On the Effectiveness of Vector-based Approach for Supporting Simultaneous Editing of Software Clones

Seiya Numata[1], Norihiro Yoshida[2] (✉), Eunjong Choi[3], and Katsuro Inoue[1]

[1] Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{s-numata,inoue}@ist.osaka-u.ac.jp,
[2] Nagoya University, Furo-cho, Chikusa, Nagoya 464-8601, Aichi, Japan
yoshida@ertl.jp,
[3] Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma, Nara 630-0192, Japan
choi@is.naist.jp,

Abstract. Code clone is one of the factors that makes software maintenance more difficult. Once a developer find a defect in a code fragment, he/she has to inspect the all of the code clones of the code fragment. In this study, we investigated the effectiveness of query-based use of a vector-based clone detection tool for supporting simultaneous fixing of buggy clones in source code and compared it with the query-based use of a token-based clone detection tool CCFinder.

Keywords: Code clone detection tool, Software maintenance, Simultaneous editing

1 Introduction

A code clone is a code fragment that has identical or similar code fragments to it in the source code [10]. So far, a lot of code clone detection techniques have been developed to capture various aspects of source code similarity [7, 10, 13].

For the detection of syntactically identical or similar code fragments, token-based and tree-based approaches detect identical token sequence and similar syntax tree in source code, respectively [7, 10]. These approaches are able to detect useful clones (i.e., code fragments to be merged [5], inconsistent code clones that are suspected to include a bug [9]), but have limitations of false positives [5] (syntactically similar but semantically different code) and false negatives [4] (syntactically different but semantically similar clones). As a more sophisticated approach, a few techniques have been proposed for the detection of only semantically similar clones from source code [7, 11, 12].

For example, Komondoor and Horwitz proposed an approach to finding isomorphic subgraphs of program dependence graphs (PDGs) in order to find semantic clones from source code [12]. Also, MeCC detects C functions implementing semantically-similar computations based on the similarity of abstract memory states between them [11]. Jiang and Su proposed an approach to compare program execution traces via random testing in order to find functionally equivalent code fragments [8]. However, those approaches have limitations. Identifying isomorphic subgraph of PDG is time-consuming as well as identifying abstract memory states of C functions [11], and comparing execution traces require a number of test suites to achieve sufficient level of test coverage. Also, it is difficult for those approaches to be applied to uncompilable source code.

In our previous research [18], we developed a vector-based approach for the lightweight detection of function clones. In our vector-based approach, a feature vector is generated for each function, based on the occurrence of identifiers and reserved keywords, and then clustering of the generated vectors is performed by means of locality-sensitive hashing (LSH) [6]. Finally, clones are detected based on the similarities between each pair of feature vectors. We confirmed the advantages of the vector-based approach over MeCC as follows:

- Detects a number of function clones but also maintains a low false positive rate in comparison to MeCC
- Detect in a shorter time
- Finds a larger number of clone-related defects and bad smells

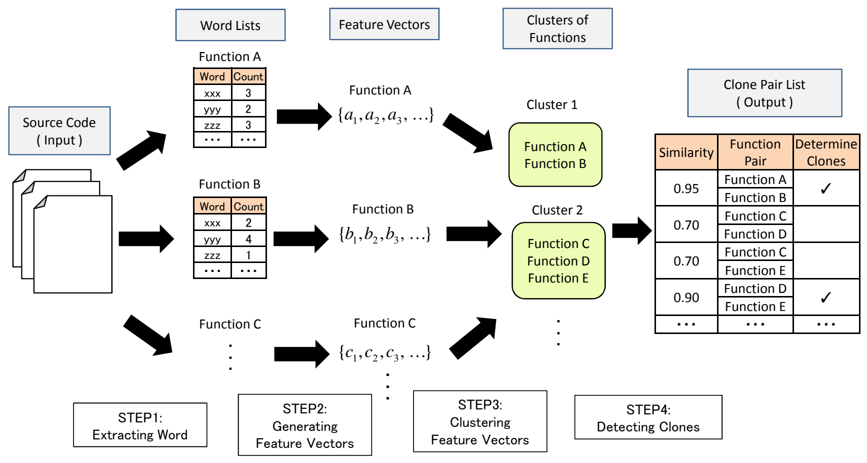
We introduced the tool based on the vector-based approach to a Japanese multinational IT company and then got feedbacks from practitioners in the company. According to the feedbacks, the practitioners need to know the effectiveness of the vector-based approach for supporting simultaneous fixing of buggy clones. They are mainly motivated to perform query-based use of the vector-based approach. When they find a defect in a function, they would like to give the function as a query to the vector-based approach and then discover clone-related defects from the detected function clones. In our previous research [18], we detected function clones from OSS and then manually confirmed that a large number of the detected function clones included defects and bad smells. However, the effectivness of query-based use of the vector-based approach is still unknown so far.

