Jiang J, He JH, Chen XY. CoreDevRec: Automatic core member recommendation for contribution evaluation. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(5): 998–1016 Sept. 2015. DOI 10.1007/s11390-015-1577-3

CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation

Jing Jiang (蒋 竞), *Member, CCF*, Jia-Huan He (贺佳欢), and Xue-Yuan Chen (陈学渊)

State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China

E-mail: jiangjing@buaa.edu.cn; lightbot.johnson@gmail.com; 429817468@qq.com

Received March 19, 2015; revised July 16, 2015.

Abstract The pull-based software development helps developers make contributions flexibly and efficiently. Core members evaluate code changes submitted by contributors, and decide whether to merge these code changes into repositories or not. Ideally, code changes are assigned to core members and evaluated within a short time after their submission. However, in reality, some popular projects receive many pull requests, and core members have difficulties in choosing pull requests which are to be evaluated. Therefore, there is a growing need for automatic core member recommendation, which improves the evaluation process. In this paper, we investigate pull requests with manual assignment. Results show that 3.2%∼40.6% of pull requests are manually assigned to specific core members. To assist with the manual assignment, we propose CoreDevRec to recommend core members for contribution evaluation in GitHub. CoreDevRec uses support vector machines to analyze different kinds of features, including file paths of modified codes, relationships between contributors and core members, and activeness of core members. We evaluate CoreDevRec on 18 651 pull requests of five popular projects in GitHub. Results show that CoreDevRec achieves accuracy from 72.9% to 93.5% for top 3 recommendation. In comparison with a baseline approach, CoreDevRec improves the accuracy from 18.7% to 81.3% for top 3 recommendation. Moreover, CoreDevRec even has higher accuracy than manual assignment in the project *TrinityCore*. We believe that CoreDevRec can improve the assignment of pull requests.

Keywords core member recommendation, contribution evaluation, pull-based software development

1 Introduction

The pull-based software development is an emerging paradigm for distributed software development[1-2]. Developers pull code changes from other repositories and merge them locally, rather than push changes to a central repository. Various open source software hosting sites, notably GitHub, provide support for pull-based development and allow developers to make contributions flexibly and efficiently.

The pull request process mainly includes three roles in GitHub: contributors, core members, and commenters. Contributors modify codes to fix bugs or improve features. When a set of changes is ready, contributors create pull requests and submit code changes to main repositories. Core members are trusted members of the community. Only experienced and excellent developers are chosen as core members, and they are granted the privilege of directly committing codes to project repositories[3]. Core members evaluate submitted codes and decide whether to merge these code changes into repositories or not. All developers in the community can become commenters of pull requests[4-5]. Commenters freely discuss code changes and leave comments. Their peer review provides suggestions for core members to make decisions[6]. Core members and commenters play different roles in the evaluation process. Core members are necessary and they make final decision of pull requests. Commenters assist core members to make judgement, but they are not essential. Some pull requests are directly evaluated by core members, without any suggestions from commenters.

Regular Paper

Special Section on Software Systems

This work is supported by the National Natural Science Foundation of China under Grant No. 61300006 and the State Key Laboratory of Software Development Environment of China under Grant No. SKLSDE-2015ZX-24.

Ideally, pull requests are assigned to core members and evaluated within a short time after their submission. However in reality, some popular projects receive many pull requests, and core members have difficulties in prioritizing pull requests which are to be merged[7]. Since a great number of code changes must be reviewed before the integration, finding appropriate core members can be a labor-intensive and time-consuming task. There is a growing need to automatically recommend suitable core members for contribution evaluation.

Current assignment of pull requests includes two ways: pull requests can be freely chosen by core members who prefer to review them; pull requests can also be manually assigned to specific core members, who are expected to do evaluation. In this paper, we set out to understand current manual allocation of pull requests in GitHub. In particular, we investigate how often pull requests are manually assigned for evaluation. We collected 18 651 pull requests through GitHub API. We also collected the assignment information to identify pull requests with manual assignment. Results show that 3.2%∼40.6% of pull requests are manually assigned to specific core members. Automatic recommendation is required to shorten the assignment time and improve the evaluation process.

We propose a method called CoreDevRec to solve the core member recommendation problem. We utilize various kinds of features, including file paths of modified codes, relationships between contributors and core members, and activeness of core members. Based on these features, we use support vector machines (SVM) to build our classifier CoreDevRec. Given a new pull request, CoreDevRec generates a list of top k most suitable core members who have the highest probability of evaluating codes. We evaluate CoreDevRec on 18 651 pull requests of five popular projects in GitHub. As there is no previous work of core member recommendation for pull requests, we modify the approach RevFinder[8] to recommend core members, and use it as the baseline approach for comparison. Results show that CoreDevRec achieves accuracy from 72.9% to 93.5% for top 3 recommendation, and achieves mean reciprocal rank from 0.63 to 0.83. In comparison with RevFinder, CoreDevRec improves the accuracy from 18.7% to 81.3% for top 3 recommendation, and improves the mean reciprocal rank from 15.3% to 70.3%. Moreover, CoreDevRec even has higher accuracy than manual assignment in the project *TrinityCore*; CoreDevRec has better performance than manual assignment for top 4 and top 5 recommendation in all projects. Therefore, we believe that CoreDevRec can improve the assignment of pull requests.

The main contributions of this paper are as follows.

• We make an exploratory study on manual assignment.

• We propose CoreDevRec to solve the core member recommendation problem in contribution evaluation. Experimental results show that CoreDevRec achieves good accuracy.

2 Methodology

Before diving into the recommendation of core members, we begin by providing background information about contribution evaluation process in GitHub. Then, we introduce how our datasets are collected. Finally, we report statistics of our datasets.

2.1 Contribution Evaluation Process

GitHub is a web-based hosting service for software development repositories. It has become one of the world's largest open source communities. The typical contribution process includes following steps[4]. First of all, a contributor forks a repository and makes changes to implement new features or fix bugs. The contributor submits a pull request when he or she wants to merge code changes into the main repository. This pull request needs to be evaluated by a core member. The pull request may be chosen by a voluntary core member, who chooses to evaluate this pull request by him/herself. The pull request may also be manually assigned to a specific core member, who is expected to do evaluation. The core member inspects code changes, evaluates potential contributions, and decides whether to merge code changes into the main repository or not. The core member sometimes asks the contributor to make updates and submit new commits for re-evaluation. Commenters sometimes participate in the evaluation process[4-5]. Commenters freely discuss the pull request and suggest improvements. Their suggestions assist the core member to make decision.

Core members and commenters play different roles in the evaluation process. Core members are necessary and they make final decision of pull requests. Commenters provide suggestions and assist core members to make judgements, but they are not essential. Some pull requests are directly evaluated by core members, without any suggestions from commenters. Core members can leave comments and become commenters, but commenters are not necessarily core members.

To illustrate the contribution process, Fig.1 shows an example of the pull request ID 1414 in the project *rails*①. Firstly, a contributor (*crx*) created a pull request and submitted it for evaluation. Secondly, a core member (*josevalim*) was assigned to evaluate this pull request. Thirdly, a commenter (*samlown*) left the suggestion. Finally, another core member (*guilleiguaran*) made a review of modified codes and decided to reject the pull request.

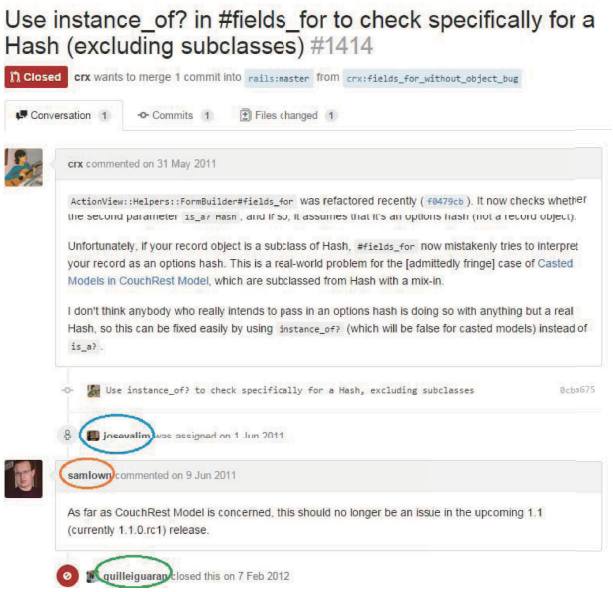

Fig.1. Example of pull request evaluation.

In the above example, the pull request was manually assigned to a core member *josevalim*. But this pull request was finally evaluated by another core member *guilleiguaran*. The manual assignment was incorrect in this pull requests. A developer *samlown* thought that code modification in the pull request was unnecessary. However, *samlown* was not a core member, and he was not allowed to make the decision of this pull request. *guilleiguaran* was a core member, and he decided to reject the pull request. In this pull request, *samlown* is a commenter, and *guilleiguaran* is the core member for evaluation. Previous studies[4-5] recommended reviewers, who were actually commenters. In this paper, we make research on the recommendation of core members, who make final decision of pull requests. Core members and commenters are different. In the above example, *samlown* was the correct recommendation by previous studies[4-5], while *guilleiguaran* is the correct recommendation by our work.

2.2 Data Collection

GitHub provides access to its internal data stores through an API②. It allows us to access a rich collection of open source software (OSS) projects, and provides valuable opportunities for research. We gather information from GitHub API and create datasets of core members and pull requests.

