Article

Incorporating Code Structure and Quality in Deep Code Search

Hao Yu, Yin Zhang * , Yuli Zhao * and Bin Zhang

Software College, Northeastern University, Shenyang 110169, China; 1901824@stu.neu.edu.cn (H.Y.); zhangbin@mail.neu.edu.cn (B.Z.)

* Correspondence: zhangyin@mail.neu.edu.cn (Y.Z.); zhaoyl@swc.neu.edu.cn (Y.Z.)

Abstract: Developers usually search for reusable code snippets to improve software development efficiency. Existing code search methods, including methods based on full-text or deep learning, have two disadvantages: (1) ignoring structural information of code snippets, such as conditional statements and loop statements, and (2) ignoring quality information of code snippets, such as naming clarity and logical correctness. These disadvantages limit the performance of existing code search methods. In this paper, we propose a novel code search method named Structure and Quality based Deep Code Search (SQ-DeepCS). SQ-DeepCS introduces a code representation method called program slice to represent structural information as well as API usage of code snippets. Meanwhile, SQ-DeepCS introduces a novel deep neural network named Method-Description-Joint Embedding Neural Network (MD-JENN) to weight the quality of code snippets. To evaluate the proposed methods, we train MD-JENN and evaluate SQ-DeepCS by searching for code snippets with respect to the top-rated questions from Stack Overflow. We use four evaluation indicators to measure the effectiveness of SQ-DeepCS: FRank, SuccessRate@k, PrecisionRate@k, and Mean Reciprocal Rank (MRR). The experimental results show that our approach can provide better results than existing techniques when searching for relevant code snippets.

Keywords: code search; attention mechanism; semantic space mapping

Citation: Yu, H.; Zhang, Y.; Zhao, Y.; Zhang, B. Incorporating Code Structure and Quality in Deep Code Search. *Appl. Sci.* **2022**, *12*, 2051. <https://doi.org/10.3390/app12042051>

Academic Editor: Byung-Gyu Kim

Received: 25 December 2021

Accepted: 12 February 2022

Published: 16 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

To develop software efficiently, software developers often find and reuse existing code snippets by searching over professional codebases, such as GitHub [1–4]. Driven by information needs, developers will submit queries expressed in natural language and expect for code snippets satisfying their needs. However, code snippets and natural language queries are heterogeneous, and thus it is hard to locate code snippets that meet user's intent [5].

Traditional information retrieval methods toward code searching are usually based on text vocabulary matching [6]. For example, Lv et al. [7] combined text similarity and API sequence matching and proposed an extended Boolean model named CodeHow. Linstead et al. [8] proposed Sourcerer, a code search tool that combines structural information with text vocabulary information using information retrieval techniques. Since code snippets and natural language queries have obvious heterogeneous characteristics [9,10], code snippets that can fulfill information needs do not necessarily contain submitted query words or natural language words with similar semantics. As a result, the performance of traditional text vocabulary-based code search methods are greatly limited.

Gu et al. [11] brought joint embedding technology to code search to deal with the flaw and proposed a code search tool named DeepCS. The key idea of joint embedding technology is to transform heterogeneous inputs into shared vector space. With joint embedding technology, DeepCS embedded code snippets and natural language descriptions into a high-dimensional vector space. As a result, a code snippet and its corresponding description will occupy nearby regions of the space. By calculating the vector similarity of the embedded vectors of the code snippet and its corresponding description, code search tools can retrieve related code snippets that are more in line with users' expectations.

With the development of machine learning technology in recent years, many different methods have gradually sprung up in the field of code search. Sachdev et al. [2] proposed an unsupervised technique named Neural Code Search (NCS). NCS extracts specific keywords from code snippets and uses only the word embedding mechanism to obtain the vector of code snippets. Yao [12] used Tree LSTM to process the abstract syntax tree of code snippets and proposed a new code search method named At-CodeSM. These methods used different techniques to extract the semantic features of code snippets and completed the code search task by comparing the similarity between vectors of code snippets and natural language queries.

Although DeepCS showed quite good results on some datasets, we could still notice certain disadvantages. Firstly, DeepCS ignored certain structural information when representing code snippets, such as conditional statements and loop statements. Structural information reflects the execution order of code snippets, and thus is an essential part of code semantics [13]. DeepCS treated structural information as chain links, which ignores the semantics contained in the structure of code snippets, and thus limited the performance of code search.

Secondly, DeepCS ignored the quality information of code snippets, such as naming clarity and logical correctness. The quality of codes from large codebases may vary. Take the two code snippets shown in Figure 1 as an example. The method name of the first code snippet cannot clearly reflect the purpose of this code, and the variable names in the second code snippet do not conform to naming conventions. Meanwhile, the purposes of the two code snippets are the same. That is to say, they should have a similar ranking order in the results returned by code search tools. However, when representing code snippets, DeepCS assigns the same weight value to all features, such as method names and tokens. As a result, the ranking order of the first code is far lower than that of the second code because of the incomprehensible method name.

<pre>func getK(m map[int]int) []int{ keys := make([]int, 0, len(m)) for k := range m { keys = append(keys, k) } return keys }</pre>	<pre>func getKeys(m map[int]int) []int{ a := make([]int, 0, len(m)) for k := range m { a = append(a, k) } return a }</pre>
---	--

Figure 1. Example code snippets.

The goal of this paper is to overcome the aforementioned problems and to improve code search performance. For this purpose, we propose a novel code search method named SQ-DeepCS. Firstly, we introduce a novel code representation method called *program slice* to preserve structural information and data information when representing code snippets. Program slice is a formal representation of function body. It preserves structural information on the basis of API linear sequence [11]. Secondly, we introduce attention mechanism [11,14] to weight the quality of code snippets and propose a novel deep learning model MD-JEnn. MD-JEnn is a bi-directional long-short term memory (BLSTM) based deep learning model. It leverages attention mechanism to weight the quality of code snippets. To evaluate the proposed methods, we train MD-JEnn and evaluate SQ-DeepCS by searching for code snippets with respect to the top-rated questions from Stack Overflow. We use four evaluation indicators to measure the effectiveness of SQ-DeepCS. The experimental results show that our approach can provide better results than existing techniques when searching for relevant code snippets.

