electronics

MDPI

Article

Boosting Code Search with Structural Code Annotation

Xianglong Kong [1,*], Hongyu Chen [1], Ming Yu [2] and Lixiang Zhang [1]

[1] School of Computer Science and Engineering, Southeast University, Nanjing 211189, China
[2] Shenyang Blower Works Group Corporation, Shenyang 110869, Chin
* Correspondence: xlkong@seu.edu.cn

Abstract: Code search is a process that takes a given query as input and retrieves relevant code snippets from a code base. The relationship between query and code is commonly built on code annotation, which is extracted from code comments or other documents. The current code search studies approximately treat code annotation as a common natural language, regardless of its hidden structural information. To address the information loss, this work proposes a code annotation model to extract features from five perspectives, and further conduct a code search engine, i.e., CodeHunter. CodeHunter is evaluated on a dataset of 7 million code snippets and query descriptions. The experimental results show that CodeHunter obtains more effective results than Lucene and DeepCS. And we also prove that the effectiveness comes from the rich features and search models, CodeHunter can work well with different sizes of query descriptions.

Keywords: code search; code annotation; deep neural networks

Citation: Kong, X.; Chen, H.; Yu, M.; Zhang, L. Boosting Code Search with Structural Code Annotation. *Electronics* **2022**, *11*, 3053. https://doi.org/10.3390/electronics11193053

Academic Editors: Scott Uk-Jin Lee, Sanghyuk Lee, Soo Kyun Kim, Asad Abbas and Seokhun Kim

Received: 18 August 2022
Accepted: 21 September 2022
Published: 25 September 2022

1. Introduction

Code search is an important technique to improve effectiveness and efficiency of software development [1–3]. Code search is firstly conducted by matching keyword text in a code base, then researchers enrich the matching model with some descriptions about candidate code snippets, i.e., code annotation [4,5]. Code annotation may contain a class name, method name, API sequence, structural relationships among code blocks, and descriptions from code comments and documents. The commonly used way to extract code annotation is using the first sentence of annotations and words that are repeated within the code [4].

Although a lot of code search techniques are proposed in recent years, there are still some problems in this area [6,7]: (1) The existing code search technology is not comprehensive in the representation of code, most of them only pay attention to the information of the code itself and ignore the code annotation information; (2) In terms of code annotation information, the existing search technology simply regards all annotation information as equally weighted information, ignoring the specificity of code annotation; (3) The search content of the existing code search technology is simple, and there is no better strategy in the face of more complex search conditions [8,9].

To address these three problems, we propose CodeHunter, which uses both code and code annotation information to represent the code. Code annotation [10–13] can be treated as an irreplaceable part of the software, it is also a core component of APIs and frameworks for enterprise development. It plays a very important role in helping communicate key program capabilities in software development. Effectively leveraging code annotation information, or encouraging the use of code annotation information, is not currently considered in code search. We propose using an abstract syntax tree-based code analysis method to select and extract code features and annotation features; then, we fuse the two types of features to generate multi-dimensional code annotations. We set up and train deep annotation embedded networks and deep query embedded networks, vectorize multidimensional code annotations, and user query bar statements with their

corresponding network. By the end of the process, we calculate vector similarity measures to determine the best code that meets the search conditions.

We select the top 100 open-source Java projects on Github, ranked by stars. Based on 7 million lines of Java code, we conduct experiments to evaluate CodeHunter. Experimental results show that CodeHunter obtains higher accuracy and average reciprocal order than Lucene (https://lucene.apache.org/ (accessed on 13 May 2019)) and DeepCS [14]. In terms of A@1, A@5, and A@10, CodeHunter's accuracy rate was 0.43, 0.78, and 0.89, respectively, higher than the other two tools. CodeHunter ranked first among the three search tools with an MRR of 0.56. Although the effectiveness of CodeHunter is limited by the integrity of code features and annotation features, CodeHunter can still work well with different sizes of query descriptions.

In summary, the paper makes the following contributions:

- We propose a structural code annotation method, which realizes the application of code annotation and structure annotation in code search.
- We build and train deep-annotation and deep-query-embedded networks. We vectorize the code annotations and search conditions using their own respective networks and match the codes that meet the search conditions by calculating the similarity of vectors.
- CodeHunter obtains more effective results than Lucene and DeepCS in our experiments. We also find out the optimal settings of the training model and prove the effectiveness is not affected by the size of query descriptions.

The rest of this paper is organized as follows: The related work is in Section 2. The technical details of CodeHunter are described in Section 3. The experimental validation for our approach is shown in Section 4. The conclusion is outlined in the Section 5.

2. Related Work

A huge body of research effort has been dedicated to investigating effective and efficient code search techniques. The current works can be divided into two categories, i.e., information retrieval-based methods and machine learning-based methods according to the construction of the search model. This section will discuss these academic studies respectively.

2.1. Code Search Based On Information Retrieval

The code search technology based on information retrieval directly matches the similarity of two parts of information by extracting text information and code information. CodeHow [7] is a code search method based on an extended Boolean model. This method extends the representation of the code while it is already identified. This technique focuses on the description of the interface related to the code, combining the description with the identity of the code itself as the extended code representation. Finally, a vector model is used to vectorize the extended representation and query. The search results are obtained by measuring the distance between vectors. Vinayakarao et al. [8] also have similar methods. The difference is that they use Stack Overflow, a developer Q&A community, to gather information. They use code identifiers to search in the Q&A community and find relevant descriptions corresponding to code identifiers. This information is then used as an enhanced description of code identity for a more complete representation of the code. Finally, the accuracy of the search is improved.

