International Journal of Electrical and Computer Engineering (IJECE)
Vol. 9, No. 5, October 2019, pp. 3804~3812
ISSN: 2088-8708, DOI: 10.11591/ijece.v9i5.pp3804-3812

Enhancing code clone detection using control flow graphs

Dong Kwan Kim
Department of Computer Engineering, Mokpo National Maritime University, Korea

Article Info

Article history:

Received Mar 6, 2019
Revised Apr 7, 2019
Accepted Apr 19, 2019

Keywords:

Clone detection
Clone types
Control flow graph
Deep learning

ABSTRACT

Code clones are syntactically or semantically equivalent code fragments of source code. Copy-and-paste programming allows software developers to improve development productivity, but it could produce code clones that can introduce non-trivial difficulties in software maintenance. In this paper, a code clone detection framework is presented with a feature extractor and a clone classifier using deep learning. The clone classifier is trained with true and false clones and then is tested with a test dataset to evaluate the performance of the proposed approach to clone detection. In particular, the proposed approach to clone detection uses Control Flow Graphs (CFGs) to extract features of a given code snippet. The selected features are used to compute similarity scores for comparing two code fragments. The clone classifier is trained and tested with similarity scores that quantify the degree of how similar two code fragments are. The experimental results demonstrate that using CFG features is a viable methodology in terms of the effectiveness of clone detection for both syntactic and semantic clones.

Corresponding Author:

Dong Kwan Kim,
Department of Computer Engineering,
Mokpo National Maritime University,
91, Haeyangdaehak-ro, Mokpo-si, Jeollanam-do, Korea.
Email: dongkwan@gmail.com

1. INTRODUCTION

Software reuse refers to a series of activities of using existing code for developing new software products. Software reuse has positive aspects of improving development productivity and quality, but simple copy-and-paste reuse can produce redundant and duplicate code (also known as code clones) across a program. Code clones can be defined as syntactically or semantically equivalent code fragments of source code. The presence of the code clone can hinder the consistency of an application in software maintenance such as bug fixes, security updates, and code refactoring. Such code changes can be involved in undesirable consequences due to the code clone. If only part of the code clone is modified without covering the whole of the code clone, potential problems can be introduced during the lifecycle of a software system. Therefore, for better software maintenance, locating and keeping track of code clones are a crucial part when modifying code of an existing program. In this paper, a novel approach to clone detection is presented with a feature extractor and a clone classifier using deep learning. The feature extractor constructs feature vectors to characterize a give code fragment for clone detection. The clone classifier is based on supervised learning which produces a deep learning model through training data consisting of input vectors and desired output values. Therefore, the clone classifier is trained with known true clones and false clones in a training phase. In a testing phase, the clone classifier predicts whether or not two method pairs have a clone relationship. The proposed approach to clone detection extracts input features from the CFGs of a code fragment. The input features are used to compute similarity scores for comparing two code fragments. The clone classifier is trained and tested with the similarity scores that quantify the degree of how similar two code fragments are.

The proposed clone detection framework represents a code fragment as control flow graphs to catch semantically similar clones. A control flow graph is a directed graph that represents the control flow of a code fragment (e.g., functions and methods). The CFG consists of a set of nodes (also known as vertices) and edges. A node represents the basic statement of the code fragment such as expressions, if-statements, and for-statements. An edge of the CFG connects one node with another and means a program control flow from between the two nodes. The node could be connected to multiple nodes if it is involved in controlling more than one program control. The CFG has a single entry node where a control flow starts and a single exit node where a control flow ends. There can be multiple control flows between the entry node and the exit node. A path in a CFG is a sequence of directed edges in which all nodes are distinct. For example, 3_Path in this paper means any three distinct nodes are connected in a CFG-based representation. The path represents structural features in code fragments and its length is various in a CFG-based representation. The code fragments are characterized by the length and the occurrence counts of the CFG path.

