Refining Duplicate Contribution Detection in Pull-Based Projects

Vazgen Tadevosyan
Department of X
University X
vt7313@rit.edu

John Gianini
Department of X
University Y
John.J.Giannini@gmail.com

Professor Mohamed Mkaouer
Golisano College of Computing and Information Sciences Rochester Institute of Technology
mwmvse@rit.edu

November 9, 2024

1 Introduction

The pull-based development model is integral to many large open-source software (OSS) projects. On this model, programmers worldwide may branch a program's existing code, alter it to fix some issue, and then submit a pull request (PR) to merge their modified code into the official build of the software. PRs are met with a process of review and revision, which ultimately results in a decision to accept the project changes or reject them. Volunteer submissions to an OSS project contribute significantly to its rapid development. According to studies, communication and coordination between contributors have always been the main challenges to open-source software (OSS) communities [1].
The open-source model has proven extremely useful, but its decentralized nature can lead to significant redundancy of effort when multiple PRs are submitted to address the same issue, as they often are in practice. In the absence of rapid detection of these duplicates, numerous groups will work on reviewing and improving solutions to the same problem when only one of these solutions can be implemented. Thus, early detection of PRs that duplicate each other's purpose would be ideal, but is rendered difficult by the size and decentralized nature of OSS projects. Furthermore, prior studies indicate that duplication is among the most common reasons for rejecting PRs [2, 3], which means wasted effort from PR duplication is commonplace. Currently, the only way of finding duplicate PRs is manual identification by teams of reviewers, but this cannot be the most efficient or the best way to do it: popular projects receive too many PRs for duplicates to be reliably spotted. A practical method for the automatic detection of duplicate PRs would thus have great utility. Our research serves the goal of developing such an automatic method.
Some work [5] has been done on detecting duplicate PRs based on the textual similarity between the descriptions that form part of a PR. But individuals often use different words to make similar statements, even without the cultural and linguistic differences that may exist between programmers on international OSS projects. This problem is illustrated in Table 1. The first pair of PR descriptions differ because one contributor describes the problem in the title and then omits any further description, the other refers to an issue number in the title. This illustrates how even when contributors largely agree in their descriptions of a fix, lack of standard formatting can lead to a lack of detectable similarity. But there will also be cases where contributors describe an issue differently because they conceive it differently, as the second description pair illustrates.We thus seek to follow the work of [?] and improve the similarity measures used to find likely duplicates by incorporating other sources of similarity information: similarity between the actual changes to project code involved in PRs. Furthermore, we aim to refine the methods used to combine similarities, i.e., textual and change similarity, into a ranking of likely duplicates thus making automated duplication more feasible.

Table 1: The Titles & Descriptions from Two Pairs of Duplicate PRs

Duplicate PR Pair 1 (2307 & 2362 from Angular.js)		
Title	Fixed incomplete merge	fix for #2361
Description	[no description]	removed merge conflict comment in angular-mocks.js
Duplicate PR Pair 2 (14069 & 14073 from Joomla-CMS)		
Title	'STRICT_TRANS_TABLES' on MySQL 5.7	Module publish up, down and checkout out time should always have a null date if not set
Description	With this parameter 'STRICT_TRANS_TABLES' on MySQL 5.7.16: When I try to save a module I get an error: Save failed with the following error: Incorrect datetime value: '' for column 'publish_up' at row 1 When I try to save the content language I get an error: Save failed with the following error: Field 'asset_id' doesn't have a default value	Pull Request for Issue #13031. Summary of Changes This is a fix aimed towards the Joomla 4.x branch that ought to be in the 3.x branch too. Currently we don't always set this value (but with STRICT_TRANS_TABLES disabled it automatically has a null date applied). In the 4.x branch we've enabled this parameter causing modules to fail to save. [...]

2 Study Design

We constructed our duplicate detection algorithm on the template of the one laid out by Li et al.[5], [**?**]], though we have altered their design at several points in hopes of obtaining increased performance. In general, the method is straightforward: calculate similarity scores based on various dimensions in which a PR may resemble another. These scores have the general form (actual similarity) / (possible similarity). Then these individual scores are aggregated and the results used to identify likely duplicates. Figure 1 provides a high-level overview of our proposed method.

Then these individual scores are aggregated, and the results used to identify likely duplicates. Figure 1 provides a high-level overview of our proposed method.

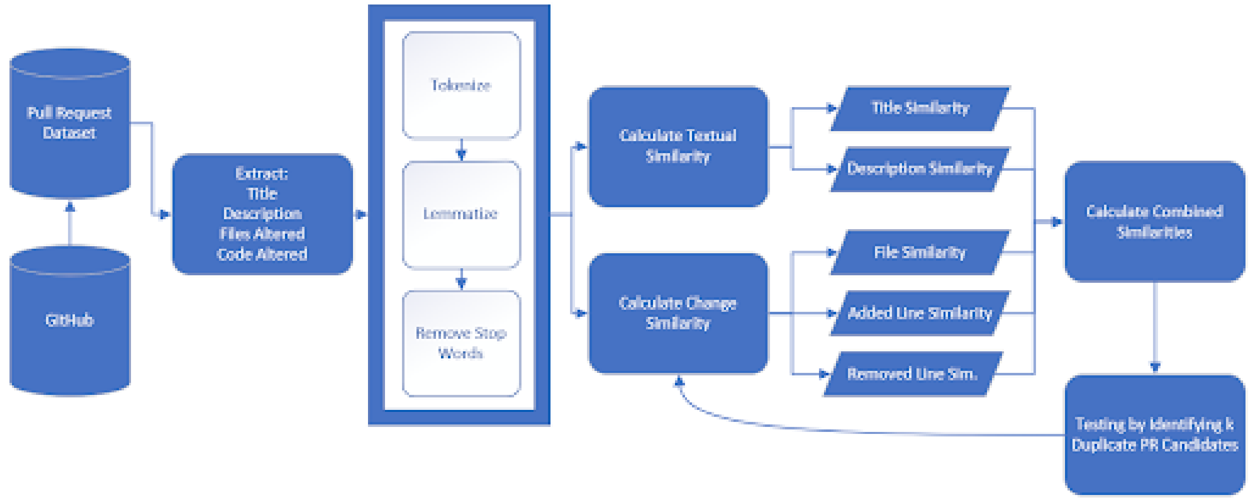
Pull Request Dataset
GitHub
Extract: Title Description Files Altered Code Altered
Tokenize
Lemmatize
Remove Stop Words
Calculate Textual Similarity
Calculate Change Similarity
Title Similarity
Description Similarity
File Similarity
Added Line Similarity
Removed Line Sim.
Calculate Combined Similarities
Testing by Identifying k Duplicate PR Candidates

