3-2018

Early prediction of merged code changes to prioritize reviewing tasks

Yuanrui FAN

Xin XIA

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

Shanping LI

Empir Software Eng
https://doi.org/10.1007/s10664-018-9602-0

Early prediction of merged code changes to prioritize reviewing tasks

Yuanrui Fan[1] · Xin Xia[2] · David Lo[3] · Shanping Li[1]

Abstract Modern Code Review (MCR) has been widely used by open source and proprietary software projects. Inspecting code changes consumes reviewers much time and effort since they need to comprehend patches, and many reviewers are often assigned to review many code changes. Note that a code change might be eventually abandoned, which causes waste of time and effort. Thus, a tool that predicts early on whether a code change will be merged can help developers prioritize changes to inspect, accomplish more things given tight schedule, and not waste reviewing effort on low quality changes. In this paper, motivated by the above needs, we build a merged code change prediction tool. Our approach first extracts 34 features from code changes, which are grouped into 5 dimensions: code, file history, owner experience, collaboration network, and text. And then we leverage machine learning techniques such as random forest to build a prediction model. To evaluate the performance of our approach, we conduct experiments on three open source projects (i.e., Eclipse, LibreOffice, and OpenStack), containing a total of 166,215 code changes. Across three datasets, our approach statistically significantly improves random guess classifiers and

Communicated by: Sven Apel

✉ Xin Xia
xin.xia@monash.edu

Yuanrui Fan
yrfan@zju.edu.cn

David Lo
davidlo@smu.edu.sg

Shanping Li
shan@zju.edu.cn

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[2] Faculty of Information Technology, Monash University, Melbourne, Australia

[3] School of Information Systems, Singapore Management University, Singapore, Singapore

Published online: 19 March 2018

two prediction models proposed by Jeong et al. (2009) and Gousios et al. (2014) in terms of several evaluation metrics. Besides, we also study the important features which distinguish merged code changes from abandoned ones.

Keywords Code review · Predictive model · Features

1 Introduction

Software code review, i.e., the practice of involving team members to critique changes made to a software system, has been an effective strategy to improve code quality before code changes are integrated into the main development branch (Shimagaki et al. 2016). In 1976, Fagan (2001) found that formal design and code inspections with in-person meetings reduce the number of post-release defects. Ackerman et al. (1984) and Aurum et al. (2002) found that code reviews improve the overall quality of software systems. Unfortunately, the cumbersome and time-consuming nature of conventional formal code inspection (e.g., in-person meetings and reviewer checklists) impedes its adoption in practice (Shull and Seaman 2008; Votta 1993).

Modern Code Review (MCR) provides an informal, lightweight, and tool-based variant of conventional formal code review, and it has been widely practiced in open source and proprietary software projects (Bacchelli and Bird 2013). During MCR, a developer submits a code change request (i.e., a patch) to a code review system (e.g., Gerrit[1]), and project moderators then assign this request to a set of reviewers (Xia et al. 2015b). Next, these code reviewers will inspect the patch, discuss it, and suggest potential improvements or fixes. However, not all of the code changes are eventually merged into a codebase. In our collected data,[2] we find 12% to 18% of code changes are eventually abandoned by reviewers.

Considering the substantial time and effort needed to be spent to inspect a code change is large, a tool which predicts early on (i.e., when a code change is submitted) whether a code change will be merged, can reduce the time and effort wasted on the inspection of abandoned code changes. Such wasted time can be used to review more promising code changes which eventually contribute to the codebase of a software project. We refer to the problem of predicting whether a change will eventually get merged early in the code review process as *merged change prediction*. In this paper, we focus on analyzing code changes in Gerrit, which is a popular code review management system, and is widely used by many open source projects such as Eclipse, LibreOffice, and OpenStack. Moreover, there have been a number of studies which analyze code changes in Gerrit (Gonzalez-Barahona et al. 2014; Mukadam et al. 2013; Baysal et al. 2013; McIntosh et al. 2014; Thongtanunam et al. 2016).

In this paper, we propose an approach to predict early on whether a code change will be merged. To do so, we extract 34 features from code changes which are grouped into five dimensions: code, file history, owner experience, collaboration network, and text. In the code dimension, we extract code features from patches, e.g., number of lines of code added or deleted in the patch, and number of changed source code files. In the file history dimension, we mine file modification history, and extract features such as number of

[1] https://code.google.com/p/gerrit/

[2] For more details, please refer to Table 1.

times files in a patch were changed, and number of developers who changed the files. In the owner experience dimension, we extract features based on previous behaviors of the code change owners, e.g., number of prior code changes submitted by the owner, and number of prior code changes he/she is assigned to inspect. In the collaboration network dimension, we first build a network based on prior collaborations of the owners and the reviewers for a code change owner. Next, we extract features of his/her corresponding node in the network such as its degree, closeness, and betweenness centrality. In the text dimension, we extract the features from natural language description of the code changes, such as number of words in the description, and whether the description contains words like "bug", "feature", or "improvement". All the 34 features can be automatically extracted when a review request is submitted initially. Using the extracted features, we leverage machine learning techniques such as random forest (Breiman 2001) to build models that can effectively predict merged code changes. Since there are only a small number of code changes which are abandoned (e.g., 17% of the code changes are abandoned in our collected data), we need to address a class imbalance problem. To solve this problem, we use a cost sensitive learning approach to improve the prediction accuracy for the minority class (i.e., abandoned code changes).

The usage scenarios of our proposed tool are as follows:

Without Tool. Bob is a developer in a large project team, and his main responsibility is to review code changes submitted by other developers. He is typically assigned to review more than 20 code changes in a single day. Without our tool, he can only randomly select code changes one at a time, inspect the patch, detect bugs, and discuss with other developers. He finds that it is hard for him to focus when he reviews more than 15 changes, and he often introduces errors when inspecting the remaining changes. Also, it wastes much of Bob's time and effort if a code change he reviews is eventually abandoned, which leads to Bob having less time to inspect other code changes.

With Tool. Bob's company adopts our tool. When Bob is assigned more than 20 code changes in a single day, he first runs our tool to prioritize the code changes. Our tool gives a ranked score for each change. The higher the rank scores, the more chance the code changes will be eventually merged. Then, Bob reviews the code changes in sequence. He finds that he can pay more attention to the changes with higher quality and more likelihood to be merged. And he spends less time and effort on the changes that are eventually abandoned.

Previous studies are related to ours (Jeong et al. 2009; Gousios et al. 2014). Jeong et al. (2009) proposed a set of features to predict whether a bug fix patch will be accepted, and these features include number of occurrences of certain keywords (e.g., boolean, break, and catch), and number of brackets appearing in a patch. Gousios et al. (2014) proposed 12 features to predict whether a pull request (PR) will be merged, and these features are divided into three dimensions: pull request, project, and developer. Our work is related but different from theirs: (1) The keyword occurrence features proposed by Jeong et al.—which detects appearance of keywords such as *interface*—are specific to Java.

Our features are programming language agnostic, which is different from Jeong et al.'s; (2) Gousios et al. focused on analyzing pull requests in Github. Some features they extracted from Github pull requests cannot be extracted from Gerrit code changes; (3) we extract more domain-specific features from code changes, and we not only extract features from patches, but also from collaboration network, modification history, and description of code

changes. Moreover, in this paper, we also make a comparison with these approaches when adapted for our problem.

To evaluate the performance of our approach, we collect code change datasets from three large open source projects, namely Eclipse, LibreOffice, and OpenStack, containing a total of 166,215 code changes. We use Area Under the Curve (AUC) (Huang and Ling 2005) and cost effectiveness (Arisholm et al. 2007; Mende and Koschke 2009; Rahman et al. 2012; Jiang et al. 2013a; Xia et al. 2015a) to evaluate the performance of our approach. AUC is a suitable metric since it evaluates the probability that a merged change is ranked higher than an abandoned change. Cost effectiveness evaluates prediction performance given limited resources, e.g., percentage of code changes to inspect. Following previous studies (Rahman et al. 2012; Jiang et al. 2013a; Xia et al. 2015a, 2016a, b; Zhang et al. 2015), we use EffectivenessRatio@20% (ER@20%), which evaluates the percentage of merged code reviews when inspecting the top 20% suspicious merged code changes returned by a prediction method, as the default cost effectiveness metric. Experimental results show our approach achieves an average AUC of 0.73, which improves random guess, weighted random guess, Jeong et al.'s approach (Jeong et al. 2009), and Gousios et al.'s approach (Gousios et al. 2014) by 46%, 46%, 30%, and 12%, respectively. Moreover, our approach achieves an average ER@20% of 0.95. Following Costa et al. (2016), we measure *normalized improvement* which considers the room for improvement, between our approach and the baseline methods in terms of ER@20%. And the *normalized improvements* of our approach over random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach are 69%, 69%, 64% and 38%, respectively.

Of the 34 features we extract, we find that merged rate of prior code changes submitted by the owner (i.e., *merged_ratio*), number of prior code changes submitted by the owner (i.e., *change_num*), number of prior code changes submitted by the owner that contain at least one subsystem[3] affected by current code change (i.e., *subsystem_change_num*) and clustering coefficient of the owner's corresponding node in collaboration network (i.e., *clustering_coefficient*) are the most important features to distinguish merged code changes from abandoned code changes across three datasets.

The paper makes the following contributions:

1. We propose an approach which includes a total of 34 features to predict whether a code change will be merged. We extend previous studies and extract features not only from patches but also from modification history, collaboration network and description of patches. We also investigate the most important features that distinguish merged code changes from abandoned code changes.
2. We construct an experiment on a broad range of datasets containing a total of 166,215 code changes from three large-scale open source projects, and the experimental results show that our approach achieves statistically significant and substantial improvements over the baselines.

Paper Organization The remainder of this paper is organized as follows. Section 2 presents results of our preliminary study, which highlights the importance to predict early on whether a code change will be merged. Section 3 presents research questions investigated in this study, elaborates how we collect data, and describes other experiment setup. Section 4

[3]Subsystem is defined in Section 3.3.

presents experiment results which answer a number of research questions. Section 5 discusses the impact of using multiple classifiers and the effectiveness of our approach for cross-project prediction. Section 6 briefly reviews related work. Section 7 concludes and mentions future work.

2 Preliminary Study

To assess the necessity of our study, we sent 200 emails to developers from Eclipse, LibreOffice, and OpenStack communities. In the emails, we only asked two simple questions: *Do you need a tool to predict whether a code change will eventually get merged early in the code review process? And why?* We received 62 replies, and 59 developers agreed that they need a tool to predict whether code changes eventually get merged. Only 2 pointed out they do not need such a tool since *it would be difficult to build such a tool with high accuracy*. Some comments we received are listed below:

- *"I often need to review about 20 code change requests in a single day. It is really hard to make the decision on whether a code change should be merged unless I comprehend the code, and try to integrate it into the codebase. A tool on merged change prediction can help* ***prioritize the 20 code review tasks*** *with different priorities, so that I can simply inspect the code changes from top to down. It is really promising to have such a tool."*
- *"A tool to predict merged code changes will help to* ***build my confidence*** *on identifying whether I should accept a code review request. Sometimes I felt that I should accept the review request but with low confidence, in such a case, this tool can aid me to make the decision."*
- *"I do not want to spend several days to inspect a code change, discuss with other people, and provide a number of comments, but finally the code change is abandoned. I* ***feel depressed towards the abandoned changes****, since most of them will* ***occupy me plenty of time.*** *Thus, I need a tool to predict which code changes should be merged."*

From the above reviewer comments, developers need a merged change prediction tool to (1) help prioritize code changes they are asked to inspect, (2) increase their confidence on whether they should merge the changes, and (3) reduce the time and effort wasted due to abandoned changes. Our finding is consistent with Gousios et al. (2015). They found that in a pull request development model (e.g., Github), integrators typically need to manage large amounts of contribution requests simultaneously and they have difficulties with prioritizing contributions that are to be merged.

As one developer replied, much time and effort of reviewers is wasted on abandoned changes. To illustrate this, we calculate the total number of abandoned code changes and characterize the amount of time these abandoned code changes were opened for review in 2015 for Eclipse, LibreOffice and OpenStack projects. In 2015, 2,394, 713 and 17,030 code changes were abandoned in Eclipse, LibreOffice and OpenStack, respectively. The total amount of time these code changes were opened for review in the three projects is 21,444, 6,696 and 184,170 hours, respectively. Some code changes had been open for a very long time until they were eventually abandoned. For example, the owner and reviewers of change 262127[4] in OpenStack spent several months continuously discussing whether to merge the

[4] https://review.openstack.org/#/c/262127/

code change but the change was eventually abandoned. Thus, reviewers waste much time on abandoned code changes.

3 Experiment Setup

In this section, we first present the steps to collect code change data. Then, we present the overall framework of our approach. Next, we elaborate the details of the 34 features we extract from code changes, which are grouped in five dimensions: code, file history, owner experience, collaboration network, and text. After that, we present evaluation setup and four baselines. Later, we present the prediction model used in our study (i.e., random forest) and a cost sensitive learning approach which is used to address the class imbalance problem. Finally, we present evaluation metrics that we use to measure the prediction performance of our approach and baselines. In this paper, we are interested in answering four research questions:

RQ1 Can we effectively predict which code changes will be merged?
RQ2 How effective is our prediction model when all features are used than when only a subset is used?
RQ3 How effective are our approach and the baseline methods when different percentages of code changes are inspected?
RQ4 Which features are most important in identifying merged code changes?

Answering RQ1 sheds light on the effectiveness of our approach to predict code changes that will be merged or abandoned, compared to random guess, weighted random guess and existing state-of-the-art approaches (Jeong et al. 2009; Gousios et al. 2014). Answering RQ2 highlights the effectiveness of our approach compared to models built using features of each dimension. Answering RQ3 sheds lights on the effectiveness of our approach and the baselines when different percentages of code changes are inspected. The answer of RQ4 presents the most important features in identifying merged code changes. Note that we have related but different goals when investigating RQ2 and RQ4. By investigating RQ2, we give an answer to the question: *"Does our approach benefit from combining features of the five dimensions?"* By investigating RQ4, we study the most important features that distinguish merged code changes from abandoned ones.

3.1 Data Collection

We collect all meta-data information (i.e., creation time, status, description of patches, commit ID, owner, and reviewers) of code changes stored in the Gerrit systems of Eclipse, LibreOffice, and OpenStack. Following a prior study (Yang et al. 2016), we use the REST APIs provided by the Gerrit systems[5] to collect the meta-data. In Gerrit, the status of a code change can be "Open", "Merged", or "Abandoned". In our study, we remove the code changes whose status are "Open" since they may not have been assigned to reviewers, and the set of reviewers may still change.

There are many sub-projects in the three projects. To avoid studying inactive/dead sub-projects, we choose those that contain more than 200 merged changes, and 54, 3, and

[5]https://gerrit.wikimedia.org/r/Documentation/rest-api.html

248 sub-projects are left in our Eclipse, LibreOffice, and OpenStack datasets, respectively. Moreover, we remove code changes whose description contains words "NOT MERGE" or "IGNORE", since the owners of these code changes clearly know these changes would be abandoned, and there is no need for further inspections. Moreover, we notice that for some code changes, the reviewers are the same as the owners. We also remove these code changes, since the owner does not require the need of others to review his/her code. Table 1 presents the statistics of the collected data. In total, we collect 166,215 code changes, and among them, 83% of code changes are eventually merged. Our findings are consistent with Gousios et al.'s findings; they found that 84.73% pull requests are eventually merged (Gousios et al. 2014).

Based on the code changes we collect, we also extract the patches corresponding to these code changes. To do so, we first use *git clone* to download all the repositories used in our study, since each sub-project uses a separate repository managed by the Gerrit system. In our study, we clone all of these repositories into our local server. Next, we extract commit IDs from the collected code changes, and we use *git ls-remote* to get a list of patch set IDs which correspond to a specific code change. In our study, we only extract the first patch set in the list of patch sets since we focus on predicting whether a code change will be merged when it is submitted initially. Then, we use *git fetch* to get the patch.