The first step of clone detection is to determine the scope of a code fragment which is a continuous segment of source code. Some clone detection tools take a function or a method as the code fragment. In some cases, a portion of the function can be considered as the code fragment. The code fragment is identified with three elements such as a source file, a starting line number, and an ending line number. If a whole method is considered as the code fragment, the starting line number and the ending line number of the code fragment will be the same as those of the method. The proposed clone detection approach takes each method in Java source files as the code fragment. Code clones are a pair of code fragments of source code that are syntactically or semantically equivalent. There are four types of code clones: Type-1 (T1), Type-2 (T2), Type-3 (T3), and Type-4 (T4). Type-1 clones are syntactically identical code fragments, except for differences in white space, layout and comments [1]. Type-2 clones are syntactically identical code fragments, except for differences in identifier names, literal values, white space, layout and comments [1]. Type-3 clones are syntactically similar code fragments that differ at the statement level. Fragments have statements added, modified and/or removed with respect to each other [1]. Finally, Type-4 clones are syntactically dissimilar code fragments that implement the same functionality [2]. The proposed approach to clone detection uses BigCloneBench [3] which is one of the popular benchmarks of code clones. The BigCloneBench data are categorized by separating Type-3 and Type4 clone pairs into four categories based on their syntactical similarity: Very-Strongly Type-3 (VST3) clones with a similarity in range 90% (inclusive) to 100%, Strongly Type-3 (ST3): 70-90%, Moderately Type-3 (MT3): 50-70%, and Weakly Type-3 or Type-4 (WT3/4): 0-50% [4].

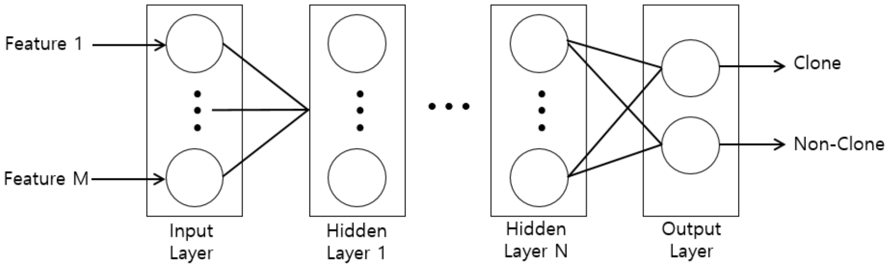

Figure 1. The deep neural network architecture

Deep learning is a specific kind of machine learning and allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction [5]. Deep learning algorithms have been applied to solve problems of artificial intelligence fields and have produced promising results for some specific tasks such as natural language processing, computer vision, and image processing [6]. These advanced learning algorithms can also contribute to solving software engineering problems that are the analogous ones in artificial intelligence. Figure 1 shows the deep neural network architecture that consists of three layers-input layer, hidden layer, and output layer. The input features (e.g., similarity scores) are given to input nodes in the input layer. The nodes in the different layers are connected with connection weights and biases. The deep learning network model can possess multiple hidden layers so to improve the performance of the code clone classifier. In the clone classifier model, the Rectified Linear Unit (ReLU) is an activation function for the hidden layers and the softmax function is used as activation function for the output layer. As two nodes reside in the output layer, the prediction result of the clone classifier is either “Clone” or “Non-Clone” on the given method pairs.

The rest of this paper is organized as follows. Section 2 overall describes the proposed approach to detect code clones using control flow graphs. Section 3 presents experimental results to demonstrate the effectiveness of the CFG-based clone detection. Section 4 summarizes related studies and Section 5 finally remarks conclusions and future research directions.

2. APPROACH

Figure 2 illustrates the overall workflow of the proposed code clone detection framework. The clone detection system has the training and testing phase. The solid arrows show the flow of the training phase which produces a trained clone classifier from a list of features. The clone classifier determines if pairwise methods are clones to each other in the testing phase which is shown the dashed arrows in Figure 2.

