Adopting Learning-to-rank Algorithm for Reviewer Recommendation

Guoliang Zhao
Queen's University
Kingston, Canada
17gz2@cs.queensu.ca

Daniel Alencar da Costa
University of Otago
Dunedin, New Zealand
danielcalencar@otago.ac.nz

Jiawen Liu
Queen's University
Kingston, Canada
jiawen.liu@queensu.ca

Ying Zou
Queen's University
Kingston, Canada
ying.zou@queensu.ca

ABSTRACT

Software code review is a software quality assurance activity in which one or several developers check the quality of a source code change. The success of a code review depends on finding appropriate reviewers to inspect the code change, e.g., a pull request (PR). Otherwise, it can result in inefficient or low-quality code reviews. To match the expertise of a reviewer with a PR, existing approaches model the expertise of reviewers using different features (e.g., the file path similarity, the textual similarity, and the social connection features). However, the weights of different features are usually handcrafted and customized for each project. It is often time-consuming as the weights used in one project cannot be propagated to other projects. In this paper, we propose a learning-to-rank (LtR) approach that can automatically learn the best weights of features to recommend reviewers. We empirically study 80 GitHub projects to test the performance of our LtR approach and compare its performance with two baselines. The experiment results show that: (1) applying the maximum aggregation scheme to compute features improves the performance of our LtR approach; (2) the LtR approach outperforms the two baseline models by 28%, 10%, and 12% with respect to top-1 and top-3 accuracy, and Mean Reciprocal Rank on average; (3) the semantic similarity feature can be used to recommend appropriate reviewers; (4) and the social connection between the contributors and the reviewers is the most important feature to recommend appropriate reviewers to review PRs.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

Code reviewer, Reviewer recommendation, Pull request, Learning-to-rank

CASCON'22, November 15 - 17 2022, Toronto, Canada

ACM Reference Format:
Guoliang Zhao, Daniel Alencar da Costa, Jiawen Liu, and Ying Zou. 2022. Adopting Learning-to-rank Algorithm for Reviewer Recommendation. In *Proceedings of 32nd Annual International Conference on Computer Science and Software Engineering (CASCON'22).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Code review has proved to be critical to improve the quality of both corporate and open source software projects [6, 8]. During the code review process, developers inspect a code change made by others and communicate the required modifications before integrating the code change into the software. Modern Code Review (MCR) [1], a lightweight and tool-based code review methodology, has been widely adopted in collaborative software development platforms, such as GitHub, GitLab and Bitbucket. Compared with the traditional code review process which requires in-person meetings, MCR supports distributed software development. For example, on GitHub, a contributor submits a code change through a pull request (PR) and notifies a set of reviewers who might have the appropriate expertise to review the PR. The reviewers then evaluate the code change and suggest modifications. Based on the suggestions, the contributor improves the PR until it is approved by reviewers and passes the integration tests. Then, the code change is merged into the central repository, and the status of the PR changes to *merged*.

For an adequate code review process, the contributor must find reviewers with the appropriate expertise to review a code change. Thongtanunam *et al.* [22] have found that 4%-30% of patches have reviewer assignment problems, indicating that it takes 12 days longer to resolve such patches. However, assigning appropriate reviewers is a challenging task as the number of potential reviewers in large-scale distributed software projects can scale up to hundreds. For example, in our 80 studied projects from GitHub, we find that the number of reviewers in each project ranges from 18 to 491, while the median is 64. Given that the number of PRs tends to increase in a popular software project [31] and the number of reviewers might be large, it could be costly to manually assign reviewers in an ad-hoc manner. In recent years, several reviewer recommendation approaches have been proposed based on measuring the expertise of reviewers using different features, including file path similarity [22], textual similarity between the previously reviewed PRs and the new PR [23], and social connection between contributors and reviewers [28]. Reviewers are then ranked using their expertise

score, and the top reviewers are recommended to review the PR. Existing studies [23, 28] have found that combining different features can produce better performance than using one single feature. When combining different features, existing studies have assigned handcrafted weights to features and sum them up to represent the expertise score of reviewers. To relieve the practitioners from the tedious tuning process of deciding the weights of features, in particular for recommending reviewers for a large number of projects, we apply a Learning-to-rank (LtR) model to rank reviewers based on their expertise for a given PR and recommend the top reviewers to review the PR.

More specifically, LtR [13] is an approach that applies machine learning algorithms to rank the most relevant documents (e.g., reviewers) given a query (e.g., PR). LtR has been widely used to solve software engineering tasks [16, 25, 32] because it automatically learns the best configuration of different features (i.e., the weights of features achieving the best performance).

Moreover, summation is one of the most commonly used aggregation schemes in the existing studies [22, 23, 28]. In existing approaches, the similarity features (e.g., textual similarity and file path similarity) are measured between the previously reviewed PRs and a new PR. Then, these similarities are summated to represent the expertise of reviewers regarding the new PR. However, other aggregation schemes are demonstrated to produce less correlation among features and outperform the summation scheme in building defect prediction models [30]. Inspired by the defect prediction field, we explore seven aggregation schemes (e.g., summation, arithmetic mean, median, standard deviation, maximum, skewness, and kurtosis) to achieve the best performance of our approach. Moreover, we model the semantic meanings of PRs and leverage the semantic similarity between PRs to recommend reviewers to review a PR. The semantic similarity between documents is demonstrated to outperform the textual similarity in text representation [12]. In particular, we address the following research questions.

RQ1. What is the performance of our LtR model for recommending appropriate reviewers to review PRs?

We study the performance of the LtR model using seven aggregation schemes, including summation, mean, median, maximum, standard deviation, skewness, and kurtosis. We find that the maximum aggregation scheme outperforms other aggregation schemes to recommend appropriate reviewers to review PRs.

RQ2. Does our LtR model provide better performance for recommending reviewers?

We compare the performance of our model with two well-known and state-of-art approaches: FPS (i.e., file path similarity) approach [22] and IR+CN (i.e., textual similarity and social connection) approach [28]. Our results suggest that our LtR model can recommend appropriate reviewers more accurately, especially at the first position of the rank.