The core of the above approach is an effective representation of code information. On the other hand, there are ways to augment the user's query information and make it easier to find satisfactory code by supplementing requirements. Lemo et al. [15] proposed a code search method based on a dictionary extended query. The innovation of this approach is that the focus shifts from code to queries. The method expands the query by searching for synonyms of keywords in the query. Finally, an expanded query is used to complete the search. Nie et al. [16] proposed a code search method that enhances the search text with question-and-answer information. This method combines the methods of

Vinayakarao [8] and Lemo [15]. Based on the search information of question-and-answer information, the query statement is extended. Although the expansion direction is different from Vinayakarao's method [8], they both have good effects. Lu et al. [17] proposed a code search method based on a lexical network. This method uses a vocabulary network to deal with queries, that is, through analysis of the query, the query vocabulary is extended with a vocabulary network, to better represent the demand. Finally, through similarity matching, the method outputs the final search code. Rahman et al. [18] proposed a tool that utilizes Q&A community data to conduct code search recommendations. This method also uses the question and answer community to find the corresponding description information through the code. Unlike the Nie method [16], the description information does not directly extend the query. Instead, it uses a Github search, a large open-source community, to directly use the description to find relevant code.

2.2. Code Search Based on Machine Learning

With the development of recognition intelligence tools, many researchers have applied some tools to code search. Haiduc et al. [19] designed queries based on machine learning. In this method, the corresponding relationship between query and code signing is studied, and finally, the query is automatically performed. Nguyen et al. [20] applied deep learning to API search and recommendation, that is, in each natural language query, the method generated an API-based ordered search. Later, Gu et al. [14] put forward DeepCS, which measures the similarity between code fragments and user queries through joint embedded learning and deep learning, and finds their connection through training. Instead of matching text similarity, DeepCS jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, in such a way that the code snippet and its corresponding description have similar vectors. Using the unified vector representation, code snippets related to a natural language query can be retrieved according to their vectors. According to the latest research results, the accuracy of code searches based on machine learning is higher than that based on information retrieval.

Chai et al. [21] proposed CDCS, a novel approach for domain-specific code search. CDCS's initial program representation model is pre-trained on a large corpus of common programming languages (such as Java and Python) and is further adapted to domain-specific languages such as Solidity and SQL. Unlike cross-language CodeBERT, which is directly fine-tuned in the target language, CDCS adapts a few-shot meta-learning algorithm called MAML to learn the good initialization of model parameters, which can be best reused in a domain-specific language. Liu et al. [22] using the advantages of DeepCS (i.e., the capability of understanding the sequential semantics in important query words) and the indexing technique in the IR-based model (accelerate the search response time substantially) proposed an IR-based model CodeMatcher. CodeMatcher first collects metadata for query words to identify irrelevant/noisy ones, then iteratively performs a fuzzy search with important query words on the codebase that is indexed by the Elasticsearch tool, and finally reranks a set of returned candidate code according to how the tokens in the candidate code snippet sequentially matched the important words in a query.

Although the code search technology has been well developed in recent years, there are still some problems such as misunderstanding of simple search statements, the quality of candidate code, and incomplete code representation. This work provides a structural code annotation method that uses two aspects of code and annotation information to represent the code, so that the code representation contains effective multi-dimensional information to improve the accuracy of search results.

3. Methodology

3.1. Motivating Example

As a centralized expression of code information, whether code annotations can completely show all the information within the code, directly affects the accuracy of code search. In previous studies, researchers annotated codes mostly with information from

the source code itself, such as Method Name or API sequence, including some identifiers in the Method body. For example, in recent deep neural network-based code technology DeepCS [14], the code feature that was extracted included: Method Name, API Sequence, and Tokens. Figure 1 shows a simple example of Tokens generated by a Java method in this technology.

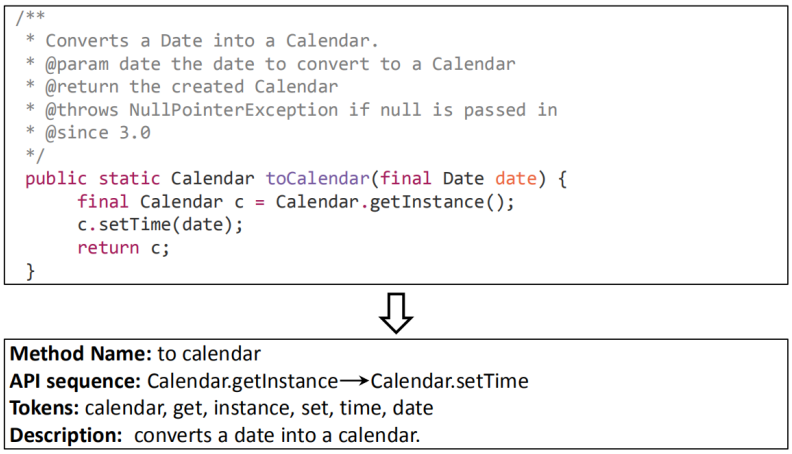

Figure 1. DeepCS code annotates the results.

Figure 1 demonstrates that although the code annotation collected contains relatively complete method body information, it did not collect annotation information. The annotation contains some important information such as interpretation of the parameters, interpretation of the return value, and code exception information. This information can help a coder interpret and understand the code. For example, *Converts a Date into a Calendar* expresses what the code logically does. *@param date the date to convert to a Calendar* and *@return the created Calendar* contain explanations of arguments and return values. *@throws NullPointerException if null is passed in* describes the exceptions that can occur and why. *@since 3.0* identifies the particular version that the class, method, or another identifier that was first added. In addition, this information may also be input by the user during the search (for example, some users may restrict the parameters and return values of the search code), so more comment information is needed for a more comprehensive code annotation.

To address this problem, the next section introduces a novel approach to code annotation. Figure 2 shows the overall framework of our method. As can be seen from the framework, the annotation method proposed in this paper not only uses the features of the code body extracted from the abstract syntax tree, but also extracts the features of the annotations, and finally completes the code annotation based on the two types of features.