2. Related Work

2.1. Recurrent Neural Network

In code search, code snippets and natural language queries are required to be embedded into vectors so that their semantic similarity can be measured. Variable-length

sequential data, such as code snippet and natural language query, are often preprocessed by recurrent neural network (RNN). RNN is composed of multiple neural network units and takes sequential data as input [15–17]. RNN has the ability to map sequential input into sequential hidden state. Compared with the ordinary fully connected network, the current time step output of the neurons in RNN hidden layers depends not only on the input of the current time step but also on the output of the previous time step. This feature of RNN is particularly suitable for processing code [18].

With the length of input sequence increasing, RNN will face the problems of long-term dependencies [19,20]. To alleviate this problem, Xu introduced bi-directional long-short term memory network (BLSTM) [21]. BLSTM combined memory cells and RNN to preserve memory information. BLSTM controls selective memory and forgetting information through three gating units: forget gate, input gate, and output gate. The forget gate determines how much information in the previous time step can be preserved to the current time step. The input gate determines how many input signals will be fused, and the output gate controls how much memory is finally output. The specific calculation method is defined as follows:

$$i_t = \text{sigmoid}(W_i X_t + V_i H_{t-1} + b_i) \quad (1)$$

$$f_t = \text{sigmoid}(W_f X_t + V_f H_{t-1} + b_f) \quad (2)$$

$$o_t = \text{sigmoid}(W_o X_t + V_o H_{t-1} + b_o) \quad (3)$$

$$g_t = \tanh(W_g X_t + V_g H_{t-1} + b_g) \quad (4)$$

where i_t, f_t, o_t, g_t represent the input state, the forget state, the output state, and the unit state of the current time step. X_t is the input signal of the current time step. H_{t-1} represents the output signal of the previous time step. W, V, b are the coefficient matrix of BLSTM to be trained. By training these weights, BLSTM can selectively ignore or strengthen the current memory c or input signals according to the current input signals and memory information. In this way, BLSTM better learns the semantic information of long sentences. c and h are determined by:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot g_t \quad (5)$$

$$h_t = o_t \cdot \tanh(c_t) \quad (6)$$

where c_t represents the memory signal of current time step t . h_t is the hidden state of t . The above process can be simplified as:

$$h_t = \tanh(W_D[h_{t-1}; w_t] + b_D), \forall t = 1, 2, \dots, N_D \quad (7)$$

where $h_t, t = 1, 2, \dots, N_D$ are the hidden states of BLSTM. $[x; y] \in R^{2d}$ is a concatenate operator to integrate x and y . $\tanh$ is a commonly used activation function. $w_t \in R^d$ represents the embedded representation of the natural language word w_t . W^D and b^D are the weight matrix and bias matrix of BLSTM. Experimental results show that BLSTM can overcome RNN when processing long sequential data in code search [11].

2.2. Attention Mechanism

When embedding sequential inputs into a vector space, RNN assigns the same information weight to each input feature. However, since different features of code snippets have different qualities, it is necessary to give higher weight to the high-quality features. The problem of assigning different weight to different features has been extensively studied, and one of the most effective methods is the attention mechanism [14,22–25].

Attention method contains a randomly initialized global attention vector $\alpha \in R^d$. For a set of combined context vectors $\tilde{c}_1, \tilde{c}_2, \dots, \tilde{c}_n$, attention method calculates an attention weight α_i for each $\tilde{c}_i$ as the normalized inner product between the context vectors and the global attention vector α :

$$\alpha_i = \frac{\exp(\tilde{c}_i^T \cdot \alpha)}{\sum_{j=1}^n \exp(\tilde{c}_j^T \cdot \alpha)} \quad (8)$$

$$v = \sum_{i=1}^n \alpha_i \cdot \tilde{c}_i \quad (9)$$

where α_i is a softmax function in conventional form. According to the properties of the softmax function, the attention weights are positive and add up to 1. Attention weight α_i can be regarded as a weighted average and can be trained to represent the importance of combined context vectors. Finally, representative vector $v \in R^d$ combines the characteristics of each context vector. In the domain of code search, Alon et al. have introduced the attention mechanism to weight the critical paths of codes [14]. In our work, we leverage attention mechanism to solve the problem of different code quality.

2.3. Joint Embedding Mechanism

Joint embedding mechanism, which is also known as multi-model embedding, is usually used to model the relation between two heterogeneous data [26]. Consider two heterogeneous datasets X and Y with some semantic association. The semantic association can be expressed as a mapping function f :

$$f : X \rightarrow Y \quad (10)$$

Since X and Y are heterogeneous datasets, different embedding technologies need to be used in order to embed X and Y into a unified vector space. The semantic relation between X and Y is measured by calculating similarity of the two embedded vectors. The goal of joint embedding is to make semantically similar concepts across the two modalities occupy nearby regions of the space [27]. This process can be expressed as:

$$f : X \xrightarrow{\varphi} v_X \rightarrow S(v_X, v_Y) \leftarrow v_Y \xleftarrow{\tau} Y \quad (11)$$

where φ, τ are the different embedding function to transform X, Y into the same semantical space by setting the dimensions of two embedded vectors v_X, v_Y to the same. $S(v_X, v_Y)$ represents a similarity measure (e.g., cosine) to evaluate the matching degrees of v_X and v_Y . In this way, the mapping function can model the semantic relation between two heterogeneous datasets X and Y .

3. Code Representation

Existing code representation methods generally treat a code snippet as three parts: the method name, the API sequence, and the tokens. The API sequence is generated by traversing the AST tree of the code snippet. The API sequence treats the calling of API functions in the code snippet as a chain link. As a result, the API sequence ignores semantics contained in the structure of the code snippet. To overcome this disadvantage, this section introduces a novel code representation method called *program slice* to preserve structural information when representing code snippets as follows:

Method Name Representation: Method name of each code snippet is divided into a sequence of tokens. These tokens are split according to the corresponding naming conventions such as camelCasing or under_scores. The token sequence is then lowercased. For example, the method name 'Write2File' will be transformed into token sequence {write, 2, file}.

