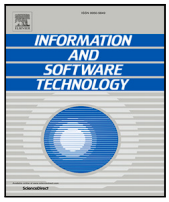
Information and Software Technology 142 (2022) 106756

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Early prediction for merged vs abandoned code changes in modern code reviews

Khairul Islam [a], Toufique Ahmed [b], Rifat Shahriyar [a], Anindya Iqbal [a], Gias Uddin [c,*]

[a] Bangladesh University of Engineering and Technology, Bangladesh
[b] University of California, Davis, United States of America
[c] University of Calgary, Canada

ARTICLE INFO

Keywords:
Code review
Patch
Early prediction
Merged
Abandoned

ABSTRACT

Context: The modern code review process is an integral part of the current software development practice. Considerable effort is given here to inspect code changes, find defects, suggest an improvement, and address the suggestions of the reviewers. In a code review process, several iterations usually take place where an author submits code changes and a reviewer gives feedback until is happy to accept the change. In around 12% cases, the changes are abandoned, eventually wasting all the efforts.

Objective: In this research, our objective is to design a tool that can predict whether a code change would be merged or abandoned at an early stage to reduce the waste of efforts of all stakeholders (e.g., program author, reviewer, project management, etc.) involved. The real-world demand for such a tool was formally identified by a study by Fan et al. (2018).

Method: We have mined 146,612 code changes from the code reviews of three large and popular open-source software and trained and tested a suite of supervised machine learning classifiers, both shallow and deep learning-based. We consider a total of 25 features in each code change during the training and testing of the models. The features are divided into five dimensions: reviewer, author, project, text, and code.

Results: The best performing model named PredCR (Predicting Code Review), a LightGBM-based classifier achieves around 85% AUC score on average and relatively improves the state-of-the-art (Fan et al., 2018) by 14%–23%. In our extensive empirical study involving PredCR on the 146,612 code changes from the three software projects, we find that (1) The new features like reviewer dimensions that are introduced in PredCR are the most informative. (2) Compared to the baseline, PredCR is more effective towards reducing bias against new developers. (3) PredCR uses historical data in the code review repository and as such the performance of PredCR improves as a software system evolves with new and more data.

Conclusion: PredCR can help save time and effort by helping developers/code reviewers to prioritize the code changes that they are asked to review. Project management can use PredCR to determine how code changes can be assigned to the code reviewers (e.g., select code changes that are more likely to be merged for review before the changes that might be abandoned).

1. Introduction

Code review is a practice where a developer submits his/her code to a peer (referred to as 'reviewer') to judge the eligibility of the written code to be included in the main project code-base. Code review helps remove errors and issues at the early stage of development. As such, code review can reduce bugs very early and improve software quality in a cost-effective way. A code review process has some distinct steps (see Fig. 1). The process starts when a developer introduces a code change by creating a patch or revision. The developer or the project moderator assigns a reviewer to examine this change request [1]. The reviewer inspects the code, discusses any possible improvement, and often suggests fixes. After the review, the developer may provide a new patch or revision addressing the review comments and generate a new review iteration. This process repeats until either the reviewer accepts the changes and it gets merged to the project, or the reviewer rejects the code changes and it gets abandoned [2]. Such a workflow is facilitated by different automated code review tools such as Gerrit [3].

Substantial efforts are spent by code reviewers to review a patch thoroughly, to make code changes, and to analyze comments/suggestions made by the authors. If a change is abandoned after some

* Corresponding author.
E-mail address: gias.uddin@ucalgary.ca (G. Uddin).

https://doi.org/10.1016/j.infsof.2021.106756
Received 26 April 2021; Received in revised form 17 September 2021; Accepted 12 October 2021
Available online 23 October 2021

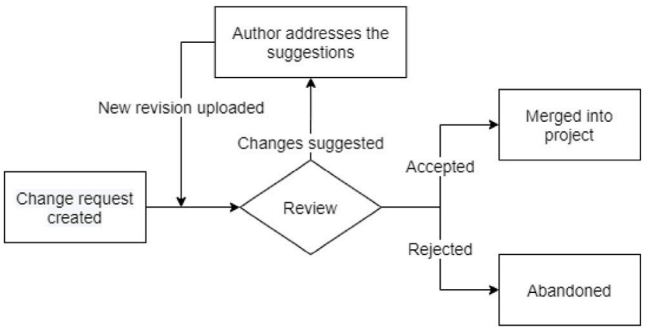


Fig. 1. Workflow of a modern code review process.

iterations, it causes significant waste of time and resources both for the code reviewer and the code author. Indeed, we found that around 12% of code changes are abandoned in our mined data in three large and popular open-source software projects (see Table 3 in Section 3). Therefore, if we can predict early whether a code change would be merged or abandoned in the long run, we can reduce the waste in effort and time by both code reviewers and authors. The prediction has to come as early as possible so that the reviewers can use it to prioritize which code change to review next. On the other hand, the management can analyze the cause of an ongoing review process with negative predictions and intervene to save resources.

The real-world demand for a tool to early predict the future merge/abandon chance of a code change was previously identified by a study by Fan et al. [4]. They surveyed 59 developers from three popular open-source software communities (Eclipse, LibreOffice, and GerritHub) and asked them whether they needed a tool to predict early if a code change will be merged or abandoned in the future. The developers agreed that they need a tool to early predict whether a change would be merged/abandoned in the future. Developers pointed out that this will (1) help prioritize code changes to review, (2) increase their confidence in merging the changes, and (3) reduce the resources wasted due to abandoned changes.

A number of techniques and tools are developed in recent years to assist code reviews with such early prediction. Jeong et al. [2] proposed a model to predict patch acceptance in the Bugzilla system. Gousios et al. [5] predicted pull request acceptance on GitHub. They calculated features when the pull request has been closed or merged. The most recent early prediction model is a shallow learning Random Forest model developed by Fan et al. [4]. They compared their performance with Jeong et al. [2] and Gousios et al. [5] and showed better results at predicting merged changes.

Unfortunately, all the above approaches suffer from one or more of the following shortcomings:

(1) Jeong et al. [2] used programming language-specific keywords as features but did not use any historical data, which can offer more contexts to predict the likelihood of future merged/abandoned states.
(2) Gousios et al. [5] predict just before a pull request is merged/abandoned, which might be too late to save efforts because a code review can span over multiple iterations and interactions between the code author and the code reviewer. Intuitively, the sooner we can predict (in this cycle of iterations), the more efforts and time we can hope to save for both stakeholders.
(3) Fan et al. [4] do not use any reviewer or project-related dimensions, which can offer useful insights that are more specific to a reviewer or a project.
(4) All the three models also suffer from bias against new authors, i.e., pull requests from new authors could be unfairly predicted as most likely to be abandoned due to lack of data.

Therefore, software developers and code reviewers can benefit from a more robust tool that can more reliably predict whether a code change would be merged or abandoned in the future.

In this paper, we have conducted an empirical study on the feasibility of developing a better classification model by addressing the above limitations of prior works. Using Gerrit API, we have mined 146,612 code changes from the code reviews of three large and popular open-source software projects (Eclipse, LibreOffice, and GerritHub). Each code change has information of whether it is merged or abandoned — this is our target variable. For each code change, we compute 25 features from five dimensions: reviewer, author, project, text, and code. We then train and test five shallow machine learning models and one deep neural network model on the dataset. We find that a LightGBM-based model offers the best overall performance. We name the model PredCR. In an empirical study involving PredCR, we answer the following five research questions:

RQ1. Can our proposed early prediction model PredCR outperform the state-of-the-art baselines? This validates the contributions of our work, compared to the prior works (Section 4.1). The most recent model on the early prediction of merged code changes was developed by Fan et al. [4]. Their model outperforms previous works (Jeong et al. [2], Gousios et al. [5], etc.). We have compared our model performance with the state-of-the-art by reproducing their work. We found that PredCR relatively improves the AUC score by 14%–23%. The normalized improvements [4,6] are 44%–54%. Therefore, our developed early prediction model PredCR offers considerable performance improvement over state-of-the-art baseline (i.e., Fan et al. [4]).

RQ2. How effective is each feature dimension in our proposed approach? This investigates how each feature dimension in PredCR performs. We have found that (Section 4.2) on average the AUC scores in models on the reviewer, author, project, text, and code dimensions are 77%, 67%, 58%, 53%, and 57% respectively. So previous experience-related features have much impact on the code change outcome. Also, when dimensions are used all together the average AUC score is around 85%. This validates that PredCR benefits from using all the dimensions.

RQ3. How well does the model handle bias against new authors? As we noted before, state-of-the-art tools to predict early merge/abandoned changes suffer from bias against new authors. One of our goals while designing PredCR was to reduce such bias so that we can facilitate better onboarding of new reviewers and authors into the software ecosystem. We have used historical data to predict merged code changes whereas new authors have little prior records in the system. We find (Section 4.3) that PredCR achieves on average 78.7% AUC score for new authors. This relatively improves the AUC scores by 21%–30% compared to the state-of-the-art [4].

RQ4. How well does our approach work while using multiple revisions? In real life, code changes generally go through multiple revisions before finally getting merged or abandoned. Intuitively, it is more difficult to predict the merge/abandoned change of a code change if we are only looking at the first revision, compared to the last revision. As such, it would be beneficial to find whether and how PredCR can improve its prediction accuracy as we add more revisions to it over time. This research question leads to exploring how well PredCR performs when predictions are updated at each new revision of the same code change. We find (Section 4.4) that if features related to prior revisions are added to the feature set, 6%–15% relative improvements are achieved in terms of the AUC score in the last revision compared to the first. Therefore, PredCR achieves better performance during the latter stages of a revision chain.

RQ5. How well does the model improve over time? As a software project evolve, it can have more data to train over time. Therefore, it is important to understand whether PredCR is able to improve its prediction accuracy, as a project evolves. We thus sliced a project data by time into 11 folds, where fold 0 contains the earliest data and fold 10 contains the most recent data. We find (Section 4.5) that PredCR gives

5%–9% better AUC scores in the second half of the folds (i.e., folds 5–10) than the first half of the folds (i.e., folds 0–4). Therefore, the performance of PredCR for an evolving software project improves over time, as we have access to more data of the software.

Our tool can help reviewers manage their review works better. It can also assist project management to make decisions regarding resource allocation. Code changes with the possibility of being merged into the main codebase can be given more focus than those predicted as abandoned. PredCR also extracts features to understand change intent: bug fix, feature implementation or refactoring (Section 3.2.4). In practice, bug fix changes have more importance than feature implementations and feature implementations have more importance than refactoring. So reviewers can use PredCR to label more important code changes (e.g., bug fixes). Then prioritize changes that are more important and have better merge probability.