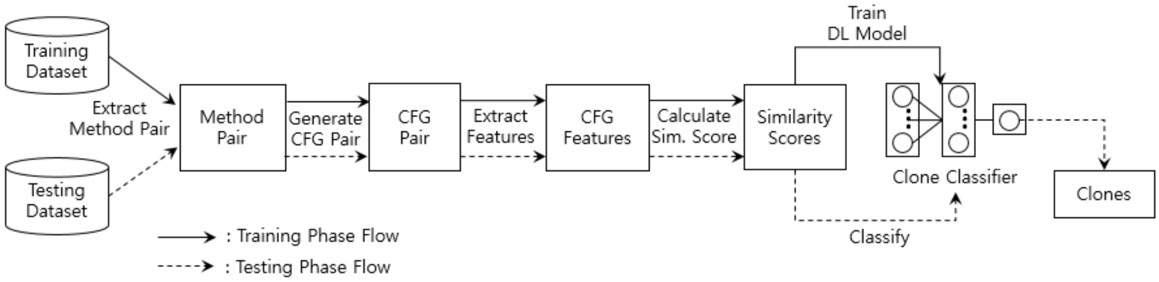

Figure 2. The overall flow of the proposed code clone detection framework

The clone detection system used BigCloneBench which is one of the popular benchmarks of code clones and has been used for clone-related studies. The BigCloneBench data are separated into the training and test datasets. The clone classifier model in the detection system is trained and optimized by the training dataset and then is evaluated by the test dataset. The first step of the training phase is to identify method modules from the given Java source files through lexical analysis and syntactic analysis. Pairwise method modules are constructed from the separate methods. The pairwise methods are used to train or test the deep learning model. The proposed clone detection system extracts syntactic and semantic features of code fragments from CFGs of the identified methods. A CFG is generated for each method. CFG features are generated from the CFG. The CFG feature sets are represented by feature vectors that could be computed effectively in the deep learning model. Similarity scores are used to quantify the degree of how similar two methods are. The similarity score of the pairwise method is calculated and is given to the input layer of the clone classifier. The range of similarity score is [0, 1] through the normalization step. The trained clone classifier can be produced after the training phase. In the testing phase, the similarity scores of the pairwise methods are created from the test dataset. The steps for generating the similarity score are the same as those of the training phase. The clone classifier can tell whether the given method pairs are clones or not using the similarity scores.

The proposed clone detection framework is based on a deep learning network model which is a clone classifier and determines the semantic similarity of the method using its corresponding control flow graph. A CFG is a directed graph and represents all possible execution paths of a method. It also encodes the behavior of the method at a higher level of abstraction. A set of features are extracted from the control flow graphs. Such a feature set includes node information and path information of the control flow graph. For the comparison of two methods, the pairwise feature sets are used to compute the similarity score. Figure 3 shows an example of Java source code and its corresponding control flow graph. In this example, the class Sample has only one method main and the control flow graph of the main method is presented on the right side of Figure 3. The node of the CFG is known as a basic block and represents expressions, if-statements, for-statements, etc. The edge of the CFG represents a possible control flow from the end of one block to the beginning of the other. As seen in Figure 3, CFGs also have a single entry and a single exit point.

```
public class Sample {
  public static String VERSION = "2.0.0";
  public static void main(String[] args) {
System.out.printf("PROGEX (Program Graph
Extractor)  [ ver. %s ]\n", VERSION);
System.out.println("Visit project website @
https://github.com/ghaffarian/progex\n");

if (args.length == 0) {
     CLI.printHelp(null);
     new GUI().show();
} else {
    new CLI().parse(args).execute();
      }
   }
}
```

