Li ZX, Yu Y, Wang T *et al.* Detecting duplicate contributions in pull-based model combining textual and change similarities. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(1): 191–206 Jan. 2021. DOI 10.1007/s11390-020-9935-1

Detecting Duplicate Contributions in Pull-Based Model Combining Textual and Change Similarities

Zhi-Xing Li[1], Yue Yu[1,*], *Member, CCF, ACM*, Tao Wang[1], *Member, CCF, ACM*
Gang Yin[1], *Member, CCF, ACM*, Xin-Jun Mao[2], *Member, CCF, ACM*, and
Huai-Min Wang[1], *Fellow, CCF, ACM*

[1] *Key Laboratory of Parallel and Distributed Computing, College of Computer, National University of Defense Technology Changsha 410073, China*

[2] *Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha 410073, China*

E-mail: {lizhixing15, yuyue, taowang2005, yingang, xjmao, hmwang}@nudt.edu.cn

Received August 15, 2019; accepted January 3, 2020.

Abstract Communication and coordination between open source software (OSS) developers who do not work physically in the same location have always been the challenging issues. The pull-based development model, as the state-of-the-art collaborative development mechanism, provides high openness and transparency to improve the visibility of contributors' work. However, duplicate contributions may still be submitted by more than one contributor to solve the same problem due to the parallel and uncoordinated nature of this model. If not detected in time, duplicate pull-requests can cause contributors and reviewers to waste time and energy on redundant work. In this paper, we propose an approach combining textual and change similarities to automatically detect duplicate contributions in the pull-based model at submission time. For a new-arriving contribution, we first compute textual similarity and change similarity between it and other existing contributions. And then our method returns a list of candidate duplicate contributions that are most similar to the new contribution in terms of the combined textual and change similarity. The evaluation shows that 83.4% of the duplicates can be found in average when we use the combined textual and change similarity compared with 54.8% using only textual similarity and 78.2% using only change similarity.

Keywords pull-request, duplicate detection, textual similarity, change similarity

1 Introduction

The rapid development and evolution of OSS benefits a lot from global volunteer contributions. Even though open source software (OSS) communities have fostered plenty of high-quality projects like Linux① and Rails②, the communication and coordination between contributors have always been the challenging issues[1, 2]. To make it more efficient for geographically-distributed software development, researchers and practitioners have never stopped exploring better solutions[3, 4]. Nowadays, the pull-based development model[5], as the state-of-the-art collaborative development mechanism proposed by GitHub, is becoming more attractive and being applied by an increasing number of OSS projects[6]. Supported by social coding sites and code version control systems, this model allows developers to fork a repository for local changes, and submit pull-requests (PR) for community discussion before merging back.

Although the openness and transparency of the pull-based model enables developers to collaborate in a more

---

Regular Paper

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004202, and the National Natural Science Foundation of China under Grant No. 61702534.

*Corresponding Author

①https://www.linux.org, Nov. 2019.

②https://rubyonrails.org, Nov. 2019.

visible and efficient way, developers' participation in OSS is still voluntary and spontaneous [7,8]. Therefore, it is inevitable that two developers might work on the same issue and submit duplicate PRs [9]. Especially for the popular projects which attract thousands of volunteers and continuously receive incoming PRs [10,11], it is hard to appropriately coordinate contributors' activities, because most of them work distributively and tend to lack the information of others' work progress. It cannot be denied that duplicate pull-requests might bring some benefits. For example, reviewers receive more than one solution targeted to the same issue and therefore have higher chance to pick a better solution after making a comparison between them. Besides, the authors of duplicate PRs might also learn how the same issue is solved differently and analyze the strengths and weaknesses of the two solutions respectively. Nevertheless, it is also important to realize the negative impacts of duplicate PRs.

The prior studies [5,12] have found that duplicate is one of the main reasons rejecting a PR. However, for now, there is no automatic detection tool for duplicate PRs in GitHub. The current practice is to count on the manual identification by reviewers. Unfortunately, the number of new PRs and active PRs may be too large to cope with for reviewers of popular projects. As a result, quite a number of duplicate PRs cannot be identified in time [13] and reviewers have to spent redundant effort on evaluating each of them separately [5,14]. Specifically, we have found in our prior work [13] that 21% of duplicates are detected after more than one week and 2.5 reviewers are involved in the redundant review discussion which contains 5.2 review comments on average. Moreover, a pull-request is iteratively reviewed and updated until it reaches the standard to be merged back to the codebase of the project [10,15,16]. That means both of the two developers might take redundant effort to update their PRs before the duplicate relation between their PRs is identified. Therefore, the later the duplicate relation is identified, the more redundant the effort of the contributors and reviewers may be wasted.

These problems highlight the need for an automatic tool which can be used to detect duplicate PRs at submission time. The timely identification of the duplicate relation between two PRs would help reviewers and contributors to be more informed so that they can make more appropriate decisions to avoid unnecessary redundant work. Our previous work [17] has tried to detect duplicate PRs based on textual similarity. However, it is possible that different developers use different expressions to describe the same concept, especially in OSS development which usually involves global developers with various backgrounds. To better reveal the duplicate relation between two PRs, we also leverage the change information of PRs in this paper. When a new PR is submitted to a project, we first compute the textual similarity and the change similarity between it and the historical PRs. And then we combine the two kinds of similarities by weights determined by a greedy search algorithm. Finally, we suggest a list of candidate PRs that have the highest combined similarity with the new PR. Based on the dataset constructed in our prior work [13] which contains more than 2 300 pairs of duplicate PRs, we evaluate our approach in terms of recall-rate. The experimental results show that about 83.4% of the duplicates can be found when the candidate list is set to 20.

The rest of paper is organized as follows. Section 2 illustrates the background. Section 3 presents the approach of our study in detail, and Section 4 elaborates the conducted experiments and reports the evaluation result. Threats and related work can be found in Section 5 and Section 6 respectively. Finally, we draw our conclusions in Section 7.

2 Background

In GitHub, a growing number of developers contribute to the open source projects by submitting PRs [5,6]. As illustrated in Fig.1, a typical contribution process based on the pull-based development model in GitHub involves the following actions.

- *Fork.* Before contributing, a contributor (e.g., Alice or Bob) should first fork the original project and get his or her own local repository.
- *Edit.* Based on the cloned local repository, the contributor is able to edit locally (e.g., fixing bugs or proposing new features) without disturbing the main branch in the original repository.
- *Submit.* When the contributor has finished the desired work, she/he packages the changed codes in the local repository and submits a PR to the original repository. In addition to commits, the contributor needs to provide a title and description to elaborate the submitted PR.
- *Review.* To guarantee the submitted PR does not break the current runnable state of the project, the core members of the project and community users will launch the process of code review to detect potential defects and discuss how to improve its quality. After receiving the feedback from reviewers, the contributor

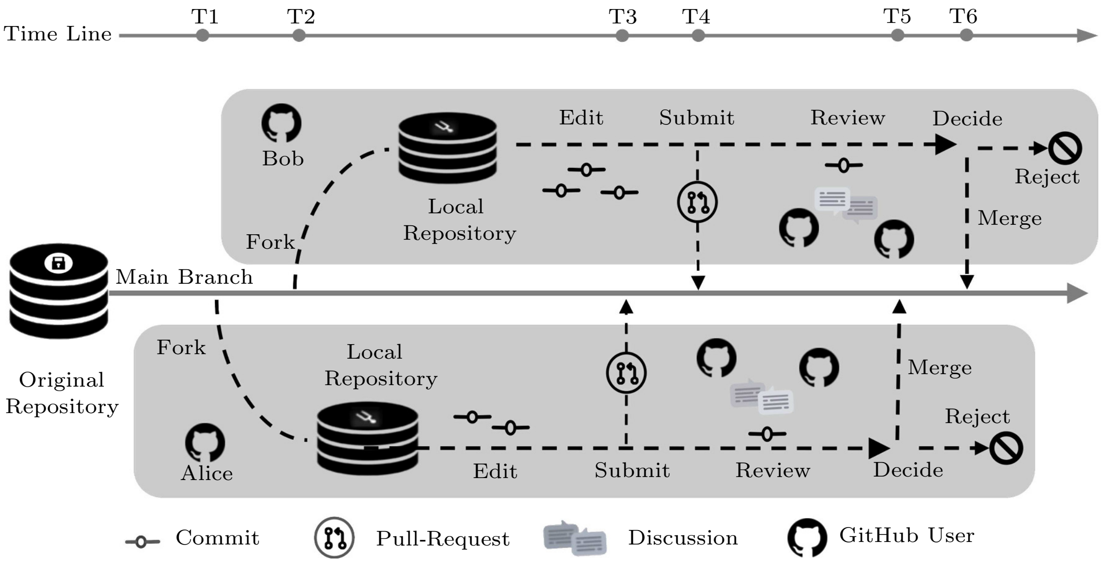


Fig.1. Contribution process in pull-based development model.

