


## Article

# Joint Embedding of Semantic and Statistical Features for Effective Code Search

Xianglong Kong<sup>1,\*</sup> , Supeng Kong<sup>1</sup>, Ming Yu<sup>2</sup> and Chengjie Du<sup>1</sup><sup>1</sup> School of Computer Science and Engineering, Southeast University, Nanjing 211189, China<sup>2</sup> Shenyang Blower Works Group Corporation, Shenyang 110869, China

\* Correspondence: xlkong@seu.edu.cn

**Abstract:** Code search is an important approach to improve effectiveness and efficiency of software development. The current studies commonly search target code based on either semantic or statistical information in large datasets. Semantic and statistical information have hidden relationships between them since they describe code snippets from different perspectives. In this work, we propose a joint embedding model of semantic and statistical features to improve the effectiveness of code annotation. Then, we implement a code search engine, i.e., JessCS, based on the joint embedding model. We evaluate JessCS on more than 1 million lines of code snippets and corresponding descriptions. The experimental results show that JessCS performs more effective than UNIF-based approach, with at least 13% improvements on the studied metrics.

**Keywords:** code search; software reuse; code embedding; statistical semantics



**Citation:** Kong, X.; Kong, S.; Yu, M.; Du, C. Joint Embedding of Semantic and Statistical Features for Effective Code Search. *Appl. Sci.* **2022**, *12*, 10002. <https://doi.org/10.3390/app121910002>

Academic Editors: Rolando Miragaia and José Carlos Bregieiro Ribeiro

Received: 12 August 2022

Accepted: 28 September 2022

Published: 5 October 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

Code search is a process that takes a user's query as input and retrieves the most relevant, appropriate and rated source code snippets from a code base [1]. During software development, 19% of working time is spent on code search, the existing code snippets in open-source communities can be reused to implement current requirements [2]. An effective code search approach can significantly improve the efficiency of software development [3–5]. Existing code search research mainly focuses on query expansion and search engine optimization, which are constructed based on semantic and statistical features. Akbar et al. used semantics and ordering to retrieve source code [6]. Balachandran conducted instance queries from large-scale code repositories [7]. The effectiveness of semantic and statistical analysis can directly impact code search techniques.

However, existing code search techniques focus on either semantic or statistical model, regardless of the potential relationships between them. For example, API invocation and code frequency can represent a code snippet from different perspectives, but they are rarely used together in traditional code search approach. The existing works already prove that either statistical features [8] or semantic features [9–11] can be used to build a code search model. Actually, semantic and statistical features are symbiotic in software projects. Technical problems that arise when combining semantic and statistical features involves the mismatch between cooperative features and multidimensional information with various patterns of representation and different levels of granularity.

To address this problem, we apply joint natural language and source code snippet models to make code search more effective, so that a code search technique can be found, i.e., JessCS. During code semantic analysis, the necessary semantic information is extracted at the method level and the class level to implement the best fit semantic information. Next, statistical features are extracted, e.g., API invocation sequence, to investigate statistical regularity. During joint embedding, the code and its corresponding natural language description are mapped into a unified vector space. Semantically, similar concepts occupy

adjacent regions in this vector space. We train a deep learning model to match the natural language query and relate code based on the joint embedding network.

To evaluate the effectiveness of JessCS, we collected 1,263,974 sets of code and a set of corresponding functional descriptions from a public API, i.e., Google BigQuery (Google BigQuery, <https://cloud.google.com/bigquery/public-data/> (accessed on 1 June 2020)). The code base is conducted based on a project filter that limits the stars, language, time period, and other factors. Methods that do not contain Javadoc comments are filtered out. Our model uses <method name, API invocation, token collection> to represent the code snippets. Experimental results show that JessCS obtained higher accuracy and success rate on code search results. We selected UNIF-based approach [12] to make the comparison. JessCS obtains commonly at least 13% improvement in various indicators, and the query success rate of returning TOP-10 results reaches 0.90. In terms of code feature selection, selecting a triad for the code feature improves the TOP-1 Precision and MRR (Mean Rank Reciprocal) by 16.6% and 0.144, respectively, which effectively improves the accuracy of code search. To summarize, the main contributions of this study include:

- We propose semantic and statistical analysis methods for code search and combines deep learning to construct a joint embedding network.
- A code search technique for Java language, i.e., JessCS is designed and a dataset for code search based on 1,263,974 pairs of code snippets and descriptions was constructed.
- We conducted a comparative experiment to evaluate the effectiveness of JessCS and analyze the impact of related factors. We found that JessCS is more effective than the selected approach, and the improvements come from both of semantic and statistical features.

The rest of the paper is organized as follows. Section 2 presents the background techniques. Section 3 introduces the details of our approach. Section 4 introduces experiment and analysis. Section 5 discusses the possible threats to validity. Section 6 introduces the related concepts and techniques. Section 7 concludes this paper.

## 2. Background

In this section, we introduce the background concepts and techniques our approach is based on.

### 2.1. Code Search

The current studies applied many kinds of models to match the given query with the candidate code snippets [13]. For example, information retrieval models commonly match the code and query at the text level. Machine learning models mine the semantic information of the code from massive datasets and builds a code search model to achieve semantic matching between the code and natural language queries.

Text-level analysis in code search usually employs language models to mine deep statistical features of text [14]. The language model from the code search is used to predict the next word based on the previous text, so it can only handle sequences of fixed length. Aiming at NNLM problems, we use LSTM [15] in the application of code embedding to maintain long-term dependencies. LSTM introduces a gating mechanism to control when and how previous information is read from memory cells and how new information is written. The vector representation of words can be obtained by training the language model. In this paper, the attention mechanism is used to weigh and sum these word vectors to obtain the vector representation of the sequence.

### 2.2. Joint Embedding

Joint embedding [16], known as multi-modal embedding [17], is a technique that can encode heterogeneous data into a unified vector space. This technique is used for many tasks [18] (e.g., Krugle (Krugle code search, <http://www.krugle.com/> (accessed on 2 August 2020)), especially in the image processing. For example, in the field of computer vision, Karpathy and Li [19] used the joint embedding technique to jointly embed images

and text into the same vector space for image labeling. In the code search field, we need to treat the code as a structural text and vectorize it.

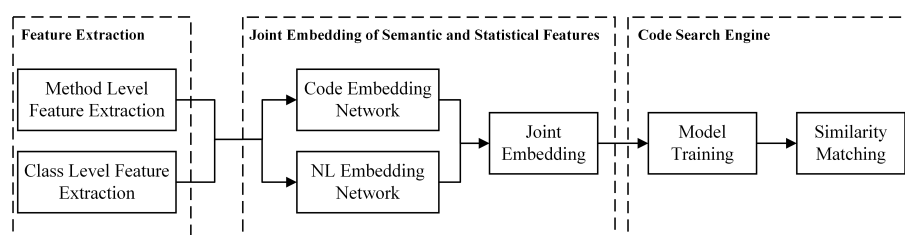
Vectorization of a text refers to the process of converting text into numerical tensors. In this paper, we decompose the text into a series of lexical tokens, and then use word embeddings and distributed representations to associate numerical vectors with these tokens. If two words have similar context, then even if the two words themselves are unrelated, they may carry similar semantic features. Since natural language queries and code fragments are heterogeneous, semantically similar codes and natural language queries may not have the same keywords or synonyms, and simply matching based on their keyword tags is prone to term mismatch. To remedy this deficiency, embeddings need to be generated separately for code and natural language descriptions.

Word embedding is an embedding technique that makes vectors of similar entities close to each other, also known as a distributed semantic model [20,21]. It represents words as fixed-length vectors so that similar words are close to each other in the vector space. Word embeddings are often implemented using machine learning models such as CBOW and Skip Gram [22]. These models build a neural network to capture the relationship between words and their contextual representation.

