5-2015

RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information

Tien-Duy B. LE
Singapore Management University, btdle.2012@phdis.smu.edu.sg

Mario Linares VASQUEZ
College of William and Mary

David LO
Singapore Management University, davidlo@smu.edu.sg

Denys POSHYVANYK
College of William and Mary

DOI: https://doi.org/10.1109/ICPC.2015.13

RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information

Tien-Duy B. Le*, Mario Linares-Vásquez†, David Lo*, and Denys Poshyvanyk†
*School of Information Systems, Singapore Management University
†Computer Science Department, The College of William and Mary
{btdle.2012,davidlo}@smu.edu.sg, {mlinarev,denys}@cs.wm.edu

***Abstract*—Links between issue reports and their corresponding commits in version control systems are often missing. However, these links are important for measuring the quality of a software system, predicting defects, and many other tasks. Several approaches have been designed to solve this problem by automatically linking bug reports to source code commits via comparison of textual information in commit messages and bug reports. Yet, the effectiveness of these techniques is oftentimes suboptimal when commit messages are empty or contain minimum information; this particular problem makes the process of recovering traceability links between commits and bug reports particularly challenging. In this work, we aim at improving the effectiveness of existing bug linking techniques by utilizing *rich contextual information*. We rely on a recently proposed approach, namely *ChangeScribe*, which generates commit messages containing rich contextual information by using code summarization techniques. Our approach then extracts features from these automatically generated commit messages and bug reports, and inputs them into a classification technique that creates a discriminative model used to predict if a link exists between a commit message and a bug report. We compared our approach, coined as *RCLinker* (Rich Context Linker), to *MLink*, which is an existing state-of-the-art bug linking approach. Our experiment results on bug reports from six software projects show that *RCLinker* outperforms *MLink* in terms of F-measure by 138.66%.**

I. Introduction

In order to improve the quality of software systems, developers often allow users and testers submit issue reports in issue tracking systems, such as JIRA or Bugzilla. Developers would then work on these issue reports and commit corresponding changes to version control systems, e.g., SVN or Git. Unfortunately, many issue reports can not be linked to their corresponding commits for many reasons [4], [3], [49]. However, these missing links are important because they can be used to support a number of development and research tasks. For example, these links can be used to find classes that are most buggy ones by counting the number of bug reports that are linked to them. Furthermore, these links can be used to generate high quality defect data (i.e., the number of defects that affect various classes in a project during its life-time), which can, in turn, be used to build effective bug prediction solutions [24], [7], [43], [28], [41], [63], [21].

Due to the importance of these links, a number of past studies have proposed approaches to recover missing links between bug reports and their corresponding commits in version control systems. Wu *et al.* and Nguyen *et al.* propose approaches named *ReLink* [60] and *MLink* [47] respectively. These approaches enumerate a set of potential links and remove the ones that do not satisfy some criteria defined based on a set of thresholds. These thresholds are learned by heuristically enumerating various values based on a training and/or a validation dataset. A main operation in *ReLink* and *MLink* is the computation of similarity between the textual contents in commit messages and issue reports. Intuitively, the more similar the textual contents are the more likely the commit is to be linked to the issue report. Unfortunately, many commit messages are empty or contain insufficient information [39], [19]. This makes it hard for *ReLink* and *MLink* to identify links between bug reports and commits.

In this paper, we propose a novel bug linking approach that addresses major weaknesses of existing solutions. First, we rely on rich contextual information for detecting links between commits and bug reports. We do so by enriching existing (oftentimes, very short or empty) commit messages with automatically generated content. We make use of a recently proposed technique, namely *ChangeScribe* [15], [36], which analyzes code changes and generates commit messages by combining several code summarization techniques. These automatically generated commit messages capture rich contextual information including commits' intent, summaries of fine-grained and structural code changes, as well as impact of committed code changes. Second, differently from past approaches, we propose a new classification-based technique to create a discriminative model that can predict if a link exists between an issue report and a commit. However, it should be noted that past approaches do not rely on classification algorithms to establish the links. We coin our approach as *RCLinker*, which stands for Rich Context Linker.

We evaluated our proposed approach on 385 bug reports and 3,249 commits from six software projects: CLI, IO, Collections, Math, Lang, and CSV. These bug reports were extracted from the projects' JIRA bug tracking systems. Our decision toward using JIRA was mostly based on the fact that Bissyande *et al.* found that bug reports extracted from JIRA tended to be better linked as compared to other bug tracking systems [5]. We perform ten-fold cross-validation by omitting some of the links and use *RCLinker* to predict some of the omitted links. We then compute precision, recall, and F-measure of our proposed approach. Aside from evaluating the effectiveness of our approach, we also compare *RCLinker*

978-1-4673-8159-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPC.2015.13

to *MLink*, the latest state-of-the-art approach for linking bug reports to their corresponding commits.

Overall, our work provides the following contributions:

1) We are first to use rich contextual information that captures intent behind the commits, fine-grained and structural source code changes, impact of code changes, and many more, for establishing links between issue reports and commits;
2) We propose a classification-based approach which extracts 20 textual and metadata features from a training set of links to build a discriminative model that can predict if a link exists between an issue report and a commit. None of the existing linking approaches is based on a classifier to tackle this task;
3) We have performed experiments on bug reports from six software systems. Our experiments demonstrate that *RCLinker* outperforms existing state-of-the-art approach *MLink* in terms of F-measure by 138.66%.

We organize the rest of the paper as follows. We briefly discuss background materials used in our work in Section II. Next, Section III mainly describes our proposed approach. We evaluate the experimental results in Section IV. Section V discusses the related work to our study. Finally, we conclude the paper with some of the directions for the future work in Section VI.

II. Background

In this section, we first describe *ChangeScribe*, an approach that can generate commit messages by analyzing code changes. Next, we describe the classification algorithm that is used in this paper, namely the random forest algorithm.

A. ChangeScribe

$ChangeScribe$ [15], [36] is an approach for automatic generation of commit messages, which combines code summarization techniques [25], [45], stereotypes detection [16], [17], [44], and impact analysis. $ChangeScribe$ extracts and analyzes the differences between two versions of the source code, and also performs a commit characterization based on the stereotype of methods modified, added and removed. The result is a commit message that provides an overview of the changes and classifies and describes in detail each of the changes; the message describes the *what* and *why* of a change using natural language. $ChangeScribe$ also allows controlling the length of the message by using an impact set-based heuristic.

$ChangeScribe$ uses change types from the change-set and fine-grained changes to generate the two parts of the commit message: general description and detailed description. The general description characterizes the change-set with a general overview of the commit, which includes the following: (i) a phrase describing whether it is an initial commit, (ii) a phrase describing commits intent, (iii) a phrase indicating class renaming operations, (iv) a sentence listing new modules, and (v) a sentence indicating whether the commit includes changes to properties or internationalization files. Sentences (i) and (iii)-(v) are generated with $ChangeScribe$ specific templates [15], and the commits intent in sentence (ii) is based on the commit stereotypes proposed by Dragan *et al.* [17].