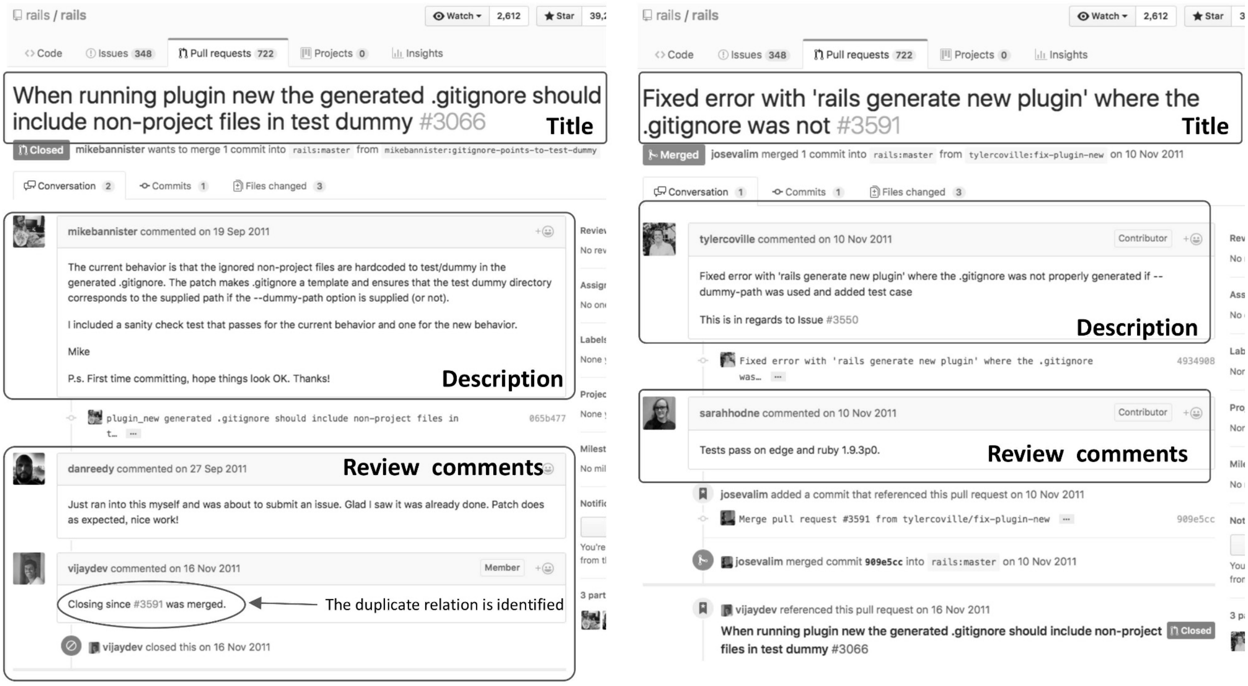
gets a chance to update the PR and attach new commits, which would trigger another round of code reviews.

- *Decide.* Finally, the PR which has went through several rounds of rigorous evaluations will be merged or rejected depending on its eventual quality by an integrator of the original repository.

The pull-based model lowers the contribution entry for community developers and improves the transparency and efficiency of collaborative development. Therefore, an increasing number of projects are adopting this development model and OSS developers have expressed high contribution enthusiasm. However, a potential risk of submitting duplicate PRs exists in the pull-based development model when more than one developer is contributing voluntarily without appropriate coordination. For example, as shown in Fig.1 two developers (i.e., Bob and Alice) fork the same original repository and edit their own local repositories to achieve the same goal. Alice first forks the original repository at time T1 after which Bob also forks the repository at T2. After forking, they conduct the offline work based on their own local repositories. Unfortunately, both of them lack awareness of each other's work and do not realize they are actually doing the same thing. Consequently, both Alice and Bob submit PRs at T3 and T4 respectively, which results in two duplicate PRs. After submission, the duplicate PRs will go through separate threads of code review until they get decisions at T5 and T6 respectively.

Fig.2 shows a pair of duplicate PRs (Rails #3066 and Rails #3591) which have been submitted to resolve the problem of .gitignore file. As shown in Fig.2, the reviewer team of a project consists of not only the core members of the project but also the community audience who are interested in the project. Moreover, there is no constrained appointment between reviewers and PRs and reviewers are free to participate in any review thread as they want. Consequently, not every PR will be reviewed by the same reviewer(s), which results in that duplicate PRs are not always possible to be detected immediately until a reviewer notices the existence of both PRs. For example, in the case shown in Fig.2, the duplicate relation is identified by the reviewer "vijaydev" after the two PRs have received several review comments. This means duplicate PRs cause redundant effort of not only the contributors' initial work before submission but also the review and update activities after submission.

In order to overcome the above challenges, an automatic tool is necessary to detect duplicate PRs at submission time. In the Bob and Alice case shown in Fig.1, such a tool can avoid redundant activities after T4, for example, Bob and Alice can be informed of each other's work and coordinated to work together for a better solution, or reviewers can prefer one PR to the other one and prevent redundant reviews and improvements on both of them. It is also possible that Bob submits his PR after Alice's PR has got a decision, i.e., T1 $<$ T2 $<$ T3 $<$ T5 $<$ T4 $<$ T6. Even in this scenario, however, detecting the duplicate relation between the two PRs at T4 makes sense to prevent reviewers wasting time on



When running plugin new the generated .gitignore should include non-project files in test dummy #3066

mikebannister wants to merge 1 commit into `rails:master` from `mikebannister:gitignore-points-to-test-dummy`

mikebannister commented on 19 Sep 2011

The current behavior is that the ignored non-project files are hardcoded to test/dummy in the generated .gitignore. The patch makes .gitignore a template and ensures that the test dummy directory corresponds to the supplied path if the --dummy-path option is supplied (or not).

I included a sanity check test that passes for the current behavior and one for the new behavior.

Mike

P.s. First time committing, hope things look OK. Thanks!

```
plugin_new generated .gitignore should include non-project files in t…    065b477
```

danreedy commented on 27 Sep 2011

Just ran into this myself and was about to submit an issue. Glad I saw it was already done. Patch does as expected, nice work!

vijaydev commented on 16 Nov 2011

Closing since #3591 was merged.

vijaydev closed this on 16 Nov 2011

Fixed error with 'rails generate new plugin' where the .gitignore was not #3591

josevalim merged 1 commit into `rails:master` from `tylercoville:fix-plugin-new` on 10 Nov 2011

tylercoville commented on 10 Nov 2011

Fixed error with 'rails generate new plugin' where the .gitignore was not properly generated if --dummy-path was used and added test case

This is in regards to Issue #3550

```
Fixed error with 'rails generate new plugin' where the .gitignore was…    4934908
```

sarahhodne commented on 10 Nov 2011

Tests pass on edge and ruby 1.9.3p0.

josevalim added a commit that referenced this pull request on 10 Nov 2011

```
Merge pull request #3591 from tylercoville/fix-plugin-new    909e5cc
```

josevalim merged commit `909e5cc` into `rails:master` on 10 Nov 2011

vijaydev referenced this pull request on 16 Nov 2011

When running plugin new the generated .gitignore should include non-project files in test dummy #3066

(a) (b)

Fig.2. A pair of duplicate PRs of rails in GitHub. (a) github.com/rails/rails/pull/3066. (b) github.com/rails/rails/pull/3591.

duplicate reviews. Finally, we would like to point out that other possible scenarios exist in that case, e.g., 1) Bob submits his PR before Alice although he forks the repository later than Alice, i.e., T4 < T3, and 2) Bob forks the repository and submits his PR after Alice submits her PR, i.e., T3 < T2. No matter which scenario happens, we are always trying to detect the relation between two actual duplicate PRs at the submission time of the PR that is submitted later.

3 Approach

The goal of our work is automatically detecting duplicate PRs at submission time. As shown in Fig.3, we first measure the similarities between a new PR and the historical PRs.