3.2. Code Feature Extraction Method Based on an Abstract Syntax Tree

This section introduces a code feature extraction method based on an abstract syntax tree. Firstly, the concept and construction method of an abstract syntax tree are introduced. Then, the required code features are extracted according to the constructed abstract syntax tree. The extracted code features will be used to generate multidimensional code annotations later. Figure 3 shows the framework of the code annotation method.

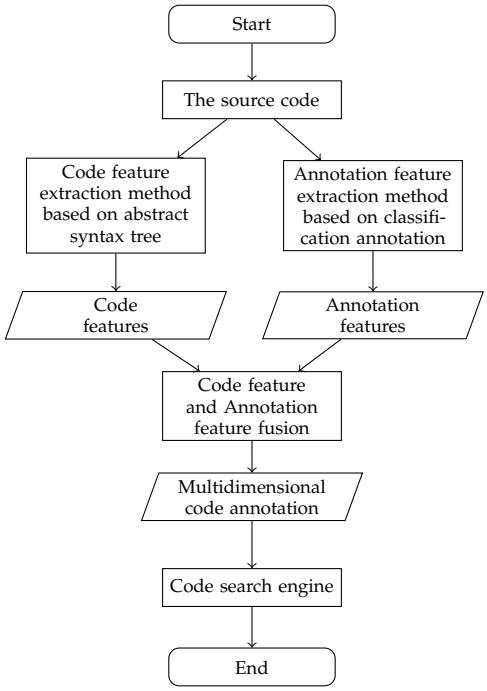

Figure 2. The overall framework of our approach.

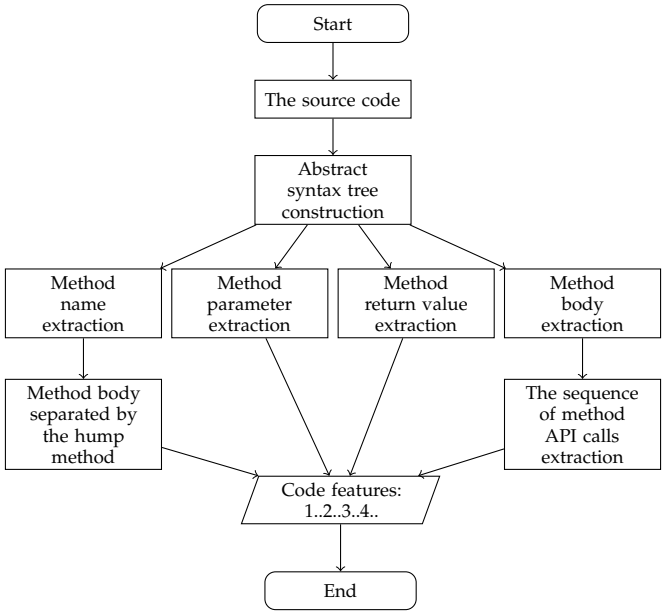

Figure 3. Code Annotation framework.

3.2.1. Program Analysis Based on Abstract Syntax Tree

Abstract syntax tree (AST) or "syntax tree" is an intermediate representation of the grammatical structure of the source code. Compared with the common code information in search technology, the code information obtained by abstract syntax trees is more comprehensive. There are many mature tools for converting source code into abstract syntax trees. This is commonly used by, for example, JDT, ANTLR, RECODER, and JavaCC. In this paper, JDT (i.e., Eclipse Java Development Tools) is selected for the construction of the abstract syntax tree of the source code. JDT is a component in Eclipse, which organizes, compiles, debugs, and runs Java programs. As Eclipse has evolved, JDT has gained more and more capabilities, including the construction of abstract syntax trees. Today, the Eclipse AST is an important component of the Eclipse JDT, defined in the package org.eclipse.jdt.core.dom, which allows researchers and developers to quickly generate abstract syntax trees.

3.2.2. Code Feature Extraction

After building the abstract syntax tree of the source code, the information on the abstract syntax tree nodes is obtained through access functions in the org.eclipse.jdt.core.dom.ASTVisitor class. As the object of this search uses Java methods, the upper node information, such as files and packages, is ignored during the access, and only the information in the org.eclipse.jdt.core.dom.ASTNode class is considered. Once the method level, method name, method parameter, method return value, and API sequence are extracted as code features, method annotations are extracted to generate annotation features in the next section.

The details of extracting the required information from the AST are described below.

- Method name. It is extracted by the getName() method in the MethodDeclaration class. After the method description is obtained, the words are separated in the method name according to the Java-based hump rule.
- Method parameter. It is extracted by the parameters() method in the MethodDeclaration class.
- Return value. It is extracted by the getReturnType() method in the MethodDeclaration class.
- Method annotation. It is extracted by the getJavadoc() method in the MethodDeclaration class.
- API sequence. The AST does not provide an API sequence interface for a method, so the getBody() method in the MethodDeclaration class is first called to extract the method body, and then the API sequence is extracted according to the extraction rules. The specific extraction rules are as follows: When a method call is used as a parameter, the parameter method is added to the API sequence first, and then the main method is added to the API sequence. For sequential statements, extract the APIs used in each statement and add the extracted APIs to the API sequence in the order of the statements. For conditional branch statements, extract the APIs used in each statement in $if - then - else$ order, and add the extracted APIs to the API sequence. For loop statements, add the API in the loop body to the API sequence only once. According to the above rules, each statement in the method body is processed to generate the API sequence for that method.

The detailed algorithm of extracting code features via an AST tree is shown as Algorithm 1. The code features of the source code are extracted via extraction Algorithm 1 and include method name, method parameters, method return value, and API sequence of four features.

Algorithm 1: Code feature extraction.