Figure 1: An Overview of our Approach

We draw data from a dataset of PRs taken from 16 GitHub projects. The data set includes pairs of duplicate PRs and PRs without a duplicate from each project in roughly equal numbers. The title, description, files

altered, and code changes made by each PR are contained in our dataset.
Broadly, we will assess the similarity of pull requests based on textual similarity, that is similarity between the descriptions of two PRs, and change similarity, that is similarity of their alterations to the code. These are further subdivided into similarity of title, description, files altered, and two sorts of code changes.

In most categories, the similarity measures will employ Natural Language Processing (hereafter NLP) to some degree. The exceptions are comparison of file paths and comparison of lines of code removed. Our specific method for calculating each sort of similarity is discussed at length below. A weighted average of these similarity measurements is then calculated, and used to search for duplicate PRs: the k most similar PRs are chosen. We consider our method successful if the actual duplicate is contained in that top-k. In turn, the success of this search will be used to tune the parameters of our similarity measurements further.

2.1 Calculating Similarities

The general formula underlying all our similarity scores is given in equation 1:

$$\text{similarity score}(a, b) = \frac{\text{summed actual similarity}(a, b)}{\text{maximum possible similarity}(a, b)} \tag{1}$$

Thus our similarity values will fall between 0 and 1 inclusive, and can intuitively interpreted, roughly, as percentage of perfect resemblance between a and b actually exhibited. However, the specifics of how actual and possible similarity are calculated varies substantially from score to score. We ultimately calculate similarity in five aspects: similarity of title (title_sim), of description (desc_sim), of files altered (fp_sim), of lines of code removed (del_sim), and of lines of code inserted (add_sim). Details of each calculation are discussed in the following sub-sections.

2.1.1 Similarity of Title and Description

Each PR has a title and description written in natural language by the submitter. Thus, a straightforward NLP-based measure of textual similarity is employed for these two elements. First, titles and descriptions in the entire dataset undergo standard NLP pre-processing: they are first lemmatized, a process by which individual tokens are reduced to a common base form such that similarity can more easily be identified. (For instance, 'fitting' and 'fitted' might both become 'fit'.) Second, stop words are removed from the vector—words too ubiquitous to be of use in calculating similarity ('the', 'and', 'it', etc.) Finally, text is tokenized—broken into a TF-IDF vector of linguistic tokens. This done, a data frame is generated to hold the per-word importance from each item. Similarity scores are generated by measuring the cosine similarity between items in this data frame based on the calculated importance weights. To illustrate the process, consider these two pull request titles from the rails/rails project in which the second duplicates the first: To illustrate the process, consider these two pull request titles from the Rails/Rails project, in which the second duplicates the first:

- **#3066** *"When running plugin new, the generated .gitignore should include a non-project file."*
- **#3591** *"Fixed error with rails generate new plugin where the .gitignore was not."*

Here is the same pair of string strings transformed by lemmatizing and removing stop words:

- **#3066** *"running plugin new generate .gitignore include non-project file"*
- **#3591** *"fixed error rail generate new plugin .gitignore"*

Notice that the lemmatization process has changed 'rails' to 'rail' and removed capitalization, while words like 'when', 'the', and 'was' have been removed as unhelpful stop words. Next, these strings must be represented as a vector. This will be done according to Equation 2, where $tf_{i,j}$ is the number of occurrences of word i in document j (in this case, a title string), df_i is the number of documents containing word i, and N is the total number of documents.

$$W_{i,j} = tf_{i,j} \cdot \log\left(\frac{N}{df_i}\right) \tag{2}$$

TF-IDF assigns an importance weight score to every word found in the documents. If a word does not appear, it receives 0. If the word appears in every document or title, it again receives 0. The importance score is high only if the word is unique to the document and rarely appears in others. Put differently, words count towards similarity to the extent that they are rare in the dataset. The vector size of each title is the number of unique words in all titles in the dataset (not just these two), which for us is 4485. Illustrating with the full TF-IDF vector is pointless because of its size, so we will use a simplified, but analogous, vectorization approach here: a vector with only eleven entries, one for each unique word found in the two titles. The counts of each word are recorded:

$$\textbf{\#3066} \quad \vec{t_1} = (1,1,1,1,1,1,1,1,0,0,0)$$
$$\textbf{\#3591} \quad \vec{t_2} = (0,1,1,1,1,1,0,0,1,1,1)$$

Cosine similarity is used to measure similarity of the produced vectors, that is, the dot product of the vectors and divided by the product of the Euclidean norms for each vector (see formula 3).

$$\text{CosineSimilarity}(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|} \tag{3}$$

This distance measure preserves the intuition that samples which contain similar words in similar proportions are very similar even when they differ markedly in length. The score for this sample pair will be ~ 0.54.

We repeat this process for pairs of titles and pairs of descriptions across our entire dataset.

2.1.2 Similarity of File Path

Each PR contains a list of project files which it alters. Intuitively, altering the very same files signifies close similarity between two PRs but, also, altering files from very similar parts of the project's directory structure signifies some degree of similarity. These intuitions are captured by two algorithms. Algorithm 1 gives a similarity measure for two file paths. It does this by splitting them into elements by the path separator and returning the number of path elements shared prior to the first divergence divided by the total number of elements in the longer path. Thus, the return value is the percentage of the longer path which is shared. Algorithm 2 combined the pairwise similarities generated by algorithm 1 into a similarity score by repeatedly selecting the best matches, and then ensuring that a given file may not be repeatedly matched. This produces a list of match qualities for all files in whichever PR altered fewer files. These qualities are summed and then divided by the maximum number of files either PR alters.

