Identifying Supplementary Bug-fix Commits

Tao Ji Jinkun Pan Liqian Chen Xiaoguang Mao
College of Computer
National University of Defense Technology, Changsha 410073, China
Email: taoji@nudt.edu.cn, pan_jin_kun@163.com, {lqchen, xgmao}@nudt.edu.cn

***Abstract*—Real-world bugs and the bug-fix activities are essential in many fields such as bug prediction and automatic program repair. Identifying bug-fix commits from version histories has received much recent attention. Linking commits to bug reports and analyzing the commits individually are common practice. However, considering the one-to-many relationship between the bug report and the bug-fix commits, analyzing commits individually will miss the relevance between commits, since several commits might fix the same bug together. In addition, some supplementary bug-fix commits which supplement or correct the identified bug-fix commit may be neglected. For empirical studies on bug-fix commits, it is important to study all the relevant commits as a whole, otherwise we will fail to understand the complete real bug-fix activities. In this paper, we investigate the relevance between bug-fix commits that are linked to the same bug-fix pull request, and utilize machine learning techniques to determine supplementary bug-fix commits for an identified bug-fix commit. Experimental results show that there indeed exist supplementary bug-fix commits (i.e., 19.8% on average) that are neglected when analyzing commits individually. The performance of our tool SupBCFinder is much better than that of using a sliding window of one hour and that of analyzing the local change. Moreover, inspired by our learning-based approach and extracted features, we propose one effective heuristic as an alternative for the cases when there are not enough pull requests for training.**

I. Introduction

Mining software repositories has provided valuable insights for developing and improving software. In the fields of bug prediction and automatic program repair, real-world bugs and the fixes are desired. For example, one can use historical bugs to build prediction models for predicting if the software contains bugs that have appeared in history and to fix them by reusing existing fix operations. However, the biases in bug-fix datasets may affect the results of such studies [1]. Much effort has been devoted to identifying bug-fix commits precisely and completely.

In general, there are two main categories of techniques for identifying bug-fix commits: analyzing the bug-fix characteristics of commits (without considering bug reports) [2] [3] and analyzing the links between commits and bug reports [4] [5]. Considering that the links between commits and bug reports may be missing and developers might fix bugs that are not reported in bug tracking systems, the remaining commits that are not linked to bug reports still need to be analyzed. To conduct a comprehensive study, these two categories of techniques are often used together to identify bug-fix commits [6] [7]. For example, Zhong and Su [6] first pick out those commits whose messages contain the numbers of bug reports, and then identify more bug-fix commits by matching keywords such as "bug" and "fix". According to their statistics on bug reports and bug-fix commits, one reported bug may have more than one bug-fix commit. It means that a bug-fix commit may have some *supplementary bug-fix commits* which supplement or correct the initial fix attempts [8]. Considering this one-to-many relationship between the bug report and bug-fix commits, some of those commits identified by keywords may be *relevant* (which means they may fix the same bug together) and other supplementary bug-fix commits may be still missing. It is important to study all these relevant bug-fix commits as a whole. For example, when we need to repair a buggy program by reusing fix operations from historical bug fixes [9], we may fail if we do not reuse supplementary bug fixes together.

Existing works [10] [11] on grouping related commits use some heuristics such as using the sliding window of a fixed time interval and determining if the authors are the same. Sliwerski et al. [12] and Yin et al. [13] find related changes if there is source code change overlap between two changes. These heuristics do not consider the characteristics of different projects and may lose some hidden relevance between relevant commits. We believe that the relevance between existing relevant bug-fix commits could help find those neglected bug-fix commits, which are supplementary to those identified individually by matching keywords.

Thanks to Git, a popular distributed version control system, relevant bug-fix commits tend to be locally gathered in the same branch and the relevance between them can be obtained easily. GitHub is a famous collaborative code hosting site built on top of Git. Based on the git-request-pull utility of Git, the "Fork & Pull" development model of GitHub becomes popular. A pull request that adds new features or fixes bugs may be accepted once the core team members agree, and the merged commits can be obtained easily from the pull request. Those commits in the same pull request always achieve the same goal and thus they are relevant. Moreover, developers tend to commit small changes with Git because of the distributed characteristics [14]. It means that developers may commit different changes several times for fixing a complicated bug. Based on these observations, we have an intuition that supplementary bug-fix commits are common and they could be identified by learning characteristics between the relevant commits of the same pull request.

In this paper, we propose a learning-based approach called SupBCFinder (**Sup**plementary **B**ug-fix **C**ommits **Finder**) to identify more bug-fix commits, as a supplement to the existing approaches. First, we identify those bug-fix pull requests and

extract their commits. For those commits from the same pull request, we assume that they are relevant. The relevance between commits indicates that they share the same goal. Second, we add some irrelevant commits as the negative samples to build a training dataset. Third, we extract features from these commits and compute the feature values for each pair of commits. Feature values reflect two commits' difference with respect to a certain feature. After that, we leverage machine learning techniques to build a discriminative classification model to predict whether two commits are relevant or not. Finally, we couple the identified bug-fix commits and their adjacent subsequent commits, and use the trained model to predict if these adjacent subsequent commits are supplementary to the identified bug-fix commits.

We have evaluated SupBCFinder on six software projects hosted on GitHub. The results indicate that SupBCFinder performs well on identifying supplementary bug-fix commits. Kalliamvakou et al. point out that some projects do not often use pull requests [15]. Thus, we further propose one heuristic to identify supplementary bug-fix commits, when there are not enough pull requests.

In short, this paper makes the following contributions:

- We make use of the relevance between adjacent commits in the same bug-fix pull request, and propose a learning-based approach to identify supplementary bug-fix commits;
- Experimental results show that the performance of our approach is much better than that of using a sliding window of one hour and that of analyzing the local changes;
- Our proposed heuristic outperforms the local change approach on most repositories (5/6) after limiting the time interval. We recommend to use this heuristic when there are not enough pull requests available.

