Identifying Redundancies in Fork-based Development

Luyao Ren
Peking University, China

Shurui Zhou, Christian Kästner
Carnegie Mellon University, USA

Andrzej Wąsowski
IT University of Copenhagen, Denmark

***Abstract*—Fork-based development is popular and easy to use, but makes it difficult to maintain an overview of the whole community when the number of forks increases. This may lead to redundant development where multiple developers are solving the same problem in parallel without being aware of each other. Redundant development wastes effort for both maintainers and developers. In this paper, we designed an approach to identify redundant code changes in forks as early as possible by extracting clues indicating similarities between code changes, and building a machine learning model to predict redundancies. We evaluated the effectiveness from both the maintainer's and the developer's perspectives. The result shows that we achieve 57–83% precision for detecting duplicate code changes from maintainer's perspective, and we could save developers' effort of 1.9–3.0 commits on average. Also, we show that our approach significantly outperforms existing state-of-art.**

***Index Terms*—Forking, Redundant Development, Natural Language Processing, Machine Learning**

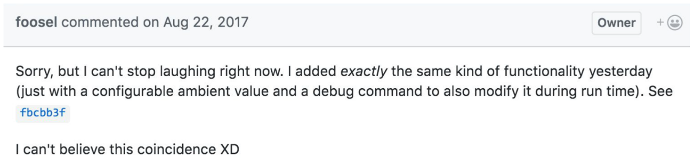
I. Introduction

Fork-based development allows developers to start development from an existing codebase, while having the freedom and independence to make any necessary modifications [1]–[4], and making it easy to merge changes from forks into upstream repository [5]. Although fork-based development is easy to use and popular in practice, it has downsides: When the number of forks of a project increases, it becomes *difficult to maintain an overview* of what happens in individual forks and thus of the project's scope and direction [6]. This further leads to additional problems, such as redundant development, which means developers may re-implement functionality already independently developed elsewhere.

For example, Fig. 1(a) shows two developers coincidentally working on the same functionality, where only one of the changes was integrated.[1] And Fig. 1(b) shows another case in which multiple developers submitted pull requests to solve the same problem.[2] And developers we interviewed previously by researchers also confirmed the problem as follows: *"I think there are a lot of people who have done work twice, and coded in completely different coding style[s]"* [6]. Gousios et al. [7] summarized nine reasons for rejected pull requests in 290 projects on GitHub, in which 23% were rejected due to redundant development (either parallel development or superseded other pull requests).

Existing works show that redundant development significantly increases the maintenance effort for maintainers [1], [8]. Specifically, Yu et al. manually studied pull requests from 26 popular projects on GitHub, and found that on average 2.5 reviewers participated in the review discussions of redundant pull requests and 5.2 review comments were generated before the duplicate relation is identified [9]. Also, Steinmacher et al. [10] analyzed quasi-contributors whose contributions were rejected from 21 GitHub projects, and found that one-third of the developers declared the nonacceptance *demotivated* them from continuing to contribute to the project. However, redundant development might not always be harmful (just as Nicolas et al. pointed out that duplicate bug report provides additional information, which could help to resolve bugs quicker [11]). We argue that pointing redundant development out, will help developers to collaborate, creating better solutions overall.

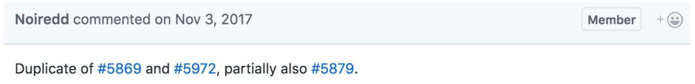
Facing this problem, our goal is (1) to help project maintainers to automatically identify redundant pull request in order to decrease workload of reviewing redundant code changes, and (2) to help developers detect redundant development as


foosel commented on Aug 22, 2017 Owner

Sorry, but I can't stop laughing right now. I added *exactly* the same kind of functionality yesterday (just with a configurable ambient value and a debug command to also modify it during run time). See fbcbb3f

I can't believe this coincidence XD

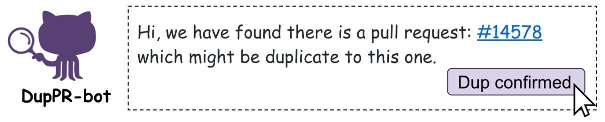
(a) Two developers work on same functionality


Noiredd commented on Nov 3, 2017 Member

Duplicate of #5869 and #5972, partially also #5879.

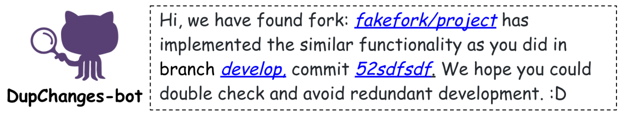
(b) Multiple developers work on same functionality

Fig. 1. *Pull requests rejected due to redundant development.*



Hi, we have found there is a pull request: #14578 which might be duplicate to this one.

(a) Helping maintainers to detect duplicate PRs



Hi, we have found fork: *fakefork/project* has implemented the similar functionality as you did in branch *develop*, commit *52sdfsdf*. We hope you could double check and avoid redundant development. :D

(b) Helping developers to detect early duplicate development

Fig. 2. *Mock up bot: Sending duplication warnings.*

[1] https://github.com/foosel/OctoPrint/pull/2087
[2] https://github.com/BVLC/caffe/pull/6029

early as possible by comparing code changes with other forks in order to eliminate potentially wasted effort and encourage developers to collaborate. Specifically, we would like to build a bot to send warnings when duplicate development is detected, which could assist maintainers' and contributors' work in open source projects. This idea also matches one of the desired features of a bot that contributors and maintainers want [12]. A mock up of the bot's output is shown in Fig. 2.

In this paper, we lay the technical foundation for such a bot by designing an approach to identify redundant code changes in fork-based development and evaluating its detection strategy. We first identify clues that indicate a pair of code changes might be similar by manually checking 45 duplicate pull request pairs. Then we design measures to calculate the similarity between changes for each clue. Finally, we treat the list of similarities as features to train a classifier in order to predict whether a pair of changes is a duplicate. Our dataset and the source code are available online.[3]

We evaluate the effectiveness of our approach from different perspectives, which align with the application scenarios introduced before for our bot: (1) helping project maintainers to identify redundant pull requests in order to decrease the code reviewing workload, (2) helping developers to identify redundant code changes implemented in other forks in order to save the development effort, and encouraging collaboration. In these scenarios, we prefer high precision and could live with moderate recall, because our goal is to save maintainers' and developers' effort instead of sending too many false warnings. Sadowski et al. found that if a tool wastes developer time with false positives and low-priority issues, developers will lose faith and ignore results [13]. We argue that as long as we show some duplicates without too much noise, we think it is a valuable addition. The result shows that our approach could achieve 57–83% precision for identifying duplicate pull requests from the maintainer's perspectives within a reasonable threshold range (details in Section IV). Also, our approach could help developers save 1.9–3.0 commits per pull request on average .

We also compared our approach to the state-of-the-art showing that we could outperform the state-of-the-art by 16–21% recall. Finally, we conducted a sensitive analysis to investigate how sensitive our approach is to different kinds of clues in the classifier. Although we did not evaluate the bot that we envision in this paper, our work lays the technical foundation for building a bot for the open-source community in the future.

To summarize, we contribute (a) an analysis of the redundant development problem, (b) an approach to automatically identify duplicate code changes using natural language processing and machine learning, (b) clues development for indicating redundant development, beyond just title and description, (c) evidence that our approach outperforms the state-of-the-art, and (d) anecdotal evidence of the usefulness of our approach from both the maintainer's and the developer's perspectives.

