Noname manuscript No.
(will be inserted by the editor)

Characterizing and Identifying Reverted Commits

Meng Yan · Xin Xia · David Lo · Ahmed E. Hassan · Shanping Li

Received: date / Accepted: date

Abstract In practice, a popular and coarse-grained approach for recovering from a problematic commit is to revert it (i.e., undoing the change). However, reverted commits could induce some issues for software development, such as impeding the development progress and increasing the difficulty for maintenance. In order to mitigate these issues, we set out to explore the following central question: can we characterize and identify which commits will be reverted?

In this paper, we characterize commits using 27 commit features and build an identification model to identify commits that will be reverted. We first identify reverted commits by analyzing commit messages and comparing the changed content, and extract 27 commit features that can be divided into three dimensions, namely change, developer and message, respectively. Then, we build an identification model (e.g., random forest) based on the extracted features. To evaluate the effectiveness of our proposed model, we perform an empirical study on ten open source projects including a total of 125,241 commits. Our experimental results show that our model outperforms

Meng Yan
College of Computer Science and Technology, Zhejiang University, China
E-mail: mengy@zju.edu.cn

Xin Xia
Faculty of Information Technology, Monash University, Melbourne, Australia
E-mail: xin.xia@monash.edu

David Lo
School of Information Systems, Singapore Management University, Singapore
E-mail: davidlo@smu.edu.sg

Ahmed E. Hassan
School of Computing, Queen's University, Canada
E-mail: ahmed@cs.queensu.ca

Shanping Li
College of Computer Science and Technology, Zhejiang University, China
E-mail: shan@zju.edu.cn

two baselines in terms of AUC-ROC and cost-effectiveness (i.e., percentage of detected reverted commits when inspecting 20% of total changed LOC). In terms of the average performance across the ten studied projects, our model achieves an AUC-ROC of 0.756 and a cost-effectiveness of 0.746, significantly improving the baselines by substantial margins. In addition, we found that "developer" is the most discriminative dimension among the three dimensions of features for the identification of reverted commits. However, using all the three dimensions of commit features leads to better performance.

Keywords Reverted Commits · Identification Model · Feature Engineering · Empirical Study

1 Introduction

As a software system evolves, its source code is modified for different purposes, e.g., bug fixing, feature implementation, and refactoring. Developers modify the source code by submitting commits to Version Control Systems (VCSs). However, when developers submit a commit, it is unrealistic to expect them to always complete the whole task on the first attempt without introducing any problems (Yoon and Myers, 2012).

In practice, a popular and coarse-grained approach for recovering from a problematic commit is to revert it (Shimagaki et al., 2016). Reverting a commit means undoing the change that the commit introduced. Prior studies show that 75% of developers feel that a backtracking tool (i.e, revert back to an earlier state by removing inserted code or by restoring removed code) is necessary (Yoon and Myers, 2012). Additionally, different version control systems, such as Git, Mercurial and Subversion, provide built-in revert commands. Developers often use version control systems to back out problematic commits by reverting to a previously, known-to-be working system state (Codoban et al., 2015).

Recently, Shimagaki et al. (2016) conducted an empirical study on 6 projects (including 4 open source projects and 2 industrial projects) to understand why commits are reverted. The main findings of their work include: (1) the percentages of reverted commits range from 1% to 5%; (2) those commits that are eventually reverted linger within the codebases for 1-35 days (median); (3) there are various reasons to revert a commit, including internal reasons (e.g., compilation error or incomplete fix) and external reasons (e.g., temporary workaround or unnecessary feature).

Although revert is a pervasive command for recovering from a problematic commit, reverted commits could induce some issues for software development and maintenance. For example, reverted commits exist for a long time within the codebase (as long as 811 days with a median of 1-35 days) potentially impeding the development and maintenance progress. The longer a commit takes to be reverted, the more difficult developers try to switch to the context of the reverted commit (Souza et al., 2015; Shimagaki et al., 2016). Additionally, by the time the commit was reverted, many other commits

might have depended on it. This would increase the difficulty of software maintenance (Souza et al., 2015).

In order to mitigate this issue, we set out to explore the following central question: *Can we characterize and identify which commits will be reverted?* To the best of our knowledge, this is the first work to address this question. The benefits of our study are as follows:

- Characterizing reverted commits by various commit features can help to improve an understanding of this phenomena, in turn helping us identify typical root causes of reverted commits and suggesting ways to mitigate them.
- Our Just-In-Time (JIT) identification of reverted commits informs developers about such commits when they are still fresh on their minds (Kamei et al., 2013). Early detection of problematic commits can save developer's effort and time for software development and maintenance (Souza et al., 2015).

The typical usage scenario of our reverted commit identification tool is to warn developers about commits that are likely to be reverted. As a result, developers can check these commits carefully at an early stage (e.g., soon after submitting them into the code repository). For example, suppose Bob is a developer in a large project team, Bob submitted a possibly problematic commit on one day.

Without our tool, the commit is reverted after one month (even worse just a day before the release of a new version). Many other source code files (i.e., are changed during the month with many of these changes depending on the code in Bobs just-reverted commit) should also be revisited or even revised. This would lead to lose efforts, increased maintenance costs, and many last minute surprises and fire-fighting in a project.

With our tool, the commit is flagged with a high likelihood score among the commits submitted in a recent period (e.g., several days). Bob or other related developers carefully check the commit after receiving the warning by our tool. As a result, they discovered the problem and fixed it by submitting another commit or reverting this commit early on. Eventually, the development effort can be saved or the possible maintenance difficulty (that is introduced by reverting these commits after a long time) can be mitigated.

Therefore, in this paper, we propose an automated model for JIT identification of commits that will be reverted. To achieve this goal, we extract 27 commit features which are grouped into three dimensions: change, developer and message. In the change dimension, we extract commit features by measuring the code differences that are made in the commit (e.g., number of modified subsystems and files) (Kamei et al., 2013). In the developer dimension, we extract the developer features by analyzing the historical activities of developers (e.g., number of previously submitted commits of the developer who submits a commit). In the message dimension, we extract the message features by analyzing the textual description of the commit message (e.g., whether or not the change is a bug fix).

Next, we build our identification model by using a random forest classifier on these extracted features. To evaluate the effectiveness of our identification model, we perform experiments on ten open source projects from different application domains with a total of 125,241 commits, namely, Hadoop, Gerrit, Hbase, Karaf, Jenkins, Spring-boot, Hive, Eclipse Platform, Egit and Eclipse JDT. In our evaluation, we adopt two performance measures (i.e., AUC-ROC and cost-effectiveness) using a 10 times 10-fold cross-validation setting. AUC-ROC is the area under the receiver operator characteristic (ROC) curve (Huang and Ling, 2005). In the ROC curve, the true positive rate (TPR) is plotted as a function of the false positive rate (FPR) across all thresholds. Cost-effectiveness evaluates the performance of a model given a certain cost threshold, e.g., a certain percentage of code to inspect (e.g., 20% of total changed LOC). It is calculated by measuring the recall considering the inspection under limited resources. In practice, when a team has limited resources to inspect changed LOC in potentially reverted commits, it is crucial that manually inspecting the top percentages of commits that are likely to be reverted can help developers discover as many problematic commits as possible. In our study, by default, we define cost-effectiveness as the number of reverted commits that can be discovered by inspecting the top 20% of totally changed LOC based on the confidence levels that a model outputs (Kamei et al., 2013; Xia et al., 2016a).

Our experimental results show that our proposed model achieves an average of AUC-ROC of 0.756, and an average of cost-effectiveness of 0.746 across the ten studied projects. In addition, we implement two baselines, Random Guess (RG) and Naive Bayes Multinomial based on the textual description of the Commit Message (NBMCM). RG identifies reverted commits randomly. NBMCM identifies reverted commits by building a Naive Bayes Multinomial classifier on commit messages. As a result, our proposed model improves over RG and NBMCM by 51.29% and 17.19% in terms of average AUC-ROC, and by 216.94% and 108.95% in terms of average cost-effectiveness respectively. Additionally, in order to understand what commit features impact reverted commit most, we perform a feature importance analysis.

In summary, the main contributions of this paper are as follows:

1. We propose the problem of identifying commits that will be reverted, and we propose a model to automatically identify such commits. Our proposed model leverage 27 commit features. To the best of our knowledge, this paper is the first work to address this problem.
2. We evaluate our proposed model on ten projects with a total of 125,241 commits. Our experimental results show that out approach achieves an average AUC-ROC of 0.756, an average cost-effectiveness of 0.746 across the ten studied projects, and a significant improvement over two baseline approaches.
3. We investigate the most discriminative dimension of features for identifying commits that are likely to be reverted. Our experimental results show

that "developer" is the most discriminative dimension among the three dimensions of features.

Paper Organization. The remainder of this paper is structured as follows. Section 2 presents the study data of our study. Section 3 presents our empirical study setup, including research questions, studied features, used classifiers, validation setting and used performance measures. In Section 4, we detail and investigate our experimental results and their analysis. In Section 5, we discuss three more findings that impact our model for identifying commit that will get reverted. Section 6 describes the threats to validity. Section 7 presents the related work of our study, including commits rework and identification models for software commits. At last, in Section 8, we draw a conclusion of this paper.

2 Empirical Study Dataset

This section presents the details of the used dataset in this study. We first describe the summary of studied open source projects. Then, we describe the methodology to identify reverted commits.

2.1 Dataset

Table 1 lists basic statistics about the studied projects. In total, we study 10 Java open source projects namely Hadoop, Gerrit, Hbase, Karaf, Jenkins, Spring-boot, Hive, Eclipse Platform, Egit and Eclipse JDT. The projects cover different application domains. Hadoop[1] is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Gerrit[2] is a free, web-based team code collaboration tool. Hbase[3] is the Hadoop database, a distributed, scalable, and big data store. Karaf[4] is a lightweight, powerful, and enterprise ready container powered by OSGi. Jenkins[5] is a self-contained, open source automation server which can be used to automate all sorts of tasks that are related to building, testing, and deploying software. Spring-boot[6] is a framework that eases the bootstrapping and development of new Spring applications. Hive[7] is a data warehouse software which facilitates the reading, writing, and managing of large datasets that reside in distributed storage using SQL. Eclipse Platform[8] defines the set of frameworks and common services that collectively make up

[1] http://hadoop.apache.org/
[2] https://www.gerritcodereview.com/
[3] http://hbase.apache.org/
[4] http://karaf.apache.org/
[5] https://jenkins.io/index.html
[6] http://projects.spring.io/spring-boot/
[7] https://hive.apache.org/
[8] https://projects.eclipse.org/projects/eclipse.platform

Table 1: Summary of Studied Projects

Project	**#Commits**	**#Reverted**	**Reverted Ratio**	**#Reverted && Defective**
Hadoop	14,564	554	3.80%	172
Gerrit	19,474	285	1.46%	133
Hbase	12,366	576	4.66%	229
Karaf	5,246	130	2.48%	28
Jenkins	18,253	220	1.21%	91
Spring-boot	9,940	211	2.12%	100
Hive	9,067	381	4.20%	60
Eclipse Platform	8,051	121	1.50%	42
Egit	5,310	107	2.02%	43
Eclipse JDT	22,970	586	2.55%	325
Total	***125,241***	***3,171***	***2.53%***	***1,223***

infrastructure required to support the use of Eclipse as a component model. Egit[9] is an Eclipse Team provider for the Git version control system. Eclipse JDT[10] provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins.

2.2 Reverted Commit Identification

A reverted commit referred to as a commit that is being reverted by a future commit (refers to as reverting commit). We identify reverted commits in two methods:

The first method is identifying reverted commits by checking commit messages. This identification method is conducted by following prior study (Shimagaki et al., 2016). This identification method consists of two steps:

Step 1: Identification of reverting commits. By following Shimagaki et al. (2016), we identify reverting commits by scanning git commit log. Since Git uses a fixed string pattern to mark reverting commits, we use the following regular expression to identify reverting commits:

*ˆRevert \".*This reverts commit ([0-9a-f]{40}).**

Step 2: Identification of reverted commits. Each reverting commit contains a SHA-1 ID in the Git log. This SHA-1 ID refers to the commit that is being reverted. In this way, we can label the commits that are reverted.

However, this identification method can only identify reverted commits that follow a standard procedure of reverting and do not modify the commit message. Additionally, this identification method can only identify reverted commits that the entire commit is reverted. The reverting commits that do not follow a standard procedure or only revert a subset of the changed files are

[9] http://www.eclipse.org/egit/

[10] https://projects.eclipse.org/projects/eclipse.jdt

ignored. For example, it is possible to start reverting a commit by following a standard procedure and then modify part of file changes and the generated message before submitting.

