A Deep Neural Network-Based Approach to Finding Similar Code Segments

Dong Kwan KIM(a), Member

SUMMARY This paper presents a Siamese architecture model with two identical Convolutional Neural Networks (CNNs) to identify code clones; two code fragments are represented as Abstract Syntax Trees (ASTs). CNN-based subnetworks extract feature vectors from the ASTs of pairwise code fragments, and the output layer produces how similar or dissimilar they are. Experimental results demonstrate that CNN-based feature extraction is effective in detecting code clones at source code or bytecode levels.

key words: code clone detection, Siamese architecture, convolutional neural network, abstract syntax tree (AST)

1. Introduction

Cloning code may occur intentionally or unintentionally during software development or maintenance. Code clones have been recognized as bad coding practices in a program [1]; they may cause inconsistent changes in a program and increase maintenance costs.

There are four clone types: Type-1 clones are identical code fragments except for variations in whitespace, layout and comments. Type-2 clones are syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments. Type-3 clones are copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments. Type-4 clones are two or more code fragments that perform the same computation but are implemented by different syntactic variants. It is challenging to detect non-trivial code clones such as Type-3 and Type-4 clones. The proposed approach to clone detection aims to find Type-3 and Type-4 clones as well as Type-1 and Type-2 clones. This paper presents a CNN-based Siamese architecture model for detecting code clones. Pairwise code fragments are given to the Siamese model. The two CNN subnetworks build feature vectors from ASTs of these code fragments. The detection model finds clones by comparing these feature vectors. The experimental results indicate that the proposed approach to CNN-based clone detection is beneficial in finding semantic clones as well as syntactic clones.

This paper makes the following contributions:

• A CNN-based clone detector that automatically constructs semantic features from ASTs of code segments.
• A clone detector for Java bytecode that is compiled into Java source code before being fed to the deep neural network.
• An experimental study that evaluates the applicability and utility of the Siamese neural network-based approach to finding code clones.

The rest of this paper is organized as follows. Section 2 introduces related work of clone detection methods and Sect. 3 presents the proposed Siamese neural network-based clone detection. Section 4 shows experimental results and Sect. 5 remarks the conclusion.

2. Related Work

Wei and Li [2] propose a supervised learning model to detect functional code clones. Their model consists of AST-based LSTM networks for feature representations of code fragments and hash functions for similarity comparison. White et al. [3] extract deep feature representations using autoencoder, a type of unsupervised learning, with recurrent neural networks to detect code clones; their approach represents terms in code fragments as continuous-valued vectors and links code patterns at the lexical level with code patterns at the syntactic level. Saini et al. [4] present a learning-based approach to find harder-to-detect semantic clones in an effective way. They use a deep neural network with Siamese architecture consisting with two identical subnetworks, a comparator network, and a classification unit. Mou et al. [5] enhance tree-based convolutional neural networks to process syntactic parse trees of programming languages; a convolution kernel extracts structural features of programs from abstract syntax trees. Bui et al. [6] introduce bilateral tree-based convolutional neural networks for program classification; these networks are constructed by adapting existing tree-based convolutional neural networks and encode AST structures of programming languages. Ragkhitwetsagul et al. [7] propose the Siamese clone search engine which is scalable and incremental, suitable for performing instant clone search on large-scale data sets. The Siamese is an acronym for Scalable, incremental, and multi-representation and has nothing to do with deep learning models. The Siamese tool has not found satisfactory results in searching for Type-4 clones.
3. Proposed Deep Learning Model