Table 1 summarizes the contributions we have made in this work. The usage scenarios of our proposed tool are as below:

- **Without PredCR:** Bob is a developer in a large project team. His responsibility is to review submitted code changes by other developers. With the expansion of the projects, the number of code changes he has to review has increased too. He inspects the code changes serially by the order of submission time or randomly. However, it is difficult for him to keep the focus on reviewing so many code changes. Also, code changes with better quality are often taking much longer to merge into the project for falling behind in the queue. Some of the code changes are being abandoned even after his effort and time. Also after giving some initial reviews in a code change, he has to go through it again to check if the author has improved it in a later revision.
- **With PredCR:** Bob and his team adopt our tool. The tool predicts the probability of getting merged for the code changes that are assigned to the reviewer. Now the code changes can be prioritized based on the suggestion of the tool. So he can focus more on those with a better chance of getting merged in the future. The tool features can also be used to filter more important changes (e.g., bug fixes) and prioritize only them. Also, there would be less delay for the better code changes, as they will be reviewed and accepted earlier. As such, Bob now can spend less time on code changes that will likely be abandoned in the long run. Moreover, the tool updates the prediction with each new revision/patch submission of the same code change. This helps Bob refine his decision to prioritize code changes for review, e.g., a code author may radically improve a new version of a code that was previously predicted to be abandoned by PredCR. With new data, PredCR can update its prediction that the updated code has now more chance of getting merged than abandoned. This will help Bob to then focus more effort on the new code changes during reviews.

Our main objective is to help reviewers review code changes assigned to them. Therefore, our tool PredCR is expected to run after the change has been assigned to him/her. This is based on our observation of how reviews are conducted in platforms like Gerrit, where authors add reviewers (both human and bot) while creating the code change. We find that on rare occasions the reviewer list might get updated later.[1]

Replication Package. https://github.com/khairulislam/Predict-Code-Changes.

Paper Organizations. The rest of the paper is organized as follows. Section 2 presents the prior works related to ours. Section 3 presents the data collection process, studied features, research questions, and evaluation metrics. Section 4 presents the answers to the research questions presented in the previous section. Section 5 discusses the major themes of our study results and highlights the finding of our study. Then in Section 6, we have presented the threats to the validity of our work. And Section 7 has the concluding remarks.

[1] https://git.eclipse.org/r/c/4diac/org.eclipse.4diac.ide/+/184346.

2. Related work

In this section, we have presented the prior works related to our study. We have discussed their motivations, working setups, features used, and limitations. Table 2 shows the summary of those works and our comparison with them.

2.1. Early prediction in code reviews

Jeong et al. [2] focused on predicting patch acceptance at any state of revisions. They suggested that patches predicted as accepted can be auto-accepted and authors can use it before submitting a patch to get feedback on it. Also, reviewers can use it to predict patch quality. Jiang et al. [11] conducted a study on the Linux kernel and examined the relationship between patch characteristics and patch reviewing/integration time. Kamei et al. [15] built a change risk model based on characteristics of a software change to predict whether or not the change will lead to a defect. However, this does not predict whether the change will be eventually merged or abandoned. Gousios et al. [5] predicted acceptance of pull requests. To obtain an understanding of pull request usage and to analyze the factors that affect such development.

Hellendoorn et al. [12] used natural language processing techniques to compute how similar a code change is to previous ones. They then predicted whether it will be approved based on the review outcomes of similar ones. Thongtanunam et al. [8] investigated the characteristics of patches that: (i) do not attract reviewers, (ii) are not discussed, and (iii) receive slow initial feedback. They calculated features just before the code change was closed and predicted acceptance for it at that moment. Gerede et al. [14] focused on predicting whether or not a code change would be subject to a revision request by any of its reviewers.

Fan et al. [4] predicted whether a code change will be merged or abandoned as soon as it was submitted. Their main objective was to prioritize the code review process by early predicting code changes that are more likely to be merged. They compared their works with Jeong et al. [2], Gousios et al. [5], and show state-of-the-art performance. Zhao et al. [9] proposed a learning-to-rank (LtR) approach to recommending pull requests that can be quickly reviewed by reviewers. Different from a binary model for predicting the decisions of pull requests, their ranking approach complements the existing list of pull requests based on their likelihood of being quickly merged or rejected. Huang et al. [13] proposed a method to predict the time-cost in code review before a submission is accepted. They focused on predicting whether a submission will be accepted on the first submission and whether it will take more than 10 submissions.

Our target is to predict the merged probability of a code change request as soon as it is submitted before any review has come. This is similar to the work by Fan et al. [4].

2.2. Review tool used

Jeong et al. [2] used the Bugzilla system in Firefox and the Mozilla Core projects. Gousios et al. [5], Zhao et al. [9], Hellendoorn et al. [12] worked with pull requests in GitHub projects. Jiang et al. [11] worked on the Linux kernel which is supported by Git repositories. Thongtanunam et al. [8], Huang et al. [13], Gerede et al. [14], and Fan et al. [4] worked on open source projects using the Gerrit tool. We have also worked with the Gerrit tool.

2.3. Feature dimensions

Jeong et al. [2] used patch metadata, patch content, and bug report related features. Bug report-related features are very specific to the Bugzilla system they worked on. However, they do not use any historical data in the feature set. Shin et al. [16] showed that without historical data fault prediction models usually have low performance. Gousios

Table 1
Research contributions made in this work.

Topic	Research contribution	Research advancement
Prioritizing review requests	Our work shows considerable performance improvement compared to the state-of-the-art [4] in early predicting outcome of code changes.	Predicting outcome of code changes has been highlighted by many prior studies [2] [7] [5] [8] [4] [9]. Our study will help to reduce the difficulties programmers are facing in the rapid growth of software projects.
Reducing prediction bias	We have shown that PredCR can reduce the prediction bias against new authors is most cases compared to the state-of-the-art [4].	Code review approach for newcomers is different [10]. So careful approach is necessary so that such a prediction model does not discourage them from contributing. Our study will help the community by reducing this bias.
Update prediction at multiple revisions	We have presented an adjusted approach that can update prediction at the submission of new revision for a code change so that efforts at later revisions are recognized.	Compared to prior arts which calculates code change related features only at initial submission [4] or just before closing [11] [5] [8], our approach adds the flexibility to also consider subsequent revisions. This is more useful as it scores based on the latest patch before any review has started on it.

Table 2
Comparison of our paper with related works.

Topic	Our works	Prior study	Comparison
Early Prediction in Code Reviews	Our goal is to predict early whether a code change will be merged or abandoned to prioritize reviews and to reduce waste of efforts on abandoned changes.	Predict whether a patch will be accepted [2,5,12], will need more than one submission to be accepted [13,14], will fail to attract reviewers [8], will be closed earlier than others [9]. Early prediction of a code change being merged [4].	Our goal is to predict merge probability early before any review starts. Similar to Fan et al. [4].
Review tool used	Gerrit code review tool	Bugzilla [2], Linux kernel [11], Github [5,9,12], Gerrit [4,13,14]	As all features available on one tool, might not be available on another, our work on Gerrit cannot be compared directly with all of them.
Feature dimensions used	Reviewer, author, project, text and code related features. Experience related features were calculated using more recent data (past 60 days).	Code or patch [2][11] [13] [5] [4], bug report [2], project [4,5,8], author [5] [4], review [11] [8], text [8] [4], reviewer [8]	We have focused on more recent performance of authors and reviewers. All features presented by us in Section 3.2 are available from the creation of the code change.
Program language dependency	We have not used any language-dependent features	Jeong et al. [2], Hellendoorn et al. [12], and Huang et al. [13] used features dependent on Java language.	PredCR can be used on any project using the Gerrit tool as it is language-independent.

et al. [5] used pull request, project, and developers' characteristics-related features. Both Jeong et al. [2] and Gousios et al. [5] used some features (time after open) which are not available when the first patch is submitted. Gousios et al. [5] also used review activities in previous revisions in the feature set (num_comments, num_participants). They calculated features at the time a pull request has been closed or merged.

Jiang et al. [11] grouped the features into six dimensions: experience, email, review, patch, commit, and development. The review group is related to review participation in the prior patches. The email feature contains information related to prior patches. Thus many of the features are not available when submitting the first patch. Kamei et al. [15] grouped the features into diffusion, size, purpose, history, experience dimensions. Thongtanunam et al. [8] extracted patch metrics in five dimensions: patch properties, history, past involvement of an author, past involvement of reviewers, and review environment. Their history feature is related to review activities in prior patches of the patch set. They calculated the features just before the code change was merged or abandoned. Fan et al. [4] grouped the features into five dimensions: code, file history, owner experience, collaboration network, and text. All of these features are available when the first revision of the code change request is being submitted.

We have grouped our features into five dimensions: reviewer, author, project text, code. All of those features are calculated after the first revision is created.

2.4. Programming language dependency

Jeong et al. [2] used Java language-specific keywords in their feature set to predict patch acceptance. Hellendoorn et al. [12] trained and tested their language models on pull requests that only contain java files. Huang et al. [13] used code modifying features and code coupling features which are java language-dependent. Therefore, they filtered

Table 3
Statistics of collected data.

Project	Time period	Changes	Merged	Abandoned
LibreOffice	2012.03.06 – 2018.11.29	56,241	51,410(91%)	4,831(9%)
Eclipse	2012.01.01 – 2016.12.31	57,351	48,551(85%)	8,800(15%)
GerritHub	2016.01.03 – 2018.11.29	33,020	29,367(89%)	3,653(11%)
Total		146,612	129,328(88%)	17,284(12%)

out any changes from their dataset which contained any non-java file. These works are programming language-dependent, so cannot be used for projects of different languages. Other previous works discussed [4,5,8,11], do not have a programming-language dependency.

Our work does not use any language-specific features. So it is programming language-independent.

3. Empirical study setup

In this section, we have described how we have collected the data from Gerrit projects and preprocessed them before using them in the experiment. Then we have explained the features extracted from the dataset, which we have grouped into five dimensions. We have presented the rationale and explained how the features were calculated. Then, we have described our evaluation metrics to measure the prediction performance. Finally, we have presented the research questions we shall answer in our work.

3.1. Data collection and preprocessing

We have used the REST API provided by Gerrit systems to collect data from three Gerrit projects LibreOffice, Eclipse, and GerritHub. The miner was created following the approach presented by Yang et al. [17]. We have collected changes with the status "merged" or "abandoned". We have mined a total of 61062, 113427, and 61989 raw code changes respectively from LibreOffice, Eclipse, and GerritHub respectively within the time period mentioned in Section 3.1.

