Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Classification Model for Code Clones Based on Machine Learning

Jiachen Yang · Keisuke Hotta ·
Yoshiki Higo · Hiroshi Igaki ·
Shinji Kusumoto

Received: date / Accepted: date

Abstract Results from code clone detectors may contain plentiful useless code clones, but judging whether each code clone is useful varies from user to user based on a user's purpose for the clone. In this research, we propose a classification model that applies machine learning to the judgments of each individual user regarding the code clones. To evaluate the proposed model, 32 participants completed an online survey to test its usability and accuracy. The result showed several important observations on the characteristics of the true positives of code clones for the users. Our classification model showed more than 70% accuracy on average and more than 90% accuracy for some particular users and projects.

Keywords filtering · classify · machine learning · code clone detector ·

1 Introduction

Great efforts have been made to detect identical or similar code fragments from the source code of software. These code fragments are called "code clones" or simply "clones", as defined in Higo et al (2008b) and Roy et al (2009). During development, code clones are introduced into software systems by various operations, namely, copy-and-paste or machine generated source code. Because code cloning is easy and inexpensive, it can make software development faster, especially for "experimental" development. However, despite their common occurrence in software development, code clones are generally considered harmful,

Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {jc-yang,k-hotta,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

as they make software maintenance more difficult and indicate poor quality of the source code. If we modify a code fragment, it is necessary to check whether each of the corresponding code clones needs simultaneous modifications. Therefore, various techniques and tools to detect code clones automatically have been proposed by many researchers, such as Johnson (1994), Baxter et al (1998) and Li et al (2006). Furthermore, these code clone detectors were studied by Bellon et al (2007), who defined the widely used categories of clones based on the following similarities:

Type-1 An exact copy of a code fragment except for white spaces and comments in the source code.
Type-2 A syntactically identical copy where only user-defined identifiers such as variable names are changed.
Type-3 A modified copy of a type-2 clone where statements are added, removed or modified. Also, a type-3 clone can be viewed as a type-2 clone with gaps in-between.

By applying code clone detectors to the source code, users such as programmers can obtain a list of all code clones for a given code fragment. Such a list is useful during modifications to the source code. However, results from code clone detectors may contain plentiful useless code clones, and judging whether each code clone is useful varies from user to user based on the user's purpose for the clone. Therefore, it is difficult to adjust the parameters of code clone detectors to match the individual purposes of the users. It is also a tedious task to analyze the entire list generated by the code clone detector. Clone detection tools (CDTs) usually generate a long list of clones from source code. A small portion of these clones are regarded as true code clones by a programmer, while the others are considered as false positives that occasionally share an identical program structure.

A previous study Krinke (2008) suggested that cloned code fragments are less modified than non-cloned code fragments. This result indicates that more clones in existing source code are stable instead of volatile. These stable clones should be filtered from the result of CDTs before applying simultaneous modifications to the volatile ones. Meanwhile, Bellon et al (2007) manually created a set of baseline code clones to evaluate several CDTs. Manual baseline clones have the following issues.

– They demand a huge amount of manpower.
– Their accuracy depends on the person who made the judgments.

Several methods have been proposed to filter out unneeded clones, including the metric-based method described in Higo et al (2008a). While these methods can summarize a general standard of true code clones, the users must have professional knowledge, such as what all the metrics mean. Furthermore, it is hard to fit the extraction method of refactoring to individual use cases, such as filtering only the code clones. Tairas and Gray proposed a classification method in Tairas and Gray (2009), which classifies the code clones by identifier names rather than the text similarity of identical clone fragments. The purpose of our study is the same as those of filtering or classification methods: to find true positives from the result of CDTs during software maintenance. However, we take a different approach to find the true positives.

Finding clones that are suitable for refactoring has also been studied. Previous studies such as Higo et al (2002) and Rysselberghe and Demeyer (2004) suggested that the clone detection process should be augmented with semantic information in order to be useful for refactoring. Tiarks et al (2009, 2011) investigated several code characteristics from different clone detection tools for type-3 clones. Their studies pointed out that clone detection tools may improve the results with feedback from the users, and that the best metrics for different datasets include source code similarity, token value similarity and identifier similarity, all of which are measured by the edit distance. Our research is based on their observations, though we compare the similarity of the token sequences by a more efficient approach and target the removal of unwanted type-2 code clones.

According to a survey we conducted, users of CDTs tend to classify clones differently based on each user's individual uses, purposes or experience about code clones. A true code clone for one user may be a false code clone for another user. From this observation, we propose the idea of studying the judgments of each user regarding clones. The result is a new clone classification method, entitled Filter for Individual user on code Clone Analysis (Fica).

The code clones are classified by Fica according to a comparison of their token type sequences through their similarity, based on the "term frequency – inverse document frequency" (TF-IDF) vector. Fica learns the user opinions concerning these code clones from the classification result. In a production environment, adapting the method described in this research will decrease the time that a user spends on analyzing code clones.

From the experimental results, we made several observations:

1. Users agree that false positive code clones are likely to fall into several categories, such as a list of assignment statements or a list of function declarations.
2. Users agree that true positive code clones are more diverse than false positives.
3. Generally, the minimum required size of the training set grows linearly with the number of categories that clone sets fall into, which is less than the total number of detected clone sets.

In the paper, Section 2 introduces a motivating example that led to this research. Section 3 introduces the working process for the proposed method, Fica, and describes how it could help the user to classify code clones. Then,

```
wcstr = 0;
slen = mbstowcs (wcstr, s, 0);
if (slen == -1)
  slen = 0;
wcstr = (wchar_t *)xmalloc (sizeof (wchar_t) * (slen + 1));
mbstowcs (wcstr, s, slen + 1);
wclen = wcswidth (wcstr, slen);
free (wcstr);
return ((int)wclen);
```

Fig. 1: execute_cmd.c in bash-4.2

Section 4 discusses in detail the proposed method and algorithms that Fica uses. After that, Section 5 shows the result of an online survey that we conducted to evaluate our proposed method. Finally, we discuss other related works in the classification of code clones in Section 6 and conclude the contributions of this paper in Section 7.

2 Motivating Example

We conducted a survey with several students[1]. We provided the students with 105 clone sets detected from `bash-4.2` from the result of CDTs on source code in the C language, then asked them whether these code clones were true code clones, based on their own experience and motivation. Table 1 shows a part of the result. In this table, a code clone set is marked as `O` if a student thought this clone is a true code clone or as `X` if not. We can see from this table that the attitudes toward these clones varied from person to person.

[1] All students are from the Graduate School of Information Science and Technology, Osaka University

Table 1: Survey of clones in bash-4.2 (O: true code clone, X: false code clone)

Clone ID	Participant			
	Y	S	M	U
1	X	O	O	O
2	X	X	O	O
3	O	X	X	O
4	X	O	X	O
5	X	O	X	O
...	...			
O count	5	24	23	25
X count	100	81	82	80

```
if (wcharlist == 0)
{
  size_t len;
  len = mbstowcs (wcharlist, charlist, 0);
  if (len == -1)
    len = 0;
  wcharlist = (wchar_t *)xmalloc (sizeof (wchar_t) * (len + 1));
  mbstowcs (wcharlist, charlist, len + 1);
}
```

Fig. 2: subst.c in bash-4.2

As an example, the source of a clone with ID 5 is shown in Figures 1 and 2. These blocks of code convert a multi-byte string to a wide-char string in C code. Because their functions are identical, S and U are considered able to be merged, so these clones are true code clones. Meanwhile, Y and M considered the fact that Figure 2 is a code fragment in a larger function having more than 100 LOCs. Because it may be difficult to apply refactoring, these clones should be false positives of the CDT.

Moreover, from Table 1 we can see that Y was more strict than the three other students. In the comment to this survey, Y mentioned that only clones that contain an entire body of a C function are candidates. This unique standard was also reflected in all five true clones he chose.

Based on the above motivating survey, we feel that a classifier based on the content of source code is necessary during a clone analysis to meet the individual usage of each user. Therefore, we developed the classification method described in the following sections.

3 Fica System

In this section, we introduce the general working process of the Fica system, which ranks detected clones based on the historical behavior of a particular user. As a complement to existing CDTs that filter unexpected clones, Fica is designed as an online supervised machine learning system, which means that the user should first classify a small portion of the input manually, and then Fica gradually adjusts its prediction while more input is given by the user. By adapting this process into a traditional code clone analyzing environment, the desired code clones could be presented to the user more quickly.

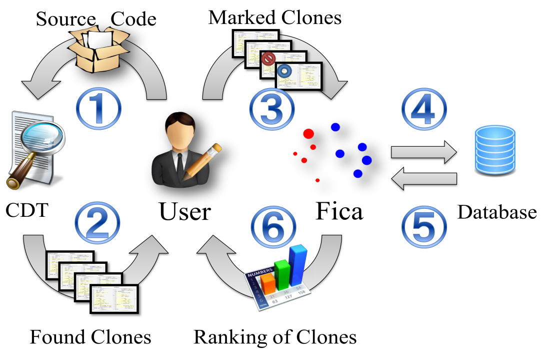


Fig. 3: Overall workflow of FICA with a CDT

3.1 Overall Workflow of FICA System

The overall workflow of FICA is described in Figure 3 and listed as follows.

1. A user submits source code to a CDT.
2. The CDT detects a set of clones in the source code.
3. The user marks some of these clones as true or false clones according to her/his own judgment, and then submits these marked clones to FICA as a profile.
4. FICA records the marked clones in its own database.
5. Meanwhile, FICA studies the characteristics of all these marked clones by using machine learning algorithms.
6. FICA ranks other unmarked clones based on the result of machine learning, which predicts the probability that each clone is relevant to the user.
7. The user can further adjust the marks on code clones and resubmit them to FICA to obtain a better prediction. FICA will also record these patterns that it has learned in a database associated with the particular user profile so that further predictions can be made based on earlier decisions.

