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%[?], while in some core parts of Linux the number is between 15% and 25%[?]. 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:

```
public static int Num(int X, int Y){
        int a = 100;
        int b = 200;
        if(X>Y){
                return b + X;
        }
        else{
                a = 80;
                return a + Y;
        }
}

public static int Num(int X, int Y){
        int a=100;
        // no blanks _Type 1
        int c = 120;
        // b to c _Type 2
        if(X>Y){
                X = X - Y;
                // add stmt _Type 3
                return c + X;
        }
        else{
                a = 80;
                return a + Y;
        }
}
```

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 time cost is quadratic in terms of number of methods in Java file: running a Java file with 100 methods takes only 2 seconds, which is very practical.

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 detailed approach is constructed as follows:

A. Method Declaration

STCD is based on methods, it counts tokens in each method. So the very first thing to do is method declaration. Here 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 than needed and can be improved in future work.

B. Tokenization

Since we have divided the whole file into methods, next is to tokenize the method body and get the frequency of each token. This includes the comment and white space removal process.

C. Categorization

Now 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 meanings. 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. Later on these categories will be given different weights, depending on their importance.

A list of categorized tokens is shown in table **??**:

TABLE I
A List of Token Frequency

Token	Category	Freq	Token	Category	Freq
int	Type	1	)	Marker	6
for	Keyword	1	}	Marker	2
i	Other1	8	{	Marker	1
System	Other1	3	+	Operator	6
out	Other1	3	=	Operator	1
println	Other1	3	-	Operator	1
toBinary	Other1	1	¡	Operator	1
Integer	Other1	1	:	Other2	2
toBinaryString	Other1	1	5.0	Num	1
(	Marker	6	33.0	Num	1

D. Pretreatment of Variable Names

Among all the categories, variable name, i.e. *methodParameter* is special. Its similarity is caculated with bigram algorithm. 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.

E. Similarity Calculation for Each Category

After the pretreatment of variable names, a similarity algorithm is applied to each category. A list of tokens with their frequencies can be taken as a vector, so the similarity for two vectors can be caculated, and we have 9 similarity numbers for each comparision.

F. Final Similarity Calculation and Result

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

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

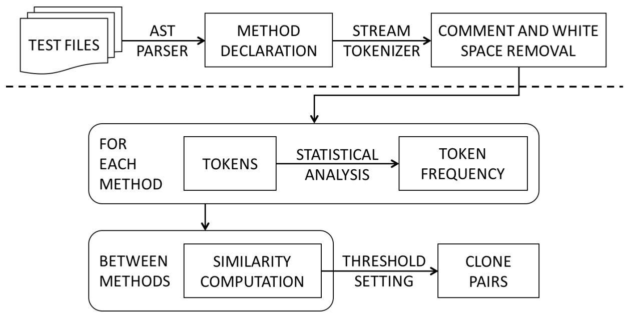


Fig. 1. Overall Project Framework

G. Machine Learning

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.

DISCUSS MORE ABOUT MLP HERE

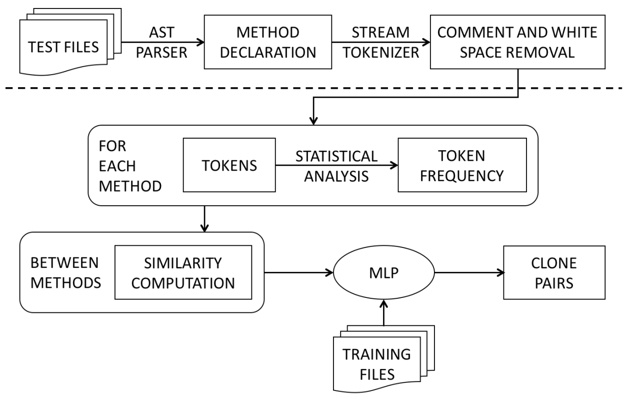


Fig. 2. Project Framework with Machine Learning

III. Implementation

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

A. Bigram Similarity

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 frequencies will be compared.

