains three parts: natural query (P_Q), context description (P_C) and desired answers (P_A). The extracted Q&A contextual pair contains the appropriate codes and their contextual description (b), the words in processed Q&A contextual pair with code structural sequence after customized bilingual text tokenizing (c), and a piece of corpus of pre-processed structural sequence (d). '[c]...[/c]' represents a code fragment, and '\n' represents a line break in code fragment. 'NR' indicates that the code fragment do not meet the requirement in length and type, and 'NP' indicates that the code fragment is not parsable. Except for 'NR' and 'NP', we call the others appropriate code fragments.

Q&A programming post		Q&A contextual pair extraction	
P_Q	Using BeautifulSoup,how do I iterate over all embedded text?	P_Q	Using BeautifulSoup,how do I iterate over all embedded text?
P_C	Let...HTML: \n [c] Hello there Hi \n [/c] \n becomes \n [c] Hll thr!H![/c] \n I...this?	P_C	Let...HTML: \n [c]'NR' [/c] 'NP' [c]'NR'[/c] \n I...this?
P_A	Suppose...variable [c]test_html[/c] has...content: \n [c]<html>...</html> \n[/c] \n Just...this: \n [c]from BeautifulSoup import BeautifulSoup \n test_html = load_html_from_above() \n soup = BeautifulSoup(test_html) \n for t in soup.findAll (text=True): \n text = unicode(t) \n for vowel in u'aeiou': \n text = text.replace (vowel, u'') \n t.replaceWith(text) \n print soup \n[/c] \n That prints: \n [c]<html>...</html>\n[/c] \n Note... untouched.	P_A	Suppose...variable [c]'NR'[/c] has...content: \n [c]'NR'[/c] \n Just...this: \n [c]from BeautifulSoup import BeautifulSoup \n test_html = load_html_from_above() \n soup = BeautifulSoup (test_html) \n for t in soup.findAll (text=True): \n text = unicode(t) \n for vowel in u'aeiou': \n text = text.replace (vowel, u'') \n t.replaceWith(text) \n print soup \n [/c] \n That prints: \n [c]'NR'[/c] \n Note ... untouched.
(a)		(b)	

Bilingual text tokenizing for the Q&A contextual pair	
P_Q	use beautiful_soup , how do i iterate over all embed text ?
P_C	let...html : 'NR' 'NP' 'NR' i...this
P_A	suppose...variable 'NP' have...content 'NP' just...this: from beautiful_soup import beautiful_soup newline var = load_html_from_above ( ) newline var = beautiful_soup (var) newline for var in var . find_all (var = true) : newline var = unicode (var) newline for var in string : newline var = var . replace (var , string) newline var . replace_with (var) newline print var that print : 'NP' note...untouched.
(c)	

Corpus of pre-processed structural sequence
use ... text ? let ... html : 'NR' 'NP' 'NR' i ... this suppose ... variable 'NP' have ... content 'NP' just ... this: from beautiful_soup import beautiful_soup newline var = load_html_from_above ( ) newline var = beautiful_soup (var) newline for var in var . find_all (var = true) : newline var = unicode (var) newline for var in string : newline var = var . replace (var , string) newline var . replace_with (var) newline print var that print : 'NP' note ... untouched.
(d)

tokens in AST nodes. Based on pre-trained structure embeddings, NJACS can introduce more structure information in neural network training to improve code search performance. This section describes the detail steps for learning structure embeddings.

3.1. Techniques of word embeddings

Word embeddings are widely used with a great success in NLP tasks to boost the accuracy of downstream learning tasks. Their main advantages, compared to local representations (e.g., one-hot encoding, bag of words), are that distributed values along dimension can capture semantic and syntactic similarities of words to reduce the dimension of text representation modeling. These language models (e.g., Word2Vec Mikolov et al., 2013a, FastText Bojanowski et al., 2017 and GloVe Pennington et al., 2014) generally compute the vector representation of words using a two-layer dense neural network that can be unsupervised trained on a large NL corpus, which can embed semantic correlations for words that shares similar contexts in the corpus.

Notice that word embedding models are originally designed for NLP, our corpus collected from software repositories are different from NL description. Nevertheless, we rely on the assumption that the distributional hypothesis also holds for program text, i.e. tokens that occur in the same context of source code also have related meanings. Recent work (Allamanis et al., 2015; Theeten et al., 2019) also supports the same hypothesis. Similar to the famous NLP example of: vec("king") $-$ vec("man") $+$ vec("woman") $\approx$ vec("queen") (Mikolov et al., 2013b), Alon et al. (2019) learns the distributed vectors of software libraries ("code embeddings") for specific programming languages, such as: "*receive* is to send as *download* is to: *upload*". With the pre-trained code embeddings, these related work (Alon et al., 2019; Python2vec, 0000; LeClair et al., 2018) has successfully improved the performance in predicting method names, recommending codes and classifying software libraries.

Unlike traditional methods (Sachdev et al., 2018; Cambronero et al., 2019; Theeten et al., 2019) which extract source codes as code fragment sequences for learning code embeddings, we parse them into the structural sequences to learn structure embeddings that leverage the structure nature of source code. Consider the examples in Post#32100 (StackOverflow, 2008), Post#25444735 (StackOverflow, 2014) and Post#3958346 (StackOverflow, 2010), three similar code fragments from SQL queries " *SELECT MAX(col) \n FROM table \n WHERE col < (SELECT MAX(col) \n FROM table)*", " *SELECT MAX(field) FROM table WHERE field < (SELECT MAX(field) FROM table)*" and " *SELECT MAX (EmpSalary) FROM employee WHERE EmpSalary < (SELECT MAX(EmpSalary) FROM employee)*" parsed as "select max(col0) from tab0 where col0 < (select max(col0) from tab0)" can better unify the code structures. From the above observations, besides the important identifiers (e.g., "select", "max"), most variable names and special characters (e.g., "field", "employee") may appear less frequently. Refer to previous related work (Iyer et al., 2016; Bui and Jiang, 2018), we strip out all comments of code fragments and to avoid being context specific. Then we normalize the token streams to remove semantic-irrelevant tokens (e.g., some variable names) and replace structural and some semantic tokens (e.g., syntactic node types, data types, and method signatures) with tokens denoting their types. In this case, we pre-train the structure embeddings for improving performance in code search.

The process of constructing structure embeddings can be summarized as follows: (1) We first extract the Q&A contextual text pairs with appropriate code fragments and NL descriptions from

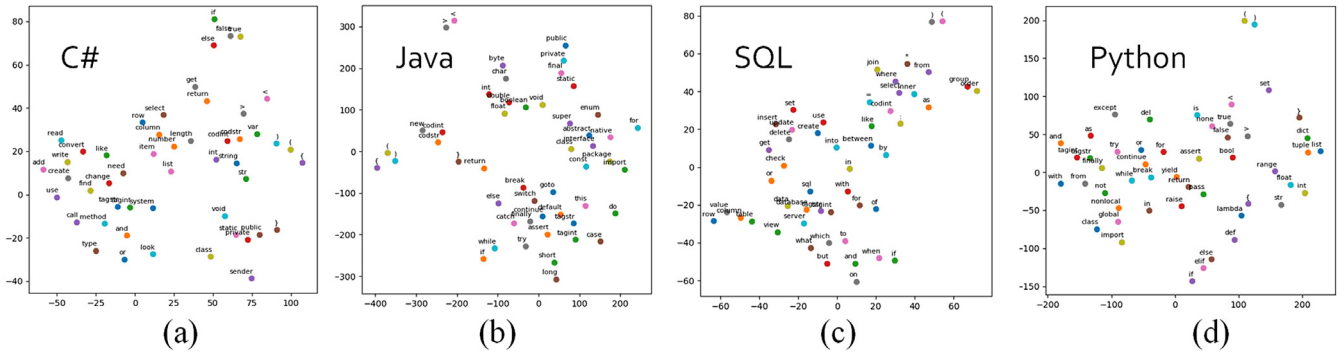

Fig. 3. Examples of 2D-visualization for structure embeddings in C# (a), Java (b), SQL (c) and Python (d) respectively.

the programming posts; (2) We then normalize the Q&A contextual text pairs into the code structural sequences combined with their mixed NL description as training corpora; (3) We finally use a variant of word2vec model named fastText (Theeten et al., 2019) to learn structure embeddings for pre-training neural code search models.

3.2. Collecting embedding corpus

We collect programming datasets from StackOverflow, a popular Q&A site for posting programming issues among software developers. Anonymized versions of XML database can be downloaded from StackExchange (0000). Using the regular expression (e.g., Tags = ".*c#.*") to match different tags, we can parse the Q&A posts for various programming languages. In our datasets (Downloaded in Step 2017), we collect 642 662, 602405, 699 700 and 434 597 posts for C#, SQL, Java and Python respectively. These Q&A posts are further normalized as code structural sequences to train structure embeddings. Table 1 shows an example of converting Post#830997 (StackOverflow, 2009b) to a code structural sequence in Python domain. We first extract the Q&A contextual text pairs from all posts and replace the inappropriate code fragments with "NR" and "NP". Then, we use the different customized bilingual tokenizers (Described in Section 5.4) to extract NL words and software-specific tokens of code fragment, and we transform the remaining tokens from camelcase names to underscored words by Python Inflection (Inflection, 0000) library. Finally, each Q&A post is processed as a piece of corpus of structural sequence for training structure embeddings. In total, we learn the structure embeddings for C#, SQL, Java and Python containing 238 344 608, 234 480 598, 180 891 654 and 178 602 792 words respectively.

3.3. Building structure embeddings

In our implementation, we use the above four kinds of large-scale codebases, including C#, Java, SQL and Python respectively. To train the structure embeddings at all granularity (i.e., identifiers, method signatures) of code fragment, we employ the fastText model with Python Gensim (Gensim, 0000) library. Table 2 shows the hyper-parameter settings during the training process. Related work (Sachdev et al., 2018; Cambronero et al., 2019) often exploits the word2vec model with a window size of 5. But in order to learn contextual similar words that are farther away in the flat structure of source code, and analyze their distribution effect in different window sizes, we finally set the window size of 10. Unlike NLP where the number of words is generally fixed, programmers often declare their own identifiers in their source codes. Thus, the number of code tokens in programming language is unlimited (Peng et al., 2015). Throughout the process, our code search corpus is derived from the original embedding corpus on the large-scale code base used for training structure embeddings. Our provided structure embeddings with the words which can completely cover the words of training corpus used in code search, so they can incorporate more pre-trained words with external word-level structure information for improving code search performance.

Table 2
Hyper-parameter settings for training structure embeddings.

Window size	Number of iterations	Vector dimension	Subsampling	Low-frequency word	Negative sampling
10	20	300	1e−5	2	5

Table 3
Examples of structure embeddings with semantic similarities.

Domain	Token	Contextual similar words (Top 4 results)
C#	convert	idconvert, sconvert, nconvert, deconvert
Java	sender	ssender, qsender, amtsender, smssender
SQL	create	ccreate, docreate, tocreate, docreate
Python	join	joinz, sjoin, pjoin, joinq

As our structure embeddings space consists billions of points (one for each token in vocab), in order to explore the spatial distribution of tokens and their local neighborhoods, we project the embedding space into 2 dimensions using t-SNE (Maaten and Hinton, 2008), which is a dimensionality reduction technique that aims to preserve local distances among points. To better present the distribution effect of structure embeddings, we select some specific identifiers, such as the keywords (e.g., "lambda", "while", "break") in Python domain. Fig. 3 shows the examples of 2D-visualization of structure embeddings for C#, Java, SQL and Python. We observe that the two groups of keywords ("while", "break" and "continue"; "if", "elif" and "else") in Fig. 3(d) are very close neighbors which are frequently used together or offering similar functionality in source code and NL description. Table 3 lists some examples of structure embeddings with semantic similarities in C#, Java, SQL and Python.

4. NJACS: Neural joint attention code search using structure embeddings

As described in Section 3, we learn the structure embeddings. In this section, we discuss the background of bi-modal embedding of heterogeneous data and introduce the neural techniques used over structure embeddings in NJACS.