Actually, we make two intuitive hypotheses: 1) if two pull-requests are duplicate, they tend to have similar textual descriptions, and 2) if two pull-requests are duplicate, they tend to have high overlap in changes. Therefore, two kinds of similarities are considered in our approach: textual similarity and change similarity. Textual similarity is calculated based on the natural language text (i.e., titles and descriptions of PRs), while change similarity is calculated by comparing the overlap of changed source files. And then, we combine different similarities with a greedy search algorithm. Finally, we suggest a list of candidate PRs ranked by the combined similarity.

In the following subsections, we will elaborate each step of the approach in detail.

3.1 Calculating Textual Similarity

From the example in Fig.2, we can see that the titles and descriptions of the two duplicate PRs share some same words, which means natural language processing (NLP) technologies can be used to measure their textual similarity [18–21]. The text content of a PR contains two components: the title and the description. Therefore, we calculate both title similarity and description similarity between two PRs. In the calculation of each of them, we adopt the standard NLP techniques [18] as follows.

Preprocessing. Firstly, preprocessing is performed on the texts including tokenization, stemming and stop words removal. Different strategies can be applied to split a sentence into tokens depending on the type of data and application domain [18]. There are some types of texts which are usually split into multiple tokens in common settings but we treat them as a single token in the context of PR. For example, code paths and hyperlinks usually indicate one concept and they should not be divided into separate words. To this end, we use

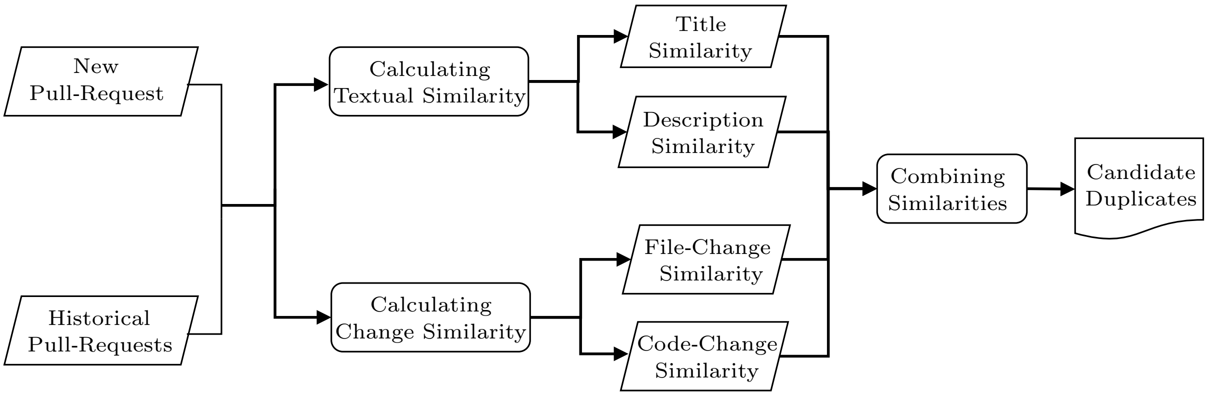


Fig.3. Overall framework of our method.

the regular tokenizer to parse the raw text. After tokenization, each word will be stemmed to its root form (e.g., "was" to "be" and "errors" to "error") with the help of Porter stemming algorithm[22]. Finally, common stop words (e.g., "the" and "a"), which appear so frequently that they have little effect on distinguishing different documents, will be removed.

Transformation. We then transform the preprocessed texts into multi-dimensional vectors which is computable in Vector Space Model (VSM). After transformation, a text is represented as a vector, for example the presentation of the i-th text is: $TextVec_i = (\boldsymbol{w}_{i,1}, \boldsymbol{w}_{i,2}, ..., \boldsymbol{w}_{i,v})$. Each dimension of the vector corresponds to an unique word in the corpus formed from all the texts. The value of $w_{i,k}$, which is the weight of the k-th item of $TextVec_i$, is computed by the TF-IDF (Term Frequency-Inverse Document Frequency) model:

$$w_{i,k} = tf_{i,k} \times idf_{i,k}. \tag{1}$$

In (1), $tf_{i,k}$ denotes the term frequency which is the frequency of the k-th term appearing in the i-th text and $idf_{i,k}$ denotes the inverse term frequency which measures the distinguishing characteristics of a term based on the number of texts containing it.

Calculation. With the texts represented as vectors, we calculate the similarity between two texts (e.g., the i-th text and the j-th text) using Cosine[23] similarity which is computed by the following formula:

$$\begin{aligned} TextSim(i,j) &= \frac{TextVec_i \times TextVec_j}{|TextVec_i||TextVec_j|} \\ &= \frac{\sum_{m=1}^{m=v}(w_{i,m} \times w_{j,m})}{\sqrt{\sum_{m=1}^{m=v} w_{i,m}^2}\sqrt{\sum_{m=1}^{m=v} w_{j,m}^2}}. \end{aligned} \tag{2}$$

For two PRs, after applying this formula to the texts of titles and texts of descriptions respectively, we can obtain two similarities Sim_title (title similarity) and Sim_desc (description similarity).

3.2 Calculating Change Similarity

It is possible that different sentences and expression formations can be used by different people when they try to describe the same thing, especially in the scenario of collaborative software development which involves developers with various backgrounds from all over world. Consequently, using only natural language text may not be enough to detect whether two PRs are duplicate. In such cases, the information of changes may be more helpful. It is intuitive that in order to carry out the same task, either fixing a bug or adding a feature, developers are likely to edit the similar or even the same files. Therefore, in addition to textual similarity, we also take into consideration the change similarity which includes file-change similarity and code-change similarity.

3.2.1 File-Change Similarity

To calculate the file-change similarity between two PRs: PR_i and PR_j, we first parse the raw change information and extract the changed files. And then Algorithm 1 is applied on the extracted files, which takes the sets of changed files as the input and outputs the file-change similarity between two PRs. The first line initializes a list $file\_sims$ to store the intermediate results generated in the process. The subsequent six lines (lines 2–7) calculate the pair-wise path similarities between two file sets and store each pair of files and their similarity to $file\_sims$. Specifically, the path similarity between two files is computed by Algorithm 2 which will be introduced in the following paragraph. And then we sort the items in $file\_sims$ according to the path similarity (line 8). Subsequently, a new empty list is

Algorithm 1. Calculating the File-Change Similarity Between Two PRs

```
Input:
    $files_i$: the files changed by $PR_i$,
    $files_j$: the files changed by $PR_j$
Output:
    the file-change similarity between $PR_i$ and $PR_j$
 1: Let $file\_sims$ be a list
 2: for $f_i$ in $files_i$ do
 3:     for $f_j$ in $files_j$ do
 4:         $fp\_sim \Leftarrow$ file path similarity between $f_i$ and $f_j$
 5:         Add tuple ($f_i$, $f_j$, $fp\_sim$) to $file\_sims$
 6:     end for
 7: end for
 8: Sort $file\_sims$ in terms of file path similarity
 9: Let $final\_sims$ be a list
10: $top\_num \Leftarrow \min(\mathrm{len}\,(files_i), \mathrm{len}\,(files_j))$
11: while $top\_num > 0$ do
12:     Let $top\_sim$ be the item in $files\_sims$ which gets the highest similarity
13:     Add $top\_sim[2]$ to $final\_sim$
14:     Delete $item$ from $files\_sims$ on condition that $item[0] == top\_sim[0]$ or $item[1] == top\_sim[1]$
15:     $top\_num - -$
16: end while
17: return $\mathrm{sum}(final\_sims)/\max(\mathrm{len}\,(files_i), \mathrm{len}\,(files_j))$
```

created (line 9) and the minimum size of the two file sets is used to determine how many items in the sorted *file_sims* would be used to calculate the final change similarity (line 10). In the following lines (lines 11–16), we iteratively review *file_sims* and select the item which has the highest similarity and store the value of similarity to *final_sims*. Let us assume the selected item is (f_m, f_n, sim_mn). In order to make the most of each changed file of the two PRs, the other items in *file_sims* which contain f_m or f_n are deleted from *file_sims* and would not be considered in the next iteration, so that the left items composed by other files will have more chance to be reviewed. Finally, all the similarity values in *file_sims* would be added together, divided by the maximum size of the two file sets, and returned as the file-change similarity (line 17).