To filter out the inactive/dead sub-projects, we have selected sub-projects with at least 200 merged code changes. Hence, 4, 64, and 48 sub-projects were left respectively from LibreOffice, Eclipse, and GerritHub. We have removed code changes where subjects contain the word "NOT MERGE" or "IGNORE" since these will eventually be abandoned. We have also removed changes where the reviewers are the same as the owners. Some changes did not have patchset data available anymore, we have also excluded them. The same preprocessing steps are applied to all three projects. Table 3 presents statistics of the finally collected dataset. We have also collected registration dates for each developer account. It was later used during feature extraction for the computing experience of the developer. In case of missing values on the date of registration, we have filled them by linearly interpolating them based on the existing dates and account_id. For example, if account_id 3 has registration date missing and the closest previous and next account_ids are 1 and 5 with registration dates 01-01-2018 and 01-05-2018 respectively, then account_id 3 will be assigned 01-03-2018 as the registration date.

3.2. Studied features

We have extracted a total of 25 features from the dataset. All features are calculated when the code change is initially submitted (same as Fan et al. [4]). Gousios et al. [5], Jiang et al. [11], Thongtanunam et al. [8] calculated all features at the time when a change has been closed. However, the review process has already been finished by then and no remedy is effective at that point. Our main goal is to predict the possibility of merging/abandonment for code changes as early as possible. For this reason, we have not used the following dimensions: history (Thongtanunam et al. [8]), review (Jiang et al. [11]), commit (Jiang et al. [11]). These are not available at the initial stage. Also, in Section 4.4 we have shown that by only adding revision numbers to the feature list, PredCR can give significant performance when the prediction is updated after submission of each new revision.

Some features were not available in the Gerrit system. For example: bug report information (Jeong et al. [2]), email (Jiang et al. [11]). When calculating past record-related features, we have generally considered recent performances (in the last 7 or 60 days). Fan et al. [4] added 'recent' prefix to features that were calculated in the last 120 days. Our approach thus is more restrictive in terms of feature history. Table 4 shows our finally selected feature list and the rationale behind choosing those. We discuss the features and dimensions below.

3.2.1. Feature dimension 1. Reviewer

Num_of_reviewers is the number of human reviewers found in the reviewer list of the code change. This feature was previously used by Thongtanunam et al. [8] and Jiang et al. [11]. Num_of_bot_reviewers are the number of bot tools added to the reviewer's list. As these accounts do not actively participate in review discussion but perform different analyses on the patch set, we have kept their number separately. Whether an account is a bot, is determined by checking whether the account name is 'do not use' or it contains any of the following words 'bot', 'chatbot', 'ci', 'jenkins', or the project name. We have calculated a reviewer's experience by the number of years s/he is registered in this system. We have calculated that using the difference of the revision upload date and the reviewer's date of registration in this project. This value is then averaged by the number of reviewers, which is feature avg_reviewer_experience. A reviewer's review count is found by calculating the number of closed (merged or abandoned) changes, in the last 60 days, where that a particular reviewer was involved in the reviewer list. This value is then averaged by the number of reviewers, which is feature avg_reviewer_review_count. Thongtanunam et al. [8] introduced similar features that calculated prior patches that a reviewer has reviewed or authored.

3.2.2. Feature dimension 2. Author

We have used the recent changes in a 60-day window when calculating author_merge_ratio, author_review_number, author_merge_ratio_in_project, changes_per_week. When calculating the merge ratio, if there are no finished changes of this author, then a default merge ratio of 0.5 is given. Author_merge_ratio is the ratio of merged changes among all finished changes created by this author. Author_review_number is the number of changes where the author is in the reviewers' list. Author_merge_ratio_in_project is the author's merge ratio in the corresponding sub-project. This sub-project name comes with the "project" key in code change response, so we have kept it in this way. Changes_per_week is the number of closed changes each week for this author in the last 60 days. Author_experience is calculated following the same way as the reviewer experience, i.e., taking the difference between the current revision upload date and the author's date of registration in years. Total_change_number is the number of changes created by this author.

3.2.3. Feature dimension 3. Project

We have calculated all project-related features in a 60 days window.

Project_changes_per_week feature is calculated using the number of changes closed every 7 days among the past 60 days for this sub-project. Changes_per_author is the number of closed changes per author in the last 60 days. Project_merge_ratio is the ratio of merged and closed changes in the last 60 days for this sub-project. If the project does not have any finished changes yet, the default merge ratio of 0.5 is given.

Table 4
List of features. The dimensions which we have used, but were not used by state-of-the-art [4] are highlighted as bold. The features for which we did not find prior studies using them, are highlighted as bold too.

Dimension	Rationale	Feature Name
Reviewer	Reviewers number and their past record affect change outcome [18] [8].	**avg_reviewer_experience** avg_reviewer_review_count [8] [19] num_of_reviewers [8] [11] **num_of_bot_reviewers**
Author	Experienced programmer has low defect probability [20]. Developer's experience significantly impacts on change outcome [11] [5]. More active developers have a better chance at merging patches [21].	author_merge_ratio [4] **author_experience** author_merge_ratio_in_project [4] total_change_number [4] [19] author_review_number [4] [11] author_changes_per_week [8]
Project	Large workload results in less review participation [19]. Project's receptiveness affects change outcome [5]	project_changes_per_week [8] **changes_per_author** **project_merge_ratio**
Text	Well explained descriptions better draw attention [18] Intent of a code change is related to the kind of feedback it receives.[22]	description_length [4] is_bug_fixing [4,8,15] is_feature [4,8] is_documentation [4,8]
Code	Modifying more directories is usually defect-prone [20]. Scattered changes are more prone to defects [23]. Larger changes are more defect-prone [24]. Touching many files is more defect-prone [25] [24].	modified_directories [8] modify_entropy [15] lines_added [2] [4] lines_deleted [2] [4] files_modified [5] files_added [4] files_deleted [4] subsystem_num [4]

3.2.4. Feature dimension 4. Text

These features are calculated on the change description provided for the code change. The aim is to identify the purpose of the code change. The description is provided in the subject of the code change when it is created. Description_length is the number of words present in the change description. The other three features have binary values, i.e., 0 or 1. We have marked a code change as documentation if the change description contains "doc", "copyright", or "license". Similarly, we categorize it as bug fixing if the change description contains "bug", "fix" or "defect". Other changes are marked as a feature. These are done following Thongtanunam et al. [8].

3.2.5. Feature dimension 5. Code

This section refers to the features which are related to the changes made in The source code. Modified_directories refer to the number of directories modified by this code change. It is calculated by extracting the bottom directories from file paths. Similarly, subsystem_num is the number of subsystems (the top directory in the file path) modified in the change. Modify_entropy is a feature previously proposed by Kamei et al. [15]. Entropy is defined as $-\sum_{k=1}^{n}(p_k * log_2 p_k)$, where n is the number of the files modified and p_k is the proportion of lines modified among total modified lines in this change. Other features such as files_added, files_deleted, files_modified, and lines_added, lines_deleted are self-explanatory. Most of these source code features have also been used in prior studies [4,5,8,15].

3.3. Performance metrics

We use a total of seven metrics to report and compare the performance of PredCR against the baselines in our three datasets. The metrics can be broadly divided into two categories: Standard Performance Metrics and Improvement Analysis Metrics. All the metrics except one (cost-effectiveness) are used from Python scikit-learnlibrary. The metrics are defined below.

3.3.1. Standard performance metrics

We report five standard performance metrics:

(1) AUC,
(2) Cost-Effectiveness,
(3) Precision,
(4) Recall, and
(5) F1-score.

AUC. Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) is a widely used performance measure for prediction models. For our case, the AUC score calculates the probability that PredCR prioritizes merged code changes more than abandoned code changes. Following related literature on the early prediction of merged/abandoned code reviews, we use the AUC score to determine the best-performing models.

Cost-Effectiveness (ER@K%). Cost-effectiveness is used to measure performance given a cost limit. As in practice, developers can only review a limited number of changes, our target is to correctly predict as many merged cases as possible within that limit. Following prior studies [1,4], we have used EffectivenessRatio@K% (ER@K% in short), which evaluates the percentage of merged code changes in the top K% code changes(sorted by decreasing order of merge probability) predicted as "Merged".

This also helps evaluate how well our model can prioritize the code changes. A larger effectiveness ratio means the model better prioritizes code changes that will eventually be merged. The state-of-the-art [4] used this metric for the same purpose. Xia et al. [1] used this metric to evaluate the prioritization of blocking bugs. Jiang et al. [7] also used this to evaluate the ranking of personalized defect prediction. The authors of these works used prediction probability from the model to prioritize.

By denoting the number of merged changes and the number of changes in top K% as N_{mk} and N_k, respectively, we get,

$$ER@K\% = \frac{N_{mk}}{N_k} \quad (1)$$

We have used ER@20% as the default cost-effective metrics. In Section 5.1.1, we have shown PredCR performance when K is varied from 10 to 90. Note that using K at 100 does not have any significance. Top 100% means all code changes are being chosen. In that case, the proportion of merged changes and all changes is constant, irrespective of the model.

Precision. The proportion of changes that are correctly labeled among all predicted examples of that class. For merged and abandoned classes, we presented this metric as P(M) and P(A).

$$P(M) = \frac{TP}{TP + FP}, P(A) = \frac{TN}{TN + FN} \tag{2}$$

Recall. The proportion of changes that are correctly labeled among changes that actually belong to that class. For merged and abandoned classes, we presented this metric as R(M) and R(A).

$$R(M) = \frac{TP}{TP + FN}, R(A) = \frac{TN}{TN + FP} \tag{3}$$

In our context, precision implies the percentage of results that are relevant. On the other hand, recall refers to the percentage of total relevant results correctly classified by our algorithm.

F1-Score. The harmonic means of precision and recall. For merged and abandoned classes we presented this metric as F1(M) and F1(A).

$$F1(M) = \frac{2 * P(M) * R(M)}{P(M) + R(M)} \tag{4}$$

$$F1(A) = \frac{2 * P(A) * R(A)}{P(A) + R(A)} \tag{5}$$

3.3.2. Improvement analysis metrics

We report two metrics:

(1) Relative Improvement (RIMPR), and
(2) Normalized Improvement (NIMPR).

Relative Improvement (RIMPR). By relative improvement, we mean the relative change between the two scores. Instead of simply calculating the difference it is better because it considers the difference relative to the old value. For example, improving a score from 20% to 40% is only a 20% increase in score. But only calculating the difference misses the fact that the new score is double the previous score. However, the improvement here is 100% which clearly shows that fact. Improvement is calculated as follows,

$$Relative\ Improvement\ (RIMPR) = \frac{new\ score - old\ score}{old\ score} \tag{6}$$

We report this metrics name as RIMPR throughout the rest of the paper.