4.1. Bi-modal embedding of heterogeneous data

Bi-modal embedding (also known as dual-modal embedding and joint embedding) is a supervised neural embedding technique that maps heterogeneous data to the unified vector space to computerize their semantic similarities. Suppose there are two heterogeneous data sets X and Y, the computation of bi-modal embedding can be formulated as:

$$X \xrightarrow{\phi} V_X \rightarrow J(V_X, V_Y) \leftarrow V_Y \xleftarrow{\psi} Y \tag{2}$$

where X and Y belong to heterogeneous data, ϕ and ψ are the embedding functions that map X and Y into a unified d-dimensional vector $(V_X, V_Y \in R^d)$ space; $J(\cdot)$ is a similarity measurement (e.g., cosine similarity) to score the matching degrees of X and Y, used to learn the mapping functions. Through bi-modal embedding, heterogeneous data can be easily correlated through their embedding vectors.

Bi-modal embedding techniques have been widely used in many tasks (Frome et al., 2013; Xu et al., 2015). For example, in software engineering, (Huo et al., 2016; Gu et al., 2018; Sachdev et al., 2018; Yao et al., 2019; Cambronero et al., 2019; Yao et al., 2018; Xu et al., 2015; Zhong et al., 2017) employ bi-modal neural models to learn the unified representations of cross-language mappings, such as CoaCor (Yao et al., 2019), BiV-HNN (Yao et al., 2018) and Seq2SQL (Zhong et al., 2017). (Sachdev et al., 2018) presented an unsupervised approach named NCS, which uses the bag-of-words-based network to embed code fragments and NL queries. And they recently proposed an improved UNIF (Cambronero et al., 2019), which replaces TF–IDF (Ramos, 2003) weights with a learned attention-based weighing scheme. But these above approaches often average (either simple or weighted) code embeddings to obtain the code representation without involving token order. In contrast, Iyer et al. (2016), White et al. (2016) and Yao et al. (2019) proposed the sequence-based embedding networks to learn the query and code representations. It is confirmed that bag-of-words-based UNIF outperforms sequence-based CODEnn (Cambronero et al., 2019), because CODEnn adopts the extracted special subelements (including method name, API sequence and tokens) of the code snippet to represent Java method instead of using overall sequential tokens. However, recent sequence-based bi-modal embedding techniques achieve state-of-the-art code search performance, such as CoaCor (Yao et al., 2019) and DeepCom (Hu et al., 2018). Thus, these sequence-based bi-modal embedding neural networks can still be better used for code search.

4.2. Overall architecture introduction

Inspired by existing bi-modal embedding techniques (Huo et al., 2016; Gu et al., 2018; Cambronero et al., 2019; Allamanis et al., 2015), we propose a novel joint-attention-based code search named NJACS for software Q&A sites. NJACS utilizes a two-way attention mechanism to embed the code fragments and NL queries into the nearby vectors of a unified space for measuring their semantic similarity.

Fig. 4 illustrates the overall architecture and the detailed structure of NJACS. As shown in Fig. 4(A), NJACS is composed of four layers (*Structure Embedding Layer*, *Feature Embedding Layer*, *Joint Attention Layer*, and *Similarity Matching Layer*), each corresponding to a component of joint attention embedding:

•**Structure Embedding Layer**: As shown in Fig. 4(B-a), the layer that first embeds the both words of NL queries and tokens of code fragments into structure vectors by lookup the matrix of pre-trained structure embeddings.

•**Feature Embedding Layer**: As shown in Fig. 4(B-b), the layer that next uses the sequence-based embedding RNNs to learn the underlying semantic representation by summarizing the sequential structure vectors of tokens and words from both directions.

•**Joint Attention Layer**: As shown in Fig. 4(B-c), the layer that further introduces global attention to compute the aligned matrix and generate the different directions of attentive pooling based weights for each other's representation.

•**Similarity Matching Layer**: As shown in Fig. 4(B-d), the layer that finally computes the similar values of the bi-modal attention-based vector representations to score the matching degree between NL queries and code fragments.

Detailed design of four layers are described as follows.

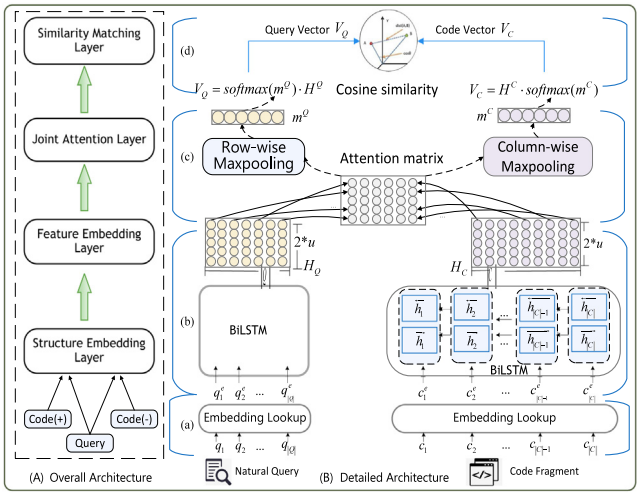

Fig. 4. Architecture of NJACS, which is trained on the triplets comprised of ⟨positive code fragment (Code+), natural query (Query), negative code fragment (Code-)⟩.

4.2.1. Structure embedding layer

We assume that the corpus of code fragments and their NL query descriptions is available for training. We denote this corpus as a collection of (Q, C), where C is an AST-based parsed code structural sequence $Q = q_1, \ldots, q_{|Q|}$ from a NL query and C is a sequence of words $C = c_1, \ldots, c_{|C|}$ from its corresponding code fragment. Instead of using one-hot representation, pre-trained word2vec embeddings, randomly initialized word embeddings and uniform or gaussian random embeddings, we use the pre-trained structure embeddings to incorporate word-level structure information. To get the structure embeddings of the sequential inputs, we first perform embedding lookup as follows:

$$q_i^e = \mathrm{E} \cdot q_i (i \in [1, |Q|]); \; c_j^e = \mathrm{E} \cdot c_j (j \in [1, |C|]) \tag{3}$$

where $\mathrm{E} \in \mathbb{R}^{d \times |V|}$ is the matrix of pre-trained structure embeddings (see in Section 3.3), d is the vector dimension of the structure embeddings ($d = 300$) and V is a fixed-sized word vocabulary (for C#, SQL, Java, Python respectively, $|V| =$ 238 344 608, 234 480 598, 180 891 654, 178 602 792). $q_i^e \in \mathbb{R}^d$ and $c_j^e \in \mathbb{R}^d$ indicate the output of structure embedding vector for word q_i and token c_j respectively.

4.2.2. Feature embedding layer

After mapping the QC sequential words into the structure embeddings, we adopt the commonly used bidirectional Long Short-Term Memory (BiLSTM) (Schuster and Paliwal, 1997) based RNN to learn the underlying representation by summarizing the contextual information from both directions. First, we describe the advantages of LTSM by embedding sequential inputs such as sentences using their internal memory. Consider a sequential

input with T words, LSTM embeds it through three multiplicative layers: an input layer which reads the word embedding of each word $x_t(t \in [1, T])$ at every time step t, a recurrent hidden layer which recurrently computes and updates a hidden state h_{t-1} after reading each input, and an output layer which utilizes the hidden state h_t for specific tasks. For simplicity, we denote the above calculation as below (the memory cell for updating state is omitted):

$$h_t = \text{LSTM}(x_t, h_{t-1}) \tag{4}$$

The vanilla LSTM's hidden state h_t takes information from the past, knowing nothing about the future. Instead of LSTM, our proposed NJACS model incorporates the improved bidirectional LSTM (BiLSTM) containing both forward LSTM and backward LSTM, which captures past and future information at each step in two directions. The forward hidden state $\overrightarrow{h_t}$ and backward hidden state $\overleftarrow{h_t}$ at each time step t for sequential inputs can be express as follows:

$$\overrightarrow{h_t} = \text{LSTM}\left(x_t, h_{t-1}^{\rightarrow}\right); \overleftarrow{h_t} = \text{LSTM}\left(x_t, h_{t+1}^{\leftarrow}\right) \tag{5}$$

Finally, the two hidden states $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ are concatenated to form the final hidden state h_t as:

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}] \tag{6}$$

where $[a; b]$ represents the concatenation of two vectors. Let the hidden unit number for each unidirectional LSTM be u, the BiLSTM-based embedding vectors updated from T-length sequential inputs can be note as $H \in \mathbb{R}^{T\times 2u}$:

$$H = [h_1, \ldots, h_t, \ldots, h_T] \tag{7}$$

Followed the process of transforming the n-length input vectors to the sequences of embedding vectors H, the BiLSTM-based embedding matrices of Q and C are denoted as $H \in \mathbb{R}^{|Q|\times 2u}$ and $H \in \mathbb{R}^{|C|\times 2u}$ that keep more historical encoding information for learning bi-modal representation.

4.2.3. Joint attention layer

Most previous code search models (Huo et al., 2016; Gu et al., 2018; Yao et al., 2019; Cambronero et al., 2019) use the pooling-based hidden states of all steps (Gu et al., 2018) or final hidden state of the last step (Yao et al., 2019) of RNNs directly to generate the final vector representations for NL query and code fragment respectively. Instead of learning QC representations independently, we leverage a two-way attention mechanism (including global attention and attentive pooling) to improve the computation of QC representations. Specifically, we first use the global attention mechanism to create an aligned joint attention matrix from the QC embedding vectors, and then apply the attentive pooling operation to generate the attention weights both in row and column directions. Finally, we multiply the attention weights with their embedding vectors to compute each other's vector representations of QC pairs.

Particularly, given the embedded vector $h_t^q \in \mathbb{R}^{2u}$ in the sequential H_Q of NL query and the embedded vector $h_t^c \in \mathbb{R}^{2u}$ in the sequential H_C of source code at each time step t, we use the global attention mechanism to construct an aligned joint attention matrix M of the QC mappings. The attention matrix $M \in \mathbb{R}^{|Q|\times|C|}$ can be formulated as below:

$$M = A^T \tanh\left(W_q h_t^q + W_c h_t^c\right) \tag{8}$$

where $W_q \in \mathbb{R}^{a\times 2u}$, $W_q \in \mathbb{R}^{a\times 2u}$, $A \in \mathbb{R}^a$ are the matrix of trainable parameters, a is the dimensions of attention, tanh is the activation function. The aligned matrix M contains the scores of a soft alignment between the hidden vectors h_t^q and h_t^c of each token in code fragment and NL query.

Next, we adopt the attentive pooling mechanism which derives the pooling-based projection vectors in different directions to guide the attention weights for QC each other's representations. Max-pooling is an operation that selects the maximum value in each fixed-sized region over a matrix. Using the row-wise and column-wise max-pooling over M, we can obtain the important score vectors $m^Q \in \mathbb{R}^{|Q|}$ and $m^C \in \mathbb{R}^{|C|}$, respectively. Formally, the kth elements of the vectors m^q and m^c are computed as follows:

$$m_k^q = \max_{1<i<|Q|}[M_{k,i}]; m_k^c = \max_{1<j<|C|}[M_{j,k}] \tag{9}$$

Each element m_k^q is treated as an importance score for the kth query-to-code alignment words. Likewise, each element m_k^c can be treated as an importance score for the kth code-to-query alignment words. In order to assign specific weights for QC, we employ the softmax activation function to compute the attentive weight vectors $\zeta_Q \in \mathbb{R}^{|Q|}$ and $\zeta_C \in \mathbb{R}^{|C|}$ as below:

$$\zeta^Q = \text{softmax}(m^Q); \zeta^C = \text{softmax}(m^C) \tag{10}$$

Finally, we get the attention-based vector representations $v_Q \in \mathbb{R}^{2u}$ and $v_C \in \mathbb{R}^{2u}$ which are multiplied by their corresponding attentive weights ζ^Q and ζ^C and BiLSTM-based output embedding vectors H^Q and H^C respectively:

$$v_Q = \zeta^Q H^Q; v_C = \zeta^C H^C \tag{11}$$

4.2.4. Similarity matching layer

We have described the transformations that map the NL query Q and code fragment C into the vectors v_Q and v_C. Since we want the vectors of NL query and code fragment to be jointly embedded, we measure the similarity between the two vectors. Similar to Paik et al. (2014) and Huo et al. (2016), we use the cosine similarity $cos(v_Q, v_C)$ for the measurement, which is defined as:

$$cos(v_Q, v_C) = \frac{v_Q^{\mathrm{T}} v_C}{\|v_Q\| \cdot \|v_C\|} \tag{12}$$

where $a^{\mathrm{T}}b$ represents the multiplication of two matrices via their transpose. The higher the similarity, the more related the code fragment is to the NL query.

