A Hybrid Approach to Code Reviewer Recommendation with Collaborative Filtering

Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, Xudong Liu
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University, Beijing, China 100191
{xiazl, wangxu, liuxd}@act.buaa.edu.cn, {sunhl, jiangjing}@buaa.edu.cn

Abstract—Code review is known to be of paramount importance for software quality assurance. However, finding a reviewer for certain code can be very challenging in Modern Code Review environment due to the difficulty of learning the expertise and availability of candidate reviewers. To tackle this problem, existing efforts mainly concern how to model a reviewer’s expertise with the review history, and making recommendation based on how well a reviewer’s expertise can meet the requirement of a review task. Nonetheless, as there are both explicit and implicit relations in data that affect whether a reviewer is suitable for a given task, merely modeling review expertise with explicit relations often fails to achieve expected recommendation accuracy. To that end, we propose a recommendation algorithm that takes implicit relations into account. Furthermore, we utilize a hybrid approach that combines latent factor models and neighborhood methods to capture implicit relations. Finally, we have conducted extensive experiments by comparing with the state-of-the-art methods using the data of 5 popular GitHub projects. The results demonstrate that our approach outperforms the comparing methods for all top-k recommendations and reaches a 15.3% precision promotion in top-1 recommendation.

Index Terms—Code reviewer, collaborative filtering, github.

I. INTRODUCTION

Peer code review, the manual inspection of code changes, has been proven to be an effective means to reduce software defects and improve software quality [1], [2]. In recent years, Modern Code Review(MCR) [3], a tool-based and lightweight code review practice, has been widely adopted by industrial companies (e.g., Facebook, Google) and open source communities. For instance, GitHub provides an online review mechanism for ensuring the quality of the code from contributors. The review mechanism is an important part of GitHub’s pull-based software development model [4]. The contributors of a project may make some changes to the source code by adding a new feature or fixing certain bugs. The changes cannot be directly committed to the project repository, instead they are submitted to the community in the form of pull requests. The pull requests are subject to review by the whole community. Each user can be a potential reviewer, who can leave comments on a pull request in terms of the code style, the necessity of a new feature, the correctness of the code logic and etc. Finally, the integrator [5] will make a decision by taking all reviewers’ opinions into consideration.

For some popular GitHub projects, there are many contributors involved and a large number of pull requests are submitted as a result. Previous research [5] indicates that integrators often have difficulty to process the incoming pull requests in time. For instance, they can become tired of prioritizing the pull requests due to the heavy workload. As a result, it often leads to high latency for processing pull requests. According to findings in [6], 15% of contributors complain the delay of pull request response. The study conducted by Thongtanunam et al. [7] reveals that review requests with code reviewer assignments take 12 more days to be approved than others. As a result, an automatic reviewer recommendation is urgently needed for alleviating integrators’ workload and speed up code review process.

In practice, identifying the appropriate reviewers for a review task is a non-trivial task. A review task may involve multiple modules in the project, thus it is hard to find the competent reviewers. Besides, there are hundreds, even thousands, of candidate reviewers in a project, which aggravates the recommendation problem. In recent years, code reviewer recommendation has received a lot of attention [7]–[15].

Most existing works recommend reviewers by using explicit relations in historical review data. For instance, some researchers analyze the review data to build explicit relations between reviewers and contributors, and make recommendation based on that relationship. However, in practice some reviewers may have never reviewed a contributor’s review requests, but the reviewers and the contributor may have common interests in some technologies that make the reviewers suitable for reviewing the contributor’s requests. Considering such implicit relations is helpful for identifying competent reviewers.

In this work, we are mainly concerned with improving the code reviewer recommendation with the implicit relations in review data. To be specific, we first analyze related works and find that they only focus on explicit relations in review data leading to greater error. Second, we design a recommendation algorithm by taking consideration of implicit relations. With respect to the approach to measuring implicit relations, the algorithm uses a hybrid approach which combines a matrix factorization based latent factor model with a neighborhood-based method. Third, we evaluate our approach in comparison with the state-of-the-art algorithms. Results show that our
algorithm improves the precision with top-1 recommendation by 5.8%~15.3%.

