2IMP25 - Software Evolution

Requirement Traceability

Quartile 3 - 2019-2020

Group 50

Full Name	Student ID	Email
Yu Zhong	1416804	y.zhong@student.tue.nl
Xin Zhao	1329553	x.zhao1@student.tue.nl

Eindhoven, March 10, 2020

Contents

1 Introduction

Clones widely exit in any software system. It is easy to understand this phenomenon since programmers may prefer copy-paste programming from existing code to writing code from scratch. Clones are divided into four categories. Type 1 clones, also named as copy-paste clones, have identical physical lines of code and may have differences in white-space and comments; Type 2 clones refers to structurally similar code fragments with rename variables, different types, or method calls; Type 3 clones are similar code fragments but different in removed, added or modified statements; Type 4 clones refers to code fragments implementing the same functionality.
However, as software grows, it becomes more difficult to maintain a clones-existing system. For example, if a bug is detected in any section of the code, correction is required in all replicated code fragments, which creates the need to detect clones in the source code and relate these fragments [10]. Apart from detecting clones within a same version of source code, finding duplicated code fragments between different versions can reveal the similarity between two versions in the process of software evolution. In [5], Livieri made use of clone detection to analyse the feature of the evolution process of linux kernel.
In this assignment, we will follow the method provided by [5] to analyse the the evolution of jQuery. Instead of D-CCFinder, we utilized a tool named JsInspect to detect clone fragments. However, before making use of it, we need to see its capability on detecting different types of clones. We created a basic snippet and a set of Type 1, 2, 3 clones and see to what extend JsInspect can detect them. The manually constructed clones are described in Section 3, the results and the capability of JsInspect are discussed in Section 4. Then we implemented the method from [5] generate a heat map. Section 2 describes our own method to calculate similarity values, scripts and parameters to generate such results. We analysed these results in Section 5.

2 Methodology

In the given file *prep.py*, different versions of jQuery from 1.0 to 3.4.1 are cloned and stored into the folder *input*. However, not all the text files in the repository are source files. Thus we need to figure out which files should be excluded when recognizing and counting the code clones. The *src* folder is usually the place where source files are located and thus we exclude the files in other folders at the same level, including *build* folder, *test* folder, *speed* folder, *dist* folder and *external* folder. This is not enough, however, since there may be some test file for small modules in the *src* folder for some versions of jQuery, for example, the javaScript file *selectorTest.js* for the version 1.1.3. Also, there are exceptions in some versions, like 1.10.0, where *jquery.js* is located outside the *src* folder.
Based on the analysis above, we extracted the names of javaScript files and folders that we want to exclude through the script *to_ignore.py* which will return a string with all the file names we want to exclude. The string we got from the script is:

- "Gruntfile\.js | grunt\.js | build | jquery\.min\.js | test | external | dist | jquery-migrate\.js | jquery-migrate\.min\.js | \.github | speed | \.git | intro\.js | outro\.js | Test\.js"

After having the string stating excluded files, we can use it as configuration when running JsInspect. Table 1 shows the configuration of running JsInspect on different versions of jQuery.

Parameters	Statement
−I	Do not match identifiers.
−L	Do not match literals.
−−ignore <regex>	Do not match files or folders whose names match the regular expression.
> <path>	Generate output text file.

Table 1: The configuration used to run JsInspect on different versions of jQuery.

Hence, the command to run JsInspect is:

- jsinspect -I -L –ignore "Gruntfile\.js | grunt\.js | build | jquery\.min\.js | test | external | dist | jquery-migrate\.js | jquery-migrate\.min\.js | \.github | speed | \.git | intro\.js | outro\.js | Test\.js" > /output/result.txt

The *result.txt* contains all the clone fragments JsInpsect detected and according to this, we can easily calculate the line of clones between two versions. The srcipt *extract_repeat.py* implements the functionality that extracts and adds up the numbers of line of code clones for each pair of jQuery versions from the *result.txt* file and generate a matrix as Figure 1 shows.

	1	1.0.1	1.0.2	1.0.3	1.0.4	1.1	1.1.1	1.1.2	1.1.3	1.1.3.1
1	0	0	0	0	0	0	0	0	0	0
1.0.1	8888	0	0	0	0	0	0	0	0	0
1.0.2	8530	8530	0	0	0	0	0	0	0	0
1.0.3	8347	8347	9632	0	0	0	0	0	0	0
1.0.4	7675	7675	7996	8050	0	0	0	0	0	0
1.1	441	441	591	612	773	0	0	0	0	0
1.1.1	441	441	591	612	730	7265	0	0	0	0
1.1.2	444	444	594	615	733	7182	7693	0	0	0
1.1.3	303	303	446	467	592	1929	2308	2522	0	0
1.1.3.1	315	315	458	479	604	1941	2320	2534	6888	0
1.1.4	318	318	357	365	393	1319	1436	1493	4882	5572

Figure 1: An example of the output matrix for *extract_repeat.py*.

We used *cloc*, a tool to count blank lines, comment lines and physical lines of code, to count how many lines of code each version of jQuery has. The configuration for *cloc* is shown in Table 2 [4].

Parameters	Statement
–not-match-d=<regex>	Do not match files in directories matching the Perl regex.
–not-match-f=<regex>	Do not match files whose basenames match the Perl regex.
–match-f=<regex>	Only count files whose basenames match the Perl regex..
–skip-uniqueness	Skip counting duplicated files (if exist) in a given version of jQuery .

Table 2: The configuration used to run JsInspect on different versions of jQuery.

Compared with *jsinspect*, *cloc* supports more regular expression features, such as operator "^" and "$", which are start and end. Hence the command to run *cloc* on each version of jQuery is:

- cloc –not-match-d " ^speed$ | ^external$ | ^dist$ | ^test$ | ^\.git$ | ^build$ | ^\.github$" –not-match-f " ^jquery-migrate\.js$ | ^jquery\.min\.js$ | ^jquery-migrate\.min\.js$ | ^grunt\.js$ | ^.*Test\.js$ | ^intro\.js$ | ^outro\.js$ | ^Gruntfile\.js$" –match-f "(\.js$) | (\.jsx$)" –skip-uniqueness

We wrote a script named *count_files.py* to run *cloc* on each version of jQuery automatically and generate *.csv* file containing counting results as Figure 2 shows.

version	files	blank	comment	code
1	4	565	3406	1202
1.0.1	4	565	3406	1202
1.0.2	5	623	4772	1381
1.0.3	4	602	4699	1302
1.0.4	4	601	4135	1345
1.1	5	513	3009	1435
1.1.1	5	517	3020	1456

Figure 2: An example of the output for *count_files.py*

After counting the numbers of line of code for the clones and the source code, we can compute similarity value for each two verion of jQuery. We implemented two method to compute the similarity value.
The first one the method is from [5] to calculate the similarity, or code clone coverage ratio of two versions of jQuery:

$$sim(v_i, v_j) = \frac{C_{ij}}{V_i + V_j} \tag{1}$$

where

C_{ij} : the total number of line of code in terms of the clone fragments between version i and j and i $\leq$ j

V_i : the number of line of code in terms of the source code in version i.