The second part of the message (i.e., detailed description) characterizes the changes made to the Java files, and the changes are organized according to packages. $ChangeScribe$ describes the class' goal and its relationships with other objects, for the case of new and removed files. Moreover, if an existing file is modified, $ChangeScribe$ describes the changes for each inserted, modified and deleted code snippet. $ChangeScribe$ generates descriptive phrases for all changes at class/method/statement level. For instance, for added/removed classes, $ChangeScribe$ describes the class responsibility based on the approach by Hill *et al.* [25], and for describing the class signatures $ChangeScribe$ uses the class stereotypes defined by Moreno *et al.* [44]. For modified classes, $ChangeScribe$ generates descriptions for all the changes at statement level by using fine-grained source code changes extracted with $ChangeDistiller$ [20] and fourteen templates aimed at describing structural changes such as class/method renames, functionality addition/removal, parameter type change, among others. For example, when a new method is added, the following sentence is generated: *Add an additional functionality to <Object>*. But, if the method is removed, the resulting sentence is *remove functionality to <Object>*. In addition, the descriptions include context information such as method's visibility or whether a method is unused: *remove an unused functionality from <Object>*. For each added, removed or modified type (i.e., class), a sentence is added to describe the impact of the change in two ways: (i) references to the type in the change set, and (ii) co-lateral changes triggered when a method was added to or removed from an existing class.

Because the detailed description of all the changes can lead to long and superfluous messages, $ChangeScribe$ controls the length of the messages by using a heuristic, which is based on impact analysis [15], [36]. In summary, the detailed description lists only the most representative classes in the change set, because representative classes contribute more to the description of the change-set and are more related to the rationale behind the commit. These classes are defined as the ones with changes that have high impact on the change-set, and the threshold for the impact is defined by the developer.

Figure 1 shows examples of commit messages generated by $ChangeScribe$ for commits of SpringSocial, Apache Solr, JFreeChart and Retrofit from GitHub.

B. Random Forest

Random forest is an ensemble classification approach that utilizes several weaker classifiers (i.e., classification algorithms) to create a more powerful classifier. Each weaker classifier is trained by a decision tree learning algorithm on a sampled subset of training data. In *RCLinker*, we combine random forest with an under-sampling strategy to construct a prediction model for recovering missing links. We utilize

a) Initial commit of Spring social
http://goo.gl/5Igx1s

Initial commit. This is a degenerate modifier commit: this change set is composed of empty, incidental, and abstract methods. These methods indicate that a new feature is planned. This commit includes changes to internationalization, properties or configuration files (.classpath, .gitignore, .project, ...). The commit includes these new modules:
- facebook
- twitter [...]

b) A commit of Apache Solr
http://goo.gl/lV6aWm

This is a state update modifier commit: this change set is composed only of mutator methods, and these methods provide changes related to updates of an object's state. This change set is mainly composed of:
1. Changes to package org.apache.solr.common.cloud:
1.1. Modifications to ClusterState.java:
1.1.1. Remove an unused functionality to get shard

c) A commit of JFreeChart
http://goo.gl/StXeJS

This is a large modifier commit: this is a commit with many methods and combines multiple roles. This commit includes changes to internationalization, properties or configuration files (pom.xml). This change set is mainly composed of:
1. Changes to package retrofit.converter:
1.1. Add a Converter implementation for simple XML converter. It allows to: Instantiate simple XML converter with serializer; Process simple XML converter simple XML converter from body; Convert simple XML converter to body
Referenced by: SimpleXMLConverterTest class

d) A commit of RetroFit
http://goo.gl/mmbxzC

This is a small modifier commit that does not change the system significantly. This change set is mainly composed of: 1. Changes to package org.jfree.chart:
1.1. Modifications to TestUtilities.java:
1.1.1. Add javadoc at serialised(Object) method
2. Changes to package org.jfree.chart.util:
2.1. ModiFIcations to LineUtilities.java:
2.1.1. Add a functionality to extend line
The added/removed methods triggered changes to RingPlot class

Fig. 1. Examples of Commit Messages Generated with ChangeScribe.

the implementation of random forest that is available in Weka toolkit with default settings [22].

III. Proposed Approach

This section describes our proposed approach in detail. We describe the overall framework in Section III-A. Section III-B describes a list of features that we extract from *ChangeScribe* generated and developer-written commit messages, and bug reports. We describe our strategy to construct a discriminative model to predict the existence of a link between a commit and a bug report in Section III-C.

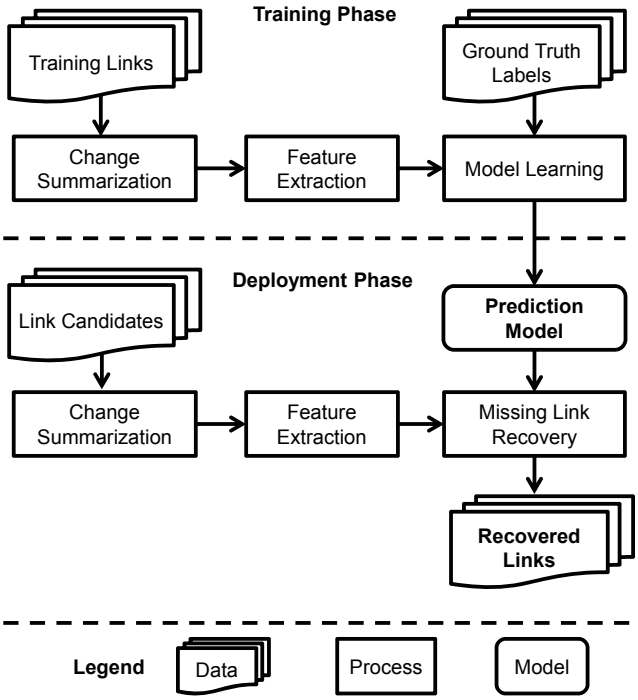

Fig. 2. Overall Framework

A. Overall Framework

Figure 2 shows the overall framework of our approach. The figure depicts the two main phases in *RCLinker*: *Training Phase* and *Deployment Phase*. The training phase takes as input a predefined set of issue-commit links. An issue-commit link is a pair of issue report and commit for which there exists either a *true link* (i.e., the commit fixes the issue report) or *false link* (i.e., the commit does not fix the issue report) between them. Figure 3 shows a sample issue report from CLI. In this work, we are especially interested with the following fields available for bug reports in JIRA: report date, last update date, reporter, priority, summary, description, and a list of comments with the identifiers of the commenters. The output of the training phase is a prediction model which is able to discover missing issue-commit links. Next, in the deployment phase, our approach takes as input the prediction model constructed in the training phase and a set of unseen link candidates. Using the prediction model, in the deployment phase, our approach recovers missing *true* issue-commit links from the set of input link candidates. The following paragraphs describe details of the two phases.

