Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 9956207, 11 pages
https://doi.org/10.1155/2021/9956207

Research Article

TagNN: A Code Tag Generation Technology for Resource Retrieval from Open-Source Big Data

Lingbin Zeng,[1] Xin Guo,[1] Cheng Yang,[2] Yao Lu,[2] and Xiao Li[1]

[1]Technical Service Center for Vocational Education, National University of Defense Technology, Changsha 410073, China
[2]College of Computer, National University of Defense Technology, Changsha 410073, China

Correspondence should be addressed to Xiao Li; xiaoli@nudt.edu.cn

Received 4 June 2021; Accepted 2 August 2021; Published 26 August 2021

Academic Editor: Mamoun Alazab

With the vigorous development of open-source software, a huge number of open-source projects and open-source codes have been accumulated in open-source big data, which contains a wealth of code resources. However, effectively and efficiently retrieving the relevant code snippets in such a large amount of open-source big data is an extremely difficult problem. There are usually large gaps between the user's natural language description and the open-source code snippets. In this paper, we propose a novel code tag generation and code retrieval approach named TagNN, which combines software engineering empirical knowledge and a deep learning algorithm. The experimental results show that our method has good effects on code tag generation and code snippet retrieval.

1. Introduction

With the vigorous development of the open-source software, the resources contained in the open-source big data are becoming increasingly abundant [1–3], including not only open-source software artifacts but also software development behavior data and auxiliary documentation such as user manuals and technique reports. On the one hand, the rapid development of the open-source big data has provided software developers with a huge amount of software code snippets. On the other hand, the explosive growth of open-source resources has brought considerable challenges to the retrieval of open-source resources, especially code snippets. Part of the current research work has been aimed at the classification of open-source resources. For example, Stack Overflow (http://stackoverflow.com/) classifies query questions into different levels with different tags. These classification levels and tag settings are determined by specialized domain experts who spend considerable time and effort. Such a manual process is difficult to scale to the explosive development of the open-source big data [4]. Take the famous Linux kernel as an example. According to statistics (http://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/), the Linux kernel has 27.8 million lines of code in the Git repository in 2020, increasing more than 1.7 million lines compared with 2019. Such a phenomenon is common in the open-source community. This brings great challenges to human annotation work.

Open-source code snippets are developing at such a rapid pace that traditional manual tagging is far from adequate. Thus, the automatic tagging and classification of open-source resources have begun to attract scholarly attention. Wang et al. [5] use the existing tag of an open-source community (e.g., Stack Overflow) to tag software using classification algorithms such as SVM. Zhou [6] puts forward a tool named *TagMulRec* which contained an efficient tag-based multiclassification algorithm that could deal with a mass of software. These tasks are aimed at constructing tags at the open-source software or project level.

However, with the development of the open-source big data, the reuse of open-source resources by software developers has dived into the level of specific source codes rather than just using general mature open-source software products. Consequently, generating tags for natural language queries is an effective method to enable software developers to quickly retrieve high-quality open-source code snippets.

The traditional manual tagging model has difficulty in supporting the generation of code tags for massive natural language query statements. In recent years, the rapid development of deep learning methods has many mature applications, such as image recognition, speech recognition, and language translation. This also brings great opportunities for code tag generation [7, 8]. Unlike traditional machine learning models, training deep learning models require massive amounts of data. At this point, the open-source big data has massive resources for model training. The rapid development of deep learning in natural language processing has a certain enlightening effect on the tag generation of open-source resources.

Based on the above survey, we propose a technology to automatically generate code tags for natural language queries, in the hope of better guiding software developers to effectively retrieve code snippets, accelerating the application of open-source resources by software developers, and promoting the growth and expansion of open-source software. Specifically, we propose a novel code tag generation and code retrieval approach named *TagNN*. The model combines the term frequency-inverse document frequency (TF-IDF) algorithm and the recurrent neural network (RNN) framework. Our experiments found that simply generating a code sequence of a certain length through a short natural language description is not ideal. Therefore, we combine the deep learning method and the empirical knowledge of software engineering to generate the code tags for the corresponding natural language description, thereby facilitating the efficiency of code retrieval. The experimental results demonstrate that our method designed in this paper offers good performance results in improving the efficiency of code retrieval. The key contributions of this paper are as follows:

(i) A new code tagging framework that combines deep learning and software engineering empirical knowledge

(ii) A large-scale dataset of Java code tags that contains 717,980 pairs of text summaries and code snippets

(iii) A novel code tag mining framework that leads to a significant improvement of code retrieval compared to the state-of-the-art methods

The rest of this paper is organized as follows. Section 2 reviews previous works. Section 3 describes our methods. Section 4 shows the experimental design and results, and the last section concludes this paper.

2. Related Work

Many works have been proposed for code tagging and code retrieval. We will review these works in three categories including code retrieval, deep learning, and tag measurement.

2.1. Code Retrieval. In recent years, code retrieval has become a research topic of interest in software engineering. Researchers have proposed a variety of code retrieval methods. These studies cover multiple aspects, provide multiple forms of input, and recommend code resources at various levels. The following reviews typical research work about code retrieval.