Program Slice Representation: Program slice extends API sequence to preserve structural information of method body [11]. Program slice is generated by two steps: (1) parsing AST of a code snippet [28,29] and (2) applying static analysis method on the ASTs [30]. For different statements of code snippet, the processing methods are as follows:

- For each variable declaration statement, program slice analyzes its corresponding variable type and adds *new* keyword before the variable type. Taking Golang as an example, *var a string* is transformed to *new string*.

- For each condition statement, program slice retains its structural information and *if* / *else* keywords. For example, *if*(s_1){ s_2 ;} *else*{ s_3 ;} is transformed to *if*(p_1){ p_2 ;} *else* { p_3 ;} , where p_1 , p_2 , and p_3 are the program slices of statements s_1 , s_2 , and s_3 .
- For each loop statement, program slice retains its judgment conditions and cycle body and adds the *for* keyword. For example, *for*(c_1 ; c_2 ; c_3){ s_4 ;} is transformed to *for*(p_2){ p_4 ;} , where p_2 and p_4 are the program slices of condition c_2 and statement s_4 .
- For each range statement, program slice retains its judgment conditions and cycle body and adds the *range* keyword. For example, *for* $i := \text{range } c_1$ { s_2 ;} is transformed to *range*(p_1){ p_2 ;} , where p_1 and p_2 are the program slices of condition c_1 and statement s_2 .
- For each return statement, program slice analyzes its returned variable type and adds *return* keyword before the variable type. For example, *return* 10 is transformed to *return int*.
- For each switch statement, program slice retains its judgment conditions and body and adds the *switch* keyword. For example, *switch* c_1 { s_2 } is transformed to *switch*(p_1){ p_2 ;} , where p_1 and p_2 are the program slices of condition c_1 and statement s_2 .
- For each call expression, program slice retains only the object type and the called method name. For example, for *a.b*() or *c*() , program slice generates *A.b* or *C*, where *A* and *C* are the names of class or struct for object *a* and *c*.
- For each nested function calls, program slice uses nested definitions to represent each layer of function calls. For example, *a.b(c.d(e.f()))* is transformed to *A.b(C.d(E.f()))*, where *A*, *C*, *E* are the name of class or struct for object *a*, *c*, *e*.
- For each operation expression, program slice preserves operation symbols and result types. For example, *a = 1 + 2* is converted to *+int*.

Token Representation: Method body of code snippets are tokenized according to blank space and processed in the same way of method name representation. Duplicated tokens and tokens in a stop word list are removed.

Figure 2 shows an example of code representation extracted from a Golang method.

Figure 2. An example of code representation.

4. Model

Existing code search methods take code snippets and the corresponding code annotations as input and use joint embedding technology to model the semantic relation between code snippets and annotations. When using RNN to embed sequential inputs into a vector with semantic information, each input has the same influential weight. However, since different features of code snippets, such as method name and program slice, have different qualities, it is necessary to give higher weight to the high-quality features. Thus, we introduce attention mechanism to weight the quality of code snippets and propose a novel deep learning model MD-Jenn.

MD-Jenn embeds code snippets and descriptions into hidden vectors using BLSTM. According to attention mechanism, the hidden vectors are integrated into vectors that respectively represent code snippets and descriptions. MD-Jenn then jointly embeds the

two vectors into the same space to calculate the similarity. By this means, a code snippet and its corresponding description can be embedded into nearby vectors through iterative training. The details of our model are introduced in the following subsections.

4.1. Architecture

Figure 3 shows the overall structure of the MD-JEnn. MD-JEnn is divided into three components: the description embedding module, the method feature embedding module, and the similarity module. Each of them corresponds to a part of joint embedding:

- *Description embedding module (DE-Module)* embeds natural language descriptions into description vectors.
- *Method feature embedding module (MF-Module)* embeds code snippets into code vectors.
- *Similarity module* calculates the degree of similarity between description vectors and code vectors.

Figure 3. The overall structure of the MD-JEnn.

The following subsections describe the detailed design of these modules. Figure 4 shows the detailed structure of MD-JEnn.

Figure 4. The detailed structure of the MD-JEnn.

4.1.1. Description Embedding Module

Description embedding module (DE-Module) embeds natural language code annotations into description vectors. The first sentence in the code annotation of a code snippet

always represents the summary of the entire code snippet. In order to obtain the embedded description vector, DE-Module processes a natural language description in the following steps:

Firstly, DE-Module takes a natural language description as input and outputs an embedded feature vector. In DE-Module, BLSTM considers a natural language description $D = \{w_1, w_2, \dots, w_{N_D}\}$ as a sequence that contains N_D words. BLSTM takes the description as input and calculates the hidden state according to each time step. BLSTM updates the hidden state h_t at time t by concentrating input word w_t and preceding hidden state h_{t-1} :

$$h_t = \tanh(W^D[h_{t-1}; \mathbf{w}_t] + b^D), \forall t = 1, 2, \dots, N_D \quad (12)$$

where $h_t, t = 1, 2, \dots, N_D$ are the hidden states of BLSTM. $[x; y] \in R^{2d}$ is a concatenate operation to integrate x and y . $\tanh$ is a commonly used activation function. $\mathbf{w}_t \in R^d$ represents the embedded representation of natural language word w_t . W^D and b^D are the weight matrix and bias matrix of BLSTM (either of the matrices is bi-directional). In this way, a description is embedded into N_D d -dimensional embedded feature vectors.

Secondly, since some words in a description are important, it is necessary to assign higher weights to these important words. For example, the words *write* and *file* are more important as they express the key semantics of the description *write to a file*. To give higher weights to important words, MD-Jenn introduce attention mechanism to aggregate the embedded vectors of the description into a represented vector by calculating a scalar weight for each vector of the description word. The individual vectors are aggregated to a represented description vector v via attention:

$$\alpha_i = \frac{\exp(\tilde{h}_i^T \cdot \alpha)}{\sum_{j=1}^n \exp(\tilde{h}_j^T \cdot \alpha)} \quad (13)$$

$$v = \sum_{i=1}^n \alpha_i \cdot \tilde{h}_i \quad (14)$$

where $\alpha \in R^d$ is initialized in random, $h_t, t = 1, 2, \dots, N_D$ are the hidden states of the previous BLSTM layer. α_i is the attention weight of each h_i . The exponents in α_i are the common form of softmax function. The vector d can be treated as the representation of the input description. The description vector d considers both lexical information and semantic information. In this way, DE-Module can emphasize the key words in a description.