The second method we came out with is basically to form the data into vectors and calculate the cosine value between two vectors. To illustrate our idea, suppose that versions v_1 and v_2 have V_1 and V_2 lines of code and C_1 and C_2 lines of code clones respectively. Then the cosine value between vector (V_1, C_1) and (C_2, V_2) is the similarity value. The similarity value will be 0 if C_1=C_2=0 and 1 if V_1=C_1 and C_2=V_2. The formula to compute similarity value is:

$$sim(v_i, v_j) = cos((V_i, C_i), (C_j, V_j)) = \frac{V_i C_j + C_i V_j}{\sqrt{V_i^2 + C_i^2}\sqrt{C_j^2 + V_j^2}} \tag{2}$$

Based on Formula (1) and (2), the functionality to compute the similarity matrix is realised in the script *calculate_sim.py*. Figure 3 shows an example output of it.

	1	1.0.1	1.0.2	1.0.3	1.0.4	1.1	1.1.1	1.1.2	1.1.3	1.1.3.1
1	1	0	0	0	0	0	0	0	0	0
1.0.1	0.859076	1	0	0	0	0	0	0	0	0
1.0.2	0.713867	0.713867	1	0	0	0	0	0	0	0
1.0.3	0.708815	0.708815	0.719934	1	0	0	0	0	0	0
1.0.4	0.68198	0.68198	0.621918	0.634658	1	0	0	0	0	0
1.1	0.043534	0.043534	0.050371	0.052941	0.070031	1	0	0	0	0
1.1.1	0.04338	0.04338	0.050217	0.052777	0.06592	0.730151	1	0	0	0
1.1.2	0.043191	0.043191	0.049987	0.052519	0.065517	0.713633	0.761683	1	0	0
1.1.3	0.02916	0.02916	0.037185	0.039506	0.052394	0.189582	0.226031	0.244262	1	0
1.1.3.1	0.030303	0.030303	0.038173	0.040507	0.053437	0.190687	0.227117	0.245329	0.65977	1
1.1.4	0.030139	0.030139	0.029373	0.030465	0.034296	0.127625	0.138463	0.142394	0.46074	0.52566

Figure 3: An example of the output similarity matrix for *extract_repeat.py*

The final step is to create the heat map. The code is wrapped in the script *heatmap.py*. It utilizes the similarity matrix and generates an interactive html page showing the heat map.

3 Manually Constructed Clones

3.1 Basic Snippet

We wrote a JavaScript function of around 10 lines of code that solve the task of distinguishing prime number. Figure 4 shows the detail for the basic snippet.

```
1   function isPrimeNum(num){
2       var flag = true;
3       var i = 2;
4       var tmp=num/2+1;
5       while(i<tmp && flag){
6           var isPrime = num%i==0;
7           if(isPrime)
8               flag = false;
9           i++;
10      }
11      return flag;
12  }
```

Figure 4: The code for the *isPrime* function

This basic snippet contains the following syntactical constructs:

- Assignment, including not only the assignment for numerical type variables but also the assignment for boolean type variables.
- Mathematical operations, such as binary operations /, +, %, logical binary operation && and unary operation ++.
- Comparison. In our case we used <.
- Looping like the while loop.

3.2 Type 1 clones

Type 1 clones are defined as follow: two code fragments are clones if they are identical up to white-space or comments. Based on the definition, we created four Type 1 clones.

- Type_1_1: The first clone for Type 1 we created is simply adding a few blank lines in the middle of the code. See line 5-7 in Figure 5a.
- Type_1_2: In the second clone, we added a single line comment after an assignment. See line 3 in Figure 5b.
- Type_1_3: In the third clone, we mixed blank lines with the multiple lines comment. We inserted blank lines from line 4 to 5 and from line 10 to 11, and we also added a multiple line comment from line 6 to 9.
- Type_1_4: In previous clones, we only insert blank lines between two lines of code. In the fourth clones, we first inserted white-space within a line, for example in the condition of the while loop (see line 6 in Figure 5d), and then we break a line into two, changing while(...){ to while(...)\n{ (see line 7 in Figure 5d).

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;



    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(a) Type_1_1: the first Type 1 clone.

```
function isPrimeNum(num){
    var flag = true;
    var i = 2; // here is a commend
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(b) Type_1_2: the second Type 1 clone.

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;


    /**
     * here is a multiple line comment.
     * This is a function to distinguish prime number.
     */


    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(c) Type_1_3: the third Type 1 clone.

```
function isPrimeNum(num)
{
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while( i < tmp && flag )
    {
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(d) Type_1_4: the fourth Type 1 clone.

Figure 5: The four Type 1 clones for the basic snippet.

3.3 Type 2 clones

Type 2 clones are defined as follow: two code fragments are clones if they are structurally identical. Based on the classification of cloned methods [2], we created five Type 2 clones:

- Type_2_1: According to the classification of cloned methods, we first considered the single-token difference in the function call. As the Figure 6a shows, in line 1 the function name changes from $isPrimeNum$ to $test$.
- Type_2_2: In the second Type 2 clone, we renamed the variable $flag$ to $result$. These changes can be observed in line 2, 5, 8, 11 of the code shown in the Figure 6b.
- Type_2_3: A cloned method given in the classification is changing the type of local Variable. Therefore we changed the type of a local variable $isPrime$ in the while loop from boolean type to integer type in line 6 of the code shown in the Figure 6c.
- Type_2_4: In the fourth clone, we considered the constant in the snippet. In line 3 of the code shown in Figure 6d, we assigned the constant of 1 instead of 2 to the variable i.

```
function test(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(a) Type_2_1: the first Type 2 clone.