Overall, NJACS takes the QC pair $\langle Q, C\rangle$ as input and predicts their cosine similarity $cos(v_Q, v_C)$.

4.3. Model training

Now we present how to train NJACS with loss function. Similar to Gu et al. (2018), we construct a triple of $\langle Q, C^+, C^-\rangle$ as a training instance: for each natural query Q, there is a positive code fragment C^+ (a desired code) as well as a negative code fragment C^- (not a desired code) randomly chosen from the sampled codes. These triplets come from our collected training corpora in Section 5.3.1. Our goal is to learn a representation function f with trainable parameters which can make the score of positive $\langle Q, C^+\rangle$ pairs larger than negative $\langle Q, C^-\rangle$ pairs.

$$f(Q, C^+) > f(Q, C^-), \forall Q, C^+, C^- \tag{13}$$

When training on the set of $\langle Q, C^+, C^-\rangle$ triples, NJACS predicts the cosine similarities of both $\langle Q, C^+\rangle$ and $\langle Q, C^-\rangle$ pairs and minimizes the triplet ranking loss (Chen and Zhou, 2018; Gu et al., 2018):

$$l(\theta) = \sum_{\langle Q, C^+, C^-\rangle \in P} max(0, \varepsilon - cos(v_Q, v_{C^+}) + cos(v_Q, v_{C^-})) + \lambda\|\theta\| \tag{14}$$

Here, P denotes the training corpora of different QC pairs (see Section 5.3.1), θ denotes the model parameters of NJACS, $\varepsilon > 0$ is a constant margin, $\lambda \in [0, 1]$ is a regularization parameter, v_Q, v_{C^+} and v_{C^-} are the embedded vector representations of Q, C^+ and C^-, respectively.

5. Experiments

5.1. Q&A datasets collection

As described in Section 4.3, NJACS requires a large-scale corpus that contains NL queries and their code fragments. To the best of our knowledge, existing code search datasets are collected without a general standard, which generally are based on human annotation or crawled from Q&A forums and online sites. Unfortunately, most Q&A forums provide personal opinions of users which are often not adequately confirmed or outdated. According to the statistics, around 74% of Python and 88% of SQL programs in the Q&A forums are not directly parsable or runnable (Iyer et al., 2016; Yang et al., 2016) and irregular (Hu et al., 2019). Nevertheless, many of them usually contain critical information to answer a program issue. Thus, they can still be used in code search after being further cleaned processing. To prevent the bias of pseudo data with high-redundancy on experimental results, unlike (Iyer et al., 2016; Chen and Zhou, 2018; Hu et al., 2018; Wong et al., 2013) extracting the Q&A datasets based on the highest upvote or individual social features (e.g., upvote and pageview), we adopt our previous designed framework named CodeMF (Hu et al., 2020) to mine high-quality Q&A datasets. CodeMF can significantly reduce the data redundancy of Q&A datasets (Hu et al., 2020), and it is also applicable to various programming languages. Q&A datasets mining may be improved with more advanced features and other supervised algorithms, which is not the focus of this paper. Our evaluation datasets containing training corpora and benchmark search codebase for code search are extracted from the high-quality Q&A datasets.

In the following sections, we describe the detailed design of CodeMF for high-quality Q&A datasets mining and further prepare the evaluation datasets.

5.2. Unsupervised Q&A posts mining

We start by extracting multilingual Q&A posts as described in Section 3.2, and we follow Iyer et al. (2016) to identify the Q&A posts with *how-to* style queries. We use these query-cleaned posts as the source of Q&A datasets. In previous work (Peng et al., 2016b,a), most behavioral attributes (e.g., poster influence, tags, and URL) are used to extract high-quality microblogs. Along with such ideas, the quality of Q&A posts can also be measured by social features such as pageviews and upvotes, etc. Thus, we proposed CodeMF for Q&A datasets mining by considering various social feature fusion.

First CodeMF extracts all social features from L posts to construct a K-dimensions features matrix $\Gamma = (\gamma_1, \ldots, \gamma_L)^T \in \mathbb{R}^{L\times K}$. Then CodeMF applies missing value completion with average value and min–max normalization to scale feature $\gamma_x (x \in [1, L])$ to [0, 1], and derives a new K-dimensions features matrix $T \in \mathbb{R}^{L\times K}$. After that, CodeMF reduces the dimension of matrix T to get a D-dimensions features matrix $F \in \mathbb{R}^{L\times D}$ based on Kernel Principal Component Analysis (KPCA) (Xu et al., 2007) . Then CodeMF trains the dimension-reduced matrix F as D signals $F_d (d \in [1, D])$ and transforms them to the frequency-domain coefficients (high-frequency: $HF_d^q (q \in [1, Q])$, low-frequency: $LF_d^q (q \in [1, Q])$) in wavelet trees based on Q-layers ($Q \leq log_2 L+1$) Wavelet Transformation (WT) (Hu et al., 2020). Next CodeMF fuses all corresponding spliced frequency-domain coefficients (high-frequency: HS^q, low-frequency: HS^q) to new high-frequency and low-frequency coefficient vectors NHF^q, NLF^q as the following steps.

Assuming that the observed distribution of spliced coefficients can be described as a D component Gaussian mixture distribution, i.e., $f(m_{di}^q) \sim N(\mu_d^q, \delta_d^{2q})$, we can estimate the probability that the fusion coefficient $m_{di}^q (i = 1, \ldots, n^q)$ can be formulated as follows:

$$f(m_{di}^q) = \sum_{d=1}^{D} w_d^q f(m_{di}^q) \tag{15}$$

where n^q denotes the dimension of frequency-domain coefficients in qth layer, w_d^q satisfies $\sum_{d=1}^{D} w_d^q = 1$, $f(m_{di}^q)$ is the probability density functions of the ith component in each coefficient vector M_d^q in the example of the spliced coefficient M. Similar to Peng et al. (2016b,a), Expectation Maximization (EM) (Dempster et al., 1977) is used to estimate the feature-fusion weights w_d^q.

Followed the above process, after we obtain the feature-fusion weights wh_d^q and wl_d^q, these new coefficient vectors NHF_q and NLF_q are calculated as follows:

$$NHF_q = \sum_{d=1}^{D} wh_d^q HS^q;\ NLF_q = \sum_{d=1}^{D} wl_d^q LS^q \tag{16}$$

Next CodeMF reconstructs a new wavelet tree based on the previous decomposed structure using Inverse Wavelet Transformation (IWT) (Hu et al., 2020). CodeMF takes the top leaf node S of the wavelet tree as the comprehensive score of L posts. CodeMF ranks the L posts in descending order based on fusion score. We define the Top-λ QC mappings as high-quality datasets that are all relative. To balance the quality of corpus with providing more training corpus, high-quality corpus can be collected with different threshold settings (λ).

In the experiments, we filter some noisy Q&A posts with no queries and code responses, retain total 171 773, 156 432, 211 215 and 150 190 cleaned mappings for C#, Java, SQL and Python, and truncate 171 760, 156 432, 221 200 and 160 176 of them to support 4-layer based ($Q = 4$) CodeMF mining framework. Finally, we take the Top-λ (λ = 170 000, 150000, 210 000, 150 000) scored posts as high-quality Q&A datasets for C#, Java, SQL and Python respectively.

5.3. Mappings extraction for code search

5.3.1. Constructing code search corpora

After extracting the high-quality Q&A datasets, we adopt a random sampling strategy to collect the Single Candidate QC Pairs (*SicQC Pairs*) and extract the Multiple Candidate QC Pairs (*MucQC Pairs*). It is worth noting that NJACS focuses on more general code fragments, not code snippets. Among these corpora, we remove the code fragments with the length less than 20 and 10 in SicQC Pairs and MucQC Pairs to ensure enough code candidates. To further optimize the NJACS model, we scale our training corpora to a kind of systematically mined StackOverflow Question-Code Pairs (Yao et al., 2018) (*StaQC Pairs*). In SicQC Pairs, MucQC Pairs and StaQC Pairs, we label the desired code fragment (a standard code solution) as "1" and "0" (other code candidates) otherwise. In these three kind of QC Pairs, we divide 80% of them for training, 10% for validation and 10% for testing to develop and optimize the code search models. Statistics of SicQC Pairs, MucQC Pairs and StaQC Pairs are summarized in Tables 4–6. Moreover, we try to obtain more original public QC pairs (not shared online in processed types) from different software Q&A sites for experiment, but (Sachdev et al., 2018; Cambronero et al., 2019) do not share them. Gu et al. (2018) provided the Q&A dataset from GitHub for code search, but they did not provide enough large-scale codebase for training corresponding structure embeddings. Our proposed the NJACS model for code search benefits from using structure embeddings. In addition, Huo et al. (2016) and Yin et al. (2018) shared their own different corpora, but they did not share the corresponding data processing methods that may be difficult to achieve the original desired results.

Table 4
Statistics of Single Candidate QC (SicQC) pairs.

Data types	C#		Java		SQL		Python	
Top-K size	170000		150000		210000		150000	
Length $\geq$ 20	102867		82461		109034		82481	
Label types	QC pairs	Labeled 1	QC pairs	Labeled 1	QC pairs	Labeled 1	QC pairs	Labeled 1
Training	246879	33.33%	197904	33.33%	261681	33.33%	197952	33.33%
Validation	30858	33.33%	24735	33.33%	32706	33.33%	24741	33.33%
Testing	30858	33.33%	24738	33.33%	32709	33.33%	24744	33.33%

Table 5
Statistics of Multiple Candidate (MucQC) pairs.

Data types	C#		Java		SQL		Python	
Top-K size	240886		214894		259900		285328	
Length $\geq$ 10	237078		210904		256602		279695	
Label types	QC pairs	Labeled 1	QC pairs	Labeled 1	QC pairs	Labeled 1	QC pairs	Labeled 1
Training	187500	34.71%	167741	34.64%	203880	35.26%	222524	30.95%
Validation	24708	32.92%	21575	33.66%	26268	34.21%	28717	29.98%
Testing	24864	32.72%	21578	33.65%	26446	33.98%	28449	30.26%

Table 6
Statistics of StackOverflow Question-Code (StaQC) pairs.

Data types	SQL		Python	
Size	101098		185906	
Label types	QC pairs	Labeled 1	QC pairs	Labeled 1
Training	80471	61.32%	147851	63.05%
Validation	10312	62.28%	18862	63.81%
Testing	10315	63.29%	19193	64.42%

5.3.2. Benchmark search codebase

To better evaluate NJACS, our experiments are performed over a search codebase, which is different from the training corpus. In practice, the search codebase could be an organization's local codebase or any codebase created from open source projects. But for open source projects, there will be a gap between the evaluation results of human judgment and machine search. We finally construct a local search codebase for machine evaluation. The reason is that we want to reduce the impact of the subjectivity of manual evaluation. The benchmark search codebase should contain more code candidates that are most semantically closed to the desired code fragment. Thus, we extract the Top 200 queries with the most candidate codes of which total 3239 C#, 3368 Java, 3295 SQL and 3693 Python QC mappings as benchmark search codebase. Compared to Yao et al. (2019) and Yao et al. (2018) using the StaQC dataset with average of 6 candidate codes, our collected benchmark search codebase is more effective in evaluating code search performance, which has the following characteristics: (1) The source dataset comes from the real code-search software Q&A site, StackOverflow, not the cleaned GitHub data. (2) Most code types are entire code fragments with more complex code structure rather than other partial code snippets such as Java method. (3) These candidate codes of all programming languages contain almost average of 7 to 8 semantically similar code fragments. In summary, the complexity of code structure and the number of code candidates in this benchmark search codebase make our code search task challenging. Fig. 5 shows the length frequency distribution of the candidate codes in the benchmark search codebase.