First of all, we choose five popular projects in GitHub. The project *rails* is a framework to create database-backed web applications according to the Model-View-Controller pattern; the project *zf2* is the framework for modern, high-performing PHP applications; the project *scala* is built for the Scala programming language; the project *xbmc* is the open source software media player and entertainment hub for digital media; the project *TrinityCore* is an MMORPG framework based mostly in C++. All these projects provide fundamental functions for software development. In GitHub, starring allows users to keep track of projects that they find interesting; the number of forks is a metric for measuring the active involvement of the developer community. While these metrics are not absolute, they provide good insights into the popularity of a project. In our datasets, all projects have more than 2 000 stars, and the project *rails* even has 23 815 stars. Moreover, all projects have more than 900 forks, and the project *rails* even has 9 897 forks. Projects in our datasets are popular in GitHub.

Next, we collected pull requests of these projects through GitHub API in July 2014. We sent queries to GitHub API, received its replies, and extracted datasets from project creation time to July 2014. For each pull request, we crawled its ID, the contributor who submitted it, the creation time, comments, the number of commits, the number of files modified, the number of added lines, and the number of deleted lines. Each pull request included paths of modified files, which were also collected in our datasets. Furthermore, we collected the close time and the developer who closed the pull request. The pull request could be closed by the contributor or a core member. We ignored pull requests closed by their contributors, because their final decisions were not made by core members and do not need core member recommendation. In GitHub, the pull request can

①https://github.com/rails/rails/pull/1414, July 2015.
②http://developer.github.com/v3/, July 2015.

be manually assigned to a specific core member for evaluation. We crawled information about whether the pull request was manually assigned, who it was assigned to and when it was assigned. This information is useful to understand current manual assignment of pull requests.

GitHub provides an API to return a list of all available assignees to which issues may be manually assigned③. These assignees include the owner and collaborators, namely core members in the project. In GitHub, pull requests belong to a special kind of issues. We collected core member lists through this API.

GitHub is a social coding site[9], and allows users to attract followers. Users build social connections and follow interesting developers[10]. Tsay *et al.* found that pull request acceptance was influenced by social connections between core members and contributors[2]. Social relationships may be useful in core member recommendation. Therefore, we also collected follower and following relationships in our datasets.

2.3 Basic Statistics

Finally, we report statistics of our datasets in Table 1. The project *rails* has 8 333 pull requests, which is a large burden for 44 core members. These pull requests should be carefully assigned to correct core members, so as to take some load off core members' backs. Other projects also face the pull request assignment problem.

Table 1. Basic Statistics of Projects

Project Owner	Project Name	# Pull Requests	# Core Members	% of Manually Assigned Pull Requests	Date of First Pull Request
rails	*rails*	8 333	44	3.2	2010/9/2
zendframework	*zf2*	4 024	13	40.6	2010/9/4
scala	*scala*	2 603	19	19.9	2011/12/1
xbmc	*xbmc*	2 150	73	15.6	2011/1/5
TrinityCore	*TrinityCore*	1 541	35	7.2	2010/12/31

Note: # stands for "Number of"; % stands for "Percentage".

In GitHub, commenters provide suggestions and assist core members to make judgement. We wonder whether commenters are necessary in the contribution evaluation. In every pull request in the project *rails*, we compute the number of comments. We also calculate this metric for other projects. Fig.2 shows cumulative distribution function (CDF) of results. In the project *rails*, 29.1% of pull requests have no comments. 65.3% of pull requests have less than three comments, and only 34.7% of pull requests have more than four comments. It shows that a part of pull requests are directly evaluated by core members, without any suggestions from commenters. Other projects have similar results, and commenters are not essential in contribution evaluation.

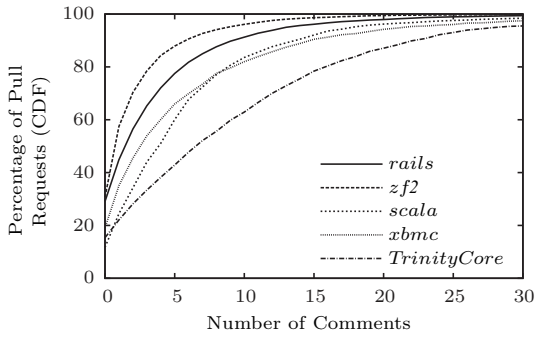

Fig.2. Number of comments.

3 Analysis of Manual Assignment

In GitHub, some pull requests are manually assigned to core members, who are expected to do evaluation. Little is known about manual assignment in GitHub. Therefore, we set out to investigate how often pull requests are manually assigned for evaluation.

As described in Subsection 2.2, our datasets include the information about whether a pull request is manually assigned, and who it is assigned to. Based on these datasets, we compute the percentage of manually assigned pull requests and plot results in Table 1. The project *zf2* has 40.6% of pull requests manually assigned to specific core members. Projects *rails*, *scala*, *xbmc*, and *TrinityCore* have 3.2%, 19.9%, 15.6%, and 7.2% of manually assigned pull requests, respectively. 3.2%∼40.6% of pull requests are manually assigned to specific core members. Results show that a part of pull requests need manual allocation.

We take a further step and explore how the percentage of manually assigned pull requests evolves over

③https://developer.github.com/v3/issues/assignees/, July 2015.

time. For each project, we compute the percentage of manually assigned pull requests each year after the first pull request. As shown in Table 1, the first pull request in the project *rails* was created in September 2010. We collected our datasets in July 2014. The interval time between the first pull request and data collection is less than four years. Therefore, we report statistics of three years for the project *rails*. We also report statistics of three years for projects *zf2*, *xbmc* and *TrinityCore*. The first pull request in the project *scala* was created in December 2011, and we report statistics of two years for the project *scala*.

Fig.3 shows the evolution trend of manually assigned pull requests. In the project *rails*, the percentage of manually assigned pull requests gradually increases over time. In the project *zf2*, none of pull requests are assigned in the first year. In the second year, 14.9% of pull requests are manually allocated to specific core members. In the third year, the percentage of assigned pull requests is even as high as 70.6%.

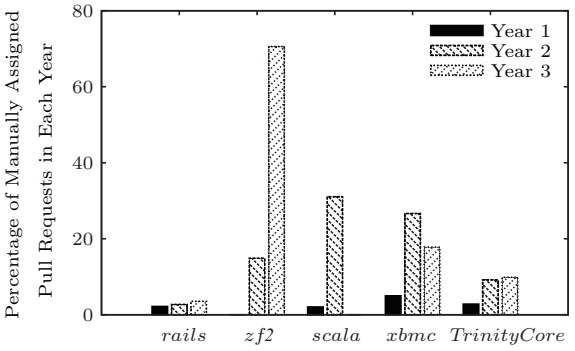

Fig.3. Evolution of the percentage of manually assigned pull requests.

Pull request assignment becomes more and more popular in the project *zf2*. In projects *scala* and *TrinityCore*, we observe similar growth trend of manually assigned pull requests. This is probably because with the evolution of projects, more pull requests are submitted and more developers become core members, which makes the pull request allocation more complex. The automatic recommendation is required to save the assignment time, and accelerate the evaluation process.

4 CoreDevRec: An SVM-Based Method to Recommend Core Members

In this section, we describe our method CoreDevRec to solve the core member recommendation problem. Fig.4 presents the overall framework of CoreDevRec. The entire framework contains two phases: a model building phase and a prediction phase. In the model building phase, our goal is to build a model from historical pull requests and their evaluators. In the prediction phase, the model is used to recommend core members.

Our framework firstly collects various fields from a set of training pull requests with known core members. As described in Subsection 2.2, we crawled modified file paths; we collected social related information, such as follower and following relations; we also collected historical records of pull requests, which could be used to calculate activeness of core members. Next, we extract path features, relationship features, and activeness features from crawled information. We describe detailed definitions and why we choose these features in Subsections 4.1, 4.2 and 4.3 respectively. To check whether features are sufficiently independent, we leverage Spearman's rank correlation coefficient (ρ)[11]. We

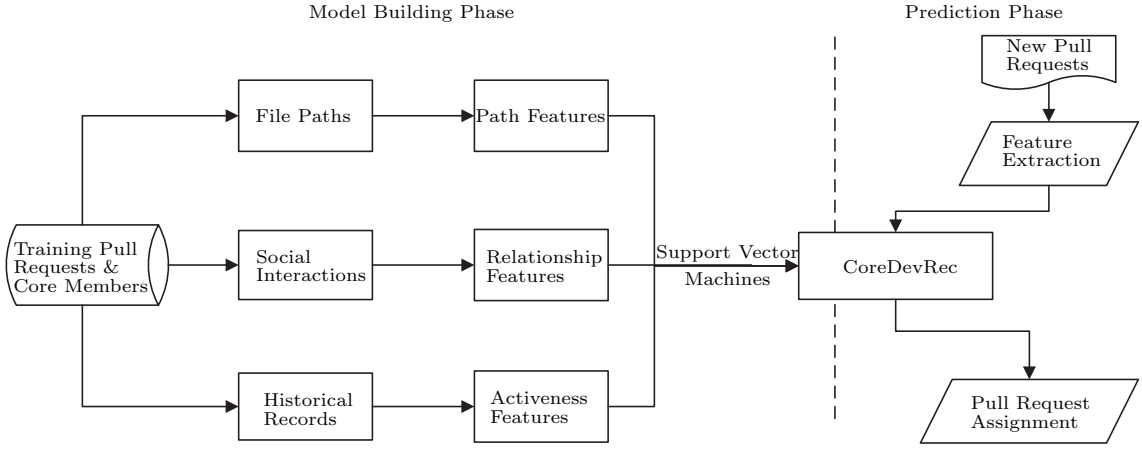

Fig.4. Overall framework for our method CoreDevRec.

set a threshold of $\rho = \pm 0.8$, above which we eliminate features. This statistical analysis ensures the independence of features. According to pull request features and their evaluators, we use support vector machine learning techniques to build our method CoreDevRec, as described in Subsection 4.4.

