Empirical Software Engineering (2024) 29:29
https://doi.org/10.1007/s10664-023-10431-7

A multi-objective effort-aware approach for early code review prediction and prioritization

Moataz Chouchen[1] · **Ali Ouni**[1]

Accepted: 27 November 2023 / Published online: 23 December 2023

Abstract

Modern Code Review (MCR) is an essential practice in software engineering. MCR helps with the early detection of defects and preventing poor implementation practices and other benefits such as knowledge sharing, team awareness, and collaboration. However, reviewing code changes is a hard and time-consuming task requiring developers to prioritize code review tasks to optimize their time and effort spent on code review. Previous approaches attempted to prioritize code reviews based on their likelihood to be merged by leveraging Machine learning (ML) models to maximize the prediction performance. However, these approaches did not consider the review effort dimension which results in sub-optimal solutions for code review prioritization. It is thus important to consider the code review effort in code review request prioritization to help developers optimize their code review efforts while maximizing the number of merged code changes. To address this issue, we propose $CostAwareCR$, a multi-objective optimization-based approach to predict and prioritize code review requests based on their likelihood to be merged, and their review effort measured in terms of the size of the reviewed code. $CostAwareCR$ uses the RuleFit algorithm to learn relevant features. Then, our approach learns Logistic Regression (LR) model weights using the Non-dominated Sorting Genetic Algorithm II (NSGA-II) to simultaneously maximize (1) the prediction performance and, (2) the cost-effectiveness. To evaluate the performance of $CostAwareCR$, we performed a large empirical study on 146,612 code reviews across 3 large organizations, namely LibreOffice, Eclipse and GerritHub. The obtained results indicate that $CostAwareCR$ achieves promising Area Under the Curve (AUC) scores ranging from 0.75 to 0.77. Additionally, $CostAwareCR$ outperforms various baseline approaches in terms of effort-awareness performance metrics being able to prioritize the review of 87% of code changes by using only 20% of the effort. Furthermore, our approach achieved 0.92 in terms of the normalized area under the lift chart (P_{opt}) indicating that our approach is able to provide near-optimal code review prioritization based on the review effort. Our results indicate that our multi-objective formulation is prominent for learning models that provide a trade-off between good cost-effectiveness while keeping promising prediction performance.

Communicated by: Yasutaka Kamei

✉ Moataz Chouchen
moataz.chouchen.1@ens.etsmtl.ca

Extended author information available on the last page of the article

Keywords Code review · Multi-Objective Optimization · Code review prioritization

1 Introduction

Code review is an essential practice in the software development process which allows developers to review code changes before merging them into the main code base. Code review was originally introduced by (Fagan 1999) and was known as code inspection. Code inspection involves developers meeting on a weekly/monthly basis to discuss and inspect code changes. Thus, code inspection was known to be a cumbersome and time-consuming process (Shull and Seaman 2008). Nowadays, code inspection has been migrated to a more lightweight asynchronous and tool-based process named Modern code review (MCR). MCR have been widely adopted in large open source projects (Beller et al. 2014) and several tools namely, Gerrit[1] and GitHub[2].

MCR allows developers to provide and/or receive feedback regarding their code changes in a timely manner. Prior studies showed that adopting MCR led to several benefits including fewer bugs and better code quality (Greiler et al. 2016). Additionally, MCR can help developers to better understand the code and improve code ownership and transfer learning between developers (Bacchelli and Bird 2013; Greiler et al. 2016; Chouchen et al. 2023).

However, adopting MCR comes with several challenges. Usually, developers receive many code review requests daily (Fan et al. 2018; Chouchen et al. 2023; AlOmar et al. 2022; Chouchen et al. 2021). Prioritizing these review requests represents one of the major challenges that developers often face daily (Fan et al. 2018; Gousios et al. 2015, 2016). (Gousios et al. 2016) reported that prioritizing code review is generally performed manually. That is, developers may check hundreds of code review requests prior to starting their review activities which is known to be a time-consuming task. Additionally, developers have usually tight deadlines and they need to merge the maximum number of code changes with the minimum review effort.

To address this issue, prior works attempted to learn to prioritize code review requests automatically based on the Machine learning (ML) approaches (Gousios et al. 2014; Fan et al. 2018; Islam et al. 2022). In particular, previous works attempted to prioritize code review requests based on their relevance (i.e., their likelihood to be merged). These studies adopted classification models to classify code review requests as merged or abandoned based on several features designed to capture several aspects related to the code review activities. Later, the probability of merge is used to prioritize the code review requests (Fan et al. 2018; Islam et al. 2022) However, to the best of our knowledge, code review efforts (Chen et al. 2018) have not been considered when building such models. Code review efforts are associated with the cost of reviewing and testing a given code change. Different changes may require a different amount of code review efforts. Smaller code changes may take considerably smaller code review effort thus they need to be inspected before the larger ones. Therefore, we believe that is important to incorporate the code review effort dimension when prioritizing code review requests.

Previous studies reported a conflicting relationship between the prediction performance of classifying whether a given code change is going to be merged/abandoned and the cost-effectiveness of software predictive models (Guo et al. 2018; Arisholm et al. 2010). To address this issue, we propose $CostAwareCR$, an effort-aware approach for code review requests

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

[2] https://github.com/features/code-review

early prediction and prioritization. Our approach helps developers predict whether a given code change is going to be merged/abandoned and prioritize code review requests based on their relevance and their code review efforts. Since there are multiple conflicting aspects to consider (i.e., prediction performance and cost-effectiveness), we adopt a Search-Based Software Engineering (SBSE) approach following the guidelines of Harman and Jones (2001) which suggests that prediction models for software engineering can be built by optimizing different conflicting aspects, such as performance and cost-effectiveness. In a nutshell, our approach consists of four steps. First, we extract various features related to code review activities from the code review history. Thereafter, we use the RuleFit algorithm (Friedman and Popescu 2008) to learn augmented features to improve the prediction performance of our approach. Subsequently, we use the Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al. 2002) to optimize the weights of a logistic regression (LR) model. NSGA-II learns the LR model weights by optimizing 2 conflicting objective functions: (*i*) maximizing prediction performance and, (*ii*) maximising the cost-effectiveness. Finally, the obtained solutions are used to predict and prioritise the code review requests.

To evaluate the performance of our approach, we conduct an empirical study based on 146,612 code review requests extracted from 3 large open-source software projects, Eclipse, LibreOffice and Gerrithub. We compared our approach to 3 (Machine Learning) ML baselines including the state-of-the-art models proposed by Islam et al. (2022) and Kamei et al. (2012). We evaluated the performance of different models using the Area Under the Curve (AUC) and two effort-aware performance metrics, the area under the lift chart (P_{opt}) and the recall at 20% (ACC) (Chen et al. 2018) following an online validation process (Islam et al. 2022). The obtained results indicate that our approach achieves prominent AUC scores higher than 0.75 for all the studied projects. Additionally, our approach achieves a relative improvement from 6% to 38% in terms of P_{opt} and from 6% to 132% in terms of ACC as compared to baseline approaches. The obtained results indicate that our approach is efficient in predicting code reviews as merged/abandoned with acceptable performance and can help to prioritize code review requests.

We summarize the contributions of this paper as follows:

- We propose a new formulation for the early code review prediction and prioritization problem as a multi-objective search problem. Our approach led to prominent prediction performance and the best cost-effectiveness compared to previous state-of-the-art approaches.
- A large empirical study based on 146,612 code reviews extracted from three long-lived projects, Eclipse, LibreOffice and Gerrithub. Through this empirical study, we evaluate the performance of $CostAwareCR$in predicting and prioritizing code review requests.
- A replication package that contains the implementation of $CostAwareCR$, the used data and the scripts to replicate and extend our results (CostAwareCR 2023).

The rest of the paper is organized as follows. Section 2 introduces the needed background for our approach. The related work to our study is discussed in Section 3. We present the details of our approach in Section 4. Section 5 presents the empirical study details. We present and discuss the obtained results in Section 6. We provide more additional analysis of $CostAwareCR$and the implications of the obtained results in Section 7. In Section 8, we discuss the threats to validity related to our study. Finally, we conclude this study with Section 9.

2 Background and Motivation

In this section, we present the main concepts to help understand our approach then we provide a motivating example to motivate the need for it.

2.1 Background

2.1.1 Code Review Workflow in Gerrit

Figure 1 shows an illustration of the Gerrit code review workflow. As described by Milanesio (2013), the Gerrit code review process starts with a developer submitting a code change to the Gerrit code review tool. Gerrit assigns reviewers to the change either by automatically suggesting reviewers or by allowing the developer/project maintainers to manually assign reviewers. Subsequently, Reviewers review the change and provide feedback on how to improve the change and avoid potential defects/issues. The author of the change addresses the reviewers' comments and improves the change and finally, a core developer or the project maintainer chooses whether he will merge the change to the main code base or abandon it.

2.1.2 NSGA-II Overview

The Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a widely used evolutionary algorithm designed for solving multi-objective optimisation problems. Introduced by Kalyanmoy (Deb et al. 2002), NSGA-II extends the classical genetic algorithm to efficiently handle multiple conflicting objectives simultaneously. Its effectiveness, efficiency, and ability

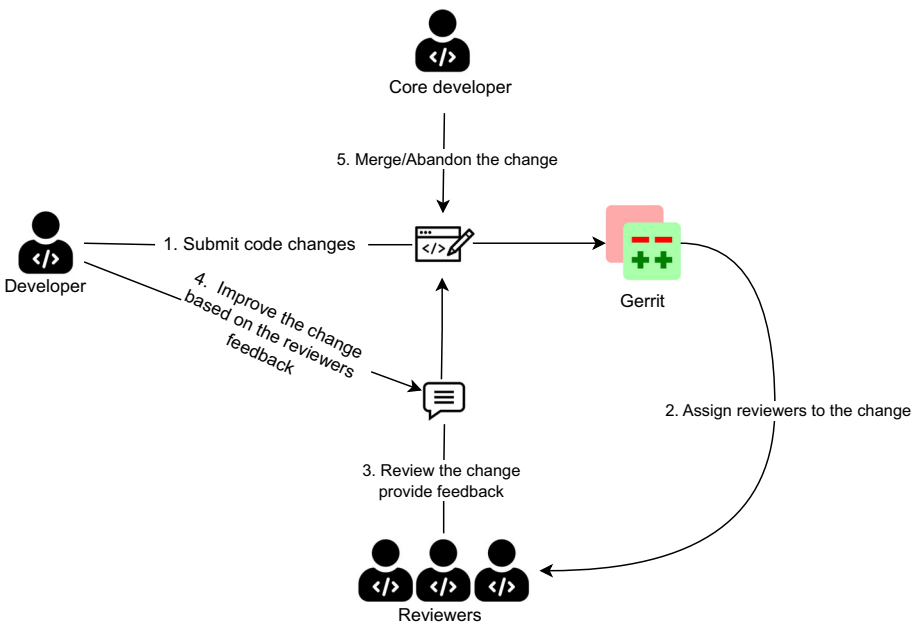


Fig. 1 An illustration of the Gerrit code review process

to find diverse Pareto-optimal solutions have made it a popular choice for various real-world optimization tasks.

Algorithm 1 NSGA-II.

Data: The size of population N, Number of generations G
Result: Set of Pareto-optimal solutions
Initialize population P of size N with random individuals;
Evaluate the fitness of each individual in P based on the objectives;
for $g \leftarrow 1$ *to* G **do**
  Create an empty list Q for offspring;
  Perform non-dominated sorting of individuals in P and put the resulting fronts in R ;
  Create an empty list P' for selected individuals;
  $i \leftarrow 0$;
  while $|P'| + |R_i| \leq N$ **do**
    Assign crowding distance to individuals in front R_i;
    $P' \leftarrow P' \cup R_i$;
    $i \leftarrow i + 1$;
  Sort individuals in front i based on crowding distance;
  $P' \leftarrow P' \cup R_i[1 : (N - |P'|)]$; `// Fill remaining slots in` P'
  Perform crossover and mutation on individuals in P' to create offspring Q;
  Evaluate the fitness of each individual in Q based on the objectives;
  Select N best individuals from $P \cup Q$ based on non-dominated sorting and crowding distance;
  $P \leftarrow$ selected individuals;

Algorithm 1 shows the pseudo-code of NSGA-II. First, the population P is initialized with random individuals, and their fitness is evaluated based on the objectives. NSGA-II then performs a series of generations, each consisting of selection, crossover, and mutation steps. The non-dominated sorting and crowding distance assignment are used for maintaining diversity in the population and identifying the Pareto-optimal front. The output of NSGA-II is a set of Pareto-optimal solutions that represent the trade-offs between the conflicting objectives. Decision-makers can use this set to make informed decisions and identify optimal solutions based on their preferences. In this study, we NSGA-II to learn models that can both predict and prioritize code review requests. In Section 4.2, we show how NSGA-II is adapted to learn early code review prediction and prioritization models.

