Master Thesis

Title

Classification Model for Code Clones Based on Machine Learning

Supervisor

Prof. Shinji KUSUMOTO

by

Jiachen YANG

February 5, 2013

Department of Computer Science

Graduate School of Information Science and Technology

Osaka University

Master Thesis

Classification Model for Code Clones Based on Machine Learning

Jiachen YANG

Abstract

Code clones have gained great attentions in recent research. Several code clone detection methods have been proposed to detect identical or similar code fragments from source code of software. These code clones are introduced into software systems by various operations during development, namely copy-and-paste or machine generated source code. Despite of its commonly occurrence in software development, code clones are generally considered harmful as they make software maintenance more difficult and indicate poor quality of source code. If we modified a code fragment, it will be necessary to check all corresponded code clones whether they need modifications simultaneously.

By applying code clone detectors to the source codes, users such as programmers can obtain a list of all code clones of a given code fragments, which 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 different purposes of them. So it is difficult to just adjust the parameters of code clone detectors and expect to get the desired code clones. It is also a painful task to analyze through the entire list that the code clone detector generated.

In this research we proposed a classification model by applying machine learning algorithm on the judgments of each individual user on code clones. And we experimented the proposed model by an on-line survey to test its usability and accuracy with 33 participants contributed.

The result showed several important observations on the characteristics about the interesting-ness of code clones for the users. And our classification model showed more than 70% accuracy in average and more than 90% accuracy for particular user and source code project. And during this research, several important observations were obtained about the interesting-ness of code clones.

Keywords

filtering

classification

machine learning

code clone detector

judgment of user

token-based

Contents

List of Figures

List of Tables

1 Introduction

Great efforts have been made to detect identical or similar code fragments from source code of software, with these code fragments called as "code clones" or simply "clones" [1–3]. These code clones are introduced into software systems by various operations during development, namely copy-and-paste or machine generated source code. Because code cloning is easy and inexpensive, it can make software development faster and can enable "experimental"development. However, despite of its commonly occurrence in software development, code clones are generally considered harmful as they make software maintenance more difficult and indicate poor quality of source code. If we modified a code fragment, it will be necessary to check all corresponded code clones whether they need modifications simultaneously. Therefore, various techniques and tools have been proposed to detect code clones automatically by many researchers [4–19].

By applying code clone detectors to the source codes, users such as programmers can obtain a list of all code clones of a given code fragments, which 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 different purposes of them. So it is difficult to just adjust the parameters of code clone detectors and expect to get the desired code clones. It is also a painful task to analyze through the entire list that the code clone detector generated.

Clone Detection Tools(referred as CDT) usually generate a long list of clones from source code, among which a small portion of clones are helpful to users in improving software quality by applying refactoring or locating similar bugs, however the rest are not so "interesting".

Several methods have been proposed to filter out those "uninteresting" clones such as metric-based method described in [20]. A general standard of judging whether a clone is "interesting" or not, can be summarized by these methods, while not only professional knowledges from users, such as what all these metrics mean, is needed, but also it is hard to fit to individual use case of users such as filtering only the code clones that can be applied extracting-method refactoring on. Tairas and Gray proposed a classification method in [21]. These existing classification methods are clustering by identifier names rather than textural similarity of the identical clone fragments.

According to the survey we conducted, users of CDTs tend to classify clones differently based on their individual use cases, purposes or experience about code clones. With this observation,

we thus propose the idea of studying judgments of each user on clones, which results a new clone classification method, entitled as FICA, Filter for Individual user on code Clone Analysis.

The code clones are classified by FICA, according to the comparison of their token type sequences through the calculation of their textual similarity, along with opinions on these code clones from users as well, who are typically programmers. In a production environment, adapting the method described in this research will decrease the time that a user may spend on analyzing the code clones.

From the experiment result, we obtained several observations:

- Un-interesting code clones are likely to fall into several categories.
- Interesting code clones are unique comparing to un-interesting ones.
- Users with more experience on code clones are more likely to agree with each other compared to users with less experience.
- The minimum required size of the training set roughly grows linearly with the number of categories that clone sets fall into, which is less than a magnitude of the total number of detected clone sets.

In the following sections, we will firstly introduce a motivating example that leads to this research, and the imaginary process of how this method is supposed to help the user to classify the code clones. Then a discussion will be led on the proposed method and algorithms that FICA used in detail. After that, we will show the result of applying FICA to an on-line survey that were conducted by us to evaluate our method. Finally we will discuss about the related works in classification of code clones.

```
2717 wcstr = 0;
2718 slen = mbstowcs (wcstr, s, 0);
2719 if (slen == -1)
2720   slen = 0;
2721 wcstr = (wchar_t *)xmalloc (sizeof (wchar_t) ...
2722 mbstowcs (wcstr, s, slen + 1);
2723 wclen = wcswidth (wcstr, slen);
2724 free (wcstr);
2725 return ((int)wclen);
```

Figure 1: execute_cmd.c in bash-4.2

```
1100 if (wcharlist == 0)
1101 {
1102   size_t len;
1103   len = mbstowcs (wcharlist, charlist, 0);
1104   if (len == -1)
1105     len = 0;
1106   wcharlist = (wchar_t *)xmalloc (sizeof (wchar_t) ...
1107   mbstowcs (wcharlist, charlist, len + 1);
1108 }
```

Figure 2: subst.c in bash-4.2

2 Motivating Example

We conducted a survey on several students[1], providing them 105 clone sets from result of CDT on source code in C language detected from `bash-4.2`, asking them whether they are willing to perform refactoring. Table 1 shows a part of the result. In this table, a code clone set is marked as `O` if a student is willing to apply refactoring on it or as `X` if not so. We can see from this table that their attitudes toward these clones vary from person to person.

As an example, source of clone with ID 5 is showed in Figure 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 thought they could be merge together. Meanwhile Y and M considered the fact that Figure 2 is a code fragment in a larger function which is more than 100 LOC therefore may be difficult to

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

be refactored.

Besides, from Table 1 we can see that Y was more strict than other three students. In the comment to this survey he mentioned that only clones that contains an entire body of C function are candidates for refactoring. This unique standard was also reflected in all 5 “interesting” clones he has chosen.

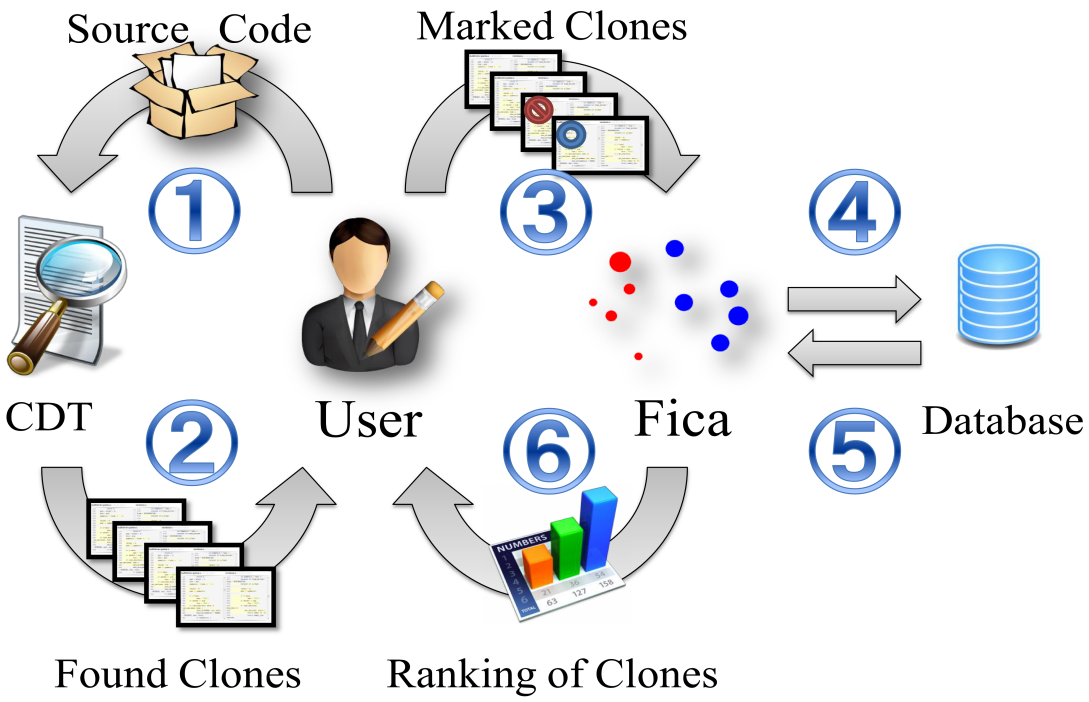


Figure 3: Overall Workflow of FICA with CDT

3 FICA System

In this section, we introduce the general working process of FICA system, which ranks detected clones based on studying historical behavior of a particular user, as a complement to existing CDTs that filtering unexpected clones. FICA is designed as a on-line supervised machine learning system, which means that the user should firstly 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 should be presented to the user more quickly.

3.1 Overall Workflow

The overall workflow of FICA is described in Figure 3.

1. User submits source code to a CDT.

2. CDT detects a set of clones in the source code.

3. User marks some of these clones as "interesting" or not according to her/his own judgment, and then submits these marked clones to FICA as a profile.

4. FICA records marked clones into its own database.

5. Meanwhile FICA studies characteristics of all these marked clones by using machine learning algorithms.

6. FICA ranks other clones remaining unmarked based on the result of machine learning, predicting the possibility of whether they are "interesting" or not to user.

7. User can further adjust the marks on code clones and re-submit them to FICA to obtain a better prediction. FICA will also record these patterns that it have learned into a database associated with the particular user profile so that further predictions can be made based on history decisions.

3.2 Requirements on Existing CDT

FICA utilizes a CDT to generate a list of code clones from source code. Those resulted output from CDT should at least contain following necessary information for each code clone:

1. Positions of the code clone fragment in original source code.

2. A sequence of tokens of the code clone fragment.

Every existing token-based clone detector should already meet this requirement or only require some minor modifications on their output format. Syntax-based clone detectors that compare AST or PDG graphs of the source code could be adjusted to generate a sequential text representation of identical code fragment as the needed input of FICA, which is not necessarily equal to lexical token sequence generated by lexical analyzer (also known as lexer). Text-based clone detectors usually only output positional information of code clones, thus code clone fragments need to be parsed through a lexical analyzer to obtain required lexical token sequence.

3.3 Marking Clones Manually

To be used by FICA as an initial training set, only a small set of clones which is found by CDT, is required to be marked manually by the user of Fica at first. Considered types of marks on clones can be boolean, numerical or tags:

- Boolean clones are marked as "interesting" or not.
- Tagged clones are marked as one of several tags or categories by users based on their use cases such as refactoring procedural, issue tracking id, etc.

As the most simple case, users need to tell FICA that they are interested in certain clones or not. Tags typed marks can be considered as possible extension of boolean typed ones that involve multiple choices.

3.4 Machine Learning

Receiving the clones from CDT and the marks from the user, FICA studies the characteristic of the marked clones by calculating similarity of lexical token sequence of these clones. This step employs machine learning algorithms which are widely used in natural language processing or text mining. The algorithm to be used will be similar with the one GMail used in detecting spam emails or the one CJK Input Methods used in suggesting available input candidates . By comparing the similarity of marked clones and unmarked ones, FICA can thus predict the possibility whether an unmarked clone is interesting or not. Detailed machine learning model and algorithm will be described in Section 4.

3.5 Cycle of Supervised Learning

FICA returns the predicted marks of all remaining clones by ranking or calculating the possibility whether the user may have "interest" in them or not. The user is allowed to correct some of these predictions and re-submit them to FICA to obtain a better prediction, which forms a cycle of supervised learning. And eventually FICA will be trained to filter all clones according to the interest of the user. Furthermore, these patterns that FICA have learned will be recorded into a database as well, which is associated with the particular user. As a result, further predictions on clones can be made based on history decisions of the same user.