INQRES [9] considers the relationship between each pair of words in the source code and interactively reconstructs the search query to optimize the query quality. Bajracharya [10] proposes a systematic model that takes a natural language query as input, finds the source code implementation of the corresponding function and the calling methods of existing code snippets, and uses the TF-IDF method and boosting technology to identify popular classes. XSnippet [11] takes a dedicated query statement as input. The advantage of XSnippet is that it divides the query into two steps to expand the scope of the query. Sourcerer [12] is a code retrieval tool based on Lucene that combines code attributes and code popularity as an indicator to evaluate the quality of recommended codes and then retrieves relevant code snippets.

The above works have their own characteristics in the study of code retrieval. They have contributed corresponding solutions to the code retrieval problem by analyzing the empirical knowledge of software engineering and the laws of natural language. However, retrieval patterns and code tag characteristics cannot be exhausted through manually constructed rules. The TagNN method proposed in this paper combines the characteristics of external rules with data-driven generalization ability through deep learning, which has certain innovations.

2.2. Application of Deep Learning in Natural Language Processing. The research area of deep learning in natural language processing has focused on sentence-level or document-level text representation and classification methods as follows.

UNIF [13] uses the attention mechanism to combine the embedding of each token in the code snippet and generates an embedding vector representation of the entire code fragment. Pennington et al. [14] propose using the global "word-word" co-occur matrix to obtain a word vector representation in the GloVe method. Hill et al. [15] propose learning distributed expressions corresponding to sentences from unlabeled data. Conneau [16] proposes using supervised learning to learn general sentence representations from natural language inference data. Tai et al. [17] improve the semantic representation model of long- and short-term memory networks with tree structures.

These works use deep learning methods to process and analyze natural language. However, few works applied deep learning methods to the code snippet data. Moreover, although the composition of the code is similar to natural language expression, there are still some differences. Different from other works, TagNN uses the empirical knowledge of software engineering to process the code snippets so that the deep learning method could effectively process the code snippets with good results.

2.3. Tag Measurement. The quality of tags is mainly measured from two aspects: similarity and generalization. The similarity measure indicates the distance between tags. The higher the value is, the closer the meaning or the stronger the association is. Wang et al. [18] integrated tag annotation through a labeling system that exists in the open-source community itself and then measured the textual similarity of open-source resources on this basis. Begelman et al. [19] measured the similarity of tags by the number of times the tags appear together.

The generalization degree of a tag represents the number of categories contained in the tag. The larger the value is, the higher the level in the tag hierarchy. Schmitz [20] adjusted the threshold to control the usage of special tags and increased the special vocabulary in the filter to improve the quality of the generated tags.

The above are all very classic works in this field which mainly judge the quality of the tag by measuring the attributes of the tag itself. In this paper, TagNN will judge whether tags are good or bad based on their effectiveness in assisting code retrieval that is different from the above work.

3. Overview of TagNN

In this section, we describe the framework of our approach. TagNN involves two main methods. One is a tag generation model based on deep learning methods, and the other uses TF-IDF to extract keywords from generated code snippets [21–24]. The deep learning model has a good effect on natural language processing, and TF-IDF has a good effect on extracting key information in the text. Thus, we combine the two types of methods for code tag generation. As shown in Figure 1, TagNN consists of five parts: data collection, training data processing, model training, model testing, and code retrieval.

Figure 1 shows the framework of our approach, which consists of five steps for code retrieval with code tags. This section describes the first three parts (i.e., data collection, training data processing, and model training), and Section 4 describes the last two parts (i.e., model testing and code retrieval).

We give a brief description of the first three parts. First, we obtain a large number of high-quality open-source projects from the open-source community, from which we extract code snippets and corresponding summary information. Then, we analyze the summary information according to the empirical knowledge of the natural language and process the code snippets according to the software engineering domain knowledge to obtain high-quality summary information and code snippets. Next, with the previously obtained dataset, we train the deep learning model. In addition, based on the trained model, we input the natural language description information to acquire the corresponding code tags. Finally, the generated code tags are passed to the corresponding code retrieval method to improve the efficiency of code retrieval.

3.1. Data Collection. We use the *Kraken* (https://forgeplus.trustie.net/projects/zenglingbin12/summer_all) [25] tool to acquire data, and the specific steps are shown in Figure 2.

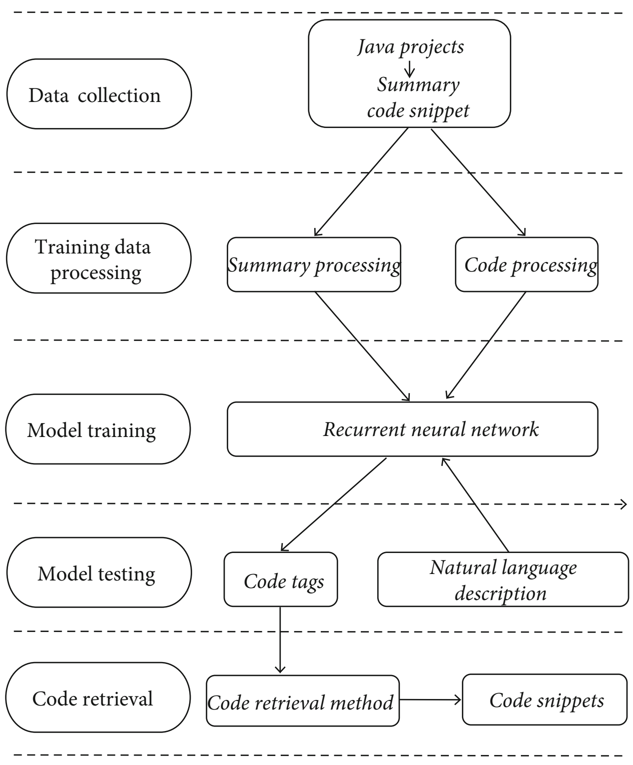

Figure 1: The framework of the TagNN.

