IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 7, 2019, accepted June 24, 2019, date of publication June 28, 2019, date of current version July 15, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2925560

Learning to Rank Reviewers for Pull Requests

XIN YE, (Member, IEEE)
California State University San Marcos, San Marcos, CA 92069, USA
e-mail: xye@csusm.edu

ABSTRACT In pull-based software development, anyone who wants to contribute to a project can request integration of the code changes to the public repository by sending a pull request to the development team. Upon receiving a pull request, a team member will review the changes and decide on merging it or not in the repository. Finding the appropriate reviewer for a pull request is a crucial step. To support reviewer recommendation, this paper introduces an adaptive ranking model to rank all the reviewer candidates for a pull request. The ranking model leverages 14 features to measure the relationships between a pull request and the reviewer candidates. The weight parameters of the ranking model are trained automatically based on previously resolved pull requests by using a learning-to-rank technique. The experimental evaluations on 12 open-source projects show that the proposed approach outperforms the baseline and a state-of-the-art approach. It can recommend the suitable reviewers within top-1 recommendation for over 80% of the pull requests in the opencv and jekyll projects. The feature selection experiments show that the most important feature is the feature that counts the number of previous pull requests sent by the requester and reviewed by a developer. The feature that measures the file path similarity between files changed in the pull request and files previously modified by a developer is another important feature.

INDEX TERMS Learning to rank, pull request, reviewer recommendation.

I. INTRODUCTION

Github is a widely used code hosting site that supports collaborative development. According to the official statistics, as of June 2018, Github has over 28 million users and 57 million repositories. It is the largest host of code in the world. Github leverages the pull-based development model [1], which is an emerging software development paradigm [1], [2], to support distributed development. In pull-based development, the code development effort is decoupled from the decision of integrating the code changes. In Github, any developers can clone any public repositories. For a public repository, external developers that do not belong to the development team (the core team) have only read access to it. In order to make contributions to the project, external developers can fork the repository to obtain a local copy with write privileges. After making changes to the cloned repository, external developers can request an integration of the code changes to the public repository by sending a pull request to the core team, which will make a decision whether to integrate the changes or not.

A pull request consists of code commits and tells the core team about changes that wait to be merged to the base branch. Upon receiving a pull request, the core team will review the changes, discuss potential modifications, and decide whether to merge it or not. A pull request may be assigned to one or more reviewers who are core team members. Reviewers may request clarifications and modifications in review comments. During the review process, there are two types of comments: review comments and normal conversation comments. Review comments are initiated by reviewers to raise review questions and request more changes. Normal conversation comments may be initiated by core team members or external developers to discuss any issues about the pull request. Github allows external developers to join the discussion about a pull request through normal conversation comments. The requester may response to both types of comments and may produce more changes in the following commits. If the changes are deemed to be satisfactory, one reviewer will approve the changes. Then a core team member will merge the changes to the base branch and close the pull request. It is noteworthy that the reviewer who approve the changes and the core team member who merge the changes may not be the same person. If the changes are considered to be problematic, the pull request will be rejected and closed.

Gousios *et al.* [3] conducted a survey with 749 core members and found that they feel overwhelmed in managing pull requests. Integrators are struggling to maintain the quality of the code and are spending a lot of time to review pull

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

requests [3]. Because it is the core members' responsibility to assue quality, it is crucial to find the appropriate core member with solid knowledge to lead the review process of pull request [4]. However, the process of assigning a reviewer to initiate the review process is commonly based on core members' interest in the pull request [5]. In some cases, it is time consuming to find the appropriate reviewer that takes the responsibility of approving or rejecting a pull request. Therefore, a key challenge of pull-development is to automatically and efficiently find the suitable reviewer that has the knowledge to review the pull request [4].

Currently there are two research areas about assigning developers to pull requests. One research area focuses on finding commenters to comment on pull requests. Such comments include review comments and normal conversation comments. Because Github allows any users comment on a pull request, a commenter may not be a core member with write access. A commenter may not be the reviewer, who finally approve or reject the changes. In this research area, Yu *et al.* [6] proposed a Vector Space Model (VSM)-based machine learning (ML) approach and a comment network (CN) approach to recommend developers to comment on pull requests. Later, they [7] used an information retrieval (IR)-based approach and a file location technique proposed by Thongtanunam *et al.* to improve the performance. Rahman *et al.* [8] also adapted the file location technique. Jiang *et al.* [9] analyzed previous pull requests to predict commenters for new pull requests.

Another research area aims at assigning core members to integrate the changes. Such integrators are reviewers, who review the changes and make the final decision to either approve or reject a pull request. In this research area, Moreira *et al.* [5], de Lima Júnior *et al.* [10], and Jiang *et al.* [11] proposed to use classification methods to recommend reviewers for pull requests. Instead of treating this problem as a classification problem, this paper considers it as a ranking problem.

To recommend reviewers for pull requests, this paper proposes a ranking approach. If a pull request is construed as a query, then the problem of recommending reviewers for a given pull request can be modeled as a ranking task in information retrieval (IR) [12]. Therefore, we propose to approach the reviewer recommendation problem as a ranking problem, in which the reviewer candidates are ranked according to their *relevance* to a given pull request. In this context, we equate relevance with the likelihood that a particular candidate has solid knowledge to review the pull request. We define the ranking function as a weighted sum of different features, where the weight parameters can be trained on previously resolved pull requests automatically using a learning-to-rank technique.

The main contributions of this paper include: approaching the reviewer recommendation problem as a ranking problem; a ranking model that leverages fourteen features to measure the relationships between a pull request and the reviewer candidates; exploiting a learning-to-rank technique to train the ranking model based on previously resolved pull requests; extensive evaluations and comparisons with the baseline and a state-of-the-art approach; a feature selection experiment that evaluates the importance of different features.

The rest of the paper is structured as follows. Section II discusses the related work. Section III introduces the ranking model. Section IV describes fourteen features in detail. Section V discusses experimental evaluations. Finally, the paper ends with conclusion in Section VI.

II. RELATED WORK

To the best of our knowledge, only Moreira *et al.* [5], de Lima Júnior *et al.* [10], and Jiang *et al.* [11] have proposed approaches to recommend core members to review pull requests. Some other approaches [6]–[9] were proposed to predict developers to comment on pull requests.