3.2 Overall Framework of Our Approach

Figure 1 shows the overall framework of our approach. The framework includes two phases: model building and prediction.

In the model building phase, our framework takes as input a set of training code changes with known labels (merged or abandoned). The framework first extracts various features from the code changes. The features are quantifiable characteristics of code changes that could potentially differentiate merged code changes from abandoned ones. In this paper, we consider features that are grouped into five dimensions: code, file history, owner experience, collaboration network and text. Then, based on these features and the labels of training code changes, our framework uses random forest to build a model. Moreover, it applies cost sensitive learning to deal with imbalanced data. Finally, the framework produces a prediction model that could predict early on whether a code change will be merged and output a likelihood score for the code change to be merged.

In the prediction phase, our framework takes as input a set of testing code changes whose labels are to be predicted. Features are first extracted from these code changes. Then, model learned in the model building phase is applied to predict the labels of these code changes by analyzing the extracted features. Finally, the framework ranks the code changes in

Table 1 Statistics of the collected data

Project	Time period	# Changes	# Merged	# Abandoned
Eclipse	2009.10–2016.12	46,600	38,488 (83%)	8,112 (17%)
Libre.	2012.03–2016.12	22,655	19,870 (88%)	2,785 (12%)
OpenS.	2015.01–2015.12	96,960	79,930 (82%)	17,030 (18%)
Total		166,215	138,288 (83%)	27,927 (17%)

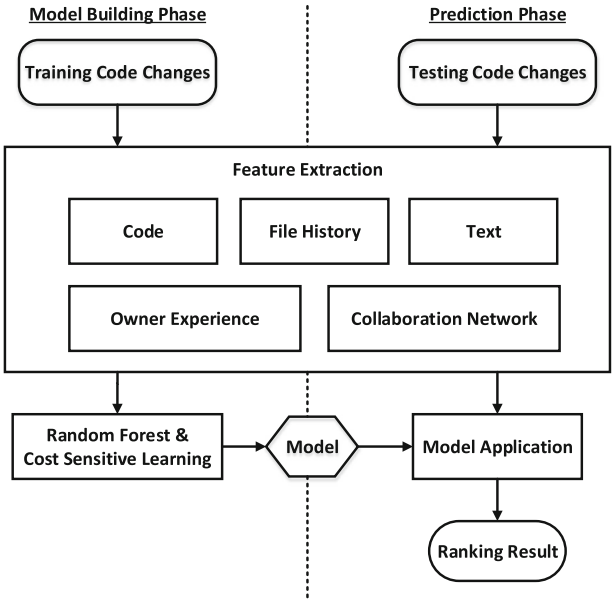


Fig. 1 Overall framework of our approach

descending order according to the likelihood scores output by the model for each of them. The ranking result can be used to prioritize these code changes for reviewers.

In Sections 3.3 and 3.5, we further elaborate the detail information of studied features and model building techniques (i.e., random forest and cost sensitive learning) in our framework, respectively.

3.3 Studied Features

Our features are mainly based on the results of previous studies. Tsay et al. (2014) found that both social and technical factors could have effect on contribution acceptance. Thus, we consider extracting features from these two aspects. With regards to social factors of a code change, we extract features to characterize change owner's experience and his/her activities in the collaboration network. Our features that characterize change owner's experience are derived based on those considered in Kamei et al. (2013) and Gousios et al. (2014). Our features that characterize change owner's activities in the collaboration network are derived based on those considered in Zanetti et al. (2013). With regards to technical factors of a code change, we extract features to characterize the source code, modification history of the files and the textual description of the change. Our features that characterize the source code of a change are derived based on those considered in Kamei et al. (2013). Also, our features that characterize modification history of the files in the change are derived based on those considered in Kamei et al. (2013). Our features that characterize the textual description of the change are derived based on the findings of Thongtanunam et al. (2016) and Herzig et al. (2013).

Totally, we extract 34 features which are potentially related to merged code changes. Table 2 summarizes the set of 34 features we investigate, which are grouped in 5 dimensions: code, file history, owner experience, collaboration network, and text.

Table 2 Studied features

Dimension	Feature name	Description
Code	lines_added_num	Number of inserted lines in this code change
	lines_deleted_num	Number of deleted lines in this code change
	changed_file_num	Number of changed files in this code change
	file_added_num	Number of added files in this code change
	file_deleted_num	Number of deleted files in this code change
	directory_num	Number of modified directories in this code change
	subsystem_num	Number of modified subsystems in this code change
	modify_entropy	Distribution of modified code across files in this code change
	language_num	Number of programming languages used in this code change
	file_type_num	Number of file types in this code change
	segs_added_num	Number of *added_code_segment*s in this code change
	segs_deleted_num	Number of *deleted_code_segment*s in this code change
	segs_updated_num	Number of *updated_code_segment*s in this code change
File history	changes_files_modified	Number of times files in this code change were modified before.
	file_developer_num	Number of developers who changed files in this code change
Owner experience	change_num	Number of prior code changes submitted by the owner of this code change
	recent_change_num	Number of prior code changes submitted by the owner of this code change that is counted according to our weighting scheme (see paragraph) in recent 120 days
	subsystem_change_num	Number of prior code changes submitted by the owner of this code change, that contain at least one subsystem affected by this code change
	review_num	Number of prior code changes the owner of this code change is assigned to inspect
	merged_ratio	Merged rate of prior code changes submitted by the owner of this code change
	recent_merged_ratio	Number of merged code changes submitted by the owner of this code change, counted according to our weighting scheme (see paragraph) in recent 120 days prior to this code change and normalized over *recent_change_num*
	subsystem_merged_ratio	Merged rate of prior code changes submitted by the owner of this code change, that contain at least one subsystem affected by this code change
Collaboration network	degree_centrality	These metrics are used to measure this code change owner's degree of activity in collaboration process of the corresponding project prior to this code change (Zanetti et al. 2013)
	closeness_centrality	
	betweenness_centrality	
	eigenvector_centrality	
	clustering_coefficient	

Table 2 (continued)

Dimension	Feature name	Description
	k_coreness	
Text	msg_length	Number of words in description of this code change
	has_bug	Whether description of this code change contains word "bug"
	has_feature	Whether description of this code change contains word "feature"
	has_improve	Whether description of this code change contains word "improve"
	has_document	Whether description of this code change contains word "document"
	has_refactor	Whether description of this code change contains word "refactor"

Code Dimension refers to features that are based on the source code in the patches. Weißgerber et al. (2008) found that size of a patch would affect whether it would be accepted. We use features *lines_added_num*, *lines_deleted_num*, *changed_file_num*, *file_added_num* and *file_deleted_num* to quantify the size of the patch. And we pre-process the features *lines_added_num* and *lines_deleted_num* by taking the logarithms of the original values, since we find the range of these two features are too large.

The diffusion of a change is also an important feature in defect prediction (Mockus and Weiss 2000). We expect that the diffusion of a change can influence likelihood whether a change would be merged. We use features proposed by Kamei et al. (2013) to measure diffusion of code changes, i.e., *directory_num*, *subsystem_num* and *modify_entropy*. We use bottom directory of a file as a directory to measure *directory_num*. We use top directory of a file as a subsystem to measure *subsystem_num*. To illustrate, for a file with a path, "org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java", its subsystem is "org.eclipse.jdt.core" and its directory is "org.eclipse.jdt.core/jdom/.../dom". To calculate *modify_entropy*, we follow Kamei et al. (2013). Entropy is defined as: $-\Sigma_{k=1}^{n}(p_k * \log_2 p_k)$. Note that n is number of files modified in the change, and p_k is calculated as the proportion of lines modified in $file_k$ among lines modified in this code change.

By observing the modified files in some code changes, we find that many modified source code files are of different types (i.e., they have different file extensions), and written in different programming languages. Thus, we use features *file_type_num* and *language_num* to measure unique number of different file extension names, and programming languages, respectively. To calculate *file_type_num*, we count number of file extensions in the code change. And we identify the programming languages used in a code change by matching files in the code change with languages' corresponding file extensions. For example, we identify a file with extension "java" as a Java file. We consider the following programming languages: Java, C/C++, Python, JavaScript, bash, html, php, Ruby, and Go. Number of extensions is not equal to number of programming languages in a code change, since a code change may contain documentation, images, or configuration files which are not related to programming languages. Also, a programming language such as C++ might use multiple types of file extensions such as "c", "h", "cpp" and "hpp".

Figure 2 presents an example patch file. We define the code between "@@" as a *code segment*. We call a *code segment* only with inserted lines as *added_code_segment* and a *code segment* only with deleted lines as *deleted_code_segment*. And a *code segment* with both inserted and deleted lines is called *updated_code_segment*. A code change with more *code segment*s modified is likely more complex and requires reviewer more effort and time to comprehend it. We use features *segs_added_num*, *segs_deleted_num* and *segs_updated_num*

```
--- a/nova/virt/libvirt/driver.py
+++ b/nova/virt/libvirt/driver.py
@@ -2501,6 +2501,9 @@ class LibvirtDriver(driver.ComputeDriver):
         if image_meta.obj_attr_is_set("id"):
             rescue_image_id = image_meta.id

+        # NOTE(dprince): for rescue console.log may already exist... chown it.
+        self._chown_console_log_for_instance(instance)
         rescue_images = {
             'image_id': (rescue_image_id or
                          CONF.libvirt.rescue_image_id or instance.image_ref),
@@ -2912,9 +2915,6 @@ class LibvirtDriver(driver.ComputeDriver):
```

code segment

Fig. 2 An example patch file

to quantify the complexity of code changes in *code segment* level. The advantage of these features is that they are general for all kinds of text files and programming languages.

File History Dimension refers to features that are based on file modification history recorded by the Gerrit systems. Graves et al. (2000) found that number of previous changes to a file is a good indicator to detect buggy files. Matsumoto et al. (2010) found that files which are previously touched by more developers are more likely to induce defects. Here, following Kamei et al.'s study (Kamei et al. 2013), we use features *changes_files_modified* and *file_developer_num* to quantify history information about files modified in a code change.

Owner Experience Dimension refers to features which are based on the experience of a code change owner. Mockus and Weiss (2000) found that developer experience is an essential piece of information for predicting software failures. Baysal et al. (2013) and Jiang et al. (2013b) found that patch writer experience significantly impacts code review outcomes.

Following Kamei et al.'s study (Kamei et al. 2013), we use features *change_num*, *recent_change_num* and *subsystem_change_num* to measure the experience of the owner. To calculate *recent_change_num*, we count number of changes submitted by the owner in past 120 days of this code change according to our weighting scheme which assigns high weights to changes submitted recently: $\frac{1}{\frac{n}{30}+1}$, where n is the number of days that has passed since a prior code change submitted by the owner to the date this code change is submitted. For example, if an owner of a newly submitted change submitted 4 changes 30 days ago, 6 changes 60 days ago, and 8 changes 90 days ago, then *recent_change_num* is 6 (i.e., $= \frac{4}{\frac{30}{30}+1} + \frac{6}{\frac{60}{30}+1} + \frac{8}{\frac{90}{30}+1}$). If a code change owner often submits code changes in recent time prior to the change, we expect that he/she will be more familiar with the recent developments of the projects, and thus the code change maybe more likely to be merged.

Also, if a code change owner has reviewed many changes submitted by other developers, he/she would be more familiar with coding standards and operations of Gerrit system. Thus, we use *review_num* as a metric to also quantify for the owner's experience.

Besides, Gousios et al. (2014) used acceptance rate prior to current pull request as a feature to predict pull request outcomes (i.e., merged or abandoned). Following their study, we use the same feature which is named as *merged_ratio* in our study. Moreover, we also extract other features: *recent_merged_ratio* and *subsystems_merged_ratio*. The feature *recent_merged_ratio* is used to measure recent performance of the owner prior to current code change. The feature *subsystem_merged_ratio* is used to quantify the owner's familiarity with the subsystems affected by this code change.

Collaboration Network Dimension Baysal et al. (2013) found that collaboration factors of the code change owners (i.e., their level of participation within the project) can influence code review outcomes. They found that the more active a developer is in the project, the faster and more likely his/her patches will be merged. Zanetti et al. (2013) proposed a technique to predict whether a bug report is valid based on collaborations of reporters. Following these studies, we construct a network based on collaborations of owners and reviewers. Next, for the code change owner, we extract features of his/her corresponding node in the network.

For a newly submitted code change, we first extract its owner and submission date. Then, following Zanetti et al. (2013), we collect code changes prior to the newly submitted code change in a window of 30 days before the submission date. Next, we construct an undirected graph based on owners and reviewers of these code changes. In the graph, nodes denote owners and reviewers of the code changes. And for each code change, the code change owner and the reviewers are connected by an edge. Subsequently, we extract the Largest Connected Component (LCC) of the generated graph and check whether the code change owner is a member of the LCC. If the code change owner exists in the LCC, we quantify the code change owner's degree of activity using the six features proposed by Zanetti et al. (2013) namely *degree_centrality*, *closeness_centrality*, *betweenness_centrality*, *eigenvector_centrality*, *clustering_coefficient* and *k_coreness*. We use the Python package NetworkX[6] to construct collaboration networks and extract features from the networks.

Text Dimension refers to features that capture textual characteristics of the commit message. Thongtanunam et al. (2016) found that the description length of a patch is related to its likelihood of receiving poor comments. We count the number of words in the commit message to measure the description length of this code change. Commit message containing more information about a code change may help reviewers comprehend the change more easily. Herzig et al. grouped code changes into six clusters: bug fixing, adding new feature, improvement, documentation, refactoring and other (Herzig et al. 2013). We extract the purpose of code changes by matching the related words in the commit message. Features *has_bug*, *has_feature*, *has_improve*, *has_document*, *has_refactor* are used to represent the purpose of code changes. To calculate these features, we simply check whether the commit message of a code change contains the related words. For example, if the commit message contains word "bug", *has_bug* will be set true—please refer to Table 2 for more words used.

3.4 Evaluation Setup & Baselines

To simulate the usage of our approach in practice, we use the same longitudinal data setup described in Bhattacharya and Neamtiu (2010), Tamrawi et al. (2011), Jeong et al. (2009), Xia et al. (2017). The code changes extracted from each repository in Table 1 are first sorted in chronological order of creation time, and then divided into 11 non-overlapping windows of equal sizes. The process proceeds as follows: First, in fold 0, we train using code changes in frame 0, and test the trained model using the code changes in frame 1. Then, in fold 1, we train using code changes in frame 0 and frame 1, and proceed in a similar way (like frame 1) to test using code changes in frame 2, and so on. In the final fold (fold-9), we train using code changes in frame 0–9, and test using code changes in frame 10. We then compute the average AUC, ER@20%, precision, recall and F1 scores for merged and abandoned code changes across the 10 folds. We use this evaluation setup in RQ1, RQ2 and RQ3.

[6] http://networkx.readthedocs.io/en/stable/

To make a comparison, we choose four baselines, i.e., random guess, weighted random guess, Jeong et al.'s approach (Jeong et al. 2009), and Gousios et al.'s approach (Gousios et al. 2014).

Random guess predicts randomly whether a code change will be merged, effectively acting as a coin flip, thus its AUC is always 0.5. Its precision is the ratio of merged code changes to total number of code changes in the dataset. Since random guess has two possible outcomes (e.g., merged/abanoned), its average recall is 0.50.

Weighted random guess is a variant of random guess. Compared to random guess, weighted random guess also randomly predicts a code change to be merged or abandoned and it also outputs a randomly ranked list for changes in the testing dataset. Thus, its AUC is also 0.5. However, unlike random guess, weighted random guess considers the class distribution in training dataset. To illustrate, let us consider the proportion of merged code changes in training dataset to be $m\%$. In the testing dataset, weighted random guess randomly chooses $m\%$ of the code changes and predicts these code changes as merged. Its average recall for merged and abandoned code changes is thus $m\%$ and $1-m\%$, respectively.