The main purpose of the FICA system is to find the true code clones from the result of a CDT, or in reverse, to filter out the false code clones, by using a classification method. As we have shown in the motivating example, whether a code clone is true or false largely depends on the user's subjective purpose. Our FICA classifier is designed to consider this subjective factor and to be helpful regardless of different users.

3.2 Classification Based on Token Sequence of Clones

Fica utilizes a CDT to obtain a list of code clones from source code. The resulting output from the CDT should contain the following information for each code clone:

1. Positions of the code clone fragment in the original source code.
2. A sequence of tokens included in the code clone fragment.

Most existing token-based CDTs meet this requirement. Text-based CDTs usually output only the positional information of code clones; in this case, code clone fragments need to be parsed through a lexical analyzer to obtain the required lexical token sequences.

Fica compares the exact matched token sequence of reported clone sets from CDTs. These clones are known as type-2 clones in other literature. Type-2 clone instances from a clone set should have the same token sequence. Fica compares the similarity of the token sequences for the following reasons:

– The similarity of the token sequences was pointed out by Tiarks et al (2009) as one of the best characteristics for filtering out unneeded clones, which are referred to as rejected candidates in that study.
– The types of each token in the token sequence are identical among all instances of a type-2 clone set.
– The required token sequence is obtainable as a side-product of a token-based CDT.

By using the token sequence from CDT, Fica can save the time of reparsing the source code, which should have been done by the CDT. Therefore, Fica can be integrated with the CDT as a complementary system.

3.3 Marking Clones Manually

To be used by Fica as an initial training set, only a small set of clones found by CDT is required to be marked manually by the user of Fica. The considered types of marks on clones can be Boolean, numerical, or tags:

– Boolean clones are marked based on whether they are true code clones.
– Range clones are marked with a numerical likelihood to be true code clones.
– Tagged clones are marked by one of several tags or categories by the users based on their use, such as refactoring, issue tracking ID, etc.

As the most simple case, users need to tell Fica what kind of clones should be treated as true code clones. Numerical type marks are an extension of Boolean type marks used in the situation that the user wants to say finding clone A is more useful than finding clone B but less useful than finding clone C. Tag type marks can be considered as a possible extension of Boolean type ones that involve multiple choices. For example, clones marked with the tag `refactoring` are suitable candidates for refactoring, or clones marked with the tag `buggy` are clones that tend to have bugs.

Also, a Fica user is allowed to have multiple profiles in the system, with each profile representing a use case of code clones. Profiles should be trained separately and Fica will treat them as individual users.

3.4 Machine Learning

Receiving the clones from the CDT and the marks from the user, Fica studies the characteristic of the marked clones by calculating the similarity of the lexical token sequence of these clones. This step employs a machine learning algorithm that is widely used in natural language processing or text mining. The algorithm used is similar to that used in GMail for detecting spam emails or the CJK Input Methods used in suggesting available input candidates. By comparing the similarity of marked and unmarked clones, Fica can thus predict the probability of an unmarked clone set being a true positive. Details on the machine learning model and algorithm are described in Section 4.

3.5 Cycle of Supervised Learning

Fica returns the predicted marks for all remaining clones by ranking or calculating the probability of the user considering them as true clones. The user is allowed to correct some of these predictions and resubmit them to Fica to obtain a better prediction. This is known as the cycle of supervised learning. Eventually, Fica is trained to filter all clones according to the interest or purpose of the user. Furthermore, the patterns learned by Fica are also recorded in a database associated with the particular user. As a result, further predictions on clones can be made based on earlier decisions of the same user.

4 Machine Learning Method

The classification process in Fica is described in Figure 4. Fica can be viewed as a supervised machine learning process with these steps:

1. Retrieving a list of clone sets by a predefined order from a CDT.
2. Calculating the text similarity among those clone sets by their cosine similarity in TF-IDF vector space.

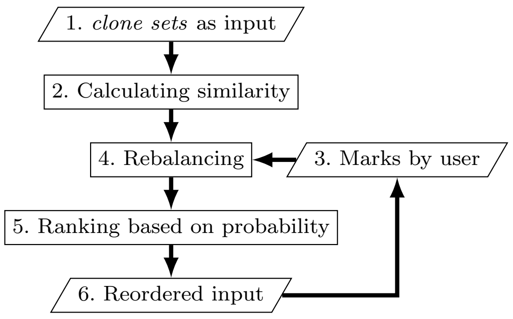


Fig. 4: Classification process in Fica. Parallelograms represent input data or output data, and rectangles represent the procedures of Fica.

3. Receiving the true or false clone marks from the user as training sets of machine learning.
4. Rebalancing the input set in order to get identical data points in each set of the training sets.
5. Calculating the probability for each clone set of the different classifications in the marked groups.
6. Ranking and reordering the clone sets based on their probabilities.

The user can adjust the marks that Fica has predicted and submit the marks again to Fica, as in step 3, which then forms the supervised machine learning cycle.

4.1 Input Format of Fica

As a support tool for CDTs, Fica needs to extract code clone information from the output of the CDTs. The structure of all the information needed by Fica is represented as the UML diagram in Figure 5.

A `project` consists of the original source code and the detected code clones represented as `clone sets`. The source code fragments are tokenized by the CDT. As Fica needs both tokenized source code for calculating the similarity and the original source code for presenting back to the user, these two forms are passed together to Fica. A clone set is a set of identical code fragments. By identical, we mean the tokenized code fragments are literally equal to each other in a `clone set`. We are targeting type-2 code clones; therefore, all the token types in a clone set are identical.

A `clone` in a `clone set` is the tokenized code clone fragment in the original source file in the project, The list of token types in the clone set should be

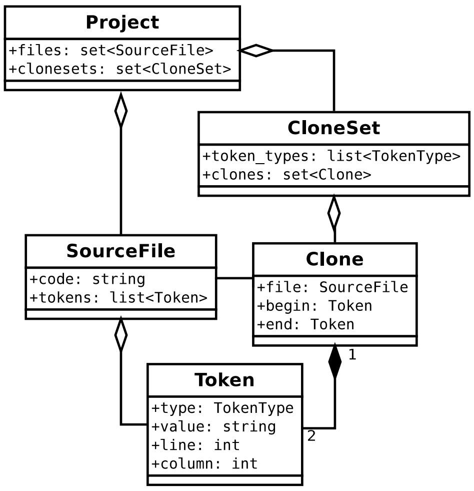


Fig. 5: Structure of input to FICA

equivalent to the types of tokens in every clone in the clone set. A token has a type and a position in the original source code. An example of how the input of FICA looks like is in Figure 6.

4.2 Calculating Similarity between Clone Sets

FICA calculates the TF-IDF vector Jones (1972) of clone sets, which is widely used in machine learning techniques in natural language processing. In FICA we define `term` t as an n-gram of a token sequence, `document` d as a `clone set`, and all documents `D` as a set of `clone sets` that can be gathered from a single `project` or several `projects`.

A previous study by Tiarks et al (2011) suggested that the similarity of token sequences is one of the best characteristics for identifying false candidates from type-3 clones. Similar to their study, we also need to calculate the similarity among clones. Therefore, we use a metric from the token sequence to capture syntactic information. In contrast to their study, we compare similar type-2 clones. Therefore, we need a metric that can tolerate structural reordering of source code, such as reordering of statements. Based on these two objectives, we use n-grams of the token sequences.

First, we divide all token sequences in clone sets into n-gram terms. Let us assume we have a clone with the following token types:

STRUCT ID TIMES ID LPAREN CONST ID ID TIMES ID RPAREN ...

```
Project: git-v1.7.9
files:
  bloc.c
    const char *blob_type = "blob";
    struct blob *lookup_blob(const ...
    ...
  tree.c
    const char *tree_type = "tree";
    static int read_one_entry_opt( ...
    ...
  builtin/fetch-pack.c
  builtin/receive-pack.c
  builtin/replace.c
  builtin/tag.c
  ...
clonesets:
  1: token_types: STRUCT ID TIMES ID LPAREN CONST ...
     clones:
       blob.c (6:1)-(23,2)
       tree.c (181,1)-(198,2)
  2: token_types: ID RPAREN SEMI RETURN INT_CONST ...
     clones:
       builtin/replace.c (39,48)-(57,10)
       builtin/tag.c (154,45)-(172,10)
  ...
```

Fig. 6: Example of input format for FICA

If we assume that N for n-gram equals 3, then we can divide this clone into n-grams as:

```
STRUCT ID TIMES
       ID TIMES ID
          TIMES ID LPAREN
                ID LPAREN CONST
                   LPAREN CONST ID
                          CONST ID ID
                                ID ID TIMES
                                   ...
```

Then we calculate all the `term frequency` values of n-grams within this document of clone set by Equation 1.

$$\mathrm{TF}(t,d) = \frac{|t : t \in d|}{|d|} \tag{1}$$

Equation 1 states that `term frequency` of `term` t in `document` d is the normalized frequency, where `term` t appears in `document` d. The result of TF of the above `clone set` is as follows:

```
RETURN ID SEMI        :0.00943396226415
ID RPAREN LBRACE      :0.0283018867925
SEMI RETURN ID        :0.00943396226415
ID SEMI IF            :0.00943396226415
ID LPAREN CONST          :0.00943396226415
TIMES ID RPAREN          :0.00943396226415
IF LPAREN LNOT           :0.0188679245283
RPAREN LBRACE ID         :0.00943396226415
LPAREN RPAREN RPAREN     :0.00943396226415
SEMI RBRACE RETURN       :0.00943396226415
RETURN ID LPAREN         :0.00943396226415
ID ID RPAREN             :0.00943396226415
TIMES ID COMMA           :0.0188679245283
...
```

Analogously, the `inverse document frequency` IDF and TF-IDF are calculated by Equations 2 and 3.

$$\mathrm{IDF}_D(t) = \log \frac{|D|}{1 + |d \in D : t \in d|} \tag{2}$$

$$\text{TF-IDF}_D(t,d) = \mathrm{TF}(t,d) \times \mathrm{IDF}_D(t) \tag{3}$$

$$\overrightarrow{\text{TF-IDF}_D d} = [\text{TF-IDF}_D(t,d) : \forall t \in d] \tag{4}$$

Equation 2 states that the textttinverse document frequency *idf* for texttttterm t in all `documents` D is the logarithm of the total number of `documents` divided by the number of `documents` containing `term` t. In Equation 2, we add 1 to the denominator to avoid division by zero. By combining tf and *idf* as Equation 3, we can then calculate the vector space $\overrightarrow{\text{TF-IDF}(d, D)}$ as in Equation 4 for each clone set in the overall documents.