Normalized Improvement (NIMPR). Normalized improvement is a measure proposed by Costa et al. [6] to evaluate the improvement between two methods in terms of an evaluation metric. The same metrics have been used by Fan et al. [4] to highlight improvements over baselines in prioritizing code changes for reviewers. It takes room for improvement into consideration. For example: let us consider accuracy is improved from 80% to 85% and F1_score is improved from 90% to 95%. In both cases, the improvement is 5%, but normalized improvement is 25% and 50%, respectively. In the latter case, the room for improvement was only 10%. Hence, a 5% improvement here has much more impact. We have used the short form of this metric as NIMPR.

$$Normalized\ Improvement\ (NIMPR) = \frac{new\ score - old\ score}{1 - old\ score} \tag{7}$$

3.4. Experimentation setup and approach

We have used the longitudinal cross-validation setup, previously used by Fan et al. [4]. Previous works have used similar setups to ensure only using past data to predict future events. Rakha et al. [26] used a similar approach in retrieving duplicate issue reports. Bangash et al. [27] used time-aware evaluation in cross-project defect prediction.

For each project, the selected code changes are first sorted in increasing order of creation time. Then they are divided into 11 non-overlapping windows of equal size following Fan et al. [4]. Instead of traditional ten-fold cross-validation, this approach is followed to ensure that no future data is used during training.

In the first fold, the model is trained using window 1 and tested on window 2. In the second fold, the model is trained using windows 1 and 2 and tested on window 3. Similarly, in the last fold(10), the model is trained on windows 1–10 and tested on window 11. At each stage, we have calculated the AUC, ER@20%, precision, recall, and F1 scores for merged and abandoned code changes. Then we have computed the average of the metrics across ten-folds for both merged and abandoned code changes. Kaggle kernels were used to run all experiments. They provide an Intel(R) Xeon(R) CPU with 16 Gigabytes of Ram, 4 CPU cores, and a 2.20 GHz processor. We have used Python as the programming language. Due to the stochastic nature of the machine learning models, it is recommended to run a model multiple times and take the average for final performance reporting. In our case, for each model, each experimentation is rerun ten times and the average result is reported to ensure stable model performance. This means that for each model we did longitudinal 10-fold cross-validation 10-times and then took the average. During each of the runs, we did hyperparameter tuning.

Model selection process. First, we have used StandardScaler to fit and transform the features of each project. Then to find the best model, we have used six machine learning classifiers GradientBoosting [28], RandomForest [29], ExtraTrees [30], LogisticRegression, LightGBM [31] and Deep Neural Network(DNN). Except for LightGBM and DNN, all other classifiers are imported from the scikit-learn library. The LightGBM classifier used is taken from lightgbm[2] library. The DNN model was created using the keras[3] library.

Handling class imbalance. As this dataset is an imbalanced one, we have considered class imbalance when training the models. We have balanced the classification loss, by setting the classifier parameter class_weight to 'balanced'. This uses the values of the target column to automatically adjust weights inversely proportional to class frequencies in the input data. This way class imbalance is taken into consideration when calculating loss. Hence, we have set class_weight = 'balanced' for all of these classifiers. Except for GradientBoosting, which automatically handles class imbalance by constructing successive training sets based on incorrectly classified examples [28].

Randomness across different runs. To introduce randomness across different runs, we set solver = 'saga' for LogisticRegression (suggested by scikit-learn documentation). And subsample=0.9, subsample_freq=1, random_state = numpy.random.randint(seed=2021) for LightGBM (this will subsample 90% of the train data each time). Otherwise, these two models produce the same results after each run, and rerunning them ten times does not have a meaning. The DNN model maintains random results because it initialized with random weights. The other models had their random_state kept to default 'None' during model initialization. We also manually validated whether each run is creating different results.

Deep Neural Network (DNN) Architecture. We have used the deep neural network model to investigate whether neural networks would outperform other machine learning classifiers. The network architecture is shown in Fig. 2. It contained three dense layers. The input dense layer contains 25 relu units, one for each feature. Then we have added a dropout layer with a 10% dropout rate, this would randomly drop 10% of the incoming values, which will help reduce overfit on the training data. Then another dense layer with 16 relu units. Then another dropout layer with a 10% dropout rate. The output layer contains one sigmoid node to convert input values within 0 to 1. This is the merged probability predicted by the model. We have used the 'adam' optimizer and 'binary_crossentropy' loss. The number of epochs was set to 10 (increasing or decreasing epoch more reduced test performance) during model training.

Parameter-tuning. We have used grid search to hyper-tune the parameters for each model and presented their best performance. We tuned n_estimators and max_depth for RandomForest and ExtraTrees

[2] https://lightgbm.readthedocs.io/en/latest/index.html.
[3] https://keras.io/.

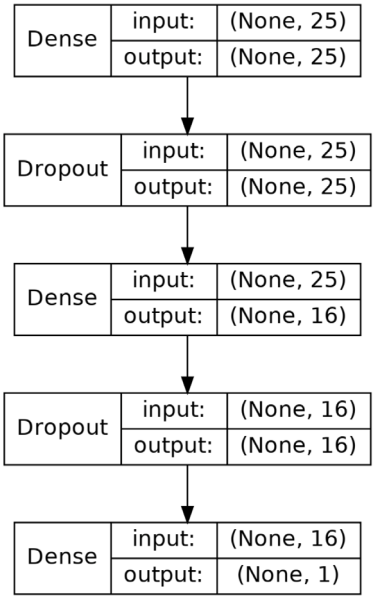


Fig. 2. DNN model architecture.

classifier, n_estimators, and learning_rate for GradientBoosting and LightGBM, max_iter for LogisticRegression. For the DNN, we have experimented by varying the number of layers, dropout rate (0.10, 0.15, 0.20), optimizers (Adam, RMSProp, SGD, Adagrad), and the number of nodes.

The best-performing one was chosen based on the AUC score, as it is mentioned [4] as the most important evaluation metrics to prioritize code changes for reviewers. The chosen model is then used to compare our longitudinal cross-validation performance with the state-of-the-art. The same classifier is later used to answer our other research questions. The hyper-tuning results for the chosen model are discussed in Section 5.1.3.

Reproducing state-of-the-art baseline. To compare our work with the state-of-the-art [4], we have followed the steps presented in their work and also publicly shared git repository[4] to reproduce it. We have preprocessed our dataset using the same steps as them. Then calculated their features. However, we found a bug in their feature calculations, which calculates the status of some code changes which have not been closed yet. The bug found in author_feature.py - class AuthorPersonalFeatures - def extract_features, separates code changes by the creation date of the current code change(for which they are calculating features). However, some of those code changes are merged after the current change was created. When calculating merge ratios, these changes were not excluded. Thus merge ratios now may contain information about the future of those code changes, potentially leaking the target label(merged or abandoned). We have fixed this issue by excluding those open changes when calculating any kind of merge ratio (merged_ratio, recent_merged_ratio, subsystem_merged_ratio). We have shared both implementations (original and fixed) in our shared repository.[5]

We have used a RandomForest classifier with class_weight='balanced' as their model, it is equivalent to their usage of RandomForest from the Weka tool with $\alpha = 1$ in cost ratio. We have hyper-tuned their model and found the best results when n_estimators are 500 and max_depth is 5. This hyper-tuned model is then used on their feature set calculated from the same dataset as us, following the longitudinal cross-validation setup. So our results are directly comparable.

[4] https://github.com/YuanruiZJU/EarlyPredictionReview.
[5] https://github.com/khairulislam/Predict-Code-Changes.

Table 5
Performance for different classifiers across the three projects.

Model	AUC		
	LibreOffice	Eclipse	GerritHub
LightGBM	**86.0**	**84.3**	**84.6**
DNN	85.3	82.9	84.0
Random Forest	85.2	83.6	83.5
GradientBoosting (GBT)	85.4	82.8	82.7
ExtraTrees	85.6	83.3	83.6
Logistic Regression	81.2	77.0	79.3

4. Empirical study results

In this section, we answer five research questions:

RQ1. Can our proposed approach outperform the state-of-the-art?

RQ2. How effective is PredCR when only one feature dimension is used?

RQ3. How well does the model handle bias against new authors?

RQ4. How well does our approach work while using multiple revisions?

RQ5. How well does the model improve over time?

Answering RQ1 will show how PredCR performs compared to the state-of-the-art works. It will also validate PredCR's effectiveness in the early prediction of merged code changes. RQ2 will highlight the performance of each feature dimension used in PredCR. This research question will also validate whether PredCR benefits from using all features rather than using a subset of them. RQ3 will explore whether PredCR has any bias against new authors. We have used author experience-related features, so this may introduce bias against new authors. By answering this research question we have explained how we have handled it and what impact it had on the performance of PredCR. RQ4 is intended to explore if PredCR is able to improve its prediction ability with subsequent revisions. And finally, RQ5 investigates if PredCR has any advantage of using a longer period of time. With time, training data can be enriched as new code changes are available in a project and hence performance improvement is expected.

4.1. Can our proposed approach outperform the state-of-the-art? (RQ1)

4.1.1. Motivation

To validate the performance of PredCR, we plan to compare our approach and performance with the state-of-the-art. The work of Fan et al. [4] is considered state-of-the-art on the early prediction of merged code changes. Their model outperforms the previous works (Jeong et al. [2], Gousios et al. [5], etc.) with respect to most of the metrics. So if PredCR is able to outperform their models in a similar setup, its superiority and applicability will be established.

4.1.2. Approach

The approach described in Section 3.4 is used here to evaluate this research question.

4.1.3. Results

Table 5 shows the results for selecting the best classifier. AUC is chosen as it is suggested to be the best metric for this task [4]. The best results for each project are in bold. **We have found that LightGBM has the best overall performance across the three projects.** Which is not surprising as LightGBM has previously shown better performance than similar gradient boosting decision trees [31]. LightGBM showed an AUC score of 86.0, 84.3, and 84.6 for the three projects LibreOffice, Eclipse, and GerritHub, respectively. **We, therefore, have picked LightGBM as the underlying model in our PredCR tool and used it throughout the remaining parts of the paper.**

Table 6, shows the results of our best model and its comparison with the state-of-the-art [4]. **PredCR outperforms the state-of-the-art**

Table 6
Longitudinal cross-validation test results with comparison.

Project	Approach	AUC	ER@20%	Merged			Abandoned		
				F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
LibreOffice	Ours	**86.0**	**99.3**	**94.1**	**95.9**	**92.4**	**48.3**	**42.2**	**58.7**
	Fan et al [4]	70.2	96.3	86.7	91.1	82.8	31.7	26.4	42.2
Eclipse	Ours	**84.3**	**97.5**	**92.3**	**92.9**	**91.8**	**57.7**	**56.0**	**60.0**
	Fan et al [4]	69.7	93.9	81.6	89.2	75.6	36.1	29.2	50.0
GerritHub	Ours	**84.6**	**99.0**	**92.0**	**95.3**	**88.9**	**50.0**	**42.2**	**62.5**
	Fan et al [4]	74.4	98.2	82.9	93.9	74.8	33.4	24.1	59.0

Table 7
Improvements (RIMPR and NIMPR) of PredCR over the baseline (Fan et al. [4]).

