Empirical Software Engineering (2022) 27: 7
https://doi.org/10.1007/s10664-021-10035-z

Reviewing rounds prediction for code patches

Yuan Huang[1] · Xingjian Liang[2] · Zhihao Chen[2] · Nan Jia[3] · Xiapu Luo[4] · Xiangping Chen[5] · Zibin Zheng[1] · Xiaocong Zhou[2]

Accepted: 20 July 2021 / Published online: 23 October 2021
© The Author(s) 2021

Abstract
Code review is one of the common activities to guarantee the reliability of software, while code review is time-consuming as it requires reviewers to inspect the source code of each patch. A patch may be reviewed more than once before it is eventually merged or abandoned, and then such a patch may tighten the development schedule of the developers and further affect the development progress of a project. Thus, a tool that predicts early on how long a patch will be reviewed can help developers take self-inspection beforehand for the patches that require long-time review. In this paper, we propose a novel method, *PMCost*, to predict the reviewing rounds of a patch. *PMCost* uses a number of features, including patch meta-features, code diff features, personal experience features and patch textual features, to better reflect code changes and review process. To examine the benefits of *PMCost*, we perform experiments on three large open source projects, namely Eclipse, OpenDaylight and OpenStack. The encouraging experimental results demonstrate the feasibility and effectiveness of our approach. Besides, we further study the why the proposed features contribute to the reviewing rounds prediction.

Keywords Code review · Code patch · Reviewing rounds · Machine learning · Discriminative feature

1 Introduction

Code review is an essential software engineering practice to reduce software defects and ensure the code quality of software, which is employed both in open source and industrial contexts (McIntosh et al. 2016). In a regular code review, a code patch (i.e., a code change) is submitted to the review tools such as Gerrit[1] by a developer. Then, one or more code

[1] https://www.gerritcodereview.com/

Communicated by: Ali Ouni, David Lo, Xin Xia, Alexander Serebrenik and Christoph Treude

This article belongs to the Topical Collection: *Recommendation Systems for Software Engineering*

✉ Xiangping Chen
chenxp8@mail.sysu.edu.cn

Extended author information available on the last page of the article.

reviewers will be assigned to inspect this code patch (Liu et al. 2019). Finally, the code patch will be merged into the code base or returned to the developers for re-modification if new defects or code conflicts are found by the reviewers (Zou et al. 2019).

In the software life cycle, code review provides good value in identifying defects in patches (Fagan 2002; McIntosh et al. 2014). However, code review is a time-consuming process since it requires reviewers to read, comprehend and critique the source code (Rigby and Storey 2011; Fan et al. 2018; Baum et al. 2019). We note that a code change might be inspected several rounds before it is eventually merged or abandoned. We investigated more than 50,000 review cases in the review history of the projects Eclipse, OpenDaylight and OpenStack. The distribution of the reviewing rounds is illustrated in Fig. 1. We can see that about half of the patches go through one round review, while the rest of the patches need more than two rounds review. 1% of the patches even need 20 or more rounds of review.

Obviously, a patch requiring multiple reviewing rounds will affect for developers (Fan et al. 2018). For example, the patches requiring multiple reviewing rounds will be suspended until the reviews are complete, which may affect the development progress of the developers, and further make the project schedule get tighter. Thus, if we can early predict how many rounds a patch will be reviewed, it will help developers allocate more effort beforehand to the patches for self-inspection and make a more effective arrangement for development. If a patch is predicted to have a long reviewing rounds, developers can take actions such as breaking the patch into smaller to speed up the review process.

However, accurately predicting the reviewing rounds is not an easy task. One main challenge is that a number of factors can affect the reviewing process. Existing research (Baysal et al. 2013) has found that both technical and non-technical factors can influence the reviewing process of a code patch. These factors include personal and organizational relationships, patch size, component, reviewer/submitter's experience, and reviewer load. Similarly, Jiang et al. (2013a) also found from their empirical study that the reviewing round is impacted by submission time, the number of affected components, and developer's experience, etc. Therefore, various factors need to be comprehensively considered and carefully selected when we build a model for predicting the reviewing round of a patch.

In this paper, we propose a learning-based method, *PMCost*, to help developers predict reviewing rounds of the patches. Specifically, we extract patch meta-features, code diff features, personal experience features and textual features as discriminative features to measure

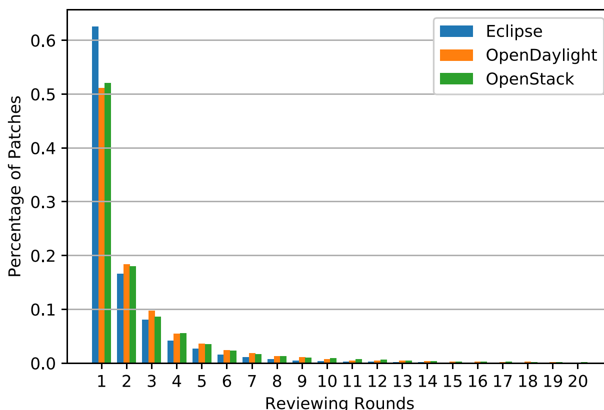


Fig. 1 The distribution of the reviewing rounds

the reviewing rounds. Among them, the patch meta-features represent non-technical factors in review process, such as reviewers, owners, subproject, etc. The code diff features represents the code churn via the number of modified methods, modified code lines, etc. The personal experience features represent the collaboration experience of the developers and the reviewers, which also indicates the activeness of a developer or reviewer in a project. The textual features represent natural language comment of a patch.

Then, we model the prediction as a triple-classification problem, i.e., patches with one-round reviewing (i.e., 1 round), patches with short-rounds reviewing (i.e., 2 to 6 rounds), patches with long-rounds reviewing (i.e., large than 6 rounds). We also try to predict the actual reviewing rounds of a patch by using the regression methods (in Section 7.3). However, the regression methods show a poor performance on the task of actual reviewing round prediction. Hence, we focused on the round range instead of actual reviewing rounds, because it can avoid giving the inaccurate prediction reviewing rounds (obtained by regression models) to developers. To evaluate the effective of *PMCost* for predicting reviewing rounds in within-project and cross-project scenarios and further explore the effective of the proposed features, we perform a case study on three large open-source projects to explore the following research questions in this paper:

- RQ1: How effective is *PMCost* in predicting reviewing rounds?
- RQ2: How effective is each subset of features in predicting reviewing rounds?
- RQ3: What features contribute the most to the reviewing rounds prediction?
- RQ4: Can *PMCost* be generalized in a cross-project scenario?

The results show that: *PMCost* with Random Forest as classifier achieves accuracies of 79.83%, 72.97% and 71.81% for Eclipse, OpenDaylight and OpenStack, respectively, which outperforms several baseline methods with other machine learning algorithms, such as Decision Tree, Multilayer Perceptron, etc.

There are two contributions to this study: 1) Four types of discriminative features are extracted to measure the reviewing rounds. 2) A case study on 3 open-source projects demonstrates the effectiveness of our method. We have uploaded the source code and datasets of the proposed method to the Github, and the URL is: https://github.com/liangxj8/PMCost. We describe the basic requirements and steps for running the proposed method.

The rest of the paper is organized as follows. The overall framework and motivating scenario are presented in Section 2; Section 3 describes the data we collected for further study. Section 4 describes the features we extracted. The setups and results of experiment are presented in Section 5. Section 6 discusses model limitation and solution. Section 7 discusses the results. The related work is presented in Section 8; Section 9 describes the threats to validity, while Section 10 summarizes our approach and outlines directions of future work.

2 Overall Framework and Motivating Scenario

2.1 Overall Framework

Figure 2 shows the overall framework of the proposed approach. The framework includes two phases: the model building phase and the prediction phase. In the model building phase, our goal is to build a prediction model based on the collected patch data. In the prediction phase, the model is used to determine the reviewing round of a patch.

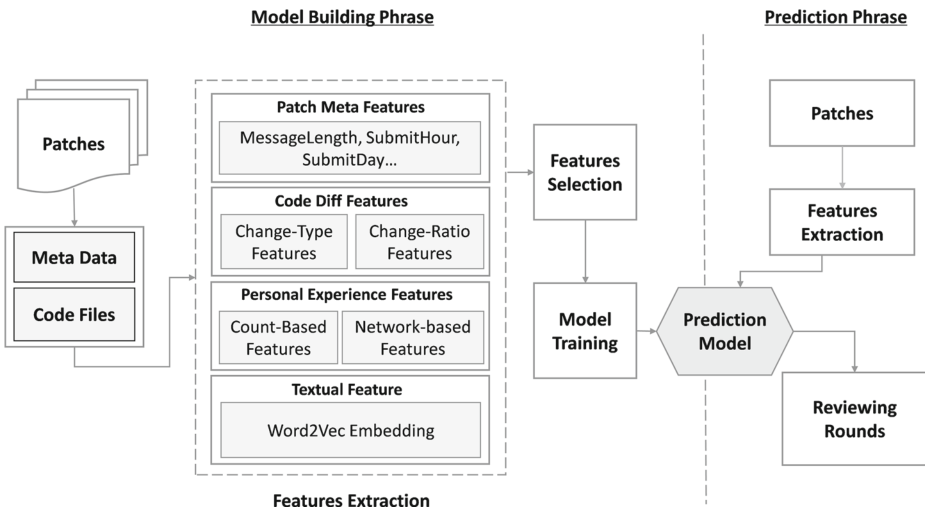


Fig. 2 The overall framework of the proposed approach

Our framework first collects the online patches from Gerrit, which includes the patch meta data (e.g., patch message) and code files. Then, our framework extracts features from the patches. Specifically, our framework analyzes the meta data to obtain the patch meta features, and analyzes code files to obtain the code diff features. Besides, personal experience features are extracted from the collaboration network between the developers and the reviewers, and the text features are extracted from the natural language comment of a patch by our framework. Meanwhile, our framework labels the classification of each patch (i.e., one-round, short-rounds and long-rounds reviewing) according to its actual reviewing rounds.

Next, our framework builds a prediction model based on machine learning algorithm. After the prediction model is constructed, it is used in the prediction phase to predict the reviewing round of a patch. For each patch, we first extract features as we do in the model building phase. Then, we input the features to the prediction model. This step outputs the prediction results, which is a label corresponding to the reviewing rounds that a patch needs.

2.2 Motivating Scenario

In this section, we use a motivating scenario to illustrate how *PMCost* helps developers/reviewers in development. Suppose developer Alice try to submit a patch to Gerrit, without the help of *PMCost*, Alice directly submits the patch to the Gerrit and waits for the response of the reviewer, i.e., merged to repository or returned back to revise. In the meantime, Alice can do another development activities. With the help of *PMCost*, Alice will get an estimated reviewing rounds. If the patch will cost a long time for reviewing, *PMCost* can also give the factors that makes the patch take a long time for reviewing. For example, *PMCost* hints that the length of the patch message makes the patch for a long time reviewing. Then, Alice can write a more concise message and then submit to the Gerrit. For reviewer Barry, without *PMCost*, he will choice a patch for reviewing according to the priorities of all the patches. Suppose *PMCost* gives each patch an estimated reviewing time, then Barry can consider a combination of patch priority and reviewing time to optimize the

patch review order, and decide which patch to review first. In such way, Barry can conduct code reviews in a more flexible way.

3 Data Collection

We collect ten thousands of patches from three open source projects in the Gerrit, including Eclipse, OpenDaylight, and OpenStack. In detail, we can directly collect the information of each patch from Gerrit, including patch meta data and code files. The patch meta data includes: patch submit time, status, patch message, code files with path names, etc. The status of each patch is labeled as “Merged” or “Abandoned”. We download these two types of patches as our dataset, because both merged and abandoned patches need to undergo reviews. Then, the reviewing rounds for both kinds of patches should be approximated. Figure 3 shows a patch (ID: 26415) of Eclipse in Gerrit. There are many subprojects in the three projects. To avoid studying inactive or dead subprojects, we choose those that contain more than 50 patches. Meanwhile, we require each patch includes at least one source file, otherwise, we will remove it from the dataset.