The contributions of this work are summarized as follows:

- We analyze the review data and have a finding that implicit relations are helpful for reviewer recommendation algorithms. To the best of our knowledge, this is the first attempt to concern implicit relations in review data for code reviewer recommendation.
- We design a recommendation algorithm using a hybrid approach, which combines latent factor models and neighborhood models, to evaluate implicit relations in review data.
- We have conducted an extensive set of experiments with real OSS data to evaluate our approach in comparison with four representative methods of code reviewer recommendation. Experimental results confirm the advantages of our approach.

The remainder of this paper is organized as follows. In Section II, we present the related work about code reviewer recommendation. Section III gives an example to illustrate the motivation of our work. In Section IV, we design an algorithm to recommend appropriate reviewers for review requests by taking implicit relations into consideration. Section V presents the evaluation results of an extensive set of experiments. In Section VI, we discuss several important factors of reviewer recommendation. Finally, Section VII concludes the work.

II. RELATED WORK

Some previous works of reviewer recommendation have a common pattern. When a new review request comes, they firstly extract some specific features from the request. The same operation is also done to all resolved requests. Secondly, the similarity score between this new request and each resolved request is calculated. Thirdly, these similarity scores are assigned to the reviewers of resolved requests. So every reviewer has a score as the expertise with respect to the new request.

Thongtanunam et al. [7], [16] propose File Path Similarity algorithm (FPS). Its idea is that files in similar directories have similar functions. So the change files in similar directories, are probably reviewed by same reviewers. Let \( f_i \) and \( f_j \) be two files in the project. Let \( PathComparison \) be a function which takes two files as input and returns an integer as output. \( PathComparison : (f_i, f_j) \mapsto \mathbb{Z} \). The research [7] provides four methods to implement \( PathComparison \). Let \( Length \) be a function which calculates the directory levels of a file, denoted as \( Length : f_i \mapsto \mathbb{Z} \). The definition of similarity between two files is as follows:

\[
Similiarity : (f_i, f_j) \mapsto \frac{PathComparison(f_i, f_j)}{\max\{Length(f_i), Length(f_j)\}}.
\]  

Let \( r_i \) and \( r_j \) be two reviews. Let \( Files \) be a function that maps a review to the set of its reviewed files. Let \( RequestSimilarity \) be a function that calculates the similarity score between two requests, defined as

\[
RequestSimilarity : (r_i, r_j) \mapsto \frac{\sum_{f_i \in Files(r_i), f_j \in Files(r_j) \text{ similarity}}{\left|\text{Files}(r_i)\right| \ast \left|\text{Files}(r_j)\right|}}.
\]

Balachandran [12] proposes an algorithm called Review Bot. It considers each code line’s history. The similarity measurement is to judge whether the resolved request changes the same line. Rahman et al. [15] use external software libraries and specialized technologies, a bag of tokens, to represent a review request. They compare two review requests, i.e. two sets of tokens, using cosine similarity.

Some algorithms focus on the build of personal capabilities. Zanjani et al. [10] calculate expertise of every reviewer for each file. The expertise of a reviewer to a file are comprised of three metrics, which are the number of review comments, number of workdays, and the latest workday. Jiang et al. [9] propose the activeness attribute in reviewer recommendation. The algorithm counts the number of reviewed requests of every reviewer with consideration of time prioritization. The main idea of this algorithm is that the more active reviewers are, the more likely to inspect requests they are.

In recent years, the relationship between developers gets attention of researchers. Yu et al. [13] propose an algorithm called CN, which captures the relationship between the submitters of review requests and reviewers. CN firstly constructs a network which depicts comment relationship in a project. When a new review request comes, CN then extracts the submitter of this request and looks for the appropriate reviewers, who cooperate with the author frequently, in the network. Yu et al. also combine CN with IR method to recommend reviewers. We denote it as IR+CN.

Some other research works mix algorithms together to get the recommendation list. Ouni et al. [8] define the reviewer recommendation problem as a search-based optimization problem and take both reviewer expertise and reviewer collaboration into consideration. Xia et al. [11] propose an algorithm called TIE, which mixes FPS and a text classifier.

All these algorithms utilize various kinds of data in historical review data and try to extract distinguished features to model expertise of reviewers. However, the implicit relations are ignored by these algorithms. For instance, if a developer does not review the request \( r \), it does not mean that this developer is not competent for this request \( r \). In this work, we aim to provide an approach to improve reviewer recommendation accuracy through leveraging the implicit relations.