Clone detection techniques need to measure the similarity of two code fragments. Siamese neural networks can be used to find similar code fragments. Two identical subnetworks in a Siamese architecture process two code fragments, another neuron nodes take their outputs, and final outputs are produced in an output layer. Figure 1 depicts the proposed CNN-based Siamese architecture. The Siamese neural network is configured with two identical CNN subnetworks with the same weights. The proposed clone detection framework takes method pairs which are extracted from Java source files. Each CNN subnetwork takes one of the method pairs in parallel and computes its feature vectors for clone detection. In this neural network model, a contrastive loss algorithm is used as a loss function to measure the absolute difference between a predictive value and an expected value on a training data instance. A contrastive loss function is a distance-based loss function and is calculated on input pairs. The contrastive loss function will try to improve the Siamese network model for clone detection. The closer the prediction value of the model is to the actual value, the smaller the difference between the two values is. In other words, if the model works well on a training dataset, the loss function will output a lower value. The proposed framework determines if pairwise methods are clones by inspecting the output value of the contrastive loss function. Figure 2 shows the overall steps of the CNN-driven program feature extraction. The input of the code detection framework is code fragments which are continuous segments of source code. A code fragment can be a function or a method. The first step of the proposed clone detection framework is to extract all methods from source files. To remove noisy data, we intentionally filtered out methods less than six LOC from a training dataset.

The extracted methods are represented as ASTs which are tree representations of the syntactic structure of source code. Each AST node denotes a programming language construct in the source code such as for-statements and if-statements. The proposed clone detection framework extracts features from the ASTs to compare two methods. The ASTs are represented as embedding vectors which are real-valued vector representations for the CNN subnetworks in the Siamese architecture. The CNN subnetwork produces the feature representation vector of a method via convolution and pooling layers.

In the convolution layer, convolution operations are performed on the vector representations of a method AST by applying dot products between a feature extractor and the local regions of the AST. The convolutional layer uses ReLU activation function. After the convolution task has been performed, a set of feature maps are generated and they contain structural features in an AST. Some pooling layers can be applied to aggregate structural features along different parts of the AST. A max pooling operation is used to take the maximum value of a sub-region of the convolutional vectors. The pooled results of two ASTs are passed to the weighted L1 distance algorithm in order to measure a distance between a pair of feature vectors. If the measured distance value is closer than a predefined threshold, the two ASTs are detected as clones.

Table 1 lists selected AST nodes which represent abstract constructs in the Java programming language.

| No. | Category | Tree Nodes |
|-----|----------|------------|
| 1   | Loop     | ForStatement, WhileStatement, DoStatement, EnhancedForControl, ForControl |
| 2   | Condition | IfStatement, SwitchStatement, SwitchStatementCase |
| 3   | Declaration| PackageDeclaration, InterfaceDeclaration, ClassDeclaration, ConstructorDeclaration, MethodDeclaration, FieldDeclaration, LocalVariableDeclaration, VariableDeclaration |
| 4   | Control   | BreakStatement, ContinueStatement |
| 5   | Exception | ThrowStatement, TryResource, TryStatement, CatchClause |
| 6   | Method    | MethodInvocation, SuperMethodInvocation |

Fig. 1 The overview of the proposed Siamese architecture model for clone detection.

Fig. 2 The overview of the CNN-driven program feature extraction.
These AST nodes can be categorized as six types of nodes: loop, condition, declaration, control, exception, and method categories. The proposed framework uses Eclipse ASTParser to parse Java source code into ASTs. The names of the tree nodes shown in Table 1 are the same as those used in the JDT library.

4. Experimental Results

BigCloneBench [8], a popular clone detection benchmark, is used as training and testing dataset to evaluate the proposed approach to clone detection. The benchmark contains the four primary clone types (Type-1 ~ Type-4) including syntactically and semantically code clones. The performance of the proposed approach to clone detection will be evaluated with all clone types. BigCloneBench covers implementations of 10 functionalities where correspond to 10 source code folders as #2 ~ #11—#2: Web download, #3: Secure hash (MD5), #4: Copy a file, #5: Decompress Zip, #6: FTP authenticated login, #7: Bubble sort, #8: Init. SGV with model, #9: SGV selection event handler, #10: Create Java project (Eclipse), and #11: SQL update and rollback. Folder #4 contains the largest number of known true clone pairs and false clone pairs and is used to train the proposed deep learning models. Five folders are used for the testing dataset—folders #2, #3, #6, #7, and #10.