```
function isPrimeNum(num){
    var result = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && result){
        var isPrime = num%i==0;
        if(isPrime)
            result = false;
        i++;
    }
    return result;
}
```

(b) Type_2_2: the second Type 2 clone.

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(c) Type_2_3: the third Type 2 clone.

```
function isPrimeNum(num){
    var flag = true;
    var i = 1;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(d) Type_2_4: the fourth Type 2 clone.

Figure 6: The five Type 2 clones for the basic snippet.

3.4 Type 3 clones

Type 3 clones are defined as follow: two code fragments are clones if they are similar but statements or expressions could have been added, removed or modified. Based on this concept, we created four Type 3 clones:

- Type_3_1: In the first clone, we removed the if statement. See Figure 7a for details.
- Type_3_2: In the second clone, we added a variable *result* into the program. The changes can be seen in line 3 and 12 of the code shown in Figure 7b.
- Type_3_3: In the third clone, we modified the *while* statement and changed it to for statement. The change is shown in line 5 in Figure 7c.
- Type_3_4: In the fourth clone, we reordered the code. We exchanged the position of two assignment statements of $flag$ and tmp.

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        i++;
    }
    return flag;
}
```

(a) Type_3_1: the first Type 3 clone.

```
function isPrimeNum(num){
    var flag = true;
    var result;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    result = flag;
    return flag;
}
```

(b) Type_3_2: the second Type 3 clone.

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    for(;i<tmp && flag;){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(c) Type_3_3: the third Type 3 clone.