Jeong et al. proposed a set of features to predict whether a bug fix patch will be accepted, and we remove features extracted from bug reports since we do not have bug reports data. Notice that the main programming language of Eclipse, LibreOffice and OpenStack are Java, C++ and Python, respectively. And the keyword occurrence features proposed by Jeong et al. are specific to Java (e.g., they used number of occurrences of keyword "interface" as a feature, but "interface" is not a keyword in C++ or Python). To build a prediction model using Jeong et al.'s approach for Eclipse dataset, we use all the keywords Jeong et al. proposed. To build prediction models using Jeong et al.'s approach for LibreOffice and OpenStack datasets, we use keywords proposed by Jeong et al. that are common in multiple programming languages. These keywords include: "if", "else", "for", "while", "break", "continue", "return" and "try". Jeong et al. used Bayesian network to build their prediction models. We find that the Bayesian network models built using their features achieve better performance than random forest models in terms of AUC score and EffectivenessRatio@20% (ER@20%). In this paper, we re-implement Jeong et al.'s approach and following Jeong et al.'s study, we use Bayesian network to implement their approach.

For Gousios et al.'s approach, we remove features *team_size* (number of core team members in a project), and *perc_ext_contribs* (the ratio of commits from external members over core team members in the last 3 months) since the concept *core team* is not defined in the Gerrit system. Gousios et al. used random forest to construct prediction models. In this paper, we re-implement Gousios et al.'s approach and following their study, we use random forest to build prediction models that are based on Gousios et al.'s features. Note that according to our empirical experimental results, random forest performs better than Bayesian network on Gousios et al.'s features in terms of AUC and ER@20%.

3.5 Approach

We use the proposed features to characterize a code change. These features are then used to learn a prediction model to predict whether a code change will be merged when it is initially submitted. In this paper, by default, we use random forest (Breiman 2001) to construct the prediction model, and we use the random forest implementation in Weka (Hall et al. 2009).

Random forest is an ensemble approach, which is specifically designed for decision tree classifiers (Breiman 2001). The general idea behind random forest is to combine multiple decision trees for prediction. Each decision tree in a random forest is built based on a random subset of the features. Random forest adopts the mode of the class labels output by

individual trees. The major advantage of random forest is that it is generally highly accurate and feature importance can be generated automatically. Since random forest unifies many trees that are learned differently, it can avoid overfitting problem and is not sensitive to outliers.

We notice that the number of merged code changes is much more than the number of abandoned ones, i.e., a class imbalance problem (He and Garcia 2009) exists. If the class imbalance problem is not handled properly, it might degrade the performance of the prediction model (Grbac et al. 2013). To address this challenge, we use a cost sensitive learning approach (Elkan 2001) which is implemented in Weka (Hall et al. 2009). Unlike cost-insensitive learning, cost sensitive learning is a type of learning that takes misclassification cost into account, i.e., it treats different misclassification cases differently. Cost sensitive learning has been utilized to deal with class imbalance problem in many software engineering studies (Khoshgoftaar et al. 2002; Zheng 2010; Liu et al. 2014).

Cost sensitive learning approach takes misclassification costs into account and the misclassification costs are defined by a cost matrix. Table 3 presents the cost matrix for cost sensitive learning used by our approach. In Table 3, $c(i, j)$—$i, j \in \{0, 1\}$—denotes the cost of labeling a sample of class i as class j. In our case, $c(0, 1)$ represents the cost of misclassifying a merged code change as abandoned, while $c(1, 0)$ represents the cost of misclassifying an abandoned code change as merged. Given a cost matrix, the aim of cost sensitive learning is to generate a prediction model with minimum misclassification cost according to the cost matrix.

There is no cost for correct classification (Liu and Zhou 2006), i.e., $c(0, 0)$ and $c(1, 1)$ are set to 0. Since the minority class in this paper is the abandoned code change, the cost of misclassifying an abandoned code change should outweigh the cost of misclassifying a merged one, i.e., $c(1, 0) > c(0, 1)$. We denote $c(1, 0)/c(0, 1)$ as *cost ratio*. A larger *cost ratio* means that the model has a stronger bias towards the minority class, i.e., abandoned code change. As Weiss et al. (2007) recommended, the specific values of $c(1, 0)$ and $c(0, 1)$ are not necessary to be determined. We only need to determine *cost ratio*. Furthermore, Weiss et al. (2007) recommended that users can first determine a ratio α, then *cost ratio* can be calculated as the production of α and the class distribution of training dataset, i.e.:

$$cost\ ratio = \alpha \times \frac{number\ of\ merged\ changes\ in\ training\ dataset}{number\ of\ abandoned\ changes\ in\ training\ dataset} \quad (1)$$

By default, we set α as 1, i.e., we set *cost ratio* as the class distribution in training dataset. The choice of α can impact the prediction performance of our approach. A larger α results in a larger *cost ratio*, then the model would have a stronger bias towards abandoned code changes. In this paper, we also vary the value of α and investigate the sensitivity of the prediction performance of our approach towards the choice of α.

3.6 Evaluation Metrics

For each code change, there would be 4 possible classification outcomes: a code change is classified as *merged* when it is truly *merged* (true positive, TP); it can be classified as

Table 3 Cost matrix for cost sensitive learning

	Actual merged	Actual abandoned
Predict merged	$c(0, 0)$	$c(1, 0)$
Predict abandoned	$c(0, 1)$	$c(1, 1)$

merged when it is truly *abandoned* (false positive, FP); it can be classified as *abandoned* when it is truly *merged* (false negative, FN); it can be classified as *abandoned* when it is truly *abandoned* (true negative, TN). Based on TP, TN, FP, and FN, we calculate AUC, cost effectiveness, precision, recall, and F1-score as follows:

AUC: Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) plot is a widely used measure in many software engineering studies (Lamkanfi et al. 2010; Lessmann et al. 2008; Romano and Pinzger 2011; Bao et al. 2017). The AUC score ranges from 0 to 1 and larger AUC value indicates better prediction performance. Notice that random guess classifier always gets an AUC of 0.5 and any prediction model achieving AUC score more than 0.5 is more effective than random guess classifier. AUC measures the prediction performance across all the thresholds and it is insensitive to cost and class distributions (Rajbahadur et al. 2017). Lessmann et al. (2008) recommends that AUC should be used as the primary accuracy indicator to compare the performance of prediction models. Romano and Pinzger (2011) concluded that a prediction model with an AUC score above 0.7 is often considered to have adequate classification performance. In our context, the AUC score evaluates the probability that a classifier ranks a randomly chosen merged code change higher than a randomly chosen abandoned code change. Since the main purpose of our study is to prioritize code changes that reviewers are assigned to review, the AUC score is the most important evaluation metric.

Cost Effectiveness: Cost effectiveness (Arisholm et al. 2007; Mende and Koschke 2009; Rahman et al. 2012; Jiang et al. 2013a; Xia et al. 2015a; Huang et al. 2017) is an evaluation metric which is used to measure prediction performance given a cost limit, and also simulates the practical usage of our tool. In practice, considering limited budget, developers can only inspect a limited number of code changes, and they would expect to identify as many merged code changes as possible. In our context, the cost is a fixed number of code changes to inspect, and the benefit is the percentage of merged code changes that can be identified. It is desirable to catch as many merged code changes as possible while minimizing the number of code changes to inspect. Cost effectiveness is an appropriate measure that can evaluate how effective is a prediction model to prioritize code changes that reviewers are assigned to inspect. We use EffectivenessRatio@K% (ER@K%), which evaluates the percentage of merged code changes over the top K% suspicious merged code changes, as our cost effectiveness metric. ER@K% is also an important evaluation metric for our study.

Specifically, we represent a prediction model as *pm*. To evaluate the ER@K% of *pm*, we first rank the code changes in descending order according to the likelihood score that *pm* outputs for each of them. Also we count the number of merged code changes that appear in the top K% in the ranking, which is denoted as $N_{K\%}(pm, merged)$. Also, we count the number of code changes in the top K% in the ranking, which is denoted as $N_{K\%}$. Based on these two numbers, ER@K% is computed as:

$$ER@K\% = \frac{N_{K\%}(pm, merged)}{N_{K\%}} \tag{2}$$

Following previous studies (Arisholm et al. 2007; Mende and Koschke 2009; Rahman et al. 2012; Jiang et al. 2013a; Xia et al. 2015a), by default, we set the number of code changes to inspect as 20% of the total number of code changes and we use ER@20% as the default cost effectiveness metric. We also investigate the performance of our approach with various K values.

Note that as mentioned in Section 3.4, both random guess and weighted random guess output randomly ranked list for the testing dataset. Thus, for the same testing dataset, the two classifiers achieve the same ER@K% value for any K from 0 to 1 according to the calculation of ER@K%.

Normalized Improvement: Normalized improvement is a measure proposed by Costa et al. (2016) to evaluate the improvement between two methods in terms of an evaluation metric (e.g., ER@20%). Normalized improvement takes the room for improvement into consideration. In our paper, all ER@20% scores of our approach and the baseline methods are very high (above 0.8). Directly comparing ER@20% scores of our approach and the baseline methods may underestimate the improvement of our approach over the baseline methods. For example, if our approach and a baseline method achieves an ER@20% of 0.95 and 0.93, respectively, our approach only improves the baseline method by 2% by directly comparing the ER@20% scores. However, the room for improving over the baseline method is only 7% (from 93% to 100%) and our approach achieves 29% (2% ÷ 7%) of this possible gain.

Thus, in addition to directly comparing ER@20% scores of our approach and the baselines, we also compute the normalized improvements of our approach over the baselines in terms of ER@20%. Following Costa et al. (2016), we normalize the percentage of improvement considering the room for improvement. We denote ER@20% scores of our approach and a baseline method as $ER_{20\%}(ours)$ and $ER_{20\%}(baseline)$, respectively. Then, the normalized improvement of our approach over the baseline method in terms of ER@20% can be computed as:

$$NormalizedImprovement = \frac{ER_{20\%}(ours) - ER_{20\%}(baseline)}{1 - ER_{20\%}(baseline)} \tag{3}$$

Merged Precision: is the proportion of code changes that are correctly labeled as *merged* among those that are predicted as *merged*, i.e.:

$$P(M) = \frac{TP}{TP + FP} \tag{4}$$

Merged Recall: is the proportion of merged code changes that are correctly labeled, i.e.:

$$R(M) = \frac{TP}{TP + FN} \tag{5}$$

Abandoned Precision: is the proportion of code changes that are correctly labeled as *abandoned* among those that are predicted as *abandoned*, i.e.:

$$P(A) = \frac{TN}{TN + FN} \tag{6}$$

Abandoned Recall: is the proportion of abandoned code changes that are correctly labeled, i.e.:

$$R(A) = \frac{TN}{FP + TN} \tag{7}$$

F1-score: is a summary metric that combines both precision and recall to measure the performance of the prediction model. This metric can evaluate if an increase in precision (recall) outweighs a reduction in recall (precision). It is calculated as the harmonic mean of Precision and Recall. For merged code changes, it is:

$$F1(M) = \frac{2 \times P(M) \times R(M)}{P(M) + R(M).} \tag{8}$$

Table 4 AUC, ER@20%, F1, precision, and recall for merged and abandoned code changes of our approach compared with the baselines

Project	Approach	AUC	ER@20%	Merged			Abandoned		
				F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	Ours	**0.71**	**0.93**	0.87	**0.87**	0.88	**0.39**	0.40	0.39
	Random guess	0.50	0.82	0.62	0.82	0.50	0.26	0.18	**0.50**
	Weighted random guess	0.50	0.82	0.83	0.82	0.83	0.17	0.18	0.17
	Jeong et al. (2009)	0.55	0.84	0.86	0.83	0.89	0.18	0.22	0.16
	Gousios et al. (2014)	0.64	0.90	**0.90**	0.83	**0.97**	0.17	**0.42**	0.11
Libre.	Ours	**0.73**	**0.96**	0.92	**0.91**	0.93	**0.31**	0.34	0.29
	Random guess	0.50	0.88	0.64	0.88	0.50	0.19	0.11	**0.50**
	Weighted random guess	0.50	0.88	0.88	0.88	0.87	0.12	0.12	0.12
	Jeong et al. (2009)	0.53	0.89	**0.93**	0.88	**0.99**	0.05	0.12	0.03
	Gousios et al. (2014)	0.64	0.94	**0.93**	0.89	**0.99**	0.08	**0.35**	0.04
OpenS.	Ours	**0.74**	**0.95**	0.89	**0.87**	0.91	**0.41**	0.45	0.37
	Random guess	0.50	0.83	0.62	0.83	0.50	0.26	0.17	**0.50**
	Weighted random guess	0.50	0.83	0.82	0.83	0.82	0.18	0.17	0.18
	Jeong et al. (2009)	0.59	0.86	**0.91**	0.83	**1.00**	0.10	**0.79**	0.06
	Gousios et al. (2014)	0.68	0.93	**0.91**	0.84	0.98	0.22	0.57	0.13

The best results are in bold

For abandoned code changes, it is:

$$F1(A) = \frac{2 \times P(A) \times R(A)}{P(A) + R(A)} \tag{9}$$

4 Results

In this section, we presents answers of the four research questions presented in Section 3.

(RQ1) Can we effectively predict which code changes will be merged?

To evaluate the performance of our approach and the baselines, we calculate the average AUC, ER@20%, F1, precision and recall for merged and abandoned code changes across the 10 folds. Table 4 presents the evaluation results of our approach compared with the baselines. On average, across the three datasets, our approach achieves an AUC, ER@20%, F1(M) and F1(A) of 0.73, 0.95, 0.89 and 0.37, respectively. Our approach shows improvements in AUC, ER@20% and F1(A) compared with the baselines. We apply Wilcoxon signed-rank test (Wilcoxon 1945) with a Bonferroni correction (Abdi 2007) to investigate whether the improvements of our approach over baselines are statistically significant (p-value < 0.05). We also use Cliff's delta (Cliff 2014),[7] which is a non-parametric effect size measure that quantifies the amount of difference between two sets of values. The statistical

[7]Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size respectively.

tests take as input the AUC, ER@20%, F1-scores for abandoned code changes calculated in the 10 evaluation folds of our approach and the baselines. Table 5 presents the adjusted p-values and Cliff's delta for our approach compared with the baselines.

Notice that the F1(M) of our approach is much higher than random guess and weighted random guess but a little lower than Jeong et al.'s approach and Gousios et al.'s approach. Since our approach deals with the issue of imbalanced data while Jeong et al.'s approach and Gousios et al.'s approach do not address the problem, our approach assigns more weight to abandoned code changes so that it achieves a much better recall for abandoned code changes than Jeong et al.'s approach and Gousios et al.'s approach. Thus, the little degradation of F1(M) is reasonable.

Notice that all ER@20% scores of our approach and baselines are above 0.8 and very high. As mentioned in Section 3.6, in addition to directly comparing the ER@20% scores of our approach and the baselines, we also use *normalized improvement* to evaluate the improvements of our approach over the baselines in terms of ER@20%.

We have the following findings:

- In terms of AUC, our approach on average improves random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.' approach by 46%, 46%, 30%, and 12%, respectively. And statistical tests show that the improvements are significant, and the effect sizes are large.
- In terms of ER@20%, our approach on average improves random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach by 13%, 13%, 10% and 3%. Statistical tests show that the improvements are significant, and the effect sizes are large. And the *normalized improvements* of our approach over random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach are 69%, 69%, 64%, and 38%, respectively.
- In terms of F1 for abandoned code changes, i.e., F1(A), our approach improves random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach by 54%, 131%, 236% and 131%, respectively. And statistical tests show that the improvements are significant, and the effect sizes are large.

Recall that the main purpose of our tool is to help code reviewers prioritize the code changes they are asked to review, and in such a case, AUC and ER@20% are the most important evaluation metrics since they both measure the probability that a merged code change is ranked higher than an abandoned code change. And in our study, we find our approach achieves substantial improvements over the baselines in terms of AUC and ER@20%. Also, our approach substantially improves the baseline methods in terms of F1(A).

(RQ2) How effective is our prediction model when all features are used than when only a subset is used?

In this study, we combine 34 features from five dimensions (i.e., code, file history, owner experience, collaboration network, and text). In this research question, we would like to investigate whether our prediction model benefits from the use of all features—as compared to its subset.