A. RECOMMENDATION APPROACHES TO ASSIGN CORE MEMBERS TO REVIEW (APPROVE OR REJECT) PULL REQUESTS

Moreira *et al.* [5] proposed to use three sets of attributes (features), among which one attribute set (set A) with 14 attributes are from their prior work [10] and one attribute set (set B) with 11 attributes are from the work of Jiang *et al.* [11]. These three attribute sets (set A, B and C) contain 36 attributes totally to measure the expertise of requesters, the social relationship between developers, the properties of pull requests and the expertise of integrators. They then employed different classification algorithms to classify pull requests to core members based on these attribute sets. They performed evaluations on 32 open-source projects collected from GitHub. They also performed feature selection in evaluation to select a more suitable set of attributes. They reported that their recommendation outperformed the state-of-the-art. They also evaluated the importance of different attributes and found that the most important attribute is the requester's login name.

Jiang *et al.* [11] proposed CoreDevRec. It is based on 11 attributes including attributes about the social relationship, attributes about file location, and attributes about the activeness of core members. In their approach, the more active of a core member the more likely he/she will review a pull request. Based on these attributes, they used Support Vector Machine (SVM) to classify pull requests to core members. They conducted experiments over five projects and reported that CoreDevRec achieved an accuracy of 72.3% for the Top-1 recommendation.

Both de Lima Júnior *et al.* and Jiang *et al.* model the training task as a classification problem in which pull requests are assigned to core members. We approach it as a ranking problem and directly train our model for ranking.

B. RECOMMENDATION APPROACHES TO PREDICT DEVELOPERS TO COMMENT ON PULL REQUESTS

Yu *et al.* [6] proposed to use VSM to convert a pull request (title and description) to a weighted vector, then use SVM to classify it to a label (developer). In the same work [6], they

proposed a comment network (CN) that depicts the comment relationship between developers. When a new pull request is received, CN will assign it to the developer that has commented on the requester frequently. Later, they [7] proposed to use an IR-based technique and a file location technique to improve the performance. The IR-based technique computes the cosine similarity between a new pull request and a resolved pull request. It then calculates a expertise score of a candidate develop by summing up the similarity scores of pull requests that he/she has commented. Developers with larger expertise scores are recommended as commenters. The file location technique was proposed by Thongtanunam *et al.* It computes file-path similarity scores between file names in a new pull request and file names in the solved pull requests. The file-path similarity scores are propagated to the developers who has commented the corresponding pull-requests. Developers are then ranked based on their expertise scores. The top-ranked developers are recommended to comment on the new pull request. They performed experiments over 84 open-source projects and reported that the IR+CN approach achieved the best result: the top-10 recommendation achieved 79.0% of recall and 33.8% of precision.

Rahman *et al.* [8] extract external libraries and specific technologies, a bag of tokens, from the source code attached to the pull requests. They compare an open pull request and a resolved pull request, i.e. two sets of tokens, using cosine similarity. They compute the expertise score of a developer by summing up the similarity values of resolved pull requests commented by him or her. Developers are ranked and recommended as commenters based on their expertise scores.

Jiang *et al.* [9] proposed attributes about the number of reviewed pull requests of every developer with consideration of a time window, the similarity between the title of an open pull request and the title of a resolved pull request, the similarity between files changes in an open pull request and files changed in a resolved pull request, and the social relationship between different developers.

III. RANKING MODEL

We defined a ranking model, which is a weighted sum of k features, to assign a matching score for any pull request r and reviewer candidate d combination. Equation 1 shows the scoring function $s(r, d)$, where each feature $\phi_i(r, d)$ represents a specific type of relationship between the reviewer candidate d and the received pull request r:

$$s(r, d) = \mathbf{w}^T \Phi(r, d) = \sum_{i=1}^{k} w_i * \phi_i(r, d) \tag{1}$$

When an arbitrary pull request r is received at test time as input, the model computes the score $s(r, d)$ for each reviewer candidate c. It uses this score to rank all the reviewer candidates in descending order. Then a ranked list of reviewer candidates is presented to the user. We expect that reviewer candidates in higher positions of the ranked list have larger chance to be suitable for the pull request i.e., more likely to have the knowledge to review the pull request.

The model parameters w_i can be trained by using a learning-to-rank technique based on previously resolved pull requests. The optimization procedure of this learning technique tries to optimize the model parameters so that the scoring function can rank the candidates that are known to be the actual reviewers of a pull request at the top of the ranked list for that pull request.

IV. FEATURE ENGINEERING

This section discusses features proposed for measuring the relationship between pull requests and core members.

A. FILE PATH SIMILARITY

If files changed in a pull request have been previously modified by a developer, this developer should have the knowledge to review this pull request. Similarly, if a developer has recently modified files that are located at similar file system paths with the changed files in a pull request, this developer may also has similar experience to review these changes. Thus, we extend a state-of-the-art approach [13] to compute the similarity scores between file paths involved in a pull request and file paths that are modified by a developer recently.

Given two files f_i and f_j, we use a slash character as a delimiter to split their file paths into components. Each component is a word. Then their file path similarity $filePathSim(f_i, f_j)$ is computed as in Equation 2, where $LCP(f_i, f_j)$ is the longest common prefix of f_i and f_j, and $Len(f_i)$ is the number of components in f_i. LCP calculates the number of common components that appear in both file paths from the beginning to the last. The intuition is that files locate at the same directory may have similar functionality [13].

$$filePathSim(f_i, f_j) = \frac{LCP(f_i, f_j)}{\max(Len(f_i), Len(f_j))} \tag{2}$$

$$\phi_1(r, d) = \begin{cases} \sum_{f_i \in F_r} \sum_{f_j \in F_d} \frac{filePathSim(f_i, f_j)}{|F_r| \times |F_d|} & \text{if} |F_d| > 0 \\ 0 & \text{if} |F_d| = 0 \end{cases} \tag{3}$$

We let F_r denotes the set of files changed in pull request r. We let F_d denotes the set of files that are modified by developer d recently (i.e., within one week). It is straightforward to get F_r from a pull request r. To get F_d, we look at the recent commits committed by developer d.

We then compute feature 1 $\phi_1(r, d)$ using Equation 3. If developer d did not modify any files recently (i.e., $|F_c| = 0$), $\phi_1(r, d)$ is set to 0. Otherwise, $\phi_1(r, d)$ is the sum of file path similarity values between files changed in the pull request and files modified by the developer recently normalized by $|F_r| \times |F_d|$.