[3]https://github.com/FancyCoder0/INTRUDE

II. Identifying Clues to Detect Redundant Changes

In this paper, we refer to *changes* when developers make code changes in a project. There are different granularities of changes, such as pull requests, commits, or fine-grained code changes in the IDE (Integrated Development Evironment). In this section, we show how we extracted clues that indicate the similarity between pull requests. Although we use pull requests to demonstrate the problem, note that the examples and our approach are generalizable to different granularities of changes: For example, we could detect redundant pull requests for an upstream repository, redundant commits in branches or forks, or redundant code changes in IDEs (detailed application scenarios are described in Sec. III-C).

A. Motivating Examples

We present two examples of duplicate pull requests from GitHub to motivate the need for using both natural language and source code related information in redundant development detection.

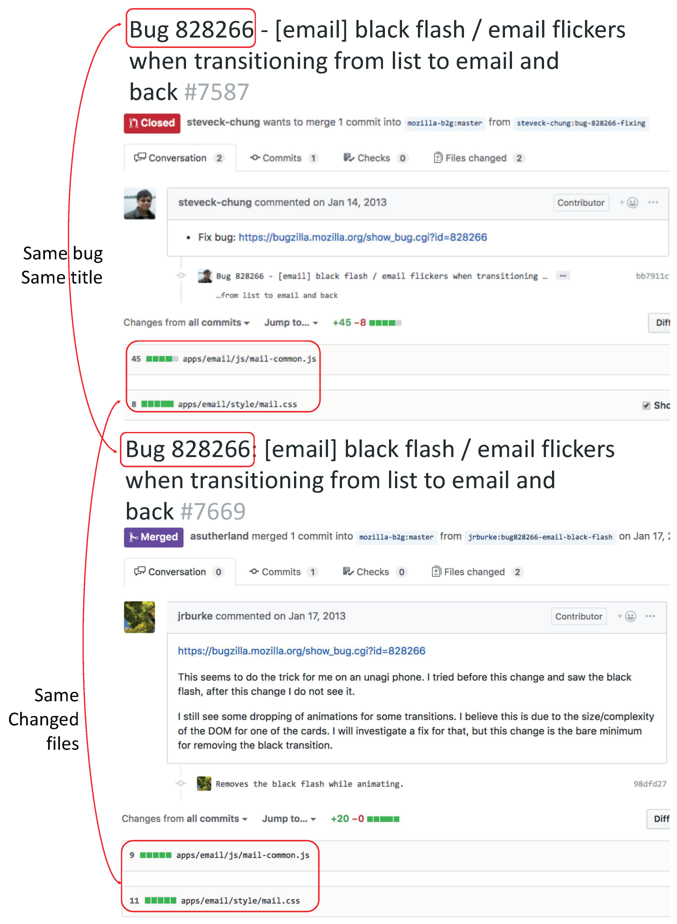
Case 1: Similar Text Description but Different Dode Changes: We show a pair of duplicate pull requests that are both fixing the bug 828266 in the *mozilla-b2g/gaia* repository in Fig. 3. Both titles contained the bug number, copied the title of the bug report, and both descriptions contain the link to the bug report. It is straightforward to detect duplication by comparing the referred bug number, and calculating the similarity of the title and the description. However, if we check the source code,[4] the solutions for fixing this bug are different, although they share two changed files. Maintainers would likely benefit from automatic detection of such duplicates, even if they don't refer to a common bug report. It could also prevent contributors from submitting reports that are duplicates, lowering the maintenance effort.

Case 2: Similar Code Changes but Different Text Description: We show a pair of duplicate pull requests that implement similar functionality for project *mozilla-b2g/gaia* in Fig. 4. Both titles share words like 'Restart(ing)' and 'B2G/b2g', and both did not include any other textual description beyond the title. Although one pull request mentioned the bug number, it is hard to tell whether these two pull requests are solving the same problem by comparing the titles. However, if we include the code change information, it is easier to find the common part of the two pull requests: They share two changed files, and the code changes are not identical but very similar except the comments at Line 8 and the code structure. Also, they changed the code in similar locations.

B. Clues for Duplicate Changes

We might consider existing techniques for clone detection [14], which aim to find pieces of code of high textual similarity on Type 1-3 clones in a system but not textual descriptions [15]. However, our goal is not to find code blocks originating from copy paste activities, but code changes

[4]https://github.com/mozilla-b2g/gaia/pull/7587/files
https://github.com/mozilla-b2g/gaia/pull/7669/files



Bug 828266 - [email] black flash / email flickers when transitioning from list to email and back #7587

Closed steveck-chung wants to merge 1 commit into `mozilla-b2g:master` from `steveck-chung:bug-828266-fixing`

Conversation 2 · Commits 1 · Checks 0 · Files changed 2

steveck-chung commented on Jan 14, 2013 · Contributor

- Fix bug: https://bugzilla.mozilla.org/show_bug.cgi?id=828266

```
Bug 828266 - [email] black flash / email flickers when transitioning …   bb7911c
…from list to email and back
```

Changes from all commits · Jump to… · +45 −8

```
45 apps/email/js/mail-common.js
8 apps/email/style/mail.css
```

Bug 828266: [email] black flash / email flickers when transitioning from list to email and back #7669

Merged asutherland merged 1 commit into `mozilla-b2g:master` from `jrburke:bug828266-email-black-flash` on Jan 17,

Conversation 0 · Commits 1 · Checks 0 · Files changed 2

jrburke commented on Jan 17, 2013 · Contributor

https://bugzilla.mozilla.org/show_bug.cgi?id=828266

This seems to do the trick for me on an unagi phone. I tried before this change and saw the black flash, after this change I do not see it.

I still see some dropping of animations for some transitions. I believe this is due to the size/complexity of the DOM for one of the cards. I will investigate a fix for that, but this change is the bare minimum for removing the black transition.

```
Removes the black flash while animating.   98dfd27
```