Project	Metric	RIMPR	NIMPR
LibreOffice	AUC	22.8	53.7
	ER@20%	3.11	81.1
	F1(M)	8.53	55.6
	F1(A)	52.4	24.3
Eclipse	AUC	20.9	48.2
	ER@20%	5.06	59.0
	F1(M)	13.1	58.2
	F1(A)	59.8	33.8
GerritHub	AUC	13.7	39.8
	ER@20%	0.81	44.4
	F1(M)	8.56	41.5
	F1(A)	49.7	24.9

in all cases. Our average AUC is around 85%, where the state-of-the-art is around 71.4%. We note that we find a slightly lower performance for Fan et al. [4] model compared to its performance as reported in the Fan et al. paper. This can be due to dataset differences and adding a fix in their feature calculation.

As defined in Section 3.3, we calculate the relative improvement (RIMPR) and normalized improvement (NIMPR) [4,6] of PredCR over the state-of-the-art baseline (i.e., Fan et al. [4]). We present the improvement of PredCR over the baseline in Table 7 using four metrics AUC, ER@20%, F1(M), and F1(A). We find that

(1) PredCR improves the AUC scores by around 14%–23% compared to state-of-the-art [4] and the normalized improvements are around 46%–58%.
(2) The ER@20% in state-of-the-art was already around 96% on average. However, PredCR still provides around 44%–81% normalized improvement.
(3) In terms of f1_score for merged changes, PredCR provides around 9%–13% relative improvements and 42%–58% normalized improvements. Though the state-of-the-art [4] were already significant for merged code changes.
(4) For abandoned changes, PredCR improves f1_score by a large margin. It gives around 50%–60% relative improvements and 24%–34% normalized improvements. Considering only 12% of the code changes are abandoned, difficulties in accurately predicting the fate of abandoned code changes are significantly higher.

Table 8 shows the importance of our studied features while running the longitudinal cross-validation process. It was calculated using the feature_importances_ attribute provided by the LightGBM classifier for each project during the longitudinal cross-validation process and averaged over all runs. The importance of the top three features for each project is in bold. The review and project dimensions added in PredCR were not used by Fan et al. [4], but Table 8 shows that they have a significant impact on prediction performance.

Table 8
List of features with importance in PredCR.

Dimension	Feature Name	Feature Importance		
		LibreOffice	Eclipse	GerritHub
Reviewer	avg_reviewer_experience	**9.67**	6.73	**10.0**
	avg_reviewer_review_count [8] [19]	**10.9**	**8.18**	**8.98**
	num_of_reviewers [8] [11]	3.64	4.94	7.73
	num_of_bot_reviewers	2.84	0.60	1.11
Author	author_merge_ratio [4]	4.72	2.68	4.24
	author_experience	**9.25**	8.07	**8.30**
	author_merge_ratio_in_project [4]	1.71	3.77	1.53
	total_change_number [4] [19]	7.23	**8.40**	7.21
	author_review_number [4] [11]	7.61	**8.55**	7.87
	author_changes_per_week [8]	4.91	5.39	7.02
Project	project_changes_per_week [8]	7.55	7.00	7.00
	changes_per_author	5.43	6.28	4.94
	project_merge_ratio	2.93	4.43	5.35
Text	description_length [4]	3.53	3.79	2.64
	is_bug_fixing [8] [4]	0.30	0.33	0.17
	is_feature [8] [4]	0.38	0.84	1.19
	is_documentation [8] [4]	0.18	0.28	0.24
Code	modified_directories [8]	2.07	0.98	0.96
	subsystem_num [15],	2.83	5.90	3.39
	modify_entropy [15]	2.47	2.02	2.42
	lines_added [2,4]	4.44	5.24	3.93
	lines_deleted [2,4]	3.33	3.32	3.21
	files_modified [5]	1.24	1.24	1.37
	files_added [4,32]	0.48	0.80	0.72
	files_deleted [4,32]	0.30	0.25	0.06

RQ1. Can our proposed approach PredCR outperform the state-of-the-art baseline? Our PredCR tool is based on the LightGBM model, which **on average**, outperforms the state-of-the-art [4] by 19% in terms of AUC score. If we compare the normalized improvement (NIMPR) metric, PredCR outperforms the state-of-the-art [4] by 48% in terms of AUC score. PredCR outperforms by 10% for merged and by 54% for abandoned changes (in terms of F1-score). The most informative two features in PredCR are avg_reviewer_experience and avg_reviewer_review_count which belong to the Reviewer dimension, none of which were used by Fan et al. [4].

4.2. How effective is PredCR when only one feature dimension is used? (RQ2)

4.2.1. Motivation

We have described the features we have used in Section 3.2. In this research question, we have investigated how much performance each feature dimension used in PredCR achieves alone. This will also validate whether PredCR benefits from using all those feature dimensions or not.

4.2.2. Approach

We have used the same longitudinal ten-fold cross-validation on all projects. We have worked first with all dimensions and later trained and tested the classifier for one feature dimension only. Then reported the performance metrics.

4.2.3. Results

Table 9 shows PredCR performance using all feature dimensions and single feature dimension. The best results for each dimension are in bold. The average AUC on models trained on all dimensions, reviewer, author, project, text, and code dimensions are 85%, 77%, 67%, 58%, 53%, 57%. In terms of the AUC score, **PredCR on average improves reviewer, author, project, text, and code models by 10%,**

Table 9
Performance of PredCR for all features and in each feature dimension.

Project	Dimension	AUC	ER@20%	F1(M)	F1(A)
LibreOffice	All dimensions	**86.0**	**99.3**	**94.1**	**48.1**
	Reviewer	81.3	97.9	92.2	42.9
	Author	67.7	96.1	90.7	25.1
	Project	50.8	91.2	76.8	11.5
	Text	52.5	92.6	73.2	14.9
	Code	53.7	92.6	81.0	14.5
Eclipse	All dimensions	**84.3**	**97.5**	**92.3**	**57.7**
	Reviewer	75.9	93.3	91.5	54.2
	Author	65.3	92.4	81.6	31.5
	Project	58.1	90.0	78.5	25.5
	Text	55.1	87.7	74.0	24.2
	Code	55.9	88.4	76.1	24.6
GerritHub	All dimensions	**84.6**	**99.0**	**91.7**	**49.3**
	Reviewer	72.7	95.3	86.8	35.1
	Author	69.3	97.5	83.6	26.9
	Project	66.4	96.8	77.7	26.2
	Text	52.4	90.2	58.2	19.2
	Code	61.2	95.8	74.8	22.5

27%, 46%, 60%, and 49% respectively. Except for the reviewer dimension, all other dimensions have poor performance for abandoned code changes.

We have presented two examples to demonstrate the importance of the reviewer dimension and how it affects the change request outcome. In project LibreOffice, for change id 65890,[6] the author was facing build failures because one of the pipeline tests was failing. The reviewer mentioned that the test failed not because of the author's change. If he would have uploaded a new revision of this patch, the tests might have run successfully. The author later abandoned this change and created another change 66203[7] for the same issue. This was later merged successfully with further help from the reviewer. Clearly, the reviewer's experience directly influenced the outcome of these changes. In project GerritHub change id 745519,[8] the reviewer suggested that the change made by the author was unnecessary since there was a better alternative. The experienced reviewer knew about this method, but the author did not. After reviewer's suggestion he abandoned the change.

To demonstrate the importance of the author dimension, we show an example from our dataset below. In LibreOffice change id 4071[9] the author gives a fix for several bugs. The reviewer compliments the author for fixing this critical problem. The author is experienced in this project and had been working for more than 1 year, with a 0.98 merge ratio in this project. At the time of this code change, he was making around 7 code changes per week and also actively reviewing other code changes.

Similarly, here is an example of the project dimension. In LibreOffice change id 4071, the code change is made for the 'core' sub-project. At the time this code change was created, this sub-project had around 103 code changes per week, a merge ratio of 0.87, and on average 8 code changes per developer. The author was making around 7 code changes per week, so he was a regular developer on that project. We see his code changes get merged with minimal review.

And, the following example shows the importance of text dimension. In project GerritHub change id 745519,[10] the change description says, "add brctl command for neutron-linuxbridge image". The number of words would be 6. And following the approach of Thongtanunam et al. [8] this code change will be labeled as a feature.

[6] https://gerrit.libreoffice.org/c/core/+/65890.
[7] https://gerrit.libreoffice.org/c/core/+/66203.
[8] https://review.opendev.org/#/c/745519/.
[9] https://gerrit.libreoffice.org/c/core/+/4071.
[10] https://review.opendev.org/#/c/745519/.

The next one demonstrates the importance of the source code-related dimension. For example, LibreOffice change id 4071 is a medium-size code change. The author made 71 line additions and 4 deletions across 6 files. So this is easier for the reviewers to inspect. We see it gets merged with minimal review. These examples demonstrate the importance of using PredCR with diverse features.

RQ2. How effective is PredCR when only one feature dimension is used? The reviewer dimension has the best average AUC score of 77% across projects for a single dimension. Also, this dimension has moderate performance on abandoned code changes. Author dimension achieved 67% AUC score on average. Project dimension achieved on average 58% AUC score, but there is a significant difference in score between LibreOffice and GerritHub. Text and code dimensions achieved around 53% and 57% AUC scores, so their impact is close. Using all dimensions together improved our AUC scores by 10%–60%. This validates that PredCR benefits from the use of all features, compared to its subset.

4.3. How well does the model handle bias against new authors? (RQ3)

4.3.1. Motivation

Table 8 shows high importance of author-related features on PredCR. For a new author, it is more likely to consider him/her as inexperienced and predict a lower possibility for merging. For example, Fan et al. [4] faced a considerable bias against new authors. Changes made by new authors were mostly being predicted as abandoned. Hence, they had to propose an adjustment approach. They predicted code changes made by new authors using a model that is only trained on code changes by new authors. They used another model trained on all code changes for experienced contributors. We also need to evaluate how much bias PredCR might have for the new authors.

4.3.2. Approach

We have labeled authors with less than ten code changes as new authors following Fan et al. [4]. Test dataset in each fold of the longitudinal ten-fold cross-validation only contains new authors. Fan et al.'s [4] results were reproduced using their adjusted approach as they suggested.

4.3.3. Results

Table 10 shows the comparison of PredCR performance with state-of-the-art [4]. We find that PredCR's average AUC score across all projects is 85% and for new authors, it is 78.7%. So the performance drop in PredCR for this case is small, considering new authors have either none or few past records. In terms of AUC scores, PredCR improves over Fan et al.'s [4] adjusted approach for LibreOffice, Eclipse, and GerritHub projects by 26%, 31%, and 21%. The normalized improvements are 43%, 47%, and 40%. In terms of ER@20%, PredCR provides 5%–17% relative improvements.