The patch code files are the exact inspecting object by reviewers, which reflect the code churn of the current patch. For each reviewing round, the developer should submit a new patch set to Gerrit (i.e., a reviewing round corresponds to a patch set). As a result, a patch often contains multiple patch sets before it is merged or abandoned. As Fig. 4 shows, there are two patch sets for the patch 26415. In our study, we only extract the diff files of the first patch set of a patch since we focus on predicting how long a patch will be merged or abandoned when it is submitted initially.

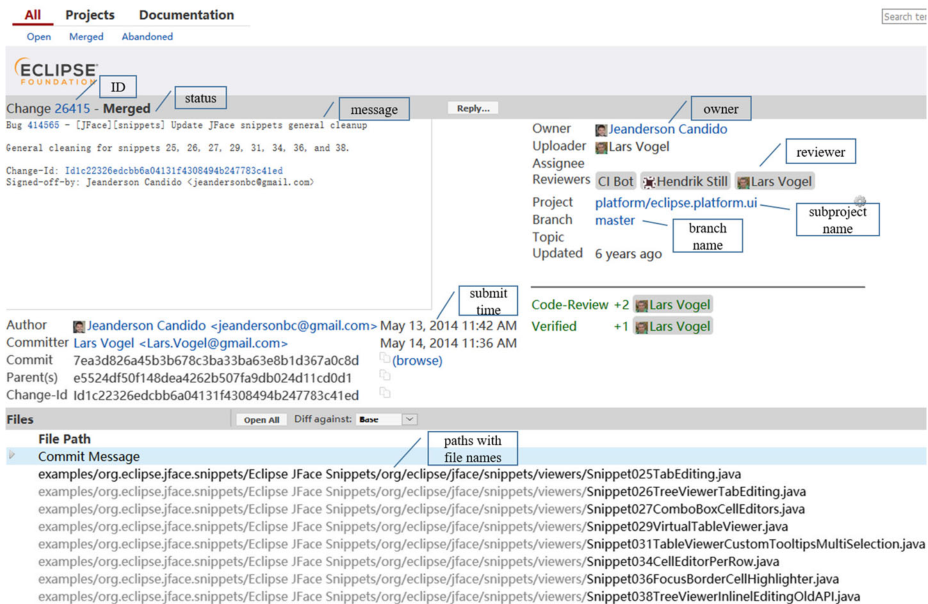


Fig. 3 A patch example

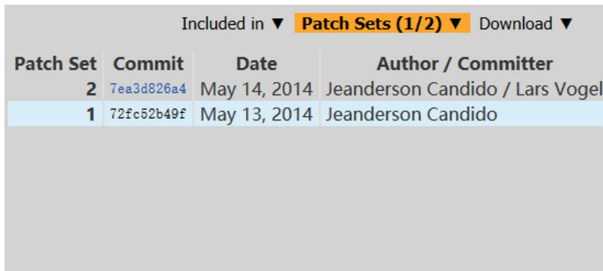


Fig. 4 Patch sets

In order to understand how long the new submitted patch will be reviewed, we count the distribution of reviewing rounds shown in Fig. 1. We found that about 50% patches need only one round of review, and the rounds of all patches are distributed in a long tail.

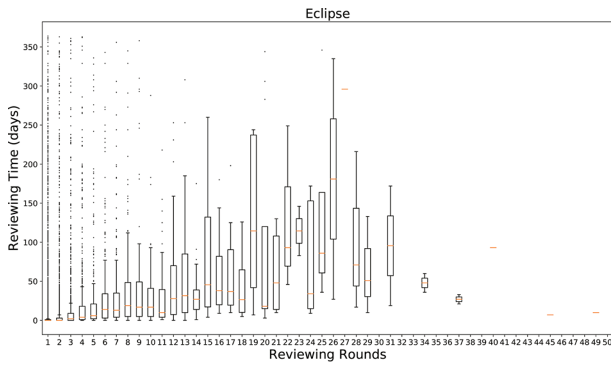
We also study the relevance between the reviewing rounds and average reviewing time on the collected data set. To provide a concrete statistic analysis, we use the box plot (Fig. 5) to show the distribution of reviewing rounds and reviewing time on the 3 projects, and the vertical axis in these figures are the number of the actual reviewing days of the patches.

We can see from Fig. 5, as the number of reviewing rounds increase, the reviewing time is also increasing. The average reviewing time of the patches with 1 reviewing round is about 0.33 day on the three datasets. The average reviewing time of the patches with 2 to 6 reviewing rounds are about 5.3 days and the average reviewing time of the patches with more than 6 reviewing rounds are about 31.3 days on datasets. To investigate correlation between the number of reviewing rounds and the reviewing time, we employ the Spearman Correlation Coefficient (2008). The results show that the Spearman Correlation Coefficient of three projects Eclipse, OpenDaylight and OpenStack are 0.68, 0.75 and 0.78, respectively, meaning that there is a positive linear correlation between the number of reviewing rounds and the reviewing time. Then, we divide the reviewing rounds of the patches into three intervals: 1 round, 2-6 rounds, more than 6 rounds, and we call them one-round reviewing, short-rounds reviewing and long-rounds reviewing patches in this paper.

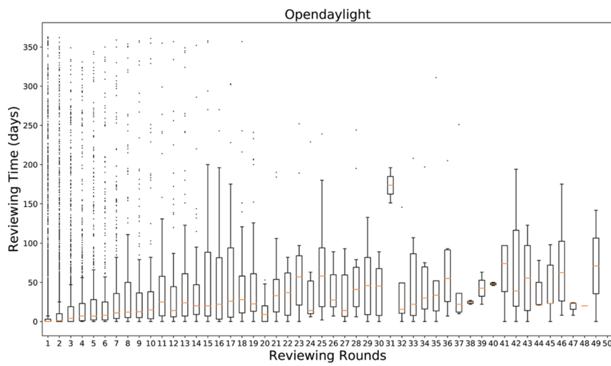
Table 1 shows the detailed information of the data sets collected from the three projects. These three projects contain several years of code review data related to different sub projects. In this study we considered the code patches of the projects that can be collected in recent time period. For example, we collect the code patches of Eclipse from October, 2020, and reverse-collect patches as the project evolves. Meanwhile, we need to make sure that there are enough data to train the model. Hence, we collect a total of 19,964 patches after removing the noise (i.e., the patches without source files) during the defined period, i.e., January, 2017 to October, 2020.

4 Feature Extraction

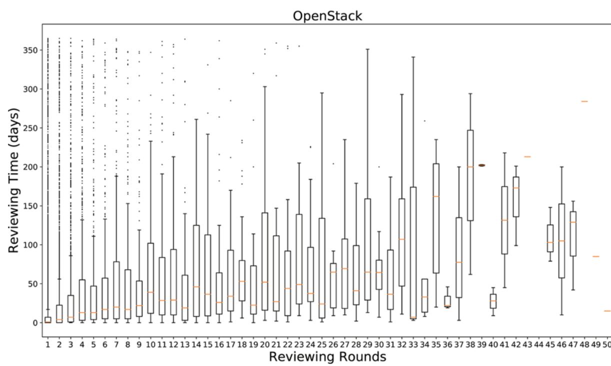
We extract four types of features in total to characterize a patch, including patch meta features (PMF), personal experience features (PEF), code diff features (CDF) and textual features (TF). The detailed introduction of the feature extraction is described as follows.



(a) Eclipse



(b) Opendaylight



(c) Openstack

Fig. 5 The reviewing time of patches with different reviewing rounds in the three projects

Table 1 Statistical characteristics of the datasets

Projects	Time span	# of Patches		
		One-round reviewing	Short-rounds reviewing	Long-rounds reviewing
Eclipse	2017.01-2020.10	12,484	6,622	858
OpenDaylight	2017.12-2020.10	10,201	7,898	1,856
OpenStack	2017.07-2020.10	10,351	7,559	1,982

4.1 Patch Meta Features

Patch meta data describes the basic attributes of a patch from the patch submitting time to the patch message. A patch can be characterized by the meta data, and we regard the basic attributes of a patch as PMF features.

We directly encode the length of a patch message (*MessageLength*) into the PMF features. Meanwhile, Eyolfson et al. (2011) found that there is strong relevance between the submitting time and the correctness of a patch. Inspired by this, we use *SubmitHour* to characterize the submitting time by normalizing the original time (e.g., "May 13, 2014, 11:42 PM") into 24-hour time format (i.e., "23"). Meanwhile, we also apply *SubmitDay* to determine the date of submitting a patch, e.g., the day "May 13, 2014" is normalized as "13".

A code change may cause ripple effects in a system, which makes a patch usually contain multiple entities (Kagdi et al. 2013; Huang et al. 2021). The diffusion of a patch represents the impacted entities in a system, which also reflects the impact scope and the scale of a patch. Then, the diffusion of a patch can be used to determine how long the patch will be reviewed. Inspired by Kamei et al. (2012), we use multilevel-diffusion features to measure the diffusion of a patch. As shown in Fig. 3, the modified files stored in Gerrit Repositories have a definite storing path like "examples/org.eclipse. jface.snippets/ Eclipse JFace Snippets/ org/eclipse/jface/ snippets/viewers/ Snippet025TabEditing.java". We split the paths by "/" and number them in the reverse order from the bottom "Snippet025TabEditing.java". Restricting the maximum level to 13, we totally extract 13 features by counting the number of unique values of each path level, defined as $FileNumLv_0 \sim FileNumLv_{12}$. We take the maximum path level as 13 because it can cover more than 90% of the file paths in our dataset. Especially, $FileNumLv_0$ equal to 5 means that a newly submitted patch contains 5 unique .java files, while $FileNumLv_1 \sim FileNumLv_{12}$ means the number of distinct folders at the 1st $\sim$ 12nd level in the patch path. All PMF are shown in Table 2.

Table 2 Patch meta features

Category	Features	Description
Meta	*MessageLength*	The number of words involved in the patch message
	SubmitHour	Submitting time normalized to 24-hour format of the current patch
	SubmitDay	Submitting day of the current patch
	$FileNumLv_0 \sim FileNumLv_{12}$	The number of unique file (or folder) in each level of the path in a patch

4.2 Code Diff Features

The finding in our previous study (Huang et al. 2018) shows that a patch with more code changes will take reviewers more time to comprehend it. Therefore, the change scale of a patch is an important indicator for predicting the reviewing rounds. To describe how many code changes occur in a patch, diff comparison between the two versions of a patch set is necessary. Each patch set has two versions, and we can extract the diff code from them. As mentioned, we only extract diff code in the first patch set since we focus on predicting how long a patch will take to be reviewed when it is submitted initially. In general, we extract two types of Code Diff Features (CDF) as follows.

Change-Type Features Code modification can be divided into inserting, updating and deleting. All the software entities such as class, statement, attribute, method, parameter and comment can be the modified objects. Then, we extract all types of modified software entities by using a static analysis tool ChangeDistiller[2]. They are: "*ClassAddition*", "*ClassDelete*", "*MethodRenaming*", "*StatementInsert*", "*StatementUpdate*" and so on. To construct these features, we count the number of all types of the modifications for all code files in a patch.

Change-Ratio Features Every type of modification has its directly influenced code range. The change range can be described by ChangeDistiller as a two-tuple like $< 144, 160 >$, in which the first element indicates the start line of change effected and the second element indicates the end line of change affected. Then, we subtract the start line from the end line to count the delta of the code lines. Finally, we add up all the delta of the code lines of all code file in a patch, and divide the total number of the code lines in all the files of the patch is the *ChangedCodeRatio* feature. All CDF are illustrated in Table 3.

4.3 Personal Experience Features

Baysal et al. (2013) and Jiang et al. (2013b) found that the experience of a patch owner significantly impacts the code review outcomes. Referring to the conclusion of Kagdi et al. (2008), the persons who are more active in the review system should have more knowledge about the system. Then, to characterize the person experience, PEF are divided into count-based and network-based features.