Similarly, we let F_s denotes the set of files in the pull requests reviewed by developer d, then we compute feature 2

using Equation 4 [7], [13].

$$\phi_2(r,d) = \begin{cases} \sum_{f_i \in F_r} \sum_{f_j \in F_s} \frac{filePathSim(f_i,f_j)}{|F_r| \times |F_s|} & \text{if} |F_s| > 0 \\ 0 & \text{if} |F_s| = 0 \end{cases} \quad (4)$$

B. TITLE SIMILARITY

When a new pull request is received, if a developer has reviewed similar pull requests before, this developer might have similar experience to review the new pull request.

Given the title of a pull request, we remove the stop words, punctuations and numerical numbers. We perform stemming using the Porter stemmer. Then we use a standard information retrieval model, the classic Vector Space Model (VSM), to represent it as a vector of term weights. For a document d (a pull request title), for each term t in the vocabulary, we compute the term weights $w_{t,d}$. We use the classical *tf.idf* weighting scheme to compute the term weights. In this weighting scheme, the term frequency factor *tf* is normalized as follows:

$$\begin{aligned} w_{t,d} &= nf_{t,d} \times idf_t \\ nf_{t,d} &= 0.5 + \frac{0.5 \times tf_{t,d}}{\max_{t \in d} tf_{t,d}} \quad idf_t = log \frac{N}{df_t} \end{aligned} \quad (5)$$

$tf_{t,d}$ refers to the *term frequency* factor that counts the number of times a given term t appears in document d. df_t refers to the *document frequency* factor counting how many documents contain term t. The total number of documents is represented by N. Using a logarithm, the *inverse document frequency* idf_t is computed to alleviate the impact of the document frequency in the term weight.

The vocabulary of all words from all pull requests is denoted by V. Let $\mathbf{r_{new}} = [w_{t,r_{new}} | t \in V]$ and $\mathbf{r_{old}} = [w_{t,r_{old}} | t \in V]$ be the VSM vector representations of a new pull request r_{new} and an old pull request r_{old} respectively. As shown in Equation 5, we use the *tf.idf* formula to compute term weights $w_{t,r_{new}}$ and $w_{t,r_{old}}$. After computing the vector space representations, we use the standard *cosine similarity* to compute the textual similarity between a new pull request and an old pull request as follows:

$$sim(r_{new}, r_{old}) = cos(\mathbf{r_{new}}, \mathbf{r_{old}}) = \frac{\mathbf{r_{new}}^T \mathbf{r_{old}}}{\|\mathbf{r_{new}}\| \|\mathbf{r_{old}}\|} \quad (6)$$

Given a core member d, let $pr(d)$ be the set of pull requests that were reviewed by d. Feature 3 is then defined as follows:

$$\phi_3(r,d) = \sum_{r_{old} \in pr(d)} sim(r, r_{old}) \quad (7)$$

The feature computes the textual similarity between the title of the new pull request r and the titles of all the pull requests in $pr(d)$.

C. SOCIAL RELATIONS

If a developer has reviewed many pull requests sent by a requester, then this developer is likely to review a new pull request sent by the same requester. As such, we design feature 4 to measure the social relations between a requester and a developer.

When a new pull request r is received, we let q denote the requester and let $pulls\_reviewed(q,d)$ denote the set of pull requests that were sent by requester q and reviewed by developer d. As shown in Equation 8, we compute feature 4 by counting the amount of such pull requests.

$$\phi_4(r,d) = |pulls\_reviewed(q,d)| \quad (8)$$

If a developer has reviewed a requester's pull requests frequently and recently, this developer is likely to review this requester's new pull request again. Thus, we consider a time window and let $pulls\_reviewed_{30}(q,d)$ denote the set of pull requests sent by requester q and reviewed by developer d in the last 30 days, we computer feature 5 in Equation 9 as the total number of such pull requests.

$$\phi_5(r,d) = |pulls\_reviewed_{30}(q,d)| \quad (9)$$

Similarly, if a developer has commented on a requester's pull requests frequently, this developer may has the experience to review this requester's new pull request. As discussed in Section I, reviews and normal conversation comments are different. While any users can leave normal conversation comments on a pull request, only the core members can review it. A reviewer is a core member who has the knowledge to approve or reject a pull request. A commenter can be any user that leaves a normal conversation comment.

$$\phi_6(r,d) = |pulls\_commented(q,d)| \quad (10)$$

$$\phi_7(r,d) = |pulls\_commented_{30}(q,d)| \quad (11)$$

If a core member has left normal conversation comments on a requester's pull requests frequently, this core member may share common interest with the requester and may be a candidate to review this requester's new pull request. As such, we design feature 6 in Equation 10, where $pulls\_commented(q,d)$ refers to the set of pull requests sent by requester q and commented by developer d.

We let $pulls\_commented_{30}(q,d)$ denote the set of pull requests sent by requester q and commented by developer d in the last 30 days and design feature 7 in Equation 11.

D. ACTIVENESS

Active members may have the knowledge to review a new pull request. We let $pulls\_reviewed(d)$ denote the set of pull requests reviewed by developer d and $pulls\_reviewed_{30}(d)$ be the set of pull requests reviewed by d in the last 30 days. We then compute feature 8 and feature 9 in Equation 12 and Equation 13 respectively.

$$\phi_8(r,d) = |pulls\_reviewed(d)| \quad (12)$$

$$\phi_9(r,d) = |pulls\_reviewed_{30}(d)| \quad (13)$$

Similarly, we let $pulls\_commented(d)$ denote the set of pull requests commented by developer d and $pulls\_commented_{30}(d)$ be the set of pull requests commented

by c in the last 30 days, then we compute feature 10 and feature 11 in Equation 14 and Equation 15 respectively.

$$\phi_{10}(r, d) = |pulls\_commented(d)| \quad (14)$$

$$\phi_{11}(r, d) = |pulls\_commented_{30}(d)| \quad (15)$$

Next, we let $last(d)$ denote the most recent review activity of developer d and $last(d).day$ be the most recent day when developer d reviewed or commented a pull request. We let $r.day$ denote the day when the new pull request r was sent. Then we design feature 12 in Equation 16 as the inverse of the distance in days between r and $last(d)$.

$$\phi_{12}(r, d) = (r.day - last(d).day + 1)^{-1} \quad (16)$$

Thus, if r was sent in the same day that d reviewed another pull request, $\phi_{12}(r, d)$ is 1. If d last reviewed another pull request one day before r was sent, $\phi_{12}(r, d)$ is 0.5.