We build a random forest model based on features in each dimension, and in total, we have five random forest models—let us refer to them as code, file history, owner experience, collaboration network, and text models. For each prediction model, we also use cost sensitive learning to deal with imbalanced data. We then compute the average AUC, ER@20%, precision, recall and F1 scores for merged and abandoned code changes across the 10 folds. Table 6 presents the AUC, ER@20%, F1, precision and recall for merged and abandoned

Table 5 Adjusted p-values and Cliff's deltas for our approach compared with the baselines

Project	Approach	AUC			ER@20%			F1(A)		
		p-value	Cliff's delta		p-value	Cliff's delta		p-value	Cliff's delta	
			Value	Desc.		Value	Desc.		Value	Desc.
Eclipse	Random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Weighted random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Jeong et al. (2009)	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Gousios et al. (2014)	2.9×10^{-3}	0.96	Large	1.4×10^{-2}	0.98	Large	2.9×10^{-3}	1	Large
Libre.	Random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.68	Large
	Weighted random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Jeong et al. (2009)	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Gousios et al. (2014)	2.9×10^{-3}	0.72	Large	2.9×10^{-3}	0.85	Large	2.9×10^{-3}	0.96	Large
OpenS.	Random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Weighted random guess	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Jeong et al. (2009)	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Gousios et al. (2014)	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.92	Large	2.9×10^{-3}	1	Large

P-value is the probability of finding the observed results when null hypothesis (H_0) of a study question is true (DeGroot and Schervish 2012). Cliff's delta is a non-parametric effect size measure that quantifies the amount of difference between two sets of values (Cliff 2014). In our case, the p-values evaluate the probabilities that our approach improves the baselines in terms of AUC, ER@20% and F1(A) by chance. The Cliff's deltas evaluate the magnitude of differences between the effectiveness results of our approach and the baselines in terms of AUC, ER@20% and F1(A). In the table, the p-values which are less than 0.05 indicate that the improvements of our approach over the baselines are statistically significant and the Cliff's deltas which are larger than 0.474 indicate that the magnitudes of the differences are large

Table 6 AUC, ER@20%, F1, precision and recall for merged and abandoned code changes of our approach compared with the random forest models built using features in each dimension

Project	Dimension	AUC	ER@20%	Merged			Abandoned		
				F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	All dimensions	**0.71**	**0.93**	**0.87**	**0.87**	**0.88**	**0.39**	**0.40**	0.39
	Code	0.53	0.83	0.83	0.83	0.82	0.21	0.21	0.22
	File history	0.51	0.83	0.62	0.83	0.50	0.27	0.18	**0.52**
	Owner experience	0.66	0.91	0.84	**0.87**	0.81	0.36	0.32	0.41
	Collaboration network	0.57	0.86	0.79	0.85	0.74	0.29	0.23	0.37
	Text	0.53	0.83	0.66	0.84	0.55	0.28	0.19	0.51
Libre.	All dimensions	**0.73**	**0.96**	**0.92**	**0.91**	**0.93**	**0.31**	**0.34**	0.29
	Code	0.54	0.89	0.85	0.88	0.82	0.16	0.13	0.21
	File history	0.51	0.89	0.67	0.88	0.55	0.19	0.12	0.46
	Owner experience	0.65	0.93	0.88	**0.91**	0.86	0.30	0.26	0.36
	Collaboration network	0.59	0.92	0.87	0.89	0.85	0.22	0.19	0.26
	Text	0.54	0.90	0.66	0.90	0.53	0.22	0.14	**0.56**
OpenS.	All dimensions	**0.74**	**0.95**	**0.89**	**0.87**	**0.91**	**0.41**	**0.45**	0.37
	Code	0.57	0.86	0.83	0.84	0.81	0.26	0.24	0.29
	File history	0.57	0.86	0.76	0.85	0.68	0.28	0.22	0.41
	Owner experience	0.68	0.93	0.86	**0.87**	0.85	0.37	0.36	0.38
	Collaboration network	0.62	0.89	0.83	0.86	0.80	0.33	0.29	0.38
	Text	0.53	0.85	0.70	0.84	0.60	0.28	0.20	**0.47**

The best results are in bold

code changes of our approach compared with the five random forest models. Similar to RQ1, we also apply Wilcoxon signed-rank test with Bonferroni correction to investigate whether the improvements of our approach over the five random forest models are statistically significant (p-value < 0.05), and Cliff's delta to measure effect size. The statistical tests take as input the AUC, ER@20%, F1-scores for merged and abandoned code changes calculated in the 10 evaluation folds of our approach and the five baselines. Tables 7 and 8 present the adjusted p-values and Cliff's delta for our approach (using all features) compared with the five random forest models.

We have the following findings:

- In terms of AUC, our approach on average improves code, file history, owner experience, collaboration network, and text models by 33%, 38%, 11%, 24% and 38%, respectively. Statistical tests show that the improvements are significant, and all the effect sizes are large.
- In terms of ER@20%, our approach on average improves code, file history, owner experience, collaboration network, and text models by 10%, 10%, 3%, 7% and 10%, respectively. Statistical tests show that the improvements are significant, and all the effect sizes are large. And the *normalized improvements* of our approach over code, file history, owner experience, collaboration network, and text models are 64%, 64%, 38%, 55% and 64%, respectively.

Table 7 Adjusted p-values and Cliff's deltas for our approach compared with the random forest models built using features in each dimension

Project	Dimension	AUC			ER@20%		
		p-value	Cliff's delta		p-value	Cliff's delta	
			Value	Desc.		Value	Desc.
Eclipse	Code	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	File history	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Owner experience	2.9×10^{-3}	0.8	Large	8.8×10^{-3}	0.78	Large
	Collaboration network	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Text	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
Libre.	Code	2.9×10^{-3}	1	Large	8.8×10^{-3}	1	Large
	File history	2.9×10^{-3}	1	Large	8.8×10^{-3}	1	Large
	Owner experience	2.9×10^{-3}	0.74	Large	8.8×10^{-3}	1	Large
	Collaboration network	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Text	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
OpenS.	Code	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	File history	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Owner experience	2.9×10^{-3}	0.96	Large	2.9×10^{-3}	0.94	Large
	Collaboration network	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Text	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large

- In terms of F1 for abandoned code changes, i.e., F1(A), our approach on average improves code, file history, owner experience, collaboration network, and text models by 76%, 48%, 8%, 32% and 42%, respectively. And our statistical tests show that only one of the improvements is not significant—it is highlighted in bold in Table 7. Furthermore, in most of the cases, the effect size is large.
- In terms of F1 for merged code changes, i.e, F1(M), our approach on average improves code, file history, owner experience, collaboration network, and text models by 6%, 31%, 3%, 7% and 33%, respectively. And our statistical tests show that the improvements are significant, and all the effect sizes are large.

As mentioned in Section 3.6, AUC and EffectivenessRatio@20% are the most important metrics in our study. In terms of these two metrics, we find that our approach achieves substantial improvements over code, file history, owner experience, collaboration network and text models. Our approach also achieves improvements in terms of F1(A) and F1(M) over the five prediction models. This demonstrates that combining the five dimensions is beneficial in improving the effectiveness of our proposed solution.

(RQ3) How effective are our approach and the baseline methods when different percentages of code changes are inspected?

By default, we evaluate the performance of the prediction models when only the top 20% of the code changes are reviewed following previous studies (Rahman et al. 2012; Jiang et al. 2013a; Xia et al. 2015a). In this research question, we would like to investigate the performance of our approach and the baseline methods when different percentages of code

Table 8 Adjusted p-values and Cliff's delta for our approach compared with the random forest models built each of the five dimension

Project	Dimension	F1(M)			F1(A)		
		p-value	Cliff's delta		p-value	Cliff's delta	
			Value	Desc.		Value	Desc.
Eclipse	Code	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	File history	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Owner experience	2.9×10^{-3}	1	Large	5.9×10^{-3}	0.34	Medium
	Collaboration network	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.88	Large
	Text	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.92	Large
Libre.	Code	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.86	Large
	File history	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.7	Large
	Owner experience	5.9×10^{-3}	0.72	Large	1.0	0.04	Negligible
	Collaboration network	2.9×10^{-3}	0.9	Large	2.9×10^{-3}	0.52	Large
	Text	2.9×10^{-3}	1	Large	1.5×10^{-2}	0.5	Large
OpenS.	Code	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	File history	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large
	Owner experience	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.66	Large
	Collaboration network	2.9×10^{-3}	1	Large	2.9×10^{-3}	0.98	Large
	Text	2.9×10^{-3}	1	Large	2.9×10^{-3}	1	Large

changes are inspected. To answer this research question, we plot the ER@K% graphs that show the percentages of merged code changes that can be caught by inspecting different percentages of code changes.

Figures 3, 4 and 5 present the ER@K% scores of our approach and the baseline methods for various K on Eclipse, LibreOffice and OpenStack datasets, respectively. We notice that for each dataset, our approach consistently achieves the best EffectivenessRatio (ER) for a wide range of K values (i.e., a wide range of the number of code changes to review). For

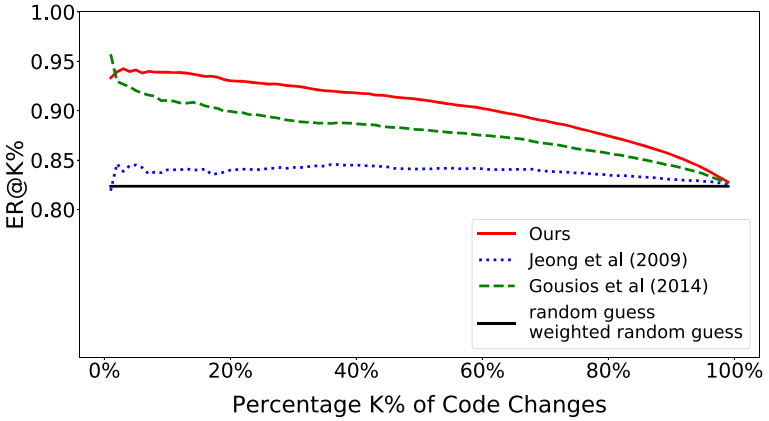


Fig. 3 EffectivenessRatio@K% (ER@K%) of our approach and baselines for various K on Eclipse dataset

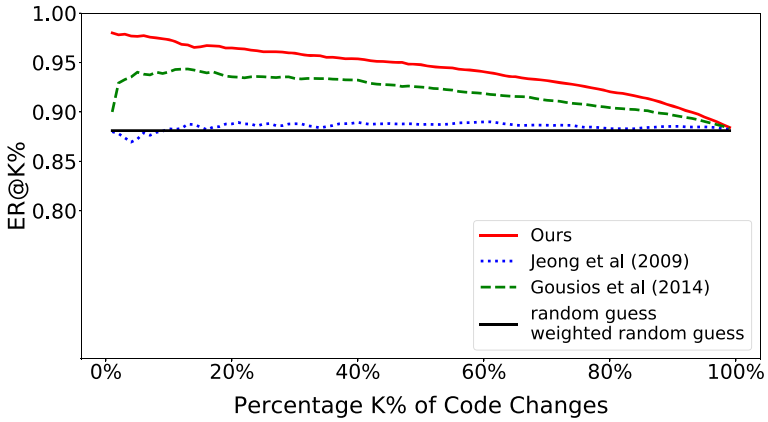


Fig. 4 EffectivenessRatio@K% (ER@K%) of our approach and baselines for various K on LibreOffice dataset

example, for Eclipse, when we set K as 25, the ER@25% scores of our approach, random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach are 0.93, 0.82, 0.82, 0.84 and 0.89, respectively; when we set K as 50, the ER@50% scores of our approach, random guess, weighted random guess, Jeong et al.'s approach and Gousios et al.'s approach are 0.91, 0.82, 0.82, 0.84 and 0.88, respectively.

We notice that for each dataset, the curve of ER@K% consistently descends as K increases. This is the case since the number of abandoned code changes in the top K% of the ranking list increases as K increases.

We also calculate the improvements and normalized improvements of our approach over the baselines in terms of ER@K% for various K on Eclipse, LibreOffice and OpenStack datasets. We vary K from 10 to 90. Tables 9, 10 and 11 present the improvements and normalized improvements of our approach over the baselines in terms of ER@K% for Eclipse, LibreOffice and OpenStack datasets, respectively. As shown in the tables, our approach outperforms the baselines in terms of ER@K% for various K across the three projects. We also

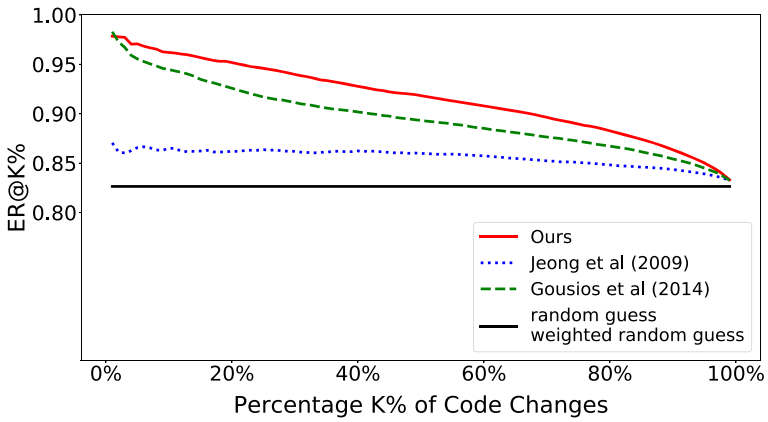


Fig. 5 EffectivenessRatio@K% (ER@K%) of our approach and baselines for various K on OpenStack dataset

Table 9 Improvements (IMPR) and Normalized Improvements (NIMPR) of our approach over the baselines in terms of ER@K% for various K on Eclipse dataset

K	Random guess Weighted random guess		Jeong et al. (2009)		Gousios et al. (2014)	
	IMPR	NIMPR	IMPR	NIMPR	IMPR	NIMPR
10	14%	65%	12%	62%	3%	32%
20	13%	61%	11%	56%	3%	31%
30	12%	58%	10%	52%	4%	32%
40	11%	54%	9%	47%	4%	28%
50	11%	50%	8%	44%	3%	25%
60	10%	45%	7%	38%	3%	22%
70	8%	38%	6%	32%	3%	17%
80	6%	29%	5%	24%	2%	12%
90	4%	18%	3%	15%	1%	7%

find that as K increases, the improvements and normalized improvements of our approach over the baselines decrease. However, across a wide range of K values (from 10 to 80), the normalized improvements of our approach over the baselines are more than 10% across the three projects.

In practice, reviewers cannot inspect all the code changes they are assigned due to various reasons such as tight development schedule. As shown in Figs. 3–5, reviewers can leverage the ranking list produced by our approach to prioritize the code changes. In such a way, they can focus more on the merged code changes, and do not waste time and effort to inspect the abandoned code changes.

(RQ4) Which features are most important in identifying merged code changes?

In this research question, we would like to find out the most important features that differentiate merged code changes from abandoned code changes in our Eclipse, LibreOffice and OpenStack datasets. Considering that merged code changes from different projects may have different characteristics, we conduct our experiment on each code change dataset.

Following previous studies (Tian et al. 2015; Bao et al. 2017), we consider to identify the most important features by leveraging random forest model. Comparing with the prediction model we implement in RQ1, we first perform feature selection to build another random forest model.

Step 1: *Correlation Analysis.* For each code change dataset, we first use variable clustering analysis implemented in the Hmisc R package[8] to look for the correlations among the features. By this analysis, we construct a hierarchical overview of the features among the 34 features and the correlated features are grouped into sub-hierarchies. We randomly select one feature from a sub-hierarchy if the correlations of features in the sub-hierarchy are larger than 0.7, which is the same setting used in the previous study (Tian et al. 2015). After this step, we remove 11, 10 and 9 features in the datasets of Eclipse, LibreOffice and OpenStack, respectively.