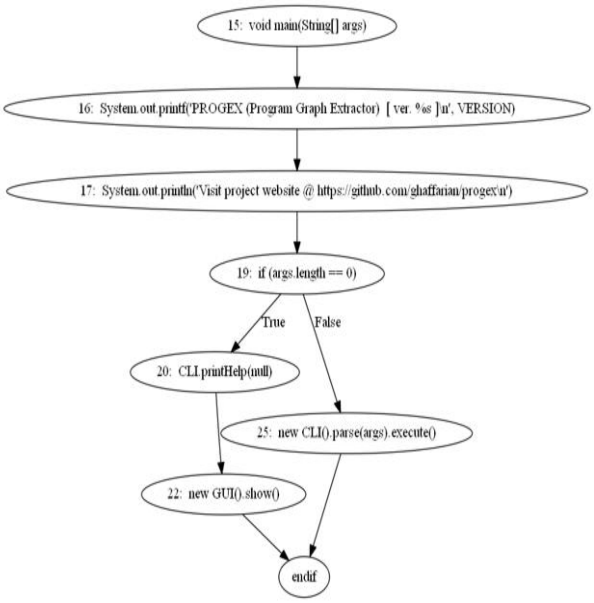
15: void main(String[] args)
16: System.out.printf('PROGEX (Program Graph Extractor) [ver. %s]\n', VERSION)
17: System.out.println('Visit project website @ https://github.com/ghaffarian/progex\n')
19: if (args.length == 0)
True
False
20: CLI.printHelp(null)
25: new CLI().parse(args).execute()
22: new GUI().show()
endif

Figure 3. Example of control flow graph

A path in a CFG is a sequence of directed edges which connect a sequence of distinct nodes. For example, 2_Path means any two distinct nodes are connected in the control flow graph. The proposed clone detection framework considers 1_Path, 2_Path, 3_Path, and 4_Path to compare pairwise methods. Table 1 lists the feature paths which are extracted from the control flow graph shown in Figure 3. The first column shows the type of paths and the second column is the total frequency of each path type in the control flow graph. The last column shows the distinct instances of each path type and their frequency. The summation of the frequency of each path instance should be equal to the total number in the second column.

Table 1. Example of feature paths extracted from a control flow graph

Type	Total	Path/Frequency
1_Path	8	[startMethod/1], [StatementExpressionContext/5], [IfStatementContext/1], [endif/1]
2_Path	8	[startMethod- StatementExpressionContext/1], [StatementExpressionContext- StatementExpressionContext/2], [StatementExpressionContext – IfStatementContext/1], [IfStatementContext-StatementExpressionContext/2], [StatementExpressionContext- endif/2]
3_Path	7	[startMethod -StatementExpressionContext-StatementExpressionContext/1], [StatementExpressionContext-StatementExpressionContext-IfStatementContext/1], [StatementExpressionContext-IfStatementContext-StatementExpressionContext/2], [IfStatementContext-StatementExpressionContext-StatementExpressionContext/1], [IfStatementContext-StatementExpressionContext-endif/1], [StatementExpressionContext-StatementExpressionContext-endif/1]
4_Path	6	[startMethod -StatementExpressionContext-StatementExpressionContext-IfStatementContext/1], [StatementExpressionContext-StatementExpressionContext-IfStatementContext-StatementExpressionContext/2], [StatementExpressionContext-IfStatementContext-StatementExpressionContext-StatementExpressionContext/1], [StatementExpressionContext-IfStatementContext-StatementExpressionContext-endif/1], [IfStatementContext-StatementExpressionContext-StatementExpressionContext-endif/1]

3. EVALUATION

A set of features can impact seriously the performance of the clone classifier. Therefore, extracting the meaningful features of a dataset is one of the key steps to develop an acceptable deep learning model. Figure 4 shows the CFG feature extraction algorithm to identify a set of CFG features from a given method. Such CFG features of each method will be used to compute the similarity score for the method comparison. Given a target method in a program, the proposed clone detection system extracts a set of CFG features from the corresponding control flow graph of the method. After creating the corresponding CFG of the input method, the detection system finds all paths on the control flow graph. The next step is to count the frequency of the CFG features. If the label of a CFG node on a CFG path is found in the feature set, the frequency of the CFG feature node increases by one. If the CFG node is found newly, it is added to the feature set. The frequency of the newly added feature node is one. The counting of the frequency of the CFG feature nodes is performed until all CFG paths are considered. Finally, through the feature extraction algorithm, the CFG feature set is produced that holds a pair of the feature node type and its frequency.

```
Input: Let MED be a target method in a program.
Output: Let FeatureSet be a set of CFG features. $FeatureSet = \{(f_1, feq_1), \cdot\cdot\cdot, (f_n, feq_n)\}$

$aCFG \Leftarrow$ generateCFG(MED)  //Creating CFGs by calling CFG generator on MED
$allPaths \Leftarrow$ findAllPaths(aCFG)  //Finding all paths on the control flow graph
foreach $path_i$ from allPaths do
    foreach $node_j$ from $path_i$ do
        if $node_j \in FeatureSet$ then
            //Getting the frequency to which $node_j$ is mapped in FeatureSet
            $feq_j \Leftarrow$ getFequency(FeatureSet, $node_j$)
            $feq_j \Leftarrow feq_j + 1$  // Increase the frequency by one
        else  //Adding $node_j$ to FeatureSet if $node_j$ is new
            $FeatureSet = FeatureSet \cup \{(node_j, 1)\}$  //Assigning one to frequency of $node_j$
        endIf
    endForeach
endForeach
return FeatureSet
```