2.1.3 The RuleFit Algorithm

The RuleFit algorithm serves as a bridge between interpretable rule-based models and conventional machine learning, offering an adaptable solution for precise and transparent predictive modelling (Friedman and Popescu 2008). The RuleFit algorithm has two stages. RuleFit first employs decision tree ensembles like Random Forest to capture intricate relationships in input features and the target variable. Subsequently, it extracts all the if-then-else rules from these ensemble decision tree models. The extracted rules are then used to extend the original dataset with new boolean features. This extension enhances the model's ability to capture non-linear interactions (Pornprasit et al. 2021). In the second stage, a sparse logistic regression model integrates the augmented dataset, resulting in the final model. Our study leverages RuleFit to enhance the $CostAwareCR$. In particular, we use its rule

importance assignment to extend the original feature set (Friedman and Popescu 2008) with the most prominent rules to accurately predict whether a given change is going to be merged/abandoned. $CostAwareCR$learns later an LR model using the extended dataset with the selected rules. The weights of all the features (i.e., the original data features and the newly added rule-based features) are then optimized by NSGA-II to maximize the predictive performance of the LR model and its prioritization ability.

2.2 Motivating Example

In this section, we provide real-world examples to motivate the need for early predicting and prioritizing code review requests. In practice, prioritizing code review tasks is often challenging for developers since it requires developers to check hundreds of code review requests that are committed by developers (Gousios et al. 2015, 2016). The lack of prioritization of code reviews properly may lead to wasted time/effort for developers (Fan et al. 2018). On one hand, reviewers can spend considerable time reviewing a code change that ends-up to be abandoned. An example of such changes can be observed in the code change number #65590[3] extracted from the project LibreOffice. This code change has a large size since it has more than 100 changed lines spanning over 7 files. We observe that despite reviewers spending more than 3 months with significant effort in reviewing the change since the change took 15 revisions and involved 4 reviewers that discussed 15 inline comments with the author of the code change. Despite the effort of the author and the reviewers, the change ended up being abandoned. Such abandoned code changes may lead to a significant waste of time and effort, with potential unforeseen release delays.

Another major problem related to code review consists of the lack of prioritizing code changes that are likely to be easily merged. An example of such problems can be observed in the code change number #57241[4] from LibreOffice. This code change was submitted on 10th July 2018 and considered as a small change impacting only one file with 31 changed lines. However, the code change has been abandoned two times due to inactivity leading to 4 months of wasted time. Finally, the reviewer provided his feedback and merged the change on the 15th of November 2018. Hence, we observe that some small changes can take an unreasonably long time to be merged despite a relatively easy code change that requires a low review effort. By a closer investigation of the change history data, we found that there were 228 open code review requests related to the repository of this change at the moment of its submission, among them, 113 are small changes (i.e., with a code churn ≤ 50). Thus, if a reviewer decides to select one of those review tasks, she/he needs to go through 228 code review requests which is considered to be a cumbersome and time-consuming task (Gousios et al. 2015, 2016). However, in practice reviewers are unlikely to go through a long list of review requests.

Hence, an efficient code review prioritization method will help reviewers optimize their efforts. We believe that a tool that prioritizes code changes based on their likelihood to be merged as well as their required review effort can help reviewers to effectively prioritize code reviews and reduce the overall review effort. Such a tool can help developers to prioritise code requests that are likely to be merged and require a small review effort. By doing so, reviewers will maximize the number of merged changes by spending minimum review effort.

[3] https://gerrit.libreoffice.org/c/core/+/65990/

[4] https://gerrit.libreoffice.org/c/core/+/57241/

3 Related Work

Existing works that are related to our work can be divided into two main categories, (1) studies attempted to understand the main factors that influence the code review outcome, and (2) studies attempted to predict the code review outcome using ML approaches.

Factors Influencing Code Review Outcome Recent works showed that several technical and non-technical factors can significantly impact the code review outcome. Several technical factors including the size of the change (Weißgerber et al. 2008; Baysal et al. 2016), the description of the change (Thongtanunam et al. 2017; Jiang et al. 2013), the used programming language, the commits count and the number of modified files (Soares et al. 2015) can influence the outcome of code reviews. These factors have been also studied by Gousios et al. (2014) to predict whether a given pull request is going to be merged/abandoned. Other non-technical factors that influence the outcome of a given code review request have been identified by researchers namely, the number of reviewers (Jiang et al. 2013), the author and the reviewers' experiences (Bosu and Carver 2014; Baysal et al. 2016), and the interpersonal relationship between developers (Egelman et al. 2020). In particular, Fan et al. (2018) extended the features used by (Gousios et al. 2014) by socio-technical features and obtained better results for early code review outcome prediction. Finally, Khatoonabadi et al. (2021) investigated factors that lead to wasted contributions in pull requests. They concluded that pull requests are abandoned due to the inability of contributors to satisfy the reviewers' needs and to the lack of review. Recently, Islam et al. (2022) showed that reviewers-related features are crucial to the code review early prediction. In their study, they showed that their reduced set of features achieves state-of-the-art results compared to the study of Fan et al. (2018).

Early Code Review Outcome Prediction and Prioritization Previous studies by Gousios et al. (2015; 2016) suggest that prioritizing code review requests remains a major challenge for project maintainers since they need to check the list of all code review requests in order to decide which requests to handle. To this end, various previous studies attempted to propose models to prioritize code review requests based on their likelihood to be merged (e.g., recommending the code review requests that are most likely to be merged in the first place) (Fan et al. 2018; Islam et al. 2022). These approaches may help project maintainers and contributors focus on the most likely merged code changes without wasting time on code review requests that are likely to be abandoned. One of the earliest works was proposed by (Gousios et al. 2014) in which they proposed an ML approach to predict whether a given pull request is going to be merged based on 14 features. Similarly, Zhao et al. (2019) proposed a learning-to-rank approach to prioritize changes that are going to be merged. Recently, Fan et al. (2018) proposed an approach based on Random Forest (RF) to predict whether a code change is going to be merged/abandoned and they use the learned merge probabilities to rank code review requests. Lately, Islam et al.(2022) improved the approach of Fan et al. (2018) by proposing $PredCR$ that uses Light Gradient Boosting Machines (LGBM) and a reduced set of features. The previous approaches focus on improving the prediction performance of their approaches. In particular, the previous approaches leverage different ML techniques namely, Logistic regression, Naive Bayes, Decision trees (Gousios et al. 2014), Random Forest (Gousios et al. 2014; Fan et al. 2018) and LGBM (Islam et al. 2022) by focusing on prioritizing code reviews by their likelihood to be predicted however, the previous approaches did not attempt to incorporate the code review effort dimension when prioritizing code reviews and only considered the prediction performance when learning their models.

In particular, developers may have limited time and tight deadlines which makes them not able to inspect all code changes (Gousios et al. 2015, 2016).

In this study, we fill this gap by introducing a novel approach that addresses this limitation by leveraging the code review effort when prioritizing code reviews. Our approach is based on multi-objective optimization to learn models that prioritize code reviews based on their likelihood to be merged while minimizing the estimated code review effort. By doing so, we can help developers review the maximum number of likely merged code changes with minimum effort. Kamei et al. (2012) proposed an effort-aware defect prediction model based on linear regression (EALR) that predicts the density of the defects instead of learning to classify whether a change is defective/non-defective. EALR was able to identify 35% of defects by inspecting 20% of the code. However, to the best of our knowledge, we are the first who propose an effort-aware early code review prediction model. Our approach is inspired by effort-aware defect prediction approaches (Chen et al. 2018; Canfora et al. 2013; Shukla et al. 2018). Our approach learns early code review prediction models with prominent prediction performance and incorporates the code review effort when performing code review request prioritization. To showcase the performance of our approach, we first compare it against the state-of-the-art approach of Islam et al. (2022) $PredCR$. Additionally, we compare our approach with two effort-aware ML-based approaches inspired by Kamei et al. (2012) using linear regression (EALR) and LGBM (EALGBM) that learns to predict the density of merge instead of predicting whether a code change is going to be merged/abandoned more details about the baselines are depicted in Section 5.3.

4 Proposed Approach

In this section, we present a multi-objective cost-aware approach for the early code review prediction and prioritization, $CostAwareCR$.

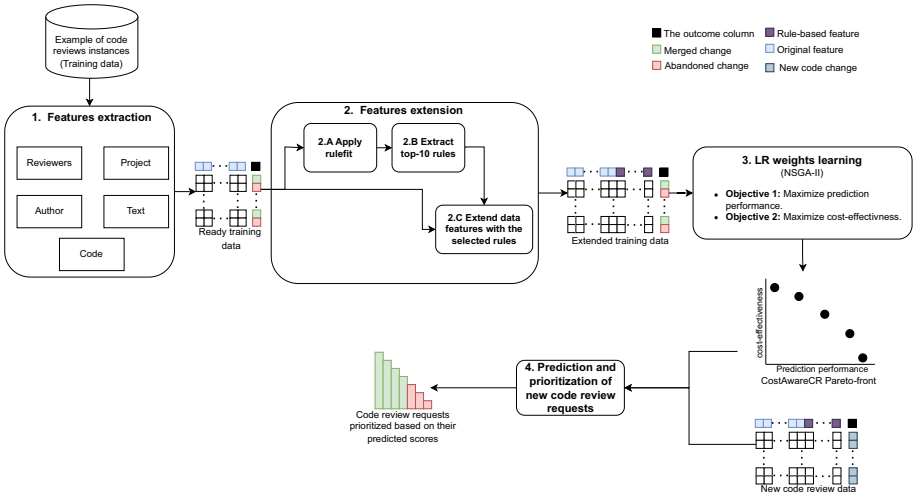


Fig. 2 Approach Overview

4.1 Approach Overview

The goal of our approach is to learn to prioritise code review requests while maintaining acceptable code review prediction performance. We adopt a search-based software engineering (SBSE) approach (Harman and Jones 2001) which allows for optimizing multiple objectives by modelling them using objective functions. Figure 2 shows an overview of our approach: $CostAwareCR$. Our approach has 4 steps.

Step 1: Feature extraction In the first step, examples of code review instances are used as a training set and we extract features related to code review. In this study, we use the features of Islam et al. (2022) to predict and prioritize code review requests since they led to the best performance compared to the state-of-the-art approaches (Islam et al. 2022). In total, 25 features are extracted spanning over 5 main dimensions, including the Reviewer, Author, Project, Text and Code. We summarize the studied features in Table 1. It is worth mentioning that these features can be computed at the moment of the submission of a code change.

Step 2: Feature extension In the next step, we use the *RuleFit* algorithm (Friedman and Popescu 2008) to extract efficient prediction rules from an ensemble decision tree model developed by Islam et al. (2022) (Step 2.A). *RuleFit* learns a sparse linear model that includes detection rules extracted from the given ensemble tree model. Our study uses the rules extracted from *RuleFit* to extend the original dataset features with more sophisticated features. To this end, we pass our data to *RuleFit* to learn the sparse linear model. Thereafter, the top-10 rules are then selected according to their importance score (Friedman and Popescu 2008) (Step 2.B). These rules are used to construct new features (Step 2.C). Thus, we obtain an extended dataset by combining the originally extracted features with the newly augmented features.

Step 3: LR weights learning The extended training dataset extracted from Step 2 is later fed to the Non-dominated Sorting genetic algorithm II (NSGA-II) (Deb et al. 2002) to learn the early code review prediction classifier. NSGA-II is a multi-objective evolutionary algorithm that employs the concept of Pareto dominance to solve multi-objective optimization problems. Additionally, NSGA-II incorporates an elitist principle, whereby the fittest individuals in a population are prioritized for selection and carried forward to the next generation (Deb et al. 2002). In this study, we use LR as a prediction model since it was previously used in a similar context (Canfora et al. 2013; Chen et al. 2018; Shukla et al. 2018) and we use NSGA-II to learn its weights since our approach is based on optimizing both the prediction performance of the model and its cost-effectiveness. To this end, NSGA-II learns the LR weights by optimizing two fitness functions: (*i*) maximize the prediction performance (*ii*) maximize the cost-effectiveness of the model. These weights are later used to calculate the probability of a given code change to be merged and all the code review requests to be prioritized according to their probabilities. At the end of this step, we obtain a set of final non-dominated solutions that we denote as the *CostAwareCR Pareto-front*.