B. Similarity Algorithm

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 of two elements 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.

Here are two examples showing the resonablity of this algorithm:

$$\begin{aligned} \mathrm{List}_x &= <a,3>, <b,3>, <c,3>, <d,2> \\ \mathrm{List}_y &= <a,3>, <b,3>, <c,3>, <d,5> \\ \mathrm{Sim}(\mathrm{List}_x, \mathrm{List}_y) &= 1 - \frac{3}{11+14} = 0.88 \end{aligned} \quad (3)$$

$$\begin{aligned} \mathrm{List}_x &= <a,3>, <b,1>, <c,3>, <d,1> \\ \mathrm{List}_y &= <a,0>, <b,5>, <c,2>, <d,5> \\ \mathrm{Sim}(\mathrm{List}_x, \mathrm{List}_y) &= 1 - \frac{3+4+1+4}{8+12} = 0.4 \end{aligned} \quad (4)$$

here $<a,3>$ means the frequency for a is 3.

This result matches our intuition: the lists in the first example are more similar than in the second example, and we are likely to take the first example as clone, which has a similarity of 0.88.

C. Machine Learning

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. That being said, STCD can also run without training, however the training process will provide a better result.

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 $\frac{100\times 99}{2} = 4950$ pairs. In these pairs, most of them are not clone, so we applied our tool to the Java file, with a low threshold of 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 many more non-clone pairs but studying more of them does not give a better result.

D. User Interface

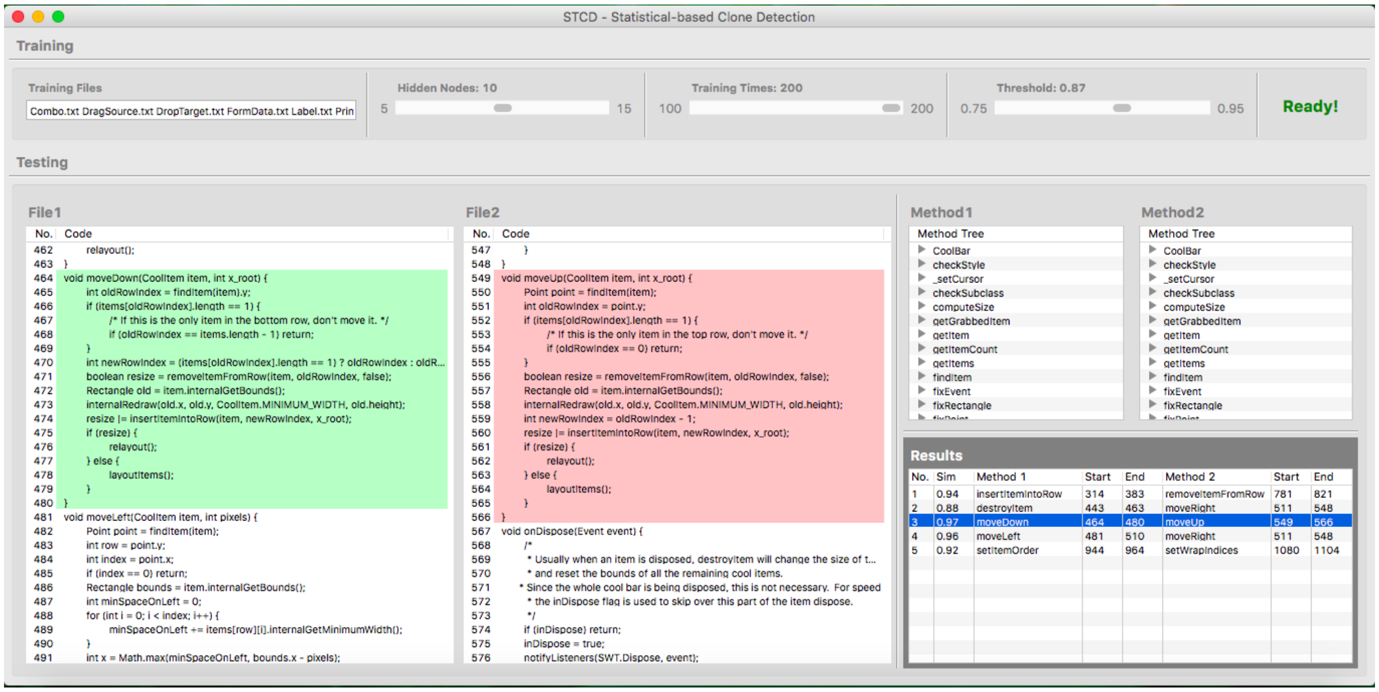
To make STCD more user friendly, we also developed a UI, as shown in Fig:**??**.

