applied sciences

MDPI

Article

A Source Code Similarity Based on Siamese Neural Network

Chunli Xie *, Xia Wang, Cheng Qian and Mengqi Wang

Department of Computer Science & Technology, Jiangsu Normal University, Xuzhou 221116, China; 6020040016@jsnu.edu.cn (X.W.); 3020172212@jsnu.edu.cn (C.Q.); 3020170269@jsnu.edu.cn (M.W.)

* Correspondence: xiechunti@jsnu.edu.cn; Tel.: +86-150-6210-7762

Received: 23 September 2020; Accepted: 22 October 2020; Published: 26 October 2020

Abstract: Finding similar code snippets is a fundamental task in the field of software engineering. Several approaches have been proposed for this task by using statistical language model which focuses on syntax and structure of codes rather than deep semantic information underlying codes. In this paper, a Siamese Neural Network is proposed that maps codes into continuous space vectors and try to capture their semantic meaning. Firstly, an unsupervised pre-trained method that models code snippets as a weighted series of word vectors. The weights of the series are fitted by the Term Frequency-Inverse Document Frequency (TF-IDF). Then, a Siamese Neural Network trained model is constructed to learn semantic vector representation of code snippets. Finally, the cosine similarity is provided to measure the similarity score between pairs of code snippets. Moreover, we have implemented our approach on a dataset of functionally similar code. The experimental results show that our method improves some performance over single word embedding method.

Keywords: code similarity; word embedding; siamese neural networks

1. Introduction

Code similarity is often used to measure the similarity degree between a piece of code snippets on text, syntax and semantic. Code similarity is a fundamental activity for some applications, such as code cloning detection [1,2], code plagiarism [3–5], code recommendation [6] and information retrieval [7]. Indeed, several efforts have been made for finding similar codes for each given code snippet. Specifically, manual defined or hand-crafted features, e.g., by analyzing the overlap among identifiers, operators, operands, lines of code, functions, types, constants and other attributes or comparing the abstract syntax trees of two code snippets, are conducted for small code snippets. However, this method is a coarse-grained measurement with low accuracy [8]. Additionally, programs have a well-defined syntax, they can be represented by a series of tokens, ASTs (Abstract Syntax Trees), PDGs (program dependency graphs) or CFGs (Control Flow Graphs), which can be successfully used to capture code patterns. Then the similarity of codes is measured by string matching [9], suffix tree matching [10], graph matching [11] and other algorithms.

With the increasing arising of valuable, widely-used, open-source software, the scale of available data, such as billions of tokens of code and millions of instances of meta-data, is massive. It's hard to extract ASTs, PDGs and CFGs from big code snippets. Then, a new approach to measure the similarity is required. Software engineering and programming languages researchers have largely focused on machine learning techniques to extract features, because they have shown high performance in Natural Language. Hindle et al. first proposed the assumption of code naturalness [12], and proved that source codes had similar characteristics to natural language, that provided a new idea for source code representation and analysis [13–15]. While there are many differences between natural language and

source codes, for example, code has formal syntax and semantics, so, it is still an open question to understand codes.

In summary, there are many unique challenges inherent in designing an effective approach for codes. First, words in code have a specific meaning, i.e., 'while', 'for' words mean loop, *'if' and 'switch'* mean choice. How to understand the semantic of codes from their contents and represent these semantics automatically is a nontrivial problem. Second, code has rich words and has higher neologism rate than text. Most of characters are identifiers and a developer must name new things using a new identifier that makes code corpora drastically change. Programs consist of identifiers, sentences and structures, in which some are user-defined words, some are reserved words. Their contributions are different for code presentations. Thus, it is necessary to capture the statistical behavior of a word appearing along with other words.

To address the challenges mentioned above, in this paper, we first build a pre-training model that embeds each word of codes into a real vector based on Word2Vec and we also design another model to learn words frequencies in code snippets. We model a code snippet as a matrix rather than averaging and summing word embedding to represent sentences. The weight of each word is the learned word frequency. Then a neural network is proposed to learn automatically semantic features underlying codes. Finally, the cosine similarity scores of source code pairs are calculated based on their representations. We call this approach Word Information for Code Embedding-Siamese Neural Networks (WICE-SNN). This method applies deep learning technology to the measurement of code similarity which can help find deep semantic information and higher abstract features underlying codes than traditional machine learning method [16–19]. We implemented our method and evaluated their recall, precision on a dataset. Experimental results reveal that WICE-SNN significantly outperforms some baselines on similar code task.

The main contributions of this paper are as follows:

- We construct a method that incorporates words statistical information in code snippets. This approach regulates the weight of each word to its corresponding sentence representation according to its contribution.
- We propose a siamese neural network that extracts semantic features by utilizing the similarity among source codes and makes code snippets with similar function mapped into similar vectors.