Input : Java source code address
Output: Code characteristics of Java methods
Begin : Define codeFeature {*name*, *para*, *return*, *APIsequence*}
Begin : Initial List(*codeFeature*)
Begin : BulidAST(*address*) as ast
for *method* ∈ *ast.methods* **do**
    name ←getName().split() ;
    para ←parameters();
    return ←getReturnType() ;
    body ←getBody() ;
    for *method* ∈ *body* **do**
        APIsequence.add (*method.getName*()) ;
    end
    List.add(codeFeature{*name*, *para*, *return*, *APIsequence*});
end
return List(*codeFeature*)

3.3. Annotation Feature Extraction Method

This section introduces an annotation feature extraction method. First, a content-based annotation classification and recognition method are proposed. Then, the annotation features of the source code are extracted according to the classified annotations, and the extracted annotation features will be used for the generation of code annotations later. Figure 4 shows the overall flow chart.

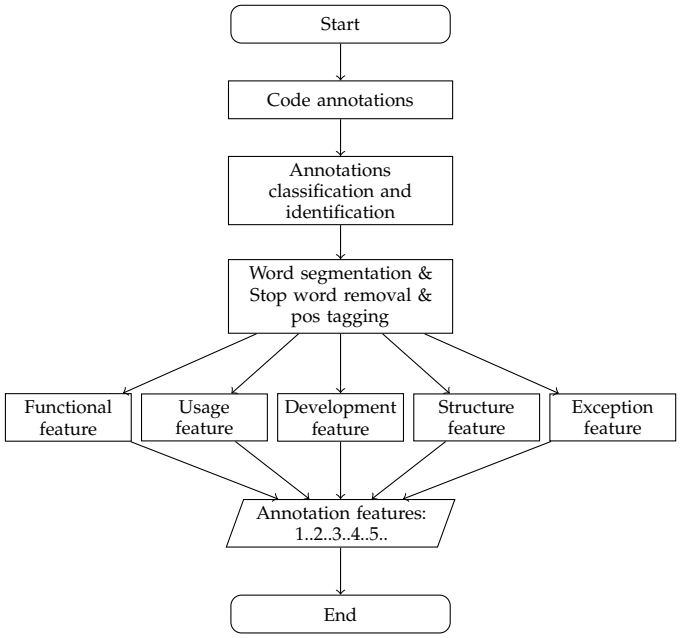

Figure 4. Code Annotation flowchart.

3.3.1. Classification of Annotation

Code annotations play an important role in software development. They can help developers understand the source code effectively; they also play a very important role in software maintenance [23]. Reading and understanding annotations is an essential activity in a developer's daily tasks, and in practice, developers typically spend more time reading and understanding code than editing it [24]. There have been studies on the analysis and classification of annotation contents, and they have classified the content in annotations according to their respective research purposes [25–28]. However, in the field of code search, the annotation of these different contents has not been considered sufficiently by researchers, and most of the research work on code annotation is focused on the function description of code [29,30]. Annotations describing functionality are indeed an important part of annotations, but the content of annotations may also include many other types of annotations [25]. Different types of annotations provide a more complete description of code when building code annotations or code indexes.

This study proposes a content-based annotation classification, which is used for code annotation generation and later code search-related work. This classification divides annotations into five categories, namely functional annotations, usage annotations, structure annotations, development annotations, and exception annotations. The details of each category are as follows:

- Functional annotations: This type of annotation contains a short description of the source code being referenced. Typically, developers use such annotations as code summaries to help other developers understand the behavior of the code without having to read the code to analyze it themselves.
- Use annotations: This type of annotation provides information about the use of the code to users who may want to use the code. Typically, annotations are preceded by metadata tags, such as @param or @return.

- Structural annotations: This type of annotation describes the relationship between this code entity and other code entities. Common tags are @see and @link.
- Development annotations: This type of annotation contains development-related information, including developer information, date, and version. Common tags are @author and @version.
- Exception annotations: This type of annotation describes the exceptions that can occur and why. The common labels are @throws and @Exception.

3.3.2. Annotation Recognition

After introducing the annotation classification, we propose an annotation recognition method to identify different types of annotations. Because the annotations extracted from the abstract syntax tree are method-level Javadoc annotations, the object of this annotation recognition method is Javadoc annotation specific. In addition, the granularity of this method is at the statement level, that is, every natural statement will be classified into the five types of annotations proposed above.

Because Javadoc annotations have their own annotation specifications, they are identified according to the key identifiers specified in the annotation specifications as follows:

- If the sentence in the annotation contains "@param" or "@return," then the sentence is annotated.
- Sentences in annotations that contain "@link" or "@see" indicate their relationship to other resources, which are structural annotations.
- If the sentence in the annotation contains "@author" and "@date", then the annotation describes the author and ownership; If the sentences in the annotation include "@version", and "@since", they describe information about the version. These sentences are development annotations.
- If the sentences in the annotation contain "@throws" or "@Exception", the exceptions that may be thrown by this method are displayed. These sentences are exception annotations.
- In practice, functional annotations are usually at the beginning of the annotation section. If the first or first sentences in an annotation do not contain the standard identifiers, identified in the bulleted items above, then they are functional annotations.

3.3.3. Annotation Feature Extraction

After identifying the annotation types, annotations are presented as different categories. This section will extract annotation features of annotation feature generation methods from annotations of different categories. Because annotations are from Natural Language text, it is necessary to use some Natural Language processing tools to preprocess the five types of annotations. Thus, NLTK (Natural Language Toolkit) is used for processing. NLTK is primarily English oriented, but many of its Natural Language Processing (NLP) models or modules are Language independent, so many of NLTK's toolkits can be reused if a Language has initial Tokenization or word segmentation.