Changes from all commits · Jump to… · +20 −0

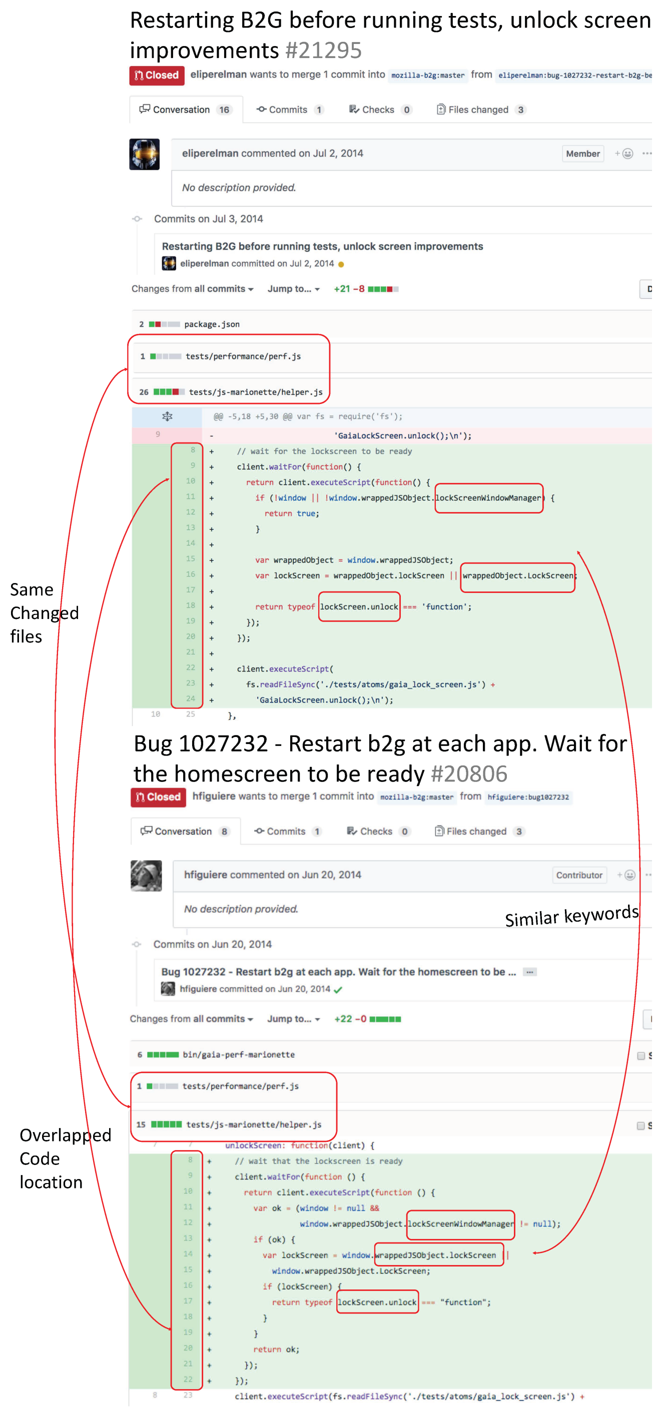
```
9 apps/email/js/mail-common.js
11 apps/email/style/mail.css
```

Fig. 3. *Duplicate PRs with similar text information*

written independently by different developers about the same functionality due to a lack of an overview in the fork-based development environment, which is conceptually close to the *Type-4 clones* [15], meaning two code changes have similar functionalities but are different in syntax.

Similarly, we have considered existing techniques for detecting duplicate bug reports [16]–[21], which compare textual descriptions but not source code (see detailed related work in Sec. VI). Different from the scenarios of clone detection and detecting duplicate bug reports, for detecting duplicate pull requests we have both textual description and source code, including information about changed files and code change locations. Thus we have additional information that we can exploit, and have opportunities to detect duplicate changes more precisely. Therefore, we seek inspiration from both lines of research, but tailor an approach to address the specific problem of detecting redundant code changes across programming languages and at scale.

To identify potential clues that might help us to detect if two changes are duplicates, we randomly sampled 45 pull requests that have been labeled as duplicate on GITHUB from five projects using (the March 2018 version of) GHTORRENT [22]. For each, we manually searched for the corresponding pull request that the current pull request is duplicate of. We then went through each pair of duplicate pull requests and inspected the text and code change information to extract clues indicating the potential duplication. The first two authors iteratively refined the clues until analyzing more duplicate pairs yielded no further clues.



Restarting B2G before running tests, unlock screen improvements #21295

Closed eliperelman wants to merge 1 commit into `mozilla-b2g:master` from `eliperelman:bug-1027232-restart-b2g-be`

Conversation 16 · Commits 1 · Checks 0 · Files changed 3

eliperelman commented on Jul 2, 2014 · Member

No description provided.

Commits on Jul 3, 2014

Restarting B2G before running tests, unlock screen improvements
eliperelman committed on Jul 2, 2014

Changes from all commits · Jump to… · +21 −8

```
2 package.json
1 tests/performance/perf.js
26 tests/js-marionette/helper.js
```

```
@@ -5,18 +5,30 @@ var fs = require('fs');
-                          'GaiaLockScreen.unlock();\n');
+    // wait for the lockscreen to be ready
+    client.waitFor(function() {
+      return client.executeScript(function() {
+        if (!window || !window.wrappedJSObject.lockScreenWindowManager) {
+          return true;
+        }
+
+        var wrappedObject = window.wrappedJSObject;
+        var lockScreen = wrappedObject.lockScreen || wrappedObject.LockScreen;
+
+        return typeof lockScreen.unlock === 'function';
+      });
+    });
+
+    client.executeScript(
+      fs.readFileSync('./tests/atoms/gaia_lock_screen.js') +
+        'GaiaLockScreen.unlock();\n');
   },
```

Bug 1027232 - Restart b2g at each app. Wait for the homescreen to be ready #20806

Closed hfiguiere wants to merge 1 commit into `mozilla-b2g:master` from `hfiguiere:bug1027232`

Conversation 8 · Commits 1 · Checks 0 · Files changed 3

hfiguiere commented on Jun 20, 2014 · Contributor

No description provided.

Commits on Jun 20, 2014

Bug 1027232 - Restart b2g at each app. Wait for the homescreen to be …
hfiguiere committed on Jun 20, 2014 ✓

Changes from all commits · Jump to… · +22 −0

```
6 bin/gaia-perf-marionette
1 tests/performance/perf.js
15 tests/js-marionette/helper.js
```

```
  unlockScreen: function(client) {
+   // wait that the lockscreen is ready
+   client.waitFor(function () {
+     return client.executeScript(function () {
+       var ok = (window != null &&
+                 window.wrappedJSObject.lockScreenWindowManager != null);
+       if (ok) {
+         var lockScreen = window.wrappedJSObject.lockScreen ||
+           window.wrappedJSObject.LockScreen;
+         if (lockScreen) {
+           return typeof lockScreen.unlock === "function";
+         }
+       }
+       return ok;
+     });
+   });
    client.executeScript(fs.readFileSync('./tests/atoms/gaia_lock_screen.js') +
```

Fig. 4. *Duplicate PRs with similar code change information*

Based on the results of the manual inspection, neither text information or code change information was always superior to the other. Text information represents the goal and summary of the changes by developers, while the corresponding code change information explicitly describes the behavior. Therefore, using both kinds of information can potentially detect redundant development precisely. Comparing to previous work [23], which detects duplicate pull requests by calculating the similarity only of title and description, our approach considers multiple facets of both the text information and the code change information.

We summarize the clues characterizing the content of a code change, which we will use to calculate change similarity:

- **Change description** is a summary of the code changes written in natural language. For example, a commit has commit message and a pull request contains title and description. Similar titles are a strong indicator that these two code changes are solving a similar problem. The description may contain detailed information of what kind of issue the current code changes are addressing, and how. However, textual description alone might be insufficient, as Fig. 4 shows.
- **References to an issue tracker** are a common practice in which developers explicitly link a code change to an existing issue or feature request in the issue tracker (as shown in Fig. 3). If both code changes reference the same issue, they are likely redundant, except for cases in which the two developers intended to have two solutions to further compare.
- **Patch content** is the differences of text changes in each file by running *'git diff'* command. The content could be source code written in different programming languages or comments from source code files, or could be plain text from non-code files. We found (when inspecting redundant development) that developers often share keywords when defining variables and functions, so that extracting representative keywords from each patch could help us identify redundant changes more precisely comparing to only using textual description (as shown in Fig. 4).
- **A list of changed files** contains all the changed files in the patch. We assume that if two patches share the same changed files, there is a higher chance that they are working on similar or related functionality. For example, in Fig. 4, both pull requests changed the *helper.js* and *perf.js* files.
- **Code change location** is a range of changed lines in the corresponding changed files. If the code change locations of two patches are overlapping, there is a potential that they are redundant. For example, in Fig. 4, two pull requests are both modifying *helper.js* lines 8–22, which increases the chance that the changes might be redundant.