In the prediction phase, we use CoreDevRec to predict whether a pull request is likely to be evaluated by a specific core member. For each new pull request, we extract values of features, put these values to CoreDevRec, and predict the probability of evaluating pull requests by core members. Core members with the highest probability are recommended.

4.1 Path Features

As described in Subsection 2.2, we collected paths of modified files of pull requests. Directory distance between files was studied in latent social structure[12] and focus-shifting patterns of OSS developers[13]. Initial study[8] observed that files located in similar paths would be reviewed by similar experienced code reviewers. Therefore, we utilize the similarity of modified file paths to recommend core members. Different from the initial study[8], we compute the path similarity by machine learning techniques, instead of string comparison techniques. The detailed steps are as follows.

First of all, we transfer modified file paths into multiple strings. For each file path, we extract consecutive substrings from the beginning of the path. We also take pull request ID 1414 in the project *rails* as an example. This pull request has one modified file path, namely "actionpack/lib/action_view/helpers/form_helper.rb". For this file path, we extract five consecutive substrings from the beginning of the path, including "actionpack", "actionpack/lib", "actionpack/lib/action_view", "actionpack/lib/action_view/helpers", "actionpack/lib/action_view/helpers/form_helper.rb". Some pull request has several modified file paths, and the pull request's path substrings include substrings extracted from all modified file paths.

Secondly, we use path substrings to build path features. Each path substring is one path feature. All unique path substrings of pull requests in the training set construct the path vocabulary. The number of path substrings in the vocabulary is the length of path features. We use term frequency-inverse document frequency (tf-idf), and assign a weighting to path substring s in pull request pr as follow:

$$weight_{pr,s} = tf_{pr,s} \times \left(\log \left(\frac{NT}{df_s} \right) + 1 \right),$$

where $tf_{pr,s}$ is the number of times that substring s occurs in pull request pr, df_s is the number of pull requests that contain path substring s, and NT is the number of pull requests. $weight_{pr,s}$ is the weight of substring s in pull request pr, namely the value of path feature s.

If two pull requests have similar file paths, their path features are also similar. We use support vector machines to identify pull requests with similar modified file paths. Details are described in Subsection 4.4. Note that we delete path substrings which only appear in one pull request. The deletion decreases the size of vocabulary, and reduces the running time of machine learning techniques.

4.2 Relationship Features

According to the previous work[2], core members prefer to accept codes, which are submitted by their followers. Relationships between contributors and core members are important factors in the code evaluation. In order to recommend suitable core members, we also consider relationships between contributors and core members. We use four features to present their social distance and prior interactions as follow.

Follower Relation. Different from traditional OSS platforms, GitHub is a social coding site[9], and allows developers to build social relationships. The contributor decides to follow the core member, when the contributor is interested in the core member's activities. The $follower\_relation$ is a dichotomous variable indicating whether the contributor is the follower of the core member or not. We use this feature as a proxy of the social closeness between the contributor and the core member.

Following Relation. It is a dichotomous variable indicating whether or not the core member follows the contributor. The following relation shows the core member's interest towards the contributor.

Prior Evaluation. The contributor *co* submitted several pull requests before, and some of them might be evaluated by the core member *cm*. The *prior_evaluation* feature is the total number of the contributor *co*'s pull requests which was evaluated by the core member *cm* before. It describes whether the core member *cm* likes to evaluate pull requests submitted by the contributor *co*.

Recent Evaluation. The feature *recent_evaluation* is the number of contributor *co*'s pull requests evaluated by core member *cm* in recent m months. The setting of m is discussed in Section 5. The *recent_evaluation*

feature describes interactions between the contributor and the core member in recent months, while the *prior_evaluation* feature shows all previous interactions between the contributor and the core member.

GitHub is a social coding site, and allows developers to directly build follower or following relations. These direct relationships are not supported in some OSS projects, such as Apache and Mozilla. In other OSS platforms, social relationships can be built by mining emails[14], synchronized co-commits[15], version control system[16] and bug tracking system[16]. Some social relationships are similar to follower and following relations, and they can be used to recommend core members.

4.3 Activeness Features

Core members have the great responsibility of code evaluation. However, core members may be inactive or short-term active in the evaluation of code contributions. Due to the principle of voluntary participation, core members always have the freedom to decide their activities and even leave the community[17]. Core members may not always be available, and may choose to leave the project[18]. Therefore, we need to recommend core members who are active and available. Some core members might evaluate similar pull requests before, but they have already left the project and they are not suitable for recommendation. In this subsection, we consider six features to predict activeness of core members.

Total Pulls. In order to measure the project scale, we count the total number of pull requests accumulated in the project. We also consider several features to describe the project scale, including the number of core members and the number of contributors who have submitted pull requests before. We compute Spearman's rank correlation coefficient (ρ) between features. We observe that ρ between any two features is more than 0.8. The number of core members and the number of contributors are both highly correlated with the total number of pull requests. Therefore, the total number of pull requests is a representative feature, and we only use this feature to describe the project scale in our method.

Evaluate Pulls. In order to measure the liveness of core members, we consider the total number of pull requests evaluated by the core member before, namely the $evaluate\_pulls$ feature.

Recent Pulls. The feature $recent\_pulls$ is the total number of pull requests evaluated by the core member in recent m months. The $recent\_pulls$ feature describes the core members' recent activeness, while the $evaluate\_pulls$ feature shows the core member's activeness after he or she joins the project.

Evaluate Time. After the contributor submits the pull request, it takes some time for a core member to evaluate the contribution and give feedback. The feature $evaluate\_time$ is computed as the average interval time between the pull request submission and the core member evaluation in recent m months. It describes whether the core member evaluates pull requests quickly or slowly. Note that the core member may take several actions towards a pull request, such as leaving comments and then closing the pull request. We use the close action in this feature. We do not consider the interval time between the pull request submission and other activities of the core member, because it is highly correlated with the $evaluate\_time$ feature.

Latest Time. We measure the close time of the latest pull request evaluated by the core member. In experiments, $latest\_time$ is computed as the interval time between the close time of the latest evaluated pull request and the start time of the test set. We use the start time of the test set here, because training set includes all core members' evaluation made before this time. The small $latest\_time$ means that the core member is recently active in evaluating pull requests.

First Time. In order to measure the core member's age in the project, we define the $first\_time$ feature, which is the interval time between the close time of the first evaluated pull request and the start time of the test set. It measures whether the core member is new or old in the project. The age of the core member in GitHub can be measured by the interval time between the core member's registration time in GitHub and the start time of the test set. We compute the Spearman's rank correlation coefficient. We find that the core member's age in the project is strongly correlated with the core member's age in the GitHub. Therefore, the $first\_time$ feature describes the core member's age in both the project and GitHub.

4.4 CoreDevRec

The final step is to use machine learning techniques, build CoreDevRec, and recommend core members for contribution evaluation.

We use the vector space model to represent each pull request as a weighted vector. Each feature is an element in the vector. For a pull request pr submitted

by the contributor *co*, we firstly compute values of NV path features in the vector. NV is the number of substrings in the path vocabulary here. Secondly, we compute the relationship features between the contributor *co* and any core member *cm* who ever evaluates pull requests in the project. It ensures that candidates include all core members who have joined the project and have the right to make evaluation. We consider four relationship features for every core member. If there are NC core members, the vector has $4 \times NC$ elements about relationships. Thirdly, activeness features of any core member *cm* are included in the vector. Since we consider six activeness features for each core member, the vector has $6 \times NC$ elements about activeness. Finally, we combine all features together to generate a large vector, which includes $NV + 4 \times NC + 6 \times NC$ elements for each pull request.

For each pull request in the training set, we know the core member who really evaluates the pull request. Each core member is considered as a category, and the pull request belongs to the category of its evaluator. We run training datasets and build the CoreDevRec classifier through support vector machines (SVM)[19]. We implement CoreDevRec on top of the tool Weka④. The SVM classifier is a supervised classification algorithm that finds a decision surface that maximally separates the classes of interest. The SVM classifier assigns labels (in our case: the core member) to a data point (in our case: a pull request) with a certain likelihood. We use the SVM classifier to get the likelihood that the pull request *pr* will be assigned to a specific core member *cm*. Core members with the highest probability are recommended in OSS projects.

Several machine learning algorithms are known to perform well and have been used in previous work involving prediction models[1,20]. In Subsection 5.7, we run each dataset through four classification algorithms, namely support vector machines, naive Bayes, decision tree C4.5, and Ripper[21]. We do not perform any additional tuning to these classification algorithms. Results show that the SVM classifier performs better than the other classifiers. Therefore, we choose the SVM algorithm to build our method CoreDevRec.

5 Evaluation

In this section, we evaluate our method CoreDevRec. The experimental environment is a Windows server 2012, 64-bit, Intel® Xeon® 1.90 GHz server with 24 GB RAM. We first present our experimental setup, evaluation metrics, and research questions (Subsections 5.1, 5.2, and 5.3 respectively). We then present our experimental results that answer these research questions (Subsections 5.4∼5.8).

5.1 Experimental Setup

In order to simulate the usage of methods in practice, previous studies[22-23] chronologically sort all bug reports, divide these bug reports into 11 folds, and perform experiments in 10 rounds. This experimental setting is suitable for methods which are not time-sensitive. We also sort pull requests in chronological order of creation time. However, we do not divide pull requests into 11 folds. Instead, pull requests created in the same month are put into a group, and the whole datasets are divided into N groups. N depends on the interval time between the first pull request and the data collection. For example, the project *rails*' first pull request was created in September 2010. We collected datasets in July 2014, which was 46 months after the first pull request. Therefore, we divide the project *rails*' pull requests into 46 non-overlapping frames. Our method CoreDevRec utilizes relationship features and activeness features, which are both real-time. If we divide pull requests into 11 folds, pull requests in the same fold may be created in completely different time, and we cannot accurately compute their activeness or relationship features.