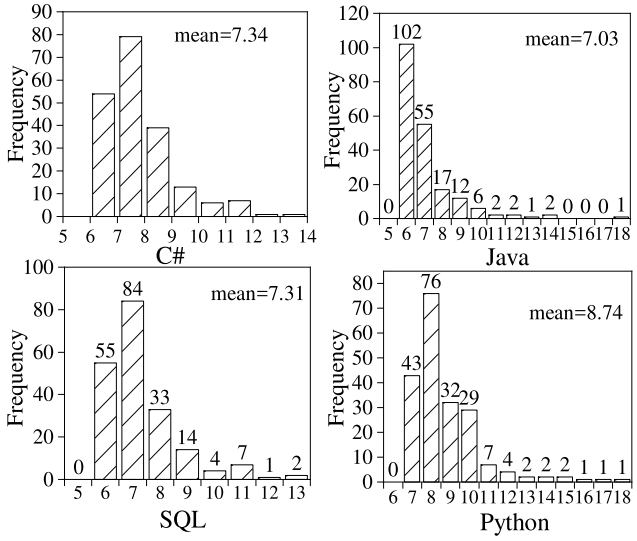

Fig. 5. Length frequency distribution of the candidate codes in Benchmark Search Codebase for C#, Java, SQL and Python.

5.4. Data preprocessing

After data collection, we extract the code structure sequences of the code fragments and the sequential words of the NL queries using different bilingual tokenizers. These bilingual tokenizers consist of two parts: the AST-based tokenizers, the NL tokenizers combined with regular matching and NLP tools. For AST-based tokenizers, we adopted the modified versions of Abstract Syntax Trees (ANTLR parser for C# Iyer et al., 2016 and Java Iyer et al., 2018, python-sqlparse for SQL Yao et al., 2018, python built-in parser for Python Yao et al., 2018) to perform the tokenization for source code. In the tokenizing processing, all literals are replaced with their corresponding types. For example, we replace the table and column names with placeholder tokens (e.g., "tab0", "colo") and number them to preserve their dependencies in SQL. In Python programs, we detect variables, numbers and strings in the code fragments and replace them with special tokens "var", "number" and "string" respectively . Moreover, we convert the camel-case names into underscores in all programming languages by Inflection (StackOverflow, 2009b) library. For NL tokenizers, we use multiple regular-expressions and NLP tools (Nie et al., 2016) (WordNet Miller, 1995, Lemmatizer Loper and Bird, 2002) to uniformly normalize other special words, such as we replace the numbers and strings with "tagint" and "tagstr". In addition, we replace some non-standard characters "c#", "C#" and "C#.Net" with "csharp" to unify the description of program types in C# domain.

5.5. Performance measurement

Code search for Q&A software sites can be formalized as a ranking problem. The goal is to rank the desired code fragment (a positive sample) higher than other candidate codes (the negative samples). Thus, we adopt two common metrics for evaluating code search performance named Recall@k (Gu et al., 2018; Sachdev et al., 2018; Cambronero et al., 2019; Hu et al., 2020) (also known as SuccessRate@k Gu et al., 2018 and Answered@k Cambronero et al., 2019) and Mean Reciprocal Rank (MRR) (Gu et al., 2018; Yao et al., 2019) to measure the effectiveness of all methods. Recall@k measures the percentage of queries for which the desired code fragment could exist in the top k ranked results. MRR is average of the reciprocal ranks of querying results. Recall@k and MRR are calculated as follows:

$$\text{Recall@}k = \frac{1}{N}\sum_{n=1}^{N}\delta(\text{Rank}_{C^+(n)} \le k) \tag{17}$$

$$\text{MRR} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{Rank_{C^+(n)}} \tag{18}$$

where N is a set of queries, $\text{Rank}_{C^+(n)}$ is the ranking position of the desired codes, and δ is an indicator function which returns 1 if the input is true and 0 otherwise. The higher the Recall@k and MRR value, the better the performance.

5.6. Baselines and variants of NJACS

To demonstrate the effectiveness of NJACS, we compare it with both the follow baselines and our proposed variants.

5.6.1. Existing baselines

•**Q**ECK (Nie et al., 2016): A query expansion method for code search based on crowd knowledge. It expands the query for search utilizing the software-specific words extracted from the high-quality pseudo relevance feedback (PRF) QC pairs. We followed the original paper to set the number of related documents and expansion words to 5, 10 respectively.

•**Q**ECK$_{\text{CodeMF}}$ (Hu et al., 2020): Compared with QECK$_{\text{Rocchio}}$, the difference is that QECK$_{\text{CodeMF}}$ introduces CodeMF framework to obtain the high-quality pseudo relevance feedback QC pairs which combined with the content score and quality score.

•**NCS** (Sachdev et al., 2018): An unsupervised model developed at Facebook, which use the word2vec embeddings to jointly embed the codes and queries for measuring their similarity. NCS sums the word embeddings for query and code representation with the average and TF–IDF weights respectively.

•**UNIF** (Cambronero et al., 2019): A supervised extension of the base NCS technique. UNIF uses a bag-of-words-based neural network and computes the attention-based weights for code representation instead of using TF–IDF weights.

•**DeepCS** (Gu et al., 2018): A developed tool applies CODEnn neural network to retrieve codes for Java method. To learn the code representations, in addition to code tokens, it also considers features like function names and API sequences, all of which are combined into a fully connected layer. In our datasets, there is no such features in SQL and Python, thus we slightly modify DeepCS to be the same as the code retrieval model in CoaCor (Yao et al., 2019). ε is set to 0.05.

•**CoaCor** (Yao et al., 2019): A code annotation model trains a reinforcement learning-based network to generate an description for source code that can be used for code retrieval. Since CoaCor can be used for code search as well, we conduct additional experiment using our evaluation datasets. The hyper-parameters for C#, Java, and Python programs are selected the same as SQL in original paper.

•**SelfATT** (Husain et al., 2019): A state-of-the-art neural technique of baseline methods for CodeSearchNet challenge, where multi-head attention (Hu et al., 2018) is used to compute the representation of each token for summarizing the sequential embedded representations of code fragments and NL queries.

Approaches like RACS (Li et al., 2016) and CodeHow (Lv et al., 2015) are excluded from the baseline, as they are limited to special code structure rather than our concerned code fragments.

5.6.2. Variants of NJACS

As described in Section 4.2, NJACS benefits from its four-layer architecture. Abandoning the Joint Attention Layer, we have the following variants based on different design of the Feature Embedding Layer:

•**CodeLSTM**: Similar to QA-LSTM (Tan et al., 2015) for question answering (QA) task, CodeLSTM utilizes the Feature Embedding Layer based on BiLSTM network. Different from DeepCS, CodeLSTM applies average-pooling to BiLSTM outputs for generating final vector representations of code fragments and NL queries.

•**CodeCNN**: CodeCNN constructs the Feature Embedding Layer similar to CNN-based QA (Feng et al., 2015). CodeCNN introduces k-maxpooling (Qiu and Huang, 2015) to keep maximum k values for each filter of CNN outputs. With n parallel filters, CodeCNN final gives the code and query representations with dimension of kn. We add the filter window sizes {1, 2, 3, 5, 7, 9} to the hyper-parameters setting.

•**CodeRCNN**: Same as QA-LSTM (Tan et al., 2015), CodeRCNN introduces both BiLSTM and CNNs for Feature Embedding Layer. CodeRCNN reconstructs the CNN structure on the outputs of max-pooling based BiLSTM for giving a more composite representation both of code fragments and NL queries.

•**CodeATT**: The architecture of CodeATT is similar to attention-based QA-LSTM (Tan et al., 2015). Unlike NJACS using concat-based global attention mechanism, CodeATT use a general-based global attention mechanism (Luong et al., 2015) to compute the query and code representations with attention-based weights.

In addition, there could be other variants. We only show these models which are most closely related to NJACS.

5.7. Implementation details

We use (Tensorflow, 0000) to implement all models. In all experiments, We compute the best Recall@1 or MRR score on the testing set to choose the best final model for prediction. Through extensive experiments on tuning parameters, we find that the optimal training parameters for one programming language are also applicable to others. For NJACS, the same hyper-parameters for all languages are set as follows. In the Structure Embedding Layer, we pad the NL queries and codes to the maximum length ($|Q| = 20$, $|C| = 200$) with "PAD". Formally, we replace the words occurring with a frequency of less than 2 in the vocabulary with "UNK" token. These "UNK" and "PAD" tokens are initialized randomly with values uniformly sampled from $[-0.25, 0.25]$. For the other NL words or code tokens, we preform embedding lookup using pre-trained structure embeddings. In the Feature Embedding Layer, we use the two-layer stacked BiLSTM with the hidden dimension u of 256. In the Joint Attention Layer, we set the dimensions of attention a to 200. In the Similarity Matching Layer, we set parameter regularization $\lambda = 0.02$ and use margin (ε) of 0.5. Besides, we set dropout probability of 0.5 in embedding lookup and BiLSTM encoding. Finally, we train our models in batch size of 256 on 2 Nvidia Titan V GPUs. Specifically, we use Adam (Zinkevich et al., 2010) with a learning rate of $2 * 10^{-4}$ on all parameters in training.

Table 7
Experimental comparison of SicQC Pairs and MucQC Pairs on Recall@1 and MRR metrics. D.T and P.T indicate data and program types. Predicting set refers to our benchmark search codebase. These footnotes "$_{+Struc2vec}$", "$_{+Word2vec}$" and "$_{+Random}$" represent different models using structure embddings, pre-trained word2vec embddings and randomly initialized embeddings respectively.