The rest of this paper is organized as follows. Section II shows motivating examples. Then, Section III describes SupBCFinder in detail. Next, we conduct experiments and evaluate experimental results in Section IV. We discuss related work in Section V. Finally, Section VI concludes.

II. Motivating Examples

In this section, we illustrate two motivating examples.

Zhong and Su [6] classify bugs into two types: *reported bugs* and *on-demand bugs*. A reported bug is linked to a set of bug-fix commits that refer to the same issue number, while an on-demand bug is linked to one bug-fix commit identified by matching keywords such as "bug" or "fix". In their empirical study, they analyze these two kinds of bugs respectively. Fig. 1 shows four commits of Aries whose bug-fix commits have been analyzed by Zhong and Su [6]. These four commits appear in the same main branch "Trunk" and two other branches are created for other different purposes. Because these four commits do not refer to any issue number, they have been classified into *on-demand* bugs. However, according to their descriptions, it is obvious that the developer was trying to fix the same bug. If we analyze them individually, the links between these four commits will be missing and the analysis results will fail to reflect the real fix operations of one bug. Imagine that, when we reuse one of those bug-fix commits to repair a similar bug in other software, we will fail to repair the bug correctly because of the loss of supplementary bug fixes.

As we can see, these four relevant commits in Fig. 1 can still be identified as bug-fix commits by matching the keyword "fix". However, in practice, some supplementary bug-fix commits may be neglected by matching keywords. Fig. 2 shows two adjacent commits of "wordpress-mobile/WordPress-Android"[1] which is the Android client of WordPress. The top commit fixes a bug explained by the message and the bottom commit modifies that fix. The message of the bottom commit does not contain any words indicating that the commit fixes a bug. However, we find that this commit is a supplementary patch of the top one. The bug will fail to be eliminated without it.

The above two examples show the importance of grouping related bug-fix commits. Although these changes of two motivating examples are respectively committed by the same author, we fail to group them together when using a sliding window of 200 seconds [10]. After inspecting these commits, we find that these relevant ones often refer to the same nouns or contain the same lines in code changes. Hence, we think that the relevance between existing relevant bug-fix commits may help us to identify supplementary bug fixes for identified bug-fix commits. Then, there is a further question we need to consider: where are those supplementary bug-fix commits? If we analyze all subsequent commits of a given bug-fix commit, the time and space consumption will be high. As shown in Fig. 1 and Fig. 2, the supplementary bug-fix commits are adjacent to the initial bug-fix commits and they are locally gathered. Moreover, considering the following two reasons, we propose to find supplementary bug fixes from those adjacent subsequent commits.

1) Git: Working with Git, developers are able to create independent branches for adding new features or fixing bugs in a very convenient way. Obviously, these commits in the same branch are relevant because of the same goal. In addition, an empirical study [14] shows that Git's commits are smaller than SVN's in the size of changes, and developers tend to split one task into more commits when using Git. Although these commits have been split, they are adjacent in the same branch. These insights motivate us to find supplementary bug fixes by examining the adjacent subsequent commits.

2) Reopened Bugs: Developers may submit sets of commits to fix one bug when the bug report is reopened. In this case, the relevant sets of commits may appear far away in different locations of the Git graph. However, reopened bugs just comprise 4-7.25% of all bugs in three projects studied by [16]. Additionally, for reopened issues, we cannot just group all those commits together for analyzing the fixes of a bug,

[1] https://github.com/wordpress-mobile/WordPress-Android. Repositories hosted on GitHub can be found by concating "https://github.com/" with their names, so we won't give these links in the following throughout the paper.

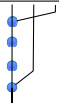
Graph	Description	Commit	Author	Date
	should be able to fix the Jenkins this time:(	690564b	Emily Jiang <ejiang@apache.org>	Mar 29, 2012, 5:56 PM
	fix the build break: revert to use the simple debug method	3002360	Emily Jiang <ejiang@apache.org>	Mar 29, 2012, 6:54 AM
	fix the build break - trying:)	6eef48b	Emily Jiang <ejiang@apache.org>	Mar 29, 2012, 5:49 AM
	fix the build break	e021d37	Emily Jiang <ejiang@apache.org>	Mar 28, 2012, 11:43 PM

Fig. 1. Aries's four adjacent commits that fix the same bug

```
commit 3af2f6ba391e6266ab4359dfcf490b5d6678b64b
Author: Mario Zorz<mariozorz@gmail.com>
Date:   Wed Jul 6 11:40:36 2016 -0300

    fixed checking connection and showing toast after text validation

index: .../org/wordpress/android/ui/people/PeopleInviteFragment.java
@@ -317,6 +317,12 @@
        ...
        if (usernamesToCheck.size() > 0) {
+
+           if (!NetworkUtils.checkConnection(getActivity())) {
+               enableSendButton(true);
+               return;
+           }
            ...
==================================================
commit 157eca164ab2029dcf80487af4ad9a9cd4700148
Author: Mario Zorz <mariozorz@gmail.com>
Date: Thu Jul 7 08:24:02 2016 -0300

        moved network connection check to only be made when the user
hits SEND as suggested by @hypes

Index: .../org/wordpress/android/ui/people/PeopleInviteFragment.java
@@ -318,11 +318,6 @@

        if (usernamesToCheck.size() > 0) {

-           if (!NetworkUtils.checkConnection(getActivity())) {
-               enableSendButton(true);
-               return;
-           }
-
            ...
@@ -423,6 +418,11 @@
        ...
+       if (!NetworkUtils.checkConnection(getActivity())) {
+           enableSendButton(true);
+           return;
+       }
+
        ...
```

Fig. 2. WordPress-Android's two adjacent commits that fix the same bug

because the running context of this reopened bug may have been changed during the evolution. Although the issue may be reopened once the bug reappeared with more test cases, previous bug-fix commits are able to relieve development pressure temporarily. Besides, from the view of automatic program repair, these previous bug-fix commits are valuable.

Based on the consideration above, we hypothesize that relevant bug-fix commits are locally gathered and supplementary bug-fix commits may be identified by learning from the relevance between existing relevant bug-fix commits.