By using TF-IDF, we define the cosine similarity $\mathrm{CosSim}_D(a,b)$ of two clone sets, a and b, with regard to a set of documents D, as in Equation 5 and 6.

$$\mathrm{Sim}_D(a,b) = \overrightarrow{\text{TF-IDF}_D a} \cdot \overrightarrow{\text{TF-IDF}_D b} \tag{5}$$

$$\mathrm{CosSim}_D(a,b) = \begin{cases} 0, & \mathrm{Sim}_D(a,b) = 0 \\ \frac{\mathrm{Sim}_D(a,b)}{\left|\overrightarrow{\text{TF-IDF}_D a}\right| \cdot \left|\overrightarrow{\text{TF-IDF}_D b}\right|}, & \text{otherwise} \end{cases} \tag{6}$$

After calculating the similarity among all clone sets within a project, we can then plot them based on the force-directed algorithm described in Kobourov (2012) to show the similarity among code clones. As an example, Figure 7a illustrates the clone sets grouped by their similarity. The details of the force-directed graph (FDG) are discussed in Subsection 5.3.

4.3 User Profile and Marks on Clone Sets

A profile represents a user's preferences with regard to the classification of clones stored in Fica. A user is allowed to keep multiple profiles for different use cases of

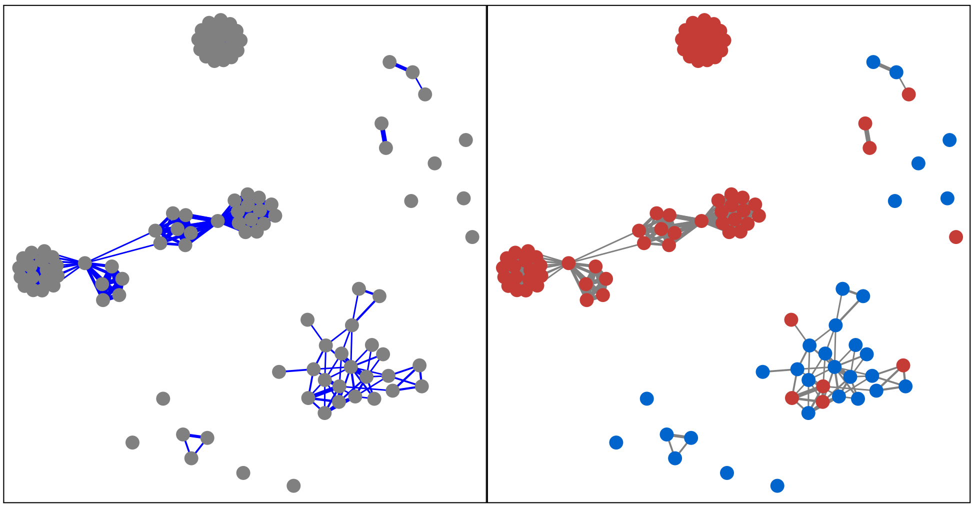

(a) clone sets grouped by similarity (b) Colored force-directed graph

Fig. 7: Force-directed graph of clone sets of bash 4.2. Blue circles mean true code clones and red means false code clones.

Fica. The implementation of Fica does not distinguish a user from the profiles, so for simplicity, the remainder of this paper assumes that every user has a single profile, and Fica needs an initial training set to provide further predictions.

A mark is made on each clone set by the user. Marks, which indicate this clone set has certain properties, are recorded in the user profile. We considered three kinds of marks that users can make on these clone sets. Boolean marks flag a clone set that the user thought to be true or false clones. Numerical marks represent a weight, or importance, assigned by each user, saying that clone set A is more important than clone set B. Tag marks are not exclusive, which is useful in multi-purpose code clone analysis for classifying the clones into multiple overlapping categories.

A training set is a set of clone sets that have been marked by the user. For Boolean marks, the training set M is divided into two sets of clone sets, M_t and M_f, each of which donates a true set and a false set, where $M_t \cap M_f = \emptyset, M_t \cup M_f = M$. For range marks, there is only one group of training sets, M, and each clone set m in M is associated with a weight $w(m)$. For the tag marks, several groups of clone sets are in the form of M_x, with each set donating a different tag. The probability calculation is the same. These groups can overlap each other, and doing so does not affect the result of the calculation.

For each clone set t that has not been marked, FICA calculates the probability that this clone set t should also be marked in the same way as those in clone set group M_x by using Equation 7.

$$\mathrm{Prob}_{M_x}(t) = \begin{cases} 1, & \sum\limits_{\forall m \in M_x} w(m) = 0 \\ \dfrac{\sum\limits_{\forall m \in M_x} \mathrm{CosSim}_{M_x}(t, m) \cdot w(m)}{\sum\limits_{\forall m \in M_x} w(m)}, & \text{otherwise} \end{cases} \tag{7}$$

Here, $w(m), m \in M_x$ is a weighting function for marked clone set m in clone set group M_x. For Boolean or tag marks, we defined $w(m) = 1$; thus, $\sum_{\forall m \in M_x} w(m) = |M_x|$, which is the size of clone set group M_x. For numerical range marks, the weighting function $w(m)$ is defined as a normalized weight in $[0, 1]$.

As expressed in Equation 7, the probability that a clone set t should also be marked as M_x is calculated by the average normalized similarity of t and every clone set in the clone set group M_x. For predicting Boolean marks, FICA can compare the calculated probability of the clone being a true or false clone, and choose the higher one as the prediction. For range marks, the average normalized similarity is multiplied by $w(m)$ from the user's input. For tagged marks, FICA needs to set a threshold for making predictions.

FICA makes predictions based on the probability and ranks all the code clones by the value calculated from $\mathrm{Prob}_{M_x}(t)$. For Boolean marks, the ratio of the probability for the true clone set and the false clone set, $\frac{\mathrm{Prob}_{M_T}(t)}{\mathrm{Prob}_{M_F}(t)}$, is used as a ranking score. For tagged marks, the probabilities for each individual tag are ranked separately, and the tag with the highest probability is chosen as the prediction result. Range marks have only one group of marks. Therefore, the probability $\mathrm{Prob}_M(t)$ is used directly for ranking.

A special case occurs when the sum of the weighting functions is equal to zero; in this case, we define the probability to be 1. This special case occurs when no such data is chosen in clone set group M_x. An arbitrary value is possible in this case as we have no learning data, but we chose 1 to prevent division by zero in calculating the ratio between probabilities.

After the prediction, all clone sets are marked either by the prediction of FICA or by the user's input. An example of all Boolean marked clone sets in the above force-directed graph is shown in Figure 7b. The result of the FICA prediction is presented to the user, so that the user can check its correctness, approve some or all of FICA's prediction or correct other predictions as needed.

After correcting part of the result, the user can resubmit the marks to Fica to get a better prediction.

5 Experiments

5.1 Implementation Details

Fica was implemented as a proof-of-concept system. We implemented the described algorithms for computations of the similarity of code clones, as well as a customized token-based CDT that outputs exactly what Fica requires. Also, we wrapped the CDT part and Fica together by using a web-based user interface, which is referred to as the Fica system from now on[2].

The Fica system manages the profiles of users as normal login sessions, like those in most websites. One of the users uploads an archive of source code to the Fica system. Then Fica unzips the source code into a database and then passes the code to the CDT part of the Fica system.

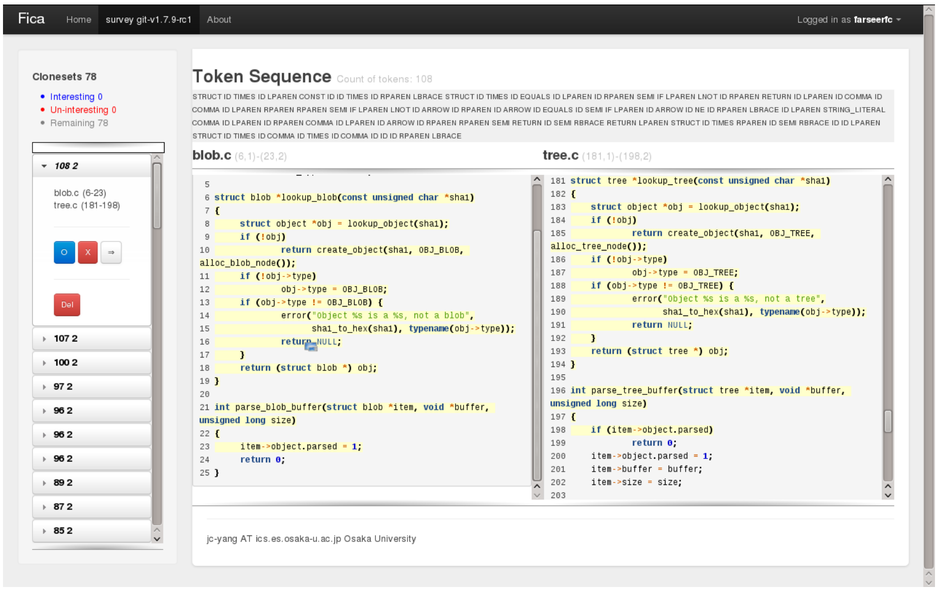
The CDT part of the Fica system implements the algorithm for constructing a *generalized suffix tree* described by Ukkonen (1995) and the algorithm of the detection of clones among multiple files in the *generalized suffix tree.* Code clones and types of token sequences are recorded in a database.