D.T		SicQC pairs				MucQC pairs			
P.T	Model	Testing set		Predicting set		Testing set		Predicting set	
		Recall@1	MRR	Recall@1	MRR	Recall@1	MRR	Recall@1	MRR
C#	QECK	–	–	0.1115	0.2383	–	–	0.1140	0.2345
	QECK$_{CodeMF}$-(Hu et al., 2020)	–	–	0.1134	0.3056	–	–	0.1146	0.3132
	NCS$_{+Struc2vec}$	–	–	0.1250	0.3456	–	–	0.1236	0.3287
	UNIF$_{+Struc2vec}$	0.5129	0.7211	0.1300	0.3641	0.3739	0.6338	0.1350	0.3704
	CodeCNN$_{+Struc2vec}$	0.7048	0.8379	0.1800	0.4013	0.4850	0.7036	0.2750	0.4924
	CodeRCNN$_{+Struc2vec}$	0.8119	0.8998	0.1800	0.4013	0.5079	0.7168	0.2900	0.5097
	CodeLSTM$_{+Struc2vec}$	0.8238	0.9063	0.1450	0.3731	0.4972	0.7104	0.3100	0.5221
	DeepCS$_{+Struc2vec}$	0.8328	0.9114	0.1800	0.3916	0.5097	0.7177	0.3550	0.5440
	SeltATT$_{+Struc2vec}$	0.8579	0.9259	0.1650	0.3842	0.5130	0.7197	0.2900	0.5137
	CodeATT$_{+Struc2vec}$	0.8666	0.9291	0.1850	0.4096	0.5119	0.7183	0.3630	0.5406
	NJACS	**0.8758**	**0.9346**	**0.1950**	**0.4107**	**0.5173**	**0.7230**	**0.3700**	**0.5636**
SQL	QECK	–	–	0.1340	0.3045	–	–	0.1580	0.3245
	QECK$_{CodeMF}$-(Hu et al., 2020)	–	–	0.1450	0.3387	–	–	0.1698	0.3347
	NCS$_{+Struc2vec}$	–	–	0.1650	0.3564	–	–	0.1620	0.3621
	UNIF$_{+Struc2vec}$	0.4466	0.6842	0.1900	0.4039	0.3985	0.6554	0.1850	0.4133
	CodeCNN$_{+Struc2vec}$	0.5351	0.7387	0.1600	0.3912	0.5038	0.7161	0.3050	0.5033
	CodeRCNN$_{+Struc2vec}$	0.7042	0.8396	0.2100	0.4175	0.5174	0.7251	0.3000	0.5251
	CodeLSTM$_{+Struc2vec}$	0.7293	0.8540	0.1950	0.4101	0.5223	0.7266	0.3400	0.5456
	DeepCS$_{+Random}$-(Yao et al., 2019)	0.7092	0.8430	0.1600	0.2852	0.5045	0.7037	0.29500	0.5010
	DeepCS$_{+Struc2vec}$	0.7464	0.8627	0.1800	0.4090	0.5164	0.7231	0.3250	0.5326
	SeltATT$_{+Struc2vec}$	0.7506	0.8654	0.1750	0.4109	0.5235	0.7277	0.3750	0.5625
	CodeATT$_{+Struc2vec}$	0.7625	0.8714	**0.2100**	0.4263	0.5143	0.7235	0.3550	0.5484
	NJACS	**0.7944**	**0.8897**	0.1900	**0.4348**	**0.5451**	**0.7291**	**0.3550**	**0.5620**
Java	QECK-(Nie et al., 2016)	–	–	0.1530	0.3577	–	–	0.1550	0.3023
	QECK$_{CodeMF}$	–	–	0.1633	0.3687	–	–	0.1632	0.3412
	NCS$_{+Word2vec}$-(Sachdev et al., 2018)	–	–	0.1735	0.3710	–	–	0.1728	0.3551
	NCS$_{+Struc2vec}$	–	–	0.1850	0.3830	–	–	0.1800	0.3680
	UNIF$_{+Word2vec}$-(Cambronero et al., 2019)	0.4466	0.6842	0.1900	0.4039	0.3985	0.6554	0.1850	0.4133
	UNIF$_{+Struc2vec}$	0.4872	0.7064	0.1950	0.4098	0.3878	0.6478	0.2050	0.4174
	CodeCNN$_{+Struc2vec}$	0.6979	0.8355	0.2250	0.4339	0.4770	0.7007	0.2400	0.4637
	CodeRCNN$_{+Struc2vec}$	0.7737	0.8786	0.2150	0.4332	0.4792	0.7013	0.2400	0.4685
	CodeLSTM$_{+Struc2vec}$	0.7867	0.8853	0.1650	0.3869	0.4759	0.7000	0.2700	0.4756
	DeepCS$_{+Random}$-(Gu et al., 2018)	0.7198	0.8477	0.1400	0.3752	0.4729	0.6978	0.2250	0.4560
	DeepCS$_{+Struc2vec}$	0.7929	0.8891	0.1600	0.4023	0.4756	0.7003	0.2250	0.4591
	SeltATT$_{+Random}$-(Husain et al., 2019)	0.4947	0.7174	0.1800	0.4053	0.4682	0.6899	0.2550	0.4547
	SeltATT$_{+Struc2vec}$	0.8153	0.9010	0.1850	0.4207	0.4810	0.7036	0.2750	0.4884
	CodeATT$_{+Struc2vec}$	0.7983	0.8914	0.2250	0.4438	0.4833	0.7029	**0.2800**	**0.4923**
	NJACS	**0.8354**	**0.9121**	**0.2350**	**0.4655**	**0.4984**	**0.7275**	0.2750	0.4804
Python	QECK	–	–	0.1200	0.2591	–	–	0.1250	0.2719
	QECK$_{CodeMF}$	–	–	0.1465	0.2678	–	–	0.1289	0.3087
	NCS$_{+Struc2vec}$	–	–	0.1850	0.2873	–	–	0.1350	0.3367
	UNIF$_{+Struc2vec}$	0.4601	0.6901	0.1500	0.3221	0.3696	0.6245	0.1300	0.3390
	CodeCNN$_{+Struc2vec}$	0.7539	0.8672	0.2000	0.4008	0.5175	0.7184	0.2350	0.4333
	CodeRCNN$_{+Struc2vec}$	0.8224	0.9056	0.1500	0.3620	0.5241	0.7233	0.2100	0.4494
	CodeLSTM$_{+Struc2vec}$	0.8354	0.9125	0.1950	0.3903	0.5269	0.7247	0.2550	0.4724
	DeepCS$_{+Struc2vec}$	0.8415	0.9160	0.1850	0.3988	0.5306	0.7268	0.2450	0.4557
	SeltATT$_{+Random}$-(Husain et al., 2019)	0.3856	0.6445	0.1550	0.3211	0.5250	0.7236	0.1950	0.4273
	SeltATT$_{+Struc2vec}$	0.8534	0.9222	0.1650	0.3733	0.5307	0.7273	0.2150	0.4494
	CodeATT	0.8591	0.9253	0.2050	0.3477	0.5353	0.7286	0.2700	0.4751
	NJACS	**0.8738**	**0.9335**	**0.2150**	**0.4128**	**0.5442**	**0.7294**	**0.2900**	**0.4812**

6. Evaluation and results

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

- RQ1: Whether our proposed two-way attention-based neural code search approach for software Q&A sites, NJACS, achieves state-of-the-art performance in all the benchmarks ?
- RQ2: Whether NJACS can effectively improve the performance of code search for various programming languages using structure embeddings instead of other embeddings ?
- RQ3: Whether joint attention-based NJACS introduced with two-way attention information can improve the performance than other representation-based neural code search models ?

6.1. RQ1: Retrieving the desired codes in use-case scenarios like StackOverflow

The first experiment is conducted to identify the effectiveness of NJACS for code search in software Q&A sites.

Unlike the existing baselines (Gu et al., 2018; Sachdev et al., 2018; Yao et al., 2019; Cambronero et al., 2019; Husain et al., 2019), which generally use randomly initialized embeddings (word2vec embeddings) or pre-trained word2vec embeddings (Learned from external software corpus, similar to pre-trained Chinese Word Vectors (Pre-trained Chinese Word Vectors, 0000)

Table 8
Three types of code embeddings with different dimensions used for pre-training neural code search models. D.T and P.T stand for different data and program types respectively. Random, Struc2vec and Word2vec represent randomly initialized embeddings (word2vec embeddings), pre-trained structure embddings and pre-trained word2vec embddings respectively.

D.T	P.T	Random	Word2vec	Struc2vec
SicQC pairs	C#	(192089,300)	(190923,300)	(189583,300)
	SQL	(33127,300)	(32997,300)	(31087,300)
	Java	(133165,300)	(131209,300)	(130469,300)
	Python	(87958,300)	(76407,300)	(85751,300)
MucQC pairs	C#	(189573,300)	(186442,300)	(184862,300)
	SQL	(39064,300)	(39058,300)	(36695,300)
	Java	(135395,300)	(133820,300)	(131067,300)
	Python	(121409,300)	(110123,300)	(116279,300)
StaQC pairs	SQL	(22849,300)	(22725,300)	(18785,300)
	Python	(142511,200)	(103355,300)	(121303,300)

used in NLP), NJACS combines pre-trained structure embeddings. To match the original baseline settings, we used the three different code embeddings as shown in Table 8. It is clear that the number of words in structure embeddings obtained by unified structural normalization is less than the number of words in normal tokenize based word2vec embeddings. Thus, structure embeddings can avoid suffering from data sparseness, as variables and rare symbols may only appear a few times.

Tables 7 and 9 show the evaluation results of NJACS and related approaches. Except for some individual cases, NJACS substantially outperforms all baselines ($+_{Struc2vec}$) and significantly perform better than all benchmarks using other embeddings ($+_{Word2vec}$ or $+_{Random}$) in original paper. This demonstrates that NJACS produces generally more relevant codes than other methods. Compared to the state-of-the-art baselines (e.g., NCS, UNIF and DeepCS), NJACS achieves more 1%∼7.5% higher Recall@1 and 0.53%∼6% better MRR on testing set and predicting set for SicQC Pairs, MucQC Pairs and StaQC Pairs. The gain of Recall@1 and MRR metrics on predicting set for SQL and Python programs in StaQC Pairs is relatively higher, probably because the systematically mined by supervised BiV-HNN model with higher quality than SicQC Pairs and MucQC Pairs extracted using the unsupervised CodeMF framework. We also find that NJACS has better predicting results on StaQC Pairs than SicQC Pairs and MucQC Pairs for SQL and Python. This also shows that increasing the size of training data gives no obvious benefit, possibly because the HQ StaQC Pairs for Python and SQL already carries critical information. NJACS substantially outperforms its CodeATT-based variant model both on Recall@1 and MRR by around 2% in C#, SQL, and Python programs, showing the two-way attention architecture preferable over using only global attention mechanism. Meanwhile, we observe that NJACS performs the worst results on MucQC Pairs for Java compared to other programming languages. The reason is that the benchmark search codebase for Java contains up to 102 similar candidate codes, with the code structure and code type of Java programs that are more complex and diverse than other programs (see Section 5.3.2). More candidates are better for testing model performance, but the more candidates, the greater risk of noisy information involved in the data. Thus, the noise information in search codebase have a negative impact on evaluation results for Java program. Moreover, we observe that NJACS achieves worse precision values on SQL than other programs, probably because the code content in each SQL program carries less critical information for making a prediction. As C#, Java, Python codes contain multiple code types, in which the informative intermediate variable names that are directly related to the objective of the code. In contrast, SQL contains only a few keywords and functions, the unified normalized tokens (e.g. "col0" and "tab0") that make up the majority of the total words. This statistical characteristic for C# and SQL is also confirmed in Iyer et al. (2016).

In addition, we collect these three different QC Pairs in different ways, with different sampling methods and data sizes, which can help us analyze the impact of data on the results. Combining the analysis of Tables 7 and 9, we draw the following conclusions: (1) Instead of increasing the data size of training corpus, using the supervised extracted high-quality QC pairs can better improve code search performance. (2) Randomly sampled data performs worse than original paired code candidates and is more sensitive

Table 9
Experimental comparison of StaQC Pairs on Recall@1 and MRR metrics. P.T stands for different program types. Predicting set refers to our benchmark search codebase. These footnotes "$_{+Random}$" and "$_{+Struc2vec}$" represent different models using randomly initialized embeddings and structure embeddings respectively.

StaQC pairs					
P.T	Model	Testing set		Predicting set	
		Recall@1	MRR	Recall@1	MRR
SQL	QECK	–	–	0.1650	0.3358
	$QECK_{CodeMF}$	–	–	0.1250	0.3456
	$NCS_{+Struc2vec}$	–	–	0.1800	0.3627
	$UNIF_{+Struc2vec}$	0.4524	0.6974	0.2000	0.4134
	$CodeCNN_{+Struc2vec}$	0.5520	0.7552	0.2750	0.4917
	$CodeRCNN_{+Struc2vec}$	**0.5849**	**0.7750**	0.3500	0.5513
	$CodeLSTM_{+Struc2vec}$	0.5823	0.7734	0.3450	0.5493
	$DeepCS_{+Struc2vec}$	0.5782	0.7713	0.3550	0.5530
	$SeltATT_{+Struc2vec}$	0.5847	0.7748	0.3600	0.5466
	$CodeATT_{+Struc2vec}$	0.5759	0.7699	0.3700	0.5690
	NJACS	0.5839	0.7738	**0.3950**	**0.5907**
Python	QECK	–	–	0.1250	0.2936
	$QECK_{CodeMF}$	–	–	0.1250	0.3456
	$NCS_{+Struc2vec}$	–	–	0.1550	0.3248
	$UNIF_{+Struc2vec}$	0.4401	0.6825	0.1500	0.3538
	$CodeCNN_{+Struc2vec}$	0.5687	0.7603	0.2600	0.4535
	$CodeRCNN_{+Struc2vec}$	0.5790	0.7662	0.3250	0.5051
	$CodeLSTM_{+Struc2vec}$	0.5787	0.7669	0.2850	0.4770
	$DeepCS_{+Struc2vec}$	0.5782	0.7713	0.3050	0.5230
	$SeltATT_{+Random}$-(Husain et al., 2019)	0.5663	0.7411	0.2750	0.4876
	$SeltATT_{+Struc2vec}$	0.5777	0.7667	0.3150	0.5036
	$CodeATT_{+Struc2vec}$	0.5797	0.7676	0.3550	0.5467
	NJACS	**0.5890**	**0.7729**	**0.3600**	**0.5677**

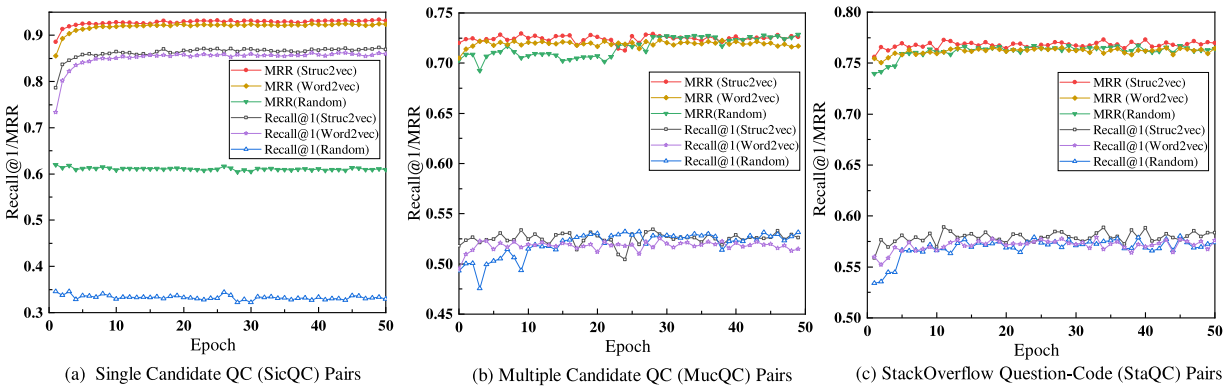

Fig. 6. Recall@1 and MRR curves on testing set for NJACS using different code embeddings in Python. Struc2vec, Word2vec and Random indicates structure embddings, pre-trained word2vec embddings and randomly initialized embeddings respectively.

to external knowledge dependencies such as structure embeddings. (3) Introducing different sampling strategies to select most unrelated candidate codes paired with NL queries in training process can also improve the performance of code search.