III. Proposed Approach

This section describes our approach called SupBCFinder in detail.

A. Overall Framework

Bug-fix commits of *reported bugs* may be grouped directly from bug reports. The remaining bug-fix commits of *on-demand bugs* may have supplementary bug fixes just as shown in Sect. II. We think that the relevance hidden in these commits of the same bug-fix pull requests can help recover missing

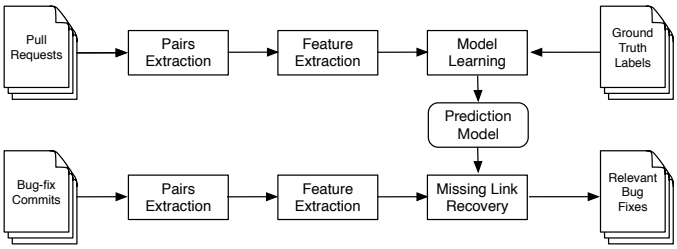


Fig. 3. Overall Framework

links between adjacent commits and the bug-fix commit of *on-demand* bugs. Our tool SupBCFinder consists of two main phases: *Training Phase* and *Predicting Phase*. Fig. 3 shows the overall framework of our approach.

During the training phase, SupBCFinder extracts *relevant* commits and *irrelevant* commits by analyzing the pull requests from the repository. Please note that we do not extract relevant commits from issues. As shown in the Algorithm 1 (Line 5), we select pull requests that contain bug-fix keywords and own N commits or less. Then we make pairs of commits, and each pair is labeled as relevant or irrelevant. After sorting the commits into an ordered list $\{C_0, C_1, ..., C_n\}$ (where C_0 is the oldest commit and C_n is the newest commit) in a bug-fix pull request, we form each two neighbor commits C_i and C_{i+1} (where $i \in [0,n)$) as a pair, which is then labeled as relevant. With Git, developers often create another branch for adding new features or fixing bugs. If tasks are relevant, developers may not create another branch. Based on these intuitions, we choose the ancestor commits $\mathbb{P}$ of the pull request to form the irrelevant pair (P_j, C_0) (where $P_j \in \mathbb{P}$). Note that, we limit the size of $\mathbb{P}$ to prevent the imbalance between relevant and irrelevant pairs (Line 7). The feature extraction component then extracts feature values from these pairs. After that, extracted features are provided to a model learning algorithm to build a model that discriminates relevant bug-fix commits from irrelevant commits.

During the predicting phase, SupBCFinder makes a pair (C_i, C_j) between a bug-fix commit C_i and one of its adjacent subsequent commits C_j. Next, the feature extraction component extracts features from pairs of commits and then provides them to the model. Finally, after predicting by the model, we obtain the supplementary bug-fix commits.

Please note that tangled commits [17] may affect the effectiveness of our approach. A study [18] on 10 open-source CVS and SVN projects shows that 8.4% of inspected code changes seem to be tangled. The proportion of tangled changes is small and we have an intuition that tangled commits

Algorithm 1 Training Set Creation

Input: $\mathbb{PR}$ is a set of all merged pull requests, $\mathbb{K}$ is a set of keywords to identify bug fixes, N is the parameter
Output: $\mathbb{V} = \{((C_i, C_j), label)\}$ is a set of commit pairs with labels

```
1: function CREATETRAINSET(ℙℝ, 𝕂, N)
2:     𝕍 ⇐ ∅
3:     for all PR ∈ ℙℝ do
4:         size = |PR.commits|
5:         if 1 < size ≤ N and 𝕂 ∩ tokenize(PR.doc) then
6:             ℂ ⇐ sort(PR.commits)
7:             ℙ ⇐ GETANCESTORS(C_0)[0 : size-1]
8:             for all i∈ [0, | ℂ |-1) do
9:                 𝕍 ⇐ 𝕍 ∪ {((C_i, C_{i+1}), 1)} /*relevant*/
10:            end for
11:            for all P_j∈ ℙ do
12:                𝕍 ⇐ 𝕍 ∪ {((P_j, C_0), 0)} /*irrelevant*/
13:            end for
14:        end if
15:    end for
16:    return 𝕍
17: end function
18:
19: function GETANCESTORS(C)
20:    for all parent∈C.parents do
21:        if ! isMerging(parent) then
22:            return {parent}
23:        end if
24:    end for
25:    ℙ ⇐ ∅
26:    for all parent∈C.parents do
27:        ℙ ⇐ ℙ ∪ GETANCESTORS(parent)
28:    end for
29:    return ℙ
30: end function
```

may appear more rarely because developers prefer to commit smaller changes when working with Git [14].

B. Training Data Acquisition

Issues is the bug tracking system of GitHub, which works like Bugzilla. Users report problems or bugs to developers by creating an issue. Developers will fix the bug or enhance existing function according to this issue. After finishing it, developers may refer the issue number in the commit message. However, not all of the relevant commits refer to the issue number. For example, the pull request `#4138`[2] of WordPress-Android says that it fixes the issue `#4137`, but only one commit of this pull request refers to this issue number while the other commit does not. It means if we pick out only those commits that refer to the issue numbers, we will still neglect some commits that are in fact relevant. To this end, we collect relevant commits from pull requests instead of issues.

[2]https://github.com/wordpress-mobile/WordPress-Android/pull/4138

TABLE I
LIST OF EXTRACTED FEATURES

Feature	Type	Description
T_1	Float	Ratio of the same nouns
D_1	Int	Time interval between two commits (in hours)
C_1	Bool	The second commit removes some lines that were added by the first commit
C_2	Float	Ratio of added-removed lines
C_3	Bool	The second commit adds some lines that were removed by the first commit
C_4	Float	Ratio of removed-added lines
C_5	Bool	The second commit adds or removes lines that also have been added or removed by the first commit
C_6	Float	Ratio of added-added and removed-removed lines

Pull Requests provides opportunities for other developers to make contributions to the repository. Once a pull request is accepted by core developers, its corresponding commits will be merged into the main branch of the repository. The information from pull requests about tasks (adding new features or fixing bugs) together with their corresponding commits is a valuable source of data for empirical studies. Commits in the same pull request are relevant and adjacent in the same branch. Therefore, if a pull request is put forward to fix a bug, the relevance between its bug-fix commits is helpful to build a discriminative model.