RQ3. What are the most significant features that affect the performance of the LtR model?

We find that the social connection between contributors and reviewers has the most impact on the performance of the LtR model.

Organization of the paper: We present the idea behind the LtR algorithm in Section 2. We describe our experiment setup and our results in Sections 3 and 4, respectively. We discuss the rationale for selecting our dataset in Section 5. The threats to validity are discussed in Section 6. Finally, we conclude the paper in Section 7.

2 LEARNING-TO-RANK APPROACHES

In LtR approaches, the relevance between a PR and a reviewer is estimated using a scoring function. The more expertise a reviewer has for a given PR, the higher the relevance between the PR and the reviewer.

$$rel(pr, re) = \sum_{i=1}^{k} w_i * f_i(pr, re) \quad (1)$$

Where $f_i(pr, re)$ represents the i_{th} feature; and w_i represents the weight of the i_{th} feature [26].

As Equation 1 shows, the scoring function is defined as a weighted sum of k features. Each feature, $f_i(pr, re)$ measures the expertise that a reviewer re has with respect to a PR, i.e., pr, from the perspective of the i_{th} feature, f_i. When a new PR comes, the LtR model computes the expertise score for the reviewers with respect to the new PR. Reviewers are ranked based on their expertise scores. The reviewers ranked at the top are recommended to review the PR.

In our approach, we apply the random forest [3] LtR algorithm, to learn the configuration (i.e., weights) of features. We choose the random forest LtR model because it has proved to outperform other LtR models in software engineering datasets [31].

3 EXPERIMENT SETUP

In this section, we introduce the process for collecting data and extracting features for building the LtR model. We also explain how we filter out correlated features. The overview of our approach is shown in Figure 1. We first select a set of experiment projects and collect the data of the selected projects. Next, we extract the features under different aggregation schemes and build a random forest LtR model using the 5 extracted features to recommend appropriate reviewers for a PR. We evaluate the performance of the LtR models using different aggregation schemes. Finally, we compute the effect that each feature has on recommending reviewers.

3.1 Collecting data

As shown in Figure 1, we first select experiment projects by leveraging project information hosted on GHTorrent (through Google BigQuery[1]). We then collect the data for our experiment by crawling information from GitHub.

Project Selection. To avoid running experiments on personal, inactive or toy projects [9], we filter out projects containing less than 2,000 PRs [5]. Our dataset contains data from November 2010 to April 2018. We filter out (*i*) projects that have less than 2,000 PRs; (*ii*) forked projects; (*iii*) and deleted projects. As a result, we obtain a set of 370 projects. Then, we apply the following criteria to refine our initial set of selected projects:

- We exclude the projects which have dominant reviewers, i.e., when a single reviewer is responsible for more than half of the PR reviews of a project. In such projects, contributors can always invite dominant reviewers to review their code changes, since they have the expertise for the majority of

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

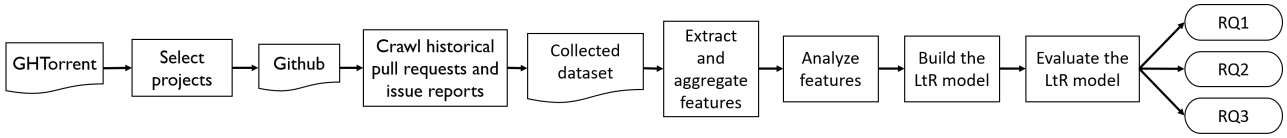

Figure 1: The overview of our approach.

the PRs. After removing projects with dominant reviewers, 135 projects remain.

- We exclude the projects that use external issue tracking systems (e.g., bugzilla and jira) as opposed to using the issue tracking system of GitHub [9]. We restrict our analyses to only issue reports on GitHub because we plan to combine PRs and issue reports. Nevertheless, our approach can be adapted to projects using other issue tracking systems, provided that PRs and issue reports can be linked together. 86 projects using the issue tracking system of Github remain.
- We manually investigate the remaining projects to exclude the projects that are not related to software development, i.e., projects which are used to store data or documents. Consequently, we remove 6 projects and obtain 80 subject projects.

Data Collection and Preprocess. We collect (*i*) the PRs and their related issue reports; (*ii*) comments; (*iii*) commits information; (*iv*) and code change information for each subject project. Contributors often mention the related issue report IDs on the PR. Thus, we extract the related issue report IDs from PRs by analyzing the HTML code on the page of each PR. The percentage of PRs that contain issue report IDs is shown in Appendix A in our online appendix[2]. For each PR, we combine the title, description, related commit messages and comments as the textual content for the PR. For the PRs that contain issue report IDs, we append the title and description of the related issues to the existing textual content of the PRs. In addition to the textual contents, we collect the code changes (i.e., code added or deleted in the PRs).

The descriptive statistics of the collected data are available in Appendix A in our online appendix[2]. In total, our dataset contains 80 GitHub projects, which have 498,084 PRs. There exist casual reviewers on GitHub who become inactive after contributing for a short period [17]. To avoid recommending casual reviewers, we recommend only active reviewers who contribute to the project within the last three months before the submission of a PR.

3.2 Feature Calculation

We use the features listed in Table 1 to capture the expertise of reviewers with respect to a PR. We include the features proposed by prior studies: the file path similarity [22], the textual similarity [23], and the social connection features [28]. The textual similarity comprises the similarity between the previously reviewed PRs and a new PR on a text level. To complement the textual similarity, we compute the semantic similarity between PRs. Both the textual similarity and semantic similarity are measured with the textual content of PRs. In the rest of this section, we introduce the rationale and computation process for each feature.