E. DAYS OF THE WEEK

Some developers temp to review pull requests during the weekdays (Monday to Friday) but not the weekend [14]. Some developers can review pull requests in the weekend (Saturday and Sunday). Developers' availability has an effect on the review activity. As such, we design feature 13 in Equation 17, where $pulls_{day}(r, d)$ refers to the set of pull requests reviewed by developer d on the day of the week when the pull request r was sent.

$$\phi_{13}(r, d) = |pulls_{day}(r, d)| \quad (17)$$

For example, if r was sent on Friday, feature $\phi_{13}(r, d)$ computes the total number of pull requests reviewed by d on Friday.

F. NUMBER OF FILES CHANGED IN A PULL REQUEST

We let $files(r)$ denote the number of files changed in pull request r. We let $avg(files(d))$ be the average number of files changed per pull request for all pull requests reviewed by d. Then we design feature 14 in Equation 18.

$$\phi_{14}(r, d) = (|files(r) - avg(files(d))| + 1)^{-1} \quad (18)$$

Thus, if the number of files changed in r equals the average number of files changed per pull request reviewed by d, $\phi_{14}(r, d)$ is 1.

G. FEATURE SCALING

Features with a wide range of values can be detrimental when training machine learning models. Many models tend to perform better when there is not a wide variation in the range of values for a given feature. To make different features be comparable to each other, we perform feature scaling to normalize all features to the same scale. Given some feature ϕ, we let $\phi.min$ denote the minimum observed value and let $\phi.max$ be the maximum for that feature in the training set. A feature may also have values present in the testing set that are larger than the observed maximum or smaller than the observed minimum found in the training set. To accommodate these scenarios, we scale the features in the testing and training dataset as follows:

$$\phi' = \begin{cases} 0 & \text{if } \phi < \phi.min \\ \dfrac{\phi - \phi.min}{\phi.max - \phi.min} & \text{if } \phi.min \leq \phi \leq \phi.max \\ 1 & \text{if } \phi > \phi.max \end{cases} \quad (19)$$

V. EMPIRICAL EVALUATION

This section discusses our empirical evaluation of the proposed reviewer recommendation approach.

A. DATA COLLECTION

We evaluate the proposed reviewer recommendation approach over 12 open-source projects collected from GitHub using GitHub API[1]. These projects are the most forked projects. They have 3,235 watches, 56,629 stars and 19,989 forks on average in GitHub. They are popular and widely used. Each contains more than 1,000 closed pull requests. Then we discard pull requests without reviews. Some pull requests have normal conversation comments but do not have reviews. Some pull requests have neither comments nor reviews. These pull requests are discarded. Because core members who submit pull requests temp to have other developers review their requests, self-reviewed pull requests do not represent such normal behavior [4]. Thus, pull requests that are reviewed by the same developers who make the requests are also discarded. Finally we collect a total number of 43,986 pull requests. For each project, those developers who have reviewed others' pull requests are identified as the reviewer candidates.

The statistics of our data are shown in Table 1. Column "Project" contains the project names. Column "Language" shows to the programming language that each project was written in. Column "Time Range" shows the time range when the pull requests were sent. Column "Pull Request Selected" shows the total number of selected pull requests for each project. Column "Top-1" shows the percentage of pull requests that are reviewed by the most active developer (i.e., the developer who review the most times). The most active developer represents the majority class for each project. Column "Top-3" contains the percentage of pull requests reviewed by the three developers who most reviewed pull requests in the project. Column "Top-5" shows the percentage of pull requests reviewed by the five developers who most reviewed pull requests in the project. Column "Commits" shows the number of commits for each project. Column "Reviewers" shows the number of reviewers who have reviewed others' pull requests in the project. These reviewers are also contributors to the project.

These 12 projects have 204 reviewer candidates on average. This number is relative large. For example, we need to identify the appropriate reviewers out of 370 candidates in

[1] https://developer.github.com/v3/

TABLE 1. Benchmark datasets.

Project	Language	Time Range	Pull Requests Selected	Majority Classes			Commits	Reviewers
				Top-1	Top-3	Top-5		
tensorflow	C++	2015-11 – 2019-03	6,335	13.12%	27.36%	39.51%	51,974	370
opencv	C++	2012-07 – 2019-03	1,603	69.49%	84.96%	90.19%	26,395	70
bitcoin	C++	2010-12 – 2019-03	2,652	22.74%	46.52%	62.84%	20,117	233
electron	C++	2013-06 – 2019-03	3,279	33.88%	68.62%	79.04%	21,631	128
swift	C++	2015-11 – 2019-03	6,843	18.35%	38.37%	48.17%	85,615	228
node	JavaScript	2015-11 – 2019-03	11,039	39.88%	55.31%	65.76%	26,762	447
react	JavaScript	2013-06 – 2019-03	2,292	35.56%	69.90%	77.72%	10,856	261
keras	Python	2015-03 – 2019-03	1,416	60.78%	91.68%	96.02%	5,104	139
pandas	Python	2015-03 – 2019-03	3,774	70.48%	80.14%	87.38%	19,218	138
scikit-learn	Python	2010-09 – 2019-03	1,973	59.91%	80.01%	89.24%	23,898	121
jekyll	Ruby	2010-11 – 2019-03	918	57.84%	78.22%	85.50%	10,448	50
rails	Ruby	2010-09 – 2019-03	1,862	30.56%	50.50%	63.23%	73,110	261

the tensorflow project. The project Node.js has the maximum number (11,039) of selected pull requests, 447 reviewers candidates and 26,762 commits. The project jekyll has the minimum number (918) of selected pull requests, 50 reviewer candidates and 10,448 commits. For tensorflow, bitcoin and swift, the Top-1 majority reviewers reviewed less than 30% of the pull requests, which indicates that there are several active reviewers in these teams. For opencv, keras and pandas, the Top-1 majority reviewers reviewed more than 60% of the pull requests, which indicates that one reviewer in each project is responsible for most the pull requests.

B. EVALUATION METRICS

We split the dataset into multiple folds. For each pull request from a test fold, to test the model, we use the learned weights to compute the weighted scoring function $s(p, c)$ for each reviewer candidate. We then rank all the candidates in descending order based on their score values. We compare the ranking result with the ideal ranking in which the candidates who are reviewers should be listed at the top. Finally, the overall system performance is computed by pooling together the ranking results from all test folds. We use the following evaluation metrics to assess the ranking performance.