```
function isPrimeNum(num){
    var tmp=num/2+1;
    var i = 2;
    var flag = true;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}
```

(d) Type_3_4: the fourth Type 3 clone.

Figure 7: The four Type 3 clones for the basic snippet.

4 JsInspect

4.1 Introduction to JsInspect

JsInspect is a delicated tool for clone detection. It is a command-line tool which understand ES6, JSX and Flow. The methodology it applies is AST-based clone detection. That is, JsInspect use an abstract syntax tree to build the structure of the source code and each node of this tree denotes a construct such as an assignment or declaration. Because of that, it performs good in finding structurally similar code clone [11].

4.2 The results of clones detection on manually construct clones

4.2.1 Type 1 clones

JsInspect successfully detected all four clones in Type 1 without any configuration. Using the command *jsinspect ./../../manually*, the result given by JsInspect is attached in Appendices A.1.

4.2.2 Type 2 clones

When detecting Type 2 clones, we should specific the configuration that it should not match identifiers or literals. Using the command *jsinspect -I -L ./../../manually*, the result given by JsInspect is attached in Appendices A.2. As we can observed in the result, all four clones can be detected.

4.2.3 Type 3 clones

JsInspect can only detect the clone *Type_3_1* if using the default threshold for the number of nodes. We adjusted the parameter to 20 as *jsinspect -I -L -t* 20 *./../../manually*. The result given by JsInspect is attached in Appendices A.3.

4.3 Discussion on the capability of JsInspect

Given the results detected by JsInspect, we can analysis the capability of JsInspect. The capability of JsInspect we discovered is shown in Table 3.

There is no doubt that JsInspect can detect Type 1 clones.
As for Type 2 clone, we can observe from the result, Appendices A.2, that the tool only detect the clones of length 6 from line 1 to line 6 instead of that of total length 12. One possible reason we found is that, in the second clone where we renamed the variable $flag$ to $result$, four lines of code are different because of the renaming. These differences might be too much for JsInspect to detect and only two of the renaming differences were reported. Other single-token different clones for Type 2 can be successfully detected. Most importantly, parameters -I -L help to detect Type 2 clones.
JsInspect does not perform well on Type 3 clones. Even though we reduced the threshold for the number of node from 30 to 20 and JsInspect did report matches between the basic snippet and the clones, we can observe that it actually did not figure out any modification like adding, removing or modifying statements. As Appendices A.3 shows, it only recognized the remaining part of the code.
Therefore, we conclude that JsInspect can only detect copy-paste clones and structurally similar clones. Besides, it may also have some restrictions on finding Type 2 clones, for example it may perform better on single-token differences.

	The first clone	The second clone	The third clone	The fourth clone
Type 1	✓	✓	✓	✓
Type 2	✓	Not fully detected	✓	✓
Type 3	✗	✗	✗	✗

Table 3: The clones detected by JsInspect

5 jQuery Evolution

5.1 jQuery and the version history of jQuery

jQuery, first released in 2006, is "a fast, small, and feature-rich JavaScript library". The latest version of jQuery only occupies 86KB. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers [3]. Briefly, tt is a lightweight JavaScript library that simplifies programming with JavaScript. It is very popular among web developers. A survey shows that 58% of the top 10 million websites use the jQuery library [1].
In order to keep pace with the need of large community, jQuery evolves and new versions appear regularly. The sizes of each version of jQuery is shown as Figure 8 .

5.2 Heat Maps

We recreated the heatmap of Livieri et al. using jQuery Data. The result is shown as Figure 9. Moreover, we used our customized method for computing similarity and generated a heatmap as Figure 10 shows.

5.3 jQuery Discussion

We can observe in the bar chart (Figure 8) that:

- The number of physical line of code basically keeps increasing from versions 1.0.0 to 1.10.2 and then shrinks at version 1.11.0. There is a gap between the number of physical code at version 2.0.0 and those of previous and later versions. We looked into the evolution archive [8] and found that versions 1.11.0 and 2.1.0 were released at the same times and in the changelog the developers focused much on removing or rewriting components to speed up the code to get less overload.

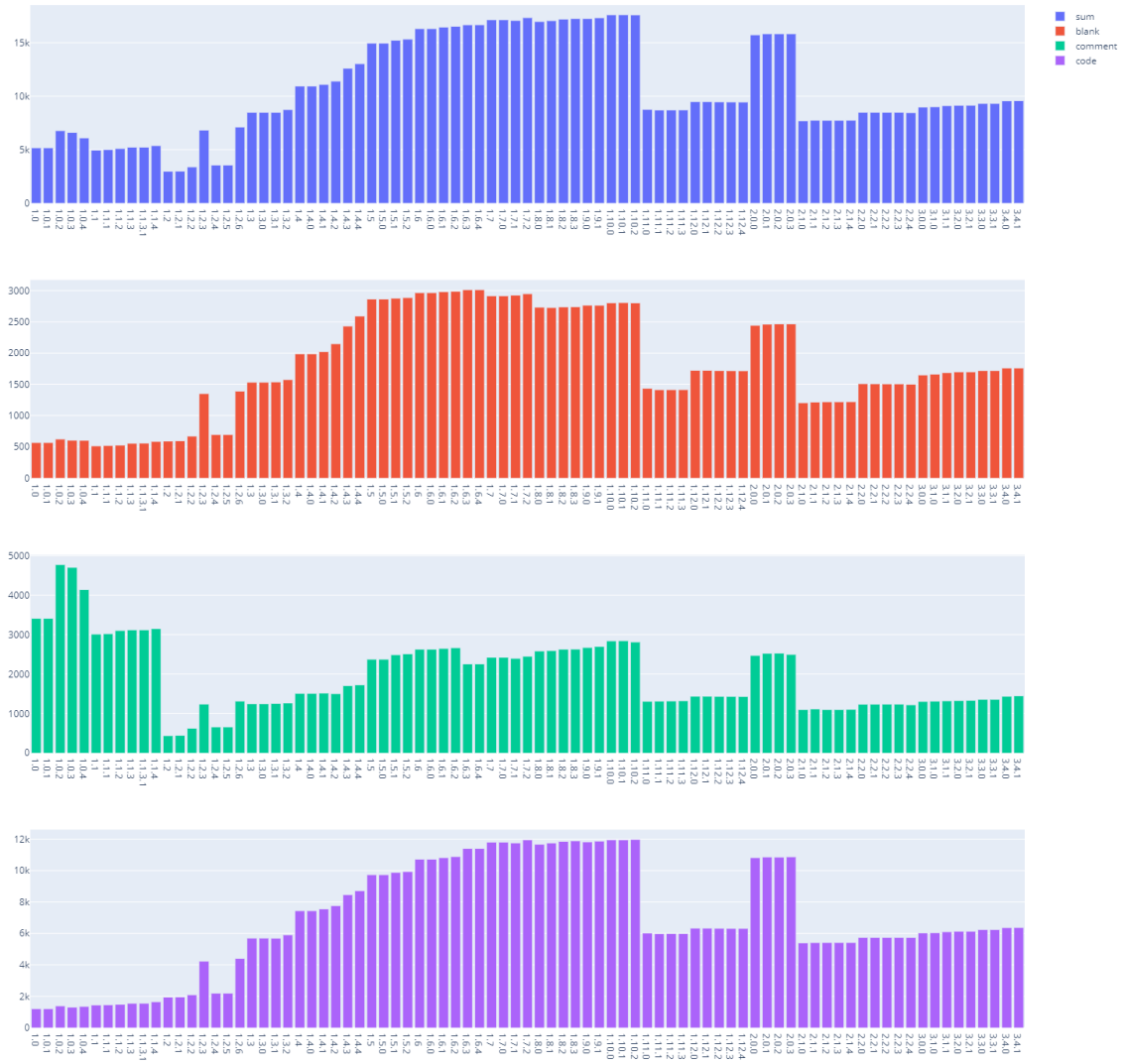

Figure 8: The bar chart plotting the lines of blank, comment, physical code and total code per jQuery version.

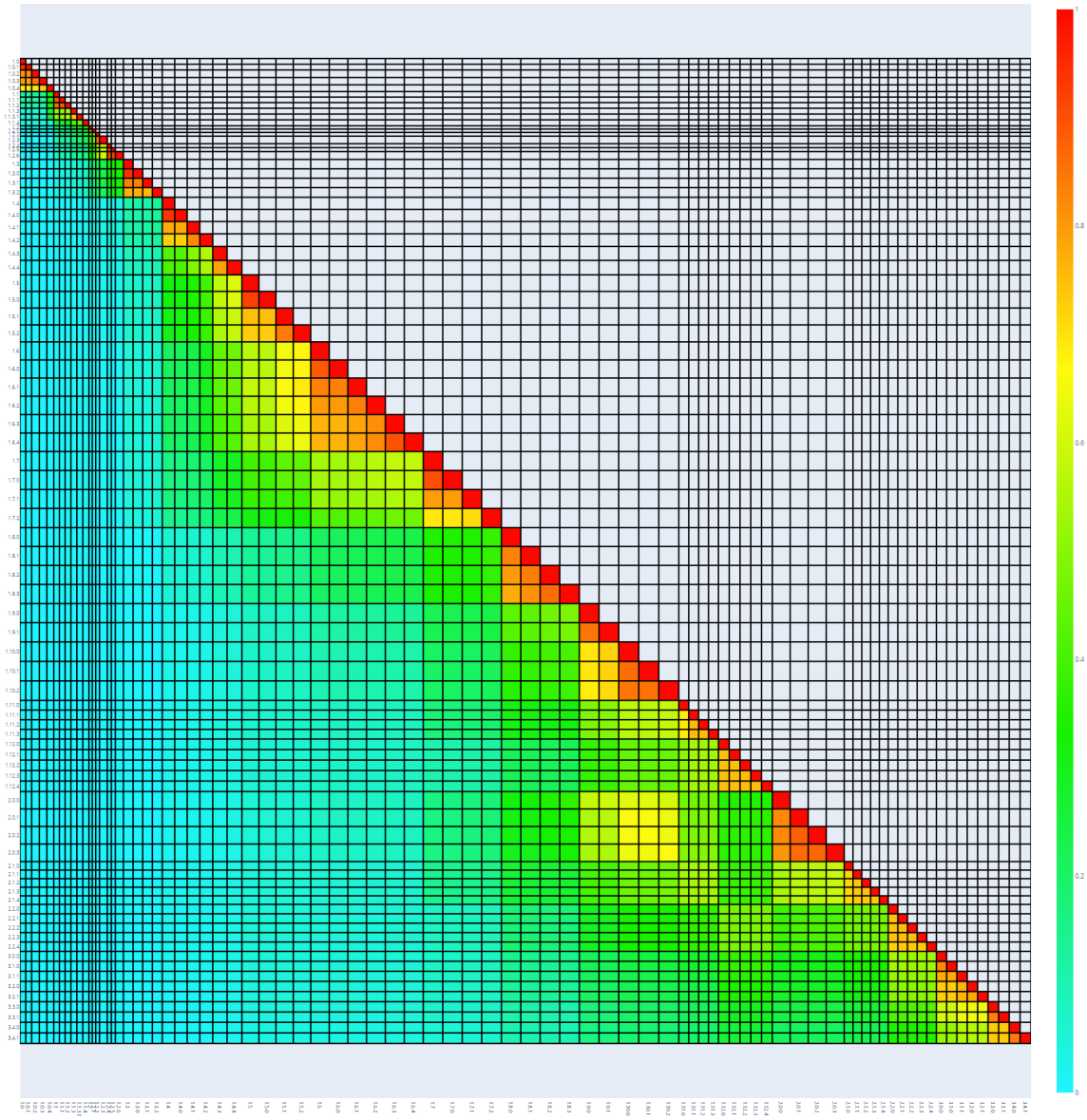

Figure 9: The heatmap of Livieri et al. using jQuery Data.

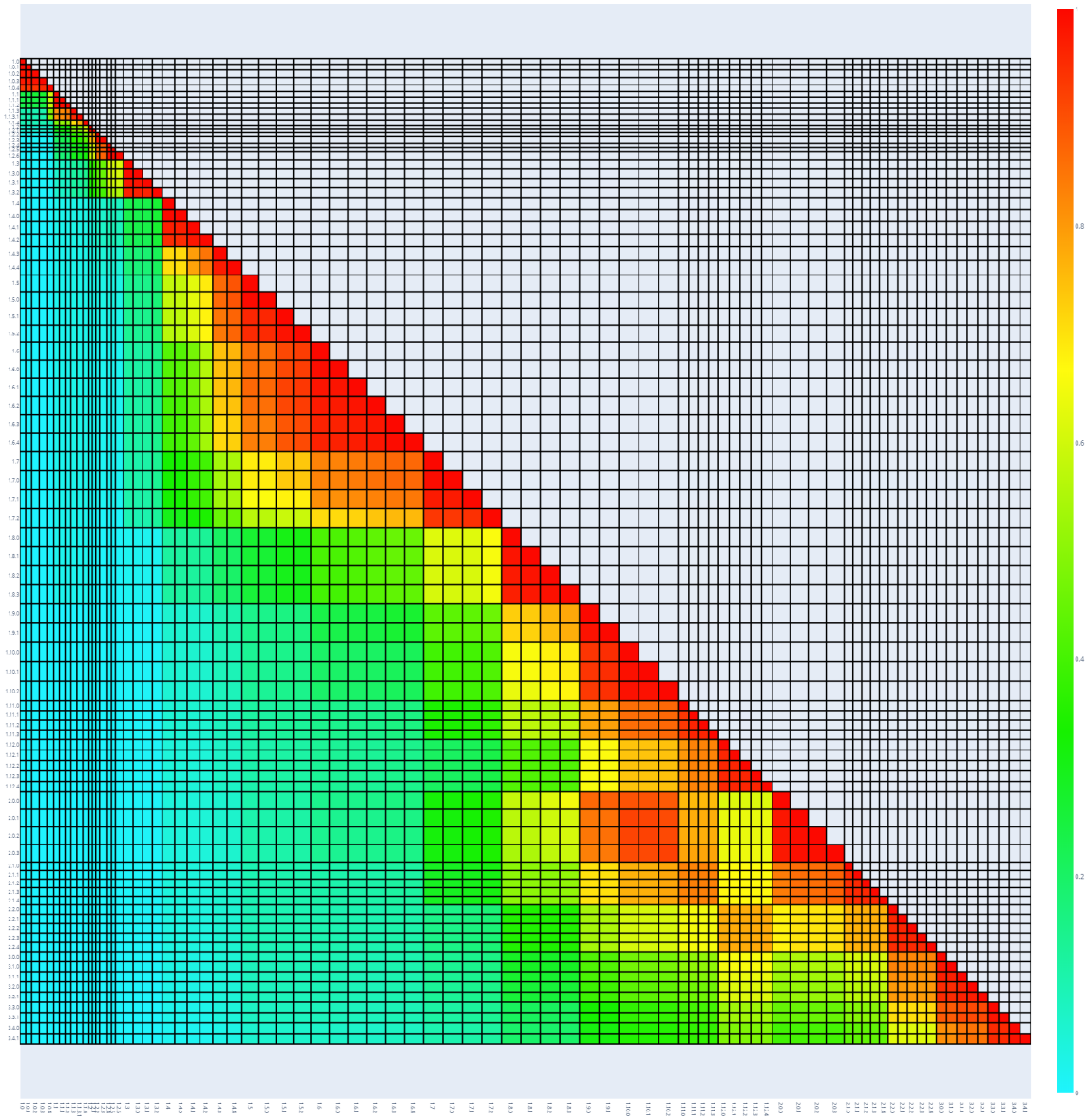

Figure 10: The heatmap of our customized method for computing similarity using jQuery Data.

- A similar pattern shows in the number of physical line of code from version 1.2.3 to 1.2.6 that gaps exit between their numbers of line of code. This is because 1.2.6 is the next release immediately following 1.2.3. Versions 1.2.4 and 1.2.5 were skipped since they were incorrect [6].
- The bar chart of numbers of total lines of code shows a similar pattern to that of physical lines of code except for versions 1.0 to 1.1.4 which have over 3000 lines of comments. In the release of version 1.2, those comments were removed and the number of total lines of code drops from 5378 to 2964.

In order to analyse the evolution of jQuery in a more accurate way, we here mainly focus on the heatmap of Livieri et al. and the data we stated below is from 9. We can observe in Figure 9 and Figure 10 that:

- Most of the highest similarity values gather near the diagonal. This is because, based on the definition in semantic versioning, changes between consecutive releases, or say kernel updates, are usually small, the code clone coverage is high [5].
- However, there are exceptions that versions 2.0.x are more similar to versions 1.10.x instead of versions 1.11.x and 1.12.x. Consequently, versions 2.1.x are more similar to 1.11.x instead of 2.0.x, and 2.2.x are more similar to 1.12.x instead of 2.0.x. In a word, versions 2.0.x are more likely to have closer relationship with 1.10.x. To prove this, we looked into the evolution archive of jQuery where [7, 8, 12] stated the releasing time of each version. Versions 1.10.0 and 2.0.1 were released at the same time in 2013. Later on versions 1.11 and 2.1 were released in 2014 while versions 1.12 and 2.2 were released in 2016. We conclude this is the reason why comparing to versions 1.10.x, versions 2.0.x are less related to 1.11.x and 1.12.x, which to a certain extent violates the definition of major updates in semantic versioning.
- Another observation is that the color patterns of the triangular sectors for most of the minor versions are the same. In detail, the similarity values for versions with the same minor number are usually above 0.65 and thus the colors used in these triangular sectors are usually within the range of red to yellow. For example, for versions 1.6.x, the color for the triangular sector is orange. This is because in semantic versioning the change in the last number should only refer to patching.
- Still there is an exception for the previous observation. For versions 1.4.x, the similarity value between 1.4.2 and 1.4.3 is below 0.65 and hence presented as green color. We looked into the evolution archive [9] and found that in the release of 1.4.3, developers not only fixed some bugs but also made plenty amount of improvements including entirely rewriting the *css.js* and released some new features. These changes results in a similarity value of 0.55.
- No obvious pattern is found for major versions. There are totally 60 releases on the top of major version 1, 14 releases on the top of jQuery 2 and 9 releases on the top of jQuery 3. On the top of jQuery 1, the last version is very different from the first version with the similarity value 0, which indicates a big change during this period. On the other hand, jQuery 2 and 3 does not show such big change.

6 Discussion

6.1 Comparison

In this section we discuss the differences between the heatmaps produced by two methods and the differences in the conclusion between our work and Livieri's work.
The only difference between two methods we used to create the heatmaps is the formula to compute similarity values and hence two heatmaps (Figure 9 and Figure 10) only differ in colors not the size of the grids. We compared the results and found that:

- Our customized method performs worse when distinguishing the similarity within minor versions. For example, our method fails to tell the differences in similarity values for kernel versions on the top of 1.7 since the grids inside this triangular sector are all red while another method can show that version 1.7.1 has closer relationship with 1.7.2 than 1.7.0. On the other hand, our method makes the pattern for kernel evolution within a minor version more clear.
- Our customized method shows more details for nonconsecutive kernel versions that are on the top of the same major number, or say it shows more moderate color transition for grids in the middle of this huge triangle. For instance in Figure 10, it indicates that compared to versions 3.4.x, versions 3.3.x are more similar to 2.2.x but in Figure9 it is less obvious.
- We can analyse from the Formula (1) and Formula (2) that our method may have preference. Take an extreme situation as example to illustrate this. Suppose that:
  - For case 1: v_1 has totally 3 lines of code and 1 line of clone and v_2 has totally 6 lines of code and 5 lines of clone.
  - For case 2: v_1 has totally 3 lines of code and 3 line of clone and v_2 has totally 6 lines of code and 3 lines.

  Basically, this situation may be an abstraction for multiple duplication and longer sector duplication. It is possible that in case 1, the part of v_1 clones duplicates five times in v_2. As in case 2 there is only one time of duplication. Livieri's method cannot tell the difference between case 1 and 2. Luckily, our method can make it by computing higher similarity value for case 2. Hence, we can say that our method prefers longer sector clones rather than multiple clones.

In [5], Livierial et al. conluded some findings on the evolution of Linux. Compared with their findings, we observe that:

- In both cases of Linux and jQuery, high similarity values gather near the diagnal.
- They have different patterns in the kernel evolution within a minor version. In the evolution of Linux, versions 2.0.x, 2.2.x and 2.4.x presents a pattern that there exits a kernel version $a.b.i$ that the similarity for kernel versions from $a.b.0$ to $a.b.i-1$ is high and so as that for $a.b.i$ to later kernel versions. However, the similarity for the former kernel versions and the latter ones is relatively low. As for jQuery, due to the limit number of kernel releases within a minor version (at most 5), no detailed pattern can be driven since they are almost all red.
- In the evolution of Linux, the kernel size continues to grow while some kernel updates in jQuery reduced the size of source code. Moreover, The releasing timelines for kernel versions on the top of major verions 1 and 2 overlaps and leads to some outstanding color blocks in the heatmap.

6.2 Reflection

In Section 5 we found that the evolution process of jQuery partially follows the instructions of semantic versioning. In this section, we mainly discuss the dependency of our results on the choices of we made when running JsInspect and the clone detection capabilities of JsInspect. As Table 1 states, JsInspect is requested to ignore rename variables and literals by adding parameter -I -L. What if we remove them, which means to drop out Type 2 clones, and only focus on copy-paste clones? Figure 11 shows the heatmap generated by such configuration. By comparing Figure 10 and Figure 11, we can find that:

- The overall pattern of the heatmaps would not change even if we drop parameters -I -L. Breifly, high similarity values gather in small triangular sectors near diagonal and outstanding blocks exit for versions 2.0.x and 1.11.x.
- The similarity values for grids inside triangular sectors change negligibly (usually around 0.01), which indicates that the a large proportion of the clones between kernel versions on the top of same minor versions are Type 1 clones. Meanwhile, the similarity values for kernel versions

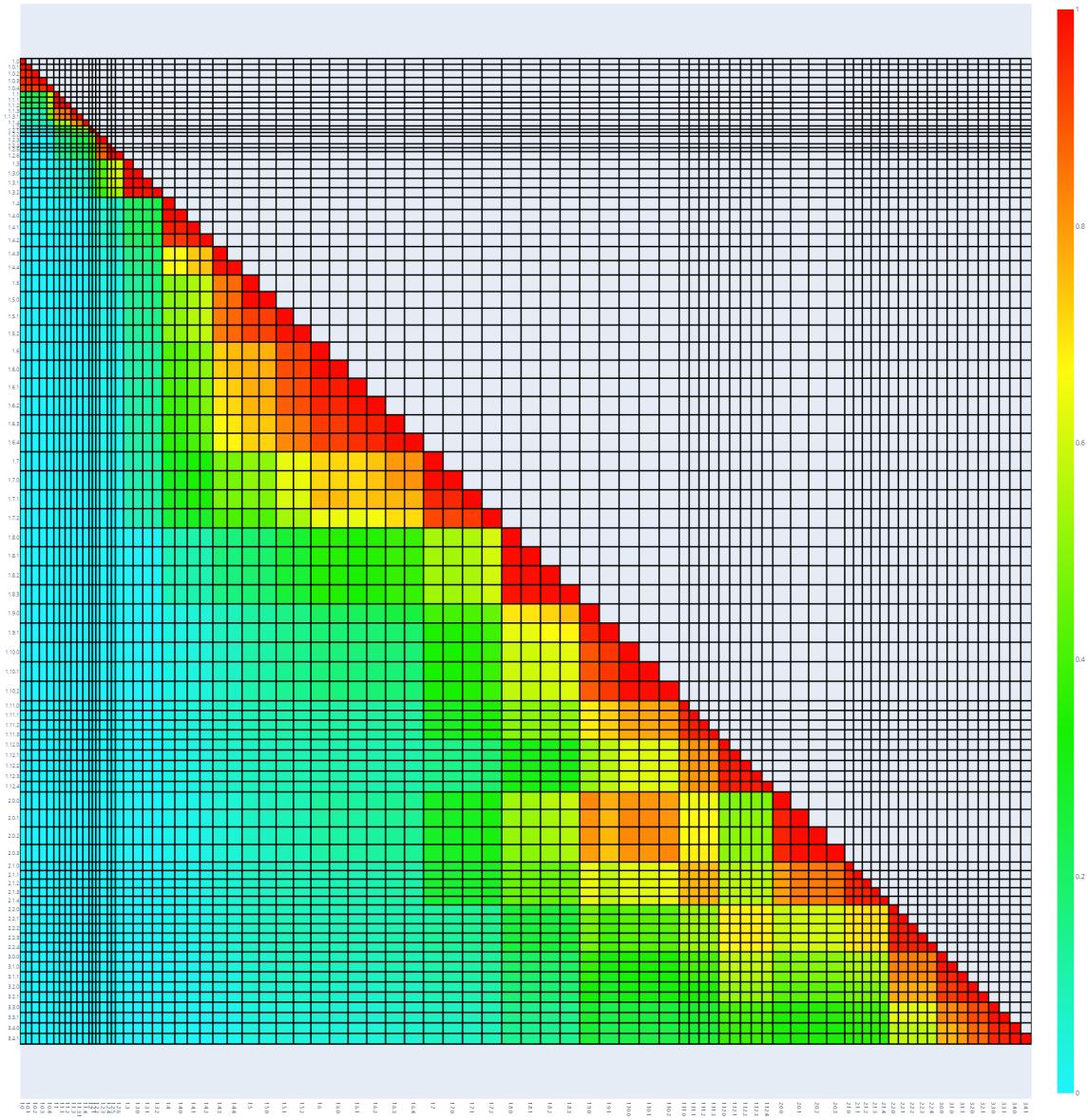

Figure 11: The heatmap of our customized method for computing similarity without parameter -I -L.