Step 1 (Project requirements). First, we need to clarify the type, quantity, and sorting criteria of the project to be obtained. However, measuring the quality of open-source projects is a complex system engineering problem. In this paper, we adopt the concept of crowd intelligence and use the star mechanism in GitHub to filter projects. Specifically, when users in the open-source community like the project, they can give the project a star. The more stars a project has, the more people acknowledge the project. Based on this concept, we collected the top-ranked projects in GitHub as the source of projects.

Step 2 (Project lists). According to the requirements mentioned before, we use the API provided by GitHub to obtain the metadata of all projects through the *Kraken* tool and then analyze the data and sort out the project list.

Step 3 (Cloning projects). According to the project lists, we use the protocol provided by the Git tool to clone and store the remote projects locally. Because the protocol provided by the Git itself is single-threaded, it is difficult to clone projects concurrently on a large scale. Thus, we design a multithreaded concurrent algorithm to clone projects.

Step 4 (Extracting code files). Every project contains many different types of files, including documentation, technical manuals, and source codes. We need to filter out the source code files. In our work, we filter the corresponding code files by file name suffix matching. In addition, it should be noted that we believe that each project has only one main programming language. For projects composed of multiple programming

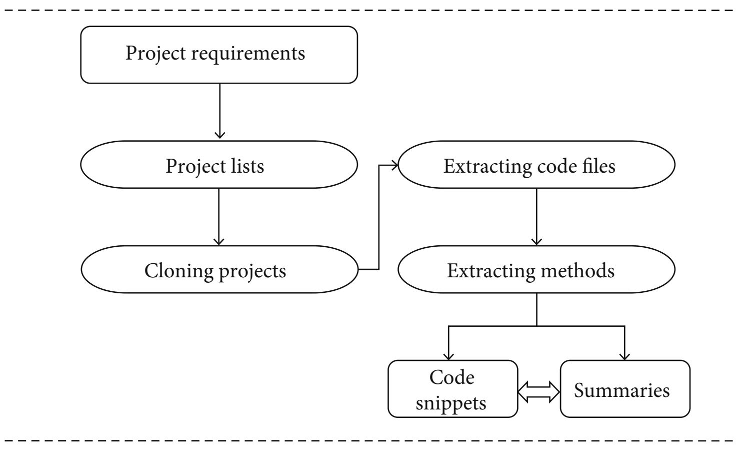

Figure 2: Data collection processing.

languages, we clarify the main programming language of the project through the information provided by GitHub.

Step 5 (Extracting methods). For the obtained source code files, we use the characteristics of the programming language and the empirical knowledge of natural language to filter out method-level code snippets and corresponding description information in the code files.

Step 6 (Paired dataset). The method-level code snippets and summaries are stored in a local database as a basic dataset for model training and testing.

3.2. Training Data Processing

3.2.1. Summary Processing. In the training data processing module, we process information of code snippets and code summaries separately. We believe that the quality of code snippets and code summaries extracted from high-quality projects in the GitHub community is generally high. However, the open-source community has a large number of contributors, and their mastery levels are uneven, which inevitably leads to uneven code summary quality. There could be invalid or nonfunctional descriptions. Based on this consideration, through the observation of the code summary information and the understanding of language grammar rules, we developed heuristic rules to filter code descriptions. Figure 3 shows our process and rules for filtering code summaries. We have a total of six steps for summary processing.

Step 1 (Remove @ block information). For programmers who use the integrated development environment, as shown in Figure 4, if they create a comment after writing the code, the integrated development environment automatically adds some predefined information for the class-level and method-level code snippets. Take the well-known integrated development environment *Eclipse* as an example. It can automatically generate information such as “@author,” “@data,” and “@return.” Although this information can help developers understand the code better, they are not functional descriptions. Therefore, in this step, we remove the code summaries that contain “@” block information.

Step 2 (Remove other @ information). After removing the “@” block information in the first step, there is still some “@” information added by the software developer in the code summaries. As shown in Figure 5, “@link” indicates the class related to the object. In addition, there is still information similar to “@deprecated” and “@code.” We use regular expressions to remove the code summary information that contains these “@” information.

Step 3 (Remove web page information). Through the analysis of the code summary data, as shown in Figure 6, we found that there are many web page tag elements to better display the code summary information. However, these web page tags are noisy data for the code description data, which is not good for model training. We use regular expressions to remove these web page tag data.

Step 4 (Remove punctuation information). The code summary is written by the software developer during the code development, which contains many punctuation marks (e.g., “,”, “.”, “?”, “!”, “:”, and “;”) and other symbols. To reduce the noisy influence of punctuation marks on the code summary information, as shown in Figure 7, we remove the punctuation marks and remove the summary information containing the question mark.