Next, we use divided frames to build training set and test set. As show in Fig.5, the validation process is proceeded as follows: first of all, the training set is built by pull requests evaluated between month 1 and month m. The test set is built by pull requests created in month $m + 1$. Then in round 2, we build a training set using pull requests created between month 1 and month $m + 1$, and build a test set using pull requests created in month $m + 2$. We use the similar way to build training set and test set for each month. Finally, in round $N - m$, the training set is built by pull requests created between month 1 and month $N - 1$. The test set is built by pull requests created in month N. We use the training set and the test set to compute the performance of CoreDevRec in each round, and then compute the average performance value of all rounds. This setup ensures that only past pull requests are used to predict pull requests submitted in the future. In the initial stage, some projects have no pull

④http://www.cs.waikato.ac.nz/ml/weka/, July 2015.

requests and the corresponding set is null. For example, in the project *rails*, no pull requests are submitted in month 5 after its creation. We ignore these rounds with null training or test tests, and do not make any prediction.

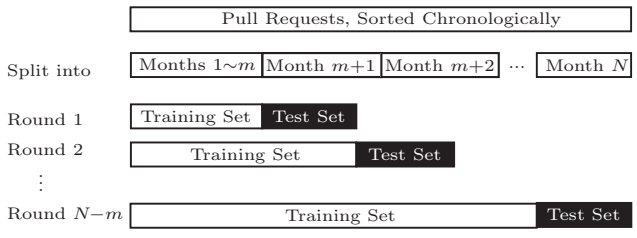

Fig.5. Experimental setup.

For each pull request, modified file paths are extracted to build path features. The core member who really evaluates the pull request is extracted as a label for the classifier. Relationship features and activeness features are computed monthly. It ensures that features are updated, and the computation overhead is acceptable. More specifically, we build the contributor set for each month, including all developers who submit pull requests in the month. We also build the core member set for each month, including all core members who have evaluated pull requests before. It ensures that these core members have already joined the project and made evaluation of code contributions. Every core member is the candidate for evaluating pull requests. Therefore, we compute relationship features for pairs between any contributor and any core member in corresponding sets. For any core member in the set, we also compute activeness features to judge whether he or she is still active in evaluating pull requests.

Three features, *recent_evaluation*, *recent_pulls*, and *evaluate_time*, depend on the number of recent months m. By default, we set m as 1, and compute recent activities of core members in recent one month. In Subsection 5.8, we explore how different parameter settings of m influence the performance of CoreDevRec. Results show that CoreDevRec always has the best performance when m is set as 1.

5.2 Evaluation Metrics

In order to evaluate our method, we use the top k accuracy, $Gain_k$, and the mean reciprocal rank (MRR). These metrics are commonly used in recommendation approaches for software engineering[8,24-26].

We evaluate experiments with the accuracy of top k predicted core members, as described in initial study[24]. The definition of the top k accuracy is as follows:

$$Accuracy_k = \frac{\sum_{pr \in PR} IsCorrect(pr)}{|PR|}, \quad (1)$$

where k is the number of recommended core members and PR is the test set of pull requests. $|PR|$ is the number of pull requests in the test set. If one of top k recommended core members really evaluates the pull request, the prediction is correct and the $IsCorrect(pr)$ function returns value of 1; otherwise, the prediction is incorrect and the $IsCorrect(pr)$ function returns value of 0. The accuracy describes the percentage of pull requests which are correctly assigned to core members. Inspired by the previous study[25], we choose k to be 1, 2, 3, 4, and 5 in the experiments.

According to the previous work[8], mean reciprocal rank (MRR) measures the average value of reciprocal ranks of correct core members in a recommendation list. The definition of MRR is as follows:

$$MRR_k = \frac{\sum_{pr \in PR} \frac{1}{rank(candidates(pr))}}{|PR|},$$

where k is the number of recommended core members and PR is the test set of pull requests. $rank(candidates(pr))$ returns the value of the first rank of actual core member in the recommendation list $candidates(pr)$. If the recommendation list does not include the actual core member, the value of $\frac{1}{rank(candidates(pr))}$ is 0. Ideally, the method with the perfect ranking should achieve the MRR value of 1.

In order to compare two methods, we define the gain to compare how method 1 outperforms method 2. As described in the initial study[22], the gain for accuracy and MRR is defined as follows:

$$Gain_{\text{accuracy}} = \frac{(Accuracy_k(1) - Accuracy_k(2))}{Accuracy_k(2)}, \quad (2)$$

$$Gain_{\text{MRR}} = \frac{(MRR_k(1) - MRR_k(2))}{MRR_k(2)}, \quad (3)$$

where $Accuracy_k(1)$ and $MRR_k(1)$ are top k performance for method 1, and $Accuracy_k(2)$ and $MRR_k(2)$ are top k performance for method 2. If the gain value is above 0, it means method 1 has better accuracy than method 2; otherwise, method 2 has better prediction results.

5.3 Research Questions

We are interested in answering following research questions.

*RQ*1: What is the performance of CoreDevRec? We propose CoreDevRec to find appropriate core members for contribution evaluation. We aim to evaluate the performance of our method in terms of accuracy and MRR. The better performing method can improve the assignment of pull requests.

*RQ*2: What is the performance of CoreDevRec in comparison with RevFinder? How much improvement can it achieve over the method proposed in [8]?

In the previous work[8], Thongtanunam *et al.* proposed a method RevFinder to recommend reviewers for submitted patches. As there is no previous work of core member recommendation for pull requests, we modify RevFinder to recommend core members, and use it as the baseline approach for comparison.

*RQ*3: What is the performance of CoreDevRec in comparison with manual assignment? How much improvement can it achieve over the manual assignment?

Some pull requests are manually assigned in projects. As shown in Fig.1, the manual assignment may be incorrect. We compare the accuracy of CoreDevRec and manual assignment.

*RQ*4: What is the benefit of SVM algorithm in core member recommendation?

Several machine learning algorithms are known to perform well and have been used in previous work involving prediction models[1,20]. To our knowledge, there are few studies that investigate the effectiveness of machine learning algorithms for the task of recommending core members. CoreDevRec uses SVM to retrieve dominant features and build the prediction model. We would like to investigate whether SVM algorithm could achieve better performance than some other machine learning algorithms. To answer this question, we run each dataset through four classification algorithms, namely SVM, naive Bayes, decision tree C4.5, and Ripper. We compare the performance of these machine learning algorithms in the recommendation of core members.

*RQ*5: Do different numbers of recent months affect the performance of CoreDevRec?

CoreDevRec has a parameter m which describes the number of recent months. By default, we set m as 1, and compute recent activities of core members in recent one month. In this research question, we would like to investigate whether different numbers of recent months affect the performance of CoreDevRec. To answer this research question, we vary the number of recent months from 1 to 6, and compute CoreDevRec's performance.

*RQ*6: How long is the allocation time of different methods?

The allocation time is defined as the interval between a pull request's submission time and its assignment time. We also compare the allocation time of CoreDevRec, RevFinder, and manual assignment.

*RQ*7: What is the performance of CoreDevRec in different rounds?

In experiments, we compute the average value of the accuracy in all rounds. We wonder whether CoreDevRec has similar or different performance in various rounds.

*RQ*8: What is the performance of CoreDevRec with definite social relationships?

Follower and following relations were cross-sectional at the time of data collection (July 2014), rather than at the time when pull requests were submitted. Without knowing exact creation time, follower and following relations are estimated in previous experiments. Though we cannot be certain about follower and following relations before July 2014, these social relationships are definite after July 2014. We collected datasets of pull requests again in November 2014. We use definite social relationships to make recommendation for pull requests submitted between July 2014 and November 2014.

5.4 RQ1: Performance of CoreDevRec

Table 2 shows the accuracy and MRR (mean reciprocal rank) of CoreDevRec by recommending different numbers of candidate core members. When we only recommend one core member for each pull request, the project *rails* has 51.3% of correctly assigned pull requests, and the project *zf2* has 72.3% of correctly assigned pull requests. The accuracy increases as the number of recommended core members grows. CoreDevRec achieves the accuracy from 72.9% to 93.5% for top 3 recommendation. When we recommend top 5 core members, projects *zf2* and *scala* have the accuracy higher than 94%. CoreDevRec achieves MRR from 0.63 to 0.83. Results show that CoreDevRec has high accuracy and MRR.

Table 2. Performance of CoreDevRec

Project	Accuracy (%)					MRR
	Top 1	Top 2	Top 3	Top 4	Top 5	
rails	51.3	70.5	79.6	84.4	87.3	0.67
zf2	72.3	88.4	93.5	95.0	95.8	0.83
scala	57.0	77.5	88.0	92.8	94.8	0.73
xbmc	55.1	70.1	77.0	79.4	81.0	0.67
TrinityCore	49.3	63.8	72.9	77.2	80.2	0.63

5.5 RQ2: Comparison Between CoreDevRec and RevFinder

Thongtanunam *et al.* proposed a method RevFinder to recommended reviewers for submitted patches[8]. Since RevFinder is recent work, we modify it to recommend core members in GitHub. RevFinder is a core-reviewer recommendation method based on the assumption that "files that are located in similar file paths would be managed and reviewed by similar experienced code-reviewers" [8]. We use RevFinder to recommend core members using modified file paths in pull requests. The calculation of RevFinder can be summarized as follows. Given a new pull request, 1) a list of candidates is generated and it includes all core members who ever evaluated pull requests before. 2) The algorithm calculates a similarity score between every past pull request and the new pull request. The score is the average path similarity value of every modified file path in the past and the new pull request. 3) The score of a core member is the sum of similarity scores of past pull requests evaluated by this core member. 4) The candidates who have the highest scores will be recommended as appropriate core members.