III. MOTIVATION

Previous works do not take implicit relations into consideration when recommending code reviewers. We take CN [13] which is a relation-centered algorithm as an example. CN captures the comment relationship between submitters of pull requests and reviewers in GitHub. Finally, a comment network is constructed where a directed edge from node \( a \) to node \( b \)
We analyze the comment network of netty project in GitHub when recommending reviewers for pull request 6146 which is titled “Wrap operations that require SocketPermission from the SecurityManager”. The network is shown in Figure 1. It can be observed that jasontedor has never reviewed tbrooks8’s pull requests. When this network is used to recommend reviewers for pull request 6146 submitted by tbrooks8, jasontedor will be ignored because there is no path from tbrooks8 to jasontedor. However, the true reviewers of pull request 6146 include jasontedor.

Pull request 6146 is about SecurityManager of JVM, while jasontedor is an expert of SecurityManager. As shown in Figure 2, normanmaurer, who is the owner of netty, left a comment in pull request 5545 to ask for jasontedor’s help because of his expertise on SecurityManager. In pull request 5561, jasontedor left a comment about SecurityManager. These historical activities indicate that jasontedor is good at SecurityManager. tbrooks8 is the submitter of pull request 6146 which covers SecurityManager while jasontedor is a contributor in this project who has the expertise of SecurityManager. This is an implicit relation between tbrooks8 and jasontedor, which CN ignores.

We try to use this case to explain a part of implicit relations in data. Implicit relations are the hidden relations between two data objects, where these two data objects are not related explicitly in the data we extracted. As what can be seen in the case above, considering these implicit relations is important for recommending reviewers.

Not just the CN, the other works also have the similar problem. For previous algorithms [7], [12], [15], they only use explicit relations, which are the real relationship between reviewers and requests. For algorithm eHRev [10], it also only utilizes the real modification operations, while ignoring implicit relations. Therefore we hope to take implicit relations into consideration when recommending reviewers to improve reviewer recommendation.

IV. A COLLABORATIVE FILTERING APPROACH TO REVIEWER RECOMMENDATION

In this section, we firstly organize the review data into the form of matrix. Secondly, we describe the logic of the recommendation algorithm. Thirdly, we elaborate the hybrid model, which is used to evaluate implicit relations.

A. Data Organization

In order to utilize the techniques of collaborative filtering to solve the reviewer recommendation problem, we need to organize the historical review data into matrix.

Let reviewerSet be the set of all reviewers in training data. Let requestSet be a set of review requests. We denote the matrix, which represents the historical data of code review, as Mat. The matrix Mat is defined as follows:

\[
Mat(i, j) = \begin{cases} 
\lambda_0 \sum_{n=1}^{m} t_1(i, j, n), & \text{if } i \text{ reviews } j \\
\text{null}, & \text{if } i \text{ does not review } j.
\end{cases}
\]

(3)

In the above equation, \( m \) is the count of review comments that are originated by reviewer \( i \) on request \( j \) and \( \lambda_0 \), which belongs to \( [0, 1] \), is the decay factor. \( \lambda_0 \) is used to weaken the influence of the number of a reviewer’s comments in the same review request. The factor of \( t_1(i, j, n) \) is to reflect the influence of time locality. We assume that recent comments are more important.

\[
t_1(i, j, n) = \frac{\text{commentTime}_{i,j,n} - \text{beginTime}}{\text{endTime} - \text{beginTime}}.
\]

\( \text{commentTime}_{i,j,n} \) is the creation time of review comment \( n \), which is commented by reviewer \( i \) on review request \( j \). \( \text{beginTime} \) and \( \text{endTime} \) are the earliest and latest creation time of review requests in training set.

The Request-Matrix is sparse because usually a reviewer only reviews a fraction of review requests. As a result, the matrix has many void entries. Assigning reasonable values to these void entries is the specific form of capturing implicit relations in review data.
B. Algorithm Logic

To capture the implicit relations, as Implicit Relation Catcher does in Section IV-C. Here, we use a function, which is denoted as eval, to represent the Implicit Relation Catcher.

\[
eval : i, j \mapsto \mathbb{R}, i \in \mathbb{Z}, j \in \mathbb{Z}. \tag{4}\]

We also define a function called transform, which takes a review request \( r \) as input and returns a subset of requestSet, which can reasonably represent the new review request.

\[
\text{transform} : r \mapsto \varphi, \varphi \in \rho(\text{requestSet}), \rho \text{ means power set.} \tag{5}\]

We use equation 2 to calculate the similarity between requests and finally choose a set of review requests with the highest similarity scores.

