Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers

Aleksandr Chueshev, Julia Lawall, Reda Bendraou, Tewfik Ziadi

Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers

Aleksandr Chueshev*, Julia Lawall†, Reda Bendraou*, and Tewfik Ziadi*
Sorbonne University/LIP6,* Inria †
Paris, France
Emails: {aleksandr.chueshev, reda.bendraou, tewfik.ziadi}@lip6.fr, {julia.lawall}@inria.fr

Abstract—**Code review is an important part of the development of any software project. Recently, many open source projects have begun practicing lightweight and tool-based code review (a.k.a modern code review) to make the process simpler and more efficient. However, those practices still require reviewers, of which there may not be sufficiently many to ensure timely decisions. In this paper, we propose a recommender-based approach to be used by open-source projects to increase the number of reviewers from among the appropriate developers. We first motivate our approach by an exploratory study of nine projects hosted on GitHub and Gerrit. Secondly, we build the recommender system itself, which, given a code change, initially searches for relevant reviewers based on similarities between the reviewing history and the files affected by the change, and then augments this set with developers who have a similar development history as these reviewers but have little or no relevant reviewing experience. To make these recommendations, we rely on collaborative filtering, and more precisely, on matrix factorization. Our evaluation shows that all nine projects could benefit from our system by using it both to get recommendations of previous reviewers and to expand their number from among the appropriate developers.**

Index Terms—**recommender systems; code review; collaborative filtering; matrix factorization;**

I. Introduction

Code review is generally accepted to be an essential pillar in any large-scale software development process. Today, the software industry is increasingly replacing heavyweight old-style code inspections, including waterfall-like procedures, an expert panel, group meetings, and other formal requirements [1], with modern code review (MCR) [2]. MCR follows a less-formal model based on asynchronous processes and focuses on reviewing code changes by a non-author, often using support tools, such as Gerrit [3], Github [4], and CodeFlow [5]. Usually, MCR involves code discussions, suggestions for fixes, and, finally, the integration of changes, once approved by reviewers, into the main version of a system. Today, MCR is regularly used in both proprietary and open-source software (OSS) projects [2], [6], [7].

Research has shown that MCR can have a large impact on improving the quality of source code and software in general [8]. However, Rigby et al. [9] have identified that code reviews might cost a lot because they require reviewers to read, understand, and critique code changes made by others. To assess changes effectively, the reviewer should have a deep understanding of the affected code [10]. As a software project grows, the process of finding an appropriate reviewer becomes more time-consuming and less effective. Recent talks at developer conferences reveal that the problem of finding appropriate reviewers has encouraged open-source communities to become interested in expanding the number of reviewers. For instance, the Intel Open Source Technology Center [11] and Apache Software Foundation (ASF) projects (e.g., Apache Spark [12], Apache Kafka [13]) claim that reviews are open for everyone, even if contributors only provide quick comments and insights.

We focus in this paper on the OSS model, which has become an important driving force in modern software development [14]. Commercial companies and startups increasingly contribute to OSS as well as opening their own projects to the open-source community [15]. Expanding the number of reviewers in OSS projects promotes knowledge sharing among the contributors and helps contributors get to know the codebase. Besides, having sufficient reviewers allows balancing the workload without putting too much burden on a few key persons. However, expanding the set of reviewers cannot be implemented by randomly selecting reviewers from among the developers. A strategy is needed that takes into account developers' current experience and knowledge about the code and the project structure, to ensure that they will be able to perform reviews successfully.

One strategy for expanding the set of reviewers is to use a recommender system. Such a system could examine various project data in order to identify appropriate developers and suggest them as possible reviewers for a given code change. In recent years, recommender systems have attracted a lot of attention and are being successfully used in various domains, such as entertaiment [16], [17], medicine [18], natural language processing [19], and software engineering [20], [21]. In our setting, a recommender system has to meet certain requirements. **R1**: Contributors should not have to indicate manually their reviewing preferences. Indeed, in an OSS setting, contributors typically participate as much or as little as they want, and are not required to participate in management-related tasks. **R2**: Since many projects are written in multiple languages, the recommender system must be independent of the development language. **R3**: The recommender system must account for the possibility that the developers have varying levels of expertise in the different parts of the system where they have previously worked, and may even be knowledgeable about related parts of the system to which they have not previously explicitly contributed. **R4**: To expand the number of reviewers, the recommender system must recommend reviewers both from contributors who have previously done reviews and from contributors who have previously only worked on code changes.

Contribution: To make recommendations addressing the defined requirements, we have designed a recommender system based on the collaborative filtering (CF) strategy, and more precisely, on matrix factorization. Our recommender system bridges the gap between reviewers and developers who have little or no reviewing experience as follows. For a given code change, the system first searches for relevant reviewers based on the history of reviews. Our approach then identifies as new possible reviewers those developers whose development history is similar to that of the found reviewers. A fundamental hypothesis of our approach is that contributors with similar development preferences may also have similar reviewing preferences. These preferences reflect abstract information about contributors' experience, respectively, in development and reviewing, and are automatically deduced while incrementally processing the project history of commits and reviews. Note that we focus on developers' competence to do reviews; another important issue is their motivation, but we leave that to future work.

This paper targets OSS projects maintained through the pull-based model of GitHub and the patch workflow of Gerrit. GitHub is a platform on which developers communicate with each other by working on issues, coding, and making reviews on proposed code changes. GitHub stores a huge collection of data, e.g., issues, commits, and reviews, and is the world's largest host for OSS projects today. The pull-based model provided by GitHub unifies the development and review processes, and thus simplifies the collection and preparation of data for our recommender system. Gerrit was originally developed by Google as a platform specifically focused on reviewing and is compatible with Git systems. Today, Gerrit is a part of the open-source community and has gained popularity among such foundations as Libreoffice, Eclipse, Openstack, etc. Like GitHub, Gerrit provides access to a collection of reviews of proposed code changes. Its model is very similar to GitHub but slightly different in how reviewers approve the proposed code changes before introducing them into the project.