Table 3 shows the performance comparison between CoreDevRec and RevFinder. The gain is calculated using (2)∼(3). It measures how CoreDevRec outperforms RevFinder. In project *scala*, CoreDevRec outperforms RevFinder by 175.4% for top 1 recommended core member. Compared with RevFinder, CoreDevRec improves the accuracy from 18.7% to 81.3% for top 3 recommendation. Its top 5 accuracy values outperform those of RevFinder by 38.8%, 11.7%, 17.2%, 26.2%, and 46.9% in projects *rails*, *zf2*, *scala*, *xbmc*, and *TrinityCore*, respectively. CoreDevRec has obviously higher accuracy than RevFinder.

Table 3 also shows the MRR values of CoreDevRec and RevFinder. In comparison with RevFinder, CoreDevRec improves MRR from 15.3% to 70.3%. This indicates that CoreDevRec can recommend correct core members at lower rank than RevFinder does.

We use the receiver operating characteristic (ROC) analysis[27-28], and compare core member rankings obtained by CoreDevRec and RevFinder. ROC curve illustrates the performance of a binary classifier, when its discrimination threshold is varied. As described in (1), k core members with the highest scores are considered as positive examples, and the other core members are considered as negative examples. We vary the threshold k from 1 to the total number of core members. We classify a candidate as a recommended core member (predicted positive) if its rank is less than or equal to k; otherwise, a candidate is not classified as recommended (predicted negative). Therefore, as k increases, both false positive rate (FPR) and true positive rate (TPR) increase[27].

Fig.6 plots ROC curves for CoreDevRec and RevFinder. For a given FPR, the method with larger TPR is considered better. When FPR is 0.2, CoreDevRec has TPR as 0.85, and RevFinder has TPR as 0.62 in the project *rails*. Other projects have similar results. ROC curves further prove that CoreDevRec behaves better than RevFinder in all projects.

We discuss why CoreDevRec outperforms RevFinder. RevFinder only uses the similarity of pre-

Table 3. Performance Comparison Between CoreDevRec and RevFinder

Project	Method	Accuracy (%)					MRR
		Top 1	Top 2	Top 3	Top 4	Top 5	
rails	RevFinder	24.3	40.5	49.1	57.0	62.9	0.42
	CoreDevRec	51.3	70.5	79.6	84.4	87.3	0.67
	Gain	**111.1**	**74.1**	**62.1**	**48.1**	**38.8**	**59.50**
zf2	RevFinder	60.8	70.2	78.8	82.8	85.8	0.72
	CoreDevRec	72.3	88.4	93.5	95.0	95.8	0.83
	Gain	**18.9**	**25.9**	**18.7**	**14.7**	**11.7**	**15.30**
scala	RevFinder	20.7	51.1	53.7	69.2	80.9	0.46
	CoreDevRec	57.0	77.5	88.0	92.8	94.8	0.73
	Gain	**175.4**	**51.7**	**63.9**	**34.1**	**17.2**	**58.70**
xbmc	RevFinder	33.2	46.3	53.7	60.3	64.2	0.48
	CoreDevRec	55.1	70.1	77.0	79.4	81.0	0.67
	Gain	**66.0**	**51.4**	**43.4**	**31.7**	**26.2**	**39.60**
TrinityCore	RevFinder	21.8	32.6	40.2	46.1	54.6	0.37
	CoreDevRec	49.3	63.8	72.9	77.2	80.2	0.63
	Gain	**126.1**	**95.7**	**81.3**	**67.5**	**46.9**	**70.30**

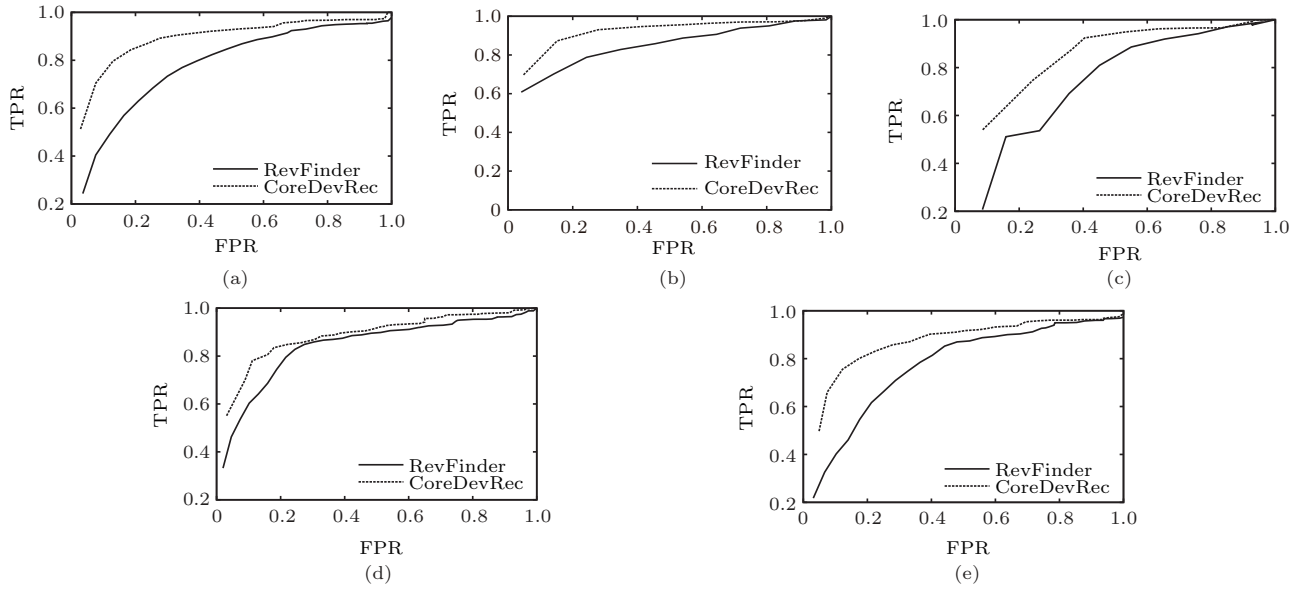

Fig.6. ROC curves for CoreDevRec and RevFinder. (a) Project *rails*. (b) Project *zf2*. (c) Project *scala*. (d) Project *xbmc*. (e) Project *TrinityCore*.

viously evaluated file paths. CoreDevRec not only utilizes file paths, but also considers the activeness of core members and relationships between core members and contributors. In the discussion section of the previous study of [8], Thongtanunam *et al.* mentioned that one threat of RevFinder was the lack of core reviewer retirement information: "It is possible that code reviewers are retired or no longer involve the code review system." CoreDevRec addresses this problem by measuring the activeness of core members. Retired core members have low values of activeness features, and they are excluded from recommendation. Furthermore, CoreDevRec uses relationships between core members and contributors, so as to improve accuracy and MRR of core member recommendation.

5.6 RQ3: Comparison Between CoreDevRec and Manual Assignment

As described in Subsection 2.1, some pull requests are manually assigned to specific core members. In order to get the accuracy of manual assignment, we compute the percentage of pull requests which are really evaluated by manually assigned core members. Then we compare results between our CoreDevRec and manual assignment in Table 4. Only one core member can be manually allocated to a pull request in GitHub. In comparison, we only consider manual assignment when one core member is recommended. Moreover, manual assignment does not provide complete orders of core members. We do not know the first rank of actual core member in the recommendation list. Therefore, we do not compare the MRR value of CoreDevRec and manual assignment.

When the number of recommended core members is 3, CoreDevRec outperforms manual assignment by 12.6%, 40.1%, 17.9%, and 72.3% in projects *rails*, *scala*, *xbmc*, and *TrinityCore*, respectively. CoreDevRec has better accuracy than the manual assignment for top 4 and top 5 recommendation in all projects. In the project *TrinityCore*, CoreDevRec has higher accuracy than manual assignment, when only top 1 core member is recommended; CoreDevRec outperforms manual assignment by 89.6% for top 5 recommendation.

Although CoreDevRec sometimes recommends more core members than manual assignment, CoreDevRec greatly reduces the size of candidate set of core members. CoreDevRec can be used to assist with manual assignment and reduce the allocation time.

5.7 RQ4: Comparison of Different Machine Learning Algorithms

Several machine learning algorithms are known to perform well and have been used in previous work involving prediction models[1,20]. In Fig.4, several machine learning algorithms can be used to retrieve dominant features and build the model for core member recommendation. In this subsection, we investigate the performance of different machine learning algorithms.

Table 4. Accuracy Comparison Between CoreDevRec and Manual Assignment

Project	Method	Accuracy (%)				
		Top 1	Top 2	Top 3	Top 4	Top 5
rails	Manual	70.7	N/A	N/A	N/A	N/A
	CoreDevRec	51.3	70.5	79.6	84.4	87.3
	Gain	−27.4	−0.3	**12.6**	**19.4**	**23.5**
zf2	Manual	94.6	N/A	N/A	N/A	N/A
	CoreDevRec	72.3	88.4	93.5	95.0	95.8
	Gain	−23.6	−6.6	−1.2	**0.4**	**1.3**
scala	Manual	62.8	N/A	N/A	N/A	N/A
	CoreDevRec	57.0	77.5	88.0	92.8	94.8
	Gain	−9.2	**23.4**	**40.1**	**47.8**	**51.0**
xbmc	Manual	65.3	N/A	N/A	N/A	N/A
	CoreDevRec	55.1	70.1	77.0	79.4	81.0
	Gain	−15.6	**7.4**	**17.9**	**21.6**	**24.0**
TrinityCore	Manual	42.3	N/A	N/A	N/A	N/A
	CoreDevRec	49.3	63.8	72.9	77.2	80.2
	Gain	**16.5**	**50.8**	**72.3**	**82.5**	**89.6**

We use different machine learning algorithms to build classifiers, including SVM, naive Bayes, decision tree C4.5, and Ripper[21]. Each classifier assigns labels (in our case: the core member) to a data point (in our case: a pull request) with a certain likelihood. We implement these classifiers on top of the tool Weka. Then we run datasets through four classification algorithms, and plot their performance in Table 5.