Step 5 (Remove non-English vocabulary). For the method level, we only consider the functional summaries of the method. Therefore, as shown in Figure 8, we remove the summary information that contains non-English words. In this step, we use Python’s *pyenchant* module to check each word described.

Step 6 (Remove the description that is too short). In this step, we remove the ambiguous code description information.

As shown in Figure 9, this code summary information can only be understood by the code writer. Generally, a complete code description in English must have at least one verb

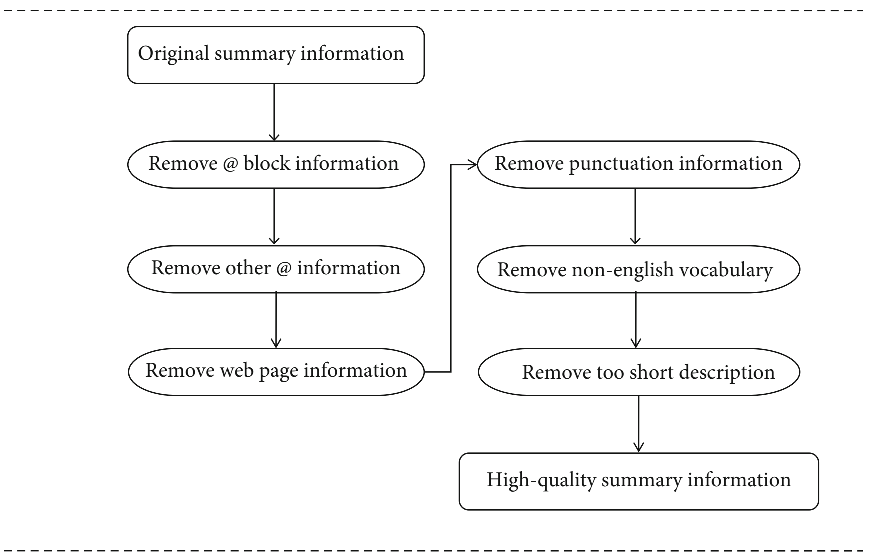
Original summary information
Remove @ block information
Remove other @ information
Remove web page information
Remove punctuation information
Remove non-english vocabulary
Remove too short description
High-quality summary information

Figure 3: Summary information processing.

```
/**Write a localized message, using the default resource bundle.
   *@param key the key for the message to be localized
   *@throws IOException if there is a problem closing the underlying stream
 */
public void write I18N (String key) throws IOException {
  write (getString (i18n, key)) ;
}
```

Figure 4: Summaries containing "@" block information.

```
/ **Returns the string for rendering the{@link IJavaElement#getElementName() element name} of
   * the given element.
   */
protected String getElementName (IJavaElement element) {
   return element.getElementName();
}
```

Figure 5: Summaries containing other "@" information.

```
/**Returns the coefficient of determination <em>R</em><sup>2</sup>.
  * @return the coefficient of determination <em>R</em><sup>2</sup>,
which is a real number between 0 and 1
*/
public double R2() {
   return R2 ;
}
```

Figure 6: Summaries containing web page information.

```
/ *??  ´  ? ´   ?  ´  /´   ?????* http://mavin-manzhan.oss.-cn-hangzhou
aliyuncs.com/  ...
*/
private static String getUrlFileName(String url) {
        String filename = null;
        String[] strings = url.split("/");
        ...
        return filename; }
```

Figure 7: Summaries containing punctuation information.

and one object. Therefore, we have deleted descriptions that are fewer than two words.

3.2.2. Code Processing. In this section, we deal with method-level code snippets to obtain code tags, as shown in Figure 10.

(1) Code Segmentation. As shown in Figure 11, a piece of the original code snippet is successfully segmented after six steps of processing.

Step 1 (Remove parentheses). Parentheses in the code snippets have no actual special meaning. Thus, we replace them with blanks in the first step.

Step 2 (Remove punctuation information). We delete the punctuation information of the code snippet as we did for the summary before.

Step 3 (Remove underlining). When writing Java codes, some programmers are accustomed to using underscores in

```
/*writes data to a random filename
 (update_<per JVM random UUID>_<COUNTER>.tmp)
*/
private static DiskFileItem write ( String dir, byte[] data ) throws IOException, Exception
   {
       return makePayload(data.length + 1, dir, dir + "/whatever", data);
    }
```

Figure 8: Summaries containing non-English vocabulary.

```
// where
void findFiles (File dir, Set<File> files) {
    for (File f: dir.listFiles()) {
        if(f.isDirectory())
          findFiles(f, files);
      else
          files.add(f);
    }
}
```

Figure 9: Overly short description.

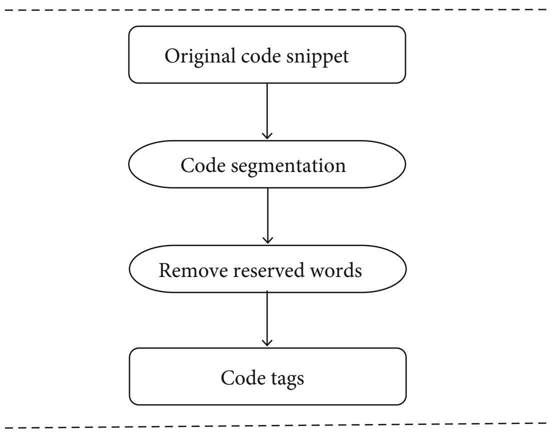

Figure 10: The key steps of code processing.

variable names to connect nouns. We delete the underscores and decompose the variable names.