on the top of same major versions go down if we do not take Type 2 clones into account. For example, the average similarity values for versions 1.7.x and 1.8.x drop from above 0.6 to around 0.5. Therefore we can say that there are more renaming between nonconsecutive versions than consecutive ones.

Since JsInspect does not perform well in detecting Type 3 clones, we followed the experience from 4.2.3 that instead of detecting the entire snippet, we adjusted the threshold -t to detect the remaining part of the snippet. Then we focus on the parameter -t, which is used as a threshold when detecting clones along ASTs. In our method, we take the default setting with threshold 30. Here we want to see whether there is any difference if we take threshold as 20. The result shows in Figure 12. Based on Figure 10 and Figure 12, we can observe that:

- The overall pattern of the heatmaps would not change even if we add the parameter -t 20.
- As expected, JsInspect detected more clones with a lower threshold. For instance, in Figure 10 the similarity for version 1.1.4 and 1.10.0 is 0 but in Figure 12 it becomes 0.04.

Here it is very tricky to pick a threshold to detect partial sectors of Type 3 clones and the correctness of this method is not proven. Hence we still cannot conclude that our results on the overall pattern are independent on the capabilities of JsInspect but it is likely to be this case due to the comparisons we made.

7 Conclusion

In this assignment, we reproduced the work by Livieri et al. on jQuery data using JsInspect and cloc. In preparation stage, we tested JsInspect on a set of manually constructed scripts and found that it performs better on detecting Type 1 and Type 2 clones. We customized our own method to compute similarity values and generated heatmaps using the metric definition of Livieri et al. and our own one. By analysing these heatmap, we had some findings for the evolution of JQuery and then proved them by looking up the evolution archive. We found that versions 2.0.x is more similar to 1.10.x even if they are not consecutive in semantic versioning. This is because they are actually consecutive releases.
Next in Section 6, we compared the method we customized with that of Livieri et al.. Our method presents detailed color transition for intermediate similarity while the one presented by Livieri et al. can see details for high similarity. Then we compared the results for the evolution of jQuey we got with the results presented in [5]. The overall styles of heatmaps are the same but patterns for kernel updates within a minor version are different. Following the comparison, we discussed to what extent the choices we made can affect our results. By dropping out Type 2 clones and reducing the threshold, the overall pattern of the heatmaps keeps the same. Hence we can say our results on the overall pattern are independent on the parameters -I, -L or -t.

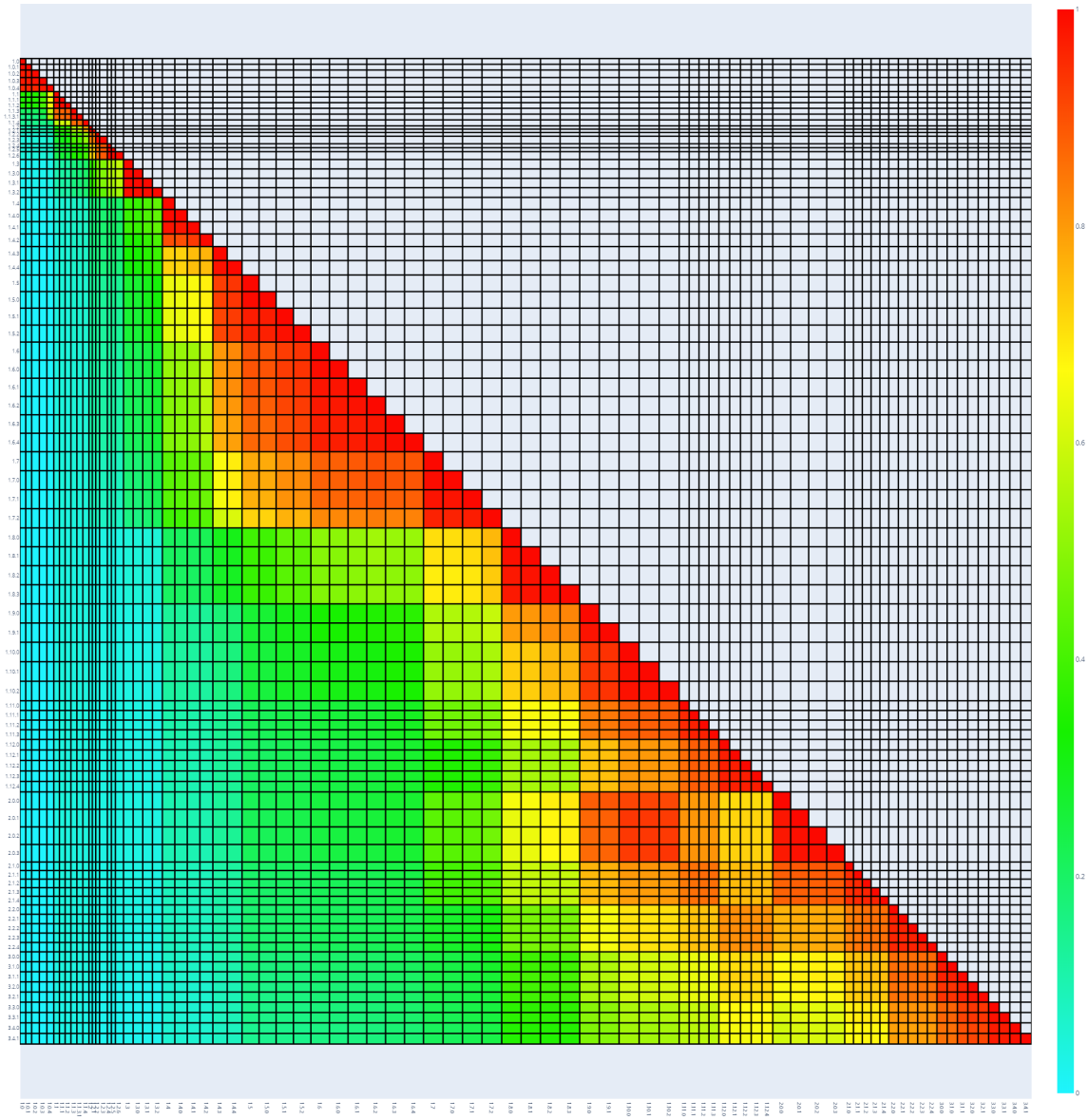

Figure 12: The heatmap of our customized method for computing similarity with parameter -t 20.

References

[1] Esben Andreasen and Anders Møller. Determinacy in static analysis for jquery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 17–31, 2014.

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, pages 292–303. IEEE, 1999.

[3] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[4] Al Danial. Cloc – count lines of code. `http://cloc.sourceforge.net/`, 2017.

[5] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 22, USA, 2007. IEEE Computer Society.

[6] Dave Methvin. jquery 1.2.6 released. `https://blog.jquery.com/2008/05/24/jquery-1-2-6-released/`, 2008.

[7] Dave Methvin. jquery 1.10.0 and 2.0.1 released. `https://blog.jquery.com/2013/05/24/jquery-1-10-0-and-2-0-1-released/`, 2013.

[8] Dave Methvin. jquery 1.11 and 2.1 released. `https://blog.jquery.com/2014/01/24/jquery-1-11-and-2-1-released/`, 2014.

[9] John Resig. jquery 1.4.3 released. `https://blog.jquery.com/2010/02/19/jquery-142-released/`, 2010.

[10] Neha Saini, Sukhdip Singh, et al. Code clones: Detection and management. *Procedia computer science*, 132:718–727, 2018.

[11] Petr Stříbný. Finding duplicate javascript code. `https://stribny.name/blog/2015/05/finding-duplicate-javascript-code`, 2015.

[12] Timmy Willison. jquery 2.2 and 1.12 released. `https://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/`, 2016.

Appendices

A The results for manually constructed clones

A.1 The result for Type 1 clones