Then the Fica system shows a comparison view of detected clone sets to the user, as shown in Figure 8. The user can mark Boolean tags on these clone sets, and the Fica system stores these marks immediately in the database associated with the user profile. While the user is marking those clone sets, the Fica system calculates the similarity among those clone sets and trains its prediction model by including the user input into its training set on the server side in the background. As a result, the feedback of user inputs in the Fica system can be gained in nearly real time.

5.2 Experimental Setup

To test the validity and user experience of the proposed method, the following experiment was conducted. We uploaded the source code of four open source projects as experimental targets, as shown in Table 2. For all the projects in Table 2, we only included `.c` files and ignored all other files as unnecessary because all four projects were developed in C language. As the parameters passed to the CDT part of the fica system, we only detected clone sets that contained more

[2] Fica with experimental data can be accessed here: `http://valkyrie.ics.es.osaka-u.ac.jp`



Clonesets 78

- Interesting 0
- Un-interesting 0
- Remaining 78

108 2

blob.c (6-23)
tree.c (181-198)

107 2
100 2
97 2
96 2
96 2
96 2
89 2
87 2
85 2

Token Sequence Count of tokens: 108

STRUCT ID TIMES ID LPAREN CONST ID ID TIMES ID RPAREN LBRACE STRUCT ID TIMES ID EQUALS ID LPAREN ID RPAREN SEMI IF LPAREN LNOT ID RPAREN RETURN ID LPAREN ID COMMA ID COMMA ID LPAREN RPAREN RPAREN SEMI IF LPAREN LNOT ID ARROW ID RPAREN ID ARROW ID EQUALS ID SEMI IF LPAREN ID ARROW ID NE ID RPAREN LBRACE ID LPAREN STRING_LITERAL COMMA ID LPAREN ID RPAREN COMMA ID LPAREN ID ARROW ID RPAREN RPAREN SEMI RETURN ID SEMI RBRACE RETURN LPAREN STRUCT ID TIMES RPAREN ID SEMI RBRACE ID ID LPAREN STRUCT ID TIMES ID COMMA ID TIMES ID COMMA ID ID ID RPAREN LBRACE

blob.c (6,1)-(23,2)

```

struct blob *lookup_blob(const unsigned char *sha1)
{
	struct object *obj = lookup_object(sha1);
	if (!obj)
		return create_object(sha1, OBJ_BLOB,
alloc_blob_node());
	if (!obj->type)
		obj->type = OBJ_BLOB;
	if (obj->type != OBJ_BLOB) {
		error("Object %s is a %s, not a blob",
		      sha1_to_hex(sha1), typename(obj->type));
		return NULL;
	}
	return (struct blob *) obj;
}

int parse_blob_buffer(struct blob *item, void *buffer,
unsigned long size)
{
	item->object.parsed = 1;
	return 0;
}
```

tree.c (181,1)-(198,2)

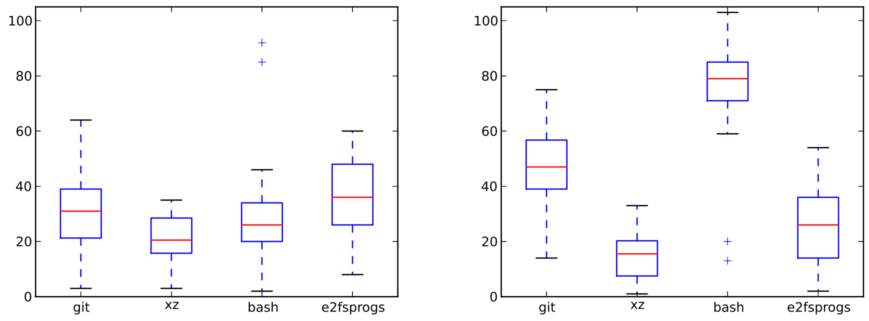
```
struct tree *lookup_tree(const unsigned char *sha1)
{
	struct object *obj = lookup_object(sha1);
	if (!obj)
		return create_object(sha1, OBJ_TREE,
alloc_tree_node());
	if (!obj->type)
		obj->type = OBJ_TREE;
	if (obj->type != OBJ_TREE) {
		error("Object %s is a %s, not a tree",
		      sha1_to_hex(sha1), typename(obj->type));
		return NULL;
	}
	return (struct tree *) obj;
}

int parse_tree_buffer(struct tree *item, void *buffer,
unsigned long size)
{
	if (item->object.parsed)
		return 0;
	item->object.parsed = 1;
	item->buffer = buffer;
	item->size = size;

```

jc-yang AT ics.es.osaka-u.ac.jp Osaka University

Fig. 8: Fica showing clone sets

than 48 tokens, and only reported those clone sets from different files. These two parameters were chosen based on experience. Namely, the length limitation less than 48 for C projects is more likely to result in small false positives, as well as for clones from the same source code file. Although too many clone sets are an obstacle to conducting this experiment, those small false positives should be detectable in Fica as well. The CCFinder, one of the well-known CDTs proposed in Kamiya et al (2002), uses a length of 30 tokens as a default parameter. This length is widely accepted in both industrial and academic studies, while its definition of tokens is compacted. For example, CCFinder treats an `IF` followed with a `LEFT_BRACE` as one compressed token, whereas our CDT considers them as two tokens. Based on our experience, a length limit of 48 tokens generates almost the same amount of code clones as CCFinder.

Altogether, 32 users participated in this experiment. Each user had different experiences about code clone detection techniques. They were required to mark those clone sets found by the CDT built in the Fica system when using Boolean marks as true or false clones. The users were told the steps of the experiments but not the internal techniques used in the Fica system. Consequently, they were not aware that the Fica system compares clone sets by the text similarity of lexical



(a) # of True Clones (b) # of False Clones

Fig. 9: # of true and false clones from the selected data of users

token types. Also, we informed the participants that the experiment would take approximately 2 hours to complete, with approximately 30 seconds per clone set.

As the order of the clone sets affects the prediction result of the system, we disabled the reordering feature of the Fica system. The users were presented with the list of clone sets in a fixed order from the original results of the CDT. This fixed order is determined by the suffix tree algorithm used in the CDT and appears to be random from the user's perspective. The number of true code clones and false code clones selected by the users are shown as box-plots in Figure 9.

The experiment was conducted on the Internet through the Fica web interface. Among the 32 participants, 13 were graduate students or professors in our school, 7 were from clone-related research groups in other universities, while the others were in industry and interested in clone related research. All of these participants are or were majors in computer science or software engineering. Regarding their programming language preferences, almost all of the participants used Java as one of their main languages, while 11 also mainly used C. Furthermore, we

Table 2: Open source projects used in experiments

Project & Version	# of clone sets	LOC	Tokens	Files	# of Participants
git v1.7.9-rc1	78	153,388	829,930	315	32
xz 5.0.3	36	25,873	113,894	113	27
bash 4.2	105	133,547	494,248	248	25
e2fsprogs 1.41.14	62	99,129	442,978	274	25
Total	281	411,937	1,881,050	950	32

ensured that none of the participants ever contributed to the development of the four target software projects.

We used 5-grams throughout the experiments. The size of the n-gram was selected based on our experience. It is certain that increasing the N value improves the theoretical accuracy of the algorithm. Theoretically, the upper bound size of the possible types of n-grams grows exponentially with the N value, although in practice this size is also limited by the size of input. Therefore, increasing the value of N increases the calculation time accordingly. From our experience of the implementation, any N value greater than 3 generates a similar ranking result, while any N value less than 6 enables the prediction of one pass to end in a time limit of 1 second, which is reasonable for interactive usage. Furthermore, we intended to capture the syntactic context of the token sequence and tolerate the reordering of statements. Since 5 is the median length per statement of a C program, we adapted the N value of 5 in our experiments.

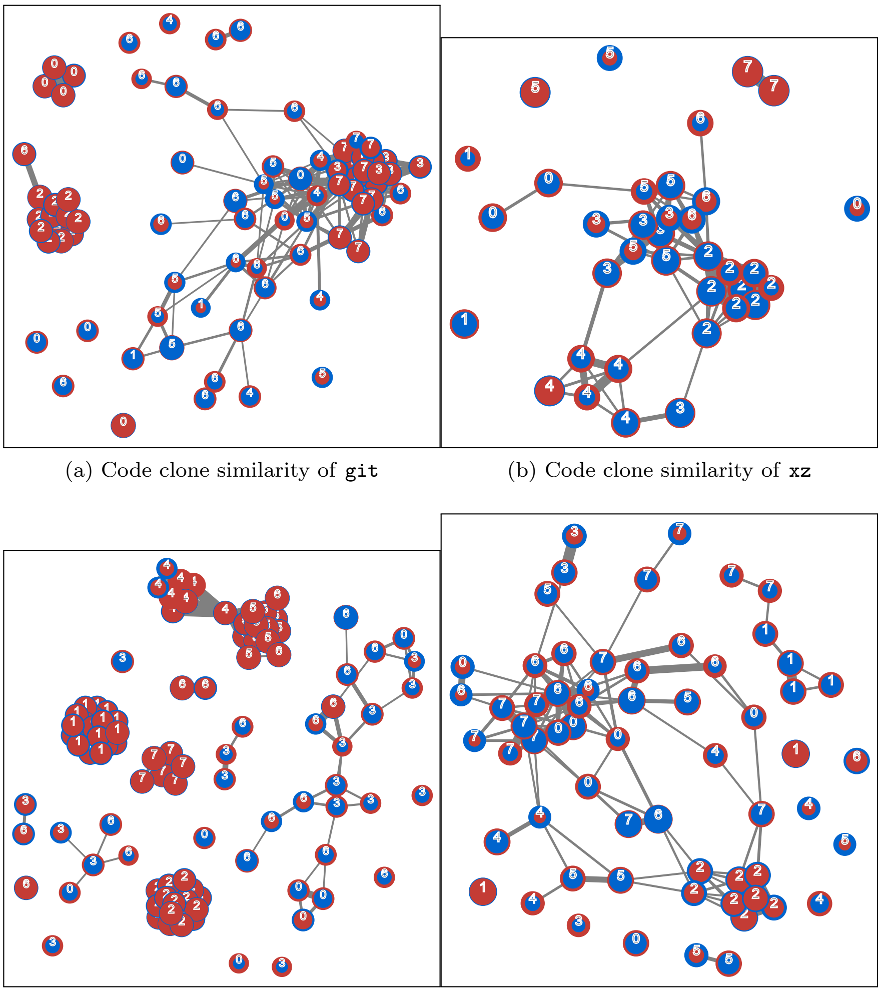
5.3 Code Clone Similarity of Classification Labels by Users