III. Identifying Duplicate Changes in Forks

Our approach consists of two steps: (1) calculating the similarity between a pair of changes for each clue listed previously; (2) predicting the probability of two code changes being duplicate through a classification model using the similarities of each clue as features.

As our goal is to find duplicate development caused by unawareness of activities in other forks, we first need to filter out pull request pairs in which the authors are aware of the existence of another similar code change by checking the following criteria:

- Code changes are made by the same author; or
- Changes from different authors are linked on GitHub by authors, typically used when one is a following work of the other, or one is intended to supersede the other with a better solution; or
- The later pull request is modifying the code on top of the earlier merged pull request.

TABLE I
Clues and corresponding machine learning features

Clue	Feature for Classifier	Value
Change description	Title_similarity	[0,1]
	Description_similarity	[0,1]
Patch content	Patch_content_similarity	[0,1]
	Patch_content_similarity_on_overlapping_changed_files	[0,1]
Changed files list	Changed_files_similarity	[0,1]
	#Overlapping_changed_files	N
Location of code changes	Location_similarity	[0,1]
	Location_similarity_on_overlapping_changed_files	[0,1]
Reference to issue tracker	Reference_to_issue_tracker	{-1, 0, -1}

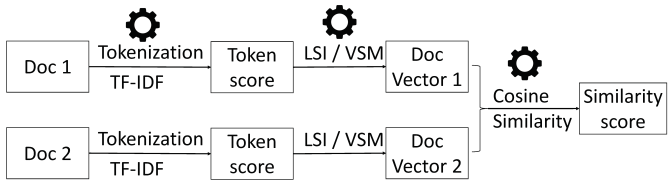


Fig. 5. Calculating similarity for description / patch content

A. Calculating Similarities for Each Clue

We calculate the similarity of each clue as features to train the machine learning model. Table I lists the features.

1) Change Description: To compare the similarity of the description of two changes, we first preprocess the text through tokenization and stemming. Then we use the well-known *Term Frequency Inverse Document Frequency* (TF-IDF) scoring technique to represent the importance of each token (its TF-IDF score), which increases proportionally to the number of times a word appears in the feature's corpus but is offset by the frequency of the word in the other feature's corpora [24]. The TF-IDF score reflects the importance of a token to a document; tokens with higher TF-IDF values better represent the content of the document. For example, in Fig. 4, the word *'LockScreen'* appears many times in both pull requests, but does not appear very often in the other parts of the project, so the *'LockScreen'* has a high TF-IDF score for these pull requests.

Next, we use Latent Semantic Indexing (LSI) [25] to calculate similarity between two groups of tokens, which is a standard natural language processing technique and has been proved to outperform other similar algorithms on textual artifacts in software engineering tasks [26], [27]. Last, we calculate the cosine similarity of two groups of tokens to get a similarity score (see Fig. 5).

2) Patch Content: We compute the token-based difference between the previous and current version of the file of each change, e.g. if original code is *func(argc1, argc2)*, and updated version is *func(argc1, argc2, argc3)*, we only extract *argc3* as the code change. We do not distinguish source code, in-line comments, and documentation files, we treat them uniformly as source code, but assume the largest portion is source code.

In order to make our approach programming languages independent, we treat all source code as text. So we use the same process as code change description to calculate the similarity, except we replace LSI by Vector Space Model (VSM), shown in Fig. 5, because VSM works better in case of exact matches while LSI retrieves relevant documents based on the semantic similarity [27].

However, this measure has limitations. When a pull request is duplicate with only a subset of code changes in another pull request, the similarity between these two is small, which makes it harder to detect duplicate code changes. During the process of manually inspecting duplicate pull request pairs (Section II-B), we found there are 28.5% pairs where one pull request is five times larger than the other at the file level. To solve this problem, we add another feature as the similarity of patch content only on overlapping files.

3) Changed Files List: We operationalize the similarity of two lists of files into computing the overlap between two sets of files by using Jaccard similarity coefficient: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ (A and B are two sets of elements). The more overlapping files two changes have, the more similar they are. As Fig. 6 shows, PR1 and PR2 have modified 2 files each, and both of them modified *File1*, so the similarity of the two lists of files is $1/3$.

Again, in case that one pull request is much bigger than the other in terms of changed files, which leads to a small ratio of overlapping files, we add a feature defined as the number of overlapping files.

4) Location of Code Changes: We calculate the similarity of code change location by comparing the size of overlapping code blocks between a pair of changes. The more overlapping blocks they have, the more similar these two changes are. In Fig. 6, block *A* overlaps with block *D* in *File1*. We define the *Location similarity* as the length of overlapping blocks divided by length of all the blocks.

Similar to our previous concern, in order to catch redundant changes between big and small size of patches in file level, we define a feature of similarity of code change location for only overlapping files. For example, in Fig. 6, block *A*, *B* and *D* belong to *File1*, but block *C* and *E* belong to different files, so the measure of *Location similarity on overlapping changed files* only consider the length of block *A*, *B* and *D*.

5) Reference to Issue Tracker: Based on our observation, we found that if two changes link to the same issue, they are very likely duplicates, while, if they link to different issues, our intuition is that the chance of the changes to be duplicate is very low. So we defined a feature as reference to issue tracker. For projects using the GITHUB issue tracker, we use the GITHUB API to extract the issue link, and, for projects using other issue tracking systems (as Fig. 3 shows), we parse the text for occurrences from a list of patterns, such as 'BUG', 'ISSUE','FR' (short for feature request). We define three possible values for this feature: If they link to the same issue, the value is 1; if they link to different issues, the value is -1; otherwise it is 0.

B. Predicting Duplicate Changes Using Machine Learning

The goal is to classify a pair of changes as duplicate or not. We want to aggregate these nine features and make a decision. Since it is not obvious how to aggregate and weigh the features, we use machine learning to train a model. There are many studies addressing the use of different machine learning algorithms for classification tasks, such as support vector machines, AdaBoost, logistic regressions, neural network, decision Trees, random forest, and k-Nearest Neighbor [28]. In this study, in order to assess the performance of these techniques for our redundancy detection problem, we have conducted a preliminary experimental study. More specifically, we compared the performance of six algorithms based on a small set of subject projects. We observed that the best results were obtained when AdaBoost [29] was used. Therefore, we focused our efforts only on AdaBoost, but other techniques could be easily substituted. Since the output of AdaBoost algorithm is a probability score whether two changes are duplicate, we set a threshold and report two changes as duplicate when the probability score is above the threshold.

C. Application Scenarios

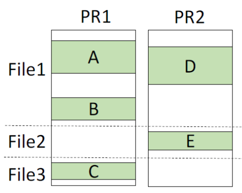
Our approach could be applied to different scenarios to help different users. Primarily, we envision a bot to monitor the incoming pull request in a repository and compare the new pull request with all the existing pull requests in order to *help project maintainers* to decrease their workload. The bot would automatically send warnings when a duplication is detected (see Fig. 2(a)).