DISCUSS MORE ABOUT SWT HERE

E. Instruction of STCD

The step-by-step 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.



Training Files: Combo.txt DragSource.txt DropTarget.txt FormData.txt Label.txt Prin

Hidden Nodes: 10 (5 – 15)

Training Times: 200 (100 – 200)

Threshold: 0.87 (0.75 – 0.95)

File1

```
No.  Code
462         relayout();
463  }
464  void moveDown(CoolItem item, int x_root) {
465         int oldRowIndex = findItem(item).y;
466         if (items[oldRowIndex].length == 1) {
467                /* If this is the only item in the bottom row, don't move it. */
468                if (oldRowIndex == items.length - 1) return;
469         }
470         int newRowIndex = (items[oldRowIndex].length == 1) ? oldRowIndex : oldR...
471         boolean resize = removeItemFromRow(item, oldRowIndex, false);
472         Rectangle old = item.internalGetBounds();
473         internalRedraw(old.x, old.y, CoolItem.MINIMUM_WIDTH, old.height);
474         resize |= insertItemIntoRow(item, newRowIndex, x_root);
475         if (resize) {
476                relayout();
477         } else {
478                layoutItems();
479         }
480  }
481  void moveLeft(CoolItem item, int pixels) {
482         Point point = findItem(item);
483         int row = point.y;
484         int index = point.x;
485         if (index == 0) return;
486         Rectangle bounds = item.internalGetBounds();
487         int minSpaceOnLeft = 0;
488         for (int i = 0; i < index; i++) {
489                minSpaceOnLeft += items[row][i].internalGetMinimumWidth();
490         }
491         int x = Math.max(minSpaceOnLeft, bounds.x - pixels);
```

File2

```
No.  Code
547         }
548  }
549  void moveUp(CoolItem item, int x_root) {
550         Point point = findItem(item);
551         int oldRowIndex = point.y;
552         if (items[oldRowIndex].length == 1) {
553                /* If this is the only item in the top row, don't move it. */
554                if (oldRowIndex == 0) return;
555         }
556         boolean resize = removeItemFromRow(item, oldRowIndex, false);
557         Rectangle old = item.internalGetBounds();
558         internalRedraw(old.x, old.y, CoolItem.MINIMUM_WIDTH, old.height);
559         int newRowIndex = oldRowIndex - 1;
560         resize |= insertItemIntoRow(item, newRowIndex, x_root);
561         if (resize) {
562                relayout();
563         } else {
564                layoutItems();
565         }
566  }
567  void onDispose(Event event) {
568         /*
569          * Usually when an item is disposed, destroyItem will change the size of t...
570          * and reset the bounds of all the remaining cool items.
571        * Since the whole cool bar is being disposed, this is not necessary.  For speed
572          * the inDispose flag is used to skip over this part of the item dispose.
573          */
574         if (inDispose) return;
575         inDispose = true;
576         notifyListeners(SWT.Dispose, event);
```

Method1

Method Tree
▶ CoolBar
▶ checkStyle
▶ _setCursor
▶ checkSubclass
▶ computeSize
▶ getGrabbedItem
▶ getItem
▶ getItemCount
▶ getItems
▶ findItem
▶ fixEvent
▶ fixRectangle