Therefore, we propose a second identification method for identifying reverted commits by comparing the detailed changed content of each changed file. We define a reverted commit as: there is at least one changed source file whose inserted code and removed code is identical with a future commit's removed code and inserted code respectively. The detailed steps of this identification method are as follows:

Step 1: Get changed content of each commit. For each commit, we get the changed content using the *git diff* command and store them as a list of 3-element vectors in the format <*file, inserted, removed*>. We refer to such a 3-element vector as a *file change vector V*, in which "file" indicates the changed file path, "inserted" refers to the text in the inserted lines in this changed file, "removed" refers to the text in the removed lines in this changed file.

Step 2: Identify reverted commits. For a commit A, we compare each of its file change vectors with those of all commits submitted after A. If we can find any commit B which satisfies the following two conditions: (1) Both A and B have a file change vector (referred to as V_a and V_b, respectively) containing the same file F. (2) The inserted lines of V_a is identical with the removed lines of V_b, and the removed lines of V_a is identical with the inserted lines of V_b. Then we label commit A as a reverted commit, B is a reverting commit that reverts A.

We combine the identified reverted commits by the above-mentioned two methods as our final reverted commits. The second identification method can identify the reverted commits which did not follow a standard reverting procedure. For example, we notice that the commit *"9992cae54120d2742922745c1f513c6bfbde67a9"* in Hadoop project might be a reverting commit, since its commit message is *"Reverting the previous trunk merge since it added other unintended changes in addition"*. However, we cannot identify the corresponding reverted commits by just checking commit message using the first identification method, since its commit message does not contain the corresponding reverted commit id. Using our proposed identification second method, we identified that this commit reverted other commits, e.g., "d00b3c49f6fb3f6a617add6203c6b55f6c345940" and "83e4b2b46962ba2f799ea5c92aa328a5f01e21b7".

In this paper, we collect the commits of the selected projects from the creation date of the projects to September 30, 2017. Shimagaki et al. (2016) found that reverted commits linger within the version control system for a median of 1-35 days. Therefore, we use the commits until June 30, 2017 to ensure that most of the studied commits are correctly labeled. In total, there are 125,241 commits in ten projects, and 3,171 of them are reverted. The ratio of reverted commits is 2.53%. In practice, it is difficult to identify reverted commits due to the class imbalance phenomenon.

Additionally, one may be concerned that the identification of reverted commits is similar to the identification of defect-introducing commits (i.e., just-in-time defect identification (Kamei et al., 2013)). A prior study has shown that commits can be reverted due to various reasons beyond the introduction of bugs (Shimagaki et al., 2016). To further clarify the difference, we compute the proportion of reverted commits in our dataset that also introduce defects. The defect-introducing commits are identified by using the commit-guru tool (Rosen et al., 2015)[11]. The commit-guru tool performs the following main steps: first, it semantically analyzes each commit message made by the developer in order to classify the commit type (e.g., corrective, feature addition) (Hindle et al., 2008). Second, it identifies the defect-introducing commits by linking them to those that are fixing commits. The linking step is performed by using a variant of the SZZ algorithm (Śliwerski et al., 2005; da Costa et al., 2017).

As a result, the column "#Reverted && Defective" in Table 1 presents the number of commits that are both being reverted and defect-introducing. It shows that only a small proportion of reverted commits are also defect-introducing commits. Thus, we believe that the identification of reverted commits is a different problem than the identification of defect-introducing commits, although a defect introducing commit is one potential reason for the reversal of a commit.

3 Empirical Study Setup

In this section, we present our experimental setup, including studied features, used classifier, validation setting, used evaluation measures and research questions.

3.1 Studied Features

We extract a total of 27 commit features, which are divided into three unique dimensions: change, developer and message. All these features are derived from the source code repository control system (e.g., Git). Table 2 presents a summary of extracted features. We choose these features because: these features are used in many prior studies of commit analysis, such as defective change identification (Mockus and Weiss, 2000; Kim et al., 2008; Kamei et al., 2013; Yang et al., 2016) or build co-change identification (McIntosh et al., 2014; Xia et al., 2015a). We present below details of the features in each dimension.

Change: The change dimension contains commit features by measuring the code changes that are made in the commit. This dimension is one of the most important dimension for identifying defective changes (Kamei et al., 2013). Here, we conjecture that the change dimension can also be leveraged

[11] http://commit.guru/

Table 2: Studied Features

Dimension	Name	Definition	Rationale
Change	NS	Number of modified subsystems	Commits touching more subsystems may be more likely to be reverted.
	ND	Number of modified directories	Commits touching more directories may be more likely to be reverted.
	NMF	Number of modified files	Larger NFM, NFA or NFD values indicate that a commit touched more files. Commit touching more files may be more likely to be reverted.
	NAF	Number of added files	
	NDF	Number of deleted files	
	NUC	Number of unique changes to the modified files	A larger NUC indicates a larger spread of modified files. Such a commit may be more likely to be reverted, since a developer will have to recall and track many previous commits.
	LA	Lines of added code	Larger LA or LD indicates that commit touched more lines of code (LOC). Commits touching more LOC may be more likely to be reverted.
	LD	Lines of deleted code	
	Entropy	Distribution of modified code across each file	Commits with high entropy may be more likely to be reverted, since a developer will have to recall and track more scattered commits across each file.
	NLC	Number of low significance code changes	The significance level expresses how strongly a code change (a commit consists of many fine-grained code changes) may impact other source code entities and whether a commit may be functionality modifying or functionality preserving(Fluri et al., 2007). Commits with different significant code changes may impact the likelihood of a commit being reverted.
	NMC	Number of medium significance code changes	
	NHC	Number of high significance code changes	
	NCC	Number of crucial significance code changes	
Developer	NDEV	Number of developers that changed the modified files	A larger NDEV indicates that the changed files have been previously touched by a large number of developers. A commit with a larger NDEV may be more likely to be reverted, since different developers have different design thoughts and coding styles.
	EXP	Developer experience	A commit with a lower EXP may be more likely to be reverted. A developer that has often modified the files in recent months may be less likely to submit a commit that will be reverted. A developer that is more familiar with the subsystems modified by a commit may be less likely to submit a commit that will be reverted.
	REXP	Recent developer experience	
	SEXP	Developer experience on a subsystem	
	MINO	Min proportion of ownership of modified files	A larger proportion of ownership of a developer for a file indicates the developer has more responsibility on the file. In a commit, if the submitting developer has a higher proportion of ownership to the modified files, the commit is less likely to be reverted. Because the developer who submit the commit is more familiar and responsible for the modified file.
	AVEO	Average proportion of ownership of modified files	
	MAXO	Max proportion of ownership of modified files	
Message	msg_length	Message length: number of words in the message	The change message contains the purpose of this commit. Past studies have found that the commit purpose has an impact on it being reverted, such as fixing bugs, introducing bugs, and refactoring (Shimagaki et al., 2016). We conjecture that the commit purpose has an impact on it being reverted. In addition, we convert the textual factors of commit messages into a numerical value using a classifier, namely Naive Bayes Multinomial score (i.e., nbm_msg_score) (McCallum et al., 1998; Xia et al., 2016b). A higher score indicates that the commit is more likely to be reverted. We calculate such score of training set and testing set separately. In terms of training set, we first split training set into two subsets. Then we train a classifier with a training subset, and use it to obtain the textual scores for commits on the other training subset. In terms of testing set, we train a classifier built on all of the commits in the training set, and use it to obtain such scores for commits on testing set.
	has_bug	Whether message of this change contains the word "bug"	
	has_feature	Whether message of this change contains the word "feature"	
	has_improve	Whether message of this change contains the word "improve"	
	has_document	Whether message of this change contains the word "document"	
	has_refactor	Whether message of this change contains the word "refactor"	
	msg_nbm_score	Naive Bayes Multinomial likelihood score of a commit likelihood to be reverted	

to determine the likelihood of identifying commits being reverted. Change dimension consists of 13 features.

NS represents the number of modified subsystems and *ND* represents the number of modified directories. The root directory name is identified as the subsystem name and the directory name is identified to calculate the modified directories. For example, in Hbase, if a commit modifies a file with the path "hbase-client/src/.../hbase/Chore.java", then the subsystem is "hbase-client", and the directory name is "hbase-client/src/.../hbase".

Entropy aims to measure the distribution of the commits across the different files (Hassan, 2009). We compute entropy of a commit as: $H(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k)$, where probability $p_k \geq 0$ and it indicates the proportion that $file_k$ is modified in a commit (i.e., modified lines in $file_k$ respects to total modified lines of a commit), thus, $(\sum_{k=1}^{n} p_k) = 1$. For example, a commit modifies three files, A, B, and C with modified lines 30, 20, and 10, respectively, The Entropy is measured as 1.46 by using the formula: $= -(\frac{30}{60}log_2\frac{30}{60} + \frac{20}{60}log_2\frac{20}{60} + \frac{10}{60}log_2\frac{10}{60})$. The formula for Entropy above has been normalized by the maximum Entropy $log_2 n$ to account for differences in the number of files across changes, similarly to Hassan (2009). The higher the normalized Entropy, the larger the spread of a change.

In addition to extract commit features on the modified subsystems and directories, we also extract features at more fine-grained level, e.g., file-level and line-level. At file-level, we extract four features, (i.e., *NAF, NMF, NDF* and *NUC*). The features *NAF, NMF, NDF* represent the number of added, modified and deleted files respectively. These features aim to measure different activities on modified files and are used in prior studies (McIntosh et al., 2014; Xia et al., 2015a). *NUC* indicates number of unique commits to the modified files before. For example, if a developer Bob submitted a commit that modified files A, B, and C, file A was previously modified in commit α, and files B and C were modified in commit β, then *NUC* would be 2 (i.e., α and β).

At the line-level, we extract two features, namely the lines of added code (i.e., *LA*) and lines of deleted code (i.e., *LD*). These features describe the size of a commit, i.e., the number of renewed LOCs in the commit. We can directly measure them from source control repository. We measure *NS, ND, Entropy, LA* and *LD* as described by Kamei et al. (2013).

Additionally, we extract 4 significance features, namely the number of low, medium, high, and crucial significance level code changes (i.e., *NLC, NMC, NHC* and *NCC*). These features aim to capture the fine-grained activities for each commit (one commit may contain many source code changes). We adopt the significance level for each code change proposed by prior studies (Fluri and Gall, 2006; Fluri et al., 2007) which classify each code change type with a significance level (i.e., low, medium, high and crucial). The significance level expresses how strong a code change may impact other source code entities (i.e., how likely other source code entities have to be changed). For example, code changes in a method body are considered to have a low or medium significance level, whereas code changes on the interface of a class have a high or crucial

significance level (Fluri and Gall, 2006). We measure the *NLC, NMC, NHC* and *NCC* features by using the ChangeDistiller framework (Fluri et al., 2007).

Developer: The developer dimension aims to capture the historical activities related to developers, such as developer's experience and ownership of modified files. Prior work observed that developer's experience and ownership to files have an impact on software quality (Kamei et al., 2013; Bird et al., 2011). Here, we conjecture that the developer dimension can also be leveraged to determine the likelihood of commits being reverted. This dimension consists of 7 features.

NDEV measures the number of developers that previously touched the modified files. For example, if a developer Bob submitted a commit that modified files A, B, and C. A was previously modified by Jim. B and C were previously modified by Mike. Then the *NDEV* would be 2 (Jim + Mike). A higher *NDEV* means that the changed files have been previously touched by more developers.

EXP measures the experience of a developer in a project. This feature is calculated by counting the number of previous submitted commits of a developer in a project. Higher *EXP* means the developer have higher experience. Recent experience (REXP) measures the experience of a developer by considering the age of historical commits. This feature is measured as the total experience of the developer in terms of commits, weighted by their age. It gives a higher weight to commits that are more recent. In detail, we use the following weighting scheme to measure REXP: $\frac{1}{n+1}$, where n is measured in years. For example, if a developer of a commit submitted three commits in the current year, four commits one year ago, and three commits two years ago, then REXP is 6 (i.e., $= \frac{3}{1} + \frac{4}{2} + \frac{3}{3}$). Subsystem experience (SEXP) measures the number of commits the developer submitted previously to the subsystems that are modified by the current commit. We measure *NDEV, EXP, REXP* and *SEXP* as described by Kamei et al. (2013).