Algorithm 1 Calculating Similarity Between Two Files

```
Input: file path $file_1$, and file path $file_2$
Output: The similarity of location between $file_1$ and $file_2$
$pl_1$, $pl_2$ ← split $file_1$ into path components, split $file_2$ into path components
 pos ← 0
 while pos ¡ min(len($pl_1$), len($pl_2$)) do
     if $pl_1$[pos] ≠ $pl_2$[pos] then
         break
     pos++
return pos / max(len($pl_1$), len($pl_2$))
```

Algorithm 2 Calculating File Path Similarity Between PRs

```
Input: files$_1$ and files$_2$, lists of files altered respectively by PR$_1$ and PR$_2$
Output: The aggregate similarity of files altered by PR$_1$ and PR$_2$
let file_sims be a list
for $f_1$ in $files_1$ do
    for $f_2$ in $files_2$ do
        sim ← algorithm_1(f$_1$, f$_2$)
        add (f$_1$, f$_2$, sim) to file_sims
sort file_sims in descending order by sim
let final_sims be a list
while top_n ¡= min(len(files$_1$), len(files$_2$)) do
    top_sim ← file_sims[0]
    add top_sim[0] to final_sims
    delete items from file_sims where item[0] == top_sim[0] or item[1] == top_sim[1]
    top_num–
return sum(final_sims) / max(len(files$_1$), len(files$_2$))
```

2.1.3 Similarity of Lines Removed

`del_sim` is the simplest of the calculated similarity scores because it can be assumed that lines of code removed are taken from the same pre-existing file and thus perfectly match one another. [**?**] Thus we can calculate the similarity of lines deleted between two PRs with the formula given in equation 4:

$$\text{del\_sim}(a, b) = \frac{N(\text{del\_lines}_a \cap \text{del\_lines}_b)}{N(\text{del\_lines}_a \cup \text{del\_lines}_b)} \tag{4}$$

This counts the overlap of the sets of removed lines in the two PRs and divides it by a count of the union of those same sets. When the overlap is total, the answer will be 1, otherwise, it will be the fraction of all lines deleted shared between PRs.

2.1.4 Similarity of Lines Added

In contrast to lines removed, calculating the similarity of lines added is complex. This complexity stems from three intuitions that guided our method: (1) partial similarity of code lines should be counted, (2) similarity between lines inserted into different files should not be counted, and (3) text in comments should be compared differently than code since comments contain natural language.
Similarity of added lines is measured using Algorithm 3 (which operates similarly to Algorithm 1, save that it proceeds file-by-file rather than making a full pairwise comparison).

Algorithm 3 Calculating The Similarity of Added Lines Between PRs

```
Input: files$_1$ and files$_2$, lists of files altered respectively by PR$_1$ and PR$_2$
Output: The aggregate similarity of per-file lines altered by PR$_1$ and PR$_2$
files_common ← intersection of files$_1$ and files$_2$
let add_sims be a list
let num_lines$_1$ and num_lines$_2$ be equal to 0
for f in files_common do
    add_lines$_1$, add_lines$_2$ ← addedLines(PR$_1$, f), addedLines(PR$_2$, f)
    num_lines$_1$ += len(add_lines$_1$)
    num_lines$_2$ += len(add_lines$_2$)
    let line_sims be a list
    for line$_1$ in add_lines$_1$ do
        for line$_2$ in add_lines$_2$ do
            tokens$_1$, tokens$_2$ ← tokenize(line$_1$, line$_2$)
            tmp_sim ← len(intersection(tokens$_1$, tokens$_2$)) / len(union(tokens$_1$, tokens$_2$))
            add (tokens$_1$, tokens$_2$, tmp_sim) to line_sims
    sort line_sims in descending order by tmp_sim
    let final_sims be a list
    while top_n ¿ 0 do
        top_sim ← line_sims[0]
        add top_sim[2] to final_sims
        delete items from line_sims where item[0] == top_sim[0] or item[1] == top_sim[1]
        top_num–
return sum(add_sims) / max(num_lines$_1$, num_lines$_2$)
```

A comparison is made only of lines added to the same files. For each such file, tokenized lists of words are compared for similarity. The `tokenize()` function differentiates between lines containing actual code and lines containing comments, and handles each differently. Lines containing code are simply broken around whitespace, while lines containing comments also have stop words and punctuation removed.
Once all pairwise comparison of added lines has been done within a file, aggregate similarity for that file is computed by repeatedly selecting the best matches, and then ensuring that a given line may not be repeatedly matched.
Finally, all per-file aggregate scores are summed then divided by the number of lines added by whichever PR added more. The resulting number is the similarity score, `add_sim`.

2.2 Combining Similarities

With all five similarity scores calculated (title_sim, desc_sim, fp_sim, del_sim, and add_sim), it remains to compute combined similarity. This is done by taking the weighted average of the individual similarity scores. Li Z., Yu Y, Wang T *et al.* use a very similar approach, but we have departed from their method in one key area: they combined del_sim and add_sim into a single similarity measure (p 196). However, it appears arbitrary to leave title_sim and desc_sim separate yet combine del_sim and add_sim into a single score. Also, this locks del_sim and add_sim into having equal weights, and caps their possible weights at half of what other scores may receive. Hence we opted to assign them separate weights, allowing for greater potential flexibility in score calculation.
For our first trial, we have assigned equal weights to all scores. Li Z., Yu Y, Wang T *et al.* report this to be only slightly less successful than a method which searches for more ideal weights, yielding an average recall of 0.826 with a k of 20, while using tuned weights only improved their average recall to 0.834 (p 202). Given the parameters of our project, we have thus opted to use equal weighting as it seems unlikely to significantly impact performance.

