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

Learning to Predict Code Review Completion Time In Modern Code Review

Moataz Chouchen · Ali Ouni · Jefferson Olongo · Mohamed Wiem Mkaouer

Received: date / Accepted: date

Abstract Modern Code Review (MCR) is being adopted in both open source and commercial projects as a common practice. MCR is a widely acknowledged quality assurance practice that allows early detection of defects as well as poor coding practices. It also brings several other benefits such as knowledge sharing, team awareness, and collaboration. For a successful review process, peer reviewers should perform their review tasks promptly while providing relevant feedback about the code change being reviewed. However, in practice, code reviews can experience significant delays to be completed due to various socio-technical factors which can affect the project quality and cost. That is, existing MCR frameworks lack of tool support to help developers estimate the time required to complete a code review before accepting or declining a review request. In this paper, we aim to build and validate an automated approach to predict the code review completion time in the context of MCR to help developers better manage and prioritize their code review tasks. We formulate the prediction of the code review completion time as a learning problem. In particular, we propose a framework based on regression machine learning (ML) models based on 69 features that stem from 8 dimensions to (*i*) effectively estimate the code review completion time, and (*ii*) investigate the main factors influencing code review completion time. We conduct an empirical study on more than 280K code reviews spanning over five projects hosted on *Gerrit*. Results indicate that ML models significantly outperform baseline approaches with a relative improvement ranging from 7% to 49%. Furthermore, our experiments show that features related to the date of the code review request, the previous owner and reviewers activities as well as their

M. Chouchen, A. Ouni, J. Olongo
ETS Montreal, University of Quebec, Montreal, QC, Canada
E-mail: moataz.chouchen.1@ens.etsmtl.ca, ali.ouni@etsmtl.ca

M. W. Mkaouer
Rochester Institute of Technology, NY, USA
E-mail: mwmvse@rit.edu

interactions history are the most important features. Our approach can help further engaging the change owner and reviewers by raising their awareness regarding potential delays based on the predicted code review completion time.

Keywords Modern Code Review · Effort estimation · Machine Learning · Software engineering.

1 Introduction

Code review (originally known as *code inspection*) is a crucial activity in software development where team members review, exchange comments and criticize code changes before merging them into the main codebase [1, 2]. Traditional code review processes were cumbersome and time-consuming, which pushed code review practices to migrate to modern code review (MCR), a lightweight, tool-based and asynchronous process [3]. Since its emergence, MCR has been widely adopted and supported with collaborative cloud-based tools in both open-source and commercial software projects such as *Gerrit*[1] and *ReviewBoard*[2]. MCR has shown its efficiency in several aspects to help developers to improve the quality of their code and reduce post-integration defects [4–6].

While MCR has its advantages, it also presents several challenges in practice. For instance, MCR best practices sets tight limits for code review delays and requires them to be completed promptly [7, 8]. Existing research pointed out that code reviews on Google, Microsoft, and open-source software (OSS) projects take approximately 24 hours [9]. For example, Google recommended code review best practices state that "*we expect feedback from a code review within 24 (working) hours*" [10]. Chromium code review guidelines recommend that they "*aim to provide some kind of actionable response within 24 hours of receipt (not counting weekends and holidays)*"[3]. Indeed, having code quickly reviewed can help developers identify more defects and reduce the deployment delays according to Facebook's previous experiences [11]. However, despite all guidelines and recommendations, having delayed code reviews in OSS projects is still common in the context of MCR. For instance, Patanamon et al. [12] have shown that code reviews can take up to 14 days to be reviewed. Additionally, Macleod et al. [4] showed that when it comes to the challenges faced during the code review, response time is a major concern as authors of code changes are often struggling to get feedback promptly.

To help manage the code review process and reduce the completion time, various approaches have been proposed to recommend appropriate reviewers for pending code reviews [12–17]. However, most of these approaches tend to recommend reviewers who are most experienced with the code change without

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

[2] `https://www.reviewboard.org`

[3] `https://chromium.googlesource.com/chromium/src/+/HEAD/docs/code_reviews.md`

considering the required time to complete the code review. Other studies proposed approaches to prioritize code review requests [18–20] in the context of MCR. However, the estimated completion time is still not considered to better prioritize code reviews. That is, in a typical software project, developers can receive several review requests daily. To better manage and prioritize their code review activities, developers need to estimate the review time for their review requests which can be tedious and error-prone, especially when multiple files are involved in the code change [7, 21, 22]. Hence, inappropriate estimation of code review completion time can lead to unforeseen delays in the whole code review process and software product delivery [5, 14]. This phenomenon has been explored by some prior research works advocating that the code review process can be impacted by several socio-technical factors such as the complexity of the code change, the context of the change, the author's experience and collaboration, and reviewers participation and the communication environment [7, 23–26]. Hence, the estimation of the code review completion time would not be straightforward, given that multiple technical and social factors can interfere with it. Thus, it is crucial to provide efficient and accurate approaches to estimate code review completion.

In this paper, we build an automated approach to predict the time required to perform a code review before accepting or declining a review request, in the context of MCR. More specifically, we formulate the code review completion time prediction as a regression problem and use Machine Learning (ML) techniques to build prediction models to estimate the code review completion time based on multiple technical and non-technical related factors since ML techniques are successful in similar contexts [18–20, 27]. Additionally, we leverage the built prediction model to investigate the most important factors that impact code review completion time.

To evaluate our approach, we perform a large empirical study on 280K code reviews collected from 5 OSS projects to understand whether our ML approach is suitable for predicting the code review completion time and identify the most influential factors on code review completion delays. The obtained results suggest that our approach achieve 0.36, 0.49 and 22.89 in terms of Standard Accuracy (SA), Mean Relative Error (MRE) and Mean Absolute Error (MAE) and improve baselines with up to 50%. Time related features are the most influential features for the code review completion time and other features related to the owner experience and owner - reviewers activities are emerging when recent data is added. This paper makes the following contributions:

- We demonstrate that code review completion time can be accurately predicted with ML.
- We study the performance of our approach under different dimensions and show that the accurate prediction of code review delay requires multiple technical and non-technical factors combined.
- We identify the most important factors related to code review completion time.

– We conduct a large scale empirical study on a dataset of 280k code reviews spanning over five OSS projects and we make our dataset publicly available [28].

The remainder of the paper is organized as follows. In section2, we present the necessary background and discuss the motivation of our approach. In section 3, we discuss the related works to our study. In section4, we introduce our framework and define its components. Section 5 is devoted to presenting the details of our experimental study. Section 6 presents the obtained results for each research question. Section 7 provides more analysis in the performance of our approach. Section 8 discusses the threats to validity of our conclusions and we conclude this paper in Section 9.

2 Background and motivation

2.1 Gerrit code review process

Figure 1 illustrates the MCR process supported by *Gerrit*, one of the most adopted MCR tools. The process starts with a developer submitting a code patch to *Gerrit* (1). *Gerrit* invites reviewers to review the patch (2). The reviewers who accepted the invitation review the patch (3) and the author address reviewers' comments through multiple revisions (4). Finally, a committer who is an experienced developer provides his final decision to 5a) Merge the patch, 5b) Request more revisions, or 5c) Reject the patch.

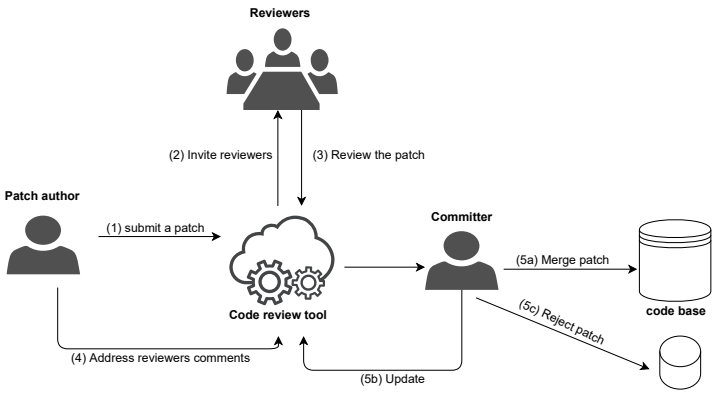


Fig. 1: An illustration of the MCR process.

The Gerrit code review process is a complex process involving several actors and activities. However, inadequately doing code reviews can result in bugs [29], suboptimal design [30] and waste of meaningful resources [18] with both short-term and long-term implications. Prior work has shown that the size

of the code changes has a significant impact on the time taken for code reviews. However, several other factors can also impact code review time. For example, Baysal et al. [23] found that the reviewer's workload and experience can affect the time taken for code reviews. Further other organizational (such as release deadlines) and geographical factors (collaboration across multiple time zones) can also influence the speed. While these factors are critical for faster code reviews they are hard to change. Often, developers are working on multiple projects and features simultaneously. They are simultaneously working on code changes while reviewing other people's code reviews. So, widespread to lose track of pending activities which might be blocking the code change review. This problem is further amplified since these code reviews are spread across multiple repositories. So, in this work, we build a regression model to help developers to estimate the completion time of a given code review request. Our prediction can be used later to alert developers when they go beyond the predicted delay.

2.2 Motivating example

Figure 2 shows an example taken from the Libreoffice project[4] on *Gerrit* illustrating how such delays can occur. The example shows that the code change was submitted on November 19th, by the developer *Pranam Lashkari* (cf. box ① from Figure 2). This change touches 3 files with 33 changed lines in total (13 added and 21 removed) thus it is considered to be a small change since the total number of changed lines does not exceeds 50. Despite being small, this change was abandoned by the Review bot (A bot that abandon code reviews having more than 2 months of inactivity) twice (cf. boxes (2.a) and (2.b) from Figure 2) since there were no activities on the change by the author and the reviewers and restored by the reviewer *Jan Holesovsky* each time thus keeping the author without any feedback for more than four months. Finally, on April 8th, the reviewer *Jan Holesovsky* apologized for the delay taken by the change (cf. box ③ from Figure 2) and approved the change. Form the discussed example, we observe that code review delays can occur and can take up to months. Despite adopting bots to alert developers when a code review is inactive for a long period (*e.g., Review bot* for Libreoffice), these approaches failed to make the developers more engaged since they relay on static thresholds (*e.g.,* 30 days notification period) instead of accurate prediction of a given code review completion time. Therefore, code review delays can be substantially reduced if the authors and reviewers were aware of the needed completion time for a given code review. In particular, the delays could be reduced if there was a mechanism (*e.g.,* an intelligent bot) integrated into *Gerrit* that can predict the code review completion time accurately and alert the responsible developers (*i.e.,* the owner and the reviewers of the code change) when a delay occur during the code review based on accurate predictions. Additionally, project

[4] `https://gerrit.libreoffice.org/c/online/+/83224`

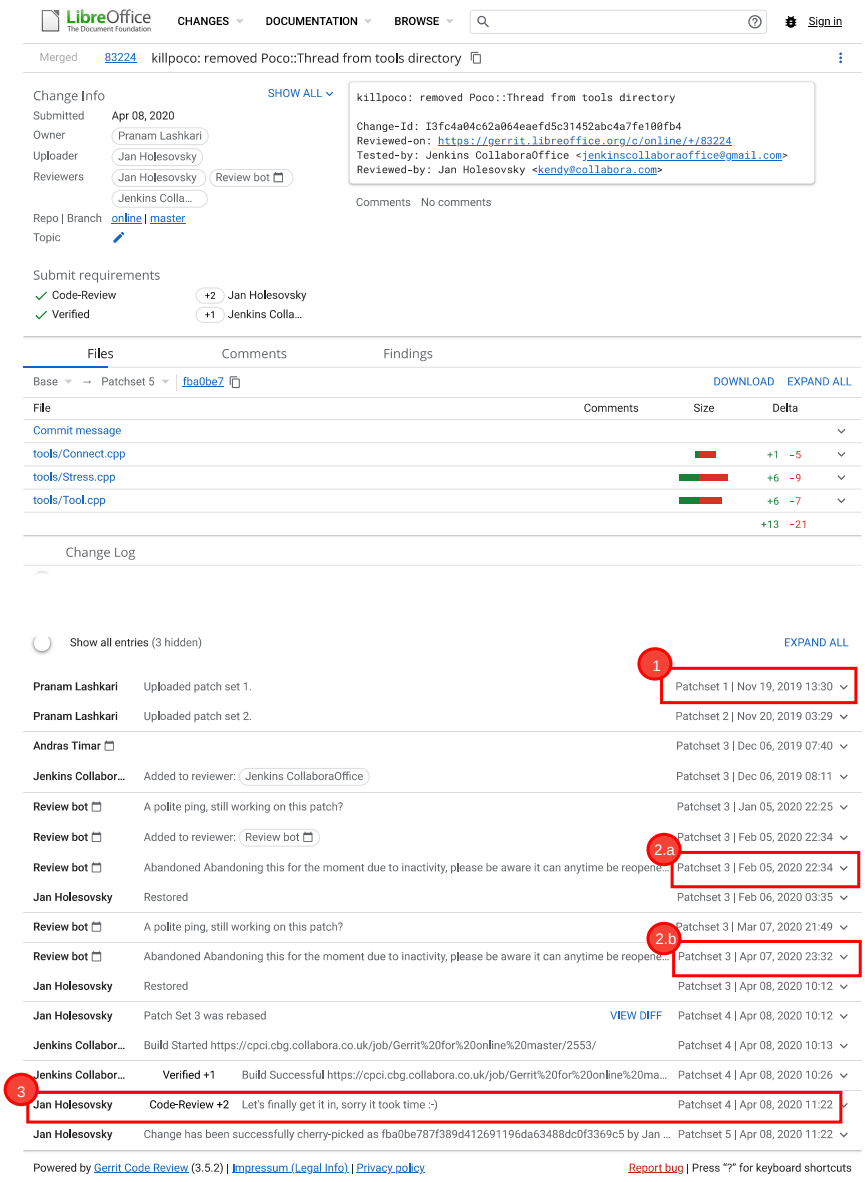


Fig. 2: An example of code review taken from Libreoffice, code change number #83224[4].

managers can use the accurate predictions of our model to prioritize review requests and identify code reviews taking unexpected long time and attempt to address them by either inviting more reviewers for a forgotten changes or by resolving any confusion and conflicts since code review process involves human interactions [25, 31, 32].

3 Related work

Factors influencing code review outcome. Recent works studied various technical and non-technical factors that can affect the code review duration. Jiang et al. [33] showed the number of reviewers might impact the review time. Bosu et al. [34] revealed that core developers receive quicker first feedback on their review requests, and complete the review process in a shorter time. Soares et al. [35] studied factors impacting pull requests (PRs) outcomes. They concluded that factors like a programming language, number of commits, files added and external developer number can influence the outcome of a PR. Additionally, Tsay et al. [36] showed that developers often use both technical and social information when evaluating potential contributions to open-source software projects. Baysal et al. [23] showed through an empirical study on *Webkit* and *Google Blink* that technical (*i.e.,* patch size and component) and non-technical (*i.e.,* organization, author experience, and reviewers' activity) factors may impact the code review duration. Similarly, Thongtanunam et al. [37] showed that code review participation can be influenced by the past activities of the author, the change description length and the nature of the change. Terrell et al. [38] studied gender bias in the context of PR acceptance. Rastogi et al. [39] studied the impact of geographical location on PR acceptance. Later, Egelman et al. [40] studied factors associated with interpersonal conflicts in code review. Recently, AlOmar et al. [24] showed that, in Xeros, refactoring code reviews take longer in terms of duration compared to non-refactoring code reviews.