Algorithm 2 . Calculating Path Similarity Between Two Files

```
Input:
    file $file_i$ and file $file_j$
Output:
    the path similarity between file $file_i$ and file $file_j$
 1: $fp_i$, $fp_j \Leftarrow file_i.path$, $file_j.path$
 2: $ps_i \Leftarrow$ split $file_i$ into components
 3: $ps_j \Leftarrow$ split $file_j$ into components
 4: $pos \Leftarrow 0$
 5: while $pos <$ len $(ps_i)$ and $pos <$ len $(ps_j)$ do
 6:     if $ps_i[pos] != ps_j[pos]$ then
 7:         break
 8:     end if
 9:     pos++
10: end while
11: return $pos/\max(\mathrm{len}\,(ps_i), \mathrm{len}\,(ps_j))$
```

Algorithm 2 is used to compute the path similarity between two files. It first gets the paths of the inputted two files (line 1). And then, it will split the two paths by the path separator (i.e., "/") (lines 2 and 3). Subsequently, the longest common sub-path will be found for the two paths (lines 4–10). Finally, the length of the longest common sub-path will be divided by the maximum length of the two component sets and returned as the path similarity (line 11).

3.2.2 Code-Change Similarity

To calculate the code-change similarity between two PRs: PR_i and PR_j, we first extract the added lines and the deleted lines in both of them and then apply the following formula.

$$CodeSim(i,j) = \frac{AddSim(i,j) + DelSim(i,j)}{\max(N(lines_i), N(lines_j))}. \quad (3)$$

In (3), function *AddSim* returns the similarity between PR_i and PR_j in terms of the added lines, and function *DelSim* returns their similarity in terms of the deleted lines. Variables $lines_i$ and $lines_j$ represent the sets of changed lines (i.e., added lines plus deleted lines) by PR_i and PR_j respectively. Function N returns the number of items contained in the given set.

The computation detail of *DelSim* is as follows:

$$DelSim(i,j) = \frac{N(del\_lines_i \ \& \ del\_lines_j)}{N(del\_lines_i \mid del\_lines_j)}. \quad (4)$$

del_lines_i and del_lines_j are the sets of lines deleted by PR_i and PR_j respectively. We divide the size of the union of the two sets by the size of their intersection and get the similarity value in terms of deleted lines.

To compute the similarity in terms of added lines, we use the token-based method as shown in Algorithm 3. The basic idea of this algorithm is similar to that of Algorithm 1. For two given PRs, PR_i and PR_j, we first get the common files $files\_common$ changed by both of them (line 1). And then we iteratively compute the code-change similarity on each of the common files (lines 3–23). Specifically, we extract all the added lines on a common file by the two PRs respectively (line 4), calculate the pair-wise similarities between two lines, and store the results in a list $line\_sims$ (lines 6–12). To compute the similarity between two lines, function *tokenize* first replaces each punctuation in the given line with whitespaces and splits the line into word tokens (line 8). Next, the similarity based on the two token sets is computed using the size of their union to divide the size of the intersection (line 9). The subsequent steps on $line\_sims$ (lines 13–21) are similar to the steps (lines 8–16) introduced in Algorithm 1. After the code-change similarity on each file has been calculated and collected (line 22), we return the sum of these similarity values (line 24).

3.3 Combining Similarities

Since title similarity (Sim_title), description similarity (Sim_desc), file-change similarity (Sim_file) and code-change similarity (Sim_code) between two PRs have been calculated, we are able to compute the combined similarity [19] as follows:

$$\begin{aligned} & Sim\_combined \\ = \; & a \times Sim\_title + b \times Sim\_desc + \\ & c \times Sim\_file + d \times Sim\_code. \end{aligned} \tag{5}$$

In (5), $Sim\_combined$ denotes the combined similarity that is composed by the four kinds of similarities with different weights (i.e., a, b, c, and d). To automatically determine the value of the four weight parameters, we use a greedy search algorithm as shown in Algorithm 4. Its inputs include the training set randomly sampled from DupPR, a dataset of historical duplicate PRs which will be introduced in Subsection 4.1, the maximum number of iterations for searching the best weight parameters, and the value of unit by which weights increase or decrease in each iteration. Finally, a list of optimized weight parameters will be returned.

In Algorithm 4, the first three lines (lines 1–3) initialize the four weight parameters, compose a list with them and get the initial fitness score for the initial weight list. The fitness score is used to evaluate the

Algorithm 3. Calculating the Code-Change Similarity Between Two PRs in Terms of Added Lines

```
Input:
    $files_i$: the files changed by $PR_i$,
    $files_j$: the files changed by $PR_j$
Output:
    the code-change similarity between $PR_i$ and $PR_j$ in terms of added lines
1: $files\_common \Leftarrow$ the intersection of $files_i$ and $files_j$
2: Let $add\_sims$ be a list
3: for $f$ in $files\_common$ do
4:     $add\_lines_{i,f}$, $add\_lines_{j,f} \Leftarrow$ addedLines($PR_i$, $f$), addedLines($PR_j$, $f$)
5:     Let $line\_sims$ be a list
6:     for $line_i$ in $add\_lines_{i,f}$ do
7:         for $line_j$ in $add\_lines_{j,f}$ do
8:             $tokens_i$, $tokens_j \Leftarrow$ tokenize($line_i$), tokenize($line_j$)
9:             $tmp\_sim \Leftarrow$ len(intersection($tokens_i$, $tokens_j$))/len(union($tokens_i$, $tokens_j$))
10:            Add ($tokens_i$, $tokens_j$, $tmp\_sim$) to $line\_sims$
11:        end for
12:    end for
13:    Sort $line\_sims$ by token similarity
14:    $top\_n \Leftarrow$ min($add\_lines_{i,f}$, $add\_lines_{j,f}$)
15:    Let $final\_sims$ be a list
16:    while $top\_n > 0$ do
17:        $top\_sim \Leftarrow line\_sims[0]$
18:        Add $top\_sim[2]$ to $final\_sims$
19:        Delete $item$ from $line\_sims$ on condition that $item[0] == top\_sim[0]$ or $item[1] == top\_sim[1]$
20:        $top\_num$ –
21:    end while
22:    Extend $add\_sims$ with $final\_sims$
23: end for
24: return sum($add\_sims$)
```

Algorithm 4. Determining Weight Parameters

```
Input:
    $DupPR\_train$: the training dataset of duplicate PRs,
    $max\_iter$: maximum number of iterations (default value = 20),
    $step$: the unit of weight change in each iteration (default value = 0.05)
Output:
    a list of weight parameters ($a$,$b$ and $c$)
1: let $a = 1, b = 1$ and $c = 1$
2: $wts = [a, b, c]$
3: $wts.fts = fitness(DupPR\_train, wts)$
4: repeat
5:     let $search\_history$ be a list
6:     for $i$ in $[0, \mathrm{len}(wts)]$ do
7:         $tmp\_wts \Leftarrow wts$    #forward search
8:         Increase $tmp\_wts[i]$ by $step$
9:         $tmp\_wts.fts \Leftarrow fitness(DupPR\_train, tmp\_wts)$
10:        Add $tmp\_wts$ to $serach\_history$
11:        $tmp\_wts \Leftarrow wts$    #backward search
12:        Decrease $tmp\_wts[i]$ by $step$ if $tmp\_wts[i] > 0$
13:        $tmp\_wts.fts \Leftarrow fitness(DupPR\_train, tmp\_wts)$
14:        Add $tmp\_wts$ to $serach\_history$
15:    end for
16:    Set $wts\_max$ to the $tmp\_wts$ which gets the max $fts$ in $search\_history$
17:    if $wts\_max.fts > wts.fts$ then
18:        $wts.fts \Leftarrow wts\_max.fts$
19:        $wts \Leftarrow wts\_max$
20:    else
21:        break
22:    end if
23:    $max\_iter--$
24: until $max\_iter > 0$
25: return $wts$
```

detection performance of duplicate PRs when a set of weight parameters are used to combine each kind of similarities. For a given PR, we expect its duplicate PR can get a higher similarity than others. To this end, the fitness function is set as the following.

$$fit(DupPR, wts) = \sum_{(pr_i,pr_j)\sim DupPR\_train} \frac{1}{rank(pr_i, SimPRs(pr_j))}. \quad (6)$$

In (6), (pr_i, pr_j) indicates each pair of duplicate PRs in $DupPR\_train$, and pr_i is submitted earlier than pr_j. Function $SimPRs$ returns a top list of PRs that are the most similar to pr_j in terms of the combined similarity, and function $rank$ computes the position of pr_i in the top list.