In Table 5, SVM has higher MRR than naive Bayes, decision tree C4.5, and Ripper in projects *rails*, *scala*, *xbmc*, and *TrinityCore*. In projects *xbmc* and *TrinityCore*, SVM also has the highest accuracy for all top k recommendation. In the project *rails*, SVM has the highest accuracy for top 2, top 3, top 4 and top 5 recommendation. The accuracy for SVM is 51.3% for top 1 recommendation, which is close to the accuracy of C4.5 algorithm. In projects *zf2* and *scala*, SVM also outperforms other three algorithms on the accuracy for top 2, top 3, top 4 and top 5 recommendation. Although SVM does not have the best performance for

Table 5. Performance Comparison of Different Machine Learning Algorithms

Project	Algorithm	Accuracy (%)					MRR
		Top 1	Top 2	Top 3	Top 4	Top 5	
rails	SVM	51.3	**70.5**	**79.6**	**84.4**	**87.3**	**0.67**
	Naive Bayes	42.9	58.4	67.3	73.4	77.5	0.58
	C4.5	**54.3**	63.9	68.9	71.9	73.9	0.64
	Ripper	52.9	60.2	66.6	71.1	74.6	0.63
zf2	SVM	72.3	**88.4**	**93.5**	**95.0**	**95.8**	**0.83**
	Naive Bayes	67.7	80.4	86.1	90.0	92.5	0.78
	C4.5	75.7	85.5	89.6	91.5	92.3	**0.83**
	Ripper	**76.8**	82.3	85.6	88.5	90.3	**0.83**
scala	SVM	57.0	**77.5**	**88.0**	**92.8**	**94.8**	**0.73**
	Naive Bayes	47.7	69.5	79.6	85.5	87.6	0.66
	C4.5	54.3	70.2	74.9	76.9	79.2	0.67
	Ripper	**58.0**	71.6	79.4	83.6	87.5	0.71
xbmc	SVM	**55.1**	**70.1**	**77.0**	**79.4**	**81.0**	**0.67**
	Naive Bayes	44.8	58.3	65.6	70.3	74.1	0.57
	C4.5	53.4	61.8	65.0	66.9	68.7	0.62
	Ripper	54.9	63.5	67.3	70.4	72.2	0.64
TrinityCore	SVM	**49.3**	**63.8**	**72.9**	**77.2**	**80.2**	**0.63**
	Naive Bayes	30.7	44.3	53.4	59.6	64.6	0.47
	C4.5	45.6	53.2	57.4	60.9	61.9	0.54
	Ripper	47.5	54.7	59.8	62.3	64.7	0.57

top 1 recommendation, its accuracy is close to that of Ripper.

Table 5 shows that SVM performs better than the other machine learning algorithms. Therefore, we choose SVM to build our method CoreDevRec.

5.8 RQ5: Effect of Varying the Number of Recent Months

CoreDevRec has a parameter m which describes the number of recent months. As described in Section 4, three features *recent_evaluation*, *recent_pulls*, and *evaluate_time* depend on the number of recent months m. In this subsection, we investigate the effect of the number of recent months on the performance of CoreDevRec. We vary the number of the recent months m from 1 to 6. Then we plot results of accuracy and MRR in Fig.7.

Fig.7(a) shows the accuracy with the parameter m varying from 1 to 6 in the project *rails*. For top 1, top 2, and top 3 recommendation, the accuracy drops substantially as the number of recent months increases. For top 4 and top 5 recommendation, the accuracy drops slightly with the parameter m varying from 1 to 6. CoreDevRec achieves the highest accuracy when it considers core members' evaluation in recent one month. Fig.7(c) shows that the project *scala* has the similar performance.

Fig.7(d) shows accuracy results of the project *xbmc*. As the number of recent months increases, the line of accuracy declines slightly for top 1 and top 2 recommendation; the line of accuracy remains stable for top 3 and top 4 recommendation. For top 5 recommendation, the accuracy with the setting of $m = 1$ is 81%, and the accuracy with the setting of $m = 5$ is 82.1%. The accuracy with the setting of $m = 1$ is lower than that of the setting of $m = 5$, but the difference is small. The setting of $m = 1$ can achieve good accuracy. Projects *zf2* and *TrinityCore* have similar performance to the project *xbmc*.

Fig.7(f) shows MRR results for five projects. It also shows stable MRR performance for projects *zf2*, *xbmc*, and *TrinityCore*, i.e., there is little difference in the varying number of recent months. In projects *rails* and *scala*, MRR lines drop slightly as the value of m increases. CoreDevRec always has the best performance when m is set as 1. Therefore, m is set as 1 in CoreDevRec by default.

5.9 RQ6: Allocation Time of Different Methods

The aim of CoreDevRec is to shorten the interval time between a pull request's submission and its assignment. In this subsection, we compare the allocation time of RevFinder, manual assignment, and CoreDevRec. For RevFinder and CoreDevRec, the computation of core members immediately begins after the pull request's submission, and the allocation time is computed as the run time of the algorithm. As described

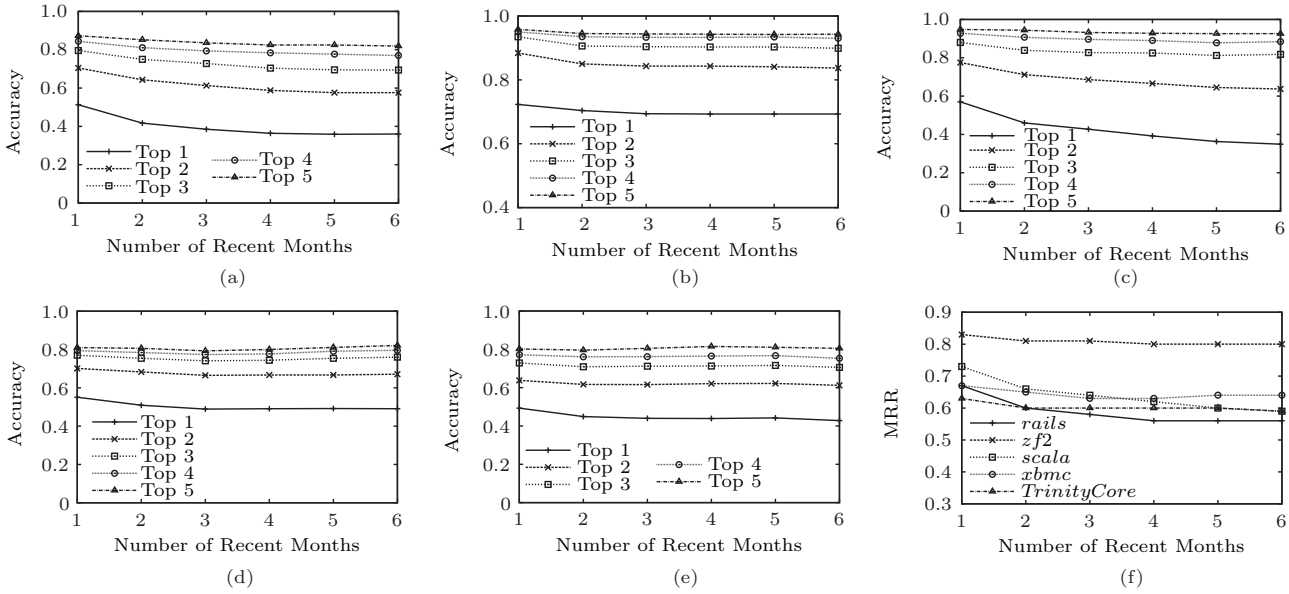

Fig.7. Performance with the number of recent months (m) varying from 1 to 6. (a) Project *rails*. (b) Project *zf2*. (c) Project *scala*. (d) Project *xbmc*. (e) Project *TrinityCore*. (f) Mean reciprocal rank (MRR).

in Subsection 2.2, we crawled information about when a pull request was assigned. For manual assignment, the allocation time is computed as the interval time between the pull request's submission and its manual assignment.

We compute allocation time for pull requests in a project. Table 6 shows the minimum, maximum, average value and median value of allocation time, respectively. In the project *rails*, the median value of allocation time is only 1.7 seconds for CoreDevRec, and it is only 4.2 seconds for RevFinder. In contrast, the median value of allocation time is 99 372.5 seconds for manual assignment. Other projects have similar results. Allocation time is very short for RevFinder and CoreDevRec, while manual assignment has much larger allocation time. This is because manual assignment waits for developers to assign pull requests. CoreDevRec has the allocation time within 3 seconds, which is much shorter than manual assignment.

5.10 RQ7: Performance in Different Rounds

As shown in Fig.5, we compute the performance of CoreDevRec in each round, and then compute the average performance value of all rounds. In this subsection, we investigate the effect of the round on the performance of CoreDevRec. In previous work[23], the accuracy increases over nearly all rounds for the bug triage model. This is because that machine learners have longer time datasets and more available information in the higher numbered rounds. We wonder whether the accuracy and MRR also increase as the number of round grows in our method.

We plot the accuracy and MRR in each round in Fig.8. As described in Subsection 5.1, only five pull requests are created before month 6 in the project *rails*, which does not provide enough information for machine learning. Therefore, we do not compute accuracy results for rounds from 1 to 7.