There are two problems that we need to solve: identifying bug-fix pull requests and ensuring that the pull request completes one single task. Multi-task pull requests may increase the burden of reviewing and lead to rejection [19]. However, we still need filter out those multi-task pull requests as many as possible to reduce the negative effect of the irrelevant groups of commits. In our approach, we first identify bug-fix pull requests by matching keywords (e.g., "fix", "bug", "error", "fault", "crash", "failure" and "fail") in titles and bodies of pull requests. Then, we control the number (e.g., we set the number < 5 in our experiments) of a bug-fix pull request's commits (Line 5 in Algorithm 1), because empirical studies show that bug-fix changes are smaller than that of adding features changes [20] and about 80% of pull requests own less than 3 commits [21]. Moreover, we have an intuition that multi-task pull requests tend to own more commits. After identifying single-task bug-fix pull requests, we generate training sets from them.

C. Feature Extraction

Once we get pairs of relevant and irrelevant commits, our feature extraction tool extracts features from those pairs of commits. The extracted features are listed in Table I.

A commit log message often illustrates the purpose of these changes or summarizes them. For example, the messages of those two commits in Fig. 2 show that changes are working on "connection". To automatically extract information from commits' messages, we utilize the natural language processing tool *nltk*[3] to extract and stem nouns which represent the working objects. If two commits' messages contain the same

[3]http://www.nltk.org

nouns, the probability of their working towards the same goal will be high [22]. Thus, we compute the ratio of the same nouns among all nouns as the feature T_1. In our tool, we have filtered some nouns such as "issue", "bug" and "error", because they are frequently used in commits' messages and do not reflect the working objects.

The time interval between two commits can also reflect the hidden relevance. Existing approaches consider commits relevant when the difference of their timestamps is below a certain threshold (e.g., 200 seconds). However, an empirical study shows that most of related commits cannot be grouped using the common sliding time window of 300 seconds and more than a half of related commits are committed over one hour apart [23]. Considered time intervals may reflect the working habit of developers, and thus we compute the time interval (in hours) between two commits as the feature D_1.

To better understand the content of a commit, we have also extracted features from code changes. Supplementary bug fixes may propagate changes of the initial bug fix into different locations of different files, revert the initial bug fix or modify some initial changes. As a result, there are some identical lines between commits' removed lines and added lines. Considering the order relation between two commits in the same pair in training sets or testing sets, we classify these cases into three categories: (1) The second commit removes some lines that are added by the first commit; (2) The second commit adds some lines that are removed by the first commit; (3) The second commit adds some lines that have been added by the first commit, or removes some lines that have been removed. These three categories correspond to the boolean features C_1, C_3 and C_5. We also compute the ratio of these lines as the features C_2, C_4 and C_6 by using the equations (1), (2) and (3).

$$C_2 = \frac{2 \times |set(A_1) \cap set(R_2)|}{|set(A_1)| + |set(R_2)|} \quad (1)$$

$$C_4 = \frac{2 \times |set(R_1) \cap set(A_2)|}{|set(R_1)| + |set(A_2)|} \quad (2)$$

$$C_6 = \frac{2 \times |(set(A_1) \cap set(A_2)) \cup (set(R_1) \cap set(R_2))|}{|set(A_1)| + |set(A_2)| + |set(R_1)| + |set(R_2)|} \quad (3)$$

where *set()* represents the function that extracts the set of all different lines from the collection of lines, A_1 and R_1 represent those added lines and removed lines respectively in the first commit, A_2 and R_2 represent those added lines and removed lines respectively in the second commit, $|\cdot|$ represents the size of the set. In our tool, some lines are neglected when considering the characteristic of different programming languages. For example, "else" often appears in one single line without any other tokens, and it is used in many completely different and irrelevant code changes.

Note that, we just analyze the adjacent commits of the bug-fix commits and these commits often appear in the same branch, which means that they may co-change because of the same goal such as adding new features. As shown in Fig. 4, a bug-fix commit may appear in the process of adding a new feature because a previous commit introduces a bug and its subsequent commit goes on adding the new feature. They are relevant due to the same goal of adding the new feature instead of fixing a bug. Our early experimental results show that more false positives will be produced if we add these features extracted from the authors, the association or the dependency of changed files. As a result, we do not use these features in SupBCFinder.

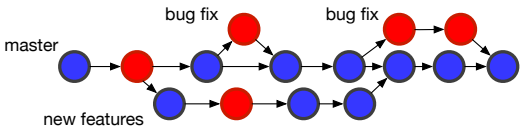


Fig. 4. A simple Git graph. The red circles represent bug-fix commits.

D. Model Learning

After extracting features from training pairs, we leverage the Support Vector Machine (SVM) algorithm to build a prediction model which is capable of differentiating relevant pairs from irrelevant pairs. Our problem is a classical binary classification problem. Existing classification algorithms such as Decision Tree and Bayesian Logistic Regression (BLR) also can be utilized in our framework. Considering that SVM is a high performance algorithm and has been widely used across different classification applications, we utilize the *scikit-learn*'s[4] implementation of SVM in SupBCFinder to conduct experiments.

IV. Experimental Evaluation

In this section, we conduct experiments to evaluate the effectiveness of SupBCFinder.

A. Dataset

Considering that our approach does not target projects written in particular languages, we collect six popular software repositories hosted on GitHub with a large number of issues and pull requests: "wordpress-mobile/WordPress-Android" (we use "WP-A" as its abbreviation), "atom/atom", "moby/moby", "opencv/opencv", "kubernetes/kubernetes" and "apple/swift". They are written in different popular programming languages and they are involved in different areas.