Count-Based Features We propose a number of count-based features to characterize the personal experience such as *OwnerPatchNum*, *OwnerFileNum*, *OwnerAvgReviewRonds*, *ReviewerPatchNum*, *ReviewerFileNum*, *ReviewerAvgReviewRonds*. *OwnerPatchNum* is the number of patches that an owner has ever participated. Intuitively, the core developers (i.e., patch owners) make more contributions to a project, and *OwnerPatchNum* reflects the frequency of a developer making code changes. Besides, the familiarity to a project for developers is different, as the number of code files they have read, created and modified is different. Therefore, we extract the number of files that a developer has involved as one of the features. Similar to the feature *FileNumLv*, we split file paths by "/" and numbered them in the reverse order from the bottom. Restricting the maximum path level

[2] https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home

Table 3 Code diff features

Category	Features	Description
Change-Type	*ClassAddition*, *ClassRenaming*, *ClassDelete* *ParentClassDelete*, *ParentClassAddition*, *ParentClassRenaming*	The numbers of class-involved modifications
	StatementDelete, *StatementInsert*, *StatementUpdate*, *StatementOrderUpdate*	The numbers of statement-involved modifications
	AttributeDelete, *AttributeInsert*, *AttributeRenaming*, *AttributeTypeUpdate*	The number of Attribute-involved modifications
	ReturnTypeUpdate, *ReturnTypeDelete*, *ReturnTypeInsert*	The numbers of return type-involved modifications
	ParameterDelete, *ParameterInsert*, *ParameterOrderingUpdate*, *ParameterRenaming*, *ParameterTypeUpdate*	The numbers of parameter-involved modifications
	MethodAddition, *MethodDelete*, *MethodRenaming*	The numbers of method-involved modifications
	CommentDelete, *CommentInsert* *CommentUpdate*	The numbers of comment-involved modifications
Change-Ratio	*ChangedCodeRatio*	The proportion of code lines influenced by all types of modification

to 13, we extract 13 levels of *OwnerFileNum* features named $OwnerFileNumLv_0 \sim OwnerFileNumLv_{12}$. For instance, $OwnerFileNumLv_0$ is the number of files a developer has ever submitted, $OwnerFileNumLv_1$ is the number of penultimate path level (i.e., the direct parent folders of the files) he/she has ever been involved, and so on. More directly, *OwnerAvgReviewRonds* calculates the average number of reviewing rounds of all patches that a developer has participated in the past.

Meanwhile, we also pay attention to the influence of reviewer experience on the reviewing rounds, and extract the features *ReviewerPatchNum*, *ReviewerFileNum*, *ReviewerAvgReviewRonds*. It should be noted that when a new patch is submitted, a project moderator may assign multiple reviewers to the patch (Rigby and Bird 2013). In this case, we first count the number of patches that each reviewer has ever reviewed, and then we add all the patches that these reviewers have reviewed. At last, feature *ReviewerPatchNum* for the patch equal to the average values of the total number of the patches. In a similar manner, we can compute the values of features *ReviewerFileNum* and *ReviewerAvgReviewRonds*. The detailed introduction of the count-based PEF can be found in Table 4.

Network-based Features Zanetti et al. (2013) proposed an approach to predict whether a bug report is valid based on the collaborations of reporters. Following their studies, we construct two modes of relational networks based on symbioses of patch owners and reviewers.

Table 4 Count-based PEF

Category	Features	Description
Count-Based	*OwnerPatchNum*	Number of patches that an owner has ever participated.
	$OwnerFileNumLv_0 \sim Lv_{12}$	Number of file path level that the owner has ever been involved.
	OwnerAvgReviewRonds	The average number of reviewing rounds of all patches that an owner has participated in the past.
	ReviewerPatchNum	Average number of patches that all reviewers has ever participated.
	$ReviewerFileNumLv_0 \sim Lv_{12}$	Number of file path level that all the reviewers have been involved in the past.
	ReviewerAvgReviewRonds	The average number of reviewing rounds of all patches that reviewers have participated in the past

Direct Collaboration Mode If an owner appears together in a patch with a reviewer, we declare that there is a direct collaboration relationship between them. We first take owners/reviewers as nodes, and build edges according to the direct collaboration relationship and finally construct a directed weighted graph. In the graph, each edge points to a reviewer from an owner.

Indirect Symbioses Mode If two people are together involved in the same file/file folder, no mater they are owners or reviewers, we declare that there is an indirect symbiosis relationship between them and build an undirected weighted edge between these two person-nodes as their relationship has no intense directivity. Here, an indirect symbiosis relationship refers to names and path levels of files or file folders, which can be split from file paths in the way mentioned above. Here, we also restrict the maximum path level to 13.

The edge weights in both modes of networks refer to how many times two persons have been both involved in a patch, a file or a folder. For example, when an owner and a reviewer have ever collaborated in 10 patches, the weight of the directed edge between them is 10. The examples of the two modes of networks are shown in Fig. 6, in which we use sets of tuples like $(Person1, Person2, EdgeWeight)$ to represent a weighted graph. Next, we introduce three network-based operationalizations.

Degree_centrality is the degree of a node, which is simply a count of how many connections (i.e., edges) a node has. E.g., when a node has 10 connections, it has a degree centrality of 10.

Closeness_centrality is to capture the average distance between a node and every other node in a network. A low *closeness_centrality* means that a person is directly connected or "just a hop away" from most others in the network (Smith et al. 2011).

Betweenness_centrality is a measure of how important the node is to the flow of information through a network. In graph theory (Bollobas 1998), *betweenness_centrality* is calculated based on shortest paths. For every pair of nodes in a connected graph, there exists

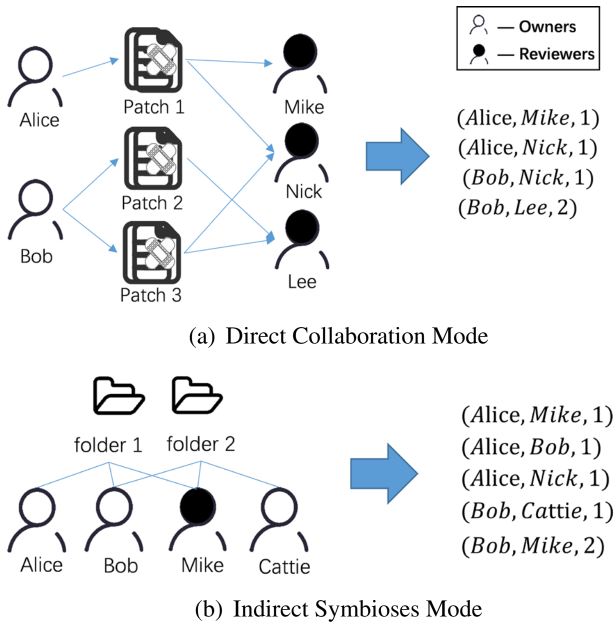


Fig. 6 Two modes of the networks

at least one shortest path between the nodes such that the sum of the weights of the edges (for weighted graphs) is minimized.

We use NetworkX[3] to construct the graphs, and further extract *degree_centrality*, *closeness_centrality*, *betweenness_centrality* of each node. Because of the one-by-one relationship between a patch and a specific owner in the networks, these network-based metrics of owner nodes can directly be used as a patch feature. Since a patch may have multiple reviewers, we calculate the mean of these metrics of the reviewers for a patch. All network-based PEF are illustrated in Table 5.

4.4 Textual Features

A patch message usually describes how a bug is fixed or a software functionality is implemented, which can partly reflect the importance and urgency of a patch (Huang et al. 2017). Therefore, the semantics of a patch message may be used to estimate the reviewing rounds of a patch, and we extract the textual features from the patch message to characterize its semantic information.

In this study, we employ word embeddings to capture the semantics of a patch message. Word embeddings are unsupervised word representations that only require large amounts of unlabeled text to learn (Mikolov et al. 2013). In this work, we collect the patch messages as software engineering text. First, we preprocess the patch message. This includes removing punctuation symbols from the patch message, and filtering out meaningless words, such as *aaa* and *xxx*. Additionally, we reduce the amount of vocabulary in the entire corpus. Specifically, we apply the stem segmentation technique. Because English verbs may appear

[3] https://networkx.github.io/

Table 5 Network-based PEF

Category	Features	Description
Network-Based	*OwnerDegreeCentrality* *OwnerClosenessCentrality* *OwnerBetweennessCentrality*	degree_centrality, closeness_centrality and betweenness_centrality of the owner in the Direct Collaboration Network.
	AvgReviewerDegreeCentrality *AvgReviewerClosenessCentrality* *AvgReviewerBetweennessCentrality*	The average degree_centrality, closeness_centrality and betweenness_centrality of the reviewers in the Direct Collaboration Network.
	$OwnerDegreeCentralityLv_0 \sim Lv_{12}$ $OwnerClosenessCentralityLv_0 \sim Lv_{12}$ $OwnerBetweennessCentralityLv_0 \sim Lv_{12}$	degree_centrality, closeness_centrality and betweenness_centrality of the owner who has ever been involved in the past, calculated from bottom to the 13rd path layer in the Indirect Symbioses Network.
	$AvgReviewerDegreeCentralityLv_0 \sim Lv_{12}$ $AvgReviewerClosenessCentralityLv_0 \sim Lv_{12}$ $AvgReviewerBetweennessCentralityLv_0 \sim Lv_{12}$	The average degree_centrality, closeness_centrality and betweenness_centrality of the reviewers who has ever been involved in the past, calculated from bottom to the 13rd path layer in the Indirect Symbioses Network.

in different tenses, such as past tense, future tense, and perfect tense, we transform verbs of different tenses into their original forms.

To obtain the vector representation of a word, we use the continuous skip-gram model to learn the word embedding of a central word (i.e., w_i) (Mikolov et al. 2013). It is well known that the required word embedding is an intermediate result of the continuous skip-gram model. Continuous skip-gram is effective at predicting the surrounding words in a context window of $2k$+1 words (generally, k=2, and the window size is 5). The objective function of the skip-gram model aims at maximizing the sum of log probabilities of the surrounding context words conditioned on the central word (Mikolov et al. 2013):

$$\sum_{i=1}^{n} \sum_{-k \leq j \leq k, j \neq 0} \log p(w_{i+j}|w_i) \tag{1}$$

where w_i and w_{i+j} denote the central word and the context word, respectively, in a context window of length $2k$+1 and n denotes the length of the word sequence. The term $\log p(w_{i+j}|w_i)$ is the conditional probability, defined using the softmax function:

$$\log p(w_{i+j}|w_i) = \frac{exp(v_{w_{i+j}}^{'T} v_{w_i})}{\sum_{w \in W} exp(v_w^{'T} v_{w_i})} \tag{2}$$

where v_w and v'_w are the input and output vectors of a word *w* in the underlying neural model, and *W* is the vocabulary of all words. Intuitively, $p(w_{i+j}|w_i)$ estimates the normalized probability of a word w_{i+j} appearing in the context of a central word w_i over all words in the vocabulary. Here, we employ the negative sampling method (Mikolov et al. 2013) to compute this probability.

After training the model, each word in the corpus is associated with a vector representation and forms a word dictionary. To obtain the semantic information of a patch message, we first identify the words in a patch message and then determined the corresponding vector representation of each word from the dictionary. Subsequently, we add the vectors of all words dimension by dimension, and then average the sum of all the vectors.

5 Evaluation

5.1 Research Questions

To verify the effectiveness of the proposed method, we perform a case study on three large open source projects, i.e., Eclipse, OpenDaylight and OpenStack, aiming to explore the following research questions in this study:

RQ1: How effective is *PMCost* in predicting patch reviewing rounds?

We try to use the features extracted from the patches to help developers estimate if their code changes can be reviewed with 1 reviewing round, or 2 to 6 rounds or more than 6 rounds. Then, we need to evaluate how effective of *PMCost* in predicting reviewing rounds. For comparison, we employ the Random Guess (RG) algorithm as a baseline. Random Guess randomly gives a classifying probability according to the proportion in the training set for each reviewing rounds level and chooses the largest one as the final predicted level. Meanwhile, we employ several popular machine learning algorithms as the classifier of *PMCost*. They are: LightGBM (LGB) (Ke et al. 2017a), Support Vector Machines (SVM) (Cortes and Vapnik 1995), Logistic Regression (LR) (Pregibon and et al 1981), Decision Tree (DT) (Quinlan 1987), Random Forest (RF) (Breiman 2001a), Multilayer Perceptron (MLP) (Pal and Mitra 1992).

RQ2: How effective each subset of features in predicting patch reviewing rounds?

In this study, four groups of features, that conceptually represent different social and technical aspects relevant to the code review, are considered to train a prediction model. With this analysis we can learn about whether each subset of features is usful in predicting patch reviewing rounds.

RQ3: What features contribute the most to the reviewing rounds prediction?