Step 4 (Dividing codes). The Java code writing rules follow the camel case naming rules, so we decompose the variable names based on the camel case rules and decompose each independent word from it.

Step 5 (Remove Arabic numbers). The Arabic numbers themselves do not represent special meanings, so we remove the Arabic numbers that exist after the code participles.

Step 6 (Lowercase vocabulary). After the final processing, we uniformly convert the vocabulary to lowercase and finally output high-quality code snippets.

(2) Remove Reserved Words. The keywords of the Java code exist in the system itself and have little meaning for the characterization of the function itself [26]. Therefore, when we obtain the code snippets processed in the first step, we process the code and delete the Java keywords.

(3) Generate Tags. After the first and second steps, the code snippets become code vocabulary sequences. We use the TF-IDF algorithm to select the most representative words for each code snippet in the entire training set. TF-IDF is a statistical method that evaluates the importance of a word to one of the documents in a document set. The importance of a word increases in proportion to the number of times it appears in the document, but at the same time, it decreases in inverse proportion to the frequency of its appearance in the corpus [27, 28]. Through the TF-IDF algorithm, we generate the top ten important words in each code snippet. The ten words are used as the tag of the code snippet to characterize it.

3.3. Model Training.

After the data are processed in the second step, TagNN implements the construction of the model through a recurrent neural network (RNN) algorithm. We choose the classic Encoder-Decoder model that is often used in natural language processing. Next, we will give a general introduction to the selected model.

3.3.1. Input. We use the natural language description processed by heuristic rules as the input of the model.

3.3.2. The Basic Theory of the Model. The RNN is a classic neural network model. It is composed of an input layer, a hidden layer, and an output layer. For the convenience of description, we use d to refer to the input layer, t to refer to the output layer, and h to refer to the hidden layer. The depth of the hidden layer can be set flexibly. The state of the hidden layer h is changed by its previous state and the influence of the state d, and finally, t is affected by the cumulative network weight propagation [29–31]. Besides, this paper uses the LSTM as an activation function [32], which has a good effect on natural language processing. Next, we will introduce the Encoder and Decoder in detail.

(1) Encoder. The Encoder is essentially an RNN. This paper takes the natural language description of the code snippet as the input sequence D which inputs the words into the model one by one from the head position to the tail position, and the state of the corresponding hidden layer changes accordingly. After the D sequence is input and processed by the hidden layer, the Encoder will output the intermediate state m, as shown in Figure 12 [30].

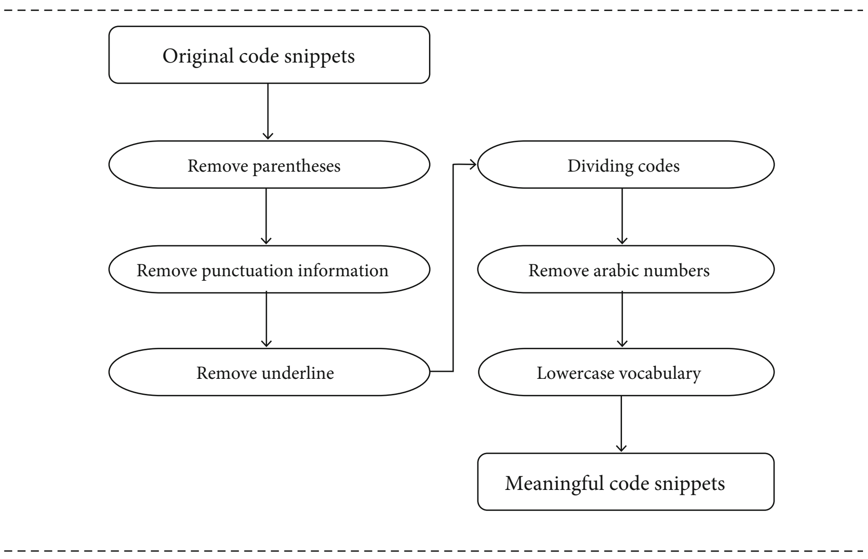

Figure 11: The key steps in code snippet processing.

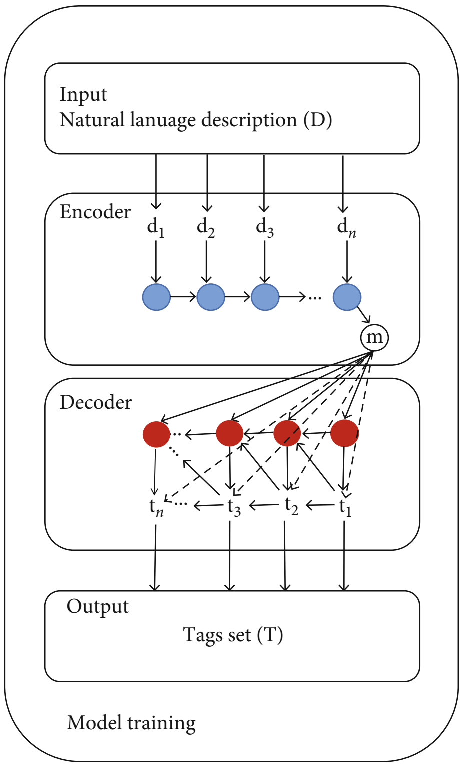

Figure 12: The architecture of model training.