a) Training Phase: The issue-commit links used for training are created from a set of issues in an issue-tracking system and a set of commits from a source code repository. Each link comes with the following information:

1) Metadata and textual description (i.e., summary, description, and comments) of the corresponding issue.
2) Commit messages and source code of the previous and updated revision of the corresponding commit.
3) A label which indicates whether there is an actual link between the issue and the commit. If there is a link

Report Date: 31/Jul/09 23:00
Last Update Date: 02/Mar/13 14:19
Reporter: Kristoff Kiefer
Priority: 3 (Major)
Summary: *Standard help text will not show mandatory arguments for first option*
Description: *The generated help text will not show "<arg>" for the first argument added ...*
Comments:
Commenter: Kristoff Kiefer
Content: *Underlying cause is something in OptionBuilder*
Commenter: Emmanuel Bourg
Content: *Kristoff what version of Commons CLI did you use? I tried your test case on the trunk and ...*

Fig. 3. Issue Report CLI-186 of CLI.

between the issue and commit, the link is labeled as *true link*. Otherwise, its label is *false link*.

In the training phase, there are three major steps. They are *Change Summarization*, *Feature Extraction*, and *Model Learning*. In the *Change Summarization* step, we apply *ChangeScribe* [15], [36] to summarize source code changes in each commit. The automatically generated message output by *ChangeScribe* for each commit is then combined with the developer-written message of the corresponding commit. After this step, we perform *Feature Extraction* to retrieve important characteristics from the training links. The extracted characteristics are forwarded to the *Model Learning* step for constructing a prediction model. The output of this phase is a model that is capable of recovering missing *true* links from a set of unseen link candidates in the deployment phase.

b) Deployment Phase: In the deployment phase, our approach takes as input the prediction model constructed in the training phase, and a set of unseen link candidates (i.e., which are not part of the training set). Link candidates are created by pairing unlinked commits with all issues in the issue tracking system. These link candidates are then processed to recover missing links. In this phase, we also execute *ChangeScribe* in the *Change Summarization* process, and extract important characteristics from each link candidate in the *Features Extraction* process. The extracted characteristics (i.e., features) are then input to the prediction model. Given a set of features from a link candidate, the prediction model classifies whether the link candidate is a true link or not. The output of our approach during the deployment phase is a list of predicted true links that are recovered from the input link candidates.

B. Feature Extraction

In this section, we describe details of features extracted from issue-commit links and link candidates. In total, we have 20 features that are divided into two different types: text features and metadata features. Table II describes the features using the notations listed in Table I. The following paragraphs describe these features in more details.

1) Text Features: Intuitively, if a commit and an issue are linked together, their textual contents are likely to be similar. Therefore, we extract text features by capturing textual similarities (measured in different ways) between issue reports and commits.

TABLE I
LIST OF NOTATIONS

Notation	Description
Commit Notations	
MSG	Human-writen commit message
CSMSG	Commit message generated by ChangeScribe
CDATE	Commit date
Issue Notations	
SUM	Summary of an issue
DES	Description of an issue
PRI	Priority of an issue (PRI $\in \{1, 2, 3, 4, 5, 6\}$)
NCOM	Number of comments posted in an issue
COM_i	i^{th} comment in an issue ($1 \leq i \leq$ NCOM)
$WS(D)$	Set of distinct words in document D
$+$	text concatenation operator
RDATE	Report date of the issue
UDATE	Last updated date of the issue
$\text{DATE}(\text{COM}_i)$	Created date of comment i^{th} in an issue

TABLE II
LIST OF EXTRACTED FEATURES. NOTATIONS SHOWN IN TABLE I ARE USED TO COMPUTE FEATURE VALUES.

Feature	Description
Text Features	
T_1	$\text{cosine}(\text{SUM} + \text{DES} + \sum \text{COM}_i, \text{MSG} + \text{CSMSG})$
T_2	average of $\text{cosine}(\text{I}_\text{a}, \text{C}_\text{b})$ $(\text{I}_\text{a} \in \{\text{SUM}, \text{DES}, \text{COM}_i\}, \text{C}_\text{b} \in \{\text{MSG}, \text{CSMSG}\})$
T_3	max of $\text{cosine}(\text{I}_\text{a}, \text{C}_\text{b})$ $(\text{I}_\text{a} \in \{\text{SUM}, \text{DES}, \text{COM}_i\}, \text{C}_\text{b} \in \{\text{MSG}, \text{CSMSG}\})$
T_4	T_2/T_3
T_5	T_2/T_1
T_6	$\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i) \cap WS(\text{MSG}+\text{CSMSG})\rvert$
T_7	$\frac{\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i) \cap WS(\text{MSG}+\text{CSMSG})\rvert}{\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i) \cup WS(\text{MSG}+\text{CSMSG})\rvert}$
T_8	$\frac{\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i) \cap WS(\text{MSG}+\text{CSMSG})\rvert}{\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i)\rvert}$
T_9	$\frac{\lvert WS(\text{SUM}+\text{DES}+\sum \text{COM}_i) \cap WS(\text{MSG}+\text{CSMSG})\rvert}{\lvert WS(\text{MSG}+\text{CSMSG})\rvert}$
Metadata Features	
M_1	Number of changed files in the commit that are mentioned in the issue text
M_2	$\frac{\text{M}_1}{\text{NCOM}+\text{PRI}}$
M_3	$\text{M}_3 = 1$ if the issue reporter is also the committer. Otherwise, $\text{M}_3 = 0$.
M_4	$\text{M}_4 = 1$ if the committer posts comments in the issue. Otherwise, $\text{M}_3 = 0$.
M_5	$\text{M}_5 = 1$ if the commit date is between the report date and the last updated date of the issue. Otherwise, $\text{M}_5 = 0$.
M_6	$\text{CDATE} - \text{RDATE}$
M_7	$\text{UDATE} - \text{CDATE}$
M_8	M_6/M_7
M_9	$\min_{1 \ldots \text{NCOM}}\{\lvert\text{CDATE} - \text{DATE}(\text{COM}_i)\rvert\}$
M_{10}	$\lvert\text{CDATE} - \text{DATE}(\text{COM}_\text{NCOM})\rvert$
M_{11}	$\lvert\text{CDATE} - \text{DATE}(\text{COM}_\text{NCOM-1})\rvert$

Before computing values of text features, we perform text preprocessing on the textual contents of issues (i.e., contents in the summary and description fields, and comments in issue reports) and commits (i.e., developer-written commit messages and automatically generated messages by *ChangeScribe*). The purpose of the preprocessing is to convert text to its normalized form which maximizes the chance of matching relevant issues and commits. We perform three text preprocessing steps, which are text normalization, stop word removal, and

stemming, described below:

1) *Text Normalization:* In this step, special symbols and punctuation marks are deleted from the text. Then, the text is separated into words. If a word follows Camel-Casing, it is split into separate tokens. We include the original word and its split tokens into the normalized text. Including original words helps increase chances of matching rare identifiers. For example, "bugLinking" is split to "bug" and "linking". Next, "bugLinking", "bug", and "linking" are all included in the normalized text.
2) *Stopword Removal:* In information retrieval (IR), English stopwords are often excluded from text documents as they frequently appear in documents and are likely to be less helpful in retrieving relevant documents. Similarly, in this step, we also remove English stopwords from normalized text. These stopwords are frequently used by developers, and are unlikely to be helpful while recovering missing links. We use the list of stopwords obtained from [46].
3) *Stemming:* In this step, we run the Porter Stemming algorithm [51] to convert words to their root forms. For example, "linked", "linkage", and "linking" are all transformed to "link".

After performing text preprocessing, we compute values of features T_1 to T_9. Among the nine text features, T_1 to T_5 are calculated based on cosine similarities. To find the values of these features, we represent the corresponding textual contents of an issue report and a commit as two document vectors whose dimensions correspond to the number of words in their contents. We assign a weight to each word by utilizing the *term frequency-inverse document frequency* (tf-idf) weighting scheme. The following is the tf-idf formula of a word w in a document d of corpus D (i.e., a collection of documents):

$$tfidf(w, d, D) = f(w, d) \times log\frac{|D|}{|\{d_i \in D | w \in d_i|} \quad (1)$$

In Equation 1, $f(w, d)$ is the number of times word w occurs in document d, and $w \in d_i$ denotes that the word w appears in the document d_i. Using this formula, the cosine similarity of two documents is calculated based on the cosine similarity of their corresponding document vectors [40].

T_1 captures the cosine similarity between the text in an issue report (i.e., summary, description, and all comments) and the text in a commit (i.e., developer-written commit message and *ChangeScribe*'s message). If T_1 is high, intuitively, the corresponding issue and commit are likely to be linked with each other. T_2 and T_3 are the average and maximum cosine similarity between the various textual parts of an issue report and a commit. Similar to T_1, T_2 and T_3 are also good indicators of textual relevance between an issue report and a commit. T_4 and T_5 are normalized forms of T_2 with respect to T_3 and T_1, respectively.

Different from the first five text features, T_6 to T_9 are computed based on the number of common words shared between the textual contents of an issue report and a commit. T_6 captures the number of shared words between all text in an issue report and a commit. T_7 is the ratio of T_6 to the number of distinct words in the corresponding issue and commit pair. Similarly, T_8 and T_9 are the ratio of T_6 to the number of distinct words in the corresponding issue and commit respectively.

Overall, our textual features measure the likelihood of a link between issues and commits based on their textual similarities.

2) Metadata Features: There are totally 11 metadata features that we extract from issues and commits. Values of these metadata features are computed based on metadata information of issues, e.g., report date, last update date, priority, reporter name, etc., and of commits, e.g., commit date, committer name, list of changed files, etc.

In particular, M_1 captures the relevance between an issue and a commit based on the number of changed files modified by the commit that are mentioned in the textual description of an issue (i.e., in the summary, description, or comments). If the value of M_1 is high, it means more changed files of the commit are being referred to in the issue report. Hence, higher M_1 is an indication of an existing link between an issue and a commit. M_2 is the ratio of M_1 to the sum of the number of comments and the priority value of an issue, where the denominator of M_2 reflects the importance level of an issue.

Next, M_3 and M_4 take into account the involvement of a committer in handling an issue. If a commit is linked with an issue, its committer likely submits at least one comment to the issue. In many cases, the reporter of an issue is also assigned to fix that issue.

Features M_5 to M_8 capture various relationships between the time commits are made and their corresponding issues are created or updated. Intuitively, in most cases, the date a commit is made and the date its corresponding issue is submitted or updated should be close to each other. M_5 is a boolean feature that indicates if the commit date is between the issue report reporting date and its last update date. M_6 and M_7 are the time difference between the date a commit is made against the date an issue report is submitted and the last update date of an issue report, respectively. M_8 is the ratio of M_6 to M_7.

The last three metadata features, i.e., M_9 to M_{11}, are computed based on the time difference between the date when a commit is made and the date when a comment of an issue report is posted. Usually, when a committer submits a fix for an issue to the repository, he or she is likely to inform other developers that the issue has been addressed by posting a comment to that issue. Following this intuition, M_9 captures the minimum absolute time difference between the commit date and the date when comments for an issue are posted. Furthermore, unless an issue is not completely resolved, there are likely to be no or only a few other comments (e.g., a thank you comment) after the comment informing the fix. To capture this intuition, for M_{10} and M_{11}, we compute the absolute time difference between the commit date and the date the last and second last comment of an issue are posted, respectively.

Overall, our metadata features capture relationships between metadata information in commits and issues that can be used to predict missing links between them.

C. Model Learning

Algorithm 1: Undersampling

Input: T: Set of training links
N: Number of selected nearest neighbors
I: Set of issues of links in T
C: Set of commits of links in T
NN_I: Nearest neighbors of issues in I
NN_C: Nearest neighbors of commits in C
Output: D: Undersampled set of training links

```
1  TrueLinks ← extract true links from T
2  FalseLinks ← extract false links from T
3  foreach tl ∈ TrueLinks do
4      curIssue ← tl.issue
5      curCommit ← tl.commit
6      nnIssCnt ← 0
7      foreach issue ∈ NN_I[curIssue] do
8          lc ← link of issue and curCommit
9          if lc ∈ FalseLinks ∧ lc ∉ D then
10             include lc to D
11             nnIssCnt ← nnIssCnt + 1
12         end
13         if nnIssCnt > N then
14             break
15         end
16     end
17     nnCmtCnt ← 0
18     foreach commit ∈ NN_C[curCommit] do
19         lc ← link of curIssue and commit
20         if lc ∈ FalseLinks ∧ lc ∉ D then
21             include lc to D
22             nnCmtCnt ← nnCmtCnt + 1
23         end
24         if nnCmtCnt > N then
25             break
26         end
27     end
28 end
29 Append TrueLinks to D
30 return D
```

Using the features extracted from the training links, we construct a prediction model, which is capable of differentiating *true links* from *false links*. In particular, we use the Random Forest algorithm [6] as part of the *Model Learning* process in *RCLinker* (Figure 2).