The process of NLTK preprocessing annotation can be divided into three steps: word segmentation, stop word removal, and pos tagging. After preprocessing, each type of annotation has been processed into a word bag with part-of-speech tagging. Now, extraction rules are formulated according to different types of annotations to extract the word features of each type of annotation, as shown below:

- For functional annotations, verbs, and non-proper nouns are extracted as features of functional annotations. Verbs include VB, VBD, VBG, VBN, VBP, VBZ, and nouns include NN and NNS.
- For usage annotation, extract non-proper nouns as usage annotation features, including NN, NNS;
- For development notes, nouns (including proper nouns), cardinal words, and loanwords are extracted as development notes features, including NN, NNS, NNP, CD, FW;
- For exception annotations, extract non-proper nouns as features of exception annotations, including NN and NNS.

A detailed annotation feature extraction algorithm is shown in Algorithm 2. Specifically, each line of code annotation is spanned, and annotations are classified according to a keyword. Then, NLTK is used to preprocess annotations, and word features are extracted according to different types of annotations. Finally, all kinds of extracted word features are integrated to complete the extraction of code annotations.

Algorithm 2: Annotation feature extraction algorithm.

Input : Java method code annotations
Output: Method annotation feature
Begin : Define annotationFeature
$\{function, usage, structure, development, exception\}$;
Begin : $ImportNLTK$;
for $method \in ast.methods$ **do**
  if $\text{“@}param\text{”or“@}return\text{”} \in sentence$ **then**
    $wordssentence.tokenize()$;
    $words.drop(stepwords)$;
    $usagewords.pos\_tag().selece(NN, NNS)$;
  end
  else if $\text{“@}link\text{”or“@}see\text{”} \in sentence$ **then**
    $wordssentence.tokenize()$;
    $words.drop(stepwords)$;
    $structurewords.pos\_tag().selece(NN, NNS)$;
  end
  else if $\text{“@}author\text{”or“@}date\text{”or“@}version\text{”or“@}since\text{”} \in sentence$ **then**
    $wordssentence.tokenize()$;
    $words.drop(stepwords)$;
    $developmentwords.pos\_tag().selece(NN, NNS, NNP, CD, FW)$;
  end
  else if $\text{“@}throws\text{”or“@}exception\text{”} \in sentence$ **then**
    $wordssentence.tokenize()$;
    $words.drop(stepwords)$;
    $exceptionwords.pos\_tag().selece(NN, NNS)$;
  end
  else if $annotationsbeginwithsentence$ **then**
    $wordssentence.tokenize()$;
    $words.drop(stepwords)$;
    $functionwords.pos\_tag().selece(VB, VBD, VBG, VBN, VBP, VBZ, NN, NNS)$;
  end
  annotationFeature.add($function, usage, structure, development, exception$)
end
return annotationFeature;

3.4. Multi-Dimensional Code Annotation Generation Based on Code Feature and Annotation Feature

Multidimensional code annotation describes code from four aspects, including function annotation, usage annotation, method body annotation, and development annotation. The content of multidimensional code annotation is composed of code features and annotation features. The specific content is shown in Table 1.

Both annotation features and code features exist in the form of word features, and the repeated words will be filtered out in the fusion process.

Table 1. Multidimensional code annotates content.

Function Annotation	Usage Annotation	Method Annotation	Development Annotation
Method names (Code features) Functional features (Annotation features)	Parameters (Code features) Return value (Code features) Usage features (Code features)	Call sequence (Code features) Exception features (Annotation features) Structural features (Annotation features)	Development features (Annotation features)

3.5. Code Search Based on Multi-Dimensional Code Annotation

In the search framework, the multi-dimensional code annotation of the source code is generated based on the original code. After finding the code mark, by matching the network code annotation and the query statement of the user input, including the code marked by deep annotations, embedded networks will be entered by the user query line statement through deep natural language query embedded networks. This is done to quantify, then calculate vector lengths and find vector similarity measures. Finally, the ranking results based on similarity will output the search results.

4. Results and Discussion

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

- RQ1: Can CodeHunter generate accurate results for the given query descriptions?
- RQ2: Does CodeHunter perform more effective than other selected techniques?
- RQ3: How do the influencing factors impact the effectiveness of CodeHunter?

The experiment environment is as follows: 64-bit Windows-10 operating system, the host CPU is Intel core i7-8700k, the GPU is Nvidia K40 G, the Java version is Java8, the running environment is JDK1.8 with the Eclipse IDE, the Python version is 3.4 with the Pycharm IDE, and the database is MySQL.

4.1. Datasets

4.1.1. Collection of Experimental Data

GitHub is a large-scale open-source software hosting platform, and many developers and development teams have stored a large number of open-source projects and version history for users to call, including a large number of excellent java projects and codes; thus, this experiment also extracted these projects and codes as experimental data. First of all, the items are screened. In this experiment, the top 100 items in Github, whose programming language is Java, are sorted by star rating. After downloading the project, a query-code pair is generated according to the project. The method body of the code is extracted through an abstract syntax tree, and then the code annotations are extracted. There are two cases, according to whether the code has annotations or not:

(1) The method code contains code annotations. For this kind of situation, the abstract syntax tree of the code is constructed, and the method body and code annotations are extracted. After that, the code annotations are simply processed, the "@" tags in the code annotations are removed, and the remaining statements are combined as code queries.
(2) The method code contains no code annotations. In real-world environments, some methods lack code annotations. For this kind of situation, the abstract syntax tree is still constructed to extract the method body; then, the method name is segmented according to java naming rules and the segmentation result is taken as the query statement of the method.

With this methodology, a total of 7,337,263 query-code pairs were generated from 100 projects.

4.1.2. Division of Experimental Data

After collecting query-code pair data, 7,337,263 query-code pairs were collected. To train the matching network of tools and measure the index of search, this experiment divides the data into a training set and a testing set according to the ratio of 20–80%. As for the training set, the input model is a ternary combination of code-positive description-negative description, so it is necessary to modify the query-code pair. The query corresponding to the code itself is taken as the positive description, and 10 positive descriptions of the 10th, 20th, 30th, . . . , 100th method code after the method code are selected as their own negative descriptions.