Figure 4. Extracting CFG features from pairwise methods to identify code clones

The proposed clone detection system needs to train a binary-class classifier for code clone detection which uses training and test dataset for both true clone pairs and false clone pairs. For the training of the classifier, the similarity scores of each feature vector and its label are provided from the training dataset. The true clones are labelled to ‘1’ while the false clones are labelled to ‘0’. Keras [7]—an open source machine learning library for deep learning— is used to train a clone classifier. Four input nodes and two output nodes are configured for the input layer and the output layer respectively. Since the proposed clone detection approach considers four features of the control flow graph, the input layer holds four nodes which independently take in four feature values. The output layer contains two nodes because the clone detection problem is the binary classification task to check whether two methods are clones or not. The proposed clone detection system configures the deep learning model to include eight hidden layers and to run 200 iterations for training.

Table 2 lists the node types of the control flow graph that should be considered to detect code clones. The node type is the construct of Java programs such as expressions, control constructs, conditional constructs, and exception handling constructs. The Java syntax of the node type is represented in the last column and follows BNF-style conventions: [expr] means zero or one occurrences of expr, {expr} means zero or more occurrences of expr, and (x | y) denotes one of either x or y. The proposed clone detection system will extract node types from methods and then generate edges and paths which are made up of the nodes.

Table 3 shows the datasets of training and testing for the clone classifier. Folder #4 in BigCloneBench is used as the training dataset since it has the largest number of true and false clone pairs. The true clone pairs in the training dataset include T1, T2, VST3, and ST3 clones and the total number of true clones is 13,399 which is equal to the total number of false clones. MT3 and WT3/4 clones are intentionally excluded from the training dataset because they could contain noisy data which may produce false alarms on detecting code clones.

For the testing of the clone classifier, the test dataset is made from Java source files in the four folders of BigCloneBench such as #2, #3, #7, and #10. The source files in the folder #4 are excluded because they are used as the training dataset. Unlike the training dataset, the test dataset contains only true clone pairs without false clone pairs. The similarity scores on the pairwise methods are computed through the same procedures as the training phase. These procedures include the CFG construction, the CFG path generation, the CFG feature extraction, and the similarity score computation.

Table 2. A list of CFG nodes

No	Node Type	Java Syntax
1	StatementExpressionContext	StatementExpression;
2	LocalVariableDeclarationContext	{VariableModifier} Type VariableDeclarators;
3	IfStatementContext	if ParExpression Statement [else Statement]
4	ForStatementContext	for (ForControl) Statement
5	WhileStatementContext	while ParExpression Statement
6	DoWhileStatementContext	do Statement while ParExpression;

7	SwitchStatementContext	switch ParExpression { SwitchBlockStatementGroups}
		SwitchBlockStatementGroups: { SwitchBlockStatementGroup }
		SwitchBlockStatementGroup: SwitchLabels BlockStatements
		SwitchLabels: SwitchLabel { SwitchLabel }
		SwitchLabel: case Expression : case EnumConstantName : default :

