A Token Frequency Based Code Clone Detection

He Feng*, Liuqing Li†,
*Department of Physics,
Virginia Tech, Blacksburg, VA, 24061
Email: fenghe@vt.edu
†Department of Computer Science,
Virginia Tech, Blacksburg, VA, 24061
Email: liuqing@vt.edu

***Abstract*—Code clone detection remains an active topic in software engineering. It is not only common in the programming process, but also leading to low maintainability.**

In this paper, we propose a simple and efficient token-frequency-based approach in Java to detect cloned codes. Our tool extracts variables in each method, using bigram to find out similar variable names, as well as counting keywords and symbols. After transforming these pieces of information into vectors, components are set with different weights by machine learning method, then a similarity algorithm is introduced to find out similar codes.

As a counting-based detection tool, it is small but very efficient: 90% in both precision and recall rate. STCD is a new and effective tool in clone detection.

I. Introduction

It is very common that people clone codes in their programs, i.e. reuse some code fragments by copying with or without modification in the coding process. Previous research shows a significant fraction of the code being cloned even in some well-known software systems. The fraction of duplicated code in X Windows System is about 19%[1], while in some core parts of Linux the number is between 15% and 25%[2]. Even if code clones can improve the efficiency of a program in some cases, such as remove method calls and reduce execution time, they propose challenges for developers to maintain software: if fixing a bug, adding or removing a feature, it has to be applied to all the similar parts. To solve this problem, clone detection has become an important tool for programmers, since it can help indentify duplicated code when making a change.

There are mainly four types of clones:

Type 1: simply includes the variations in whitespace, layout and comments.

Type 2: in addition to *Type 1*, it allows more variations in identifiers, literals and types.

Type 3: contains *Type 2* and allows further modifications such as changed, added or removed statements.

An example of *Type 1, 2, 3* clones is shown below:

AN EXAMPLE GOES HERE

Type 4: code fragments perform the same computation or results but are implemented in different ways.

Few tools have been developed to detect *Type 4* clones so far.

Based on the clone types, a number of techniques and tools have been implemented. Token-based method shows the best performance among all these techniques. However, each tool has its own limitations, just like the time cost of CCFinder. We propose to develop and implement a scalable clone detection based on tokens to detect code clone in token variation and statement modification.

Our approach is based on the following insight: most developers would not make dramatic changes in code clone process except for some token modifications including keywords, types, variables and operators. We try to design and improve a simple statistical method to calculate the similarity between code fragments. Specifically, to detect whether fragment A is a clone of fragment B or not, we first tokenize each fragment and design a flexible token filter. Any useful tokens, such as types, variables, identifiers, operators, will be counted, so the information is transformed into a token frequency vector. The second step is the pretreatment of variable names, take v_A and v_B, compare and find out similar variables with small variations in prefix, suffix and substitution. After finding matching variable names, the third step is to use a similarity algorithm to measure the two vectors, by putting different weights on different levels of tokens, and the weights can be set by machine learning, which enhances the accuracy. Finally, a threshold is introduced and optimized, depending on the precision and recall from actual experiments.

After actual testing, STCD reaches 90% in both precision and recall rate, which means STCD is very effective.

The structure of the rest of the paper is as follows: in Section 2, we introduce the mechanism of STCD in detail; in Section 3, we describe the implementation of STCD; in Section 4, we evaluate STCD and get the precision and recall rate, and also the time cost; in Section 5, we analyze the threats to validity; Section 6 is the conclusion; Section 7 lives the related work.

Our solution is new and we believe it's aggressive in clone detection field.

II. Approach

Our approach is constructed as follows:

First is method declaration. We used AST Parser Tool to catch each method and get the information, which means, the *startLineNumber, endLineNumber, methodName, methodParameters, methodType, methodBody*. This costs extra time and can be improved in future work.

Second is the tokenization of method body and get the frequency of each token, which includes the comment and white space removal process.

Third, for each method, we have a list of tokens, these tokens fall into 9 categories: *methodParameter, methodType, tokenListNumber, tokenListType, tokenListKeyword, tokenListMarker, tokenListOperator, tokenListOther1, tokenListOther2*. The purpose of doing so is that these categories have different weights, for example, two pieces of code both have 2 type names: "int", that provides no useful information, however, if two pieces of code both have 2 method parameter names: "drawVerticalLine", that means these pair are more likely the same.