Table II describes the details of our dataset. The column "HEAD" represents the first ten digits of the latest commit's SHA number, demonstrating the version of the project we adopt. By calling the APIs of GitHub, we are able to download the pull requests of each repository. The subcolumn "All" of the column "Pull Requests" represents the number of all pull requests downloaded from GitHub before the date of the latest commit. Note that some pull requests are in the state of "open", which means that they have not been accepted. The subcolumn "Merged" represents the number of pull requests that have been merged into the repository. We identify bug-fix pull requests by matching their titles and bodies with keywords. To

[4]http://scikit-learn.org

TABLE II
SIX PROJECTS STUDIED IN EXPERIMENTS

Projects	Language	HEAD	Pull Requests					Commits		
			All	Merged	Size=2	Size=3	Size=4	All	Bug-Fix	SupBC
WordPress-Android	Java	2a35d280ca	2,379	2,110	323	182	81	18,844	1,677	19(/100)
Atom	JavaScript, CoffeeScript	9a42f82895	3,710	2,514	157	83	59	31,728	2,018	22(/100)
Moby	Go	916e9ad754	17,294	13,742	572	161	52	32,521	3,926	28(/100)
OpenCV	C++	03aa69da99	6,202	4,821	261	100	40	21,197	4,621	17(/100)
Kubernetes	Go	b58c7ec456	26,912	22,358	761	255	114	48,920	4,673	16(/100)
Swift	C++, Swift	72a1176d7b	10,279	9,118	464	210	91	54,221	11,378	17(/100)

reduce the noise from multi-task pull requests, we select those pull requests that contain less than 5 commits. The subcolumn "Size=*N*" represents the number of those bug-fix pull requests that contain *N* commits.

The subcolumn "All" of the column "Commits" represents the number of all commits and the subcolumn "Bug-Fix" represents the number of bug-fix commits that do not appear in a bug-fix pull request which contains less than 5 commits, because these pull requests will be used to generate the training sets in our experiments. We identify bug-fix commits by matching keywords that have been used to identify bug-fix pull requests. Then by matching the keyword "squashed", we filter out those commits that are squashed from many other commits. And we also filter out those owning two parent commits, which means they complete the task of merging other commits into the branch. Considering that issues are not always about fixing bugs, we do not match the issue numbers for identifying bug-fix commits.

As Table II shows, the number of each project's identified bug-fix commits is huge (more than 1600). Because we have to manually identify their supplementary bug fixes at first, we decide to randomly pick out 100 bug-fix commits from each project. Among these 100 bug commits, the number of commits that have supplementary bug-fix commits is shown in the subcolumn "SupBC". To get the testing sets, we select the adjacent subsequent commit which is the direct child of the selected bug-fix commit as the candidate supplementary bug-fix commit. In practice, each bug-fix commit may have more than one supplementary bug-fix commit, but the difficulty that we manually identify supplementary bug fixes will also increase when more commits are involved. Thus, to trade off, we make pairs between the identified bug-fix commits and their direct child commits to conduct experiments. Over testing sets, we find that the number of bug-fix commits that actually have supplementary bug fixes account for 19.8% on average. This ratio shows the importance of finding Supplementary bug fixes in practice.

B. Experimental Settings

After generating training sets, we extract features from them. However, different projects have their own characteristics. In our experiments, we have considered more details while computing the code features C_1-C_6. For example, we have neglected some lines such as "end" and "endif" while computing those code features for OpenCV, because they may appear in different structures meaninglessly.

During the training phase, we use the linear kernel SVM to produce discriminative models and we set its parameter *C* to the default value 1.0. Different values of this parameter may produce different hyperplanes of the SVM.

C. Research Questions

Our experimental evaluation seeks to address the following research questions:

1) ***RQ***$_1$*:* How effective is SupBCFinder in identifying supplementary bug-fix commits?

Considering that the sliding window approach has been widely used in grouping related commits, we compare SupBCFinder with it. And we also compare SupBCFinder with the strategy that considers the local change.

2) ***RQ***$_2$*:* Which of the extracted features best discriminate true supplementary bug-fix commits from the others?

In this question, we examine which of the extracted features are most helpful in differentiating true samples from others. We use Fisher score which has been used to estimate how discriminative the features are in the fields of both machine learning and software engineering [3].

3) ***RQ***$_3$*:* Are there any other general heuristics that are able to achieve similar performance?

Some repositories hosted on GitHub do not own enough pull requests to generate training sets for SupBCFinder. What's worse, some popular projects even do not support the pull-requests-based development at all [15]. In these cases, SupBCFinder will fail to work while some general heuristics will be still helpful.

D. Experimental Results

To evaluate the effectiveness of SupBCFinder, we compute precision, recall and F-measure that have been widely used in machine learning.

1) ***RQ***$_1$*:* Effectiveness of the Proposed Approach

The parameter *N* of SupBCFinder is used to control the generation of training sets from different sizes of pull requests. For example, if we set *N* to 3, we will get the training set generated from bug-fix pull requests that contain 2 or 3 commits just as shown in the algorithm (Line 5). As a result, we can use SupBCFinder (*N*=2) to predict the training set without generating from pull requests contain 2 commits. To assess the effectiveness of our approach, we use SupBCFinder with different parameters to cross validate its effectiveness. As shown in Fig. 5, we use SupBCFinder (*N*=2) to predict those commit pairs generated from bug-fix pull requests that

contain 3 or 4 commits and use SupBCFinder (N=3) to predict those commit pairs from bug-fix pull requests that contain 4 commits. When we use SupBCFinder (N=2) to predict the set of commit pairs generated from the pull requests contain 4 commits, we achieve the worst results. However, when we include the commit pairs from the pull requests contain 3 commits to guide our learning-based approach, our approach works better just as shown in the third subgraph of Fig. 5.

Furthermore, we use a sliding window of one hour and the local change approach to perform comparative experiments on the labeled testing sets. An empirical study [23] shows that most related commits cannot be grouped using the sliding window of 300 seconds and more than half of related commits are committed over one hour. Therefore, we set the time interval of the sliding window to one hour so that the sliding window approach is able to achieve high recall. For the local change approach, we treat a subsequent commit as the supplementary bug-fix commit if it changes the code that appears nearby (+/-25 lines, which is used by Yin et al. [13]) the changes introduced by the previous commit.