Table 10 also shows that for metrics related to merged code changes, PredCR under-performs in terms of F(M) and R(M). This is because PredCR has less bias against abandoned code changes. For metrics related to abandoned code changes, PredCR significantly outperforms in terms of F(A) and R(A). But under-performs in terms of P(A). However, this shows that Fan et al.'s [4] adjusted approach has a considerable bias towards merged code changes.

We have concluded that in Fan et al.'s [4] original approach, the bias to experienced authors was introduced by using many features related to the author's past records. For the new authors, these feature values are mostly zero and thus cause a bias against them increasing the likelihood of predicting them as abandoned. Even the adjusted approach ends up having a bias towards merged code changes. To reduce such bias, we have decided not to use the collaborative dimension

Table 10
Performance on changes created by new authors.

Project	Approach	AUC	ER@20%	Merged			Abandoned		
				F1(M)	P(M)	R(M)	F1(A)	P(A)	R(A)
LibreOffice	Ours	**78.3**	**94.4**	52.9	**92.3**	41.6	**49.6**	37.1	**85.2**
	Fan et al. [4]	62.1	83.2	**85.3**	76.0	**97.2**	13.9	**49.7**	8.31
Eclipse	Ours	**78.9**	**89.9**	71.5	**87.2**	61.2	**54.0**	42.7	**75.8**
	Fan et al.[4]	60.6	76.6	**79.4**	68.7	**98.2**	9.70	**54.3**	5.38
GerritHub	Ours	**78.9**	**91.1**	64.6	**89.8**	51.9	**49.3**	36.5	**78.9**
	Fan et al.[4]	65.0	86.9	**84.9**	75.6	**97.1**	11.3	**41.6**	6.81

which considers the collaborative history between author and reviewers. This could have resulted in a decrease in overall performance. However, our addition of features related to reviewer and project dimensions makes up for that deficiency and also improves the overall model performance.

RQ3. How well does the model handle bias against new authors? PredCR achieved on average 78.7% AUC score in the longitudinal cross-validation test for new authors, where the state-of-the-art [4] achieved around 63%. PredCR gives a more balanced prediction for both classes, while still maintaining a better AUC score. Also, our model performance for new authors (78.7% AUC) is not far behind the overall model performance (85% AUC).

4.4. How well does our approach work while using multiple revisions? (RQ4)

4.4.1. Motivation

So far we have trained and tested with only the initial submission of code. But in real life, a code change generally goes through several revisions before finally getting merged or abandoned. Each revision contains updated files based on reviews received in the previous revisions. Thus an outcome predicted based on the first revision might be improper for later revisions. The prediction model needs to be able to update prediction given a code change when a new revision is pushed. Besides the initial submission, the stakeholders can still be significantly benefited if a good prediction is available after early-stage revisions.

Many of the changes are not ready for review during the initial submission. The reasons can be: (i) build failure, (ii) pipeline test failure, (iii) work in progress, (iv) merge conflict, (v) unintentionally included changes, and (vi) dependent on any other change. For this, the author has to push more patches. Multiple patches are already uploaded before the review even starts. A merge prediction made only on the first patch would miss any of these cases. For example, in project Eclipse, for change-id 167412,[11] the initial patch was labeled work in progress. The second patch faced a build fail. On the third patch, it was labeled as ready-for-review and finally was merged after the eighth patch. In project LibreOffice, for change-id 100373,[12] it took the author five patches to fix build fails. Only then the change was ready for review and finally was merged at the sixth patch.

4.4.2. Approach

We have designed two adjusted approaches for the merge prediction of a code change in revision rounds. In the first approach, we have added only the review number to the existing feature set. This approach does not train on any previous activities within the patchset. In the second approach, we have added features related to reviews and other activities of previous revisions in the feature set. Similar approach was followed by Gousios et al. [5], Jiang et al. [11] and Thongtanunam et al. [8]. In both approaches, we have used the longitudinal data setup during validation. Change features are sorted according to their creation time.

We have used two different approaches because they will show how PredCR performs with or without considering review activities from previous revisions. **One important difference is that for this approach our features are calculated right after a new revision is uploaded. So that we can give updated predictions on the code change before reviewers have to do any review.** Both Gousios et al. [5] and Thongtanunam et al. [8] calculated features just before the pull request or the code change is closed. However reviews are already done at that point, so predicting at that point would not be helpful for reviewers.

Table 11
AUC(%) for multiple revisions with revision number (Approach 1).

Project	Total	First revision	Last revision	RIMPR	NIMPR
LibreOffice	85.2	86.2	92.5	7.7	47
Eclipse	77.8	83.7	86.1	2.9	15
GerritHub	82.0	85.1	86.5	1.6	9.4

Table 12
AUC(%) for multiple revisions with previous revision related features (Approach 2).

Project	Total	First revision	Last revision	RIMPR	NIMPR
LibreOffice	88.2	86.2	98.8	15	92
Eclipse	79.5	83.7	89.0	6.1	33
GerritHub	82.6	85.1	90.0	5.8	33

4.4.3. Results

Table 11 shows the test results with the first approach. Column RIMPR and NIMPR show the improvement and normalized improvement [4,6] in the AUC scores at the last revision compared to the first. Here we have added 'revision_number' in the feature set so that the model knows at which stage of review this code change belongs. Jiang et al. [11] used patch_no when predicting whether a patchset will be accepted in the git repository of the Linux kernel. Note that this result is not comparable with the one shown in Table 6 because the test set is different. However, the average AUC is still significant.

'Total' presents results when the test fold contains all changes of that fold. 'First revision' presents the result when the test fold only contains changes at their first revision. The last revision means when the code change was finally merged or abandoned. 'Last revision' presents the result when the test fold only contains changes at that revision. Table 11 shows that in all cases the AUC score has improved in the last revision. That is expected because the fate of the code change is almost set at that time.

For the second approach we have used revision number [11], weighted_approval_score, avg_delay_between_revisions [8], and number_of_messages as extra features. Weighted approval score is calculated at each revision by adding label values of previous revisions multiplying by the fraction of current_revision_no and current_revision_no + 1. This will add more weight to the labels in the later revisions.

Avg_delay_between_revisions is calculated in days. Table 12 shows the average test AUC scores achieved during the experiments. Column RIMPR and NIMPR show the improvement and normalized improvement [4,6] of the AUC scores at the last revision compared to the first.

Overall AUC scores and AUC scores in the last revision both have improved in this approach. During the first revision, these previous revision-related features do not exist. However, this result shows that adding previous revision-related features can improve prediction performance in later revisions. Also, we have found that for changes with only one revision the AUC scores are 86%, 85.2%, and 83.5% in project

[11] https://git.eclipse.org/r/c/platform/eclipse.platform.swt/+/167412.
[12] https://gerrit.libreoffice.org/c/core/+/100373.

Table 13
AUC score in each fold.

Fold	LibreOffice	Eclipse	GerritHub
1	82.6	76.6	86.4
2	80.7	82.1	72.9
3	81.0	81.2	80.6
4	80.3	86.9	79.7
5	87.0	86.1	87.5
6	87.5	**88.6**	84.8
7	88.2	84.3	85.5
8	89.6	85.6	89.6
9	90.9	87.1	88.4
10	**91.8**	84.9	**91.1**

LibreOffice, GerritHub, and Eclipse, respectively. However, for changes with multiple revisions, their AUC scores at the last revision (when the change was finally closed) are 98.6%, 89.4%, and 86%, respectively. But since our primary goal is to give better results during the initial submission, we have not focused too much on this point.

RQ4. How well does our approach work while using multiple revisions? PredCR achieves around 78%–88% AUC score when predictions are updated at the submission of each new revision. PredCR can improve prediction at the last revision up to 8%, compared to the prediction performance at the first revision without using previous revision activity-related features. Using previous revision activity-related features can improve the performance up to 15%. So PredCR can be adjusted with significant results to update predictions at later revision.

4.5. How well does the model improve over time? (RQ5)

4.5.1. Motivation

In real-life scenarios, the number of changes will keep increasing over time. Hence, the model can be trained on a larger dataset. But it is important to validate whether increasing the size of the training dataset will increase the performance of PredCR.

4.5.2. Approach

We have followed a longitudinal ten-fold cross-validation setup to calculate model performance in each project. As explained in the approach of RQ1, this validation setup ensures no future data is used during training. The code changes are sorted by their time of creation and the model trained on past data is used to predict future code changes. The performance of subsequent folds of validation presents the outcome of the model over time. Therefore, we have used the results achieved in each fold of the longitudinal cross-validation setup performed in RQ1, to validate whether PredCR performance improves in later folds.

4.5.3. Results

Table 13 shows the prediction performance of the model during each fold by AUC score. The results show that the performance does not monotonically increase over time. However, the performance in the last half is better on average than that in the first half. In the last fold, both LibreOffice and GerritHub achieved the best results. Eclipse achieved the best AUC score in the 6th fold. Average AUC score for LibreOffice, Eclipse, and GerritHub in fold 1–5 are respectively 82%, 82%, and 81.4%. And in fold 6–10 they are 89.2%, 86.0%, and 87.9%. So AUC scores on average improved 9%, 5%, and 8% in the latter half of the longitudinal cross-validations.

Table 14
ER@20% for different K.

K	LibreOffice		Eclipse		GerritHub	
	Ours	Fan et al [4]	Ours	Fan et al [4]	Ours	Fan et al [4]
10	**99.5**	97.3	**97.9**	95.2	**99.4**	98.2
20	**99.2**	96.3	**97.5**	93.9	**99.0**	98.2
30	**98.9**	95.1	**96.8**	93.1	**98.6**	97.3
40	**98.6**	94.4	**96.4**	92.4	**98.0**	96.3
50	**98.1**	93.7	**95.8**	91.4	**97.3**	95.8
60	**97.8**	92.8	**95.2**	90.5	**96.8**	94.7
70	**97.4**	92.0	**94.4**	89.4	**96.2**	93.8
80	**96.6**	90.0	**93.3**	88.1	**95.5**	92.5
90	**95.7**	89.7	**91.8**	85.9	**94.3**	91.2

RQ5. How well does the model improve over time? The longitudinal cross-validation setup sorts data by time and after each fold, one more window is added to the training data, so train data size increases too. In this real-world scenario, PredCR has improved 5%–9% in terms of AUC scores in the latter half of the fold. This validates that, in an active project, with the passage of time, PredCR will be able to improve its performance as more changes are created.

5. Discussions

In this section, we first offer more detailed insights into the performance of PredCR by analyzing the performance based on hyper-parameters and run-time (Section 5.1). We then discuss the implications of PredCR and our study findings to the field of software engineering practitioners and research in Section 5.2.