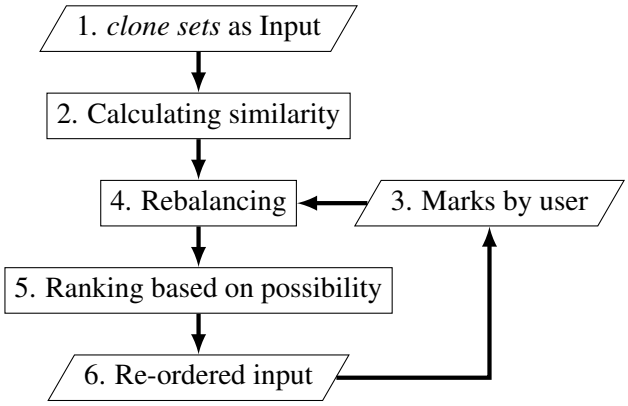


Figure 4: Classification Process in FICA

4 Machine Learning Model

The classification process in FICA is described in Figure 4, which 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 textual similarity among those clone sets by their cosine similarity in tfidf vector space.
3. Given the interesting or un-interesting marks from the user as training sets of machine learning.
4. Re-balancing the input set in order to get identical data points in each set of the training sets.
5. Calculating the possibility for each clone set of their different classification to the marked groups.
6. Finally the clone sets will be ranked and re-ordered as these possibilities, and presented to the user.

User can adjust the marks that FICA has predicted and submit the marks again to FICA as in step 3, which then forms a supervised machine learning circle.

4.1 Input Format of FICA

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

A `project` consists of the original source code and detected code clones which are represented as `clone sets`. The source code are tokenized by CDT. As FICA needs both tokenized source code for calculating the similarity and original source code for representing to the user, these two forms are pass together to FICA. A clone set is a set of identical code fragments. By saying identical, we mean the tokenized code fragments are literally equal to each other in a `clone set`. A `clone` in a `clone set` is the code clone fragment which has a file indicating the source file in the project, and the begin and end token in the file. The list of token types in the clone set should be equal 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 really looks like is in Figure 6.

4.2 Calculating Similarity between clone sets

Firstly FICA calculates the "term frequency – inverse document frequency" (TF-IDF) weight [22] of clone sets. In FICA we define a `term` t as a N-Gram of token sequence, `document` d as a `clone set`, and all documents `D` as a set of `clone sets` which can be the clone sets from a single `project` as well as several `projects`.

Firstly, we divide all token sequences in clone sets as N-Gram terms. Given a clone set with token types as following:

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

If we assume that N for N-gram equals 3, then we can divide this clone set 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
...

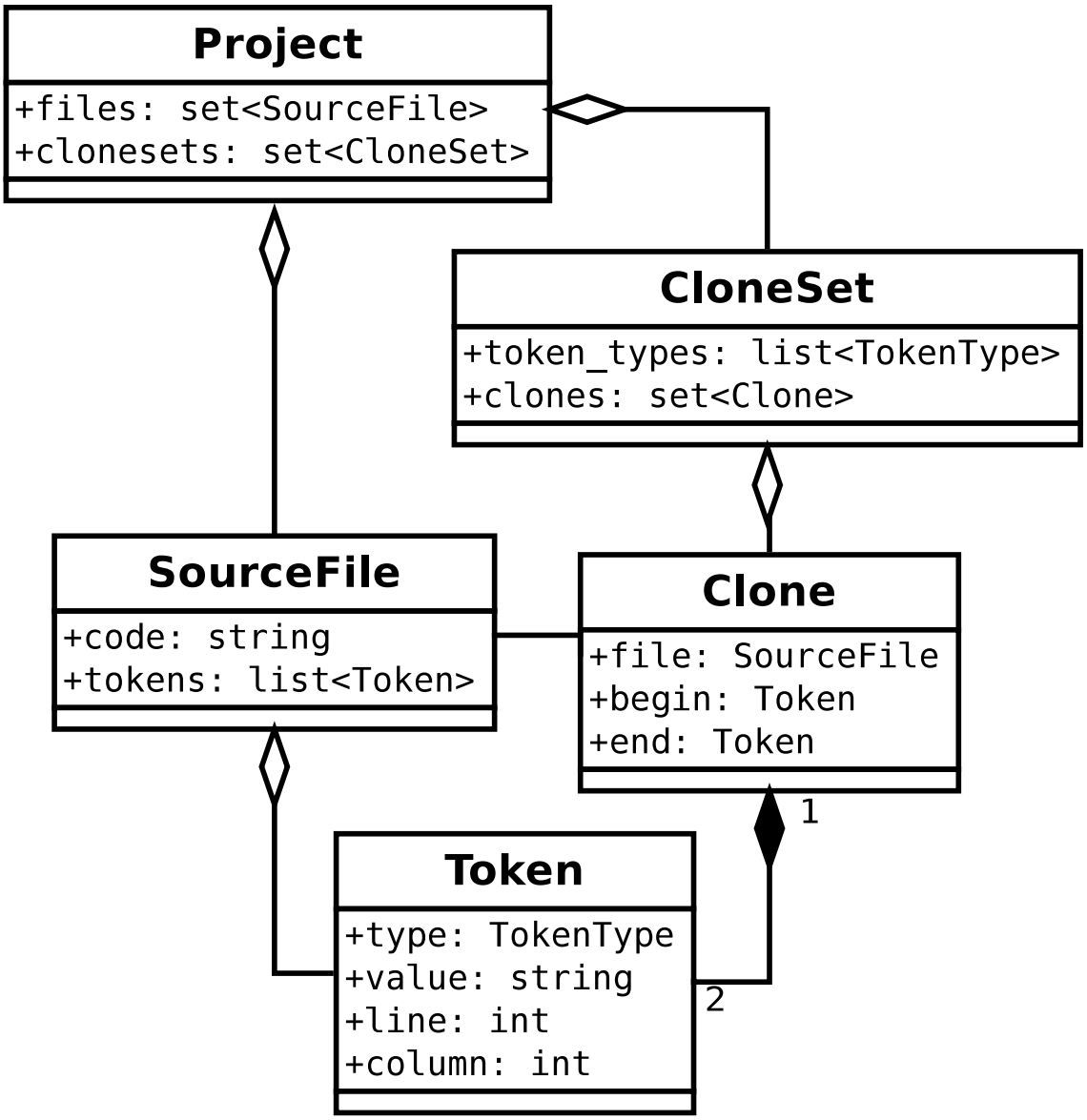


Figure 5: Structure of input to FICA

```
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)
  ...
```

Figure 6: Example of Input Format for FICA

Then we calculate all these `term frequency` of N-Grams with in this document of clone set using Equation 1.