Additionally, we envision a bot to monitor forks and branches, and compare the commits with other forks and with existing pull requests, in order to *help developers* detect early duplicate development. Researchers have found that developers think it is worth spending time checking for existing work to avoid redundant development, but once they start coding a pull request, they never or rarely communicate the intended changes to the core team [30]. We believe it is useful to inform developers when potentially duplicate implementation is happening in other forks, and encourage developers to collaborate as early as possible instead of competing after submitting the pull request. A bot would be a solution (see Fig. 2(b)).

Also, we could build a plug-in for a development IDE, so we could detect redundant development in real time.

IV. EVALUATION

We evaluate the effectiveness of our approach from different perspectives, which align with the application scenarios intro-



$$\textit{Changed\_files\_similarity} = \frac{\{File\ 1\}}{\{File\ 1, File2, File3\}} = \frac{1}{3}$$

$$\textit{\#Overlapping\_changed\_files} = 1$$

$$\textit{Location\_similarity} = \frac{loc(A) + loc(D)}{loc(A) + loc(B) + loc(C) + loc(D) + loc(E)}$$

$$\textit{Location\_similarity\_on\_overlapping\_files} = \frac{loc(A) + loc(D)}{loc(A) + loc(B) + loc(D)}$$

Fig. 6. *Similarity of changed files and similarity of code change location (loc: Lines of code)*

duced in Sec. III-C: (1) helping project maintainers to identify redundant pull requests in order to decrease the code reviewing workload, (2) helping developers to identify redundant code changes implemented in other forks in order to save development effort. To demonstrate the benefit of incorportating multiple clues, we compared our approach to the state-of-the-art that uses textual comparison only. Finally, beyond just demonstrating that our specific implementation works, we explore the relative importance of our clues with a sensitivity analysis, which can guide other implementations and future optimizations. Thus, we derived four research questions:

- *RQ1: How accurate is our approach to help maintainers identify redundant pull requests?*
- *RQ2: How much effort could our approach save for developers in terms of commits?*
- *RQ3: How good is our approach identifying redundant pull requests comparing to the state-of-the-art?*
- *RQ4: Which clues are important to detect duplicate changes?*

A. Dataset

To evaluate approaches, an established ground truth is needed—a reliable data set defining which changes are duplicate. In our experiment, we used an established corpus named DupPR, which contains 2323 pairs of duplicate pull requests from 26 popular repositories on GitHub [9] (Table. II). We picked half of the DupPR dataset, which contains 1174 pairs of duplicate PRs in twelve repositories as the *positive samples* in the training dataset (highlighted) to calibrate our classifier (see Sec. III-B), and the remaining 1149 pairs from 14 repositories as testing dataset.

While this dataset provides examples of duplicate pull requests, it does not provide negative cases of pull request pairs that are not redundant (which are much more common in practice [7]). To that end, we randomly sampled pairs of merged pull requests from the same repositories, as we assume that if two pull requests are both merged, they are most likely not duplicate. Overall, we collected 100,000 negative samples from the same projects, 50,000 for training, and 50,000 for testing.

TABLE II
SUBJECT PROJECTS AND THEIR DUPLICATE PR PAIRS

Repository	#Forks	#PRs	#DupPR pairs	Language
symfony/symfony	6446	16920	216	PHP
kubernetes/kubernetes	14701	38500	213	Go
twbs/bootstrap	62492	8984	127	CSS
rust-lang/rust	5222	26497	107	Rust
nodejs/node	11538	12828	104	JavaScript
symfony/symfony-docs	3684	7654	100	PHP
scikit-learn/scikit-learn	15315	6116	68	Python
zendframework/zendframework	2937	5632	53	PHP
servo/servo	1966	12761	52	Rust
pandas-dev/pandas	6590	9112	49	Python
saltstack/salt	4325	29659	47	Python
mozilla-b2g/gaia	2426	31577	38	JavaScript
rails/rails	16602	21751	199	Ruby
joomla/joomla-cms	2768	13974	152	PHP
angular/angular.js	29025	7773	112	JavaScript
ceph/ceph	2683	24456	104	C++
ansible/ansible	13047	24348	103	Python
facebook/react	20225	6978	74	JavaScript
elastic/elasticsearch	11859	15364	62	Java
docker/docker	14732	18837	61	Go
cocos2d/cocos2d-x	6587	14736	57	C++
django/django	15821	10178	55	Python
hashicorp/terraform	4160	8078	52	Go
emberjs/ember.js	4041	7555	46	JavaScript
JuliaLang/julia	3002	14556	42	Julia
dotnet/corefx	4369	17663	30	C#

* The upper half projects (highlighted) are used as training dataset, and lower half projects are used as testing dataset.

B. Analysis and Results

RQ1: How accurate is our approach to help maintainers identify redundant contributions?

In our main scenario, we would like to notify maintainers when a new pull request is duplicate with existing pull requests, in order to decrease their workload of reviewing redundant code changes (e.g., a bot for duplicate pull request monitoring). So we simulate the pull request history of a given repository, compare the newest pull request with all the prior pull requests, and use our classifier to detect duplication: If we detect duplication, we report the corresponding pull request number.

Research method: We use the evaluation set of the DupPR dataset as ground truth. However, based on our manual inspection, we found the dataset is incomplete, which means it does not cover all the duplicate pull requests for each project. This leads to several problems. First, when our approach detects a duplication but the DupPR does not cover the case, the precision value is distorted. To address this problem, we decided to manually check the correctness of the duplication warnings; in another word, we complement DupPR with manual checking result as ground truth (shown in Table III). Second, it is unrealistic to manually identify all the missing pull request pairs in each repository, so we decided

TABLE III
RQ1: Simulating PR history

PR history	Our_result	DupPR	Manual checking	Warning correctness
1	-	?		
2	-	?		
3	-	?		
4	2	2		✓
5	-	?		
6	5	?	5	✓
7	-	?		
8	-	6		
9	4	?	✗	✗

Ground Truth

to randomly sample 400 pull requests from each repository for computing precision.

Table III illustrates our replay process. The *PR_history* column shows the sequence of the coming PRs, *our_result* column is our prediction result, for example, we predict 4 is duplicate with 2, and 6 is duplicate with 5, and 9 is duplicate with 4; *DupPR* column shows that 2 and 4 are duplicate, and 8 and 6 are duplicate; the *manual_checking* column shows that the first 2 authors manually checked and confirmed 5 and 6 are duplicate, and 9 and 4 are not duplicate. The warning correctness shows that the precision of this example is $2/3$.

For calculating recall, we use a different dataset because even for 400 pull requests per project, we need to manually check a large number of pull requests in order to find all the duplicate pull request pairs, which is very labor intensive. Thus, we only use the evaluation section of the DupPR dataset (lower half of Table. II) to run the experiment, which contains 1149 pairs of confirmed duplicate pull requests from 14 repositories.

Result: Figure 7 shows the precision and recall at different thresholds. We argue that within a reasonable threshold range of 0.5925–0.62, our approach achieved 57-83% precision and 10-22% recall. After some experiments, we pick a reasonable default threshold of 0.6175, where our approach achieves 83% precision and 11% recall (dash line in Fig. 7). Tables IV and V show the corresponding precision and recall for each the project separately at the default threshold.