Step 4: Prediction and prioritization of new code review requests In this step, we use the *CostAwareCR Pareto-front* to prioritize and predict the code review requests. There are multiple approaches that help select solutions from a given Pareto front like knee point (Zhang et al. 2014). Additionally, the best solution can be chosen based on the preferences of practitioners (i.e., whether the practitioner favours the prediction performance or the cost-effectiveness). Finally, a subset of the *CostAwareCR Pareto-front* solutions can be used to

Table 1 The list of used features (Islam et al. 2022)

Dimension	Feature	Description
Reviewer	avg_reviewer_experience	The averge experience of the reviewers of the studied change (Islam et al. 2022).
	avg_reviewer_review_count	The average number of code reviews per the studied change reviewers (Thongtanunam et al. 2017; Baysal et al. 2016).
	num_of_reviewers	The number of the reviewers of the studied change (Thongtanunam et al. 2017; Jiang et al. 2013).
	num_of_bot_reviewers	The number of the bot reviewers of the studied change (Islam et al. 2022).
Author	author_experience	The number of days from the change's date to the author's account registration day (Islam et al. 2022).
	author_merge_ratio	The number of merged changes by the author divided by the total number of his changes (Fan et al. 2018).
	author_merge_ratio_in_project	The ratio of the merged changes by the author in the studied change's repository (Fan et al. 2018).
	total_change_number	The total number of code changes written by the studied change author (Fan et al. 2018; Baysal et al. 2016).
	author_review_number	The number of changes reviewed by the current change's author (Fan et al. 2018; Jiang et al. 2013).
	author_changes_per_week	The number of previously closed changes by the studied change's author (Thongtanunam et al. 2017).
Project	project_changes_per_week	The closed changes per week for the studied change's repository (Thongtanunam et al. 2017).
	changes_per_author	The total number of closed changes per author (Islam et al. 2022).
	project_merge_ratio	The ratio of merged changes to the total closed changes in the last 60 days for the change's repository (Islam et al. 2022).

Table 1 continued

Dimension	Feature	Description
Text	description_length	The change's description word count (Fan et al. 2018).
	is_bug_fixing	1 the change is related to bug fixing, 0 otherwise (Fan et al. 2018; Thongtanunam et al. 2017; Kamei et al. 2012).
	is_feature	1 if the change introduces a new feature, 0 otherwise (Fan et al. 2018; Thongtanunam et al. 2017).
	is_documentation	1 if the change is related to documentation, 0 otherwise (Fan et al. 2018; Thongtanunam et al. 2017).
Code	modified_directories	The number of the modified directories of the studied change (Thongtanunam et al. 2017).
	modify_entropy	The change's entropy (Kamei et al. 2012).
	lines_added	The number of the added lines of the studied change (Jeong et al. 2009; Fan et al. 2018).
	lines_deleted	The number of the deleted lines of the studied change (Jeong et al. 2009; Fan et al. 2018).
	files_modified	The number of the modified files of studied change (Gousios et al. 2014).
	files_added	The number of the newly added files of the studied change (Fan et al. 2018).
	files_deleted	The number of the deleted files of the studied change (Fan et al. 2018).
	subsystem_num	The number of the modified sub-systems of the studied change (Fan et al. 2018).

predict and prioritize the code review requests by either performing the majority voting (Dietterich 2000) or considering the mean/median of the score of all the solutions.

4.2 NSGA-II Adaption

In our approach, we adopt NSGA-II to optimize the LR model weights since NSGA-II has been shown to be efficient in previous related studies (Deb et al. 2002; Chen et al. 2018; Shukla et al. 2018; Canfora et al. 2013). In Section 7.2, we motivate our choice of NSGA-II by comparing it to other multi-objective evolutionary algorithms, namely, NSGA-III, UNSGA-III and AGEMOEA. To adopt any multi-objective search-based algorithm such as NSGA-II, some problem-specific elements need to be specified namely, (*i*) the solution representation, (*ii*) the population initialization, (*iii*) solution variation operators and (*vi*) the fitness function. The next paragraphs are devoted to defining these elements.

4.2.1 Solution Representation

In this study, we use NSGA-II to optimize the LR weights models that will be used to compute the probability of merging a given code change. Specifically, given a dataset with N features, the goal of our approach is to learn a solution encoded as a vector of weights $W = \langle w_0, w_1, .., w_N \rangle$ for the LR model. The probability of merge of a given code change i is calculated as follows:

$$merge\_prob(i, W) = \frac{1}{1 + e^{-(w_0 + w_1 * m_{i,1} + ... + w_n * m_{i,n})}} \tag{1}$$

where $m_{i,j}$ denotes the value of feature j for the code change i. Finally, we predict if a given code change is merged/abandoned based on its probability of merge calculated earlier. The outcome of our approach, $y(i, W)$, is defined as follows:

$$y(i, W) = \begin{cases} 1(Merged) & \text{if } merge\_prob(i,W) > 0.5 \\ 0(Abandoned) & \text{if } merge\_prob(i,W) <= 0.5 \end{cases} \tag{2}$$

4.2.2 Population Initialization

NSGA-II is an evolutionary algorithm that requires creating an initial solutions population. Since we use NSGA-II to learn the weights vector of the LR model, a population of weights vectors should be initialized. In this study, a solution (i.e., a weights vector) is randomly initialized as a vector of real numbers where each item belongs to the interval $[-1000, 1000]$. The interval $[-1000, 1000]$ is chosen to allow NSGA-II to explore more combinations of weights. In practice, different weight intervals can be tried.

4.2.3 Solution Variation Operators

Solution variation operators are applied to generate the new population and guide the search for optimal solutions. In this study, we use Simulated Binary Crossover (SBX) (Deb et al. 2007). We also use Polynomial Mutation (PM) (Deb and Agrawal 1999). These operators

have been previously used in similar contexts (Deb et al. 2007; Chen et al. 2018; Shukla et al. 2018).

4.2.4 Fitness Functions

To assess the quality of the candidate solution, appropriate evaluation criteria should be established. Prior studies show that prediction performance metrics can be used to evaluate the fitness of candidate solutions (Harman and Clark 2004; Harman et al. 2012). The goal of this study is to learn to prioritize code review requests based on their code review efforts while maintaining acceptable prediction performance. We thus design our fitness function toward this goal. Our fitness function has the following objectives:

1. **Maximize prediction performance**. We use the Area Under the Curve (AUC) (Hossin and Sulaiman 2015) as an indicator of the prediction performance. The AUC is adopted since the early code review prediction problem is characterized by having highly imbalanced data (Islam et al. 2022; Fan et al. 2018). The AUC value is calculated based on the probability of merge defined in (1).
2. **Maximize cost-effectiveness**. Cost-effectiveness reflects the number of correctly identified/recommended "Merged" code reviews with respect to the needed effort to investigate them by practitioners representing the trade-off between the number of correctly predicted code reviews as merged with respect to the required effort to investigate these changes. In other words, maximizing the cost-effectiveness function aims to maximize the number of correctly identified merged changes while reducing the needed effort to inspect them. To this end, given a solution W, we define *cost_effectiveness(W)* as follows:

$$Cost\_effectiveness(W) = \alpha * Benefit(W) + (1 - \alpha) * (1 - effort(W)) \tag{3}$$

where *Benefit* denotes the total number of correctly predicted code reviews as "Merged" to the total number of "Merged" changes. The *Benefit* is defined as:

$$Benefit(W) = \frac{|\{i, wherey(i, W) = 1\} \cap \{i, wherethechange\_iis"Merged"\}|}{|\{i, wherechange\_iis"Merged"\}|} \tag{4}$$

The *Effort(W)* represents the cost of inspecting the code reviews predicted as "Merged" and is defined as:

$$Effort(W) = \frac{\sum_{i=1}^{|C|} y(i, W) * cost(c_i)}{\sum_{i=1}^{|C|} cost(c_i)} \tag{5}$$

Similar to previous studies (Canfora et al. 2013; Shukla et al. 2018; Chen et al. 2018), we use the code churn (i.e., the total number of the added and deleted lines of code in the studied change) since larger changes require more effort to inspect (Kamei et al. 2012) compared to the smaller changes. The cost function measures the code review effort for a given change c_i. Finally, we use the parameter $\alpha \in [0, 1]$ to control the trade-off between the *Benefit* and the *Effort*. In practice, the parameter α can be set based on the preferences of the practitioners.

It is worth mentioning that these functions are conflicting. For instance, if we only consider maximizing AUC while ignoring cost-effectiveness may lead classifiers with acceptable prediction performance with poor prioritization performance (Guo et al. 2018). On the other hand, if we only maximize the cost-effectiveness, the obtained model may opt to recommend

code reviews that are likely to be merged having low cost with higher false positives (i.e., abandoned changes) leading thus to low prediction performance.

5 Empirical Study Design

This section presents the details of our empirical study.

5.1 Research Questions

We evaluate our approach to whether it can effectively prioritize code review requests based on their effort while maintaining acceptable merged/abandoned prediction performance. To this end, we define the following research questions:

RQ1. **(Single-Objective vs Multi-Objective):** *Does our approach outperform single-objective GA?* The primary motivation of this research question is to investigate whether a multi-objective search is required to learn code review request prediction and prioritization. Our fitness function is based on 2 objectives to maximize the prediction performance and maximize the cost-effectiveness Section 4.2.4). The goal of RQ1 is to show that optimizing both objectives simultaneously is required to achieve the optimal model. Specifically, in this RQ, we aim to show empirically that optimizing both objectives simultaneously provides better performance than either optimizing only one objective or combining them into a single objective.

RQ2. **(*CostAwareCR* vs ML baselines):** *How does our approach perform compared to the state-of-the-art ML techniques?* To show the effectiveness of our approach, we compare the performance of our approach against the state-of-the-art ML-based approaches namely, *PredCR* proposed by Islam et al. (2022), EALR and EALGBM by Kamei et al.(2012).

RQ3. **(Performance at k%):** *How does the performance of our approach change when different portions of code review are inspected?* In practice, the availability of developers may change. In some periods, developers may have enough time to inspect large amounts of changed codes. However, in other periods, developers may have tight/limited time and effort preventing them from inspecting all code changes. Therefore, it is crucial to evaluate the performance of *CostAwareCR* at different amounts of inspected code. In this RQ, we want to investigate the performance of our approach when different percentages of code review effort are inspected and determine how much effort is needed to identify most of the merged code changes.

RQ4. **(Features importance analysis):** *What are the most important features to predict and prioritize code review requests?* We believe it is important to identify the most crucial features since this information helps practitioners debug the model and gain a deeper idea about its decision. In particular, by looking at the features' importance, developers can understand why their change is either classified to be merged or abandoned or why their code review is prioritized. Developers can then use this information to improve their code changes to have higher chances of being merged and prioritized. Therefore, RQ4 is devoted to identifying the most important features related to predicting and evaluating code review requests.

Table 2 Statistics of collected data

Project	Studied period	#Changes	Merged	Abandoned
LibreOffice	2012.03.06 - 2018.11.29	56,241	51,410 (91%)	4,831 (9%)
Eclipse	2012.01.01 - 2016.12.31	57,351	48,551 (85%)	8,800 (15%)
Gerrithub	2016.01.03 - 2018.11.29	33,020	29,367 (89%)	3653 (11%)
Total		146,612	129328(88%)	17,284(12%)

5.2 Studied Projects

We evaluate the performance of our approach using the data studied by (Islam et al. 2022). The data is extracted from 3 open source projects: Eclipse[5]. LibreOffice[6] and Gerrithub[7]. Similar to the study of (Islam et al. 2022), several filtering criteria have been applied to remove irrelevant code review instances. In particular, we apply the following criteria:

- Code reviews from repositories that have less than 200 code changes are removed since these code reviews belong to likely inactive repositories. Therefore, we keep 4 repositories for LibreOffice, 64 for Eclipse and 48 for Gerrithub.
- We discard changes that are likely to be abandoned. These changes can be identified as changes having keywords “NOT MERGE" and/or “IGNORE" in their titles.
- We remove changes that are only accepted by their owner without any other reviewer involved.
- Changes with missing data (i.e., commits data) are removed.

In Table 2, we present some statistics related to the final datasets for the studied projects.