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

Equation 1 says 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 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
...
```

Then the `inverse document frequency` idf and tfidf is calculated using Equation 2 and 3.

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

$$\text{tfidf}_D(t, d) = \text{tf}(t, d) \times \text{idf}_D(t) \tag{3}$$

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

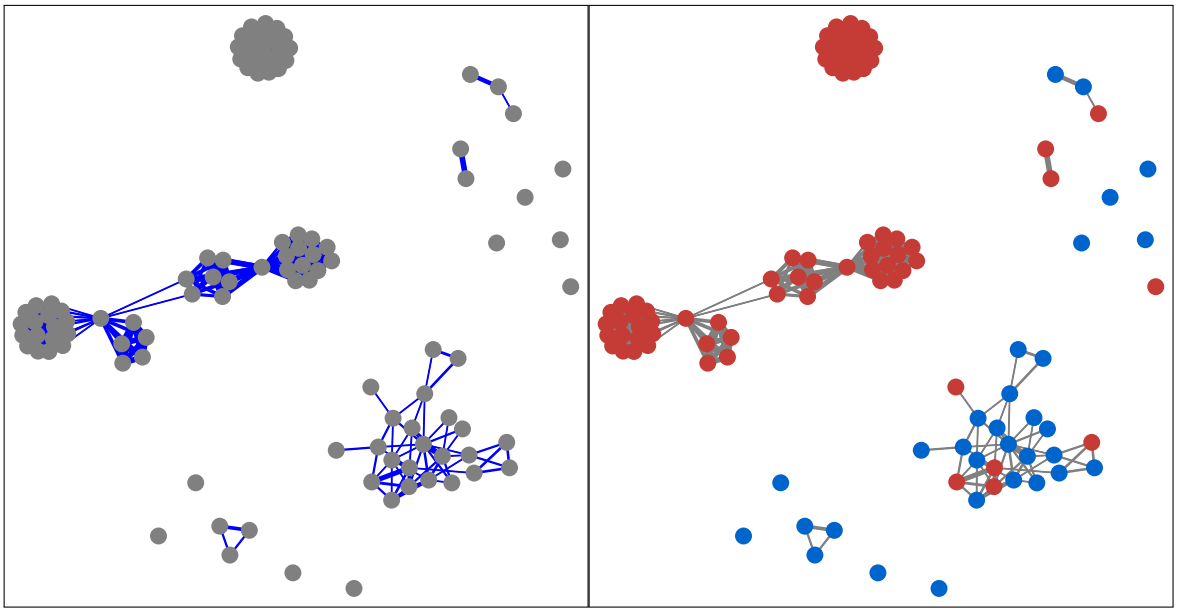
Equation 2 says that `inverse document frequency` *idf* of `term` t in all `documents` D is logarithm of dividing the total number of `documents` by the number of `documents` containing `term` t. In the Equation 2, we plus one to the denominator so as to avoid zero-division. By combining tf and idf as Equation 3 we can then calculate a vector space $\overrightarrow{tfidf(d, D)}$ as in Equation 4 for each clone set in the overall documents.

By using $tfidf$, we define the normalized similarity $\text{NS}_D(a, b)$ of two clone sets, a and b, with regarding a set of documents D as in Equation 5 and 6.

$$\text{S}_D(a, b) = \overrightarrow{\text{tfidf}_D a} \cdot \overrightarrow{\text{tfidf}_D b} \tag{5}$$

$$\text{NS}_D(a, b) = \begin{cases} 0, & \text{S}_D(a, b) = 0 \\ \frac{\text{S}_D(a,b)}{\left|\overrightarrow{\text{tfidf}_D a}\right| \cdot \left|\overrightarrow{\text{tfidf}_D b}\right|}, & \text{otherwise} \end{cases} \tag{6}$$

After calculating the similarity among all clone sets within a project, we can then plot them based on force-directed algorithm as in Figure 7a to illustrate the clone sets grouped by their similarity. The details of force-directed graph will be discussed in next section.



(a) clone sets Grouped by Similarity (b) Colored Force-Directed Graph

Figure 7: Force-Directed Graph of clone sets of bash 4.2

4.3 User Profile and Marks on clone sets

A profile represents a history of user's judgments on classification of clones stored in FICA. A user is allowed to keep multiple profiles in different use cases of FICA. The implementation of FICA does not differ a user from profiles, so for simplicity, the following of this paper assumes that a single user has a profile. FICA needs a initial training set to provide further predictions.

A mark is made on a clone set by the user, which will recorded in the user profile, that says this clone set has some property. There are three kinds of marks that user can make on these clone sets. Boolean marks are two group of clone set that user thought as interesting or not interesting, which are exclusive to each other. Tagged marks are made individually on each clone set, which are not necessary exclusive.

A training set is a set of clone sets that have been marked by user. We only talk about boolean marks here, but the same algorithm is also applicable on tagged marks.

For the boolean marks, the training set M is divided into two sets of clone sets, i.e. M_i and M_u, each of them donates the interesting set and un-interesting set, where $M_i \cap M_u =, M_i \cup M_u = M$.

For each clone set t that not been marked, FICA calculate the possibility whether this clone set

t also marked as those in clone set group M_x by using the Equation 7.

$$\mathrm{P}_{M_x}(t) = \begin{cases} 1, & |M_x| = 0 \\ \frac{\sum_{\forall m \in M_x} \mathrm{NS}_{M_x}(t,m)}{|M_x|}, & \text{otherwise} \end{cases} \tag{7}$$

Equation 7 says the possibility whether clone set t should also has mark M_x are calculated by the average normalized similarity of t and every clone set in clone set group M_x.

For predicting boolean marks, FICA can compare the calculated possibility of both interesting mark and un-interesting mark, choose the higher one as the prediction. For tagged marks, FICA needs to set a threshold for making judgment.

After prediction, all clone sets are marked either by prediction of FICA or by user's input. An example of all boolean marked clone sets in above force-directed graph is shown in Figure 7b. The result of FICA's prediction is presented to user, so that user can check its correctness, approve some of FICA's prediction and correct other predictions. After correcting part of the result, user can re-submit the marks to FICA to get a better prediction.

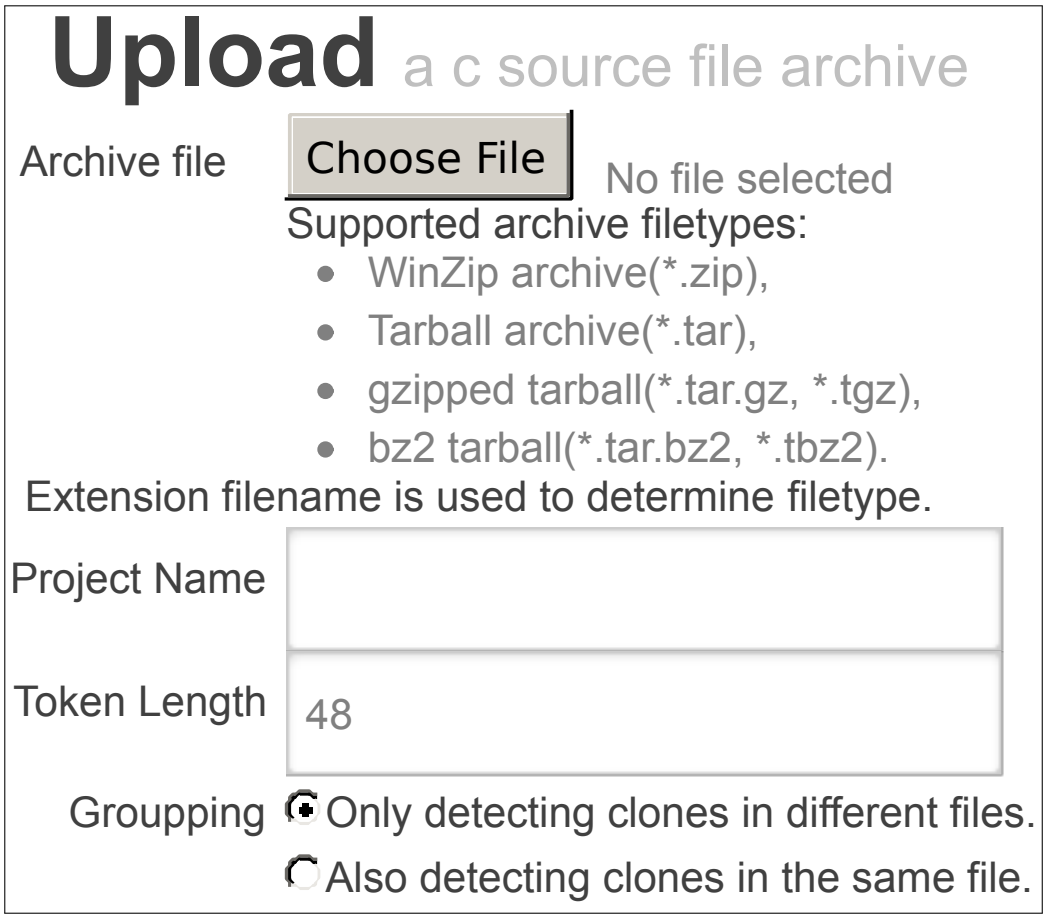


Figure 8: Upload Achieve of Source Code to FICA

5 Implementation Details

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

The FICA system manages profiles of users as normal login sessions like in mostly all websites. One of the users uploads an archive of source code to the FICA system. Then FICA will unzip the source code into a database and then pass to the CDT part of FICA system as in Figure 8.

The CDT part of the FICA system implements the algorithm of generalized suffix tree [23] and algorithm of detection of clones among multiple files in a generalized suffix tree. Code clones and