[2]https://github.com/liuj888/ReviewerRecommendationLtR/blob/main/ReviewerRecommendation_Appendix.pdf

Table 1. Features used to measure the expertise of reviewers

Feature	Description
social network connection	measures the social network connection between the contributor of a PR and a potential reviewer
file path similarity	measures the similarity between file paths of a new PR and file paths of the PRs reviewed by a reviewer
semantic similarity	measures the similarity between the semantic meaning of a new PR and the semantic meaning of the previous PRs of a reviewer
text based similarity	measures the similarity between the textual information of a new PR and the textual information of the previous PRs of a reviewer
source code similarity	measures the similarity between the source code of a new PR and the source code of the previous PRs of a reviewer

File Path Similarity Feature. The rationale for using the file path similarity feature is that code changes to the same files or files located in similar locations are likely to be reviewed by the same reviewers. For a new PR, we compute the similarity between the file paths of the new PR and the file paths of the previous PRs reviewed by each of the active reviewers following the process proposed in the prior studies [2, 22]. We show the algorithm used to calculate the file path similarity feature in Appendix B in our online appendix[2] .

Textual Similarity Feature. The rationale for using the textual similarity feature is that the similar PRs are often described in a similar manner (i.e., containing similar words in the titles and descriptions). A reviewer probably has the expertise to review a new PR if the new PR is described similarly to the PRs that the reviewer reviewed before. First, for an active reviewer, we compute the textual similarities between the previously reviewed PRs and the new PR. Next, we aggregate the similarities to compute the textual similarity feature for the active reviewer. To compute the textual similarities between PRs, we use the vector space model to represent each PR as a vector of term weights. Next, for each pair of PRs (i.e., one PR from the previously reviewed PRs and the new PR), we use the cosine similarity between the vectors of the two PRs to represent the textual similarity between them.

To represent PRs as vectors, we first process the textual content of PRs to remove stop words and perform stemming using the Python gensim[3] library. Then, we build the vector space model and

[3]https://radimrehurek.com/gensim/

produce vectors to represent PRs using the Python sklearn library[4]. The vector of each PR consists of the weights of the terms in the textual content of the PR. We capture the weight of each term using term frequency-inverse document frequency (tf-idf) as shown in Equation 2.

$$tf - idf(t, pr, PRs) = log(\frac{n_t}{N_{pr} + 1}) * log\frac{N_{PRs}}{pr_t \in PRs : t \in pr_t} \quad (2)$$

where t is a term in a pr, and PRs is the corpus of all PRs in a project. n_t and N_pr are the number of occurrences of a term t in a pr and the total number of terms in a pr, respectively. N_PRs is the total number of PRs.

Code Change Similarity Feature. The rationale for the code change similarity feature is that the PRs which have similar code changes (code added or deleted in the PRs) can be probably reviewed by the same reviewer. In our data collection phase, we collect the code changes for all PRs. The process for measuring the similarity of the code change is similar to the process of measuring the similarity of the textual content.

Semantic Similarity Feature. When computing the textual similarity, each PR is treated as a bag of words, which ignores the semantic meaning [10, 18]. The textual content of PRs and issue reports are manually written by contributors. Different contributors might choose different words to describe the same functionality or choose similar words to describe different contents. In addition, the semantic similarity between documents has proven to achieve better performance for the text classification [11, 14] than the textual similarity. Therefore, we measure the semantic similarity between the previous PRs of a reviewer and the new PR.

To compute the semantic similarities between the PRs, we build a document-to-vector (dtv) model and represent each PR as a semantic vector. The dtv model is a neural network model that can infer semantically similar documents. The dtv model learns the semantic meanings of documents based on the semantic meaning of words and the order of words in documents [12, 15]. The entries of the semantic vectors are the semantic features learned by the dtv model [7]. We use the processed textual content of the PRs that are used to compute the textual similarity to build a dtv model using the *Paragraph Vector* algorithm [12]. The *paragraph Vector* algorithm is implemented in the Python gensim library.

Each PR is represented as a vector. Then we compute the cosine similarity between the vectors to measure the semantic similarity between the PRs. For an active reviewer, we compute the semantic similarities between the previously reviewed PRs and the new PR. We then aggregate the similarities to compute the value of the semantic similarity feature.

Social Connection Feature. The rationale for the social connection feature is that the reviewers who share the common interests with a contributor are more likely to review the PRs from the contributor. As studied by Yu *et al.* [27, 28], the common interests between reviewers and contributors can be measured by the interactions (e.g., comments on PRs) between reviewers and contributors. The interactions can be obtained by mining the history of PR reviews. The computation of the social connection feature between an active reviewer (i.e., *RE*) and a contributor (i.e., *CO*) of a PR is shown in Equation 3. k is the total number of PRs submitted by the contributor (i.e., *CO*) and reviewed by the reviewer (i.e., *RE*). For each PR (i.e., PR_i), w_i is the social connection score related to the PR (i.e., PR_i) and m is the total number of comments from the reviewer (i.e., *RE*). λ is the decay factor used to distinguish between the comments submitted to multiple PRs and the comments submitted to the same single PR. When the reviewer (i.e., *RE*) places multiple comments on the same PR, the decay factor λ controls the social connection score that the reviewer (i.e., *RE*) can obtain. The decay factor is set to 0.8, based on the previous studies [27, 28]. $t(i, n)$ is a time-sensitive factor of a comment calculated in Equation 4. Time-sensitive $t(i, n)$ ensures that the recent comments are more important than the old comments.

$$SC(RE, CO) = \sum_{i=1}^{k} w_i = \sum_{i=1}^{k} \sum_{n=1}^{m} \lambda^{n-1} * t(i, n) \quad (3)$$

$$t(i, n) = \frac{timestamp(i, n) - start\_time}{end\_time - start\_time} \in (0, 1] \quad (4)$$

$timestamp(i, n)$ is the date at which the reviewer RE places the comment n in PR i. The start_time and end_time are the time when the first PR and the latest PR of a project were reviewed, respectively.

We compute the values of the aforementioned features to model the expertise of reviewers to review a new PR. However, when we collect the textual content of a new PR, we exclude the comments, since we cannot have the comments before reviewers start reviewing the new PR.