4.1.2. Method Feature Embedding Module

Method feature embedding module (MF-Module) embeds code representations into code vectors. A code snippet can be represented into a code representation $C = [M, S, K]$ using the code representation method described in Section 3, where $M = m_1, m_2, \dots, m_{N_M}$ is the sequence of N_M tokens and represents the method name of the code snippet; $S = s_1, s_2, \dots, s_{N_S}$ is the program slice with N_S consecutive tokens; $K = \{k_1, k_2, \dots, k_{N_K}\}$ is the collection of tokens that appeared in the code snippet. Each part of the code snippet is embedded into partial embedding vectors. In order to highlight the high-quality parts of the code snippet, these partial embedding vectors are concentrated into a represented code vector with attention mechanism. MF-Module processes the code representation C according to the following steps:

Firstly, MF-Module embeds the method name M into a sequence of separated tokens using BLSTM:

$$h_t = \tanh(W^M[h_{t-1}; \mathbf{m}_t] + b^M), \forall t = 1, 2, \dots, N_M \quad (15)$$

where $h_t, t = 1, 2, \dots, N_M$, are the hidden states of BLSTM. $m_t \in R^d$ represents the embedded representation of tokens. W^M and b^M are the weight matrix and bias matrix of BLSTM.

Secondly, MF-Module embeds program slice S into N_S d -dimensional hidden vectors h_t :

$$h_t = \tanh(W^S[h_{t-1}; s_t] + b^S), \forall t = 1, 2, \dots, N_S \quad (16)$$

where $h_t, t = 1, 2, \dots, N_S$, are the hidden states of the BLSTM, $s_t \in R^d$ represents the embedded representation of the tokens in program slice s_t , W^S and b^S are the weight matrix and bias matrix of BLSTM.

Finally, as the tokens K are not strictly ordered, MF-Module embeds the tokens K by a fully connected layer:

$$h_t = \tanh(W^K k_t), \forall t = 1, 2, \dots, N_K \quad (17)$$

where $h_t, t = 1, 2, \dots, N_K$, are the embedding vectors of tokens. $k_t \in R^d$ represents the embedded representation of tokens. W^K is the weight matrix of the fully connected layer.

Since different features of code snippets have different qualities, it is necessary to give higher weight to the high-quality features. After embedding the three components (method name, program slice, and tokens) of a code snippet, MF-Module emphasizes the high-quality parts of the code snippet with attention mechanism:

$$h_t = [\mathbf{m}; \mathbf{s}; \mathbf{k}] \quad (18)$$

$$\alpha_i = \frac{\exp(\tilde{h}_i^T \cdot \alpha)}{\sum_{j=1}^n \exp(\tilde{h}_j^T \cdot \alpha)} \quad (19)$$

$$c = \sum_{i=1}^n \alpha_i \cdot \tilde{h}_i \quad (20)$$

where $[x; y; z]$ is a concatenate operate to integrate x , y , and z . $\alpha \in R^d$ is initialized in random, $h_t, t = 1, 2, \dots, N_H$, are H d -dimensional vectors. α_i is the attention weight of each $\tilde{h}_i$. In this way, the code vector c can be viewed as the final representation of the code snippet.

In summary, a code snippet is first processed into three features: the method name, the program slice, and the tokens. In the training phase, each feature of <method name, program slice, tokens> is embedded into a feature vector by BLSTM, BLSTM, and MLP, accordingly. These feature vectors are then concatenated into a code vector through an attention layer.

4.1.3. Similarity Module

Joint embedding mechanism needs a similarity calculation method to form a unified vector space. In order to joint embed the code vectors c and the description vectors d , similarity module uses cosine similarity as the measurement:

$$\cos(c, d) = \frac{c^T d}{\|c\| \|d\|} \quad (21)$$

where c and d are the code vector and description vector. Since the similarity marks the degree of correlation, similarity module aims to make semantically similar vectors occupy adjacent spatial regions. To summarize, MD-JEnn receives an input as <code, description> pair and calculates the cosine similarity $\cos(c, d)$ of them. The similarity module is only responsible for calculating the similarity. The semantic relation between code snippets and natural language descriptions is obtained through model training described in the next subsection.

4.2. Model Training

To train MD-JEnn, a training dataset containing triples $\langle C, D+, D- \rangle$ needs to be constructed, where $D+$ represents a similar description that describes the functionality of code C , and $D-$ represents a dissimilar description. In principle, the similarity value

of C and $D+$ should be higher than C and $D-$. Therefore, the training algorithm can be expressed as:

$$L = \begin{cases} 0, & m(C, D+) - m(C, D-) < \lambda \\ \lambda - (m(C, D+) - m(C, D-)), & \text{else} \end{cases} \quad (22)$$

where $m(A, B)$ denote the cosine similarity of A and B . L is the loss function. λ is a threshold that the similarity of positive sample should be higher than negative sample. The advantage of loss function L is that it does not force the classification of a single sample but learns the relation of samples. This method reduces the difficulty of building datasets.

Figure 5 is an example of a training dataset. In each epoch of model training, the ranking loss encourages the cosine similarity between a code snippet and its correct description to go up and the cosine similarities between a code snippet and incorrect descriptions to go down. In this way, the semantic relation between code snippets and natural language descriptions is established.

Figure 5. An example of training dataset.

5. Evaluation

5.1. Experimental Setup

In order to evaluate the performance of SQ-DeepCS, we constructed a codebase to train our model and varify the result. The codebase consists of Golang code snippets from GitHub repos. To ensure data quality, we only chose repos with more than 20 stars. For code snippets with comments, we treat the comments as the descriptions of the code snippets. The code snippets and descriptions are represented into triples <number, code representation, description representation>. For code snippets without descriptions, we represented them as <number, code representation, none>. The codebase consists of 218,072 triples with descriptions and 566,103 triples without descriptions. As in [11], we use code snippets with descriptions for training and all the code snippets for result verification.