(2) Decoder. The Decoder is also an RNN like the Encoder. When the Encoder outputs the state m to the Decoder, the Decoder will output t_i one by one, which is a tag that is used to measure the code snippets. Finally, the Decoder will output the sequence T which is the set of tags.

3.3.3. Output. The output of the model is a set of tags that we want to measure the code snippets.

4. Experimental Design and Effect Verification

To demonstrate the validity of our TagNN model, we designed two related problems and conducted corresponding experiments. This section introduces experimental data and evaluates our experimental results.

4.1. Experimental Setup

4.1.1. Model Settings. Table 1 shows the basic parameters of TagNN. We implement TagNN with the famous TensorFlow [33] framework. The model has six hidden layers, each of which has 128 neurons. The neuron type is LSTM, and the learning rate is set as 0.5.

4.1.2. Data Settings. We design an experiment based on GitHub's Java projects, from which we selected the top 5,000 Java projects based on the ranking of stars. After screening and distinguishing abstracts, we selected 717,980 summary-code pairs that met the conditions. As shown in Table 2, we use 80% of the data for the training set, 10% for the validation set, and 10% for the test set.

4.2. Research Question. To study the effect of TagNN on tag generation and whether the generated tags can help improve the search and retrieval of open-source codes through natural language, we propose the following two research questions:

(1) Question 1. What is the effect of generating code tags for natural language through TagNN?

(2) Question 2. Can the code tags generated by TagNN improve the accuracy of natural language retrieval codes?

For the first question, we analyzed the accuracy of the code tags generated by TagNN, and for the second question,

Table 1: Basic parameters of TagNN.

Training tools	TensorFlow
Hidden layers	6
Number of neurons per layer	128
Neuron type	LSTM
Learning rate	0.5

Table 2: Experimental dataset.

The total amount of experimental data	717,980
Training set	80%
Validation set	10%
Test set	10%

Table 3: The effect of TagNN.

The total amount of test data	71,798
Accuracy rate	78.03%
Recall rate	31.00%

we used the traditional code search matching method plus the code tag data to see how it affects the search results.

4.3. Metric Methods. For problem one, the experiment uses the accuracy index. When the predicted tag generated by the model matches the true tag of the code snippet, the accuracy index is one; otherwise, the value is zero. In addition, we use the index of the recall rate, which is the probability that the true tags of the code snippets appear in the tags generated by the model.

4.4. Results. This section presents the final results of the above two experiments and targeted comparative analysis.

4.4.1. The Effect of Generating Code Tags for Natural Language through TagNN. We use the TagNN model trained by the RNN to read the 71,798 natural language descriptions in the test set and correspondingly generate 71,798 code tags. As shown in Table 3, the accuracy rate of tag generation is 78.03%, and the recall rate is 31.00%. The experimental results prove that our model has a high accuracy rate and a reasonable recall rate. To the best of our knowledge, this is the first time that code tags have been generated through natural language by deep learning methods to facilitate the retrieval of code snippets in natural language. The accuracy rate of 78.03% demonstrates that TagNN has a good effect in generating code tags for natural language.

4.4.2. The Impact of Code Tags Generated by TagNN on Natural Language Retrieval Codes. TagNN generates code tags by describing natural language to improve the efficiency of code retrieval. The role of TagNN tags is to allow existing code retrieval methods to achieve better results after using tags.

We selected the classic TF-IDF algorithm for code retrieval. Through the TF-IDF algorithm, we measure the similarity scores of a single natural language description and each piece of code and sort them with the similarity score from highest to lowest. We believe that if the target code snippet appears in the top ten code snippets, then the search task is successful. As shown in Table 4, before adding tags, the retrieval accuracy rate was 34.2%, while after adding tags, it was 40.03%, an increase of approximately 17.04%. This means that the code tags generated by TagNN are of great help in improving the accuracy of natural language retrieval codes.

Table 4: The boost effect of TagNN.

Algorithm	Accuracy rate
Classic TF-IDF algorithm	34.2%
TF-IDF algorithm with code tags	40.03%
Boost effect	17.04%

Table 5: Response categories for relevance evaluation.

Scale	Response category
5	Very relevant
4	Relevant
3	Neither relevant nor irrelevant
2	Irrelevant
1	Very irrelevant

4.5. User Study. Through the above data analysis, we verified the accuracy of code tag generation and the effect of code tag-assisted code retrieval, which effectively proved the effectiveness of TagNN at the data level. To further measure the effectiveness of TagNN, we conducted a user study.

We invited 16 students with different backgrounds to evaluate the results. Among them, there are eight software development engineers, four master's students, and four doctoral students, all of whom have no less than five years of Java software development experience. They were asked to analyze the relevance of our tags and code snippets and the effectiveness of tags to help code retrieval.