- *Accuracy@k* represents the percentage of pull requests that at least one correct recommendation appears within the top k positions in the ranked list.
- *Mean Average Precision (MAP)* is defined as the mean of the Average Precision (AvgP) values across all the queries. It is a standard metric measuring the overall ranking performance of an IR system [12].

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, \quad AvgP = \sum_{k \in K} \frac{Prec@k}{|K|} \tag{20}$$

Here Q denotes the set of all queries (i.e., pull requests), K denotes the set of the positions of the positive instances (the candidates who are reviewers) in the ranked list. $Prec@k$ denotes the precision over the top k instances in the ranked list:

$$Prec@k = \frac{\text{\# of relevant docs in top k}}{k} \tag{21}$$

- *Mean Reciprocal Rank (MRR)* [15] is a metric measuring the ranking performance on the first recommendation, where $first_q$ is the position of the first positive instance (the first candidate who is a reviewer) in the ranked list, for each query q:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{first_q} \tag{22}$$

C. LEARNING TO RANK & HYPERPARAMETERS

As shown in Equation 1, our ranking model $s(p, c)$ is a weighted sum of different features, where each feature capture a specific type of relationships between a pull request p and a reviewer candidate d. To train the model parameters w_i, we use the SVM^{rank} package [16], which is a implementation of a learning-to-rank approach [17]. In this learning framework, learning $\mathbf{w}$ means solving the optimization problem shown in Equation 23, where $\mathcal{R}$ denotes the set of pull requests in a training set, $\mathcal{P}(r)$ is the set of positive instances (reviewers for pull request r), and $\mathcal{N}(r)$ is the set of negative instances (non-reviewers for pull request r).

The optimization procedure of this learning technique tries to optimize the model parameters so that the scoring function can rank the candidates that are known to be the actual reviewers of a pull request at the top of the ranked list for that pull request.

$$\begin{aligned} &\text{minimize: } J(\mathbf{w}, \xi) = \frac{1}{2}||\mathbf{w}||^2 + C \sum \xi_{rpn} \\ &\text{subject to: } \mathbf{w}^T \Phi(r, p) - \mathbf{w}^T \Phi(r, n) \geq 1 - \xi_{rpn} \\ &\qquad \xi_{rpn} \geq 0 \\ &\qquad \forall r \in \mathcal{R},\ p \in \mathcal{P}(r),\ n \in \mathcal{N}(r) \end{aligned} \tag{23}$$

To tune the size of the training dataset $\mathcal{R}$ and the number of negative instances per pull request $\mathcal{N}(r)$, we compute learning curves for two different projects: tensorflow and rails. For the tensorflow project, the latest 1000 pull requests are used for testing, the next 500 pull requests are used for validation, the rest 4835 pull requests are used in the pool of training examples. Similarly, for the rails project, the latest 1000 pull requests are used for testing, the next 500 are used for validation, and the rest 9539 are used in the training pool.

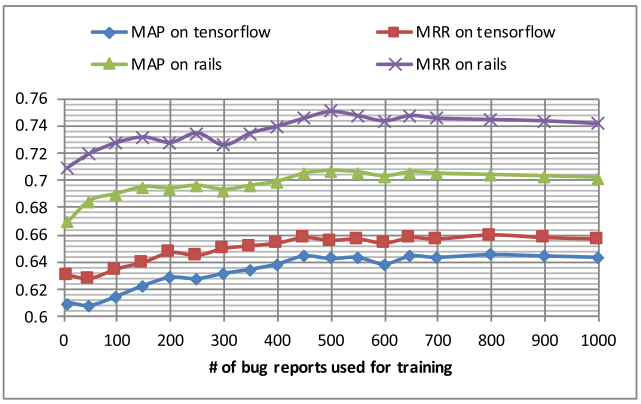

FIGURE 1. MAP and MRR as a function of $|\mathcal{R}|$.

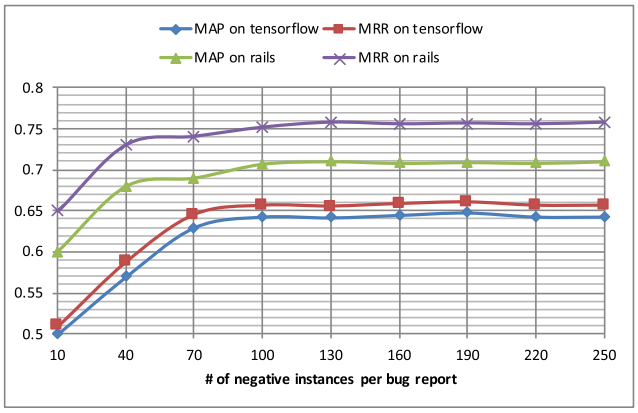

FIGURE 2. MAP and MRR as a function of $|\mathcal{N}|$.

To compute the learning curves, we repeatedly train the ranking model on $|\mathcal{R}|$ pull requests. At the beginning, we use the newest 10 pull requests from the training pool in $\mathcal{R}$. Then we use the newest 50 pull requests in $\mathcal{R}$. With an increments of 50, we continue to use the newest 100 pull requests and then up to all the pull requests from the training pool. Figure 1 shows the learning curves for $|\mathcal{R}|$. Both MAP and MRR grow obviously when the number of training pull requests in $\mathcal{R}$ grows from 10 to 500. However, beyond 500 pull requests, the performance in terms of MAP and MRR stays mostly flat. We observe that the two different projects have similar behavior. Therefore, we choose training sets $|\mathcal{R}|$ to be 500 for all twelve projects in the remaining experiments.

To tune the number of negative instances (non-reviewers for pull requests) $\mathcal{N}(r)$, we use a similar approach. First, we use the latest 500 pull requests from the training pool as the training examples $\mathcal{R}$. Then we repeatedly train the ranking model as we increase the value of $\mathcal{N}(r)$. At the beginning, we set $\mathcal{N}(r)$ to be 10. Then we increase $\mathcal{N}(r)$ with an increment of 30 and tune $\mathcal{N}(r)$ up to 250. Figure 2 shows the learning curves for $\mathcal{N}(r)$, where the performance in terms of MAP and MRR grows as $\mathcal{N}(r)$ increases from 10 to 100. Beyond 100, the learning curves of $\mathcal{N}(r)$ stay flat. Therefore, we fix $\mathcal{N}(r)$ to be 100 in the following experiments.

