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

Improving the Pull Requests Review Process Using Learning-to-rank Algorithms

Guoliang Zhao · Daniel Alencar da Costa · Ying Zou

Received: date / Accepted: date

Abstract Collaborative software development platforms (such as GitHub and GitLab) have become increasingly popular as they have attracted thousands of external contributors to contribute to open source projects. The external contributors may submit their contributions via pull requests, which must be reviewed before being integrated into the central repository. During the review process, reviewers provide feedback to contributors, conduct tests and request further modifications before finally accepting or rejecting the contributions. The role of reviewers is key to maintain the effective review process of the project. However, the number of decisions that reviewers can make is far superseded by the increasing number of pull requests submissions. To help reviewers to perform more decisions on pull requests within their limited working time, we propose a learning-to-rank (LtR) approach to recommend pull requests that can be quickly reviewed by reviewers. Different from a binary model for predicting the decisions of pull requests, our ranking approach complements the existing list of pull requests based on their likelihood of being quickly merged or rejected. We use 18 metrics to build LtR models and we use six different LtR algorithms, such as ListNet, RankNet, MART and random forest. We conduct empirical studies on 74 Java projects to compare the performances of the six LtR algorithms. We compare the best performing algorithm against two baselines obtained from previous research regarding pull requests prioritization: the first-in-and-first-out (FIFO) baseline and the small-size-first baseline. We then conduct a survey with GitHub reviewers to understand the perception of code reviewers regarding the usefulness of our approach. We observe that: (1) The random forest LtR algorithm outperforms other five well adapted LtR algorithms to rank quickly merged pull requests. (2) The random forest LtR algorithm performs better than both the FIFO and the small-size-first baselines, which means our

Guoliang Zhao
School of Computer, Queen's University, Kingston, Ontario, Canada
E-mail: 17gz2@queensu.ca

Daniel Alencar da Costa and Ying Zou
Department of Electrical and Computer Engineering, Queen's University,
Kingston, Ontario, Canada
E-mail: {daniel.alencar, ying.zou}@queensu.ca

LtR approach can help reviewers make more decisions and improve their productivity. (3) The contributor's social connections and contributor's experience are the most influential metrics to rank pull requests that can be quickly merged. (4) The GitHub reviewers that participated in our survey acknowledge that our approach complements existing prioritization baselines to help them to prioritize and to review more pull requests.

Keywords Pull requests, learning-to-rank, merged, rejected

1 Introduction

The pull-based development model has become a standard for distributed software development and has been adopted in several collaborative software development platforms, such as GitHub, GitLab and Bitbucket [1]. Compared with other classic distributed development approaches (e.g., sending patches to development mailing lists [2]), the pull-based development model provides built-in mechanisms for tracking and integrating external contributions [3]. For example, with pull-based development model, some pull requests may be integrated with just one click, without manual intervention.

Under the pull-based development model, contributors can fetch their local copies of any public repository by *forking* and *cloning* them. Next, contributors can modify their clones to fix a bug or implement a new feature as they please. Ultimately, contributors may request to have their code changes merged into the central repository through pull requests (PRs) [4]. Once contributors submit PRs, they become available to reviewers (i.e., PRs undergo the opened state). If a PR is approved by reviewers and passes the integration tests, its respective code is merged into the central repository (i.e., the status of the PR changes to *merged*). Such collaborative software development platforms offer social media functionalities to allow contributors to easily interact with each other, such as following projects and communicating with reviewers.

As Gousios *et al.* [5] mentioned, the role of reviewers is key to maintain the quality of projects. Therefore, reviewers should carefully review PRs and decide whether a PR is worth integrating. Code reviewers should also communicate the modifications that are required before integrating a PR to external contributors. Nevertheless, reviewers typically have to manage large amounts of open PRs simultaneously. Through our preliminary study of 74 projects hosted on GitHub, we observe that the workload (measured by the number of open PRs) of reviewers tends to rise while the decisions made by reviewers tend to remain roughly constant. As shown in Figure 1, the average number of open PRs is below five on January 2014. Overtime, the average number of open PRs keeps on increasing and reaches 30 on May 2016. However, the average number of reviewers and the average number of decisions made every day by reviewers remain roughly the same (e.g., an average of one decision per day). In fact, Steinmacher *et al.* [6] observed that 50% of PRs are submitted by casual-contributors, which considerably inflate the number of PR submissions that must be processed.

Gousios *et al.* [5] mentioned that prioritizing multiple PRs is important for reviewers when they face an increasing workload. The authors propose a PR prioritization tool, called PRioritizer, to automatically recommend the top PRs that reviewers should first deal with [7]. The tool can help reviewers select the PR that

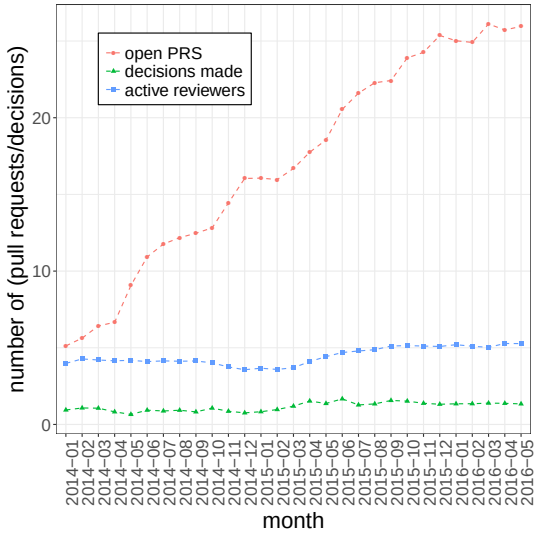

Fig. 1 The average number of open PRs and the average number of decisions made every day among 74 Java projects on GitHub.

needs their immediate attention. To increase the decisions made by reviewers, the time taken by a reviewer to make decisions on PRs (either rejected or merged) should be considered when the tool recommends PRs to reviewers. In our preliminary study based on 74 projects, the median time taken by a reviewer to make decisions on PRs is 16.25 hours as shown in Figure 2. There are PRs that can be reviewed quickly (e.g., the minimum time taken to review a PR is 8 minutes), while some other PRs take a long time to review (e.g., the maximum time is more than 400 hours). More specifically, recommending reviewers PRs that can be quickly reviewed allows them to handle more contributions and give expedite feedback on PRs, which could be very useful when reviewers have only a limited time (e.g., half an hour or few minutes) to review PRs. The fast turn-around could ultimately improve the productivity of developers and reduce the waiting queue of open PRs so that contributors do not need to wait for a long time to receive feedback. Therefore, it is desirable to provide an approach that can recommend PRs that can be merged or rejected in a timely fashion. Instead of replacing the working practices, our approach aims to complement the existing practices of reviewers if they decide to take a number of quick decisions in a limited period of time. We adopt learning-to-rank (LtR) algorithms [8] [9] to rank PRs that are likely to be quickly merged or rejected. In particular, we address the following research questions.

RQ1. What learning-to-rank algorithm is the most suitable for ranking PRs?

We study six LtR algorithms including pairwise LtR algorithms and listwise LtR algorithms, such as RankBoost, MART, RankNet and the random forest. We find that the random forest based algorithm outperforms the other algorithms with respect to rank PRs that can receive quick decisions.

RQ2. Is our approach effective to rank PRs that can receive decisions quickly?

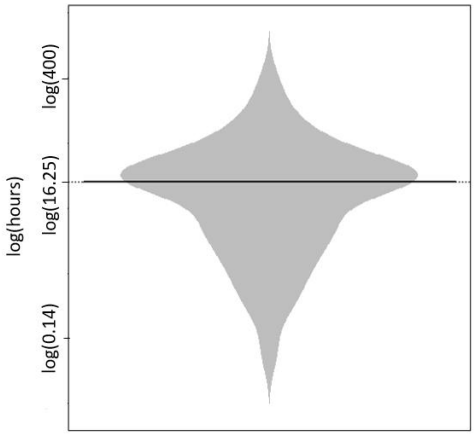

Fig. 2 The time taken to make decisions on PRs among 74 Java projects on GitHub.

Gousios *et al.* [5] observed that there are two well-adapted prioritizing criteria among reviewers: first-in-and-first-out (FIFO) criterion and small-size-first criterion that recommends the PRs having the minimum source code churn. We compare the performance of the random forest algorithm with the FIFO baseline and the small-size-first baseline. Our LtR approach outperforms both the FIFO baseline and the small-size-first baseline. Our results suggest that our LtR approach can help reviewers to make decisions regarding PRs more efficiently.

RQ3. What are the most significant metrics that affect the ranking?

To test the effect of each metric, we exclude a given metric and re-run the learning-to-rank approach without the metric. Then, we obtain the decreasement in the accuracy of LtR algorithm without that metric. The larger the decreasement, the more important a metric is. We find that metrics that represent the reputation of contributors (e.g., the previous PRs merged percentage) and the social connection of contributors (e.g., the number of followers and the previous interaction) affect the ranking the most.

RQ4. How do GitHub reviewers perceive the usefulness of our approach?

We conduct a survey with GitHub reviewers to evaluate the usefulness of our approach. We also aim to investigate whether our approach can complement the existing criteria used by reviewers to prioritize PRs (e.g., FIFO and small-size-first criteria). 86% of reviewers who participate in our survey believe that our approach can help them to obtain a higher throughput of reviewed PRs within their limited working time.