8	LabelStatementContext	Identifier : Statement
9	ReturnStatementContext	return [Expression] ;
10	BreakStatementContext	break [Identifier] ;
11	ContinueStatementContext	continue [Identifier] ;
12	SynchBlockStatementContext	synchronized ParExpression Block
13	TryStatementContext	try Block (CatchClause | [CatchClause] FinallyBlock)
14	TryWithResourceStatementContext	try ResourceSpecification Block [CatchClause] [FinallyBlock] ResourceSpecification : (Resources [;]) Resources : Resource {; Resource} Resource : {VariableModifier} ClassOrInterfaceType VariableDeclaratorId = Expression
15	ThrowStatementContext	throw Expression ;

Table 3. Datasets of training and testing

Dataset		True Clones						False Clones
		T1	T2	VST3	ST3	MT3	WT3/4	
Training(#4)		4,500	3,103	1,204	4,592	0	0	13,399
Testing	#2	1,552	7	22	1,411	2,687	10,107	
	#3	630	583	523	2,734	24,335	825,838	
	#7	33	4	21	211	1,627	10,665	
	#10	151	61	282	922	1159	296	

Figure 5 shows the experimental result on the test dataset. The proposed clone classifier has been evaluated with different similarity thresholds in order to check how the similarity threshold affects the performance of the clone detection. The classifier finds clones when the likelihood value is greater than or equal to the similarity threshold. In these experiments, the clone classifier is configured with 8 hidden layers and 200 epochs. The proposed detection system effectively identified T1, T2, and VST3 clones-the recall results are close to 100% (except the folder #2 dataset) with the different similarity thresholds: 0.95, 0.96, 0.97, and 0.98. The proposed classifier also detects effectively ST3 clones on the datasets #2 and #10-the recall values are greater than 90%. With the datasets #3 and #7, the performance of the clone detection goes down a little bit as the detection thresholds become larger. The detection system did not show desirable detection performance on the MT3 and WT3/4 clones. It is still challenging to detect semantic clone types even though the proposed approach is based on control flow graphs which may represent more abstract perspectives of semantically similar code blocks than token-based approaches.

Deep neural network models may be affected by hyperparameters such as hidden layer and epoch. Tables 4 and 5 show the recall results with different parameter settings for training the proposed clone detection system. Five clone detection models are created with the different numbers of hidden layers such as 2, 4, 6, 8, and 10. The default number of the epoch is 100. Although the training models are not affected severely by the number of hidden layers in this experiment, the trained model with 8 hidden layers achieves the best performance. Table 5 shows the number of epochs impacts on the performance of the clone detection. The proposed clone classifier works best when the number of epochs is 200.