The subsequent lines (lines 4–24) iteratively search better weight parameters until a local optimal result is found or the limitation of iterations is reached. In line 5, we first create a list to store the search histories in each iteration which will be compared to determine the local optimal solution. In each iteration, we try to change every weight parameter (lines 6–15) from two directions: forward search (lines 7–10) and backward search (lines 11–14). A copy of the current optimal weight list is first created in both kinds of search attempt (line 7 and line 11). In forward search, the corresponding item in weight list is increased by one unit of weight change (line 8), while in backward search the item is decreased by one unit (line 12).

The changed weight list is then used to calculate the new fitness score (line 9 and line 13) and the new weight lists will be stored in *search_history* (line 10 and line 14). After all the items in the weight list have been inspected forward and backward, *wts_max*, which gets the maximum the fitness score, will be selected from *search_history* (line 16). If the fitness score of *wts_max* is higher than the fitness score of the current optimal weight list, it will update the current optimal weight list and the next iteration begins (lines 17–19) until it reaches the limitation of the maximum iterations (lines 23 and 24). Otherwise, the procedure has reached a local optimum and we terminate the iteration process (lines 20 and 21). In the end, the final optimal weight list is returned (line 25).

3.4 Suggesting Candidates

After the combined similarities between the new PR and the historical PRs have been computed, we rank the historical PRs according to the combined similarity. Among the ranked PRs, we suggest the top-k items as candidate duplicates so that reviewers can examine whether the new PR duplicates a suggested existing PR.

4 Experiment and Evaluation

4.1 Dataset

The experiments are conducted on the dataset DupPR[13] which is collected from 26 open source projects in GitHub③. Each pair of duplicate PRs in DupPR has been manually verified after an automatic identification process, which would guarantee the quality of this dataset. The construction process of DupPR is shown in Fig.4.

- *Random Sampling.* For each project, 200 review comments are randomly sampled, containing at least one reference to another PR.
- *Manual Examination.* Each sampled comment is manually examined to see if it is used to point out the duplicate relation among PRs. Such kind of comments are called indicative comments which can help to reconstruct the duplicate relations.
- *Rules Extraction.* All the manually-identified indicative comments are reviewed to extract rules (regular expressions) which can be applied lately to automatically judge whether a given comment is an indicative comment. The following items are some simplified rules.
  - `closed by (?:\w+:?  ){,5} (?:#(\d+))`
  - `(?:#(\d+)):?  (?:\w+:?  ){,5} dup(?:licate)?`

The first rule would match comments which contain the keywords `closed by` followed by several words and a pull-request reference like "Closed by lucky number #2000 because it's a cleaner PR". The second rule would match comments that contain a pull-request reference followed by several words and the keywords `dup` or `duplicate` like "PR #16509 is duplicate of this PR".

- *Automatic Identification.* If a review comment is automatically identified as an indicative comment according to the identification rules, the PR references contained in the comment will be extracted to form a couple of candidate duplicates with the PR that the indicative comment belongs to. In total, 3 580 pairs of candidate duplicate pull-requests are detected.
- *Manual Verification.* It is inevitable that automatic identification may introduce false-positive errors. To exclude the misidentified duplicates, all the candidate duplicate PRs are manually verified. Finally, 2 323 pairs of duplicate pull-requests pass the manual verification.

For each project in DupPR, we randomly select half of the duplicates as the training set and the remaining duplicates are used as the test set. In the paper, for each pair of duplicate PRs in DupPR, the early submitted one is called master PR and the late submitted one is called duplicate PR. Our research goal is trying to detect the corresponding master PR given a duplicate PR.

4.2 Evaluation Metrics

To evaluate the performance of our method, we apply the *recall-rate@k* metric proposed by Runeson *et al.*[18] which has been widely applied by other studies[24, 25] related to duplicate detection. (7) defines how *recall-rate@k* is calculated.

$$recall\text{-}rate@k = \frac{n\_detected}{n\_tatol}. \tag{7}$$

In (7), $n\_detected$ is the number of duplicate PRs whose corresponding master PRs are detected in the suggested candidate list, while n_total is the total number of duplicate PRs in the test set. In terms of recall-rate, detection approaches can be assessed by calculating the percentage of duplicate PRs for which the master PRs are in the suggested candidate list. Moreover, k in *recall-rate@k* varies from 1 to 20 respectively in the experiments.

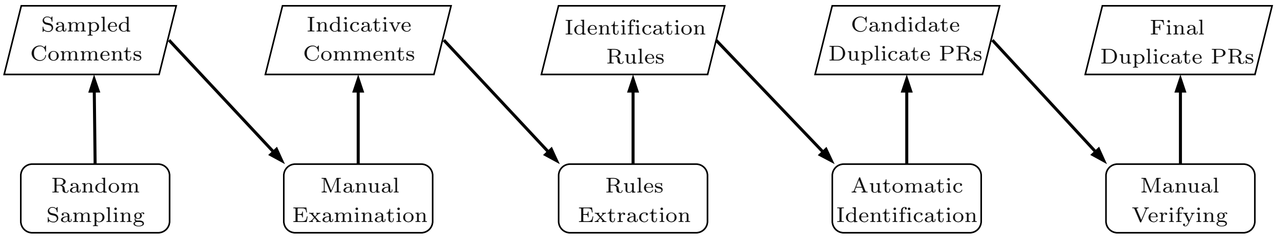


Fig.4. Approach to getting historical duplicate PRs.

---

③https://github.com/whystar/MSR2018-DupPR/blob/master/project_list.md, Nov. 2019.

4.3 Research Questions and Results

In this subsection, we present the experiments with respect to our three research questions.

*RQ*1. Is textual similarity or change similarity more helpful to detect duplicate PRs?

Experimental Setup. As previously discussed, a PR usually contains two kinds of information, i.e., textual information and change information. Hence, we firstly want to investigate how the detection performance defers when different information is separately used. To answer this question, we conduct experiments with five options by using title similarity, description similarity, text similarity, file-change similarity, and code-change similarity respectively. Text similarity is calculated by adding together title similarity and description similarity. To offer an overall evaluation on the detection performance of different similarities, we compute the weighted average *recall-rate*@k on all the 26 studied projects. Obviously, the larger the variable k, the higher the weighted average *recall-rate*@k would be. However, a larger k would also cause the top list to contain more irrelevant items that would make the automatic detection less applicable in practice. Consequently, we follow the prior studies [20, 24–26] and make k range from 1 to 20 in the experiment.

Evaluation Result. Fig.5 shows the evaluation result of different similarities. We use *Sim_title*, *Sim_desc*, *Sim_text*, *Sim_file*, and *Sim_code* as abbreviations for the five experimental options, title similarity, description similarity, text similarity, file-change similarity, and code-change similarity respectively. From the result, we can see that change similarities perform better than textual similarities and code-change similarity is the best no matter how the size of candidate list changes. For example, when the size of top list is set to 20, *Sim_code* is able to find about 78.2% duplicates for each project in average, while *Sim_file*, *Sim_title*, *Sim_desc*, and *Sim_text*, can only find 61.0%, 45.2%, 40.1% and 54.8% respectively.

Summary. Change similarity performs better than textual similarity in detecting duplicate PRs.

*RQ*2. Can combining textual similarity and change similarity achieve better detection performance?

Experimental Setup. Furthermore, we would like to examine whether combining textual similarity and change similarity can improve the detection performance compared with using each of them separately. To answer our second research question, we conduct another experiment using combined textual and change similarity (*Sim_combined*) to detect duplicate PRs. As shown in (5), all the four kinds of basic similarities (i.e., title similarity, description similarity, file-change similarity and code-change similarity) are added by a linear model with weights determined by a greedy search algorithm. As we do in the experiment of RQ1, we still use weighted average *recall-rate* to evaluate the detection performance.