Previous studies attempted to understand the main factors impacting code review duration. We build upon the results obtained in these studies to extract features characterizing code review completion time in MCR.

Effort estimation in software engineering. Our research is based on effort prior estimation techniques to determine the amount of time needed to conclude a code review request in *Gerrit*. Software effort estimation is a field of software engineering research that has been studied extensively in the past four decades [41–47]. Early attempts of effort estimation in software engineering attempted to use regression analysis to extract the factors influencing software effort needed to complete a software project or a feature [41]. One of the earliest effort estimation models was the COCOMO model proposed by Barry W. Boehm in his 1981 book, Software Engineering Economics [41], which he later updated to COCOMO 2.0 in 1995 [48]. Later, a variety of techniques have been proposed for effort estimation based on ML techniques namely, decision trees [49], support vector machines [50] and genetic programming [51–53].

In the context of MCR, multiple techniques have been proposed to guide developers to prioritize their code review effort. Fan et al. [18] proposed an

approach to detect whether a given code review will be merged, helping developers focus their effort on code reviews that are likely to be merged. Later Saini et al. [19] proposed a tool for prioritizing code reviews based on Bayesian networks. Later, Islam et al. [20] proposed an approach to early predict if a code change is going to be merged and they compared their tool with the previous state-of-art approach by Fan et al. [18]. Additionally, multiple studies attempted to predict the effort required by a code review request. In the context of pull request development, Yu et al. [54] studied the different factors contributing to a given PR. They conclude that PR latency prediction is a complex problem requiring multiple independent factors. Maddila et al. [22,55] proposed a tool for pull request duration based on multiple factors related to the PR itself, the PR author and the development process. Later, Huang et al. [56] use Hidden Markov Models (HMMs) to predict whether a code review takes more than time than the median code review durations. In the context of *Gerrit* development, prior studies attempted to formulate code review effort estimation as a classification problem. Huang et al. [56] defined three classes of code reviews based on the number of revisions: easy if it requires only one revision, regular if it requires between 2 and 6 revisions and hard if requires more than 7 revisions. Later they used code-based features to build a classifier to classify the code review instances into the defined classes. Recently, Wang et al. [57, 58] attempted to use change intents to characterize large effort reviews. They formulate code review effort estimation as a binary classification problem and defined code reviews with large effort as those who have undergone more than two revisions suggested by Microsoft developers. We summarize the details of the presented studies and ours in Table 1. To the best of our knowledge, the first study that attempted to formulate the required time for a PR was released in 2015 meaning that the field is relatively new. Additionally, to the best of our knowledge, no prior study attempted to develop a regression model for code reviews using *Gerrit* code review tool. Prior studies only studied *Github* PRs latency as a regression problem. Since the Gerrit-based code review process and pull request code review processes slightly differ, we fill this gap by examining the code review duration prediction problem in the context of the Gerrit-based code review process. Additionally, to the best of our knowledge, we validate our approach on more than 280K code review instances spanning five open source projects while prior works only use approximately 130K (Wang et al. [58]). Moreover, our study provide the largest metrics for characterizing different aspects of code review activities, with 69 features spanning over eight dimensions (cf. Table 2). Furthermore, prior code review effort studies used 10-fold cross validation that may lead to data leakage (*i.e.,* the model learns from future data to predict values for the past data.) since code review data is time ordered. To mitigate this, we apply an appropriate validation process: Online validation process repeated 100 times similar to previous works [18, 20, 59]. Finally, we release the used data and code of our experiments and make it available for reproducibility and extension [28].

Table 1: Related works summary.

Study	**Year**	**Code review tool**	**Outcome**	**Dataset**	**Evaluation metrics**	**Validation process**	**Features**	**Artifact available**
Yu et al. [54]	2015	Github	Regression: PR latency	GHtorrent(40,848)	R_square	?	4 dimensions, 25 features	No
Maddila et al. [22,55]	2019	Github	Regression: PR latency	10 Microsoft repositories(2875)	MAE, MRE + case study	10 fold cross validation	29 features	No
Huang et al. [56]	2019	Gerrit	Multiclass: based on #revision	Eclipse(9455), OpenDaylight(6403), Total(15,858)	P, R, F1	10 fold cross validation	2 dimensions, 28 features	No
Zhang et al. [60]	2020	Github	Binary: $duration > medianduration$	Vue(?), React(?), React-Native(?), Node(?), Boostrap(?)	P, R, F1, Accuracy	10 fold cross validation	No features	No
Wang et al. [57,58]	2021	Gerrit, microsoft	Binary: $\#revisions > 2$	Qt(23,041), Openstack(6430), android(7120) + Microsoft(100K), Total(136,561)	P, R, F1, AUC + case study	10 fold cross validation	3 dimensions, 15 features + word2vec	Yes
Ours	2022	Gerrit	Regression: Code review completion time	Libreoffice(12777), Qt(44470), Eclipse(20534), Android(63087), Openstack(145418), Total(286277)	SA, MAE, MRE	10 fold online validation repeated 100 times	8 dimensions (cf. Table2), 69 features	Yes

? Not mentioned by the authors.
P, R denote Precision and Recall.

4 Framework overview

In this study, we investigate whether it is possible to formulate code review completion time as a learning problem. Moreover, we investigate the most important factors influencing the completion time of a code review. Our framework depicted is depicted in Figure 3. First, we collect raw code review training data from open source projects that practice MCR, focusing, in particular, on Gerrit-based projects (Step A). Then, we filter our irrelevant data and outliers from the code review data (Step B). Thereafter, different features are calculated for the whole historical data (Step C). This historical data is used later to build and validate our regression models (Step D). Finally, the built regression model will be used to predict the completion time for new code reviews (Step E). Further, we analyze the build model to rank features and explore factors influencing code review duration (Step F). In the following, we explain in more detail each step of our framework.

4.1 Step A: Crawl historical code review data

We start by extracting code reviews data and commits from various open-source software (OSS) projects from the *Gerrit* since it is a widely adopted code review tool for large projects using the Gerrit REST API[5]. Since the first revision commit data is not usually available in the cloned repositories for performance issues according to projects developers, we used the `git fetch` command to fetch the first revision data from the remote servers.

4.2 Step B: Filter historical data

To build robust models with good performance, training and validation data instances, *i.e.,* code reviews should be carefully selected from the studied projects [22]. Hence, we use apply following filtering steps in sequence to clean our datasets:

1. Re-opened code reviews since these reviews have been already reviewed.
2. Since there is no review activity in there, code reviews that are self-reviewed, *i.e.,* self-merged or self-abandoned by the owner of the code review request [26].
3. Code reviews with a short delay, *i.e.,* with delay $\leq$ 12 hours since these reviews can be related to urgent issues (security change or bug with high severity) and should be reviewed at a fast pace [22].
4. Code reviews with long delays, *i.e.,* more than 30 days since these Code reviews can be forgotten by the authors or have a low priority similar to previous studies [22].

[5] `https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html`

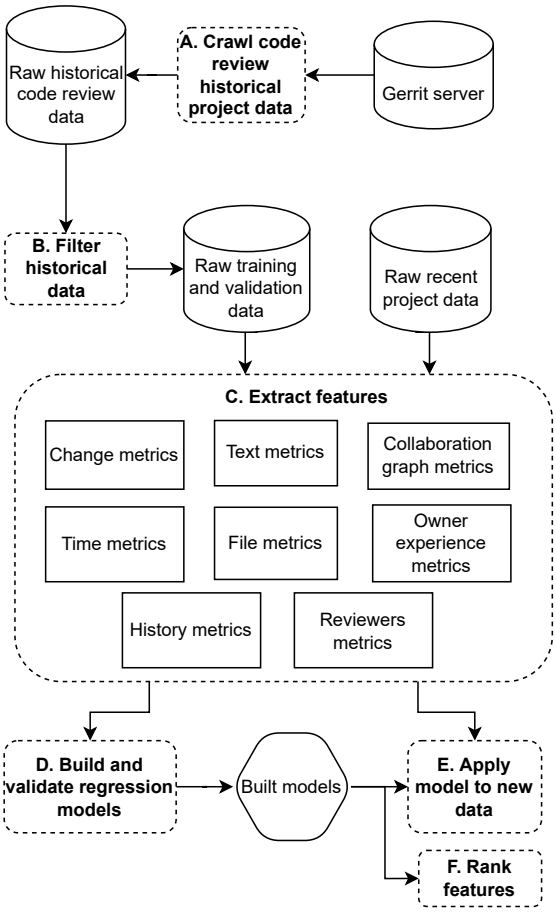


Fig. 3: Framework overview.

5. Outlier code reviews. Like other software engineering datasets such as effort estimation [61], code review data can suffer from outliers. We apply `Boxplot filtering` [61] and `IsolationForest` [62] in cascade similar to previous software effort estimation works [61]. First, we compute the first and the third quantiles Q_1 and Q_3 for all changes completion time and remove changes having duration lower than $Q_1 - (Q_3 - Q_1) * 1.5$ and changes having higher duration than $Q_3 + (Q_3 - Q_1) * 1.5$. In the next step, we remove code review instances identified as 'anomaly' by `IsolationForest` based on their completion time values. We use the default IsolationForest implementation provided by Scikit-learn[6].

After filtering irrelevant code reviews, we store the obtained review data and use it to train and our regression models.

[6] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

4.3 Step C: Extract features

In our study, we use and extend the set of features defined by Fan et al. [18] since it is tightly related to our context. In total, we extracted 69 features that are related to code review completion time prediction. Table 2 summarizes our list of features grouped into eight main dimensions: time features, collaboration graph features, change features, text, file, history, owner and reviewers features. All these features are computed at the submission of the code review request. Although most of our features have been studied in related studies [18,27,33,37], some of the dimensions mentioned previously have been discarded. For example, we discarded the commit dimension introduced by Jiang et al. [33] since it cannot be computed at the submission time of the review request. Similar to Fan et al. [18], when calculating past record-related features, we have generally considered recent activities (in the last 30 days from the studied code review submission day). Fan et al. [18] added the 'recent' prefix to features that were calculated in the last 120 days. Our approach is thus more restrictive in terms of feature history hence it require less computational resources and overhead. In the following sections, we provide the details of the studied dimensions and features.

Table 2: The list of features considered in our studies.

Dimension	Feature name	Rationale/ Conjecture
Time	created_weekday_utc created_weekday_owner_tz created_hours_utc created_hours_owner_tz	Change submission time may impact the responsiveness of the reviewers since the developers may have different time-zones [37, 40].
Collaboration graph	degree_centrality closeness_centrality betweenness_centrality eigenvector_centrality clustering_coefficient core_number	More collaboration between developers makes code review faster.
Change	insertions deletions num_files num_files_type num_directory num_programming_languages modify_entropy code_churn sum_loc sum_complexity num_binary_files	Larger changes are harder to review and more defect prone [63–65]. Touching many files is more defect-prone [66, 67].
Text	subject_length subject_word_count description_length description_word_count is_non_fonctional is_refactoring is_corrective is_preventive has_feature_addition is_merge	Better explained changes are easier to review. Reviewers feedback may depend on the nature of the change [57].
Files	avg_num_dev_modified_files num_files_changes num_files_changes_avg files_changes_duration_avg files_revisions_duration_avg files_num_branch_changes num_file_changes_for_revrs_avg_recent num_file_changes_for_revrs_max_recent file_changes_time_revrs_avg_recent file_changes_time_revrs_max_recent	Previous activities on files may impact the code review process [68].
Owner	num_owner_prior_changes num_owner_open_changes_recent owner_age prior_owner_rate owner_changes_messages owner_changes_messages_avg owner_time_between_message_avg owner_merged_ratio num_prior_owner_project_changes prior_owner_project_changes_ratio owner_revrs_commons_changes_recent owner_revrs_commons_msgs_avg_recent	The changes of more experienced developers are reviewed first [34]. Experienced developers have a higher chances of quickly merging their changes [33, 69]
History	num_prior_changes num_prior_project_changes num_open_changes_recent num_project_open_changes_recent	Previous activities on the project may impact the studied code change duration.
Reviewers	revrs_changes_recent revrs_changes_avg_recent revrs_merged_changes_recent revrs_merged_changes_avg_recent revrs_abandoned_changes_recent revrs_abandoned_changes_avg_recent revrs_open_changes_recent revrs_open_changes_avg_recent revrs_num_review_recent revrs_num_review_avg_recent revrs_previous_msgs_recent revrs_previous_msgs_avg_recent	Reviewers activities and their past record may affect the code review process [37, 70]

- **Time dimension:** Usually, large open-source projects involve developers from different time zones. Prior studies showed that having different time zones in the team may lead to delays in code reviews [37, 40]. To characterize such behavior, we introduced the date dimension with 4 features. The *created_weekday_utc* and *created_hours_utc* measures denote the submission day and hour according to the UTC timezone. The *created_weekday_owner_tz* and *created_hours_owner _tz* measures denote the submission day and hour according to the owner's timezone. Note that the owner timezone is extracted from the `GitPersonInfo` sub-field provided by *Gerrit*[7]
- **Collaboration graph dimension:** Recent studies on code review suggested that the collaboration degree of the code change owners (*e.g.,* their level of participation within the project) can influence code review outcomes. For instance, Baysal et al. [23, 71] suggested that the more active a developer is in the project, the faster and more likely their patches will be merged. Following these studies, we construct a developers network based on the collaborations of owners and reviewers. Thereafter, for each code change owner, we extract features of their corresponding node in the network. We first extract its owner and submission date for a newly submitted code change. Then, following Zanetti et al. [72], we collect all code reviews performed before the submitted code change in a window of 30 days that precede the submission date. Then, we construct an weighted undirected graph based on the owners and reviewers of these code changes. In the graph, nodes denote developers (*i.e.,* owners and reviewers), and edges weights represent their review collaboration frequency, (*i.e.,* for each code change, the change owner and reviewers are connected by an edge and the edge weights increase by one). Subsequently, we extract the Largest Connected Component (LCC) of the generated graph and check whether the code change owner is a member of the LCC. If the code change owner exists in the LCC, we use the six features proposed by Zanetti et al. [72] namely *degree_centrality*, *closeness_centrality*, *betweenness_centrality*, *eigenvector_centrality*, *clustering_coefficient* and *core_number* to quantify the code change owner's degree of activity. All these the metrics are computed using the Python package Networkx[8].
- **Change dimension:** Change metrics are used to characterize the complexity and the volume of the changed code. Prior studies found that larger and complex changes are harder to review [5, 6, 23, 63, 73]. This dimension involves a set of features related to the changes made in the source code and have been investigated in previous works [18, 22, 27]. The features *insertions* and *deletions* denote the number of added and deleted lines. The *code_churn* is the sum of *insertions* and *deletions*. The *num_files*, *num_files_types*, *num_binary_files* and *num_directory* features denote, re-

[7] `https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html#git-person-info`

[8] `https://networkx.org`

spectively, the number of files, the number of file types, the number of binary files and the number of directories modified by this code change. The *num_programming_languages* denotes the total number of programming languages used in the change. The *modify_entropy* is a feature previously proposed by Kamei et al. [74] and 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 studied change. The *sum_loc* and *sum_complexity* features denote the total lines of code and the total cyclomatic complexity of the changed files. We extracted these features using Pydriller[9].