Additionally, we extract 3 features related to ownership of the developer on the modified files of a commit, namely the *minimum, average* and *maximum* proportion of ownership on modified files in a commit (i.e., *MINO*, *AVEO* and *MAXO*). Ownership is used to describe whether one developer has responsibility for a software component (Bird et al., 2011). The proportion of ownership of a developer for a file is the ratio of number of commits that the developer has made relative to the total number of commits for the file. We measure the proportion of ownership as described by Bird et al. (2011). For example, if Bob has made 20 commits to "a.java", and there are a total of 100 commits to "a.java", then the proportion of ownership of Bob to "a.java" is 20%. Since a commit may modify many files, we calculate minimum, average and maximum proportion of ownership to capture the ownership features of a commit.

Message: Message is the commit log that is written by submitting developer when submitting a commit. The message dimension consists of 7 features. These features are calculated based on the commit messages.

The features *has_bug*, *has_feature*, *has_improve*, *has_document* and *has_refactor* aim to describe the purpose of a commit. Prior work observed that the commit message can indicate the purpose of a commit (Fu et al., 2015; Yan et al., 2016) and grouped them into six categories: fixing bug, adding feature, improvement, documentation, refactoring and other (Herzig et al., 2013). We identify these categories by simply checking whether the message contains the related words as shown in Table 2 (Mockus and Votta, 2000; Hassan, 2008). And we use the *msg_length* to represent the length of the commit message by counting the number of words.

Additionally, we convert the textual factors of the commit messages into a numerical value (i.e., *msg_nbm_score*) using a classifier (i.e., Naive Bayes Multinomial). Since prior work found that the commit message has an impact on reverted commits (Shimagaki et al., 2016), we assume that analyzing the commit message can help on the identification of reverted commits. The msg_nbm_score is calculated using the following steps:

- **Pre-processing commit messages:** We pre-process the textual description of each commit message by tokenizing, stop-word removal, and stemming. We first tokenize each commit message into words, phrases, symbols, or other meaningful name element tokens. Then, we remove the stop-words such as "the", "I", "of" which provide little information for understanding the text. And we perform a stemming step to reduce inflected tokens to their stem, base or root form. We use the resulting textual tokens and count the number of times each token appears in a commit message.
- **Converting textual factors of commits in the training set into numerical values:** We split training set into two subsets by leveraging stratified random sampling, so that the distribution and number of reverted commits in both training subsets is the same (Valdivia Garcia and Shihab, 2014; Xia et al., 2016b). Then, we extract the textual factors from the training subsets, and create a word frequency table based on the extracted factors. We train a classifier with the first training subset, and use it to obtain the textual scores (i.e., confidence scores that a commit is to be reverted) for commits on the second training subset. We also train a classifier with the second training subset, and use it to obtain the textual scores on the first training subset. As a result, the likelihood values of all the training data are calculated.
- **Converting textual factors of commits in the test set into numerical values:** In the identification phase, given a new commit, we leverage the text mining classifiers built on all of the commits in the training set to convert the textual factors into a textual score. Note that we do not use the labels of the testing set.

In this paper, we leverage the Naive Bayes Multinomial (NBM) to build the text mining classifier. NBM is one of the variants of Naive Bayes algorithm which builds a classifier based on multinomially distributed data (McCallum et al., 1998). We adopt NBM as the default classifier for a commit message

since it is a simple text classification technique that have been used in many prior software text analysis studies (Xia et al., 2014, 2016b; Huang et al., 2017).

3.2 Performance Measures

We mainly consider two widely used performance measures, i.e., AUC-ROC and cost-effectiveness:

AUC-ROC: AUC-ROC represents the area under the Receiver Operating Characteristic (ROC) curve. In the ROC curve, the true positive rate (TPR) is plotted as a function of the false positive rate (FPR) across all thresholds. The value of AUC-ROC ranges from 0 to 1, and higher AUC-ROC values indicate a better performance. AUC-ROC is also widely used in many software engineering studies (Lessmann et al., 2008; Xia et al., 2015a; Nam and Kim, 2015). An AUC-ROC of 0.7 is considered as a promising performance score (Lessmann et al., 2008; Nam and Kim, 2015). We choose AUC-ROC as our performance measure because it is threshold independent and has a statistical interpretation (Bradley, 1997; Lessmann et al., 2008).

Cost-effectiveness: Cost-effectiveness measures the performance considering the limited inspection resources. It is calculated by measuring the recall considering the inspection under limited resources. It has been widely used for evaluating identification models that leverage software engineering data, such as software commits (Jiang et al., 2013; Kamei et al., 2013; Xia et al., 2016a; Yang et al., 2016). The main idea of cost-effectiveness is to simulates the practical usage of the proposed model. In practice, due to the limited resources, developers can only inspect a limited number of software commits. In our context, we consider the required effort to inspect those commits identified as reverted.

In detail, by following prior studies (Jiang et al., 2013; Kamei et al., 2013; Yang et al., 2016), the cost-effectiveness in our study denotes the recall of reverted commits when using 20% of the entire required effort to inspect all commits to inspect the top ranked commits produced by our model. And the total number of lines modified by a commit ($LA + LD$) as a measure of the required effort to inspect a commit.

3.3 Classifier

After data collection and feature extraction, we build our identification model by using a classifier. In default, we adopt Random Forest (RF) as our classifier and we use its implementation in Weka (Hall et al., 2009). Random Forest is an ensemble approach that is specifically designed for decision tree classifier (Breiman, 2001). The basic idea behind random forest is to combine multiple decision trees for classification. Each decision tree is built by using a random subset of the extracted features. The advantages of random forest

are: 1) it is generally highly accurate and the importance of features can be captured automatically; 2) Since random forest unifies many trees that are learned differently, it can mitigate overfitting problems and is not sensitive to outliers.

3.4 Validation Settings

To evaluate the effectiveness of our identification model, we adopt a widely used validation setting, namely 10-fold cross-validation. In detail, we perform a 10 times stratified 10-fold cross-validation. In each stratified 10-fold cross validation, we randomly divide the dataset into ten folds by using stratified random sampling. The objective of stratified random sampling technique is to keep the class distribution of each fold the same as the original dataset. Then, nine folds are used to train the classifier, while the remaining one fold is used to evaluate the performance of the built model. This process is repeated 10 times, so that each fold is used exactly once as the testing set. We perform 10-fold cross validation 10 times to reduce the bias. As a result, there are 100 performance values for each project and we report the average of the 100 values for each performance measure.

3.5 Research Questions

In this study, we are interested in answering the following four research questions:

RQ1: Can we effectively identify the commits that will be reverted?

RQ2: How effective is our model when built on a subset of commit features?

RQ3: Which commit features are most discriminative for identifying reverted commits?

RQ4: How effective is our approach when considering more evaluation measures?

In RQ1, we evaluate the effectiveness of our approach and compare it with baseline models in terms of AUC-ROC and cost-effectiveness. In RQ2, we investigate how effective our identification model is when built on a subset of commit features (i.e., each dimension of features). In RQ3, we examine the most discriminative commit features for identifying reverted commits. In RQ4, we evaluate our approach and baselines considering more evaluation measures.

4 Results

In this section, we provide the results of the aforementioned three research questions. To answer RQ1, we conduct an experiment on ten projects to evaluate the effectiveness of the proposed identification model and compare it

Table 3: The AUC and cost-effectiveness for our model (Ours) compared with the baselines. The best performance values are highlighted in bold.

Project	**AUC-ROC**			**Cost-effectiveness**		
	RG	**NBMCM**	**Ours**	**RG**	**NBMCM**	**Ours**
Hadoop	0.500	0.683	**0.732**	0.301	0.456	**0.839**
Gerrit	0.500	0.577	**0.775**	0.242	0.387	**0.733**
Hbase	0.500	0.671	**0.781**	0.242	0.265	**0.863**
Karaf	0.500	0.630	**0.734**	0.168	0.410	**0.746**
Jenkins	0.500	0.671	**0.765**	0.245	0.366	**0.743**
Spring-boot	0.500	0.741	**0.782**	0.238	0.372	**0.722**
Hive	0.500	0.634	**0.768**	0.243	0.333	**0.796**
Platform	0.500	0.579	**0.717**	0.282	0.259	**0.625**
Egit	0.500	0.666	**0.797**	0.169	0.488	**0.693**
JDT	0.500	0.604	**0.715**	0.223	0.235	**0.700**
Average	*0.500*	*0.646*	***0.756***	*0.235*	*0.357*	***0.746***
Improved	*51.29%*	*17.19%*		*216.94%*	*108.95%*	

Table 4: Adjusted P-values and Cliff's Delta comparing AUC-ROC and cost-effectiveness scores for our approach with baselines.

Project	**AUC-ROC**		**Cost-effectiveness**	
	RG	**NBMCM**	**RG**	**NBMCM**
Hadoop	1.00 (L) ***	0.64 (L) ***	1.00 (L) ***	0.97 (L) ***
Gerrit	1.00 (L) ***	1.00 (L) ***	1.00 (L) ***	0.90 (L) ***
Hbase	1.00 (L) ***	0.96 (L) ***	1.00 (L) ***	1.00 (L) ***
Karaf	1.00 (L) ***	0.72 (L) ***	1.00 (L) ***	0.94 (L) ***
Jenkins	1.00 (L) ***	0.85 (L) ***	0.99 (L) ***	0.94 (L) ***
Spring-boot	1.00 (L) ***	0.50 (L) ***	1.00 (L) ***	0.97 (L) ***
Hive	1.00 (L) ***	0.97 (L) ***	1.00 (L) ***	1.00 (L) ***
Platform	1.00 (L) ***	0.80 (L) ***	0.97 (L) ***	0.90 (L) ***
Egit	1.00 (L) ***	0.70 (L) ***	1.00 (L) ***	0.58 (L) ***
JDT	1.00 (L) ***	0.96 (L) ***	1.00 (L) ***	1.00 (L) ***
W/T/L	*0/0/10*	*0/0/10*	*0/0/10*	*0/0/10*

***p<0.001, **p<0.01, *p<0.05, –p>0.05

L, M, and S represent Large, Medium and Small effect size according to Cliff's delta

with two baseline approaches. To answer RQ2, we conduct three experiments by considering three dimensions (i.e., change, developer and message) of our features respectively. To answer RQ3, we perform feature importance analysis, which consists of three steps: correlation analysis, redundancy analysis and importance feature identification.

4.1 RQ1: Can we effectively identify the commits that will be reverted?

Motivation: Previous studies note that reverting commits can impede the development progress and increase the difficulty of maintenance (Souza et al., 2015; Shimagaki et al., 2016). In this RQ, we would like to investigate whether our proposed model can effectively identify reverted commits (i.e., that will likely be reverted in the future). The benefits of identifying such commits lie

in two aspects: first, the likely reverted commits can be highlighted early on (i.e., at check in time). Once a reverted commit is identified, we can provide a timely warning to the development team. Early identification of problematic commits can save developer's effort and time. Second, identifying the likely reverted commits can help to understand the commit context by tracing the commit features. Understanding the context of reverted commits can help to avoid likely reverted commits in the future.

Approach: To answer this RQ, we conduct an empirical study on ten open source projects with a total of 125,241 commits. We built our model using a Random Forest classifier and adopt 10 times stratified 10-fold cross validation to estimate the accuracy of our model. In this way, there will be 100 effectiveness values for each project. We report the average value and perform a statistical test on the 100 effectiveness values.

Additionally, we implement two baselines to compare with our model:

Baseline 1: Random Guess (RG). This baseline is usually used when there is no previous method for our research question (Xia et al., 2016b). In this baseline, it randomly determines a list of commit as reverted. Since we use a stratified 10-fold cross-validation, we implement the RG baseline according to the data distribution in training set. For example, the ratio of reverted commits in Hadoop is 3.80% in the training set of one fold, the RG baseline will randomly determines approximately 3.80% of commits in testing set as reverted. Note that the AUC-ROC of RG is always 0.5 (Xia et al., 2016b). The cost-effectiveness of RG depends on the order of commits in testing set. We randomly sort the commits in testing set, but always keep the commits which are determined as reverted by RG at the front of the order. To reduce the bias of randomly sorting, we repeat the RG 10 times to get the average performance.

Baseline 2: Naive Bayes Multinomial based on Commit Message (NBMCM). Since the commit messages may also indicate that a commit is likely to be reverted, this baseline is a text classification baseline based on the commit messages using a Naive Bayes Multinomial (NBM). The baseline is built by using following steps. First, we preprocess all commit messages by using the aforementioned preprocessing steps, including tokenization, stop-word removal and stemming. Second, we use the resulting textual tokens and count the number of times each token appears in each commit message. Third, we construct a classifier based on the textual representation by using the NBM classifier. Note that both baseline 1 and 2 are conducted with the same training and testing set as ours.