In this study, we investigated the effectiveness of query-based use of the vector-based clone detection tool for supporting simultaneous fixing of buggy clones in source code. In the investigation, we used the collection of clone-related defects that was collected by Li and Ernst for the evaluation of a cloned buggy code detector CBCD [14, 15].

The remainder of this paper is organized into the following sections. Section 2 details a vector-based approach to detecting code clones. Section 3 describes a method to investigate the effectiveness of query-based use of a clone detection approach. Section 4 explains the investigation result, Section 5 reviews related work and finally, Section 6 summarizes this study.

2 Vector-based approach to detecting clones

Figure 1 provides an overview of the vector-based approach. The Vector-based approach takes source code is used as the input, and the the output consists of



Similarity	Function Pair	Determine Clones
0.95	Function A Function B	✓
0.70	Function C Function D	
0.70	Function C Function E	
0.90	Function D Function E	✓
⋯	⋯	⋯

Fig. 1. Overview of the Vector-based approach

a list of clone pairs (i.e., pairs of function clones that are identical or similar to one another).

Hereafter, we use the term *word* to represent a set of identifiers (e.g., variables and function names) and reserved keywords (e.g., conditional statements and interactive statements).

2.1 STEP1:Extracting Word

In this step, *words* are extracted from functions in the source code. During this process, if an identifier consists of more than one word, it is divided into different words as follows:

– It is divided using a delimiters such as hyphens or underscores, between words.
– It is split by using a capital letter for each word using CamelCase

2.2 STEP2:Generating Feature Vectors

In this step, feature vectors are generated based on the weights of extracted words from STEP1. For this, we use Term Frequency Inverse Document Frequency(*TF-IDF*) [2], a popular technique, in IR, for weighting each word. Uramoto et al. used *TF-IDF* to weight newspaper articles for the relation of multiple news articles [17]. In this study, we use *TF-IDF* to weight words in the source code.

TF-IDF combines Term Frequency(TF) weights and Inverse Document Frequency(IDF) weights. For example, suppose N_w represents the occurrences of w in a function, and N_{all} represents the total number of occurrences of all words in a function. In addition, *Function* represents the set of all functions in the source code, while $Function_w$ represents the set of functions that contain word w exists. The weighting of w using *TF-IDF* is defined as follows :

$$tf_w = \frac{N_w}{N_{all}} \qquad idf_w = \log \frac{|Function|}{|Function_w|} \qquad tfidf_w = tf_w \times idf_w$$

2.3 STEP3:Clustering Feature Vectors

Clustering is used to identify candidates for clone pairs, and is conducted. prior to clone detection (STEP4) for the sake of time efficiency. In this step, the feature vectors generated in STEP 2 are clustered using Locality-Sensitive Hashing(LSH) [6], which is known to be an efficient nearest neighbor search algorithm.

To cluster feature vectors, in this study we usesdE^2LSH[4][1], which implements the LSH algorithm. Given a feature vector as a query, E^2LSH performs clustering of feature vectors approvimate to the query from the dataset, based on Euclidean distance. Using a dataset of functions, with each feature vector (function) as a query, a set of similar functions is returned.

2.4 STEP4:Detecting Clones

Clone pairs are detected based on the *cosine similarity* between all of the feature vectors of the clustered feature vectors obtained in STEP 3. *Cosine similarity* identifies similarities between multidimensional vectors. Formally, the *cosine similarity* between two vectors $\boldsymbol{a}$, and $\boldsymbol{b}$ whose dimension is d are determined as follows :

$$sim(\boldsymbol{a}, \boldsymbol{b}) = cos(\boldsymbol{a}, \boldsymbol{b}) = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} {a_i}^2}\sqrt{\sum_{i=1}^{d} {b_i}^2}}$$

Cosine similarity takes a value between 0 and 1, because feature values only have positive values, as seen in the formulation of the TF-IDF described in STEP 2. If the *cosine similarity* value between two feature vectors is higher than the threshold, these two vectors are regarded as clone pairs. In this study, we set the threshold at 0.9 to reduce the probability of false positive results.