3.3 Aggregation Schemes

We measure the file path similarity, the textual similarity, the semantic similarity, and the code change similarity between the new PR and the previously reviewed PRs first at the PR-level and aggregate the similarities. Summation is one of the most commonly used aggregation schemes in the existing studies [22, 23, 28]. However, other aggregate schemes are demonstrated to result in less correlation between features, and outperform summation in the research of building defect prediction models [30]. Inspired by the observation of Feng *et al.* [30], we explore seven different aggregation schemes, i.e., summation, arithmetic mean, median, maximum, standard deviation, kurtosis, and skewness, to verify which aggregation scheme can obtain the best performance in our approach.

Summation. Summation captures the accumulative similarities between the previously reviewed PRs and a new PR. Reviewers with higher accumulative similarities for a new PR are more likely to have the expertise to review the new PR.

Central Tendency. The average similarity can be captured using the central tendency similarity. In this paper, we use the arithmetic mean and median to measure the central tendency. Reviewers with higher average similarities for a new PR are more likely to have the expertise to review the PR.

Dispersion. Dispersion comprises the standard deviation and the maximum aggregation schemes. The standard deviation measures the spread of similarities with respect to the central tendency. For example, for a reviewer with a high standard deviation, some previously reviewed PRs are more similar to the new PR, while some other previously reviewed PRs are less similar to the new PR. The maximum scheme measures the largest value of similarities. A

[4]https://scikit-learn.org/stable/

reviewer with a higher maximum value has reviewed a PR that is more similar to the new PR.

Shape of distribution. For a reviewer, the similarities between the previously reviewed PRs and a new PR form a distribution of values. In addition to leveraging the values of the distribution (i.e., similarities), we explore the shape of the distribution. We particularly study the skewness and kurtosis, which are two commonly used measurements to capture the shape of a distribution.

3.4 Correlation and redundancy analysis

Considering highly correlated and redundant features can result in redundant computations and increased processing time, we conduct correlation and redundancy analysis to remove highly correlated and redundant features.

We use the Spearman rank correlation because it can handle non-normally distributed data [29]. We use the *cor()* function in R to compute the correlation coefficient between features. If the correlation coefficient between two features is larger than or equal to 0.7, the pair of features is considered to be highly correlated and we select only one of the features [4]. We compute the correlation between the described features using each aggregation scheme and the result is shown in Table 2. The Aggregation Scheme column indicates in which aggregation scheme the correlation of the features occurs. Feature 1 and Feature 2 columns specify the features that are correlated. The Frequency column stores the number of projects in which the features are correlated. As shown in Table 2, the correlations mostly exist in the summation aggregation scheme. The Feature 2 column also indicates the feature that we choose for the analyses when the correlation pair occurs. We always keep the social connection because it has improved the performance of reviewer recommendation techniques proposed by the existing approaches [27, 28]. Similarly, we prefer the semantic similarity over the textual and code change similarity because the semantic meaning of documents has proved to achieve a better performance compared to the textual similarity.

Table 2. The correlated features

Aggregation Scheme	Feature 1	Feature 2	Frequency
maximum	code change	social connection	4
summation	code change	textual	63
summation	textual	semantic	60
summation	code change	semantic	44
summation	semantic	social connection	2
summation	code change	social connection	1

After performing the correlation analysis, we conduct a redundancy analysis on the features using the *redun()* function in R. Redundant features can be explained by other features in the dataset and do not bring extra values to the LtR model. We find that there is no redundant feature in our data.

3.5 Dataset labeling

To build a LtR model, we need to label the relevance between each active reviewers and a PR. There are two relevance levels in querying appropriate active reviewers to review a PR.

- Relevance level 0: indicates that a reviewer does not have the expertise required to review the PR.
- Relevance level 1: indicates that a reviewer has the expertise required to review the PR.

We label the relevance between each PR and all active reviewers based on the historical data. For example, if a PR (i.e., PR_1) is reviewed by a reviewer (i.e., RE_1) only, the relevance between the reviewer (i.e., RE_1) and the PR (i.e., PR_1) is 1 and the relevance between other active reviewers and the PR (i.e., PR_1) is 0.

4 RESULTS

In this section, we present the motivation, approach, and results for our three research questions.

4.1 RQ1. What is the performance of our LtR model for recommending appropriate reviewers to review PRs?

Motivation. In previous studies, the textual similarity, the file path similarity, the semantic similarity, and the code similarity are measured between the previously reviewed PRs and a new PR at the PR-level. The similarities at the PR-level are then aggregated to represent the expertise of reviewers regarding the new PR. Summation is one of the most commonly used aggregation schemes in the existing studies [22, 23, 28]. However, other aggregation schemes are demonstrated to result in less correlation among features and outperform using the summation scheme in the research on defect prediction [30]. To optimize the performance of our approach, it is crucial to select and deploy the most suitable aggregation scheme.

Approach. We use LtR model to rank reviewers who have the highest expertise to review the PRs. LtR models rank a set of documents (e.g., reviewers) based on their relevance to a given query (e.g., appropriate reviewers to review the new PR). The overall steps of our approach for recommending reviewers are shown in Figure 2.

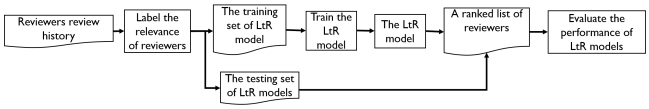

Figure 2: An overview of our LtR model.

4.1.1 The LtR algorithm and aggregation schemes selection. As explained in Section 2, we choose a random forest LtR algorithm because it outperforms other LtR algorithms in modeling PRs [31]. To aggregate the similarities at the reviewer level, we use seven aggregation schemes as introduced in Section 3.3.

4.1.2 Training phase. The LtR model is trained using a set of queries (i.e., PRs) $Q = \{q_1, q_2, \ldots, q_n\}$ and their related set of documents (i.e., active reviewers) $D = \{d_1, d_2, \ldots, d_n\}$. Each active reviewer can be interpreted as a document d for the query q of a PR. The active reviewers are then labeled with the relevance r in relation to the query q. During the training process, for each query, the LtR algorithm computes the relevance between each reviewer and the query using the feature vector V_s of the reviewer. In our study, we define d, r, q and V_s as follows:

- Document d is a reviewer.
- Query q is a query to identify the appropriate reviewers for a PR.
- The relevance r is whether d is the appropriate reviewer for the PR (i.e., 1) or not (i.e., 0).
- The feature vector V_s is a set of features used to build the LtR models, as discussed in Section 3.2.