For a natural language query, SQ-DeepCS returns top K most relevant code snippets calculated by the MD-JEnn model. The use of SQ-DeepCS consists of three steps: (1) offline training, (2) offline codebase embedding, and (3) online code searching. As follows:

- (1) *Offline training*: SQ-DeepCS is trained by the method described in Section 4 through code snippets with descriptions in the codebase.
- (2) *Offline search codebase embedding*: SQ-DeepCS embeds all the code snippets in the codebase to a set of code vectors. These vectors will be cached for fast similarity calculation.
- (3) *Online code searching*: In this part, when a user submits a natural language query, SQ-DeepCS embeds the query into a query vector using the DE-Module and calculates the cosine similarity between the query vector and the vectors in the codebase. The top K relevant results will be returned to the user.

Before offline training, the codebase is preprocessed using the method described in Section 3 to train MD-JEnn. The offline training parameters are as follows: All BLSTMs have 200 hidden units in each direction. The dimension of word embedding is 100. MD-JEnn has two types of multilayer perceptron (MLP). The number of hidden units of the MLP for embedding individual tokens is 100. The number of hidden units of MLP for combining the

embedding vectors of different aspects is 400. We consider K as 1, 5, and 10 when returning the top K relevant results.

MD-Jenn model is trained via the mini-batch Adam algorithm. The batch size is set as 128. We limit the size of vocabulary to 20,000 words that are the most frequently used words in the training dataset. We build our model on Keras, an open-source deep learning framework and train models on a server with one NVidia 1080Ti GPU. The training lasts for nearly 13 h with 200 epochs.

5.2. Evaluation Method

5.2.1. Evaluation Questions

To evaluate the effectiveness of SQ-DeepCS, we establish an evaluation question set that consists of 45 top voted Golang programming questions collected from Stack Overflow. We follow the following criteria to choose the 45 questions from the list of top voted Golang questions in Stack Overflow:

- (1) The question should be a concrete and achievable programming task. There are various questions in Stack Overflow, but some problems are not related to programming tasks. For example, ‘When is the `init()` function run’, ‘What should be the values of `GOPATH` and `GOROOT`’, and ‘Removing packages installed with `go get`’. We only retain questions related to concrete and achievable programming tasks and filter out other problems such as knowledge sharing and judgement.
- (2) The question should not be a duplication of another question. We only retain the unique questions and remove all similar questions.
- (3) The question should have an accepted answer and the answer should contain a code snippet to solve the current question.

5.2.2. Evaluation Metrics

We submit 45 questions to SQ-DeepCS and obtained the corresponding search results. In order to evaluate the results, five experienced developers were invited to determine whether the results can or cannot resolve the submitted questions. The decisions made by the developers are binary. When the developers hold different opinions on the results, they will discuss to reach an agreement.

We use four evaluation indicators to measure the effectiveness of SQ-DeepCS: FRank, SuccessRate@ k , PrecisionRate@ k , and MRR. All of them are widely used in the research domain of information retrieval and code searching.

FRank (also known as the best hit rank) is the rank of the first correct result in the query result list. The assumption behind FRank is that users often browse down from the first search result. The smaller the FRank of the search result, the less effort the users make to obtain correct results. We use FRank to evaluate the effectiveness of a single search and record the corresponding value of each query.

SuccessRate@ k (also known as the success percentage at k) represents the percentage of the queries for which more than one correct result could exist in the top k ranked results [31–33]:

$$SuccessRate@k = \frac{1}{|Q|} \sum_{q=1}^Q \delta(FRank_q \leq k) \quad (23)$$

where Q is the number of all queries. $\delta(\cdot)$ is a function which returns 0 if the input is false and 1 otherwise. SuccessRate@ k counts the number of queries with correct results in the first k results of all the queries and then divides it by the total number of queries. A good code search engine should help users find correct search results in shorter time so as to save their query cost. The higher the SuccessRate@ k , the better the code search performances.

Precision@k is a variant of accuracy which calculates the average value of the quality of query results. In our evaluations, it is calculated as:

$$Precision@k = \frac{1}{|Q|} \sum_{q=1}^Q \frac{relevant_{q,k}}{k} \quad (24)$$

where $relevant_{q,k}$ represents the number of search results related to the query statement in the top k results of the q th query. Precision@k is important because developers often inspect multiple results of different usages to learn from [34]. A better code search algorithm should allow less noisy results so that users can obtain more relevant results. In our experiment, the higher the Precision@k, the better the code search performances. For FRank, Success@k, and PrecisionRate@k, we recorded their values when k is set to 1.5 and 10. For MRR, we only recorded the value when k equals 10.

MRR is the average of the reciprocal ranks in the results of all queries. The reciprocal rank of a query is the reciprocal of the ranking of the first correct search result. MRR is calculated by the following formula:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^Q \frac{1}{FRank_q} \quad (25)$$

The idea behind MRR is the reciprocal of the sorting results: the MRR score of the first result is 1, the score of the second result is 0.5, and the score of the n th match result is $1/n$. In our experiment, the score of not finding (NF) the correct result in the top 10 returned results is recorded as $1/11$. The higher the MRR value, the better the code search performance.

5.3. Compared Method

We compare our method with DeepCS. DeepCS embeds a code snippet and the corresponding description into four vectors: the method name vector, the API sequence vector, the token vector, and the description vector. DeepCS uses BLSTM and maxpooling to embed method names, API sequences, and descriptions. For code snippet tokens, DeepCS simply uses an MLP. The vectors of method name, API sequence, and tokens are fused into one vector through a fully connected layer.