The rest of this paper is organized as follows: Section 2 introduces the current situation of related research, summarizes and analyzes it. In Section 3, we present our similarity computing framework and explain the constitution of our model. Section 4 demonstrates an experiment to evaluate the proposed method and presents the obtained results. Section 5 concludes our work.

2. Related Work

2.1. Source Code Similarity

The research on source code similarity originated in the 1970s. Assessing source code similarity is to measure similar implementation of functions, which is a fundamental activity in software engineering and it has many applications. Many approaches for code similarity have been proposed in literatures. Most of these approaches focus on syntactic similarity rather than semantic similarity. These techniques can be classified into two categories: attributed-based and structure-based. The early approach proposed by Halstead measure similarity based on properties of software, such as numbers of different operators, operands, variable types and other attributes in the program [20]. The programs with similar attributes have higher similarity. The attributed-based methods are simple, but are easy to be interfered by variable substitution. Structure-based approaches include text-based, token-based, tree-based and graph-based. Text-based approaches treat code snippets as two string sequences and compare their similarity. One of the widely-used methods of string similarity proposed by Roy and Cordy is to find a longest common subsequence. This technique is independent of the programming language,

but ignores the syntax and structure information of the program [9]. For token-based technology, source codes can be transformed into tokens. A stream of tokens is an abstract representation of a program and useless characters, spaces, comments, etc., are filtered out. This method is not affected by textual difference, but it does not consider the order of the code, and it ignores the structural information in code snippets [1]. Tree-based and graph-based code similarity measurement focus on structural information between two programs and can avoid lexical differences [10,21], but the cost of computation is very large, especially graph-based algorithms are mostly NP-complete.

Text-based, token-based, tree-based and graph-based approaches show good performance in small-scale programs. However, with the development of open sources on Internet, more and more open source codes with similar functions but different forms are available. The above methods are not appropriate anymore. Researchers try to introduce machine learning techniques to solve the similarity task. Machine learning methods can be divided into keywords-based, vector space-based, and deep learning-base approaches. N-gram similarity and Jaccard similarity are the two mainly keywords-based algorithms. It measures similarity by the number of common substrings in two code fragments. Jaccard method measures similarity from the ratio of intersection and union of word sets between two texts. Word2vec is the common vector-based methods, in which words are mapped into vectors and similarity is the distance between two corresponding vectors [22]. As deep learning techniques has so excellent performance in natural language processing and computer vision [23], researchers has begun to apply it to software engineering, such as programming analysis [24], code plagiarism [25], code clone [26], code abstract generation, fault location and other applications [27,28]. These new deep learning approaches use representation learning to extract automatically useful information from the amount of unlabeled code data. The methods reduce the cost of modeling code and improve performance because they are more efficient than human's at discovering hidden features in high dimensional spaces.

2.2. Source Code Representation

Source code representation plays a key role for code similarity. In this section, we review some recent and representative work about code similarity tasks. There are two common code representation methods: statistical-based and vector-based approach. Statistical-based methods introduce some probabilistic models to represent source codes. TF-IDF is a popular statistical representation method usually used in Information Retrieval applications. The representation of text consists of a term-document matrix which usually describes the word frequencies in the document. The TF-IDF model assigns relatively low weights to very frequent words and high weights to rare words. For a source code snippet c, it is represented by N reversed words, $c = \{v_1, v_2, \ldots, v_N\}$, where the weight of v_i is f_i, $f_i \in F = \{f_1, f_2, \ldots, f_N\}$, F is a TF-IDF set. Word embedding is a vector-based representation that produces a low-dimension continuous distributed vector for word. The word2vec model proposed by Mikolov et al. is the most common word embedding method, which constructs a three-tier shallow neural network to predict the distributed vector of each word [29]. It is also introduced into the representation of source code. Ye et al. trained a low-dimensional vector for code snippets extracted from API documents to measure the similarity of documents. This model were used to software search and fault locating tasks based on the trained vector space distance [30,31]. Chen et al. extracted a large number of code corpus from the stack overflow and other programming Q&A communities, and processed the synonym problems based on the word vector. It provided the basis for code conversion and other work [32]. Hao et al. parsed the abstract syntax tree for the source code, and mapped it to its corresponding real-valued vector representations, so that similar AST node have similar vectors [33]. Mou extracted abstract syntax trees from codes, and provided a tree-based CNN to learn word vectors of codes which was used in code classification and searching [34]. Nguyen proposed a new DNN network to extract lexical and grammatical features of code in each layer [24]. White proposed an auto-encoder deep learning framework applied in code clone detection,

which learn lexical and syntactic feature of code [35]. Wang used a deep belief network to learn semantic features from token vectors extracted from ASTs to perform defect prediction [36].