We showed in Figure 7 that the clone sets from a given project have a tendency to group into clusters. To further illustrate this phenomenon, we use a semi-supervised learning approach that compares the result of a clustering algorithm with the selection of the users. Note that this semi-supervised clustering approach is used only to show the underneath relations among our dataset of code clones, but it is not part of the Fica machine learning process.

We applied the KMeans clustering algorithm on the TF-IDF vector of our datasets with $K = 8$ as the target cluster number for each software project. As a result, the KMeans algorithm labeled each clone set with a cluster id from 0 to 7, and all four software projects converged in at most four iterations. The K value of KMeans is determined by trial, and we experimented with values from 5 to 10. In practice, the K value grows with the scale of the targeted software project.

Then we overlaid the labels from the KMeans algorithm onto the FDG, as in Figure 10. For each project in FDG, a clone set is represented as a circle drawn in a mixture of red and blue. Blue circles mean true code clones and red means false code clones. Partially red and blue circles indicate clonesets for which people had different opinions, and the relative area occupied by the two colors reflects the relative number of opinions. The central color always represents the majority opinion. The numerical text in each circle is the label of the clustering result by the KMeans algorithm.

For each pair of clone sets, a link exists between the circles if the similarity between the clone sets is greater than a threshold, which is adjustable from the web interface in real time.The values of similarity were assigned as the force strength on the links between the circles of clone sets. We used the `d3.js` javascript library



(a) Code clone similarity of `git` (b) Code clone similarity of `xz`

(c) Code clone similarity of `bash` (d) Code clone similarity of `e2fsprogs`

Fig. 10: Code Clone Similarity with classification by users

by Bostock (2012) to generate this FDG. Some other technical details about this graph are listed in Table 3.

To show the distance among the clones in each cluster by KMeans, we calculated the average normalized cluster distance, as in Table 4. The distances in this table were normalized from 0 to 1, where the larger number indicates a larger distance among the clone sets and the most distance clone sets have a score of 1.

Figure 10 shows that groups of false positives are more similar than groups of true positives. Thus, these false positives are closer to each other in the figure and are clustered with the same label by KMeans. For example, in the `git` project, the cluster with ID 2 has an average normalized distance of 0.21; this means that the false clone sets appear to be closer than the cluster with ID 6, whose average normalized distance is 0.9. This result indicates that the probability of false clone sets calculated by Equation 7 is more accurate than the probability of true clone sets, because false clones are grouped into categories more tightly.

We calculated the homogeneity and completeness scores defined by Rosenberg and Hirschberg (2007). These two scores compare the labels of the clustering result with the label of ground truth, which comprises the selected marks by the users in our dataset. These two scores reveal the different characteristics of clustering:

homogeneity scores high when each cluster contains only members of a single class.

completeness scores high when all members of a given class are assigned to the same cluster.

Table 3: Parameters for force-directed graph

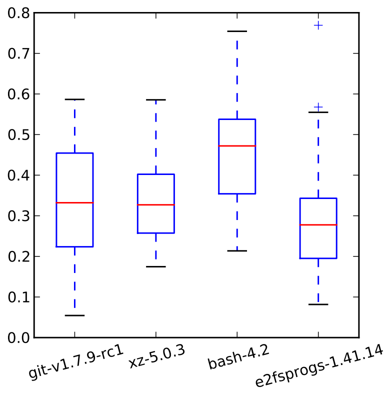
Parameter	Value
force.size	800
force.charge	100
link.linkDistance	$1/link.value$
link.linkStrength	$link.value/16 + 0.2$
link.stroke-width	$10 \cdot \sqrt{link.value}$
node.r	$13 \cdot major/(minor/1.5 + major)$ 8
node.stroke-width	$8 \cdot minor/(minor/1.5 + major)$ 9

$$major = \max\{|\texttt{true}|, |\texttt{false}|\} \tag{8}$$

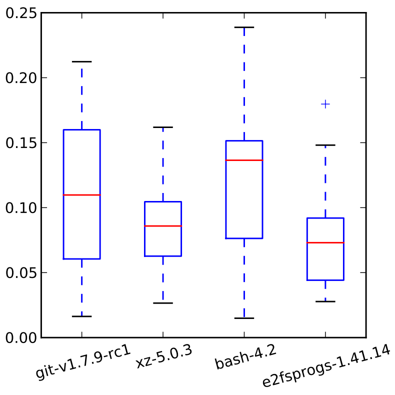
$$minor = \min\{|\texttt{true}|, |\texttt{false}|\} \tag{9}$$

Table 4: Average normalized distance of clusters by KMeans

	0	1	2	3	4	5	6	7
git	0.78	0.46	0.21	0.36	0.73	0.78	0.9	0.55
xz	0.64	0.46	0.37	0.66	0.58	0.77	0.45	0.0
bash	0.79	0.18	0.19	0.87	0.25	0.06	0.9	0.14
e2fsprogs	0.76	0.74	0.42	0.33	0.71	0.76	0.79	0.79



(a) Homogeneity



(b) Completeness

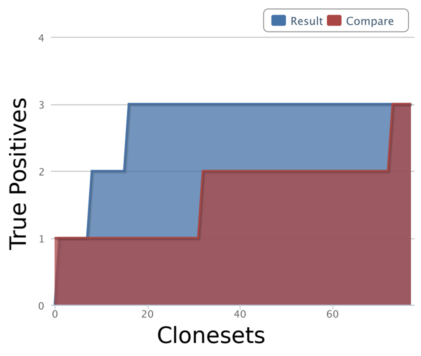
Fig. 11: Homogeneity and completeness between users' marks and clusters by KMeans

As we have eight clusters by the KMeans algorithm and two classes from the marks of users, the homogeneity is expected to be high and the completeness is expected to be low. For each software project, we compared the marks of each user with the KMeans labels and used a boxplot to show these two scores in Figure 11. We can see the co-relationship of users' marks with the clusters from the KMeans algorithm.

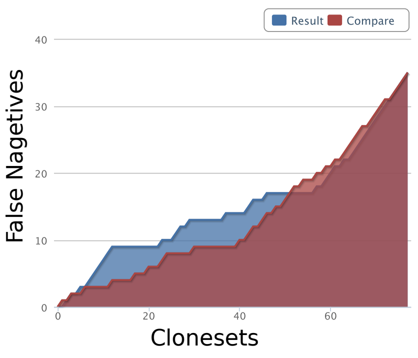
The internal reason for this result is the fact that almost all false clone sets fall into categories of meta-clones with certain obvious characteristics of their token types, such as a group of `switch-case` statements, or a group of assignments. Meanwhile, the true clones are harder to classify into similar categories. In general, true code clones usually contain rare sequences of tokens. Thus, we have observations 1 and 2, regardless of the subjective judgment of participants,

Observation 1 (Categories of False Clones) *False code clones are more likely to fall into several categories of meta-clones by comparing the literal similarity of their token types.*

Observation 2 (Diversity of True Clones) *True code clones composed of sequences of tokens are more diverse than false clones by comparing the literal similarity.*



(a) Average True Positives Found



(b) Average False Negatives Found

Fig. 12: Average True Positives Found and Average False Negatives Found for different users on git project

5.4 Ranking Clone Sets

To measure the quality of supervised machine learning of FICA, we adapted a measure called *Average true Positive Found* (referred to as ATPF) proposed by Lucia et al (2012). Their study is similar to ours in that they also reordered the clone list according to the structure similarity. Here, the word *positive* refers to the clones predicted by FICA that have a high likelihood of being true clones, rather than the result from the CDTs.

The measure of ATPF is visualized as a graph capturing the cumulative proportion of true positives found as more clones are inspected by the users. In the cumulative true positives curve, a larger area under the curve indicates that more true positives were found by the users early; this means the refinement process effectively re-sorts the clone list.

An example of an ATPF graph is shown in Figure 12a. In the ATPF graph, both the X-axis and the Y-axis are the number of clone sets, where the value of the X-axis is the number of clone sets presented to the user while the value of the Y-axis is the number of clone sets that the user considers as true clones. The graph has two cumulative curves. The *"result"* curve in blue is the result after reordering by our tool, while the *"compare"* curve in red is the original order presented by the CDTs. For our embedded CDT, the output order of clones is in alphabetical order of the token sequence because of the suffix tree algorithm.

In Figure 12a, 78 clone sets are found by the CDT, but only 3 of them are regarded as true clones by the user. As we can see from the *"compare"* curve, these 3 true clone sets are distributed in the very beginning, the middle, and near the tail of the clone set list. In the *result* curve, our method successfully rearranged the order of the clone list, so that all three true clone sets appeared at the front part of the clone set list. In a real working environment, this will significantly increase the efficiency of the user.