A problem that makes effective learning of this model hard is imbalanced training data [27]. Usually, there are fewer true links than false links in the training links as an issue is typically only linked to a few commits. For that reason, we balance the set of training links by using undersampling. Balancing data not only helps reducing the cost of learning but also improves the performance of the prediction. In our undersampling strategy, for each true link, we select a number of false links whose issues and commits are nearest neighbors to the true link in terms of their textual descriptions.

TABLE III
DATASET: COLUMN "#FISSUES" INDICATES THE NUMBER OF FIXED ISSUES OBTAINED FROM THE ISSUE TRACKING SYSTEMS. COLUMN "#CHANGES" IS THE NUMBER OF COMMITS EXTRACTED FROM SOURCE CODE REPOSITORIES. COLUMN "#LINKS" IS THE NUMBER OF TRUE ISSUE-COMMIT LINKS INDICATED IN THE ISSUE TRACKING SYSTEM (I.E., JIRA).

Project	#FIssues	#Changes	#Links	Study Period
CLI	56	410	70	2006–2013
Collections	63	617	102	2012–2013
CSV	49	766	75	2010–2014
IO	65	371	97	2009–2011
Lang	64	648	105	2011
Math	88	437	160	2010
Total	385	3249	609	

Our undersampling algorithm is shown in Algorithm 1. It takes as input a set of training links T, the number of selected nearest neighbors N, the sets of issues and commits of links in T, as well as their lists of nearest neighbors. The output of our algorithm is an undersampled set of training links where the degree of imbalance is proportional to the value of N. To construct the list of nearest neighbors of an issue report, we sort the other issue reports in descending order of the cosine similarities (c.f., Equation 1) between their textual contents (i.e., summary, description, and comments). Similarly, for each commit, we sort the other commits in descending order of the cosine similarities between their textual contents (i.e., developer-written message and *ChangeScribe* message) to generate the list of its nearest neighbors.

In Algorithm 1, at line #3 we enumerate each link labeled as a true link in the training data. Next, for each true link, lines #4 to #5, we extract its issue report and commit. Lines #7 to #16 select issues that are the nearest to the true link's issue (i.e., $NN_I[curIssue]$). These issues are then paired with the true link's commit to create false links. These false links are then included in the output set D if they are in the training data and not part of D yet. We stop this process after we have added N links to D. Lines #18 to #26 select false links by finding commits that are the most similar to the commit of the true link (i.e., $NN_C[curCommit]$). These commits are paired with the true link's issue to create false links. These false links are then included in the output set D if they are in the training data and not part of D yet. We stop this process after we have added an additional N links to D. Finally, line #28 merges the set of true links to the output set D, and our algorithm returns the undersampled set of links at line #30.

IV. EXPERIMENTAL EVALUATION

A. Dataset

In our experiments, we collected a large number of issues and commits coming from several Apache software projects. The projects included in our evaluation are Commons CLI [9], Commons IO [12], Commons Collections [10], Commons Math [14], Commons Lang [13], Commons CSV [11]. All these projects use JIRA[1] as the issue

[1]https://issues.apache.org/jira

tracking system. For each project, a study period was chosen (see Table III), and all the issues and changes that had been submitted and committed in that study period were included, respectively. With the collected changes, we executed *ChangeScribe* to generate commit messages summarizing all source code changes. We also collected true links in JIRA. The number of true links is shown in column "#Links" in Table III. Noticeably, according to the table, the number of true links accounts for a small percentage (i.e., less than 1%) over the number of possible combinations between issues and commits (i.e., link candidates). Table III also describes other detailed information of our dataset.

B. Evaluation Metrics

To evaluate the performance of our approach, we computed Precision, Recall, and F-measure. These metrics are widely used in machine learning and data mining to assess the effectiveness of classification algorithms. We estimated the values of Precision, Recall, and F-measure based on four statistics: True Positives (TP), False Positives (FP), True Negatives (TN), False Negatives (FN). Their definitions are as follows:

True Positives:	Number of *true* issue-commit links that are predicted *correctly*.
False Positives:	Number of *false* issue-commit links that are predicted *incorrectly*.
True Negatives:	Number of *false* issue-commit links that are predicted *correctly*.
False Negatives:	Number of *true* issue-commit links that are predicted *incorrectly*

We use the above statistics to estimate Precision, Recall, and F-measure using the following formulas:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$\textit{F-measure} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Both Precision and Recall reflect the effectiveness of our prediction model. According to the above formulas, Precision is the ratio between the number of true positives over the number of link candidates that are predicted as true links by our model. On the other hand, Recall is the percentage of the number of true positives over the total amount of true links. Importantly, between Precision and Recall, there is usually an inverse relation where higher Precision might come with lower Recall, and vice versa. Thus, F-measure, which is the harmonic mean of Precision and Recall, is used to combine the two metrics into one single summary measure.

C. Experimental Settings

Ten-fold cross-validation is a standard way of estimating the accuracy of a prediction engine in data mining [23]. Its purpose is to evaluate how the result of a prediction engine generalizes to an independent test data. In our experiments, we also conducted ten-fold cross-validation on the set of issue-commit links to assess the performance of our proposed approach. Our experiments are performed on a Intel Xeon E5-2667 server with 189 GB RAM running Linux 2.6 OS.

Furthermore, we observed that most of issue-commit links labeled by JIRA or other issue tracking systems were explicit links (i.e., issue IDs were mentioned in commit messages, or revision IDs were mentioned in issues). In such cases of explicit links, it was trivial to determine true links between issues and commits. For that reason, we excluded issue IDs from commit messages, and vice versa, to ensure that all the links were implicit in the deployment phase. In our experiments, we set the number of nearest neighbors N used for the undersampling process to 5. With this setting, the number of true links accounts for approximately 9% of the undersampled training data.

D. Research Questions

a) ***RQ***$_1$*: How effective is our proposed approach in recovering missing traceability links between issues and committed changes?* In this research question, we evaluate the performance of *RCLinker* using the dataset, metrics, and settings described earlier. We set the number of selected nearest neighbors to $N = 5$.

b) ***RQ***$_2$*: How effective is our approach as compared to other state-of-the-art approaches for detecting issue-commit links?* There are several approaches for recovering missing issue–commit links where *MLink* [47] is one of the state-of-the-art techniques. In this research question, we compare our approach to *MLink*. Similarly to ***RQ***$_1$, we also perform ten-fold cross-validation on our proposed approach as well as *MLink*. Finally, we evaluate the two approaches based on the metrics.

c) ***RQ***$_3$*: What is the impact of varying the number of nearest neighbors N on the performance of* RCLinker*?* In this research question, we investigate the performance of approach when varying the number of selected nearest neighbors N in the Algorithm 1 used in *Model Learning* process (see Section III-C). By default, we set $N = 5$ where the number of true links accounts for approximately 9% of the training data. When the value of N increases, there are more negative instances (i.e., false links) included in the training data, and vice versa. To understand the impact of N, we considered the values of $N \in \{1, 5, 10, 15, 20\}$ and compared the changes in the resulting metrics.