[8] https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf

Table 10 Improvements (IMPR) and Normalized Improvements (NIMPR) of our approach over the baselines in terms of ER@K% for various K on LibreOffice dataset

K	Random guess Weighted random guess		Jeong et al. (2009)		Gousios et al. (2014)	
	IMPR	NIMPR	IMPR	NIMPR	IMPR	NIMPR
10	10%	77%	10%	77%	3%	55%
20	9%	70%	9%	69%	3%	45%
30	9%	66%	8%	64%	3%	39%
40	8%	61%	7%	58%	2%	32%
50	8%	56%	7%	54%	2%	30%
60	7%	50%	6%	46%	2%	27%
70	6%	43%	5%	40%	2%	23%
80	4%	33%	4%	32%	2%	17%
90	3%	21%	2%	18%	1%	9%

Step 2: *Redundancy Analysis.* After reducing collinearity among the features using correlation analysis, we detect redundant features which do not have unique signal as compared to the other features. We use the redun function in the rms R package[9] to do the redundancy analysis and find that in the datasets of Eclipse, LibreOffice and OpenStack, none of the remaining features are redundant. So we do not remove any feature in this step.

Step 3: *Important Feature Identification.* After the above two steps, there are 23, 24 and 25 features remaining in Eclipse, LibreOffice and OpenStack, respectively. We implement a random forest model for each project using the bigrf R package.[10] To identify the importance of features that differentiate merged code changes from abandoned ones, we use the varimp function provided by bigrf to compute the importance of a feature in training process based on an internal error estimate of a random forest classifier which is called out of the bag (OOB) estimate (Wolpert and Macready 1999). Its core idea is to permute each feature randomly one by one and see whether the OOB estimate will be reduced significantly or not.

We use 10-fold cross validation and for each run of a 10-fold cross validation, we get an importance value for each feature. To determine which of the features are most important for the datasets, we input the importance values taken from all 10 runs to the Scott-Knott test (Scott and Knott 1974). We use the SK function provided by the ScottKnott R package.[11] This test takes a set of distributions (one for each feature) as input and identifies groups of features that are statistically significantly different from one another.

Figures 6, 7 and 8 present the Scott-Knott test results of Eclipse, LibreOffice and OpenStack when comparing the importance value of the features, respectively. In the figures, features are grouped by different colors and the importance values of different feature groups are significantly different from one another (p-value <

[9] https://cran.r-project.org/web/packages/rms/rms.pdf

[10] https://github.com/aloysius-lim/bigrf

[11] https://cran.r-project.org/web/packages/ScottKnott/ScottKnott.pdf

Table 11 Improvements (IMPR) and Normalized Improvements (NIMPR) of our approach over the baselines in terms of ER@K% for various K on OpenStack dataset

K	Random guess Weighted random guess		Jeong et al. (2009)		Gousios et al. (2014)	
	IMPR	NIMPR	IMPR	NIMPR	IMPR	NIMPR
10	16%	78%	11%	72%	2%	31%
20	15%	72%	10%	65%	3%	35%
30	14%	65%	9%	56%	3%	32%
40	12%	58%	8%	47%	3%	32%
50	11%	53%	7%	42%	3%	26%
60	10%	53%	7%	42%	3%	23%
70	8%	40%	5%	30%	2%	16%
80	7%	32%	4%	23%	2%	12%
90	5%	21%	2%	13%	1%	7%

0.05). From the figures, we find that *merged_ratio*, *msg_length* and *change_num* are the three most important features that affect the random forest models to differentiate merged code changes from abandoned code changes across Eclipse, LibreOffice and OpenStack. Note that Jeong et al.'s approach does not include any of the three most important features and Gousios et al.'s approach includes *merged_ratio* and *change_num*.

Step 4: *Effect of Important Features.* To understand the impact of these important features, we compare the values of top ten important features in merged and abandoned code changes in each code change dataset. We use the Wilcoxon rank-sum test (Mann and Whitney 1947) to analyze the statistical significance of the difference between merged and abandoned code changes. We compute the Cliff's delta introduced in RQ1 to investigate the effect size of the difference between the two groups of code changes. Table 12 shows the p-values and Cliff's delta for the top ten important features of Eclipse, LibreOffice and OpenStack.

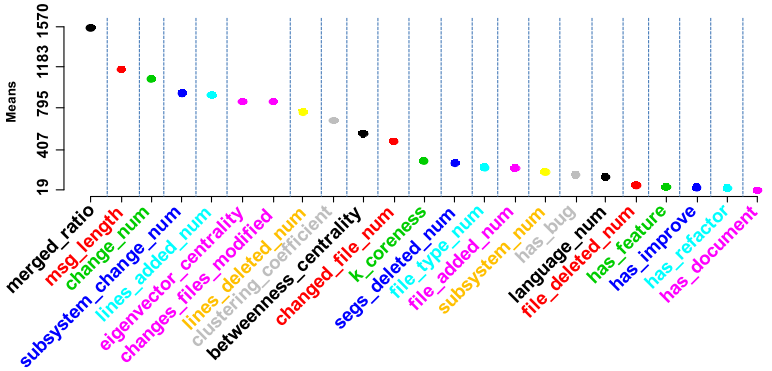


Fig. 6 Scott-Knott test results of Eclipse

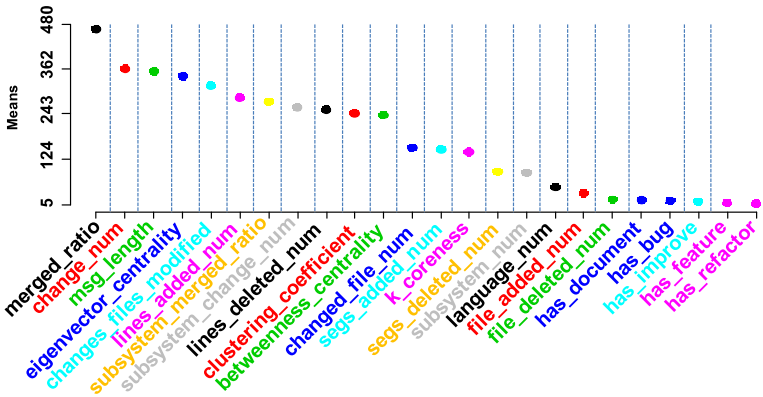


Fig. 7 Scott-Knott test results of LibreOffice

Based on the results shown in Figs. 6–8 and Table 12, we find that the features *merged_ratio*, *change_num*, *subsystem_change_num* and *clustering_coefficient* are the most important features to distinguish merged code changes from abandoned code changes and they have statistically significantly and non-negligible positive effect across three datasets. The results suggest that a code change owner's experience and his/her activity in the collaboration network are the most important factors that positively influence the likelihood of the code change to be merged.

In Table 12, we present the top ten important features for Eclipse, LibreOffice and OpenStack datasets. We would like to do more quantitative and qualitative analysis on these important features. For each dimension, we randomly choose one feature which is one of the top ten important features shown in Table 12. These features are *change_num*, *clustering_coefficient*, *lines_added_num*, *changes_files_modified* and *msg_length*.

For each feature, we analyze the class distribution of changes when the feature is of different values. To do this, for each feature, we calculate its median value for each project and use the median value to divide the dataset into two sets. The first set contains changes whose feature values are less than the median value and the second set contains the remaining changes. Then we calculate the class distribution of the two sets. We use Fisher's exact

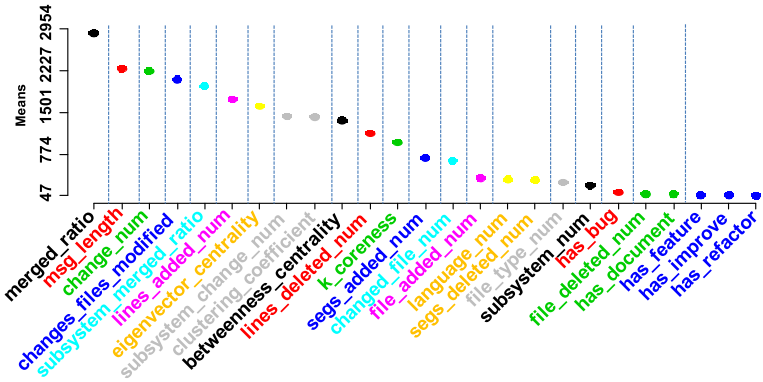


Fig. 8 Scott-Knott test results of OpenStack

Table 12 P-values and Cliff's deltas for the top ten most important features of Eclipse, LibreOffice and OpenStack

Project	Features	p-value	Cliff's delta	
			Value	Desc.
Eclipse	merged_ratio	2.2×10^{-16}	0.33	Medium
	msg_length	2.2×10^{-16}	0.13	Negligible
	change_num	2.2×10^{-16}	0.18	Small
	subsystem_change_num	2.2×10^{-16}	0.16	Small
	lines_added_num	2.2×10^{-16}	−0.08	Negligible
	eigenvector_centrality	2.2×10^{-16}	0.14	Negligible
	changes_files_modified	1.5×10^{-2}	0.02	Negligible
	lines_deleted_num	2.3×10^{-1}	−0.01	Negligible
	clustering_coefficient	2.2×10^{-16}	0.17	Small
	betweenness_centrality	7.0×10^{-14}	0.05	Negligible
Libre.	merged_ratio	2.2×10^{-16}	0.39	Medium
	change_num	2.2×10^{-16}	0.27	Small
	msg_length	2.2×10^{-16}	0.17	Small
	eigenvector_centrality	2.2×10^{-16}	0.29	Small
	changes_files_modified	1.7×10^{-1}	−0.02	Negligible
	lines_added_num	9.5×10^{-7}	−0.06	Negligible
	subsystem_merged_ratio	2.2×10^{-16}	0.32	Small
	subsystem_change_num	2.2×10^{-16}	0.21	Small
	lines_deleted_num	2.2×10^{-16}	−0.10	Negligible
	clustering_coefficient	2.2×10^{-16}	0.19	Small
OpenS.	merged_ratio	2.2×10^{-16}	0.38	Medium
	msg_length	2.3×10^{-1}	0.01	Negligible
	change_num	2.2×10^{-16}	0.25	Small
	changes_files_modified	2.2×10^{-16}	−0.06	Negligible
	subsystem_merged_ratio	2.2×10^{-16}	0.35	Medium
	lines_added_num	2.6×10^{-5}	−0.02	Negligible
	eigenvector_centrality	2.2×10^{-16}	0.23	Small
	subsystem_change_num	2.2×10^{-16}	0.21	Small
	clustering_coefficient	2.2×10^{-16}	0.20	Small
	betweenness_centrality	2.2×10^{-16}	0.20	Small

test (Upton 1992) to determine whether class distributions of the two sets are statistically significantly different (p-value < 0.05) or not. The test takes as input the number of merged and abandoned code changes in the two sets. Furthermore, we also qualitatively analyze some change examples manually.

In the owner experience dimension, we analyze the feature *change_num*. The feature refers to number of previous changes submitted by the owner of a change. For each project, we use the median value of *change_num* to divide the dataset into two sets. Table 13 presents

Table 13 Class distribution of changes when *change_num* is of different values and results of Fisher's test for Eclipse, LibreOffice and OpenStack datasets

Project	Feature value	% of Merged	% of Abandoned	p-value
Eclipse	<104	79%	21%	2.2×10^{-16}
	≥104	86%	14%	
Libre.	<87	84%	16%	2.2×10^{-16}
	≥87	91%	9%	
OpenS.	<134	78%	22%	2.2×10^{-16}
	≥134	87%	13%	

the class distributions of the two sets for each project and results of Fisher's test. We find that across the three projects, class distributions of the two sets of changes are statistically significantly different (p-value < 0.05). It indicates that *change_num* is an effective feature to distinguish merged and abandoned changes. Experienced developers' changes are more likely to be merged than non-experienced developers' changes. For example, the owner of change 73252[12] in Eclipse had submitted more than 1,000 changes previous to this change. The reviewers only gave some simple comments and after the owner addressed the comments, the change was eventually merged. As a contrasting example, in Eclipse, the owner of change 58215[13] had not submitted any change previous to this change. The change was eventually replaced by a better change and abandoned.

In the collaboration network dimension, we analyze the feature *clustering_coefficient*. The feature evaluates owner's degree of activity in the collaboration network. A larger *clustering_coefficient* indicates that the change owner has a higher centrality in the collaboration network and more collaboration activities. The value of *clustering_coefficient* ranges from 0 to 1. For each project, we use the median value of *clustering_coefficient* to divide the dataset into two sets. Table 14 presents the class distributions of the two sets for each project and results of Fisher's test. We find that across the three projects, class distributions of the two sets of changes are statistically significantly different (p-value < 0.05). It indicates that *clustering_coefficient* is also an effective feature to distinguish merged and abandoned changes. A change owner with more activities in the collaboration network is more likely to get his/her changes to be merged. For example, for change 242578[14] of OpenStack, the owner's node in our constructed network has a *clustering_coefficient* larger than 0.5. Notice that in Table 14, across the three projects, the median value of *clustering_coefficient* is less than 0.1. Thus, the *clustering_coefficient* of the owner's node is large, which indicates that the owner has many collaboration activities in the project. And the change was eventually merged.

In the code dimension, we analyze the feature *lines_added_num*. The feature refers to number of added lines in a change. For each project, we use the median value of *lines_added_num* to divide the dataset into two sets.. Table 15 presents the class distributions of the two sets for each project and results of Fisher's test. We find that across the three projects, class distributions of the two sets are statistically significantly different (p-value < 0.05). By comparing the results shown in Tables 13, 14 and 15, we find that

[12] https://git.eclipse.org/r/#/c/73252/

[13] https://git.eclipse.org/r/#/c/58215/

[14] https://review.openstack.org/#/c/242578/

Table 14 Class distribution of changes when *clustering_coefficient* is of different values and results of Fisher's test for Eclipse, LibreOffice and OpenStack datasets

Project	Feature value	% of Merged	% of Abandoned	p-value
Eclipse	<0.0756	79%	21%	2.2×10^{-16}
	≥0.0756	86%	14%	
Libre.	<0.0161	85%	15%	2.2×10^{-16}
	≥0.0161	91%	9%	
OpenS.	<0.0454	78%	22%	2.2×10^{-16}
	≥0.0454	87%	13%	

lines_added_num is not as effective as *change_num* and *clustering_coefficient*. But statistical tests indicate that *lines_added_num* is an effective feature to distinguish merged changes from abandoned ones. Small changes are more likely to be merged than large changes. For example, in Eclipse, the change 59345[15] was eventually abandoned since it was too large (the change contains 3,423 added lines and 699 deleted lines). The reviewers suggested the change owner to split the change into small changes. It indicates that reviewers prefer small changes.

In the file history dimension, we analyze the feature *changes_files_modified*. The feature is computed as the total number of times that the files in a change were modified previous to the change. For each project, we use the median value of *changes_files_modified* to divide the dataset into two sets. Table 16 presents the class distributions of the two sets of changes. We find that class distributions of the two sets are similar in LibreOffice, while in Eclipse and OpenStack, class distributions of the two sets are statistically significantly different (p-value < 0.05). From Table 16, we also find that the effect of *changes_files_modified* is different in Eclipse and OpenStack. In Eclipse, the set containing changes with smaller *changes_files_modified* has more abandoned changes than the other set, while in OpenStack, the set containing changes with smaller *changes_files_modified* has less abandoned changes than the other set. Our findings indicate that *changes_files_modified* may influence a change's likelihood to be merged in various ways.

On one hand, a change with larger *changes_files_modified* may contain files which are modified many times in previous changes. Files which are modified many times are more likely to be important files in the development process of the project. Since experienced developers are familiar with the development process of the project, they are more likely to modify these files. As mentioned earlier, these changes are more likely to be merged since they are submitted by experienced developers. For example, in Eclipse, files that were affected by change 7424[16] had been totally modified more than 600 times and the owner had submitted more than 700 changes before the change was submitted. And the change was eventually merged.