- **Text dimension:** Text related features are mainly extracted from the change description and title provided with a submitted code change. The aim is to identify the purpose of the code change. The subject is provided in the field *subject* while the description is provided under the field *message* of the first commit of the code change when it is created. *subject_length* and *description_length* are the number of characters in the change subject and description respectively. The *subject_word_count* and *description_word_count* represent the word counts for the subject and description of the change, respectively. The other six features's values are binary, *i.e.,* 0 or 1. These features are computed based on the previous work of Hindle et al. [75] where for each change, we mark the category as 1 if one of the keywords in the Table 3 exists in the change description to classify them into non-functional, refactoring, corrective, preventive or a new feature code change.
- **File dimension:** This dimension consists of a set of features based on the file modification history as recorded within *Gerrit*. For instance, Graves et al. [76] found that several previous changes to a file are a good indicator to detect buggy files. Furthermore, Matsumoto et al. [77] found that files which more developers previously touch are more likely to induce defects. In our context, we conjecture that previous review activity on files can be a good predictor of the code review completion time. The *avg_num_dev_modified_files* feature is the average number of developers who modified a code change files. The *num_files_changes* is the total number of changes affecting at least one of the current change files. The *num_files_changes_avg* is the average of the number of the changes involving the present change files. The *files_changes_duration_avg* is the average code reviews completion time of the changes involving at least one modified file of the current change. Similarly, *files_revisions_duration_avg* is the average completion time of revisions (*i.e.,* Patchsets) involving at least one file of the present change files. The *files_num_recent_branch_changes* is the total number of branches in previous file changes. The features *num_file_changes_for_revrs_avg_recent* and *num_file_changes_fo_ revrs_max_recent* are the averages and the max of changes involving at least one of the present change files involving one of the potential reviewers as a reviewer (cf. Sec-

[9] `https://pydriller.readthedocs.io`

tion 4.3) in the last 30 days. Similarly, *file_changes_time_revrs_avg_recent* and *file_changes_time_revrs_max_recent* denote the average and the max of code reviews completion times involving at least one file of the present change files and one potential reviewer as a reviewer.

- **Owner dimension:** The owner related features characterize the experience and the history of the activities of the owner of the present change. Previous works showed that the owner's experience could be highly significant in predicting software failures [64] and code review outcomes [33]. Similar to prior works [18,27], we use various owner features to predict the code review completion time. The *num_owner_prior_changes* and *num_owner_prior _changes_recent* features denote the total number of the changes done by the owner previously and in the last 30 days. The *owner_age* is the number of days counting from the creation of the owner account until the date of the change submission. The *prior_owner_rate* denotes the number of changes done by the owner per day. The *owner_changes_messages* is the total number of messages exchanged in previous owner's contributions. Similarly, the *owner_changes_messages_avg* denotes the number of messages in the owner prior changes per change. The *owner_merged_ratio* denotes the ratio of the number of merged changes of the owner to the *num_owner_prior_change*. The *num_prior_ owner_project_changes* denotes the number of prior changes done by the owner in the same repository of the current change. Additionally, the *prior_owner_project_changes_ratio* denotes the ratio of the number of changes performed by the owner to the repository of the current change by the total number of changes performed by the owner. Finally, the *owner_revrs_commons_ changes_recent* and *owner_revrs commons_ msgs_avg_recent* denote the common changes number and the total number of exchanged messages between the owner and the potential reviewers in the last 30 days.
- **History dimension:** The history dimension is devoted to characterizing the evolution of the studied project and repository of the present change. *num_prior_changes* is the total number of code changes done before the studied change. Similarly, *num_prior _changes_project* is the total number of code changes done before the studied change acting on the same repository of the present change. Similarly, *num_open _changes_recent* denotes the total number of currently reviewed changes submitted in the last 30 days (counting from the date of the submission of the studied code change) while *num_project_open_changes_recent* is the number of the currently reviewed changes to the studied change's repository in the last 30 days.
- **Reviewers dimension:** Prior studies showed that reviewers' records and activities might impact the outcome of a given code review [37, 70]. Since the reviewers of a given code change might not be known when a code review request is submitted, we define the potential reviewers as the top 5 reviewers with the highest number of reviews for the present change owner. We compute the metrics for this dimension based on the potential reviewers. All these metrics are computed based on their activities in the last 30 days (where the prefix *recent* is added for all of them). The

revrs_changes_recent is the total number of changes authored by the potential reviewers while the *revrs_changes_avg_recent* is the average of reviewed changes per potential reviewer. Similarly, the *revrs_merged_changes_recent*, *revrs_merged_changes _avg_recent*, *revrs_abandoned_changes_recent* and *revrs_abandoned _changes _avg_recent* indicate the sums and the averages of the merged and abandoned prior changes authored by the potential reviewers. The *revrs_open changes_recent* and *revrs open_changes_avg_recent* denote the sum and average of the recently opened code changes by potential reviewers. The *revrs_num_review_recent* and *revrs_num_review_avg_recent* denote the sum and the average of the prior reviewed changes by the potential reviewers. Finally, the *revrs_previous _msgs_recent* and *revrs_previous_msgs_avg_recent* denote, respectively, the total number and the average number of messages (*i.e.,* comments) written by the potential reviewers in the recently reviewed code changes.

Table 3: Used keywords for each code change intent based on the work of Hindle et al. [75]

Type	**Keyword**
is_non_functional	doc*, docs, document, documentation, help.
has_feature_addition	new, add, requirement, initial, create, implement, implementing, insert, inserted, insertions.
is_corrective	bug, wrong, problem, patch, fix*, fixed, fixes, fixing, close*, closed, closes, closing, bug*, bugs, error, errors, defect, defects, issue*, fault*, fail*, crash*, except, fail*, resolves, resolved, solve*, crash, make sure.
is_merge	merge, merg*, combine, unite, join, consolidate, incorporate, fuse, mix*.
is_preventive	test*, junit, coverage, asset, unit*, unit test, UnitTest, prevent*, cppunit
is_refactoring	clean, better, bugs, cleanup, cohesion, comment, complexity, consistency, decouple, duplicate, fix, performance, readability, redundancy, regression, risks, simply, testing, uncomment, modernize, modif, modularize, optimize, refactor, remove, rewrite, restructure, reorganize, generalise, ->, =>, move, deprecated, deprecate, code, readability, test, bad code, bad management, best practice, break out, extract class, loose coupling, not documented, open close, splitting logic, strategy pattern, single responsibility, ease of use, improving code quality, remove legacy code, replace hard coded, stress test results, single level of abstraction per function, allow easier integration with

(ABC*) denotes any word starting with ABC

4.4 Step D: Build and validate regression models

In this step, we build and validate various regression models to predict the code review completion time (in hours). In our approach, the outcome variable is

Table 4: ML pre-processing steps configurations.

Step	Scikit/Python algorithm
(1) Log transformation	`np.log1p`
(2) Data standardization	`sklearn.preprocessing.RobustScaler`
(2) Quantile Transformation	`sklearn.preprocessing.QuantileTransformer(n_quantiles=1000, output_distribution = 'uniform')`
(4) PCA	`sklearn.decomposition.PCA(n_components = 40,whiten =True,svd_solver = 'full')`

the difference $T_c - T_s$, where T_c denotes the time when the code review is completed (*i.e.,* either merged or abandoned), and T_s denotes the time of the creation of a given code review. We use the historical data extracted in steps A and B to build various regression models and validate their performances. We build and validate our ML models using *Scikit learn*[10]. Before training the regressor, we follow a four-step data pre-processing approach as summarized in Table 4. In particular, our data pre-processing consists of the following steps:

Log transformation. Code review data is highly skewed data. Therefore we apply log transformation to reduce the skew effect in the data similar to prior studies [78, 79]. Hence, we use the `np.lop1p` function from the Numpy[11] package as presented in Table 4 to perform the log transformation $y = log(x + 1)$ for the outcome variable and the features since some values may have near 0 values. Note that to compute the prediction performance metrics (defined later in Section 5.4.1), we transform back the outcome variable and the predictions of our models to their original values and scale using the `np.exp1p` function, *i.e.,* the inverse of the `np.lop1p` function.

Data standardization. ML regressors typically require features to have a close scale [80]. The scale difference between features can influence the performance of an ML regressor. Since code review data can have noise, we use Sklearn's `RobustScaler` which uses Median and IQR to center and scale the data instead of mean and variance.

Quantile transformation. Since our approach involves a large number of features, some features may have different scales and distributions due to the noise in the data related to the human aspect in the code review activities. Therefore, we transform our features and the outcome variable to have a uniform distribution using the `QuantileTransformer` function from Sklearn, similar to prior studies [81]. Note that the outcome variable values and the predictions of our model are restored to their original values and scales using the inverse transformations based on Sklearn.

[10] `https://scikit-learn.org/stable/index.html`
[11] `https://numpy.org/`

Principle Component Analysis (PCA). Since our learning problem involves highly dimensional code review data, our datasets may suffer from multi-collinearity and redundancy similar to previous studies [18]. To alleviate such a negative impact, we perform the principle component analysis (PCA) since it was widely adopted in previous software engineering prediction problems [82, 83]. We use the `PCA` function from Sklearn to apply PCA for on our features. Note that we only transformed numerical variables and excluded binary variables (*i.e.,* most of the Text dimension metrics).

After performing data preprocessing steps, we train and validate our regression model to learn to predict the code review completion time. In this step, we study the performance of multiple ML regressors namely, *Extra Trees* (*ET*), *Random Forest* (*RF*), *Decision Tree* (*DT*), *Extreme Boost* (*XGBOOST*) and *AdaBoost* with *DT* (*AdaBoost*). All of the studied models are tree-based ML models. This choice is motivated by two reasons: (*i*) Tree models does not have any hypothesis regarding the outcome distribution thus beneficial in our case since we are dealing with highly skewed outcome variable (cf. Section 5.3 and Table 6). (*ii*) The tree models can provide meaningful insights regarding the most influential features based on Gini score [84]. Thus, it can be used to explain the predictions of the models. Note that fitting ML regressors involves learning their hyper-parameters since their values have proven to affect ML algorithms' performances in several software engineering prediction problems [20, 85–88]. To this end, we apply grid search to learn the best hyper-parameters for each algorithm to ensure a fair comparison. Table 5 shows the studied hyper-parameters for each studied ML algorithm. For each studied project, we perform a grid search to search for the optimal parameters for each ML algorithm and then we compare the studied ML algorithms using their optimal setting. We use SA (cf. Section 5.4.1) to choose the best setting for ML algorithms *i.e.,* for each ML algorithm, we use the hyper-parameters leading to the highest SA.

4.5 Step E: Apply the built model to new data

After training and validating the performance of each ML regressor, we select the best regressor model based on evaluation metrics defined in 5.4.1. The selected model can be deployed in the code review process tool and can be applied to predict the completion time for a newly submitted code review request.

4.6 Step F: Rank features

One of the goals of our study is to understand the most influential features impacting the delay of a code review. Knowing this piece of information can help developers and project managers to plan informed retro-actions leading to predicting and reducing review delays. To measure the importance of the

Table 5: Optimized hyperparameters for each ML algorithm.

ML model	Parameter	Values
Extra Trees	`criterion`	['squared_error']
	`max_depth`	[5, 10, None]
	`n_estimators`	[100, 500]
Random Forest	`criterion`	['squared_error', 'poisson']
	`max_depth`	[5, 10, None]
	`n_estimators`	[100, 500]
XGBOOST	`max_depth`	[5, 10, None]
	`n_estimators`	[100, 500]
	`learning_rate`	[0.01, 0.1, 0.3]
AdaBoost	`n_estimators`	[100, 500]
	`learning_rate`	[0.01, 0.1, 0.3]
	`loss`	['linear', 'square', 'exponential']
Decision Tree	`criterion`	['squared_error', 'friedman_mse', 'absolute_error', 'poisson']
	`max_depth`	[5, 10, None]
	`splitter`	['best', 'random']

features, we leverage the previously trained ML regressor and use it to extract feature importance to get a meaningful insights about the features that impact the code review completion time. Since our studied are based on tree models, we use their Gini importance values [84] to rank learn features influence on our outcome similar to previous works [18, 20]. More details are provided in Section 6.3.2.

5 Empirical study design

In this section, we start by setting the research questions. Thereafter, we describe the experimental process that will drive our study to answer our research questions.

5.1 Research questions

The main goal of our approach is to investigate whether it is possible to learn and predict the completion time of a given code review. In addition, we aim to further investigate the main factors that impact the delay of a code review. Our research is driven by the following research questions:

RQ1. **(ML algorithms performance):** *How effective are machine learning models for code review duration prediction?*
RQ2. **(All features vs features dimensions):** *How effective is our prediction model when all features are used as compared to when only a dimension is used?*

***RQ3.* (Features importance analysis):** *What are the most critical features influencing code review completion time?*

5.2 Experimental setting overview

In practice, code review data changes over time, especially for long-lived projects. Thus, it is important to consider the code review completion time prediction as an online learning problem since it is more similar to what happens in practice [18,20,59,89]. To measure the performance of the different ML algorithms, we use the online validation process as depicted in Figure 4. Initially, we sort code reviews according to their creation time in an ascending order. Then, we split the whole dataset into 11 folds of equal sizes. Thereafter, we select the last fold i as testing data while we keep the earlier folds $fold_{0,1,2..,i-1}$ as the training set. This process is repeated 100 times to alleviate the stochastic nature of the data as well as the ML algorithms.