In this paper, we want to find the corresponding functional code snippet for the natural language query entered by the user. Considering the heterogeneity of code and natural language, we use a joint embedding technique to map the code and its corresponding language description into a unified vector space to achieve the purpose of code search. Due to many syntactic and structural features unique to the source code, we choose a collection of method names, API calls, and tokens to represent the code, and then combine them into a semantic vector representing the entire code entry.

### 3. Approach

We present the technical framework of JessCS in Figure 1. JessCS consists of three steps: feature extraction, joint embedding of semantic and statistical features, and construction of a code search engine. First, we extract information from source code and query descriptions to build the code embedding network and natural language embedding network. The query description, which includes word stemming, keyword synonyms and word frequency, is obtained by tokenizing the identifier according to the camel case naming rule. Second, we implement the joint embedding method based on semantic and statistical features to overcome the incongruence of data dimension. Finally, we construct a code search engine based on the matching strategy of the query and source code.



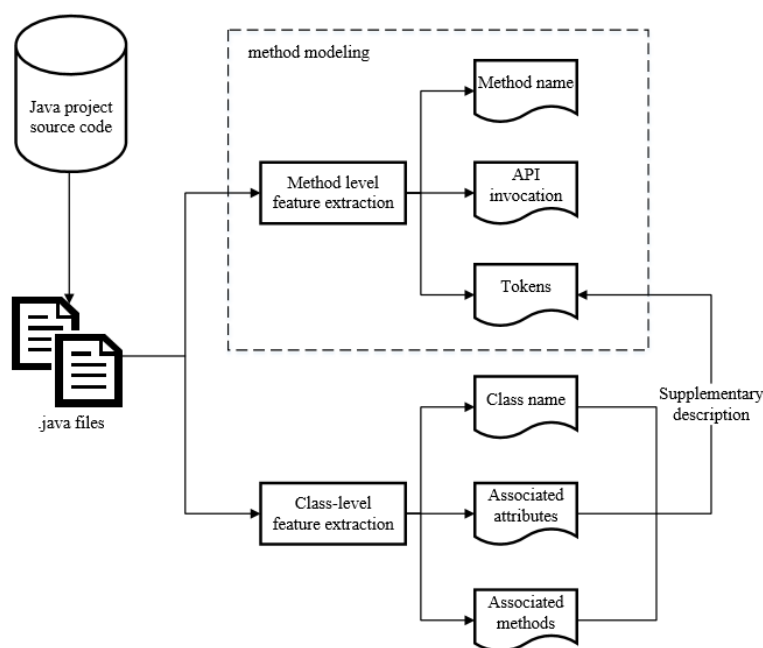
**Figure 1.** Feature extraction for source code.

#### 3.1. Feature Extraction with Different Granularities

To learn the deep statistical features of source code through deep learning methods [23], we employ semantic analysis and modeling on the source code. To extract the semantic information in the code as comprehensively as possible and eliminate the semantically irrelevant noise, this work proposes a source code feature extraction method.

Figure 2 demonstrates how the solution uses the method in the Java file as the basic unit and models the source code from the method level and the class level. For each method in the class, the necessary semantic information related to the appropriate method is extracted from the class where the method is located to supplement the description. In

Java, a method is the smallest unit of source code functionality implementation. The method level does not exist alone, it exists as part of a class level.



**Figure 2.** Feature extraction for Java source code.

### 3.1.1. Feature Extraction at the Method Level

For code search with method as the granularity as a parameter, the semantic information at the method level can be reflected in the method name, API invocation, string, variable name, method comment, and other information. This study uses the triplet,  $\langle \text{method name, API invocation, token collection} \rangle$  to model the method [9]. The specific information extraction method is as follows:

- **method name:** For each Java method, we extract its method name and parse it according to camel case. For example, the method name “ContentAddExecutor” will be resolved as “content, add, executor”.
- **API invocation:** A sequence of API invocations can represent the semantics of the source code [24]. We present the detailed methods of feature extraction through the following examples: (a) For each constructor invoking  $\text{new } C()$ ,  $C.\text{new}$  is generated and appended to the API sequence. (b) For each method invoking  $o.m()$ , where  $o$  is an instance of class  $C$ ,  $C.m$  is generated to append to the API sequence. (c) For method invocations passed as parameters, the method is attached before invoking the method. For example,  $o_1.m_1(o_2.m_2(), o_3.m_3())$ , we generate a sequence  $C_2.m_2-C_3.m_3-C_1.m_1$ , where  $C_i$  is the class to which the instance  $o_i$  belongs. (d) For the statement sequence  $s_1, s_2, \dots, s_N$ , the API sequence  $a_i$  is extracted from each  $s_i$  and concatenated to the API sequence  $a_1-a_2-\dots-a_N$ . (e) For conditional statements such as  $\text{if}(s_1)s_2;\text{elses}_3;$ , we create a sequence from all possible branches, i.e.,  $a_1-a_2-a_3$ , where  $a_i$  is the API sequence extracted from statement  $s_i$ . (f) For loop statements such as  $\text{while}(s_1)s_2$ , a sequence  $a_1-a_2$  is generated, where  $a_1$  and  $a_2$  are API sequences extracted from statements  $s_1$  and  $s_2$ .
- **token collection:** According to the hump rule, the various names in the entire method entity are split. The words, after filtering the stop words and keywords are added to the token collection. For example, stop words (e.g., *and* and *in*) and keywords in Java language are removed because they do not distinguish semantically in source code.

### 3.1.2. Feature Extraction at the Class Level

In Java, a method exists as part of a class. This work extracts class-level semantic information from three features: class name, association attribute, and association method. The class-level semantic information is used as a supplementary description of the token collection in the method-level semantic information. The specific methods are as follows:

- class names: The scope covered by some descriptive names (such as class names, method names, variable names) varies according to the objects they describe. The smaller the coverage, the more accurate the semantic information can be represented in the name. Thus, our scheme uses the class name as a supplement to the token collection in the method description information.
- Associated attributes and associated methods: A class contains several attributes and methods, in which class methods can directly access class attributes or invoke other class methods. To extract class-level semantic information for methods, it is necessary to dig deep-level sub-method calls and attribute access to analyze the functions of upper-level methods. Since the semantic information at the class level is less important than the semantic information at the method level, the upper layer and the bottom layer method name are parsed and added to the token collection of the middle layer. At the same time, the class name of the attribute accessed by the upper-level method is also added to the token collection.

The deep semantic features of the code are the ternary features generated after filtering the method name, API invocation, and token collection extracted from the method level features. Code searching based on statistical features extracts code semantics via statistical methods, and then matches them with the semantic features of the query language. Statistical features, which are Java code features extracted at the class level, include class names, associated attributes, and associated methods. In addition to considering the information of the method itself, the statistical features in the class where the method is located are a complementary description of the semantic information of the method.

### 3.2. Joint Embedding of Semantic and Statistical Features

A joint embedding model consists of three parts: code embedding, natural language description embedding, and similarity measurement. The code and natural language description embedding encodes the code and natural language description into vectors. The similarity calculation measures the similarity between the code vector and the natural language description vector. The core idea of the joint embedding is to represent code and natural language descriptions in the same vector space.

#### 3.2.1. Code Embedding Network

Based on source code feature extraction and modeling, we propose a deep learning model that represents code fragments as continuous distributed vectors. They represent code fragments as a fixed-length code vector, which can be used to predict the semantic properties of code fragments. Deep learning models can help mine the relationship between query descriptions and code. Compared with traditional text retrieval methods, deep learning-based code search using deep neural networks to learn code and natural language representations can obtain more effective results [12,25,26]. Through this network model, the deep statistical features of the code can be extracted and applied to the downstream code search task. The code embedding network is shown in Figure 3. According to the triplet of code <method name, API invocation, token collection> obtained by modeling, we generate embeddings for them, respectively, and integrate the embedding vectors of three different features into a code representation vector  $E_c$ .