The main logic is shown in Algorithm 1. Line 5 uses function transform to transform a review request to some similar requests in requestSet. Line 6 to 19 iterate each reviewer in reviewerSet to calculate the reviewer’s score. Line 11 to 14 calculate the score with regard to a specific reviewer and a specific col. If the entry is void, line 14 uses eval to calculate, which evaluates the implicit relations. Line 20 to 21 sort the reviewers by their scores in descending order and return the top \( k \) reviewers as the result. How function eval works is explained in Section IV-C.

**Algorithm 1 Reviewer Recommendation Algorithm**

**Input:** \( r \) : a new review request, \( k \) : the length of recommendation list

**Output:** \( C \) : a list of reviewers

1: function REVIEWERRECOMMENDATION(r, k)
2: Construct matrix \( \text{Mat} \)
3: Get all reviewers as a set reviewerSet
4: \( C \leftarrow \emptyset \)
5: \( \text{colSet} \leftarrow \text{transform}(r) \)
6: for reviewer \( \in \text{reviewerSet} \) do
7: \( i \) : Get label of reviewer in reviewerSet
8: \( \text{score} \leftarrow 0 \)
9: for col \( \in \text{colSet} \) do
10: \( j \) : Get label of col in requestSet
11: if \( \text{Mat}(i, j) \) is not void then
12: \( \text{score} \leftarrow \text{score} + \text{Mat}(i, j) \) //utilize explicit relations
13: else
14: \( \text{score} \leftarrow \text{score} + \text{eval}(i, j) \) //catch implicit relations
15: end if
16: end for
17: reviewer.score \( \leftarrow \text{score} \)
18: \( C \leftarrow C \cup \{\text{reviewer}\} \)
19: end for
20: Sort elements in \( C \) by score of every reviewer in descending order
21: Return the top \( k \) elements in the \( C \) as a list
22: end function

C. Implicit Relation Catcher

\( \text{eval} \) function undertakes the responsibility to capture the implicit relations in data. The specific form of \( \text{eval} \) is to assign values to void entries in matrix, while collaborative filtering algorithms [17] are good at this. As a result, we utilize the techniques of collaborative filtering to solve this problem. We use a hybrid approach, which is a 2-tier model for recommendations [18]. The first tier is a latent factor model and the second tier is a neighborhood method. This approach can consider not only the localized strong associations between data objects but also the holistic structure of the data.

1) Latent Factor Model: We use a latent factor model to capture implicit relations because of its strong ability to extract characteristics. As shown in Figure 3, we use historical review data to train the latent factor models. As a result, all reviewers and requests in the training data are represented in a common latent factor space, i.e., reviewers and requests are points in this space.

The latent factors may measure the degree of contribution, technologies, skills, modules in project or some uninterpretable dimensions. These factors are extracted automatically by latent factor models and can represent some hidden characteristics. The exact implication of each factor is hard to interpret. It is preferable to take charge of catching implicit relations in data.

We use an SVD(Singular Value Decomposition)-like lower rank decomposition of the matrix [19]. Each reviewer is associated with a vector \( p_i \in \mathbb{R}^f \), and each request is presented by a vector \( q_j \in \mathbb{R}^f \), \( f \) is the order of the common space. The value of \( \text{Mat}(i, j) \) is calculated by taking an inner product, i.e., \( p_i^T q_j \). So our equation is as follows:

\[
b_{ij} = p_i^T q_j. \tag{6}\]

\( b_{ij} \) is the estimated value for \( \text{Mat}_{ij} \).

This model can capture the holistic structure of data, i.e., it can mine the latent rules of the whole data in the matrix. However, it is poor at detecting the strong associations with a small portion of closely related items [18]. To improve this
model, we incorporate a neighborhood model. The latent factor model is as a baseline model.