Besides, we conduct a statistical test to investigate whether the difference between the proposed model and the baseline is statistically significant. In particular, we adopt the Wilcoxon signed-rank test (Wilcoxon, 1992) with a Bonferroni correction (Abdi, 2007) at 95% significance level. The Wilcoxon signed-rank test is a non-parametric hypothesis test that is used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ.

A Bonferroni correction is used to counteract the problem of multiple comparisons. In addition, we also compute the Cliff's delta to measure the effect size. Cliff's delta is a non-parametric effect size measure that can evaluate the amount of difference between two approaches (Long et al., 2003). It defines a delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "Negligible", "Small", "Medium", "Large" effect size, respectively (Romano et al., 2006).

Results: Table 3 presents the AUC-ROC and cost-effectiveness comparison between our identification model (Ours) and the baselines (i.e., RG and NBMCM). We use "Improved." to represents the improvement ratio over the baseline, "Improved" is computed as $\frac{Ours-baseline}{baseline} * 100\%$.

Table 4 presents the adjusted p-values and Cliff's Delta comparing the AUC-ROC and cost-effectiveness scores for our approach with baselines. The row "W/T/L" reports the number of projects for which the corresponding model obtains a significantly better, equal, and worse performance than our model.

From the two tables, in terms of the average performance across the ten studied projects, our model achieves an AUC-ROC of 0.756 and a cost-effectiveness of 0.746, improves over the baseline RG by 51.29% and 216.94% in terms of AUC-ROC and cost-effectiveness respectively, improves over NBMCM by 17.19% and 108.95% in terms of AUC-ROC and cost-effectiveness respectively. In addition, all the improvements are statistically significant over the baselines across the ten projects in terms of both AUC-ROC and cost-effectiveness, and the effect sizes are large in all the cases.

In terms of the average performance across the ten studied projects, our model achieves an AUC-ROC of 0.756, and a cost-effectiveness of 0.746 significantly improving over the two baselines by a substantial margin.

4.2 RQ2: How effective is our model when built on a subset of commit features?

Motivation: We extracts three dimensions of commit features namely change, developer, and message. These features characterize commits along different aspects. Some aspects may be more important for identifying whether commits will be reverted. In default, our model uses three dimensions of features, we are also interested in which dimension of features is more important. To answer this RQ, we are interested in two questions: first, whether our model benefits from all dimensions of features; second, which dimension is the most discriminative for identifying reverted commits.

Approach: We build three models (one model for each dimension) and denote each model by the dimension name (i.e., change, developer, and message). In each dimension model, we keep the classifier (i.e., Random Forest) and validation setting (i.e., 10 times stratified 10-fold cross validation) as the same

Table 5: AUC-ROC and Cost-effectiveness for the three models that are built on three dimensions of features. The best performance between the three models built on a dimension of features is underlined. The performance highlighted in bold means that the performance is the best in comparison with our model (i.e., using all features).

Project	AUC-ROC				Cost-effectiveness			
	Change	Developer	Message	All	Change	Developer	Message	All
Hadoop	0.696	0.647	0.604	**0.732**	0.741	0.790	0.660	**0.839**
Gerrit	0.668	0.692	0.579	**0.775**	0.542	0.609	0.367	**0.733**
Hbase	0.691	0.731	0.593	**0.781**	0.748	0.812	0.619	**0.863**
Karaf	0.688	0.690	0.592	**0.734**	0.586	0.688	0.584	**0.746**
Jenkins	0.654	0.694	0.524	**0.765**	0.490	0.615	0.395	**0.743**
Spring-boot	0.698	0.701	0.585	**0.782**	0.535	0.645	0.524	**0.722**
Hive	0.684	0.704	0.546	**0.768**	0.717	0.772	0.542	**0.796**
Platform	0.653	0.640	0.504	**0.717**	0.590	0.509	0.614	**0.625**
Egit	0.774	0.709	0.707	**0.797**	0.644	0.620	0.538	**0.693**
JDT	0.675	0.656	0.523	**0.715**	0.633	0.617	0.669	**0.700**
Average	0.688	0.686	0.576	**0.756**	0.623	0.668	0.551	**0.746**

Table 6: Adjusted P-values and Cliff's Delta comparing AUC-ROC and cost-effectiveness scores of the models built using a particular dimension of features against the model with all the features (Ours).

Project	AUC-ROC			Cost-effectiveness		
	Change	Developer	Message	Change	Developer	Message
Hadoop	0.51(L)***	0.89(L)***	0.98(L)***	0.50(L)***	0.29(S)***	0.79(L)***
Gerrit	0.88(L)***	0.75(L)***	1.00(L)***	0.81(L)***	0.59(L)***	1.00(L)***
Hbase	0.92(L)***	0.68(L)***	1.00(L)***	0.75(L)***	0.39(M)***	0.98(L)***
Karaf	0.30(S)***	0.31(S)***	0.79(L)***	0.60(L)***	0.25(S)***	0.63(L)***
Jenkins	0.82(L)***	0.59(L)***	1.00(L)***	0.91(L)***	0.64(L)***	0.97(L)***
Spring-boot	0.73(L)***	0.72(L)***	0.99(L)***	0.81(L)***	0.41(M)***	0.80(L)***
Hive	0.86(L)***	0.71(L)***	1.00(L)***	0.53(L)***	0.18(S)**	0.96(L)***
Platform	0.46(M)***	0.52(L)***	0.96(L)***	0.16(S)-	0.45(M)***	0.06(N)-
Egit	0.14(N)*	0.55(L)***	0.56(L)***	0.18(S)**	0.28(S)***	0.53(L)***
JDT	0.61(L)***	0.79(L)***	1.00(L)***	0.49(L)***	0.57(L)***	0.20(S)**
W/T/L	*0/0/10*	*0/0/10*	*0/0/10*	*0/1/9*	*0/0/10*	*0/1/9*

***p<0.001, **p<0.01, *p<0.05, –p>0.05

L, M, S and N represent Large, Medium, Small and Negligible effect size according to Cliff's delta

with our all-dimensions model. In the comparison, in order to confirm a fair comparison, we keep the commits in the training and testing sets as the same with our all-dimensions model.

Additionally, in order to investigate whether the difference between our all-dimensions model and the single dimension models that is statistically significant, we also adopt the Wilcoxon signed-rank test with a Bonferroni correction at a 95% significance level and compute the Cliff's delta to measure the effect size.

Result: Tables 5 present the AUC-ROC and cost-effectiveness comparison for our model (in column "All features") with three models that are built on

each dimension of features (in column "Change", "Developer", and "Message", respectively). The best performing model across the three dimensions is underlined for each studied project. We highlight the best performance among the four columns in bold.

The results show that the "Developer" dimension model outperforms the "Change" and "Message" dimension model in most of the studied projects. In terms of the average AUC-ROC, "Change" and "Developer" models achieve a comparable performance and outperform "Message" model. In terms of average cost-effectiveness, the most discriminative dimension is "Developer". Our all-dimensions model achieves the best in terms of both AUC-ROC and cost-effectiveness in all studied projects.

Table 6 present the adjusted p-values and Cliff's Delta comparing the AUC-ROC and cost-effectiveness scores for our all-dimensions model with the three single dimension models. The row "W/T/L" reports the number of projects for which the corresponding single dimension model obtains a significantly better, equal, and worse performance than our all-dimensions model.

From the table, in terms of AUC-ROC and cost-effectiveness, we observe that the improvements of our all-dimensions model over the three single dimension models are statistically significant (p-value$<$0.05) with a non-negligible effect size in most cases. There is one case without a non-negligible effect size in the Egit project for the "Change" dimension in terms of AUC-ROC. There is one case without a statistically significant difference in the Platform project for the "Change" and "Message" dimensions.

"Developer" is the most discriminative dimension among the three dimensions of features for identifying reverted commits. However, using all three dimensions of commit features is better.

4.3 Which commit features are most discriminative for identifying reverted commits?

Motivation: There are 27 commit features in our identification model. In addition to find the most discriminative dimension of features for identifying reverted commits, we are also interested in understanding which commit features are most discriminative in identifying the reverted commits. For developers, knowing which features are most discriminative would help them avoid such commits.

Approach & Results: Our feature importance analysis consists of three steps as proposed by prior studies (Tian et al., 2015; Tantithamthavorn et al., 2015; Kabinna et al., 2016; Li et al., 2017).

Step 1: Correlation Analysis. This step aims to reduce collinearity among the features. For each project, this step will compute the correlations among the features by using a variable clustering analysis approach implemented in

the R package *Hmisc*[12]. As a result, the approach will produce a hierarchical overview of the correlations among all the features. The correlated features are grouped into sub-hierarchies. To remove correlated features, we use the same method as in prior work (Tian et al., 2015; Tantithamthavorn et al., 2015; Kabinna et al., 2016; Li et al., 2017). If the correlations of features in the sub-hierarchy are above 0.8, we try to keep the one that is easier to understand and calculate from each pair of highly-correlated features (Li et al., 2017). Additionally, we try to drop the same feature set for all the studied projects (Li et al., 2017).

After step 1, we remove 7, 10, 5, 4, 7, 11, 6, 5, 6, and 5 features in the Hadoop, Gerrit, Hbase, Karaf, Jenkins, Spring-boot, Hive, Platform, Egit and JDT projects, respectively. As a result, there are 20, 17, 22, 23, 20, 16, 21, 22, 21 and 22 features remaining in the Hadoop, Gerrit, Hbase, Karaf, Jenkins, Spring-boot, Hive, Platform, Egit and JDT projects, respectively.

Step 2: Redundancy Analysis. After reducing the collinearity among the features, this step aims to remove any redundant features that do not have a unique signal relative to the other features. In this step, we use the redun function in the rms[13] R package. After step 2, none of the remaining features are redundant in all the studied projects.

Step 3: Important Feature Identification. This step aims to determine the importance of each feature. We use the *bigrf*[14] R package to calculate the importance of each feature. This package leverages a random forest model with 10-times stratified 10-fold cross-validation to determine the most important features. The feature importance evaluation is based on an internal error estimate of a random forest classifier, which is called an "Out Of the Bag" (OOB) estimate (Wolpert and Macready, 1999). The key idea behind this approach is to check whether the OOB estimate will be reduced significantly or not when features are randomly permuted one by one.

We use the "importance" function in the R package "randomForest" to evaluate the importance of the features. In each run of the 10-fold cross-validation, we have 10 importance values for each feature. To determine which of the features are the most important, we apply the Scott-Knott Effect Size Difference (ESD) test for the importance values taken from all 10 iterations of the 10-fold cross-validation (Li et al., 2016; Tantithamthavorn et al., 2017; Xia et al., 2017). Note that the Scott-Knott ESD test is different from Scott-Knott test (Scott and Knott, 1974). The Scott-Knott test assumes that the data is normally distributed. This might cause that the created groups are trivially different from one another. The Scott-Knott ESD test corrects the non-normal distribution of an input dataset and merges adjacent groups that have a negligible effect size.

[12] http://cran.r-project.org/web/packages/Hmisc/index.html
[13] https://cran.r-project.org/web/packages/rms/rms.pdf
[14] http://cran.r-project.org/web/packages/bigrf/bigrf.pdf

Table 7: Importance of features as ranked according to the Scott-Knott ESD test

Hadoop		Gerrit		Hbase		Karaf	
Group	**Feature**	**Group**	**Feature**	**Group**	**Feature**	**Group**	**Feature**
1	LA	1	LA	1	REXP	1	NUC
2	LD	2	Msg_length	2	EXP	2	LD
	NUC	3	EXP		LA	3	EXP
3	NDEV		Msg_nbm_score	3	NUC	4	LA
4	EXP	4	LD	4	NDEV		ND
5	REXP	5	REXP		SEXP	5	NMF
6	MINO	6	Has_document	5	LD		NDEV
	ND	7	NDEV		AVEO	6	Msg_nbm_score
7	AVEO	8	ND	6	NMF		SEXP
8	Msg_length		NMF	7	ND	7	REXP

Jenkins		Spring-boot		Hive		Platform	
Group	**Feature**	**Group**	**Feature**	**Group**	**Feature**	**Group**	**Feature**
1	REXP	1	LD	1	NDEV	1	LD
2	EXP	2	NDEV	2	LA	2	LA
	LA	3	LA		LD	3	AVEO
3	NDEV	4	EXP	3	NUC		NLC
	NUC		NUC		ND	4	Entropy
4	AVEO	5	ND	4	REXP		Has_bug
	Msg_length	6	Msg_nbm_score		EXP	5	NMF
5	LD	7	Msg_length		SEXP	6	NMC
6	Msg_nbm_score		NMF	5	AVEO	7	EXP
7	ND	8	AVEO	6	MINO	8	Msg_length

Egit		JDT		All Projects	
Group	**Feature**	**Group**	**Feature**	**Group**	**Feature**
1	ND	1	LD	1	LA
	Msg_length	2	LA		REXP
2	NMF	3	AVEO	2	EXP
3	SEXP	4	NMF	3	LD
	EXP		EXP	4	NLC
	LD	5	Entropy	5	Msg_nbm_score
4	LA	6	NLC	6	NMC
5	Msg_nbm_score		Msg_length		SEXP
6	NMC	7	NDEV	7	NDEV
	Entropy	8	NMC		NMF

After step 3, Tables 7 present the group of top 10 most discriminative features as ranked according to Scott-Knott ESD test results in the studied ten projects, respectively.