Table 2: A Pair of Scored PRs from Angular.js

	PR 3848 content	**PR 4308 content**	**Score**
Title	feat($parse): support unicode identifier names	feat(expressions): allow non-English Unicode letters in identifiers	0.463
Description	Support unicode identifier names as defined in Section 7.6 Identifier Names and Identifiers, ECMAScript Language Specification (http://www.ecma-international.org/ecma-262/5.1/#sec-7.6), except for unicode escape sequences which is hard to implement without changing too much existing code. Closes #3847	Allow most of the non-English Unicode letters in identifiers inside AngularJS expressions. Please see #2174 for the discussion. Closes #2174	0.209

Lines Removed	return 'a' <= ch && ch <= 'z' ||
	'A' <= ch && ch <= 'Z' ||
	'_' == ch ||ch == '$';
	if (ch == '.' ||isIdent(ch) ||
	isNumber(ch))
		[none removed]	0.0

Lines Added	var UNICODE = (function () {
	var letterRanges = {
	[continued for 131 lines]
		var unicodeLetterRanges =
		'\u0041-\u005A\u0061-\u007A\u00AA\u00B5\u00BA' +
		[continued for 120 lines]
			0.048

Files Altered	src/ng/parse.js test/ng/parseSpec.js	src/ng/parse.js test/ng/parseSpec.js	1.0
Combined			0.343

[2]Actually, what may be an error in Li Z., Yu Y, Wang T *et al.* renders their method on this point unclear. Their algorithms and formulas, as printed, seem to result in their deleted line similarity score being divided by its possible maximum value twice, once individually (see their formula 4) and once along with the added line score (see their formula 3). Effectively, their raw score would be normalized twice. This would often render the deleted line similarity score quite small relative to other scores, minimizing its impact for no discernible reason. It is unclear to us whether this is the procedure they actually used or a mistake in their description.

2.3 Searching Based on Similarity Scores

The final step in our procedure employs the combined similarity scores to find duplicate PRs, operating on the assumption that duplicate PRs will be more similar to one another than they will be to non-duplicates. For each project in our dataset, each PR with a duplicate is compared to every other PR and a similarity score is generated. These similarity scores and associated PR ids are then put into a list and sorted, and we record the place in the ordering the actual duplicate appears as the *match_rank*. (A *match_rank* of 0 indicates that the duplicate was more similar than any other PR, and a *match_rank* of 4 that it was fifth most similar, etc.)

Conveniently, those PRs with a *match_rank* less than k are those for which our method would return their duplicates in a search of k candidates. This property is used for validation below.

Figure 3 illustrates our method with an actual pair of PRs from the Angular.js project. Note that many of the individual similarity scores are low, including a complete lack of overlap in lines removed as one PR removed no lines. The overlap between the many lines added is also quite slight. Title and description each enjoy more noticeable similarity, and the lists of files altered are a perfect match. This resulted in a combined similarity of 0.343, making 4308 the PR most similar to 3848. The next runner-up for similarity to 3848 has a combined similarity of only 0.241—it too alters the same two files, but lacks as high a degree of similarity elsewhere. Since 3848 and 4308 are duplicates of one another, this means that each would receive a *match_rank* of 0, and that our method of finding duplicate PRs would succeed on this pair at any k.

3 Experiments

3.1 RQ1. Which sorts of similarity measurements are most helpful for detecting duplicate PRs?

Though Li Z., Yu Y, Wang T *et al.* report that combined similarity measures incorporating text, files altered, etc. is the most effective, they report that similarity of code (lines added and lines deleted computed together) is quite effective on its own, and that file path similarity comes in second. We want to explore further the relative contribution of the different similarity measures to successfully identify duplicates. This information would be useful for weighing and making decisions about where effort to improve similarity scoring would be most useful.

To that end, we've measured the success of all five sorts of similarity in identifying duplicates. We follow many other studies in using *recall-rate@k* as a measure of success. This measure has the added benefit of measuring something which would be of great importance if the method we are developing were to be employed in development, i.e., does a search for k candidates find true duplicate PRs? For a given k, the *recall-rate@k* is defined by formula 5.

$$\text{recall-rate@}k = \frac{n_{\text{detected}}}{n_{\text{total}}} \tag{5}$$

n_{detected} is the number of duplicate PRs detected by a search for k candidate duplicates, and n_{total} is the number of PRs in our dataset with duplicates that could be detected.

[3]Here and elsewhere in the paper we round results to three decimal places.