3 Investigation Method

We investigated the effectiveness of query-based use of the vector-based clone detection tool for supporting simultaneous fixing of buggy clones in source code and compared it with the query-based use of a token-based clone detection tool CCFinder [10]. Please note that we set 10 tokens as minimum length of a token sequence for CCFinder because most clone-related defects in the dataset are 10 tokens or smaller in source code.

As we mentioned in Section 1, this research was triggered by the feedbacks from the Japanese multinational IT company. Because this company has used CCFinder for several years, they would like to know the effectiveness comparison of the vector-based approach and CCFinder.

[4] `http://www.mit.edu/~andoni/LSH/`

Table 1. The numbers of N1, N2, N3 and N4

	Vector-based approach		CCFinder
	threshold = 0.9	threshold = 0.5	
N1	11	10	10
N2	22	13	11
N3	4	11	16
N4	1	4	1

3.1 Dataset

For our investigation, we used the dataset of clone-related defects that was collected by Li and Ernst [14, 15]. The clone-related defects in the dataset are from the OSS repositories of Git, Linux kernel and PostgreSQL that are written by C/C++.

The dataset also includes commit IDs of not only clone-related defects but also code clones of those defects [14]. Please note that we removed the instances if a defect and its code clones in the same function because the purpose of this study is the investigation of effectiveness of query-based use of the vector-based approach.

3.2 Effectiveness Criteria

We used not only precision/recall and F-measure but also a categorization proposed by Li and Ernst [15]. Li and Ernst proposed the following categorization for each instance in their dataset of clone-related defects.

- N1: no false positives, no false negatives
- N2: no false positives, some false negatives
- N3: some false positives, no false negatives
- N4: some false positives, some false negatives

After clones of each clone-related defect are detected, each clone detection tool can be characterized by the numbers of N1, N2, N3 and N4.

Precison/recall and F-score for each approach are calculated from the total numbers of true positives, detected functions and buggy functions that are involved in the dataset.

4 Investigation Results

Table 1 shows the numbers of N1, N2, N3 and N4 and Table 2 shows recall, precision and F-score for each approach.

According the numbers of N2 in Table 1, the vector-based approach with threshold=0.9 is the most efficient for supporting simultaneous fixing of buggy

Table 2. Recall, Precison and F-score

	Vector-based approach		CCFinder
	threshold=0.9	threshold=0.5	
Recall	0.41	0.53	0.53
Precision	0.59	0.11	0.01
F-score	0.48	0.18	0.02

clones because N2 means no false positive. When developers have only a limited time, the vector-based approach with threshold=0.9 is the most suitable.

In terms of the numbers of N3 in Table 1, CCFinder is the highest. For the development of a high-reliability software system, CCFinder is the most suitable because N3 means no false negative.

N1, N2, N3 and N4 take account of the existence of false postives and negatives and do not take account of the numbers of them precisely. On the other hand, Recall/Precision takes account of the numbers of them precisely.

According to Table 2, the precision of CCFinder is extremely low. In several instances of clone-related defects in the dataset, CCFinder detects a large number of false positives (max. 218). This means that CCFinder is unsuitable when developers have only a limited time. The vector-based approach with threshold=0.9 is most suitable when developers have only a limited time according to the precision in Table 2.

In terms of recall, the all of the score are almost same. The vector-based approach with threshold=0.5 and CCFinder are the highest score between them. Since the precision of CCFinder is extremely low, the vector-based approach with threshold=0.5 is more suitable for the development of a high-reliability software system.

5 Related Work

Thus far, various techniques have been proposed for the detection of code clones from source code. For the detection of syntactically identical or similar code fragments, the token-based and tree-based approaches detect identical token sequences and similar syntax trees in the source code, respectively [7, 10]. However, these approaches may result in false positives (syntactically similar but semantically different clones) [5] and false negatives (syntactically different but semantically similar clones) [4].