We improved DeepCS by introducing program slice and attention mechanism. To understand the impact of introducing program slice and attention mechanism, we designed several comparison methods based on DeepCS:

- (1) *P-DeepCS*: Program slice based DeepCS (P-DeepCS) uses program slice instead of API sequence in DeepCS. This is to explore the impact of introducing program slice.
- (2) *A-DeepCS*: Attention based DeepCS (A-DeepCS) uses attention layers instead of maxpooling layers in DeepCS. This is to explore the impact of introducing attention mechanism.
- (3) *PA-DeepCS*: Program slice and attention based DeepCS (PA-DeepCS) use program slice to represent method body and use attention mechanism to weight embedded vector. However, when fusing the vectors of method name, program slice, and tokens, PA-DeepCS still uses a fully connected layer.
- (4) *SQ-DeepCS*: SQ-DeepCS is our proposed method. The key difference of PA-DeepCS and SQ-DeepCS is that SQ-DeepCS adopts an attention layer to concentrate the vectors of method name, program slice, and tokens.

We also formulate experiments in the context of the code search techniques mentioned in Section 1:

- (1) *NCS*: NCS extracts some specific keywords as semantic features of code snippets, such as method name, method invocations, and enums. NCS combines these keywords

with fastText [35] and conventional IR techniques, such as TF-IDF. The encoder of NCS adopts unsupervised training mode.

- (2) *At-CodeSM*: *At-CodeSM* used Tree LSTM [36] to process the abstract syntax tree of code snippets. *At-CodeSM* extracted three features of code snippets: method name, token, and AST. For tokens of code snippets, *At-CodeSM* use LSTM and attention mechanism to obtain the vector of tokens. When fusing the vectors of method name, AST, and tokens, *At-CodeSM* uses a fusion layer.

5.4. Results

Table 1 shows the evaluation queries and the corresponding FRank(DCS: DeepCS; AD: A-DeepCS; PD: P-DeepCS; PAD: PA-DeepCS; SQD: SQ-DeepCS; NCS and *At-CodeSM* are not given in the table due to space limitations). In Table 1, 'NF' represents Not Found, which means there is no relevant result in the top K results in this query (here we consider K as 10). A FRank value of 1 indicates that a user can obtain reusable code snippets in the first search result. The FRank value of 1 that appeared in DeepCS and SQD is 8 and 10. The number of 'NF' in DeepCS and SQD is 20 and 16. Therefore, SQD outperforms DeepCS from the perspective of FRank. The FRank value of 1 that appeared in AD, PD, and PAD is 6, 7, and 9. The number of 'NF' in AD, PD, and PAD is 22, 21, and 16. The results show that AD and PD perform even worse than DeepCS, while PAD outperforms DeepCS a bit but still worse than SQD. For NCS and *At-CodeSM*, the number of 'NF' is 20 and 17. In this regard, the effect of NCS is similar to that of DeepCS, slightly inferior to that of *At-CodeSM*, while SQD still maintains a good effect. The FRank value of 1 that appeared in NCS and *At-CodeSM* is 7 and 9. It means that when we only consider the top search results, SQD still performs well.

Figure 6 shows the box-plot of FRank for the several approaches. The vertical axis represents the FRank values from 1 to 11, where we regard 'NF' as the FRank value of 11. The symbol '+' and the horizontal line in the box represent the mean and median of FRank. We can observe that the average FRank score of SQ-DeepCS is 6.09, which is lower than the average FRank score of DeepCS (6.94). This result shows that, from the perspective of the ranking of first useful search results, we have improved about 0.85 on the basis of DeepCS. The average FRank scores of AD, PD, and PAD are 6.96, 7.24, and 6.51. For NCS and *At-CodeSM*, the average FRank scores are 7.09 and 6.22. The results show that at the FRank level, the performance of SQD is ahead of other methods.

Figure 6. The box-plot of FRank for SQ-DeepCS and related approaches.

Table 2 shows the overall accuracy of SQ-DeepCS and related approaches. The results show that the performance differences of A-DeepCS, P-DeepCS, and DeepCS are not significant. For example, S@10 of PD is 0.533, which is 2.3% lower than that of DeepCS. From these results, we could see that the introduction of program slice or attention mechanism may not improve the performance of DeepCS.

Compared with DeepCS, S@k of PAD are improved by 4.4%, 2.2%, 8.8%, and P@k of PAD are improved by 4.4%, 1.8%, and 2.5%. For MRR, the improvement to DeepCS is 3.3%. The result shows that PA-DeepCS outperforms DeepCS a bit.

Table 1. The evaluation queries and the corresponding FRank.

No.	Query	FRank				
		DCS	AD	PD	PAD	SQD
1	how to check if a map contains a key in go?	NF	NF	1	7	1
2	how to convert an int value to string in go?	NF	2	7	2	3
3	how to check if a file exists in go?	1	2	1	1	2
4	how to find the type of an object in go?	NF	2	1	1	4
5	read a file in lines	1	NF	NF	6	NF
6	how to multiply duration by integer?	8	3	4	1	2
7	how to generate a random string of a fixed length in go?	5	1	NF	1	NF
8	checking the equality of two slices	3	NF	NF	NF	3
9	how can i read from standard input in the console?	NF	10	NF	10	NF
10	how to write to a file using go	2	1	1	1	1
11	how do i send a json string in a post request in go	3	5	2	4	10
12	getting a slice of keys from a map	NF	NF	1	9	8
13	convert string to int in go?	4	1	10	9	5
14	how to get the directory of the currently running file?	NF	1	6	NF	1
15	convert byte slice to io.Reader	NF	NF	NF	NF	NF
16	how to set headers in http get request?	4	1	1	1	3
17	how to trim leading and trailing white spaces of a string?	NF	4	8	NF	1
18	how to connect to mysql from go?	9	NF	NF	NF	NF
19	how to parse unix timestamp to time.Time	NF	NF	2	2	10
20	subtracting time.duration from time in go	3	2	NF	NF	NF
21	convert string to time	3	NF	9	6	1
22	how to set timeout for http get requests in golang?	NF	NF	2	NF	5
23	how to set http status code on http responsewriter	NF	2	3	10	NF
24	how to stop a goroutine	NF	NF	NF	NF	7
25	how to find out element position in slice?	NF	NF	NF	5	NF
26	partly json unmarshal into a map in go	6	3	NF	1	6
27	how to index characters in a golang string?	10	NF	1	NF	NF
28	how to get the name of a function in go?	3	NF	NF	NF	1
29	convert go map to json	7	2	NF	2	3
30	convert interface to int	1	NF	9	10	NF
31	how to convert a bool to a string in go?	NF	9	2	2	1
32	obtain user's home directory	NF	4	9	NF	NF
33	how do i convert a string to a lower case representation	NF	1	NF	1	6
34	how do i compare strings in golang?	NF	4	7	6	NF
35	mkdir if not exists using golang	NF	3	NF	3	NF
36	sort a map by values	NF	NF	1	5	2
37	how to check if a string is numeric in go	NF	NF	NF	NF	4
38	convert an integer to a float number	1	5	NF	NF	NF
39	delete max in BTree	1	NF	3	5	1
40	how to clone a map	1	NF	NF	NF	1
41	string to date conversion	2	NF	NF	NF	NF
42	get database config	1	NF	4	2	1
43	convert string to duration	1	3	NF	1	2
44	generate random integers within a specific range	7	NF	NF	NF	NF
45	generate an md5 hash	5	NF	NF	3	3