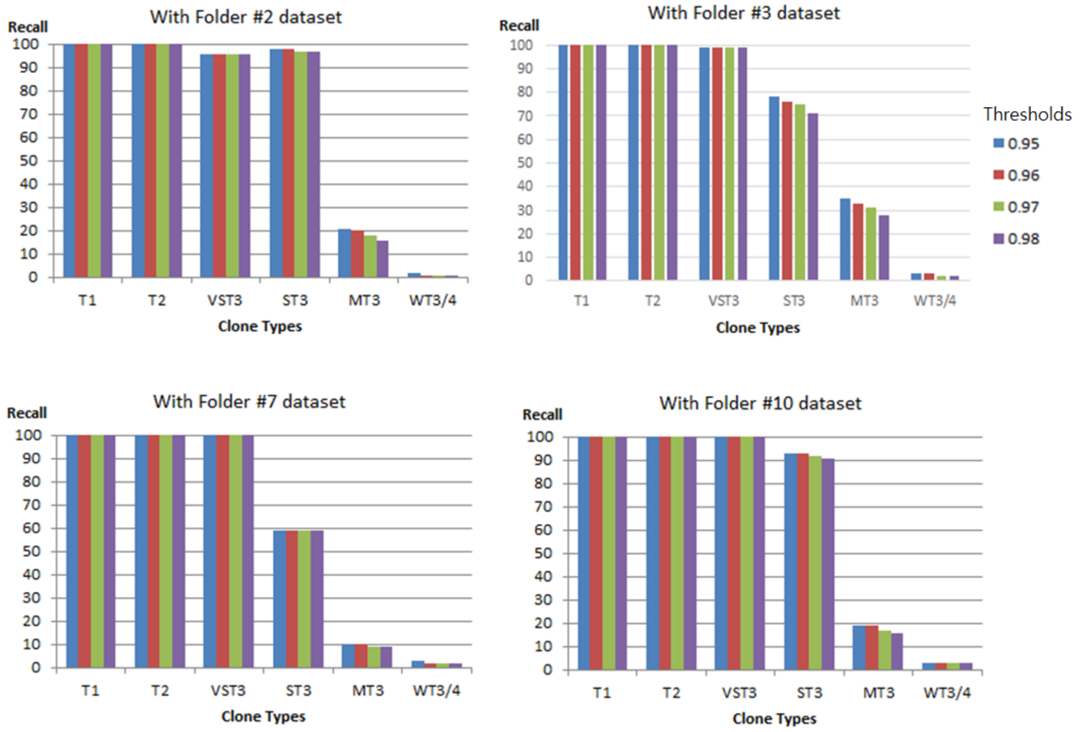

Figure 5. Recall results with different similarity thresholds (0.95~0.98)

Table 4. Recall results with different hidden layers (Threshold = 0.98)

# of hidden layers	T1	T2	VST3	ST3	MT3	WT3/4
2	100	100	99	74	28	2
4	100	100	99	73	27	2
6	100	100	99	71	28	2
8	100	100	99	74	29	2
10	100	100	99	70	25	2

Table 5. Recall results with different epochs (Threshold = 0.98)

# of epochs	T1	T2	VST3	ST3	MT3	WT3/4
50	100	100	99	72	27	2
100	100	100	99	71	28	2
150	100	100	99	72	28	2
200	100	100	99	73	28	2
250	100	100	99	72	27	2

4. RELATED WORK

Many studies have been conducted to overcome research challenges in detecting code clones across source code. Most of existing clone detection methodologies can be categorized into five types: text-based [8], token-based [9], tree-based [10], graph-based [11], and metrics-based [12] approaches. One of the emerging research trends is to leverage deep learning algorithms to enhance the performance of the existing clone detection strategies. The promising outcomes of deep learning affect many other fields beyond artificial intelligence communities. Software engineers have been involved in actively applying deep learning to solve typical problems in software engineering such as clone detection, bug prediction, and security prediction.

Sheneamer and Kalita's clone detection approach [13] uses typical machine learning algorithms so to detect semantic code clones. They use a supervised learning approach where semantic features are extracted from ASTs and PDGs and training data are labelled with clones or non-clones. Their approach is based on machine learning algorithms, not deep learning algorithms. White et al. [14] use a deep learning approach to detect code clones by combining recurrent neural network with recursive neural network at method and file levels. The experimental results show their methodology is feasible in some cases, but they still need to conduct more case studies on popular clone benchmarks. Li et al. [15] provide a token-based clone detection approach using deep learning-based clone classifier. They extract feature vectors by tokenizing method pairs and then compute similarity scores of the feature vectors. The clone classifier is trained with eight similarity scores of known true clones and false cones. In the testing phase, the trained clone classifier predicts code clones in a codebase of unknown clones. This approach still has room for improvement on finding semantic clones like Type-3 and Type-4 clones. Saini et al. [16] propose a clone detection approach to focus on harder-to-detect semantic clones. Their approach is based on a deep neural network with Siamese architecture where information retrieval and metric-based methods are combined. To detect semantic clones, their approach excludes semantically dissimilar clones using a semantic filter instead of using semantic features. Phan et al.'s work [17] show CFG-based deep learning can be used for software defect prediction as well as clone detection. They apply convolutional neural networks for predicting software defect using control flow graphs. Control flow graphs are built from assembly files and then are represented as vectors which are given to convolutional neural networks. The convolutional neural network explores the behavior of target code using vector representations and reports software defects in unseen datasets after being trained with training datasets.