In this research question, we would like to find out the most important features in estimating the patch reviewing rounds. Meanwhile, we expect to know why the features are useful in determining the reviewing rounds of a patch. Feature analysis is also important for the developers. Since we find out which features are important in the prediction model, we can give a suggestion to the developers for quickly reviewing their patches. For example, if developers know what is causing the patch review to slow down, he can focus on that and modify their patches accordingly so as to get a faster review.

RQ4: Can *PMCost* be generalized in a cross-project scenario?

We expect to build a prediction model that can be applied to different projects and even the projects based on different program languages. So we conduct a cross-project and cross-language evaluation. We define three ways of generalization: same-language-cross, language-cross, union-training. The same-language-cross way indicates that we first train a

model based on data of a project and then apply it to another project, while the two projects are based on the same program language. The language-cross way indicates that we first train a model based on data of a project and then apply it to another project, while the two projects are based on different program languages. The union-training way indicates that we first train a model based a union dataset of several projects and then apply the model to test the dataset coming from each project.

5.2 Evaluation Criteria

To evaluate the effectiveness of *PMCost*, we employ the precision (P), recall (R), accuracy (ACC), and F1-score (F1) to measure the results. The precision, recall, F1-score are computed as follows:

$$precision = \frac{TP}{TP + FP} \tag{3}$$

$$recall = \frac{TP}{TP + FN} \tag{4}$$

$$accuracy = \frac{TP + FN}{TP + FN + TN + FP} \tag{5}$$

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{6}$$

$$F1(overall) = \frac{1}{3} * (F1(a) + F1(b) + F1(c)) \tag{7}$$

where *TP* is the number of true positives, *FP* is the number of false positives, *TN* is the number of true negatives, and *FN* is the number of false negatives. Therefore, the precision is the percentage of positive instances identified by our classifier that are actually positive instances. The recall is the percentage of true positive instances that are successfully retrieved by our classifier.

Accuracy is the proportion of true results (both true positives and true negatives) among the total number of cases examined. These four metrics are also suitable for negative instances. The F1 is the weighted harmonic mean of the precision and the recall and can be used as a comprehensive indicator of the combined precision and recall values. There are three types of instances in our dataset, and then number of each types of instances is different. So we introduce the overall F1 shown in (7) to measure the performance of our model in the multi-classification task. We firstly calculate F1 for each label such as *a*, *b*, and *c*, and the calculate the unweighted mean of the three F1. In the evaluation, we use the 5-fold cross-validation strategy and average 5-fold tested results as the final result.

5.3 Results and Discussion

5.3.1 RQ1: How effective is PMCost in predicting reviewing rounds?

To evaluate the performance of *PMCost*, we first extract 4 types of features from the dataset, i.e., Eclipse, OpenDaylight, and OpenStack. Then, we label all patches according to the actual reviewing rounds they had been through. At last, we calculate the average precision, recall and F1 for the one-round reviewing, short-rounds reviewing and long-rounds reviewing patches.

Based on previous experience of parameter adjustment (Huang et al. 2020), the following hyper-parameters are adjusted for LightGBM, including: the *BoostingType* is the gbdt, and the *LearningRate* is 0.05. The feature selection ratio of building a tree (i.e., *FeatureFraction*) is 0.9. The sampling ratio of building a tree (i.e., *FeatureFraction*) is also 0.9. Meanwhile, the hyper-parameter of Random Forest includes: *BagSizePercent*=30; for each classifier of the Random Forest, 30% of the original training set is randomly selected for training; *NumFeatures*=6. For each classifier of the Random Forest, 6 features of the feature set are randomly selected for training; *NumIterations*=300 means the Random Forest contains 300 classifiers. For decision tree, we use a fast decision tree learner, which builds a decision tree using information gain and prunes it using reduced-error pruning. For SVM, we use radial basis function as the kernel function. To speed up the model training, we set *BathSize*=200, *CacheSize*=100. In addition, the other hyper-parameters of the machine learning algorithms are set to the default values. Table 6 shows the hyper-parameters for all the machine learning algorithms.

Table 7 presents the evaluation results. RF and DT achieve better F1 values on the one-round reviewing patches across the three datasets (i.e., Eclipse, OpenDaylight and OpenStack), respectively, which shows the improvements compared with other machine learning algorithms. Meanwhile, we can observe from Table 7 that RF achieve better precision values when being applied on the short-rounds reviewing patches on datasets Eclipse and OpenStack compared with other machine learning algorithms.

Similarly, the precision of RF is higher than other machine learning algorithms on the three datasets when being applied on the long-rounds reviewing patches, while the recall is lower than the ones of some other algorithms. Improving the precision is the main purpose of our method in the scenario of predicting the reviewing rounds, as telling developers the accurate reviewing rounds is what we are pursuing.

Besides, RF outperforms other models in terms of the overall ACC and F1 metrics, which shows that RF is probably more suitable to handle the mixed-type features and able to handle reviewing time prediction better. Then, we choice RF algorithm to build *PMCost* in the following experiments.

5.3.2 RQ2: How effective is each subset of features in predicting reviewing round?

Furthermore, we evaluated the effect of different features (i.e., CDF, TF, PMF and PEF) on the prediction accuracy, as presented in Table 8. The results shown in Table 8, is testing CDF alone and then gradually add up another group of features until all the features. Meanwhile,

Table 6 The hyper-parameters of the machine learning algorithms

Algorithms	Hyper-parameters
LightGBM	*BoostingType=gbdt*, *LearningRate*=0.05, *FeatureFraction*=0.9
Random Forest	*BagSizePercent*=30, *NumFeatures*=6, *NumIterations*=300
Decision Tree	*Fast Learner*, *Reduced-error Pruning*
SVM	*Radial Basis Function*, *BathSize*=200, *CacheSize*=100
Logistic Regression	*Default*
Multilayer Perceptron	*Default*

Table 7 The performance comparison for different machine learning algorithms

	Models	One-round reviewing			short-rounds reviewing			Long-rounds reviewing			Overall	
		P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	RG	61.17	33.65	42.80	33.89	32.19	31.63	4.06	34.43	7.07	55.50	27.17
	LR	92.71	28.74	38.52	50.84	59.45	52.90	15.02	57.20	23.58	69.95	38.33
	SVM	59.98	16.51	20.59	49.01	51.72	44.54	13.92	64.26	21.91	66.00	29.01
	MLP	83.63	52.44	63.10	52.22	66.85	58.43	18.50	43.11	24.74	75.86	48.76
	DT	72.15	69.33	70.60	46.90	47.99	47.32	14.22	29.81	18.99	74.61	45.64
	RF	79.60	67.08	72.56	57.40	54.05	55.57	25.85	23.03	23.82	79.83	50.65
	LGB	34.98	8.01	12.78	29.98	23.72	25.51	10.72	12.40	10.63	68.99	16.31
OpenDaylight	RG	49.62	32.89	39.42	39.82	32.26	35.56	9.65	32.31	14.58	55.41	29.85
	LR	72.69	43.92	54.27	52.39	16.97	24.47	26.05	52.64	33.41	68.37	37.38
	SVM	69.64	28.45	37.83	10.93	1.32	2.16	30.03	47.64	35.98	67.35	25.32
	MLP	73.45	54.79	60.33	54.62	33.49	38.02	25.62	49.35	33.02	69.85	43.79
	DT	58.58	63.66	60.68	47.02	41.65	43.85	22.36	31.54	25.52	66.55	43.35
	RF	69.45	62.77	65.21	55.74	35.18	42.57	38.76	26.28	29.61	72.97	45.80
	LGB	59.68	31.32	38.17	24.92	10.64	14.57	31.09	27.62	26.78	68.88	26.51
OpenStack	RG	54.47	33.66	41.34	37.28	33.79	35.29	8.90	33.25	13.65	55.52	30.09
	LR	71.94	22.93	31.65	63.23	1.17	2.13	24.33	33.39	27.13	66.67	20.30
	SVM	82.79	12.80	19.62	34.36	0.40	0.78	39.83	21.72	26.23	67.67	15.54
	MLP	74.23	44.23	49.92	45.53	15.33	18.34	26.96	53.05	32.99	68.14	33.75
	DT	62.43	58.60	59.88	41.91	40.10	40.63	20.23	32.84	24.57	66.08	41.69
	RF	70.64	54.75	61.19	47.51	25.70	33.24	41.04	28.35	32.27	71.81	42.23
	LGB	37.47	23.75	28.99	35.15	4.29	7.24	20.77	17.05	17.65	69.76	17.96

Table 8 The performance comparison for accumulating different feature groups

Projects	Features	One-round reviewing			Short-rounds reviewing			Long-rounds reviewing			Overall	
		P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	CDF	55.52	15.73	24.37	40.31	4.46	7.68	6.37	2.78	3.70	67.26	11.92
	CDF+TF	77.18	51.15	61.08	50.39	42.19	44.10	13.19	11.29	10.38	74.57	38.52
	TF+PMF	80.52	46.47	57.92	48.32	56.96	51.26	16.02	38.33	22.24	72.93	43.81
	CDF+TF+PMF	82.77	60.00	69.29	55.62	51.07	52.56	28.18	22.87	24.36	78.60	48.74
	CDF+TF+PEF	82.47	62.57	70.91	54.71	50.64	52.01	27.81	25.42	25.45	78.68	49.46
	CDF+TF+PMF+PEF	79.60	67.08	72.56	57.40	54.05	55.57	25.85	23.03	23.82	79.83	50.65
OpenDaylight	CDF	43.01	4.99	8.84	47.69	7.54	12.99	17.30	3.15	5.15	65.91	8.99
	CDF+TF	68.42	42.51	51.97	49.73	22.01	30.35	23.99	8.07	11.89	69.97	31.40
	TF+PMF	69.90	39.12	49.63	49.64	43.96	46.31	24.00	33.61	27.15	68.03	41.03
	CDF+TF+PMF	73.75	49.38	58.90	53.38	30.36	38.42	35.28	21.07	25.42	72.06	40.91
	CDF+PMF+PEF	65.41	63.18	63.51	56.41	33.88	41.92	33.88	23.85	26.71	71.91	44.05
	CDF+TF+PMF+PEF	69.45	62.77	65.21	55.74	35.18	42.57	38.76	26.28	29.61	72.97	45.80
OpenStack	CDF	45.04	2.02	3.85	41.41	6.23	10.77	23.40	7.35	10.88	65.73	8.50
	CDF+TF	71.30	31.08	42.65	42.06	19.63	26.51	27.78	12.66	17.19	68.89	28.78
	TF+PMF	68.04	44.84	53.89	42.04	38.40	39.80	26.38	20.90	23.23	68.15	38.97
	CDF+TF+PMF	75.65	37.33	49.60	46.69	21.20	29.02	38.61	22.56	28.02	70.86	35.55
	CDF+PMF+PEF	69.07	52.01	58.66	47.05	25.62	33.07	36.54	26.08	29.48	70.88	40.40
	CDF+TF+PMF+PEF	70.64	54.75	61.19	47.51	25.70	33.24	41.04	28.35	32.27	71.81	42.23

we also test the random combinations of the features such as ”TF + PMF”. Each result in Table 8 represents the value of a metric obtained with the RF algorithm.

Generally, every type of features is useful. We can see that when only applying CDF to train the learning model, *PMCost* achieves accuracies of 67.26%, 65.91% and 65.73% for Eclipse, OpenDaylight and OpenStack, respectively. When combining CDF with TF, the accuracies increase to 74.57%, 69.97%, and 68.89%, respectively. Then, when taking PMF (i.e., ”CDF + TF + PMF”) into the learning model, the accuracies on the three datasets exhibit obvious improvements, and this result reveals that the patch meta features provide a good indication on the reviewing round prediction. Lastly, when adding PEF (i.e., ”CDF + TF + PMF + PEF”) into the learning model, the accuracies have increased considerably.

In summary, we can observe that the prediction accuracy presents a continuous step-like increase when adding new types of features into the learning model. This continuous accuracy increase can also confirm that the four types of features are useful in characterizing the patch reviewing round from four different perspectives.

5.3.3 RQ3: What features actually contributes to the prediction?

To further understand the discriminative power of the extracted features, we ranked the 15 most significant features according to their information gain values in Fig. 7. The information gain computation is based on the dataset of the 3 projects, hence the information gain value of each feature reflects the contribution to the class in the global dataset.