d) ***RQ***$_4$*: Which of the extracted features best discriminate true links from the other ones?* In this research question, we examine which of the proposed features (see Table II) are most helpful in differentiating true links from the other ones. To do that, we rank the extracted features based on their Fisher scores. In machine learning, Fisher score is a standard measurement to estimate how discriminative the features are [18]. In software engineering, several studies also employ Fisher scores to evaluate the importance of features [56], [33], [34],

TABLE IV
RCLINKER: PRECISIONS, RECALLS, AND F-MEASURES

Project	Precision	Recall	F-measure
CLI	45.71%	91.43%	60.95%
Collections	43.32%	92.16%	58.93%
CSV	39.05%	88.00%	54.10%
IO	58.74%	86.60%	70.00%
Lang	57.89%	94.29%	71.74%
Math	60.73%	83.13%	70.18%
Average	50.91%	89.27%	64.32%

[35]. Fisher score of a feature is calculated as follow:

$$FS(j) = \frac{\sum_{class=1}^{\#class}(\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class}(\frac{1}{n_{class}-1}\sum_{i=1}^{n_{class}}(x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2)} \quad (2)$$

In Equation 2, $FS(j)$ is the Fisher score of the j^{th} feature, n_{class} is the number of data points (i.e., number of links) with label $class$ (i.e., *true link* or *false link*), $\bar{x}_j$ is the average value of the j^{th} feature over all data points (i.e., links), $\bar{x}_j^{(class)}$ is the average value of the j^{th} feature over all data instances with label $class$. According to the formula, if a feature has a Fisher score of zero, that feature is not discriminative enough to distinguish true links from the other ones. On the other hand, a feature is very discriminative if its Fisher score is considerably larger than zero. The more discriminative a feature is, the more important it is. In this research question, for each project, we calculate and rank the features based on their Fisher scores.

E. Experimental Results

1) ***RQ***$_1$ *– Effectiveness of Our Approach:* In this RQ, we inspect the performance of our approach on different projects in the dataset. Table IV shows the Precision, Recall, and F-measure of our approach. According to the table, our approach achieves an average Precision, Recall, and F-measure of 50.91%, 89.27%, and 64.32%, respectively. Among the six projects, Lang obtains the best F-measure (i.e., 72.90%), and CSV is the one that has the lowest F-measure (i.e., 41.91%). Noticeably, our approach has higher Recall as compared to Precision for all the projects. On average, our approach's Recall (i.e., 89.27%) is higher than its Precision (i.e., 50.91%). We believe recall is more important than precision since finding true links is important, and not too many links are generated, and thus developers can quickly inspect the generated candidate links to remove the false positives.

2) ***RQ***$_2$ *– Our Approach vs. the Baseline:* In this RQ, we perform ten-fold cross-validation to compare the performance of *MLink* [47] with our approach. In particular, *MLink* takes as input a set of issue reports, a set of commits, and four parameters θ_p, θ_n, θ_t, and θ_a. Thus, we perform the Hill-Climbing algorithm for tuning θ_p, θ_n, θ_t, and θ_a for *MLink* (following the description in [47]) in each fold. *MLink* uses explicit links mentioned in the commit logs to train a set of thresholds to infer missing links. In each fold of our experiment, we inject explicit links into the commit logs based on the training data. To do that, we labeled commit messages with issue IDs whose pair forms a training true link. Using the tuned parameters, we used *MLink* to recover missing links. Table V presents the Precision, Recall, and F-measure of *MLink* on recovering missing links in our dataset. According to the table, *MLink* achieves an average Precision, Recall, and F-measure of 56.40%, 17.96%, 26.95%. Among the projects, CLI has the highest F-measure (i.e., 44.90%), and CSV has the lowest F-measure (i.e., 7.06%). Comparing to Table IV, even though *MLink* has higher average Precision (i.e., 56.40% vs. 50.91%), *RCLinker* still outperforms *MLink* in terms of average Recall and F-measure by 397.05% and 138.66% respectively. Furthermore, F-measure of *RCLinker* is higher than *MLink* on all the projects. F-measure is a standard metric that is often used to evaluate if an increase in recall (precision) outweighs a reduction in precision (recall).

TABLE V
MLINK: PRECISIONS, RECALLS, AND F-MEASURES

Project	Precision	Recall	F-measure
CLI	78.57%	31.43%	44.90%
Collections	68.42%	25.49%	37.14%
CSV	30.00%	4.00%	7.06%
IO	34.48%	10.31%	15.87%
Lang	58.06%	17.14%	26.47%
Math	68.89%	19.38%	30.24%
Average	56.40%	17.96%	26.95%

3) ***RQ***$_3$ *– Varying Amount of Training Data:* In this research question, we varied the value of N in Algorithm 1, and inspected the impact of N on the effectiveness of *RCLinker*. Table VI indicates the average Precision, Recall, and F-measure when varying $N \in \{1, 5, 10, 15, 20, 25\}$. According to the table, *RCLinker* significantly loses its effectiveness when $N = 1$ as compared to the other values of N. At $N = 1$, *RCLinker* achieves a high Recall of 92.34%, but it loses its effectiveness due to low Precision (i.e., 19.70%). This is because the amount of false links in undersampled training data is not sufficiently enough for the prediction model to distinguish between false links and true links. As the result, the rate of false positives (FP) increases as the prediction model misinterprets many false link instances as true link instances. For that reason, our F-measure when $N = 1$ is as low as 31.76%. Next, for $\in \{5, 10, 15, 20, 25\}$, the Precision of *RCLinker* is improved, and ranges from 50.91% and 73.73%. As Precision increases, Recall of our approach slightly decreases, but still maintains high rate (which ranges from 84.50% to 89.27%). Overall, the value of N has substantial effect on the effectiveness of our approach where large values of N help improving *RCLinker*'s effectiveness. However, larger values of N increases the runtime cost of *RCLinker*. The average execution time of *RCLinker* across the six projects increases steadily, between 2.42 to 5.26 seconds, as we increase N from 1 to 25.