Method2

Method Tree
▶ CoolBar
▶ checkStyle
▶ _setCursor
▶ checkSubclass
▶ computeSize
▶ getGrabbedItem
▶ getItem
▶ getItemCount
▶ getItems
▶ findItem
▶ fixEvent
▶ fixRectangle

Results

No.	Sim	Method 1	Start	End	Method 2	Start	End
1	0.94	insertItemIntoRow	314	383	removeItemFromRow	781	821
2	0.88	destroyItem	443	463	moveRight	511	548
3	0.97	moveDown	464	480	moveUp	549	566
4	0.96	moveLeft	481	510	moveRight	511	548
5	0.92	setItemOrder	944	964	setWrapIndices	1080	1104

Fig. 3. User Interface

IV. Evaluation

A. *Test Result Without Machine Learning*

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 of 0.65 to rule out most of the pairs and manually compare the rest.

Using low threshold to rule out most of the non-clones is reasonable because STCD can also work without training. Apply STCD to test files without training will get the following result:

```
Test Files: Spinner.java
Similarity = 0.98

Line #: 304-313
public void cut () {
        checkWidget ();
        if ((style & SWT.READ_ONLY) != 0) return;
        NSText fieldEditor = textView.currentEditor();
        if (fieldEditor != null) {
                fieldEditor.cut(null);
        } else {
                //TODO
        }
}

Line #: 563-572
public void paste () {
        checkWidget ();
        if ((style & SWT.READ_ONLY) != 0) return;
        NSText fieldEditor = textView.currentEditor();
        if (fieldEditor != null) {
                fieldEditor.paste(null);
        } else {
                //TODO
        }
}

Test Files: Tree.java
Similarity = 0.89

Line #: 368-377
void clear (TreeItem parentItem, int index, boolean all) {
        TreeItem item = _getItem (parentItem, index, false);
        if (item != null) {
                item.clear();
                item.redraw (-1);
                if (all) {
                        clearAll (item, true);
                }
        }
}

Line #: 379-391
void clearAll (TreeItem parentItem, boolean all) {
        int count = getItemCount (parentItem);
        if (count == 0) return;
        TreeItem [] children = parentItem == null ? items : par
        for (int i=0; i<count; i++) {
                TreeItem item = children [i];
                if (item != null) {
                        item.clear ();
                        item.redraw (-1);
                        if (all) clearAll (item, true);
                }
        }
}
```

These two pairs are clones and are found by our tool. After manual inspection, we found that almost all pairs with similarity lower than 0.65 are not clones. Actually, many pairs with similarity between 0.65 and 0.87 are not clones either. So we consider the process of manually finding ground truth is realiable, and the clones ruled out by the similarity of 0.65, i.e. false negative, is small.

B. Test Result With Machine Learning

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. After tranining with MLP, the test result is shown in Table **??**:

Test Files	TP + FN (Actual Clones)	TP + FP (Detected Clones)	TP
Button.java	0	4	0
CoolBar.java	11	12	10
Menu.java	1	0	0
Spinner.java	2	2	2
TabFolder.java	3	2	2
TableItem.java	20	20	18
ToolBar.java	4	4	4
ToolItem.java	3	3	3
Tracker.java	1	1	1
Tree.java	11	13	9

TABLE II
TEST RESULT

Here we used a threshold of 0.87. A lower threshold will result in more "clones" to be detected, which reduces the precision rate, while a higher threshold will result in more false negatives, thus a lower recall rate. After actual testing, a threshold of 0.87 balances the precision and recall rate.

In the table above, the detected clones are very close to actual clones, 4 out of 10 files got exact match.

One thing needs to be mentioned is about *ToolItem.java*, the process of manually finding "ground truth" results in 2 false negatives. At first we decided there is only 1 "actual" clone, but after training and testing, STCD found 3. After checking by eye, we decided the extra 2 pairs are actually clones, they were ruled out earlier by the low threshold of 0.65. This being said, manual selection does result in false negatives, however on the other hand, this implies the training process is effective.