We use different N to investigate the effectiveness of SupBCFinder. As shown in Fig. 6, SupBCFinder achieves higher precision and recall than the sliding window approach and the local change approach over all six projects. The weak performance of the sliding window approach indicates that we cannot group the bug-fix commit with its subsequent commits just by considering the time interval. The local change approach also outperforms the sliding window approach. We compute the p-value of Mann-Whitney-Wilcoxon test between the F-measures of the local change approach and the SupBCFinder (N=2), and get the p-value 0.004998. Given the 0.05 significance level, the improvement of SupBCFinder over the local change approach is statistically significant. SupBCFinder (N=2) achieves the best performance in experiments. When the N increases, the effectiveness decreases slightly. We recommend to set N to 2 when using SupBCFinder.

However, comparing the prediction results of SupBCFinder shown in Fig. 5 and Fig. 6, we find that SupBCFinder achieves low precision and high recall while predicting the testing sets. There are two main reasons for this phenomenon. After inspecting the training sets, we find that the relevant commits are always in the same branch of Git while irrelevant commits are not. Appearing in different branches brings a big difference, which is easy to be discriminated by the prediction model. However, the irrelevant commits in the testing set sometimes appear in the same branch which is often forked for some particular goals such as adding new features or fixing bugs. And this will increase the difficulty of discriminating and lead to a relatively low precision. Note that the testing sets are labeled by ourselves manually, some supplementary bug-fix commits may still be neglected because of the unawareness of software background knowledge. This will lead to the relatively high recall achieved by SupBCFinder.

The analysis above shows that our approach can effectively identify supplementary bug-fix commits.

2) ***RQ***$_2$*:* Important Features

TABLE III
RANKED FEATURES

	WP-A	Atom	Moby	OpenCV	Kubernetes	Swift
1	C_1	C_1	T_1	C_1	T_1	T_1
2	C_2	T_1	C_1	C_2	C_1	C_1
3	T_1	C_2	C_2	T_1	C_5	C_2
4	C_5	C_5	C_5	C_5	C_2	C_5
5	C_6	C_6	C_6	C_3	C_6	C_3
6	C_3	C_3	C_3	C_6	D_1	C_6
7	C_4	C_4	C_4	C_4	C_3	D_1
8	D_1	D_1	D_1	D_1	C_4	C_4

Table III presents the features in descending order of their Fisher scores computed over the training sets constructed by those pull requests that contain two commits. From this table, we see C_1 is the most discriminative feature in WordPress-Android, Atom and OpenCV. In Moby, Kubernetes and Swift, it is also in the top two features. C_1 is a code feature which represents whether the subsequent commit removes a line which is added in the previous commit. Other code features such as C_2 and C_5 are also important features that are always ranked in the first half of all features. The text feature T_1 also achieves high fisher scores. The feature D_1 which represents the time interval, fails to be ranked in the top. This explains why we fail to achieve high precision and recall in prediction experiments above when just considering the time interval.

During the implementation of SupBCFinder, we have tried to extract some other features to improve the effectiveness of our approach. Similar messages of different commits may reflect similar work goals just like the three bottom commits in Fig.1. We compute the Levenshtein distance ratio [24] between two commits' messages as a feature. Although this feature ranks well in the training sets, it fails to improve the prediction results over testing sets. We also examine the features such as Latent Semantic Analysis (LSA) [25] and File Association Distance [26] to get a predictive model. These models with different parameters N work better on predicting other training sets, but achieve lower F-measure on predicting testing sets.

3) ***RQ***$_3$*:* General Heuristics

As shown in Table III, the text feature T_1 and the code features C_1 and C_5 achieve high rankings among all features. Inspired by this result, we wonder if there are some heuristics that are able to be used to find supplementary bug-fix commits. By combining the features T_1, C_1, C_3 and C_5, we propose one simple heuristic:

Heuristic: If two commits refer to the same nouns or the same lines existing in the *diff* contents of each commit, we consider them as relevant, otherwise irrelevant.

We use this heuristic to perform prediction experiments on the testing sets. As shown in Table IV, most relevant commits are identified correctly (the recall ranges from 87.5% to 100%), but more false positives have been produced (the precision ranges from 30.2% to 54.2%). Although this heuristic always outperform the sliding window approach, it achieves worse effectiveness than the local change approach over the projects OpenCV and Swift.

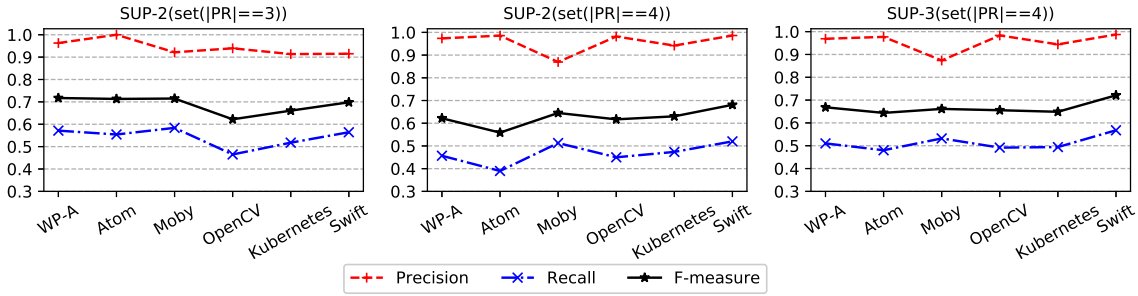


Fig. 5. Precision, recall and F-measure of SupBCFinder with different parameters on different training sets. "SUP-*N*" represents the SupBCFinder with the parameter *N*. "set(|PR|)==*N*" represents the set of labeled commit pairs generated from the pull requests that contain *N* commits.