> NJACS can significantly retrieve more code solutions than baselines and its variants, and it is also applicable to both general-purpose imperative and declarative programming languages such as C# and SQL.

6.2. RQ2: Comparison of performance metrics using different code embeddings

We conduct the second experiment to compare NJACS against NJACS using other code embeddings and further explore the influence of our proposed pre-trained structure embeddings on the performance of code search. Take Python for example, the performance metrics of NJACS on various QC pairs in Python domain are presented in Fig. 6.

Results in Fig. 6 show the effect of structure embeddings used in NJACS in comparison with alternative of other randomly initialized embeddings (word2vec embeddings) or pre-trained word2vec embddings. Specifically, we use the randomly initialized embeddings from a uniform distribution that follows $[-0.25, 025]$. Due to space constraints, we do not show the result of other randomly initialized embeddings that obey other random distribution such as Gaussian distribution and Exponential distribution. These pre-trained word2vec embddings used for comparison are trained from the three above QC pairs in which the code fragments are processed as unstructured code fragment sequences. Unsurprisingly, we find that over structure embeddings, the trend of curves of Recall@1 and MRR metrics on testing set is relatively high, and converges fast. This means that NJACS tuned with structure embeddings basically achieves the best performance. And we discover that word2vec embeddings built with unstructured software repositories can also improve baseline performance. Although the NJACS model (including structure embeddings) achieves only minor improvements on the MucQC pair and StaQC pair compared to using pre-trained word2vec embedding. However, almost none of the existing neural code search methods use any pre-trained techniques like word embeddings. Compared to currently used randomly initialized embeddings in NJACS, the gain on testing set is relatively higher especially in SicQC Pairs. More concretely, the integration of structure embeddings in NJACS brings most 50% and 30% improvements on Recall@1 and MRR On SicQC Pairs, which shows the effectiveness of using structure embeddings for pre-training neural network. In addition, it is worth noting that DeepCS creates training corpus based on random sampling strategy similar to SicQC Pairs but use randomly initialized word embeddings. In reality, we incorporate the structure embeddings in all baseline experiments, which means that our experimental results have improved significantly. Thus, the introduction of structure embeddings as the external structure information is very useful in improving code search performance of all neural models.

Table 10
Comparison of experimental results on Recall@1 and MRR metrics in attention-based (INE) and representation-based (REN) neural models using structure embeddings. D.T, P.T and M.T stand for different data, program and model types respectively. Predicting set refers to our benchmark search codebase.

D.T	P.T	M.T	Testing set		Predicting set	
			Recall@1	MRR	Recall@1	MRR
SicQC pairs	C#	REN	0.8579	0.9259	0.1800	0.3916
		INE	**0.8666**	**0.9291**	**0.1850**	**0.4096**
	SQL	REN	0.7506	0.8654	**0.2100**	0.4175
		INE	**0.7625**	**0.8714**	0.1900	**0.4263**
	Java	REN	0.8153	0.9019	0.2250	0.4339
		INE	**0.7983**	**0.8914**	**0.2250**	**0.4438**
	Python	REN	0.8534	0.9222	0.1950	**0.3988**
		INE	**0.8591**	**0.9253**	**0.2050**	0.3477
MucQC pairs	C#	REN	0.5130	0.7197	0.35500	**0.5440**
		INE	**0.5119**	**0.7183**	**0.3630**	0.5406
	SQL	REN	**0.5164**	**0.7266**	0.3750	**0.5625**
		INE	0.5143	0.7235	**0.3550**	0.5484
	Java	REN	0.4810	0.7036	0.2750	0.4884
		INE	**0.4833**	**0.7029**	**0.2750**	**0.4804**
	Python	REN	0.5307	0.7273	0.2550	0.4724
		INE	**0.5353**	**0.7286**	**0.2700**	**0.4751**
StacQC pairs	SQL	REN	**0.5849**	**0.7750**	0.3600	0.5530
		INE	0.5759	0.7699	**0.3700**	**0.5690**
	Python	REN	0.5790	0.7713	0.3150	0.5230
		INE	**0.5797**	**0.7729**	**0.3550**	**0.5467**

> NJACS can improve the performance of training neural models on various large-scale codebases using code structure embeddings that can be leveraged for the software task targeting deep program understanding.

6.3. RQ3: Comparison of attention-based and representation-based models

As the existing neural code search models are mostly using representation-based models (e.g., NCS, UNIF, DeepCS), we

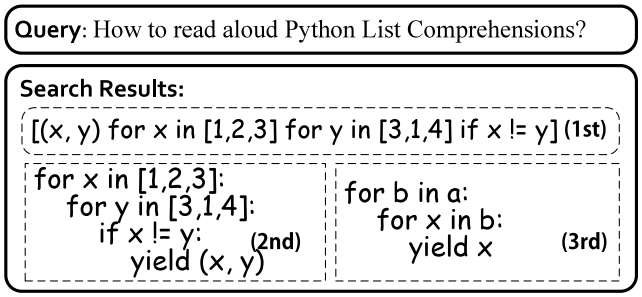

Fig. 7. Examples of code fragments searched by NJACS for a natural query in Python domain.

conduct the third experiment to explore the influence of joint attention features on code search performance.

Firstly, we divide the experimental neural models into two groups: attention-based models and representation-based models. The representation-based models are composed of NCS, UNIF, CodeCNN, CodeRCNN, CodeLSTM, DeepCS and SelfATT. The attention-based models refer to CodeATT and NJACS. Next, we compare the best performance of the representation-based models with the worst performance of the attention-based models. The evaluation results are shown in Table 10.

As shown in Table 10, besides SQL programs, the attention-based models completely outperform the representation-based models on testing set for C#, Python and Java in all experiments. We also find that the attention-based models are mostly better than the representation-based models on predicting set in six groups of experiments. The results indicate that the joint attention information both from global attention mechanism and attentive pooling operation are beneficial to improve the performance of code search. Moreover, combined with the first experiment, we observe that the two-way attention-based architecture contained in NJACS can perform better results.

> Compared to the representation-based models, the attention-based models such as CodeATT and NJACS adopting the attention features of heterogeneous languages can improve the performance of code search.

6.4. Case study

We now provide concrete examples of code search results for benchmark search codebase that demonstrate the advantages of NJACS. Take Python for example, we consider the query "how to read aloud python list comprehensions?" in Post# 9061760 (StackOverflow, 2012) whose responses contains 8 code fragments. As shown in Fig. 7, NJACS is able to retrieve the desired code fragment (code solution). And we find that the top 3 code fragments of the search results can be used as alternative code solution with similar functions. As NL query often contain noisy information (e.g., Post#17777182 (StackOverflow, 2013) with irrelevant or noisy keywords "dictonary"), but this does not affect the performance of NJACS. The ability of query understanding enables NJACS to perform a more robust code search. Moreover, Fig. 7 also shows the results of two similar candidate codes (2nd and 3rd). The two candidate codes have most common keywords and may be difficult to distinguish via IR-based methods. NJACS can understand the meaning of the two queries and return relevant fragments. Due to space constraints, we do not show other more examples. Apparently, NJACS has the ability to recognize NL query and code semantics.

We can attribute the above to the two-way attention mechanism used by NJACS. Take Python as an example, when searching "xor matrix multiplication for AES mix column stage", NJACS can also successfully identify the importance of different keywords and use them to focus on key parts of code fragments. Fig. 8 shows the relative magnitudes of the joint attention based weights between NL query and its corresponding code fragment. Darker regions represent stronger weights.

These three experiments RQ1, RQ2 and RQ3 show that NJACS can effectively search the code solution from multiple semantically similar candidate codes. It means that NJACS is suitable for various application scenarios (Multi-type NL queries and code fragments) in software Q&A sites. While there could be other examples, but we do not show them due to space constraints.

7. Discussions

7.1. Why does NJACS work?

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

Constructing bi-modal embedding model using joint attention information. Unlike traditional representation-based models such as NCS and DeepCS, NJACS adopts a joint attention based neural network instead of RNN-based bi-modal embedding techniques for combining sufficiently detailed matching signals and incorporates code and query terms importance learning using a two-way attention mechanism. This novel value-weighted model is able to give different attention of both the query and code terms in their respective directions, thus achieving a focus on the query description.

Parsing code structural sequence for embedding entire code fragments. Compared to text documents, source codes have their own unique properties, such as code structures and code identifiers. Most of C# fragments contain loops and conditionals, and SQL queries often comprise one or more subqueries. Most tokens such as variable names denoting their types are often meaningless to semantic expression of code. Therefore, we normalize the code fragment into a structural sequence that reduce the sparsity of the data and maintains the homology of the code fragments.

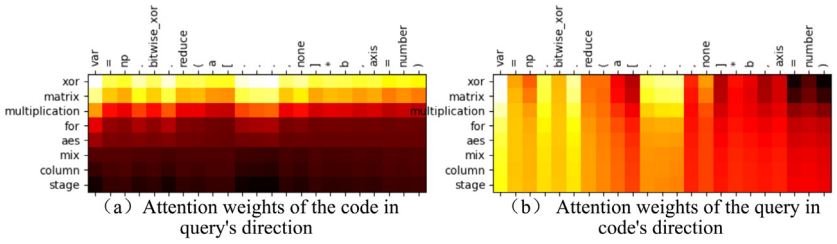

Fig. 8. Heatmap of attention weights $m(i,j)$ for example Python code fragments in StaQC Pairs. NJACS learns to align the key words in natural query with the corresponding tokens in code fragment.

Incorporating structure embeddings for pre-training code search models: Word2vec embeddings and BERT have been used to great effect in NLP to boost the accuracy of downstream learning tasks, such as neural translation (Devlin et al., 2018) and answer selection (Tan et al., 2015). Compared to learning the representation of code tokens based on standard sparse "one-hot" encoding or normal word embeddings, the superiority of our work is that the code structure embeddings is likely to provide a higher boost in the performance of code search, which involves the reasoning of structure information of source code.

7.2. Limitation of NJACS

Despite the advantages such as associative search, NJACS could still return inaccurate results. It sometimes returns the first code that is not a standard code solution. Affected by the interference candidate codes (code#1 differs from code#2) by only one word which appears less frequently in the dictionary, NJACS may treat them as the same code fragments for query semantic matching. In future work, manually annotated benchmark search datasets could be considered in our model to further adjust the results.

7.3. Threats to validity

Threats to the effectiveness of NJACS include:

Quality of collected query-code mappings: We currently only consider the first code fragment to be a standalone code solution or not. In many cases, most code fragments in responses of QC pairs serve as multiple steps and should be merged to form a complete solution (Yao et al., 2018). What is more, it is confirmed in Yao et al. (2018), Wong et al. (2013) that the first code fragment may not be completely semantically paired with query. This is another challenging task of mining software repositories in related work (Hu et al., 2020; Yao et al., 2018; Yin et al., 2018). Nevertheless, the first code fragment usually contains critical information to answer a question.