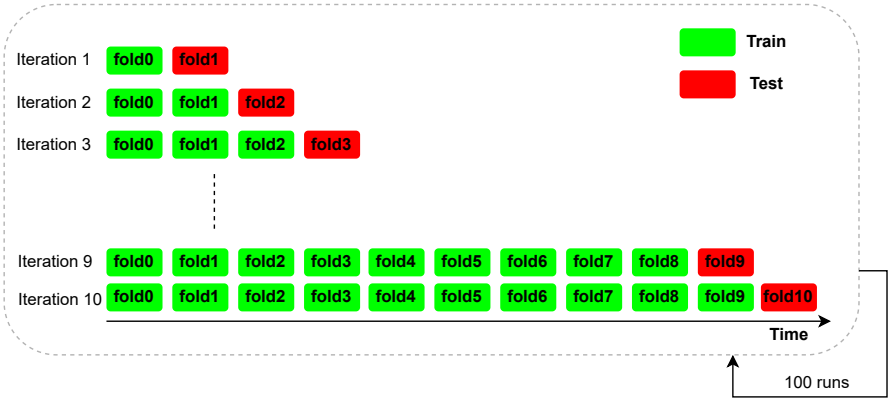


Fig. 4: Online validation process

5.3 Data collection

We validate the results of our study using five OSS projects, namely, Android[12], QT[13], Eclipse[14], LibreOffice[15] and Openstack[16] projects data since they have been widely studied in previous research in MCR [13, 14, 18, 26, 90].

[12] https://android-review.googlesource.com
[13] https://codereview.qt-project.org/
[14] https://git.eclipse.org
[15] https://gerrit.libreoffice.org
[16] https://review.opendev.org/

All data have been collected from 01-01-2015 to 27-04-2021. We present in Table 6 a summary of the collected data that we plan to use in our experiments. The same filtering steps discussed in Section 4.2 are applied to all five projects. Table 6 presents statistics of the final collected datasets. In Table 6, we also provide some descriptive statistics related to the distribution of the outcome variable (*i.e.,* the code review completion time) for each project. In particular, we provide the skewness [91] and the kurtosis [92] values of the outcome variable to understand the location and variability of our outcome variable. From Table 6 we can read that most of the skewness values are close to one which indicate high right skewness. In particular, the projects Libreoffice and Eclipse have a skewness values higher than 1 which is interpreted as a sever skewness. Additionally, we observe that only Libreoffice has a positive kurtosis value indicating that its outcome variable distribution is characterized as leptokurtic distribution [92] having a heavier tail compared to a normal distribution. These characteristics indicate that learning to predict such outcome variable is challenging due to the high skewness and positive kurtosis (for Libreoffice). Thus our pre-processing steps and the choice of tree-based models are motivated to reduce such impact.

Table 6: Descriptive statistics of the studied projects.

Project	**#Changes**	**#Accounts**	**Avg (Std) outcome***	**Max outcome***	**Skewness**	**Kurtosis**
Android	63,078	4,197	51.45 (36.69)	145.04	0.98	-0.18
Qt	44,470	1,315	53.27 (36.37)	146.94	0.99	-0.07
Eclipse	20,534	1,378	58.17 (43.92)	169.12	1.04	-0.07
Openstack	145,418	8,177	46.31 (29.62)	123.57	0.92	-0.19
Libreoffice	12,777	612	36.38 (20.48)	96.29	1.04	0.25

*Mean, Std and Max of the outcome are expressed in hours.

5.4 Performance metrics

5.4.1 Standard performance metrics

To evaluate our different ML algorithms, we use three standard regression metrics: Mean Absolute Error (MAE), Mean Relative Error (MRE) and Standard Accuracy (SA) and and since they have been commonly used in a similar context [22].

Given a project with n code review instances, each is characterized by actual duration ($actual_i$) and estimated duration ($estimated_i$). MAE and MRE across the n code review instances are respectively defined as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|actual_i - estimated_i| \quad (1)$$

$$MRE = \frac{1}{n}\sum_{i=1}^{n}\frac{|actual_i - estimated_i|}{max(\epsilon, |actual_i|)} \tag{2}$$

where ϵ is an arbitrary small yet strictly positive number to avoid undefined results when $actual_i$ is zero.

SA is widely used in evaluating regression models in software engineering [53, 93, 94]. SA is based on MAE and defined as follows:

$$SA = 1 - \frac{MAE_{p_i}}{MAE_{p_0}} \tag{3}$$

where MAE_{p_i} is the MAE of the prediction model p_i being evaluated and MAE_{p_0} is the MAE of a large number (usually 1,000 runs) of random guessing. For a prediction model p_i whose accuracy outperforms random guessing, SA will produce a number in the range [0, 1]. An SA value closer to zero is discouraging since it means that the predictor p_i performs just a little better than random guessing [52]. SA can also produce negative values, for those predictors which are performing even worse than random guessing. A high-performance prediction model should have a lower MAE, lower MRE, and higher SA than its competitors.

5.4.2 Improvement analysis metric

To report the improvement of our approach compared to baseline, we use to Relative Improvement (RIMPR) previously used by Fan et al. [18] and Islam et al. [20]. 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 an SA score from 0.25 to 0.40 is only a 0.15 increase in score. But only calculating the difference misses the fact that the new score is almost the double the previous score. However, the improvement here is 60% which clearly shows that fact. RIMPR is calculated as follows:

$$RIMPR = \frac{new\_score - old\_score}{old\_score} \tag{4}$$

5.5 Statistical analysis method

Since our experiments are based on ranking different configurations, we use the Scott-Knott Effect Size Difference (SK-ESD) test elaborated by Tantithamthavorn et al. [95] to find statistically distinct ranks. SK-ESD extends the Scott-Knott test with normality and effect size corrections as follows:

- Normality correction: SK-ESD applies log-transformation $y = log(x + 1)$ to alleviate the skewness of the data.

– Effect size correction: SK-ESD uses Cohen's d [96] to measure the effect size between different clusters and merge clusters having negligible effect size, *i.e.,* having $d < 0.2$.

In this study, we use the implementation provided by Tantithamthavorn et al.[17] since it was updated according to the recommendation of Herbold [97] allowing us to verify Scott-Knott assumptions before applying the test.

6 Empirical study results

In this section, we provide results answering our research questions outlined in Section 5.1. Answering RQ1 allows us to validate the performance of our approach and determine the best regression model. Answering RQ2 allows us to compare our approach when using all features against focusing on only one dimension thus justifying the need of using all features instead of a single dimension. Finally, answering RQ3 provides more granular insights regarding the most important features influencing the code review completion time. For the sake of the replicability, we made our datasets and our approach's code available [28].

6.1 How effective are machine learning models for code review duration prediction? (RQ1)

6.1.1 Motivation

This RQ act as a sanity check of our approach. The main goals of RQ1 are to (*i*) identify the best ML regressor for the task that we will use in the later analysis and check (*ii*) whether our approach is suitable to accurately predict the code review completion time.

6.1.2 Approach

To answer RQ1, we compare the various regressors including *Extra Trees*, *Random Forest*, *XGBOOST*, *AdaBoost*, and *Decision Tree* (cf. Section 4.4) to identify the best regressor for our problem instance. Additionally, we check whether our approach is suitable to accurately predict the code review completion time as compared to three baselines commonly used in other software engineering problems: *Mean*, *Median* and *Random Guess* [52, 53, 93]. *Random Guess* is a naive benchmark used to assess if an estimation model is useful [98]. *Random Guess* performs random sampling (with equal probability) over the set of code review instances with known completion time. It chooses randomly one instance from the sample, and uses the completion time of that code review instance as the estimate of the target instance. *Random Guess* does not

[17] https://github.com/klainfo/ScottKnottESD/tree/master

use any information associated with the target code review instance. Thus, any useful estimation model should outperform *Random Guess*. *Mean* and *Median* completion time estimations use the mean or median completion time of the past code reviews to estimate the completion time of the target code review. Note that the samples used for all the naive baselines (*i.e., Mean*, *Median* and *Random Guess*) were from the training set. To compare the performance of the different algorithms, we follow the experimental process established in Section 5.2. We evaluate all ML regressors using our three performance metrics MAE, MRE, and SA (cf. Section 5.4.1). Finally, we rank all ML regressors using SK-ESD (cf. Section 5.5).

6.1.3 Results

ML regressors comparison: Table 7 outlines the obtained results for the considered regressors across the five studied projects. The results with group 1 (*i.e.,* best results) are highlighted in bold for each project. The obtained results indicate that *RF* and *ET* have competitive results across all projects achieving higher performance compared to the other ML regressors. From Table 7 we can see that while *ET* achieves SA scores ranging from 0.33 (Libreoffice) to 0.39 (Android) having an SK-ESD rank of 1 for each project. *RF* achieves very similar SA score and ranks ranging from , 0.33 (Libreoffice) to 0.39 (Android). The same observation also applies to MRE and MAE. In terms of SA we notice that *ET* achieves relatively lower score of 0.33 for Libreoffice compared to the other projects (0.35 - 0.39) this can be explained by the some characteristics related to the distribution of the code reviews completion time. For instance, for Libreoffice, we can read from Table 6 that its outcome variable distribution has a skewness value above 1 that can be interpreted as substantially right skewness in the distribution [91]. Additionally, Libreoffice has a positive kurtosis (compared to negative kurtosis values for the other projects) which can be interpreted as having a heavier tail and is called a leptokurtic distribution [92]. These characteristics make it harder for the learning algorithm to learn generalizing on such data, thus, lesser performance is expected. Additionally, we observe that while our approach achieve MRE scores ranging from 0.46 (Android) and 0.50 (Openstack), MRE score for Eclipse (*i.e.,* 0.56) is slightly higher. This can be explained by the higher Std value compared to other projects (cf. Table 6). The same observation applies also to MAE where MAE values for Eclipse are slightly higher than those achieved in the other projects.

Overall, we observe that *ET* achieve 0.36, 0.49 and 22.89 with an SK-ESD rank of 1 for SA, MRE and MAE. In our context, these performances are acceptable since we are dealing with highly skewed data (Skewness values are close to 1 for all projects according to Table 6) RF achieves competitve scores with *ET* and both *ET* and *RF* outperform the other ML regressors. In our study, both *ET* and *RF* achieve competitive scores however *ET* are known to be more computationally efficient [99] hence we adopt it as the underlying

model for our approach and it throughout the remaining parts of the study. In Section 7.4 we present more details about the time efficiency of our approach.

Comparaison with baselines: To understand whether our approach is suitable for code review completion time estimation, we compare the performance of our approach against three baselines: *Median*, *Mean* and *Random Guess*. We report the obtained SA, MRE and MAE scores of our approach with the baselines in Table 7. Table 7 shows that *ET* significantly outperforms all the studied baselines achieving better SK-ESD ranks in all the studied projects for all performance metrics. For example, we can see that for the project Qt, *ET* achieves an SA score of 0.39 compared to 0.27 for *Median* and *Mean* and and 0 for Random Guess. Additionally, *ET* achieves an MRE score of 0.49 compared to 0.65 for both *Median* and *Mean*, and 1.02 for *Random Guess*. Furthermore, Table 8 shows the RIMPR of *Extra Tress* compared to the baselines. Overall, we have the following observations:

- In terms of SA, our approach achieves a RIMPR between 20% (Libreoffice) and 50% (Android) as compared to *Median* and *Mean* baselines.
- In terms of MRE, our approach acheives a RIMPR between 5% (Libreoffice) and 29% (Android) compared to Mean and Median baselines and RIMPR a between 46% (Libreoffice) and 53% (Android) compared to *Random Guess* baseline.
- In terms of MAE, our approach acheives a RIMPR between 7.5% (Libreoffice) and 17.3% (Android) compared to Mean and Median baselines and RIMPR between 31% (Libreoffice) and 39% (Android) compared to *Random Guess* baseline.

Note that these results are in accordance with previous works by Maddila et al. [22] who only achieved a RIMPR of 17.7% and 11.8% for MRE and MAE, respectively compared to *Mean*.

RQ1 summary: The *Extra Trees* model achieves the best performance among the other regressors as well as the considered baselines with a score of 0.36 in terms of SA, 0.48 of MRE and 22.89 of MAE. Thus, our formulation successfully passes the sanity check required for RQ1.

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

6.2.1 Motivation

In section 4.3, we defined 8 dimensions characterizing code review from different aspects. The main goal of this RQ is to understand the performance of our approach when only one features dimensions is used each time. Thus, we can validate whether our approach can benefit from all the features compared to

Table 7: The achieved performance by each models across all projects.

Project	Model	SA	MRE	MAE
Android	ExtraTrees	**0.388 (1)**	**0.463 (1)**	**24.101 (1)**
	RandomForest	**0.388 (1)**	**0.465 (1)**	**24.123 (1)**
	AdaBoost	0.385 (2)	**0.463 (1)**	24.246 (2)
	DecisionTree	0.381 (3)	0.474 (3)	24.397 (2)
	Mean	0.261 (4)	0.653 (4)	29.135 (3)
	Median	0.261 (4)	0.65 (4)	29.139 (3)
	XGBOOST	**0.389 (1)**	0.469 (2)	**24.095 (1)**
	RandomGuess	-0.001 (5)	1.057 (5)	39.46 (4)
Qt	ExtraTrees	**0.386 (1)**	**0.491 (1)**	**24.158 (1)**
	RandomForest	**0.386 (1)**	**0.491 (1)**	**24.162 (1)**
	AdaBoost	0.375 (2)	0.517 (3)	24.568 (2)
	DecisionTree	0.375 (2)	0.508 (2)	24.565 (2)
	Mean	0.272 (3)	0.647 (4)	28.606 (3)
	Median	0.272 (3)	0.65 (5)	28.621 (3)
	XGBOOST	**0.385 (1)**	**0.493 (1)**	**24.188 (1)**
	RandomGuess	0.0 (4)	1.017 (6)	39.304 (4)
Eclipse	ExtraTrees	**0.353 (1)**	**0.564 (1)**	**30.495 (1)**
	RandomForest	**0.351 (1)**	**0.564 (1)**	**30.557 (1)**
	AdaBoost	0.343 (2)	0.583 (3)	30.951 (2)
	DecisionTree	0.337 (3)	0.59 (4)	31.214 (2)
	Mean	0.275 (4)	0.699 (5)	34.167 (3)
	Median	0.274 (4)	0.702 (5)	34.169 (3)
	XGBOOST	**0.349 (1)**	0.574 (2)	**30.668 (1)**
	RandomGuess	-0.003 (5)	1.185 (6)	47.2 (4)
Openstack	ExtraTrees	**0.356 (1)**	**0.499 (1)**	**20.827 (1)**
	RandomForest	**0.356 (1)**	**0.498 (1)**	**20.827 (1)**
	AdaBoost	0.345 (4)	0.508 (2)	21.192 (3)
	DecisionTree	0.352 (3)	0.506 (2)	20.986 (2)
	Mean	0.264 (5)	0.583 (3)	23.816 (4)
	Median	0.264 (5)	0.581 (3)	23.817 (4)
	XGBOOST	0.355 (2)	**0.498 (1)**	**20.889 (1)**
	RandomGuess	-0.0 (6)	0.93 (4)	32.376 (5)
Libreoffice	ExtraTrees	**0.328 (1)**	**0.409 (1)**	**14.863 (1)**
	RandomForest	**0.326 (1)**	**0.409 (1)**	**14.905 (1)**
	AdaBoost	0.319 (2)	0.414 (2)	15.069 (2)
	DecisionTree	0.299 (3)	0.437 (4)	15.487 (3)
	Mean	0.273 (4)	0.429 (3)	16.076 (4)
	Median	0.272 (4)	0.436 (4)	16.086 (4)
	XGBOOST	0.322 (2)	0.413 (2)	14.981 (2)
	RandomGuess	-0.001 (5)	0.726 (5)	22.122 (5)
All	ExtraTrees	**0.362 (1)**	**0.485 (1)**	**22.889 (1)**
	RandomForest	**0.361 (1)**	**0.486 (1)**	**22.915 (1)**
	AdaBoost	0.353 (2)	0.497 (2)	**23.205 (1)**
	DecisionTree	0.349 (2)	0.503 (2)	**23.33 (1)**
	Mean	0.269 (3)	0.602 (3)	26.36 (2)
	Median	0.269 (3)	0.604 (3)	26.366 (2)
	XGBOOST	**0.36 (1)**	**0.489 (1)**	**22.964 (1)**
	RandomGuess	-0.001 (4)	0.983 (4)	36.092 (3)

Results are in the format of mean (SK-ESD rank), where mean is calculated as the average of the performance metrics across 100 runs. The lower is the SK-ESD rank, the better is the performance.

using only a dimension. Having a higher performance when using all features compared to using only one dimension each time indicates that our approach benefits from all features combined.