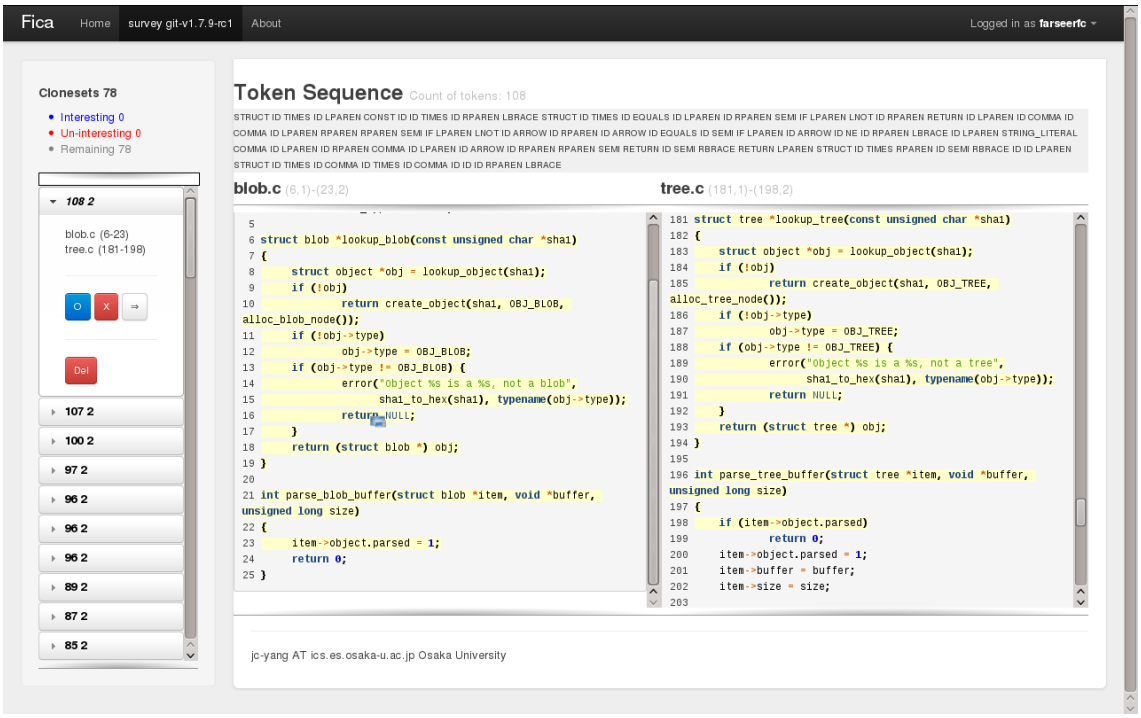


Figure 9: FICA Showing clone sets

types of token sequences are recorded in a database.

And then the FICA system shows a comparison view of detected clone sets to the user, as in Figure 9. User can mark boolean tags on these clone sets, and the FICA system will store these marks immediately into database associated with the user profile. While the user is marking those clone sets, the FICA system is calculating the similarity among those clone sets and training its prediction model by including the user input into its training set, in server side at background. As a result, the feedback of user inputs can be gained nearly real-time in the FICA system in this approach.

6 Experiments

6.1 Experimentation Setup

In order to test the validity and user experience of the proposed method, the following experiment is conducted. We uploaded the source code of 4 open source projects as experimenting 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 these 4 projects are developed in C language. As the parameters passed to CDT part of FICA system, we only detected clone sets that contains more than 48 tokens, and only reported those clone sets from different files. These two parameters are chosen by our experience. Namely, the length limitation less than 48 for C projects are more likely to result un-interesting small clones, as well as for clones from the same source code file. Although too many clone sets will be an obstacle to conduct this experiment, however, those un-interesting small clones should be practical in FICA as well. The CCFinder [4], one of the well-known CDTs, uses a length of 30 tokens as default parameter, which is widely accepted in both industrial and academic studies, while its definition of a token is compacted.

There are all together 33 people participated in this experiment with different experiences about code clone detecting techniques. They were required to mark those clone sets found by the CDT built in FICA system as interesting or not. They were told the steps of the experiments but not the internal methods to be used in FICA system, which means that they were not aware that FICA system will compare these clone sets by textual similarity of lexical token types.

6.2 Code Clone Similarity with Classification by Users

We adapted Force-Directed Graph [24](referred as FDG) to show the similarity among code clones in 4 projects, as in Figure 10. For each project in FDG, a clone set is represented as a circle drawn with red and blue. Blue circles means interesting code clones and red means un-interesting code clones. There are circles with half red and half blue, which means people have different opinions on these clone sets, with larger portion of color represents more people, and the color in the middle of a circle is always the majority of selection.

For each pair of clone sets, there is a link between the circle if the similarity between the clone sets is greater than a threshold, which is adjustable from the web interface in real-time.Values of similarity are assigned as force strength on the links between circles of clone sets. We used the

Table 1: Survey of clones in bash-4.2

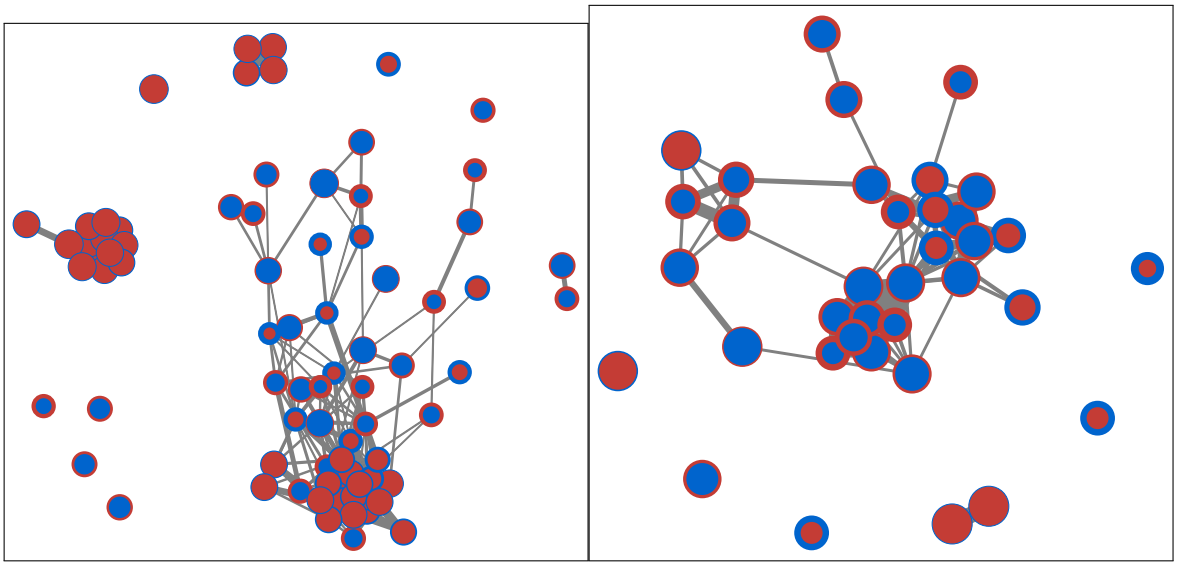
Clone ID	Participant			
	Y	S	M	U
1	X	O	O	O
2	X	X	O	O
3	O	X	X	O
4	X	O	X	O
5	X	O	X	O
...	...			
Count of O	5	24	23	25
Count of X	100	81	82	80

Table 2: Open Source Projects Used in Experiments

Project & Version	CSs [a]	LOC	Tokens	Files	P [b]
git v1.7.9-rc1	78	153,388	829,930	315	33
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	33

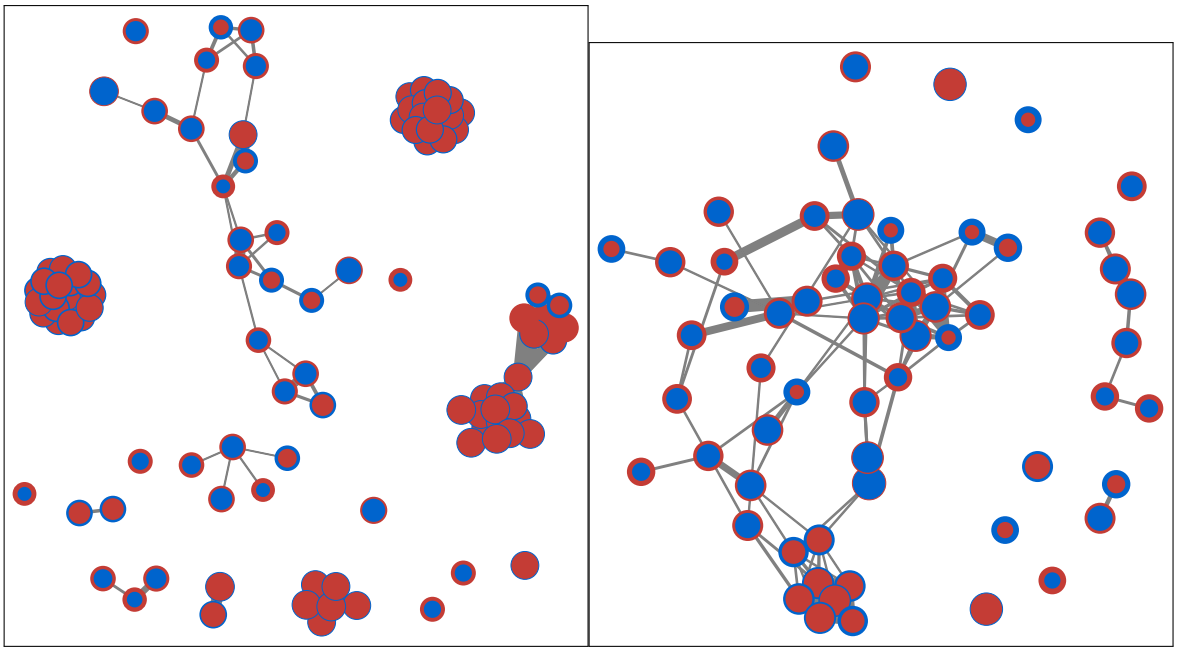
[a] The number of clone sets found in projects.

[b] The number of subjects who have finished the experiment of this project.



(a) Code Clone Similarity of `git`

(b) Code Clone Similarity of `xz`



(c) Code Clone Similarity of `bash`

(d) Code Clone Similarity of `e2fsprogs`

Figure 10: Code Clone Similarity with Classification by Users

`d3.js` javascript library [25] to generate this force-directed graph. Some other technical details about this graph are in Table 3.

The Figure 10 shows that groups of un-interesting clones are more similar, thus their distance are closer to each other in the figure, than groups of interesting ones. This phenomenon indicates that the possibility of un-interesting clone sets calculated by Equation 7 is more accurate than the possibility of interesting clone sets, because un-interesting clones are grouped into categories more tightly. The internal reason that caused this observation is the fact that almost all un-interesting clone sets are fall into categories with certain obvious characteristics of their token types, such as a group of `switch-case` statements, or a group of assignments. Meanwhile the interesting clones are harder to classify into similar categories. To retell in general, interesting code clones are interesting because they are rare and unique.