5.3 Studied Baselines

The goal of this study is to show that multi-objective evolutionary search can help developers to prioritize code review requests based on their efforts. Thus, it is necessary to show that optimizing the two defined objectives in Section 4.2.4 simultaneously leads to better results compared to optimizing just one objective or combining them in one objective. Therefore, we first compare our approach with single-objective GA baselines optimizing either one of the objectives or a combination of the fitness function components to motivate the need for a multi-objective search to handle the conflicting nature of the components of the optimized fitness functions. To this end, first, we compare our approach against GA baselines that optimizes only AUC or $cost - effectivness$ denoted as *GA-AUC* and *GA-effectivness*. Additionally, we compare our approach with *Mono-GA* that maximizes the *Mono-fitness* which is a combination of our two objective functions, i.e., AUC (representing the prediction performance) and the *cost_effectivness* (cf. (3)) defined as:

$$Mono - fitness = \frac{AUC + cost\_effectivness}{2} \tag{6}$$

In addition, we compare our approach against state-of-the-art ML approaches investigated in pr e vious studies. In particular, we compare our approach against $PredCR$ proposed by

[5] https://git.eclipse.org/r/q/status:open+-is:wip

[6] https://gerrit.libreoffice.org/q/status:open+-is:wip

[7] https://review.gerrithub.io/q/status:open+-is:wip

Table 3 The studied hyper-parameters for each ML and GA approaches

Approach	Parameter	Value
EALR	`fit_intercept`	True
PredCR, *EALGBM*	`learning_rate`	0.01
	`n_estimators`	500
CostAwareCR, *Mono-GA*, *GA-AUC*, *GA-effectivness*	`Population size`	800
	`Max generations`	2400
	`SBX rate`	0.7
	`PM rate`	0.1
	`SBX eta`	15
	`PM eta`	20

Islam et al. (2022) using the same pre-processing and hyper-parameters used by their study. Finally, inspired by Kamei et al. (2012), we evaluated our work against effort-aware ML approaches that learn to predict the merge density defined as $D(c) = \frac{Y(C)}{cost(c)}$, where $Y(c)$ is 1 if the code change is merged c and 0 if it is abandoned. We study two baselines based on LGBM and linear regression, Effort-Aware Light Gradient machine (*EALGBM*) and Effort-Aware Linear Regression (*EALR*) (Kamei et al. 2012). It is worth mentioning that our approach and the single objective GA are implemented based on the Pymoo framework (Blank and Deb 2020), while the ML-based approaches are implemented using Scikit-Learn (Pedregosa et al. 2011), and the LightGBM library[8]. In Table 3, we present the hyper-parameters setting for our approach and the studied baselines. The hyper-parameters for our approach and the single objective GA were set by trial and error (Harman et al. 2012). The ML baseline parameters were using the grid search procedure (LaValle et al. 2004).

5.4 Performance Metrics

To evaluate the prediction performance of our approach, we use AUC as a common ML performance metric (Islam et al. 2022; Fan et al. 2018). In particular, a value of AUC higher than 0.75 is considered to be adequate for a predictive model (Romano and Pinzger 2011; Lessmann et al. 2008). Additionally, we consider the code review effort to evaluate the performance of our approach. In particular, we used two common effort-aware performance metrics used by (Kamei et al. 2012), namely, the ratio of the correctly predicted merged changes when inspecting 20% of the code that we denote as ACC, and the normalized area under the effort-based lift chart denoted as P_{opt} (Mende and Koschke 2010; Arisholm et al. 2010; Kamei et al. 2012; Chen et al. 2018). ACC is the ratio of the correctly predicted merged changes when 20% of the total effort (i.e., code churn) is investigated to prioritize code review requests. Although ACC can be computed for different code ratios, it is worth mentioning that the choice of 20% is motivated by the previous study of Ostrand et al. (2005) in which they found that most of the code issues exist in only 20% of the code. In Section 6.3, we investigate the performance of *CostAwareCR* in different ratios of inspected code. P_{opt} is the normalized area under the cumulative effort-based lift chart. Figure 3 shows an example of an effort-based lift chart. We denote *Optimal* as the model that has the perfect prioritization performance and *Worst* as the model having the worst prioritization performance. We also

[8] https://lightgbm.readthedocs.io/en/latest/index.html

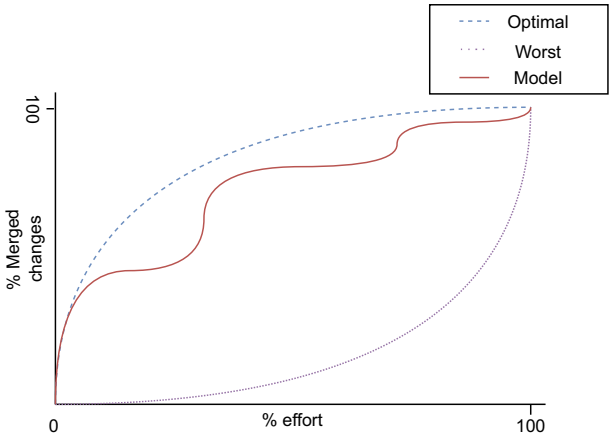


Fig. 3 An example of an effort-based lift cumulative chart

denote A_{Best}, A_{Worst} and A_{Model} as the areas under the effort-aware cumulative lift chart for the *Optimal*, *Worst* and the studied model respectively. Then P_{opt} is calculated as follows:

$$P_{opt} = 1 - \frac{A_{Best} - A_{Model}}{A_{Best} - A_{Worst}} \tag{7}$$

The higher the P_{opt} the best is the performance of the model.

5.5 Statistical Testing

In our study, we compare our approach with different baselines and configurations. To perform multiple statistical comparisons, we use Scott-Knott-ESD (SK-ESD)[9] test elaborated by Tantithamthavorn et al. (2018). 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 (Cohen 2013) to measure the effect size between different clusters and merge clusters having negligible effect size, i.e., having $d < 0.2$.

5.6 Analysis Method

Figure 4 depicts the experimental design of our study that we adopt to answer the proposed research questions. First, we follow the 10-fold online validation process to divide our dataset into training and testing sets similar to previous studies (Islam et al. 2022; Fan et al. 2018; Saidan et al. 2020; Saidani et al. 2022), since code review data is chronologically ordered. Given a dataset (i.e., a given project data), we first sort it according to the creation date of code reviews, and then we divide the dataset into 11 equal folds. At the iteration i (where $1 <= i <= 10$), we use the folds $fold_0$, ..., $fold_{i-1}$ as training dataset and we keep the $fold_i$ as a testing dataset. Thereafter, we use the training dataset to train the ML models as

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

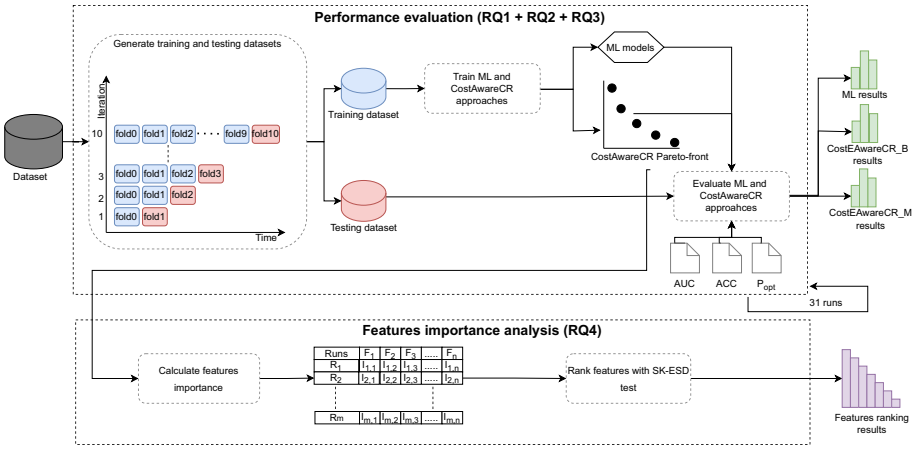


Fig. 4 Experimental setup

well as our approach. Since our approach is based on optimizing multiple objective functions, the training process of our approach will result in multiple solutions (i.e., the Pareto-front solutions). Thus, we evaluate the ML algorithms and the Pareto-front solutions obtained by our approach using the AUC, ACC, and P_{opt} performance metrics (cf. Section 5.4) on the testing dataset. To analyse the performance of our approach, we report the median of the scores obtained by our approach in the Pareto front solutions denoted as $CostAwareCR_M$ and the best score denoted as $CostAwareCR_B$ similar to Chen et al. (2018). This process is repeated 31 times to reduce the effect of the randomness for all the approaches (Arcuri and Briand 2011). Thus, our results are based on 310 independent runs (i.e., 31 runs $\times$ 10 folds).

To answer RQ1, we compare $CostAwareCR$ against the $Mono\text{-}GA$ baseline (cf. Section 5.3) in terms P_{opt}, ACC and AUC metrics (cf. Section 5.4).

To answer RQ2, we compare our approach against the ML three baselines, $PredCR$ (Islam et al. 2022), *EALGBM* (Kamei et al. 2012) and *EALR* (Kamei et al. 2012) (cf. Section 5.3). We follow the same experimental process depicted earlier. We compare the studied models based on the P_{opt} and ACC metrics defined in Section 5.4 and we use the SK-ESD test to rank all the approaches.

To answer RQ3, we plot the recall at the K% code review effort for different percentages for all the studied projects and for our approach and the ML baselines.

RQ4 is devoted to identifying the most crucial features that our model uses when predicting and prioritizing code review requests. To answer this RQ, we use the obtained Pareto-front models obtained for all the independent runs. For each of the obtained models, we compute its features' importance. Since our model learns the weights for a combination of the original features (i.e., linear terms) and rule-based features from the RuleFit algorithm, we refer to the approach of Friedman and Popescu (2008). For a given linear term for variable x_j denoted as $l(x_j)$, we define its importance as:

$$I_{l(x_j)} = |\alpha_j| * std(x_j) \tag{8}$$

where α_j is the learnt weight for $l(x_j)$ and $std(x_j)$ is the standard deviation of variable x_j. Additionally, given a rule term r_k, we define its importance as:

$$I_{r_k} = |\beta_k| * \sqrt{s_k * (1 - s_k)} \tag{9}$$

where β_k is the learnt coefficient for the rule term r_k and s_k is the support of the rule r_k (i.e., the ratio of the number of the data points where r_k applies).

Finally, since a variable x_j may occur in different rules, we define its importance as

$$I_{x_j} = I_{l(x_j)} + \sum_{x_j \in r_k} \frac{I_{R_k}}{size(r_k)} \tag{10}$$

where $size(r_k)$ denotes the number of the literals in the rule r_k. For a given model, in a given run R_i, we compute the importance of all variables. Then, we use the median of the importance values across all the Pareto-front models to obtain the final importance of the variable x_j in the run R_i since we are dealing with a set of solutions. Finally, we pass the importance values for all the runs to the SK-ESD test (cf. Section 5.5) to rank the features according to their importance values.

5.7 Data Availability Statements

To further extend and replicate our study, we provide our comprehensive replication package (CostAwareCR 2023) that includes the datasets, the implementation of $CostAwareCR$and the necessary scripts used in our experiments.

6 Empirical Study Results

In this section, we present the obtained results of the empirical study for all the research questions.

6.1 (RQ1) Does Our Approach Outperform Single-objective GA?

In Fig. 5, we show a concrete example of the conflicting nature of the AUC and the *cost-effectiveness* metric. From the Figure we can observe that when maximizing the AUC, the *cost-effectiveness* is being degraded. This shows that the fitness function components are conflicting.

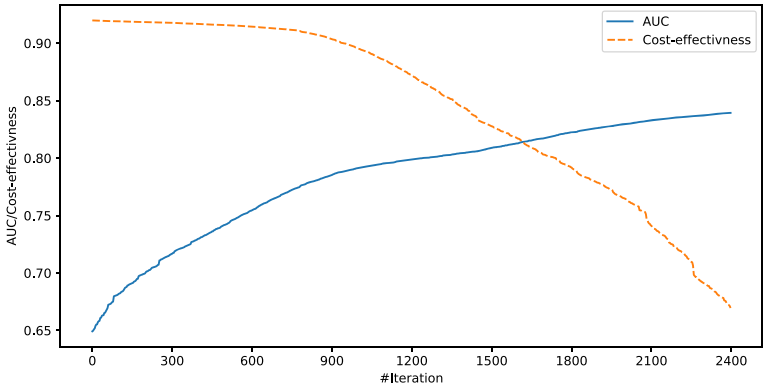
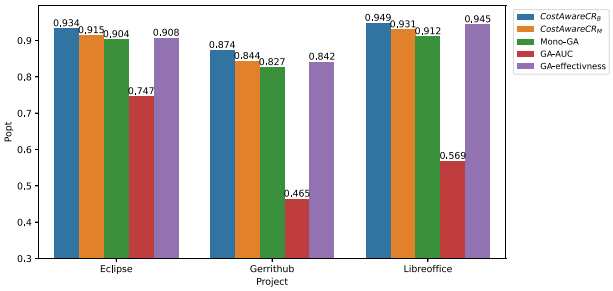


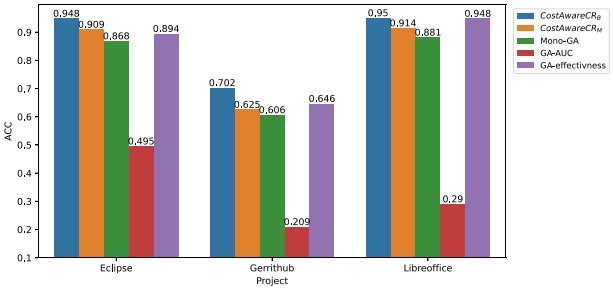
Fig. 5 AUC and Cost-effectiveness are conflicting

Additionally, Fig. 6 shows the P_{opt}, ACC, and AUC values achieved by $CostAwareCR_B$, $CostAwareCR_M$, $Mono\text{-}GA$, $GA\text{-}AUC$ and $GA\text{-}effectivness$ across the studied projects. In Fig. 6, we observe that $CostAwareCR$achieves the best trade-off between the prediction and the prioritisation performances. In all the studied projects, we observe that optimizing one objective only leads to biased performance. For example, if we only optimize AUC using GA, we end up with models having high AUC scores ($\sim$ 0.8) accompanied by low P_{opt} and ACC scores (a maximum of 0.75 for P_{opt} and 0.49 for ACC). Similarly, optimizing only the cost-effectiveness leads to models having high P_{opt} and ACC scores with poor predictive performance (i.e., low AUC of $\sim$ 0.56). Furthermore, from Fig. 6, we observe that $CostAwareCR$achieves the best trade-off between the predictive and prioritisation performances achieving acceptable AUC, ACC and P_{opt} scores.