It can be seen clearly from the Fig. 7 that the most significant feature is *AvgOwnerReviewRounds*, which is used to character the average reviewing rounds for all the patches of a developer in the dataset. Because *AvgOwnerReviewRounds* directly reflects the past review efficiency of the patches written by a developer, it plays a most important role in determining the reviewing rounds. To reveal the reasons that make the feature *AvgOwnerReviewRounds* to be significant, we will further discuss it in Section 7.2.

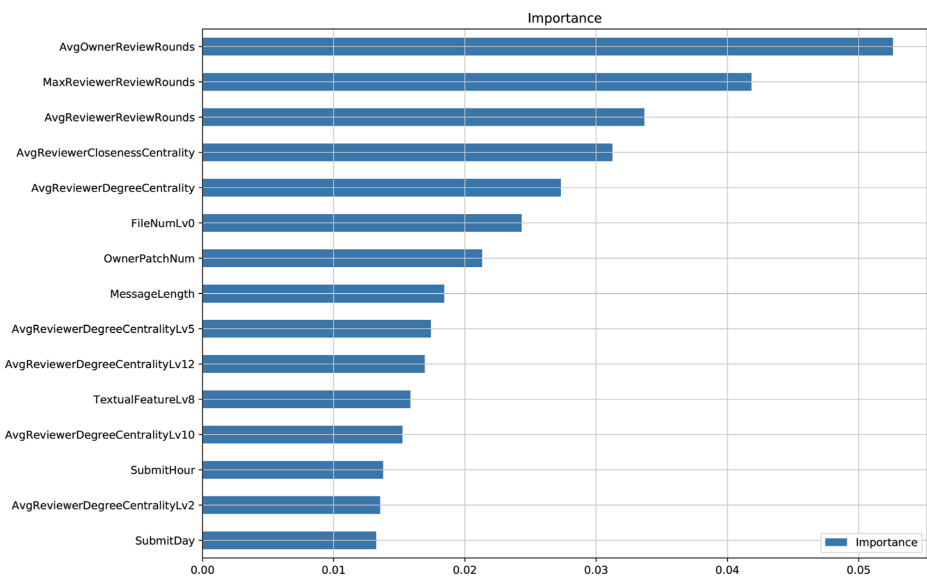


Fig. 7 The top 15 most important features

Another worth mentioning features are *MaxReviewerReviewRounds* and *AvgReviewer-ReviewRounds*. As the name suggests, *AvgReviewerReviewRounds* shows the max reviewing rounds in all the patches of a reviewer, which reflects the upper limit of the reviewing rounds by a reviewer. *AvgReviewerReviewRounds* represents the average reviewing rounds for all the patches of a reviewer in the dataset.

Meanwhile, the network-based PEF also play an import role in determining the reviewing rounds, such as *AvgReviewerClosenessCentrality* and *AvgReviewerDegreeCentrality*. These two features represent the degree of collaboration between the reviewers.

In addition, a patch meta feature is found in the 15 most important features, e.g., *MessageLength*. *MessageLength* usually describes how a bug is fixed or a software functionality is implemented, then the message length may reflect the priority level and scale of a patch, and further determines the reviewing round. More reasons that make the feature significant will be discussed in Section 7.2.

Therefore, based on the results we observed, we can draw a conclusion that the review efficiency of a patch depends heavily on factors of participants.

5.3.4 RQ4: Can PMCost be generalized in a cross-project scenario?

In our experiment setting in RQ1, for each project, we train a model by leveraging the historical patches with known reviewing round within the project (i.e., within-projects scenario). 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 scenario.

To do a cross-project evaluation, we build three generalization modes. For same-language-cross, we first train a Random Forest model based on the dataset of OpenDaylight and then apply it to Eclipse, while the two projects are both implemented by Java. For language-cross, we also train a model on the dataset of OpenDaylight and then apply the model to the test set of OpenStack, and OpenStack is based on Python. For union-training, we first equally and randomly split the 3 projects into training set and testing set. Next, we train a model based on the training set and then apply it to the testing set. Meanwhile, we also apply the union-training model to the three projects for testing, respectively.

The results are shown in Table 9. *PMCost* achieves a lower overall F1 in the same-language-cross evaluation. In the language-cross evaluation, *PMCost* shows a higher F1 values on the one-round, short-rounds and long-rounds reviewing patches compared with the ones in the same-language-cross scenario. However, the performance of *PMCost* in both same-language-cross and language-cross scenarios are inferior to ones in within-projects scenarios. The reason may be that different projects have their special collaborating and maintaining schemes and it is difficult to generalize them from one project to another.

Besides, we can observe that all the union evaluations show decline in overall F1 compared with the ones in same-language-cross and language-cross scenarios, and the F1 values on the datasets of Eclipse, OpenDaylight and OpenStack are 28.40%, 34.81% and 27.41%, respectively. Considering the training set in the union evaluation includes the patches coming from three different projects, and training a model on such union dataset will be much more difficult than the one in a within-projects scenario.

Therefore, based on the results we observed, we can draw a conclusion that the performance of *PMCost* shows some degradation in the same-language-cross, language-cross and union training scenarios.

Table 9 Generalization evaluation

Train	Test	One-round reviewing			Short-rounds reviewing			Long-rounds reviewing			Overall	
		P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
OpenDaylight	Eclipse	63.82	58.42	61.00	27.45	20.96	23.77	2.67	32.05	4.93	51.85	29.90
OpenDaylight	OpenStack	51.04	87.12	64.37	40.44	23.57	29.79	16.47	10.44	12.78	64.46	35.65
Union	Eclipse	62.92	99.12	76.97	23.82	2.75	4.93	10.69	1.98	3.34	74.28	28.40
	OpenDaylight	51.11	99.92	67.63	46.09	2.09	4.00	18.92	1.51	2.79	67.21	24.81
	OpenStack	52.05	99.98	68.46	46.68	6.51	11.42	16.34	1.26	2.34	67.74	27.41

6 Model Limitation and Solution

Our findings in RQ3 reveal that owner's experience (e.g., *OwerAvgMergedRounds*, *OwerPatchNum*) is one of the most important factors to determine the reviewing round of a patch. It indicates that *PMCost* may have a bias against new developers. The information gain score reflects the contribution of each feature to the classification in the datasets, the higher the value, the greater the contribution. Therefore, features *AvgOwnerReviewRounds* and *OwnerPatchNum* play important role in the reviewing round prediction. In general, the new developers have less experience for submitting patches, and there will be very little data available to represent the developers experience. As a result, the important feature (e.g., *OwnerPatchNum*) will play a diminished role in prediction. In this section, we first investigate whether *PMCost* has such a shortcoming. Then, we investigate how to overcome this possible shortcoming of *PMCost*.

To investigate whether *PMCost* has a bias against new developers, we need to evaluate the performance of our approach on the patches submitted by new developers. To do this, we randomly use the 80% of dataset as training set to build model, and the rest of dataset (i.e., 20%) as test set. Then, we will check the testing set that how many correct prediction coming from the patches owned by new developers and how many correct prediction coming from the patches owned by experienced developers. To determine the patches owned by new or experienced developers, we rank the owners in the test set in a descending order according to the number of patches they submitted, and then we divide the test set into set 1 and set 2. We add the patches from the owners who submitted the least patches to the set 1 until the number of patches reaches 50% of the total test set. The remaining 50% of the patches are added into set 2. We consider the patches in set 1 as the ones submitted by new developers. At last, we calculate the accuracy of the patches in set 1.

Table 10 shows the performance of *PMCost* for the patches owned by new developers. We can observe that *PMCost* achieves a F1 of only 41.24%, 40.80% and 33.42% on the Eclipse, OpenDaylight and OpenStack, respectively. As shown in Table 7, for all patches (submitted by new and experienced developers), *PMCost* achieves a F1 of 54.12%, 47.92% and 48.44% by using Random Forest algorithm on the Eclipse, OpenDaylight and OpenStack datasets respectively. Thus, *PMCost* achieves a much lower F1 score for new developers' patches compared to all patches. It means that *PMCost* inclines to incorrectly predict the labels of new developers' patches.

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

- Model 1: From the above analysis, the features in the owner experience dimensions cause the bias of our approach against new developers. To deal with the bias, we can remove these features and train a model based on the remaining features. To do this, we first remove the owner related features from PEF in the training and testing dataset and we build the prediction model on the training dataset. Then, we use the model to predict the labels of new developers' patches in the testing dataset. Table 11 presents precision, recall and F1 of the model for new developers' patches. By comparing the results shown in Tables 10, this model achieves a higher ACC and F1 for the three datasets. This method thus achieves acceptable performance for the new developers' patches.
- Model 2: We notice that half of the patches in the test set are regarded as the new developers submitted patches. We conjecture that the training set induces the bias of *PMCost* against new developers. To verify our conjecture, we only use the new developers' patches to build the prediction model. Then, we use this model to predict the

Table 10 The performance of *PMCost* owned by new developers

Projects	one-round reviewing			short-rounds reviewing			long-rounds reviewing			overall	
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	64.29	78.80	70.71	58.43	30.47	39.16	16.90	12.70	13.85	74.73	41.24
OpenDaylight	59.66	80.66	68.16	57.77	20.14	28.35	49.38	19.12	25.90	71.63	40.80
OpenStack	50.90	81.36	62.21	43.81	8.45	13.97	57.38	17.71	24.09	67.31	33.42

Table 11 The performance of *PMCost* built on the features except PEF

Projects	one-round reviewing			short-rounds reviewing			long-rounds reviewing			overall	
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	81.65	50.59	61.96	56.36	58.95	56.57	19.31	19.93	19.21	75.13	45.91
OpenDaylight	74.39	49.80	59.33	54.38	31.43	39.60	34.05	21.27	25.33	72.28	41.42
OpenStack	76.39	38.44	50.67	44.50	21.89	29.18	39.87	23.40	28.94	70.82	36.26

labels of new developers' patches in testing dataset. As shown in Table 12, the model achieves F1 values of 43.63%, 36.70% and 36.76% on the Eclipse, OpenDaylightand, OpenStack, which shows a slight improvements on datasets of Eclipse and OpenStack when comparing with the ones in Table 10.

By comparing the results of the two models, we choose the first model to predict the reviewing round for new developers' patches. Then, to deal with the bias of our approach against new developers, we retain two models to predict the reviewing round, and one is the model trained using all features and the another built on the features except PEF. When applying our models in the practice, we choose one of the model to predict the reviewing round, i.e., if a patch is submitted by a new developer (determining by the number of the patches he/she submitted), we use the model built on the features except PEF. Otherwise, we use the model built on all features.

7 Discussion

7.1 Feature Selection and Imbalanced Dataset

In this paper, we try to select the useful features. Inspired by Shivkumar et al. (Shivaji et al. 2013), we use the information gain to select a subset of features that are useful for making prediction. The information gain score reflects the contribution of each feature to the classification in the datasets. If the information gain score of a feature is greater than 0, it indicates that this feature plays a positive role in the classification. After calculating the information gain score of each feature, we found that the information gain scores of 16 features are less than or equal to 0. Then, we filter out these 16 features. Table 13 shows the performance of *PMCost* built on total features and selected features (after filtering the 16 features). We can observe a performance improvement when *PMCost* builds on the selected features. Then, we do all the experiments by applying the selected features (i.e., after removing the 16 features).

We divide the reviewing rounds into three classes, i.e., one-round reviewing, short-rounds reviewing and long-rounds reviewing. As shown in Table 1, each classes in the dataset has different number. Therefore, to balance the instances in each class, we employed SMOTE (Chawla et al. 2002) to create synthetic instances. As a result, the three classes (i.e., one-round, short-rounds and long-rounds patches) had an equal number of data instances in the training set. It is worth noting that the proportion of the three types of instances in the test set always remained unchanged. Figure 8 shows the overall F1 values before and after applying the SMOTE on the three datasets by using the Random Forest algorithm. We observed that the overall F1 values after applying SMOTE are higher than that of before applying SMOTE, and this indicates that using SMOTE to deal with imbalanced dataset can improve the performance of the proposed model. Then, we do all the experiments by applying the SMOTE to balance the classes.