The proposed Siamese network model takes a pair of Java methods as input and the features of each Java method are represented as embedding vectors through the CNN sub-network. The deep learning model for clone detection uses training dataset with both true clone and false clone pairs and testing dataset with unknown method pairs. In a training stage, a pair of methods can be extracted from a training dataset and they are represented with a set of features and a corresponding label which denotes a clone or a non-clone; true clones are labelled to ‘1’ while false clones are labelled to ‘0’. After constructing the deep learning model, a testing dataset is used to evaluate the performance of the trained model with unknown method pairs. The clone detection framework is implemented in TensorFlow and Keras which are an open source machine learning library for deep learning. All the measurements were performed on a server computer with an Intel Xeon W-2123 (3.9 GHz) processor, an NVIDIA RTX 2080 GPU, and 32 GB of RAM, running the 64-bit version of Windows 10.

BigCloneBench classifies Type-3 and Type4 clones into four types: Very-Strongly Type-3 (VST3), Strongly Type-3 (ST3), Moderately Type-3 (MT3), and Weakly Type-3 or Type-4 (WT3/4). The degree of syntactical similarity is as follows: VST3 > ST3 > MT3 > WT3/4. Table 2 shows the experimental results on the dataset of BigCloneBench. The dataset is split with a training data and a testing data. The training dataset includes 35,246 true clones and 12,023 false clones. These true clones include T1, T2, VST3, ST3, MT3, and WT3/4 clone types. The testing dataset includes 969,802 true clones and 1,158 false clones. The false clone pairs with different methods were created and manually verified by the author. Table 3 summarizes the parameters used in the proposed CNN models for the experiments. The pooling type of the proposed model is max pooling. The Keras library provides four built-in pooling types: MaxPooling, AveragePooling, GlobalMaxPooling, and GlobalAveragePooling. The proposed DL model was evaluated with all pooling types and the pooling type MaxPooling showed the best performance. In case of AveragePooling, the performance of the model was not satisfactory.

The detection performance of the proposed approach is compared with machine learning algorithms. Various traditional machine learning algorithms have been used in detecting code clones. The code features are extracted from source code and are given to machine learning algorithms for classification tasks. In terms of feature selection, various code features can be identified to characterize code clones by analyzing code text, abstract syntax trees, control flow graphs, and program dependency graphs. On the basis of code features, machine learning algorithms are used in detection model training and prediction with the selected code features. The output of the machine learning algorithms would be the similarity ratio with which the two code segments are related to each other. The proposed detection model has been compared with the three machine learning algorithms such as Support Vector Machine (SVM), Logistic Regression (LR), and Decision Tree (DT). The Python’s Scikit-learn library was used to implement these algorithms as clone classifiers. Table 4 shows the accuracy comparison between the proposed DNN and traditional ML algorithms (%).

Table 2 Datasets used in the experiments.

| Dataset | True Clones | False Clones |
|---------|-------------|--------------|
| T1      | 13,667      | 3,059        |
| T2      | 3,059       | 1,135        |
| VST3    | 1,135       | 3,826        |
| ST3     | 3,826       | 6,969        |
| MT3     | 6,969       | 6,590        |
| WT3/4   | 6,590       | 12,023       |

Table 3 Parameters setting of CNNs.

| Parameters | Values | Parameters | Values |
|-----------|--------|------------|--------|
| # nodes (Hidden layers) | 100 | Activation (output layer) | Sigmoid |
| Activation (other layers) | ReLU | Optimization algorithm | Adam |
| Loss function | Cross Entropy | Learning rate | 0.00002 |
| Pooling type | Max pooling | Batch size | 32 |

Table 4 Accuracy comparison between the proposed DNN and traditional ML algorithms (%).

| Proposed | T1 | T2 | VST3 | ST3 | MT3 | WT3/4 |
|----------|----|----|------|-----|-----|-------|
| SVM      | 99 | 83 | 87   | 54  | 2   | 0     |
| LR       | 100| 99 | 97   | 81  | 20  | 2     |
| DT       | 100| 93 | 91   | 71  | 15  | 1     |

that it is very challenging for the proposed detector to identify syntactically different but semantically equivalent code snippets. The three machine learning algorithms perform well in classifying T1, T2, and VST3 clone types. LR performs better than the two algorithms, SVM and DT over all clone types. In particular, the LR algorithm on the ST3 clones shows a higher accuracy of 81% than those of SVM and DT. The performance results of the three machine learning algorithms drop rapidly over MT3 and WT3/4 clones. The experimental results demonstrate the proposed detection approach outperforms the three machine learning algorithms over all clone types. The accuracy of the proposed clone classifier on the false clone pairs is approximately 96% which means the false positive rate of the classifier is 4%.