Table 8: The relative improvement of the ExtraTrees model compared to baselines.

Project	Model	SA	MRE	MAE
Android	Mean	48.949%	29.117%	17.279%
	Median	49.003%	28.88%	17.291%
	RandomGuess	> 100%	56.245%	38.924%
Qt	Mean	41.572%	24.082%	15.549%
	Median	41.742%	24.445%	15.594%
	RandomGuess	> 100%	51.691%	38.535%
Eclipse	Mean	28.426%	19.297%	10.747%
	Median	28.472%	19.577%	10.752%
	RandomGuess	> 100%	52.38%	35.392%
Openstack	Mean	34.914%	14.385%	12.548%
	Median	34.939%	14.199%	12.552%
	RandomGuess	> 100%	46.352%	35.67%
Libreoffice	Mean	20.056%	4.641%	7.546%
	Median	20.296%	6.161%	7.605%
	RandomGuess	> 100%	43.691%	32.815%
All	Mean	34.644%	19.415%	13.168%
	Median	34.755%	19.657%	13.19%
	RandomGuess	> 100%	50.644%	36.583%

6.2.2 Approach

To answer this RQ, we use the same online validation method defined in Section 5.2, and we trained the *Extra trees* regression model since it has shown the best performance in RQ1. We start first by training the model using all features. Then, we only use one dimension of the features presented in Section 4.3. Finally, we evaluate the performance of the obtained models using SA, MRE and MAE and we rank them using the SK-ESD test. Note that in this RQ, we do not perform PCA for our data (Step (4) in Table 4) and use the original features instead.

6.2.3 Results

Table 9 outlines the SA, MRE and MAE scores obtained for *Extra Trees* trained using all features compared to *Extra Trees* trained on each dimension of features for all studied dimensions with the best results highlighted in bold. Based on the obtained results, we highlight the following findings:

- The time dimension achieves the best SA, MRE and MAE scores compared to the other dimensions in Android.
- The collaboration graph dimension achieves the best SA, MRE and MAE scores compared to the other dimensions for Qt and Libreoffice.
- Reviewers dimension achieves the best SA, MRE and MAE scores compared to the other features dimensions for Openstack.
- Both the Files and Owner dimensions achieve the highest SA, MRE and MAE scores compared to the other features subsets with similar SK-ESD ranks (*i.e.,* 2) for Libreoffice and Eclipse.
- In terms of SA, all features have a RIMPR between 23% (Libreoffice) and 45% (Qt) compared to using only one dimension of features.

Table 9: The performance of our approach on all features and in each dimension across all projects.

Project	Model	SA	MRE	MAE
Android	All dimensions	**0.38 (1)**	**0.494 (1)**	**24.454 (1)**
	time	0.266 (2)	0.698 (6)	28.925 (2)
	Collaboration graph	0.248 (4)	0.632 (3)	29.631 (4)
	Change	0.229 (6)	0.658 (5)	30.383 (6)
	Text	0.208 (7)	0.706 (6)	31.202 (7)
	Files	0.239 (5)	0.647 (4)	29.999 (5)
	Owner	0.253 (3)	0.628 (3)	29.424 (3)
	History	0.232 (6)	0.695 (6)	30.287 (6)
	Reviewers	0.25 (4)	0.605 (2)	29.547 (4)
Qt	All dimensions	**0.381 (1)**	**0.507 (1)**	**24.341 (1)**
	time	0.25 (3)	0.701 (5)	29.478 (3)
	Collaboration graph	0.262 (2)	0.639 (2)	29.001 (2)
	Change	0.221 (6)	0.697 (5)	30.646 (6)
	Text	0.212 (7)	0.71 (6)	30.99 (7)
	Files	0.246 (4)	0.651 (3)	29.653 (4)
	Owner	0.254 (3)	0.643 (2)	29.331 (3)
	History	0.24 (5)	0.657 (3)	29.9 (5)
	Reviewers	0.238 (5)	0.682 (4)	29.967 (5)
Eclipse	All dimensions	**0.349 (1)**	**0.586 (1)**	**30.662 (1)**
	time	0.234 (4)	0.806 (7)	36.071 (3)
	Collaboration graph	0.252 (3)	0.712 (3)	35.228 (2)
	Change	0.237 (4)	0.738 (4)	35.916 (3)
	Text	0.213 (5)	0.781 (6)	37.037 (4)
	Files	0.253 (3)	0.713 (3)	35.166 (2)
	Owner	0.257 (2)	0.696 (2)	35.015 (2)
	History	0.251 (3)	0.706 (3)	35.288 (2)
	Reviewers	0.237 (4)	0.747 (5)	35.938 (3)
Openstack	All dimensions	**0.345 (1)**	**0.511 (1)**	**21.199 (1)**
	time	0.221 (6)	0.688 (8)	25.219 (5)
	Collaboration graph	0.254 (3)	0.594 (2)	24.14 (2)
	Change	0.232 (5)	0.62 (6)	24.864 (4)
	Text	0.219 (6)	0.645 (7)	25.282 (5)
	Files	0.246 (4)	0.608 (5)	24.406 (3)
	Owner	0.246 (4)	0.602 (4)	24.413 (3)
	History	0.246 (4)	0.602 (4)	24.404 (3)
	Reviewers	0.258 (2)	0.599 (3)	24.018 (2)
Libreoffice	All dimensions	**0.318 (1)**	**0.412 (1)**	**15.075 (1)**
	time	0.226 (4)	0.521 (6)	17.129 (4)
	Collaboration graph	0.256 (2)	0.454 (2)	16.462 (2)
	Change	0.234 (3)	0.489 (4)	16.931 (3)
	Text	0.213 (5)	0.51 (5)	17.414 (5)
	Files	0.257 (2)	0.451 (2)	16.423 (2)
	Owner	0.253 (2)	0.468 (3)	16.506 (2)
	History	0.211 (5)	0.505 (5)	17.442 (5)
	Reviewers	0.237 (3)	0.47 (3)	16.87 (3)

Results are in the format of mean (SK-ESD rank), where mean is calculated as average of the performance metrics across 100 runs. The lower is the SK-ESD rank the better is the performance.

- In terms of MRE, all features have a RIMPR between 7% (Openstack) and 21% (Qt) compared to using only one dimension of features.
- In terms of MAE, all features have a RIMPR between 8% (Libreoffice) and 16% (Qt) compared to using only one dimension of features.

Overall, we observe that the importance of the features dimension may vary from one studied to another. For instance, the time dimension may be in-

fluential for some projects (*e.g.,* Android) since these projects may involve developers from different time zones. Other dimensions tend to be important for other projects. For example, the reviewer dimension quantifies the activities of reviewers and their interaction with the owner has an SK-ESD rank of 2 for Openstack. This result is expected since previous studies have already shown that social aspects may impact the outcome of a code review [23,33,34]. Furthermore, the File and the Owner dimensions can also play an essential role in the completion time for some projects (*e.g.,* LibreOffice). Finally, we observe that our approach substantially improves the SA, MRE and MAE scores when using all features compared to using only one dimension of features. This demonstrates the effectiveness of our features when they are used together. The obtained results are motivated by the fact that each dimension can capture each unique piece of characteristics to the code review process.

RQ2 summary: The best dimension can vary from one project to another depending on their characteristics. Time, collaboration graph, and reviewers dimensions tend to be the most important dimensions for code review completion time. Our approach benefits from using all the features combined and can lead to an improvement between 8% and 45% in terms of SA, MRE and MAE score.

6.3 What are the most important features influencing code review completion time (RQ3)

6.3.1 Motivation

In this research question, we would like to identify the most important features that influence the completion time of the code reviews in the studied projects: Android, Qt, Eclipse, LibreOffice and Openstack. Identifying these features might help developers improve their code review process. Since every project has its unique characteristics as shown in RQ2, we study each project separately.

6.3.2 Approach

We investigate in this RQ identifying the most influential features by leveraging the *Extra Trees* model since it was identified as the best performing model in RQ1. We first perform all the pre-processing steps defined in Section 4.4 except the PCA (Step4). Then, following previous studies [18,87,100], we reduce our feature space by performing the following steps:

Step 1: ***Correlation analysis:*** This step is performed to reduce the collinearity in our datasets. For each dataset of a given project, we first use variable clustering analysis based on the `varclus` function implemented in the Hmisc R package[18] to find the correlations among the studied features.

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

Through this analysis, we construct a hierarchical clustering of the features among all the features and the correlated features are grouped into sub-hierarchies. Similar to prior settings used in the previous studies [18, 87, 100], we randomly select one feature from a given sub-hierarchy if the correlations of features in the sub-hierarchy are larger than 0.7.

Step 2: ***Redundancy Analysis:*** After reducing collinearity among the features using correlation analysis (Step 1), we remove redundant features that do not have a unique signal as compared to the other features. We use the `redun` function implemented in the rms R package[19] to perform the redundancy analysis.

Step 3: ***Important features identification:*** In this step, we train an *Extra Trees* model using Scikit-learn similar to RQ1. To identify the importance of features, we use the `feature_importance` field provided by the *ExtraTreeRegressor* object from Scikit-learn containing the value of the importance of a given feature based on the normalized total reduction of the criterion brought by that feature also known as the Gini importance. We use the online validation described in Section 5.2 and for each iteration, we get an importance value for each feature. To determine which of the features are most important for the datasets, we input the importance values taken from all 10 iterations to the SK-ESD test.

6.3.3 Results

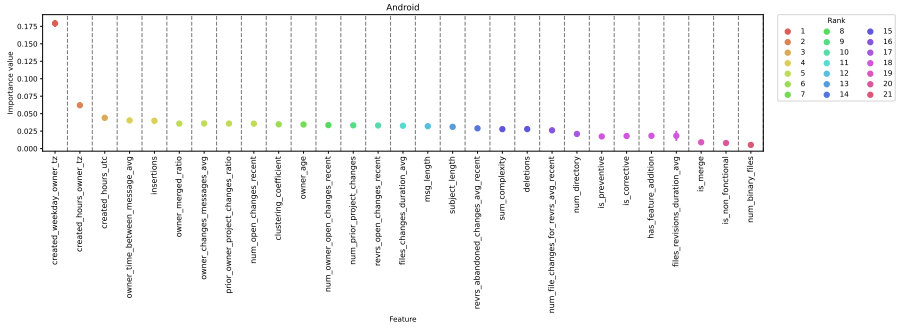
For the sake of replicability, we list the features that are selected after performing **Step 1** and **Step 2** defined in Section 6.3.2 in our replication package[20].

Figure 5 shows the results of features ranking obtained from **Step 3** (cf. Section 6.3.2) for each studied project. For all the studied projects, we notice that date-related features namely, *created_hours_utc*, *created_hours_owner_tz*, *created_weekday_utc*, *created_weekday_owner_tz* are consistently among the most important features. For example, for the Android project, we can observe from Figure 5a that *created_weekday_owner_tz*, *created_hours_owner_tz* and *created_weekday_hours_utc* are the top-3 important features. Similarly, *created_hours_utc*, *created_weekday_utc*, and *created_hours_owner_tz* are the top-3 important features for Libreoffice.

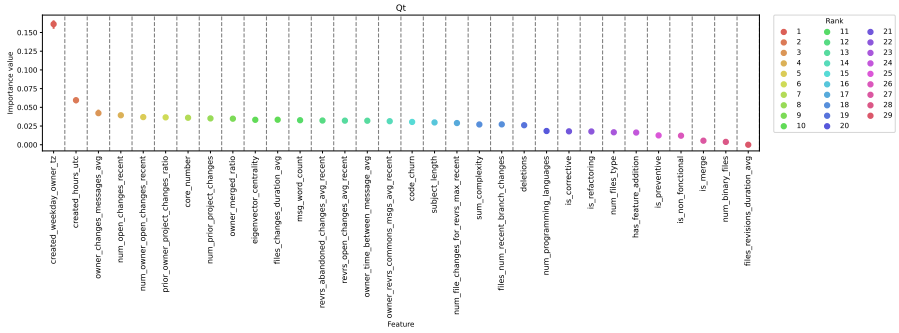
Additionally, other features are important for the code review completion time and depend on the projects. For example, history-related features, *i.e.,* features that describe the code review process can help to predict code review completion time. For example, *num_open_changes_recent* which counts the total number of the submitted changes of the projects in the last 30 days appears among the top-10 features for Qt. Similarly, the same feature appears as the 9th, 4th and 5th for Qt, Libreoffice and Eclipse respectively. Similarly, the feature *num_project_open_changes_recent* which indicates the number of

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

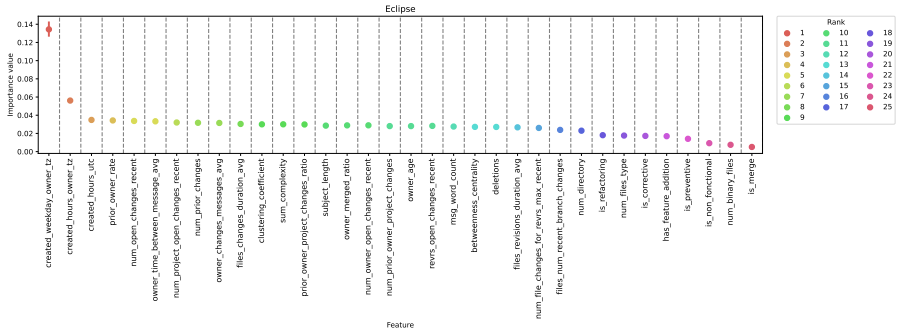
[20] `https://github.com/stilab-ets/MCRDuration/blob/main/RQ3_selected_features.md`



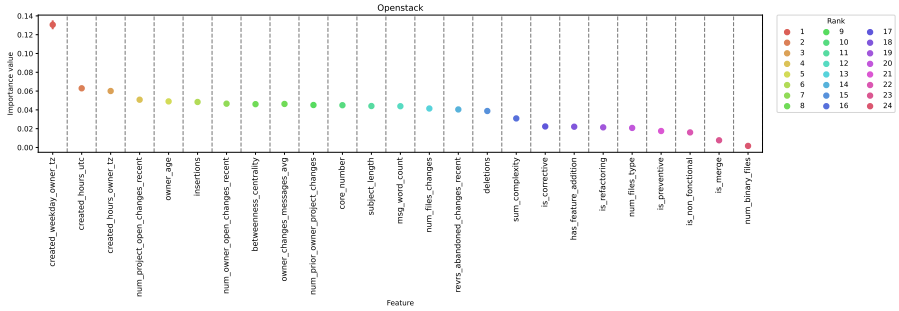
(a) SK-ESD feature ranking for Android.