In the project *rails*, the accuracy does not increase as the round number grows in Fig.8(a). The accuracy even drops greatly in some rounds. The accuracy also sometimes rises and drops as the round number grows in projects *zf2*, *scala*, *xbmc*, and *TrinityCore*. Fig.8(f) also shows the fluctuation of MRR values in all projects. More available training datasets do not increase the prediction performance of CoreDevRec. This is probably because that some features in our method are real-time, and longer time datasets do not provide more useful information for SVM. For example, the activeness of core members mainly depends on their activities in recent months, and longer time datasets do not include more useful information.

5.11 RQ8: Performance with Definite Social Relationships

Follower and following relations were cross-sectional at the time of data collection (July 2014), rather than at the time when pull requests are submitted. Without knowing exact creation time, we cannot be certain about the direction of causality for these latter features using our datasets. But we are sure that these social relationships were established before July 2014. We collected datasets of pull requests again in November 2014. We evaluate the performance of different methods for pull requests submitted between July 2014 and Novem-

Table 6. Comparison of Allocation Time (s)

Project	Method	Minimum	Maximum	Average	Median
rails	RevFinder	0.6	10.8	4.7	4.2
	CoreDevRec	1.1	2.0	1.7	1.7
	Manual	7.0	39 298 305.0	2 903 953.6	99 372.5
zf2	RevFinder	0.3	17.4	5.2	4.2
	CoreDevRec	1.2	1.9	1.5	1.5
	Manual	8.0	54 929 265.0	1 008 702.8	252 849.5
scala	RevFinder	1.0	8.4	3.9	4.0
	CoreDevRec	0.8	1.6	1.2	1.2
	Manual	4.0	4 235 916.0	271 776.7	25 599.0
xbmc	RevFinder	0.1	2.7	0.8	0.5
	CoreDevRec	1.1	2	1.7	1.7
	Manual	5.0	30 596 537.0	1 377 610.4	34 827.0
TrinityCore	RevFinder	0.2	7.1	2.5	1.8
	CoreDevRec	1.4	2.3	1.9	1.9
	Manual	57.0	9 447 707.0	1 040 281.2	90 014.0

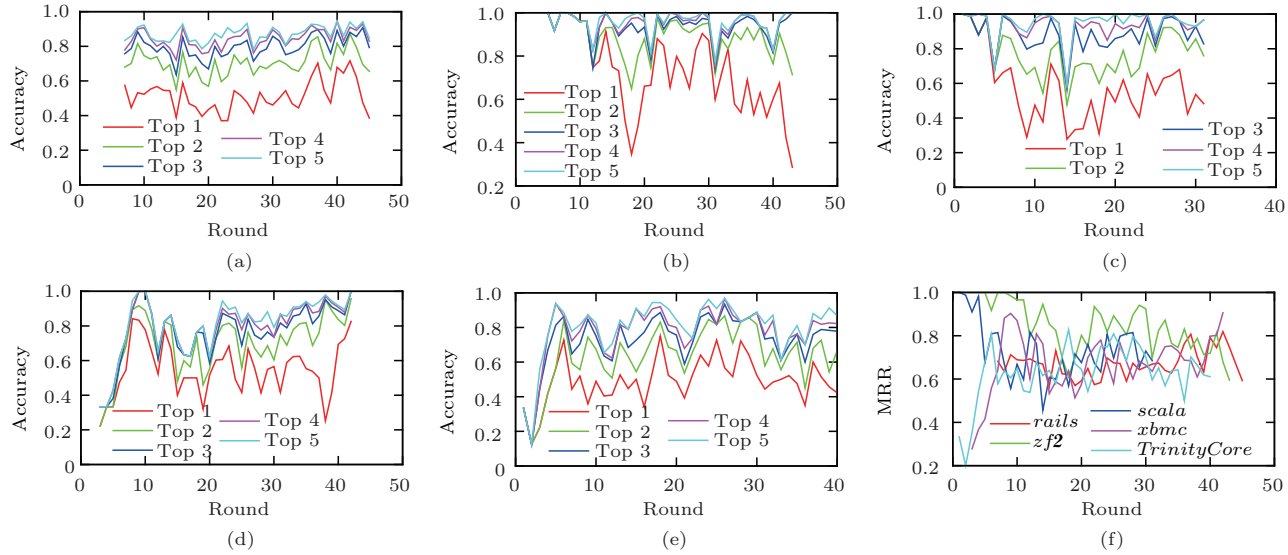

Fig.8. Performance in different rounds. (a) Project *rails*. (b) Project *zf2*. (c) Project *scala*. (d) Project *xbmc*. (e) Project *TrinityCore*. (f) Mean reciprocal rank (MRR).

ber 2014. In this experiment, follower and following relations are definite.

Table 7 shows the accuracy of RevFinder, manual assignment, and CoreDevRec. CoreDevRec has obviously higher accuracy than RevFinder in all projects. In comparison with manual assignment, CoreDevRec has better accuracy for top 1 recommendation in projects *xbmc* and *TrinityCore*. CoreDevRec has better accuracy than manual assignment for top 2 recommendation in project *scala* and top 3 recommendation in project *rails*. Results show that CoreDevRec also achieves good performance, when definite social relationships are used.

6 Threats to Validity

Threats to internal validity relate to experimenter bias and errors. Firstly, we extract features to predict the activeness of core members in future. *follower_relation* and *following_relation* are cross-sectional at the time of data collection, rather than at the time when pull requests are submitted. Without knowing exact values of features, we cannot be certain about the direction of causality for these latter features using our datasets. In order to validate our results, we use definite social relationships to make recommendation, and CoreDevRec also achieves good performance in Subsection 5.11. Secondly, CoreDevRec uses dif-

Table 7. Accuracy (Recommending Core Members for Pull Requests Submitted Between July 2014 and November 2014)

Project	Method	Accuracy (%)				
		Top 1	Top 2	Top 3	Top 4	Top 5
rails	RevFinder	24.6	37.9	42.6	45.6	48.2
	CoreDevRec	48.3	68.5	77.4	83.1	87.6
	Manual	72.0	N/A	N/A	N/A	N/A
zf2	RevFinder	7.1	10.9	11.8	12.5	19.4
	CoreDevRec	35.3	44.6	54.5	90.6	93.9
	Manual	94.8	N/A	N/A	N/A	N/A
scala	RevFinder	13.4	14.5	29.8	44.1	69.8
	CoreDevRec	43.9	74.6	88.3	94.8	97.7
	Manual	57.9	N/A	N/A	N/A	N/A
xbmc	RevFinder	16.4	44.6	50.5	52.6	53.9
	CoreDevRec	47.8	70.8	79.4	83.4	83.7
	Manual	39.4	N/A	N/A	N/A	N/A
TrinityCore	RevFinder	24.2	37.3	39.5	42.6	44.6
	CoreDevRec	38.1	62.7	75.5	79.2	81.0
	Manual	12.5	N/A	N/A	N/A	N/A

ferent kinds of features to recommend core members. Some features may be more important than the others in the recommendation. In future work, we will study the relative significance of features and find important features.

Threats to external validity relate to the generalizability of our study. Firstly, our empirical results are limited to five projects, *rails*, *zf2*, *scala*, *xbmc*, and *TrinityCore*. We cannot claim that the same results would be achieved in other projects. Our future work will focus on the evaluation in other projects to better generalize results of our method. Secondly, our empirical findings are based on open source projects in GitHub, and it is unknown whether our results can be generalized to other OSS platforms. In the future, we plan to study a similar set of research questions from other platforms, and compare their results with our findings in GitHub. Thirdly, the goal of our method is to recommend core members who are likely to evaluate pull requests. In experiments, we mainly evaluate whether one of top k recommended core members really evaluates the pull request. However, we do not consider whether recommended core members are responsible and they make correct decision of pull requests. The quality of evaluation is beyond the scope of this paper, but it is important for the evolution of OSS projects. Correct evaluation not only merges good codes into projects, but also encourages developers to make contribution.

7 Related Work

Several previous studies explored review process of code contribution. Nurolahzade *et al.* discovered that core members were often overwhelmed with many patches they had to review[29]. Rigby *et al.* observed that if modified codes are not reviewed immediately, they are likely not to be reviewed[30]. Rigby and Storey further understood broadcast based peer review in open source software projects[6]. They found that code reviews were expensive, because they needed core reviewers to read, understand and judge code changes. Bosu and Carver made empirical studies to evaluate code review process using a popular open source code review tool in OSS communities[31]. Baysal *et al.* found that nontechnical factors significantly impacted contribution review outcomes[32]. However, none of these studies addresses the problem of automatic recommendation of core members. Moreover, none of them investigates the use of manual recommendation. Different from above studies, we make a study of manual recommendation, and design a method to automatically recommend appropriate core members.

Finding relevant expertise is an important need in collaborative software engineering. We mainly describe developer recommendation in bug triage, change request, and code review.

Firstly, many studies designed approaches to assign bug reports or change requests[26,33-41]. Hossen *et al.* considered source code authors, maintainers, and change proneness to triage change requests[36]. Anvik *et al.* applied a machine learning algorithm and suggested a small number of developers suitable to resolve bug reports[38]. Linares-Vásquez *et al.* utilized source code authorship for assigning expert developers to change requests[33]. Jeong *et al.* used bug tossing history to assign developers to bug reports[34]. Matter *et al.* used a text-based method to identify expertise of developers for bug reports[35]. Core member recommendation is different from above studies. Recommending expertise for bugs or new features is to find suitable developers, who write codes and satisfy requirements. Any developer can be a candidate. Recommending core members for pull requests is to find suitable core members and make evaluation. Only a core member can be a candidate. Furthermore, the core member does not need to write codes, but he or she needs to review modified codes and decide whether to merges codes into projects.