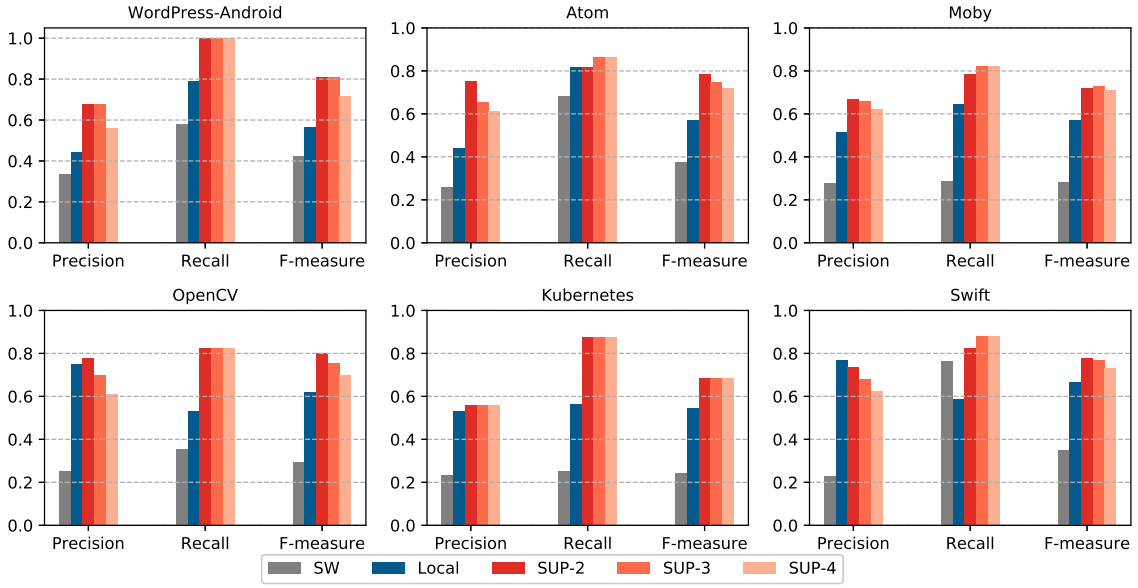


Fig. 6. Precision, recall and F-measure of three different approaches on different projects. "SW" represents the sliding window approach. "Local" represents the local change approach. "SUP-*N*" represents the SupBCFinder with the parameter *N*.

TABLE IV
PRECISION, RECALL AND F-MEASURE ACHIEVED BY OUR HEURISTIC

Projects	Precision	Recall	F-measure
WordPress-Android	50.0%	100.0%	66.7%
Atom	52.6%	90.9%	66.7%
Moby	54.2%	92.9%	68.4%
OpenCV	42.9%	88.2%	57.7%
Kubernetes	45.2%	87.5%	59.6%
Swift	30.2%	94.1%	45.7%

Although the date feature D_1 fails to achieve a high ranking among all features, we wonder if it can help improve the effectiveness of this proposed heuristic. We set the D_1 to different numbers (in hours) to analyze the results of our proposed heuristic. As shown in Fig. 7, the precision achieved on OpenCV and Swift increase after limiting the time interval between two commits to smaller numbers. The time-limit heuristic (e.g., 100 hours) is able to outperform the local change approach on most projects (5/6). In the case that the number of pull requests is not large enough for training a prediction model, SupBCFinder may fail to work well. However, this proposed heuristic still works in such a case, outperforming the sliding window approach and the local change approach in identifying supplementary bug-fix commits. Thus, we recommend to use this heuristic when there are not enough pull requests available.

E. Threats to Validity

Internal validity threats correspond to the construction of training sets. We identify bug-fix pull requests just by matching keywords, which may introduce false positives. We control the number of each pull request's commits to exclude those multi-task ones. However, some multi-task pull requests may still exist in our training sets.

External validity threats correspond to the generalization of our proposed approach. To our knowledge, there is no existing work on identifying supplementary bug-fix commits without matching issue numbers. For each studied repository, we randomly pick out 100 bug-fix commits by matching keywords,

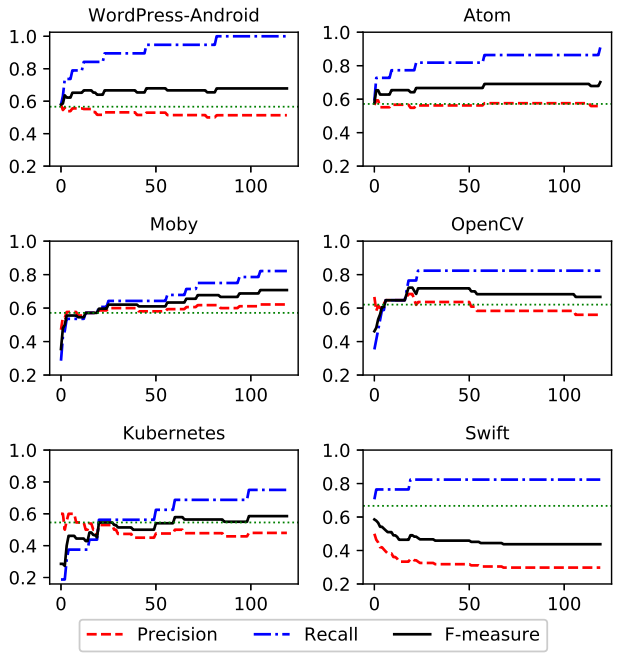


Fig. 7. Precision, recall and F-measure achieved after setting different time intervals (in hours showed on x-axis) to our proposed heuristic. The dash line of each subgraph is the F-measure achieved by the local change approach.

and then we construct the testing set. Before predicting, we manually put labels on all commit pairs. Although we have checked the labels many times, some wrong labels may exist due to the unawareness of field background knowledge.

Construct validity threats correspond to the appropriateness of our evaluation metrics (i.e. precision, recall and F-measure). These metrics are widely used in many different studies and we believe the construct validity threats have been reduced.

V. Related Work

In this section, we describe related work on identifying bug-fix commits, grouping related commits and supplementary bug fixes.