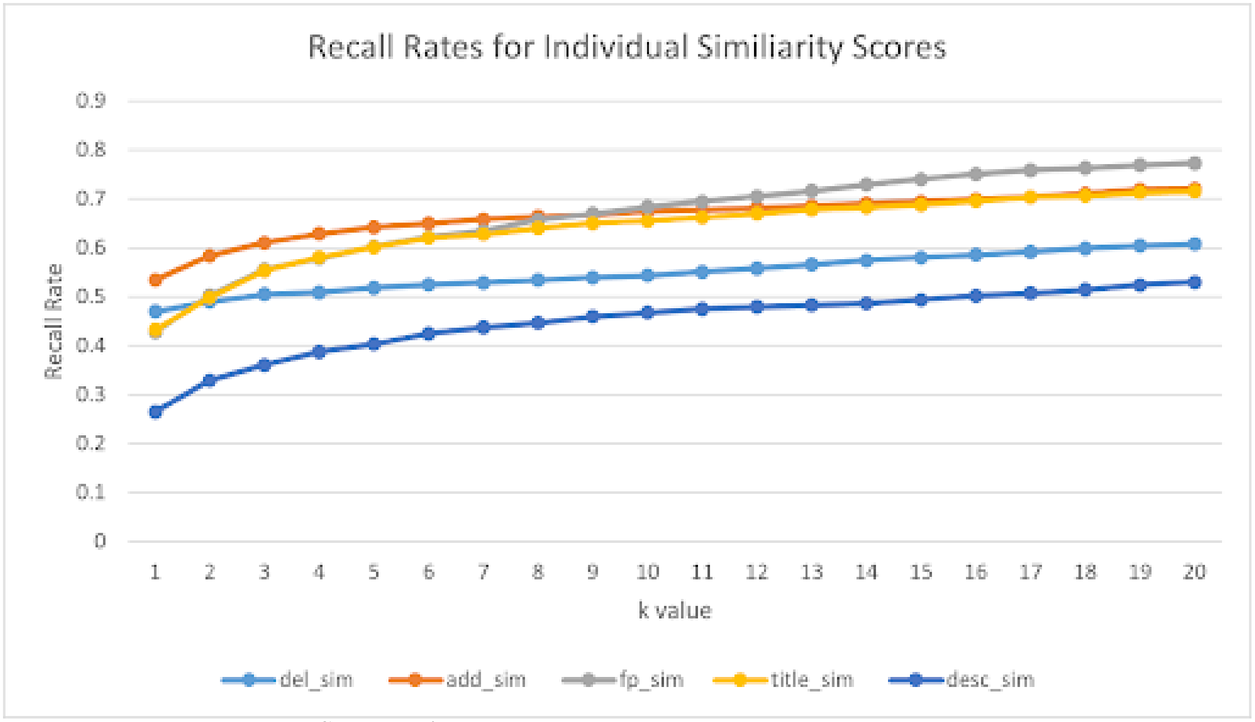

Figure 2: Success of search based on individual similarity measures alone

Figure 2 compares the average recall rates associated with using the various scoring methods on their own to find candidate duplicate PRs. It reveals `add_sim` as the most helpful individual indicator of similarity for $k < 10$, while for $k \geq 10$, `fp_sim` is the most helpful. File path similarity is remarkably effective, significantly more so than reported by Li Z., Yu Y, Wang T *et al.*, though its success also varies considerably by project (from as little as 0.474 to as much as 0.925 for $k = 20$).
Another intriguing finding is that, at $k = 0$, `del_sim` outperforms most of the other measures. So, if one wanted to find only a single candidate duplicate, similarity of lines of code deleted might be a better measure than similarity of title or files.

3.2 RQ2. To what extent does combining similarity measurements improve the most successful individual similarity measures at identifying duplicate PRs?

Despite the great success of some individual measures, our data categorically confirms that combining similarity measures improves overall recall. Figure 3 shows the average recall rates obtained by combined similarity compared to `add_sim` and `fp_sim`. Combined similarity is noticeably more successful at all k values. For example, the average recall for the combined method with a k of 20 is 0.89 while the most successful individual measure, file similarity, alone has an average recall of 0.772 at the same k. Also, the combined measure is much more consistent across projects, as shown in Figures **??** and **??**, which respectively show the recall rate by project of combined similarity and of `fp_sim` alone.

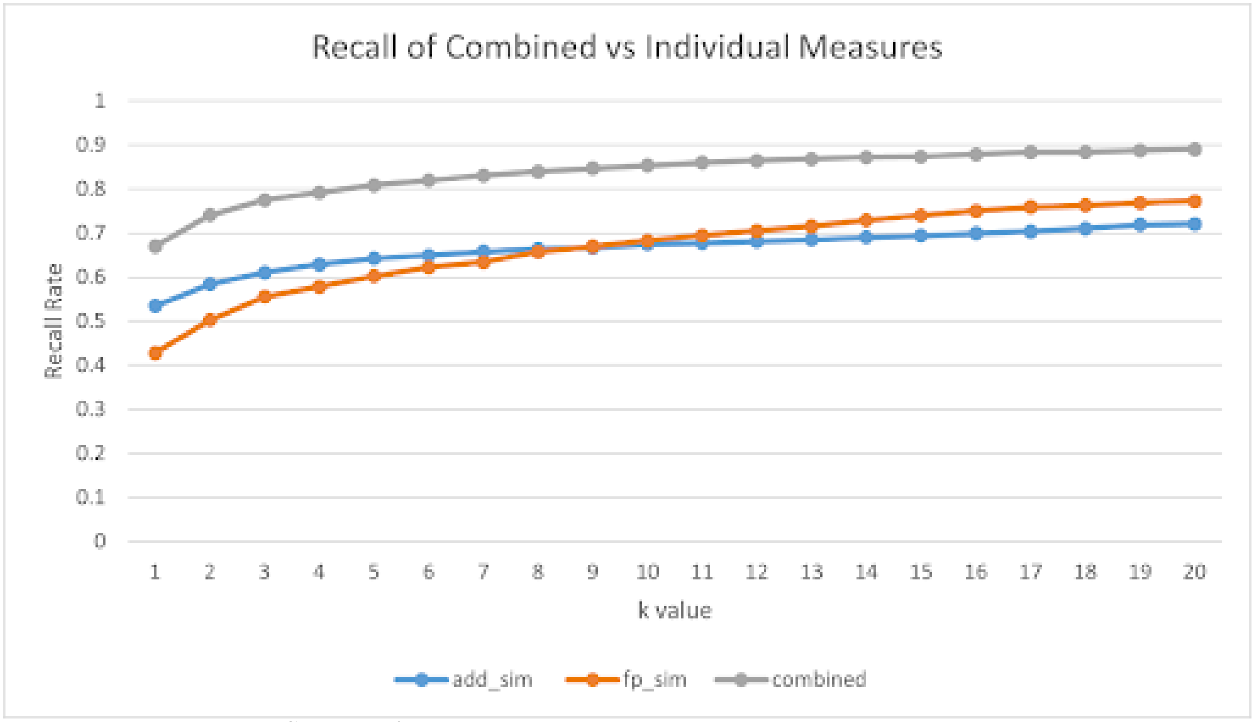

Figure 3: Success of search based on combined measure vs individual measures

Examining some pairs of pull requests in our dataset will help illustrate why combined similarity seems to improve performance in practice. Figure 6 contains one of the pairs of duplicate PRs from the introduction along with similarity scores. This pair highlights the messiness in actual PRs which makes detecting duplicates a challenge: PR 2307 has no description; and one PR uses a description as a title while another references an issue number. As a result, title and description do not register as similar. They also lack any similarity in lines added since only one of them adds anything. So the duplication would be missed entirely if description, title, or code added were used individually or in combination.
The two PRs do have perfect similarity in files altered, but they share this feature with two other PRs in our dataset (and likely with a number more if the entire project were examined). Thus, using file path alone would give us some idea where to look for a duplicate, but not enough to confidently identify it.