Most of the aforementioned works are developed for a specific task, not purely code representation. Unlike such approaches, our method uses deep learning only for training word embedding to learn more complex semantic and syntactic structure features underlying source code. Then we model a code snippet as a weighted series of word vectors and the weights were learned from their TF-IDF values.

3. WICE-SNN Framework

We first give the formal definition of source code similarity in this section and introduce the code embedding method based on statistics information proposed in this paper.

3.1. Definition of Source Code Similarity

Similar code snippets should have same objective in function related with the semantics of codes. For any two code snippets c_1, c_2, we measure the similarity score between c_1 and c_2. This score is a real number in [0, 1]. The higher the score is, the more similar c_1 and c_2 are. Without loss of generality, the problem of similar code snippets can be formulated as:

Definition: Given a set of code snippets pairs $C = \left\{\left(c_a^1, c_b^1\right), \left(c_a^2, c_b^2\right), \ldots \left(c_a^n, c_b^n\right)\right\}$, a set of real numbers $Y = \{y_1, y_2, \ldots y_n\}$, Y is the set of standard similarity score manually annotated for the set of code snippets pairs C. $\left(c_a^i, c_b^i\right)(1 \le i \le n)$ is a pair of code snippets, y_i is 1, if c_a^i, c_b^i are similar; y_i is 0, if c_a^i, c_b^i are not similar. The objective of the task is to learn a similarity model with a function f, $f(c_a, c_b, \theta) \rightarrow R$, where $c_a, c_b \in C$ and θ is a hyper-parameter of f to be trained in model. Using this model, we can measure the similarity score for arbitrary code snippets.

As shown in Figure 1, in order to tackle the above problem, we propose a Siamesed CNN model which can be used to capture the similarity semantic feature between two code snippets. The first step is word embedding and word frequency. We split the source code into variables, function name, operators, reserved words, constant value and others. Each of the word is mapped to its corresponding vector with a frequency. Then, the second step is source code representation, the input are code matrices got from the previous step. The hidden feature of code snippet are trained and mined in the model. In the last step, the similarity value is calculated based on the hidden features. We describe our model in detail as follows.

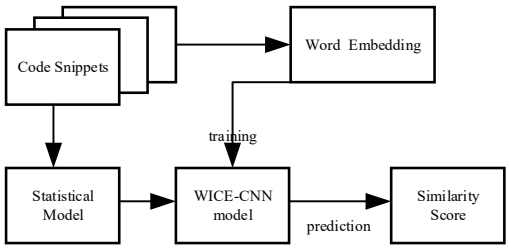

Figure 1. The flowchart of our work.

3.2. Details of WICE-SNN

In this section, we will introduce the technical details of WICE-SNN framework. As shown in Figure 2. This model mainly contains three parts: Word Information for Code Embedding layer (WICE), Siamese Neural Networks layer (SNN) and similarity score layer. WICE layer maps discrete words to real-valued vectors and extract the weight for each word in code snippet. SNN layer learns semantic representation of code snippets using the weighted word embedding as initialized value. Similarity layer calculates the similarity score between two code snippets with their semantic representations, which can be used to rank candidate code snippets to find its similar ones.

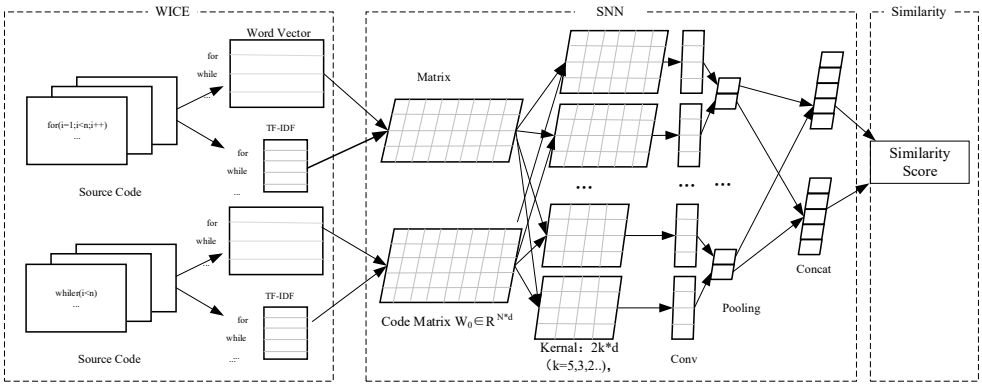

Figure 2. WICE-SNN Framework.