Evaluation Result. The detection performance of *Sim_combined* is shown in Fig.6. In addition, to provide a direct and intuitive comparison among the combined similarity and separate similarities introduced in RQ1, we also present *Sim_code* (code-change similarity) in the figure, which achieves the best performance among them. For each k (varying from 1 to 20) of *recall-rate*@k, we use a box plot to present the detection results on all the projects. The marker in each box represents the weighted average recall-rate for the corresponding k and all the markers are connected as a line which outlines the overall performance as in Fig.5. In Fig.6 we can see that *Sim_combined* achieves better performance than *Sim_code*, which means *Sim_combined* is also better than *Sim_file*,

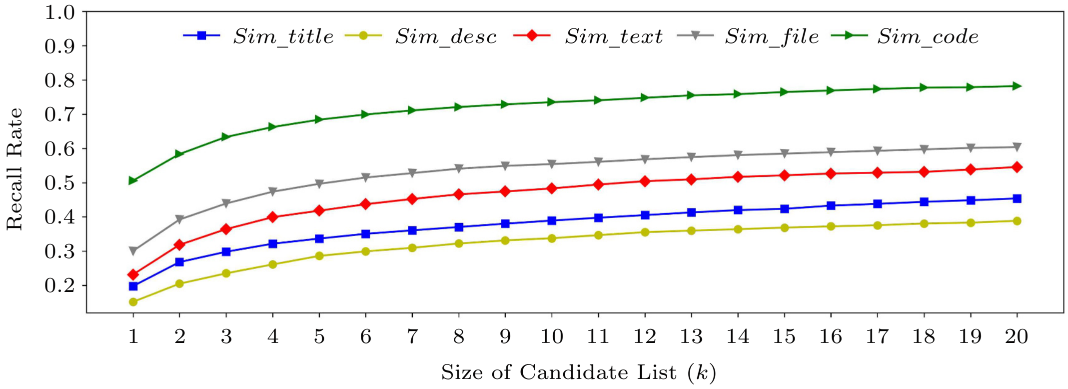


Fig.5. Detection performance of each kind of similarities.

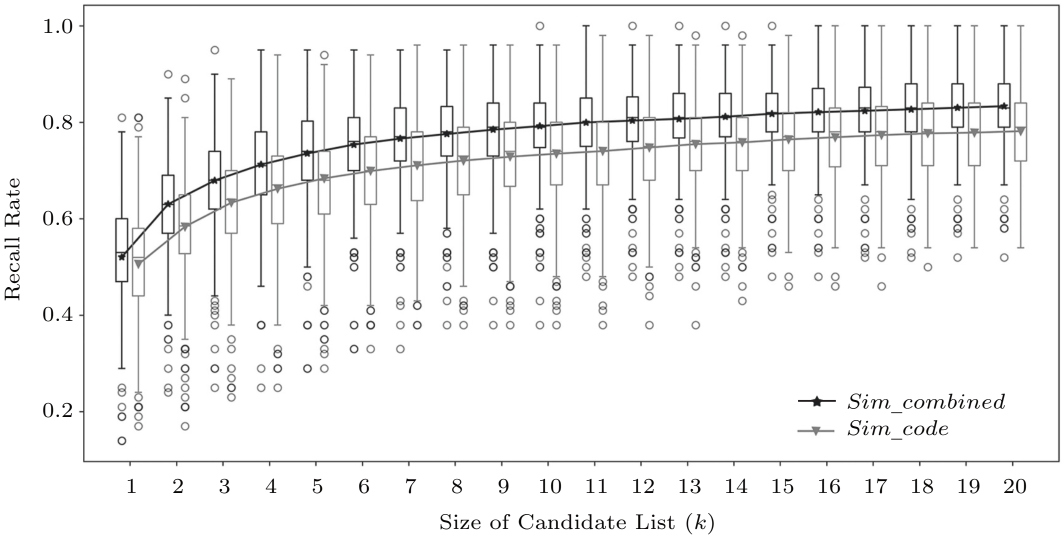


Fig.6. Detection performance of *Sim_combined* and *Sim_code*.

Sim_title, *Sim_desc*, and *Sim_text*. For example, *Sim_combined* can find 83.4% duplicates when k is set to 20, which exceeds the results in RQ1.

Moreover, we conduct Mann-Whitney-Wilcoxon (MWW) test [27] to explore whether the performance improvement is significant. Specifically, we divide each comparison result into four intervals to see how the significance changes when the size of candidate list varies. Table 1 shows the test results and we can see from the table that all the $p.values$ are less than 0.05, which means compared with all the textual similarities and change similarities, the improvement of the combined similarity is significant.

Summary. The combined similarity outperforms either textual similarity or change similarity and achieves significant improvement in detection performance.

*RQ*3. Does the greedy search algorithm achieve reasonable weight parameters?

Experimental Setup. As previously discussed in Subsection 3.3, the combined similarity is derived from four different similarities with weight parameters (i.e., a, b, c, and d) that are determined by the greedy search method. Weight parameters have significant impact on the final detection performance; therefore we would like to explore the actual effect of this algorithm. To the end, we randomly generate 20 sets of weight parameters and test their effect. In addition, we also want to examine what happens if each kind of similarities is treated equally, that is, each kind of similarities gets the same weight. Finally, the performance of these 21 sets of weight parameters is explored together with that determined by the greedy search method.

Evaluation Results. Table 2 shows the experimental result where the 22 different sets of weight parameters are organized as three groups. *WT_GS* indicates the weight parameters determined by the greedy search methods, *WT_EQ* indicates the equal weight parameters, and *WT_RD* indicates the randomly generated weight parameters. Since *WT_GS* is determined for each specific project, we do not show the exact weight values for each project, and instead we use hyphens as the placeholders in the table. From the table we can see that *WT_GS* achieves better performance than *WT_RD* and *WT_EQ*. Moreover, the detection per-

Table 1. Results of Mann-Whitney-Wilcoxon Test

Group	p			
	$0 < k \leqslant 5$	$5 < k \leqslant 10$	$10 < k \leqslant 15$	$15 < k \leqslant 20$
Sim_combined vs *Sim_title*	1.227512e-213	1.951276e-214	6.003839e-214	2.926856e-214
Sim_combined vs *Sim_desc*	8.972839e-214	1.921972e-214	1.91315e-214	1.895662e-214
Sim_combined vs *Sim_text*	4.650869e-212	1.387071e-212	1.274065e-213	8.634096e-214
Sim_combined vs *Sim_file*	2.263567e-191	2.182324e-195	3.805687e-200	2.368933e-200
Sim_combined vs *Sim_code*	1.482833e-33	2.722179e-69	7.054996e-79	4.830722e-69

Table 2. Comparison of Detect Performance for Different Weights

Group	a	b	c	d	RR@1	RR@5	RR@10	RR@15	RR@20
WT_GS	–	–	–	–	0.520	0.735	0.792	0.818	0.834
WT_EQ	1.00	1.00	1.00	1.00	0.488	0.710	0.769	0.805	0.826
WT_RD	0.89	0.24	0.21	0.62	0.415	0.668	0.737	0.774	0.796
	0.80	0.77	0.41	0.47	0.418	0.641	0.724	0.763	0.801
	0.90	0.00	0.60	0.06	0.243	0.437	0.531	0.591	0.623
	0.47	0.16	0.94	0.41	0.451	0.650	0.715	0.747	0.764
	0.75	0.61	0.28	0.19	0.329	0.533	0.610	0.661	0.700
	0.77	0.36	0.18	0.64	0.471	0.696	0.757	0.793	0.804
	0.34	0.65	0.38	0.57	0.499	0.716	0.781	0.807	0.823
	0.24	0.22	0.86	0.39	0.497	0.660	0.722	0.750	0.771
	0.12	1.00	0.05	0.11	0.249	0.404	0.461	0.499	0.526
	0.35	0.54	0.39	0.52	0.501	0.726	0.785	0.811	0.830
	0.64	0.57	0.52	0.17	0.351	0.572	0.658	0.713	0.741
	0.83	0.97	0.22	0.58	0.405	0.637	0.711	0.757	0.786
	0.89	0.20	0.05	0.30	0.295	0.491	0.580	0.621	0.649
	0.65	0.35	0.47	0.66	0.507	0.713	0.780	0.806	0.828
	0.12	0.58	0.23	0.05	0.268	0.464	0.554	0.598	0.627
	0.21	0.43	0.24	0.47	0.510	0.729	0.786	0.807	0.827
	0.07	0.58	0.77	0.73	0.509	0.687	0.758	0.795	0.814
	0.46	0.67	0.75	0.51	0.476	0.695	0.753	0.781	0.803
	0.50	0.12	0.16	0.05	0.260	0.453	0.525	0.577	0.609
	0.73	0.44	0.36	0.88	0.546	0.720	0.783	0.815	0.830

Note: "RR" is the abbreviation of "recall-rate"

formance of *WT_RD* is not stable; it can achieve good detection result of 83.0% for *recall-rate*@20, while it can also result in a bad result which is only 52.6%.