Quantity of candidate codes in codebase: Compared to the large-scale benchmark codebase used in code search (Gu et al., 2018; Cambronero et al., 2019), the quantity of code candidates in our search codebase are still not considered enough similar to Yao et al. (2019) and Yao et al. (2018). Although Q&A software sites like StackOverflow have contributed a large number of software repositories, due to their open and unrestricted nature, many of their code candidates may not match user queries. It is difficult for us to balance the pairing quality of QC pairs in the search codebase with providing more relevant candidate codes. Thus, there are very limited code candidates from actual code search scenarios.

Setting of evaluation methods and metrics: NJACS aims to focus on the code search of software Q&A sites, but the existing public datasets are flawed in the application of software Q&A sites (Yan et al., 2020). In the software Q&A sites, collecting more code candidates may not match the query, but will introduce more noise information. Similar to many software tasks like code annotation (Iyer et al., 2016; Chen and Zhou, 2018) and code mining (Yao et al., 2018), we can only use QC pairs with one relevant code to confirm the validity of NJACS. Thus, we do not adopt other evaluation indicators such as Precision@k (Gu et al., 2018) and NDCG (Husain et al., 2019), which are used to evaluate multiple candidate relevant codes.

8. 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.

Feature extraction-based code search: There are many work studies on code search using IR techniques. For instance, McMillan et al. (2011) proposed Portfolio, which extracts relevant functions for motivation query via utilizing keyword matching and PageRank. Rahman and Roy (2018) proposed RACK to convert queries into list of API classes for collecting relevant codes. As NL query is often too short, Haiduc et al. (2013), Paik et al. (2014), Lu et al. (2018), Sirres et al. (2018) and Iyer et al. (2018) turn to use query expansion and reformulation. For instance, Haiduc et al. (2013) proposed to reformulate queries based on machine learning. Lv et al. (2015) proposed CodeHow, which combines text similarity and API matching using an extended Boolean model. Hill et al. (2011) reformulated the queries with NL phrasal representations of method signatures. Moreover, Lu et al. (2015), Lemos et al. (2014) and Zhang et al. (2017b) expanded queries using external knowledge similar to Iyer et al. (2018). For example, Lu et al. (2015) proposed a Pwordnet technique by leveraging synonyms generated from WordNet. Similarly, Lemos et al. (2014) automatically extended the query based on WordNet and the code-related thesaurus. Zhang et al. (2017b) proposed an automated approach that recommends semantically related API classnames using Word2vec. Similarly, Balaneshin-kordan and Kotov (2017) proposed SWDM, a PRF-based query expansion technique using word embeddings. In addition, code-to-code search tools such as FaCoY (Kim et al., 2018) and Aroma (Luan et al., 2019) were also proposed to use a code snippet as a query and retrieve the relevant code snippets. These existing techniques often face the term mismatch problem. NJACS differs from them in that it can learn semantic associations between NL query and code fragment without sharing enough common terms.

Semantic matching-based code search: Due to the development of NLP tasks, Gu et al. (2018), Sachdev et al. (2018), Yao et al. (2019), Cambronero et al. (2019), Hu et al. (2020), Siddaramappa et al. (2013) and Siddaramappa et al. (2013) have increasing attention on semantic-based code search. For example, Luan et al. (2019) proposed a semantic-based query technique for searching source code. Ke et al. (2015) proposed SearchRepair, an approach that used for automatic defect repair. Sachdev et al. (2018) proposed NCS, which combines bi-modal embedding, TF–IDF and cosine similarity search techniques. Tekchandani et al. (2018) computed semantic code clones on AST. However, they only considered the respective semantic transformations. Thus, more researchers recently have investigated the application of deep learning techniques to code search. Gu et al. (2018) proposed CODEnn for combining embedding and deep learning to measure the similarity between code snippets and user queries. Yao et al. (2019) proposed CoaCor, a novel perspective of code annotation for code retrieval with reinforcement learning. Compared to NCS (Sachdev et al., 2018), Cambronero et al. (2019) proposed the improved UNIF model to compute code representation with attention mechanism. In addition, DeepSim (Zhao and Huang, 2018) was proposed to perform deep learning code functional similarity. However, these methods are often designed for special code granularity at method or class level. Moreover, some of them build the code representation using splitting-fusion or overall-sequence embedding techniques, which ignore the structural sequential information beyond the textual code representation. For NJACS, it can embed entire continuous code fragment with structure information and retrieve general code fragments of various code structures and code types.

Deep learning of cross-language pairs: Besides code search, most existing work have explored unified learning representation of both natural and programming languages for enhancing the performance of other software task, such as code summarization, code generation, code recommendation etc. For example, Iyer et al. (2016) presented CODE-NN, an attention-based LSTM neural network that produces NL descriptions for C# and SQL. Gu et al. (2016) proposed DeepAPI, a neural approach to generate API usage sequences for a given NL query. And Allamanis et al. (2016) proposed a novel convolutional attentional network that successfully performs extreme summarization of source code. Besides NL-to-program cross-language learning, Bui and Jiang (2018) proposed a new way to learn code element mappings across programming languages that may be used for program translation. Zhong et al. (2017) proposed a TreeLSTM-based network to synthesize SQL queries from NL questions. In addition, White et al. (2016) also used deep learning to find code clones. Huo et al. (2016) applied a CNN-based deep neural network to learn unified features from bug report and code for locating buggy files. Zhang et al. (2017a) proposed a URL framework for top-N recommendation with heterogeneous information sources — including texts, images and many others. In our work, NJACS incorporates more sophisticated components such as pre-trained structure embeddings and two-way attention mechanism, which guide more attention on program structure from query in code search (like eye tracking).

Learning representation of source code: In addition to program analysis, a line of work has focused on source code representation. Carpineto and Romano (2012), Haiduc et al. (2013), Paik et al. (2014), Lu et al. (2018), Sirres et al. (2018) and Sachdev et al. (2018) used traditional IR and machine learning methods for token-based textual code representation. For instance, Deerwester et al. (1990) and Blei et al. (2003) applied the Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA) to analyze source code. However, the common problem of these approaches is that they lack of introducing any structure information in text-based or token-based methods. Recently neural approaches (Peng et al., 2015; Mou et al., 2016; Wei and Li, 2017; Zhang et al., 2019; Allamanis et al., 2017) have attracted much attention to learn distributed representation of source code. Peng et al. (2015) created a novel "coding criterion" to build vector representations of code AST nodes. And the others combined AST and Tree-LSTM (Mou et al., 2016), Tree-based CNN (TBCNN) (Wei and Li, 2017) or AST-based Neural Network (ASTNN) (Zhang et al., 2019) to capture both the lexical and syntactical information of ASTs in source code, thus obtaining a better representation than traditional token-based methods. Among them, Tree-LSTM (Mou et al., 2016) is proposed to represent the functionality semantics of code fragments. TBCNN (Wei and Li, 2017) used the custom CNN on ASTs to learn vector representations of code snippets. ASTNN (Zhang et al., 2019) parsed the ASTs into the statement-tree sequences and further constructed them for code representation, rather than processed on the entire ASTs of source codes. Furthermore, Allamanis et al. (2017) trained the Gated Graph Neural Networks (GGNN) on program graphs to predict variable names and detect variable misuses. However, these above methods of AST-based code representation have not been empirically analyzed in code search. In NJACS, we explore the application of AST-based structure embeddings of source code and joint attention-based neural network for code search.

9. Conclusion & future work

In this paper, we propose NJACS (Neural Joint Attention Code Search) as a novel, scalable and effective code search model for software Q&A sites. In addition, NJACS is also suitable for retrieving more general code fragments, such as code elements (e.g., method body in C#) and code libraries (e.g., function body in Python). NJACS introduces both special code structure embeddings and joint attention-based framework to better model the unified vector representations between code fragments and NL queries. Experiments show that the two-way attention mechanism including global attention operation and attentive pooling mechanism used in NJACS is effective.

Moreover, the framework we designed in NJACS can theoretically improve the performance of other software tasks such as code annotation, code clone detection and program repair, most of which currently use the representation-based models without considering the deep joint attention-based semantic representation of cross-language pairs. In addition, our proposed structure embeddings that capture semantic similarity enable applications such as contextual search and automated categorization of libraries.

In future work, we will investigate more aspects of source code such as intrinsic semantic structure and context description information to better represent high-level semantics of code fragment and design better structural neural networks such as GNN (Graph Neural Network) and ON-LSTM (Ordered Neurons LSTM) for constructing code search models.

Our source code and data of the NJACS model is published at: https://github.com/hoogang/NJACS.

CRediT authorship contribution statement

Gang Hu: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Min Peng:** Validation, Formal analysis, Visualization, Software, Supervision, Funding acquisition. **Yihan Zhang:** Validation, Formal analysis, Visualization. **Qianqian Xie:** Resources, Writing - review & editing, Supervision, Data curation. **Mengting Yuan:** Formal analysis, Writing - review & editing, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Wewould like to thank anonymous reviewers for insightful comments on earlier drafts of this paper. The authors gratefully acknowledge the financial support of the National K&D Program of China Grants 2018YFC1604000 and 2018YFC1604003, and this work is also supported partially by Natural Science Foundation of China Grants 71950002 (Special Program), 61872272 and 61772382 (General Program).

References

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning (ICML). JMLR, pp. 2091–2100.

Allamanis, M., Tarlow, D., Gordon, A., et al., 2015. Bimodal modelling of source code and natural language. In: International Conference on Machine Learning (ICML). pp. 2123–2132.

Alon, U., Zilberstein, M., Levy, O., et al., 2019. Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. (POPL 3 (40), 135–146.

Balaneshin-kordan, S., Kotov, A., 2017. Embedding-based query expansion for weighted sequential dependence retrieval model. In: International Conference on Research and Development in Information Retrieval (SIGIR). pp. 1213–1216.