Organization of the paper: The related work is presented in Section 2. We describe our experiment setup and our results in Sections 3 and 4, respectively. In Section 5, we discuss the soundness of our chosen threshold, the experience levels of reviewers and the importance of the PRs recommended by our approach. The

threats to validity are discussed in Section 6. Finally, we conclude the paper in Section 7.

2 Related work

In recent years, researchers have conducted various studies regarding the factors that impact on the final decisions on PRs [5] [10] [11]. Researchers have also striven to help reviewers by, for example, recommending the right reviewer to a given PR [7] [12] [13] [14] [15]. We use this section to situate our work with respect to the related research and highlight our contributions.

2.1 Understanding the final decisions made upon PRs

Understanding the decisions made upon PRs can guide contributors to submit high quality PRs. Gousios *et al.* [10] conducted an empirical study on 291 GitHub projects to obtain the important metrics that affect both the time to review and the decision to merge PRs. They found that the final decision of a PR was significantly affected by whether it touches recently modified code of the system. They also observed that the contributor's previous successful rate (the percentage of previous PRs that a contributor has successfully merged) played an important role in the time to review PRs. Different from Gousios *et al.*, we use LtR algorithms to rank a list of PRs based on how quickly a decision to merge or reject a PR can be made. We consider additional dimensions of metrics in models, such as social connection and complexity metrics.

In the recent research, Steinmacher *et al.* [6] analyzed the PRs of casual-contributors and conducted surveys with casual-contributors and reviewers to obtain the reason for rejected PRs from casual-contributors. They found that the mismatch between developer's and reviewer's opinions was among the main reason for the rejection of PRs. Our goal is to recommend PRs with potential to be quickly processed. Our approach can help reviewers to prioritize PRs and allow them to perform more decisions on PRs within their limited working time.

In addition, Gousios *et al.* [5] surveyed various metrics that reviewers examine when evaluating a PR. The authors observed that reviewers typically focus on checking whether the PR matches the current style of a project and the source code quality of the PR. Because of the observations made by Gousios *et al.* [5], we include metrics to measure the source code quality of the PRs (e.g., cyclomatic complexity of the source code and number of comments in the source code). Tsay *et al.* [11] analyzed the social connection of contributors in the process of evaluating contributions in GitHub. The authors found that PRs from contributors with a strong social connection to the project are more likely to be merged. Considering the observations made by Tsay *et al.* [11], we use social connection metrics (e.g., social distance between contributors and reviewers, and the number of prior interactions of contributors) in our experiment.

Differently from the aforementioned research, our study focuses on recommending PRs that can quickly receive decisions instead of predicting the final decision on a PR. With a recommended ranked list, reviewers have the option to use part of their time to quickly make several decisions before digging into a PR that will

likely take more time to review. Thus, our approach is intended to complement the existing criteria to prioritize PRs and help reviewers to make more decisions whenever they are interested.

2.2 Helping reviewers evaluate contributions

Assigning incoming PRs to highly relevant reviewers can reduce the time needed to review PRs. Thongtanunam *et al.* [12] [13] took full advantage of the previously reviewed file paths to assign appropriate reviewers of new PRs automatically. Balachandran *et al.* [14] introduced a reviewer recommendation tool by integrating several automatic static analyses (i.e., automatically checking for coding standard violations and common defect patterns) into the code reviewing process. Besides focusing on the information about PRs themselves, Yu *et al.* [15] [16] implemented a reviewer recommendation approach by leveraging both the textual semantic information of PRs and the social network of contributors.

Van *et al.* [7] proposed a PR prioritization tool, called PRioritizer, to automatically recommend the top PRs that need immediate attention of reviewers. Li *et al.* [17] leveraged the textual similarity between incoming PRs and other existing PRs to warn reviewers about duplicate PRs and prevent reviewers from wasting time on them.

Despite the great advances of prior research, the time required to make a decision regarding a PR has not yet been explored when recommending a ranked list of PRs to reviewers. This is important to quickly give feedback to contributors and increase the number of PRs that can be processed by reviewers. In addition, our work builds on top of prior research and explore metrics but not yet being explored in the recommendation approaches for PRs, such as the complexity metrics and the social connection metrics.

3 Experiment Setup

In this section, we present the process for collecting data and extracting metrics for building LtR models. We also explain the analysis of correlated metrics. The overview of our approach is shown in Figure 3. We first select a set of experiment projects and collect the data of the selected projects (e.g., PRs, issues, commits). Next, we build LtR models on the 18 extracted metrics to recommend PRs that are likely to receive decisions quickly. We evaluate the performance of the LtR models and calculate the effect of each metric on ranking PRs.

3.1 Collecting data

As shown in Figure 3, we collect the data of our experiment from the GHTorrent database [18] and GitHub using the GitHub API.

Project Selection. To avoid working on personal, inactive or toy projects [19], we exclude projects containing less than 500 PRs. Our dataset contains data from November 2010 until May 2016. Given that we plan to analyze code quality metrics, we restrict our analyses to only one programming language (i.e., Java). This

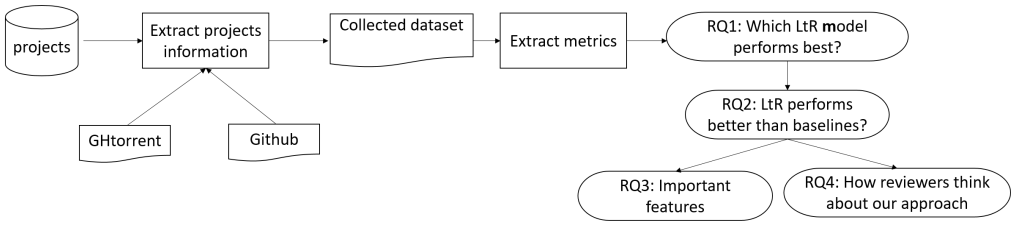

Fig. 3 The overall view of our approach.

choice allows us to implement scripts that automatically compute, for example, the addition of code complexity or code comments, because we do not need to adapt to many different programming languages. Nevertheless, our approach can be adapted to other programming languages, provided that the metrics are properly extracted. We start with a set of 303 Java projects that contain more than 500 PRs. Then, we apply the following criteria to exclude projects from the initial selection:

- We exclude projects have been deleted from GitHub. For deleted projects, GHTorrent database still records its information (e.g., created time, and owner) but we cannot find the PRs of deleted projects, since all PRs have been cleaned.
- We exclude projects that are forked from other existing GitHub projects. As Gousios *et al.* [10] mentioned, more than half of the GitHub repositories are forks of other repositories. To ensure that we study projects that receive external contributions, we only include original projects.
- The PRs of a project must be correctly labeled (*merged* or *closed*) for a recommender system to work correctly. However, as Steinmacher *et al.* [6] observed, many projects do not adopt pull based development. Instead, these projects prefer to merge contributions via git Command Line Interface (CLI). For example, through our manual analysis, we find PRs that are *closed* with comment *"Cherry picked. Thanks."* meaning that their commits were *merged*. As a result, the vast majority of PRs in such projects would be labeled as *closed* when their commits were actual *merged* into the main repository. We analyze the 94 remaining projects after excluding the deleted and forked projects. In the projects that do not adopt the pull based development, we find that the ratio of PRs labeled as *merged* over the total number of PRs is usually lower than 40%. In projects that use the pull based development, the ratio of merged PRs is around 73% [10]. To avoid studying noisy PRs, we exclude projects for which the merge ratio is less than 40% [10]. We discuss the impact of using different merge ratio thresholds in Section 5.1.

After filtering the projects, the number of our analyzed projects is reduced to 74.

Data Collection. After the project selection step, we collect the PRs, issue reports, comments and commits of the subject projects. We download the information by querying the GHTorrent (through Google BigQuery[1]) and searching GitHub repositories using the GitHub API. The GHTorrent database includes the meta information of PRs and commits, such as the id, the related project id and

[1] http://ghtorrent.org/relational.html

related PRs ids. However, information (e.g., textual information of PRs and the code difference information of commits) is not available in the GHTorrent database to compute our metrics (more details in Section 3.2). To collect such information, we use the GitHub API to search commits and PRs in GitHub repositories.

In short, our final dataset consists of 74 Java projects containing 100,120 PRs with 74,060 merged PRs and 26,060 rejected PRs.

3.2 Metrics Calculation

Given that we need to predict PRs that can receive a quick decision, we only collect the metrics that are available before the final decision (i.e., merged or rejected) of a PR. We select the metrics based on prior studies that investigate PRs [5] [7] [10] [11]. We categorize the selected metrics into four categories: source code metrics, textual metrics, contributor's experience metrics and contributor's social connection metrics.

Source code metrics. Our source code metrics describe the quality and the size of the modified source code in a PR. In total, we collect nine source code metrics, listed as follows.

- *test_inclusion (whether a contributor modifies test files in the PR).* Previous research [20] finds that PRs containing test files are more likely to be merged and reviewers perceive the presence of testing code as a positive indicator [5].
- *test_churn (the source code churn in test files).* Besides checking whether contributors change test files, we record the source code churns of the changes in test files. The source code churns measure the number of lines of code that are modified. A high code churn may indicate a better quality PR because the contributor invests effort in the tests.
- *src_churn (the source code churns in other files that are not test files).* Prior research [21] reports that the size of a code patch plays an important role in both the acceptance of the patch and the acceptance time of the patch.
- *files_changed (the number of files touched in a PR).* We use the number of modified files as a measure of the scale of a PR. Large or small scale PRs might affect the time to merge or reject a PR.
- *num_commits (the number of commits included in a PR).* We use the number of commits as a measure of the scale of a PR. Large or small scale PRs in terms of the number of commits might affect the time to merge or reject a PR.
- *commits_files_changed (the number of total commits on files modified by a PR three months before submitting the PR).* Our goal is to check whether PRs modifying an active part of the system is more likely to be merged or rejected quickly.
- *bug_fix (whether a PR is an attempt to fix a bug).* Gousios *et al.* [5] finds that reviewers would consider reviewing the PRs that fix bugs before reviewing the PRs related to enhancements.
- *ccn_added, ccn_deleted (the cyclomatic complexity of newly added code and deleted code in a PR).* We use *ccn_added and ccn_deleted* to evaluate the source code quality of PRs because prior research [5] [10] finds that the source code quality of PRs could affect the decisions made on PRs.