3.2.1. WICE Layer

Word embedding: Word2Vec is a widely word embedding model applied in NLP. Word2vec learns a vector representation for each word using a simple neural network. There have two network architectures: continuous bag-of-words (CBoW) model and skip-gram model. CBoW model uses the average of a context vector, also known as an input vector, to predict a word with softmax weight. Skip-gram model uses a single word to predict surrounding words. They consists of an input layer, a projection layer (hidden layer), and an output layer. Formally, a code snippet c is split into a word sequences, $c = \{v_1, v_2, \ldots, v_N\}$, where N is the number of words in code snippet c. The input of CBoW model is $\{x_1, x_2, \ldots, x_k\}$, $x_i \in R^{1\times N}$, is the one-hot vector of v_i, window size is k, vocabulary size is N. The converted vector u_i for one-hot vector of v_i is expressed as:

$$u_i = x_i \times W \tag{1}$$

where weight matrix $W \in R^{N\times d}$ are the paremeters of the embedding layer and $u_i \in R^{1\times d}$, where d is the dimension of hidden layer. Output layer is a log probability vector. Each word vector is trained to maximize the probability of neighboring words. The word which has the largest probability in the vector is the output word. As a result, the representation of code snippet is transformed into a matrix U which is a part of the weight matrix W.

Composition in Statistical Information: When using word vectors to represent code snippet, the weight of all word vectors are equally. But, we know the contribution of each word in code is different. For example, there are three main types of vocabulary in source codes, reserved words, user-defined words and operations. Reserved words that represent data types, control structures, library files and others; second, user-defined variable names, function names, etc.; third, various operation symbols. For source code snippets, the weight should reflect the structure of source code. In order to improve the effect of code presentation, the words of control structure should have different coefficients and multiplied by the word frequency of it. So, the vector of code snippet is represented as $U = \{f_1 \cdot w_1, \ldots, f_i \cdot w_i, \ldots, f_m \cdot w_m\}'$, where $w_i \in W$ is the i-th line of embedding matrix W, $\{f_1, f_2, \ldots f_m\}$ is the TF-IDF value of each word, m is the number of words in code snippet.

3.2.2. SNN Layer

In previous layer, we got the word embedding matrix U. One simple representation for code snippet can be modeled as a vector by a weigthed sum aggregated result of U. But, different words have different contributation to code snippet. Moreover, this representation will lost their program structure information. A CNN model is proposed to learn a representation for an input source code by integrating its semantic and structural materials. As shown in Figure 2, the CNN model contains five layers, i.e., input layer, convolution layer, pooling layer, connection layer and output layer. In the following, we will describe one by one.

Input Layer: We take a pair of pre-trained word embedding matrix U^A, U^B in previous part as input for WICE-SNN model, $U^A \in R^{N_1 \times d}$, $U^B \in R^{N_2 \times d}$, here N_1, N_2 is the number of words respectively in source code A and B, d is the vector dimension. We pad the two code snippets with 0 to have the same length $N = max\{N_1, N_2\}$. After filling with 0, the initialized marix $U \in R^{N \times d}$.

Convolution Layer: Each kernal $K \in R^{s \times d}$ does convolution operation in the word sequence $\{v_1, v_2, \ldots, v_N\}$.

$$p_i = U_i * K \tag{2}$$

Here, $*$ is convolution operator, $U_i = [u_i, u_{i+1,\ldots}, u_{i+s-1}]$, that is the embedding matrix of word sequence $\{v_i, v_{i+1}, \ldots, v_{i+s-1}\}, 1 \leq i \leq N - S + 1$. p_i is a real number, because the dimension of kernal and word vector are same. $P = [p_1, \ldots p_i, ..p_{N-s+1}]' \in R^{d_1}$, where $d_1 = N - s + 1$.

Pooling Layer: Pooling (including min, max, average pooling) is commonly used to extract robust features from convolution, to reduce its dimension. A convolution layer transforms an input feature map U with d columns into a new feature map P with one column. We get the maximum after max-pooling over each vector P, which can be expressed as

$$x_i = \max\{p_i\}, i = 1, 2, \ldots M \tag{3}$$

Here, M is the filter number we set in convolution layer.

Connection Layer: In connection layer, we concatenate each x_i, which get from pooling layer, into a vector for source code A and B.

$$X = x_1 \oplus x_2 \ldots \oplus x_M \tag{4}$$

where $\oplus$ is the operation that merges two vectors into a long vector, X is a new feature map for source code.