4) ***RQ***$_4$ *– Important Features:* In this research question, we sort features in descending order of their Fisher scores. Table VII presents top-10 features that have the highest Fisher scores in each project. From the table, T_3 is the most discriminative feature in all the projects. T_3 is a text feature that captures the maximum cosine similarity between the various textual parts of an issue report (i.e., summary, description,

TABLE VI
VARYING NUMBER OF NEAREST NEIGHBORS N: AVERAGE PRECISIONS, RECALLS, AND F-MEASURES

N	Precision	Recall	F-measure
1	19.70%	92.34%	31.76%
5	50.91%	89.27%	64.32%
10	61.95%	87.04%	72.12%
15	66.23%	85.96%	74.62%
20	72.87%	85.30%	78.41%
25	73.73%	84.50%	78.68%

TABLE VII
TOP-10 IMPORTANT FEATURES

	CLI	Collections	CSV	IO	Lang	Math
1	T_3	T_3	T_3	T_3	T_3	T_3
2	M_5	T_2	T_2	T_1	T_1	T_2
3	T_1	T_1	T_1	T_2	T_2	T_1
4	T_2	T_5	T_5	T_7	M_5	T_5
5	T_5	T_8	T_7	T_8	T_7	T_7
6	T_7	T_7	T_8	T_6	T_5	T_4
7	T_4	T_4	M_5	M_1	T_9	T_8
8	T_8	M_5	M_8	T_5	T_4	M_5
9	T_9	M_{10}	T_4	T_4	T_8	T_6
10	M_9	T_9	T_6	T_9	M_1	M_1

and comments) and a commit (i.e., commit message and ChangeScribe message). In addition to T_3, T_1 and T_2 are also important as they are in the top-3 features of at least five out of the six projects. T_1, T_2, and T_3 capture the cosine similarities between issue reports and commits (but in different ways). Among M_1 to M_{11} (i.e., metadata features), M_5 are in the top-10 important features of five out of the six projects. M_5 is a boolean feature that captures if the commit date is between the report date and last updated date of an issue report. Noticeably, text features account for at least 80% of the important features in each project. This indicates that text features are more discriminative than metadata features.

F. Threats to Validity

There are a number of threats that might affect the validity of our study. These are broadly classified into internal, external and construct validity threats. Threats to *internal validity* relate to experimenter's errors. We inspected our implementation many times, however, there might still be hidden errors that we were not aware of. Threats to *external validity* relate to the generalizability of our findings. In the experiments, we only used six Apache projects with 385 issue reports and 3,249 commits. Furthermore, all of the projects are relying on SVN as the version control system and JIRA as an issue tracking system. All these projects are written in Java programming language. In the future, we are planning on minimizing this threat by experimenting on more projects written in various programming languages. We also plan to conduct experiments on more issue reports and commits. Threats to *construct validity* relate to the suitability of our evaluation metrics. We used standard metrics to evaluate the performance of our approach. Precision, recall and F-measure metrics are well known [23] and have been used in many software engineering studies [60], [47]. Thus, we believe that we sufficiently minimized threats to construct validity in our study.

V. RELATED WORK

In this section, we highlight research studies that are closely related to our work. These are studies in recovering missing links and application classification algorithm for software engineering tasks. The following two subsections present and discuss these two main lines of research.

A. Recovering Missing Links

Several studies proposed approaches for recovering missing links between issue reports and their corresponding commits. The latest studies include the following:

1) Wu *et al.* proposed *ReLink* [60], which recovers missing links by generating a set of candidate links and filtering them based on three criteria: textual similarity between commit messages and issue reports, time duration between issue report submission and code change commit, and mapping between committers and bug report commenters. ReLink learns several thresholds based on a training set of true links that are explicitly marked by developers. To learn these thresholds, *ReLink* enumerates a set of threshold values and selects values that work best for the training data.
2) Bissyande *et al.* proposed a simple information retrieval model based on vector space modeling to recommend missing links and showed that its performance is not too far away from that one of *ReLink* [5]. They are also the first to show that JIRA dataset is suitable for evaluating bug linking techniques since the ground truth answers are typically well maintained by developers. Also, JIRA provides an easier mechanism (as compared to Bugzilla) to link commits and bug reports.
3) Nguyen *et al.* proposed *MLink* which is a multi-layered approach for recovering missing links between issue reports and commits [47]. *MLink* generates a set of candidate links and filters them based on a set of criteria including textual similarity between commits and code changes with bug reports, similarity between changed source code and code fragments in bug reports, and time constraint that a relevant commit must be made in-between bug report open and close time. MLink also learns several thresholds by enumerating a set of threshold values and selecting those that work best for a training data. Differently from *ReLink*, *MLink* analyzes code changes in addition to commit logs, however it ignores mapping between committers and bug report comments, and time duration between bug report submission and code change commit. *MLink* has been shown to outperform *ReLink*.

Among the bug report linking techniques, *MLink* is the state-of-the-art approach since it has been shown to outperform *ReLink*. We have shown that our approach *RCLinker* is able to outperform *MLink* by a substantial margin. Our work is different from these existing approaches in the following ways. First, we use *ChangeScribe* to generate rich contextual information for code changes. *ChangeScribe* is able to extract rich information including code elements that are not changed

by a commit but are affected by it. Second, we make use of machine learning, in particular a classification algorithm with our undersampling strategy, to predict the existence of links between a commit and an issue report. None of the past approaches leverages the power of a classification algorithm. Third, features used by our approach to characterize true and false links capture additional dimensions that are not captured by the criteria used by *ReLink* and those used by *MLink*. Indeed, our features capture criteria that are considered by *ReLink* but ignored by *MLink* and vice versa, as well as additional criteria.

In addition to *ReLink* and *MLink*, there are also other research studies on recovering missing links. Cleland-Huang *et al.* proposed two approaches that link regulatory codes to product specific requirements [8]. The first approach computes the probability of a word to relate to a regulation and use these probabilities to compute the probability of a requirement document to be linked to a regulation. The second approach uses a web mining approach to link requirement document to regulation. There also a number of techniques that infer missing links between duplicated bug reports [52], [59], [26], [55], [54], [58], [48], [1]. These approaches typically compare textual similarities between bug reports to recommend a list of bug reports that are very similar to each other. Different from these past studies, we focus on a different problem, namely the linking of bug reports to their corresponding commits. One of the latest approaches by Alipour *et al.* compares bug reports based on their contextual similarities measured by comparing each bug report to a list of contextual words (e.g., words related to efficiency, functionality, maintainability, etc.) [1]. In this work, we also use contextual information, however this information is inferred using code summarization techniques.

B. Application of Classification Algorithm in Software Engineering Tasks

There have been many research studies that apply classification techniques to automate various software engineering tasks. We present some of these studies.

Classification techniques have been applied to predict important information of issues opened in bug tracking systems. Menzies and Marcus, Lamkafi *et al.*, and Tian *et al.* made use of various classification algorithms to predict the values of the severity fields of bug reports [42], [31], [32], [57]. Lamkafi *et al.* predicted coarse-grained bug severity labels, while Menzies and Marcus and Tian *et al.* predicted fine-grained severity labels. Tian *et al.* built a statistical model that considers multiple factors to predict the priority of bug reports [61]. Different from severity, which is assigned based on a user perspective, priority is assigned based on the developer perspective. Tian *et al.* considered different familes of features including temporal, textual, author, related-report, severity, and product features to predict the priority of bug reports. Antoniol *et al.* introduced a framework to predict whether an issue is a feature request or a bug report [2]. Ko and Myers investigate and compare linguistic characteristics of issue report summaries and descriptions to differentiate between bug reports and feature requests [29]. Kochhar *et al.* extended Antoniol *et al.* work by predicting fine-grained issue reclassifications [30]. In their work, an issue can be classified as a bug, a request for improvement, documentation, refactoring task, etc. Zhang *et al.* proposed a classification-based technique that can estimate if the time needed to resolve a bug report will be short or long [62].