– *comments_added (the number of comments that were introduced in the source code of a PR).* Well commented source code is easy to understand [22] and may accelerate the reviewing process.

Textual information metrics. Textual information metrics describe the general textual properties of a PR. We select the following three important metrics:

– *title_length (the number of words in the title of a PR).* A longer title may contain more useful information about a PR and may help reviewers easily understand a PR.
– *description_length (the number of words in the description of a PR).* A longer description may contain more meaningful information about a PR for reviewers to understand.
– *readability (the Coleman-Liau index [23] (CLI) of the title and description messages of a PR as well as its commit messages).* CLI has been adopted to measure the text readability in bug reports [24] and education material [25]. CLI represents the level of difficulty to comprehend a text, ranging from 1 (easy) to 12(hard). The equation for CLI is shown in Equation 1

$$CLI = 0.0588 * L - 0.296 * S - 15.8 \quad (1)$$

where L is the average number of characters per 100 words and S is the average number of sentences per 100 words.

Contributor's experience metrics. The contributor's experience metrics describe the experience of the contributors who submit PRs. We select two important metrics:

– *contributor_succ_rate*. The percentage of PRs from a contributor that have been merged before submitting the current PR.
– *is_reviewer*. Whether a contributor is also a reviewer in the same project. Code reviewers are much more experienced and more familiar with their projects and code reviewing process than the external contributors.

Contributor's social connection metrics. The contributor's social connection metrics represent the social connection that the contributor has on GitHub. We measure the social distance and the prior interactions of contributors.

– *social_distance* measures the social closeness between a contributor and a reviewer. If a contributor follows a reviewer, social_distance is set to 1, otherwise, it is 0.
– *prior_interaction* is used to count the number of events that a contributor has participated in the same project before submitting a PR [11]. Events include submissions of issue reports, PRs and comments.
– *followers* measures the number of followers that a contributor has. A high number of followers indicates a contributor's popularity and influence.

For each PR of the subject projects, we extract each aforementioned metric by performing a query in the collected datasets (e.g., counting the total number of PRs and the merged PRs of a contributor to calculate *contributor_cuss_rate*). Next, we store the metrics related to PRs into a table and train and test LtR models for each project.

3.3 Correlation and redundancy analysis

Highly correlated and redundant metrics can prevent us from measuring the effect of each metric. Therefore, we conduct correlation and redundancy analyses to remove highly correlated and redundant metrics.

We apply the Spearman rank correlation because it can handle non-normally distributed data [26]. We use the *cor()* function in R to calculate the correlation coefficient. If the correlation coefficient between two metrics is larger than or equal to 0.7, we consider the pair of metrics to be highly correlated and we select only one of the metrics. We intend to train and test LtR models on each project, which requires us to remove highly correlated metrics for each project. Instead obtaining the correlated pairs of metrics and select metrics for all projects manually, we run the Spearman rank correlation for each pair of metrics in all projects and record all possible highly correlated pairs. Then, the first and the second authors discussed which metric should be kept in the occurrence of every possible correlation pair. Through several discussions and after reaching consensus, we produced a *decision table* as shown in Appendix A. For example, when *src_churn* and *ccn_addad* are highly correlated, we choose to keep *ccn_added* because we consider that the complexity of code is more important than the code churns. When building the LtR models for our projects, our approach reads the table and automatically chooses the metrics based on our produced *decision table*.

After performing the correlation analysis, we conduct a redundancy analysis on the metrics using the *redun()* function of the R *rms* package. Redundant metrics can be explained by other metrics in the data and do not aggregate values for models. We find that there exist no redundant metrics in our data.

3.4 PRs labeling

Before training and testing the LtR models, the relevance between each PR and a query (e.g., a query refers to searching for the PRs that are most likely to be quickly merged, more details in Section 4.1) is described by (i) a label based on the decision made about the PR and (ii) the time taken to make the decision (as shown in Figure 4). Through our preliminary study, we find that the median time for reviewers to review PRs in each project ranges from 1.15 hours to 448.7 hours. We use the median time to review the PRs of a project as the threshold to split PRs into quickly reviewed PRs and slowly reviewed PRs. We use the median value because it is more robust to outliers [27]. Specifically, we use three relevance levels in querying PRs that are the most likely to be quickly merged:

- Relevance level 0 indicates PRs that are rejected.
- Relevance level 1 indicates PRs that take a long time to review and merge.
- Relevance level 2 indicates PRs that are quickly reviewed and merged.

Similarly,we use three relevance levels for identifying the PRs that are the most likely to be quickly rejected:

- Relevance level 0 indicates PRs that can be merged.
- Relevance level 1 designates PRs that are rejected but take a long time to be reviewed.
- Relevance level 2 specifies that a PR can be quickly rejected.

The number of PRs in each label is shown in Table 1.

Table 1 The number of PRs with different relevance levels. For querying the quickly merged PRs, (i) 0 indicates rejected PRs, (ii) 1 indicates slowly merged PRs and (iii) 2 indicates quickly merged PRs. For querying the quickly rejected PRs, (i) 0 indicates merged PRs, (ii) 1 indicates slowly rejected PRs and (iii) 2 indicates quickly rejected PRs.

	Querying the quickly merged PRs	Querying the quickly rejected PRs
Relevance level 0	26060	74060
Relevance level 1	35140	15322
Relevance level 2	38920	10738

4 Results

In this section, we describe the three research questions. We present the motivation, approach and results for each question.

4.1 RQ1. Which learning-to-rank algorithm is the most suitable for ranking PRs?

Motivation. To optimize the performance of our approach, it is crucial to select and deploy the most suitable LtR algorithm. We use the learning-to-rank framework, called RankLib[2], which consists of a set of LtR algorithms. Given the several options of LtR algorithms, it is important to evaluate the best LtR algorithm that suits our goal.

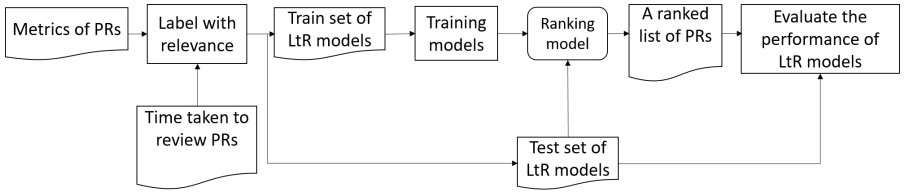

Fig. 4 An overview of our ranking approach.

Approach. We use LtR models to rank PRs that allow a reviewer to make speedy decisions. LtR models have been widely applied in the information retrieval field [28] [29]. LtR models rank a set of documents (e.g., PRs) based on their relevance to a given query (e.g., PRs that can be quickly merged to the project). LtR algorithms are supervised approaches and, as such, they have training and testing phases. The overall steps of our approach for ranking PRs are shown in Figure 4.

[2] https://sourceforge.net/p/lemur/wiki/RankLib/

4.1.1 LtR algorithm selection

There are three categories of LtR algorithms based on their training process: *(1) pointwise algorithms* compute the absolute relevance score for each PR; *(2) pairwise algorithms* transform the ranking problem into a classification problem to decide which PR is more likely to receive a quick decision in a given pair of PRs; and *(3) listwise algorithms* take ranked lists of PRs as instances in training phase and learn the ranking model. In our approach, we explore 6 well-known and widely adopted pairwise (RankNet [30] and RankBoost [31]) and listwise (MART [32], Coordinate Ascent [33], ListNet [34] and random forest [35]) LtR algorithms. We exclude pointwise algorithms, since pairwise algorithms and listwise algorithms have been empirically proved to consistently outperform pointwise LtR algorithms [36].

4.1.2 Training phase

Every LtR model is trained using a set of queries $Q = \{q_1, q_2, \ldots, q_n\}$ and their related set of documents (i.e., studied PRs) $D = \{d_1, d_2, \ldots, d_n\}$. More specifically, a query refers to searching for the PRs that are most likely to be quickly merged. Each document d is represented by one studied PR, $P = \{p_1, p_2, \ldots, p_n\}$. For each query q, the related PRs are labeled with their relevance to query q. During the training process, for each query, the LtR algorithm computes the relevance between each PR and the query using the metric vector V_s of the PR. In our study, we use one query (i.e., q_1) to identify PRs that are most likely to be quickly merged and another query (i.e., q_2) to identify PRs that are most likely to be quickly rejected. We define d, r, q_1, q_2 and V_s as follows:

- Document d is a PR.
- Relevance r is the likelihood of d being quickly merged or rejected (depending on the query).
- Query q_1 is a query to identify the PRs that are the most likely to be quickly merged.
- Query q_2 queries which PRs are the most likely to be quickly rejected.
- Metric vector V_s denotes a set of metrics used to build the LtR models as discussed in Section 3.2.