Thus we conclude our observation 1 and 2:

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

Observation 2 (Uniqueness of Interesting Clones)**.** Interesting code clones are unique comparing to un-interesting ones by means of comparing literal similarity.

6.3 Similarity among Users' Selection

We also draw a FDG of users to illustrate the similarity among their selection, as in Figure 11. Different color in this graph represent the user's experience with code clone detection technology, which is manually selected by these participants during the on-line survey, with options given as in Table 4.

As we can see from Figure 11, selections from participants with more knowledge with code clone techniques(those who selected *author* or *study*) are more similar to each other, while selections by participants with less experience with code clone tends to vary differently. This leads to observation 3.

Observation 3 (Experts Share Common Opinions)**.** Users with more experience on code clones are more likely to agree with each other compared to users with less experience.

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

$$major = \max\{|\texttt{interesting}|, |\texttt{un-interesting}|\} \quad (8)$$

$$minor = \min\{|\texttt{interesting}|, |\texttt{un-interesting}|\} \quad (9)$$

Table 4: User Experience of Code Clone Categories

Category	Means
Author	"I Implemented a Clone Detection Tool"
Study	"Code clone is one of my research topics"
Use	"Using code clone detectors in works"
Lecture	"Heard a lecture of code clone"
Never	"Never heard of code clone"

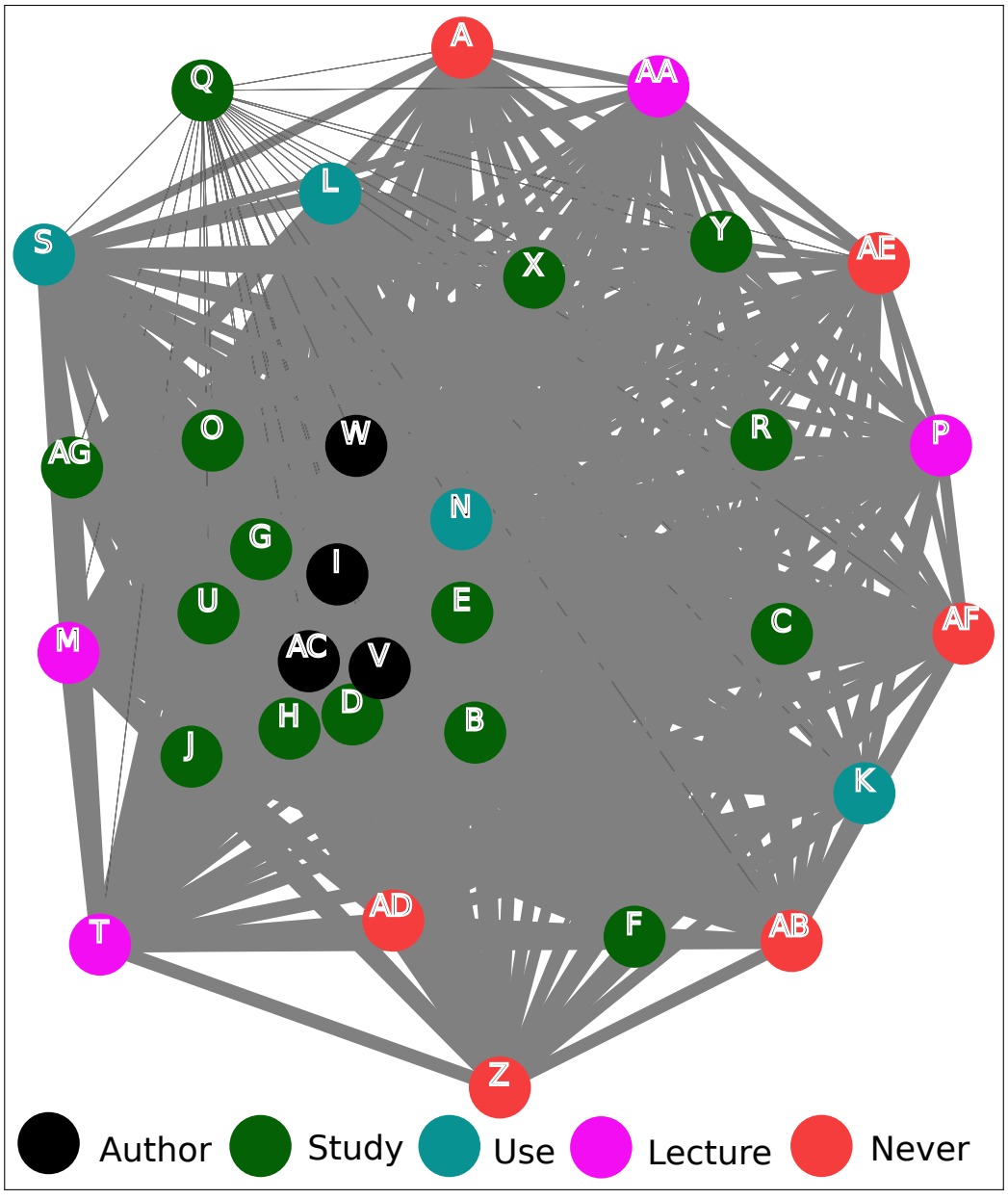


Figure 11: Similarity among Users' Selection

6.4 Ranking clone sets

To measure the quality of supervised machine learning process of FICA, we adapt a measure called *Average Percentage true Positive Found*(referred as APPF) that proposed by Lucia et al. [26]. Their study is similar to our problem as they also reordered the clone list according to the structure similarity.

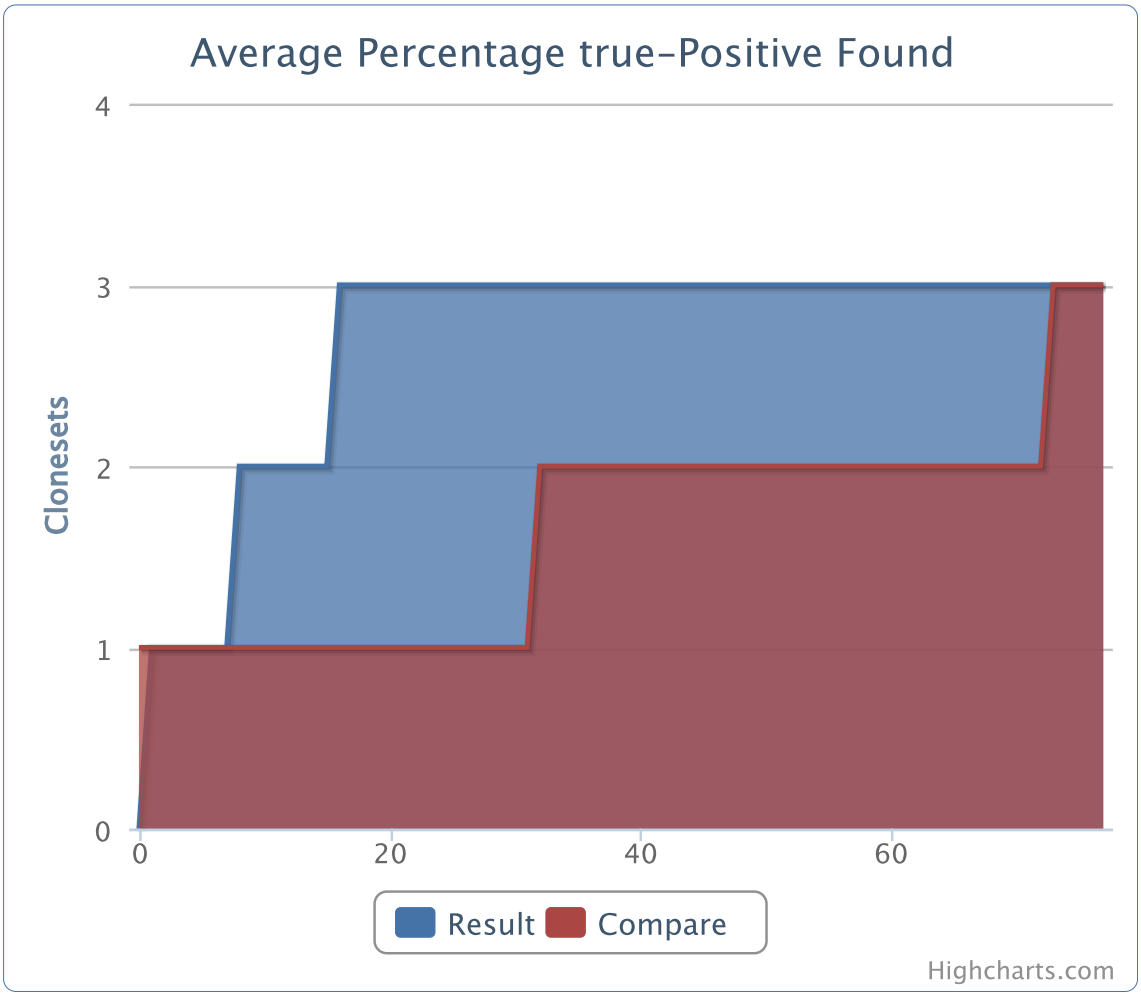
We treat the *true* and *false*, *positive* and *negative* result as the following definition. For each clone set in evaluation set, if the user marks it as interesting then it is considered as *true* result, otherwise as *false* result, and if FICA predicted it as interesting, then it is considered as *positive* result, otherwise as *negative* result.

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

An example of APPF graph is in Figure 12a. In APPF graph both X-axis and Y-axis are the number of clone sets, where the value of X-axis is the number of clone sets that presented to the user while the value of Y-axis is the number of clone sets that the user found interesting. There are two cumulative curves on the graph. The *result* curve in blue is the result after re-ordering by our tool, while the *compare* curve in red is the original order presented by CDTs.

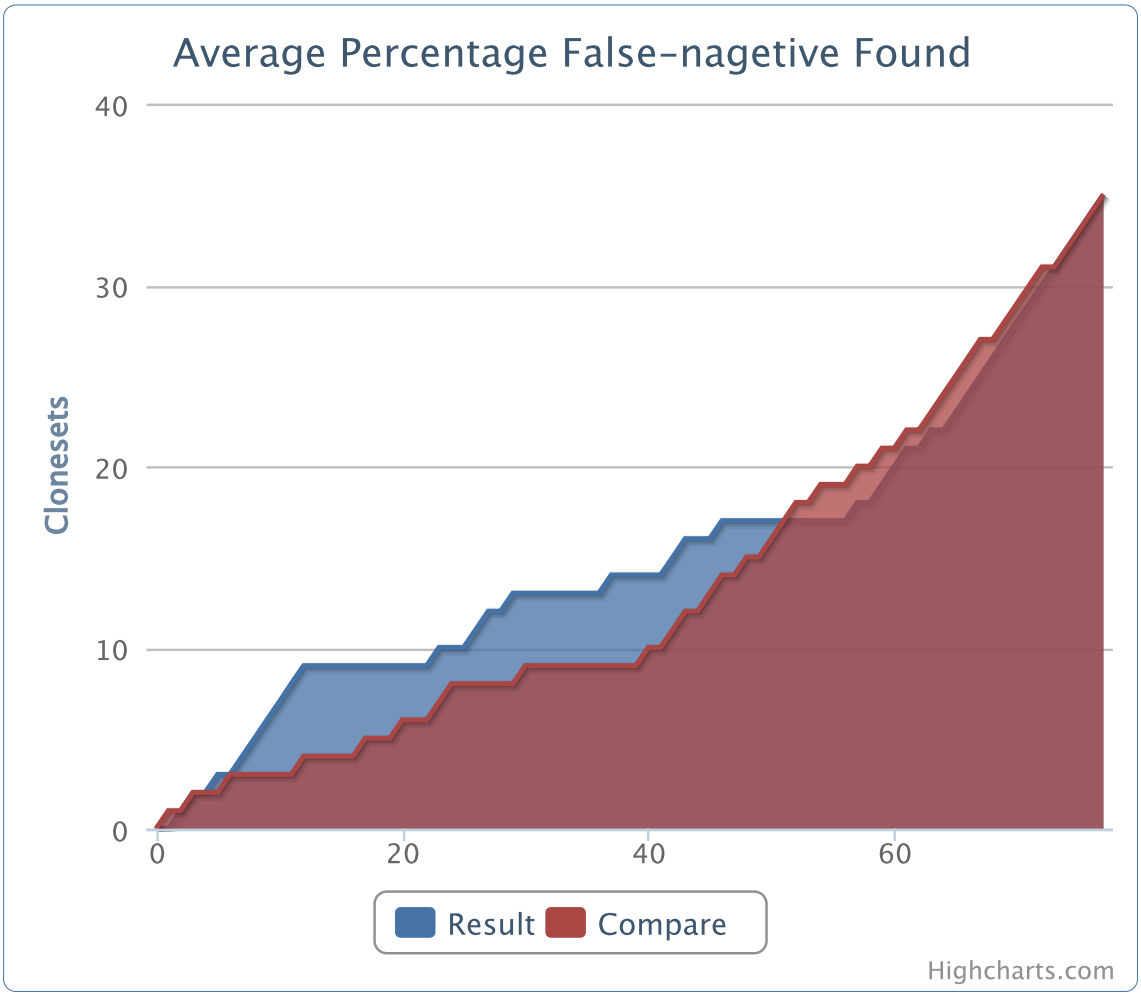
In Figure 12a, there are 78 clone sets found by CDT, only 3 out of them are thought as interesting by the user. As we can see from the *compare* curve, these 3 interesting clone sets are distributed in the very beginning, the middle, and near the tail of clone set list. In the *result* curve, our method successfully rearranged the order of clone list, so that all 3 interesting clone sets appeared at the relatively 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 un-interesting clone sets, we also defined *Average Percentage False negative Found*(referred as APFF) similar to the definition of APPF, which is shown in Figure 12b. As we can see in APFF graph, in the beginning the algorithm generate more false-negatives, and then after around 64% of the clone list, the result fails to be no better than the compared result. This behavior of the algorithm is expected as by re-ordering the list of clones, there is a small possibility to move an desired clone to the end of the list if it is



(a) APPF

Figure 12: APFF of 2 Users on Git Project



(b) APFF

Figure 12: APFF of 2 Users on Git Project (cont.)

similar to those are previously marked as not desired by the user.

6.5 Accuracy of Prediction by FICA

We trained the FICA system by using the marked data of 8 users on all clone sets for each project. The accuracy of machine learning was shown in Figure 13.

The horizontal axis of the figure is the percentage of the training set and the vertical axis is the radio of how many predictions by FICA are equal to the selection of the user. There were three steps 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 are used as the comparison set.

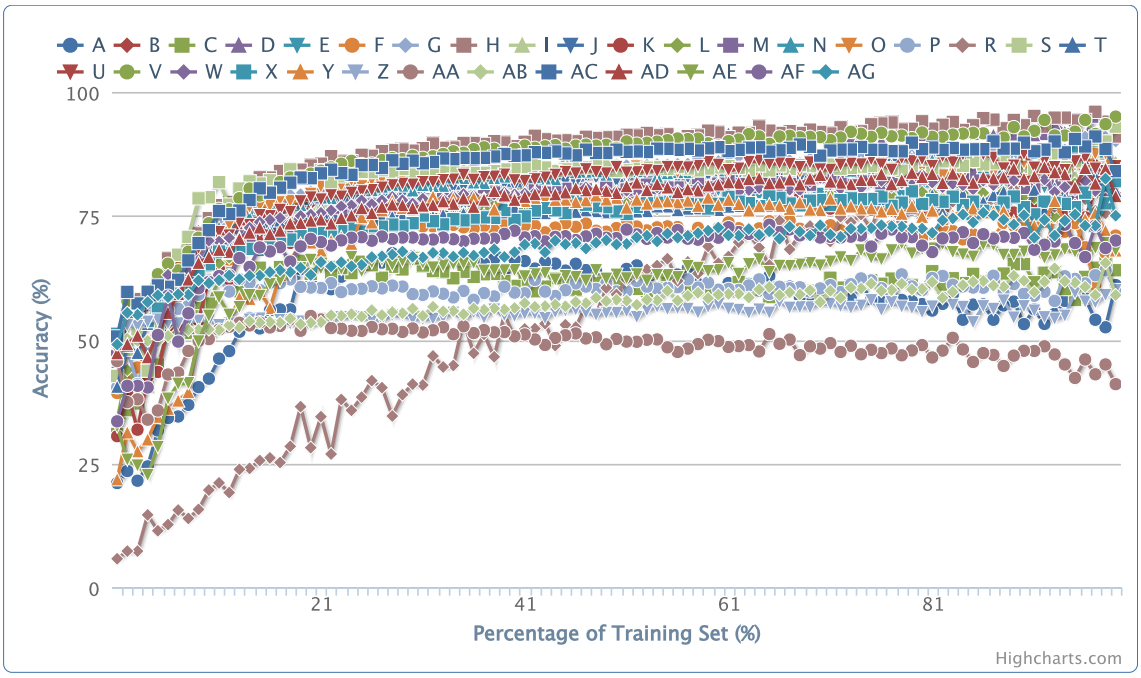
2. For a division of training and comparison set, FICA trained its machine learning model with the training set and calculate a prediction for each clone set in the comparison set.