4.1.3 Testing phase. It is critical to consider the time-sensitive nature of our data, i.e., the expertise of reviewers evolves over time, to test the LtR model. The time-sensitive nature requires us to test the model using PRs that are submitted after the PRs used in our training data. Therefore, we evaluate the performance of the LtR model following a time-sensitive validation approach as shown in Figure 3. First, PRs are sorted based on their creation time. Later, all PRs are split into 5 equally sized folds $F = \{f_1, f_2, \ldots, f_5\}$. Each f_i is then split into a training set t_i and a testing set τ_i. In the first iteration, we train a LtR model using the training set t_1 of PRs as queries, while we test the model by querying the PRs in the testing set τ_1. In the second iteration, a new LtR model is trained using the t_1, τ_1, t_2 sets (i.e., all PRs before τ_2) and tested using the PRs in τ_2 set. The iteration terminates when a LtR model is tested upon the τ_5 set. We then compute the average performance to represent the performance of the LtR model.

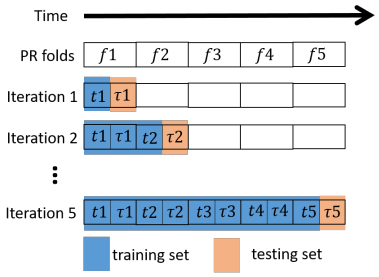

Figure 3: An overview of our time-sensitive evaluation

To evaluate the performance of our LtR model, we use the top-k accuracy and the Mean Reciprocal Rank (MRR). Both metrics have been applied to evaluate the recommendation approaches for solving software engineering problems [19, 21–24].

- Top-k accuracy measures the percentage of PRs where the appropriate reviewers are correctly recommended within the top-k positions of the result list. By considering the previous studies on reviewer recommendation, we set the k value to 1, 3, 5, and 10 in our study [22, 23].

$$Top-k\ accuracy(R) = \frac{\sum_{r \in R} isCorrect(r, Topk)}{|R|} * 100\% \quad (5)$$

Where $isCorrect(r, Topk)$ returns value of 1 if at least one of top-k reviewers are appropriate, otherwise, it returns 0. R represents a set of PRs and $|R|$ represents the number of PRs in set R.

- Mean Reciprocal Rank (MRR) computes the average position of the appropriate reviewer in the resulted list [16]. The higher MRR value indicates a better performance of the LtR model for ranking appropriate reviewers.

$$MRR(R) = \frac{1}{|R|} \sum_{r \in R} \frac{1}{rank(candidates(r))} \quad (6)$$

Where $\frac{1}{rank(candidates(r))}$ is the multiplicative inverse of the rank of the first appropriate reviewer in a rank list. If there is no appropriate reviewers in the rank list, the value of $\frac{1}{rank(candidates(r))}$ is 0. The result range from 0.0 to 1.0.

For textual similarity, file path similarity, semantic similarity, and code change similarity, we use the aggregation schemes mentioned in 3.3 to aggregate the similarities at the PR-level, then compute the features. Social connection feature represents the interactions between reviewers and contributors rather than the similarities between the new PR and the previously reviewed PRs. Thus, social connection feature is computed without the aggregation schemes. For each aggregation scheme, we obtain four aggregated features and the social connection feature and apply the random forest LtR algorithm. We follow the time-sensitive validation method as shown in Figure 3 to rank the reviewers to review the PRs of each project. We compute top-k accuracy and MRR for each project. After applying the random forest LtR algorithm to all projects, we obtain distributions of the top-k accuracy metrics at each position k (i.e., 1, 3, 5, 10) and the MRR metric. For each metric, we draw a beanplot[5] to show the performance of each aggregation scheme.

We compare the performance of the LtR model under different aggregation schemes. We use Cliff's delta to quantify the magnitude of the differences. Cliff's delta values range from -1 to +1, where a zero delta value signifies that two distributions have the same magnitude of values. A negative value indicates that the values of the first distribution tend to be smaller than the values of the second distribution, while a positive value indicates the opposite [30]. We use the *cliff.delta()* method of the *effsize* R package to compute the Cliff's delta. We map Cliff's delta values (i.e., d) to significance levels as proposed by Romano *et al.* [20]:

Negligible:	$0 \leq \|d\| < 0.147$
Small:	$0.147 \leq \|d\| < 0.330$
Medium:	$0.330 \leq \|d\| < 0.474$
Large:	$0.474 \leq \|d\| < 1$

Results. The LtR model achieves the best performance under the maximum aggregation scheme. The performance of different aggregation schemes are shown in Figures 4 and 5. For example, our LtR model achieves a median accuracy of 52% at the top-1 position under the maximum aggregation scheme. 52% top-1 accuracy signifies that for 52% of PRs our LtR model ranks the appropriate reviewer at the first position. The Cliff's delta between the maximum aggregation scheme and the other aggregation schemes are shown in Table 3. All Cliff's delta values shown in Table 3 are positive, which indicates that the maximum aggregation scheme outperforms other aggregation schemes. Under the maximum aggregation scheme, the LtR model achieves 51% and 81% median accuracy at top-1 and top-3 positions, respectively. For top-5 and