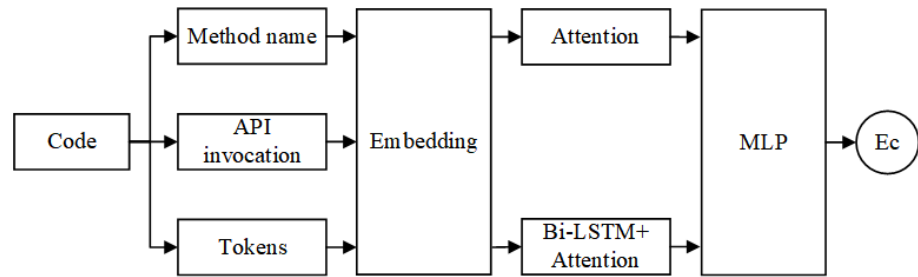


Figure 3. Code embedded in the network.

Method name embedding. We suppose the method name containing T words, which are represented as  $M = x_1, x_2, x_3, \dots, x_T$ . The specific scheme is as follows:

- Embedding layer: The role of this layer is to convert each word  $x_i$  into a corresponding vector representation  $e_i$ . A word vector matrix  $W_M \in \mathbb{R}^{d \times |V|}$  is defined, where  $V$  is a fixed-size vocabulary,  $d$  is the dimension of the word vector, and  $W_M$  is the parameter matrix obtained by training. Each entry of the word vector matrix (a word), can be converted into a representation of its word vector:  $e_i = W_M v^i$ , where  $v^i$  is a one-hot vector with dimension  $|V|$ . Then, the method named  $M$  is transformed into a matrix  $emb_M = e_1, e_2, \dots, e_T$  as the input of the next layer of the model.
- Attention layer: This work uses the attention mechanism to assign normalized weights to each word according to its weight to generate sentence vectors. For the output generated by the previous layer  $emb_M = e_1, e_2, \dots, e_T \in \mathbb{R}^{d \times T}$ , we use the following formula to calculate the normalized weight  $\alpha_i$  of each word vector  $e_i$ :

$$\alpha_i = \frac{\exp(a_m \cdot e_i^T)}{\sum_{i=1}^n \exp(a_m \cdot e_i^T)}, \tag{1}$$

where  $a_m$  is a trainable d-dimensional vector. After calculating the normalized weights, we weighted and summed each individual word vector to obtain the vector representation of the method name  $E_M$ :

$$E_M = \sum_{i=1}^T \alpha_i \cdot e_i. \tag{2}$$

API invocation embedding. The API invoked in one method are sequential nature. In this paper, the network structure of Bi-LSTM + attention is used to generate embeddings for API invocation sequences, which not only considers the weight relationship between different words, but also retains the position information of the sequence. The specific network structure is shown in Figure 4.

- Embedding layer: Like the embedding layer at the method name embedding, this layer converts the sequence of API invokes into a matrix of real numbers and passes it to the next layer of the model.
- Bi-LSTM layer: The word vector matrix input at the upper layer is converted into a hidden state matrix output. The LSTM structure of this layer includes an input gate, forget gate, cell state, and output gate:
  - Input gate. Used to decide how much new information to add:  $i_t = \sigma(W_{x_i}x_t + W_{h_i}h_{t-1} + W_{c_i}c_{t-1} + b_i)$ , where  $x_t$  represents the current input,  $h_{t-1}$  represents the previous hidden state, and  $c_{t-1}$  represents the previous cell state.
  - Forgotten door.Used to decide how much old information to discard:  $f_t = \sigma(W_{x_f}x_t + W_{h_f}h_{t-1} + W_{c_f}c_{t-1} + b_f)$ .
  - Cell state. It contains the last cell state and new information generated based on the current input and the last hidden state information:  $c_t = i_t u_t + f_t c_{t-1}$ ;  $u_t = \tanh(W_{x_c}x_t + W_{h_c}h_{t-1} + W_{c_c}c_{t-1} + b_c)$ .
  - Output gate. Used to decide which information is output:  $o_t = \sigma(W_{x_o}x_t + W_{h_o}h_{t-1} + W_{c_o}c_t + b_o)$ .

Finally, the current hidden state of the output is obtained by multiplying the current cell state by the weight of the output gate:

$$h_t = o_t \tanh(c_t). \tag{3}$$

- Attention layer: Denote the vector set of the output of the LSTM layer as  $H : [h_1, h_2, \dots, h_T]$ . We calculate the normalized weight of each hidden state  $h_i$  according to Formula (3). These individual hidden state vectors are then weighted and summed to obtain a vector representation of the API invocation.

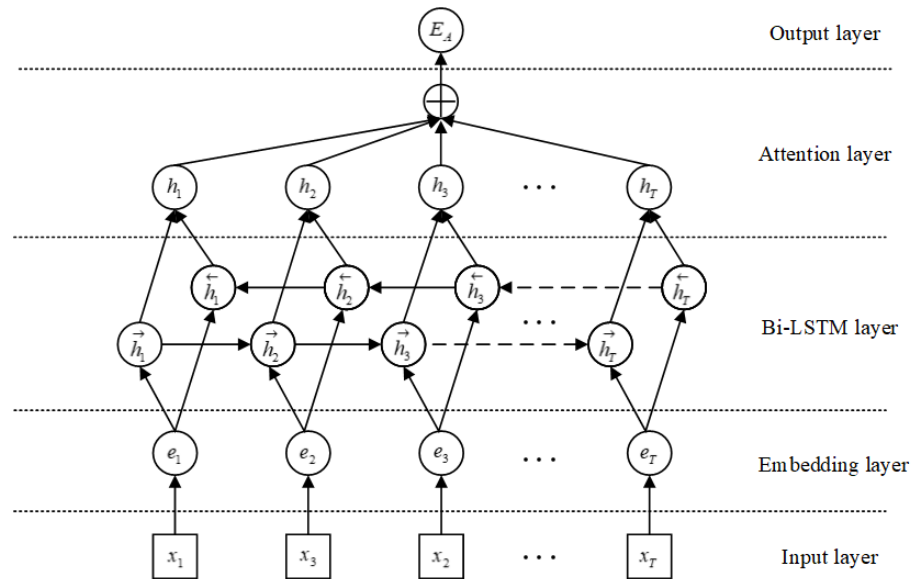


Figure 4. API invocation embedding.

Tokens collection embedding. The embedding method focuses on embedding layer and attention layer.

- Embedding layer: Like the embedding layer embedded in the method name, this layer converts the token collection into a real matrix and passes it to the next layer.
- Attention layer: Like the attention layer embedded in the method name, the weight of each word vector  $e_i$  is calculated according to Formula (3). These individual word vectors are then weighted and summed to obtain a vector representation of the set of tokens.

Feature fusion. The vectors of these three features are fused into a final code representation vector through the fully connected layer. The fully connected layer is a multi-layer perceptron, which can convert the feature vector into the output of the specified dimension.

The specific calculation formula for fusing the three code feature vectors is as follows:

$$E_c = \tanh(W_c[E_M, E_A, E_{TS}]), \tag{4}$$

where  $[E_M, E_A, E_{TS}]$  represents the simple concatenation of the vectors of these three features and  $W_c$  is the matrix of trainable parameters in the MLP. The output vector  $E_c$  represents the final embedding of the code segment.