4.5.1. Relevance Evaluation. We ask students to evaluate the relevance between the tags generated by TagNN and the corresponding code snippets. The evaluation uses the Likert-type method [34], which has a variety of different expression choices and can effectively measure users' agreement with the relevance of tags and code snippets. Users need to choose one of five candidate options to express their agreement with the degree of relevance. Table 5 indicates these options.

As shown in Figure 13, among the 10 samples, there were six samples with a median of 4, three samples with a median of 4.5, and one sample with a median of 3.5. The median average value is 4.1, which is more than 4. This shows that the tags generated by our TagNN method are highly correlated with the code snippets, reaching the relevant level.

4.5.2. Usability Evaluation. To determine whether the tags generated by TagNN are helpful for code retrieval, we organize users to evaluate the usability of tags for code retrieval. As before, we use the Likert-type method. As shown in

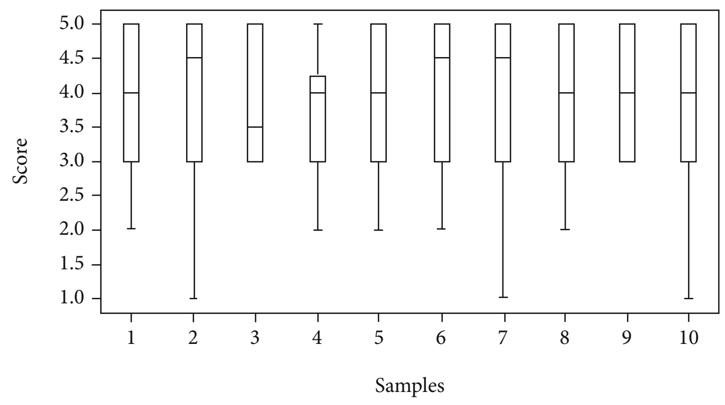

Figure 13: User evaluation on relevance.

Table 6: Response categories for usability evaluation.

Scale	Response category
5	Very useful
4	Useful
3	Not sure
2	Useless
1	Very useless

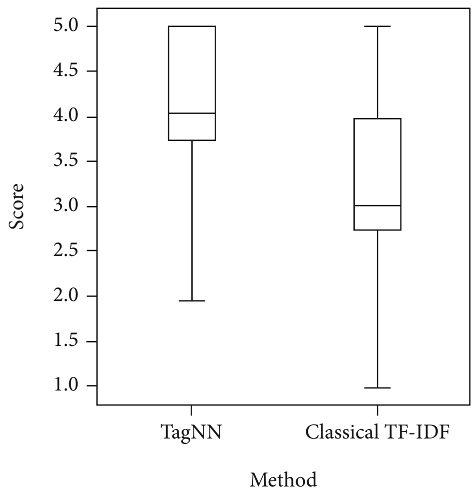

Figure 14: User evaluation on usability.

Table 6, users need to choose one of the five different options that best expresses their attitude.

As shown in Figure 14, the median value of the TagNN method is 4, and the average value is 4.06. The median value of the classic TF-IDF is 3, and the average value is 3.13. Experimental results show that from the views of users, the tags generated by TagNN have reached a useful level for code retrieval and are superior to the classic TF-IDF method.

4.5.3. Discussion. Through user evaluation, we found that the TagNN method demonstrates outstanding performance in relevance because the TagNN method combines the characteristics of deep learning methods with natural language and code language, which reflect the important characteristics of the code snippets.

As for the usability evaluation, TagNN has a better performance than the traditional TF-IDF. The reason is that the tags generated by the TF-IDF method are all derived from the code snippet itself, so there will be no vocabulary outside of the code snippet. However, TagNN uses deep learning methods and is trained based on a large amount of data to generate tags that may not be contained by the code snippet itself, which is more flexible and broad-sourced.

5. Conclusion and Future Work

In this paper, we aim to address the difficulty of retrieving massive codes in the open-source community to help developers quickly retrieve code resources, thereby speeding up the development efficiency of software developers and realizing the reuse and dissemination of high-quality code resources in the open-source community.

Based on the massive code snippets and natural language description information of the open-source community, we propose a novel code tag generation and code retrieval approach named TagNN, which combines software engineering empirical knowledge and a deep learning framework. Our method generates corresponding code tags for natural language descriptions through the RNN, thereby improving the retrieval effect. With large-scale experiments on the high-quality Java open-source project dataset collected from the GitHub community, we empirically evaluate the code tag generation effect of the model and the tags' role in improving the retrieval of code snippets.

There are still several shortcomings in our work. One is that when we selected the tags of code snippets in the training set, we simply applied the relatively rudimentary TF-IDF algorithm and did not choose a more effective weight measurement algorithm based on the actual situation of the code snippets. Second, we added the tag data directly to the natural language query to retrieve the code snippets without making more effective use of the tag data.

In future work, we will try to carefully analyze and observe the characteristics of the code snippets themselves and propose a more effective method of extracting code tags.

In addition, we will explore a more reasonable and effective application of tags to optimize the code retrieval effect.

Data Availability

The data used to support the findings of this study are available from the corresponding author (email address: zenglingbin16@nudt.edu.cn) upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (2020AAA0108802), the National Natural Science Foundation of China (61702532), and the scientific research project of the National University of Defense Technology (No. ZK20-47).