2) **Integrated Model**: Neighborhood models [20] rely on some closely related items so it is good at capturing localized associations. The integrated model is as follows:

$$M_{\hat{t}_{ij}} = \frac{p_i^T q_j}{b_{ij}} + \sum_{h \in R^d(i;j)} (Mat_{ih} - b_{ih})w_{jh}.$$  

(7)

To explain the above equation, we need to give some definitions. In neighborhood model, an appropriate measure is needed to evaluate similarity between two items. We use a shrunk correlation coefficient [18]: $s_{jh} \overset{\text{def}}{=} \frac{n_{jh}}{n_{jh} + 100\text{pearson}_{jh}}$. The variable $n_{jh}$ means the count of rows where both their elements in $j$ and $h$ columns are not void. The variable $\text{pearson}_{i,j}$ means the Pearson correlation coefficient of $j$ and $h$ columns’ data in matrix. Let $S^d(j)$ be a set of $d$ items most similar to $j$ according to the above similarity measure. $d$ is called neighborhood size. Let $R(i)$ be a set of items, defined as $\{j | Mat(i,j) \neq \text{null} \}$. The $R^d(i;j)$ in equation 7 is an intersection set of $S^d(j)$ and $R(i)$. $R^d(i;j) = R(i) \cap S^d(j)$. The $w_{jh}$ in equation 7 is a weight parameter and $|R^d(i;j)|^{-\frac{1}{2}}$ is a constant factor.

To inspect equation 7, we know that $p_i, q_j, w_{jh}$ are all parameters, which need to be estimated. Let $\kappa = \{(i,j)|Mat(i,j) \neq \text{null}\}$. These parameters are learnt by solving the regularized least squares problem:

$$\min_{p_i, q_j, w_{jh}} \sum_{(i,j) \in \kappa} (Mat_{ij} - M_{\hat{t}_{ij}})^2$$

$$+ \lambda_2 \left( \|p_i\|^2 + \|q_j\|^2 + \sum_{b \in R^d(i;j)} w^{2}_{jb} \right).$$  

(8)

We use Stochastic Gradient Descent(SGD) [21] algorithm to solve this problem. After the learning process, we can get the estimated value of every parameter. So we know each reviewer’s vector $p_i$ and each item’s vector $q_j$. After getting these representations, it is straightforward to use equation 7 to evaluate the implicit relations.

V. EVALUATION

A. Data Collection and Preprocess

To collect data for our experiments, we choose five popular GitHub projects including angular/angular.js, netty/netty, salt-stack/salt, ipython/ipython and symfony/symfony. These five projects are very popular and have 1643 watches, 18580 stars and 9017 forks on average in GitHub. Additionally, each of them is widely used and plays a significant role in their technical domains. In GitHub, we regard all comments on pull requests as independent reviews and regard the authors of these comments as reviewers. So we use GitHub API\(^2\) to crawl relevant data of these five projects.

After data collection, we filter out pull requests with less than two different reviewers according to previous works [22]–[24]. In addition, we remove stop words of pull requests’ title and description. After that, we do stemming using Porter stemmer and pull requests with less than five words are filtered out. The statistics of our data are shown in Table I.

B. Evaluation Metrics

We use precision and recall to evaluate the performance of various recommendation algorithms. In this paper, we evaluate top-1, top-2, top-3, top-4 and top-5 recommendation precisions and recalls.

$$\text{Precision} = \frac{|\text{ActualReviewers} \cap \text{RecommendedReviewers}|}{|\text{RecommendedReviewers}|}$$

$$\text{Recall} = \frac{|\text{ActualReviewers} \cap \text{RecommendedReviewers}|}{|\text{ActualReviewers}|}$$

C. Experiment Setup

We denote our algorithm based on Request-Matrix as PR-CF. For every project, we sort all pull requests in chronological order and choose the latest 600 pull requests as test set. Other pull requests are training set. For IR+CN [13] and PR-CF, we split the test set into 6 stages, where each stage contains 100 pull requests. After each stage, we rebuild the model to utilize the previous stage’s data. For FPS [7], Activeness [9] and TIE [11], it is not necessary to rebuild the models because they can use each latest pull request immediately.

For PR-CF, we always restrict the training set size to $\delta$ latest pull requests due to the consideration of high contributor turnover. Previous works [25]–[28] show that OSS projects have a high contributor turnover. According to the analysis [27], the mean turnover ratio in GitHub projects is 66% for each quarter. As a result, only considering a fixed number of latest pull requests is reasonable. In addition, restricted training set can decrease the size of the matrix, which automatically reduce the computational complexity.