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

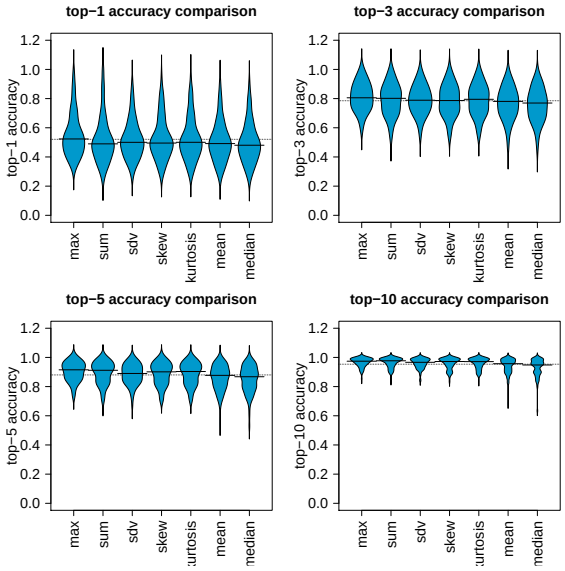

Figure 4: The top-k performance of the LtR model to rank the appropriate reviewers to review PRs using different aggregation schemes

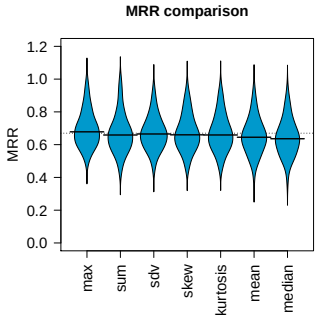

Figure 5: The MRR performance of the LtR model to rank the appropriate reviewers to review PRs using different aggregation schemes

top-10 positions, the median accuracy of the maximum aggregation scheme is close to 100%. Therefore, we select the maximum aggregation scheme to implement the LtR model and compare its performance with the existing approaches.

The maximum aggregation scheme outperforms other aggregation schemes to recommend the appropriate reviewers to review PRs.

4.2 RQ2. Does our LtR model provide better performance for recommending reviewers?

Motivation. After selecting the aggregation scheme used in the LtR model, we need to verify the effectiveness of the LtR model by comparing it with the existing approaches [22, 28].

Table 3. The magnitude of differences and Cliff's delta estimates between the maximum aggregation scheme and other schemes. In this table, 'neg' stands for negligible, 'max' for maximum, 'sum' for summation, 'std' for standard deviation, and 'skew' for skewness.

Comparison	Magnitude of differences and Cliff's delta estimates				
	MRR	top-1	top-3	top-5	top-10
max-sum	neg (0.14)	small (0.16)	neg (0.11)	neg (0.08)	neg (0.03)
max-mean	small (0.21)	small (0.17)	small (0.23)	small (0.25)	small (0.28)
max-median	small (0.26)	small (0.20)	small (0.27)	small (0.30)	medium (0.35)
max-std	neg (0.12)	neg (0.11)	neg (0.13)	small (0.15)	neg (0.10)
max-kurtosis	neg (0.11)	neg (0.10)	neg (0.11)	neg (0.11)	neg (0.10)
max-skew	neg (0.12)	neg (0.11)	neg (0.13)	neg (0.12)	neg (0.11)

Approach. We select two well-known and state-of-art reviewer recommendation approaches as the baselines. The two baselines are listed as follows:

- File Path Similarity (FPS) approach. The FPS approach recommends reviewers with the highest file path similarities to review a PR. In addition to use the longest common prefix to compute the similarity between file paths, the approach uses other three algorithms (i.e., the longest common suffix, the longest common substring, and the longest common subsequence) and combines the results of all algorithms together [22].
- IR+CN (i.e., the textual similarity and the social connection) approach. The IR+CN approach sums the textual similarity and the social connection score to recommend reviewers with the highest scores for a PR [28].

Following the same time-sensitive approach that we use to test the performance of the LtR model (see the approach of Section 4.1), we evaluate the two baselines. Similarly, we measure the performance of each baseline to rank reviewers who are appropriate to review PRs. Next, we use Equations 5 and 6 to calculate the top-k accuracy and MRR metrics. Similar to Section 4.1, after running the two baselines in all projects, we use beanplots and Cliff's delta to analyze the performance of the two baselines and compare them with the LtR model.

Results. **The random forest LtR model outperforms both the FPS and IR+CN baselines for recommending appropriate reviewers to review PRs.** Figures 6 and 7 show the performance of the LtR model and the FPS and IR+CN baselines for recommending reviewers for PRs. The Cliff's deltas between the performance of LtR and two baselines are shown in Table 4. As shown in Table 4, we observe a medium Cliff's delta (Cliff's Delta estimate = 0.42 and 0.40) between the LtR model and the two baselines at the top-1 accuracy. At the top-3 position, the Cliff's delta is medium between the LtR model and the FPS baseline and small between the LtR model and the IR+CN baseline.

Despite the high performance of the IR+CN approach observed by Yu *et al.* [28] (i.e., 0.7 top-1 accuracy), we observe in our study

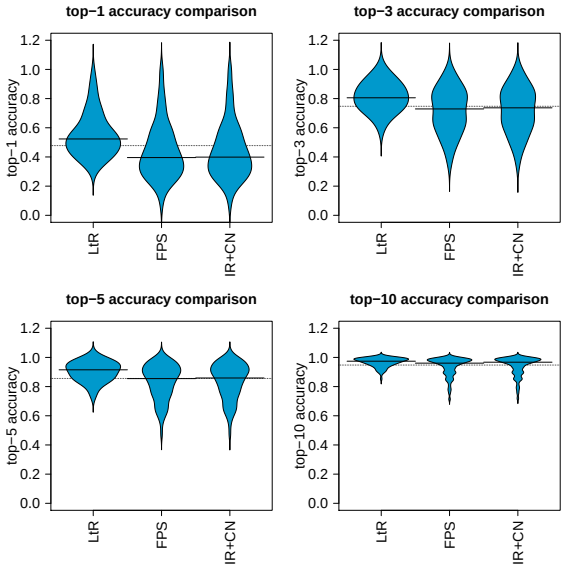

Figure 6: The top-k performance of the LtR model and two baselines to rank appropriate reviewers to review PRs