Once the search database is finally generated, there are 7,337,726 total methods (including queries). The data for training CodeHunter to match the network is 5,869,810 (80% of the total number of methods). There are 58,698,100 < code, positive-description, negative-description > triples composed of query-code pairs. There are 1,468,453 queries for measuring indicators (20% of the total number of methods).

4.2. Baselines

(1) Lucene is an open-source java full-text search engine. It has a complete query engine and index engine, and some text word segmentation engines. Lucene aims to provide a simple and easy-to-use toolkit for software developers, which facilitates the full-text retrieval function in the target system or helps to build a complete full-text retrieval engine based on it. Lucene is a subproject of Apache, which can be downloaded and used directly.

(2) DeepCS [14] is an effective code search tool in academia, which does not depend on information retrieval technology. It measures the similarity between code fragments and user queries through the collection of code information and deep learning. According to the latest research results, the code search accuracy of DeepCS is higher than that of the search method using code information for information retrieval. This tool provides an open-source download address on Github, which can be downloaded and used directly.

4.3. Metrics

This section introduces two indicators to measure search results in this experiment.

(1) Accuracy (accuracy is abbreviated as A)

In this experiment, the accuracy rate refers to the ratio of correct search results appearing among the top k results [31], and its calculation formula is as shown in Equation (1):

$$Accuracy@K = \frac{1}{S}\sum_{s=1}^{S} f(s < k) \quad (1)$$

where S refers to the total number of searches and $f()$ is a judgment function. The value is 1 when the correct search result is in the first K of the list, or 0 otherwise. In the experiment, the value of K is 1, 5, 10, and the corresponding preparation rate is expressed as A@1, A@5, and A@10. Finally, the higher the accuracy value, the better the search performance.

(2) Mean Reciprocal Rank (*MRR*)

In this experiment, the recommended results are presented in a sorted list, and another metric for evaluating a search in the search and recommendation domain is the position of the correct results in the list of results. In the measurement of accuracy, as long as the result appears, whether the first position or the last position has the same influence on the result; however, the reality is that the evaluation of the search tool is likely related to the ranking position. Therefore, it is necessary to introduce a weighted factor of position into the evaluation system. The *MRR* method is mainly used in Navigational Search or Question Answering [32,33], which is insensitive to recall rate and pays more attention to

whether the correct document retrieved by the Search tool is ranked at the top of the list of results. The calculation formula is

$$MRR = \frac{1}{S}\sum_{s=1}^{S}\frac{1}{N} \tag{2}$$

where, S refers to the total number of searches, and N is the position of the correct result in the list. For example, if the correct result is the first, N is 1. In this experiment, the maximum number of result lists is 10. If no correct answer appears in the result list, N is 11. Thus, the larger the *MRR* value, the better the search performance.

4.4. Experimental Process

For each studied subject, we performed the following steps:

- Construct multi-dimensional code annotations of all experimental data through CodeHunter;
- Use the code of the training set to mark and query the training CodeHunter matching network to determine the weight of the matching network;
- The method in the data set of this experiment is used as the common search code set of Lucene, DeepCS, and CodeHunter;
- Input the test set query into Lucene, DeepCS, and CodeHunter. For CodeHunter, the similarity of all code annotations is calculated via the matching network. The search results of the top ten similar queries are retained and output, and then the top ten search results of the three are counted;
- Calculate metrics A@1, A@5, A@10, and *MRR* based on the correct and actual results of the query.

4.5. RQ1: Effectiveness of CodeHunter Search Results

After the output of the results, the distribution of correct results in the CodeHunter search results is shown in Figure 5.

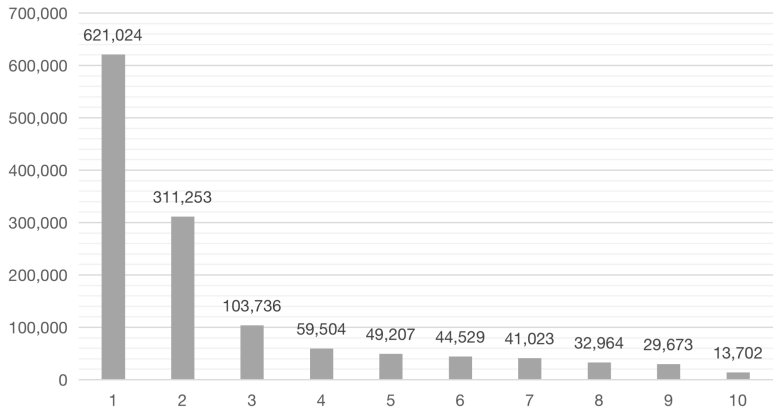

Figure 5. CodeHunter distribution of correct result locations.

As can be seen from Figure 5, in CodeHunter's test queries, there are 621 K queries with correct search results in the first recommended position, 1145 K queries with correct search results in the top 5 recommended position, 1306 K queries with correct search results, and 162 K queries with no correct search results. Based on the search results, the values of the metrics are shown in Table 2.

As can be seen from Table 2, the average probability of correct results appearing in the first search result is 0.43. The average probability of correct results in the first five results is 0.78, that is, the average probability of correct results in the second to fifth place is 0.35; The average probability of a correct result appearing in the top 10 search results is 0.89, which means the average probability of a correct result appearing in the 6th to 10th places is 0.12.

MRR has a draw value of 0.56, which means that the average correct search result ranks 1.8 in the list.

Table 2. The evaluation of CodeHunter search results.

A@1	A@5	A@10	MRR
0.43	0.78	0.89	0.56

4.6. RQ2: The Comparison of the Studied Code Search Techniques

Figure 6 shows the distribution comparison of correct results among the search results of the three tools.