D. Parameter Setting

There are several parameters need to be set in our model. Because there are no specific rules to determine these parameters, we set parameters by trial-and-error. For replicability, the parameters of our approaches and the compared approaches are shown in Table II and Table III respectively. The parameters of compared approaches are also finely tuned.

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>PARAMETER SETTINGS OF PR-CF ALGORITHM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_0$</td>
<td>0.7</td>
</tr>
</tbody>
</table>

\(^2\)https://developer.github.com/v3/
TABLE I
STATISTICS OF COLLECTED DATA

<table>
<thead>
<tr>
<th>projects</th>
<th>angular/angular.js</th>
<th>netty/netty</th>
<th>saltstack/salt</th>
<th>ipython/ipython</th>
<th>symfony/symfony</th>
</tr>
</thead>
<tbody>
<tr>
<td>reviewers</td>
<td>1341</td>
<td>218</td>
<td>475</td>
<td>349</td>
<td>1536</td>
</tr>
<tr>
<td>issues</td>
<td>8552</td>
<td>3457</td>
<td>15156</td>
<td>5295</td>
<td>8332</td>
</tr>
<tr>
<td>filtered pull requests</td>
<td>3361</td>
<td>1530</td>
<td>4378</td>
<td>2116</td>
<td>7146</td>
</tr>
<tr>
<td>language</td>
<td>JavaScript</td>
<td>Java</td>
<td>Python</td>
<td>Python</td>
<td>Php</td>
</tr>
</tbody>
</table>

TABLE III
PARAMETER SETTINGS OF COMPARED ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>time decay factor</td>
<td>0.8</td>
</tr>
<tr>
<td>Activeness</td>
<td>time window</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>time decay factor</td>
<td>1</td>
</tr>
<tr>
<td>TIE</td>
<td>weight for text classifier</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>time window</td>
<td>100</td>
</tr>
</tbody>
</table>

E. Research Questions and Results

RQ1: Is PR-CF better than FPS, Activeness, IR+CN and TIE?

Whether PR-CF outperforms state-of-art algorithms is a crucial question because it relates to whether our work is valuable and solid. We choose an algorithm from every category of our classification and a mixed algorithm. So four algorithms are selected to be compared with our algorithm PR-CF. These four algorithms are FPS [7], Activeness [9], IR+CN [13] and TIE [11]. We run these five algorithms over our data set and calculate the average precision and recall of top-1, top-2, top-3, top-4 and top-5 recommendations for each project.

Table IV presents the results of comparisons. Best result for each top-k recommendation in every project is in bold. We can see that in most cases PR-CF beats all other algorithms. Specifically, PR-CF gets 21 times the best results in total 25 precision comparisons. We also calculate the average precision and recall of all projects and show that via the chart of precision vs. recall. As shown in Figure 4, every point in the curve represents the average performance of a top-k recommendation. PR-CF outperforms all other four algorithms in all top-k recommendations. Compared with FPS, Activeness, IR+CN and TIE in top-1 recommendation, PR-CF gains 6.9%, 5.8%, 15.3% and 13.2% promotion of precision and 7.4%, 6.6%, 16.4% and 14.1% promotion of recall respectively.

RQ2: Do implicit relations improve the performance of PR-CF?

We want to measure how much PR-CF benefits from implicit relations. We conduct an experiment comparing PR-CF with PR-CF without implicit relations. The algorithm PR-CF without implicit relations, which is denoted as PR-CF*, does not evaluate implicit relations, i.e., the eval function always returns 0 for any input. We run these two algorithms over five projects and calculate total average of precision and recall.

Table V presents the result. In all top-k recommendations, PR-CF outperforms PR-CF* distinctly. In top-1 recommendation, PR-CF gains a 13.0% promotion of recall, compared with PR-CF*. This result shows that taking implicit relations into consideration brings a significant performance improvement.