On the other hand, for the files that are modified many times, developers are more likely to modify them at the same time, which may cause merge conflict. And merge conflicts cause many changes to be abandoned. For example, files that were affected by change 172781[17] in OpenStack had been totally modified more than 1,000 times before the change

[15] https://git.eclipse.org/r/#/c/59345/

[16] https://git.eclipse.org/r/#/c/7424/

[17] https://review.openstack.org/#/c/172781/

Table 15 Class distribution of changes when *lines_added_num* is of different values and results of Fisher's test for Eclipse, LibreOffice and OpenStack datasets

Project	Feature value	% of Merged	% of Abandoned	p-value
Eclipse	<52	84%	16%	2.2×10^{-16}
	≥52	81%	19%	
Libre.	<22	89%	11%	1.4×10^{-5}
	≥22	87%	13%	
OpenS.	<18	83%	17%	2.9×10^{-2}
	≥18	82%	18%	

was submitted. And the change was determined as merge conflict and it was eventually abandoned. Another example is change 261950[18] in OpenStack.

In the text dimension, we analyze the feature *msg_length*. The feature refers to number of words in the description of a change. For each project, we use the median value of *msg_length* to divide the dataset into two sets. Table 17 presents the class distributions of the two sets of changes. From the table, we find that in OpenStack, the class distributions of the two sets are similar while in Eclipse and LibreOffice, the class distributions of the two sets are statistically significantly different (p-value < 0.05). It indicates that *msg_length* is an effective feature in Eclipse and LibreOffice but it is not effective in OpenStack. A larger *msg_length* indicates that the change contains longer description. In Eclipse and LibreOffice, changes with long description are more likely to be merged than those with short description. Descriptions that are too short may not provide sufficient information; as a result, reviewers may not be confident enough to merge the corresponding changes. For example, consider change 15274[19] of LibreOffice; the description only contains one sentence. Reviewers had to ask the owner a question for more information. The owner did not reply and the change was eventually abandoned due to the lack of information.

The results of the above analysis are consistent with our findings shown in Table 12. The analysis corroborates our conclusion—change owner's experience and collaboration activities in the collaboration network are the most important factors that can be used to identify merged code changes.

5 Discussion

5.1 Impact of Different Underlying Classifiers

In our study, we use random forest as the default underlying classifiers to evaluate the performance of our approach. Here, we would like to investigate the impact of different underlying classifiers on the performance of prediction. In addition to the random forest model, we also use other classifiers namely Naive Bayes, Decision tree and Bayesian network (Han et al. 2011). These classifiers are widely used in past software engineering studies (Kim et al. 2008; Ratzinger et al. 2007; Jiang et al. 2013a; Fenton et al. 2007). Table 18 presents

[18] https://review.openstack.org/#/c/261950/

[19] https://gerrit.libreoffice.org/#/c/15274/

Table 16 Class distribution of changes when *changes_files_modified* is of different values and results of Fisher's test for Eclipse, LibreOffice and OpenStack datasets

Project	Feature value	% of Merged	% of Abandoned	p-value
Eclipse	<9	82%	18%	1.0×10^{-2}
	≥9	83%	17%	
Libre.	<13	88%	12%	$\mathbf{8.4 \times 10^{-1}}$
	≥13	88%	12%	
OpenS.	<24	84%	16%	2.2×10^{-16}
	≥24	81%	19%	

the AUC, ER@20%, F1 for merged code changes (i.e., F1(M)) and F1 for abandoned code changes (i.e., F1(A)) for different underlying classifiers.

We notice that random forest model achieves the best performance in terms of AUC, ER@20%, F1(M) and F1(A) compared with the other three classifiers. Thus, in practice, we recommend developers to use random forest as the prediction model.

5.2 Cross-project Prediction

In our experiment setting in RQ1 and RQ2, for each project, we train a model by leveraging the historical code changes with known labels within the project. However, for a new project, it may not have sufficient data to build a prediction model. Here, we would like to investigate the performance of our approach in the cross-project setting. We build a model using data from one project (i.e., source project) and use it to predict merged code changes in the other projects (i.e., target projects).

In the cross-project setting, to apply Jeong et al.'s approach, we use keywords common in multiple programming languages—see Section 3.4. For the Eclipse dataset, in the within-project setting, we use all the keywords (including Java-specific ones) that Jeong et al. proposed in their original paper; however, for the cross-project setting, we can only use the common keywords.

Table 19 presents the AUC, ER@20%, F1 for merged code changes (i.e., F1(M)) and F1 for abandoned code changes (i.e., F1(A)) of our approach compared with the baselines. Our approach achieves AUC scores ranging from 0.70 to 0.72, and ER@20% scores ranging from 0.92 to 0.97. Compared with Jeong et al.'s approach and Gousios et al.'s approach, our approach achieves the best performance in terms of AUC, ER@20%, and F1(A). We

Table 17 Class distribution of changes when *msg_length* is of different values and results of Fisher's test for Eclipse, LibreOffice and OpenStack datasets

Project	Feature value	% of Merged	% of Abandoned	p-value
Eclipse	<26	80%	20%	2.2×10^{-16}
	≥26	85%	15%	
Libre.	<21	85%	15%	2.2×10^{-16}
	≥21	90%	10%	
OpenS.	<31	82%	18%	$\mathbf{9.4 \times 10^{-1}}$
	≥31	82%	18%	

Table 18 AUC, ER@20%, F1 for merged code changes, i.e., F1(M) and F1 for abandoned code changes, i.e., F1(A) of the prediction models

Project	Prediction models	AUC	ER@20%	F1(M)	F1(A)
Eclipse	Random forest	**0.71**	**0.93**	**0.87**	**0.39**
	Naive Bayes	0.63	0.88	0.84	0.30
	Decision tree	0.60	0.82	0.82	0.35
	Bayesian Network	0.64	0.90	0.76	0.35
Libre.	Random forest	**0.73**	**0.96**	**0.92**	**0.31**
	Naive Bayes	0.66	0.92	0.86	0.28
	Decision tree	0.58	0.86	0.88	0.27
	Bayesian Network	0.66	**0.96**	0.81	0.27
OpenS.	Random forest	**0.74**	**0.95**	**0.89**	**0.41**
	Naive Bayes	0.68	0.91	0.72	0.37
	Decision tree	0.63	0.85	0.83	0.37
	Bayesian Network	0.69	0.94	0.79	0.38

notice that our approach achieves a slightly lower F1(M) than Jeong et al.'s approach and Gousios et al.'s approach. Since Jeong et al.'s approach and Gousios et al.'s approach do not deal with imbalanced data while our approach addresses the problem, the two baselines

Table 19 The AUC, ER@20%, F1 for merged code changes, i.e., F1(M) and F1 for abandoned code changes, i.e., F1(A) of our approach compared with two baselines on cross-project prediction

Source & Target Project	Approach	AUC	ER@20%	F1(M)	F1(A)
Eclipse ⇒ LibreOffice	Ours	**0.72**	**0.96**	0.90	**0.37**
	Jeong et al. (2009)	0.54	0.89	**0.92**	0.10
	Gousios et al. (2014)	0.63	0.93	**0.92**	0.15
Eclipse ⇒ OpenStack	Ours	**0.71**	**0.93**	0.89	**0.39**
	Jeong et al. (2009)	0.55	0.85	**0.90**	0.12
	Gousios et al. (2014)	0.59	0.87	**0.90**	0.17
LibreOffice ⇒ Eclipse	Ours	**0.70**	**0.92**	0.87	**0.37**
	Jeong et al. (2009)	0.54	0.84	0.87	0.17
	Gousios et al. (2014)	0.62	0.89	**0.90**	0.11
LibreOffice ⇒ OpenStack	Ours	**0.70**	**0.92**	0.87	**0.38**
	Jeong et al. (2009)	0.54	0.83	0.89	0.13
	Gousios et al. (2014)	0.64	0.90	**0.90**	0.16
OpenStack ⇒ Eclipse	Ours	**0.71**	**0.93**	0.89	**0.34**
	Jeong et al. (2009)	0.55	0.84	**0.90**	0.03
	Gousios et al. (2014)	0.64	0.90	**0.90**	0.13
OpenStack ⇒ LibreOffice	Ours	**0.71**	**0.97**	0.90	**0.34**
	Jeong et al. (2009)	0.55	0.88	**0.91**	0.03
	Gousios et al. (2014)	0.64	0.93	**0.91**	0.13

The best results are in bold

have a strong bias towards the majority class (i.e., merged code changes) due to imbalanced data. This results in that the two baselines achieve a slightly better F1(M) than our approach while their F1(A) scores are much lower than F1(A) scores of our approach.

From Table 19, we notice that the AUC results of all the cross-project prediction models are above 0.7. As mentioned in Secion 3.6, a prediction model with an AUC score above 0.7 is often considered to have adequate classification performance (Romano and Pinzger 2011). Thus, the cross-project prediction models using our proposed features can be considered to have adequate prediction performance.

5.3 Sensitivity of Our Approach to the Choice of α

As mentioned in Section 3.5, the choice of the parameter in cost sensitive learning (i.e., α) can influence the prediction performance of our approach. In this section, we would like to investigate the effect of varying α on the performance of our approach. We vary α from 0.2 to 2.

For the three projects, we plot the resultant AUC, ER@20%, F1-scores for merged and abandoned code changes in Figs. 9, 10, 11 and 12, respectively. As shown Figs. 9 and 10, AUC and ER@20% are relatively stable with various α. For example, consider the results for the Eclipse dataset; the average AUC scores only vary between 0.706 ($\alpha = 0.4$) and 0.717 ($\alpha = 1.6$) and the average ER@20% scores only vary between 0.928 ($\alpha = 0.4$) and 0.938 ($\alpha = 1.8$). Thus, AUC and ER@20% are not sensitive towards the choice of α. In Fig. 11, F1(M) decreases as α increases across the three projects. Moreover, the curves become steeper as α increases. When α is less than 0.8, F1(M) decreases slightly as α increases. When α is larger than 0.8, F1(M) decreases more visibly as α increases. It indicates that increasing α negatively impacts prediction performance of our approach for merged code changes. In Fig. 12, across the three projects, F1(A) increases when α is less than 1.2 but slightly changes when α is more than 1.2.

In summary, in terms of AUC and ER@20%, the prediction performance of our approach is relatively stable when varying α. However, the choice of α can impact the prediction performance of our approach in terms of F1-scores for merged and abandoned code changes. In practice, we recommend developers to choose an α between 0.8 and 1.2. And to choose an appropriate α, developers need to make a trade-off between merged and abandoned code

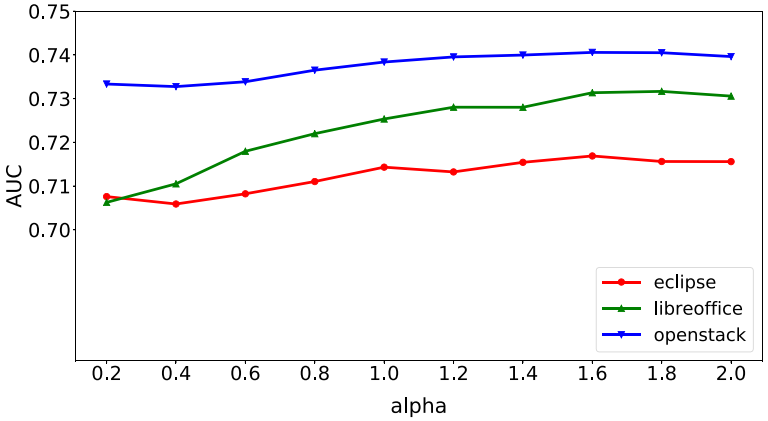


Fig. 9 Sensitivity of AUC of our approach to the choice of α for Eclipse, LibreOffice and OpenStack datasets

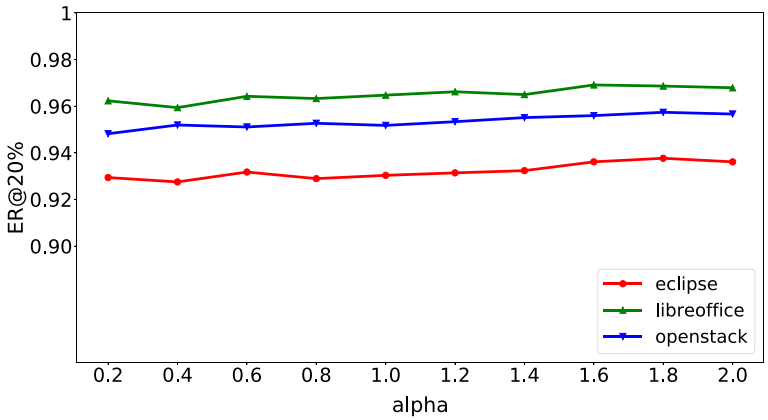


Fig. 10 Sensitivity of ER@20% of our approach to the choice of α for Eclipse, LibreOffice and OpenStack datasets

changes according to their practical requirements. If they hope to avoid time wasted on reviewing abandoned code changes, they should set a higher α so that our approach can achieve a better F1-score for the abandoned changes. However, developers may also hope to avoid wasting change owners' effort since it may slower the developing process of the project. In this case, they will put more weights on the merged changes, and set a lower α so that our approach can achieve a better F1-score for the merged changes.

5.4 Time Efficiency

The time efficiency of our approach may influence its usability. Thus, in this section, we investigate whether the runtime of our approach is reasonable. We calculate the model building time and prediction time in each evaluation fold of our approach compared with the baselines for Eclipse, LibreOffice and OpenStack datasets. Model building time refers to

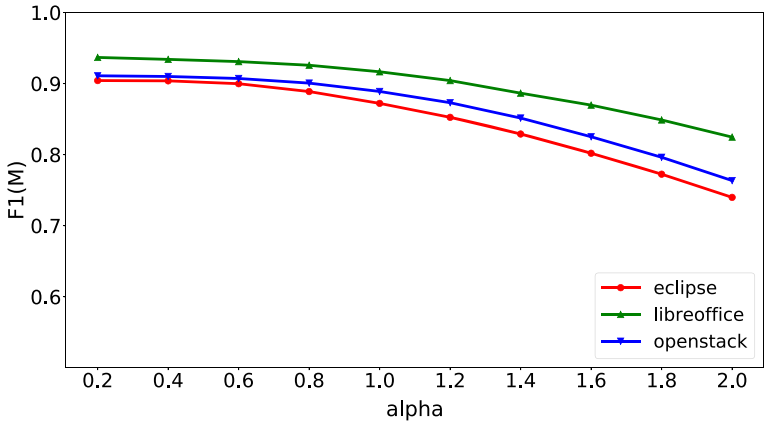


Fig. 11 Sensitivity of F1-score for merged code changes, i.e., F1(M), of our approach to the choice of α for Eclipse, LibreOffice and OpenStack datasets

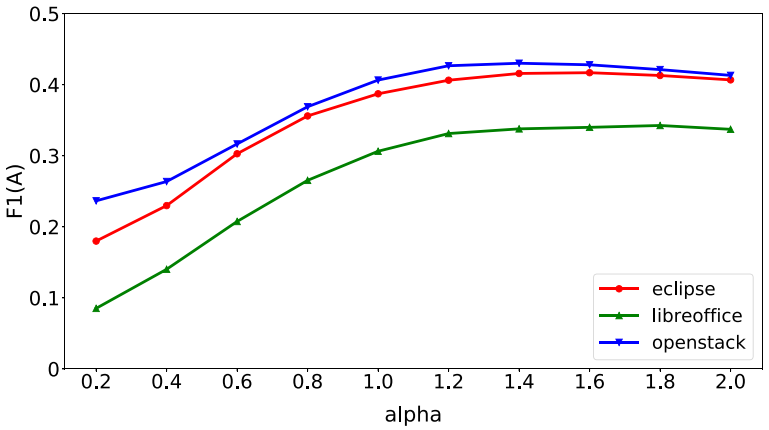


Fig. 12 Sensitivity of F1-score for abandoned code changes, i.e., F1(A), of our approach to the choice of α for Eclipse, LibreOffice and OpenStack datasets

the time that is needed by a classifier to build a model from a training dataset. And prediction time refers to the time that the classifier needs to predict the labels of code changes in a testing dataset. We use a Windows 7, 64-bit, Intel(R) Core i5 3.2GHz machine with 8GB RAM.