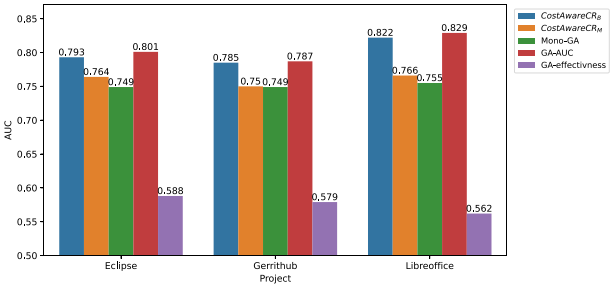
Looking at P_{opt} and ACC, we observe that $CostAwareCR$outperforms $Mono\text{-}GA$ in the studied projects. For example, for Gerrithub, while $CostAwareCR_B$and



(a) P_{opt} scores.



(b) ACC scores.



(c) AUC scores.

Fig. 6 The achieved P_{opt}, ACC, and AUC for $CostAwareCR$and $Mono\text{-}GA$ across the studied projects

$CostAwareCR_M$ achieved P_{opt} scores of 0.87 and 0.84 and ACC scores of 0.7 and 0.95 respectively, we found that the $Mono\text{-}GA$ achieves 0.83 for P_{opt} and 0.61 for ACC. The same observations apply to the other studied projects. Additionally, we observe that $CostAwareCR_B$ leads always to better results compared to the $Mono\text{-}GA$ baseline. This can be explained by the fact that the multi-objective search can better cover the search space leading to exploring better-performing solutions since it maintains a diverse set of solutions (Segura et al. 2016).

Additionally, we found that $CostAwareCR_M$ achieves a median AUC score greater or equal to 0.75 for all the studied projects. We observe that for all the studied projects, the obtained AUC values are close ranging from 0.75 to 0.77 where $CostAwareCR_M$ achieved 0.76 for Eclipse and LibreOffice and 0.75 for Gerrithub. Therefore, we conclude that our approach has promising early code review prediction performance which indicates that our approach can be used to predict whether a given code review request is merged/abandoned similar to the previous state-of-the-art approaches (Islam et al. 2022; Fan et al. 2018). Additionally, we observe that $CostAwareCR_M$ achieves similar or better AUC scores compared to $Mono\text{-}GA$ indicating that $CostAwareCR$ has similar or better prediction performance.

Therefore, we conclude that optimizing the prediction performance and cost-effectiveness objectives simultaneously instead of using a single objective search leads to a better trade-off in performance.

RQ1 summary: *$CostAwareCR$ achieves the best trade-off between predictive performance and prioritization capabilities. $CostAwareCR$ outperforms the single-objective GA baseline $Mono\text{-}GA$. $CostAwareCR$ achieves promising AUC scores above 0.75 for all the studied projects. Thus, a multi-objective search is more efficient in our context.*

6.2 (RQ2) How Does Our Approach Perform Compared to the State-of-the-Art ML Techniques in the Within-Project Context?

We report the obtained results in Table 4. Based on the obtained results we can see that $CostAwareCR_M$ and $CostAwareCR_B$ the best P_{opt} and ACC scores across the studied projects. For example, for LibreOffice, we observe that $CostAwareCR_M$ achieves 0.93 in terms of P_{opt} outperforming $EALGBM$ (0.80), $PredCR$ (0.65) and $EALR$ (0.57). The same observation also applies to the other studied projects. Looking at the ACC results, we observe that $CostAwareCR$ outperforms the baselines with large margins. For example, in LibreOffice, $CostAwareCR_M$ achieved 0.91 compared to 0.66 for $EALGBM$ (the best ML baseline for LibreOffice). Additionally, we observe that $CostAwareCR$ keeps the highest SK-ESD ranks in all the studied projects. For instance, $CostAwareCR_B$ always achieves the rank 1 while $CostAwareCR_M$ achieves the SK-ESD test rank 2 making it similar or better compared to the studied ML baselines. It is worth mentioning that while ACC scores of $CostAwareCR_M$ are above 0.9 for Eclipse and LibreOffice, $CostAwareCR_M$ achieved an ACC score of 0.7 for Gerrithub. This can be explained by the fact that Gerrithub has considerably lesser data than the other projects. In some cases, we observe that $CostAwareCR_M$ and $EALGBM$ achieve competitive performances with equal SK-ESD tests (i.e., P_{opt} for Eclipse and ACC for Gerrithub) this can be due to the fact that while $EALGBM$ is based on LGBM model (i.e., involving an ensemble of decision trees), our model is only using a Logistic regression model making it more transparent. Despite this, we still observe that $CostAwareCR$ outperforms the ML baselines in most cases. Overall, $CostAwareCR_M$ achieved a P_{opt} score of 0.89 and an ACC score of 0.81 with a relative

Table 4 The achieved P_{opt} and ACC scores by $CostAwareCR$compared to the ML baselines

Project	Approach	$P_{opt}(rank^*)$	$ACC(rank^*)$
Eclipse	$CostAwareCR_B$	**0.934 (1)**	**0.948 (1)**
	$CostAwareCR_M$	0.915 (2)	0.909 (2)
	$EALGBM$	0.912 (2)	0.83 (3)
	$EALR$	0.769 (3)	0.617 (4)
	$PredCR$	0.802 (3)	0.629 (4)
LibreOffice	$CostAwareCR_B$	**0.949 (1)**	**0.95 (1)**
	$CostAwareCR_M$	0.931 (2)	0.914 (2)
	$EALGBM$	0.805 (3)	0.665 (3)
	$EALR$	0.575 (5)	0.227 (5)
	$PredCR$	0.652 (4)	0.435 (4)
Gerrithub	$CostAwareCR_B$	**0.874 (1)**	**0.702 (1)**
	$CostAwareCR_M$	0.844 (2)	0.625 (2)
	$EALGBM$	0.823 (3)	0.644 (2)
	$EALR$	0.607 (4)	0.278 (4)
	$PredCR$	0.586 (4)	0.378 (3)
Average	$CostAwareCR_B$	**0.919 (1)**	**0.867 (1)**
	$CostAwareCR_M$	0.897 (2)	0.816 (2)
	$EALGBM$	0.846 (3)	0.713 (3)
	$EALR$	0.651 (4)	0.374 (5)
	$PredCR$	0.68 (4)	0.481 (4)

* *Rank* is the SK-ESD rank. The lower the rank the better the performance
Approaches with *Rank* 1 are in bold

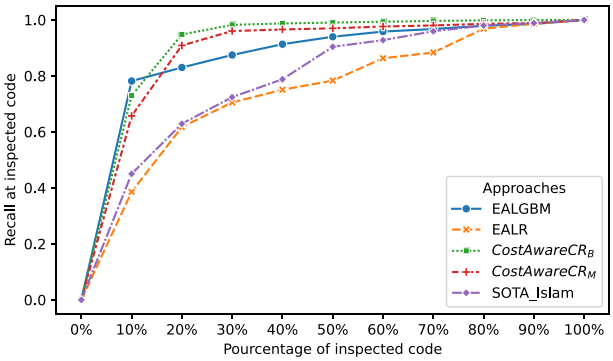
improvement ranging from 6% to 38% in terms of P_{opt} and an improvement ranging from 6% to 132% in terms of ACC. Thus, we conclude that our approach is more effective in prioritizing code review requests compared to the studied baselines.

Despite using a simple model based on logistic regression, our approach is able to outperform a more complex ML approach based on complex ensemble decision tree models. This can be explained by the fact that while the ML approach uses greedy search strategies (i.e., gradient descent (Ruder 2016) and gradient boosting (Natekin and Knoll 2013)) that can quickly overfit, our approach through the evolutionary search can explore the solutions search space more efficiently which helps $CostAwareCR$to find more optimal solutions.

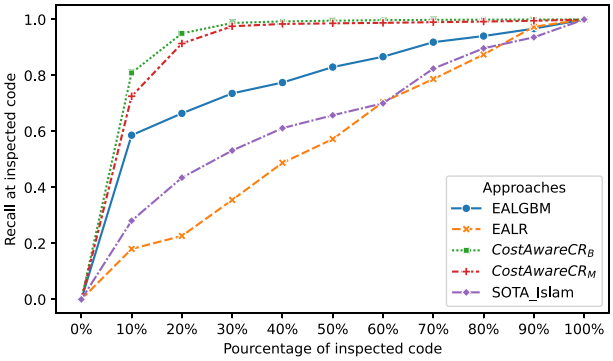
> ***RQ2 summary:*** *$CostAwareCR$achieves 0.90 and 0.82 in terms of P_{opt} and ACC scores significantly outperforming the ML baselines. Thus, $CostAwareCR$is suitable for code review requests prioritization.*

6.3 (RQ3) How Does the Performance of our Approach Change When Different Portions of Code are Inspected?

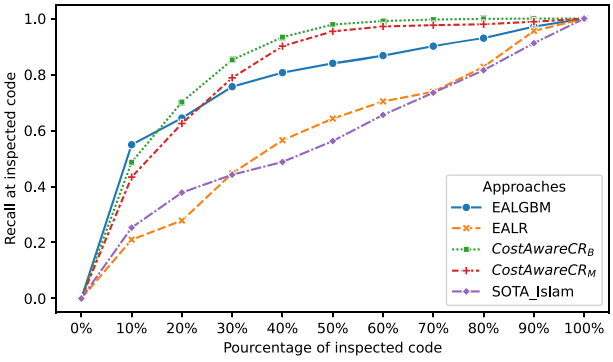
Figure 7 shows the recall at K % for our approach and the baselines for all the studied projects. We observe that for each studied project, our approach achieves consistently better recall at different inspected percentages. In particular, we observe that starting from K = 20%,



(a) Recall at % K for Eclipse.



(b) Recall at % K for LibreOffice.



(c) Recall at % K for Gerrithub.

Fig. 7 Recall at % K for our approach and the studied baselines for the studied projects

our approach outperforms the other baselines. For example for LibreOffice, when we set K to 30% $CostAwareCR_M$ achieves a recall of 0.98 compared to 0.74 for $EALGBM$, 0.53 for $PredCR$ and 0.36 for $EALR$. These results indicate that using only 30% of the effort, developers can identify nearly 98% of the merged code reviews compared to only 74% if they use $EALGBM$. Overall, we observe that the recall of our approach converges to 1 quicker than the other approach. In particular, we observe that starting from $K = 40\%$ our approach can identify more than 95% of the merged code reviews for all the studied projects, while the studied baseline scores identify only 80% of the merged changes. Therefore, our approach can help developers identify most of the merged code reviews while only inspecting 40% of the code reviews. In particular, developers cannot inspect all the code assigned to them due to several reasons such as tight deadlines. Therefore, developers can use our approach to prioritize code reviews based on their likelihood to be merged and their code review effort to get the maximum number of code reviews merged with the minimum code review effort.