SQ-DeepCS outperforms PA-DeepCS in most indicators expected for P@10. The key difference of PA-DeepCS and SQ-DeepCS is that SQ-DeepCS adopts an attention layer to concentrate the vectors of method name, program slice, and tokens. The result implies that considering the quality of different parts of code snippets can significantly improve the search performance. By using attention mechanism when fusing method name, API sequence, and tokens of the code snippet, SQ-DeepCS gives higher weight to the high-quality parts, so as to better improve the search performance.

For NCS and At-CodeSM, we can see that SQ-DeepCS once again takes the lead in the search performance evaluated by S@k, P@k, and MRR. Compared with NCS, MRR of SQ-DeepCS is improved by 12.3%. In addition, it is worth noting the difference between PA-DeepCS and At-CodeSM. PA-DeepCS outperforms At-CodeSM in most indicators. The key difference between them is the representation method of code snippets. The result

implies that program slice performs better than AST in expressing the semantic information of code snippets. We speculate that this is because program slicing removes the special features in AST and uses more generalized common features instead.

Table 2. Overall accuracy of SQ-DeepCS and related approaches.

Approach	S@1	S@5	S@10	P@1	P@5	P@10	MRR
DeepCS	0.156	0.422	0.556	0.156	0.164	0.162	0.315
A-DeepCS	0.133	0.467	0.511	0.133	0.169	0.153	0.304
P-DeepCS	0.156	0.356	0.533	0.156	0.16	0.149	0.313
PA-DeepCS	0.2	0.444	0.644	0.2	0.182	0.187	0.348
NCS	0.156	0.356	0.511	0.156	0.147	0.142	0.251
At-CodeSM	0.178	0.467	0.578	0.178	0.178	0.16	0.323
SQ-DeepCS	0.222	0.511	0.644	0.222	0.182	0.171	0.374

In the training process, we use MRR as an indicator of model convergence. We take the result that MRR is no longer improved in 30 consecutive epochs as the condition to stop training. Figure 7 shows the MRR of each epoch of validation set in the training process. We can observe that SQ-DeepCS achieves better MRR over nearly all epochs and stops training in the 144th epoch, 26.5% less than DeepCS (196th epoch). The results show that our model converges faster than DeepCS.

Figure 7. The MRR of each epoch in validation set.

Compared with DeepCS, our model needs to train fewer parameters. DeepCS has nearly 11 million parameters, while our model only needs to train 8 million, which is 27.3% lower than DeepCS. This is reflected in the training time of each epoch where DeepCS cost 140 s and we only need 100 s.

6. Discussions

We take a brief discussion about the performance of SQ-DeepCS and other code search models in Section 5.4. We propose a new representation method of code snippets named program slice. Compared with API sequence used in DeepCS and AST used in NCS, program slice contains more structural information of code snippets. Program slice extracts common features based on AST, which is more in line with the programming idea of code snippets and makes SQ-DeepCS perform better than At-CodeSM. The introduction of the attention mechanism enables different parts of the code snippets to obtain different weights. The attention mechanism weights the quality of code snippets and gives higher weight to the high-quality parts of code snippets.

Via comparison from Section 5.4, we find that limitations still exist in our research. Although the overall performance of our model is better than the existing methods, we find that in the results of some individual natural language queries, the accuracy of DeepCS, NCS, and At-CodeSM exceeds that of our model. In the experiment, we found that when the functions described by query statements are complex, the performance of the five

methods is not satisfactory. We speculate that the function of many Golang methods collected in GitHub in the dataset is parameter setting or intermediate variables updating. These kinds of code snippets are less reusable and will thus affect search performance.

7. Conclusions

In this paper, we propose a novel code search method named SQ-DeepCS. SQ-DeepCS introduces a code representation method called program slice to represent the structural information as well as API usage of code snippets. Meanwhile, SQ-DeepCS introduces a novel deep neural network named MD-JEnn to weight the quality of code snippets. We train the model and search for code snippets with respect to the top-rated questions from Stack Overflow. We use four evaluation indicators to measure the effectiveness of SQ-DeepCS: FRank, SuccessRate@k, PrecisionRate@k, and MRR. The experimental results show that our approach can provide better results than existing techniques when searching for relevant code snippets. Our potential future research may focus on extracting features of complex code snippets. Meanwhile, we also suggest studying why the proposed method performed worse than existing methods on some simple code snippets.