AN EXAMPLE GOES HERE

Fourth, similarity of *methodParameter* is caculated with bigram similarity. If "drawVerticalLine" and "VerticalDrawLine" appear in two code fragments respectively, we don't want to take them as different. So two similar variable names will be compared by bigram similarity, if they are similar, they will be taken as the same and their frequencies will be compared later.

Fifth, after the pretreatment of variable names, a similarity algorithm is applied to each category, and the number of similarity will be caculated, so we have 9 similarity numbers.

Finally, the 9 similarity numbers are given different weights and the final similarity will be calculated. A threshold is introduced, if the similarity is higher than the threshold, the result will be printed out: clone group number, similarity number, method name 1, start and end line number, method name 2, start and end line number.

A diagram describing the whole process is shown in Fig:1.

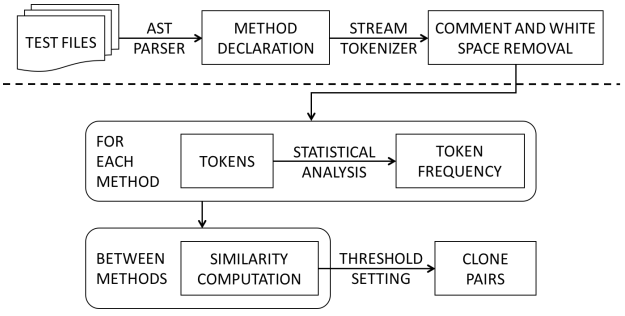

Fig. 1. Overall Project Framework

During the final step, different weights are given to different categories, however, picking 9 weights arbitrarily may not be resonable. So we used Machine Learning method to set the weights. Provided training files with "ground truth", machine learning process will decide the most proper weights, which is much more resonable than setting by hand. After actual testing, it turns out machine learning does provide a better result.

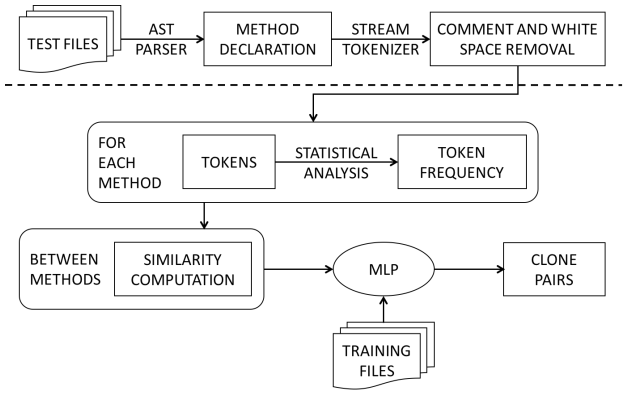

Fig. 2. Project Framework with Machine Learning

III. Implementation

Following the idea of our approach, we have implemented STCD. The source code is available at https://github.com/CSCC5704/SourceCodewithUI.

After catching all the tokens and accumulating their frequencies, in the pretreatment of variable names, we introduced bigram similarity. After actual testing, the threshold is set to 0.7 in our project. If the similarity of two variable names is higher than 0.7, then they are considered the same and their frequency will be compared.

For each category of tokens, to calculate the similarity, we used am algorithm similar to Levenshtein Distance:

$$\mathrm{Sim}(\mathrm{List}_x, \mathrm{List}_y) = 1 - \frac{\sum \mathrm{Diff}(\mathrm{Elem}_{x_i}, \mathrm{Elem}_{y_i})}{\sum(\mathrm{Freq}_{x_i} + \mathrm{Freq}_{y_i})} \quad (1)$$

where the difference is defined by:

$$\mathrm{Diff}(\mathrm{Elem}_{x_a}, \mathrm{Elem}_{y_a}) = \mathrm{Abs}(\mathrm{Freq}_{x_a} - \mathrm{Freq}_{y_a}) \quad (2)$$

This is a well-defined calculation, because no difference in the lists gives a similarity of 1, while totally different lists give similarity of 0.

An example shows the resonablity of this algorithm:

AN EXAMPLE GOES HERE

Before using machine learning to set the weights, we set the weights arbitrarily, which may not provide the best result, but it works in finding clones.