In this section, we extracted three code features based on semantic analysis and modeling of the source code and we constructed a code embedding network based on multi-feature modeling, which is used to represent the code fragment as a fixed dimension that can reflect a code semantics vector.

### 3.2.2. Natural Language Embedding Network

Like code embedding networks, query embedding networks embed natural language descriptions into a vector space. Suppose a natural language description  $D = w_1, w_2, \dots, w_{N_D}$  is a sequence of  $N_D$  words. The specific embedded network is as follows:

- **Embedding layer:** Like the Code Embedding Network, the role of this layer is to convert each word  $w_i$  into a corresponding vector representation  $e_i$ . For each the word vector matrix  $W \in \mathbb{R}^{d \times |V|}$ , each word can be converted into a representation of its word vector:  $e_i = Wv^i$ . The natural language description named  $D$  is transformed into a matrix  $emb_D = e_1, e_2, \dots, e_{N_D}$  as the input of the next layer of the model.
- **Attention layer:** Like the Code Embedding Network, this study uses the attention mechanism to assign normalized weights to each word according to its importance to generate sentence vectors. For the output generated by the previous layer  $emb_D = e_1, e_2, \dots, e_{N_D}$ , we use the following formula to calculate the normalized weight  $\alpha_i$  of each word vector  $e_i$ :

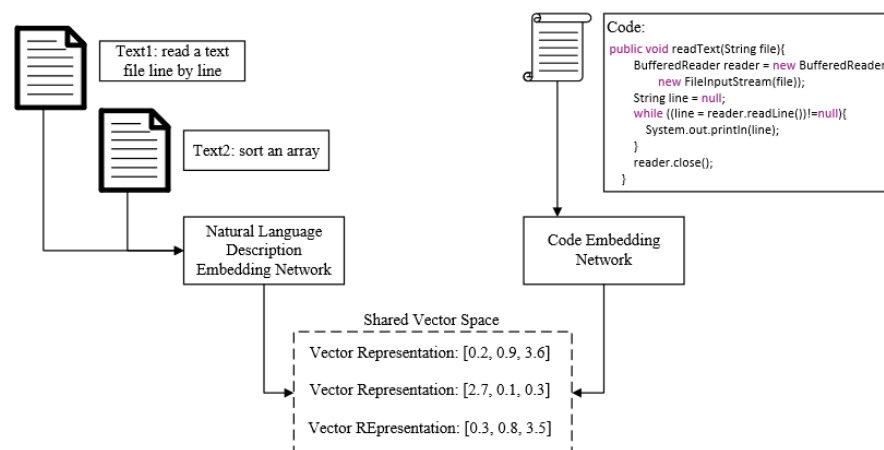
$$\alpha_i = \frac{\exp(a_d \cdot e_i^T)}{\sum_{i=1}^{N_D} \exp(a_d \cdot e_i^T)}, \tag{5}$$

where  $a_d$  is a trainable d-dimensional vector. Then, the weighted sum of these individual word vectors is used to obtain the final natural language description embedding vector  $E_D$ :

$$E_D = \sum_{i=1}^{N_D} \alpha_i \cdot e_i. \tag{6}$$

### 3.2.3. Joint Embedding

Figure 5 shows an example of a joint embedding between code snippet and natural language query. There are two natural language description texts (*Text1*: “read a text file line by line”; *Text2*: “sort an array”) and a code snippet (*Code*), which is a function that implements the function of reading text in a file line by line. After embedding *Text1*, *Text2*, and *Code* into the same vector space with the natural language description embedding network and code embedding network, respectively, *Text1* and *Code* with similar functions are closer in space. A supervised code search model [12] is selected as the baseline model of the search, and it is extended on this basis.



**Figure 5.** Schematic diagram of joint embedding of code and natural language query.

A joint embedding model consists of three parts: two different embedding functions  $\varphi : X \rightarrow \mathbb{R}^d$  and  $\psi : Y \rightarrow \mathbb{R}^d$ . These are combined to form the similarity measure function  $J(\cdot, \cdot)$ . They are connected through deep neural networks. The entire code search model consists of three modules, which correspond to the three components of the joint embedding



technology: code embedding, query embedding network, and similarity calculation. This is a general framework of the designed code search model based on supervised learning.

### 3.3. Code Search Engine

We combine the code embedding network, adjusted and optimized it, and proposed a new code search model JessCS. The specific mode structure is shown in Figure 6.

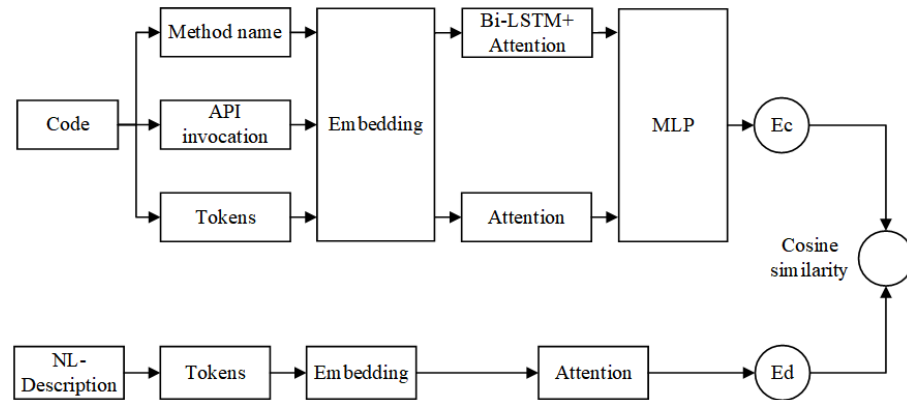


Figure 6. The framework of the code search engine.

JessCS divides the model into three sub-modules, which generate embeddings for code and natural language descriptions separately, and then use a cosine similarity function to measure the similarity between these two embedding vectors. After generating the embeddings for both the code and the natural language description, a similarity measure function is needed to measure the similarity between the two embedding vectors. This study uses the cosine function to measure the similarity between the two embedding vectors:

$$\cos(c, d) = \frac{c^T d}{\|c\| \|d\|}, \tag{7}$$

where  $c$  and  $d$  are the code vector and query vector, respectively. The higher the similarity, the more relevant the code is to the description.

Embedding both codes and descriptions into a unified vector space requires training a code search model. Figure 6 shows a schematic diagram of model training.

When training the model, we construct each training instance as a triplet  $\langle C, D+, D- \rangle$ . For each code fragment  $C$ , there is a positive description  $D+$  (the correct description of  $C$ ) and a negative description  $D-$  (the wrong description of  $C$ ) that is randomly selected from the pool of all  $D+$ . When trained on the triplet  $\langle C, D+, D- \rangle$ , the code search model predicts the cosine similarity between  $\langle C, D+ \rangle$  and  $\langle C, D- \rangle$  and minimizes the following loss function:

$$\mathcal{L}(\theta) = \sum_{\langle C, D+, D- \rangle \in P} \max(0, \epsilon - \cos(c, d+) + \cos(c, d-)), \tag{8}$$

where  $\theta$  represents the model parameter and  $P$  represents the training dataset.  $c$ ,  $d+$ , and  $d-$  are the embedding vectors of  $C$ ,  $D+$ , and  $D-$ . The  $\epsilon$  represents the significance level in the validation experiment. It is set to 0.05 in this work. This loss function drives up the cosine similarity between the code snippet and its correct description.