5.1. A deeper dive into PredCR performance

In Section 5.1.1, we first analyze the effectiveness of PredCR based on the presence of more/fewer code changes.

In Section 5.1.2, we report how much time PredCR takes to train. In Section 5.1.3, we report how the performance of PredCR changes based on different values of hyper-parameters. We have used the PredCR in Section 4 after the hyper-parameter tuning. In Section 5.1.4 we discuss our model results after excluding each dimension from the feature set. Finally, Section 5.1.5 shows the efforts developers spent per code changes in our dataset.

5.1.1. Effectiveness of PredCR with gradual increase in code changes

In this section, we will investigate the performance of PredCR with an increased percentage of inspected code changes. Since reviewing code changes is a costly and time-consuming task, it is not feasible to inspect all the reviews. Like previous studies, we use 20 as the default value for K in ER@K%. To observe the performance of PredCR with increased K, we increase the value from 10 to 90 and repeat the experiment. Table 14 presents that PredCR outperforms Fan et al. [4] for all the projects at every K value. Though the ER is supposed to decrease as K increases (the number of abandoned changes increases in the top K% of the list), still PredCR performs well.

5.1.2. Time efficiency

In this section, we discuss the time needed to train the model and its prediction time. If the model takes too long to predict, then the reviewers would not get the updated predictions in time, thus discouraging them from applying it. Moreover, new changes keep coming, and it would be challenging to update the model if it takes too long. Our used environment provides 16 Gigabytes of Ram, 4 CPU cores, and a 2.20 GHz Intel Xeon CPU. In Table 15, we have presented model training times in seconds. For LibreOffice, Eclipse, and GerritHub,

Table 15
Model training time (seconds) in each fold.

Fold	Libreoffice		Eclipse		GerritHub	
	Ours	Fan's	Ours	Fan's	Ours	Fan's
1	1.64	1.99	1.22	2.85	1.28	1.99
2	2.13	3.09	1.65	5.29	1.77	3.01
3	2.69	4.20	1.82	7.68	1.79	4.07
4	3.07	5.24	1.98	10.1	2.06	5.04
5	3.30	6.32	2.21	12.5	2.38	6.00
6	3.78	7.58	2.46	15.3	2.49	7.10
7	4.19	8.65	2.71	18.0	2.87	8.18
8	4.61	9.75	3.08	20.7	2.90	9.13
9	5.02	10.9	3.22	23.4	2.94	10.2
10	5.17	12.0	3.54	25.8	3.24	11.3
Average	**3.56**	**6.97**	**2.39**	**14.2**	**2.37**	**6.60**

Table 16
Hyper-tuning of PredCR.

Project	n_estimators	learning_rate	AUC	F1(M)	F1(A)
LibreOffice	100	0.1	84.8	94.9	48.5
	100	0.01	85.6	91.7	43.4
	500	0.1	83.3	**96.3**	**48.8**
	500	0.01	**86.0**	94.2	48.1
Eclipse	100	0.1	84.0	92.6	57.6
	100	0.01	83.4	90.9	55.2
	500	0.1	83.1	**93.9**	**59.4**
	500	0.01	**84.3**	92.3	57.7
GerritHub	100	0.1	83.8	93.3	51.6
	100	0.01	83.9	89.1	44.2
	500	0.1	82.8	**95.8**	**56.2**
	500	0.01	**84.6**	92.0	50.0

PredCR training across all 10 folds takes on average 3.56, 2.39, and 2.37 s. Where the state-of-the-art [4] takes on average 6.97, 14.2, and 6.60 s.

5.1.3. Impact of hyper tuning of PredCR

Hyper-tuning the parameters of the selected classifier is needed to ensure best model performance during practical use [33]. We have hyper-tuned the selected LightGBM classifier with varying the number of estimators and learning rate. The results are shown in Table 16. The best parameters are n_estimators = 500 and learning_rate = 0.01 based on AUC score.

5.1.4. Impact of excluding each dimension

This section presents the importance of each feature dimension. We exclude one feature at a time and rerun the experiment. From Table 17 we can see that both the reviewer and author dimensions have significant impacts on the model performance. However, removing the author dimension still gives around 82% AUC score. So if potential bias against new authors becomes a concern, removing this dimension will not make the model unusable.

5.1.5. Developer effort spent for changes

In this section, we show how much effort the developers on average spent on code changes. We consider its duration in days, the number of messages, and the number of revisions as effort. Duration is measured as the number of days spent from the creation of the code change till it gets merged or abandoned. We found there were occasional large values of these metrics and removing such outliers as noises is a standard statistical process [34]. Following Tukey et al. [34], we have removed values outside these two ranges: (a) Lower limit: first quartile −1.5 * IQR (b) Upper limit: third quartile + 1.5 * IQR. Where IQR (Inter Quartile Range) is calculated by subtracting the first quartile from the third. After removing the outliers, we calculated the mean of those values and presented them in Table 18. From Table 18 we see, abandoned changes are generally taking more time to close, have fewer messages and revisions per change. These stats are consistent with the results found by Wang et al. [35] who investigated in detail why code changes get abandoned.

Table 17
Performance of PredCR after excluding one feature dimension at a time.

Project	Excluded dimension	AUC	ER@20%	F1(M)	F1(A)
LibreOffice	Reviewer	70.0	96.9	91.7	26.8
	Author	82.4	98.4	93.5	45.7
	Project	85.9	99.2	93.8	46.8
	Text	85.9	99.2	94.0	47.9
	Code	85.7	99.1	93.9	47.6
Eclipse	Reviewer	68.4	94.1	84.1	33.4
	Author	81.1	96.4	92.9	56.4
	Project	84.0	97.3	91.9	57.3
	Text	84.1	97.5	92.2	57.3
	Code	83.6	97.1	92.3	57.1
GerritHub	Reviewer	73.3	97.9	84.6	30.7
	Author	82.8	98.5	91.3	47.8
	Project	83.5	98.9	91.6	47.7
	Text	84.4	98.9	91.8	49.4
	Code	84.5	98.9	91.8	49.5

Table 18
Developer effort spent on the code changes.

Project	Approach	Duration in days	Messages	Revisions
LibreOffice	Total	0.90	5.81	2.28
	Merged	0.89	5.87	2.36
	Abandoned	0.98	5.21	1.42
Eclipse	Total	2.11	8.75	2.24
	Merged	2.09	9.18	2.36
	Abandoned	2.30	6.43	1.62
GerritHub	Total	1.60	9.16	2.00
	Merged	1.57	9.37	2.04
	Abandoned	1.90	7.50	1.68

Table 19
Performance of PredCR across projects.

Source Project	Target Project	AUC	ER@20%	F1(M)	F1(A)
LibreOffice	Eclipse	64.3	95.1	85.2	23.0
	GerritHub	66.9	92.5	88.7	32.2
Eclipse	LibreOffice	77.5	97.5	84.6	30.6
	GerritHub	76.6	97.8	88.0	36.0
GerritHub	LibreOffice	79.2	98.3	84.4	30.7
	Eclipse	81.1	95.9	87.6	58.2

5.1.6. Cross project performance

For new projects, there might not be enough data to start giving predictions. In those cases, models pre-trained on other projects might be useful during the initial stage of the project. Here we have evaluated PredCR performance in cross-project settings. We have trained the model on a complete dataset of one project and tested it on a complete dataset from another project.

From Table 19 we see that PredCR maintains around considerable performance even across different projects. So PredCR pre-trained on other projects can be effectively used for new projects. Notice that this result is not comparable to the ones from Section 4 as it does not follow longitudinal cross-validation.

5.2. Implications of findings

As described in Section 1, the basic usage scenario of our tool is to provide early warnings to authors, reviewers, and the management about review iterations that will eventually be abandoned. As such, PredCR can be effective for the diverse major stakeholders in software engineering:

(1) **Project Manager and leads** to prioritize code review and code change efforts based on the recommendation from PredCR,
(2) **Software Developers** to save time and efforts by focusing on code changes that will most likely be merged (as predicted by PredCR), and
(3) **Software Engineering Researchers** to further investigate useful features like reviewer dimension in relevant early prediction tools.

Project Manager. This tool can provide benefits to software project management. If the management can predict the negative outcome of a review-iteration early, they may analyze the cause and take necessary steps if required. Multiple reasons may act behind the abandonment of changes such as resource allocation, job environment, efficiency mismatch between the author and the reviewer, and even their relations. Some of these may be addressed well by the early intervention of the management and thus revert the result of a particular review-iteration. Thus the company may save lots of time and resources.

Indeed, code review is a very important aspect of modern software engineering. Large software companies, as well as Open Source projects, are practicing it with care. Researchers are trying to generate insight from large repositories of code review and try to bring efficiency in the process to save the cost of production. In this work, we study a relatively under-studied problem of predicting whether a code change will eventually be merged or abandoned at an early stage of the code review process. We design a machine learning model to apply carefully selected features generated as a result of communication between the developers and the reviewers. Our developed tool PredCR is expected to save wastage of effort or help to recover from being abandoned by inviting early intervention of the management.

Software Developers. The code review process requires serious effort and also is time-consuming. The authors and reviewers involved in a review iteration are likely to get frustrated if they see that their effort goes in vain, i.e., a patchset has to be abandoned wasting their efforts for quite some time. If they get an early indication from our tool that their current review process is going in a negative direction, they may become cautious, seek external/management help, or at least be prepared mentally. If the management makes a decision early, their efforts would be saved. Thus it would benefit the practitioners.

Software Engineering Researchers. Prior SE researchers followed different approaches including statistical methods, parametric models, and machine learning (ML) methods for software effort and duration prediction [36], software cost prediction [37], software fault or defect prediction [38,39], etc. Search-based peer reviewer recommendation [40] is another related area. Different prediction models were introduced in the SE (Software Engineering) domain such as predicting the question quality [41] and response time [42] in Stack Overflow. In these models, the authors exploit the interactions among users in different contexts related to software engineering. Bosu et al. [43] identified factors that lead to useful code reviews. Some prior research suggests that a higher number of reviewers reduces the number of bugs and increases the probability of acceptability [44,45]. The experience of the reviewers sometimes leads to useful code changes [21,45].

In addition to the prediction of our model, other implications from our study are: **(1) Fairness in machine learning models** needs to be ensured before deploying them in practice and their potential bias should be thoroughly investigated **(2) Using Code Review Bots** is important in reducing the number of abandoned changes as well as reviewer workloads.

Fairness in machine learning models is important as lots of manual work in the software life-cycle is being replaced by automated tools. In Section 4.3 we have shown that prediction models can be biased against new developers. This implies that similar issues can happen in software defect prediction [7,27,39], issue classification [26], and other machine learning models if not properly investigated.