C. Precision and Recall

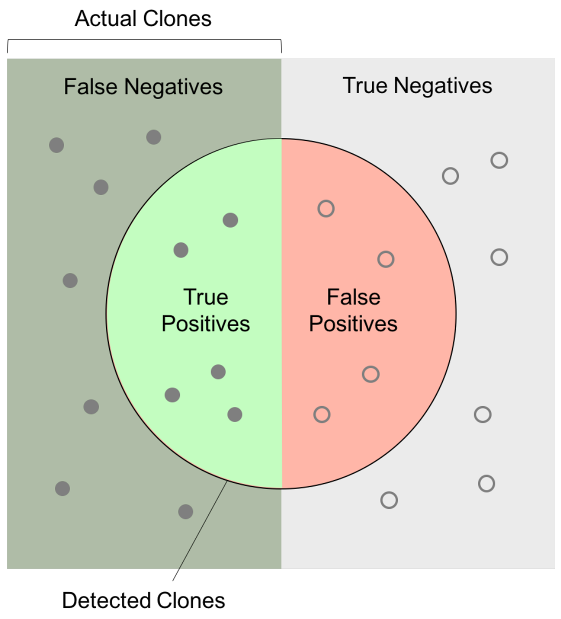


Fig. 4. Calculation of Precision and Recall

Fig:**??** shows the relationship between actual clones and detected clones, based on the definition of Precision and Recall,

$$\text{Precision} = \frac{\text{True Positives}}{\text{Detected Clones}} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (5)$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{Actual Clones}} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (6)$$

Except for *Button.java* and *Menu.java* which we are unable to calculate the Precision and Recall, the other 8 files have Average Precision of 92.75% and Average Recall of 91.25%. The precision and recall for each file is given in Fig:**??**, they are high enough to make STCD practical.

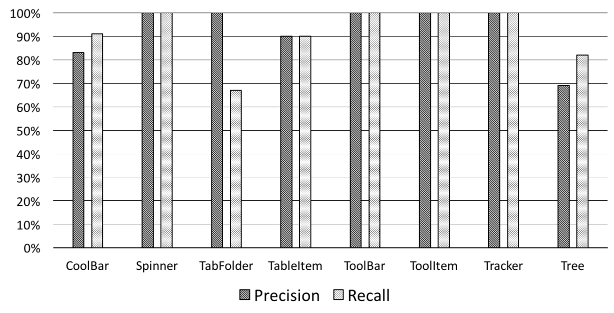


Fig. 5. Precision and Recall of STCD

D. Time Cost Evaluation

The time cost of STCD is also evaluated, and is given in Fig:**??**. This experiment is performed on a Macbook Air, time cost varies with the configuration of computer. 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.

A quadratic time cost is tolerable, especially with the fact that running a file with 100 methods takes only 2 seconds. Most Java files won't have more than hundreds of methods, so our tool is very practical.

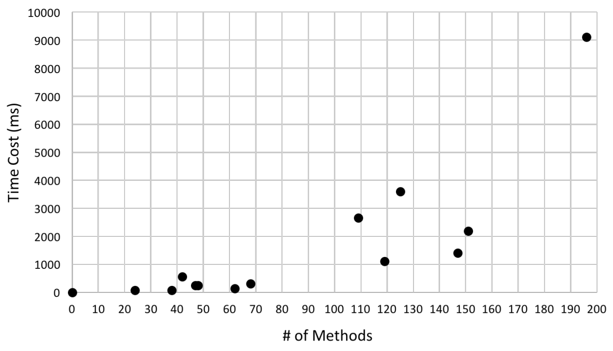


Fig. 6. Time Cost of STCD

V. Validity

Our approach has a high efficiency in finding clones, but there are still a number of threats to the validity:

1) Statistical limitation. STCD is a statiatical based tool, so each method needs to be big enough to get statistical data. In the actual testing, for methods less than 8 lines, the comparision is not reasonable, non-similar pairs can be taken as clones. So during the implementation of STCD, we ignored methods with less than 8 lines to reduce the false positives.
2) Data set too small. Both training data and test data set are small, we selected only 10 files in training and 10 files in testing, with actual clones no more than 20 in each file.
3) Data is chosen only from SWT. Both training data and test data are chosen from SWT, the good part of doing so is that machine learning can learn the developer's behavior, however on the other hand, it may not work so well for other projects. Training with user-defined files may help, but we don't know if the test result is as good as in SWT.
4) False negatives caused by manual selection. As discussed in the test results, by ruling out most unrelated comparisions with a low threshold, we got 2 actual clones ruled out, i.e. false negatives. It is unsure if there are more false negatives like this, so our result may differ from the truth.
5) Ambiguous match. Ambiguous match is realized by threshold. First is the shreshold of 0.7 used in the pretreatment of variable names. If the variable names are more similar than 0.7, they are considered the same and compared. However this may not be true. By a threshold of 0.7 we may either put unrelated variables together, or take similar variables as different.
   Second is the final threshold of 0.87. A lower threshod will result in a lower precision while a higher threshold will result in a low recall. But that is not absolute. It may either rule out actual clones or print out unrelated pairs.
6) Extra time caused by ASTParser Tool. In the first step of implementation, method declaration, we used ASTParser Tool to divide the file into methods and get the information for each method. ASTParser Tool is based on Abstract Syntax Trees, by transforming the original file into trees, extra time cost is introduced, which is not needed. A simpler method can be used to reduce this part of time.

VI. Conclusion

In this paper, we implemented a statistical-based code clone detector for Java files. As a counting based clone detector, it is simple and fast with a quadratic time cost. With the size being small, it maintains high precision and recall of over 90%.

To improve the detecting efficiency, we used machine learning to find the most proper weights. Machine learning can learn the developer's behavior, so users can use their own codes to train the tool. Its enhancement of efficiency is proved by actual testing.

Using SWT to develop a user interface makes STCD more user-friendly.

Overall in this paper, we built a MLP embeded framework to detect code clones in Java files, based on token frequency. It is efficient and aggressive in clone detection field.

Our future work includes the improvement of time cost by replacing ASTParser Tool with more simple methods, as well as enlarge the training and test data set to validate the result, and compare with other tools.

VII. Related Work

A. Types of Detection Tool

Based on the four 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.

Among all the techniques and tools, token-based method performs well in both clone type detection and time cost.

B. Token-based Detection Tool

Many token-based methods have been developed, like CCFinder[**?**], CMCD[**?**], Boreas[**?**], RTF[**?**].

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.

C. Similarity Algorithm

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 does not fit for the token frequency case. For example, if two vectors have components $< a, 1 >, < b, 2 >$ and $< a, 2 >, < b, 4 >$ respectively, then this algorithm will give a similarity of 1, which is not true.

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.

The similarity is given by:

$$Sim(v_a, v_b) = \frac{1}{1 + ED(v_a, v_b)} \tag{7}$$

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.

Take the example in our implementation:

$$\begin{aligned}
\text{List}_x &= < a, 3 >, < b, 3 >, < c, 3 >, < d, 2 > \\
\text{List}_y &= < a, 3 >, < b, 3 >, < c, 3 >, < d, 5 > \\
\text{Sim}(\text{List}_x, \text{List}_y) &= \frac{1}{1 + 3} = 0.25
\end{aligned} \tag{8}$$

$$\begin{aligned}
\text{List}_x &= < a, 3 >, < b, 1 >, < c, 3 >, < d, 1 > \\
\text{List}_y &= < a, 0 >, < b, 5 >, < c, 2 >, < d, 5 > \\
\text{Sim}(\text{List}_x, \text{List}_y) &= \frac{1}{1 + 6.48} = 0.13
\end{aligned} \tag{9}$$

Intuitatively we'll take the first case as clone, but the actual similarity is only 0.25.

Thus a similarity related to Levenshtein Distance is designed in our work.

Reference: AnDarwin, MLP, LD, Precision Recall, Benchmark, machine learning CCD, bigram

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