We use a large-scale corpus to train the model according to the above method, and continuously adjust and optimize the model parameters during the experiment to ensure the model achieves the best effect. Our models are built on Keras (Keras, <http://keras.io/> (accessed on 2 August 2020), an open-source deep learning framework. The hyperparameters we used in training are as follows: the number of hidden units in each direction of all LSTMs is set to 200; the dimension of the word embedding is set to 100; the dimension of

the output vectors of all attention layers is set to 100; the number of hidden units of feature fusion MLP is set to 100. The JessCS is trained with the mini-batch-Adam algorithm [2,27], the batch size (i.e., the number of instances per batch) is set to 128. We limit the vocabulary size to 10,000 words, which are the most used words in the training dataset. Through this training process, JessCS can embed code and natural language descriptions into a unified vector space.

JessCS focuses on comprehensive extraction of deep statistical features from source code and performing semantic analysis and modeling on source code. DeepCS [4] uses CODEnn model to represent code snippets with some directly available descriptions (e.g., high-quality comments or keywords). Due to the lack of deep investigation on code representation, DeepCS is outperformed by UNIF-based approach in recent study [12]. So we select the UNIF-based approach to make the comparison in our experiments.

#### 4. Experiment and Analysis

This section describes the conducted experiments and analysis, including evaluation of the effectiveness of JessCS and the analysis of the influencing factors.

##### 4.1. Research Questions

In this work, we aim to answer the following research questions:

- RQ1: Is JessCS effective?
- RQ2: How do influencing factors impact the effectiveness of JessCS?

We evaluate the effectiveness of JessCS in the answer of RQ1 from two perspectives: we firstly check the accuracy of code search model, then we conduct an evaluation on the searching results from 30 real code search attempts. We also investigate how the selection of code features and dimension of embedding vectors impact the effectiveness of JessCS in RQ2.

##### 4.2. Code Base

In the method-based code search, we obtain the *Javadoc* comments corresponding to each method by parsing the abstract syntax tree of the Java source code. Feature information is extracted as a natural language description of the method to obtain the large dataset. The steps needed to construct the training set are as follows:

- We use the SQL statement in Figure 7 for data fetching. This SQL statement is executed through Google BigQuery API. We obtained 63,015 Java projects with at least 2 stars in the GitHub database from 2016 to 2019, including 7,190,099 Java code files according to the conditions. The methods that do not contain *Javadoc* comments are filtered out in this step.
- For each annotated Java method, we extract semantic information, and use  $\langle \text{method name}, \text{API invocation}, \text{token collection} \rangle$  to represent the code snippets.
- For the *Javadoc* comment in each method, functional information is extracted as a natural language description. The specific methods used in this step are as follows:
  - (a) We delete the statements beginning with *@param* and *@return* in the comments, which introduce information about parameters and return values. These statements are not closely related to code functions;
  - (b) We delete the statements in the comments that contain the keywords *createdby* and *author*, which introduce the relevant information introduced by the creator and the author. These statements have nothing to do with the function of the code;
  - (c) We delete the comments led by *TODO* and *FIXME*, which have nothing to do with the function of the code;
  - (d) We delete the content containing URL and date in the comment, which has nothing to do with the function of the code;
  - (e) We delete the statement containing *Copyright/LICENSE* in the comment, which describes the copyright and other information and has nothing to do with the code function;

(f) We remove comments containing *@link* and *@see* that indicate relationships to other resources, not related to code functionality.

```

SELECT max(concat(f.repo_name, ' ', f.path)) as repo_path, c.content
FROM 'bigquery-public-data.github_repos.files' as f
JOIN 'bigquery-public-data.github_repos.contents' as c on f.id = c.id
JOIN (
  SELECT repo FROM(
    SELECT repo.name as repo
    FROM 'githubarchive.year.2016' WHERE type=WatchEvent''
    UNION ALL
    SELECT repo.name as repo
    FROM 'githubarchive.year.2017' WHERE type=WatchEvent''
    UNION ALL
    SELECT repo.name as repo
    FROM 'githubarchive.year.2018' WHERE type=WatchEvent''
    UNION ALL
    SELECT repo.name as repo
    FROM 'githubarchive.year.2019' WHERE type=WatchEvent''
  ) as r on f.repo_name = r.repo
WHERE
  f.path like '%.java' and c.size < 15,000
group by c.content

```

Figure 7. The SQL statement used in data extraction.

For the content-filtered annotations, the first sentence is selected as the natural language description of the method. Because the first sentence of the functional annotation is an overall description of the code function. Based on the above conditions for the source code, we conducted a dataset containing 1,263,974 sets of code snippets and corresponding functional descriptions. To ensure the consistency of the data, the experiments in this study are all carried out on this dataset.

#### 4.3. Metrics

The experiments select three indicators commonly used in the field of information retrieval to evaluate code search methods [28–32]:

##### 4.3.1. Success Rate of the First $N$ Results ( $SuccessRate@N$ )

For a query that may have multiple correct results, if the TOP- $N$  results [29] returned by the query contain at least one correct result, the query is deemed successful. The specific formula of  $SuccessRate@N$  is as follows:

$$SuccessRate@N = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \delta(FRank_q \leq N), \quad (9)$$

where  $Q$  is the set of queries,  $\delta()$  is a functional method, and  $FRank_q$  represents the ranking of the first correct query result for the query statement  $q$  [31]. The values of  $N$  in this experiment are 1, 5, and 10. From the TOP-10 search results, we mark the position of the first correct search result of each query statement as *Frank*. If there is no correct result in the query results of TOP-10, it will be recorded as *NF* (Not Found).

#### 4.3.2. The Accuracy of the First $N$ Results ( $Precision@N$ )

The evaluation index evaluates the proportion of correct search results in the top  $N$  returned search results for a query  $q$  [11]. The calculation formula is:

$$Precision@N = \frac{|relevant_{q,n}|}{N}, \quad (10)$$

where  $relevant_{q,N}$  represents the number of correct search results in the first  $N$  search results returned for the query statement  $q$ . The index is not sensitive enough to the location of the query result. In the standard calculation of TOP- $N$  metric, ranking the 1st and  $N$ th results has the same impact on the precision. However, developer mainly focus on the first result in the real development [33]. So we set  $N = 1$  in our experiments.

#### 4.3.3. Mean Rank Reciprocal (MRR)

The average sort reciprocal is suitable for situations where you only care about the position of the first correct search result [30,32]. The formula is as follows:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{FRank_q}, \quad (11)$$

where  $FRank_q$  represents the sorting position of the first correct query result for the query  $q$ .

We also checked the  $p$ -value and standard error to measure statistical features. During the measurement, the significance level is set to 0.05 according to the widely-used configurations [34–36]. The  $p$ -value represents the probability that the sample or observed value will occur under the conditions of the null hypothesis. If the  $p$ -value is less than the significance level, the null hypothesis can be rejected and a statistically significant conclusion can be drawn. The standard error estimates the variability across multiple samples of a population, which can also indicate the objectivity of our experiments.