To get training data, we use manual selection to find "ground truth". However it is not so easy to manually find "ground truth": if a Java file has 100 methods, then we need to compare $\frac{100\times99}{2} = 4950$ pairs. In these pairs, most of them are not clone, so we applied our tool to the Java file, with a low threshold–0.65 to rule out most of the pairs, and manually checked the rest. This reduced our work from thousands of comparasions to less than a hundred. This process may introduce false negatives but we consider it to be small.

We selected training files from SWT–the tool to develop our UI. The advantage to use data from the same project is that machine learning can learn the developer's behavior. We chose

10 Java files, namely: *Combo.java, DragSource.java, DropTarget.java, FormData.java, Label.java, Printer.java, Program.java, Shell.java, Text.java, WebKit.java*. The number of methods ranges from 12 to 125.

In the training process, MLP studies 50 clone pairs and 150 non-clone pairs to decide the best weights. There are much more non-clone pairs but studying more of them does not give a better result.

To make STCD more user friendly, we also developed a UI, as shown in Fig:3.

The instruction to use STCD:

1) From the training menu users are able to select training files. STCD can also run the testing without training, then starts from Step 4.
2) After the selection of training files, users can adjust Hidden Nodes and Training Times in the MLP training process, as well as the threshold.
3) Run the training until it's ready, that takes less than seconds.
4) From the test menu users can choose the Java file to test. A second file can be selected, if not, the test will be within the same file.
5) Run the test until the result is shown.
6) From the Results window, click on the clone and the clone pairs will be shown in green and red color. The Method 1 and Method 2 windows show the token frequencies in each method.

IV. Evaluation

In the process of finding test data, there is a benchmark of Detection of Software Clones: http://www.bauhaus-stuttgart.de/clones/. It is a general repository and information center for Detection of Software Clones, it accepts files with labeled clones for clone detection tool evaluation. However its format does not meet our requirement, so again we use manual selection to find "ground truth" for testing, just as we did in finding training data: using a low threshold to rule out most of the pairs and manually compare the rest.

We select test data from SWT–since machine learning is based on SWT files. The 10 files we used for testing are: *Button.java, CoolBar.java, Menu.java, Spinner.java, TabFolder.java, TableItem.java, ToolBar.java, ToolItem.java, Tracker.java, Tree.java*. The number of methods ranges from 38 to 147.

Since we have our tool and test data, we are ready to do evaluation. Table shows the test result:

A TABLE GOES HERE

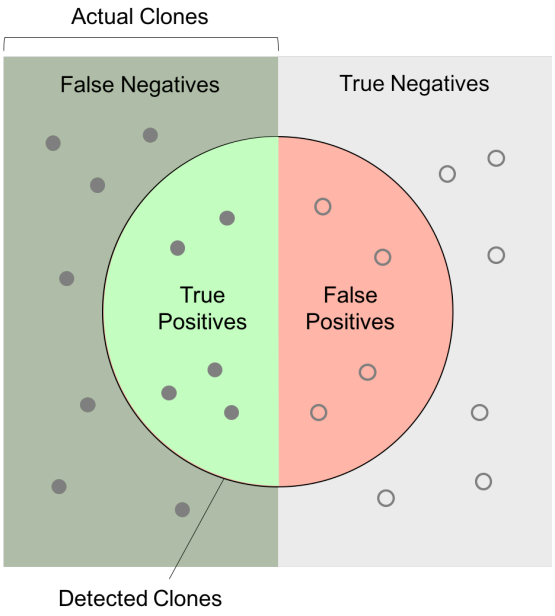

Fig. 4. Calculation of Precision and Recall

From Fig:4, based on the definition of Precision and Recall,

$$\text{Precision} = \frac{\text{True Positives}}{\text{Detected Clones}} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{Actual Clones}} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4}$$

we are able to evaluate our tool, the precision and recall is given in Fig:5.

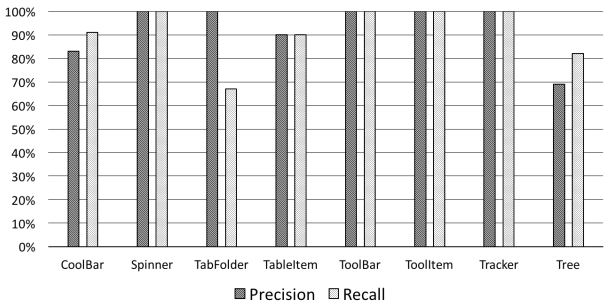

Fig. 5. Precision and Recall of STCD

The time cost of STCD is also evaluated, and is given in Fig:6. Since the comparasion for n methods is $\frac{n(n-1)}{2}$, we expect the time cost to be quadratic, and it turns out to be so– with some variations, which is the result of different method lengths.