Summary. The greedy search algorithm can achieve reasonable weight parameters to combine each kind of similarities.

5 Threats to Validity

In this section, we discuss some threats to validity which may affect the experimental results of our study.

External Validity. Our experiments are conducted based on some of the popular open source projects hosted in GitHub. The projects are developed by various programming languages and applied in different domains. However, it is unknown whether our method can be generalized to all the projects in GitHub and open source projects hosted in other platforms.

Internal Validity. Firstly, the dataset of historical duplicate PRs may contain false negative, since the extraction rules may not match all the indicative comments. Moreover, some reviewers may just close the duplicate PRs and do not leave any comment. In the future, we plan to collect more projects and enrich the dataset to further validate the effectiveness of our method.

Secondly, in order to determine the weight parameters for the four kinds of similarities, we use a greedy search algorithm. In our experiment, this algorithm performs better than treating each kind of similarities equally or randomly assigning weights to them, but we cannot ensure that the algorithm has certainly produced the most optimal result.

6 Related Work

6.1 Duplicate Detection

Researchers have paid plenty of attention on recognizing duplicate bug reports. Runeson *et al.*[18] evaluated how NLP techniques support duplicate reports identification and found about 40% duplicates can be detected. Wang and Zhang[19] proposed an approach to detect duplicate bug reports by comparing the natural language information and execution information between the new report and the existing reports. Sun *et al.*[24] used discriminative models to detect duplicates and their evaluation on three large software bug repositories showed that their method achieved improvements compared with methods using natural language. Later, Sun *et al.*[25] proposed a retrieval function to fully utilize the information available in a bug report and measure the similarity between two bug reports. Nguyen *et al.*[20] modeled each bug report as a textual document and took advantage of both IR-based features and topic-based features to learn the sets of different terms used to describe the same problems. Thung *et al.*[28] developed a tool implementing the approach proposed by Runeson *et al.*[18] and integrated it into the existing bug tracking systems. Lazar *et al.*[21] made use of a set of new textual features and trained several binary classification models to improve the detection performance. Moreover, Zhang *et al.*[26] investigated to detect duplicate questions in Stack Overflow. They measured the similarity of two questions by comparing observable factors including titles, descriptions, and tags of the questions and latent factors corresponding to the topic distributions learned from the descriptions of the questions.

6.2 Pull-Request

Although the research on PRs is in its early stages, several studies have been conducted to analyze how PRs are applied and evaluated. Gousios *et al.*[5] conducted a statistical analysis of millions of PRs from GitHub and analyzed the popularity of PRs, the factors affecting the decision to merge or reject a PR, and the time to merge a PR. Furthermore, Gousios *et al.*[9,14] studied on the work habits and challenges in pull-based development model from integrators' and contributors' perspectives respectively. Tsay *et al.*[29] examined how social and technical information are used to evaluate PRs. Yu *et al.*[6] conducted a quantitative study on the PR evaluation in the context of CI. Moreover, Yu *et al.*[10] proposed an approach that combines information retrieval and social network analysis to recommend potential reviewers. van der Veen *et al.*[30] presented PRioritizer, a prototype PR prioritization tool, to recommend the top PRs the project owner should focus on.

6.3 Code Review

Code review is employed by many software projects to examine the change made by others in source codes, find potential defects, and ensure software quality before they are merged[31,32]. Traditional code review proposed by Fagan[33] has been performed since the 1970s, but it did not get universally applied for its cumbersome and synchronous characteristics[34]. In recent years, Modern Code Review (MCR)[35] is adopted by an increasing number of software companies and teams. Different from formal code inspections, MCR is a lightweight mechanism[11,36] that is less time consuming and supported by various tools. Several perspectives of code review have been widely studied, such as automation of review task[11,37–39], factors influencing review outcomes[29,31,40] and challenges involved in code review[34,41]. The impact of code review on software quality[32,42] is also investigated by many studies in terms of code review coverage and code review participation[43], and code ownership[44]. While the main motivation for code review was believed to be finding defects to control software quality, recent research has revealed that defect elimination is not the sole motivation. Bacchelli and Bird[34] reported additional expectations, including knowledge transfer, increased team awareness, and the creation of alternative solutions to problems.

7 Conclusions

In this paper, we proposed an approach to automatically detect duplicate PRs in GitHub. Our method employs textual information and change information to calculate the similarity between two PRs and returns a candidate list of historical PRs that are most similar to the new-arriving PR. We evaluated our approach on a dataset of historical duplicates collected based on 26 popular projects hosted in GitHub. The evaluation results showed that using the combined textual and change similarity can achieve the best performance which finds about 83.4% of the duplicates compared with 54.8% using only textual similarity and 78.2% using only change information.

In the future, we plan to explore more features that can be employed to detect duplicate PRs. In addition, we would like to investigate what kind of contribution patterns tend to result in duplicate PRs and we can propose some strategies to prevent developers submitting duplicate contributions.

References

[1] Herbsleb J D, Mockus A. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 2003, 29(6): 481-494. DOI: 10.1109/TSE.2003.1205177.

[2] Espinosa J, Slaughter S, Kraut R, Herbsleb J. Team knowledge and coordination in geographically distributed software development. *Journal of Management Information*

Systems, 2007, 24(1): 135-169. DOI: 10.2753/MIS0742-1222240104.

[3] Storey M A, Singer L, Cleary B, Filho F M, Zagalsky A. The (r)evolution of social media in software engineering. In *Proc. the 2014 International Conference on Future of Software Engineering*, May 31–June 7, 2014, pp.100-116. DOI: 10.1145/2593882.2593887.

[4] Zhu J, Zhou M, Mockus A. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp.871-882. DOI: 10.1145/2950290.2950364.

[5] Gousios G, Pinzger M, van Deursen A. An exploratory study of the pull-based software development model. In *Proc. the 36th International Conference on Software Engineering*, May 2014, pp.345-355. DOI: 10.1145/2568225.2568260.

[6] Yu Y, Yin G, Wang T, Yang C, Wang H. Determinants of pull-based development in the context of continuous integration. *SCIENCE CHINA: Information Sciences*, 2016, 59(8): Article No. 080104. DOI: 10.1007/s11432-016-5595-8.

[7] Ye Y, Kishida K. Toward an understanding of the motivation of open source software developers. In *Proc. the 2003 IEEE/ACM International Conference on Software Engineering*, May 2003, pp.419-49. DOI: 10.1109/ICSE.2003.1201220.

[8] Barcomb A, Kaufmann A, Riehle D, Stol K J, Fitzgerald B. Uncovering the periphery: A qualitative survey of episodic volunteering in free/libre and open source software communities. *IEEE Transactions on Software Engineering*, 2020, 46(9): 962-980. DOI: 10.1109/TSE.2018.2872713.

[9] Gousios G, Zaidman A, Storey M A, van Deursen A. Work practices and challenges in pull-based development: The integrator's perspective. In *Proc. the 37th International Conference on Software Engineering*, May 2015, pp.358-368. DOI: 10.1109/ICSE.2015.55.

[10] Yu Y, Wang H, Yin G, Wang T. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology*, 2016, 74: 204-218. DOI: 10.1016/j.infsof.2016.01.004.

[11] Thongtanunam P, Tantithamthavorn C, Kula R G, Yoshida N, Iida H, Matsumoto K. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Proc. the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, March 2015, pp.141-150. DOI: 10.1109/SANER.2015.7081824.

[12] Steinmacher I, Pinto G, Wiese I S, Gerosa M A. Almost there: A study on quasi-contributors in open-source software projects. In *Proc. the 40th International Conference on Software Engineering*, May 2018, pp.256-266. DOI: 10.1145/3180155.3180208.

[13] Yu Y, Li Z, Yin G, Wang T, Wang H. A dataset of duplicate pull-requests in GitHub. In *Proc. the 15th International Conference on Mining Software Repositories*, May 2018, pp.22-25. DOI: 10.1145/3196398.3196455.

[14] Gousios G, Storey M A, Bacchelli A. Work practices and challenges in pull-based development: The contributor's perspective. In *Proc. the 38th International Conference on Software Engineering*, May 2016, pp.285-296. DOI: 10.1145/2884781.2884826.

[15] Yu Y, Wang H, Yin G, Ling C X. Reviewer recommender of pull-requests in GitHub. In *Proc. the 2014 International Conference on Software Maintenance and Evolution*, September 2014, pp.609-612. DOI: 10.1109/ICSME.2014.107.