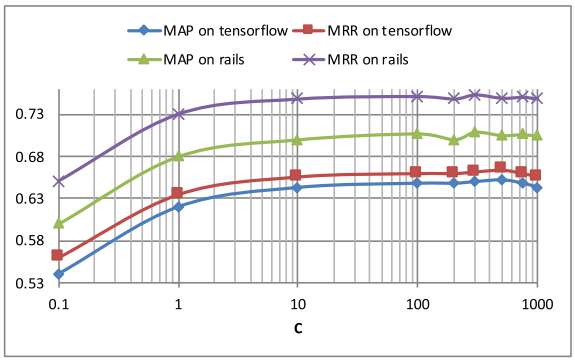

FIGURE 3. MAP and MRR as a functions of C.

Next, as shown in Equation 23, we tune the capacity parameter C of the ranking SVM model. First, we set the training size $\mathcal{R}$ to be 500 and the number of negative instances per pull request $\mathcal{N}(r)$ to be 100. Then we repeatedly train the ranking model by increasing C from 0.1 to 1000. The learning curves of C is shown in Figure 3, where both MAP and MRR increase substantially as C increase from 0.1 to 100. The learning curves become stable when C grows from 100 to 1000. Therefore, we let C to be 100 in the remaining experiments.

D. SPLITTING THE DATASET

Before splitting the dataset, we sort the pull requests of each project chronologically based on their create timestamp. For all the projects, we split their sorted pull requests into K equally sized folds $fold_1$, $fold_2$, …, $fold_K$, where $fold_1$ contains the oldest pull requests while $fold_K$ contains the most recent pull requests. Based on the tuning results to be discussed in the following section, we let each fold contain 500 pull requests. Then the total number of folds K can be computed as the number of pull requests divided by 500:

$$K = \frac{\text{\# of pull requests}}{500} \tag{24}$$

Therefore, the large projects tensorflow, bitcoin, electron, swift, node.js, react and pandas are split into 13, 6, 7, 14, 22, 5 and 8 folds, respectively. The pull requests from opencv, keras, scikit-learn, jekyll and rails are split into 4, 3, 4, 2 and 4 folds respectively. The ranking model is then trained on $fold_k$ and tested on $fold_{k+1}$, for all $1 \leq k < K$.

E. RESULTS AND COMPARISONS

We compared our learning-to-rank approach with the baseline and a state-of-the-art approach:

1) The majority class is used as the baseline. It shows the percentage of pull requests reviewed by the top-k (k = 1, 3, 5) developers who most reviewed pull requests in the project.
2) CoreDevRec [11] uses 11 attributes to measure the social relationship, file location and activeness of developers. It uses these 11 attributes to classify pull requests to different developers.

TABLE 2. Accuracies per project.

Project	Baseline Accuracy			CoreDevRec Accuracy					Learning-to-Rank Accuracy				
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	MAP	MRR	Top-1	Top-3	Top-5	MAP	MRR
tensorflow	13.12%	27.36%	39.51%	43.71%	61.90%	70.43%	0.507	0.585	**51.80%**	**74.80%**	**82.50%**	**0.643**	**0.656**
opencv	69.49%	84.96%	90.19%	85.51%	93.99%	95.73%	0.873	0.911	**85.59%**	**96.19%**	**98.50%**	**0.899**	**0.911**
bitcoin	22.74%	46.52%	62.84%	37.24%	63.57%	73.79%	0.401	0.540	**45.30%**	**70.80%**	**81.10%**	**0.525**	**0.608**
electron	33.88%	68.62%	79.04%	46.13%	70.44%	82.97%	0.581	0.601	**55.30%**	**76.60%**	**86.40%**	**0.611**	**0.687**
swift	18.35%	38.37%	48.17%	34.20%	51.50%	60.70%	0.357	0.513	**46.80%**	**69.30%**	**81.70%**	**0.580**	**0.608**
node	39.88%	55.31%	65.76%	41.10%	67.60%	72.40%	0.529	0.572	**48.70%**	**71.90%**	**76.20%**	**0.429**	**0.622**
react	35.56%	69.90%	77.72%	34.30%	62.70%	73.10%	0.397	0.513	**37.30%**	**70.30%**	**80.00%**	**0.492**	**0.543**
keras	60.78%	91.68%	96.02%	**62.71%**	95.33%	97.19%	0.713	**0.762**	61.37%	**96.33%**	**99.01%**	**0.731**	0.746
pandas	70.48%	80.14%	89.24%	67.30%	88.10%	94.90%	0.703	0.794	**73.80%**	**91.70%**	**96.70%**	**0.788**	**0.834**
scikit-learn	59.91%	80.01%	89.24%	65.90%	83.60%	88.10%	0.688	0.771	**73.50%**	**93.30%**	**97.10%**	**0.771**	**0.837**
jekyll	57.84%	78.22%	85.50%	70.10%	87.90%	95.60%	0.753	0.816	**83.10%**	**97.50%**	**99.20%**	**0.849**	**0.907**
rails	30.56%	50.50%	63.23%	46.30%	70.40%	81.80%	0.593	0.603	**62.20%**	**84.20%**	**90.60%**	**0.707**	**0.751**

Table 2 shows the experimental result. For tensorflow and swift, their baseline Top-1 majority classes are less than 20%, their Top-3 majority classes are less than 40%. Thus, multiple developers in these two projects frequently review pull requests. Our learning-to-rank approach achieves better performance than the baseline and the CoreDevRec approach in these two projects. More specifically, our learning-to-rank approach reaches 51.8%, 74.8% and 82.5% for Top-1, Top-3 and Top-5 recommendations respectively for the tensorflow project. For swift, our approach reaches 46.8%, 69.3% and 81.7% for Top-1, Top-3 and Top-5 recommendations. In terms of MAP and MRR, our approach outperforms CoreDevRec as well.

For bitcoin, electron, node.js, react and rails, their baseline Top-1 majority classes are less than 40%, their Top-3 majority classes are less than 70%. These projects have a small group of developers taking charge of the review work. In these projects, our learning-to-rank approach outperforms the baseline and the CoreDevRec approach. For example, the baseline Top-5 majority classes are 62.84%, 65.76% and 63.23% for bitcoin, node.js and rails respectively. The Top-5 accuracies of CoreDevRec for these three projects are 73.79%, 72.4% and 81.8%. Our learning-to-rank approach achieves the Top-5 accuracy of 81.8%, 76.2% and 90.6% in these three projects. For all these five projects, our learning-to-rank approach achieves higher MAP and MRR than the CoreDevRec approach.