As the user can also change the sorting order to filter out the false clone sets, we defined *Average False Negative Found* (referred to as AFNF), which is similar to the definition of ATPF shown in Figure 12b. Analogously, the word *negative* refers to the clones that have a high likelihood to be false clones by FICA. As we can see in the AFNF graph, the algorithm generates more false negatives in the beginning, and then after approximately 64% of the clone list is processed, the result fails to be better than the compared result. This behavior of the algorithm is expected. By reordering the list of clones, moving the desired clone to the end of the list is a small probability if the clone is similar to those previously marked as not desired by the user.

5.5 Predictions for Each Project

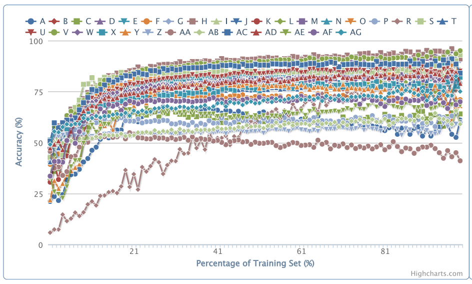
To measure the accuracy of predictions made by FICA with the marked labels from the user, we define a metric called "accuracy" as the percentage of how many predictions by FICA are equal to the selections of the user, that is, the percentage of both true positives and true negatives in all clone sets.

We trained the FICA system by using the marked data of eight users on all clone sets for each project. The accuracy of our prediction model is shown in Figure 13. The horizontal axis of the figure is the percentage of training sets and the vertical axis is the previously defined "accuracy" of prediction. Three steps were required to perform a prediction:

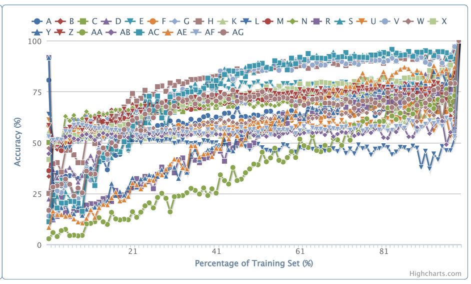
1. The FICA system randomly selected a part of all clone sets from a project as the training set and the remaining were used as the comparison set.
2. For the division of training and comparison sets, FICA trained its machine learning model with the training set and calculated a prediction for each clone set in the comparison set.
3. FICA compared the predicted result with the mark made by a user, and calculated the accuracy.

We repeated these three steps 256 times for each training set size, and the plotted values of accuracy in the Figure 13 are the averaged results of the 256 times prediction.

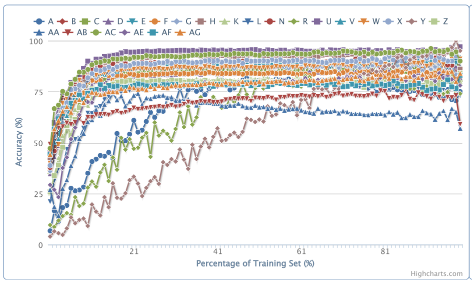
We can see from Figure 13 that the prediction results change with the users and the target projects. For all four projects and nearly all users, we can see



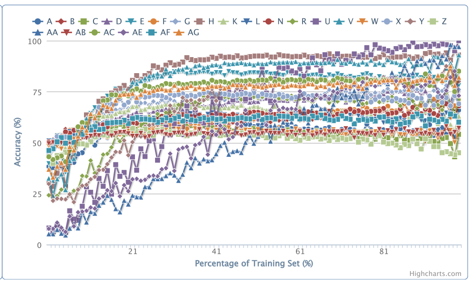
(a) FICA Prediction on `git`



(b) FICA Prediction on `xz`



(c) FICA Prediction on `bash`



(d) FICA Prediction on `e2fsprogs`

Fig. 13: Accuracy of machine learning model by FICA

that the accuracy of the prediction model grows with the portion of the training set. As the training set grows, in the beginning the accuracy of the prediction increases quickly until it reaches a point, between 10% to 30%, where it increases more slowly.

Among all four projects, the `bash` project shows the most desirable result, namely, most of the prediction accuracies are over 80% when the training set is larger than 16%, which is approximately 17 of all 105 clone sets. The results of the `git` project and the `e2fsprogs` project largely depend on the user, for example, the result of user `H` always achieves more than 90%. Meanwhile, the results of user `A` and `C` converge to approximately 60% and even decrease when the training set is growing. The reason why the result did not converge to 100% and the accuracy dropped is discussed in Section 5.8.

By combining the Observation 1 with the details from the result of Figure 13, we got our Observation 3.

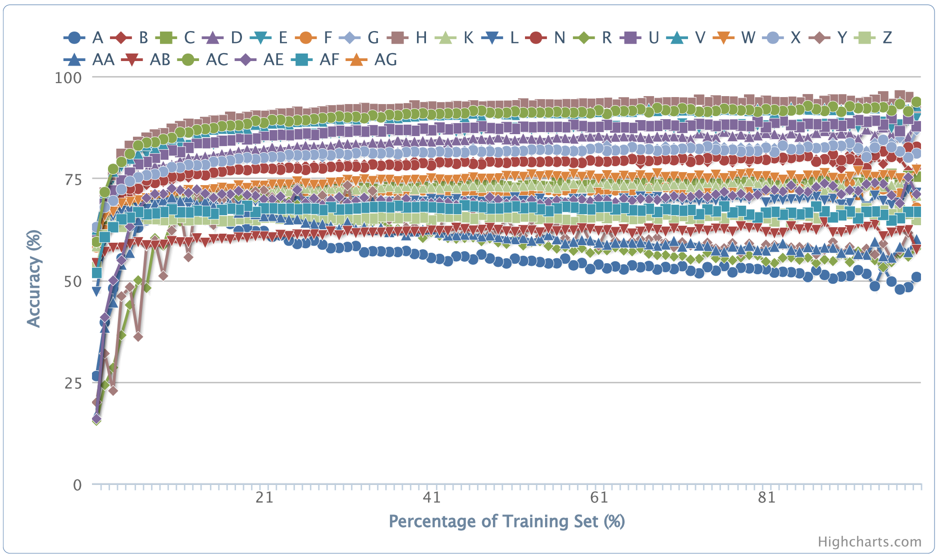


Fig. 14: Merged result of all projects

Observation 3 (Minimum Size of Training Set) *The minimum required size for the training set grows roughly linearly with the number of categories that the clone sets fall into, which is less than the total number of detected clone sets.*

We can also observe from Figure 13 and 14 that the different prediction results of users are separated into different levels. The predictions for some particular users, namely user `R` and user `V`, are always low, which means less literal similarity is found among the clone sets they marked as true clones. Also, the prediction for user `H` is high for all projects. This result shows the consistency of user behaviors.

5.6 Cross Project Evaluation

To illustrate that the training data of selections by the users can be applied across different projects, we merged all clone sets from the four projects into a single project and repeated the above experiment again. The result is shown in Figure 14. The result is actually better than the result for a single project, except for the `bash` project.

To simulate a real cross-project analysis, we also did a cross-project evaluation. For each user and each pair of the four projects, we trained the model with the data gathered from one project and evaluated our method on the other.

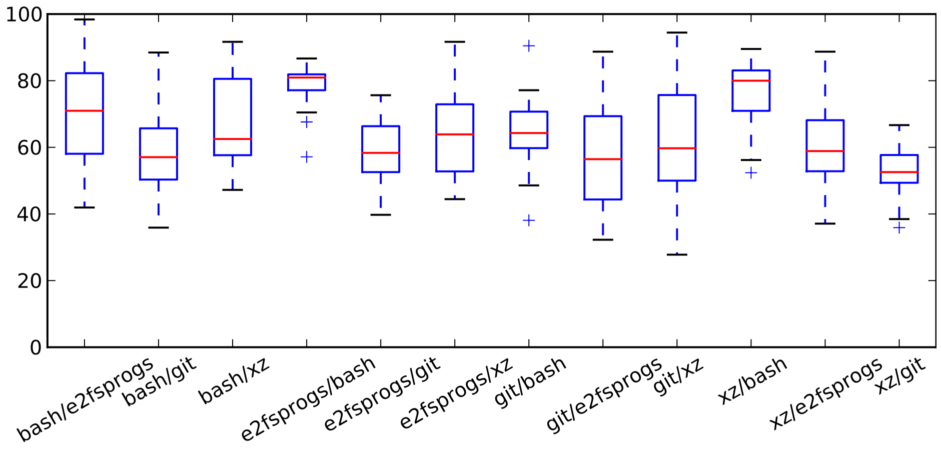


Fig. 15: Accuracy of predictions with training/evaluation projects

We used the same accuracy definition and drew the boxplot of the users in Figure 15. The result varies with the project: `e2fsprogs/bash` is a desirable result and `xz/git` is an undesirable result. We expect this kind of variation in the results, because each project has its own characteristics for code clones. In practice, the user of the FICA system is expected to cross-analyze similar projects when finding similar code clone categories in both projects.