Based on the results shown in Table 7, we observe that "LA", "LD", "EXP", "NDEV", "NUC" are the discriminative features for identifying reverted commits. In particular, we have the following findings:

1. "LA" and "LD" are ranked in the top 5 discriminative groups for all the studied projects. This suggests that the number of lines of added and the number of lines of deleted by a commit have a strong association with being reverted. We observe that commits with a large number of "LA" or "LD" are less likely being reverted. This may be observed since reverting

commits with a larger "LA" or "LD" may impact more classes, functions or commits which correlated with the reverted code. This would increase the difficulty for software maintenance. This finding also indicates that developer is cautious when reverting a commit.

2. "EXP" is ranked in the top 5 discriminative groups for 9 projects. This indicates that the number of commits submitted previously by the developer of a commit has a strong association with being reverted. We observe that commits submitted by more experienced developers are less likely being reverted. This may be observed since less experienced developers may have a higher likelihood to submit a problematic commit which are eventually reverted.
3. "NDEV" is ranked in the top 5 discriminative groups for 6 projects. This suggests that the number of developers that previously changed the modified files by the commit has a strong association with being reverted. This may be observed since files are changed by more developers tend to be more complex and defect-prone. It may take more effort to fix a problematic commit related to these files compared with reverting it.
4. "NUC" is ranked in the top 5 discriminative groups for 6 projects. This suggests that the number of unique commits to the modified files previously has an association with being reverted. Similar to "NDEV", this may be observed since files are changed by more commits tend to be more complex and defect-prone. It may take more effort to fix a problematic commit related to these files compared with reverting it.

Additionally, we also conducted the feature importance analysis by combining all the dataset of ten projects without removing any features. The last column of Table 7 presents the top 10 most discriminative features on the entire combined dataset. This column shows that "LA", "EXP" and "LD" rank the top 3 groups similar with the above-mentioned findings. However, there are some differences as compared with the results of project-by-project analysis. The reason is that we removed some features due to collinearity. For example, REXP ranks at the first group on the combined dataset, but REXP is removed in 6 projects (Hadoop, Gerrit, Jenkins, Spring-boot, Platform, and JDT) due to the high collinearity with EXP. Note that we conducted the feature importance by each project because different projects may have different data distribution and collinearity. This project-specific analysis method is also used by many prior studies (Li et al., 2016; Fan et al., 2018b; Yan et al., 2018; Fan et al., 2018a; Li et al., 2018). Although the result of feature importance may be different in different projects, we aim to identify important features shared by most of our studied projects.

In summary, the features "LA", "LD", "EXP", "NDEV", and "NUC" are the most discriminative features for identifying reverted commits.

4.4 RQ4: How effective is our approach when considering more evaluation measures?

Motivation: In previous research questions, we evaluate our approach using AUC-ROC and cost-effectiveness. These two measures evaluate the performance of prioritization. Actually, we aim to prioritize commits by classifying the relevant ones. Thus, we now investigate the effectiveness of our approach when considering additional classification measures, including AUC-PR, True Positive Rate (TPR) and True Negative Rate (TNR). AUC-PR represents the area under the precision-recall (PR). Since our dataset is highly skewed, when dealing with highly skewed datasets, Precision-Recall(PR) curves give a more informative picture of an algorithm's performance (Davis and Goadrich, 2006). TPR and TNR measure the proportion of actual positives and negatives that are correctly identified. In our use case, TPR measures to what extent we can save the development effort by identifying reverted commits, TNR measures to what extent we can avoid false alarm by identifying not-reverted commits.

Approach: For each commit, there would be 4 possible classification outcomes: a commit is classified as reverted when it is truly reverted (true positive, **TP**); it can be classified as reverted when it is truly not-reverted (false positive, **FP**); it can be classified as not-reverted when it is truly reverted (false negative, **FN**); it can be classified as not-reverted when it is truly not-reverted (true negative, **TN**). Based on TP, FP, FN and TN, we calculate TPR, TNR and AUC-PR to evaluate the effectiveness of proposed approach as follows:

True Positive Rate (TPR): TPR (also called the recall for the positive class) measures the proportion of actual positives that are correctly identified. In our context, TPR measures the percentage of reverted commits that are correctly identified as reverted. TPR is calculated as: $TP/(TP + FN)$.

True Negative Rate (TNR): TNR measures the proportion of actual negatives that are correctly identified. In our context, TNR measures the percentage of not-reverted commits that are correctly identified as not reverted. TNR is calculated as: $TN/(TN + FP)$.

AUC-PR: AUC-PR represents the area under the precision-recall (PR) curve. In the PR curve, it plots Recall on the x-axis and Precision on the y-axis. Recall is the same as TPR, precision measures that fraction of instances classified as positive that are truly positive. The AUC-PR of RG is equal to precision of RG, i.e., ratio of truly identified reverted commits among all the commits that are identified as reverted, since the average precision is always the same when varying the values of recall in the PR curve. We use *PRROC*[15] R package to calculate and plot AUC-PR.

Imbalance handling approach. To handle the high imbalance of our dataset, we use a threshold tuning approach in our classifier. A threshold

[15] https://cran.r-project.org/web/packages/PRROC/PRROC.pdf

indicates a decision boundary to differentiate reverted commits from not-reverted commits. A classifier will classify a commit to be a reverted commit if its likelihood score to be reverted is higher than the threshold. Usually, the default threshold is 0.5, however, the high imbalanced data causes a classifier to favor the majority class. Thus, we use a threshold tuning approach to automatically determine an imbalanced decision boundary using the training set (Xia et al., 2015b). With the imbalanced decision boundary, a commit will be classified to be a reverted commit when its likelihood score to be reverted is larger than the decision boundary. Note that we did not use threshold tuning approach in RQ1 and RQ2, because the AUC and cost-effectiveness would not be impacted by threshold.

We tune the threshold as follows: first, we randomly sample 20% instances of the training set (following a stratified sampling procedure) as the validation set, and use the remaining 80% instances of the training set as our new training set to build a classifier. Second, we use the built classier to evaluate the F1-score score on the validation set by varying the threshold from 0.01 to 1.00, stepped by 0.01. Third, we determine the threshold that achieves the best F1-score on the validation set. We choose the F1-score score as our optimization target since we aim to classify the minority class with both high precision and recall. Finally, we use the new training set and the determined threshold to evaluate performance on our testing set.

Baseline 3: ZeroR. In addition to the two baselines considered in RQ1, we added a third baseline ZeroR. The ZeroR classifier simply classifies all the commits as the majority class (i.e., not-reverted). Due to the high imbalance of reverted/not-reverted classes in the studied projects, we would like to investigate whether the proposed model can outperform ZeroR. The AUC-PR of ZeroR is 0, since the precision and recall of ZeroR for the reverted class is 0. Note that in this subsection, we use the same experimental data, setting and baselines with RQ1.

Results: The results are shown in Tables 8 and 9. Same to RQ1, we perform statistical tests using the Wilcoxon signed-rank test with a Bonferroni correction at a 95% significance level. In addition, we compute the effect size using Cliffs delta. In summary, we have the following findings:

In terms of TPR, our approach improves RG and NBMCM by 811.18% and 145.87% on average in a statistical significant manner moreover the effect sizes are large in all cases. The TPR of ZeroR is 0, since TP of ZeroR is 0. In terms of TNR, there is no statistically significant difference between our approach, RG and NBMCM. ZeroR achieves the best TNR, since it simply classifies all commits as not reverted. RG can achieve a high TNR since we implement RG according to the data distribution as described in RQ1. Based on TPR, we observe that our approach can save the development effort by correctly identifying 22.2% of the reverted commits. Concretely, if our approach can warn developers about a reverted commit in advance, they can check and modify them carefully. In this way, 22.2% of the reverted commits may not be reverted or reverted in an early stage (hence reducing the amount

of wasted effort and reducing the need for last minute emergency efforts (e.g., new commits or changes to features) to cope with these late-stage reverted commits). As a result, the development effort on these commits may not be wasted and the maintenance and project challenges that are introduced by the reversal of such commits after a long time can be eliminated. Based on TNR, we observe that our approach can correctly identify 97.9% of the not-reverted commits – ensuring a small ratio of false alarms among not-reverted commits, i.e., only 2.1% of the not-reverted commits are false alarms.

In terms of AUC-PR, our approach improves RG and NBMCM by 679.04% and 106.70% on average with statistical significance and all the effect sizes are large. The AUC-PR of ZeroR is 0, since precision and recall are 0. To better understand the AUC-PR of our approach and baselines, we plot an example of a PR curve in our dataset. Since we use 10 times 10 fold cross validation, it is difficult to plot PR curve of each fold. Thus, we plot the PR curve of one fold (1454 commits in total and 56 reverted commits) randomly selected from Hadoop project as Figure 1 shows; the AUC-PR of this fold is 0.283. Since the AUC-PR of ZeroR is 0, we just plot three methods: Ours, NBMCM and RG. The figure shows that the area under the curve of our proposed approach is much larger than the baselines. The AUC-PR value is not high due to the high imbalance of our dataset. The precision drops quickly along with increase in recall, highlighting that we need check more commits to find a truly reverted commit. Prior study also stated that the skew of datasets has an impact on the AUC-PR value (Lampert and Gançarski, 2014). It is difficult to achieve a highly AUC-PR value in a high imbalanced dataset. For example, Lampert and Gançarski (2014) reported that the AUC-PR is 0.1088 and 0.1307 of two methods when the ratio of positive instances is 0.01. Boyd et al. (2012) reported that the AUC-PR is 0.36 when the ratio of positive instances is 0.04.

Our approach significantly improves over the baselines by a substantial margins in terms of AUC-PR and TPR. There is no significant difference between our approach, RG and NBMCM in terms of TNR. From TPR and TNR, our approach can save the development effort by correctly identify 22.2% of the reverted commits and only 2.1% of the not-reverted commits are false alarms.

5 Discussion

As shown in previous sections, we characterize the commits that are being reverted and build a identification model. However, there are some other observations worthy of further investigation. In this section, we will report other observations including seven aspects: (1) how effective is our model when varying the levels of inspection effort? (Section 5.1) (2) Whether different underlying classifiers will affect the performance of our approach? (Section 5.2) (3) How effective is our model for cross-project setting? (Section 5.3) (4) How effective is outlier detection method? (Section 5.4) (5) How effective is

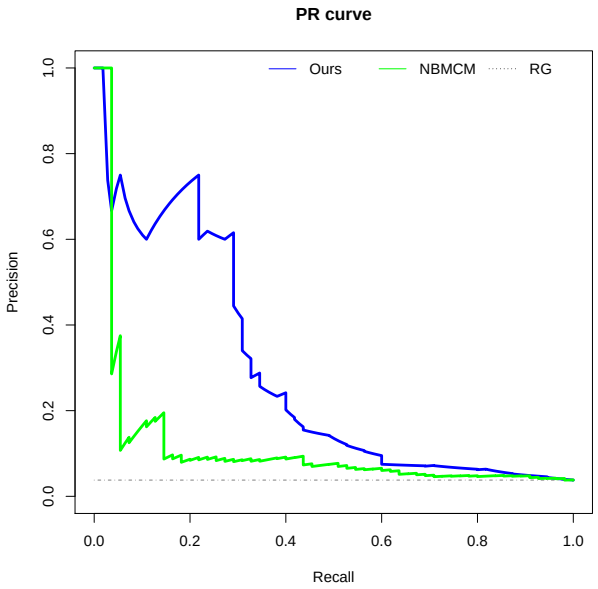


Fig. 1: An example PR curve for Hadoop

Table 8: TPR and TNR of our approach and baselines. The best performance is highlighted in bold. The underlined case indicates that our approach outperforms baseline with statistical significance and large effect size.