7.2 Significant Features Analysis

To understand why these features have strong discriminability in Section 5.3.3, we further analyze the reasons that make them discriminative. We notice that the feature *Message-Length* is included in the 15 most important features, and we analyze the reasons that make

Table 12 The performance of *PMCost* built on the patches submitted by new developers

Project	one-round reviewing			short-rounds reviewing			long-rounds reviewing			overall	
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	65.83	83.00	73.22	67.74	26.58	36.37	94.87	15.55	21.29	76.15	43.63
Opendaylight	58.28	86.90	66.76	61.43	18.35	26.76	89.03	11.22	16.57	69.58	36.70
OpenStack	54.28	80.27	61.89	58.50	13.06	19.35	88.88	23.17	29.03	67.22	36.76

Table 13 The performance of *PMCost* built on total and selected features

Projects		One-round reviewing			Short-rounds reviewing			Long-rounds reviewing			Overall	
		P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	ACC (%)	F1 (%)
Eclipse	total	77.77	60.19	67.25	51.00	52.79	51.66	20.29	24.59	21.67	76.73	46.86
	selected	79.60	67.08	72.56	57.40	54.05	55.57	25.85	23.03	23.82	79.83	50.65
OpenDaylight	total	62.16	60.84	61.10	53.26	33.03	39.74	30.11	29.18	28.42	70.14	43.09
	selected	69.45	62.77	65.21	55.74	35.18	42.57	38.76	26.28	29.61	72.97	45.80
OpenStack	total	68.87	44.72	53.96	42.03	35.88	38.34	32.49	22.80	26.47	68.78	39.59
	selected	70.64	54.75	61.19	47.51	25.70	33.24	41.04	28.35	32.27	71.81	42.23

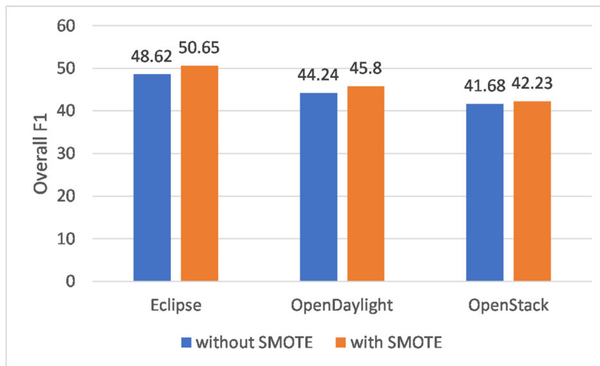


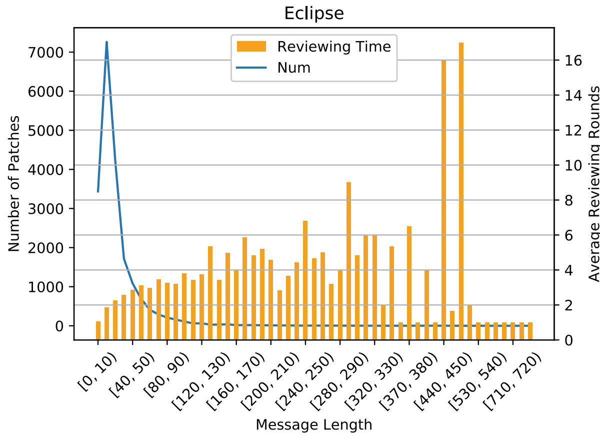
Fig. 8 The overall F1 values before and after applying the SMOTE on the three datasets

it significant. Figure 9 shows the average reviewing round of the patches with different message length in the three projects. The message length represents the number of the words contained in a patch. We use 10 words to build the interval, e.g., (40,50] means that the number of the words contained in a patch is greater than 40 and less than or equal to 50. In the project Eclipse, the number of the words contained in the patches ranges from 0 to 720. We can see from Fig. 9 (a) that more than 99% of the messages are less than 160 words. Meanwhile, there is no patch located in some intervals, such as the interval [360, 370). We also notice that the patches with different message length have different average reviewing rounds, e.g., the average reviewing round of the patches whose message length locates in the interval of [130, 140) is 5.36, while the ones whose message length locates in the interval of [210, 220) is 2.83. Obviously, the average reviewing rounds for the patches with different message length has significant difference, hence feature *MessageLength* is very informative for the reviewing round prediction.

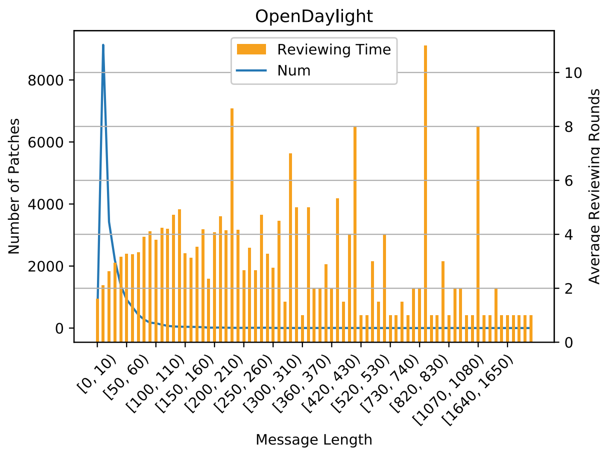
Figure 9 (b) show the average reviewing round of the patches with different message length in the project OpenDaylight. We notice that the message length of some patches is larger than 500 words, and these patch messages usually describe the complete process of how a bug occurs, as the example shown in patch 24976[4] in OpenDaylight. Meanwhile, we can also observe that the average reviewing rounds for the patches with different message length is different, which is consistent with one in Fig. 9 (a). Figure 9 (c) shows the average reviewing round of the patches with different message length in the project OpenStack. The message length of the patches locate in the interval of [850, 860) need the longest average reviewing round, i.e., 11, while the one locates in the interval of [520, 530), [670, 680), [740, 750) need the shortest average reviewing round, i.e., 1. Therefore, the variations of the average reviewing round can be observed in the three projects, which make the feature *MessageLength* be discriminative.

We learn about why *MessageLength* is so discriminative, while we don't know whether a patch with longer message or shorter message is more likely to be quickly reviewed. Then, we analyze the relationship between the reviewing rounds and the length of the patch message. We found that the patches having longer messages tend to be reviewed longer, as shown in Fig. 10. So, to speed up patch review, we can give a suggestion to the developers to avoid writing a longer and redundant patch message.

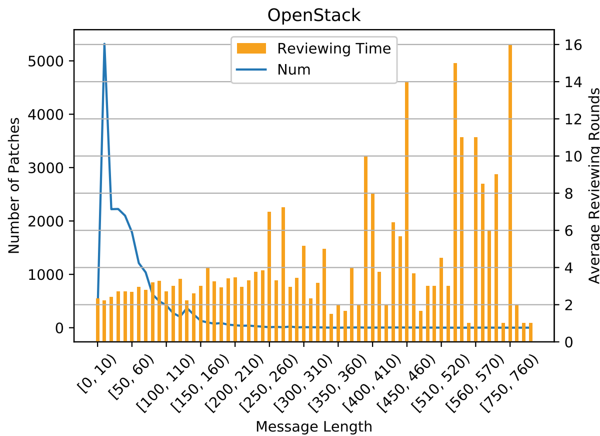
[4]https://git.opendaylight.org/gerrit/c/neutron/+/24976/1



(a) Average reviewing rounds of the patches with different message length in Eclipse



(b) Average reviewing rounds of the patches with different message length in OpenDaylight



(c) Average reviewing rounds of the patches with different message length in OpenStack

Fig. 9 Average reviewing rounds of the patches with different message length in the three projects

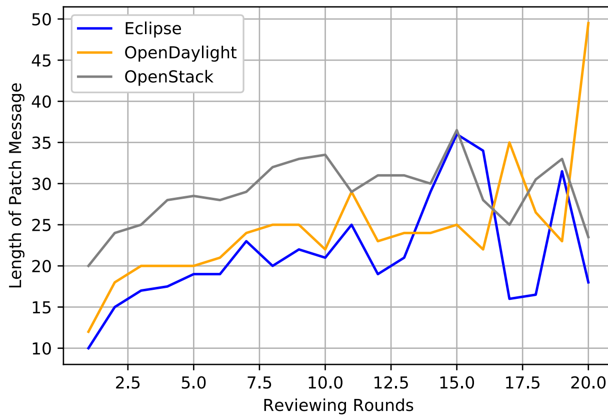


Fig. 10 The reviewing round of the patches with different message length

In addition, we observe from Fig. 7 that the most significant feature is *AvgOwnerReviewRounds*, which is used to character the average rounds for all the patches of a developer in the dataset. To further explore the implicit information reflected by this feature, we analyze the developers with most average reviewing rounds (called top developers) and the ones with least average reviewing rounds (called bottom developers). To make a fair comparison, we select the 100 top developers with the most average reviewing rounds and 100 bottom developers with least average reviewing rounds, respectively. Then, for the 100 top and bottom developers, we count the average Java files, code lines, and changed code lines involved by their patches, respectively. Table 14 shows the difference between the patch scale owned by the top and bottom 100 developers. We can observe that the average reviewing rounds shows the significant difference in the three datasets. For example, the average reviewing rounds of the top 100 developers are 17.09 on dataset OpenStack, while this number is 1 for the bottom 100 developers. Meanwhile, we also observe that the average Java files and the average code lines involved in the patches owned by the top and bottom 100 developers are significantly different. Besides, the average changed code lines in the patches owned by the top 100 developers are also greater than the ones of the bottom 100 developers. Therefore, we can conclude that the patch scale can directly affect its reviewing rounds, and lager patches have more code and require more rounds for reviewing by the reviewers. Hence, to speed up patch reviewing, we recommend that programmers patch logically unrelated changes separately as much as possible to reduce the size of a single patch.

Table 14 The patch scale owned by the top and bottom 100 developers

	Developers	Avg Reviewing Rounds	Avg Java files	Avg Code lines	Avg Changed code lines
Eclipse	Top 100	5.68	80.6	679.46	85.45
	Bottom 100	1	10.09	331.04	81.85
OpenDaylight	Top 100	8.47	12.2	237.39	49.01
	Bottom 100	1	5.81	134.64	48.8
OpenStack	Top 100	17.09	7.11	813.05	98.79
	Bottom 100	1	3.37	447.75	59.6

We also study the cyclomatic complexity (Gill and Kemerer 1991) of the patches owned by the top and bottom 100 developers. Specifically, we calculated the cyclomatic complexity of the Java files in each patch, then calculated the average cyclomatic complexity of each patch. The results in Table 15 show that the patches with more reviewing rounds means more code complexity. It is common sense that the more complex the code, the more rounds it takes to understand and review by the reviewers. We can also give some suggestions that developers can reduce the number of patch reviewing rounds by writing less complex code, e.g., using less nesting of loops or conditional statements.

7.3 Actual Reviewing Round Prediction

We try to train a model to predict the actual reviewing round. Specifically, we firstly collect the actual reviewing round of each patch. Then, since each patch has an actual reviewing round, we treat the prediction of actual reviewing round as a regression problem. We employ 5 learning-based regression models (including: SVR (Chang and Lin 2011), MLP (Hinton 1989), DT (Li et al. 1984), RF (Breiman 2001b), LGB (Ke et al. 2017b)) for the prediction on the 3 datasets, i.e., Eclipse, OpenDaylight, and OpenStack. To measure the effectiveness of the regression models, we employ MSE, MAE and R2 to evaluate the results. MSE and MAE are mean squared error and mean absolute error, which are measure of errors, and the optimization goal of the model is to make it as small as possible until zero. R2 is the coefficient of determination, the proportion of the variance in the dependent variable that is predictable from the independent variable. The closer the value of R2 is to 1, the better the model fits the data.

Table 16 shows that LGB achieves the better results in predicting the actual reviewing round in the 3 datasets when comparing with other four models. Even so, the MSE value achieved by LGB is much bigger than 0 and the R2 value is much less than 1, indicating that LGB shows a poor performance in the task of the actual reviewing round prediction. So, all these regression models achieve poor performance in the task of the actual reviewing round prediction on the 3 datasets.