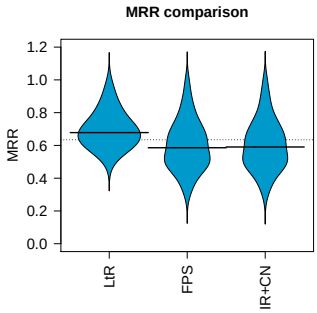

Figure 7: The MRR performance of the LtR model and two baselines to rank appropriate reviewers to review PRs

Table 4. The magnitude of differences and Cliff's delta estimates between the LtR model and two baselines

Comparison	Magnitude of differences and Cliff's delta estimates				
	MRR	top-1	top-3	top-5	top-10
LtR - FPS	medium (0.40)	medium (0.42)	medium (0.34)	small (0.34)	small (0.20)
LtR - IR+CN	medium (0.38)	medium (0.41)	small (0.32)	small (0.26)	negligible (0.1)

that the performance of the IR+CN is not as high in our dataset. We believe that is due to our dataset containing different types of PRs. In Yu *et al.*'s dataset, they kept only PRs that are reviewed by at least two reviewers to build and test their model, while we include PRs with any number of reviewers in our dataset. It is more challenging to recommend reviewers to review PRs in our dataset because there are 40.83% of PRs for which only one reviewer has the appropriate expertise.

Table 5. Effect of each feature on recommend appropriate reviewers.

Test feature	top-1	top-3	top-5	top-10
social connection feature	14.20%	6.45%	3.35%	1.20%
textual similarity feature	2.37%	1.16%	0.70%	0.31%
file path similarity feature	1.95%	0.70%	0.64%	0.22%
code change similarity feature	1.90%	1.11%	0.70%	0.30%
semantic similarity feature	1.24%	0.69%	0.50%	0.35%

The LtR model outperforms the FPS and IR+CN baselines when recommending reviewers who are appropriate to review a PR. GitHub contributors could use the LtR model to help them find appropriate reviewers to accelerate the PRs review process.

4.3 RQ3. What are the most significant features that affect the performance of the LtR model?

Motivation. In our experiment, we leverage five features to capture the expertise of reviewers related to a PR. In this research question, we investigate the effect of each feature on recommending the reviewers who are appropriate to review a PR. Next, we observe the most influential metrics in the LtR model.

Approach. To test the effect of a given feature, we first exclude the feature from the training dataset. Next, we build the LtR model under the maximum aggregation scheme on the new dataset and test its performance following the approach mentioned in Section 4.1. We measure the difference in the performance of the LtR model in terms of top-k accuracy metric without the feature. Then, the effect of the tested feature in the LtR model is measured by the drop in the performance.

Results. **The social connection is the most significant feature in the LtR model.** The effect of each feature is shown in Table 5. The percentages listed in Table 5 show the decreases in the performance as measured by the top-k accuracy of the LtR model after removing the tested feature.

Without using the social connection feature, the performance of the LtR model decreases the most (14.17% and 6.45% in the top-1 and top-3 position, respectively). Our result suggests that the interest shared by contributors and reviewers has a heavy influence when recommending reviewers to review a PR. It is probably due to the fact that the reviewer may be equally keen to integrate the changes made by the contributor. Similarly, the social connection feature has been observed to be the most important feature in predicting the review time of PRs [31]. Another important metric is the textual similarity feature, which measures the expertise of reviewers regarding a PR by calculating the textual similarity between the previously reviewed PRs and a new PR.

In addition, we observe that the file path similarity feature and the code change similarity feature have similar effects on the LtR model for recommending reviewers. The file path similarity feature measures the expertise of reviewers by analyzing the file paths of the modified files in PRs, while the code change similarity feature focuses on the modified code lines in PRs. Both features leverage

the code change information in the PRs. Moreover, we observe that the file path similarity and the code change similarity are equally important to recommend reviewers and can complement each other. The semantic similarity feature holds the least effect in recommending reviewers (i.e., only 1.25% off in the top-1 position). However, the percentage of the performance drop is still positive, which indicates that the semantic similarity feature can be used to measure the expertise of reviewers.

5 DISCUSSION

On GitHub, there exist projects where one reviewer is responsible for over 50% of the PRs reviews (i.e., dominant reviewers). In this discussion section, we elaborate on the need for us to build a recommendation model for projects with dominant reviewers.

We randomly select 10% (i.e., 23) of the 235 GitHub projects with dominant reviewers and collect data for these 23 projects. Next, we build a simple model which ranks reviewers based on the number of the previously reviewed PRs. Reviewers with the most number of reviews are recommended for a new PR. Then, we test the performance of the simple model and the LtR model on these 23 projects by following the time-sensitive approach as mentioned in Section 4.1. Figures 8 and 9 show the performance of the LtR model and the simple model. We use Cliff's delta to measure the difference between the performance of the LtR model and the simple model. The difference is listed in Table 6.

Table 6. The magnitude of differences and Cliff's delta estimates between the LtR model and the simple model

Comparison	Magnitude of differences and Cliff's delta estimates				
	MRR	**top-1**	**top-3**	**top-5**	**top-10**
LtR - simple	small (-0.25)	medium (0.44)	small (0.22)	small (-0.18)	medium (-0.38)

As shown in Table 6, the LtR model outperforms the simple model for ranking appropriate reviewers at the first and third position with medium and small difference magnitudes. However, the simple model achieves a better performance in the top-5 and top-10 accuracy with differences in the magnitude of small and medium, respectively. For the overall ranking quality (i.e., MRR), the simple model outperforms the LtR model with a small difference magnitude. Based on the comparison results, we observe that the LtR model does not outperform the simple model to rank reviewers in projects with dominant reviewers. Therefore, we advocate that building an LtR model for projects with dominant reviewers is not required instead a simple solution, i.e., recommending reviewers with the most number of reviews, is more effective. We also test the performance of the simple model in our studied projects without dominant reviewers. We observe that the simple model obtains a 30% accuracy at the top-1 position. This result indicates that the LtR model outperforms the simple model in projects with no dominant reviewers.