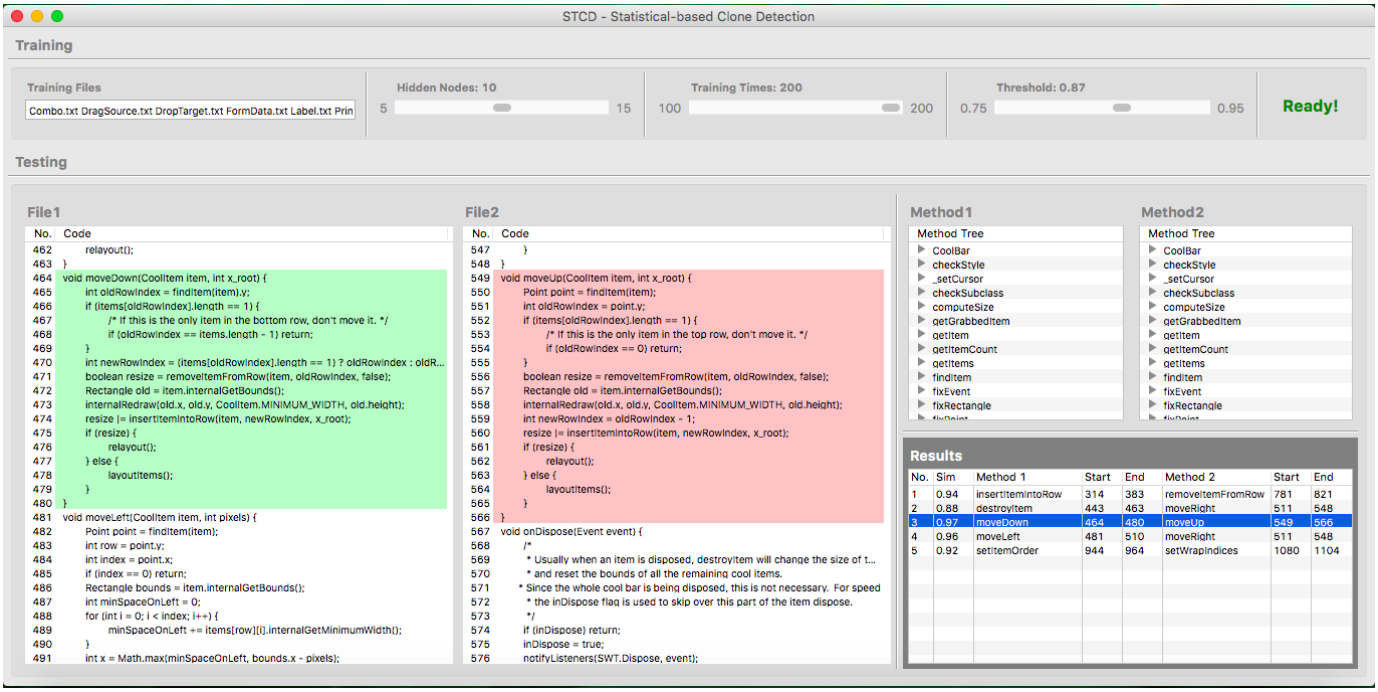

Fig. 3. User Interface

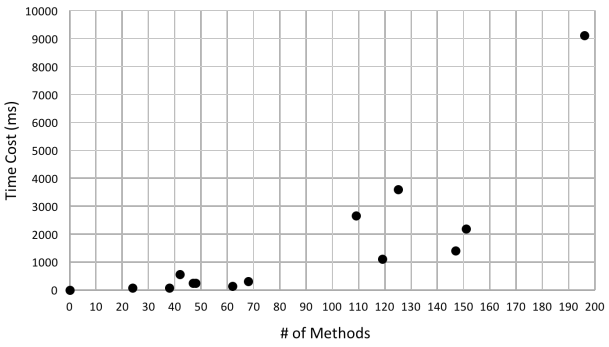

Fig. 6. Time Cost of STCD

V. Validity

the Validity

VI. Conclusion

the conclusion

VII. Related Work

Based on the cloned types, researchers have developed different techniques and tools.