Table 3: A Pair of Scored PRs from Angular.js

	PR 2307 content	**PR 2362 content**	**Score**
Title	Fixed incomplete merge	fix for #2361	0.0
Description	No description provided.	removed merge conflict comment in angular-mocks.js	0.0
Lines Removed	`<<<<<< HEAD` * *NOTE*: this function is also published on window for easy access. * *NOTE*: Only available with @link http://pivotal.github.com/jasmine/ jasmine ======== ¿¿¿¿¿¿ 8dca056... docs(mocks): fix typos	`<<<<<< HEAD` * *NOTE*: this function is also published on window for easy access. * *NOTE*: Only available with @link http://pivotal.github.com/jasmine/ jasmine ======== ¿¿¿¿¿¿ 8dca056... docs(mocks): fix typos	1.0
Lines Added	* *NOTE*: Only available with @link http://pivotal.github.com/jasmine/jasmine/	[none added]	0.0
Files Altered	src/ngMock/angular-mocks.js	src/ngMock/angular-mocks.js	1.0
Combined			0.4

This pattern is observed in many places in the data: one or two types of similarity are high for a pair of duplicate PRs while the other measures fail to indicate significant similarity. And which are the indicative measures of similarity varies greatly from case to case even if some are more likely to be of use (again see Figure 4). But, just as in the above example, aggregating types of similarity frequently allows the one or two high scores to predominate, in this case identifying the duplicate as the most similar candidate despite the many dimensions on which similarity is imperfect.
Of course, our method performs poorly when duplicate PRs simply lack detectable similarity to one another in any dimension. But, more interestingly, there is the risk that combining scores will sometimes dilute the successful indicators with the less indicative ones—and we do observe instances of this in our data. See Figure 7 for an illustrative pair from the joomla-cms project.
array tabularx

Table 4: A Pair of Scored PRs from joomla-cms

	PR 14069 content	**PR 14073 content**	**Score**
Title	’STRICT_TRANS_TABLES’ on MySQL 5.7	Module publish up, down and checkout out time should always have a null date if not set	0.0
Description	With this parameter ’STRICT_TRANS_TABLES’ on MySQL 5.7.16: When I try to save a module I get an error: Save failed with the following error: Incorrect datetime value: ” for column ’publish_up’ at row 1 When I try to save the content language I get an error: Save failed with the following error: Field ’asset_id’ doesn’t have a default value	Pull Request for Issue #13031. Summary of Changes This is a fix aimed towards the Joomla 4.x branch that ought to be in the 3.x branch too. Currently we don’t always set this value (but with STRICT_TRANS_TABLES disabled it automatically has a null date applied). In the 4.x branch we’ve enabled this parameter causing modules to fail to save. [...] Documentation Changes Required	∼0.2
Lines Removed	’STRICT_TRANS_TABLES’, ’STRICT_TRANS_TABLES’, ’STRICT_TRANS_TABLES’ (the same thing was removed from three separate files)	[none removed]	0.0
Lines Added	[none added]	`public function store($updateNulls = false) {... }` [comments omitted]	0.0

Files Altered	`libraries/joomla/database/driver/mysql.php`
	`libraries/joomla/database/driver/mysql_i.php`
	`libraries/joomla/database/driver/pdomysql.php`
		`libraries/legacy/table/module.php`	∼0.06

Combined			0.052

These two PRs address the same issue in different ways, and thus have little to no similarity in every dimension but description. Since both mention ‘STRICT_TRANS_TABLES’, ‘module’ etc., the duplicate would be in the top 5 candidates if description similarity alone were used. When it is aggregated with the other scores, the duplicate is not even in the top 20—it would be the 26th candidate.

These two PRs address the same issue in different ways and thus have little to no similarity in every dimension but description. Since both mention ‘STRICT_TRANS_TABLES’, ‘module’, etc., the duplicate would be in the top 5 candidates if description similarity alone were used. When it is aggregated with the other scores, the duplicate is not even in the top 20—it would be the 26th candidate.

Still, overall, the method that combines similarities performs significantly better than any individual measure as cases where the indicative scores predominate are more frequent than cases where they are swamped.