3. FICA compared the predicted result with the mark of user had made, and calculated the accuracy.

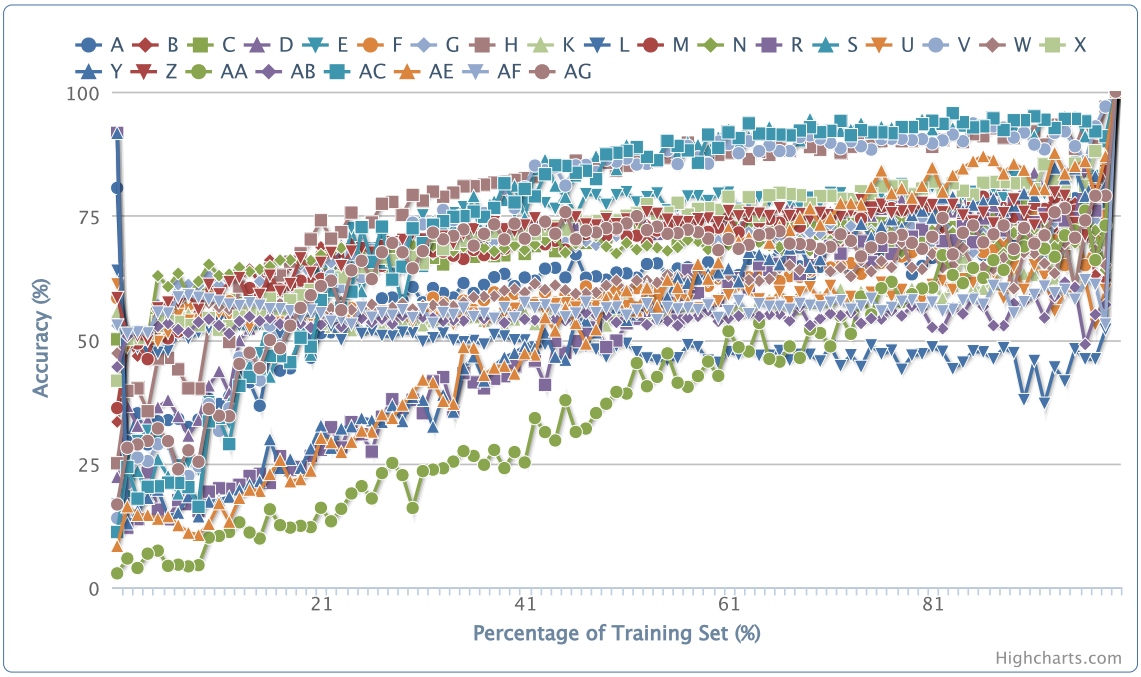
We repeated these 3 steps for 256 times for each size of the training set, and the plotted value of accuracy in the Figure 13 is the average of result of 256 times prediction.

We can see from Figure 13 that the prediction results are changing with users and target projects. For all those 4 projects and nearly all users, we can see that the accuracy of the prediction model is growing with the portion of training set. As the training set growing, in the beginning the accuracy of the prediction grows fast until it reached a point, in between 10 % to 30 %, where the growing speed of accuracy is slowed down.

Among all 4 projects, the `bash` project shows the most desirable result that most of the accuracy of prediction is over 80% when the training set is larger than 16%, which is around 17 of all 105 clone sets. The result of `git` project and `e2fsprogs` project is largely depend on the user, where the result of user `H` always achieves more than 90%, meanwhile the result of user `A` and `C` are converging to around 60% and even decreasing when the training set is growing. The reason why the result is not converging to 100% as well as the dropping of accuracy is further discussed in the next subsection.

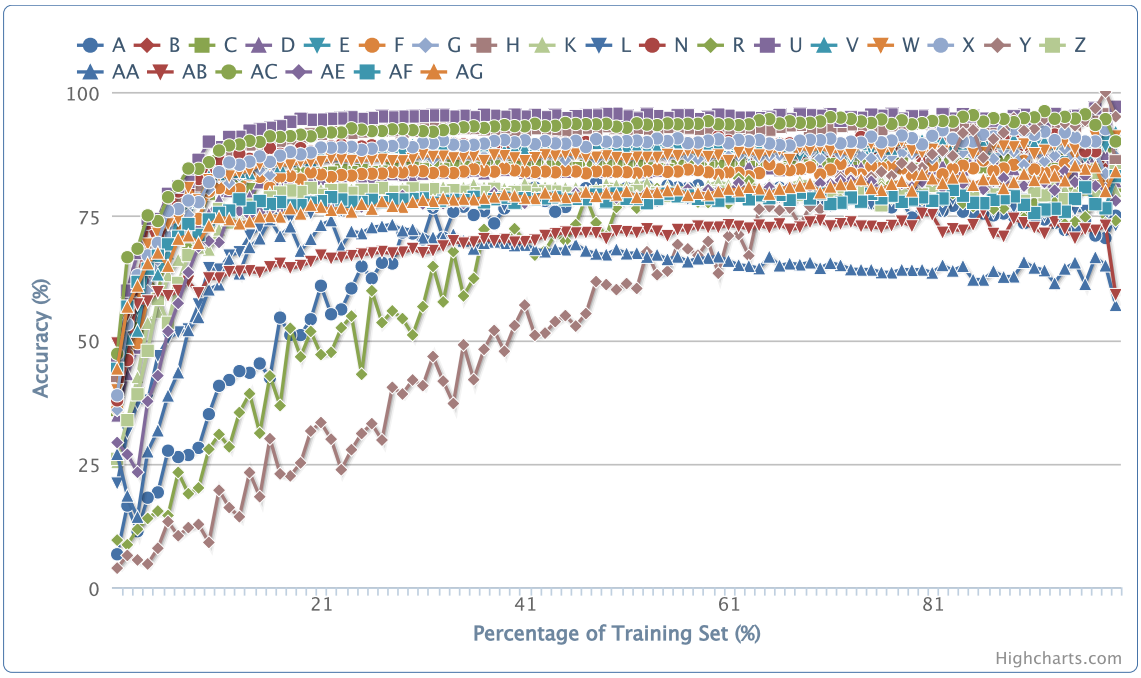


(a) FICA Prediction on `git`

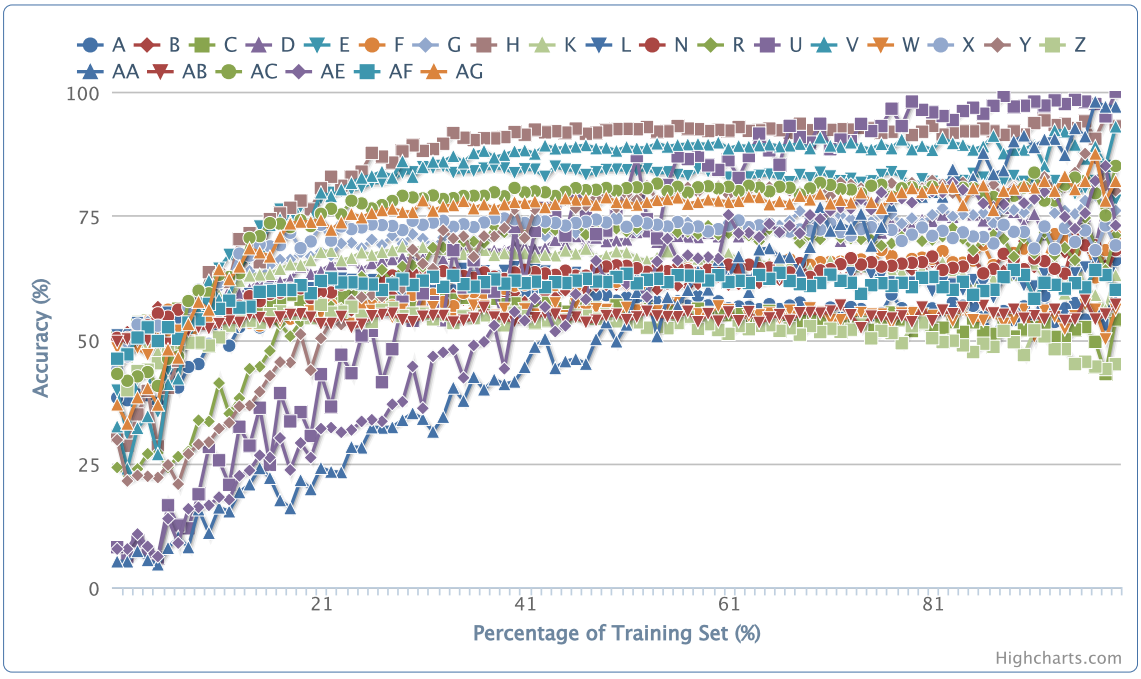


(b) FICA Prediction on `xz`

Figure 13: Accuracy of Machine Learning by FICA



(c) FICA Prediction on `bash`



(d) FICA Prediction on `e2fsprogs`

Figure 13: Accuracy of Machine Learning by FICA(cont.)

An interesting observation of this "slowered" point is that, it does not change linearly with the number of clone sets grows among projects, as shown in Table 5. The percentage of the "slowered" point is changed along with the scale of the project, but the number of clone sets is not changed greatly in all projects. To further support this conclusion, we merged all clone sets from 4 projects into a single project and repeated the above experiment again, the result is shown in Figure 14 and the "slowered" point is also measured in last line of Table 5. As we can conclude from this phenomenon, the number of clone sets that are necessary for machine learning does not grows linearly with the scale of the project or the amount of the detected clone sets.

By combing the Observation 1, we got our Observation 4.

Observation 4 (Minimum Size of Training Set)**.** The minimum required size of the training set roughly grows linearly with the number of categories that clone sets fall into, which is less than a magnitude of the total number of detected clone sets.

To illustrate that the training data of selections by users can be applied across different projects, we merged all clone sets from 4 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 of a single project except for `bash` project.

We can also observe from Figure 13 and 14 that the different behavior of users are clearly separated into different levels. Predictions for some particular user, namely user `R` and user `V`, are always low, which means there is less literal similarity among the clone sets he marked as interesting. And prediction for User `H` is always high for all projects. This result shows the

Table 5: "Slowered" point of Growing in Predicting Accuracy

Project	Total	Slowered (%)	Slowered clone sets
git	78	21%	16
xz	36	43%	15
bash	105	16%	17
e2fsprogs	62	28%	17
All projects	281	8%	22

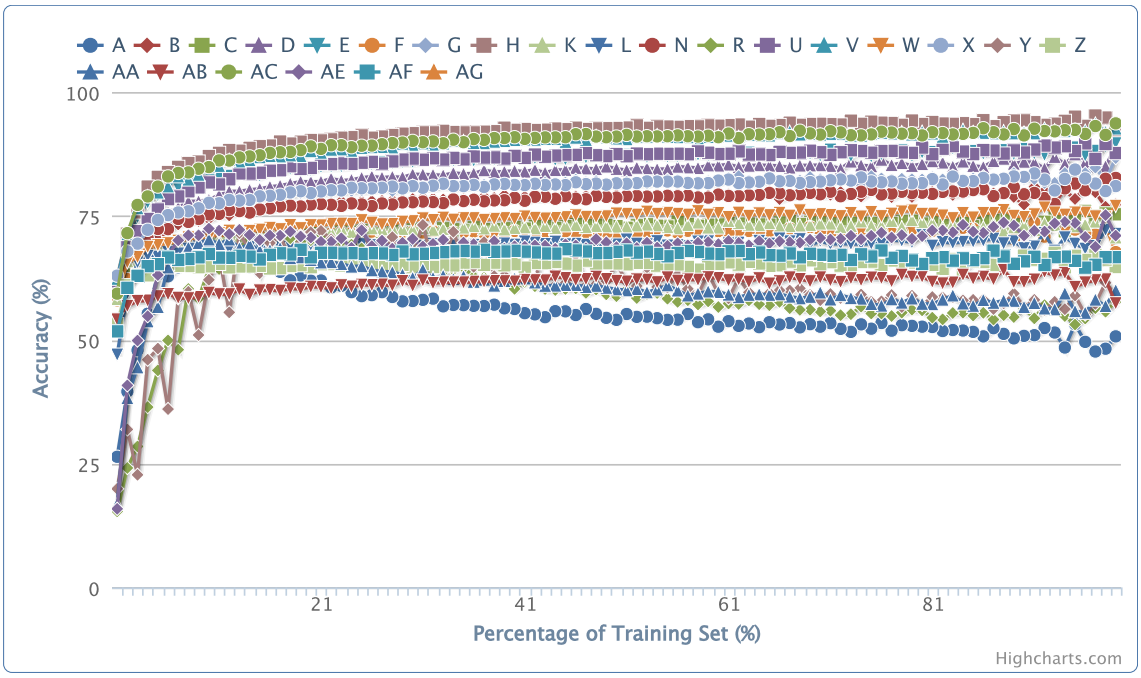


Figure 14: Merged Result of All Projects

consistency of user behaviors.

6.6 Recall and Precision of FICA

We also measured the recall and precision for separated true positives and false negatives of FICA, as a complementary to our definition of accuracy. Firstly we referred *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.

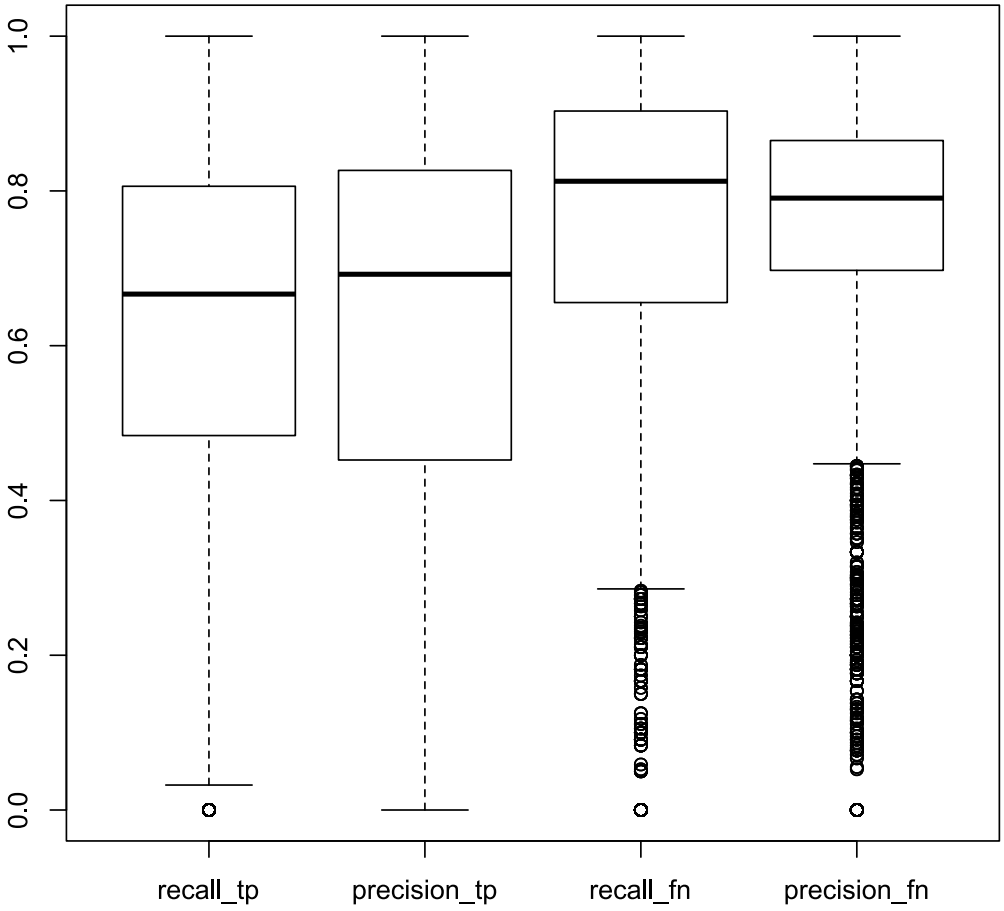
$$recall_{tp} = \frac{tp}{tp + tn} \tag{10}$$

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

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

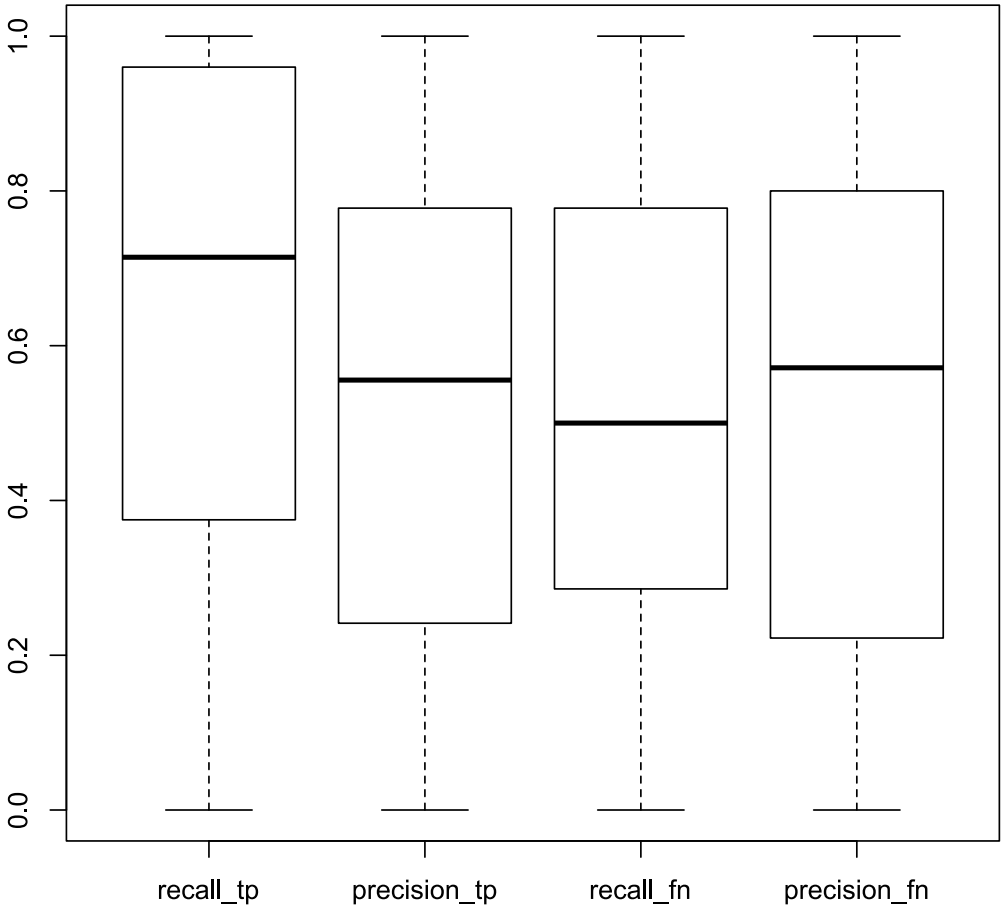
$$precision_{fn} = \frac{fn}{fn + tn} \tag{13}$$

We took 20% of all the clone sets as the training set, the remaining as the evaluation set, and repeated the experiment for 100 times. The result is shown as boxplot in Figure 15. From the



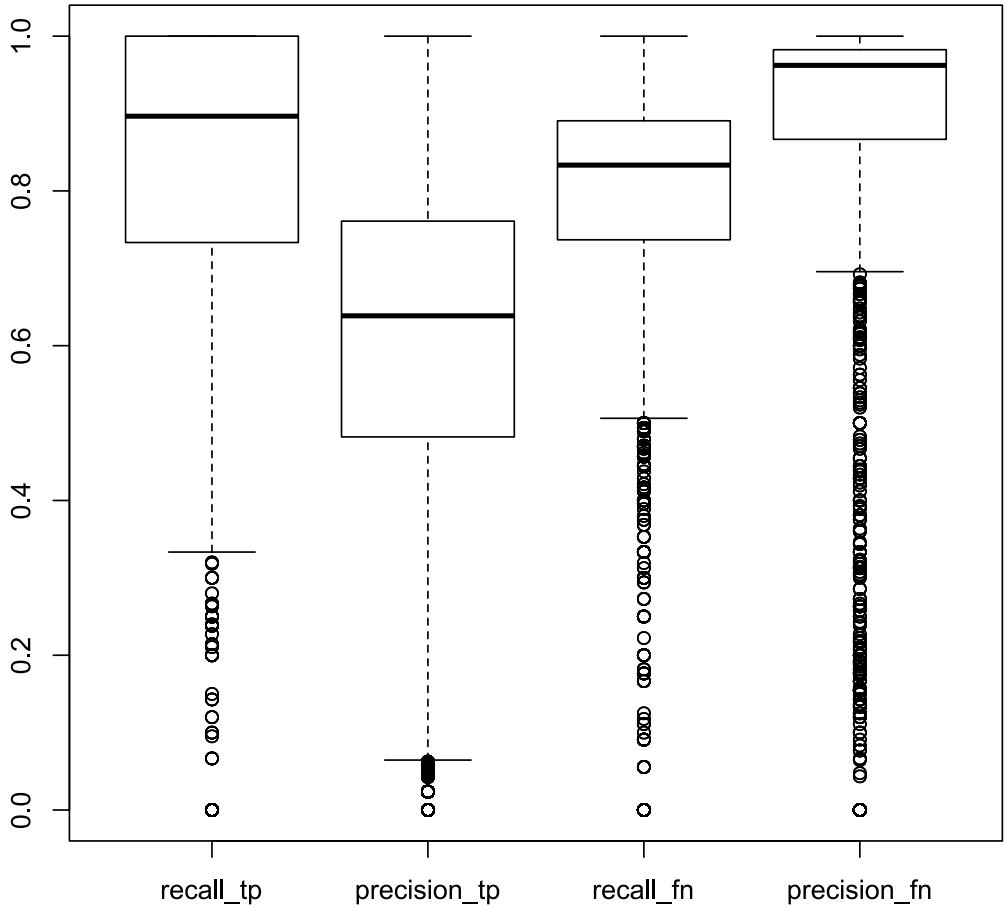
(a) Recall and Precision in `git`

Figure 15: Recall and Precision of FICA in each Project



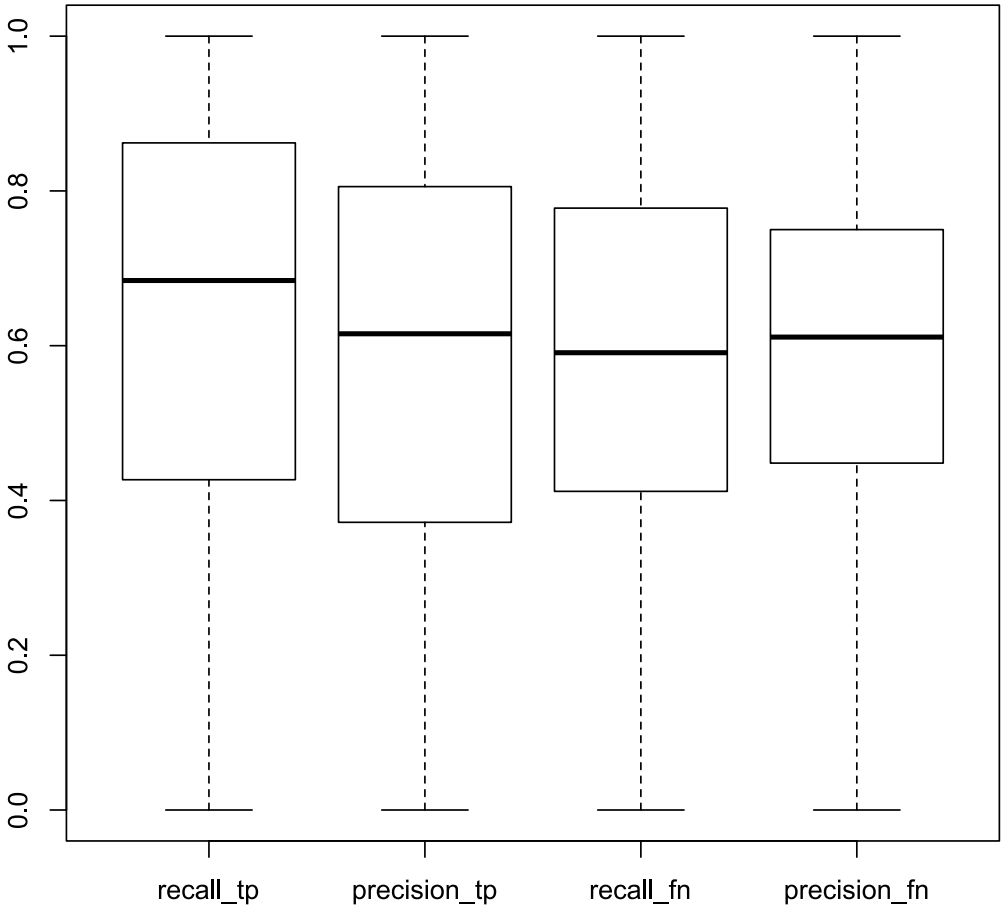
(b) Recall and Precision in `xz`

Figure 15: Recall and Precision of FICA in each Project (cont.)



(c) Recall and Precision in `bash`

Figure 15: Recall and Precision of FICA in each Project (cont.)



(d) Recall and Precision in `e2fsprogs`

Figure 15: Recall and Precision of FICA in each Project (cont.)