6 THREATS TO VALIDITY

Threats to external validity is related to the generalizability of our results with respect to other project settings. In this paper, we conduct experiments on 80 GitHub projects. Although we analyze

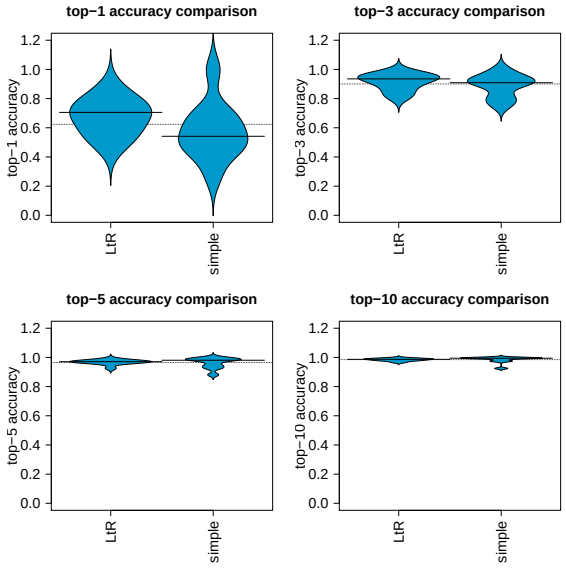

Figure 8: The top-k performance of the LtR model and the simple model to rank appropriate reviewers to review PRs

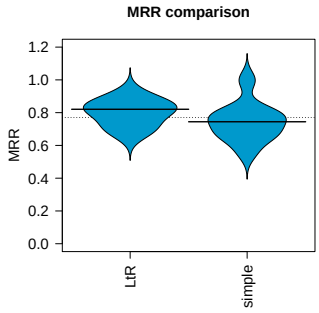

Figure 9: The MRR performance of the LtR model and the simple model to rank appropriate reviewers to review PRs

projects in a diverse range of programming languages, such as Python, Java, C++, and JavaScript, we have not investigated all other projects that use external issue tracking systems. We plan to conduct further studies to include projects using external issue tracking systems.

Threats to internal validity concern the uncontrolled factors that may affect the experiment results. One internal threat to our results is the availability of reviewers when a PR is submitted. For example, reviewers might become inactive in a project after contributing to the project for a short period. Unfortunately, there is no information to distinguish between inactive and active reviewers. In our work, we only consider reviewers who contribute to a project within the last three months before the submission time of a PR. However, it is inevitable that there might be noise in our approach (i.e., inactive reviewers).

Threats to construct validity are related to the degree to which our analyses measure what we claim to analyze. In our experiment, we label the relevance between reviewers and PRs based on

the PRs review history (i.e., whether a reviewer has the expertise to review a PR). However, we might overlook appropriate reviewers to review PRs. If a PR *PR*1 is reviewed by reviewer *RE*1 only, we assume that only reviewer *RE*1 has the expertise to review *PR*1. However, there might exist other appropriate reviewers. For example, reviewer *RE*2 might also have the right expertise, but reviewer *RE*2 can be occupied reviewing other PRs and has no time for *PR*1. In our work, we attempt to measure the expertise of reviewers from five different dimensions and automatically learn how to combine them together using the LtR algorithm.

7 CONCLUSION

Software code review has proven to be critical to improve the quality of both open source and corporate software projects. During the code review process, assigning appropriate reviewers to PRs can accelerate the code review process. Several features have been proposed by the existing research to model the expertise of reviewers from the review history. In this paper, we propose a new semantic similarity feature to measure the expertise of reviewers by analyzing the semantic meanings of PRs. We explore seven different aggregation schemes to compute the features and verify that maximum aggregation scheme is the best for measuring reviewers expertise. Moreover, we apply the LtR algorithm to automatically learn the best configuration of the five features for recommending appropriate reviewers. We observe that the social connection feature affects the LtR model for recommending reviewers most significantly. In the future, we plan to include more GitHub projects using external issue tracking system.

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[2] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. 2012. Cohesive and isolated development with branches. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 316–331.

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, and Ahmed E Hassan. 2016. The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues-An Empirical Study of the Mozilla Firefox Project. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 374–385.

[5] Manoel Limeira de Lima Júnior, Daricélio Moreira Soares, Alexandre Plastino, and Leonardo Murta. 2015. Developers assignment for analyzing pull requests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 1567–1572.

[6] Michael E Fagan. 1999. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 38, 2/3 (1999), 258.

[7] Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* (2014).

[8] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. 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, 358–368.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[10] Han Kyul Kim, Hyunjoong Kim, and Sungzoon Cho. 2017. Bag-of-concepts: Comprehending document representation through clustering words in distributed representation. *Neurocomputing* 266 (2017), 336–352.

[11] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *International Conference on Machine Learning*. 957–966.

[12] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International Conference on Machine Learning*. 1188–1196.

[13] Hang Li. 2011. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems* 94, 10 (2011), 1854–1862.

[14] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. 2015. Support vector machines and word2vec for text classification with semantic features. In *2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI* CC)*. IEEE, 136–140.

[15] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[16] Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22, 1 (2017), 259–291.

[17] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. 2016. More common than you think: An in-depth study of casual contributors. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 112–123.

[18] Ankita Rane and Anand Kumar. 2018. Sentiment Classification System of Twitter Data for US Airline Service Analysis. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 769–773.

[19] Shivani Rao and Avinash Kak. 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 43–52.

[20] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*. 1–33.

[21] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. 2011. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 365–375.

[22] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. 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, 141–150.

[23] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 261–270.

[24] Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang. 2014. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 275–278.

[25] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 191–200.

[26] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 689–699.

[27] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. 2014. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, Vol. 1. IEEE, 335–342.

[28] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.

[29] Jerrold H Zar. 1998. Spearman rank correlation. *Encyclopedia of Biostatistics* (1998).

[30] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. 2017. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering* 43, 5 (2017), 476–491.

[31] Guoliang Zhao, Daniel Costa, Alencar da, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering* (2019). https://doi.org/10.1007/s10664-019-09696-8

[32] Jian Zhou and Hongyu Zhang. 2012. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 852–861.