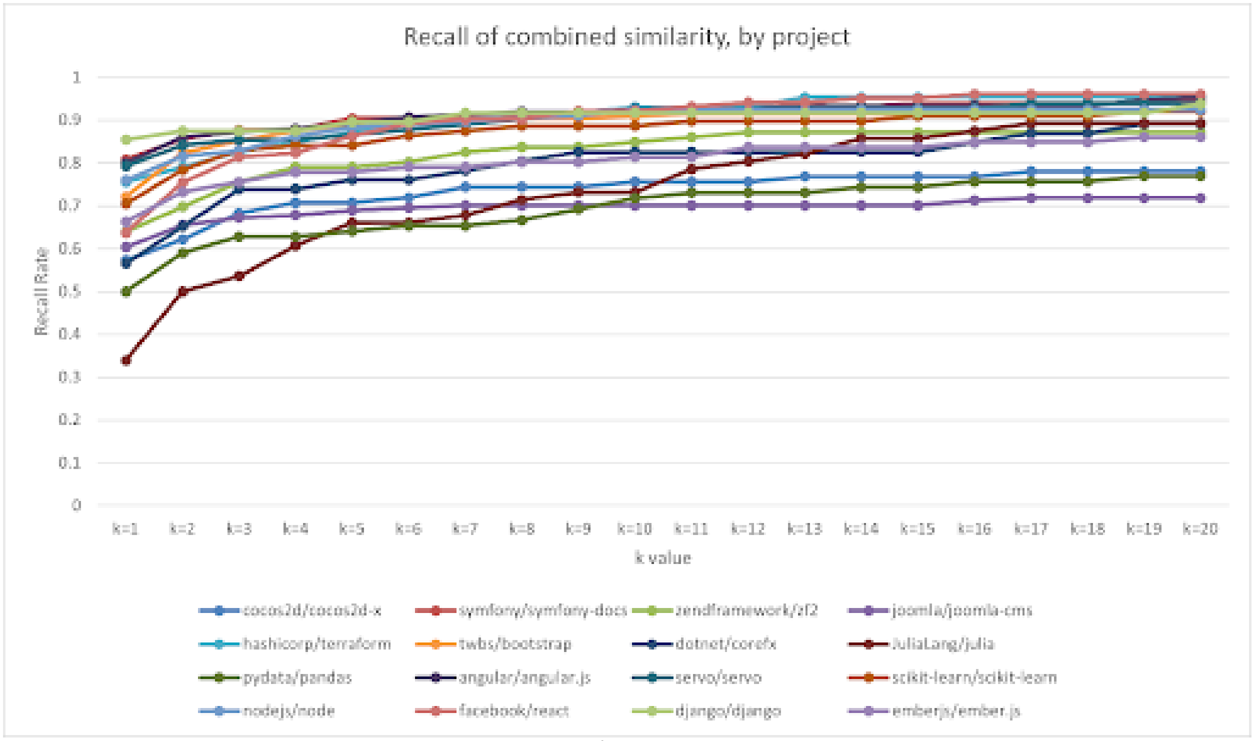

Figure 4: Per-project success of search based on combined measure

3.3 RQ3. Are there ways to improve the measurements of similarity for detecting duplicate PRs?

This entire project has been an attempt to refine the methods featured in prior work such as Li Z., Yu Y., Wang T et al. As such, we identified several points at which we think improvements are possible in the paper above: (1) splitting removed lines and added lines into separate measures of similarity, (2) processing comments differently than code for finding similarity, (3) using lemmatization instead of stemming for title and description text.

Our recall-rate@k numbers suggest that we may have succeeded. No direct comparison of results is possible, as we are working with a different data set. But we've achieved higher average recall values than Li Z., Yu Y., Wang T et al. reported, and without searching for more ideal weights as they did, see table 5. And, while they tested their method on more total PRs—2,323 pairs from 26 projects compared to our 1,631 pairs from 16 projects—that also means that we have more PRs per-project on average, meaning our findings shouldn't be less robust. That said, we are cautious in our conclusions due to the size of our dataset. (See the Threats to Validity section below.)

In working on this procedure, we've also identified some directions for further improvement. For one, we found that differences in whitespace formatting can throw off the success of removed line similarity measurement. This suggests removing whitespace prior to measurement.

Table 5: Comparison of Average Recall Rates

	K=1	K=5	K=10	K=15	K=20
Our average recall	0.67	0.809	0.853	0.873	0.89
Li Z., et al. average recall	0.520	0.735	0.792	0.818	0.834

In one project, Joomla-cms, employing comparison of title and description noticeably decreased recall. Further exploration of this surprising result revealed that many PRs have a title, description, or both that employ boilerplate language. This throws off similarity measurement because many PRs have nearly identical descriptive components (which are nonetheless unique enough globally to register as highly similar).

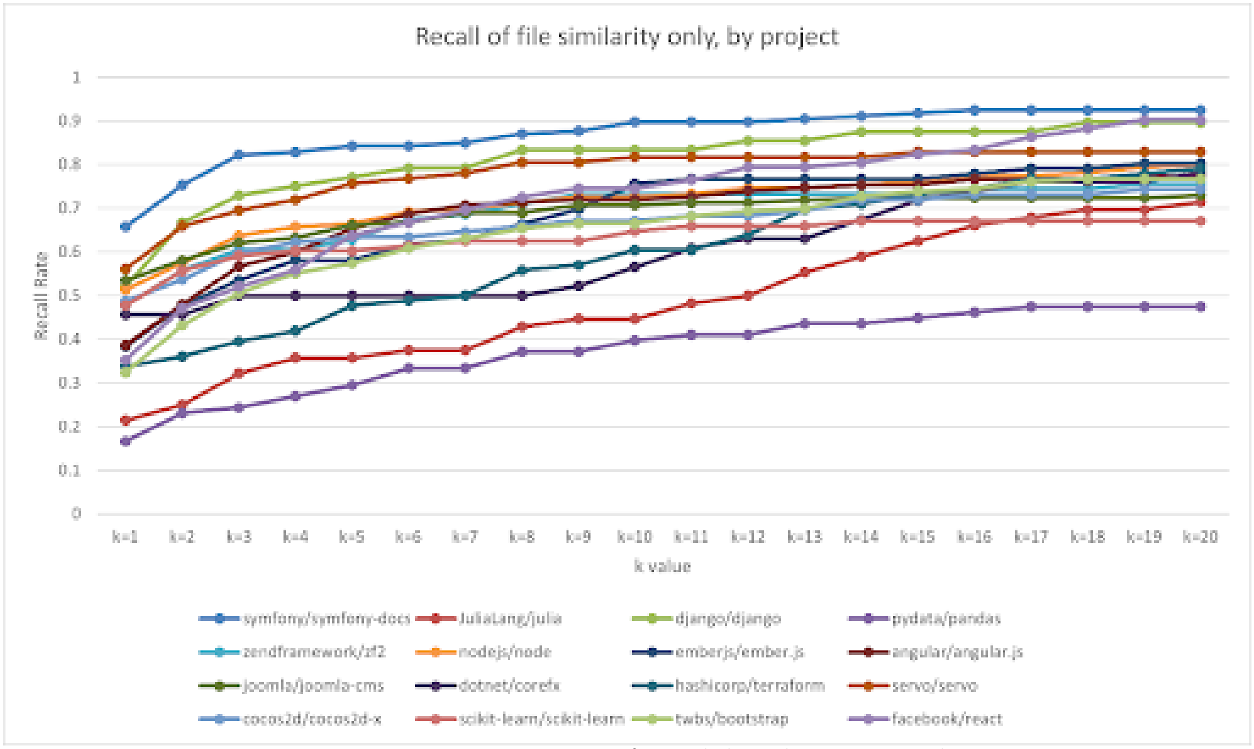

Figure 5: Per-project success of search based on `fp_sim` alone

This suggests that, for such projects, it would be worthwhile to develop a way to "zero out" these unhelpful bits of repeated text.