4.1.3 Testing phase

We measure the performance of the LtR model for ranking PRs that are most likely to be quickly merged and rejected separately, since ranking quickly merged and rejected PRs are trained using different sets of queries.

Given that software projects evolve over time [37], we need to consider the time-sensitive nature of our data. For example, we cannot test models using PRs that were closed before the PRs that we use in our training data (i.e., predicting the past). For this reason, we use a time-sensitive validation approach to evaluate the LtR models as shown in Figure 5. First, we sort our PRs based on their closing time. Then, we split our data into 5 folds $F = \{f_1, f_2, \ldots, f_5\}$. Each f_i is split into a training set t_i and a testing set τ_i. In the first iteration, we train a ranking model using the t_1 training set of PRs, while we use the τ_1 testing set as a query to test the model. In the second iteration, a new ranking model is trained using

the t_1, τ_1, t_2 sets (i.e., all PRs before τ_2) and tested using the τ_2 set as a query. This process continues until a ranking model is tested upon the τ_5 set. Finally, we compute the average performance of the ranking models in these five iterations.

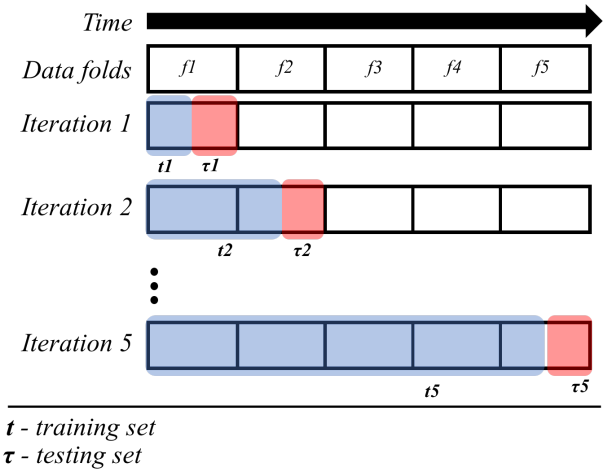

Fig. 5 An overview of our time-sensitive evaluation

Table 2 An example of the ranking result of three PRs of the MovingBlocks/Terasology project for query q_1 : *quickly merged PRs*

Rank	Pull request	Relevance
1	Develop - Block Manifestor cleanup + some...	quickly merged
2	PullRequest - Cleanup	rejected
3	AddedEclipse-specific content to .gitignore...	slowly merged

Table 3 The optimal rank of PRs in Table 2

Rank	Pull request	Relevance
1	Develop - Block Manifestor cleanup + some...	quickly merged
2	AddedEclipse-specific content to .gitignore...	slowly merged
3	PullRequest - Cleanup	rejected

To evaluate the performance of an LtR model, we use the normalized discounted cumulative gain at position k ($NDCG@k$), which is a well-adapted measure of ranking quality. $NDCG$ is a weighted sum of degree of relevance of the ranked PRs [38]. Search engine systems also use a cut-off top-k version of $NDCG$, referred to as $NDCG@k$ [38]. We opt for using $NDCG@k$ because the precision at position k ($P@k$) is not suitable for multiple labels [39]. In addition, Recall does not suit our case because hundreds of open PRs are likely to be merged quickly, while in our approach we rank only 20 PRs for reviewers to work on.

We first calculate the discounted cumulative gain at position k ($DCG@k$) [35]. DCG has an explicit position discount factor in its definition (i.e., the $\frac{1}{\log_2(1+j)}$ as shown in equation 2). PRs with a high relevance r but having a low ranking would negatively affect the DCG metric. $DCG@k$ is calculated as follows [35]:

$$DCG@k = \sum_{j=1}^{k} \frac{2^{r_j} - 1}{\log_2(1+j)} \tag{2}$$

where r_j is the relevance label of a PR in the j_{th} position in the ranking list.

For the example shown in Table 2, $DCG@1 = 3$, $DCG@2 = 3$, $DCG@3 = 3.5$. Then, the $DCG@k$ is normalized using the optimal $DCG@k$ value ($IDCG@k$) to get the $NDCG@k$ as shown in Equation 3.

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{3}$$

For the example shown in Table 2, the optimal ranking is shown in Table 3, and $IDCG@3 = \frac{2^2-1}{\log_2(1+1)} + \frac{2^1-1}{\log_2(1+2)} + \frac{2^0-1}{\log_2(1+3)} = 3.63$ and $NDCG@3 = \frac{DCG@3}{IDCG@3} = \frac{3.5}{3.63} = 0.96$.

We apply each LtR algorithm to rank the PRs of each project following the time-sensitive validation method as shown in Figure 5. Then, we compute the $NDCG@k$ metric for $k = 1, \ldots, 20$ for each project. After applying one LtR algorithm to all projects, we obtain a distribution of the $NDCG$ metric at each ranking position k (1,. . . ,20). For each ranking position k, we draw a beanplot[3] to show the performance of each algorithm at position k (e.g., the beanplot of $NDCG@1$ of RankNet algorithm as shown in Figure 6).

To compare the performance of different LtR models in the same ranking k position, we use the Cliff's delta to measure the magnitude of differences between performance distributions of LtR models at the same position. The larger the delta value, the larger the difference between two distributions. We use the *cliff.delta()* method of the *effsize* package in R to calculate the Cliff's delta.

Results. **The random forest model outperforms the other LtR models in ranking both quickly merged PRs and quickly rejected PRs.** As shown in Figure 6, for ranking quickly merged PRs, the median $NDCG@k$ values of the random forest model is almost 0.8, while the median values of the other models are all around 0.6. In addition, the magnitude of the differences in performance between the random forest model and the other models are small and medium in all the k positions as shown in Table 4.

Table 4 The median Cliff's Delta estimate of all k positions and the Cliff's Delta magnitude

Algorithm	Cliff's Delta Estimate	Cliff's Delta Magnitude
RankNet	0.334	medium
RankBoost	0.198	small
Coordinate Ascent	0.341	medium
MART	0.451	medium
ListNext	0.350	medium

[3] https://cran.r-project.org/web/packages/beanplot/vignettes/beanplot.pdf

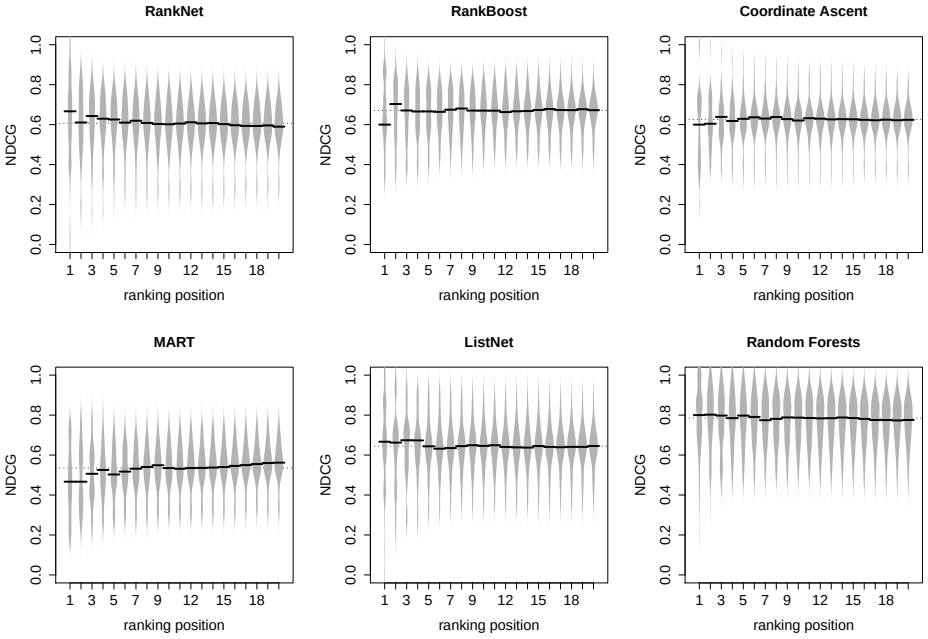

Fig. 6 The performance of the LtR models to rank PRs that can be quickly merged

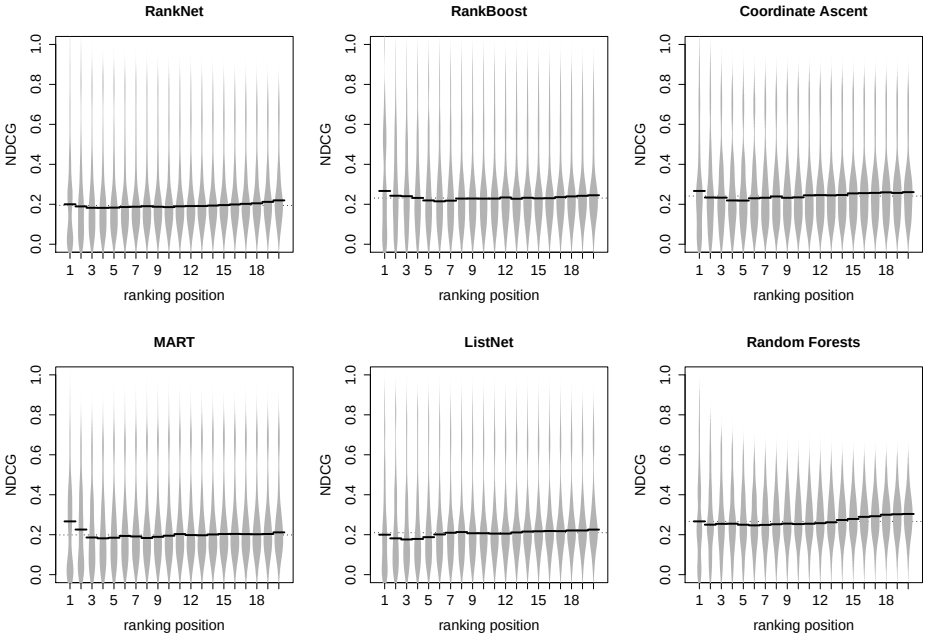

Fig. 7 The performance of the LtR models to rank PRs that can be quickly rejected

Regarding ranking quickly rejected PRs, the performances of the six models are not promising as shown in Figure 7. The most probable reason is that the ranking models are not well trained, since the quickly rejected PRs only account for 10% of the total set of PRs as shown in Table 1. Another possible reason is that there may still exist some noise in the rejected PRs in our dataset although we have excluded projects with a merge ratio lower than 0.4. Nevertheless, the random forest model performs better than the other 5 models when ranking quickly rejected PRs.