In this paper, we use three intervals to represent the reviewing round span, i.e., patches with 1 reviewing round, patches with 2 to 6 reviewing rounds, patches with more than 6 reviewing rounds. Defining the intervals of reviewing rounds has following advantage. It can avoid to give the inaccurate prediction results (obtained by regression models) to developers. Since the regression models achieve poor performance when predicting the actual reviewing round, in this case, it is better to give the developers a time span than an inexact reviewing round. Therefore, we predict the intervals of reviewing rounds instead of the actual reviewing round in this paper.

Table 15 The cyclomatic complexity of the patches owned by the top and bottom 100 developers

	Developers	Avg Cyclomatic Complexity
Eclipse	Top 100	87.01
	Bottom 100	26.53
OpenDaylight	Top 100	51.43
	Bottom 100	11.21
OpenStack	Top 100	112.65
	Bottom 100	62.37

Table 16 Performance of regression models

Projects	Algorithm	MSE	MAE	R2
Eclipse	SVR	9.23	0.38	−0.67
	MLP	64.48	1.42	−5.65
	DT	153.40	1.46	−29.12
	RF	68.79	1.46	−6.51
	LGB	4.83	0.17	−0.10
OpenDaylight	SVR	612.00	3.96	−0.73
	MLP	625.70	5.33	−1.05
	DT	861.20	5.15	−2.47
	RF	571.80	4.97	−1.15
	LGB	79.66	0.88	−0.14
OpenStack	SVR	45.90	1.23	−1.15
	MLP	60.33	1.83	−4.10
	DT	100.90	1.75	−9.12
	RF	58.82	1.77	−3.11
	LGB	6.91	0.33	−0.55

7.4 Prediction for Reviews with Multiple Rounds

As we can see in Fig. 1, 70% of the reviews take about 1 or 2 rounds. When we pay attention to the actual reviewing time (days) of the patches only needing 1 or 2 reviewing rounds, they represent only a small fraction of the reviewer's total review time, as shown in Fig. 5, while the reviews with multiple rounds cost a lot of time, i.e., most of the patches with 1-2 reviewing rounds cost less than 7 days, while most of the patches with more than 2 reviewing rounds span 7 to 120 days reviewing time. Therefore, from a reviewer's point of view, patches with multiple rounds are important to them, as they take up the majority of their total reviewing time.

Then, if a patch requires long-rounds for reviewing determined by our prediction method, we can tell the developers to do two things ahead of time. Firstly, developers take self-inspection on the patch to found what reasons make the patch need so much time for reviewing. Secondly, if a long-rounds reviewing is inevitable, developers can plan other software development activities ahead of time instead of waiting for this patch to be reviewed. Therefore, it is necessary to predict the number of reviewing rounds as well as the patches with multiple reviewing rounds.

7.5 Can Reviewing Rounds Reflect Reviewing Effort?

We found that some patches have less reviewing rounds, but takes a very long time to review. For example, in Fig. 11 a patch [5] coming from Eclipse has only 3 reviewing rounds, but it takes about one years to finish the reviewing. We can see from the change log that the first patch starts on Mar 08, 2019, and the last patch is on Mar 04, 2020. In this case, the discussion and the reviewing rounds are not proportional to each other.

[5] https://git.eclipse.org/r/c/scout/org.eclipse.scout.rt/+/138383

Change Log | Comment Threads

Only comments EXPAND ALL

Beat Schwarzentrub	Uploaded patch set 1.	Patchset 1	Mar 08, 2019 20:14
CI Bot	Build Started https://ci.eclipse.org/scout/job/org.eclipse.scout.rt_5_2_and_higher_gerrit/4166/	Patchset 1	Mar 08, 2019 20:14
CI Bot	Verified +1 Build Successful https://ci.eclipse.org/scout/job/org.eclipse.scout.rt_5_2_and_higher...	Patchset 1	Mar 08, 2019 20:25
Beat Schwarzentrub	Uploaded patch set 2.	Patchset 2	Mar 04, 2020 00:38
CI Bot	Build Started https://ci.eclipse.org/scout/job/org.eclipse.scout.rt_5_2_and_higher_gerrit/5332/	Patchset 2	Mar 04, 2020 00:38
CI Bot	Added to reviewer: CI Bot		Mar 04, 2020 00:50
CI Bot	Verified +1 Build Successful https://ci.eclipse.org/scout/job/org.eclipse.scout.rt_5_2_and_higher...	Patchset 2	Mar 04, 2020 00:50
Beat Schwarzentrub	Code-Review +2	Patchset 2	Mar 04, 2020 01:05
Beat Schwarzentrub	Change has been successfully cherry-picked as fa24d859aa0d43d5c32a6d164c644179b23c3198 by Bea...	Patchset 3	Mar 04, 2020 01:05

Fig. 11 A example of patch with less reviewing rounds but long reviewing time

However, in most cases the number of reviewing rounds and the reviewing effort are roughly positive correlated. Figure 12 shows the distributions between the reviewing rounds and the reviewing effort on the 3 datasets. The vertical axis in these figures are the number of the messages (they come out of a discussion between developers and reviewers, as shown in Fig. 11) in the discussions, which reflects the actual reviewing effort cost in the reviewing process.

To investigate correlation between the number of reviewing rounds and the reviewing effort, we apply the Spearman Correlation Coefficient. The results in Table 17 shows that the Spearman Correlation Coefficient of three projects Eclipse, OpenDaylight and OpenStack are 0.775, 0.834 and 0.788, respectively, which means that there is indeed a positive linear correlation between the number of reviewing rounds and the discussion rounds. From this perspective, the number of rounds can reflect the actual reviewing effort implied in the discussions.

7.6 Time Constraints of Training and Test Sets

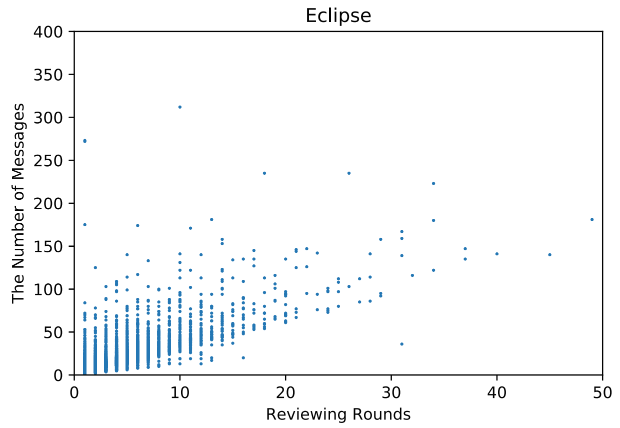
In the experiment, we employ a 5-fold cross validation metric. To avoid the future patches are used to predict past patches, we follow time constraints in the 5-fold cross validation. To achieve this goal, we sort the patches by their submission time through using the Time-SeriesPlit of Scikit-Learn (Cerqueira et al. 2020). Scikit-Learn is designed to deal with time series data by the related studies (Cerqueira et al. 2020).

Then, we employ TimeSeriesPlit to divide the dataset into 6 parts, from 0 to 5, as following shows. In the first time, the part 0 is used as training set, and part 1 is used as test set. At the second time, the parts 0 and 1 is used as training set, and part 2 is used as test set. By analogy, we can guarantee that the future patches cannot use to predict past patches at each time.

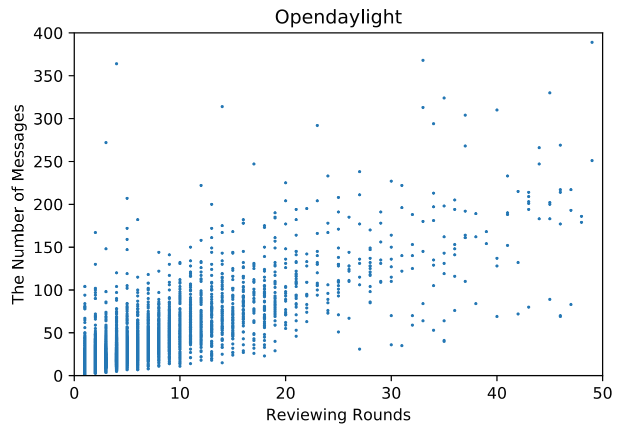
Case 1: TRAIN: [0] TEST: [1]
Case 2: TRAIN: [0 1] TEST: [2]
Case 3: TRAIN: [0 1 2] TEST: [3]
Case 4: TRAIN: [0 1 2 3] TEST: [4]
Case 5: TRAIN: [0 1 2 3 4] TEST: [5]

When we construct the network-based features, we also consider the time constraints to guarantee that the future data cannot be used to predict past data. Taking Case 1 as an example, when we construct the network-based features for the patches in the training set,

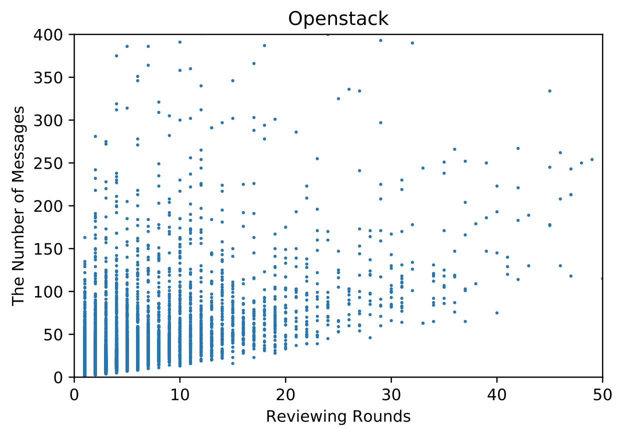
Fig. 12 The distribution of reviewing rounds and discussion time in the three projects



(a) Eclipse



(b) Opendaylight



(c) Openstack

Table 17 Spearman correlation coefficient on the three datasets

Datasets	Spearman Correlation Coefficient
Eclipse	0.775
Opendaylight	0.834
Openstack	0.788

we only take the owners and reviewers appearing together in a patch or folder in ”TRAIN: [0]”, and the cases that they appear together in a patch or folder in the ”TEST: [1]” are not considered. Thus, we can guarantee that the future data cannot be used to predict past data.

8 Related Work

Studies on factors impacting code review A large body of work has qualitatively analyzed the modern code review process. Kononenko et al. (2015) explored a set of factors impacting code review quality from the personal and social perspectives. Their findings showed that both personal factors (such as reviewer workload and experience) and participation factors (such as the number of involved developers) are associated with the quality of the code review. Thongtanunam et al. (2017) took a case study of 196,712 reviews spread across the open-source projects of Android, Qt and OpenStack and found that the amount of review participation in the past is a significant indicator of patches that will suffer from poor review participation. Moreover, the length of a patch message shares a relationship with the likelihood of receiving poor reviewer participation or discussion. Baysal et al. (2013) investigated the influence of non-technical factors on code review. They described an empirical study of the code review process for WebKit and their findings suggested that nontechnical factors can significantly impact code reviews, such as patch size, priority and component. Rigby et al. (2014) studied on six large and mature OSS (i.e., Open Source Software) projects to make an empirical understanding of OSS peer review. They found that OSS peer reviews are conducted asynchronously by empowered experts who focus on changes that are in their area of expertise. Meanwhile, they also found that reviewers trend to provide timely, regular feedback on small changes, and the OSS review is drastically different from traditional inspection. These research results inspired us to consider multiple influence factors (such as: reviewer experience, patch message, patch size, etc) in predicting the reviewing rounds in code review.

Studies on prediction of code patches There are many studies that focus on predicting the merged code patches. Fan et al. (2018) proposed an approach to predict whether a patch will eventually get merged or not. They extracted 34 features to characterize a code patch. Some of their proposed features are same with us, while some are different, such as patch meta feature, reviewer experience, etc. In the experiment, Fan et al. employed Random Forest to build a prediction model and performed evaluation on three open source projects containing a total of 166,215 patches. Jeong et al. (2009) 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. 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. Gousios et al.

(2014) 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. Their study focused on pull-requests while ours focuses on code patches coming from Gerrit. There are two obvious differences between our study and the previous ones. First, we try to predict how long a patch will be reviewed, while existing studies try to determine whether a patch is eventually merged or abandoned. Second, we model the reviewing round prediction as triple-classification problem, i.e., one-round reviewing , short-rounds reviewing, long-rounds reviewing, while the existing studies model their problem as binary-classification, i.e., merged or abandoned.