3.2.3. Similarity Score Layer

Similarity Score: This Layer targets at calculating the similarity score of each source code pair, which can be used to rank candidate code snippets to find similar ones for any source code. As shown in Figure 2, the similarity score of the input pair is computed by using cosine function on new feature vectors X which leverage their semantic representations and structure representations.

$$\text{sim}(X_A, X_B) = \frac{X_A \cdot X_B}{\| X_A \|_2 \| X_B \|_2} \tag{5}$$

where $\cdot$ is inner product of vector X_A and X_B, where $X_A = \| X_A \|_2$ an $\| X_B \|_2$ is their 2-norm. Those code snippets with the largest similarity score will be returned as similar codes of the given one.

4. Experiments

In this section, we present our empirical evaluation. In our experiments, we aim (1) to evaluate our method's accuracy in a dataset; (2) to compare it with the state-of-the-art.

4.1. Dataset

We evaluate our approach on a dataset that is collected from a programming open judge (OJ) platform (http://cstlab.jsnu.edu.cn). There are a large number of programming problems on the OJ system. Students submit their source codes as the solution to certain problems and the OJ system judges the validity of submitted source codes automatically. We select 35 C++ problems on the website which include greatest common divisor, narcissus number, string matching, insertion sorting, linked list inserting, breadth-first searching and others. We download the source codes and the corresponding problem IDs as our datasets and extract functions from source codes. At the same time, when we download the training sample codes, we will select the code that passed the OJ test to ensure they are

correct codes for the problems. The source codes for the same problem are considered to have similar function, and the source codes of different topics are not similar. As a result, in the dataset, 940 functions are extracted, so we get 817,216 pairs, including 38,460 similar pairs and 778,756 dissimilar pairs of code snippets totally after pruning.

4.2. Implementation and Comparisons

For WICE-SNN, word2vec, N-gram [12] models, we use TensorFlow to implement the neural networks models. We also compared WICE-SNN with the state-of-the-art approach: code2vec [37]. Code2vec is a neural model for code embedding that represents code snippets as a collection of paths and aggregates these paths into a single fixed-length code vector. In our experiment, we get the available from Github [38]. Because the already-trained model on Github is for java, so we train a new model using our preprocessed dataset.

For validating the effectiveness of WICE-SNN, we employ hold-out on the labeled source codes in the dataset, split the dataset into 3 parts: 80% for training, 10% for validating and 10% for testing. We repeat each training 10 times and the reported result is an average value over 10 times. We use python language to archive our programs, the specific version is 3.7 and adopt the popular framework Tensorflow 1.4.0 and Keras 2.2.0 to implement the deep learning module in this paper.

Training process: In our experiments, we use Word2Vec tool to map words into real vectors and count the frequency of words based on TF-IDF algorithm. The weighted matrix is the input of WICE-SNN model that used to train a classifier. We use a cross entropy method to evaluate the difference between the prediction result $\hat{y}$ and the marked value y as follows:

$$L(y, \hat{y}) = -\sum_{i=1}^{d} (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i) \tag{6}$$

where d is the size of output layer, represents the dimension of the vector y. We use the Adam optimization algorithm to update parameters. We train on a computer with a Nvidia GPU. A single training epoch takes about 40 min, and it takes about 7 h to completely train a model.

Parameters Setting: For WICE layer, we set vocabulary size N to 1024, hidden size d to 64, window size to 10. In CNN model, we have three kinds of filter, their window size are 2, 3 and 5. The number of filters is 32. We set the dropout rate to 0.5, learning rate to 0.001, mini-batch size to 32. The model is trained on the training set, and hyper-parameters are tuned for maximizing F1 score on validation set.

Evaluation Measure: We adopt the Precision, Recall and F1 measure, which are widely used in many fields. The expressions are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{9}$$

where TP represents true positive candidates, TN represents true negative candidates, FP represents false positive candidates, FN represents false negative candidates.

4.3. Experimental Results

In this paper, to investigate the performance of WICE-SNN and baseline on the task, we have carried out the experiments about 10 times on dataset, and took the average value as the experimental result. The performance results of all model are shown in Table 1. We also used word2vec to get

each word vector and compute the cosine value based on their average weights(Word2Vec) or TF-IDF weights(WICE) as the similarity value. We can find our proposed method achieves the best performance, with the improvement by up to 0.67, 0.83, and 0.74 in Precision, Recall and F1. Our method improve 0.41 in recall and 0.18 than word embedding and WICE, however, the precision has been reduced than the previous two methods. For N-gram, we set n from 1 to 5, and get the average values as the result. From the results, we can find N-gram model have almost same performance with Code2Vec one this dataset. In our experiment, we also found N-gram gets the best performance when n is 2. For Code2Vec is only designed for functions, so in the data preprocessing, the *main* function or the function that implements the main algorithm is retained, and other codes are deleted. In addition, we deleted all comments in code snippets as the astminer library can not process comments when extracting code blocks into abstract syntax trees.

Table 1. Results on Dataset1.