```
PS C:\Assignment-2-2020-master\jsinspect\bin> jsinspect ./../../manually
-------------------------------------------------------------

Match - 5 instances

./../../manually\basic_snippet.js:1,12
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}

./../../manually\type1_1.js:1,15
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;



    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}

./../../manually\type1_2.js:1,12
function isPrimeNum(num){
    var flag = true;
    var i = 2; // here is a commend
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
```

```
    return flag;
}

./../../manually\type1_3.js:1,20
function isPrimeNum(num){
    var flag = true;
    var i = 2;


    /**
     * here is a multiple line comment.
     * This is a function to distinguish prime number.
     */


    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}

./../../manually\type1_4.js:1,14
function isPrimeNum(num)
{
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while( i < tmp && flag )
    {
        var isPrime = num%i==0;
        if(isPrime)
            flag = false;
        i++;
    }
    return flag;
}

1 match found across 5 files
```

A.2 The result for Type 2 clones

```
PS C:\Assignment-2-2020-master\manually> jsinspect -I -L

---------------------------------------------------------------

Match - 5 instances

.\basic_snippet.js:1,6
```

```
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;

.\type_2_1.js:1,6
function test(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;

.\type_2_2.js:1,6
function isPrimeNum(num){
    var result = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && result){
        var isPrime = num%i==0;

.\type_2_3.js:1,6
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i;

.\type_2_4.js:1,6
function isPrimeNum(num){
    var flag = true;
    var i = 1;
    var tmp=num/2+1;
    while(i<tmp && flag){
        var isPrime = num%i==0;

1 match found across 5 files
```

A.3 The result for Type 3 clones

```
PS C:\Assignment-2-2020-master> jsinspect -I -L -t 20 ./manually

---------------------------------------------------------

Match - 3 instances

./manually\basic_snippet.js:1,4
function isPrimeNum(num){
    var flag = true;
```

```
    var i = 2;
    var tmp=num/2+1;

./manually\Type_3_1.js:1,4
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;

./manually\Type_3_3.js:1,4
function isPrimeNum(num){
    var flag = true;
    var i = 2;
    var tmp=num/2+1;

------------------------------------------------------------

Match - 3 instances

./manually\basic_snippet.js:5,10
while(i<tmp && flag){
    var isPrime = num%i==0;
    if(isPrime)
        flag = false;
    i++;
}

./manually\Type_3_2.js:6,11
while(i<tmp && flag){
    var isPrime = num%i==0;
    if(isPrime)
        flag = false;
    i++;
}

./manually\Type_3_4.js:5,10
while(i<tmp && flag){
    var isPrime = num%i==0;
    if(isPrime)
        flag = false;
    i++;
}

2 matches found across 5 files
```