Summary of RQ1: The random forest model performs the best when ranking PRs based on their likelihood of being quickly merged.

4.2 RQ2. Is our approach effective to rank PRs that can receive decisions quickly?

Motivation. After selecting the best performing LtR model in our approach, we need to verify the effectiveness of our proposed LtR model by comparing it with the existing prioritizing criteria that are studied by Gousios *et al.* [5].

Approach. There is no universal conclusion on how reviewers prioritize PRs in practice. Nevertheless, Gousios *et al.* [5] observed that there are two well-adapted prioritizing criteria among reviewers through their large-scale qualitative study [5]. The two criteria are listed as follows:

- *First-in-first-out.* As mentioned in their research, many reviewers prefer a first-in-first-out prioritization approach to select the PRs [5], based on the age of the PRs. In this criterion, reviewers would first focus on the PRs that come earlier than others.
- *Small-size-first.* Besides the age of the PR, reviewers use the size of the patch (source code churn of the PRs) to quickly review small and easy-to-review contributions and process them first.

We build two baselines based on the aforementioned two criteria, i.e., the first-in-first-out (FIFO) baseline and the small-size-first baseline.

We test the two baselines using the same time-sensitive approach that we use to test the performance of the random forest model (see the approach section of RQ1). Similarly, we measure the performance of each baseline to rank PRs that are the most likely to be quickly merged and rejected separately. Next, we use Equation 3 to calculate the $NDCG@k$ metric. Similar to Section 4.1, after running the two baselines in each project, we use beanplots and Cliff's delta measures to show the performance of two baselines and compare them with the random forest model.

Results. **The random forest model outperforms both the FIFO and small-size-first baselines for ranking the PRs that can be either quickly merged or quickly rejected.** Figures 8 and 9 show the performance of the random forest model, the FIFO baseline and small-size-first baseline for ranking the PRs that can receive quick decisions. We observe a large Cliff's delta (median Cliff's Delta estimate = 0.60) between the random forest model and the FIFO baseline. Thought the Cliff's delta between the random forest model and the small-size-first baseline is small (median Cliff's Delta estimate = 0.18), the random forest model can help reviewers to merge more contributions in a shorter time as shown in the following ranking example.

To show the application of our ranking approach, we run the random forest LtR model and the small-size-first baseline on the same set of PRs of the libgdx/libgdx

project. The libgdx/libgdx project has been attracting external contributions since 2012. There are more than 70 open PRs waiting for reviewers to review every day in this project. The ranking list of the random forest LtR model and the ranking list of the small-size-first baseline are shown in Tables 5 and 6, respectively. Based on the top 10 PRs in both tables, we can observe that our LtR approach ranks PRs that can be merged in a shorter time at the top positions when compared with the small-size-first approach.

Table 5 Example ranking top 10 PRs results of the LtR random forest model

Position	Pull request	label	Observed review time (hours)
1	Move DebugDrawer	2	0.013
2	Fix linker flags...	2	0.628
3	error msg for...	2	10.416
4	Javadoc minor typos	2	0.518
5	Update CHANGES	2	1.02
6	Everyone misspells...	2	0.035
7	iOS: Force linked...	1	50.233
8	Implement material...	2	0.001
9	Move gwtVersion to...	2	0.041
10	Update fetch.xml	2	0.037

Table 6 Example ranking top 10 PRs results of the small-size-first baseline

Position	Pull request	label	Observed review time (hours)
1	Removed Unused variable...	2	4.988
2	Merge pull request #2...	0	0.011
3	update	0	0.007
4	Merge pull request #1...	0	0.028
5	Very Minor Update...	2	8.424
6	Gradle: Enforced OpenGL...	2	12.169
7	Update CHANGES	2	0.054
8	MathUtils: Fixed isEqual...	2	0.157
9	Stop IDE's thinking...	0	2.483
10	Everyone misspells...	2	2.483

We also observe that the small-size-first baseline performs as good as the other LtR models (e.g., RankNet, RankBoost and ListNet) when ranking the PRs that can be quickly merged or rejected. This observation suggests that the small-size-first baseline is, to a certain extent, optimized for ranking PRs that can receive quick decisions. Breaking large PRs into several smaller PRs (whenever possible) could increase the likelihood of quickly deciding on a PR.

Summary of RQ2: The LtR using the random forest model outperforms the FIFO and small-size-first baselines when ranking PRs that can be quickly merged or rejected. By considering the high performance of the LtR model, reviewers could use the model to aid them in merging more contributions in a shorter time.

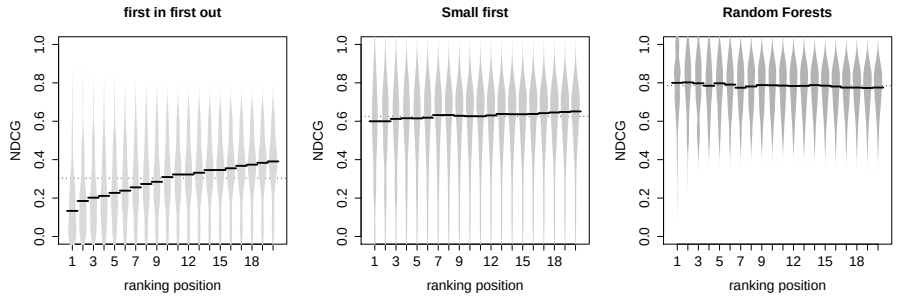

Fig. 8 The performance of our LtR approach and two baselines to rank PRs that can be quickly merged.

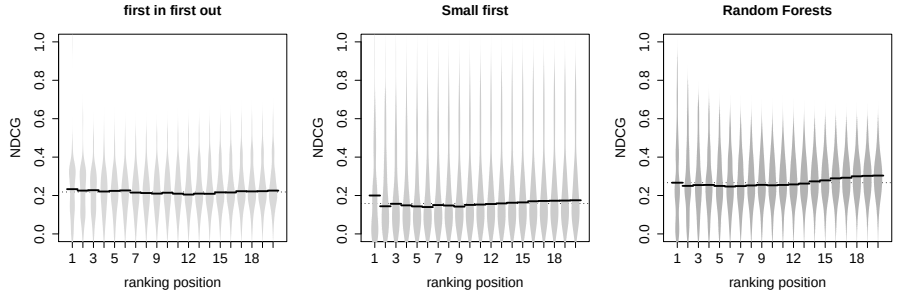

Fig. 9 The performance of our LtR approach and two baselines to rank PRs that can be quickly rejected.

4.3 RQ3. What are the most significant metrics that affect the ranking?

Motivation. In our experiment, we leverage many metrics that capture different aspects of a PR. In this research question, we investigate the effect of each metric on ranking the PRs that can be quickly merged and observe the most influential metrics in the LtR model. This is important to better understand how a PR can receive a decision more quickly with a small set of metrics that have the most significant impact on the ranking.

Approach. To test the effect of each metric, we first exclude a given metric from the training dataset. Next, we build the random forest model on the new dataset and test its performance. Finally, we measure the difference in the performance in terms of $NDCG@k$ regarding ranking PRs that can be quickly reviewed of the ranking model without the metric. The larger the drop in the performance, the higher the effect of the tested metric in the LtR model.

Results. **The metrics related to the social connection of a contributor are the most significant metrics.** Table 7 shows the effects of each metric. The percentages in Table 7 show the decreases in the performance measure $NDCG@k$ of the random forest model after removing the tested metric.

Without using *soical_distance* and *followers*, the performance of the random forest model decreases the most (4.13% and 3.20% off in the first position respectively). Both *soical_distance* and *followers* are related to the social connections of

a contributor. The first two most important metrics suggest that the social connections of the contributor play an influential role in the decision making process of PRs. Our approach could be used to attract the attention of reviewers when a PR is submitted by a contributor with a good social connection. Another two important metrics are *num_commits* and *ccn_deleted*, which represent the scale of the PRs and the source code quality of PRs respectively, as we explained in Section 3.2. We also find that the experience of the contributors is not as significant as the social connection of contributors and source code quality of PRs, especially for *prior_interaction*, based on the negative values shown in Table 7. When we remove *prior_interaction* from the LtR model, the performance of the LtR model increases by 3.26% at the first position.

In addition, we observe that *description_length* and *readability* are important, which suggests that the title and description texts of PRs have an important effect in the reviewing process. The negative effect of *src_churn* on LtR model indicates that the churn in PRs is not an important prioritization criterion in all of the cases for which a quick decision could be taken.