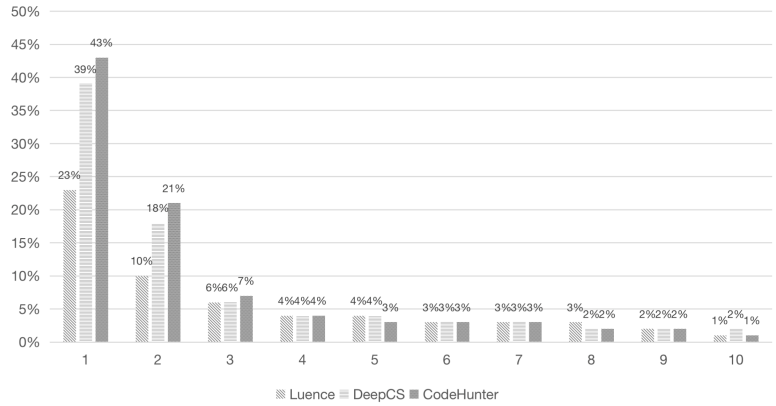

Figure 6. The distribution of Lucene, DeepCS, CodeHunter correct result.

As you can see from Figure 6, CodeHunter is significantly better than the other two tools for correct results in the first and second place. Search results are output in the form of ranking, and users get the answers they want at the forefront of ranking, which will significantly improve user experience and more accurately measure the index results as shown in Table 3.

Table 3. The evaluation of Lucene, DeepCS and CodeHunter search results.

	Luence	DeepCS	CodeHunter
A@1	0.23	**0.49**	0.43
A@5	0.47	0.72	**0.78**
A@10	0.59	0.84	**0.89**
MRR	0.32	0.52	**0.56**

The metrics show that CodeHunter's search performance is significantly better than Lucene's, and slightly better than DeepCS's. As a tool for searching using deep neural networks, DeepCS only considers the information of the code body and does not consider the code annotations when forming code annotations, and DeepCS only considers the simple functional statements when setting query statements. CodeHunter not only includes the code information but also considers the code annotation information when generating code annotations. In addition, in the setting of queries, CodeHunter's query statements can be multiple natural statements containing all aspects of code information. As a result, the actual CodeHunter search performed better when the test set of queries varied in length according to the developer's requirements. It can also be seen that CodeHunter is superior in actual query results in the face of more complex query contents.

4.7. RQ3: Analysis of Influencing Factors of CodeHunter

4.7.1. Training Set Size

To evaluate the influence of the size of training data on CodeHunter, 40%, 50%, 60%, 70%, 80%, and 90% data sets were used to construct training sets respectively, and 10% of the remaining data sets were extracted as test sets to analyze the influence. Table 4 shows the accuracy of training data of different sizes.

Table 4. The effectiveness of CodeHunter with different sizes of the training set.

	A@1	A@5	A@10
40%	0.35	0.63	0.71
50%	0.36	0.64	0.73
60%	0.39	0.71	0.79
70%	0.41	0.76	0.84
80%	0.43	0.77	0.88
90%	0.43	0.76	0.82

Table 5 demonstrates that the accuracy rate is the highest when the training set is about 80% of the data set, which is also the reason why the training test set is divided into 20–80% in this study.

4.7.2. User Query Complexity

In the test query, some query statements are short, while some query statements are long and contain more content. To analyze the impact of user query complexity on CodeHunter's accuracy, query statements are divided according to length. Thus, the query statements with larger lengths are more complex, and vice versa. In the test set, query statements are divided into six groups of 1–5, 6–10, 11–15, 16–20, 21–25, and 26+ according to the number of query statements. The accuracy results are shown in Table 5.

Table 5. The effectiveness of CodeHunter with different complexity of query descriptions.

	A@1	A@5	A@10
1–5	0.38	0.67	0.79
6–10	0.41	0.73	0.85
11–15	0.43	0.79	0.90
16–20	0.44	0.79	0.89
21–25	0.42	0.75	0.87
26+	0.41	0.74	0.86

Table 5 shows the search accuracy will be slightly reduced when the user query is short with too few words. However, in general, CodeHunter is not sensitive to the change in query statement complexity and can better complete the search task for query statements of different lengths.

5. Conclusions

This work proposes a code search technique, i.e., CodeHunter to improve the effectiveness of search results. Different from existing methods, CodeHunter constructs code features from both code and code annotation information and uses deep neural networks to match code annotations with query statements, and finally obtains search results. The code annotation model is conducted based on the information from five perspectives, i.e., functionality, usage, structure, development, and exception handling. CodeHunter is evaluated on more than 7 million code snippets and the experimental results show that it is more effective than other code search techniques, i.e., DeepCS and Luence. CodeHunter obtains an average 5% improvement over other selected techniques in terms of *Accuracy@K* and *MRR* values. we

also find that the improvements come from the high quality of code and annotation features. And CodeHunter cannot be impacted by the various sizes of query descriptions.

In the future, we plan to continuously optimize the matching network and feature selection to improve the accuracy of the search. There is also some space for improvements on applicability, e.g., extending the model to other programming languages or different granularity of code snippets. The related experiments will be extended to more subject projects and more code search techniques.

Author Contributions: Conceptualization, X.K. and H.C.; methodology, X.K.; software, M.Y.; validation, X.K., L.Z. and M.Y.; investigation, X.K.; resources, M.Y.; data curation, M.Y.; writing—original draft preparation, X.K.; writing—review and editing, H.C.; visualization, L.Z.; 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.