5.7 Recall and Precision of FICA

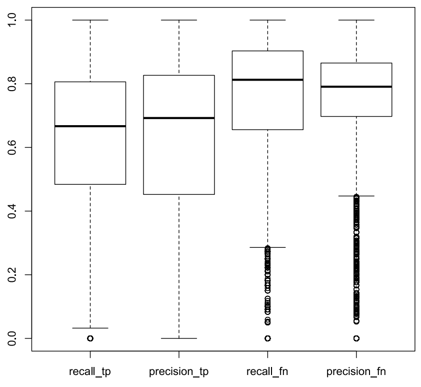
We also measured the recall and precision for separated true positives and false negatives of FICA as support for our definition of accuracy. First, we referred to *true positive* as *tp*, *true negative* as *tn*, *false positive* as *fp*, *false negative* as *fn*. We then defined the recall and precision for *tp* and *fn* as in Equations 10 to 13.

$$recall_{tp} = \frac{tp}{tp + fn} \tag{10}$$

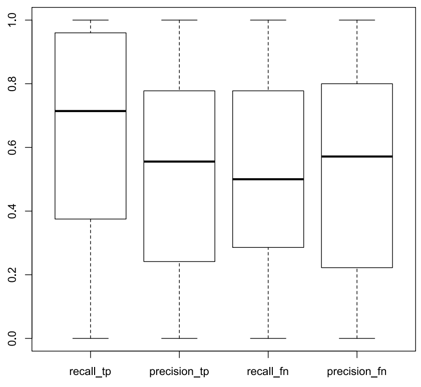
$$precision_{tp} = \frac{tp}{tp + fp} \tag{11}$$

$$recall_{fn} = \frac{tn}{tn + fp} \tag{12}$$

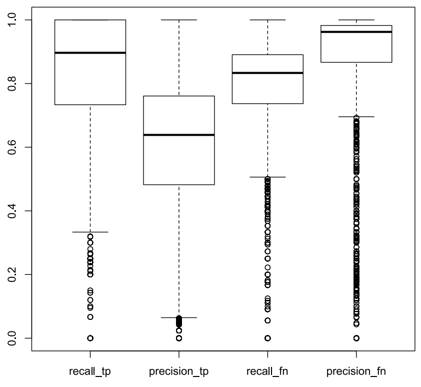
$$precision_{fn} = \frac{fn}{fn + tn} \tag{13}$$



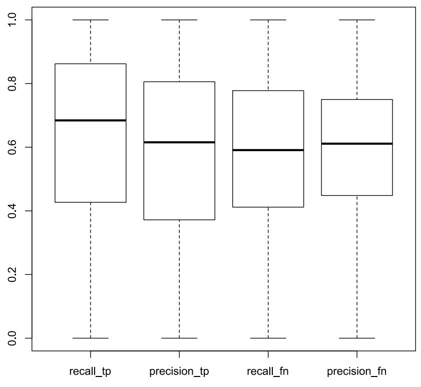
(a) `git`



(b) `xz`



(c) `bash`



(d) `e2fsprogs`

Fig. 16: Recall and precision of FICA in each project

We took 20% of all the clone sets as the training set, the remaining as the evaluation set, and then repeated the experiment 100 times. The result is shown as the boxplot in Figure 16. From the boxplot we can see a similar result to that of Figure 13, as the `git` and `bash` projects show good results while the results for `xz` and `e2fsprogs` are not so comparable. These recall and precision charts show a trend similar to the accuracy graph in Figure 13; that is, when the project has clustered clone categories, the result is more appropriate.

5.8 Reason for Converging Results

For all projects in Figure 13 and 14, the accuracy of predictions made by FICA converges to approximately 70% to 90% and it is difficult to achieve 100%. In this section, we discuss some code fragments to show the reason.

The first example comes from the `xz` project in Figure 17. These three code fragments have two code clone pairs. The code from Figure 17a in lines 130 to 142 and the code from Figure 17b are the code fragments of the first clone pair, referred to as clone α. The code from Figure 17a in lines 136 to 145 and the code from Figure 17c are the code fragments of the second clone pair, referred to as clone β. As we can see from the source code, each instance of clone α consists of a complete function body. Clone β consists of only two half parts of functions. Thus, from the viewpoint of the ease of refactoring, clone α is much easier than is clone β. On the other hand, the calculated similarity between clone α and β is greater than 43% by Equation 5. This is a very high percentage among all the other similarities between clone sets, because these two clones share a large amount of identical code fragments. As the result, 7 out of 8 users thought that clone pair α were true clones and 6 out of 8 users thought that clone pair β were false clones, while FICA always thought they belonged to the same group.

Another example comes from the `e2progs` project shown in Figure 18. It is clear that two code clone pairs are found in Figure 18. The code from Figure 18a and that from Figure 18b form the first code clone pair, referred to as clone γ, and the code from Figure 18c and Figure 18d forms the second, referred to as clone δ. Users can tell these two clone pairs are different because clone γ consists of two function bodies and can be merged into one function, while clone δ consists of two lists of function declarations. However, the calculated similarity between clone γ and δ is greater than 6.4%, which is large enough to affect the overall result. The reason why FICA regarded them as similar is that they share many common N-grams, such as `STATIC ID ID LPAREN CONST` or `CONST ID TIMES ID COMMA`, which results in the high value of similarity.

Based on the above discussion, we learned from the examples that comparing code clones by only their literal similarity has some limitations. We will continue to learn how much this limits the result and whether we can improve the accuracy by combining other methods, such as the hybrid token-metric based approach.

```
    }
    return LZMA_PROG_ERROR;
}
static void
block_encoder_end(lzma_coder *coder, lzma_allocator *allocator)
{
    lzma_next_end(&coder->next, allocator);
    lzma_free(coder, allocator);
    return;
}
static lzma_ret
block_encoder_update(lzma_coder *coder, lzma_allocator *allocator,
        const lzma_filter *filters lzma_attribute((__unused__)),
        const lzma_filter *reversed_filters)
{
    if (coder->sequence != SEQ_CODE)
```

(a) src/liblzma/common/block_encoder.c

```
    }
    return LZMA_OK;
}
static void
alone_encoder_end(lzma_coder *coder, lzma_allocator *allocator)
{
    lzma_next_end(&coder->next, allocator);
    lzma_free(coder, allocator);
    return;
}
static lzma_ret
alone_encoder_init(lzma_next_coder *next, lzma_allocator *allocator ,
        const lzma_options_lzma *options)
```

(b) src/liblzma/common/alone_encoder.c

```
    else
        lzma_free(coder->lz.coder, allocator);
    lzma_free(coder, allocator);
    return;
}
static lzma_ret
lz_encoder_update(lzma_coder *coder, lzma_allocator *allocator ,
        const lzma_filter *filters_null lzma_attribute((__unused__)),
        const lzma_filter *reversed_filters)
{
    if (coder->lz.options_update == NULL)
```

(c) src/liblzma/lz/lz_encoder.c

Fig. 17: Example of source code in xz

```
    return 0;
}
static int parse_block(const char *request,
      const char *desc, const char *str, blk_t *blk)
{
    char *tmp;
    *blk = strtoul(str, &tmp, 0);
    if (*tmp) {
        com_err(request, 0, "Bad %s - %s", desc, str);
        return 1;
    }
    return 0;
}
static int check_brel(char *request)
```

(a) tests/progs/test_rel.c

```
    return 1;
}
static int parse_inode(const char *request,
      const char *desc, const char *str, ext2_ino_t *ino)
{
    char *tmp;
    *ino = strtoul(str, &tmp, 0);
    if (*tmp) {
        com_err(request, 0, "Bad %s - %s", desc, str);
        return 1;
    }
    return 0;
}
void do_create_icount(int argc, char **argv)
```

(b) tests/progs/test_icount.c

```
static errcode_t test_write_blk64(io_channel channel,
    unsigned long long block, int count, const void *data);
static errcode_t test_flush(io_channel channel);
static errcode_t test_write_byte(io_channel channel,
    unsigned long offset, int count, const void *buf);
static errcode_t test_set_option(io_channel channel,
    const char *option, const char *arg);
static errcode_t test_get_stats(io_channel channel, io_stats *stats);
static struct struct_io_manager struct_test_manager = {
```

(c) lib/ext2fs/test_io.c

```
static errcode_t unix_write_blk(io_channel channel,
    unsigned long block, int count, const void *data);
static errcode_t unix_flush(io_channel channel);
static errcode_t unix_write_byte(io_channel channel,
    unsigned long offset, int size, const void *data);
static errcode_t unix_set_option(io_channel channel,
    const char *option, const char *arg);
static errcode_t unix_get_stats(io_channel channel, io_stats *stats)
;
static void reuse_cache(io_channel channel, ...
```

(d) lib/ext2fs/unix_io.c

Fig. 18: Examples of source code in e2fsprogs

5.9 Threats to Validity

The internal and external validity of our methodology faces several threats.

First, we focused on whether a code clone is a true clone, which depends on the subjective judgment of the particular user. Therefore, the subjective nature of the experiment leads to a major threat to the internal validity of our work. We sent out invitations to participate in the experiment through mailing lists and twitter; therefore, most of the authors participated in the experiment and most of them shared an academic background. We tried to reach software developers in industry, but failed to collect enough amount of data to be useful in this study.

Second, while conducting the experiment, we could not enforce a general rule for the participants, so we depended on the participants to maintain consistency during the experiment, which is not a trivial task.

Third, the accuracy and precision of our work were evaluated against the set of true clones, which can be affected by the subjective emotions of the participants. We also recorded the time spent by each participant during the selections, and found that the participant with the worst prediction accuracy spent the longest time. However, we could not find a similar correlation among other participants.

Moreover, the implementation of Fica and the way we conducted the experiment may also be a threat to internal validity. Although Fica did not enforce the exact order in which the participants viewed the code clones, it is quite possible that these participants were affected by the order in which Fica listed the clones. They could gain experience during the former part of the experiment and apply the experience during the latter part.

Our experiment with participants assumed that these participants were independent. We sent out invitations to participate in the online experiment through email and twitter. It is quite possible that many of these anonymous participants were from the same clone-related research group in our university. Therefore, they may share some common knowledge that was not common for all participants. We tried to reach more participants outside our research group, for example, experts with an industrial background. However, it was very hard to collect data from programmers who had no interest in code clone research.

With regard to external validity, our current implementation and experiments only covered a small part of our overall methodology. We combined Fica with a token-based CDT without proving that it can work with other CDTs, although the method should be CDT neutral. Also, Fica only experimented with Boolean marks of the code clones. Whether other kinds of marks such as range marks or tag marks will affect the result is uncertain.