Text-based, which means comparing whole lines to each other textually.

Token-based, it is still line-based comparison, but each sequence is summarized by a functor that abstracts the identifiers and literals, this encoding process preserves the structure of the whole sentence.

Metric-based, by gathering different metrics for code fragments, these metric vectors are compared.

Tree-based, usually means the abstract syntax trees(AST), a program is transformed into an abstract syntax tree, then the leaves and subtrees are compared.

Graph-based, program dependency graphs(PDG), control and data flow dependencies of a function can be represented by a program dependency graph, and comparison of subgraphs will be performed.

There are also other techniques but will not be discussed here. Among all the techniques and tools, token-based method performs well in both clone type detection and time cost.

Many token-based methods have been developed, like CCFinder[5], CMCD[6], Boreas[7], RTF[8].

CCFinder is a classical tool, it first makes a parameter-replacement, like replaces identifiers with a token, and compares each code portion to all other portions. This is very long and detailed comparison, which results in a high accuracy, however that could also costs a huge amount of time.

CMCD uses Count Matrix, it counts many aspects for each variable, and constructs a count vector for that variable, then the count vectors are compared to find out corresponding variables. This method works well in extracting variables those are copied but in different names, but is doesn't include other contributions, like keywords and symbols.

Boreas is based on CMCD, it uses CMCD to compare variables, as well as taking keywords and symbols into account, but still has room for improvement, both in executing time, and detection accuracy.

RTF uses flexible tokenization, but it doesn't use access modifiers like *private, protected, public* and type names *int, short, long, float, double*, so it loses information to some degree.

As for the similarity calculation, different methods such as Cosine similarity, Jaccard similarity, Euclidean distance and Manhattan distance can be applied in specific cases. The Cosine similarity function calculates cosine of the angle between two vectors v_a and v_b:

$$CosSim = cos(\alpha) = \frac{v_a \cdot v_b}{||v_a|| \cdot ||v_b||}$$

If v_a and v_b are in the same direction, then $cos(0)$ gives a similarity of 1, otherwise the cosine is between 0 and 1. This function takes care of vector length, but may not fit for the token frequency case.

CMCD gives a very good equation to derive the difference between two count vectors, which is expressed by the Euclidian Distance between them in space:

$$ED(v_a, v_b) = ||v_a - v_b||_2 = \sqrt{\sum_{i=1}^{n}(v_{ai} - v_{bi})^2}$$

where n is the dimension of the vectors, v_i is the ith component of v.

However the distance of two vectors may not reflect their actual difference because of their lengths, long vectors are more likely to have longer distance, while short vectors distance tend to be shorter, so it leads to some trouble while setting the threshold. A suitable similarity measurement will be designed in our work.

REFERENCES

[1] Brenda S. Baker, *On Finding Duplication and Near-Duplication in Large Software Systems*, Working Conference on Reverse Engineering(WCRE), pp. 86-95, 1995.

[2] G. Antoniola, U. Villanob, E. Merloc, M. Di Penta, *Analyzing cloning evolution in the Linux kernel*, Special Issue on Source Code Analysis and Manipulation(SCAM), 44(13): 755-765, 2002.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, *Comparison and Evaluation of Clone Detection Tools*, Transactions on Software Engineering, 33(9): 577-591, 2007.

[4] M.Gabel, L.Jiang and Z.Su, *Scalable Detection of Semantic Clones*, Proceedings of the 30th International Conference on Software Engineering(ICSE), pp. 321-330, 2008.

[5] T. Kamiya, S. Kusumoto and K. Inoue, *CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code*, IEEE Transactions on Software Engineering, Vol. 28, No. 7, 2002.

[6] Y. Yuan and Y. Guo, *CMCD: Count Matrix based Code Clone Detection*, Software Engineering Conference(APSEC), pp. 250-257, 2011

[7] Y. Yuan and Y. Guo, *Boreas: An Accurate and Scalable Token-Based Approach to Code Clone Detection*, Automated Software Engineering(ASE), pp. 286-289, 2012

[8] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin and S. Jarzabel, *Efficient Token Based Clone Detection with Flexible Tokenization*, Proceedings of the 6th European Software Engineering Conference and Foundations of Software Engineering(ESEC/FSE), pp. 513-515, 2007