We did not calculate the precision for lower threshold because when the threshold gets lower, the manual check effort becomes infeasible. Here we argue that a higher precision is more important than recall in this scenario, because our goal is to decrease the workload of maintainer, so that we hope all the warnings that we send to them are mostly correct, otherwise, we will waste their time to check false positives. In the future, it would be interesting to interview stakeholders or design experiment with real intervention to see acceptable levels about the acceptance rate of false positives in the real scenario, so we could allow users to set the threshold for different tolerance rate of false positives.

RQ2: How much effort could our approach save for developers in terms of commits?

TABLE IV
RQ1, precision at default threshold

Repository	TP / TP + FP	Precision
django/django	5 / 5	100%
facebook/react	3 / 3	100%
hashicorp/terraform	3 / 3	100%
ansible/ansible	2 / 2	100%
ceph/ceph	2 / 2	100%
joomla/joomla-cms	2 / 2	100%
docker/docker	1 / 1	100%
cocos2d/cocos2d-x	6 / 7	86%
rails/rails	5 / 6	83%
angular/angular.js	3 / 4	75%
dotnet/corefx	2 / 3	67%
emberjs/ember.js	2 / 4	50%
elastic/elasticsearch	1 / 2	50%
JuliaLang/julia	1 / 2	50%
Overall	38 / 46	83%

TABLE V
RQ1, recall at default threshold

Repository	TP / TP + FN	Recall
ceph/ceph	31 / 104	30%
django/django	14 / 55	25%
hashicorp/terraform	8 / 52	15%
elastic/elasticsearch	7 / 62	11%
cocos2d/cocos2d-x	6 / 57	11%
rails/rails	20 / 199	10%
docker/docker	6 / 61	10%
angular/angular.js	11 / 112	10%
joomla/joomla-cms	12 / 152	8%
ansible/ansible	7 / 103	7%
emberjs/ember.js	3 / 46	7%
facebook/react	3 / 74	4%
JuliaLang/julia	0 / 42	0%
dotnet/corefx	0 / 30	0%
Overall	128 / 1149	11%

The second scenario focuses on developers. We would like to detect redundant development as early as possible to help reduce the development effort. A hypothetical bot monitors forks and branches and compares un-merged code changes in forks against pending pull requests and against code changes in other forks.

Research Method: To simulate this scenario, we replay the commit history of a pair of duplicate pull requests. As shown in Fig. 8, when there is a new commit submitted, we use the trained classifier to predict if the two groups of existing commits from each pull request are duplicate.

We use the same testing dataset as described in Sec. IV-A). We calculate the number of commits to represent the saved development effort because number of commits and lines of added/modified code are highly correlated [31]. Since we are checking if our approach could save developers' effort in terms of commits, we first need to filter out pull request pairs that have no chance to predict the duplication early. For instance, two pull requests both contain only one commit, or the later pull request has only one commit. After this filtering, the final dataset contains 408 positive samples and 13,365 negative samples.

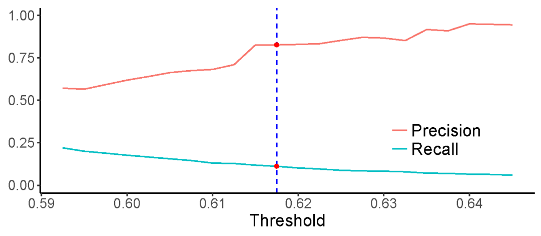


Fig. 7. *RQ1: Precision & Recall at different thresholds, dashed line shows the default threshold*

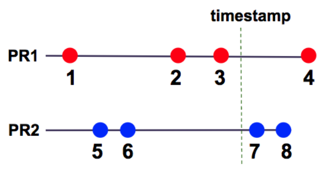


Fig. 8. *Simulating commit history of a pair of PRs. If PR1 is duplicate with PR2, we first compare commit 1 and 5, if we do not detect duplication, then we compare 1 and (5,6), and so on. If we detect duplication when comparing (1, 2, 3) with (5, 6), then we conclude that we could save developers of PR1 one commit of effort or PR2 two commits.*

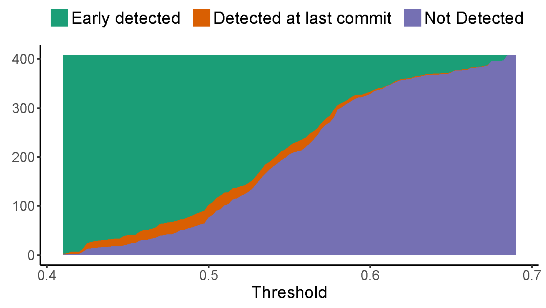
Result: Based on the classification result, we group the pairs of duplicate pull requests (positive dataset) into three groups: Duplicate detected early, duplication detected in the last commit, and duplication not detected. In addition, we check how much noise our approach introduces, we calculate the number of false positive cases among all the 13,365 negative cases, and get the false positive rate.

We argue that within a reasonable threshold range of 0.52–0.56, our approach achieved 46–71% recall (see Fig. IV-B), with 0.07–0.5% false positive rate (see Fig. IV-B). Also, we could save 1.9–3.0 commits per pull request within the same threshold range (see Fig. IV-B).[5]

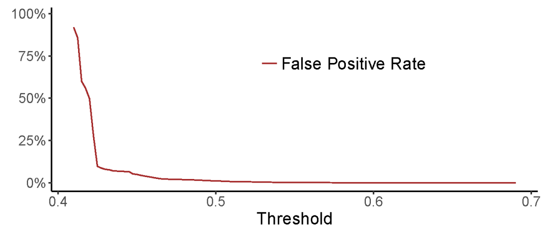
RQ3: How good is our approach identifying redundant PRs comparing to the state-of-the-art?

Research Method: Yu et al. proposed an approach to detect duplicate pull requests with the same scenario as we described in RQ1 [23], that is, for a given pull request, identifying duplicate pull requests among other history pull requests. However, there are three main differences between their approach and ours: (1) they calculate the textual similarity between a pair of pull requests only on title and description, while we consider patch content, changed files, code change location, and reference to issue tracking system when calculating similarities (9 features) (see Sec. II-B); (2) their approach returns top-K duplicate pull requests among existing pull requests by ranking them by arithmetic average of the two similarity values, while our approach reports duplication warnings only when the similarity between two pull requests is above a threshold; (3) they get the similarity of two pull requests by calculating the arithmetic average of the two

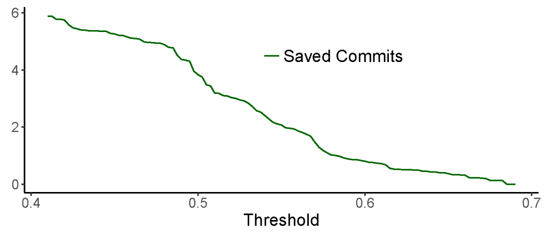
[5]Comparing to RQ1 scenario, we set a lower default threshold in this case, and we argue that developers of forks are more willing to inspect activities in other forks [6], [30]. But again, in the future, we would give developers the flexibility to decide how many notifications they would like to receive.



(a) Distribution for prediction result on positive data (duplicate PR pairs)



(b) False positive rate



(c) Saved #commits per pull request

Fig. 9. *RQ2: Can we detect duplication early, how much effort could we save in terms of commits, and corresponding false positive rate at different threshold*

similarity values, while we adopt a machine learning algorithm to aggregate nine features.