A number of classification techniques have been used to predict modules that are likely to contain bugs and vulnerabilities. Lu *et al.* proposed approaches that utilize semi-supervised learning and active learning with dimensionality reduction to predict defect prone modules [38], [37]. Panichella *et al.* predicted defect prone software entities by leveraging defect data from another project [50]. Scandariato *et al.* proposed a vulnerability prediction model learned from text features extracted from source code files to predict which components of a software application are more likely to contain security vulnerabilities [53].

VI. Conclusion and Future Work

Links between issue reports and their corresponding commits are often missing. However, these links are important for software maintenance tasks including assessing the reliability of a particular part of a software system or predicting future defective software components (e.g., classes, files, etc.). To deal with this issue, a number of past approaches have been proposed to link bug reports to their corresponding commits, however, their performance still has significant room for improvement. In this work, we propose a new bug linking approach *RCLinker* that leverages rich contextual information that are generated by *ChangeScribe* and a text classification solution that creates a discriminative model based on 20 different features to differentiate between true and false links. We have compared *RCLinker* against *MLink*, which is the latest state-of-the-art bug linking approach. Our experiments on bug reports from six projects demonstrate that *RCLinker* outperforms *MLink* in terms of F-measure by 138.66%.

In the future, we are planning on improving the effectiveness of *RCLinker* even further. We will experiment with various classification algorithms and investigate their effectiveness. We will also investigate additional features that can be more effective than the current set of features. We are also planning on further minimizing the threats to external validity by experimenting with more bug reports from more software projects. We also plan to perform additional experiments (e.g., wrapper subset evaluation) to gain more insight into the features by investigating if we can use a reduced set of features to obtain similar effectiveness.

Acknowledgement. We would like to thank the authors of *MLink* especially Anh Tuan Nguyen and Tien Nguyen for releasing the binary code of *MLink* and helping us to use and tune the tool in our experiments. The authors from W&M are supported in part by the NSF CCF-1218129 and NSF CCF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detec tion," in *MSR*, 2013.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[3] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, 2010, pp. 97–106.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 121–130.

[5] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 89–98.

[6] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[7] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences*, vol. 179, no. 8, pp. 1040–1058, 2009.

[8] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 155–164.

[9] Commons CLI, *http://commons.apache.org/proper/commons-cli/*.

[10] Commons Collections, *http://commons.apache.org/proper/commons-collections/*.

[11] Commons CSV, *http://commons.apache.org/proper/commons-csv/*.

[12] Commons IO, *http://commons.apache.org/proper/commons-io/*.

[13] Commons Lang, *http://commons.apache.org/proper/commons-lang/*.

[14] Commons Math, *http://commons.apache.org/proper/commons-math/*.

[15] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2014, pp. 275–284.

[16] N. Dragan, M. Collard, and J. Maletic, "Reverse engineering method stereotypes," in *ICSM'06*, 2006, pp. 24–34.

[17] N. Dragan, M. Collard, M. Hammad, and J. Maletic, "Using stereotypes to help characterize commits," in *ICSM'11*, 2011, pp. 520–523.

[18] R. Duda, P. Hart, and D. Stork, *Pattern Classification*, 2nd ed. Wiley Interscience, 2000.

[19] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *ICSE'13*, 2013, pp. 422–431.

[20] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725 –743, 2007.

[21] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[23] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[24] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[25] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *ICSE'09*, 2009, pp. 232–242.

[26] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, 2008, pp. 52–61.

[27] N. Japkowicz and S. Stephen, "The class imbalance problem: a systematic study," *Intelligent Data Analysis*, vol. 6, no. 5, pp. 429–449, 2002.

[28] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, 2008.

[29] A. Ko and B. Myers, "A linguistic analysis of how people describe software problems," in *IEEE Visual Languages and Human-Centric Computing*, 2006, pp. 127–134.

[30] P. S. Kochhar, F. Thung, and D. Lo, "Automatic fine-grained issue report reclassification," in *2014 19th International Conference on Engineering of Complex Computer Systems, Tianjin, China, August 4-7, 2014*, 2014, pp. 126–135.

[31] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2010, pp. 1–10.

[32] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 249–258.

[33] T.-D. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 310–319.

[34] T. B. Le, F. Thung, and D. Lo, "Predicting effectiveness of ir-based bug localization techniques," in *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, 2014, pp. 335–345.

[35] T.-D. Le, D. Lo, and F. Thung, "Should i follow this fault localization tool's output?" *Empirical Software Engineering*, pp. 1–38, 2014.

[36] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *37th IEEE/ACM International Conference on Software Engineering (ICSE'15), Formal Research Tool Demonstration*, 2015, p. to appear.

[37] H. Lu, B. Cukic, and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 314–317.

[38] H. Lu, E. Kocaguneli, and B. Cukic, "Defect prediction between software versions with active learning and dimensionality reduction," in *25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2014.

[39] W. Maalej and H. Happel, "Can development work describe itself?" in *MSR'10*, 2010, pp. 191–200.

[40] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1.

[41] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[42] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *IEEE International Conference on Software Maintenance (ICSM 2008)*. IEEE, 2008, pp. 346–355.

[43] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 47–54.

[44] L. Moreno and A. Marcus, "Jstereocode: automatically identifying method and class stereotypes in java code," in *ASE'12*, 2012, pp. 358–361.

[45] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *ICPC'13*, 2013, pp. 23–32.

[46] MySQL Stopwords, *http://dev.mysql.com/doc/refman/5.1/en/fulltext-stopwords.html*.

[47] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 63.

[48] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 70–79.

[49] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "A case study of bias in bug-fix datasets," in *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, 2010, pp. 259–268.

[50] A. Panichella, R. Oliveto, and A. D. Lucia, "Cross-project defect prediction models: L'union fait la force," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, 2014, pp. 164–173.

[51] M. F. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980.

[52] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 499–510.

[53] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," 2014.

[54] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[55] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 45–54.

[56] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[57] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *19th Working Conference on Reverse Engineering (WCRE), 2012*. IEEE, 2012, pp. 215–224.

[58] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, 2012, pp. 385–390.

[59] 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 *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 461–470.

[60] 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.

[61] Yuan Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 200–209.

[62] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: an empirical study of commercial software projects," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 1042–1051.

[63] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 531–540.