[16] Li Z X, Yu Y, Yin G, Wang T, Wang H M. What are they talking about? Analyzing code reviews in pull-based development model. *Journal of Computer Science and Technology*, 2017, 32(6): 1060-1075. DOI: 10.1007/s11390-017-1783-2.

[17] Li Z, Yin G, Yu Y, Wang T, Wang H. Detecting duplicate pull-requests in GitHub. In *Proc. the 9th Asia-Pacific Symposium on Internetware*, September 2017, Article No. 20. DOI: 10.1145/3131704.3131725.

[18] Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. In *Proc. the 29th International Conference on Software Engineering*, May 2007, pp.499-510. DOI: 10.1109/ICSE.2007.32.

[19] Wang X, Zhang L, Xie T *et al.* An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. the 30th International Conference on Software Engineering*, May 2008, pp.461-470. DOI: 10.1145/1368088.1368151.

[20] Nguyen A T, Nguyen T T, Nguyen T N, Lo D, Sun C. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proc. the 27th International Conference on Automated Software Engineering*, September 2012, pp.70-79. DOI: 10.1145/2351676.2351687.

[21] Lazar A, Ritchey S, Sharif B. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proc. the 11th Working Conference on Mining Software Repositories*, May 2014, pp.308-311. DOI: 10.1145/2597073.2597088.

[22] Porter M F. An algorithm for suffix stripping. In *Readings in Information Retrieval*, Jones K S, Willett P (eds.), Morgan Kaufmann Publishers Inc., 1997, pp.313-316.

[23] Manning C D, Schütze H. Foundations of Statistical Natural Language Processing. MIT Press, 1999.

[24] Sun C, Lo D, Wang X, Jiang J, Khoo S C. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. the 32nd International Conference on Software Engineering*, May 2010, pp.45-54. DOI: 10.1145/1806799.1806811.

[25] Sun C, Lo D, Khoo S C, Jiang J. Towards more accurate retrieval of duplicate bug reports. In *Proc. the 26th International Conference on Automated Software Engineering*, November 2011, pp.253-262. DOI: 10.1109/ASE.2011.6100061.

[26] Zhang Y, Lo D, Xia X, Sun J. Multi-factor duplicate question detection in stack overflow. *Journal of Computer Science and Technology*, 2015, 30(5): 981-997. DOI: 10.1007/s11390-015-1576-4.

[27] Mann H B, Whitney D R. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 1947, 18(1): 50-60.

[28] Thung F, Kochhar P S, Lo D. DupFinder: Integrated tool support for duplicate bug report detection. In *Proc. the 29th International Conference on Automated Software Engineering*, September 2014, pp.871-874. DOI: 10.1145/2642937.2648627.

[29] Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In *Proc. the 36th International Conference on Software Engineering*, May 2014, pp.356-366. DOI: 10.1145/2568225.2568315.

[30] van der Veen E, Gousios G, Zaidman A. Automatically prioritizing pull requests. In *Proc. the 12th Working Conference on Mining Software Repositories*, May 2015, pp.357-361. DOI: 10.1109/MSR.2015.40.

[31] Baysal O, Kononenko O, Holmes R*et al.* Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 2016, 21(3): 932-959. DOI: 10.1007/s10664-015-9366-8.

[32] Mcintosh S, Kamei Y, Adams B *et al.* An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 2016, 21(5): 2146-2189. DOI: 10.1007/s10664-015-9381-9.

[33] Fagan M E. Design and code inspections to reduce errors in program development. In *Pioneers and Their Contributions to Software Engineering*, Broy M, Denert E (eds.), Springer, 2001, pp.301-334. DOI: 10.1007/978-3-642-48354-7_13.

[34] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.712-721. DOI: 10.1109/ICSE.2013.6606617.

[35] Rigby P C, Storey M A. Understanding broadcast based peer review on open source software projects. In *Proc. the 33rd International Conference on Software Engineering*, May 2011, pp.541-550. DOI: 10.1145/1985793.1985867.

[36] Thongtanunam P, McIntosh S, Hassan A E, Iida H. Investigating code review practices in defective files: An empirical study of the Qt system. In *Proc. the 12th Working Conference on Mining Software Repositories*, May 2015, pp.168-179. DOI: 10.1109/MSR.2015.23.

[37] Jiang J, He J H, Chen X Y. CoreDevRec: Automatic core member recommendation for contribution evaluation. *Journal of Computer Science and Technology*, 2015, 30(5): 998-1016. DOI: 10.1007/s11390-015-1577-3.

[38] Rahman M M, Roy C K, Collins J A. CORRECT: Code reviewer recommendation in GitHub based on cross-project and technology experience. In *Proc. the 38th International Conference on Software Engineering Companion*, May 2016, pp.222-231. DOI: 10.1145/2889160.2889244.

[39] de Lima Júnior M L, Soares D M, Plastino A, Murta L. Developers assignment for analyzing pull requests. In *Proc. the 30th Annual ACM Symposium on Applied Computing*, April 2015, pp.1567-1572. DOI: 10.1145/2695664.2695884.

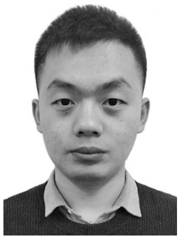
[40] Baum T, Liskin O, Niklas K, Schneider K. Factors influencing code review processes in industry. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp.85-96. DOI: 10.1145/2950290.2950323.

[41] Beller M, Bacchelli A, Zaidman A, Jürgens E. Modern code reviews in open-source projects: Which problems do they fix? In *Proc. the 11th Working Conference on Mining Software Repositories*, May 2014, pp.202-211. DOI: 10.1145/2597073.2597082.

[42] Morales R, Mcintosh S, Khomh F. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. In *Proc. the 22nd International Conference on Software Analysis, Evolution and Reengineering*, March 2015, pp.171-180. DOI: 10.1109/SANER.2015.7081827.

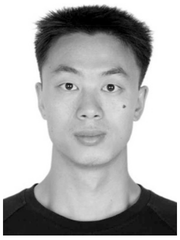
[43] Mcintosh S, Kamei Y, Adams B, Hassan A E. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proc. the 11th Working Conference on Mining Software Repositories*, May 2014, pp.192-201. DOI: 10.1145/2597073.2597076.

[44] Thongtanunam P, Mcintosh S, Hassan A E, Iida H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proc. the 38th International Conference on Software Engineering*, May 2016, pp.1039-1050. DOI: 10.1145/2884781.2884852.



Zhi-Xing Li is a Ph.D. candidate in the College of Computer at National University of Defense Technology (NUDT), Changsha. He received his Master's degree in computer science from NUDT, Changsha, in 2017. His work interests include software engineering, data mining, and knowledge discovering in open source communities.

Yue Yu is an assistant professor in the College of Computer at NUDT, Changsha. He received his Ph.D. degree in computer science from NUDT, Changsha, in 2016. His research findings have been published on MSR, FSE, IST, ICSME APSEC and SEKE. His current research interests include software engineering, spanning from mining software repositories and analyzing social coding networks.



Tao Wang is an assistant professor in the College of Computer at NUDT, Changsha. He received his Ph.D. degree in computer science from NUDT, Changsha, in 2015. His work interests include open source software engineering, machine learning, data mining, and knowledge discovering in open source software.

Gang Yin is an associate professor in the College of Computer at NUDT, Changsha. He received his Ph.D. degree in computer science from NUDT, Changsha, in 2006. He has worked in several grand research projects including National 973, 863 projects. He has published more than 60 research papers in international conferences and journals. His current research interests include distributed computing, information security, software engineering, and machine learning.

Xin-Jun Mao is a professor in the College of Computer, NUDT, Changsha. He received his Ph.D. degree in computer science from NUDT, Changsha, in 1998. His research findings have been published on Transaction on SMC, ICSE, Journal of Software: Evolution and Process, IJSEKE, and APSEC. His research interests include software engineering, multi-agent system, robot system, self-adaptive system, and crowdsourcing.

Huai-Min Wang received his Ph.D. degree in computer science from NUDT, Changsha, in 1992. He is now a professor and vice-president for academic affairs of NUDT, Changsha. He has been awarded the "Chang Jiang Scholars Program" professor and the Distinct Young Scholar, etc. He has published more than 100 research papers in peer-reviewed international conferences and journals. His current research interests include middleware, software agent, and trustworthy computing.