Kikas et al. (2016) proposed a method to predict whether or not a pull requests issue will close after a given time horizon. The proposed method is different from the existing ones that it considers the dynamic features evolving throughout an issue's lifetime. van der Veen et al. (2015) applied the issue lifetime prediction to the pull request prioritization. They trained a machine learning-based model to predict the probability of current pull requests will receive user updates in the following time window. They thought the pull requests with the highest probability should be ranked at the top of the integrated list of the pull requests. The difference between these studies and our study is that: we try to predict how long a patch will be reviewed after submitting, while these studies tries to predict whether a pull requests will close at a given time window.

Studies on prediction of reopened bugs Theoretically, the reviewing round prediction is somewhat similar to the issue of reopened bugs prediction. Generally, bugs are reported, fixed, verified and closed. However, in some cases, bugs have to be re-opened (Shihab et al. 2013). For example, the unclear description given by the bug reporter, the developer misunderstood the root cause of the bug, the bug reappeared in the current version of the system although it was fixed in the previous system (Xia et al. 2013). Similarly, if a patch submitted by developers is not accepted by reviewers, the patch will be sent back to developers. In this sense, the patch is "reopened" to developers. Many researchers have pay attention to the issue of reopened bug such as Xia et al. (2013), they presented a high-level view of how to use learning algorithms to predict reopened bug reports. Later, Xia et al. (2015) proposed ReopenPredictor, which is an automatic predictor of reopened bugs. ReopenPredictor uses a number of features, such as textual features, to achieve high accuracy prediction of reopened bugs. In addition, they proposed two algorithms that are used to automatically estimate various thresholds to maximize prediction performance. Mi and Keung (2016) drew a conclusion that it is quite possible to reduce the bug reopening rate through the adoption of appropriate methods, such as promoting effective and efficient communication among bug reporters and developers. Souza et al. (2013) provided another idea that if rapid release cycles impact the bug reopening rate. Their result showed that the bug reopening rate of versions developed in rapid cycles was about 7% higher.

9 Threats to Validity

In this section we focus on the threats that could affect the results of our case studies. The main threat to validity is the scale of the dataset. Since we need to extract reviewing data from the open-source projects, it requires the reviewing data in the selected projects should be public. We have collected 3 projects from Gerrit, and these projects include more than ten thousand patches in total. In the future, we need to collect more high-quality projects to extend our repository.

Another threat to validity is the suitability of our evaluation measure. We use a conventional measure to evaluate the effectiveness of the proposed approach in this paper. Because the issue in this study can be modeled as a multi-classification, we introduce the precision, recall, accuracy, F-measure to evaluate the effectiveness of our method. Meanwhile, we compare the performances of our method with different machine learning models. All these metrics can evaluate the effectiveness of our method. Thus, we believe there is little threat to suitability of our evaluation measure.

The last threat to validity is the generalizability of our model. Our method is used to predict the reviewing round. The selected projects are written by Java and Python language. The features we designed in this study is programming language-independent. Therefore, when applying our approach to the projects written by other programming languages, such as C++, there is little threat to generalizability of our model.

10 Conclusion and Future Work

In this paper, we aim to construct a machine learning based model to predict the reviewing round that a newly submitted patch takes to be reviewed. We formulate this problem as a triple-classification task. We extract 4 types of features such as patch meta features, code diff features, personal experience features and textual features, and input them into Random Forest to train a triple-classes prediction model. Comprehensive experiments on three open-source projects show that our approach outperforms baselines that based on other machine learning algorithms. Our comparative experiments and analysis show that some factors can affect the reviewing round significantly, such as the experience of participants. The future research agenda mainly focuses on the accuracy improvement of our approach. For example, taking more dimensions of features relating to code changes or review into consideration.

Acknowledgments The work is supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164002), the National Natural Science Foundation of China (Nos. 61902441, 61902105, 62032025), HK RGC Project (No. PolyU 152239/18E), Guangdong Basic and Applied Basic Research Foundation(No. 2020A-1515010973), the Fundamental Research Funds for the Central Universities (No. 20lgpy129). Xiangping Chen is the corresponding author.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

(2008) Spearman rank correlation coefficient. In: The concise encyclopedia of statistics. Springer New York, New York, pp 502–505. https://doi.org/10.1007/978-0-387-32833-1_379

Baum T, Schneider K, Bacchelli A (2019) Associating working memory capacity and code change ordering with code review performance. Empir Softw Eng 24(4):1762–1798

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

Bollobas B (1998) Modern graph theory. Graduate Texts in Mathematics 184
Breiman L (2001a) Random forests. Mach Learn 45(1):5–32
Breiman L (2001b) Random forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324
Cerqueira V, Torgo L, Mozeti I (2020) Evaluating time series forecasting models: an empirical study on performance estimation methods. Mach Learn 109(11):1–32
Chang CC, Lin CJ (2011) Libsvm: A library for support vector machines. ACM Trans Intell Syst Technol 2(3). https://doi.org/10.1145/1961189.1961199
Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: Synthetic minority over-sampling technique. J Artif Int Res 16(1):321–357
Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297
Eyolfson J, Tan L, Lam P (2011) Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th working conference on mining software repositories. ACM, pp 153–162
Fagan M (2002) Design and code inspections to reduce errors in program development. In: Software pioneers. Springer, pp 575–607
Fan Y, Xia X, Lo D, Li S (2018) Early prediction of merged code changes to prioritize reviewing tasks. Empir Softw Eng 23(6):3346–3393. https://doi.org/10.1007/s10664-018-9602-0
Gill GK, Kemerer CF (1991) Cyclomatic complexity density and software maintenance productivity. IEEE Trans Softw Eng
Gousios G, Pinzger M, Av Deursen (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International conference on software engineering, ICSE 2014. Association for Computing Machinery, New York, pp 345-355. https://doi.org/10.1145/2568225.2568260
Hinton GE (1989) Connectionist learning procedures. Artif Intell 40(1–3):185–234. https://doi.org/10.1016/0004-3702(89)90049-0
Huang Y, Zheng Q, Chen X, Xiong Y, Liu Z, Luo X (2017) Mining version control system for automatically generating commit comment. In: 2017 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM). pp 414–423. https://doi.org/10.1109/ESEM.2017.56
Huang Y, Jia N, Chen X, Hong K, Zheng Z (2018) Salient-class location: Help developers understand code change in code review. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ACM, pp 770–774
Huang Y, Hu X, Jia N, Chen X, Xiong Y, Zheng Z (2020) Learning code context information to predict comment locations. IEEE Trans Reliab 69(1):88–105. https://doi.org/10.1109/TR.2019.2931725
Huang Y, Jiang J, Luo X, Chen X, Huang G (2021) Change-patterns mapping: A boosting way for change impact analysis. IEEE Trans Softw Eng PP(99):1–1
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 Y, Adams B, German DM (2013a) 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 2013). IEEE Computer Society, Los Alamitos, pp 101–110. https://doi.org/10.1109/MSR.2013.6624016
Jiang Y, Adams B, German DM (2013b) Will my patch make it? and how fast?: Case study on the linux kernel. In: Proceedings of the 10th working conference on mining software repositories. IEEE Press, pp 101–110
Kagdi H, Hammad M, Maletic JI (2008) Who can help me with this source code change? In: 2008 IEEE international conference on software maintenance. IEEE, pp 157–166
Kagdi H, Gethers M, Poshyvanyk D (2013) Integrating conceptual and logical couplings for change impact analysis in software, vol 18, pp 933–969. https://doi.org/10.1007/s10664-012-9233-9
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773
Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu TY (2017a) Lightgbm: A highly efficient gradient boosting decision tree. In: Advances in neural information processing systems. pp 3146–3154
Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu TY (2017b) Lightgbm: A highly efficient gradient boosting decision tree. In: Proceedings of the 31st international conference on neural information processing systems, NIPS'17. Curran Associates Inc., Red Hook, pp 3149–3157
Kikas R, Dumas M, Pfahl D (2016) Using dynamic and contextual features to predict issue lifetime in github projects. In: Proceedings of the 13th international conference on mining software repositories, MSR '16. Association for Computing Machinery, New York. pp 291–302. https://doi.org/10.1145/2901739.2901751
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

Li B, Friedman J, Olshen R, Stone C (1984) Classification and regression trees (CART). vol 40. https://doi.org/10.2307/2530946
Liu Z, Xia X, Treude C, Lo D, Li S (2019) Automatic generation of pull request descriptions, IEEE
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
McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empir Softw Eng 21(5):2146–2189
Mi Q, Keung J (2016) An empirical analysis of reopened bugs based on open source projects. In: Proceedings of the 20th international conference on evaluation and assessment in software engineering. ACM, p 37
Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th international conference on neural information processing systems, NIPS'13. Curran Associates Inc., pp 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959
Pal SK, Mitra S (1992) Multilayer perceptron, fuzzy sets, and classification. IEEE Trans Neural Netw 3(5):683
Pregibon D et al (1981) Logistic regression diagnostics. Ann Stat 9(4):705–724
Quinlan JR (1987) Simplifying decision trees. Int J Man-mach Stud 27(3):221–234
Rigby PC, Bird C (2013) Convergent software peer review practices. In: Proceedings of the the joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). ACM. https://www.microsoft.com/en-us/research/publication/convergent-software-peer-review-practices/, preprint available upon request to cbird@microsoft.com or peter.rigby@concordia.ca
Rigby PC, Storey M (2011) Understanding broadcast based peer review on open source software projects. In: 2011 33rd international conference on software engineering (ICSE). pp 541–550
Rigby PC, German DM, Cowen L, Storey MA (2014) Peer review on open-source software projects: Parameters, statistical models, and theory. ACM Trans Softw Eng Methodol (TOSEM) 23(4):35.1–35.33
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Ki Matsumoto (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042
Shivaji S, Whitehead EJ, Akella R, Kim S (2013) Reducing features to improve code change-based bug prediction. IEEE Trans Softw Eng 39(4):552–569. https://doi.org/10.1109/TSE.2012.43
Smith M, Shneiderman B, Milic-Frayling N, Rodrigues E, Barash V, Dunne C (2011) Analyzing (social media) networks with nodexl. In: Proceedings on communities and technologies 2009, pp 255–264. https://doi.org/10.1002/9781118257463.ch9
Souza R, Chavez C, Bittencourt R (2013) Patterns for cleaning up bug data. In: 2013 1st international workshop on data analysis patterns in software engineering (DAPSE). IEEE, pp 26-28
Thongtanunam P, McIntosh S, Hassan AE, Iida H (2017) Review participation in modern code review. Empir Softw Eng 22(2):768–817
van der Veen E, Gousios G, Zaidman A (2015) Automatically prioritizing pull requests. In: Proceedings of the 12th working conference on mining software repositories, MSR '15. IEEE Press, pp 357–361
Xia X, Lo D, Wang X, Yang X, Li S, Sun J (2013) A comparative study of supervised learning algorithms for re-opened bug prediction. In: 2013 17th European conference on software maintenance and reengineering. IEEE, pp 331–334
Xia X, Lo D, Shihab E, Wang X, Zhou B (2015) Automatic, high accuracy prediction of reopened bugs. Autom Softw Eng 22(1):75–109
Zanetti MS, Scholtes I, Tessone CJ, Schweitzer F (2013) Categorizing bugs with social networks: a case study on four open source software communities. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 1032-1041
Zou W, Xuan J, Xie X, Chen Z, Xu B (2019) How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. Empir Softw Eng

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Yuan Huang[1] · Xingjian Liang[2] · Zhihao Chen[2] · Nan Jia[3] · Xiapu Luo[4] · Xiangping Chen[5] · Zibin Zheng[1] · Xiaocong Zhou[2]

Yuan Huang
huangyuan5@mail.sysu.edu.cn

Xingjian Liang
liangxj8@mail2.sysu.edu.cn

Zhihao Chen
chenzhh95@mail2.sysu.edu.cn

Nan Jia
jianan_0101@163.com

Xiapu Luo
csxluo@comp.polyu.edu.hk

Zibin Zheng
zhzibin@mail.sysu.edu.cn

Xiaocong Zhou
isszxc@mail.sysu.edu.cn

1 School of Software Engineering, Sun Yat-sen University, Zhuhai 510006, China

2 School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China

3 School of Information Engineering, Hebei GEO University, Shijiazhuang 050031, China

4 Department of Computing, The Hong Kong Polytechnic University, Hongkong, China

5 Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-sen University, Guangzhou 510006, China