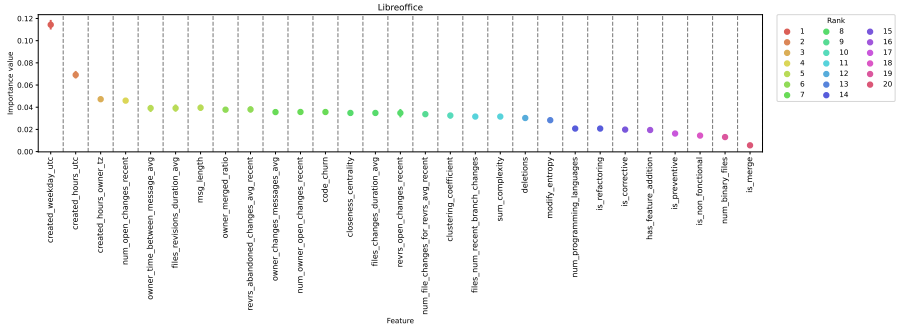
(b) SK-ESD feature ranking for Qt.



(c) SK-ESD feature ranking for Eclipse.



(d) SK-ESD feature ranking for Openstack.



(e) SK-ESD feature ranking for Libreoffice.

Fig. 5: SK-ESD feature ranking for the studied projects.

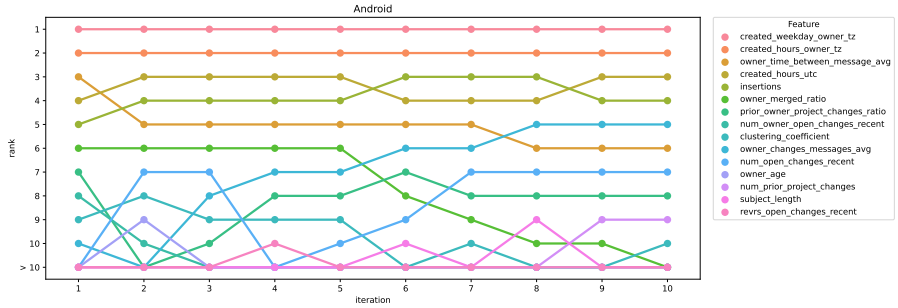
submitted changes in the last 30 days having the same repository with the studied change appears as the 4th most important feature for Openstack. Furthermore, owner experience and activity can play an important role in the code review completion time. For instance, *owner_time_between_messages_avg* and *owner_merged_ratio* are ranked as the 6th and 7th features respectively, in Android. The *owner_time_between_messages_avg* feature is ranked as the 3rd important feature for Qt. The *prior_owner_rate* is ranked as the 4th most important feature for Eclipse.

For deeper insights, we study the evolution of the most important features ranking for each studied project since code review practices may over with time. To achieve this, we report the top-10 most important features for each validation iteration for each project. Figure 6 shows the evolution ranking of the features rankings for each iteration. We can observe that the first and second ranked features that are related to time do not change for all the studied projects. Additionally, we observe that features related to owner and reviewers' activity and the interaction between them are emerging to be ranked among the top-10 features as we progress in time. For example, for Libreoffice and Android, we observe that *created_weekday_owner_tz* and *created_hours_owner_tz* keep the first and the second rank as the most important features for all the iterations. Additionally, we observe that feature ranking may change over time since code review practices can change as the project and development community evolve. For example, we observe from Figure 6c that the feature *prior_owner_rate* was not among the top-10 most important features in the first iteration then it reached the 3rd and the 4th ranking in iterations 7 and 10. Similarly, we observe from Figure 6e that the feature *revrs_abandoned_changes_avg_recent* move from not being ranked in the top-10 in iteration 3 to the 3rd most important feature for iterations 8, 9 and 10. From Figure 6d, we can also observe that the features related to previous interaction between developers (*i.e., betweeness_centrality*) and owner activities and workload (*i.e., num_owner_open_changes_recent*) are becoming among the top-5 most important features in the last iteration. Overall, the obtained results indicate the importance of the social aspect in code review which was advocated by prior studies [36, 37, 71, 90]. Additionally, having feature ranking that changes overtime indicate that deployed models should be updated regularly as new recent data is emerging.

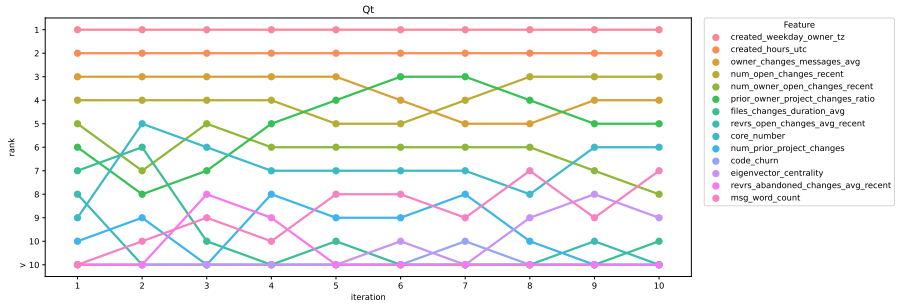
RQ3 summary: Date features are ranked as the best important features for code review completion time prediction. Feature importance ranking may vary through time and features representing owner activities and owner-reviewers interactions are emerging among the top-10 important features for recent data.

7 Discussions

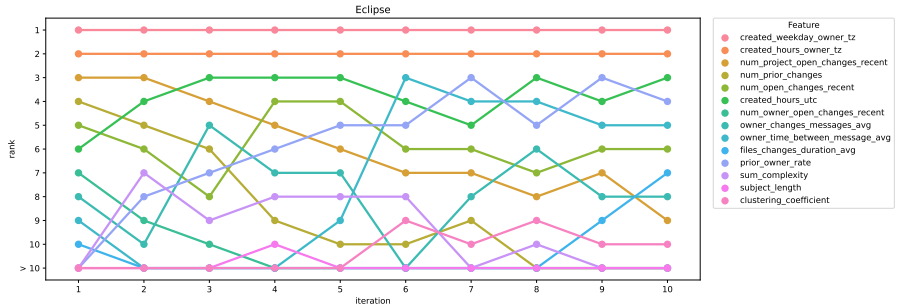
In this section, we provide deeper analysis of the performance of our approach. First, we provide a deeper analysis of our approach by analysing the mean



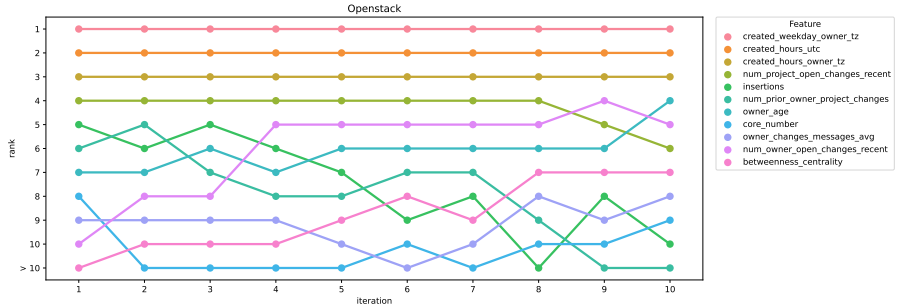
(a) Top-10 features for each validation iteration for the Android.



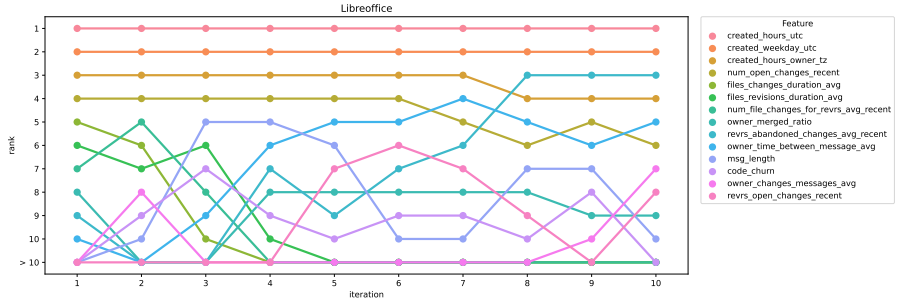
(b) Top-10 features for each validation iteration for the Qt.



(c) Top-10 features for each validation iteration for the Eclipse.



(d) Top-10 features for each validation iteration for the Openstack.



(e) Top-10 features for each validation iteration for the Libreoffice.

Fig. 6: Top-10 features for each validation iteration for the studied projects.

absolute and relative errors of our approach. Later, we study the time evolution of the performance of our approach. Moreover, we study the performance of our approach in cross-project scenario. Furthermore, we report the time efficiency of our approach to motivate the our choice of *ET*. Finally, we discuss the implications of our findings.

7.1 MAE and MRE values analysis

The obtained results suggest that our approach achieves MRE scores between 0.4 and 0.57 and MAE scores between 14.9 and 31. To get a deeper understanding of the performance, we consider the Qt project as an example and study its performance in more detail. In RQ1, we observe that our approach achieves 0.39, 0.49 and 24.15 scores for SA, MRE and MAE respectively for Qt with an improvement of 24% and 15% in terms of MRE and MAE compared to the *Mean* and *Median* baselines. To understand whether 24.15 of MAE is reasonable, we first need to understand a few characteristics related to the data that we use for inference. From Table 6, we observe that the outcome for Qt has a mean value of 53.27, an Std value of 36.37, a minimum value of 12, (cf. Section 4.2) and a max value of 146.94. Basically, we have a prediction MAE of 24.15 for data points (code reviews) ranging from 12 hours to 149.94 hours and an Std of 36.37. Figure 7 shows the distribution of the MAE values for all the studied projects. One important aspect to notice is that a significant portion of the values are lesser than approximately the half of the mean of the outcome of Qt which only a small portion of the data exceeds the mean outcome value for each project. For example, we observe for the Qt project that most of the MAE values fall behind 25 (*i.e.,* approximately the half of the mean of the outcome for the Qt project). Additionally, figure 8 shows the distribution of the MRE values, from which we can also observe that the most of the values fall behind 1 for all the studied projects. As with any regression problem, one of the other important metrics to understand is tolerance. Tolerance levels may vary based on the domain we are operating in and the users who are consuming our prediction results. For instance, a tolerance of 50% in real estate or finance is not acceptable. However, in effort estimation for code review, going off by +/- 50% hours is reasonable and can be used to alert developers regarding the code reviews taking an unexpected long completion time similar to previous studies [22]. We list the distribution of the predictions based on the tolerance level in Table 10.

7.2 The evolution of the performance of our approach over time

Figure 9 shows the prediction performance of the model during each iteration based on the SA, MRE and MAE scores. The results show that the performance does not monotonically increase over time. However, In two of the studied projects *i.e.,* Eclipse and Libreoffice, the performance in the last half

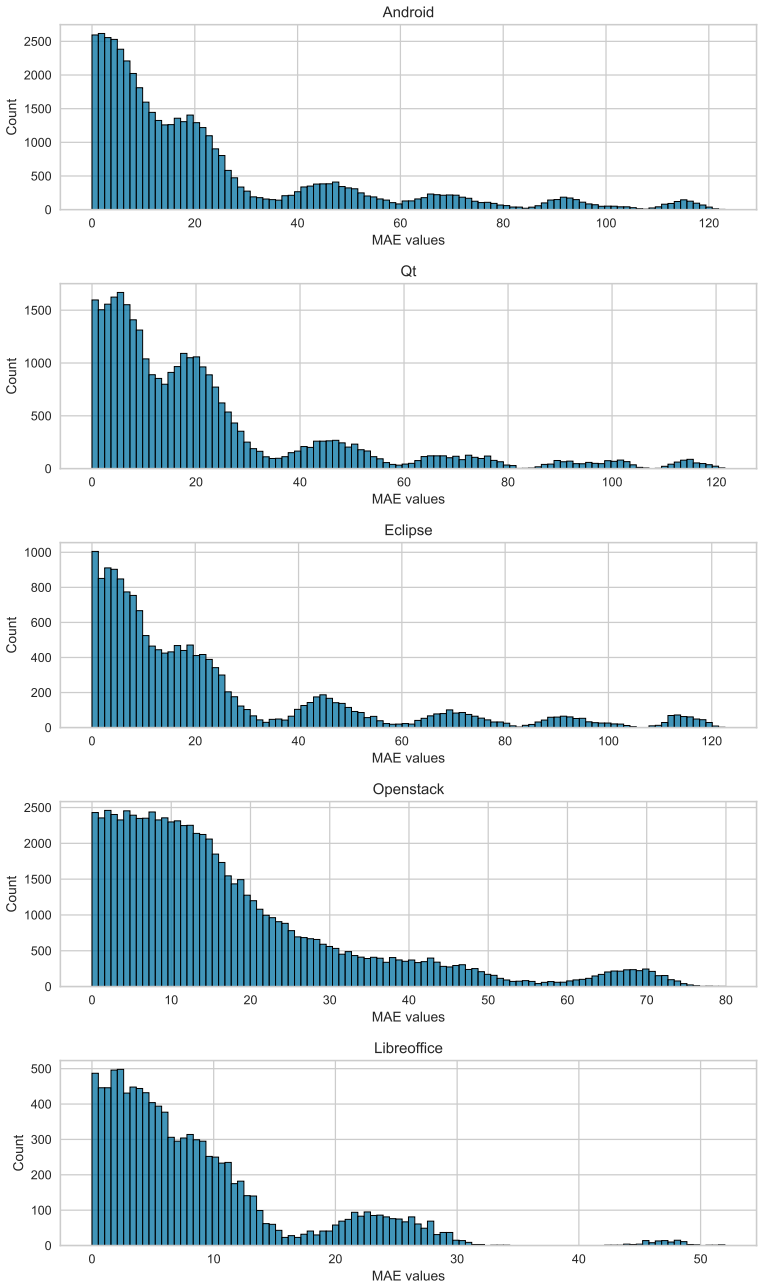


Fig. 7: MAE values distributions for the all the studied projects.