Project	**TPR**				**TNR**			
	RG	**NBMCM**	**ZeroR**	**Ours**	**RG**	**NBMCM**	**ZeroR**	**Ours**
Hadoop	0.040	0.140	0.000	**0.226**	0.962	0.968	**1.000**	0.987
Gerrit	0.012	0.040	0.000	**0.185**	0.986	0.987	**1.000**	0.969
Hbase	0.047	0.104	0.000	**0.286**	0.954	0.962	**1.000**	0.976
Karaf	0.026	0.127	0.000	**0.194**	0.976	0.981	**1.000**	0.986
Jenkins	0.011	0.037	0.000	**0.178**	0.988	0.990	**1.000**	0.989
Spring-boot	0.021	0.112	0.000	**0.184**	0.979	0.982	**1.000**	0.974
Hive	0.042	0.133	0.000	**0.285**	0.958	0.967	**1.000**	0.959
Platform	0.013	0.032	0.000	**0.157**	0.985	0.988	**1.000**	0.985
Egit	0.009	0.102	0.000	**0.354**	0.981	0.989	**1.000**	0.992
JDT	0.023	0.078	0.000	**0.174**	0.975	0.979	**1.000**	0.971
Average	0.024	0.090	0.000	**0.222**	0.974	0.979	**1.000**	0.979
Improved	811.18%	145.78%	-		0.45%	0.00%	-2.12%	

our model for project written in other programming languages? (Section 5.5)
(6) What are the implications for practitioners and researchers?(Section 5.6)

Table 9: AUC-PR of our approach and baselines. The best performance is highlighted in bold. The underlined case indicates that our approach outperforms baseline with statistical significance and large effect size.

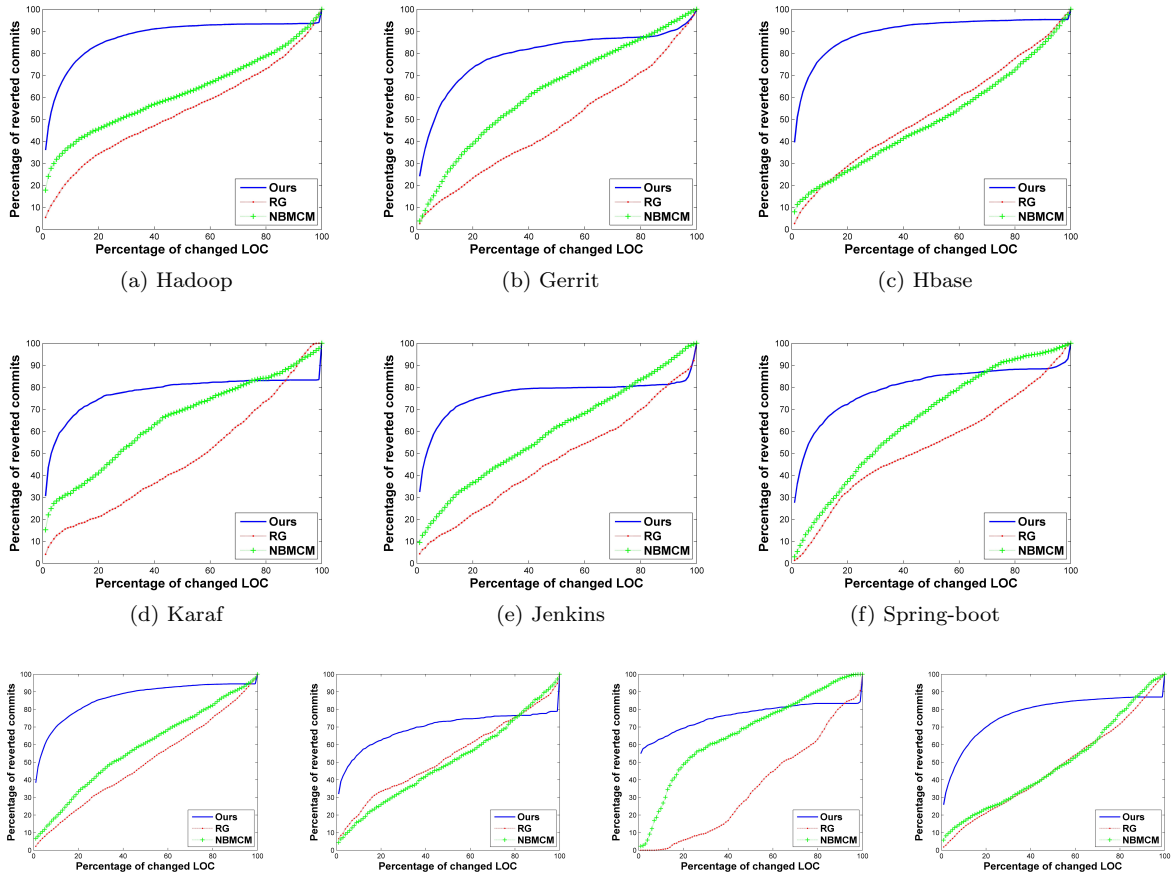
Project	**AUC-PR**			
	RG	**NBMCM**	**ZeroR**	**Ours**
Hadoop	0.040	0.174	0.000	**0.263**
Gerrit	0.012	0.032	0.000	**0.061**
Hbase	0.047	0.195	0.000	**0.299**
Karaf	0.027	0.145	0.000	**0.226**
Jenkins	0.011	0.042	0.000	**0.132**
Spring-boot	0.021	0.065	0.000	**0.129**
Hive	0.042	0.115	0.000	**0.208**
Platform	0.013	0.037	0.000	**0.121**
Egit	0.010	0.062	0.000	**0.367**
JDT	0.023	0.065	0.000	**0.121**
Average	*0.025*	*0.093*	0.000	***0.193***
Improved	*679.04%*	*106.70%*	-	

5.1 Investigating the effectiveness of our model when different percentages of changed LOC are inspected

Although we would like to capture all of the reverted commits, there is always a conflicting interest between the amount of effort one allocates and the amount of reverted commits they can detect. Therefore, it is important to investigate the cost-effectiveness of our models by considering different amount of effort.

By default, we set the percentage of changed LOC to inspect as 20% to compute the cost-effectiveness score. In this subsection, we investigate the cost-effectiveness of our model when different percentages of LOC are inspected. We plot the cost-effectiveness graphs that show the percentages of detected reverted commits by inspecting different percentages of LOC. As a result, there are 100 cost-effectiveness values in each plot. Note that for each percentage, we use the average effectiveness value of 100 values produced by 10 times 10-fold cross validation. In addition, we also plot the cost-effectiveness graphs of two the used baselines in RQ1, i.e., RG and NBMCM.

Figure 2 visualizes the cost-effectiveness graphs of the ten studied projects using our model (Ours), RG and NBMCM. We observe that our model is better than the baselines for a large range (i.e., $> 70\%$) of percentages of changed LOC to inspect in most of the studied projects, including Hadoop, Gerrit, Hbase, Karaf, Jenkins, Spring-boot, Hive, Platform and Jdt. In project Egit, our model is better than the baselines for the range (i.e., $> 66\%$) of percentages of changed LOC to inspect. In summary, our model is better than the baselines for the large early range of percentages of changed LOC to inspect. In particular, our model has a relatively flat range in the curve with a stable cost-effectiveness for the a few last range of percentages of changed LOC to inspect. The reason for the flat range is due to the class imbalance phenomenon in our dataset (i.e., less than 3% commits are reverted in average). If a model identifies most of the reverted commits at the top of



(a) Hadoop (b) Gerrit (c) Hbase

(d) Karaf (e) Jenkins (f) Spring-boot

(g) Hive (h) Platform (i) Egit (j) JDT

Fig. 2: Cost-effectiveness graphs for the ten studied projects

the ranking list and leaves a few reverted commits at the bottom of the list, the cost-effectiveness will be stable for a large range of percentage of LOC until inspecting all the changed LOC.

In summary, our model is better than the baselines at the early large range of percentages of changed LOC to inspect, but it performs worse than the baselines after a certain range. However, in practice, developers would not inspect such a high number of changed LOC due to limited project budget and tight project schedule. Therefore, our model is still cost effective.

Table 10: AUC-ROC and Cost-effectiveness of different classifiers.

Project	AUC-ROC				Cost-effectiveness			
	NBM	**LR**	**BN**	**RF**	**NBM**	**LR**	**BN**	**RF**
Hadoop	0.676	0.730	0.716	**0.732**	0.886	0.713	0.767	**0.839**
Gerrit	0.511	**0.800**	0.761	0.775	0.299	**0.746**	0.641	0.733
Hbase	0.675	0.755	0.739	**0.781**	0.859	0.763	0.774	**0.863**
Karaf	0.644	**0.761**	0.700	0.734	0.517	**0.858**	0.622	0.746
Jenkins	0.616	0.726	0.715	**0.765**	0.462	**0.784**	0.658	0.743
Spring-boot	0.481	**0.801**	0.770	0.782	0.343	**0.782**	0.733	0.722
Hive	0.695	0.756	0.724	**0.768**	**0.835**	0.743	0.740	0.796
Platform	0.596	0.692	0.699	**0.717**	0.434	0.728	**0.740**	0.625
Egit	0.704	0.790	0.751	**0.797**	0.737	**0.752**	0.653	0.693
JDT	0.623	**0.724**	0.709	0.715	0.204	**0.718**	0.566	0.700
Average	*0.622*	*0.753*	*0.728*	***0.756***	*0.558*	***0.759***	*0.689*	*0.746*

5.2 Investigating the impact of different classifiers

In our identification model, we use Random Forest as the default classifier. However, we can leverage other classifiers. In order to investigate the impact of other underlying classifiers, we investigate three more classifiers, Naive Bayes Multinomial (NBM), Logistic Regresion (LR) and Bayes Net (BN). NBM have been briefly introduced in RQ1. We describe the LR and BN classifiers briefly in this subsection.

Logistic Regression (LR). Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables (Han et al., 2011).

Bayes Network (BN). BN is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). It is probabilistic because it is built from probability distributions and uses the laws of probability for classification and anomaly detection, for reasoning and diagnostics, decision making under uncertainty and time series prediction (Han et al., 2011).

We implement the aforementioned underlying classifiers on top of the Weka tool (Hall et al., 2009). Table 10 presents the performances of different models built on our extracted commit features using different classifiers. We highlight the best classifier in bold for each project. The experiment is conducted under the same validation setting with the RQ1. The results show that RF outperforms other three classifiers in average in terms of AUC-ROC. In terms of cost-effectiveness, RF achieves comparable performance with LR and outperforms other two classifiers. Thus, in practice, we suggest the use of RF classifier as the underlying classifier to build our identification model.

Table 11: AUC-ROC of cross-project identification

Source \ Target	Hadoop	Gerrit	Hbase	Karaf	Jenkins	Spring-boot	Hive	Platform	Egit	Jdt
Hadoop		0.74	0.72	0.66	**0.69**	0.71	0.68	**0.67**	**0.75**	**0.69**
Gerrit	0.60		0.64	0.67	0.65	0.72	0.60	0.66	0.75	0.65
Hbase	0.66	0.67		0.57	0.67	0.68	**0.70**	0.66	0.43	0.67
Karaf	0.46	0.72	0.60		0.65	0.64	0.63	0.65	0.78	0.63
Jenkins	0.61	0.74	0.67	0.69		0.71	0.66	0.65	0.67	0.67
Spring-boot	0.48	0.70	0.55	0.66	0.59		0.56	0.60	0.69	0.54
Hive	**0.69**	0.71	**0.73**	**0.70**	0.69	0.66		0.65	0.72	0.67
Platform	0.63	0.75	0.68	0.69	0.66	**0.75**	0.68		0.51	0.64
Egit	0.58	0.64	0.57	0.59	0.50	0.72	0.56	0.53		0.48
JDT	0.65	**0.77**	0.69	0.67	0.68	0.74	0.69	0.64	0.61	
Average	*0.59*	*0.72*	*0.65*	*0.66*	*0.64*	*0.70*	*0.64*	*0.64*	*0.66*	*0.63*

Table 12: Cost-effectiveness of cross-project identification

Source \ Target	Hadoop	Gerrit	Hbase	Karaf	Jenkins	Spring-boot	Hive	Platform	Egit	Jdt
Hadoop		0.70	0.71	0.84	0.75	0.69	0.68	0.71	**0.78**	**0.71**
Gerrit	0.63		0.68	0.69	0.68	0.62	0.66	0.56	0.71	0.55
Hbase	**0.65**	0.69		**0.85**	0.75	0.70	**0.73**	0.71	0.76	0.69
Karaf	0.60	0.68	0.70		**0.76**	0.66	0.72	0.63	0.77	0.60
Jenkins	0.60	0.64	0.63	0.67		0.62	0.67	0.55	0.74	0.59
Spring-boot	0.59	0.61	0.63	0.75	0.66		0.66	**0.73**	0.78	0.68
Hive	0.64	0.71	0.70	0.81	0.74	0.71		0.69	0.76	0.70
Platform	0.65	0.71	**0.72**	0.85	0.74	**0.72**	0.72		0.69	0.59
Egit	0.58	0.62	0.70	0.80	0.66	0.71	0.68	0.57		0.54
JDT	0.64	**0.72**	0.72	0.85	0.76	0.71	0.71	0.64	0.77	
Average	*0.62*	*0.68*	*0.69*	*0.79*	*0.72*	*0.68*	*0.69*	*0.64*	*0.75*	*0.63*

5.3 Investigating the effectiveness of cross-project identification

In the experimental setting of RQs 1 and 2, we train our identification model by learning from the historical labeled dataset within the project. However, for new projects or projects with limited development history, there is often not enough labeled data for building a model. An alternative solution is to learn from other projects that have enough labeled data, i.e., cross-project identification. In this subsection, we investigate how effective is our model for cross-project identification.