VI. DISCUSSION

A. Variation between Projects

The performance of a specific algorithm varies between projects. As shown in Figure 5, PR-CF algorithm has different precision performance in different projects. Other algorithms are also like this. Algorithms perform well in netty project, but perform poorly in salt project. There is a noticeable performance gap between netty and salt. What is the reason behind this? We carefully analyze the data of some projects including netty and salt. We find that pull requests in netty are classified more carefully using labels [29], [30] by the core team members of the project. Almost every pull request in netty is assigned a label such as defect, cleanup, improvement, feature and so on. These labels are good for external contributors to understand the pull requests and help them choose their interested pull requests to review. It shows that review rules are made in some projects and the core team members cooperate with each other to assign the labels and review pull requests.
TABLE IV

<table>
<thead>
<tr>
<th>Project</th>
<th>k</th>
<th>precision@k</th>
<th>recall@k</th>
</tr>
</thead>
<tbody>
<tr>
<td>angular</td>
<td>1</td>
<td>0.664</td>
<td>0.285</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.643</td>
<td>0.279</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.641</td>
<td>0.273</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.648</td>
<td>0.275</td>
</tr>
<tr>
<td>netty</td>
<td>1</td>
<td>0.698</td>
<td>0.323</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.546</td>
<td>0.316</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.557</td>
<td>0.328</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.530</td>
<td>0.312</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.530</td>
<td>0.320</td>
</tr>
<tr>
<td>salt</td>
<td>1</td>
<td>0.766</td>
<td>0.504</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.646</td>
<td>0.493</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.722</td>
<td>0.497</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.722</td>
<td>0.487</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.748</td>
<td>0.483</td>
</tr>
<tr>
<td>ipython</td>
<td>1</td>
<td>0.796</td>
<td>0.678</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.737</td>
<td>0.687</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.694</td>
<td>0.656</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.694</td>
<td>0.656</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.694</td>
<td>0.656</td>
</tr>
<tr>
<td>symfony</td>
<td>1</td>
<td>0.810</td>
<td>0.753</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.733</td>
<td>0.742</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.657</td>
<td>0.727</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.657</td>
<td>0.727</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.657</td>
<td>0.727</td>
</tr>
</tbody>
</table>

In a word, the review work of such projects is well organized. As a result, the data of these projects can reflect the reviewers' expertise accurately and the prediction precision is high. So for the reviewer recommendation problem, there are more further works to be done by considering the characteristics of projects and we believe that project specific algorithms will get better results. We leave this as our future work.

B. Time Sensitivity

GitHub grows quickly and the popular projects in GitHub also develop rapidly [4]. In GitHub, both development and contributor have a high turnover [4], [26]. As a result, time is an important factor for code reviewer recommendation problem in GitHub. Hannebauer et al. [14] classify reviewer recommendation algorithms into three categories: time-local algorithms, forgetful algorithms and time-global algorithms. PR-CF belongs to both time local algorithms and forgetful algorithms.

Thongtanunam et al. [7], [16] show that when the time decay factor of FPS is set to 1, FPS behaves the best for Android, OpenStack, Qt projects, which use Gerrit-based code review systems. In other words, recommending reviewers for these projects is not time sensitive. This may be attributed to the stability of the developers of these projects. However, it is not applicable to GitHub projects. According to our experiments, FPS performs better and more stably when setting the time factor to 0.8 instead of 1. So time factor is vital for reviewer recommendation in GitHub. In our algorithm, we take time factor into consideration when constructing the matrix.

C. Threats to Validity

First, as we use five popular projects for experiments, the performance of our algorithms in other projects is yet to be studied. Second, our experiments are based on the GitHub dataset, and we do not know the results of applying it to other platforms, such as Bitbucket. Third, we use the real reviewers of pull requests as the true result, however, the true reviewers may not be the best reviewers. So our recommendation has the risk of recommending reviewers that are not the best suitable.

VII. Conclusion

Automatic reviewer assignment in MCR is urgently needed in large-scale software development, especially for OSS projects where there often involve a large number of developers who hardly know each other well. Code reviewer recommendation is widely regarded as an effective means that can speed up code review process and improve development efficiency. Previous works on reviewer recommendation focus...
on the feature extraction of the historical review data and try to model expertise of reviewers through explicit relations. However, they neglect the implicit relations that cannot be observed directly from the data. In this work, we aim at incorporating collaborative filtering techniques to capture the implicit relations that affect whether a reviewer will take a review task. As a result, we implement a specific algorithm called PR-CF that focuses on implicit relations in data. Comparing experiments with previous works were conducted and the results show that our algorithm beats all other algorithms on average.

ACKNOWLEDGEMENT

This work was supported partly by the National Key Research and Development Program of China under Grant No. 2016YFB1000804, and the State Key Laboratory of Software Development Environment under Grant No. SKLSDE-2017ZX-14. And the authors would thank the anonymous reviewers for the helpful comments and suggestions to improve this paper.

REFERENCES