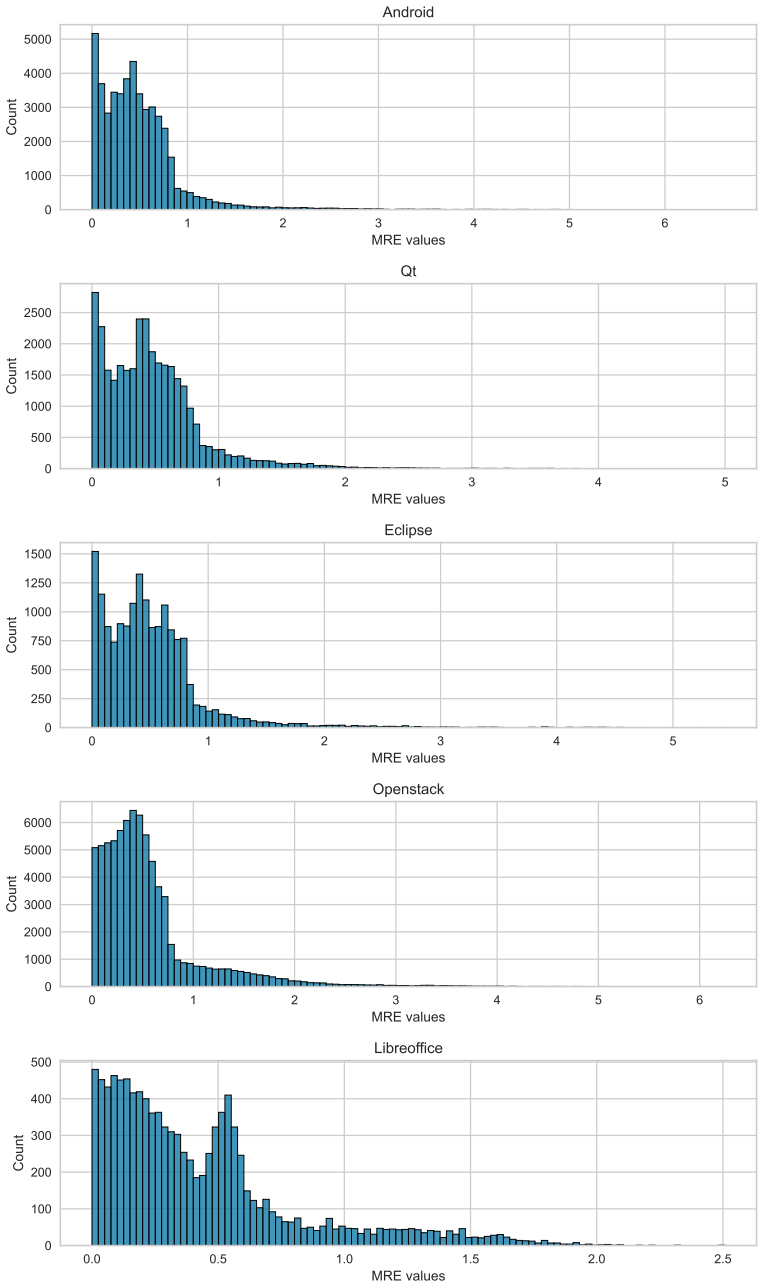


Fig. 8: MRE values distributions for each studied project.

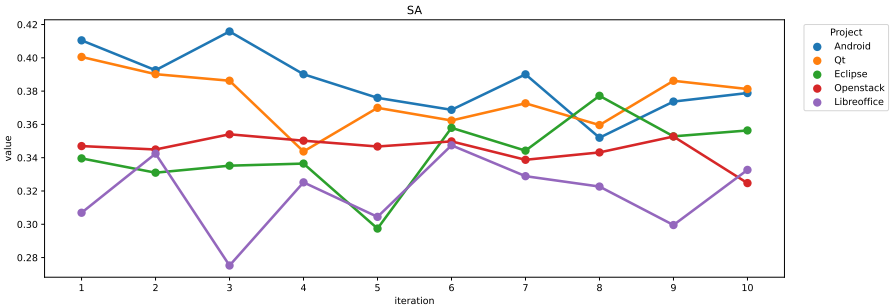
	% of the predictions within the tolerance level				
Tolerence level	**Android**	**Qt**	**Eclipse**	**Openstack**	**Libreoffice**
5%	8.41	8.66	8.3	5.32	8.61
10%	15.18	15.64	14.76	10.64	16.97
15%	20.21	20.47	19.85	15.98	25.46
20%	24.67	24.82	23.7	21.52	33.18
30%	35.67	34.71	33.21	33.02	46.65
40%	47.33	46.98	44.76	45.84	56.97
50%	60.3	60.08	57.48	58.96	65.71
70%	79.06	79.79	77.15	77.67	83.02
90%	90.54	90.12	89.75	84.91	87.91
100%	92.27	92.13	91.68	86.66	89.95

Table 10: Tolerance level distribution among each project.

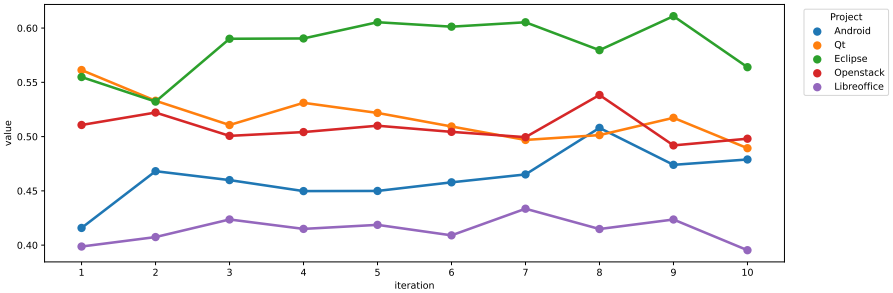
is better on average than in the first half. For the other projects (*i.e.,* Android, Openstack and Qt), the performance of our approach tend to degrade but still keep an acceptable performance over time. This suggests that the models need to be maintained and updated regularly over time since code review practices may evolve.

7.3 Cross-project performance of our approach

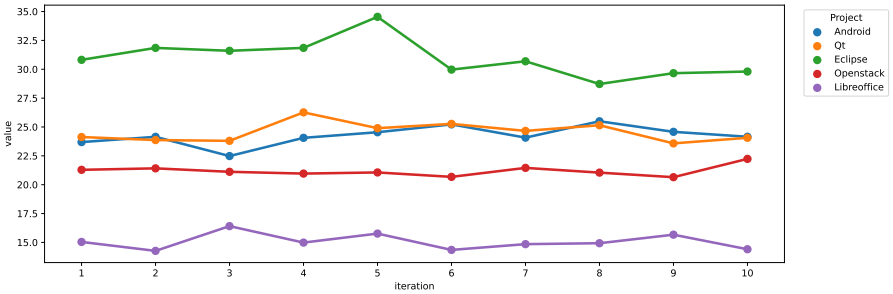
While our approach has shown acceptable performance for within-project prediction, it is important to investigate the cross-project performance of our approach since there might be insufficient data for new projects to train within-project models and generate accurate predictions. In those cases, models pre-trained on other projects might be useful during the initial stage of the project. Hence, we evaluated our approach's performance in cross-project settings. We trained an *ET* model on a complete dataset of one project and tested it on the datasets for the other projects. Figure 10 shows the results of *ExtraTrees* in a cross-project setting. We observe from figure 10 that the performance metrics depend slightly on the source data. For example, when using Android as training data, we observe that cross-project settings can achieve reasonable results in terms of SA, MRE and MAE since code review practices in these projects can be similar. However, when using Libreoffice as a training data to predict completion time for the other projects, our approach can have low performance. These results might be explained by the fact that Libreoffice have the highest skeweness and kurtosis values (cf. Table 6) which indicate that its outcome variable may contain more outliers compared to the other projects. Thus, it is hard for the ML model to learn to generalize for the other projects when using Libroffice data. Therefore, we encourage researchers to study the problem of selecting source data for a given target project with small or no available data to maximize the prediction performance of the cross-project scenario in more depth. Note that these results are not comparable to our previous results since the online validation differs from the cross-project validation.



(a) SA values evolution for each iteration for each studied project.



(b) MRE values evolution for each iteration for each studied project.

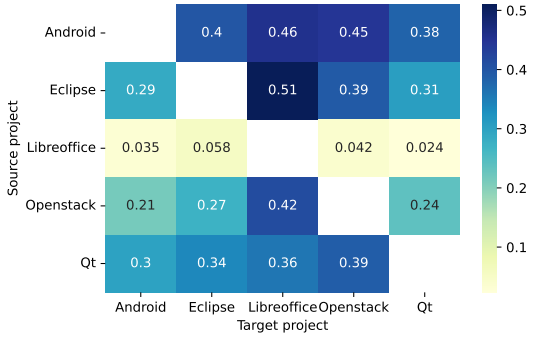


(c) MAE values evolution for each iteration for each studied project.

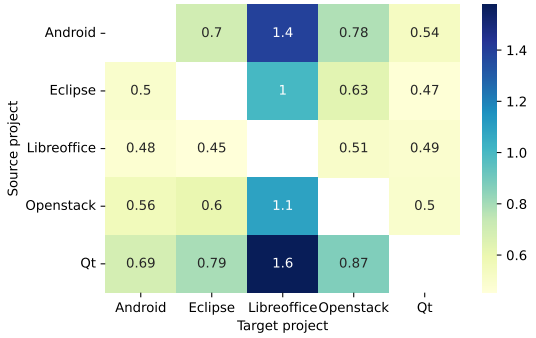
Fig. 9: Performance metrics values evolution for each iteration for each studied project.

7.4 Time efficiency of our approach

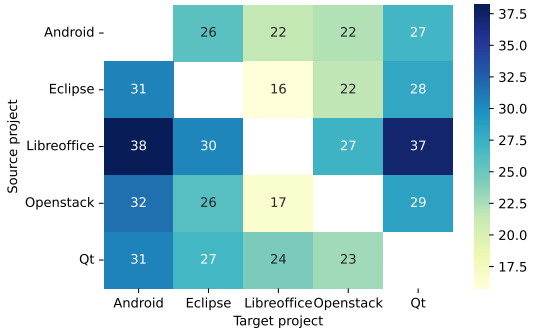
While performance is important, we also investigate the time needed to fit the model of our model. New changes keep coming, and it would be challenging to update the model if it takes too long. Recall that we mentioned in Section 6.1.3 that although *ET* and *RF* achieve similar results, we choose *ET* as ML regressor for our approach for its time efficiency [99]. Table 11 outlines the fit time for *ET* and *RF*. The results were obtained using a Desktop with 32GB of RAM and a 12 core and a 3.20GHz Intel i7-8700 CPU. The results show



(a) SA values for cross-project validation.



(b) MRE values for cross-project validation.



(c) MAE values for cross-project validation.

Fig. 10: SA, MRE and MAE values for cross-project validation.

that using *ET* is more time-efficient than *RF* which motivates its adoption as the default ML regressor for our approach.

Table 11: *ET* and *RF* training time (seconds) for each fold.

Iteration	**Android**		**Qt**		**Eclipse**		**Openstack**		**Libreoffice**	
	ET	RF	ET	RF	ET	RF	ET	RF	ET	RF
1	**2.74**	6.52	**2.07**	3.51	**1.02**	1.94	**5.03**	12.26	**0.76**	1.09
2	**5.18**	15.01	**4.04**	8.25	**2.18**	4.47	**9.48**	25.47	**1.58**	2.45
3	**7.21**	22.8	**5.97**	13.41	**3.28**	6.98	**13.78**	38.44	**2.04**	3.85
4	**9.72**	30.85	**7.55**	18.19	**4.04**	9.88	**17.53**	54.57	**2.82**	5.48
5	**11.88**	39.69	**8.98**	23.61	**5.24**	12.66	**21.9**	70.15	**3.3**	6.84
6	**14.06**	49.26	**11.22**	28.43	**6.19**	16.06	**26.0**	84.58	**3.77**	8.24
7	**16.21**	58.46	**12.86**	33.63	**7.07**	19.03	**30.49**	99.93	**4.23**	10.09
8	**18.83**	68.44	**14.47**	38.48	**7.85**	22.18	**34.24**	114.02	**4.96**	11.8
9	**21.07**	77.93	**16.01**	44.62	**8.81**	25.15	**39.19**	130.14	**5.44**	13.51
10	**23.4**	88.11	**17.63**	50.12	**10.02**	28.27	**42.41**	144.34	**5.89**	15.07
Average	**13.03**	45.71	**10.08**	26.22	**5.57**	14.66	**24.01**	77.39	**3.48**	7.84

7.5 Implications of our findings

Our study's aim is to show that we can use ML to accurately predict the code review completion time. Thus, our approach can be used to provide early warnings to authors, reviewers, and the management about review iterations that will eventually take an unexpectedly long time. As such, our approach can be effective for the diverse major stakeholders in software engineering:

Project Manager and leads. Our approach helps prioritize code review and code change efforts based on the predictions thus bringing benefits to the software project management. If project managers can predict the completion time of a given code review early, they may analyze the cause of why such a code review may take an expected time and take necessary steps if required. Multiple reasons may act behind the unexpected delay of code reviews such as differences in geographical locations and time-zones, resource allocation, job environment, efficiency mismatch between the author and the reviewer, and even their relationships. Some of these aspects may be addressed by the early intervention of the project managers and thus revert the result of a particular review iteration. Therefore, the company may save lots of time and resources. Indeed, code review is a very important aspect of modern software engineering. Large software companies, such as Open Source projects, are practicing it with care. Researchers are trying to generate insight from large repositories of code review and trying to bring efficiency to the process to save the cost of production. In this work, we study a relatively under-studied problem: predicting the completion time of a given code change at an early stage of the code review process. We design a machine learning model to apply carefully selected features generated from the communication between the developers and the reviewers. Our approach is expected to save the waste of effort by alerting managers about code reviews that may take a long time.

Software engineers. Our tool can be used to alert reviewers about code reviews that are taking unexpected delays. The code review process requires significant effort and time. The code authors and reviewers involved in a review iteration are likely to get frustrated if they see that their effort

goes in vain, *i.e.,* their code reviews take an unexpectedly long time to get reviewed. Suppose they get an early indication from our approach about the expected completion time of their submitted code review. In this case, 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. Additionally, practitioners can benefit from our approach by understanding the root causes of code review delays by investigating the features importance thus answering the question: what are the most important factors that influence the completion time of given code review. Thus it would benefit the practitioners to exploit such valuable insights to design strategies to improve their code review strategies.

Tool builders. We showed in Section 2.2 that using dummy review bot that sends reminders after 1 month of inactivity is not able to engage developers to reduce the completion time of forgotten code reviews. For this reason, our approach can help tool builders to build more intelligent mechanisms that use our reliable prediction for the completion time of a given code review request. These mechanisms can be integrated in the code review tool and can be used to alert developers about forgotten code reviews based on more intelligent predictions. Thus, leads to a significant gain in terms effort and delays. Additionally, tool builders can extend our approach to provide more insights for the developers by explaining the predictions of the approach using local explanation approaches investigated in other software engineering studies [101, 102]. We are planning to support our approach with local explanation approaches to provide meaningful insights regarding the generated prediction of our approach.