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

References

1. Liang, L.; Li, Y.; Wen, M.; Liu, Y. KG4Py: A toolkit for generating Python knowledge graph and code semantic search. *Connect. Sci.* **2022**, *34*, 1384–1400. [CrossRef]
2. Xie, Y.; Shibata, K.; Mizoguchi, T. A brute-force code searching for cell of non-identical displacement for CSL grain boundaries and interfaces. *Comput. Phys. Commun.* **2022**, *273*, 108260. [CrossRef]
3. Brandt, J.; Guo, P.J.; Lewenstein, J.; Dontcheva, M.; Klemmer, S.R. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In Proceedings of the 27th International Conference on Human Factors in Computing Systems, Boston, MA, USA, 4–9 April 2009; pp. 1589–1598.
4. Gkonis, P.K.; Trakadas, P.T.; Sarakis, L.E. Non-orthogonal multiple access in multiuser MIMO configurations via code reuse and principal component analysis. *Electronics* **2020**, *9*, 1330. [CrossRef]
5. Yu, H.; Zhang, Y.; Zhao, Y.; Zhang, B. Incorporating Code Structure and Quality in Deep Code Search. *Appl. Sci.* **2022**, *12*, 2051. [CrossRef]
6. Chatterjee, S.; Juvekar, S.; Sen, K. *SNIFF: A Search Engine for Java Using Free-Form Queries*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2009; pp. 385–400.
7. 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, Lincoln, NE, USA, 9–13 November 2015; pp. 260–270.
8. Vinayakarao, V. Spotting familiar code snippet structures for program comprehension. In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 31 August–4 September 2015; pp. 1054–1056.
9. Reiss, S.P. Semantics-based code search. In Proceedings of the International Conference on Software Engineering, Edmonton, AB, Canada, 20–26 September 2009; pp. 243–253.
10. Pinheiro, P.; Viana, J.C.; Fernandes, L.; Ribeiro, M.; Ferrari, F.; Fonseca, B.; Gheyi, R. Mutation Operators for Code Annotations. In Proceedings of the Brazilian Symposium on Systematic and Automated Software Testing, Sao Carlos, Brazil, 17–21 September 2018; pp. 77–86.
11. Lima, P.; Guerra, E.; Nardes, M.; Mocci, A.; Bavota, G.; Lanza, M. An Annotation-Based API for Supporting Runtime Code Annotation Reading. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, Vancouver, BC, Canada, 23–27 October 2017; pp. 6–14.
12. Schramme, M.; Macías, J.A. Analysis and measurement of internal usability metrics through code annotations. *Softw. Qual. J.* **2019**, *27*, 1505–1530. [CrossRef]
13. Pinheiro, P.; Viana, J.C.; Ribeiro, M.; Fernandes, L.; Ferrari, F.C.; Gheyi, R.; Fonseca, B. Mutating code annotations: An empirical evaluation on Java and C# programs. *Sci. Comput. Program.* **2020**, *191*, 102418.
14. Gu, X.; Zhang, H.; Kim, S. Deep Code Search. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944.
15. Lemos, O.; Paula, A.; Zanichelli, S.; Lopes, C.V. Thesaurus-Based Automatic Query Expansion for Interface-Driven Code Search. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad India, 31 May–1 June 2014; pp. 212–221.
16. Nie, L.; He, J.; Ren, Z.; Sun, Z.; Li, X. Query Expansion Based on Crowd Knowledge for Code Search. *IEEE Trans. Serv. Comput.* **2017**, *9*, 771–783. [CrossRef]
17. Lu, M.; Sun, X.; Wang, S.; Lo, D.; Duan, Y. Query expansion via WordNet for effective code search. In Proceedings of the 22nd IEEE International Conference on Software Analysis, Montreal, QC, Canada, 2–6 March 2015; pp. 545–549.

18. 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.
19. Haiduc, S.; Bavota, G.; Marcus, A.; Oliveto, R.; Lucia, A.D.; Menzies, T. Automatic query reformulations for text retrieval in software engineering. In Proceedings of the 35th IEEE/ACM International Conference on Software Engineering, Silicon Valley, CA, USA, 11–15 November 2013; pp. 842–851.
20. Nguyen, T.T.; Pham, H.V.; Vu, P.M.; Nguyen, T.T. Learning API usages from bytecode: A statistical approach. In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016; Dillon, L.K., Visser, W., Williams, L.A., Eds.; IEEE: Piscataway, NJ, USA, 2016; pp. 416–427.
21. Chai, Y.; Zhang, H.; Shen, B.; Gu, X. Cross-Domain Deep Code Search with Meta Learning. In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering, Pittsburgh, PA, USA, 22–24 May 2022; pp. 487–498.
22. Liu, C.; Xia, X.; Lo, D.; Liu, Z.; Hassan, A.E.; Li, S. CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 12:1–12:37. [CrossRef]
23. Maalej, W.; Robillard, M.P. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1264–1282. [CrossRef]
24. Marcus, A.; Maletic, J.I.; Sergeyev, A. Recovery of Traceability Links between Software Documentation and Source Code. *Int. J. Softw. Eng. Knowl. Eng.* **2005**, *15*, 811–836. [CrossRef]
25. Pascarella, L.; Bruntink, M.; Bacchelli, A. Classifying code comments in Java software systems. *Empir. Softw. Eng.* **2019**, *24*, 1499–1537. [CrossRef]
26. Padioleau, Y.; Tan, L.; Zhou, Y. Listening to programmers — Taxonomies and characteristics of comments in operating system code. In Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA, 16–24 May 2009; pp. 331–341.
27. Steidl, D.; Hummel, B.; Jürgens, E. Quality analysis of source code comments. In Proceedings of the 21st International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013; pp. 83–92.
28. Subramanian, S.; Inozemtseva, L.; Holmes, R. Live API Documentation. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 643–652.
29. Hammad, M.; Babur, Ö.; Basit, H.A.; van den Brand, M. Clone-Seeker: Effective Code Clone Search Using Annotations. *IEEE Access* **2022**, *10*, 11696–11713. [CrossRef]
30. Yao, Z.; Peddamail, J.R.; Sun, H. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In Proceedings of the The World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 2203–2214.
31. 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. [CrossRef]
32. Li Xuan, W.Q.; Zhi, J. Code search method based on enhanced description. *J. Softw.* **2017**, 1–11.
33. Ye, X.; Bunescu, 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.