Code Review Bots have been widely being used in OSS projects. In Table 8, the feature importance for reviewer dimension shows their significant impact on the code change outcomes. So useful feedback from review tools can help alleviate the reviewer's burden as well as improve change outcome [46]. Our shared raw dataset can be used to more thoroughly analyze how much these tools helped to save maintenance resources.

6. Threats to validity

Threats to internal validity refers to errors in our implementation. We have cross-checked all data mined to ensure the data used is valid and contains all changes available within that period. We have also removed changes for which full data was not available (i.e. some old changes were missing patch data from Gerrit response). We have rerun the pre-processing steps several times to ensure the same statistics of the final dataset. We have removed changes for which the outcome is obvious (i.e. changes labeled "WIP" or "DO NOT MERGE" etc.). So that the dataset only contains changes for which prediction is needed. To make the comparisons compatible with the state-of-the-art, we have followed the process presented in their work and reproduced their feature set and experimentation. Our experiments follow the longitudinal setup, which prevents past data to be used in training. This same setup has been followed by prior works [2,4] in similar scenarios. As the dataset is an imbalanced one, we need to prevent the model from being biased on the majority class. We have balanced classification loss to counter the data imbalance problem. We have also shown the effect of hyper-tuning on model performance. Then made comparisons with the state-of-the-art [4] using all metrics presented by them.

Threats to external validity refers to the generalization of our tool. For PredCR, it is validated by our test results for unseen data (Section 4.1). Also despite having large feature importance for experience-related features, for new contributors, PredCR still outperforms the state-of-the-art (Section 4.3). So there is no threat to use this model even developers who are new or past tracks are missing. Also with the increase of the training dataset, we have shown a positive impact in test results in later half folds of our longitudinal cross-validation result (Section 4.5). Even when prediction is updated for each revision, PredCR shows significant performance (Section 4.4). With different numbers of K, PredCR will still show better predictions than the state-of-the-art (Section 5.1.1). PredCR takes little time to train and test (Section 5.1.2) which validates its practical usability. We have also shown in Section 5.1.6 PredCR pre-trained on a project can still perform well for external projects. Even within projects, we found there are sub-projects from different domains and new sub-projects keep getting added to the project over time. PredCR still maintains a significant overall performance. However, the prioritization given by PredCR does not imply the importance of the code change or how fast it will be closed. So users need to be cautious when using PredCR in such scenarios.

Threats to construct validity refers to the suitability of our evaluation metrics. We have used the evaluation metrics following prior works in the same domain. Both AUC and cost-effectiveness have been widely used in the prediction models of software engineering studies [1,4,7]. We have presented precision, recall, and f1-scores for both classes so that model performance for both of them can be investigated. Also, the metrics have been calculated after averaging over multiple runs of the experimentation. So we believe there is little threat to the validity of our work in practice.

7. Conclusion

Modern code review is an integral part to ensure the quality and timely delivery of software systems. Unfortunately, around 12% of the code changes in a software system are abandoned, i.e, they are not merged to the main code base of the software system. As such, any tool to detect such abandoned changes well in advance can assist software teams with reduced time and effort (e.g., by prioritizing code changes

for review that is most likely going to be merged). In this paper, we present a tool named PredCR that can predict at an early stage of a code review iteration whether a code change would be merged or abandoned eventually. This tool is developed using a LightGBM-based classifier following a supervised learning approach including features related to the reviewer, author, project, text, and code changes and a dataset of 146,612 code changes from three Gerrit open source projects. PredCR outperforms the state-of-the-art tool by Fan et al. [4] by 14%–23% (in terms of AUC score) and achieves around 85% AUC score on average. We have conducted an empirical study on the applicability of PredCR. We find that the new features like reviewer dimensions that are introduced in PredCR are the most informative. We also find that compared to the baseline, PredCR is more effective towards reducing bias against new developers. PredCR uses historical data in the code review repository and as such the performance of PredCR improves as a software system evolves with new and more data. Therefore, PredCR offers more accuracy over the state-of-the-art baseline to early predict merged/abandoned code changes in diverse use cases. As such, PredCR can help to reduce the waste of time and efforts of all stakeholders (e.g., program author, reviewer, project management, etc.) involved in code reviews with early prediction, which can be used to prioritize efforts and time during the triaging of code changes for reviews.

CRediT authorship contribution statement

Khairul Islam: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing. **Toufique Ahmed:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing. **Rifat Shahriyar:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing. **Anindya Iqbal:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing. **Gias Uddin:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

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

[2] G. Jeong, S. Kim, T. Zimmermann, K. Yi, Improving code review by predicting reviewers and acceptance of patches, 2009, pp. 1–18, Research on Software Analysis for Error-Free Computing Center Tech-Memo (ROSAEC MEMO 2009-006).

[3] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: 2013 35th International Conference on Software Engineering, ICSE, IEEE, 2013, pp. 712–721.

[4] Y. Fan, X. Xia, D. Lo, S. Li, Early prediction of merged code changes to prioritize reviewing tasks, Empir. Softw. Eng. (2018) 1–48.

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

[6] C. Costa, J. Figueiredo, A. Sarma, L. Murta, TIPMerge: recommending developers for merging branches, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 998–1002.

[7] T. Jiang, L. Tan, S. Kim, Personalized defect prediction, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, Ieee, 2013, pp. 279–289.

[8] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Review participation in modern code review, Empir. Softw. Eng. 22 (2) (2017) 768–817.

[9] G. Zhao, D.A. da Costa, Y. Zou, Improving the pull requests review process using learning-to-rank algorithms, Empir. Softw. Eng. 24 (4) (2019) 2140–2170.

[10] V. Kovalenko, A. Bacchelli, Code review for newcomers: is it different? Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering, 2018, pp. 29–32.

[11] Y. Jiang, B. Adams, D.M. German, Will my patch make it? and how fast? case study on the linux kernel, in: 2013 10th Working Conference on Mining Software Repositories, MSR, IEEE, 2013, pp. 101–110.

[12] V.J. Hellendoorn, P.T. Devanbu, A. Bacchelli, Will they like this?: Evaluating code contributions with language models, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, 2015, pp. 157–167.

[13] Y. Huang, N. Jia, X. Zhou, K. Hong, X. Chen, Would the patch be quickly merged? in: International Conference on Blockchain and Trustworthy Systems, Springer, 2019, pp. 461–475.

[14] Ç.E. GEREDE, Z. Mazan, Will it pass? Predicting the outcome of a source code review, Turk. J. Electr. Eng. Comput. Sci. 26 (3) (2018) 1343–1353.

[15] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2013) 757–773.

[16] Y. Shin, R. Bell, T. Ostrand, E. Weyuker, Does calling structure information improve the accuracy of fault prediction? in: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, IEEE, 2009, pp. 61–70.

[17] X. Yang, R.G. Kula, N. Yoshida, H. Iida, Mining the modern code review repositories: A dataset of people, process and product, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 460–463.

[18] P.C. Rigby, M.-A. Storey, Understanding broadcast based peer review on open source software projects, in: 2011 33rd International Conference on Software Engineering, ICSE, IEEE, 2011, pp. 541–550.

[19] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, Investigating technical and non-technical factors influencing modern code review, Empir. Softw. Eng. 21 (3) (2016) 932–959.

[20] A. Mockus, D.M. Weiss, Predicting risk of software changes, Bell Labs Tech. J. 5 (2) (2000) 169–180.

[21] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The influence of non-technical factors on code review, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 122–131.

[22] S. Wang, C. Bansal, N. Nagappan, A.A. Philip, Leveraging change intents for characterizing and identifying large-review-effort changes, in: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, 2019, pp. 46–55.

[23] A.E. Hassan, Predicting faults using the complexity of code changes, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 78–88.

[24] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th International Conference on Software Engineering, 2008, pp. 181–190.

[25] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 452–461.

[26] M.S. Rakha, C.-P. Bezemer, A.E. Hassan, Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports, IEEE Trans. Softw. Eng. 44 (12) (2017) 1245–1268.

[27] A.A. Bangash, H. Sahar, A. Hindle, K. Ali, On the time-based conclusion stability of cross-project defect prediction models, Empir. Softw. Eng. 25 (6) (2020) 5047–5083.

[28] J.H. Friedman, Stochastic gradient boosting, Comput. Statist. Data Anal. 38 (4) (2002) 367–378.

[29] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.

[30] P. Geurts, D. Ernst, L. Wehenkel, Extremely randomized trees, Mach. Learn. 63 (1) (2006) 3–42.

[31] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, Lightgbm: A highly efficient gradient boosting decision tree, in: Advances in Neural Information Processing Systems, 2017, pp. 3146–3154.

[32] P. Weißgerber, D. Neu, S. Diehl, Small patches get in! in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, 2008, pp. 67–76.

[33] P. Probst, A.-L. Boulesteix, B. Bischl, Tunability: Importance of hyperparameters of machine learning algorithms, J. Mach. Learn. Res. 20 (53) (2019) 1–32.

[34] J.W. Tukey, et al., Exploratory Data Analysis, Vol. 2, Reading, Mass., 1977.

[35] Q. Wang, X. Xia, D. Lo, S. Li, Why is my code change abandoned? Inf. Softw. Technol. 110 (2019) 108–120.

[36] M. Shepperd, G. Kadoda, Using simulation to evaluate prediction techniques [for software], in: Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International, IEEE, 2001, pp. 349–359.

[37] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Trans. Softw. Eng. 33 (1) (2007).

[38] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.

[39] M. Shepperd, D. Bowes, T. Hall, Researcher bias: The use of machine learning in software defect prediction, IEEE Trans. Softw. Eng. 40 (6) (2014) 603–616.
[40] A. Ouni, R.G. Kula, K. Inoue, Search-based peer reviewers recommendation in modern code review, in: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on, IEEE, 2016, pp. 367–377.
[41] A. Baltadzhieva, G. Chrupała, Predicting the quality of questions on stackoverflow, in: Proceedings of the International Conference Recent Advances in Natural Language Processing, 2015, pp. 32–40.
[42] J. Goderie, B.M. Georgsson, B. van Graafeiland, A. Bacchelli, Eta: Estimated time of answer predicting response time in stack overflow, in: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 414–417.
[43] A. Bosu, M. Greiler, C. Bird, Characteristics of useful code reviews: An empirical study at microsoft, in: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 146–156.
[44] G. Bavota, B. Russo, Four eyes are better than two: On the impact of code reviews on software quality, in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 81–90.
[45] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M.W. Godfrey, Investigating code review quality: Do people and participation matter? in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 111–120.
[46] M. Wessel, A. Serebrenik, I. Wiese, I. Steinmacher, M.A. Gerosa, Effects of adopting code review bots on pull requests to oss projects, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 1–11.