Software Engineering Researchers. We encourage researchers to further investigate multiple aspects related early prediction MCR tools. Prior software engineering researchers investigated different approaches including statistical methods, parametric models, and ML methods for software effort and duration prediction [44]. In this study, we investigated whether it is possible to predict the duration of a code review based on multiple technical and non-technical aspects. We believe that researchers can build upon our work and investigate more features/dimensions related to code review that may enhance our model. Additionally, researchers may investigate the performance of models in more detail. For instance, researchers may investigate the fairness of our models and see if there is some bias against new developers joining the development team [69]. Additionally, we showed in Section 7.3 that our approach can have acceptable performance in the cross-project scenario. The cross-project scenario can be helpful when we have a new project with a small amount of data. Therefore, cross-project prediction scenarios can be also investigated in more detail by researchers. Finally, our approach uses using black-box machine learning techniques (*i.e., ET* and *RF*). Despite their acceptable performance, the interpretability of these models is limited. In practice it is challenging for practitioners to understand why such models predicted a given

outcome since these models are usually a complex ensemble of decision trees. Recently, multiple techniques have been proposed to help practitioners understand the outcome of a black-box model [103–106]. Therefore, we encourage researchers to investigate the interpretability of our models in more detail. Finally, we made our data publically available for researchers who can use it to build upon our work [28].

8 Threats to validity

Threats to internal validity can be related to potential errors in the implementation of our approach. To mitigate this threat, we based our implementation on different Python packages supported by a large community of contributors namely, Scikit-learn for ML data pre-processing and ML algorithms training and evaluation, NetworkX to build social graphs and compute collaboration graph metrics and Pydriller to process commits data and compute metrics related to the code change. Additionally, we tested our approach with different runs (*i.e.,* 100) and we used the average of metrics as a performance indicator to remove to address the stochastic nature of the ML algorithms. However, there could still be some errors that we might not notice.
Threats to external validity are related to the generalizability of our results. To mitigate this threat, we studied the performance of our approach using five OSS projects ranging from different domains. Thus, our findings are based on a total of five projects and more than 280K code review instances. However, we do not claim the generalizability of our results for other software projects. Instead, our key message is that our approach can have acceptable results if applied to some projects. We encourage researchers to experiment with our approach under different settings and datasets. Finally, we plan to validate the performance of our solution in an industrial case study.
Threats to construct validity are related to the use of a limited set of metrics to compare the performances of our approach. To mitigate this threat and ensure the reliability of our findings from a relatively small set of metrics, we used a robust experimental setup. We used online validation repeated 100 times as a validation process with different performance metrics to quantify different aspects. Additionally, we considered grid search to fine-tune ML classifiers hyper-parameters. Using other hyper-parameters techniques may lead to different observations. In this study, we focused on commonly-used metrics in regression including SA, MRE and MAE to compare ML regression models. We also use the SK-ESD test to rank different ML regression approaches. We encourage researchers to study the performance of various regressors under a different set of metrics in the future.

9 Conclusion

Modern code review is an integral part of the software development process to ensure the software systems' quality and timely delivery. However, some code

reviews can take an unexpected long completion time leading to the degradation of the whole process. In this study, we build an approach to predict the code review completion time using *Extra trees* regressor. Our approach is based on 69 features spanning over eight dimensions related to the time, change, collaboration graph, owner, files, history, text and reviewers. To validate the performance of our approach, we conducted an empirical study on a dataset containing more than 280K code changes extracted from five open source projects hosted in Gerrit. The empirical study results indicate that our approach can be used effectively to predict code review time. In addition, we show that our approach's performance improves when using all features are used together. Further, we show that time features (*i.e.,* the submission time and day of the code review request) can play an important role in the code review completion time prediction and features related to the owner activity and the owner-reviewers' previous interactions emerge as important features when more recent data is added. Furthermore, our approach uses historical data in the code review repository and as such the performance of our approach improves as a software system evolves with new and more data. Finally, we showed that our approach has reasonable fit time and acceptable cross-project performance. As such, our approach can help 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 handling code review requests.

The obtained results encourage us to investigate and study the usability of our approach in more depth by deploying it in real projects. Additionally, we plan to support our approach with explainable AI techniques to leverage local prediction explanations for the provided predictions.

Acknowledgements This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2018-05960.

References

1. M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.
2. A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13–40.
3. M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.
4. L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.
5. A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
6. C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

7. M. Chouchen, A. Ouni, R. G. Kula, D. Wang, P. Thongtanunam, M. W. Mkaouer, and K. Matsumoto, "Anti-patterns in modern code review: Symptoms and prevalence," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 531–535.
8. E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, p. 106737, 2022.
9. P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.
10. T. Winters, T. Manshreck, and H. Wright, *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020.
11. T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 21–30.
12. T. Patanamon, T. Chakkrit, G. K. Raula, Y. Norihiro, I. Hajimu, and M. Ken-ichi, "Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.
13. A. Ouni, R. G. Kula, and K. Inoue, "Search-based peer reviewers recommendation in modern code review," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 367–377.
14. M. Chouchen, A. Ouni, M. W. Mkaouer, R. G. Kula, and K. Inoue, "Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review," *Applied Soft Computing*, vol. 100, p. 106908, 2021.
15. V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?" *IEEE Transactions on Software Engineering*, 2018.
16. C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn, "Automatically recommending code reviewers based on their expertise: An empirical comparison," in *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 99–110.
17. V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 931–940.
18. Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3346–3393, 2018.
19. N. Saini and R. Britto, "Using machine intelligence to prioritise code review requests," in *IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 11–20.
20. K. Islam, T. Ahmed, R. Shahriyar, A. Iqbal, and G. Uddin, "Early prediction for merged vs abandoned code changes in modern code reviews," *Information and Software Technology*, vol. 142, p. 106756, 2022.
21. G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 358–368.
22. C. Maddila, C. Bansal, and N. Nagappan, "Predicting pull request completion time: a case study on large scale cloud services," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 874–882.
23. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "Investigating technical and non-technical factors influencing modern code review," *Empirical Software Engineering*, vol. 21, no. 3, pp. 932–959, 2016.
24. E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 348–357.
25. T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "Code reviews with divergent review scores: An empirical study of the openstack and qt communities," *IEEE Transactions on Software Engineering*, 2020.

26. S. Ruangwan, P. Thongtanunam, A. Ihara, and K. Matsumoto, "The impact of human factors on the participation decision of reviewers in modern code review," *Empirical Software Engineering*, vol. 24, no. 2, pp. 973–1016, 2019.
27. G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 345–355.
28. ___, MCRDuration Replication Package: https://github.com/stilab-ets/MCRDuration, 2022.
29. O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 111–120.
30. A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 511–522.
31. F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion in code reviews: Reasons, impacts, and coping strategies," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 49–60.
32. ——, "An exploratory study on confusion in code reviews," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–48, 2021.
33. Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? case study on the linux kernel," in *Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 101–110.
34. A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *Int. Symp. on Empirical Software Eng. and Measurement*, 2014, pp. 1–10.
35. D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "Acceptance factors of pull requests in open-source projects," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1541–1546.
36. J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th international conference on Software engineering*, 2014, pp. 356–366.
37. P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," *Empirical Software Engineering*, vol. 22, no. 2, pp. 768–817, 2017.
38. J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings, "Gender differences and bias in open source: Pull request acceptance of women versus men," *PeerJ Computer Science*, vol. 3, p. e111, 2017.
39. A. Rastogi, N. Nagappan, G. Gousios, and A. van der Hoek, "Relationship between geographical location and evaluation of developer contributions in github," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–8.
40. C. D. Egelman, E. Murphy-Hill, E. Kammer, M. M. Hodges, C. Green, C. Jaspan, and J. Lin, "Predicting developers' negative feelings about code review," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 174–185.
41. B. Barry *et al.*, "Software engineering economics," *New York*, vol. 197, 1981.
42. S. Chulani, B. Boehm, and B. Steece, "Bayesian analysis of empirical software engineering cost models," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 573–583, 1999.
43. L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of systems and software*, vol. 51, no. 3, pp. 245–273, 2000.
44. A. Trendowicz, J. Münch, and R. Jeffery, "State of the practice in software effort estimation: a survey and literature review," in *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, 2008, pp. 232–245.

45. R. Britto, V. Freitas, E. Mendes, and M. Usman, "Effort estimation in global software development: A systematic literature review," in *2014 IEEE 9th International Conference on Global Software Engineering*. IEEE, 2014, pp. 135–144.
46. N. Bettenburg, M. Nagappan, and A. E. Hassan, "Towards improving statistical modeling of software engineering data: think locally, act globally!" *Empirical Software Engineering*, vol. 20, no. 2, pp. 294–335, 2015.
47. P. Sharma and J. Singh, "Systematic literature review on software effort estimation using machine learning approaches," in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*. IEEE, 2017, pp. 43–47.
48. B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," *Annals of software engineering*, vol. 1, no. 1, pp. 57–94, 1995.
49. E. Kocaguneli, T. Menzies, J. Keung, D. Cok, and R. Madachy, "Active learning and effort estimation: Finding the essential content of software effort estimation data," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1040–1053, 2012.
50. A. L. Oliveira, "Estimation of software project effort with support vector regression," *Neurocomputing*, vol. 69, no. 13-15, pp. 1749–1753, 2006.
51. F. Ferrucci, C. Gravino, R. Oliveto, and F. Sarro, "Genetic programming for effort estimation: an analysis of the impact of different fitness functions," in *2nd International Symposium on Search Based Software Engineering*. IEEE, 2010, pp. 89–98.
52. F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 619–630.
53. V. Tawosi, F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation: A replication study," *IEEE Transactions on Software Engineering*, 2021.
54. Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *2015 IEEE/ACM 12th working conference on mining software repositories*. IEEE, 2015, pp. 367–371.
55. C. Maddila, S. S. Upadrasta, C. Bansal, N. Nagappan, G. Gousios, and A. van Deursen, "Nudge: Accelerating overdue pull requests towards completion," *arXiv preprint arXiv:2011.12468*, 2020.
56. Y. Huang, N. Jia, X. Zhou, K. Hong, and X. Chen, "Would the patch be quickly merged?" in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2019, pp. 461–475.
57. S. Wang, C. Bansal, N. Nagappan, and 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.
58. S. Wang, C. Bansal, and N. Nagappan, "Large-scale intent analysis for identifying large-review-effort code changes," *Information and Software Technology*, vol. 130, p. 106408, 2021.
59. I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Information and Software Technology*, vol. 128, p. 106392, 2020.
60. W. Zhang, Z. Pan, and Z. Wang, "Prediction method of code review time based on hidden markov model," in *International Conference on Web Information Systems and Applications*. Springer, 2020, pp. 168–175.
61. Y.-S. Seo and D.-H. Bae, "On the value of outlier elimination on software effort estimation research," *Empirical Software Engineering*, vol. 18, no. 4, pp. 659–698, 2013.
62. F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 eighth ieee international conference on data mining*. IEEE, 2008, pp. 413–422.
63. M. Greiler, C. Bird, M.-A. Storey, L. MacLeod, and J. Czerwonka, "Code reviewing in the trenches: Understanding challenges, best practices and tool needs," 2016.
64. A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.
65. 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.

66. R. Moser, W. Pedrycz, and 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.
67. N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452–461.
68. P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 168–179.
69. V. Kovalenko and A. Bacchelli, "Code review for newcomers: is it different?" in *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2018, pp. 29–32.
70. P. C. Rigby and 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.
71. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *2013 20th working conference on reverse engineering (WCRE)*. IEEE, 2013, pp. 122–131.
72. M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1032–1041.
73. O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1028–1038.
74. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.
75. A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 99–108.
76. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.
77. S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, 2010, pp. 1–9.
78. K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens, "Data mining techniques for software effort estimation: a comparative study," *IEEE transactions on software engineering*, vol. 38, no. 2, pp. 375–397, 2011.
79. P. A. Whigham, C. A. Owen, and S. G. Macdonell, "A baseline model for software effort estimation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–11, 2015.
80. D. Singh and B. Singh, "Investigating the impact of data normalization on classification performance," *Applied Soft Computing*, vol. 97, p. 105524, 2020.
81. S. Prykhodko, "Developing the software defect prediction models using regression analysis based on normalizing transformations," in *Modern problems in testing of the applied software"(PTTAS-2016), Abstracts of the Research and Practice Seminar, Poltava, Ukraine*, 2016, pp. 6–7.
82. Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, "Software defect prediction based on kernel pca and weighted extreme learning machine," *Information and Software Technology*, vol. 106, pp. 182–200, 2019.
83. M. Mustaqeem and M. Saqib, "Principal component based support vector machine (pc-svm): a hybrid technique for software defect detection," *Cluster Computing*, vol. 24, no. 3, pp. 2581–2595, 2021.
84. G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," *Advances in neural information processing systems*, vol. 26, 2013.

85. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th international conference on Software Engineering*, 2016, pp. 321–332.
86. H. Wei, C. Hu, S. Chen, Y. Xue, and Q. Zhang, "Establishing a software defect prediction model via effective dimension reduction," *Information Sciences*, vol. 477, pp. 399–409, 2019.
87. Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 301–310.
88. L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 170–181.
89. M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 99–108.
90. P. Thongtanunam and A. E. Hassan, "Review dynamics and their impact on software quality," *IEEE Transactions on Software Engineering*, 2020.
91. A. Leguina, "A primer on partial least squares structural equation modeling (pls-sem)," 2015.
92. P. H. Westfall, "Kurtosis as peakedness, 1905–2014. rip," *The American Statistician*, vol. 68, no. 3, pp. 191–195, 2014.
93. M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A deep learning model for estimating story points," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 637–656, 2018.
94. T. Xia, J. Chen, G. Mathew, X. Shen, and T. Menzies, "Why software effort estimation needs sbse," *arXiv preprint arXiv:1804.00626*, 2018.
95. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization for defect prediction models," 2018.
96. J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.
97. S. Herbold, "Comments on scottknottesd in response to" an empirical comparison of model validation techniques for defect prediction models"," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1091–1094, 2017.
98. M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.
99. P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.
100. G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 135–145.
101. C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "Jitbot: an explainable just-in-time defect prediction bot," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1336–1339.
102. D. Rajapaksha, C. Tantithamthavorn, C. Bergmeir, W. Buntine, J. Jiarpakdee, and J. Grundy, "Sqaplanner: Generating data-informed software quality improvement plans," *IEEE Transactions on Software Engineering*, 2021.
103. M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.
104. A. Messalas, Y. Kanellopoulos, and C. Makris, "Model-agnostic interpretability with shapley values," in *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*. IEEE, 2019, pp. 1–7.
105. J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Transactions on Software Engineering*, 2020.

106. X. Yang, H. Yu, G. Fan, Z. Huang, K. Yang, and Z. Zhou, "An empirical study of model-agnostic interpretation technique for just-in-time software defect prediction," in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2021, pp. 420–438.