```
130     }
131     return LZMA_PROG_ERROR;
132 }
133 static void
134 block_encoder_end(lzma_coder *coder, ...)
135 {
136     lzma_next_end(&coder->next, allocator);
137     lzma_free(coder, allocator);
138     return;
139 }
140 static lzma_ret
141 block_encoder_update(lzma_coder *coder, ... ,
142         const lzma_filter *filters,
143         const lzma_filter *reversed_filters)
144 {
145     if (coder->sequence != SEQ_CODE)
```

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

Figure 16: Example of source code in xz

boxplot we can see a similar result with Figure 13, as `git` and `bash` project show good results while the results for `xz` and `e2fsprogs` are not so comparable. These recall and precision charts showed a trend similar to the accuracy graph in Figure 13, as when there are clustered clone categories in the project, the result will be more appreciate.

6.7 Reason of Converging Results

For all projects in Figure 13 and 14, the accuracy of predictions made by FICA is converging into around 70% to 90% and is hard to achieve 100%. In this sections we will discuss some code fragments to show the reason.

The first example comes from the `xz` project as in Figure 16. There are 2 code clone pairs in these 3 code fragments. The code from Figure 16a in lines 130 to 142 and the code from Figure 16b are the code fragments of the first clone pair, referred as clone α. The code from Figure 16a in lines 136 to 145 and the code from Figure 16c are the code fragments of the second clone pair, referred as clone β. As we can see from the source code, clone α consists of two complete function body, meanwhile clone β consists only two half parts of functions. Thus, from the point of view