NiCad is selected to demonstrate the effectiveness of the proposed clone detector. NiCad is one of the state-of-the-art clone detectors in detecting near-miss intentional clones. In the experiment, it is measured among all clone pairs detected by NiCad, how many of them are identified by the proposed clone detector. Eight Java projects are used for this experiment. The comparison of NiCad is parameterized by a difference threshold that allows for near-miss detection. In Table 5, a threshold (Th) of 0.0 detects only exact clones. 0.1 detects clone pairs that may differ by up to 10% of their normalized lines, 0.2 by up to 20%. Table 5 shows the number of clone pairs found by NiCad for each Java Project. The detection ratio of the proposed clone detector is also shown as a percentage. For example, in the case of the Log4J Java project, NiCad found 3, 11, and 242 clone pairs according to the threshold and the proposed clone detector identified all of them without any missing clone pairs. The findings of this experiment demonstrate that the performance of the proposed clone detector is not much behind that of NiCad.

The qualitative evaluation on some of the experimental results was manually conducted by the author. It might be time-consuming to manually examine all reported clone pairs of the proposed model. In general, the performance of the clone detection tool would be poor when it tries to find semantic clones such as Type-3 and Type-4 clones. For a qualitative evaluation, some data of the reported semantic clones were sampled and manual inspections were performed by the author. In the future, external developers will be involved to ensure the objectivity of the manual inspection. Furthermore, to evaluate the applicability of the proposed approach, the Siamese model was compared with the NiCad tool, one of the state-of-the-art clone detection tools. Table 5 shows the clone detection performance of the proposed approach is almost the same as that of the NiCad tool on eight Java projects. To properly evaluate the precision of the proposed model and NiCad, each tool’s reported clone pairs were randomly sampled. The sampled method pairs were manually examined to check whether they were true clones.

5. Conclusions

This paper presents a practical approach to clone detection using a CNN-based Siamese architecture model. The CNN subnetworks build feature vectors from ASTs and then the feature vectors are compared to determine whether the two code fragments are similar. The proposed detection approach has been compared with three popular machine learning algorithms and NiCad in terms of the accuracy of clone detection. The overall experimental results demonstrate that the proposed clone detection methodology is feasible even though there is room for improvement in finding semantic clone pairs.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2019R1I1A3A01056368) and Mokpo National Maritime University in 2018.

References

[1] C.J. Kapser, Toward an Understanding of Software Code Cloning as a Development Practice, Ph.D. thesis, University of Waterloo, 2009.
[2] H. Wei and M. Li, “Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code,” Proc. Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), pp.3034–3040, 2017.
[3] M. White, M. Tufano, C. Vendome, and D. Podhyvanyak, “Deep learning code fragments for code clone detection,” Proc. 31st IEEE/ACM International Conference on Automated Software Engineering, pp.87–98, 2016.
[4] V. Saini, F. Farnahinifarahan, Y. Lu, P. Baldi, and C. Lopes, “Oreo: Detection of Clones in the Twilight Zone,” The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp.354–365, 2018.
[5] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” Proc. Thirtieth AAAI Conference on Artificial Intelligence, pp.1287–1293, 2016.
[6] N. Bui, L. Jiang, and Y. Yu, “Cross-language learning for program classification using bilateral tree-based convolutional neural networks,” NL4SE Workshop, AAAI’18, 2018.
[7] C. Ragkhitwetsagul and J. Krinke, “Siamese: scalable and incremental code clone search via multiple code representations,” Empirical Software Engineering, vol. 24, no. 4, pp. 2236–2284, 2019.

[8] J. Svajlenko, J.F. Islam, I. Keivanloo, C.K. Roy, and M.M. Mia, “Towards a big data curated benchmark of inter-project code clones,” 30th IEEE International Conference on Software Maintenance and Evolution, Canada, 2014.