We argue that for this scenario, our goal is to decrease maintainers' workload for reviewing duplicate pull requests, instead of assuming maintainers periodically to go through a list of potential duplicate pull request pairs. In our solution, we therefore also prefer high precision over recall. But in order to make our approach comparable, we reproduced their experimental setup and reimplemented their approach, even though it does not align with our goal.

Research Method: We follow their evaluation process by computing *recall-rate@k*, as per the following definition:

$$\textit{recall-rate}@k = \frac{N_{\text{detected}}}{N_{\text{total}}} \quad (1)$$

N_{detected} is the number of pull requests whose corresponding duplicate one is detected in the candidate list of top-k pull requests, N_{total} is the total number of pairs of duplicate pull requests for testing. It is the ratio of the number of correctly retrieved duplicates divided by the total number of actual duplicates. The value of k may vary from 1 to 30, meaning the potential k duplicates.

Result: As shown in Fig. 10, our approach achieves better results than the state-of-the-art by 16–21% *recall-rate*@k.

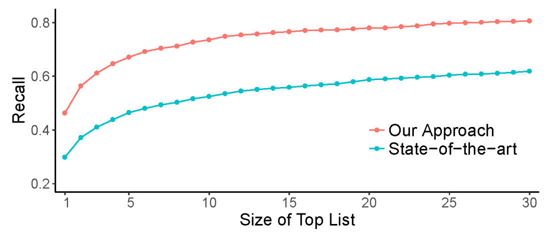


Fig. 10. *RQ3: How good is our approach identifying redundant PRs comparing to the state-of-the-art?*

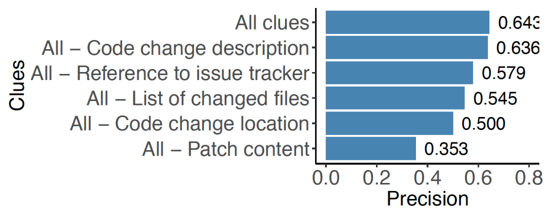


Fig. 11. *RQ4: Sensitive analysis, removing one clue at a time. Precision at recall fixed at 20%*

The reason is that we considered more features and code change information when comparing the similarity between two changes. Also, we use a machine learning technique to classify based on features.

RQ4: Which clues are important to detect duplicate changes?

We aim to understand the clues that influence the effectiveness of our approach. Specifically, we investigate how sensitive our approach is to different kinds of clues in the classifier.

Research Method: We design this experiment on the same scenario as RQ1, which is helping maintainers to detect duplication by comparing new pull request with existing pull requests from each project as testing dataset (see Sec. IV-B). However, we used a smaller testing dataset of 60 randomly sampled pull requests, because for calculating precision we need to manually check the detected duplicate pull request pairs every time, which is labor intensive.

We trained the classifier five times, and we removed one clue each time. So the combined absolute values of features change every time in the classifier's sum. This means that using a single cut-off thresholds for all the rounds does not make sense – the measured objective function changes all the time. Therefore, we pick the threshold for each model such that it produces a given recall (20%) and compare precision at that threshold.

Result: Fig. 11 shows that when considering all the clues, the precision is the highest (64.3%). Removing the clue of patch content affects precision the most, which leads to 35.3% precision, and removing the text description has the least effect (63.6% precision). The result shows that patch content is the most important clue in our classifier, which likely explains the improvement in RQ3 as well. In the future, we could also check the sensitivity for each feature, or different combinations.

V. Threats to Validity

Regarding external validity, our experimental results are based on an existing corpus, which focuses on some of the popular open source projects on GitHub, which covers different domains and programing languages. However, one needs to be careful when generalizing our approach to other open source projects. Also, for our sensitivity analysis, we only ran it on the experimental setup, which means the conclusion might not generalize to RQ2.

Regarding construct validity, the corpus of DupPR was validated with manual checking [9], but we found the dataset to be incomplete. So we manually checked the duplicate pull request pairs that our approach identified by the first and second author independently. We are not experts on these projects, so we may misclassify pairs despite careful analysis. All inconsistencies identified by the two authors were discussed until consensus was reached.

In this paper, we are using changes to represent all kinds of code changes in general, and we use pull requests to demonstrate the problem. While we believe that our approach can be generalized to other kinds of granularities of changes, future research is needed to confirm.

VI. Related Work

A. Duplicate Pull Request Detection

Li et al. proposed an approach to detect duplicate pull requests by calculating the similarity on title and description [23], which we used as baseline to compare with (Sec IV-B). Different from their approach, we considered both textual and source code information, used a machine learning technique classify duplicates, and evaluated our approach in scenarios from the maintainer's and developer's perspectives. Later, Yu et al. created a dataset of duplicate pull request pairs [9], which we have used as part of our ground truth data.

Zhang et al. analyzed pull requests with a different goal: They focused on competing pull requests that edited the same lines of code, which would potentially lead to merge conflicts [32], which is roughly in line with the merge conflicts prediction tool Palantír [33] and Crystal [34]. Even though we also look at change location, we focus on a broader picture: We detect redundant (not only conflicting) work, and encourage early collaboration. In the future, we could report conflicts since we already collect the corresponding data as one feature in the machine learning model.

B. Duplicate Bug Report Detection

We focus on detecting duplicate pull requests, but there have been other techniques to detect other forms of duplicate submissions, including bug reports [16]–[18], [35], [36] and StackOverflow questions [37]. On the surface, they are similar because they compare text information, but the types of text information is different. Zhang et al. [38] summarized related work on duplicate-bug-report detection. Basically, existing approaches are using information retrieval to parse two types of resource separately: One is natural-language based [35], [36] and the other is execution information based [16], [18].

Further, Wang et al. [17] combined execution information with natural language information to improve the precision of the detection. Beyond information retrieval, duplicate bug-report classification [19] and the Learn To Rank approach were also used in duplicate detection [20], [21]. In contrast, our approach focuses on duplicate implementations for features or bug fixing, where we can take the source code into account.

C. Clone Detection

Our work is similar to the scenario of detecting *Type-4* clones: Two or more code fragments that perform the same computation but are implemented by different syntactic variants [15]. We, instead, focus on detecting work of independent developers on the same feature or bug fix, which is a different, somewhat more relaxed and broader problem. Researchers investigated different approaches to identify code clones [14]. There are a few approaches attempting to detect pure *Type-4* clones [39]–[41], but these techniques have been implemented to only detect C clones, which are programming language specific. Recently, researchers started to use machine learning approaches to detect clones [42]–[45] including *Type-4* clones. Different from the scenario we proposed in this paper, clone detection uses source code only, while we also consider textual description of the changes. So we customize the clone detection approaches and applied them in a different scenario, that is identifying redundant changes in forks. As a future direction, it would be interesting to integrate and evaluate more sophisticated clone detection mechanisms as a similarity measure for the patch content clue in our approach.

D. Transparency in Fork-based Development

Redundant development is caused by lacking an overview and not enough transparency in fork-based development. In current modern social coding platforms, transparency has been shown to be essential for decision making [46], [47]. Visible clues, such as developer activities or project popularity, influence decision making and reputation in an ecosystem. However, with the increase of forks, it is hard to maintain an overview. To solve this problem, in prior work, we designed an approach to generate a better overview of the forks in a community by analyzing unmerged code changes in forks with the aim of reducing inefficiencies in the development process [6]. In this paper, we solve a concrete problem caused by a lack of an overview, which is predicting redundant development as early as possible.