Author Contributions: Conceptualization, Y.Z. (Yin Zhang) and Y.Z. (Yuli Zhao); methodology, H.Y. and Y.Z. (Yin Zhang); software, H.Y.; validation H.Y., Y.Z. (Yin Zhang) and Y.Z. (Yuli Zhao); formal analysis, H.Y. and Y.Z. (Yin Zhang); investigation, B.Z.; resources, B.Z.; data curation, H.Y.; writing—original draft preparation, H.Y.; writing—review and editing, Y.Z. (Yin Zhang) and Y.Z. (Yuli Zhao); visualization, H.Y.; supervision, Y.Z. (Yin Zhang) and Y.Z. (Yuli Zhao), project administration, B.Z.; funding acquisition Y.Z. (Yuli Zhao) and B.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Key Project of National Natural Science Foundation of China (U1908212), the National Natural Science Foundation of China (Grant Nos. 61977014, 61902056), and the Fundamental Research Funds for the Central Universities (N2017016, N2017013, N2017014).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yan, S.; Yu, H.; Chen, Y.; Shen, B.; Jiang, L. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 344–354. [\[CrossRef\]](#)
2. Sachdev, S.; Li, H.; Luan, S.; Kim, S.; Sen, K.; Chandra, S. Retrieval on Source Code: A Neural Code Search. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Philadelphia, PA, USA, 18–22 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 31–41. [\[CrossRef\]](#)
3. Stolee, K.T.; Elbaum, S.; Dobos, D. Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.* **2014**, *23*, 1–45. [\[CrossRef\]](#)
4. Chen, Q.; Zhou, M. A neural framework for retrieval and summarization of source code. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 826–831.
5. Campbell, B.A.; Treude, C. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 628–632. [\[CrossRef\]](#)
6. Cambroneiro, J.; Li, H.; Kim, S.; Sen, K.; Chandra, S. When Deep Learning Met Code Search. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 964–974. [\[CrossRef\]](#)
7. Lv, F.; Zhang, H.; Lou, J.G.; Wang, S.; Zhang, D.; Zhao, J. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 260–270. [\[CrossRef\]](#)
8. Linstead, E.; Bajracharya, S.; Ngo, T.; Rigor, P.; Lopes, C.; Baldi, P. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.* **2009**, *18*, 300–336. [\[CrossRef\]](#)

9. Sivaraman, A.; Zhang, T.; Van den Broeck, G.; Kim, M. Active Inductive Logic Programming for Code Search. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 292–303. [\[CrossRef\]](#)
10. Ke, Y.; Stolee, K.T.; Goues, C.L.; Brun, Y. Repairing Programs with Semantic Code Search (T). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 295–306. [\[CrossRef\]](#)
11. Gu, X.; Zhang, H.; Kim, S. Deep Code Search. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944. [\[CrossRef\]](#)
12. Meng, Y. An Intelligent Code Search Approach Using Hybrid Encoders. *Wirel. Commun. Mob. Comput.* **2021**, *2021*, 9990988. [\[CrossRef\]](#)
13. Ai, L.; Huang, Z.; Li, W.; Zhou, Y.; Yu, Y. SENSORY: Leveraging Code Statement Sequence Information for Code Snippets Recommendation. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; Volume 1, pp. 27–36. [\[CrossRef\]](#)
14. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [\[CrossRef\]](#)
15. Balog, M.; Gaunt, A.L.; Brockschmidt, M.; Nowozin, S.; Tarlow, D. DeepCoder: Learning to Write Programs. *arXiv* **2016**, arXiv:1611.01989.
16. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to sequence learning with neural networks. *arXiv* **2014**, arXiv:1409.3215.
17. Tufano, M.; Pantiuchina, J.; Watson, C.; Bavota, G.; Poshvyanyk, D. On learning meaningful code changes via neural machine translation. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 25–36.
18. Shuai, J.; Xu, L.; Liu, C.; Yan, M.; Xia, X.; Lei, Y. Improving code search with co-attentive representation learning. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Korea, 13–15 July 2020; pp. 196–207.
19. Siami-Namini, S.; Tavakoli, N.; Namin, A.S. The performance of LSTM and BiLSTM in forecasting time series. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 3285–3292.
20. Zhou, Q.; Wu, H. NLP at IEST 2018: BiLSTM-attention and LSTM-attention via soft voting in emotion classification. In Proceedings of the 9th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis, Brussels, Belgium, 31 October 2018; pp. 189–194.
21. Xu, G.; Meng, Y.; Qiu, X.; Yu, Z.; Wu, X. Sentiment analysis of comment texts based on BiLSTM. *IEEE Access* **2019**, *7*, 51522–51532. [\[CrossRef\]](#)
22. Allamanis, M.; Peng, H.; Sutton, C. A Convolutional Attention Network for Extreme Summarization of Source Code. In Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016; Balcan, M.F., Weinberger, K.Q., Eds.; PMLR: New York, NY, USA, 2016; Volume 48, pp. 2091–2100.
23. Iyer, S.; Konstant, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Long Papers, Berlin, Germany, 7–12 August 2016; Volume 1, pp. 2073–2083.
24. Wan, Y.; Shu, J.; Sui, Y.; Xu, G.; Zhao, Z.; Wu, J.; Yu, P.S. Multi-modal attention network learning for semantic source code retrieval. *arXiv* **2019**, arXiv:1909.13516.
25. Kang, H.J.; Bissyandé, T.F.; Lo, D. Assessing the generalizability of code2vec token embeddings. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1–12.
26. Wang, G.; Li, C.; Wang, W.; Zhang, Y.; Shen, D.; Zhang, X.; Henao, R.; Carin, L. Joint Embedding of Words and Labels for Text Classification. *arXiv* **2018**, arXiv:1805.04174.
27. Karpathy, A.; Fei-Fei, L. Deep visual-semantic alignments for generating image descriptions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 3128–3137.
28. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API Learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 631–642. [\[CrossRef\]](#)
29. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 783–794.
30. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. *arXiv* **2017**, arXiv:1711.00740.
31. Keivanloo, I.; Rilling, J.; Zou, Y. Spotting working code examples. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 664–675.
32. Li, X.; Wang, Z.; Wang, Q.; Yan, S.; Xie, T.; Mei, H. Relationship-aware code search for JavaScript frameworks. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 690–701.

33. Ye, X.; Bunescu, R.; Liu, C. Learning to rank relevant files for bug reports using domain knowledge. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 689–699.
34. Raghothaman, M.; Wei, Y.; Hamadi, Y. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 357–367.
35. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [[CrossRef](#)]
36. Tai, K.S.; Socher, R.; Manning, C.D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv* **2015**, arXiv:1503.00075.