GitHub contributors can leverage the important metrics found in our LtR model to attract the reviewers' attention. For example, a new external contributor should seek interactions with reviewers in the same project to build a social connection with the reviewers. The external contributor is encouraged to follow reviewers on GitHub and comment on PRs. By doing so, the external contributor may engage in discussions with reviewers before submitting a PR to the project. In addition, contributors should write meaningful descriptions, titles and commits messages of PRs, so that reviewers feel inclined to work on the submissions of the contributors.

Summary of RQ3: The social connection of a contributor is more important than the source code quality of a PR and the experience of the contributor. And reviewers should not only consider the source code churn of the PRs as a single criterion for prioritizing reviews.

Table 7 Effects of each ranking metric on PRs.

Test metric	k=1	k=3	k=5
social_distance	4.13%	2.67%	2.17%
followers	3.20%	2.34%	1.99%
num_commits	1.96%	0.00%	0.04%
description_length	1.82%	2.08%	2.00%
ccn_deleted	1.36%	0.35%	0.39%
readability	1.19%	0.65%	1.41%
commit_file_changed	0.09%	2.05%	1.57%
bug_fix	0.47%	0.00%	0.72%
comments_added	0.46%	-0.08%	1.37%
contributor_succ_rate	-0.04%	0.05%	1.00%
status	-0.27%	-0.82%	0.17%
src_churn	-0.62%	0.18%	-0.25%
ccn_added	-1.17%	-0.86%	-0.24%
test_churn	-1.22%	-0.38%	-0.18%
files_changed	-1.85%	0.04%	-0.02%
prior_interaction	-3.26%	-1.45%	-0.91%

4.4 RQ4. How do GitHub reviewers perceive the usefulness of our approach?

Motivation. The goal of our approach is to help reviewers handle the increasing workload of PRs. Therefore, it is important to understand how reviewers perceive the usefulness of our approach. We also aim to investigate whether our approach can complement the existing practices for reviewing PRs.

Approach. We first collect reviewers that have e-mail addresses information on GitHub. We include reviewers from all projects on GitHub regardless of whether their projects are included in our experiment projects set to have a large scale generalized responses. We find 2,225 reviewers on GitHub that have e-mail addresses information and we send our survey to them. The survey consists of 4 open ended questions and 1 Likert scale question as shown in Table 8. The Likert scale question asks reviewers to rate the perceived usefulness of our approach from 1 (not useful) to score 5 (extremely useful). Question 2 asks reviewers the reasons as to why our approach can be useful. In Question 4, we ask reviewers whether our approach complements the existing prioritization approaches (i.e., FIFO and small-size-first). After designing our survey, we send it to code reviewers. Our e-mail template is available in Appendix B. To compensate reviewers for their time, we offer 10$ amazon gift cards to 20% of the reviewers who participated in our survey. Unfortunately, we could not reach 351 reviewers due to the obsolete e-mails. We receive 74 responses to our survey (our responses rate is 3.7% after excluding the 351 reviewers whose e-mail addresses are not reachable).

To obtain the overall perception of reviewers regarding the usefulness of our approach, we calculate the percentage of each score that we receive in Question 3 as shown in Figure 10. For the open ended Questions 2 and 4, we manually analyze all responses of reviewers and summarize the common reasons as to why our approach can be useful or complement the existing prioritization approaches (see Table 9). We also summarize the reviewers' concerns about our approach in Table 10.

Table 8 Description of the questions of our survey

Question	Type
1. How do you choose a pull request to review when you are facing several open pull requests?	Open ended
2. We are developing an approach to recommend reviewers with pull requests that are likely be reviewed in a short time. For example, you open the project page and you have the possibility of viewing a list of pull requests that are likely to be reviewed in a short time (i.e., regardless of size or topic). Do you think our approach would be useful? And why?	Open ended
3. Please provide how useful you think this approach would be in a scale from 1 (not useful) to score 5 (extremely useful)?	Likert scale
4. Prior research mentioned that developers prioritize pull requests based on the size of the patches and the age of the pull requests. Compared to these approaches, do you think that prioritizing pull requests based on their time to be reviewed (i.e., our approach) could complement them? And why?	Open ended
5. Do you have suggestions to improve the prioritization of pull requests in general?	Open ended

Table 9 Summary of the reasons on the usefulness of our approach

Reasons	Number of reviewers
1. Our approach helps reviewers to merge more PRs in a limited time	18/74
2. Our approach helps reviewers to prioritize potentially quick reviews besides the size and age of PRs	24/74
3. Our approach prevents several PRs from being blocked by the review of a single time-consuming PR	5/74

Table 10 Summary of the reviewers' concerns about our approach

Concerns	Number of reviewers
1. Our approach recommends PRs based on their possibility of being quickly reviewed instead of their usefulness or urgency	8/74
2. Our approach is incompatible with the specific settings of a project	3/74

Results. **Overall, the usefulness of our approach is acknowledged by code reviewers on GitHub.** As shown in Figure 10, 42% and 38% of reviewers who participated in our survey mark our approach as moderately useful and useful respectively. 7% of reviewers report that our approach is very useful. Finally, we receive no negative response stating that our approach is not useful. As shown in Table 9, 18 reviewers state that our approach can be useful since our approach enables them to merge more PRs in a limited time. As R2 and R72 state: *"In some cases e.g., at the end of the day or when short of time, getting some pull requests out of the way keeps development work going on"* and *"I want the time spent reviewing to have some result, rather than spending a lot of time on one large PR that might turn out to be the wrong approach or gets abandoned"*.

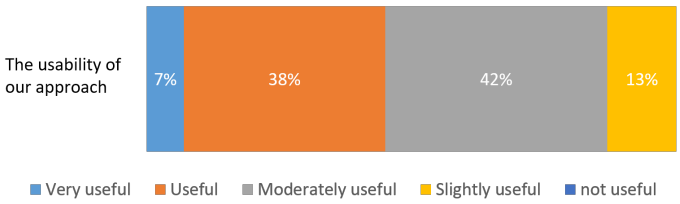

Fig. 10 The evaluation result of the usefulness of our approach.

24 reviewers mention that our approach can complement their existing working environment by helping them to prioritize potentially quick reviews besides the size and age of PRs. As R4, R24, and R45 mention in: *"Your approach can speed up a way how a reviewer will choose a right pull request without reading all existing pull request one by one."*, *"I would be able to choose the availability I have, and which factors I prefer at that moment (if I want low hanging fruits because I am short in time, for example)"* and *"I think it would complement the other prioritization*

criteria. Clearing off quick PRs is a good way to lower the cognitive load of having too many PRs open. It also improves the contributor experience and encourages concise, easy to review PRs."

As our approach recommends PRs that can be quickly reviewed first, 5 reviewers also think that our approach ensures that several PRs would not be blocked by the review of a single time-consuming PR. As R17 and R27 state: *"That would be helpful because I think it'd be best to review pull requests that are quicker before pull requests that are harder. That would ensure that several requests aren't blocked by the review of a single time-consuming pull request"* and *"Reviewing those PRs quickly may unblock other team members or tasks"*.

There are some concerns raised by reviewers as shown in Table 10. The most common concern is recommending PRs solely based on their possibility of being reviewed in a short time. If reviewing time was the single criterion for prioritizing PRs, factors, such as usefulness or urgency (e.g., a key bug fix) would be compromised. As R55 and R71 mention the following: *"whilst it may be useful to reviewers, it could be damaging for projects. Priority should be based on usefulness"* and *"A scheduling optimization strategy like this will lead to a Larger number of PRs being reviewed, but may postpone the review of more complex PRs. If those PRs are critical, the proposed strategy may starve them"*. We acknowledge that the mentioned concerns are valid and that is why our approach aims to complement the existing PRs prioritization criteria instead of replacing them. Reviewers can certainly follow their own prioritization rules and focus on PRs with a higher priority at first (e.g., PRs fixing critical bugs or implementing customer requested features). Moreover, as discussed in Section 5.2, we verify that our approach mostly recommends PRs that are related to software enhancements and bug fix rather than the trivial PRs (e.g., documentation PRs).

Other concerns are related to the specific settings of a project. For example, R15 mentioned: *"For us, this approach is not useful. Business priority is always most important"*.

In Question 5, we ask reviewers to provide suggestions to improve the prioritization of PRs. Reviewers suggest that the types of PRs can be used to prioritize PRs. As R63 mention the following: *"For example, cosmetic changes, bug fixes, functionality changes, refactor, and documentation changes may all have different priorities, depending on what the project managers deem most important"*. In addition, the reviewers suggest other aspects of PRs that can be leveraged to prioritize PRs, such as the labels of PRs, the number of times it takes until developers correct requested changes of reviewers, and the number of merge conflicts a PR causes.

5 Discussion

In this section, we discuss the soundness of our threshold to exclude noise PRs. Next, we discuss whether the PRs that our approach recommends are in fact useful and not trivial (i.e., simple modifications to documentation). We also discuss whether our approach is suitable to reviewers of diverse levels of experience.

5.1 Threshold to remove noise PRs

We conduct a sensitivity analysis to verify that the merge ratio threshold of 0.4 enables us to eliminate most of wrongly labeled PRs and keep correctly labeled PRs. We apply three different merge ratio thresholds (i.e., 0.4, 0.5, and 0.6). We do not include threshold of 0.7 because the normal merge ratio of PRs is around 0.7 and using 0.7 would exclude projects where the PRs are correctly labeled [10]. For each PR merge ratio threshold, we conduct the same experiment and evaluation approach covered in Sections 3 and 4. Figures 11 and 12 show the performance of our model of using different thresholds to exclude projects where PRs are incorrectly labeled.