For each target project, we built the identification model by learning from other alternative projects (refer to as source project). In this way, there are 9 source projects for each target project. In the evaluation, we use one source project as training data, and the target project as testing data. In this way, there are 9 effectiveness values for each target project.

Tables 11 and 12 present the results for cross-project setting. The results show that the performance of cross-project setting is poorer compared to within-project setting in terms of the average performance among different source projects. However, the performance can be reasonable when carefully

selecting a source project. We highlight the best source project in bold for each target project. The results show that the performance is sensitive to the selection of source project. For example, when choosing one project from <Hadoop, Hbase or Hive> as a target project, by choosing the other two projects as source project, the model would be better than choosing other projects as source project. These three projects come from the same community (i.e., Apache). Additionally, although Platform, Egit, JDT come from the same community (i.e., Eclipse), the best source projects are not Eclipse projects. Thus, the community is not a definite reason for selection of source project. One good criterion for selecting a suitable source projects may be the data distribution of reverted vs. non-reverted commits. The ratio of reverted commits in Hadoop, Hive and Hbase is similar and higher than other projects. Projects with similar data distribution and more positive instances (i.e., reverted commits) tend to be better for training our model. In summary, our identification model can achieve a reasonable performance when carefully selecting a source project.

5.4 Investigating the effectiveness of outlier detection method

Due to the high imbalance of the studied dataset, the problem of identifying reverted commits may also be solved by using outlier detection methods. The assumption is that the reverted commits are more likely to be outliers. Thus, in this discussion, we investigate whether or not we can identify reverted commits using outlier detection methods.

We use two outlier detection methods: One-class SVM and local outlier factor (LOF).

One-class SVM. One-class SVM algorithm learns a decision function for outlier detection: classifying new instance as similar or different to the training set. The basic idea of One-class SVM is that the SVM is trained on only one class (i.e., "normal" class). It infers the properties of normal instances and from these properties can predict which instances in testing set are unlike the normal. Thus, in our context, we only use the non-reverted commits as training set. We use the e1071[16] R package as the implementation of one-class SVM method. We use the same experimental setting used to answer RQ1, i.e., 10 x 10 stratified cross validation, but we removed the reverted commits from training set. The output of One-class SVM contains both a class result (i.e., reverted or not-reverted) and a likelihood value.

Local outlier factor (LOF). LOF (Local Outlier Factor) is an unsupervised algorithm for identifying local outliers in a multidimensional dataset (Breunig et al., 2000). The outlier factor is local means only a restricted neighborhood of each instance is taken into account. The basic idea is that the local density of an instance is compared with that of its neighbors. If the former is significantly lower than the latter, the instance is in a sparser region than its neighbors,

[16] https://cran.r-project.org/web/packages/e1071/e1071.pdf

Table 13: Performance of LOF

Project	One-class SVM		LOF	
	AUC-ROC	**Cost-effectiveness**	**AUC-ROC**	**Cost-effectiveness**
Hadoop	0.455	0.464	0.499	0.193
Gerrit	0.591	0.469	0.573	0.070
Hbase	0.421	0.496	0.503	0.190
Karaf	0.419	0.498	0.526	0.178
Jenkins	0.504	0.503	0.604	0.141
Springboot	0.559	0.531	0.627	0.081
Hive	0.455	0.452	0.515	0.175
Platform	0.512	0.575	0.564	0.158
Egit	0.353	0.203	0.460	0.159
JDT	0.465	0.469	0.546	0.175
Average	*0.473*	*0.466*	*0.542*	*0.152*

which suggests that it is an outlier. We use the DMwR [17] R package as the implementation of LOF method. The LOF value of an instance is based on the single parameter of the number of nearest neighbors, we set the default parameter as 10. The output of LOF is the degree of an instance to be an outlier.

Tables 13 presents the results of One-class SVM and LOF. We can observe that the performance is poor compared with our approach. For example, in terms of AUC-ROC, the average performance of one-class SVM and LOF is close to random guess (i.e., 0.5). In terms of cost-effectiveness, our average performance (i.e., cost-effectiveness of 0.746) also improves the outlier detection methods by a substantial margin. Thus, outlier detection methods are not effective enough for reverted commit identification.

5.5 Investigating the effectiveness on projects written in other programming languages

In the above-mentioned experiments, we studied 10 Java projects. However, the reverted commits identification may also be needed in projects written in other programming languages, such as C++ and Python. Thus, in this section, we would like to investigate how effective our model is for projects written in other programming languages.

We selected four popular projects written in C++ and Python that received many stars on Github. Table 14 provides a summary of the studied projects. CNTK[18] is the Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit. Pytorch[19] is an open source machine learning library for Python, based on Torch. Scikit-learn[20] is a simple and efficient tool

[17] https://cran.r-project.org/web/packages/DMwR/DMwR.pdf
[18] https://www.microsoft.com/en-us/cognitive-toolkit/
[19] https://pytorch.org/
[20] http://scikit-learn.org/

Table 14: Summary of studied projects written in other programming languages

Project	# All Commits	# Reverted	Reverted Ratio	Programming language
CNTK	15,061	243	1.61%	C++
Pytorch	7,430	140	1.88%	Python, C++
Scikit-learn	22,249	230	1.03%	Python, C, C++
Django	24,984	234	0.94%	Python
Total	***69,724***	***847***	***1.21%***	

Table 15: Performance on projects written by other programming languages

Project	AUC-ROC			Cost-effectiveness		
	RG	NBMCM	Ours	RG	NBMCM	Ours
CNTK	0.500	0.700	**0.719**	0.195	0.333	**0.669**
Pytorch	0.500	0.650	**0.734**	0.165	0.431	**0.645**
Scikit-learn	0.500	0.683	**0.715**	0.192	0.328	**0.592**
Django	0.500	0.628	**0.714**	0.194	0.295	**0.570**

set for machine learning and data analysis in Python. Django[21] is an web framework written in Python, which follows the model-view-template (MVT) architectural pattern. The average ratio of reverted commits is 1.21%, which is smaller than that for our studied Java projects.

In this subsection, we use the same data preparation, feature extraction, and experimental setting and evaluation method as RQ1. One difference is the feature extraction step. We did not consider four features, i.e., NLC, NMC, NHC and NCC in this subsection. Because these four features depend on ChangeDistiller tool, that only supports Java projects.

Table 15 shows the results of our model and the two baselines investigated to answer RQ1. In summary, the results show that our model achieves a reasonable performance which outperforms the two baselines in all the cases.

5.6 Implications

Usage scenarios, benefits and costs of our approach. As described in the Introduction, the typical usage scenario of our approach is to provide early warnings to developers about commits that will be eventually reverted. Our approach provides its early warnings through the classification of such commits.

Considering AUC-ROC and cost-effectiveness, we prioritize commits by the likelihood score for them to be reverted. Commits with a higher likelihood score are recommended for developers to check. Our average AUC-ROC and cost-effectiveness is 0.756 and 0.746 respectively; implying that by following our approachs recommendation (i.e., a list of commits sorted by the likelihood score to be reverted), developers can find 74.6% of the reverted commits by

[21] https://www.djangoproject.com/

inspecting 20% of the total changed LOC. This is useful for developers when they have to check a number of commits in a limited time.

Considering TPR and TNR, we prioritize commits by recommending possibly reverted commits identified by our approach for a close examination. Our average TPR and TNR is 0.222 and 0.979 respectively; implying that we can save the development effort by correctly identifying 22.2% of the reverted commits. As a result, the development effort on these commits may not be wasted and the maintenance difficulty that is introduced by reverting these commits can be mitigated. Meanwhile, our approach can correctly identify 97.9% of the not-reverted commits. This assures a small ratio of false alarms among not-reverted commits, i.e., only 2.1% of the not-reverted commits are false alarms. Such a performance indicates that the cost of our approach, i.e., developers would end up wasting time by checking 2.1% of the not-reverted commits while in the same time saving time by identifying 22.2% of the reverted commits. Due to the impact of the reverted commits (e.g., impeding the development progress, increasing the difficulty of maintenance, or leading to last minute pre-release changes), we believe that this a rational and worthwhile cost.

Implications for practitioners. First, we found that the ratio of reverted commits is 2.53% on average. Due to the potential issues induced by reverting a commit, such as impeding the development progress or increasing the difficulty of maintenance, the analysis and handling of reverted commits cannot be ignored. Second, using our tool, possibly reverted commits can be detected in an early stage. Then, developers can fix them or revert them early. Third, there are some common important features in our studied projects, such the developer related features "EXP" and "NDEV". It means that developers should pay more attention to the commits made by non-expert developers of the project. Additionally, the commits that modify "hot" files (i.e., files modified by many developers) should also be checked carefully. Fourth, when applying our approach on a new project, we can train the model by learning from another project. But note that the use of a source project that has a similar data distribution and with more reverted commits would result in better performance.

Implications for researchers. First, the detection method for reverted commits should be revisited. A prior detection method by Shimagaki et al. (2016) (i.e, only checking the commit message) can detect 0.81% commits that are reverted on average in our dataset. However, the reverted commits that did not follow a standard procedure are ignored. Thus, we propose to add another detection method, i.e., detecting reverted commits by comparing the changed code content of two commits. As a result, we detected 2.53% commits that are reverted on average in our dataset. (2) The developer-related features and change-related features are more discriminative for building an identification model. Further research for enhancing the performance can pay more attention on these two dimensions. (3) It is hard to achieve a high TPR and AUC-PR for reverted commits due to the high imbalance data distribution of reverted

commits. Since our paper is the first attempt on this problem, further works are needed on how to enhance TPR and AUC-PR.

6 Threats to Validity

Threats to internal validity refer to errors in our implementation. One potential threat to validity is the potential errors in commit feature extraction. For example, we measure EXP by counting the number of previously submitted commits of the developer in the project. In this way, the computation of EXP ignores the developer expertise in other projects. Also, it may consider some less meaningful commits, such as commits that only modify comments. Still, all of our considered commits are accepted, and not abandoned by project owners. These commits can also indicate that the developer has certain knowledge on this project. Thus, we believe this threat is minor. Additionally, to mitigate the threat, we compute the same features by following the method in previous studies (Kamei et al., 2013), and use a third-party library, such as ChangeDisttler (Fluri et al., 2007) that has been used in past studies for commit feature extraction. In addition, to reduce the errors in our implementation, we double-checked and fully tested the code, but there could still exist some errors we did not find.
Threats to external validity refer to the generalizability of our identification tool. We have analyzed 125,241 software commits from ten different open source projects. Further studies are needed to reduce this threat further by analyzing even more dataset including both open source and commercial projects.
Threats to construct validity refer to the suitability of our evaluation metrics. In this study, one potential threat is that we use AUC and cost-effectiveness as the performance measures, and use Wilcoxon signed-rank test to investigate whether the improvement of our proposed model over the baselines is significant. One or all of the measures and tests have been used in past studies (Lessmann et al., 2008; Nam and Kim, 2015; Kamei et al., 2013; Xia et al., 2016a; Yang et al., 2016; Yan et al., 2017).

7 Related Work

We divide our related work into two parts: studies on commit rework, and identification models on software commits.