For opencv, keras, pandas, scikit-learn and jekyll, their baseline Top-1 majority classes are larger than 50%. More than half of the pull requests were reviewed by a single developer. This shows that there is a developer who takes charge of much of the review work in each of these project. Again, for these projects except keras, our learning-to-rank approach outperforms the baseline and the CoreDevRec approach. For keras, CoreDevRec has a higher Top-1 accuracy and higher MRR value. But learning-to-rank has higher Top-3, Top-5 and MAP values than CoreDevRec in keras.

For all the twelve projects, our learning-to-rank approach outperforms the baseline in terms of Top-1, Top-3 and Top-5 recommendations. Our learning-to-rank approach has higher MAP values than CoreDevRec for all these projects.

Overall, our approach recommends the suitable reviewers within the top-1 recommendation for over 80% of the pull requests in opencv and jekyll. It recommends the suitable reviewers within the top-5 recommendations for over 90% of the pull requests in opencv, keras, pandas, scikit-learn, jekyll and rails. Our approach outperforms the baseline and CoreDevRec.

F. FEATURE SELECTION

The results reported in the previous subsection are obtained using all 14 proposed features. However, when used in conjunction with other features, some features may be redundant. Some features may also be irrelevant to the task. In this subsection we seek to justify these features' utility for this task. More exactly, a feature selection technique is applied in order to answer two research questions:

1) Can feature selection selects a subset of features that give better performance than using all the features?
2) What features are more important for the reviewer recommendation task?

In order to answer the two research questions, we perform feature selection by using the greedy backward elimination algorithm [18], [19]. According to the algorithm, it greedily removes a feature from the current feature set at each iteration to maximize the MAP on the testing dataset. Then it returns the feature set that achieves the best MAP across all iterations.

First, we use the splitting strategy described above to split a dataset into K folds. Then we set $D_{train} = fold_k$ and $D_{test} = fold_{k+1}$ for all folds $1 \leq k \leq K-1$ when running the greedy backward feature elimination algorithm. We then compute the MAP by pooling together the results from $K-1$ testing folds: $fold_2$ to $fold_K$.

1) ANSWER TO QUESTION 1

We only performed feature selection on the larger size projects including tensorflow, bitcoin, electron, swift, node.js and pandas. We applied the Mann-Whitney U Test [20] to test if there exists a significant difference between the MAP obtained using a selection of features and the MAP obtained using all the features. Table 3 shows the feature selection results. Results show that applying feature selection achieves

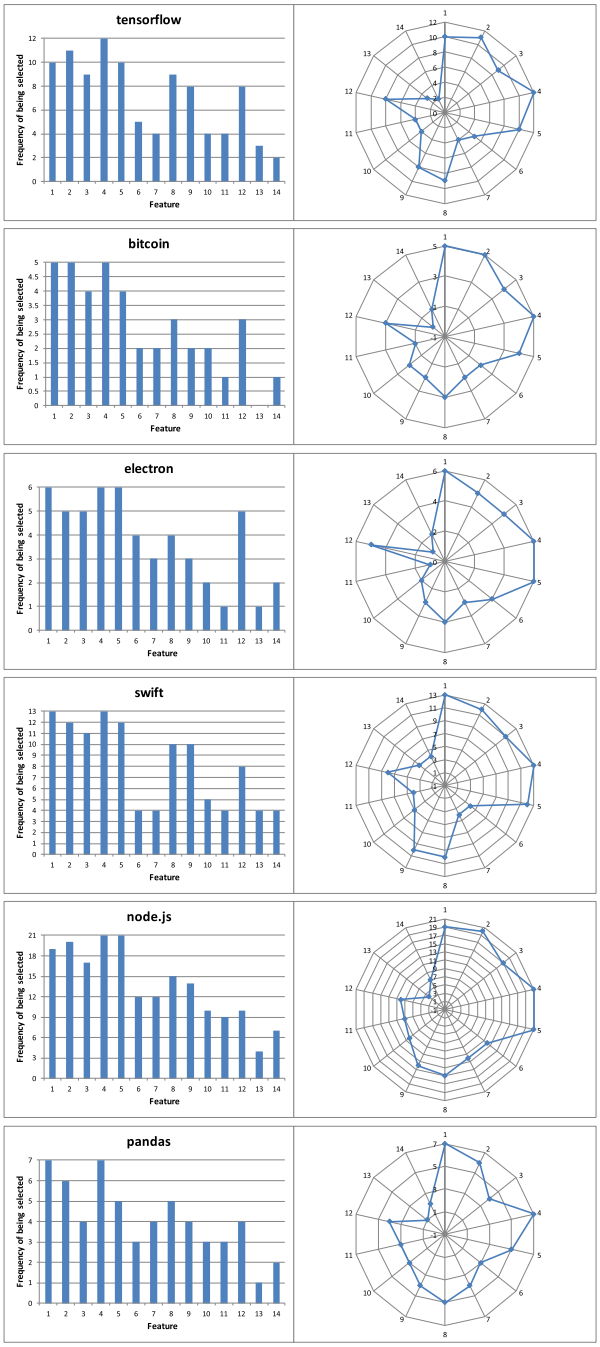

FIGURE 4. Feature selection histograms.

better performance in terms of MAP for all six projects. However, the $p - value$ in Table 3 are all greater than 0.7, which means that there is no a significant difference in terms of MAP between using feature selection and using all features.

Hence, to answer the first question, when compared with using all the features, automatic feature selection can select a subset of features but overall achieves similar performance.

2) ANSWER TO QUESTION 2

Figure 4 shows a histogram for each project. Each histogram shows the number of test folds that a feature was selected.

TABLE 3. MAP comparison: Feature selection versus all features.

Project	# of test folds	MAP across all test folds		p value
		Feature Selection	All Features	
tensorflow	12	0.652	0.643	0.738
bitcoin	5	0.531	0.525	0.746
electron	6	0.620	0.611	0.881
swift	13	0.588	0.580	0.837
node	21	0.437	0.429	0.792
pandas	7	0.792	0.788	0.843

The corresponding radar charts are shown on the right hand side. Along the circumference are different features. The radius indicates the number of times that each feature is selected. Take the tensorflow project for example, there are $K = 12$ test folds. We select a feature set for each of the 12 test folds. Figure 4 shows that features ϕ_4 is selected in all 12 folds, whereas feature ϕ_{14} is selected only twice.