In Figures 11 and 12, we observe that the performance of our approach keeps at the same level when we increase the merge ratio threshold. The Cliff's difference delta is negligible between the performances. The performance of recommending quickly merged PRs increases slightly (median Cliff's Delta estimate = 0.05) by raising the PRs merge ratio threshold, while the performance of recommending quickly closed PRs decreases slightly (median Cliff's Delta estimate = 0.04). A possible reason for the slight change in the performance is that a higher merge ratio threshold can exclude more wrongly labeled PRs, but also remove correctly labeled *closed* PRs. Therefore, we infer that the merge ratio threshold of 0.4 is able to eliminate most of wrongly labeled PRs while keeping correctly labeled PRs.

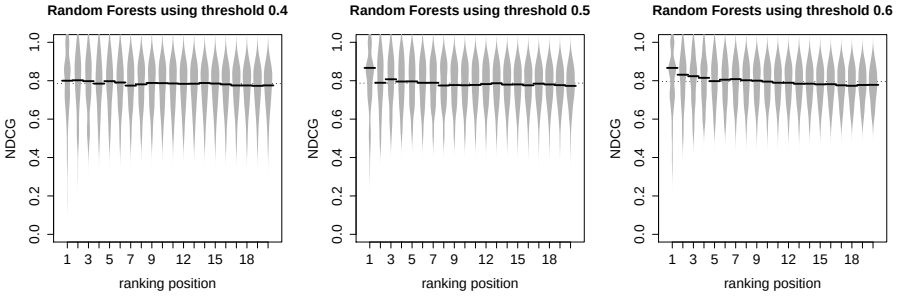

Fig. 11 The performance of our LtR approach to rank PRs that can be quickly merged under three different thresholds.

5.2 Types of the recommended PRs

The paramount goal of code reviewing is to maintain or improve the quality of the projects [5]. Therefore, it is critical to examine the impact of using our approach on improving the quality of projects. If our approach only recommends PRs that modify only project documentation, the benefit of improving the quality of a project is trivial compared with recommending bug fixing or feature implementation PRs. Although reviewers can always follow their own priority of PRs, our approach would not be useful to reviewers if we only recommended trivial PRs (e.g., documentation PRs).

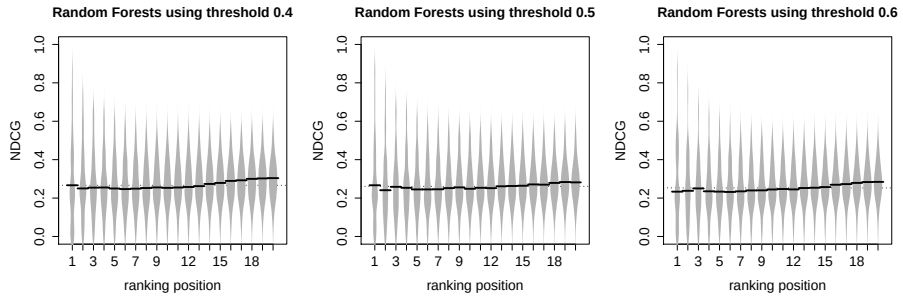

Fig. 12 The performance of our LtR approach to rank PRs that can be quickly rejected under three different thresholds.

Our approach recommends a total number of 1,480 PRs among the 74 projects in the experiment project set. We take a statistical sample of 305 PRs with 95% confidence level and with a 5% confidence interval. We randomly select 305 PRs from the entire population of recommended PRs. In prior research [40], different categories for commits are proposed to group commits. In our approach, we focus on PRs which consist of commits. Thus, we adopt the commits categories mentioned in research [40] to classify our PRs. We classify a PR based on the information contained in the description and title. We manually classify each PR in the sample into one of the types shown in Table 11. The number and percentage of PRs in each type are shown in Table 12.

Table 11 Types of pull requests [40]

Types of Pull Requests	Description
Bug fix	A pull request fixes one or more bugs.
Build	A pull request focuses on changing the build or configuration system flies (e.g., pom.xml files).
Clean up	A pull request cleans up the unused attributes, methods, classes.
Documentation	A pull request is designed to update documentation of a system (e.g., method comments).
Feature implementation	A pull request adds or implements a new feature.
Enhancement	A pull request performs activities common during a maintenance cycle (different from bug fixes, yet, not quite as radical as adding new features).
Test	A Pull request related to the files that are required for testing.
Others	A Pull request related to language translation, platform specific, token rename and source code refactoring.

Table 12 indicates that the majority of recommended PRs are related to enhancement, bug fixing, and new feature implementation. Conversely, documentation PRs only account for 7% of PRs recommended by our approach. A possible reason for the low percentage of documentation PRs is that our LtR model is built on the previous reviewed PRs. Reviewers usually prefer to review enhancement, bug fixing, and new feature implementation PRs rather than trivial PRs (i.e., PRs that do not aggregate much value for the project). Therefore, we conclude

that PRs recommended by our approach could help reviewers bring more positive contributions to the project and improve the quality of projects.

Table 12 The number of pull requests in each type

Pull request categories	Number of pull requests	Percentage of pull requests
Enhancement	78	26%
Bug fix	65	22%
Feature implementation	42	14%
Build	31	10%
Test	28	9%
Documentation	22	7%
Clean up	12	4%
Others	24	8%

5.3 PRs reviewed by reviewers of different experience levels

In this section, we check whether reviewers having different experience levels are interested in various types of PRs. This investigation is important because it checks whether our approach is suitable for reviewers of diverse levels of experience. For example, new reviewers may focus only on simple reviews (i.e., documentation or clean up PRs), while experienced reviewers are more willing to review complex PRs.

For each PR within the sample that we use in Section 5.2, we measure the experience of the reviewers who reviewed the PR. We select the following three metrics to measure the experience of reviewers:

- *prs_reviewed*. The number of PRs that one reviewer has reviewed before reviewing a specific PR.
- *prs_submitted*. The number of PRs that one reviewer has submitted to the project before reviewing a specific PR.
- *commits*. The number of commits that one reviewer has preformed before reviewing a specific PR.

We extract these three experience metrics by querying our collected dataset as explained in Section 3.1. Next, we group the experience levels of reviewers based on the type of PRs. After obtaining a set of experience levels for the reviewers of each PR type, we use the standard deviation to measure the variation on the experience levels (shown in Table 13).

We observe that, the standard deviation is high (the smallest value is 119.73) for each type of PR, which indicates that all PRs of each type are reviewed by reviewers with different levels of experience. Besides standard deviation, we also perform a Kruskal-Wallis H test [41] to test whether the experience levels of reviewers for different types of PRs have similar values. The null and alternative hypotheses are:

- *null hypothesis* H_0: The experience levels of reviewers of different types of PRs are similar.
- *alternative hypothesis* H_1: The experience levels of reviewers of different types of PRs are significantly different.

Based on the result of the Kruskal-Wallis H test shown in Table 14, the p-values of the three experience metrics are 0.31, 0.80 and 0.90, which are far above 0.05. Thus, we cannot reject the null hypothesis that all types of PRs are reviewed by reviewers of all levels of experience.

Therefore, based on the experiment results of standard deviation and Kruskal-Wallis H test, we observe that there is no apparent relationship between the experience levels of reviewers and the types of PRs that they are willing to review.

Table 13 The standard deviation of experiences of reviewers

Types of pull request	Standard deviation of experience of reviewers		
	Measured in *prs_reviewed*	Measured in *prs_submitted*	Measured in *commits*
Enhancement	295.25	225.32	818.59
Bug fix	295.19	177.31	641.26
Feature implementation	278.98	143.49	709.07
Build	325.42	237.34	725.38
Test	317.27	124.12	474.32
Documentation	274.98	215.52	894.90
Clean up	140.00	119.73	515.61
Others	253.77	207.45	851.70

Table 14 The result of Kruskal-Wallis H test

Reviewers experiences	p-value
Measured in *prs_reviewed*	0.31
Measured in *prs_submitted*	0.90
Measured in *commits*	0.80

6 Threats to validity

In this section, we discuss the threats to the validity of our study.

Threats to external validity concern whether the results of our approach are able to be generalized for other situations. In our experiment, we include PRs from 74 GitHub Java projects. We filter out projects to ensure that our analyzed projects are well-developed and popular among GitHub contributors. However, projects in other programming languages (e.g., Python and Javascript) may have different reviewing process for PRs, and our findings might not hold true for non-Java projects [6]. For example, the metrics that capture the characteristics of source code might be important for reviewing PRs in other language projects. To address this threat, further research including other language projects is necessary to obtain more generalized results.

Threats to internal validity concern the uncontrolled factors that may affect the experiment results. One of the internal threats to our results is that we assume that the behavior of reviewers does not change over time. As Gousios *et al.* [5] and Steinmacher *et al.* [6] mentioned, checking whether PRs follow the current developing goal of a project is the top priority for reviewers when evaluating

PRs. Also, it is possible that, at different stages of a project, the project enforces different developing policies (e.g., reviewers may focus more on bug fixing contributions than feature enhancement when the release time is approaching). However, it is challenging to capture all of these metrics (i.e., the changing goals of the project) without closely contacting the core contributors of each project. In spite of this challenge, we attempt to cover metrics from four dimensions: source code metrics, social connection metrics, experience metrics, and textual information metrics. Besides the changing developing policies, different reviewers may have different expertise. Ideally, our approach should be customized to recommend PRs based on the preferences of each reviewer. However, the majority of PRs in a project are reviewed by a few numbers of reviewers [16], which means we cannot obtain enough reviewing history for most reviewers. It is problematic to train a LtR model on each reviewer. Therefore, we only use objective metrics related to PRs to build a generalized approach that can work for all reviewers. Our goal is not to replace reviewers' prioritization criteria of PRs, but rather to be another tool at their hands to improve their working environment. Additionally, there may exist noise (i.e., incorrectly labeled PRs) in the rejected PRs, which can have a negative impact on the performance of the LtR models as mentioned in Section 4.1.