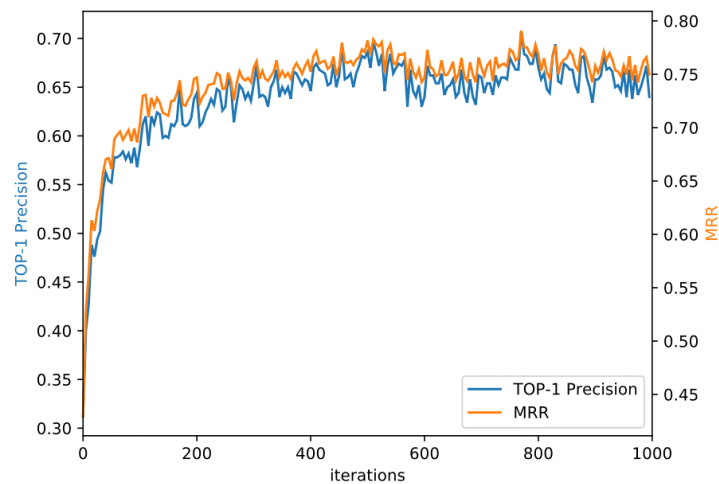
### 4.4. Results Analysis

#### 4.4.1. Rq 1: Effectiveness Evaluation of Code Search Approaches

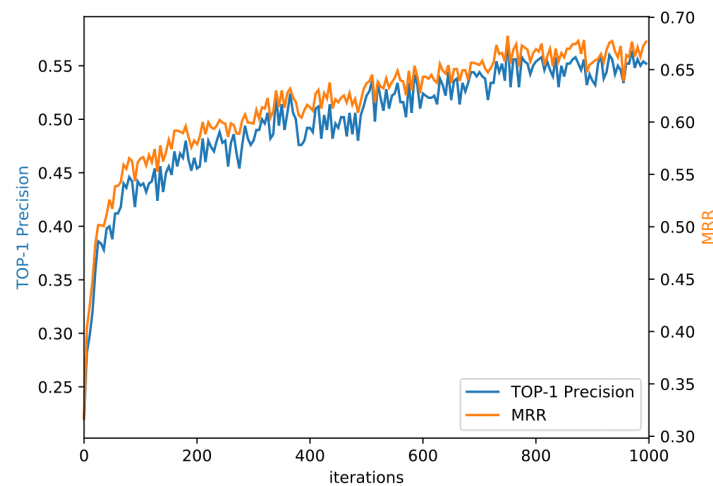
We verify the effectiveness of JessCS in terms of search model and search results. An automated evaluation method is used to evaluate the code search model on the validation set. During model training, we evaluate the model on 1000 validation sets every 5 iterations and record the TOP-1 precision and MRR values.

UNIF [12] is a supervised extension model based on NCS [26]. This work draws on the ideas in UNIF to build and implement its network structure for comparative experiments, which is referred to as UNIF-based here. The two techniques, UNIF-based and JessCS, were trained for 1000 epochs each. According to the TOP-1 precision and MRR value recorded in the model training, the curve of the TOP-1 precision and MRR value of the two models with the number of model iterations were drawn.

Figures 8 and 9 represent how the TOP-1 precision and MRR values change with the increase in iterations. According to the figures, the JessCS model converges when the iteration reaches about 500th round and tends to stabilize after a slight decrease. The UNIF-based model converges when the iteration reaches the 700th round, and then tends to be stable. In this study, the models with the best performance in the training process of these two different network structures are compared in Table 1. We measured the statistical indicators, i.e.,  $p$ -value and standard error, for the values of TOP-1 precision and MRR in Figures 8 and 9. The significance level is set to 0.05 in the measurement. The  $p$ -values for Figures 8 and 9 are 0.45 and 0.39, which are much greater than the significance level. The corresponding standard errors are 0.0432 and 0.0448. These statistical indicators show that the TOP-1 precision and MRR values are statistically determined. Both of the two measurements can indicate the effectiveness of the studied approaches.



**Figure 8.** The indicators of effectiveness for JessCS with the number of model iterations.



**Figure 9.** The indicators of effectiveness for UNIF-based approach with the number of iterations.

**Table 1.** Performance comparison of UNIF-based and JessCS.

	Iteration	TOP-1 Precision	MRR
JessCS	775	0.708	0.791
UNIF-based	755	0.578	0.682

As shown in Table 1, JessCS obtained the optimal model when iterated to 775 rounds, its TOP-1 precision was 0.708 and the MRR value was 0.791. The UNIF-based approach obtained the optimal model when iterating to 755 rounds, its TOP-1 precision was 0.578, and the MRR value was 0.682. The code search model proposed in this paper is 13% higher than UNIF-based approach in terms of the TOP-1 precision, and 0.109 higher than UNIF-based approach in terms of the MRR indicator.

We collected 30 query sentences from the high-frequency query set of the StackOverflow (StackOverflow, <https://stackoverflow.com/> (accessed on 3 September 2020)) and the commonly used high-frequency queries in the existing research [10,37,38]. Table 2 presents the collected query descriptions. We use JessCS and UNIF-based approach to code search for each query statement. We use Frank to indicate the effectiveness of code search approach in real attempts. The *NF* result means the approach cannot search a correct code snippet. The summary of search results is shown in Table 3:

**Table 2.** The collected query descriptions in real code search attempts.

Query ID	Query Description
1	create a folder
2	queue an event to be run on the thread
3	get current time and date
4	how to split string into words
5	read line by line
6	how to deserialize XML document
7	open a URL in Android's web browser
8	convert an input stream to a string
9	open url in html browser
10	copy paste data from clipboard
11	converting String to DateTime
12	how to delete all files and folders in a directory
13	how to generate random int number
14	how to execute a sql select
15	if a folder does not exist create it
16	how to get mac address
17	how to save image in png format
18	remove cookie
19	count how many times same string appears
20	how can I test if an array contains a certain value
21	append string to file
22	how can I convert a stack trace to a string
23	how to convert a char to a string in java
24	convert input stream to byte array in java
25	ping a hostname on the network
26	how to modify a char in string
27	how to read a large text file line by line using java
28	converting string to int in java
29	read an object from an xml file
30	get content of an input stream as a string using a specified character encoding

**Table 3.** The Frank values of the UNIF-based approach and JessCS search results.

Query ID	FRank		Query ID	FRank	
	JessCS	UNIF-Based		JessCS	UNIF-Based
1	1	1	16	1	1
2	1	3	17	4	3
3	5	1	18	1	1
4	1	1	19	NF	NF
5	9	NF	20	1	7
6	8	2	21	1	1
7	1	3	22	1	NF
8	1	1	23	1	6
9	1	2	24	2	1
10	NF	6	25	1	3
11	4	1	26	1	NF
12	1	2	27	3	2
13	1	1	28	1	1
14	NF	NF	29	1	1
15	2	1	30	1	NF

After evaluating the two code search models on 30 query statements, the query success rate and the average ranking reciprocal are calculated, as shown in Table 4:

**Table 4.** The accuracy of UNIF-based approach and JessCS results.

	SucRate@1	SucRate@5	SucRate@10	MRR
UNIF-based	0.43	0.70	0.80	0.56
JessCS	0.63	0.83	0.90	0.71
Improvements of JessCS	47%	19%	13%	27%

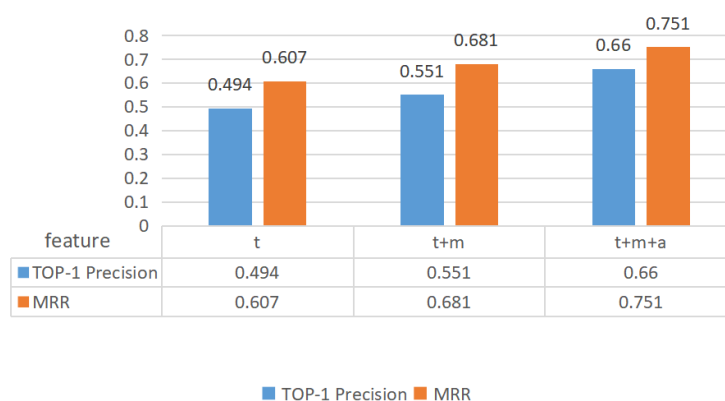
From the data in Table 4, we can find that JessCS performs more effective on all the indicators used in our experiments. For JessCS, the query success rate of returning TOP-10 results reaches 0.90 and 63% of the queries found the expected code snippet in the first search result. JessCS obtains 47%, 19%, 13%, and 27% improvements on SucRate@1, SucRate@5, SucRate@10, and MRR, respectively. The improvements come from the quality of *Javadoc* comments since we set a filter on the projects selection. *Javadoc* comments are widely used in the real projects, especially for the large-scale projects, so we can claim that Jess is effective.