Model	Precision	Recall	F1-Score
Word2Vec	0.79	0.42	0.55
WICE	0.69	0.60	0.64
N-gram	0.71	0.59	0.63
Code2Vec	0.69	0.56	0.62
WICE-SNN	0.67	0.83	0.74

At the same time, in order to test the best performance of each method, we change the similarity threshold, which increases from 0 to 1 in steps of 0.1. The test results are shown in Figure 3. The abscissa in the figure shows the change of the threshold value. It can be seen from the figure that when the threshold value of the first two methods is about 0.9, the F1 performance is the best, and the third method, when the threshold value is about 0.6, the performance is the best.

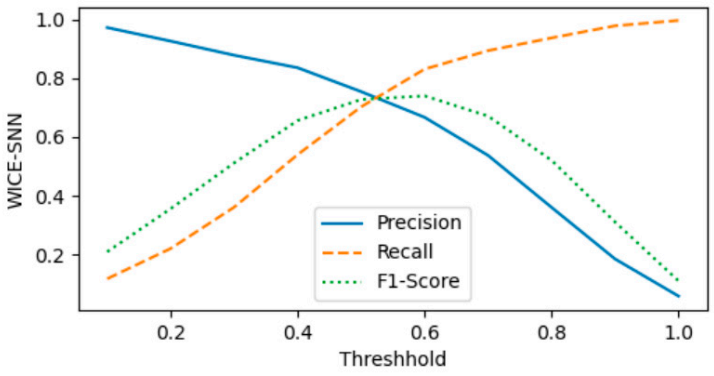

Figure 3. Performance Change with Parameter (Threshhold ∈ [0.1, 1]).

N-gram is a statistical method that needs less computing resources and it takes about 11.96 s to get results. Word2Vec and WICE belong to shallow neural networks. They take about 57 min to complete the process. Our model and code2vec are deep neural networks. Their computational cost is decided by their size of networks. They will take more time than shallow neural network. Especially, code2vec need to parse all source codes into an abstract syntax tree (AST) and extract the paths from their ASTs. So preprocessing is also a time-consuming thing. For our model, no more preprocessing is required, and the size of kernels are small. The computational cost is mainly spent on model training. It takes about 7 h to completely train a model. For code2vce, we take about 10.5 h to train a model.

5. Conclusions and Future Works

In this paper, we presented a neural network framework to measure similarities of code snippets. This framework contains three parts: embedding layer, representation layer and similarity layer.

The embedding layer integrates the frequency of word into word embedding, maps a code snippet into a real matrix. The representation layer provides a Siamese CNN model that initialized by the pre-trained matrix to mine the deep features of similar codes by training a classifier. The similarity layer computes the value of cosine similarity. We evaluate our approach to a dataset. In contrast with previous approaches, the performance of our proposed method is better than the general word embedding, than some statistical-based methods.

This work can be a foundation for improving many other applications of codes, such as bug detection, code recommendation, and code search. To serve this purpose, we will design more downstream tasks to optimize our model. Moreover, the different types of words in source code, such as reserved words, user-defined words and operators are not considered in our method. Therefore, in the future, we will further refine the word types to verify the experimental results.

Author Contributions: Conceptualization, methodology, C.X.; code, C.Q.; formal analysis, X.W.; Data Preprocessing, M.W.; Writing—Original draft preparation, C.X.; Writing—Review and editing, C.X., X.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (61502212,61773185, 61877030), Innovation and Entrepreneurship Training Program for College Students of Jiangsu Province (201910320134) and Cooperative Education Project of the Ministry of Education.

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

Appendix A

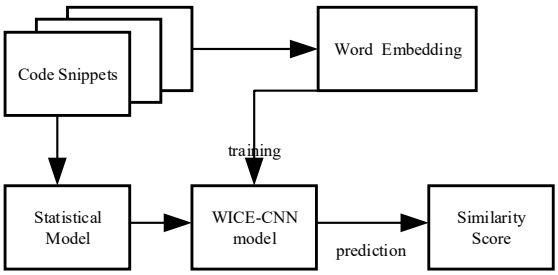

Figure A1. The flowchart of our work.

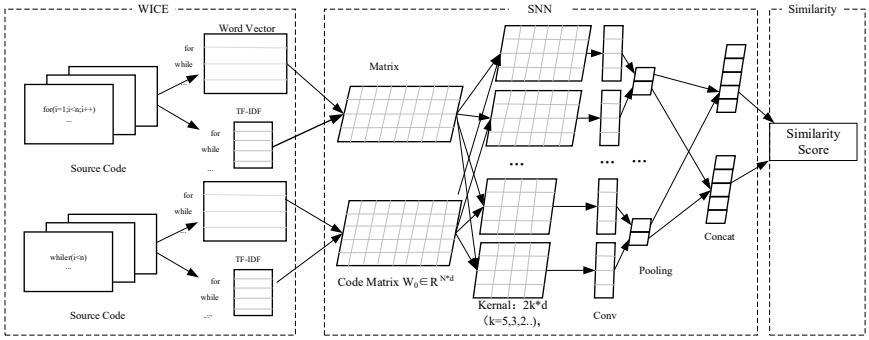

Figure A2. WICE-SNN Framework.