6 Related Research

Some related works have reported on combining machine learning or text mining techniques with code detection to classify or clustering code clones. As a complement to existing code clone detection methods, Marcus and Maletic (2001) proposed a method to identify high-level concept clones, such as different implementations of the algorithm of linked lists, directly from identifiers and comments from source code as a new method of clone detection. A similar approach by Kuhn et al (2007) found semantic topics instead of clones from comments and identities from source code.

Another method proposed by Tairas and Gray (2009) shares some common ideas with the above two methods. They compare code clones by using information retrieved from identifiers; such an approach focuses more on the semantic information or behavior of source code rather than the syntactic or structural information of source code. In contrast, the method described in this paper compares the tokenized source code of code clones, which focuses on the syntactic similarity between code clones. The works by Lucia et al (2012) share many general ideas with our work, although their focus is finding clones with bugs. They compare the structure of code clones and thus require an AST-based CDT to extract structure information. Furthermore, although not mentioned in their paper, we contacted the first author and confirmed that the judgment of whether a clone is buggy depends on the opinion of the authors, and thus relies on more subjective judgments.

Besides comparing text similarity, other methods have been proposed to filter unneeded code clones from the detection result. Higo et al (2008a) proposed a metric-based approach to identify code clones with higher refactoring opportunities. Their method calculates six different metrics for each code clone and then represents the plotted graph of these metrics as a user-friendly interface to allow users to filter out the unneeded code clones. This user-defined metric-based filtering method was further automated by Koschke (2012). Koschke is targeting a different objective, which is identification of license violations, but has also used metrics and machine learning algorithms on those metrics to form a decision tree. The result of the decision tree limits the types of metrics to only two, which are PS (Parameter Similarity) and NR (Not Repeat). These metric-based methods all have a limited identification target and thus result in higher accuracy than the method proposed in this paper. More recent research Wang et al (2013) uses a search-based solution to repeatedly change the configurations of several well-known code clone detectors in order to improve the result. They experimented on the true clone dataset of Bellon et al (2007). They tried to improve the general agreement as well as individual agreement results for each software project. We have a different target, which is filtering out false code clones for an individual user's purpose. Therefore, we took a different approach.

Other works on the classification or taxonomy of code clones focus on proposing fixed schemes of common clone categories. Balazinska et al (1999) proposed a classification scheme for clone methods with 18 different categories. The categories detail what kind of syntax elements have been changed and also how much of the method has been duplicated. Bellon et al (2007) defined three different clone types (exact clones, parameterized clones, and clones that have had more extensive edits) for the sake of comparison between different detection tools. Their aim was to test the detection and categorization capabilities of different tools.

Several visualization methods have also been proposed to aid the understanding of code clones. A popular approach that has been implemented in most CDTs is the scatter plot Church and Helfman (1993); Higo (2006). A scatter plot is useful for selecting and viewing clones on a project scale, but it is difficult to illustrate the relations among clones. Johnson proposed a method in Johnson (1994) that uses Hasse Diagrams to illustrate clusters of files that contain code clones. Johnson also proposed navigating files containing clone classes by using hyper-linked web pages citeJohnson1996Navigating. The force-directed graph used in the Fica system described in this paper is highly interactive and has all the benefits of a modern web system. Thus, Fica should be useful in the analysis of code clones.

Jiang and Hassan (2007) proposed a framework for understanding clone information in large software systems by using a data mining technique framework that mines clone information from the clone candidates produced by CCFinder. First, a lightweight text similarity is applied to filter out false positive clones. Second, various levels of system abstraction are used to further scale down the filtered clone candidates. Finally, an interactive visualization is provided to present, explore and query the clone candidates, as with the directory structure of a software system. Their method shares some common features with Fica. Compared to their method, Fica depends more on the marks of code clones by users, and thus is more customized for the individual user but also requires more operations by users.

Wang et al (2012) proposed a machine learning based method for predicting the harmfulness of code clones. Their predictor employs a Bayesian Network algorithm and learns three categories of metrics from code clone fragments, namely, the history, code and destination metrics. Their work defines the harmfulness of a clone set based on inconsistent changes among the clones, which is an objective standard. As in the idea of our APPF and APNF, they also define conservative and aggressive scenarios and evaluate their effectiveness accordingly. Although they noted in their paper that each category of metric contributes to the overall effectiveness, they do not illustrate the contribution of the literal similarity of source code. Compared to their work, our Fica system shows that the literal similarity alone is enough to classify the clones for varied purposes.

7 Conclusions

We have shown that users of CDTs have different opinions on whether a code clone is indeed a true clone, which means "useful" or "interesting" according to their particular purpose. This observation suggested that filtering code clones should take user judgments into consideration to generate more useful list of code clones.

With this observation, we proposed a classification model based on applying machine learning on code clones. We built the described system Fica, which is a web-based system, as a proof of concept. The system consists of a generalized suffix-tree-based CDT and a web-based user interface that allows the user to mark detected code clones and shows the ranked result.

We conducted an experiment on the Fica system with 32 participants. Our classification model showed more than 70% accuracy on average and more than 90% accuracy for particular users and source code projects.

We analyzed the experiment in detail, and obtained several observations from the experiments about true code clones:

1. Users agree that false positive code clones are likely to fall into several categories, such as a list of assignment statements or a list of function declarations.
2. Users agree that true positive code clones are more diverse than false positives.
3. The minimum required size for the training set generally grows linearly with the number of categories that the clone sets fall into, which is less than the total number of detected clone sets.

Acknowledgements This study was supported by Grant-in-Aid for Scientific Research (S) (25220003), Grant-in-Aid for Exploratory Research (24650011) from the Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (A) (24680002) from the Ministry of Education, Culture, Sports, Science and Technology.

References

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999) Measuring clone based reengineering opportunities. In: Sixth International Software Metrics Symposium, IEEE, pp 292–303

Baxter I, Yahin A, L Moura MA, Bier L (1998) Clone detection using abstract syntax trees. In: Proc. of the 14th International Conference on Software Maintenance, pp 368–377

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering 33(9):577–591

Bostock M (2012) D3.js, Data-Driven Documents. `http://d3js.org/`, [Online; accessed 1-May-2012]

Church K, Helfman J (1993) Dotplot: A program for exploring self-similarity in millions of lines of text and code. Journal of Computational and Graphical Statistics pp 153–174

Higo Y (2006) Code clone analysis methods for efficient software maintenance. PhD thesis, Osaka University

Higo Y, Ueda Y, Kamiya T, Kusumoto S, Inoue K (2002) On software maintenance process improvement based on code clone analysis. In: Oivo M, Komi-Sirvi S (eds) Product Focused Software Process Improvement, Lecture Notes in Computer Science, vol 2559, Springer Berlin Heidelberg, pp 185–197, DOI 10.1007/3-540-36209-6_17, URL `http://dx.doi.org/10.1007/3-540-36209-6_17`

Higo Y, Kusumoto S, Inoue K (2008a) A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. Journal of Software Maintenance and Evolution: Research and Practice 20(6):435–461

Higo Y, Kusumoto S, Inoue K (2008b) A survey of code clone detection and its related techniques. IEICE Transactions on Information and Systems 91-D(6):1465–1481, (in Japanese)

Jiang Z, Hassan A (2007) A framework for studying clones in large software systems. In: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, pp 203–212

Johnson J (1994) Visualizing textual redundancy in legacy source. In: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, p 32

Jones K (1972) A statistical interpretation of term specificity and its application in retrieval. Journal of documentation 28(1):11–21

Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28(7):654–670

Kobourov S (2012) Spring embedders and force directed graph drawing algorithms. arXiv preprint arXiv:12013011

Koschke R (2012) Large-scale inter-system clone detection using suffix trees. In: 2012 16th European Conference on Software Maintenance and Reengineering, IEEE, pp 309–318

Krinke J (2008) Is cloned code more stable than non-cloned code? In: Proc. of the 8th International Working Conference on Source Code Analysis and Manipulation, pp 57–66

Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: Identifying topics in source code. Information and Software Technology 49(3):230–243

Li Z, Myagmar S, Lu S, YZhou (2006) Cp-miner : Finding copy-paste and related bugs in large-scale software code. IEEE Transcations on Software Engineering 32(3):176–192

Lucia, Lo D, Jiang L, Budi A, et al (2012) Active refinement of clone anomaly reports. In: 34th International Conference on Software Engineering, IEEE, pp 397–407

Marcus A, Maletic J (2001) Identification of high-level concept clones in source code. In: 16th Annual International Conference on Automated Software Engineering, IEEE, pp 107–114

Rosenberg A, Hirschberg J (2007) V-measure: A conditional entropy-based external cluster evaluation measure. In: EMNLP-CoNLL, vol 7, pp 410–420

Roy C, Cordy J, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 74(7):470–495

Rysselberghe FV, Demeyer S (2004) Evaluating clone detection techniques from a refactoring perspective. In: Proceedings of the 19th IEEE international conference on Automated software engineering, IEEE Computer Society, pp 336–339

Tairas R, Gray J (2009) An information retrieval process to aid in the analysis of code clones. Empirical Software Engineering 14:33–56, 10.1007/s10664-008-9089-1

Tiarks R, Koschke R, Falke R (2009) An assessment of type-3 clones as detected by state-of-the-art tools. In: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on, IEEE, pp 67–76

Tiarks R, Koschke R, Falke R (2011) An extended assessment of type-3 clones as detected by state-of-the-art tools. Software Quality Journal 19(2):295–331

Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260

Wang T, Harman M, Jia Y, Krinke J (2013) Searching for better configurations: A rigorous approach to clone evaluation. Compare 31(58.5):81–1

Wang X, Dang Y, Zhang L, Zhang D, Lan E, Mei H (2012) Can i clone this piece of code here? In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp 170–179