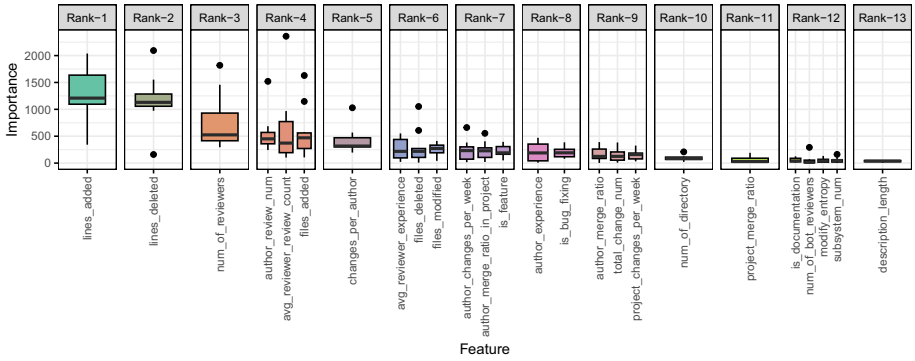
RQ3 summary: *$CostAwareCR$outperforms the ML baselines when different levels of code are inspected. Our approach needs only 40% of the effort to identify more than 90% of the merged code reviews.*

6.4 (RQ4) What are The Most Crucial Features for Predicting and Prioritizing Code Review Requests?

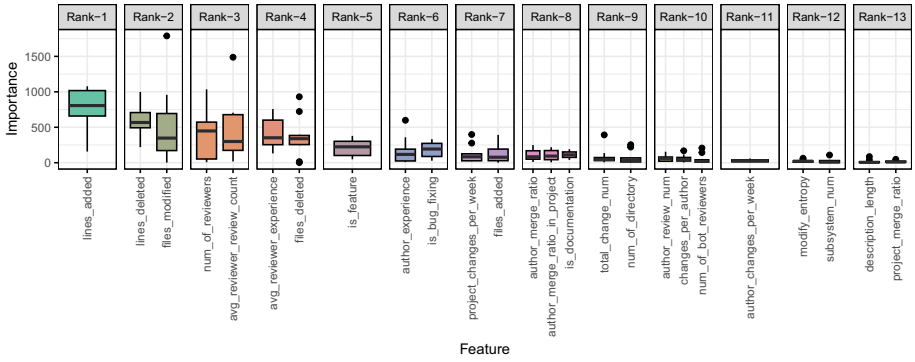
Figure 8 shows the feature importance ranking for the studied projects.

Link the code change size We observe that features that characterize the size of the code change namely, `lines_added`, `lines_deleted`, `files_added`, `files_modified` and `files_deleted` are among the top 10 features across all the projects. In particular, we observe that features `lines_added` and `lines_deleted` are ranked as the first and the second important feature for Eclipse and LibreOffice. The feature `files_added` is the most important feature for Gerrithub and the features `files_modified` and `files_deleted` appear among the top 10 important features for all the studied projects. This observation can be explained by (*i*) the fact that our approach learns to prioritize code review requests based on their complexity and (*ii*) complex changes are less relevant for developers and thus have lower odds of being merged (MacLeod et al. 2017; Weißgerber et al. 2008).

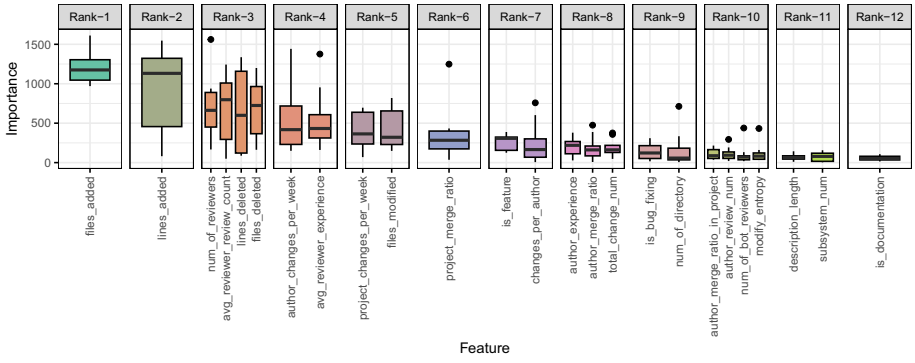
Link the Developers' Experience and Involvement Our analysis reveals that the authors'/reviewers' experience and involvement in the project play a major role when predicting and prioritizing a given code review request. The obtained results suggest that having more reviewers will increase the likelihood of merging changes since we have the feature `num_reviewers` ranked as the third feature for all the studied projects. Developers should focus their efforts on changes having a higher number of reviewers since they are more likely to be merged. A similar result has been investigated in previous studies where it is observed the number of reviewers of a given change is linked to its likelihood to be merged (Islam et al. 2022; Rigby and Bird 2013). Additionally, we observe that reviewers'/authors' experiences are also crucial for code review prediction and prioritization. From Fig. 8, we also observe that features related to developers' experience like `avg_reviewer_review_count`, `avg_reviewer_review_experience`, `author_review_count`, and `author_changes_per_week` are among the top ten features for all the studied projects. The impact of developers' experience and their past involvement have been shown to impact the code review outcome in previous studies (Thongtanunam et al. 2017; Jiang et al. 2013).



(a) Features ranking for Eclipse.



(b) Features ranking for LibreOffice.



(c) Features ranking for Gerrithub.

Fig. 8 Features ranking for the studied projects

RQ4 summary: *Features related to the code change size and developers' experience and involvement are the most prominent features for predicting and prioritizing code review requests.*

7 Discussions and Implications

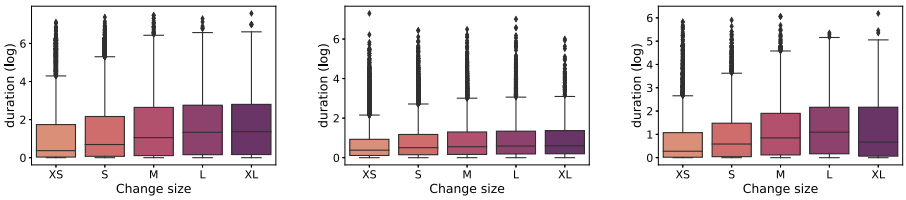
In this section, we present further analysis regarding the design and the performance of our approach then, we present the implication of the obtained results.

7.1 Does Code Churn Reflect the Review Effort?

Following previous studies (Canfora et al. 2013; Shukla et al. 2018; Chen et al. 2018), we use code churn as a proxy of the code review effort. In this section, we show that indeed code churn can be used as a proxy for the review effort. Hence, we investigate whether having a higher code churn leads to a higher review effort. We characterize the code review effort using three metrics, (1) the review duration (expressed in hours), (2) the number of review iterations, and (3) the number of exchanged messages. In this analysis, we follow the size convention of Gerrit and divide the code reviews into five groups based on their code churn size. We label code reviews with a code churn < 10 as extra small (XS). Code reviews having a code churn ≥ 10 and < 50 are labelled as small (S). Code reviews having a code churn ≥ 50 and < 250 are labelled as medium (M). Large code reviews (L) are those that have a code churn ≥ 250 and $< 1,000$. Finally, code reviews having a code churn $\geq 1,000$ are labelled as extra-large (XL). Furthermore, we calculate the Spearman correlation (De Winter et al. 2016) between the code churn and the review effort metrics.

Figures 9, 10, and 11 show the boxplots of the durations of the code reviews for the different code churn labels across the studied projects. Note that since the code review data is highly skewed, we apply the log transformation $y = \log(1 + x)$ to reduce the skewness of the code review effort metrics. From Fig. 9, we observe that larger code reviews take generally a higher duration to close them compared to code reviews with smaller sizes. The same observation also applies to the number of review iterations where we observe that code reviews having a higher code churn require more review iterations. Finally, we observe from Fig. 11 that developers tend to exchange more messages when integrating larger code changes. In Table 5, we show the Spearman correlation scores between the code churn and the review effort metrics with their p-values. The obtained results indicate a weak yet significant correlation between the code churn and code review effort metrics. In particular, the highest Spearman values are observed between the code churn and #Revisions (0.25 for Eclipse, 0.23 for LibreOffice, and 0.28 for Gerrithub).

From the previous experiments, we observe that larger changes require more review effort. However, due to the small positive Spearman values, we could not confirm a strong correlation between code review effort and code churn despite having p-values that are near 0. In practice, it is challenging to know the duration of the #Revisions of a given code review request at the moment of its submission as shown in previous studies (Chen et al. 2022; Huang et al. 2022; Chouchen et al. 2023). Therefore, code-churn is only used as a proxy of effort in our study and future research should focus on designing more sophisticated and context-aware code review effort predictors.

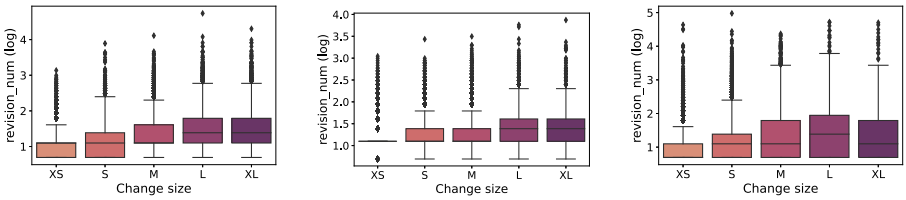


(a) Duration vs churn size for Eclipse.

(b) Duration vs churn size for LibreOffice.

(c) Duration vs churn size for Gerrithub.

Fig. 9 Review duration boxplots with respect to the different code churn sizes across the studied projects (The duration is in the log scale)

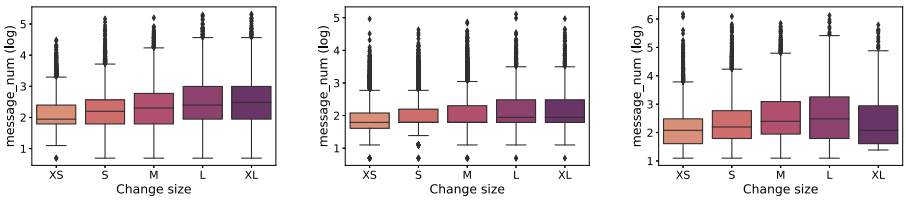


(a) #iterations vs churn size for Eclipse.

(b) #iterations vs churn size for LibreOffice.

(c) #iterations vs churn size for Gerrithub.

Fig. 10 Review iterations count boxplots with respect to the different code churn sizes across the studied projects (the review iterations count is in the log scale)



(a) #messages vs churn size for Eclipse.

(b) #messages vs churn size for LibreOffice.

(c) #messages vs churn size for Gerrithub.

Fig. 11 The number of the exchanged messages between developers counts boxplots with respect to the different code churn sizes across the studied projects (the number of the exchanged messages is in the log scale)

Table 5 The Spearman correlation scores between the code churn and the review effort metrics: duration, #messages, and #revisions

Project	Duration (P-value)	#Messages (P-value)	#Revisions (P-value)
Eclipse	0.168 (0.00)	0.222 (0.00)	0.249 (0.00)
LibreOffice	0.118 (0.00)	0.176 (0.00)	0.226 (0.00)
Gerrithub	0.223 (0.00)	0.207 (0.00)	0.276 (0.00)

Table 6 The achieved P_{opt}, ACC, and AUC scores by $CostAwareCR$ using different MOEAs

Project	Approach	$P_{opt}(rank)$	$ACC(rank)$	$AUC(rank)$
Eclipse	NSGA-II_M	0.915 (2)	0.909 (2)	**0.764 (1)**
	NSGA-III_M	0.909 (3)	0.905 (2)	**0.767 (1)**
	UNSGA-III_M	**0.922 (1)**	**0.926 (1)**	**0.77 (1)**
	AGEMOEA_M	0.917 (2)	**0.918 (1)**	**0.764 (1)**
LibreOffice	NSGA-II_M	**0.931 (1)**	**0.914 (1)**	0.766 (2)
	NSGA-III_M	**0.929 (1)**	**0.905 (1)**	**0.773 (1)**
	UNSGA-III_M	0.924 (2)	**0.891 (1)**	0.762 (2)
	AGEMOEA_M	0.923 (2)	**0.905 (1)**	0.765 (2)
Gerrithub	NSGA-II_M	**0.844 (1)**	**0.625 (1)**	**0.75 (1)**
	NSGA-III_M	**0.842 (1)**	**0.626 (1)**	**0.743 (1)**
	UNSGA-III_M	**0.842 (1)**	**0.629 (1)**	**0.745 (1)**
	AGEMOEA_M	**0.843 (1)**	**0.627 (1)**	**0.743 (1)**
Average	NSGA-II_M	**0.897 (1)**	**0.816 (1)**	**0.76 (1)**
	NSGA-III_M	**0.893 (1)**	**0.812 (1)**	**0.761 (1)**
	UNSGA-III_M	**0.896 (1)**	**0.815 (1)**	**0.759 (1)**
	AGEMOEA_M	**0.894 (1)**	**0.816 (1)**	**0.757 (1)**