References

[1] A. Terminanto, A. N. Hidayanto, and F. B. Utomo, "Implementation open source system resource planning in sustainable supply chain management of small and medium enterprise," *International Journal of Supply Chain Management*, vol. 9, no. 3, pp. 472–495, 2020.

[2] N. Zöller, J. H. Morgan, and T. Schröder, "A topology of groups: what GitHub can tell us about online collaboration," *Technological Forecasting and Social Change*, vol. 161, article 120291, 2020.

[3] O. G. Glazunova, O. V. Parhomenko, V. I. Korolchuk, and T. V. Voloshyna, "The effectiveness of GitHub cloud services for implementing a programming training project: students' point of view," *Journal of Physics Conference Series*, vol. 1840, no. 1, article 012030, 2021.

[4] E. M. Kavuk and A. Tosun, "Predicting Stack Overflow question tags: a multi-class, multi-label classification," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, Seoul, Republic of Korea, 2020.

[5] T. Wang, H. Wang, G. Yin, C. Yang, X. Li, and P. Zou, "Hierarchical categorization of open source software by online profiles," *IEICE Transactions on Information and Systems*, vol. E97.D, no. 9, pp. 2386–2397, 2014.

[6] P. Zhou, "Scalable tag recommendation for software information sites," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria, 2017.

[7] L. S. Ambati, "Factors influencing the adoption of artificial intelligence in organizations-from an employee's perspective," in *MWAIS 2020 Proceedings*, Des Moines, Iowa, 2020.

[8] S. Sophia and S. P. Rajamohana, "A survey on feature selection based spam review detection using deep learning techniques," *International Journal of Advanced Information and Communication Technology*, pp. 102–108, 2020.

[9] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou, "Interactive query reformulation for source-code search with word relations," *IEEE Access*, vol. 6, 2018.

[10] S. K. Bajracharya, "Sourcerer: a search engine for open source code supporting structure-based search," Companion to the Acm Sigplan Symposium on Object-oriented Programming Systems ACM, 2006.

[11] N. Sahavechaphan and K. Claypool, "XSnippet: mining for sample code," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Portland Oregon USA, 2006.

[12] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: an internet-scale software repository," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, Vancouver, BC, Canada, 2009.

[13] J. Cambronero, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn, Estonia, 2019.

[14] J. Pennington, R. Socher, and C. D. Manning, "GloVe: global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, Doha, Qatar, 2014.

[15] F. Hill, K. Cho, and A. Korhonen, "Learning distributed representations of sentences from unlabelled data," 2016, https://arxiv.org/abs/1602.03483.

[16] A. Conneau, "Supervised learning of universal sentence representations from natural language inference data," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen,Denmark, 2017.

[17] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *Computer Science*, vol. 5, no. 1, 2015.

[18] S. Wang, L. O. David, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy, 2012.

[19] G. Begelman, P. Keller, and F. Smadja, "Automated tag clustering: improving search and exploration in the tag space," in *collaborative web tagging workshop at WWW2006*, Edinburgh, Scotland, 2006.

[20] P. Schmitz, "Inducing ontology from Flickr tags," in *Collaborative Web Tagging Workshop at WWW2006*, vol. 50, Edinburgh, Scotland, 2006.

[21] G. T. Reddy, M. P. K. Reddy, K. Lakshmanna et al., "Analysis of dimensionality reduction techniques on big data," *IEEE Access*, vol. 8, pp. 54776–54788, 2020.

[22] N. Deepa, "A survey on blockchain for big data: approaches, opportunities, and future directions," 2020, https://arxiv.org/abs/2009.00858.

[23] M. Z. Asghar, F. Subhan, H. Ahmad et al., "Senti-eSystem: a sentiment-based eSystem-using hybridized fuzzy and deep neural network for measuring customer satisfaction," *Software: Practice and Experience*, vol. 51, no. 3, pp. 571–594, 2021.

[24] G. R. Bojja and J. Liu, "Impact of IT investment on hospital performance: a longitudinal data analysis," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, Hawaii, USA, 2020.

[25] L. B. Zeng, "Kraken: a continuous incremental data acquisition system for GitHub and Git repositorie," in *Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering*, Beijing, 2017.

[26] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, Addison-Wesley Longman Publishing Co. Inc., 1996.

[27] L. Havrlant and V. Kreinovich, "A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation)," *International Journal of General Systems*, vol. 46, no. 1, pp. 27–36, 2017.

[28] J. Ramos, "Using TF-IDF to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, Piscataway, NJ, 2003.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, 2014.

[30] K. Cho, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, https://arxiv.org/abs/1406.1078.

[31] J. C.-W. Lin, Y. Shao, Y. Djenouri, and U. Yun, "ASRNN: a recurrent neural network with an attention model for sequence labeling," *Knowledge-Based Systems*, vol. 212, article 106548, 2021.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] M. Abadi, "TensorFlow: a system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, Savannah, GA, 2016.

[34] S. Jamieson, "Likert scales: how to (ab)use them," *Medical Education*, vol. 38, no. 12, pp. 1217-1218, 2004.