As a more sophisticated approach, a few techniques have been proposed for the detection of only semantically similar clones from the source code [8, 11, 12]. For example, Komondoor and Horwitz proposed for finding the isomorphic subgraphs of PDGs in order to find the semantic clones from the source code [12]. Additionally, MeCC [11] detects C functions implementing semanticallysimilar computations, based on the similarity of their abstract memory states. Jiang and Su proposed an approach to comparing program execution traces via random testing in order to find functionally equivalent code fragments [8]. The vector-based approach in this paper is inspired by the existing vector-based approach that is proposed by Marcus et al. [16]. Their original approach uses a LSI-based clustering technique to form all clusters of similar entities. LSI-based retrieval is a considerable idea to improve the recall of the vector-based approach in our study. However, we do not use LSI because it leads the increase of the detection time.

Various applications of the detection of code clones from source code have been proposed. For example, several studies have been conducted on the support of clone refactoring using clone detection techniques. Balazinska et al. proposed a code clone classification method for the identification of reengineering opportunities [3]. Higo et al. [5] proposed a set of metrics to represent the difficulty of merging clones detected by the token-based clone detection tool CCFinder [10]. Yoshida et al. proposed an approach for extracting clone clones that are related to each other from the output of CCFinder, and suggesting these be used as a large-scale reengineering opportunity [19]. Combining these approaches with the vector-based approach appears to be a promising solution for achieving the efficient support of clone refactoring.

6 Summary

In this study, we investigated the effectiveness of the query-based use of the vector-based clone detection tool for supporting simultaneous fixing of buggy clones in source code. In the investigation, we used the collection of clone-related defects that was collected by Li and Ernst for the evaluation of a cloned buggy code detector CBCD [14, 15].

The summary of the investigation result is as follows:

- The detection result of the vector-based approach with threshold=0.9 is highest precesion.
- The detection results of the vector-based approach with threshold=0.5 and CCFinder are highest recall.
- Since the precision of CCFinder is extremely low, the vector-based approach with threshold=0.5 is more suitable for the development of a high-reliability software system.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Numbers 25220003, 26730036, 15H06344 and 16K16034.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. CACM 51(1), 117–122 (2008)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval: The Concepts and Technology behind Search (2nd Edition) (ACM Press Books). Addison-Wesley Professional (2011)

3. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Measuring clone based reengineering opportunities. In: Proc. of METRICS '99. pp. 292–303 (1999)
4. Deissenboeck, F., Heinemann, L., Hummel, B., Wagner, S.: Challenges of the dynamic detection of functionally similar code fragments. In: Proc. of CSMR '12. pp. 299–308 (2012)
5. Higo, Y., Kusumoto, S., Inoue, K.: A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. Journal of Software Maintenance and Evolution 20(6), 435–461 (2008)
6. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proc. of STOC '98. pp. 604–613 (1998)
7. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: scalable and accurate tree-based detection of code clones. In: Proc. of ICSE '07. pp. 96–105 (2007)
8. Jiang, L., Su, Z.: Automatic mining of functionally equivalent code fragments via random testing. In: Proc. of ISSTA '09. pp. 81–92 (2009)
9. Jiang, L., Su, Z., Chiu, E.: Context-based detection of clone-related bugs. In: Proc. of ESEC-FSE '07. pp. 55–64 (2007)
10. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28(7), 654–670 (2002)
11. Kim, H., Jung, Y., Kim, S., Yi, K.: MeCC: memory comparison-based clone detector. In: Proc. of ICSE '11. pp. 301–310 (2011)
12. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Proc. of SAS '01. pp. 40–56 (2001)
13. Krinke, J.: Identifying Similar Code with Program Dependence Graphs. In: Proc. of WCRE '01. pp. 301–307 (2001)
14. Li, J., Ernst, M.D.: CBCD: Cloned buggy code detector. Tech. Rep. UW-CSE-11-05-02, University of Washington Department of Computer Science and Engineering (2011)
15. Li, J., Ernst, M.D.: CBCD: Cloned buggy code detector. In: Proc. of ICSE '12. pp. 310–320 (2012)
16. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: Proc. of ASE '01. pp. 107–114 (2001)
17. Uramoto, N., Takeda, K.: A method for relating multiple newspaper articles by using graphs, and its application to webcasting. In: Proc. of ACL '98. pp. 1307–1313 (1998)
18. Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K.: A high speed function clone detection based on information retrieval technique. IPSJ Journal 55(10), 2245–2255 (2014), in Japanese
19. Yoshida, N., Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: On refactoring support based on code clone dependency relation. In: Proc. of METRICS '05. pp. 16:1–16:10 (2005)