Tables 20, 21 and 22 present the model building time and prediction time in each evaluation fold of our approach compared with the baselines for Eclipse, LibreOffice and OpenStack datasets, respectively. We notice that our approach, Jeong et al.'s approach and Gousios et al.'s approach can quickly predict the labels of code changes in a testing dataset (the prediction time is less than 1 s). On average across the 10 folds, our approach takes 218, 78 and 551 s to build models in the experiment on the Eclipse, LibreOffice and OpenStack datasets, respectively. The model building time of our approach is relatively slower

Table 20 Model building time and prediction time in each fold of our approach compared with the baselines in the evaluation experiment on Eclipse dataset (in seconds)

Fold	Model building time			Prediction time		
	Ours	Jeong et al. (2009)	Gousios et al. (2014)	Ours	Jeong et al. (2009)	Gousios et al. (2014)
0	25.82	0.03	2.01	0.11	0.03	0.14
1	57.79	0.05	4.59	0.14	0.02	0.17
2	97.89	0.13	7.57	0.17	0.02	0.22
3	139.32	0.14	10.09	0.17	0.02	0.23
4	181.80	0.25	13.11	0.19	0.03	0.22
5	230.05	0.28	16.57	0.20	0.02	0.23
6	278.74	0.45	19.64	0.22	0.02	0.23
7	332.28	0.57	23.31	0.22	0.02	0.22
8	388.42	0.78	26.69	0.23	0.03	0.25
9	450.81	0.88	30.84	0.23	0.02	0.27
avg.	218.29	0.36	15.44	0.19	0.02	0.22

Table 21 Model building time and prediction time in each fold of our approach compared with the baselines in the evaluation experiment on LibreOffice dataset (in seconds)

Fold	Model building time			Prediction time		
	Ours	Jeong et al. (2009)	Gousios et al. (2014)	Ours	Jeong et al. (2009)	Gousios et al. (2014)
0	9.14	0.05	0.86	0.05	0.02	0.05
1	20.72	0.05	1.53	0.03	0.02	0.05
2	34.77	0.03	2.65	0.05	0.02	0.07
3	50.40	0.03	3.77	0.05	0.00	0.05
4	67.21	0.03	4.95	0.06	0.00	0.06
5	84.10	0.03	6.23	0.06	0.02	0.06
6	100.20	0.05	7.27	0.06	0.02	0.06
7	117.59	0.06	8.41	0.08	0.00	0.08
8	137.62	0.06	9.64	0.06	0.02	0.08
9	160.03	0.06	11.12	0.08	0.00	0.08
avg.	78.18	0.05	5.64	0.06	0.01	0.06

than Jeong et al.'s approach and Gousios et al.'s approach. However, we believe that the model building time is reasonable and our approach can be used in practice, since our model does not need to be updated all the time. We also notice that the model building time of our approach does not grow vigorously as training data size grows. For example, in the evaluation experiment on OpenStack dataset, the training dataset in fold 0 and fold 9 contains 8,814 and 88,140 code changes, respectively, and in fold 0 and fold 9, our approach takes 62 s and 1,144 s, respectively. In practice, developers can empirically limit the training data size to control the model building time of our approach without reducing the prediction performance.

Table 22 Model building time and prediction time in each fold of our approach compared with the baselines in the evaluation experiment on OpenStack dataset (in seconds)

Fold	Model building time			Prediction time		
	Ours	Jeong et al. (2009)	Gousios et al. (2014)	Ours	Jeong et al. (2009)	Gousios et al. (2014)
0	62.23	0.02	4.76	0.31	0.02	0.33
1	145.03	0.05	10.73	0.44	0.02	0.41
2	239.21	0.08	17.21	0.44	0.03	0.48
3	346.68	0.14	24.65	0.47	0.02	0.55
4	456.43	0.23	31.79	0.52	0.02	0.56
5	578.48	0.30	41.09	0.53	0.02	0.59
6	708.23	0.38	49.97	0.59	0.03	0.66
7	840.56	0.45	59.55	0.58	0.02	0.66
8	989.96	0.50	69.34	0.61	0.02	0.69
9	1,144.03	0.61	81.57	0.62	0.03	0.70
avg.	551.08	0.28	39.07	0.51	0.02	0.56

5.5 Bias Against New Contributors

Our findings in RQ4 reveal that owner's experience and his/her collaboration activities are the most important factors which impact the likelihood of a code change to be merged. It indicates that our approach may have a bias against new contributors, i.e., our approach inclines to predict new contributors' code changes as abandoned. In this section, we first investigate whether our approach has such a shortcoming. Then, we investigate how to overcome this possible shortcoming of our approach.

To investigate whether our approach has a bias against new contributors, we need to evaluate the performance of our approach on new contributors' changes. We consider a change owner as a new contributor if number of his/her previous changes is less than ten. And we call a change owner as an experienced contributor if he/she is not a new contributor. We cannot simply count contributors who have submitted one change as experienced contributors. Ten is a more reasonable threshold—as compared to one.

We first exclude experienced contributors' changes in testing dataset of each fold in RQ1. In such a case, we only have new contributors' changes in testing dataset in each fold. Then, for each fold, we train a model based on the same training dataset in RQ1 and evaluate the performance on the testing dataset. We calculate the average AUC, ER@20%, precision, recall, F1-scores for merged and abandoned code changes across the 10 folds.

Table 23 presents the AUC, ER@20%, F1-scores, precision and recall for new contributors' merged and abandoned changes of our approach. With respect to new contributors' merged changes, our approach achieves a recall score of 0.64, 0.62 and 0.63 on the Eclipse, LibreOffice and OpenStack datasets, respectively. As shown in Table 4, for *all* merged changes (submitted by new and experienced contributors), our approach achieves a recall of 0.88, 0.93 and 0.91 on the Eclipse, LibreOffice and OpenStack datasets, respectively. Thus, our approach achieves a much lower recall score for new contributors' merged changes—as compared to *all* merged changes. It means that our approach inclines to incorrectly predict the labels of new contributors' merged changes. Such new contributors' merged changes would be given a low priority to be reviewed and it may discourage new contributors to submit their code. Thus, we need to overcome this shortcoming.

To deal with the bias of our approach against new contributors, a specialized model for new contributors' changes is needed. We consider two methods to build the model:

Method 1. From the above analysis, the features in the owner experience and collaboration network dimensions cause the bias of our approach against new contributors.

Table 23 AUC, ER@20%, F1-score, precision, recall for new contributors' merged and abandoned changes of our approach

Project	AUC	ER@20%	Merged			Abandoned		
			F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	0.69	0.87	0.72	0.83	0.64	0.49	0.40	0.64
Libre.	0.67	0.88	0.71	0.84	0.62	0.46	0.36	0.63
OpenS.	0.74	0.85	0.70	0.81	0.63	0.58	0.50	0.71

Testing datasets only include new contributors' change

To deal with the bias, we can remove these features and train a model based on remaining features (i.e., features in the code, file history and text dimensions). To do this, we first remove the features in the owner experience and collaboration network dimensions from training and testing dataset of each fold and we build prediction model on the training dataset. Then, we use the model to predict the labels of new contributors' changes in testing dataset. Finally, we calculate the average AUC, ER@20%, F1-scores, precision and recall for merged and abandoned changes across the 10 folds. Table 24 presents the AUC, ER@20%, F1-scores, precision and recall of the model for new contributors' merged and abandoned changes. By comparing the results shown in Tables 23 and 24, this model achieves a much lower AUC and ER@20% for the Eclipse and LibreOffice datasets. This method thus cannot achieve acceptable performance for new contributors' changes in the two datasets.

Method 2. We notice that only 11%–15% of changes belong to new contributors in our datasets. Thus, our model (trained using all changes) is mainly built based on experienced contributors' changes. We conjecture that it induces the bias of our approach against new contributors. To verify our conjecture, we exclude experienced contributors' changes in training dataset of each fold and only use new contributors' changes to build prediction model. Then, we use this model to predict the labels of new contributors' changes in testing dataset. Finally, we calculate the average AUC, ER@20%, F1-scores, precision and recall for merged and abandoned changes across the 10 folds. Table 25 presents the AUC, ER@20%, F1-scores, precision and recall for new contributors' merged and abandoned changes of the model. As shown in Table 25, with respect to new contributors' merged changes, the model achieves a recall score of 0.86, 0.86 and 0.88 on the Eclipse, LibreOffice and OpenStack datasets, respectively. Thus, only using new contributors' changes to learn a prediction model can improve the recall score for new contributors' changes—as compared to the results shown in Table 23. Moreover, we notice that AUC scores of the model are around 0.7. It indicates that the model achieves an acceptable performance when predicting the labels of new contributors' changes.

By comparing the results of the two methods, we choose method 2 to build a specialized model for new contributors' changes, i.e., we build a model using only new contributors' changes.

Table 24 AUC, ER@20%, F1-score, precision, recall for new contributors' merged and abandoned changes of the model built on the features in code, file history and text dimensions

Project	AUC	ER@20%	Merged			Abandoned		
			F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	0.56	0.75	0.81	0.75	0.89	0.24	0.40	0.18
Libre.	0.55	0.77	0.81	0.76	0.87	0.24	0.35	0.19
OpenS.	0.67	0.78	0.79	0.73	0.86	0.45	0.56	0.37

Testing datasets only include new contributors' changes. (Method 1)

Table 25 AUC, ER@20%, F1-score, precision, recall for new contributors' merged and abandoned changes of the model built using new contributors' changes

Project	AUC	ER@20%	Merged			Abandoned		
			F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	0.68	0.85	0.82	0.79	0.86	0.43	0.50	0.38
Libre.	0.68	0.87	0.83	0.80	0.86	0.40	0.45	0.36
OpenS.	0.73	0.84	0.81	0.75	0.88	0.52	0.65	0.43

Testing datasets only include new contributors' changes. (Method 2)

Adjusted Approach To deal with the bias of our approach against new contributors while still creating an approach that is effective for experienced contributors, we incorporate the prediction results of our original model (trained using all changes in training dataset) and the model built using only new contributors' changes in training dataset.

In our testing data, for new contributors' changes, we use the model built using new contributors' changes. For experienced contributors' changes, we use the model built using all changes. When applied to a change, both models output a likelihood score for the change to be merged. We use these likelihood scores to predict the labels of changes in testing dataset and rank the changes.

Since the likelihood scores output by both models range from 0 to 1 and both models use 0.5 as the threshold to determine the labels of changes in testing dataset, our method of incorporating the results of the two models is reasonable.

We find that this adjusted approach can achieve a higher recall for new contributors' merged changes than our original model (built using all changes). We also calculate the average AUC, ER@20%, F1-score, precision and recall for *all* merged and abandoned changes (made by both new and experienced contributors) of the adjusted approach across the 10 folds. Table 26 presents the AUC, ER@20%, F1-score, precision and recall for merged and abandoned changes of the adjusted approach. By comparing the results shown in Tables 4 and 26, we find that in terms of AUC and ER@20%, incorporating results of the model built using new contributors' changes does not negatively impact prediction performance of our approach. Thus, incorporating results of the model built using new contributors' changes

Table 26 AUC, ER@20%, F1-score, precision, recall for merged and abandoned changes of the adjusted approach

Project	AUC	ER@20%	Merged			Abandoned		
			F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
Eclipse	0.71	0.93	0.88	0.86	0.90	0.36	0.42	0.32
Libre.	0.72	0.97	0.92	0.90	0.95	0.27	0.37	0.22
OpenS.	0.73	0.95	0.90	0.87	0.93	0.38	0.48	0.31

Testing datasets include both new and experienced contributors' changes

can help our approach overcome the bias against new contributors while still achieving good *overall* prediction performance.

5.6 Model Built Using Top Ten features

In RQ4, we study the most important features in identifying merged code changes and Table 12 presents the top ten important features that distinguish merged code changes from abandoned ones for Eclipse, LibreOffice and OpenStack datasets. In this section, we would like to investigate whether we can build a light-weight prediction model using only the top ten features.

For each project, we build a random forest model using the top ten features shown in Table 12. And we use cost sensitive learning to deal with imbalanced data. We then compute the average AUC, ER@20%, precision, recall and F1-scores for merged and abandoned code changes across the 10 folds. Table 27 presents the AUC, ER20%, F1-scores, precision and recall for merged and abandoned code changes of the model built using top ten features. On average, across the three projects, the model achieves an AUC, ER@20%, F1(M) and F1(A) of 0.70, 0.94, 0.88 and 0.35, respectively. We also record the building time of the model and calculate the average building time across the 10 folds (using the same machine described in Section 5.4). On average, across the 10 folds, the model takes 186, 66 and 453 s in the building phase for Eclipse, LibreOffice and OpenStack datasets, respectively.

By comparing the results shown in Tables 4 and 27, we find that in terms of AUC, ER@20%, F1(M) and F1(A), the model built using top ten features consistently achieves slightly lower performance than our approach (using all features). And in terms of AUC, ER@20% and F1(A), the model achieves better performance than the baselines. However, the model still needs a relatively long time in building phase. For example, to build a model based on 88,140 code changes in OpenStack (in fold 10), our approach (using all features) needs 1,144 s (i.e., 19min) and the model built using top ten features needs 926 s (i.e., 15 min). Thus, in practice, we recommend that developers use all our features to build prediction model so that the model can achieve the best prediction performance.

5.7 Implications for Researchers

In this paper, we propose an approach to predict early on whether a code change will be merged. Our approach outperforms the baselines in terms of AUC and ER@20%. However, on average across Eclipse, LibreOffice and OpenStack, our approach can only achieve an average recall for abandoned changes of 0.35. Thus, many abandoned changes could not be identified by our approach and there is a room for improvement.

A further study is needed to explain more deeply why developers abandon code changes. Researchers can design more domain-specific features inspired by additional insights on

Table 27 AUC, ER@20%, F1-scores for merged, i.e., F1(M) and F1-scores for abandoned code changes, i.e., F1(A) of the model built using top ten features in Table 12

Project	AUC	ER@20%	F1(M)	F1(A)
Eclipse	0.68	0.92	0.86	0.36
Libre.	0.70	0.96	0.91	0.30
OpenS.	0.72	0.94	0.88	0.39

abandoned changes. For example, we find that many code changes are eventually abandoned since they cause merge conflicts (e.g., changes 172781[20] and 261950[21] in OpenStack). A technique that can correctly detect merge conflicts would help our approach a lot with respect to identifying abandoned changes. Recently, researchers have conducted many studies on merge conflicts. For instance, Leßenich et al. (2016) performed a survey and empirical study on merge conflicts. They surveyed 41 developers and based on the results of the survey, they inferred seven potential indicators that could identify merge conflicts. However, they found that the seven indicators are not fully effective to predict merge conflicts. Follow-up studies are needed to investigate more potential indicators to determine merge conflicts.

Moreover, in this study, we find that although features in the owner experience and collaboration network dimensions are the most important in identifying merged changes, the features induce a bias against new contributors into our approach. Without addressing the bias, our approach may not be acceptable for developers. In Section 5.5, we propose a method to deal with the bias. In the future, we recommend that researchers should not only focus on how to achieve good results based on their proposed approaches, but also consider the practical usage of the techniques and whether the techniques have potential shortcomings in practice.

5.8 Threats to Validity

Threats to internal validity

refer to errors in our datasets and experiments. We have double-checked our datasets and experiments. However, there could still be errors that we did not notice. To address a common bias in training dataset selection, i.e., future data is used as training data to predict the past, we use a longitudinal data setup in our experiment. This strategy has been used in many previous studies to address the same bias (Bhattacharya and Neamtiu 2010; Tamrawi et al. 2011; Jeong et al. 2009; Xia et al. 2017).

Moreover, we adapt Jeong et al.'s approach (which is specific to Java) to be able to handle projects written in other programming languages (i.e., LibreOffice which is written in C++ and OpenStack which is written in Python). In particular, we omit a number of keyword occurrence features that are specific to Java (e.g., number of occurrences of keyword "interface"). We acknowledge that our adaptation strategy may not be an optimal one.