```
61      }
62      return LZMA_OK;
63 }
64 static void
65 alone_encoder_end(lzma_coder *coder, ...)
66 {
67      lzma_next_end(&coder->next, allocator);
68      lzma_free(coder, allocator);
69      return;
70 }
71 static lzma_ret
72 alone_encoder_init(lzma_next_coder *next, ...,
73          const lzma_options_lzma *options)
```

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

Figure 16: Example of source code in xz (cont.)

```
474      else
475          lzma_free(coder->lz.coder, allocator);
476      lzma_free(coder, allocator);
477      return;
478 }
479 static lzma_ret
480 lz_encoder_update(lzma_coder *coder, ...,
481          const lzma_filter *filters_null,
482          const lzma_filter *reversed_filters)
483 {
484      if (coder->lz.options_update == NULL)
```

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

Figure 16: Example of source code in xz (cont.)

```
48      return 0;
49 }
50 static int parse_block(const char *request,
51       const char *desc, const char *str, blk_t *blk)
52 {
53      char *tmp;
54      *blk = strtoul(str, &tmp, 0);
55      if (*tmp) {
56           com_err(request, 0, "Bad %s - %s", desc, str);
57           return 1;
58      }
59      return 0;
60 }
61 static int check_brel(char *request)
```

(a) tests/progs/test_rel.c

Figure 17: Example of source code in e2fsprogs

on the ease of refactoring, clone α is much easier than clone β. One the other hand, the calculated similarity between clone α and β is greater than 43% by Equation 5, which is very high among all other similarity between clone sets, because they share a large part of identical code fragments. As the result, 7 out of 8 users thought that clone pair α was interesting and 6 out of 8 users thought that clone pair β was un-interesting, while FICA always thought they belongs to the same group.

Another example comes from the `e2progs` project as in Figure 17. It is clearly that there are 2 code clone pairs in Figure 17. The code from Figure 17a and Figure 17b forms the first code clone pair, referred as clone γ, and the code from Figure 17c and Figure 17d forms the second, referred as clone δ. People can tell these two clone pairs are different because clone γ are two function bodies and people can simply merge them into one function. But clone δ are two lists of function declarations. But 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 thought they are similar is that they share many common N-grams, such as `STATIC ID ID LPAREN CONST` or `CONST ID TIMES ID COMMA`, which results the high value of similarity.

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

```
44      return 1;
45 }
46 static int parse_inode(const char *request,
47        const char *desc, const char *str, ext2_ino_t *ino)
48 {
49      char *tmp;
50      *ino = strtoul(str, &tmp, 0);
51      if (*tmp) {
52           com_err(request, 0, "Bad %s - %s", desc, str);
53           return 1;
54      }
55      return 0;
56 }
57 void do_create_icount(int argc, char **argv)
```

(b) tests/progs/test_icount.c

Figure 17: Example of source code in e2fsprogs (cont.)

```
68 static errcode_t test_write_blk64(io_channel channel,
69      unsigned long long block, int count, const void *data);
70 static errcode_t test_flush(io_channel channel);
71 static errcode_t test_write_byte(io_channel channel,
72      unsigned long offset, int count, const void *buf);
73 static errcode_t test_set_option(io_channel channel,
74      const char *option, const char *arg);
75 static errcode_t test_get_stats(io_channel channel, ...);
76 static struct struct_io_manager struct_test_manager = {
```

(c) lib/ext2fs/test_io.c

Figure 17: Example of source code in e2fsprogs (cont.)

```
102 static errcode_t unix_write_blk(io_channel channel,
103     unsigned long block, int count, const void *data);
104 static errcode_t unix_flush(io_channel channel);
105 static errcode_t unix_write_byte(io_channel channel,
106     unsigned long offset, int size, const void *data);
107 static errcode_t unix_set_option(io_channel channel,
108     const char *option, const char *arg);
109 static errcode_t unix_get_stats(io_channel channel, ...)
110 ;
111 static void reuse_cache(io_channel channel, ...
```

(d) lib/ext2fs/unix_io.c

Figure 17: Example of source code in e2fsprogs (cont.)

token-metric based approach.

7 Related Research

There were some works on combining machine learning or text mining techniques with code detection to classify or clustering code clones. Marcus and Maletic 2001 proposed a method in [27] 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, rather as a complementary of existing code clone detecting methods. A similar approach by Kuhn et al. 2007 [28] found semantic topics instead of clones from comments and identifies from source code. Another method by Tairas and Gray 2009 proposed in [21] shares some common ideas with the method by above two methods that they both compares code clones by using information retrieved from identifiers, which cares more about semantic information or behavior of the source code rather than syntactic or structure information of the source codes. As a contrast, the method described in this paper compares the tokenized source code of code clones, which focused on the syntactic similarity between code clones. The works by Lucia, et al 2012 [26] shared many general ideas with our works, but they were comparing the structure of code clones, thus enforce more strict requirements on the fragment of found clones, and their experiments are undertaken by only the authors, thus relied more subjective judgments.

Besides comparing text similarity, there were other methods have been proposed to filter unneeded code clones from detecting result. Higo et al. proposed a metric-based approach in [20] to identify code clones with higher refactoring opportunities. Their method calculate 6 different metrics for each code clone, and then represented the plotted graph of these metrics as a user-friendly interface to allow user filter out those unneeded code clones. This user-defined metric-based filtering method has been further automated by Koschke in [29]. Koschke is targeting a different object which is identification of license violations, but has also used metric-based approach, and then applying machine learning algorithms on those metrics to form a decision tree. The result of the decision tree limits the types of metrics to only 2, which are PS(Parameter Similarity) and NR(Not Repeat). These metrics-based methods all have a limited identifying target and thus results a higher accuracy compared to the method proposed in this paper.

Other works on classification or taxonomy of code clones were focused on proposing fixed schemes of common clone categories. Balazinska et al. [30] propose 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 [31] defined three different clone types for the sake of a comparison between different detection tools: exact clones, parameterized clones, and clones that have had more extensive edits. They were aiming at testing 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 implemented in most of CDTs is the scatter plot [32, 33]. Scatter plot is useful to select and view clones in a project scale, but hard to illustrate the relations between clones. Johnson proposed a method in [34] that use Hasse Diagrams to illustrate clusters of files that contains code clones. Johnson also proposed to navigate the web of files that contains clone classes by using hyperlinked web pages in [35]. The force-directed graph used in FICA system described in this paper is highly interactive with all the benefits of a modern web system, thus should be useful in aiding analysis of code clones.

Jiang and Hassan [36] have 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 textual similarity is applied to filter out false positive clones. Second, various levels of system abstraction were used for further scaling 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 what FICA achieved by us. Compared to their method, FICA is more depend on the marks of code clones by users, thus more sensitive but requires more operations by users.

8 Conclusions

We have shown the fact that users of CDT may have different opinions on whether a code clone is "useful" or "interesting" to them. This observation suggested that filter of code clones should as well 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 algorithm on code clones. We build the described system FICA, as a web-based system for proof of concept research purpose. The system consists of a generalized suffix tree [23] based CDT and a web-based user interface that allows the user marks detected code clones and shows ranked result.

We conducted an experiment on FICA system with 33 participants contributed to the result. The result showed several important observations on the characteristics about the interesting-ness of code clones for the users. And our classification model showed more than 70% accuracy in average and more than 90% accuracy for particular user and source code project.

Furthermore, we obtained several observations from the experiments about the interesting-ness of code clones.

- Un-interesting code clones are likely to fall into several categories.
- Interesting code clones are unique comparing to un-interesting ones.
- Users with more experience on code clones are more likely to agree with each other compared to users with less experience.
- The minimum required size of the training set roughly grows linearly with the number of categories that clone sets fall into, which is less than a magnitude of the total number of detected clone sets.

Acknowledgements

I recieved great help and contributions from many people during writing this thesis, including but not limited to my supervisor, professors and upperclassmen. This work could not have been possible without their kindly efforts.

First, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, at the Osaka University, for his considerate support, encouragement, and adequate guidance for this work.

Also, I would like to thank to Associate Professor, Kozo Okano, at the Osaka University for his guidance, valuable suggestions and discussions for this work.

I am also deeply grateful to Associate Professor, Hiroshi Igaki, at the Osaka University for his helpful comments and valuable suggestions.

I would like to express my heartfelt appreciation to Assistant Professor, Yoshiki Higo, at the Osaka University for his zealous coaching, continuous support, and encouragement throughout this work.

My sincere thanks go to all the subjects who spare the time to the survey and experiment, for their effort, comments, and close cooperation for this work.

I would like to thank all of my upperclassmen and friends in the Department of Computer Science of Osaka University, especially the members in Kusumoto Laboratory, including Keisuke Hotta, Yui Sasaki, Yoshihiro Nagase, Kentaro Hanada and Kazuki Yoshioka, for their helpful advices, suggestions and assistance.

Finally, I would like to give extra credits to my fiancée, for her encourage, support and comments about this thesis.

References

[1] C.K. Roy, J.R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[2] Y. Higo, S. Kusumoto, and K. Inoue. A survey of code clone detection and its related techniques. *IEICE Transactions on Information and Systems*, Vol. 91-D, No. 6, pp. 1465–1481, June 2008. (in Japanese).

[3] T. Kamiya, Y. Higo, and N. Yoshida. Evolving and hot topics on code clone detection techniques. *Journal of Computer Software*, Vol. 28, No. 3, pp. 28–42, Aug. 2011. (in Japanese).

[4] 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, pp. 654–670, 2002.

[5] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the 14th International Conference on Software Maintenance*, pp. 368–377, Mar. 1998.

[6] J.H. Johnson. Substring matching for clone detection tools. In *Proc. of the 10th International Conference on Software Maintenance*, pp. 120–126, Sep. 1994.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the 15th International Conference on Software Maintenance*, pp. 109–118, Aug. 1999.

[8] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Transcations on Software Engineering*, Vol. 32, No. 3, pp. 176–192, Mar. 2006.

[9] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 253–262, Oct. 2006.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *Proc. of the 29th International Conference on Software Engineering*, May 2007.

[11] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, 2001.

[12] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. the 8th Working conference on Reverse Engineering*, pp. 301–309, Oct. 2001.

[13] Y. Higo and S. Kusumoto. Code clone detection on specialized pdgs with heuristics. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 75–84, Mar. 2011.

[14] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the 12th International Conference on Software Maintenance*, pp. 244–253, Nov. 1996.

[15] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 283–292, Sep. 2011.

[16] Y. Sasaki, T. Yamaoto, Y. Hayase, and K. Inoue. File clone detection for a large scale software system. *IEICE Transactions on Information and Systems*, Vol. J94-D, No. 8, pp. 1423–1433, Aug. 2011. (in Japanese).

[17] N. Göde and R. Kosheke. Incremental clone detection. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering*, pp. 219–228, Mar. 2009.

[18] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Proc. of the 26th International Conference on Software Maintenance*, pp. 1–9, Sep. 2010.

[19] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental code clone detection: A pdg-based approach. In *Proc. of the 18th Working Conference on Reverse Engineering*, pp. 3–12, Oct. 2011.

[20] Y. Higo, S. Kusumoto, and K. Inoue. 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*, Vol. 20, No. 6, pp. 435–461, 2008.

[21] Robert Tairas and Jeff Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, Vol. 14, pp. 33–56, 2009. 10.1007/s10664-008-9089-1.

[22] K.S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, Vol. 28, No. 1, pp. 11–21, 1972.

[23] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, Vol. 14, No. 3, pp. 249–260, 1995.

[24] S.G. Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

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

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

[27] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *16th Annual International Conference on Automated Software Engineering*, pp. 107–114. IEEE, 2001.

[28] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, Vol. 49, No. 3, pp. 230–243, 2007.

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

[30] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Sixth International Software Metrics Symposium*, pp. 292–303. IEEE, 1999.

[31] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[32] K.W. Church and J.I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, pp. 153–174, 1993.

[33] Y. Higo. *Code clone analysis methods for efficient software maintenance*. PhD thesis, Osaka University, 2006.

[34] J.H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, p. 32. IBM Press, 1994.

[35] J. Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pp. 16–. IBM Press, 1996.

[36] Z.M. Jiang and A.E. Hassan. A framework for studying clones in large software systems. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 203–212. IEEE, 2007.