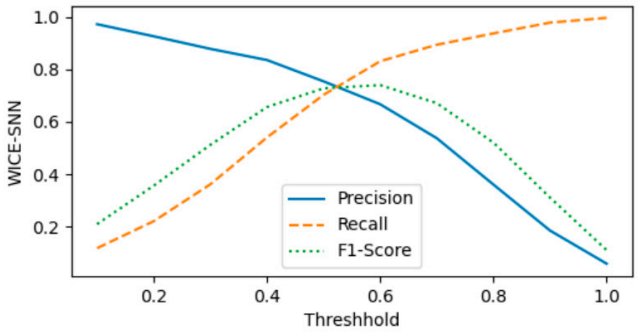

Figure A3. Performance Change with Parameter (Threshhold ∈ [0.1, 1]).

Table A1. Results on Dataset1.

Model	Precision	Recall	F1-Score
Word2Vec	0.79	0.42	0.55
WICE	0.69	0.60	0.64
N-gram	0.71	0.59	0.63
Code2Vec	0.69	0.56	0.62
WICE-SNN	0.67	0.83	0.74

References

1. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
2. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
3. Liu, C.; Chen, C.; Han, J.; Yu, P.S. GPLAG: Detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; Association for Computing Machinery ACM: New York, NY, USA, 2006; pp. 872–881. [CrossRef]
4. Cosma, G.; Joy, M. Towards a definition of source-code plagiarism. *IEEE Trans. Educ.* **2008**, *51*, 195–200. [CrossRef]
5. Cosma, G.; Joy, M. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Comput.* **2011**, *61*, 379–394. [CrossRef]
6. Mens, K.; Lozano, A. *Source Code-Based Recommendation Systems, Recommendation Systems in Software Engineering*; Springer Science and Business Media LLC, Springer: Berlin/Heidelberg, Germany, 2014; pp. 93–130.
7. Mcmillan, C.; Poshyvanyk, D.; Grechanik, M.; Xie, Q.; Chen, F. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.* **2013**, *22*, 1–30. [CrossRef]
8. Ragkhitwetsagul, C.; Krinke, J.; Clark, D. A comparison of code similarity analysers. *Empir. Softw. Eng.* **2017**, *23*, 2464–2519. [CrossRef]
9. Roy, C.K.; Cordy, J.R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In Proceedings of the 16th IEEE International Conference on Program Comprehension, Amsterdam, The Netherlands, 10–13 June 2008; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2008; Volume 2008, pp. 172–181. [CrossRef]
10. Baxter, I.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the International Conference on Software Maintenance, ICSM 2003, Amsterdam, The Netherlands, 22–26 September 2003; Institute of Electrical and Electronics Engineers, IEEE: Piscataway, NJ, USA, 2002; Volume 2006, pp. 368–377. [CrossRef]
11. Chae, D.K.; Ha, J.; Kim, S.W.; Kang, B.J.; Im, E.G. Software plagiarism detection: A graph-based approach. In Proceedings of the 22nd ACM International Conference on Information and Knowledge Management. CIKM 2013, San Francisco, CA, USA, 27 October–1 November 2013; Association for Computing Machinery ACM: New York, NY, USA, 2013; pp. 1577–1580. [CrossRef]

12. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On the naturalness of software. In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2012; Volume 2012, pp. 837–847. [CrossRef]
13. Karaivanov, S.; Raychev, V.; Vechev, M. Phrase-Based statistical translation of programming languages. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software-Onward! '14; ACM 2014, Portland, OR, USA, 20–24 October 2014; Association for Computing Machinery ACM: New York, NY, USA, 2014; Volume 2014, pp. 173–184. [CrossRef]
14. Raychev, V.; Vechev, M.; Yahav, E. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; Association for Computing Machinery ACM: New York, NY, USA, 2014; Volume 49, pp. 419–428. [CrossRef]
15. Nguyen, A.T.; Nguyen, T.T.; Nguyen, T.N. Divide-and-Conquer approach for multi-phase statistical migration for source code (T). In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2015; pp. 585–596. [CrossRef]
16. Zhang, F.Y.; Peng, X.; Chen, C.; Zhao, W.Y. Research on Code Analysis based on Deep Learning. *Comput. Appl. Softw.* **2018**, *35*, 15–23. (In Chinese)
17. Chen, Q.Y.; Li, S.P.; Yan, M.; Xia, X. Code clone detection: A literature review. *J. Softw.* **2019**, *30*, 962–980. (In Chinese)
18. Tufano, M.; Watson, C.; Gabriele, B.; Di, P.; White, M.; Poshyvanyk, D. Deep learning similarities from different representations of source code. In Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, 28–29 May 2018; pp. 542–553. [CrossRef]
19. Hellendoorn, V.J.; Devanbu, P. Are deep neural networks the best choice for modeling source code? In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017; Association for Computing Machinery, ACM: New York, NY, USA, 2017; pp. 763–773. [CrossRef]
20. Halstead, M.H. *Elements of Software Science*; Elsevier North-Holland: New York, NY, USA, 1977.
21. Komondoor, R.; Horwitz, S. Using slicing to identify duplication in source code. In *International Static Analysis Symposium, Lecture Notes in Computer Science*; Springer Science and Business Media LLC, Springer: Berlin/Heidelberg, Germany, 2001; pp. 40–56.
22. Ignacio, A.F.; Carlos Francisco, M.; Gerardo, S.; Juan Manuel, T.M.; Grigori, S. Unsupervised Sentence Representations as Word Information Series: Revisiting TF-IDF. *Comput. Speech Lang.* **2019**, *56*, 107–129. [CrossRef]
23. He, X.F.; Ai, J.L.; Song, Z.T. Multi-Source data fusion for health monitoring of unmanned aerial vehicle structures. *Appl. Math. Mech.* **2018**, *39*, 395–402. (In Chinese)
24. Nguyen, A.T.; Nguyen, T.D.; Phan, H.D.; Nguyen, T.N. A deep neural network language model with contexts for source code. In Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2018; pp. 323–334. [CrossRef]
25. Ottenstein, K.J. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bull.* **1976**, *8*, 30–41. [CrossRef]
26. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, 3–7 September 2016; Association for Computing Machinery ACM: New York, NY, USA, 2016; pp. 87–98. [CrossRef]
27. Lam, A.N.; Nguyen, A.T.; Nguyen, H.A.; Nguyen, T.N. Combining deep learning with information retrieval to localize buggy files for bug reports (N). In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2015; Volume 2015, pp. 476–481. [CrossRef]
28. Huo, X.; Thung, F.; Li, M.; Lo, D.; Shi, S.-T. Deep Transfer Bug Localization. *IEEE Trans. Softw. Eng.* **2019**, *99*, 1. [CrossRef]
29. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed representations of words and phrases and their compositionality. In Proceedings of the 26th International Conference on Neural Information

Processing Systems (NIPS), Lake Tahoe, CA, USA, 5–10 December 2013; Curran Associates Inc.: Red Hook, NY, USA, 2013; Volume 26, pp. 3111–3119.

30. Ye, X.; Shen, H.; Ma, X.; Bunescu, R.; Liu, C. From word embeddings to document similarities for improved information retrieval in software engineering. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Austin, TX, USA, 14–22 May 2016; Association for Computing Machinery ACM: New York, NY, USA, 2016; pp. 404–415. [CrossRef]
31. Nguyen, T.D.; Nguyen, A.T.; Phan, H.D.; Nguyen, T.N. Exploring API embedding for API usages and applications. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2017; Volume 2017, pp. 438–449. [CrossRef]
32. Chen, C.; Xing, Z.; Wang, X. Unsupervised software-specific morphological forms inference from informal discussions. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2017; pp. 450–461. [CrossRef]
33. Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; Jin, Z. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*; Lecture Notes in Computer Science; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2015; Volume 9403, pp. 547–553. [CrossRef]
34. Mou, L.; Li, G.; Jin, Z.; Zhang, L.; Wang, T. Convolutional neural network over tree structure for programming language processing. In Proceedings of the 30th AAAI Conference on Artifical Intelligence, Menlo Park, CA, USA, 12–17 February 2016; pp. 1287–1293. [CrossRef]
35. White, M.; Tufano, M.; Martinez, M.; Monperrus, M.; Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; Institute of Electrical and Electronics Engineers IEEE: Piscataway, NJ, USA, 2019; pp. 479–490. [CrossRef]
36. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering-ICSE '16, Austin, TX, USA, 14–22 May 2016; Association for Computing Machinery ACM: New York, NY, USA, 2016; pp. 297–308. [CrossRef]
37. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. Code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*; Association for Computing Machinery ACM: New York, NY, USA, 2019; Volume 3, pp. 1–29. [CrossRef]
38. Github. Available online: https://github.com/tech-srl/code2vec (accessed on 1 November 2019).