Rank is the SK-ESD rank. The lower the rank the better the performance
Approaches with *Rank* 1 are in bold

Table 7 The achieved HV and GD scores by $CostAwareCR$ using different MOEAs

Project	Approach	$HV(rank^*)$	$GD(rank^*)$
Eclipse	NSGA-II	**0.755 (1)**	**0.302 (1)**
	NSGA-III	0.753 (2)	**0.295 (1)**
	UNSGA-III	0.751 (2)	**0.295 (1)**
	AGEMOEA	0.752 (2)	**0.301 (1)**
LibreOffice	NSGA-II	**0.761 (1)**	**0.358 (1)**
	NSGA-III	0.752 (2)	**0.358 (1)**
	UNSGA-III	0.751 (2)	**0.358 (1)**
	AGEMOEA	0.75 (2)	0.366 (2)
Gerrithub	NSGA-II	**0.724 (1)**	**0.373 (1)**
	NSGA-III	0.716 (3)	**0.381 (1)**
	UNSGA-III	0.718 (2)	**0.38 (1)**
	AGEMOEA	0.71 (4)	0.394 (2)
All	NSGA-II	**0.747 (1)**	**0.344 (1)**
	NSGA-III	0.74 (2)	**0.345 (1)**
	UNSGA-III	0.74 (2)	**0.344 (1)**
	AGEMOEA	0.737 (2)	**0.354 (1)**

*$Rank$ is the SK-ESD rank. The lower the rank the better the performance
Approaches with *Rank* 1 are in bold

7.2 What is the performance of *CostAwareCR* When Using Other Multi-Objective Evolutionary Algorithms?

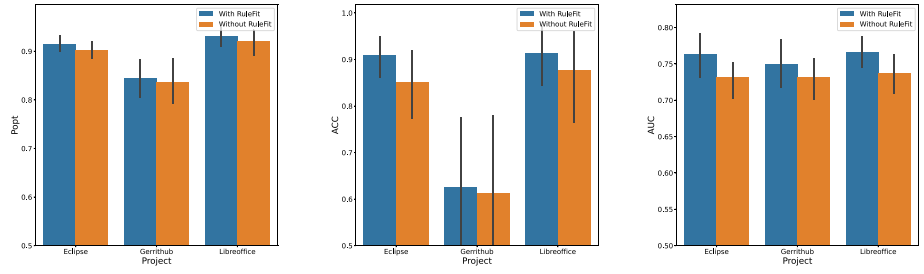
In our study, we used NSGA-II as a search algorithm to optimize the weights for our LR model. We thus investigate the performance of other common Multi-Objective Evolutionary Algorithms (MOEAs). Hence, we compare NSGA-II with other existing MOEAs namely, NSGA-III (Deb and Jain 2013), UNSGA-III (Seada and Deb 2015) and AGEMOEA (Panichella 2019). As a first step, we compare the selected MOEAs in terms of P_{opt}, ACC and AUC. We summarize the obtained results in Table 6. Overall, we observe that all the studied MOEAs obtained competitive P_{opt}, ACC and AUC scores across the studied projects. As a second step, we evaluate the solutions set qualities of the different MOEAs since all of them are based on the Pareto dominance. To achieve this, we compare the studied MOEAs using the following solution quality indicators that are well-adopted in the SBSE community (Wang et al. 2016; Riquelme et al. 2015):

- **Hypervolume (HV)**: is the volume of the space covered by the non-dominated solutions. Higher values of HV are preferred since they indicate better exploration of the search space (Zitzler and Thiele 1998).
- **Generational Distance (GD):** is the average distance from the obtained Pareto-front solutions to the true Pareto-front solutions (Van Veldhuizen et al. 1998). In our case, the true Pareto-front is a single solution having an AUC score of 1 and a P_{opt} of 1 since having a P_{opt} equal to 1 indicates a perfect trade-off between the $Benefit$ and $effort$ (cf. (3))

We report the obtained HV and GD scores for the different MOEAs in Table 7. Overall, we observe that while the MOEAs have competitive HV values, NSGA-II still achieves the highest scores having the lowest SK-ESD scores. Considering the GD scores, we observe that all MOEAs achieve similar scores. These results justify the choice of NSGA-II as the default search algorithm for $CostAwareCR$since it leads to better coverage of the search space compared to the other studied MOEAs.

7.3 Does the Use of RuleFit Improve the Performance of *CostAwareCR*?

In our study, we use RuleFit to extend the original features with rule-based features to improve the performance of our approach. In this section, we show that using RuleFit helps improve



(a) P_{opt} scores (b) ACC scores (c) AUC scores

Fig. 12 The performance of $CostAwareCR$when using *RuleFit* compared to the performance without using *RuleFit*

our approach's performance. Figure 12 shows the performance of $CostAwareCR$ when using RuleFit compared to $CostAwareCR$ without RuleFit in terms of P_{opt}, ACC and AUC. We observe that using RuleFit leads to an improvement in the performance of our approach. In particular, we observed an improvement ranging from 2% to 6% in terms of ACC and AUC however we observe that both approaches have similar P_{opt} scores. Therefore, adding RuleFit can improve the overall performance of our approach in terms of prediction and cost-effectiveness. The benefit of the performance when adding the features extracted from RuleFit is explained by the fact that these features are extracted from high-performing ensemble decision tree approaches like LGBM. It is worth mentioning that there are multiple pre-processing techniques like feature reduction techniques that may help improve the performance of $CostAwareCR$. Investigating such techniques can be done as future work.

7.4 Implications

This study aims to learn to predict and prioritize code review requests based on their likelihood to be merged and their required review effort. To this end, $CostAwareCR$, a multi-objective search-based method has been proposed to maximize the prediction performance and the cost-effectiveness of an LR model. Therefore, $CostAwareCR$ can be helpful for different actors in the software engineering field.

7.4.1 Implications for Software Practitioners

Our approach can help developers to predict and prioritise code review requests. Our empirical study showed that our approach achieves promising AUC values and the best P_{opt} and ACC values compared to the ML baselines. These results make the prediction of our approach reliable. In practice, our approach can be used in two ways: (*i*) $CostAwareCR$ can help developers to identify code reviews that are likely to be merged since our approach achieved a prominent AUC value above 0.75 across all the studied projects. (*ii*) $CostAwareCR$ can help developers prioritize code review requests. In particular, developers can decide how much effort they can allocate for code review in terms of lines of code and $CostAwareCR$ can provide them with a list of code review requests that are likely to be merged within their allocated effort. In fact, in Section 6.3, we showed that $CostAwareCR$ can identify most of the merged code reviews within an effort of 40%. Therefore, we believe that $CostAwareCR$ is useful when developers have limited review time.

7.4.2 Implications for Researchers

In this study, we proposed $CostAwareCR$ which helps developers to predict and prioritise code reviews. In Section 6.1, we compare $CostAwareCR$ against the single objective GA baselines. The obtained results (Cf. RQ1 summary) indicate that $CostAwareCR$ achieves the best balance between the predictive performance (measured in terms of AUC) and prioritization performance (measured in terms of P_{opt} and ACC). This finding indicates that early code review prediction and prioritization are conflicting tasks, and future research should focus on multi-objective approaches instead of single-objective approaches.

In Section 6.2, we showed that CostAwareCR achieved the best P_{opt} and ACC scores improving the state-of-art-approach (i.e., *PredCR*) performance with 35% and 80% respec-

tively (cf. RQ2 summary). We believe that there is still room for improvement in our approach since the state-of-the-art approach reported AUC scores ranging from 0.82 to 0.84 compared to 0.75 for $CostAwareCR$. That is, $CostAwareCR$attempted to balance the performance and cost-effectiveness for the early code review outcome prediction. We believe that researchers can use our work as a starting to conduct more investigations and further improvements. In particular, we showed that search-based approaches can be useful to bridge high-performance prediction capabilities with cost-effectiveness and future research should improve the prediction performance based on our approach by investigating different models than LR, inspecting other approaches to extend the original features in lieu of RuleFit and check the performance of $CostAwareCR$under different data and hyper-parameters. Furthermore, we used the code churn as a proxy to the code review effort since we showed that bigger changes require more review effort (cf. Section 7.1). However, we believe that other than the merged likelihood and the effort, other prioritization factors should be investigated such as the priority, the quality and the defective proneness of the code change. Therefore, we encourage researchers to investigate these factors more deeply.

In Section 6.3, we investigated the performance of CostAwareCR when different portions of code were investigated. The obtained results suggest that $CostAwareCR$outperforms the studied baselines at different inspected code portions. Additionally, we observe that when inspecting 40% of the total code, $CostAwareCR$identifies more than 90% of the merged changes. This finding is analogous to the finding of Ostrand et al. (2005) in software defects prediction in which they observe that more than 80% of the defects are located in only 20% of the code. Therefore, we recommend researchers incorporate the notation of effort when building code review request prioritization approaches since this may help practitioners optimize their efforts when doing code review activities.

In Section 6.4, we identified the most prominent features for code review prediction and prioritisation using $CostAwareCR$. Similar to previous studies (Islam et al. 2022; Thongtanunam et al. 2017; Jiang et al. 2013), we found that change complexity features (e.g., the number of added/deleted lines, number of modified files etc.,) and reviewers' experience (e.g., number of reviews and average reviewers experience). We encourage researchers to investigate $CostAwareCR$with more features.

7.4.3 Implications for Tool Builders

Our tool can be used to build automated bots that help developers predict and prioritise code review requests. In particular, this bot can be part of the *Gerrit* ecosystem and can be used to remind developers regarding code review requests that are hanging for a long time. For example, the bot can inspect newly submitted code reviews and help prioritize them. In practice, project maintainers can have up to hundreds of code reviews to inspect (Gousios et al. 2016) which is a time-consuming and tedious process. An example of such an issue can be observed in many large projects. In particular, in the subsystem core in LibreOffice, we observe that on 28th May 2016, the LibreOffice/core had 140 open changes, of which 105 were submitted between 28 and 30 May. It is cumbersome for the project maintainer and developers to inspect all these submitted changes in a timely manner. Therefore, $CostAwareCR$can be used to prioritize these changes and provide the project maintainer with a sorted list of these changes according to their relevance and their expected review effort. Finally, our tool can be configured to be personalised based on the preference of the developers in terms of the allocated effort. Therefore, tool builders exploit this aspect to build tools customized by the developers or the project owner instead of tools with static thresholds.

7.5 *CostAwareCR* Limitations

Although we showed that $CostAwareCR$achieves prominent performance in terms of predictive performance and prioritization, there are still some limitations that can be addressed to improve $CostAwareCR$. In particular, similar to previous works (Islam et al. 2022; Fan et al. 2018; Gousios et al. 2014), $CostAwareCR$does not take into account the importance and severity of the proposed code change and considers all code changes to have the same importance. To alleviate this issue, features representing the purpose of the change such as `is_bug_fixing`, `is_feature` and `is_documentation` (Cf. Table 1) have been added. Hence, incorporating the severity of a code change in the fitness function can help our approach to prioritize code reviews with high severity while still optimizing the effort of review. In practice, it is not trivial to estimate the severity and the importance of a given code change. Therefore, this can be estimated by project maintainers who extend/personalize the fitness function to their needs and their expertise.

Another limitation related to $CostAwareCR$is its fit time. We found that $CostAwareCR$ takes an average of 5h as the fit time when using a workstation with 32GB of RAM and an i9-12700H intel processor. Therefore, the performance of $CostAwareCR$in terms of time efficiency can be improved either by using more computational power or by using surrogate models to estimate the values of fitness function (Zhou et al. 2021).

Finally, it is worth mentioning that although our approach is built based on the features of (Islam et al. 2022), we think that this set of features can be extended with more sophisticated features that help capture socio-technical code review aspects (Huang et al. 2022) which can be useful to predict and prioritise code review requests.

8 Threats to Validity

This Section presents the threats to validity related to our empirical study.

Threats to Internal Validity can be related to the factors that we missed that threaten our conclusions and the obtained results like implementation errors. To mitigate this threat, we first used common libraries and frameworks to implement our approach. In particular, $CostAwareCR$is implemented based on Pymoo (Blank and Deb 2020) and the other ML baselines are implemented using SciKit-Learn (Pedregosa et al. 2011). In addition, we ran our approach and the ML baselines several times to test their correctness and robustness.

Threats to External Validity are related to the generalizability of our approach. To mitigate these threats, we first adopted the dataset curated by (Islam et al. 2022) since this dataset has been extensively studied and was collected by following the recommendations of (Yang et al. 2016). Nevertheless, we believe that future studies should the performance of our approach by using different datasets and features. In particular, we cannot claim the generalizability of our approach to proprietary projects. Thus, future works should investigate this direction.