**Finding 1.** JessCS performs more effective than UNIF-based approach in our experiments, with at least 13% improvements on the metrics.

#### 4.4.2. Rq 2: Analysis of the Impacts on JessCS Effectiveness from Influencing Factors

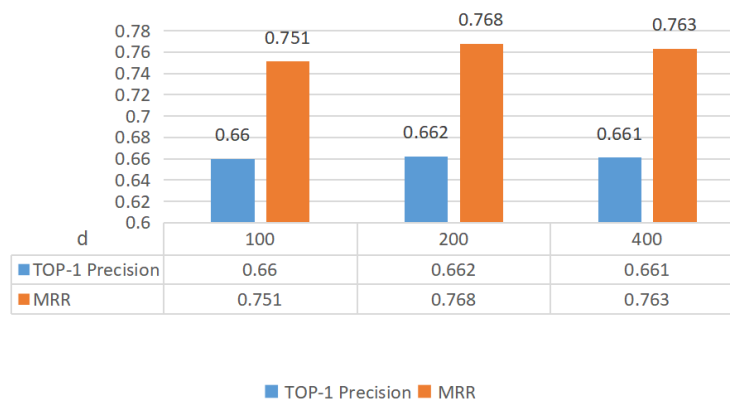
This part mainly analyzes the influencing factors of code search effectiveness from the selection of code features and embedding vector dimension. Based on the code search model in this study, only the selection of code features in the code embedding network is changed, and all other conditions are kept unchanged for experiments. We use  $\langle tokens \rangle$ ,  $\langle tokens + method name \rangle$ , and  $\langle tokens + method name + API invocation \rangle$  as code features to train the model, and train 300 rounds on the same dataset, and record the index values for comparison.

As shown in Figure 10, the horizontal axis feature represents different code feature selections,  $t$  represents the token set,  $m$  represents the method name, and  $a$  represents the API invocation. In terms of code feature selection, selecting  $t + m + a$  as the code feature improves the TOP-1 Precision and MRR indicators by 16.6% and 0.144, compared with  $t$  and  $t + m$ , which effectively improves the accuracy of code search.

**Figure 10.** The impacts of code features on the effectiveness of JessCS.

According to the code search model in this work, only the dimensions of the code embedding and natural language description embedding vector in the code embedding network are changed, and all other conditions are kept unchanged for the experiments. Three embedding vector dimensions (i.e., 100, 200, and 400) were set and each of them was trained for 300 rounds on the same dataset and model structure. The index values were recorded for comparison.

As shown in Figure 11, the horizontal axis  $d$  represents the dimension of the embedded vector of the code and its corresponding natural language description. For the code search model, changing the dimension of the embedding vector has a limited impact of JessCS effectiveness.



**Figure 11.** The impacts of embedding vector dimension on the effectiveness of JessCS.

**Finding 2.** *The joint embedding of method name and API invocation can significantly improve the effectiveness of JessCS. The use of joint embedding network can obtain more effective results than the single use of code tokens. The dimension of embedding vectors cannot impact JessCS significantly. The different dimensions (i.e., 100, 200, and 400) obtain similar results.*

## 5. Threats to Validity

**Threats to internal validity.** The construction of code base can pose a threat to internal validity. Although we extract projects from a well-maintained repository, i.e., Google BigQuery, the selected projects still cannot represent all the commonly-used search attempts. Additionally, we use the functional annotation of the code to approximate the natural language query of the code snippets. There is still a certain gap between the functional annotation and the real natural language query statement, but there are very few high-quality annotation (Codes, Natural Language Queries) datasets. The cost and difficulty of labeling such datasets manually is unacceptable for our team. To reduce this treat, we plan to extend the experiments on more subject projects.

The complex structure of neural architecture of JessCS model can also pose threats to internal validity. The multi-layer network may introduce cascading errors, which may result in over-fitting problems for our model. Additionally, we only investigate the impacts of two parameters, i.e., code features and embedding vector dimension, in the experiments. All the other parameters are set according to the existing work and our experience. These threats may impact the results of evaluation. We plan to design joint training models to check the effectiveness of joint embedding methods [39], and extend our experiments with various settings of parameters.

**Threats to external validity.** We selected UNIF-based approach [12] to make the experimental comparison, because it is proved to be more effective than some code search approaches (e.g., DeepCS [4]). However, there are still some other code search approaches, especially in software industry (e.g., Apache Lucene). To reduce this treat, we plan to extend the experiments on more techniques which can be used in code search.

The candidate code snippets are marked as independent functions, with the granularity of method level. Software developers may not always search for a method. So we plan to improve the embedding approach to meet the features from different levels of granularity for code snippets.



## 6. Related Work

Code search methods are widely used in both industry and academia. The open-source communities (e.g., *GitHub*, *SourceForge*) regard source code as text documents when implementing code search tasks and use information retrieval models to retrieve the snippet of code that matches the given query. These methods mainly rely on the matching text between the source code and the natural language query, while ignoring the semantic understanding of the natural language query and the source code. Code and natural language belong to two heterogeneous languages, and semantically relate code fragments and query sentences may not have common keywords, so this text-based matching method is prone to term mismatch, which leads to low search queries accuracy.

A huge body of research effort is dedicated to investigating effective and efficient code search techniques. Hindle et al. believe that although code and natural language are two heterogeneous languages, they have similar properties, and the code implementations of similar functions often have different degrees of similarity, so it is feasible to use statistical laws to analyze the code [40]. In the code search task, text mining is distributed massive heterogeneous code documents stored in code corpus. GuSites like StackOverflow are powerful production tools because they make it easy to search for code related to user questions expressed in natural language. Existing code search techniques can be divided into two categories: code search based on information retrieval and code search based on machine learning. The former treats code as text and uses information retrieval models to retrieve relevant code fragments that match a given query. The mainstream code search engines include Krugle, GrepCode, to name a few. This technique mainly relies on the textual similarity between source code and user query. The latter mainly uses statistical methods to mine the semantics of codes in massive datasets and builds code search models to further achieve semantic matching between codes and natural language queries. Combining information retrieval and supervised learning techniques, Niu [41] et al. used the RankBoost training set to train a model to learn ranking patterns based on the correlation between code snippets and their corresponding query sentences. Jiang et al. proposed ROSF [38]. The method is divided into two processes: coarse-grained search and fine-grained reordering.

In this work, unlike traditional code search techniques, JessCS achieves deep semantic understanding of source code and user queries combining deep learning techniques. We extract deep statistical features of source code through deep learning and perform semantic analysis and modeling on source code. In order to extract the semantic information in the code as comprehensively as possible, this work proposes an extraction method of cooperative features. Source code features can be extracted separately at different granularities. To solve the term mismatch problem in existing code search research caused by different languages of query sentences and code snippets [42], we also use a joint embedding method.

## 7. Conclusions

This work focuses on an appropriate usage of semantic and statistical features in code search. We propose an effective code search approach, i.e., JessCS, based on joint embedding network. JessCS combines deep learning techniques to construct a multi-feature model that extracts the deep statistical semantics. We evaluated the effectiveness of JessCS on a code base of more than 1 million lines of code. The experimental results show that JessCS performs more effective than another high-quality code search technique, i.e., UNIF-based approach. We also proved that the joint embedding of method name and API invocation can significantly improve the effectiveness of JessCS. In the future, we will extend the experiments on more subject projects and more code search approaches, and we also plan to design a joint training model to leverage the joint embedding methods.