Threats to construct validity concern whether the setup and measurement in the study reflect real-world situations. In our study, we treat the time interval between the submitted time of a PR and the close time of the PR as the reviewing time. However, the time interval is a rough estimation of the actual time spent by reviewers on evaluating the PRs. In practice, there is always a delay for reviewers to notice the PRs after its submission and the delay should be counted in the reviewing time. Unfortunately, there is no information indicating the exact time when a reviewer started reviewing a PR. On the other hand, the delay time might reflect the priority of PRs. The longer a PR waits, the lower the priority of that PR might be. Therefore, we find it reasonable to use the overall time interval as an approximation of reviewing time, which is used to separate the studied PRs into quickly reviewed PRs and slowly reviewed PRs. In addition, it is difficult to quantify the exact amount of time saved from reviewers due to the unknown delay for reviewers to notice the PRs. Nevertheless, we observed that our approach can be useful for reviewers to review more PRs within a limited time.

7 Conclusion

The pull-based development model has become increasingly popular in collaborative software development platforms (e.g., GitHub, Gitlab and Bitbucket). In this development model, reviewers play an important role in maintaining the quality of their open source projects. We observe that the workload for reviewers tends to increase while the number of decisions made on PRs remains roughly the same over time.

In this paper, we propose a LtR model to recommend quick-to-review PRs to help reviewers to make more speedy decisions on PRs during their limited working time. We summarize the major contributions of this paper as follows:

- **We apply more metrics in the context of modeling PRs**. We use metrics to capture the quality of the source code and textual information of PRs, such as the source code cyclomatic complexity, the number of comments in the

source code, and the readability of the title and description of PRs. Besides, we integrate different aspects of metrics to build LtR models (e.g., metrics about PRs themselves and metrics reflecting the contributor's experience and social connections).
- **We test the effectiveness of different LtR algorithms on ranking PRs.** Our results suggest that the random forest model performs better than other five LtR models (RankNet, RankBoost, Coordinate Ascent, MART, and ListNet). The random forest model also outperforms two baseline models (i.e., the first-in-first-out baseline and the small-size-first baseline).
- **We observe that the social connection metrics affect the ranking the most**, while the source code quality metrics of PRs and experience metrics of contributors are less important.
- **The usefulness of our approach has been positively perceived by GitHub code reviewers.** Reviewers believe that our approach allows them to merge more PRs in a limited time and ensures that several PRs are not blocked by reviewing a single time-consuming PR. Reviewers also mention that our approach provides reviewers more options to prioritize PRs by complementing the other two prioritization criteria (i.e., FIFO and small-size-first).

In the future, we intend to include more projects written in different programming languages. Next, we plan to explore other machine learning techniques to recommend PRs for reviewers. We also intend to categorize reviewers into several groups and apply different PRs recommendation strategy for each group of reviewers.

References

1. G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 285–296.
2. C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in oss projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 26.
3. R. Padhye, S. Mani, and V. S. Sinha, "A study of external community contribution to open-source projects on github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 332–335.
4. B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.
5. G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 358–368.
6. I. Steinmacher, G. Pinto, I. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *ICSE18. 40th International Conference on Software Engineering*, 2018, p. 12.
7. E. Van Der Veen, G. Gousios, and A. Zaidman, "Automatically prioritizing pull requests," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 357–361.
8. R. Herbrich, T. Graepel, and K. Obermayer, "Support vector learning for ordinal regression," 1999.
9. R. Herbrich, "Large margin rank boundaries for ordinal regression," *Advances in large margin classifiers*, pp. 115–132, 2000.

10. G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.
11. J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th international conference on Software engineering*. ACM, 2014, pp. 356–366.
12. P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 2014, pp. 119–122.
13. P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 141–150.
14. V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 931–940.
15. Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer recommender of pull-requests in github," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 609–612.
16. ——, "Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 1. IEEE, 2014, pp. 335–342.
17. Z. Li, G. Yin, Y. Yu, T. Wang, and H. Wang, "Detecting duplicate pull-requests in github," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 2017, p. 20.
18. G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th working conference on mining software repositories*. IEEE Press, 2013, pp. 233–236.
19. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.
20. R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 112–121.
21. P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 67–76.
22. T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.
23. D. R. McCallum and J. L. Peterson, "Computer-based readability indexes," in *Proceedings of the ACM'82 Conference*. ACM, 1982, pp. 44–48.
24. P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.
25. M. Colaco, P. F. Svider, N. Agarwal, J. A. Eloy, and I. M. Jackson, "Readability assessment of online urology patient education materials," *The Journal of urology*, vol. 189, no. 3, pp. 1048–1052, 2013.
26. J. H. Zar, "Spearman rank correlation," *Encyclopedia of Biostatistics*, 1998.
27. C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," *Journal of Experimental Social Psychology*, vol. 49, no. 4, pp. 764–766, 2013.
28. H. Li, "A short introduction to learning to rank," *IEICE TRANSACTIONS on Information and Systems*, vol. 94, no. 10, pp. 1854–1862, 2011.
29. J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 852–861.
30. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 89–96.
31. Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *Journal of machine learning research*, vol. 4, no. Nov, pp. 933–969, 2003.

32. J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.
33. D. Metzler and W. B. Croft, "Linear feature-based models for information retrieval," *Information Retrieval*, vol. 10, no. 3, pp. 257–274, 2007.
34. Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 129–136.
35. L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
36. H. Li, "Learning to rank for information retrieval and natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 7, no. 3, pp. 1–121, 2014.
37. K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proceedings of the international workshop on Principles of software evolution*. ACM, 2002, pp. 76–85.
38. Y. Wang, L. Wang, Y. Li, D. He, W. Chen, and T.-Y. Liu, "A theoretical analysis of ndcg ranking measures," in *Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)*, 2013.
39. S. Niu, J. Guo, Y. Lan, and X. Cheng, "Top-k learning to rank: labeling, ranking and evaluation," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 751–760.
40. A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 99–108.
41. W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

Appendices

Appendix A Decision table

Appendix B Survey e-mail template

Dear %s,

My name is Guoliang Zhao. I am a PhD student at Queen's University, Canada. I am inviting you to participate in a survey (consists of only 5 questions and will not take you more than 5 minutes) about improving the pull requests review process because you are an active GitHub code reviewer. We apologize in advance if our email is unwanted or a waste of your time. Your feedback would be highly valuable to our research. This study will help us understand how Github code reviewers would react to our pull requests recommending approach. There are no mandatory questions in our survey. To participate, please click on the following link: https://docs.google.com/forms/d/e/1FAIpQLScUWVQkpS42fVNNfsh EKwrKKgb7EKJb5HxLVLbKpNJYbnqk1A/viewform?usp=sf_link

To compensate you for your time, you may win one $10 Amazon gift card if you complete the full questionnaire. We will randomly select 20% of participants as winners. We would also be happy to share with you the results of the survey, if you are interested.

If you have any questions about this survey, or difficulty in accessing the site or completing the survey, please contact Guoliang Zhao at g.zhao@queensu.ca or Daniel Alencar da Costa at daniel.alencar@queensu.ca. Thank you in advance for your time and for providing this important feedback!

Best regards,

Guoliang

Table 15 Decision table for highly correlated pairs

Variable1	Variable2	Choice
test_inclusion	*test_churn*	*test_churn*
src_churn	*files_changed*	*src_churn*
src_churn	*ccn_added*	*ccn_added*
src_churn	*ccn_deleted*	*ccn_deleted*
files_changed	*commit_file_changed*	*commit_file_changed*
ccn_added	*ccn_deleted*	*ccn_added*
src_churn	*comments_added*	*src_churn*
ccn_added	*comments_added*	*ccn_added*
test_churn	*src_churn*	*src_churn*
files_chnaged	*ccn_added*	*ccn_added*
contributor_succ_rate	*is_reviewer*	*contributor_succ_rate*
is_reviewer	*prior_interaction*	*prior_interaction*
src_churn	*commit_file_changed*	*commit_file_changed*
contributor_succ_rate	*prior_interaction*	*contributor_succ_rate*
test_churn	*files_changed*	*test_churn*
followers	*is_reviewer*	*is_reviewer*
test_churn	*ccn_added*	*test_churn*
test_churn	*comments_added*	*test_churn*
test_churn	*commit_file_changed*	*test_churn*
test_inclusion	*src_churn*	*test_inclusion*
test_inclusion	*files_changed*	*test_inclusion*
test_inclusion	*commit_file_changed*	*test_inclusion*
test_inclusion	*ccn_added*	*test_inclusion*
test_inclusion	*comments_added*	*test_inclusion*
files_changed	*comments_added*	*comments_added*
commit_file_changed	*ccn_added*	*commit_file_changed*
commit_file_changed	*comments_added*	*commit_file_changed*
followers	*prior_interaction*	*prior_interaction*
contributor_succ_rate	*followers*	*contributor_succ_rate*
followers	*ccn_deleted*	*ccn_deleted*
is_reviewer	*ccn_deleted*	is_reviewer