VII. Discussion

Usefulness: On the path toward building a bot to detect duplicate development, there are more open questions. We have confirmed the effectiveness, but in the following we discuss the broader picture of whether our approach is useful in practice and how to achieve the goal.

We have opportunistically introduced our prototype to some open-source developers with public email addresses on their GitHub profiles, and other discussions in person at conferences and received positive feedback. For example, one of the maintainers from *cocos2d-x* project commented that "*this is quite useful for all Open Source Project on GitHub, especially for that with lots of Issues and PRs*." And another maintainer from *hashicorp/terraform* also replied "*this would definitely help someone like us managing our large community*."

We had also left comments to some of the pull requests that we detected as duplicate on GitHub in order to explore the reaction of developers. We commented on 23 pull requests from 8 repositories, and 16 cases were confirmed as duplication, while 3 cases were confirmed as not duplicate, and we have not heard from the rest. We have received much positive feedback on both the importance of the problem and our prediction result. Interestingly, for the three false positive cases, even for developers of the projects it was not straightforward to decide whether they were duplicate or not, due to differences in solutions and coding style. We believe that although our tool does report false positives, it might be still valuable and worth to bring developers' attention to discuss the potentially duplicate or closely related code changes together.

In the future, we plan to implement a bot as sketched in Fig. 2 and design a systematic user study to evaluate the usefulness of our approach.

Detecting Duplicate Issues: We received feedback from project maintainers that duplication does not only happen in code changes, but appears also in issue trackers, forums, and so on. We have seen issues duplicating a pull request, as some developers directly describe the issue in the pull request that solves it, missing the fact that the issue has already been reported elsewhere. This makes detecting redundant development event harder. However, since our approach is natural-language-based, we believe that with some adjustment, we could apply our approach to different scenarios.

False Negatives: Because the reported recall is fairly low, we manually investigated some false negative cases, *i.e.*, duplicates that we could not detect. We found that for a large number of duplicate pull request pairs, none of our clues work. But there are spaces we could improve, such as hard-coding specific cases, improving the training dataset, adding more features, or may be a domain vocabulary is needed to improve the result. We argue that even if we can only detect duplicate changes with a low recall of 22%, it is still valuable for developers.

VIII. Conclusion

We have presented an approach to identify redundant code changes in forks as early as possible by extracting clues of similarity between code changes, and building a machine learning model to predict redundancies. We evaluated the effectiveness from both the maintainer's and the developer's perspectives. The result shows that we achieve 57–83% precision for detecting duplicate code changes from maintainer's perspective, and we could save developers' effort of 1.9–3.0 commits on average. Also, we show that our approach significantly outperforms existing state-of-art and provide anecdotal evidence of the usefulness of our approach from both maintainer's and developer's perspectives.

References

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[2] J. Bitzer and P. J. Schröder, "The impact of entry and competition by open source software on innovation activity," *The economics of open source software development*, pp. 219–245, 2006.

[3] N. A. Ernst, S. Easterbrook, and J. Mylopoulos, "Code forking in open-source software: a requirements perspective," *arXiv preprint arXiv:1004.2889*, 2010.

[4] G. R. Vetter, "Open source licensing and scattering opportunism in software standards," *BCL Rev.*, vol. 48, p. 225, 2007.

[5] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 2012, pp. 1277–1286.

[6] S. Zhou, Ş. Stănciulescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner, "Identifying features in forks," in *Proc. Int'l Conf. Software Engineering (ICSE)*. New York, NY: ACM Press, 5 2018, pp. 105–116.

[7] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[8] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, 2015, pp. 151–160.

[9] Y. Yu, Z. Li, G. Yin, T. Wang, and H. Wang, "A dataset of duplicate pull-requests in Github," in *Proc. Int'l Conf. on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2018, pp. 22–25.

[10] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open source software projects," in *Proc. Int'l Conf. Software Engineering (ICSE)*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 256–266.

[11] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 2008, pp. 337–345.

[12] M. Wessel, B. M. de Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa, "The power of bots: Characterizing and understanding bots in OSS projects," *Proc. ACM Hum.-Comput. Interact.*, vol. 2, no. CSCW, p. 182, Nov. 2018.

[13] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, pp. 58–66, 2018.

[14] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng. (TSE)*, 2007.

[15] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, pp. 470–495, 2009.

[16] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2003, pp. 465–475.

[17] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 461–470.

[18] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei, "JDF: detecting duplicate bug reports in jazz," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 315–316.

[19] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 52–61.

[20] K. Liu, H. B. K. Tan, and H. Zhang, "Has this bug been reported?" in *Proc. Working Conf. Reverse Engineering (WCRE)*, Oct 2013, pp. 82–91.

[21] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM '12. New York, NY, USA: ACM, 2012, pp. 852–861.

[22] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th working conference on mining software repositories*. IEEE Press, 2013, pp. 233–236.

[23] Z. Li, G. Yin, Y. Yu, T. Wang, and H. Wang, "Detecting duplicate pull-requests in Github," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 2017, p. 20.

[24] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[25] T. K. Landauer and S. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge," *Psychological Review*, vol. 104, no. 2, pp. 211–240, 1997.

[26] M. M. Rahman, S. Chakraborty, G. E. Kaiser, and B. Ray, "A case study on the impact of similarity measure on information retrieval based software engineering tasks," *CoRR*, 2018.

[27] I. Chawla and S. K. Singh, "Performance evaluation of vsm and lsi models to determine bug reports similarity," in *2013 Sixth International Conference on Contemporary Computing (IC3)*, 2013, pp. 375–380.

[28] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*. Springer, 2013, vol. 112.

[29] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, pp. 119–139, 1997.

[30] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2016, pp. 285–296.

[31] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: Multitasking on GitHub projects," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2016, pp. 994–1005.

[32] Z. Xin, C. Yang, G. Yongfeng, Z. Weiqin, X. Xiaoyuan, J. Xiangyang, and X. Jifeng, "How do multiple pull requests change the same code: A study of competing pull requests in Github," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, 2018, p. 12.

[33] A. Sarma, D. F. Redmiles, and A. van der Hoek, "Palantír: Early detection of development conflicts arising from parallel code changes," *IEEE Trans. Softw. Eng. (TSE)*, vol. 38, no. 4, pp. 889–908, 2012.

[34] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011.

[35] L. Hiew, "Assisted detection of duplicate bug reports," 2006.

[36] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. Int'l Conf. Software Engineering (ICSE)*, May 2007, pp. 499–510.

[37] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions in stack overflow," in *Proc. Int'l Conf. on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2016, pp. 402–412.

[38] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, p. 1–24.

[39] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008.

[40] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 96–105.

[41] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 81–92.

[42] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. Int'l Conf. on Mining Software Repositories (MSR)*, 2018, pp. 542–553.

[43] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. New York, NY, USA: ACM, 2018, pp. 354–365.

[44] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on*. IEEE, 2016, pp. 1024–1028.

[45] R. Tekchandani, R. K. Bhatia, and M. Singh, "Semantic code clone detection using parse trees and grammar recovery," in *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, 2013, pp. 41–46.

[46] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in GitHub: Transparency and collaboration in an open software repository,” in *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. New York: ACM Press, 2012, pp. 1277–1286.

[47] ——, “Leveraging transparency,” *IEEE software*, vol. 30, no. 1, pp. 37–43, 2013.