**Author Contributions:** Conceptualization, X.K. and S.K.; methodology, X.K.; software, M.Y.; validation, X.K., C.D. and M.Y.; investigation, X.K.; resources, M.Y.; data curation, M.Y.; writing—original draft preparation, X.K.; writing—review and editing, S.K.; visualization, C.D.; supervision, X.K.; project administration, X.K. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

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

## References

1. Gabel, M.; Su, Z. A study of the uniqueness of source code. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, NM, USA, 7–11 November 2010; pp. 147–156.
2. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA, 7–9 May 2015; pp. 1–15.
3. Yu, H.; Zhang, Y.; Zhao, Y.; Zhang, B. Incorporating Code Structure and Quality in Deep Code Search. *Appl. Sci.* **2022**, *12*, 2051. [[CrossRef](#)]
4. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 40th International Conference on Software Engineering. ACM, Gothenburg, Sweden, 27 May–3 March 2018; pp. 933–944.
5. Mathew, G.; Stolee, K.T. Cross-language code search using static and dynamic analyses. In Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 205–217.
6. Akbar, S.; Kak, A. SCOR: Source Code Retrieval with Semantics and Order. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, Montreal, QC, Canada, 26–27 May 2019; pp. 1–12.
7. Balachandran, V. Query by example in large-scale code repositories. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Bremen, Germany, 29 September–1 October 2015; pp. 467–476.
8. David, A.; Larsen, K.G.; Legay, A.; Mikučionis, M.; Poulsen, D.B.; van Vliet, J.; Wang, Z. Stochastic semantics and statistical model checking for networks of priced timed automata. *arXiv* **2011**, arXiv:1106.3961.
9. Lemos, O.; Paula, A.; Zanichelli, S.; Lopes, C.V. Thesaurus-based automatic query expansion for interface-driven code search. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 212–221.
10. Rahman, M.M.; Roy, C.K.; Lo, D. RACK: Code Search in the IDE using Crowdsourced Knowledge. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, Buenos Aires, Argentina, 20–28 May 2017; pp. 51–54. [[CrossRef](#)]
11. Nie, L.; Jiang, H.; Ren, Z.; Sun, Z.; Li, X. Query Expansion Based on Crowd Knowledge for Code Search. *IEEE Trans. Serv. Comput.* **2016**, *9*, 771–783.
12. Cambroner, J.; Li, H.; Kim, S.; Sen, K.; Chandra, S. When deep learning met code search. In Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, 26–30 August 2019; pp. 964–974. [[CrossRef](#)]
13. Liu, C.; Xia, X.; Lo, D.; Gao, C.; Yang, X.; Grundy, J.C. Opportunities and Challenges in Code Search Tools. *ACM Comput. Surv.* **2022**, *54*, 1–40. [[CrossRef](#)]
14. Farahat, A.K.; Kamel, M.S. Statistical semantics for enhancing document clustering. *Knowl. Inf. Syst.* **2011**, *28*, 365–393. [[CrossRef](#)]
15. Palangi, H.; Li, D.; Shen, Y.; Gao, J.; He, X.; Chen, J.; Song, X.; Ward, R. Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval. *Audio Speech Lang. Process. IEEE/Acm Trans.* **2016**, *24*, 694–707.
16. Allamanis, M.; Tarlow, D.; Gordon, A.; Wei, Y. Bimodal Modelling of Source Code and Natural Language. In Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 7–9 July 2015; pp. 2123–2132.
17. Xu, R.; Xiong, C.; Chen, W.; Corso, J.J. Jointly Modeling Deep Video and Compositional Text to Bridge Vision and Language in a Unified Framework. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, TX, USA, 25–30 January 2015; pp. 2346–2352.
18. Weston, J.; Bengio, S.; Usunier, N.N. Wsabie: Scaling up to large vocabulary image annotation. In Proceedings of the International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, 16–22 July 2011; pp. 2764–2770. [[CrossRef](#)] [[PubMed](#)]
19. Karpathy, A.; Fei-Fei, L. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *2017*, 664–676.

20. Turian, J.; Ratinov, L.A.; Bengio, Y. Word Representations: A Simple and General Method for Semi-Supervised Learning. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, Uppsala, Sweden, 11–16 July 2010; pp. 384–394.
21. Frome, A.; Corrado, G.S.; Sens, J.; Bengio, S.; Dean, J.; Ranzato, M.A.; Mikolov, T. DeViSE: A Deep Visual-Semantic Embedding Model. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 1–9.
22. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space. In Proceedings of the 1st International Conference on Learning Representations, Scottsdale, AR, USA, 2–4 May 2013; pp. 1–12. [[CrossRef](#)]
23. Lee, S.; Lee, J.; Kang, S.; Ahn, J.; Cho, H. Code Edit Recommendation Using a Recurrent Neural Network. *Appl. Sci.* **2021**, *11*, 9286.
24. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API Learning. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 631–642.
25. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
26. 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, New York, NY, USA, 18 June 2018; pp. 31–41.
27. Mu, L.; Tong, Z.; Chen, Y.; Smola, A.J. Efficient mini-batch training for stochastic optimization. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014; pp. 661–670. [[CrossRef](#)]
28. Kong, X.; Han, W.; Liao, L.; Li, B. An analysis of correctness for API recommendation: Are the unmatched results useless? *Sci. China Inf. Sci.* **2020**, *63*, 190103.
29. 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.
30. Lv, F.; Zhang, H.; Lou, J.; Wang, S.; Zhang, D.; Zhao, J. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015; IEEE Computer Society: Washington, DC, USA, 2015; pp. 260–270.
31. 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.
32. Ye, X.; Bunesco, R.C.; 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. [[CrossRef](#)]
33. Peng, Y.; Li, S.; Gu, W.; Li, Y.; Wang, W.; Gao, C.; Lyu, M.R. Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We? *IEEE Trans. Softw. Eng.* **2021**. [[CrossRef](#)]
34. Kumar, P.; Singh, S.N.; Dawra, S. Software component reusability prediction using extra tree classifier and enhanced Harris hawks optimization algorithm. *Int. J. Syst. Assur. Eng. Manag.* **2022**, *13*, 892–903.
35. Barakaz, F.E.; Boutkhoum, O.; Moutaouakkil, A.E. *Feature Selection Method Based on Classification Performance Score and p-Value*; Springer: Cham, Switzerland, 2022. [[CrossRef](#)]
36. Diwaker, C.; Tomar, P.; Solanki, A.; Nayyar, A.; Jhanjhi, N.Z.; Abdullah, A.B.; Supramaniam, M. A New Model for Predicting Component-Based Software Reliability Using Soft Computing. *IEEE Access* **2019**, *7*, 147191–147203.
37. Campbell, B.A.; Treude, C. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Shanghai, China, 17–22 September 2017; pp. 628–632. [[CrossRef](#)]
38. Jiang, H.; Nie, L.; Sun, Z.; Ren, Z.; Kong, W.; Zhang, T.; Luo, X. ROSF: Leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets. *IEEE Trans. Serv. Comput.* **2019**, *12*, 34–46.
39. Luan, Y.; Wadden, D.; He, L.; Shah, A.; Ostendorf, M.; Hajishirzi, H. A general framework for information extraction using dynamic span graphs. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MI, USA, 2–7 June 2019; pp. 3036–3046. [[CrossRef](#)]
40. Hindle, A.; Barr, E.T.; Gabel, M.; Su, Z.; Devanbu, P. On the naturalness of software. *Commun. ACM* **2016**, *59*, 122–131. [[CrossRef](#)]
41. Niu, H.; Keivanloo, I.; Zou, Y. Learning to rank code examples for code search engines. *Empir. Softw. Eng.* **2017**, *22*, 259–291. [[CrossRef](#)]
42. Mcmillan, C.; Grechanik, M.; Poshyvanyk, D.; Fu, C.; Xie, Q. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Trans. Softw. Eng.* **2012**, *38*, 1069–1087.