Threats to external validity refer to the generalizability of our prediction tool. We have analyzed 166,215 code changes from three open source projects written in multiple programming languages. Thus, we believe that our prediction tool is general for multiple programming languages. And in the above discussion, we present that our approach is more accurate than two baselines to do cross-project prediction.

Threats to construct validity refer to the suitability of our evaluation metrics. In this study, we mainly focus on the AUC scores and cost effectiveness of the prediction models. AUC is a widely used evaluation metric that are often used in past software engineering studies (Lamkanfi et al. 2010; Lessmann et al. 2008; Romano and Pinzger 2011). Cost effectiveness is also a widely used measure to quantify prediction performance in previous software engineering studies (Arisholm et al. 2007; Mende and Koschke 2009; Rahman et al. 2012; Jiang

[20] https://review.openstack.org/#/c/172781/

[21] https://review.openstack.org/#/c/261950/

et al. 2013a; Xia et al. 2015a). Therefore, we believe that there is little threat to construct validity in our study.

6 Related Work

Studies on prediction of merged code changes. To our best knowledge, Jeong et al. (2009) and Gousios et al. (2014) are the most related work to our paper on prediction of merged code changes.

Jeong et al. proposed a set features to predict whether a given bug-fix patch in two open source project (i.e., Firefox and Mozilla Core) will be accepted (Jeong et al. 2009). Their features included number of occurrences of certain keywords in the patch and features extracted from bug reports. Thus, they focused on predicting the acceptance of bug-fix patches written in a specific programming language, while our prediction tool is to predict whether a code change submitted to Gerrit will eventually get merged or not. Different from Jeong et al., our features are programming language agnostic. Experiment results show that our approach achieves statistically significantly better performance than Jeong et al.'s approach in terms of AUC score, EffectivenessRatio@20% and F1 score for abandoned code changes.

Gousios et al. proposed 12 features to predict whether a pull request will be merged and these features are grouped into three dimensions: pull request, project and developer (Gousios et al. 2014). Gousios et al.'s study focused on pull-requests while ours focuses on Gerrit code changes. Our work considers new features not considered by Gousios et al. We not only extract features from patch, but also from modification history, collaboration network and description of code changes. We combine features from 5 dimensions (i.e., code, file history, owner experience, collaboration network and text), and our approach achieves statistically significantly better performance than Gousios et al.'s approach in terms of AUC score, EffectivenessRatio@20% and F1 score for abandoned code changes.

Studies on factors impacting code review. Many studies have been conducted to investigate factors impacting code review in open source and proprietary software projects.

Rigby et al. performed an empirical study on code review process in four open source projects, namely GCC, Linux, Mozilla, and Apache (Rigby and German 2006). They described a number of review patterns, and quantitatively analyzed the code review data extracted from the Apache project's developer and commit mailing lists. Later, Rigby et al. examined the two peer review techniques used by Apache server project, and investigated the review process, frequency of reviews, level of participation, review interval and review inducing defect in projects (Rigby et al. 2008).

Weißgerber et al. analyzed email archives and code repositories of two small open source projects, namely FLAC and OpenAFS, and found that code changes with small size have more chances to be accepted (Weißgerber et al. 2008). Jiang et al. studied large-scale patch reviews on Linux to understand factors impacting patch acceptance and reviewing time in Linux kernel project (Jiang et al. 2013b). Baysal et al. studied the influence of non-technical factors (e.g., reviewer load and activity, patch writer experience) on the outcome of code reviews in the WebKit project, and found that non-technical factors including organizational and personal factors have influence both on review time and likelihood of a patch being accepted (Baysal et al. 2013). However, they did not propose an automated technique to predict patch acceptance.

McIntosh et al. studied the impact of code review coverage and participation on software quality through a case study on the Qt, VTK, and ITK projects (McIntosh et al. 2014).

Kononenko et al. reported an empirical study on code review quality for Mozilla, and found that personal and social metrics would impact code review quality (i.e., whether reviewed changes introduced bugs in the code) (Kononenko et al. 2015). Thongtanunam et al. analyzed 196,712 reviews across the Android, Qt, and OpenStack open source projects, and found that amount of review participation in the past, the description length of a patch, and purpose of introducing new features can impact review participation and code review quality (Thongtanunam et al. 2016).

Bacchelli et al. performed a study on expectations, outcomes and challenges of modern code reviews. They observed, interviewed and surveyed developers and managers, and manually categorized review comments from multiple teams in Microsoft (Bacchelli and Bird 2013). Gousios et al. investigated work practice and challenges in pull-based development model from integrator perspective by surveying 742 integrators in Github (Gousios et al. 2015). They provided insights of the factors impacting integrators' decision on accepting or rejecting a contribution. They found that integrators struggle to maintain the quality of their projects and have difficulties with prioritizing contributions that are to be merged.

7 Conclusion

In this paper, we propose an approach to predict whether a code change will be merged. We extract 34 features to characterize a code change; these features are grouped into five dimensions: code, file history, owner experience, collaboration network and text. We use random forest to build a prediction model and use cost sensitive learning to deal with imbalanced data. To investigate the effectiveness of our approach, we perform a large-scale experiment on code change dataset from three large open source projects containing a total of 166,215 code changes. The experimental results show that our approach can achieve an average AUC of 0.73 across the three projects, which improves random guess, Jeong et al.'s approach (Jeong et al. 2009), and Gousios et al.'s approach (Gousios et al. 2014) by 46%, 30%, and 12%, respectively. Moreover, our approach achieves an average ER@20% of 0.95; the *normalized improvements* of our approach over random guess, Jeong et al.'s approach and Gousios et al.'s approach are 69%, 64% and and 38%, respectively. Our approach achieves statistically significantly improvements over random guess, Jeong et al.'s approach, and Gousios et al.'s approach in terms of AUC and ER@20%, with large effect sizes. We also find that among the 34 features, *merged_ratio*, *change_num*, *subsystem_change_num*, *clustering_coefficient* are the most important ones to distinguish merged from abandoned code changes across the three projects.

In the future, we plan to evaluate our approach with more code changes from more software projects, and we also plan to design a better approach to improve the AUC and EffectivenessRatio further.

Acknowledgments This work was partially supported by NSFC Program (No. 61602403 and 61572426).

References

Abdi H (2007) Bonferroni and Šidák corrections for multiple comparisons. Encycl Meas Stat 3:103–107

Ackerman AF, Fowler PJ, Ebenau RG (1984) Software inspections and the industrial production of software. In: Proceedings of a symposium on software validation: inspection-testing-verification-alternatives. Elsevier North-Holland Inc., pp 13–40

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: The 18th IEEE international symposium on software reliability, 2007. ISSRE'07. IEEE, pp 215–224

Aurum A, Petersson H, Wohlin C (2002) State-of-the-art: software inspections after 25 years. Software Testing. Verif and Reliab 12(3):133–154

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 712–721

Bao L, Xing Z, Xia X, Lo D, Li S (2017) Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR). IEEE, pp 170–181

Baysal O, Kononenko O, Holmes R, Godfrey MW (2013) The influence of non-technical factors on code review. In: 2013 20th working conference on reverse engineering (WCRE). IEEE, pp 122–131

Bhattacharya P, Neamtiu I (2010) Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: 2010 IEEE international conference on software maintenance (ICSM). IEEE, pp 1–10

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Cliff N (2014) Ordinal methods for behavioral data analysis. Psychology Press, New York

Costa C, Figueiredo J, Sarma A, Murta L (2016) TIPMerge: recommending developers for merging branches. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 998–1002

DeGroot MH, Schervish MJ (2012) Probability and statistics. Pearson Education, Boston

Elkan C (2001) The foundations of cost-sensitive learning. In: International joint conference on artificial intelligence. Lawrence Erlbaum Associates Ltd, vol 17, pp 973–978

Fagan ME (2001) Design and code inspections to reduce errors in program development. In: Pioneers and their contributions to software engineering. Springer, Berlin, pp 301–334

Fenton N, Neil M, Marsh W, Hearty P, Marquez D, Krause P, Mishra R (2007) Predicting software defects in varying development lifecycles using Bayesian nets. Inf Softw Technol 49(1):32–43

Gonzalez-Barahona JM, Izquierdo-Cortazar D, Robles G, del Castillo A (2014) Analyzing gerrit code review parameters with bicho. Electron Commun EASST

Gousios G, Pinzger M, Deursen AV (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering. ACM, pp 345–355

Gousios G, Zaidman A, Storey MA, Van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. In: Proceedings of the 37th international conference on software engineering-volume 1. IEEE Press, pp 358–368

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

Grbac TG, Mausa G, Basic BD (2013) Stability of software defect prediction in relation to levels of data imbalance. In: SQAMIA, pp 1–10

Hall MA, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. Sigkdd Explor 11(1):10–18

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

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21(9):1263–1284

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

Huang J, Ling CX (2005) Using AUC and accuracy in evaluating learning algorithms. IEEE Trans Knowl Data Eng 17(3):299–310

Huang Q, Xia X, Lo D (2017) Supervised vs unsupervised models: a holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 159–170

Jeong G, Kim S, Zimmermann T, Yi K (2009) Improving code review by predicting reviewers and acceptance of patches. In: Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006), pp 1–18

Jiang T, Tan L, Kim S (2013a) Personalized defect prediction. In: 2013 IEEE/ACM 28th international conference on automated software engineering (ASE). IEEE, pp 279–289

Jiang Y, Adams B, German DM (2013b) Will my patch make it? and how fast? Case study on the linux kernel. In: 2013 10th IEEE working conference on mining software repositories (MSR). IEEE, pp 101–110

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. IEEE Trans Softw Eng 39(6):757–773

Khoshgoftaar TM, Geleyn E, Nguyen L, Bullard L (2002) Cost-sensitive boosting in software quality modeling. In: 7th IEEE international symposium on high assurance systems engineering, 2002. Proceedings. IEEE, pp 51–60

Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196

Kononenko O, Baysal O, Guerrouj L, Cao Y, Godfrey MW (2015) Investigating code review quality: Do people and participation matter? In: 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 111–120

Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: 2010 7th IEEE working conference on mining software repositories (MSR). IEEE, pp 1–10

Leßenich O, Siegmund J, Apel S, Kästner C, Hunsen C (2016) Indicators for merge conflicts in the wild: survey and empirical study. Autom Softw Eng 1–35. https://doi.org/10.1007/s10515-017-0227-0

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496

Liu XY, Zhou ZH (2006) The influence of class imbalance on cost-sensitive learning: an empirical study. In: Sixth international conference on data mining, 2006. ICDM'06. IEEE, pp 970–974

Liu M, Miao L, Zhang D (2014) Two-stage cost-sensitive learning for software defect prediction. IEEE Trans Reliab 63(2):676–686

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat 50–60

Matsumoto S, Kamei Y, Monden A, Ki Matsumoto, Nakamura M (2010) An analysis of developer metrics for fault prediction. In: Proceedings of the 6th international conference on predictive models in software engineering. ACM, p 18

McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 192–201

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th international conference on predictor models in software engineering. ACM, p 7

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Tech J 5(2):169–180

Mukadam M, Bird C, Rigby PC (2013) Gerrit software code review data from android. In: 2013 10th IEEE working conference on mining software repositories (MSR). IEEE, pp 45–48

Rahman F, Posnett D, Devanbu P (2012) Recalling the imprecision of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM, p 61

Rajbahadur GK, Wang S, Kamei Y, Hassan AE (2017) The impact of using regression models to build defect classifiers. In: Proceedings of the 14th international conference on mining software repositories. IEEE Press, pp 135–145

Ratzinger J, Pinzger M, Gall H (2007) EQ-Mine:predicting short-term defects for software evolution. In: International conference on fundamental approaches to software engineering. Springer, Berlin, pp 12–26

Rigby PC, German DM (2006) A preliminary examination of code review processes in open source projects. Tech. rep., Technical Report DCS-305-IR, University of Victoria

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: Proceedings of the 30th international conference on Software engineering. ACM, pp 541–550

Romano D, Pinzger M (2011) Using source code metrics to predict change-prone java interfaces. In: 2011 27th IEEE international conference on software maintenance (ICSM). IEEE, pp 303–312

Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics 30(3):507–512

Shimagaki J, Kamei Y, McIntosh S, Hassan AE, Ubayashi N (2016) A study of the quality-impacting practices of modern code review at sony mobile. In: Proceedings of the 38th international conference on software engineering companion. ACM, pp 212–221

Shull F, Seaman C (2008) Inspecting the history of inspections: an example of evidence-based technology diffusion. IEEE Softw 25(1):88–90. https://doi.org/10.1109/MS.2008.7

Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN (2011) Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, pp 365–375

Thongtanunam P, McIntosh S, Hassan AE, Iida H (2016) Review participation in modern code review. Empir Softw Eng 1–50

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: 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 301–310

Tsay J, Dabbish L, Herbsleb J (2014) Influence of social and technical factors for evaluating contribution in GitHub. In: Proceedings of the 36th international conference on Software engineering. ACM, pp 356–366

Upton GJ (1992) Fisher's exact test. J R Stat Soc A Stat Soc 155(3):395–402

Votta LG (1993) Does every inspection need a meeting? ACM SIGSOFT Softw Eng Notes 18(5):107–114

Weiss GM, McCarthy K, Zabar B (2007) Cost-sensitive learning vs. sampling: which is best for handling unbalanced classes with unequal error costs? DMIN 7:35–41

Weißgerber P, Neu D, Diehl S (2008) Small patches get in! In: Proceedings of the 2008 international working conference on mining software repositories. ACM, pp 67–76

Wilcoxon F (1945) Individual comparisons by ranking methods. Biol Bull 1(6):80–83

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

Xia X, Lo D, Shihab E, Wang X, Yang X (2015a) Elblocker: predicting blocking bugs with ensemble imbalance learning. Inf Softw Technol 61:93–106

Xia X, Lo D, Wang X, Yang X (2015b) Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 261–270

Xia X, Lo D, Pan SJ, Nagappan N, Wang X (2016a) Hydra: massively compositional model for cross-project defect prediction. IEEE Trans Softw Eng 42(10):977–998

Xia X, Lo D, Wang X, Yang X (2016b) Collective personalized change classification with multiobjective search. IEEE Trans Reliab 65(4):1810–1829

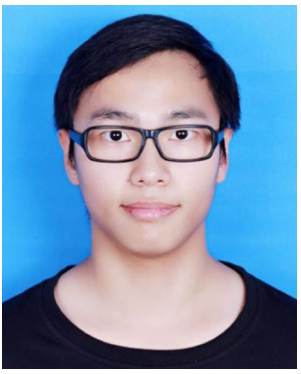
Xia X, Lo D, Ding Y, Al-Kofahi JM, Nguyen TN, Wang X (2017) Improving automated bug triaging with specialized topic model. IEEE Trans Softw Eng 43(3):272–297

Yang X, Kula RG, Yoshida N, Iida H (2016) Mining the modern code review repositories: a dataset of people, process and product. In: Proceedings of the 13th international conference on mining software repositories. ACM, pp 460–463

Zanetti MS, Scholtes I, Tessone CJ, Schweitzer F (2013) Categorizing bugs with social networks: a case study on four open source software communities. In: 2013 35th international conference on software engineering (ICSE). IEEE, pp 1032–1041

Zhang Y, Lo D, Xia X, Xu B, Sun J, Li S (2015) Combining software metrics and text features for vulnerable file prediction. In: 2015 20th international conference on engineering of complex computer systems (ICECCS). IEEE, pp 40–49

Zheng J (2010) Cost-sensitive boosting neural networks for software defect prediction. Expert Syst Appl 37(6):4537–4543



Yuanrui Fan is currently a PhD student in the College of Computer Science and Technology, Zhejiang University, China. His research interests include mining software repositories and empirical software engineering.

Xin Xia is a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).

Shanping Li received his Ph.D. degree from the College of Computer Science and Technology, Zhejiang University in 1993. He is currently a professor in the College of Computer Science and Technology, Zhejiang University. His research interests include Software Engineering, Distributed Computing, and the Linux Operating System.