7.1 Commits Rework

Commits rework leads to the additional effort during software development. Rework can be done due to by different reasons, such as rejecting a commit or reverting a commit. Before a commit is checked into the main repository, it

needs to be checked by code reviewers. A problematic commit can be rejected for various reasons.

Beller et al. (2014) and Mäntylä and Lassenius (2009) investigate the reasons that commits need to be reworked during the code review process. As a result, they find that functional commits and evolvable (i.e., non-functional) commits might need to be reworked. Tao et al. (2014) conduct an empirical study to investigate a more fine-grained reason for commits to be rejected or for reworked. They provide a comprehension list of commit rework reasons (e.g., test failures and introducing new bugs) by manually inspecting 300 rejected commits in the Eclipse and Mozilla projects and a large-scale online survey of Eclipse and Mozilla developers and the literature.

If a problematic commit has been checked into the main code repository, a popular and coarse-grained approach for recovering from such a problematic commit is to revert it. A few studies have been proposed to empirically investigate this problem.

Yoon and Myers (2012) present a preliminary study that investigates when and how the developers revert a commit. They conduct an exploratory study with 12 professional developers and a follow-up online survey. As a result, they find that 75% of developers need a reverting tool, and more robust reverting assistance tools would help developers write code more correctly and efficiently. Codoban et al. (2015) find that developers often use version control systems (e.g., SVN and Git) to back out problematic commits (e.g., reaching a dead end in implementation when doing trial and error programming) by reverting to previously working state. Souza et al. (2015) conduct an empirical study to investigate the relationship between rapid releases and reverted commits on Mozilla. As a result, they find that rapid releases have an association with reverted commits. Namely, the reverted commits rate (i.e., the backout rate) increases after shortening the software release cycle. Shimagaki et al. (2016) conduct an empirical study to better understand why commits are reverted in large software systems. They quantitatively and qualitatively investigate two industrial and four open source projects in terms of three aspects, the ratio of reverted commits, the survival time of reverted commits and manually summarize the reasons for being reverted.

Our work is motivated by these prior research efforts. The difference is that our work focuses on proposing a identification model to identify whether or not a commit will be reverted.

7.2 Identification Models of Software Commits

Many studies have been proposed to identify various characteristic of a commit, such as defective commit (Mockus and Weiss, 2000; Kim et al., 2008; Kamei et al., 2013; Yang et al., 2016) and build co-change (McIntosh et al., 2014; Xia et al., 2015a; Macho et al., 2016).

Defective commit identification aims to identify whether or not a commit is a defect inducing commit. For example, Mockus and Weiss (2000) assess

the risk of software changes (i.e., the probability that changes are defect inducing) in 5ESS network switch project. Kim et al. (2008) classify each software change as buggy or clean by using the identifiers in added and deleted source code and textual features in change logs. Kamei et al. (2013) conduct a large-scale empirical study of just-in-time quality assurance on a variety of open source and commercial projects from multiple domains. They first apply effort-aware evaluation (i.e., considering review effort for inspecting defective changes) on defective change identification. Following their work, Yang et al. (2016) propose the use of simple unsupervised models for defective commit. They found that simple unsupervised models outperform supervised models for identifying defective changes.

Build co-change identification aims to identify whether or not a commit requires build co-change. For example, McIntosh et al. (2014) build a classifier that identify whether or not a software change lead to a build co-changing. Xia et al. (2015a) propose cross-project build co-change identification to improve the performance of build co-change identification in projects in the initial development phases. Subsequently, Macho et al. (2016) improve the existing performance by taking into account detailed information about source code changes.

The similarity between our work and these aforementioned work is the used commit features. Many of the commit features that are used in our work are inspired by the prior work, such as diffusion metrics (Kamei et al., 2013; McIntosh et al., 2014; Xia et al., 2015a), message metrics (Kim et al., 2008), experience metrics (Kamei et al., 2013; Yang et al., 2016) and history metrics (Kamei et al., 2013; Yang et al., 2016). The difference between our work and these aforementioned work is that we aim to identify whether or not a commit will be reverted. To the best of our knowledge, our work is the first work for reverted commit identification.

8 Conclusion

In this paper, we characterize reverted commits and propose a model to identify reverted commits by extracting 27 commit features that are divided into three dimensions, namely change, developer and message. The model can identify whether or not a commit will be reverted in the future. To evaluate the effectiveness of our identification model, we perform an empirical study on ten open source projects with totally 125,241 software commits.

Our experimental results show that: (1) Our approach achieves a promising performance and outperforms baselines by a substantial margin in terms of AUC-ROC, cost-effectiveness, AUC-PR and TPR. In terms of the average performance across the ten studied projects, our approach achieves an AUC-ROC of 0.756, a cost-effectiveness of 0.746, a TPR of 0.222 and an AUC-PR of 0.193 which significantly improves over the baselines in a substantial margin; achieves a TNR of 0.979 which achieves a comparable performance compared with baselines. (2) “Developer” is the most discriminative dimension

among the three dimensions of studied features. However, using all the three dimensions of commit features leads to better performance. (3) The features "LA", "LD" "EXP", "NDEV", and "NUC" are the most discriminative features for identifying reverted commits.

Acknowledgment. This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904), NSFC Program (No. 61602403) and China Postdoctoral Science Foundation (No. 2017M621931).

References

Abdi H (2007) Bonferroni and šidák corrections for multiple comparisons. Encyclopedia of measurement and statistics 3:103–107

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 202–211

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code!: examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, pp 4–14

Boyd K, Costa VS, Davis J, Page CD (2012) Unachievable region in precision-recall space and its effect on empirical evaluation. In: Proceedings of the International Conference on Machine Learning, NIH Public Access, vol 2012, p 349

Bradley AP (1997) The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern recognition 30(7):1145–1159

Breiman L (2001) Random forests. Machine learning 45(1):5–32

Breunig MM, Kriegel HP, Ng RT, Sander J (2000) Lof: identifying density-based local outliers. In: ACM sigmod record, ACM, vol 29, pp 93–104

Codoban M, Ragavan SS, Dig D, Bailey B (2015) Software history under the lens: a study on why and how developers examine it. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, pp 1–10

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering 43(7):641–657

Davis J, Goadrich M (2006) The relationship between precision-recall and roc curves. In: Proceedings of the 23rd international conference on Machine learning, ACM, pp 233–240

Fan Y, Xia X, Lo D, Hassan AE (2018a) Chaff from the wheat: Characterizing and determining valid bug reports. IEEE Transactions on Software Engineering

Fan Y, Xia X, Lo D, Li S (2018b) Early prediction of merged code changes to prioritize reviewing tasks. Empirical Software Engineering pp 1–48

Fluri B, Gall HC (2006) Classifying change types for qualifying change couplings. In: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, IEEE, pp 35–45

Fluri B, Wuersch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33(11)

Fu Y, Yan M, Zhang X, Xu L, Yang D, Kymer JD (2015) Automated classification of software change messages by semi-supervised latent dirichlet allocation. Information and Software Technology 57:369–377

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD explorations newsletter 11(1):10–18

Han J, Pei J, Kamber M (2011) Data mining: concepts and techniques. Elsevier

Hassan AE (2008) Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM symposium on Applied computing, ACM, pp 837–841

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp 78–88

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 392–401

Hindle A, German DM, Holt R (2008) What do large commits tell us?: a taxonomical study of large commits. In: Proceedings of the 2008 international working conference on Mining software repositories, ACM, pp 99–108

Huang J, Ling CX (2005) Using auc and accuracy in evaluating learning algorithms. IEEE Transactions on knowledge and Data Engineering 17(3):299–310

Huang Q, Shihab E, Xia X, Lo D, Li S (2017) Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering pp 1–34

Jiang T, Tan L, Kim S (2013) Personalized defect prediction. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, pp 279–289

Kabinna S, Shang W, Bezemer CP, Hassan AE (2016) Examining the stability of logging statements. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, vol 1, pp 326–337

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. Software Engineering, IEEE Transactions on 39(6):757–773

Kim S, Whitehead Jr EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2):181–196

Lampert TA, Gançarski P (2014) The bane of skew. Machine learning 97(1-2):5–32

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34(4):485–496
Li H, Shang W, Zou Y, Hassan AE (2016) Towards just-in-time suggestions for log changes. Empirical Software Engineering pp 1–35
Li H, Shang W, Zou Y, Hassan AE (2017) Towards just-in-time suggestions for log changes. Empirical Software Engineering 22(4):1831–1865
Li H, Chen THP, Shang W, Hassan AE (2018) Studying software logging using topic models. Empirical Software Engineering pp 1–40
Long JD, Feng D, Cliff N (2003) Ordinal analysis of behavioral data. Handbook of psychology
Macho C, McIntosh S, Pinzger M (2016) Predicting build co-changes with source code change and commit categories. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, vol 1, pp 541–551
Mäntylä MV, Lassenius C (2009) What types of defects are really discovered in code reviews? IEEE Transactions on Software Engineering 35(3):430–448
McCallum A, Nigam K, et al. (1998) A comparison of event models for naive bayes text classification. In: AAAI-98 workshop on learning for text categorization, Madison, WI, vol 752, pp 41–48
McIntosh S, Adams B, Nagappan M, Hassan AE (2014) Mining co-change information to understand when build changes are necessary. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, pp 241–250
Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: icsm, pp 120–130
Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180
Nam J, Kim S (2015) Clami: Defect prediction on unlabeled datasets (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, pp 452–463
Romano J, Kromrey JD, Coraggio J, Skowronek J, Devine L (2006) Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices. In: annual meeting of the Southern Association for Institutional Research, Citeseer
Rosen C, Grawi B, Shihab E (2015) Commit guru: analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 966–969
Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics pp 507–512
Shimagaki J, Kamei Y, McIntosh S, Pursehouse D, Ubayashi N (2016) Why are commits being reverted?: A comparative study of industrial and open source projects. In: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on, IEEE, pp 301–311
Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: ACM sigsoft software engineering notes, ACM, vol 30, pp 1–5

Souza R, Chavez C, Bittencourt RA (2015) Rapid releases and patch backouts: A software analytics approach. IEEE Software 32(2):89–96

Tantithamthavorn C, McIntosh S, Hassan AE, Ihara A, Matsumoto K (2015) The impact of mislabelling on the performance and interpretation of defect prediction models. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, IEEE, vol 1, pp 812–823

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering 43(1):1–18

Tao Y, Han D, Kim S (2014) Writing acceptable patches: An empirical study of open source project patches. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, pp 271–280

Tian Y, Nagappan M, Lo D, Hassan AE (2015) What are the characteristics of high-rated apps? a case study on free android applications. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, pp 301–310

Valdivia Garcia H, Shihab E (2014) Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 72–81

Wilcoxon F (1992) Individual comparisons by ranking methods. Breakthroughs in Statistics pp 196–202

Wolpert DH, Macready WG (1999) An efficient method to estimate bagging's generalization error. Machine Learning 35(1):41–55

Xia X, Lo D, Qiu W, Wang X, Zhou B (2014) Automated configuration bug report prediction using text mining. In: Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual, IEEE, pp 107–116

Xia X, Lo D, McIntosh S, Shihab E, Hassan AE (2015a) Cross-project build co-change prediction. In: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, IEEE, pp 311–320

Xia X, Lo D, Shihab E, Wang X, Yang X (2015b) Elblocker: Predicting blocking bugs with ensemble imbalance learning. Information and Software Technology 61:93–106

Xia X, Lo D, Pan SJ, Nagappan N, Wang X (2016a) Hydra: Massively compositional model for cross-project defect prediction. IEEE Transactions on software Engineering 42(10):977–998

Xia X, Shihab E, Kamei Y, Lo D, Wang X (2016b) Predicting crashing releases of mobile applications. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, p 29

Xia X, Bao L, Lo D, Kochhar PS, Hassan AE, Xing Z (2017) What do developers search for on the web? Empirical Software Engineering pp 1–37

Yan M, Fu Y, Zhang X, Yang D, Xu L, Kymer JD (2016) Automatically classifying software changes via discriminative topic model: Supporting

multi-category and cross-project. Journal of Systems and Software 113:296–308

Yan M, Fang Y, Lo D, Xia X, Zhang X (2017) File-level defect prediction: Unsupervised vs. supervised models. In: Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on, IEEE, pp 344–353

Yan M, Xia X, Shihab E, Lo D, Yin J, Yang X (2018) Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering

Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H (2016) Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 157–168

Yoon Y, Myers BA (2012) An exploratory study of backtracking strategies used by developers. In: Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering, IEEE Press, pp 138–144