Secondly, we review researches in code review. Yu *et al.* proposed a method to predict relevant commenters of incoming pull requests in GitHub[4-5]. As shown in Subsection 2.3, commenters are not essential in contribution evaluation. Many pull requests are directly evaluated by core members, without any comments from commenters. Core member recommendation is required in contribution evaluation, and it is much different from reviewer recommendation in GitHub. Lee *et al.* proposed a graph-based method to automatically recommend suitable reviewers for patches[42]. Balachandran designed a tool called Review Bot to predict developers who modified related code sections frequently as appropriate reviewers[25]. Thongtanunam *et al.* proposed a method RevFinder to recommend developers who examined files with similar directory paths[8]. These studies recommend code reviewers in traditional open source software platforms, while CoreDevRec recommends core members in a pull-based development platform. Moreover, we modify RevFinder to recommend core members in GitHub. Experiments in Subsection 5.5 show that our method CoreDevRec has better accuracy than RevFinder.

8 Conclusions

In this paper, we empirically investigated pull requests with manual assignment. Results showed that 3.2%∼40.6% of pull requests are manually assigned to specific core members. Automatic recommendation is required to shorten the assignment time and improve the evaluation process.

We proposed CoreDevRec to recommend core members for contribution evaluation in GitHub. CoreDevRec considers different kinds of features and uses SVM to predict suitable core members. We evaluated CoreDevRec on 18 651 pull requests of five popular projects in GitHub. Results showed that CoreDevRec achieves accuracy from 72.9% to 93.5% for top 3 recommendation, and achieves mean reciprocal rank from 0.63 to 0.83. In comparison with RevFinder, CoreDevRec improves the accuracy from 18.7% to 81.3% for top 3 recommendation, and improves the mean reciprocal rank from 15.3% to 70.3%. Moreover, CoreDevRec has better accuracy than manual assignment for top 4 and top 5 recommendation. CoreDevRec has the best performance when SVM is used and the parameter m is set as 1. Therefore, we believe that CoreDevRec can improve the assignment of pull requests.

References

[1] Gousios G, Pinzger M, van Deursen A. An exploratory study of the pull-based software development model. In *Proc. the 36th ICSE*, May 31-June 7, 2014, pp.345-355.

[2] Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In *Proc. the 36th ICSE*, May 31-June 7, 2014, pp.356-366.

[3] Zhou M, Mockus A. What make long term contributors: Willingness and opportunity in OSS community. In *Proc. the 34th ICSE*, June 2012, pp.518-528.

[4] Yu Y, Wang H, Yin G, Ling C X. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *Proc. the 21st APSEC*, December 2014, pp.335-342.

[5] Yu Y, Wang H, Yin G, Ling C X. Reviewer recommender of pull-requests in GitHub. In *Proc. the 30th ICSME*, September 29-October 3, 2014, pp.609-612.

[6] Rigby P C, Storey M A. Understanding broadcast based peer review on open source software projects. In *Proc. the 33rd ICSE*, May 2011, pp.541-550.

[7] Gousios G, Zaidman A, Storey M A, van Deursen A. Work practices and challenges in pull-based development: The integrator's perspective. In *Proc. the 37th ICSE*, May 2015.

[8] Thongtanunam P, Tantithamthavorn C, Kula R G, Yoshida N, Iida H, Matsumoto K. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Proc. the 22nd SANER*, March 2015, pp.141-150.

[9] Dabbish L, Stuart C, Tsay J, Herbsleb J. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proc. CSCW*, February 2012, pp.1277-1286.

[10] Jiang J, Zhang L, Li L. Understanding project dissemination on a social coding site. In *Proc. the 20th WCRE*, October 2013, pp.132-141.

[11] Lehmann E L, D'Abrera H J M. Nonparametrics: Statistical Methods Based on Ranks. Prentice-Hall, 1998.

[12] Bird C, Pattison D, D'Souza R, Filkov V, Devanbu P. Latent social structure in open source projects. In *Proc. the 16th SIGSOFT FSE*, November 2008, pp.24-35.

[13] Xuan Q, Okano A, Devanbu P, Filkov V. Focus-shifting patterns of OSS developers and their congruence with call graphs. In *Proc. the 22nd SIGSOFT FSE*, November 2014, pp.401-412.

[14] Bird C, Gourley A, Devanbu P, Gertz M, Swaminathan A. Mining email social networks. In *Proc. the 2006 MSR*, May 2006, pp.137-143.

[15] Xuan Q, Filkov V. Building it together: Synchronous development in OSS. In *Proc. the 36th ICSE*, May 31-June 7, 2014, pp.222-233.

[16] Zhang W Q, Nie L M, Jiang H, Chen Z Y, Liu J. Developer social networks in software engineering: Construction, analysis, and applications. *SCIENCE CHINA Information Sciences*, 2014, 57(12): 1-23.

[17] Crowston K, Wei K, Howison J, Wiggins A. Free/libre open source software development: What we know and what we do not know. *ACM Computing Surveys*, 2012, 44(2): 7:1-7:35.

[18] Bird C, Gourley A, Devanbu P, Swaminathan A, Hsu G. Open borders? Immigration in open source projects. In *Proc. the 4th MSR*, May 2007, Article No. 6.

[19] Boser B E, Guyon I M, Vapnik V N. A training algorithm for optimal margin classifiers. In *Proc. the 5th Annual Workshop on Computational Learning Theory*, July 1992, pp.144-152.

[20] Xia X, Lo D,Wang X, Yang X, Li S, Sun J. A comparative study of supervised learning algorithms for re-opened bug prediction. In *Proc. the 17th CSMR*, March 2013, pp.331-334.

[21] Park B, Bea J K. Using machine learning algorithms for housing price prediction: The case of Fairfax county, Virginia housing data. *Expert Systems with Application*, 2015, 42(6): 2928-2934.

[22] Xia X, Lo D, Wang X, Zhou B. Accurate developer recommendation for bug resolution. In *Proc. the 20th WCRE*, October 2013, pp.72-81.

[23] Bettenburg N, Premraj R, Zimmermann T, Kim S. Duplicate bug reports considered harmful ... Really? In *Proc. the 24th ICSM*, September 28-October 4, 2008, pp.337-345.

[24] Xuan J, Jiang H, Ren Z, Zou W. Developer prioritization in bug repositories. In *Proc. the 34th ICSE*, June 2012, pp.25-35.

[25] Balachandran V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proc. the 35th ICSE*, May 2013, pp.931-940.

[26] Kagdi H, Gethers M, Poshyvanyk D, Hammad M. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 2012, 24(1): 3-33.

[27] Zweig M H, Campbell G. Receiver-operating characteristic (ROC) plots: A fundamental evaluation tool in clinical medicine. *Clinical Chemistry*, 1993, 39(4): 561-577.

[28] Xuan Q, Fu C, Yu L. Ranking developer candidates by social links. *Advances in Complex Systems*, 2014, 7(17): 1550005:1-1550005:19.

[29] Nurolahzade M, Nashehi S M, Khandkar S H, Rawal S. The role of patch review in software evolution: An analysis of the Mozilla Firefox. In *Proc. IWPSE-Evol*, August 2009, pp.9-17.

[30] Rigby P C, Germán D M, Storey M A. Open source software peer review practices: A case study of the Apache server. In *Proc. the 30th ICSE*, May 2008, pp.541-550.

[31] Bosu A, Carver J C. Peer code review in open source communities using review-board. In *Proc. the 4th PLATEAU*, October 2012, pp.17-24.

[32] Baysal O, Kononenko O, Holmes R, Godfrey M W. The influence of non-technical factors on code review. In *Proc. the 20th WCRE*, October 2013, pp.122-131.

[33] Linares-Vásquez M, Hossen K, Dang H, Kagdi H, Gethers M, Poshyvanyk D. Triaging incoming change requests: Bug or commit history, or code authorship? In *Proc. the 28th ICSM*, September 2012, pp.451-460.

[34] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In *Proc. the 17th ESEC/FSE*, August 2009, pp.111-120.

[35] Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proc. the 6th MSR*, May 2009, pp.131-140.

[36] Hossen M K, Kagdi H, Poshyvanyk D. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proc. the 22nd ICPC*, June 2014, pp.130-141.

[37] Hu H, Zhang H, Xuan J, Sun W. Effective bug triage based on historical bug-fix information. In *Proc. the 25th ISSRE*, November 2014, pp.122-132.

[38] Anvik J, Hiew L, Murphy G C. Who should fix this bug? In *Proc. the 28th ICSE*, May 2006, pp.361-370.

[39] Wu W, Zhang W, Yang Y, Wang Q. DREX: Developer recommendation with K-nearest-neighbor search and expertise ranking. In *Proc. the 18th APSEC*, December 2011, pp.389-396.

[40] Cubranic D, Murphy G C. Automatic bug triage using text categorization. In *Proc. the 16th SEKE*, June 2004, pp.92-97.

[41] Liu H, Ma Z, Shao W, Niu Z. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 2012, 38(1): 220-235.

[42] Lee J B, Ihara A, Monden A, Matsumoto K. Patch reviewer recommendation in OSS projects. In *Proc. the 20th APSEC*, December 2013, pp.1-6.

Jing Jiang received her B.S. and Ph.D. degrees in computer science from Peking University, Beijing, in 2007 and 2012, respectively. She is now an assistant professor in the State Key Laboratory of Software Development Environment of Beihang University, Beijing. Her research interests include software engineering, data mining, human factors and social aspects of software engineering.

Jia-Huan He received his B.E. degree in software engineering from Beihang University, Beijing, in 2009. He is now a master candidate in software engineering of Beihang University. His research interests include empirical software engineering, data mining, and machine learning.

Xue-Yuan Chen is now an undergraduate student in computer science of Beihang University, Beijing. His research interests include software engineering, data mining, and artificial intelligence.