Blei, D.M., Ng, A., Jordan, M., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. (JMLR) 993–1022.
Bojanowski, P., Grave, E., Joulin, A., et al., 2017. Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. (TACL) 5, 135–146.
Bui, N.D.Q., Jiang, L., 2018. Hierarchical learning of cross-language mappings through distributed vector representations for code. In: International Conference on Software Engineering (ICSE). pp. 33–36.
Cambronero, J., Li, H., Kim, S., et al., 2019. When deep learning met code search. In: International Symposium on Foundations of Software Engineering (FSE). pp. 964–974.
Carpineto, C., Romano, G., 2012. A survey of automatic query expansion in information retrieval. Acm Computing Surveys (CSUR) 44 (1), 1–50.
Chen, Q., Zhou, M., 2018. A neural framework for retrieval and summarization of source code. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 826–831.
Deerwester, S., Dumais, S.T., Furnas, G.W., et al., 1990. Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. Technol. (JASIST) 41 (6), 391–407.
Dempster, A.P., Laird, N.M., Rubin, D.B., 1977. Maximum likelihood from incomplete data via the EM algorithm. J. R. Stat. Soc. (JSTOR) 39 (1), 1–22.
Devlin, J., Chang, M.W., Lee, K., et al., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
Feng, M., Xiang, B., Glass, M.R., et al., 2015. Applying deep learning to answer selection: A study and an open task. In: International Conference on Automatic Speech Recognition and Understanding (ASRU). IEEE, pp. 813–820.
Frome, A., Corrado, G.S., Shlens, J., et al., 2013. Devise: A deep visual-semantic embedding model. In: Advances in Neural Information Processing Systems (NIPS). pp. 2121–2129.
Gensim. Available: https://radimrehurek.com/gensim/.
Github. Available: https://github.com/.
Grechanik, M., Fu, C., Xie, Q., et al., 2010. A search engine for finding highly relevant applications. In: International Conference on Software Engineering (ICSE). ACM, pp. 475–484.
Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: International Conference on Software Engineering (ICSE). IEEE, pp. 933–944.
Gu, X., Zhang, H., et al., 2016. Deep API learning. In: International Symposium on Foundations of Software Engineering (FSE). pp. 631–642.
Haiduc, S., Bavota, G., Marcus, A., et al., 2013. Automatic query reformulations for text retrieval in software engineering. In: International Conference on Software Engineering (ICSE). IEEE, pp. 842–851.
Hill, E., Pollock, L., Vijay-Shanker, K., 2011. Improving source code search with natural language phrasal representations of method signatures. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 524–527.
Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.
Hoffmann, R., Fogarty, J., Weld, D.S., 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In: International Conference on User Interface Software and Technology (UIST). pp. 13–22.
Hu, X., Li, G., Xia, X., et al., 2018. Deep code comment generation. In: International Conference on Program Comprehension (ICPC). pp. 200–210.
Hu, X., Li, G., Xia, X., et al., 2019. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. (ESE) 25 (3), 2179–2217.
Hu, G., Peng, M., Zhang, Y., et al., 2020. Unsupervised software repositories mining and its application to code search. In: Software: Practice and Experience (SPE), vol. 50, 3. pp. 299–322.
Huo, X., Li, M., Zhou, Z.H., 2016. Learning unified features from natural and programming languages for locating buggy source code. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 1606–1612.
Husain, H., Wu, H.H., Gazit, T., et al., 2019. CodeSearchNet challenge: evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.
Inflection. Available: https://inflection.readthedocs.io/en/latest/.
Iyer, S., Konstas, I., Cheung, A., et al., 2016. Summarizing source code using a neural attention model. In: International Conference on Association for Computational Linguistics (ACL). pp. 2073–2083.
Iyer, S., Konstas, I., Cheung, A., et al., 2018. Mapping language to code in programmatic context. arXiv preprint arXiv:1808.09588.
Ke, Y., Stolee, K.T., Le Goues, C., et al., 2015. Repairing programs with semantic code search. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 295–306.
Kim, K., Kim, D., et al., 2018. FaCoY: a code-to-code search engine. In: International Conference on Software Engineering (ICSE). ACM, pp. 946–957.
Krugle. Available: http://opensearch.krugle.org/.
LeClair, A., Eberhart, Z., McMillan, C., 2018. Adapting neural text classification for improved software categorization. In: International Conference on Software Maintenance and Evolution (ICSME). pp. 461–472.
Lemos, O.A.L., de Paula, A.C., Zanichelli, F.C., et al., 2014. Thesaurus-based automatic query expansion for interface-driven code search. In: International Conference on Mining Software Repositories (MSR). pp. 212–221.
Li, X., Wang, Z., Wang, Q., et al., 2016. Relationship-aware code search for javascript frameworks. In: International Symposium on Foundations of Software Engineering (FSE). pp. 690–701.
Loper, E., Bird, S., 2002. NLTK: the natural language toolkit. arXiv preprint arXiv:0205028.
Lu, M., Sun, X., Wang, S., et al., 2015. Query expansion via wordnet for effective code search. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 545–549.
Lu, J., Wei, Y., Sun, X., et al., 2018. Interactive query reformulation for source-code search with word relations. IEEE Access 75660–75668.
Luan, S., Yang, D., Barnaby, C., et al., 2019. Aroma: code recommendation via structural code search. Proc. ACM Program. Lang. (PACMPL) 3 (152), 1–28.
Luong, M.T., Pham, H., Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025.
Lv, F., Zhang, H., Lou, J., et al., 2015. Codehow: Effective code search based on api understanding and extended boolean model. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 260–270.
Maaten, L., Hinton, G., 2008. Visualizing data using t-SNE. J. Mach. Learn. Res. (JMLR) 9, 2579–2605.
McMillan, C., Grechanik, M., Poshyvanyk, D., et al., 2011. Portfolio: finding relevant functions and their usage. In: International Conference on Software Engineering (ICSE). pp. 111–120.
Mikolov, T., Sutskever, I., Chen, K., et al., 2013a. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems (NIPS). pp. 3111–3119.
Mikolov, T., Yih, W., Zweig, G., 2013b. Linguistic regularities in continuous space word representations. In: International Conference on Association for Computational Linguistics (ACL). pp. 746–751.
Miller, G.A., 1995. Wordnet: a lexical database for english. Commun. ACM (CACM) 38 (11), 39–41.
Mnih, V., Kavukcuoglu, K., Silver, D., et al., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
Mou, L., Li, G., Zhang, L., et al., 2016. Convolutional neural networks over tree structures for programming language processing. In: AAAI Conference on Artificial Intelligence (AAAI). pp. 1287–1293.
Nie, L., Jiang, H., Ren, Z., et al., 2016. Query expansion based on crowd knowledge for code search. IEEE Trans. Serv. Comput. (TSC) 9 (5), 771–783.
Paik, J.H., Pal, D., Parui, S.K., 2014. Incremental blind feedback: An effective approach to automatic query expansion. ACM Trans. Asian Lang. Inf. Process. (TALIP) 13 (3), 1–22.
Pattanayak, S., 2017. Convolutional neural networks. In: Pro Deep Learning with TensorFlow.
Peng, M., Dai, X., et al., 2016a. KPCA-WT: an efficient framework for high quality microblog extraction in time-frequency domain. In: International Conference on Web-Age Information Management (WAIM). Springer, pp. 304–315.
Peng, M., Gao, B., et al., 2016b. High quality information extraction and query-oriented summarization for automatic query-reply in social network. Expert Syst. Appl. (ESWA) 44, 92–101.
Peng, H., Mou, L., Li, G., et al., 2015. Building program vector representations for deep learning. In: International Conference on Knowledge Science, Engineering and Management (KSEM). Springer, pp. 547–553.
Pennington, J., Socher, R., Manning, C., 2014. Glove: global vectors for word representation. In: International Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1532–1543.
Pre-trained Chinese Word Vectors. Available: https://https://github.com/Embedding/Chinese-Word-Vectors/.
Python2vec: Word embeddings for source code. Available: https://gab41.lab41.org/python2vec-word-embeddings-for-source-code-3d14d030fe8f.
Qiu, X., Huang, X., 2015. Convolutional neural tensor network architecture for community-based question answering. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 1305–1311.
Rahman, M.M., Roy, C., 2018. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In: International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 473–484.
Ramos, J., 2003. Using tf-idf to determine word relevance in document queries. In: International Conference on Machine Learning (ICML). pp. 133–142.
Sachdev, S., Li, H., Luan, S., et al., 2018. Retrieval on source code: a neural code search. In: International Conference on Machine Learning and Programming Languages (MAPL). pp. 31–41.
Sadowski, C., Stolee, K.T., Elbaum, S., 2015. How developers search for code: a case study. In: International Conference on Foundations of Software Engineering (FSE). pp. 191–201.
Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. (TSP) 45 (11), 2673–2681.
Siddaramappa, N.N., Sindhgatta, R., Sarkar, S., et al., 2013. Semantic-based query techniques for source code.
Sirres, R., Tegawende, F., Bissyande, Kim, D., et al., 2018. Augmenting and structuring user queries to support efficient free-form code search. Empir. Softw. Eng. (ESE) 23 (5), 2622–2654.
StackExchange. Available: https://stackexchange.com/.

StackOverflow. Available: https://www.stackoverflow.com.
StackOverflow, 2008. What-is-the-simplest-sql-query-to-find-the-second-largest-value? https://stackoverflow.com/questions/32100/.
StackOverflow, 2009a. How to check if a column exists in SQL Server table? https://stackoverflow.com/questions/133031/.
StackOverflow, 2009b. Using beautiful soup, how do i iterate over all embedded text? https://stackoverflow.com/questions/830997.
StackOverflow, 2010. How-to-find-the-second-largest-value-from-a-table? https://stackoverflow.com/questions/3958346/.
StackOverflow, 2012. How to read aloud python list comprehensions? https://stackoverflow.com/questions/9061760.
StackOverflow, 2012a. Extract parts of text using regex in c#? https://stackoverflow.com/questions/11621614/.
StackOverflow, 2012b. Load data from Azure/IEnumerable to ListBox? https://stackoverflow.com/questions/19370921/.
StackOverflow, 2013. How to add elements from a dictonary of lists in python? https://stackoverflow.com/questions/17777182.
StackOverflow, 2014. Sql-how-do-i-select-a-next-max-record. https://stackoverflow.com/questions/25444735/.
Tan, M., Santos, C., Xiang, B., et al., 2015. Lstm-based deep learning models for non-factoid answer selection. arXiv preprint arXiv:1511.04108.
Tekchandani, R., Bhatia, R., Singh, M., 2018. Semantic code clone detection for Internet of things applications using reaching definition and liveness analysis. J. Supercomput. 74 (9), 4199–4226.
Tensorflow. Available: https://tensorflow.google.cn/.
Theeten, B., Vandeputte, F., Van Cutsem, T., 2019. Import2vec learning embeddings for software libraries. In: International Conference on Mining Software Repositories (MSR). IEEE, pp. 18–28.
Turian, J., Ratinov, L., et al., 2010. Word representations: a simple and general method for semi-supervised learning. In: International Conference on Association for Computational Linguistics (ACL). pp. 384–394.
Wan, Y., Zhao, Z., Yang, M., et al., 2018. Improving automatic source code summarization via deep reinforcement learning. In: International Conference on Automated Software Engineering (ASE). pp. 397–407.
Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 3034–3040.
White, M., Tufano, M., Vendome, C., et al., 2016. Deep learning code fragments for code clone detection. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 87–98.
Wong, E., Yang, N.J., Tan, N.L., 2013. Autocomment: mining question and answer sites for automatic comment generation. In: International Conference on Automated Software Engineering (ASE). IEEE, pp. 562–567.
Xu, R., Xiong, C., Chen, W., et al., 2015. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In: AAAI Conference on Artificial Intelligence (AAAI). pp. 2346–2352.
Xu, Y., Zhang, D., Song, F., et al., 2007. A method for speeding up feature extraction based on KPCA. Neurocomputing 70 (4-6), 1056–1061.
Yan, S., Yu, H., et al., 2020. Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 344–354.
Yang, D., Hussain, A., Lopes, C.V., 2016. From query to usable code: an analysis of stack overflow code snippets. In: International Conference on Mining Software Repositories (MSR). IEEE, pp. 391–402.
Yao, Z., Peddamail, J.R., Sun, H., 2019. CoaCor: Code annotation for code retrieval with reinforcement learning. In: International Conference on World Wide Web (WWW). pp. 2203–2214.
Yao, Z., Weld, D.S., Chen, W.P., et al., 2018. Staqc: a systematically mined question-code dataset from stack overflow. In: International Conference on World Wide Web (WWW). pp. 1693–1703.
Yin, P., Deng, B., Chen, E., et al., 2018. Learning to mine aligned code and natural language pairs from stack overflow. In: International Conference on Mining Software Repositories (MSR). IEEE, pp. 476–486.
Zhang, Y., Ai, Q., Chen, X., et al., 2017a. Unified representation learning for top-n recommendation with heterogenous information sources. In: International Conference on Information and Knowledge Management (CIKM). pp. 2091–2100.
Zhang, F., Niu, H., Keivanloo, I., et al., 2017b. Expanding queries for code search using semantically related api class-names. IEEE Trans. Softw. Eng. (TSE) 44 (11), 1070–1082.
Zhang, J., Wang, X., Zhang, H., et al., 2019. A novel neural source code representation based on abstract syntax tree. In: International Conference on Software Engineering (ICSE). IEEE, pp. 783–794.
Zhao, G., Huang, J., 2018. Deepsim: deep learning code functional similarity. In: International Conference on the Foundations of Software Engineering (FSE). ACM, pp. 141–151.
Zhong, V., Xiong, C., Socher, R., 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. arXiv preprint arXiv:1709.00103.
Zinkevich, M., Weimer, M., Li, L., et al., 2010. Parallelized stochastic gradient descent. In: Advances in Neural Information Processing Systems (NIPS). pp. 2595–2603.

Gang Hu received the MS degree in signal and information processing from Yunnan Minzu University, Kunming, China, in 2016. He is currently working toward the Ph.D. degree in Wuhan University. His current research focus areas include information retrieval and search-based software engineering.

Ming Peng received the MS and Ph.D. degree from the Wuhan University of China, in 2002 and 2006. She is currently a professor at School of Computer Science, Wuhan University. Currently, she works on NLP as information retrieval and knowledge graph. She is a member of the CCF.

Yihan Zhang is currently working toward the BS degree in National University of Singapore. His research interests include information retrieval and code recommendation.

Qianqian Xie received the BS degree from Jiangxi Normal University, China, in 2016. She is currently working toward the Ph.D. degree in the School of Computer Science at Wuhan University. Her research interests include sparse coding and deep learning.

Mengting Yuan received the MS and Ph.D. degree from Wuhan University, Wuhan, China, in 2006. He is currently an associate professor at School of Computer Science, Wuhan University. His current research interest includes mining software repositories and deep learning.