As shown in Figure 4, feature ϕ_4 and ϕ_5, which measures the number of pull requests that were sent by the requester and reviewed by a developer, are the most important features for all six projects. Feature ϕ_4 is selected in all the test folds across all six projects. Feature ϕ_1, which measures the file path similarity between files changed in a pull request and files modified by a developer, is another important features. Feature ϕ_1 is selected in all test folds for four projects. Feature ϕ_2, which measures the file path similarity between files changed in a pull request and files reviewed by a developer, helps on all projects as well. Feature ϕ_3, which measures the lexical similarity between the new pull request and old pull requests reviewed by a developer, although less important, is selected in many test folds for all projects except pandas. Feature ϕ_8 and ϕ_9, which count the number of pull requests reviewed by a developer, are useful for tensorflow and swift. Feature ϕ_{12}, which measures the recency of a developer's review activity, is useful for electron.

To summarize the answer to the second question, feature phi_1, ϕ_2, ϕ_4 and ϕ_5 are the most important features. Feature ϕ_3, ϕ_8, ϕ_9 and ϕ_{12} also provides complementary information that helps improve the ranking performance for the reviewer recommendation task.

VI. CONCLUSION

The pull-based development model is an emerging software development paradigm to support distributed development. A developer who wants to contribute to a project can request an integration of the code changes by sending a pull request. Upon receiving a pull request, the development team will review the changes and make a decision to either accept or reject the changes. In this work, we introduced a learning-to-rank approach to emulate the reviewer recommendation process employed by the development team. Our approach uses a ranking model to rank all the reviewer candidates for a given pull request. The ranking model leverages fourteen features to characterize useful relationships between a pull request and reviewer candidates. Experimental evaluations on twelve open-source projects show that our approach can recommend

the suitable reviewers within the top-5 recommendations for over 76% of the pull requests in all the projects. Furthermore, the proposed approach outperforms the baseline and a state-of-the-art approach CoreDevRec.

Feature evaluation experiments employing greedy backward feature elimination show that the most important features are feature ϕ_4 and ϕ_5, which count the number of previously resolved pull requests sent by the requester and reviewed by a developer. The feature ϕ_1, which computes the file path similarity between files changed in the pull request and files previously modified by a developer, is another important feature.

The proposed adaptive ranking approach is general enough to be applied to software projects that contain a sufficient amount of commit history information and a sufficient number of previously resolved pull requests.

In future work, we plan to integrate additional types of features such as features that leverage the comment network [7]. We also plan to perform evaluations of the proposed approach on more software projects.

REFERENCES

[1] G. Gousios, M. Pinzger, and A. V. Deursen, "An exploratory study of the pull-based software development model," in *Proc. 36th Int. Conf. Softw. Eng.* New York, NY, USA: ACM, 2014, pp. 345–355. doi: 10.1145/2568225.2568260.

[2] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *Proc. 36th Int. Conf. Softw. Eng.* New York, NY, USA: ACM, 2014, pp. 356–366. doi: 10.1145/2568225.2568315.

[3] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proc. 37th Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 358–368.

[4] M. L. de Lima Junior, D. Moreira, A. Plastino, and L. Murta, "Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes," *J. Syst. Softw.*, vol. 144, pp. 181–196, Oct. 2018.

[5] D. Moreira, M. L. de Lima Júnior, A. Plastino, and L. Murta, "What factors influence the reviewer assignment to pull requests?" *J. Syst. Softw.*, vol. 98, pp. 32–43, Jun. 2018.

[6] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration," in *Proc. 21st Asia–Pacific Softw. Eng. Conf.*, vol. 1, Dec. 2014, pp. 335–342.

[7] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" *Inf. Softw. Technol.*, vol. 74, pp. 204–218, Jun. 2016. doi: 10.1016/j.infsof.2016.01.004.

[8] M. M. Rahman, C. K. Roy, and J. A. Collins, "CORRECT: Code reviewer recommendation in GitHub based on cross-project and technology experience," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 222–231.

[9] J. Jiang, Y. Yang, J. He, X. Blanc, and L. Zhang, "Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development," *Inf. Softw. Technol.*, vol. 84, pp. 48–62, Apr. 2017. doi: 10.1016/j.infsof.2016.10.006.

[10] M. L. de Lima Júnior, D. M. Soares, A. Plastino, and L. Murta, "Developers assignment for analyzing pull requests," in *Proc. 30th Annu. ACM Symp. Appl. Comput.* New York, NY, USA: ACM, 2015, pp. 1567–1572. doi: 10.1145/2695664.2695884.

[11] J. Jiang, J.-H. He, and X.-Y. Chen, "CoreDevRec: Automatic core member recommendation for contribution evaluation," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 998–1016, Sep. 2015. doi: 10.1007/s11390-015-1577-3.

[12] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge Univ. Press, 2008.

[13] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 141–150.

[14] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for It: Determinants of pull request evaluation latency on GitHub," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 367–371.

[15] E. M. Voorhees, "The TREC-8 question answering track report," in *Proc. TREC*, Nov. 1999, pp. 77–82.

[16] T. Joachims, "Training linear SVMs in linear time," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*. New York, NY, USA: ACM, Aug. 2006, pp. 217–226. doi: 10.1145/1150402.1150429.

[17] A. Joachims, "Optimizing search engines using clickthrough data," in *Proc. KDD*, Jul. 2002, pp. 133–142.

[18] A. W. Whitney, "A direct method of nonparametric measurement selection," *IEEE Trans. Comput.*, vol. 20, no. 9, pp. 1100–1103, Sep. 1971. doi: 10.1109/T-C.1971.223410.

[19] S. Ramaswamy, P. Tamayo, R. Rifkin, S. Mukherjee, C. H. Yeang, M. Angelo, C. Ladd, M. Reich, E. Latulippe, J. P. Mesirov, and T. Poggio, "Multiclass cancer diagnosis using tumor gene expression signatures," *Proc. Nat. Acad. Sci.*, vol. 98, no. 26, pp. 15149–15154, Dec. 2001. [Online]. Available: http://www.pnas.org/content/98/26/15149.abstract

[20] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: http://dx.doi.org/10.2307/2236101

XIN YE received the Ph.D. degree in computer science from Ohio University. He is currently an Assistant Professor with California State University San Marcos. His main research interests include software engineering and machine learning with a recent focus on mining software repository to support software development. He is also working on applying software engineering principles in educational game design.