Threats to Construct Validity are linked to the design of our empirical study. To mitigate these threats, we first used three widely studied performance metrics to evaluate $CostAwareCR$in terms of prediction performance and cost-effectiveness. In particular, we used AUC to evaluate the performance prediction since it was previously used in similar contexts (Islam et al. 2022; Fan et al. 2018; Wang et al. 2019). To evaluate the cost-effectiveness

of $CostAwareCR$, we used P_{opt} and ACC scores which have been widely adopted in previous studies (Mende and Koschke 2010; Chen et al. 2018; Shukla et al. 2018). Moreover, we followed the online validation process since our data is chronologically ordered and to prevent data leaks (Islam et al. 2022; Saidan et al. 2020; Saidani et al. 2022). Furthermore, we summarize the results of each obtained Pareto-front by reporting the median and best metrics for each run following the previous works of (Chen et al. 2018) and Arcuri and Briand (Arcuri and Briand 2011). Our results are based on 310 runs per baseline per project and the obtained scores are then averaged and presented. Finally, we adopted the SK-ESD test to perform statistical multiple comparisons. Nevertheless, we believe that our study should be investigated under multiple other performance metrics and using other validation processes.

9 Conclusion

In this paper, we aimed to incorporate the code review effort dimension when prioritizing code review requests while maintaining acceptable early code review prediction performance. Hence, we proposed $CostAwareCR$, an effort-aware multi-objective approach that can be used to early predict whether a given code review is going to be merged/abandoned and to prioritize code review requests based on their likelihood to be merged and their code review efforts. $CostAwareCR$is based on a multi-objective search-based approach that optimizes two objective functions namely, the prediction performance and the cost-effectiveness to learn the weights of an LR model. We conduct an empirical study on 146,612 across 3 large and long-lived projects to evaluate the performance of $CostAwareCR$. The obtained results indicate that the $CostAwareCR$achieves acceptable prediction performance since we found that it achieves AUC scores above 0.75. Additionally, $CostAwareCR$achieves the best P_{opt} and ACC scores outperforming the studied baselines with 6% - 38% and 6% - 132% respectively. Future work can focus on improving the prediction performance of our approach by investigating other complex ML models (e.g., LGBM) and other MOEAs. Furthermore, our approach can be evaluated using a longitudinal and user study to further investigate its effectiveness in different contexts. Finally, while we used code churn as a proxy for code review effort following (Kamei et al. 2012), we believe that other more specific measures related to the effort of code review could be adopted as part of our future work.

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

References

AlOmar EA, Chouchen M, Mkaouer MW, Ouni A (2022) Code review practices for refactoring changes: An empirical study on openstack. In: Proceedings of the 19th international conference on mining software repositories, pp 689–701

Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd international conference on software engineering, pp 1–10

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Syst Softw 83(1):2–17

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: 2013 35th International conference on software engineering (ICSE). IEEE, pp 712–721

Baysal O, Kononenko O, Holmes R, Godfrey MW (2016) Investigating technical and non-technical factors influencing modern code review. Empir Softw Eng 21(3):932–959
Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix?. In: Proceedings of the 11th working conference on mining software repositories, pp 202–211
Blank J, Deb K (2020) Pymoo: Multi–objective optimization in python. IEEE Access 8:89,497–89,509
Bosu A, Carver JC (2014) Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In: Int. symp. on empirical software eng. and measurement, pp. 1–10
Canfora G, De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S (2013) Multi-objective cross-project defect prediction. In: 2013 IEEE Sixth international conference on software testing, verification and validation. IEEE, pp 252–261
Chen X, Zhao Y, Wang Q, Yuan Z (2018) Multi: multi-objective effort-aware just-in-time software defect prediction. Inf Softw Technol 93:1–13
Chen D, Fu W, Krishna R, Menzies T (2018) Applications of psychological science for actionable analytics. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 456–467
Chen L, Rigby PC, Nagappan N (2022) Understanding why we cannot model how long a code review will take: an industrial case study. In: Proceedings of the 30th ACM Joint European software engineering conference and symposium on the foundations of software engineering, pp 1314–1319
Chouchen M, Ouni A, Olongo J, Mkaouer MW (2023) Learning to predict code review completion time in modern code review. Empir Softw Eng 28(4):82
Chouchen M, Ouni A, Kula RG, Wang D, Thongtanunam P, Mkaouer MW, Matsumoto K (2021) Anti-patterns in modern code review: symptoms and prevalence. In: 2021 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 531–535
Cohen J (2013) Statistical power analysis for the behavioral sciences. Academic press
CostAwareCR Replication Package (2023) https://github.com/stilab-ets/CostAwareCR
DA Van Veldhuizen, GB Lamont et al (1998) Evolutionary computation and convergence to a pareto front. In: Late breaking papers at the genetic programming 1998 conference. Citeseer, pp 221–228
De Winter JC, Gosling SD, Potter J (2016) Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: a tutorial using simulations and empirical data. Psychol Methods 21(3):273
Deb K, Jai H (2013) An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. IEEE Trans Evol Comput 18(4):577–601
Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: nsga-ii. IEEE Trans Evol Comput 6(2):182–197
Deb K, Agrawal S (1999) A niched-penalty approach for constraint handling in genetic algorithms. In: Artificial neural nets and genetic algorithms. Springer, pp 235–243
Deb K, Sindhya K, Okabe T (2007) Self-adaptive simulated binary crossover for real-parameter optimization. In: Proceedings of the 9th annual conference on genetic and evolutionary computation, pp 1187–1194
Dietterich TG (2000) Ensemble methods in machine learning. In: Multiple classifier systems: first international workshop, MCS 2000 Cagliari, Italy, June 21–23, 2000 Proceedings 1. Springer, pp 1–15
Egelman CD, Murphy–Hill E, Kammer E, Hodges MM, Green C, Jaspan C, Lin J (2020) Predicting developers' negative feelings about code review. In: 2020 IEEE/ACM 42nd International conference on software engineering (ICSE). IEEE, pp 174–185
Fagan ME (1999) Design and code inspections to reduce errors in program development. IBM Syst J 38(2.3):258–287
Fan Y, Xia X, Lo D, Li S (2018) Early prediction of merged code changes to prioritize reviewing tasks. Empir Softw Eng 23(6):3346–3393
Friedman JH, Popescu BE (2008) Predictive learning via rule ensembles. Ann Appl Stati 916–954
Gousios G, Pinzger M, Deursen AV (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp 345–355
Gousios G, Storey MA, Bacchelli A (2016) Work practices and challenges in pull-based development: the contributor's perspective. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, pp 285–296
Gousios G, Zaidman A, Storey M–A, Van Deursen A (2015) 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, pp 358–368
Greiler M, Bird C, Storey M–A, MacLeod L, Czerwonka J (2016) Code reviewing in the trenches: understanding challenges, best practices and tool needs

Guo Y, Shepperd M, Li N (2018) Bridging effort-aware prediction and strong classification: a just-in-time software defect prediction study. In: Proceedings of the 40th international conference on software engineering: companion proceeedings, pp 325–326
Harman M, Jones BF (2001) Search-based software engineering. Inf softw Technol 43(14):833–839
Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Computing Surveys (CSUR) 45(1):1–61
Harman M, Clark J (2004) Metrics are fitness functions too. In: 10th International symposium on software metrics, 2004. proceedings. Ieee, pp 58–69
Hossin M, Sulaiman MN (2015) A review on evaluation metrics for data classification evaluations. Int J Data Min Knowledge Manag Process 5(2):1
Huang Y, Liang X, Chen Z, Jia N, Luo X, Chen X, Zheng Z, Zhou X (2022) Reviewing rounds prediction for code patches. Empir Softw Eng 27:1–40
Islam K, Ahmed T, Shahriyar R, Iqbal A, Uddin G (2022) Early prediction for merged vs abandoned code changes in modern code reviews. Inf Softw Technol 142:106756
Jeong G, Kim S, Zimmermann T,. Yi (2009) Improving code review by predicting reviewers and acceptance of patches. Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006), pp 1–18
Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast? case study on the linux kernel. In: 2013 10th Working conference on mining software repositories (MSR). IEEE, pp. 101–110
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773
Khatoonabadi S, Costa DE, Abdalkareem R, Shihab E (2021) On wasted contributions: understanding the dynamics of contributor-abandoned pull requests: a mixed-methods study of 10 large open-source projects. ACM Trans Softw Eng Methodol
LaValle SM, Branicky MS, Lindemann SR (2004) On the relationship between classical grid search and probabilistic roadmaps. Int J Robot Res 23(7–8):673–692
Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496
MacLeod L, Greiler M, Storey M-A, Bird C, Czerwonka J (2017) Code reviewing in the trenches: challenges and best practices. IEEE Softw 35(4):34–42
Mende T, Koschke R (2010) Effort-aware defect prediction models. In: 2010 14th European conference on software maintenance and reengineering. IEEE, pp 107–116
Milanesio L (2013) Learning Gerrit Code Review. Packt Publishing, vol 144
Natekin A, Knoll A (2013) Gradient boosting machines, a tutorial. Front Neurorobotics 7:21
Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355
Panichella A (2019) An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization. In: Proceedings of the genetic and evolutionary computation conference, pp 595–603
Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
Pornprasit C, Tantithamthavorn C, Jiarpakdee J, Fu M, Thongtanunam P (2021) Pyexplainer: explaining the predictions of just-in-time defect models. In: 2021 36th IEEE/ACM International conference on automated software engineering (ASE). IEEE, pp 407–418
Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, pp 202–212
Riquelme N, Von Lücken C, Baran B (2015) Performance metrics in multi-objective optimization. In: 2015 Latin American computing conference (CLEI). IEEE, pp 1–11
Romano D, Pinzger M (2011) Using source code metrics to predict change-prone java interfaces. In: 2011 27th IEEE international conference on software maintenance (ICSM). IEEE, pp 303–312
Ruder S (2016) An overview of gradient descent optimization algorithms. arXiv:1609.04747
Saidani I, Ouni A, Chouchen M, Mkaouer MW (2020) Predicting continuous integration build failures using evolutionary search. Inf Softw Technol 128:106392
Saidani I, Ouni A, Mkaouer MW (2022) Improving the prediction of continuous integration build failures using deep learning. Autom Softw Eng 29(1):1–61
Seada H, Deb K (2015) A unified evolutionary optimization procedure for single, multiple, and many objectives. IEEE Trans Evol Comput 20(3):358–369
Segura C, Coello CAC, Miranda G, León C (2016) Using multi-objective evolutionary algorithms for single-objective constrained and unconstrained optimization. Ann Oper Res 240:217–250

Shukla S, Radhakrishnan T, Muthukumaran K, Neti LBM (2018) Multi-objective cross-version defect prediction. Soft Comput 22(6):1959–1980
Shull F, Seaman C (2008) Inspecting the history of inspections: an example of evidence-based technology diffusion. IEEE Softw 25(1):88–90
Soares DM, de Lima Júnior ML, Murta L, Plastino A (2015) Acceptance factors of pull requests in open-source projects. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp 1541–1546
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2018) The impact of automated parameter optimization for defect prediction models
Thongtanunam P, McIntosh S, Hassan AE, Iida H (2017) Review participation in modern code review. Empir Softw Eng 22(2):768–817
Wang S, Ali S, Yue T, Li Y, Liaaen M (2016) A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In: Proceedings of the 38th international conference on software engineering, pp 631–642
Wang S, Bansal C, Nagappan N, Philip AA (2019) 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, pp 46–55
Weißgerber P, Neu D, Diehl S (2008) Small patches get in!. In: Proceedings of the 2008 international working conference on mining software repositories, pp 67–76
Yang X, Kula RG, Yoshida N, Iida H (2016) Mining the modern code review repositories: a dataset of people, process and product. In: Proceedings of the 13th international conference on mining software repositories, pp 460–463
Zhang X, Tian Y, Jin Y (2014) A knee point-driven evolutionary algorithm for many-objective optimization. IEEE Trans Evol Comput 19(6):761–776
Zhao G, da Costa DA, Zou Y (2019) Improving the pull requests review process using learning-to-rank algorithms. Empir Softw Eng 24(4):2140–2170
Zhou Q, Wu J, Xue T, Jin P (2021) A two-stage adaptive multi-fidelity surrogate model-assisted multi-objective genetic algorithm for computationally expensive problems. Engi Comput 37:623–639
Zitzler E, Thiele L (1998) Multiobjective optimization using evolutionary algorithms–a comparative case study. In: International conference on parallel problem solving from nature. Springer, pp 292–301

Authors and Affiliations

Moataz Chouchen[1] · **Ali Ouni**[1]

Ali Ouni
ali.ouni@etsmtl.ca

[1] ETS Montreal, University of Quebec, Montreal, QC, Canada