4 Related Works

In this section, we look at the related works surrounding the paper and how things related to our project like duplicate PR detection, pull-request evaluation, code review, etc., then we compare the works done by other papers as compared to how it's done in our project.

Firstly, we look at how the evaluation of pull requests has been done previously and what we can learn from those studies. Several studies have been carried out to examine how PRs are applied and evaluated, even though the research on PRs is still in its early stages. Millions of PRs from GitHub were statistically analyzed by Gousios et al. [2] to determine which PRs were popular, what factors affected the decision to merge or reject a PR, and when to merge a PR. Additionally, Gousios et al. [2] studied the work practices and difficulties in the pull-based development model from the perspectives of integrators and contributors, respectively.

Secondly, we look at how duplicate PR detection has been done previously and what we can learn from those studies. Duplicate bug reports have received a lot of attention from researchers. When Runeson et al. [6] examined how NLP techniques support the identification of duplicate reports, they discovered that 40% of the duplicates could be found. Sun et al. [7]'s evaluation of three sizable software bug repositories revealed that their method outperformed methods that used natural language to detect duplicates. Later, Sun et al. [7] proposed a retrieval function to make the most of the data in a bug report and compare the two bug reports' similarities. Additionally, Zhang et al. [8] investigated finding duplicate Stack Overflow questions. They compared observable factors, such as the questions' titles, descriptions, and tags, as well as latent factors corresponding to the topic distributions inferred from the descriptions of the questions, to determine how similar the two questions were.

Finally, we look at how other papers have looked at code review and how they have examined changes made by others in source codes and found potential defects in them and therefore ensure software quality. As proposed by Fagan [9], traditional code review has been used since the 1970s, but due to its synchronous and cumbersome nature, it has not been widely adopted. Modern Code Review (MCR) has recently gained popularity among software teams and companies. Numerous aspects of code review have been extensively researched, including the automation of review tasks, variables affecting review conclusions, and difficulties encountered during code review. Numerous studies have examined code review's influence on software quality in terms of coverage, participation, and code ownership. While finding defects to control software quality was once thought to be the primary driver of code review, more recent research has shown that this is not always the case.

5 Threats to Validity

The purpose of a threats to validity section is to acknowledge any factors that might have undue influence on the research or skew the data being collected. The four main types of validity threats are conclusion, internal, construct, and external, and they have been elaborated on below for this project.

5.1 Conclusion

Threats to conclusion validity concern the relationship between the treatment and the outcome. It is mainly about how the steps taken in the project affect the ability to draw the correct conclusion about the relations between treatment and outcome.

We are concerned that our dataset's relative size may lead to higher recall numbers than would be observed in an actual project environment. Our data contains many pull requests, but they are broken up by project such that we can only compare a few hundred from any one project. As a result, it is reasonably unlikely that we'll have more than a few PRs which modify the same files, delete the same lines of code, etc. But presumably, many similar non-duplicate PRs would be present in an actual project.

As a result, we are tentative about claiming that our method could obtain similar success when employed on the full PR history of an open-source project. This is a shortcoming shared with other papers in the literature and would be a good subject for further investigation.

5.2 Internal

Threats to internal validity consider causality between an action taken and the resulting change that can be observed.

Since the extraction rules may match not all indicative comments, the dataset of historical duplicate PRs may contain false negatives. Additionally, some reviewers might simply close the duplicate PRs without making any comments. To further validate the efficacy of our method, we would need to gather more projects and enrich the dataset in the future.

5.3 Construct

Threats to construct validity refer to the extent to which the experiment setting reflects the theory factors to consider.

So for this, we must consider whether we are choosing the correct similarities to compare with and whether the selected similarities matter and give the correct answer. We compare five types of similarities and then say that this work individually and work even better when combined which shows that we have chosen the correct similarities to compare.

5.4 External

Threats to external validity relate to the utility of the results in a research study. Are the researcher's findings applicable in a real-world setting?

Our research is based on a few of the well-known open-source projects available on GitHub. The projects are created using a variety of programming languages and used in various fields. We do not yet know if our approach can be applied to all GitHub projects as well as open-source projects hosted on other platforms.

6 Conclusion

In this project, we put forth a method for automatically detecting duplicate PRs on GitHub. Our method uses textual and change similarity to determine how similar two PRs are and then returns a list of historical PRs that are most like the newly arrived PR. We tested our method using a dataset of historical duplicates gathered from 16 well-known projects hosted on GitHub.
For all the five similarity scores (title_sim, desc_sim, fp_sim, del_sim, and add_sim), we compute a combined similarity which is done by taking the weighted average of the individual similarity scores. We have departed from the method used by Li Z., Yu Y., Wang T et al. [5], [?], as we have decided to give separate weights to all the similarities, unlike the previous method of combining del_sim and add_sim into a single similarity measure. Our method allows for potentially greater flexibility in the final score calculations. Also, Li Z., Yu Y., Wang T et al. [5], [?] employs a greedy search algorithm to find the most ideal weights for all the similarities whereas in our implementation we have given equal weights to all the scores, as their extra step did not get a significant gain on the equal weights' method.
We intend to investigate additional features in the future that can be used to identify duplicate PRs. Additionally, we want to explore the patterns of contributions that frequently lead to duplicate PRs so that we can suggest some methods for developers to avoid making duplicate contributions.

References

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

[2] Gousios, G., Pinzger, M., & van Deursen, A. (2014). 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.

[3] Steinmacher, I., Pinto, G., Wiese, I. S., & Gerosa, M. A. (2018). 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.

[4] Li, Z., Yin, G., Yu, Y., Wang, T., & Wang, H. (2017). 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.

[5] Li, Z., Yu, Y., Wang, T., Yin, G., & Wang, H. (2021). Detecting Duplicate Contributions in Pull-Based Model Combining Textual and Change Similarities. Journal of Computer Science and Technology, 36(1), 191-206. DOI: 10.1007/s11390-020-9935-1.

[6] Runeson, P., Alexandersson, M., & Nyholm, O. (2007). 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.

[7] Sun, C., Lo, D., Wang, X., Jiang, J., & Khoo, S. C. (2010). 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.

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

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