5. CONCLUSIONS AND FUTURE WORK

This paper presents a code clone detection framework to find effectively clone types with a deep learning-based clone classifier. The proposed approach to clone detection is based on the extraction of features from CFGs of given code fragments. The CFG features are represented as feature vectors so that they can be compared to determine whether code fragments are similar or dissimilar. The clone detection classifier is trained and tested with similarity scores that are computed from the feature vectors. The proposed detection framework effectively found syntactic clone types such as T1, T2, and VST3 clones. It also identified ST3 clones with the acceptable but not excellent recall results. In the case of sematic clone types such as MT3 and WT3/4 clones, the detection performance still needs to be improved. Although the proposed approach to clone detection needs to be improved further, the promising experimental results suggest that deep learning-based clone detection classifiers can be effective in finding code clones. In the future, more code clones will be explored to enhance functions of the proposed clone detection on semantic clone types. Furthermore, unsupervised deep learning algorithms can be considered to improve the weakness of the clone classifier using supervised learning. The clone detection framework will be applied for other programming languages to extend the generality of the proposed clone detection methods.

ACKNOWLEDGEMENTS

This research was financially supported by Mokpo National Maritime University in 2018.

REFERENCES

[1] S. Bellon, et al., "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering,*, vol. 33, no. 9, pp. 577–591, 2007.

[2] C. K. Roy, et al., "A survey on software clone detection research," *Queen's University, Tech. Rep. 2007-541,* 115 pp, 2007.

[3] J. Svajlenko, et al., "Towards a big data curated benchmark of inter-project code clones," *30th IEEE International Conference on Software Maintenance and Evolution,* Victoria, BC, Canada, September 29 - October 3, 2014.

[4] J. Svajlenko, et al., "BigCloneEval: A clone detection tool evaluation framework with BigCloneBench," *32nd IEEE International Conference on Software Maintenance and Evolution,* pp. 596-600, 2016.

[5] Y. LeCun, et al., "Deep learning," *Nature,* vol. 521, pp. 436-444, May 2015.

[6] P. Bielik, et al., "Programming with Big Code: Lessons, Techniques and Applications," *The Inaugural Summit on Advances in Programming Languages,* 2015.

[7] "Keras," https://keras.io/, accessed: Nov. 2018.

[8] S. Ducasse, et al., "A language independent approach for detecting duplicated code," *In Proceedings of the IEEE International Conference on Software Maintenance,* pp. 109–118, 1999.

[9] T. Kamiya, et al., "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* vol. 28, issue 7, pp. 654–670, 2002.

[10] L. Jiang, et al., "DECKARD: scalable and accurate tree-based detection of code clones," *International Conference on Software Engineering,* pp. 96–105, 2007.

[11] J. Krinke, "Identifying similar code with program dependence graphs," *In Proceedings of the Eighth Working Conference on Reverse Engineering,* pp. 301–309, 2001.

[12] J. Mayrand, et al., “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics,” *In Proceedings of International Conference on Software Maintenance*, 1996.
[13] A, Sheneamer, et al., “Semantic Clone Detection Using Machine Learning,” *15th IEEE International Conference on Machine Learning and Applications,* Dec., 2016.
[14] M. White, et al., “Deep learning code fragments for code clone detection,” *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.
[15] L. Li, et al., “CCLearner: A Deep Learning-Based Clone Detection Approach,” *IEEE International Conference on Software Maintenance and Evolution*, Sep. 2017.
[16] V. Saini, et al., “Oreo: Detection of Clones in the Twilight Zone,” *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2018.
[17] A. V. Phan, et al., “Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction,” *IEEE 29th International Conference on Tools with Artificial Intelligence*, Nov. 2017.

BIOGRAPHY OF AUTHOR

Dong Kwan Kim is an associate professor in the Department of Computer Engineering at Mokpo National Maritime University. His research interests include deep learning, software evolution, run-time systems, and mobile programming.