A. Identifying Bug-fix Commits

Bird et al. [1] point out that the biases in bug-fix datasets can reduce the effectiveness of studies.

Mockus and Votta [2] classify a change as "corrective" if the commit message contains one of following keywords: "fix", "bug", and "error". Tian et al. [3] extract features from commits and integrate two classification algorithms to predict if a commit fixes a bug. Martinez and Monperrus [20] define different bags of commits according to the change size because they assume small transactions are very likely to contain a bug fix instead of a new feature.

Linking commits to bug reports helps us identify bug-fix commits. However, the links may be missing when developers do not refer to the bug report ID [27]. ReLink [4] learns criteria of features such as time interval, bug owner and change committer and text similarity from explicit links to recover missing links. Also based on learning techniques, FRlink [5] utilizes non-source documents in commits and discards those files containing no similar code terms as those in bug reports to reduce data noise.

However, considering the complexity and the effectiveness, [7] [6] prefer to identify bug-fix commits in two steps. First, they find those commits whose messages contain the issue numbers. After that, they determine a commit as a bug fix by matching keywords such as "bug" or "fix". As a supplement to this approach, SupBCFinder is able to find supplementary bug-fix commits that may be neglected.

B. Grouping Related Commits

Canfora et al. [28] point out that it is useful to consider changes related to the same development/maintenance activity as a whole when performing historical analysis.

Zeller proposes six different criteria to group related changes and he group commits of the same date to reduce the number of unresolved test cases in delta debugging [29]. Zimmermann et al. [10] group the individual per-file changes by using the sliding window approach: if the authors are the same and the time interval is smaller than 200 seconds. Kagdi et al. [11] use the committer, the time interval and the combination of them to group related commits to recover traceability links. However, an empirical study conducted by Miura et al [23] shows that 83% of related revisions cannot be grouped using the common sliding window of 300 seconds.

Yamauchi et al. [30] generate feature vectors by extracting identifiers from code changes and use the Repeated Bisection to cluster commits. This clustering-based approach is able to group textual similar commits, but may fail to find related commits that are totally dissimilar in textual representation.

C. Supplementary Bug Fixes

Sliwerski et al. [12] identify bug-fix commits and then locate fix-inducing changes if they get undone by these bug-fix commits. Gu et al. [16] propose an approach that combines distance-bounded weakest precondition with symbolic execution to evaluate fixes and detect bad ones that need supplementary bug fixes. Yin et al. [13] use the local change approach that we have used in this paper and the examining the issue numbers to identity those partial fixes. An empirical study of supplementary bug fixes conducted by Park et al. [8] shows that only 7% to 14% of supplementary patches have a content that has at least 5 lines similar to its initial patch. And only 17% to 36% of supplementary patch locations have co-changed with the initial patch locations within 50 days after the date of the initial patch. Little overlap occurred among code clone, structural dependency and historical co-change analyses. And the remaining supplementary patch locations do not have direct dependence on nor do they overlap the initial patch locations. An et al. [31] study the relationship between supplementary bug fixes and reopened bugs. They find that 21.6% to 33.8% of supplementary bug fixes are related to reopened bugs. Miura et al. [23] recommend that software evolution studies should be performed at the work item level.

These works and their findings on supplementary bug fixes motivate our work on identifying them.

VI. Conclusion

In this paper, we propose a learning-based approach that leverages existing groups of relevant bug-fix commits to build a discriminative model, and use this model to identify supplementary bug-fix commits for those commits that have been identified as bug-fix commits by matching keywords.

Our experiments on six open source projects hosted on GitHub show that our approach SupBCFinder is able to find most of the supplementary bug-fix commits with a few false positives. Prediction results show that these discriminative models can achieve the best performance (i.e., achieving an average F-measure of 76.2%). Considering that many projects do not have enough pull requests for training, our approach may not work well in this case. To this end, after analyzing the important features extracted in our approach, we propose one simple heuristic to help us identify supplementary bug-fix commits in more scenes. And the experimental results show that the heuristic also outperforms the sliding window approach with higher precision and recall. After limiting the time interval, this heuristic is able to outperform the local change approach on most repositories (5/6).

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2017YFB1001802) and the National Natural Science Foundation of China (No. 61672529, 61502015).

References

[1] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th ESEC/FSE*. ACM, 2009, pp. 121–130.

[2] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases." in *Proceedings of International Conference on Software Maintenance*. IEEE, 2000, pp. 120–130.

[3] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 386–396.

[4] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[5] Y. Sun, Q. Wang, and Y. Yang, "Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance," *Information and Software Technology*, 2016.

[6] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE, 2015, pp. 913–923.

[7] S. Kim, K. Pan, and E. Whitehead Jr, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 35–45.

[8] J. Park, M. Kim, and D.-H. Bae, "An empirical study of supplementary patches in open source projects," *Empirical Software Engineering*, vol. 22, no. 1, pp. 436–473, 2017.

[9] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 447–456.

[10] T. Zimmermann and P. Weißgerber, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE, 2004, pp. 563–572.

[11] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *Proceedings of the 15th IEEE International Conference on Program Comprehension*. IEEE, 2007, pp. 145–154.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[13] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[14] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 322–333.

[15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[16] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 55–64.

[17] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories,*. IEEE, 2013, pp. 121–130.

[18] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 351–360.

[19] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE, 2015, pp. 358–368.

[20] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.

[21] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.

[22] W. Maalej and H.-J. Happel, "Can development work describe itself?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 191–200.

[23] K. Miura, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "The impact of task granularity on co-evolution analyses," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, pp. 47:1–47:10.

[24] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," vol. 10, no. 8, pp. 707–710, 1966.

[25] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[26] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan, "Recovering commit dependencies for selective code integration in software product lines," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 202–211.

[27] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

[28] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[29] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6. Springer-Verlag, 1999, pp. 253–267.

[30] K. Yamauchi, J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, "Clustering commits for understanding the intents of implementation," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 406–410.

[31] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. reopened bugs," in *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 205–214.