To evaluate our approach, we have used two data sets with a different frequency of code changes, number of commits, files, and contributors. The first data set consists of five pull-based open-source ASF projects hosted on GitHub: Beam [22], Flink [23], Kafka [13], Spark [12], and Zookeeper [24]. Spark is the largest of those projects with around 25,000 commits. Zookeeper, in contrast, is the smallest with only 2,018 commits. To compare our system with the current state of the art, we have additionally used a second data set of four software communities that host their reviews on Gerrit: Android [25], Openstack [26], QT [27], and Libreoffice [28]. Android is the largest among the selected communities with more than 3,000,000 commits across the 112 subprojects, and QT is the smallest with over 149,000 commits across the 35 subprojects. In all cases, our recommender system works successfully, identifying relevant reviewers from the set of actual reviewers and potential ones from among the developers.

In summary, this paper makes the following contributions:

- A rich collection of review and development data, including information about reviewers, developers and their commits within five large ASF projects.
- An extension of an existing dataset from four software communities using Gerrit, further updated and populated with development history data.
- An exploratory study on the distribution of reviewers and developers, showing how reviewers and developers participate in the given projects.
- A recommender-based approach for OSS projects to recommend regular reviewers and to expand their number from among the appropriate developers.
- An evaluation of our approach in simulation mode, showing how the system can make online recommendations of relevant potential reviewers.

The rest of this paper is organized as follows. Section II describes the context of our work, providing a description of the used data sets and an exploratory study of how developers and reviewers are distributed and involved within each project. Section III introduces the principles that underlie our approach and explains how the recommender system works. Section IV provides the results and the corresponding validation. Section V presents related work. In Section VI, we discuss the benefits and possible biases of our approach, as well as how to integrate the recommender system into a software development process. We discuss the findings and conclude in Section VII.

II. Data exploration

This section presents our data sets and an exploratory study used to understand the current state of reviewing in OSS projects. After establishing some terminology, we first describe the process of extracting data from the studied OSS projects. We then present the exploratory study, which investigates how developers and reviewers are distributed and how they participate in the given projects.

A. Terminology

In our study, we use five GitHub projects and four Gerrit software communities. Technically, each of these Gerrit communities consists of several subprojects. However, following previous work [29], [30], we do not split the Gerrit communities and process each one as a single project. Thus, hereafter, we use the term "project" to refer to both GitHub projects and Gerrit communities.

GitHub and Gerrit both support review processes for new code changes delivered to the system, but there is a difference in the terminology used. GitHub calls proposed code changes

TABLE I: Statistics on the collected data sets

Project	Time Period	# Rev.	# Dev.	# PR	# Commits	# Files
Beam	12-2014 - 01-2020	227	700	10,469	24,933	29,611
Flink	12-2010 - 01-2020	262	831	10,726	19,904	39,295
Kafka	08-2011 - 01-2020	294	737	7,881	6,941	5,630
Spark	03-2010 - 01-2020	755	2000	27,042	26,089	27,554
Zookeeper	11-2007 - 01-2020	72	150	1,195	2,053	4,361
Android	10-2008 - 01-2020	3,058	29,573	167,554	3,016,120	1,948,916
Openstack	07-2011 - 01-2020	3,769	5,122	135,357	180,654	62,692
QT	05-2011 - 01-2020	1,050	2,103	175,040	149,857	175,184
Libreoffice	03-2012 - 01-2020	214	1,283	74,127	205,918	180,418

Notes: Rev. - reviewers; Dev. - developers

"pull requests" or "PR". Gerrit calls those code changes simply "changes". Semantically, both the terms "pull requests" and "changes" describe the same process that has an author, requested reviewers and other contributors who participate in discussions. Thus, hereafter, we use the term "pull request" or simply "PR" when talking about both GitHub and Gerrit.

A cornerstone of our approach is to distinguish between different kinds of contibutors to a software project. There are indeed three roles that the contributor can follow that are relevant to our approach: a "plain developer" who only codes, a "plain reviewer" who only does reviews, and a "developer-reviewer" who does both. When we say "reviewers", without any clarifications, we mean contributors with both "plain reviewer" and "developer-reviewer" roles. Likewise, when we say "developers", we mean contributors with the roles of "plain developer" and "developer-reviewer".

B. Building the data sets

In this section, we explain the process of collecting two data sets from GitHub and Gerrit, respectively. These data sets are used to evaluate our approach and to compare with the current state of the art, respectively. Table I provides complete statistics on the collected data.

a) ***GitHub data set:*** To build the GitHub data set, we use the review and development history of five open-source ASF projects: Beam, Flink, Kafka, Spark, and Zookeeper. Beam, Flink, Kafka, and Spark are among the top 20 repositories of ASF by the number of commits made in 2019 [31]. Zookeeper is a smaller project but is used by Kafka, Hadoop [32], HBase [33], Hive [34], etc. All these projects are maintained through the pull-based model that involves the submission of proposed changes via pull requests. Pull requests contain commits that will be pushed into a repository if a reviewer approves them. In general, once a pull request is opened, contributors can review proposed changes, add review comments, contribute to the pull request discussion, and even add additional commits [35].

Using GitHub's public GraphQL API v4 [36], we have extracted the information about all of the almost 58,000 pull requests from the five selected projects. Using the Git *log* command, we have additionally extracted the more than 80,000 commits to the master branches and the 106,558 files created since the first extracted commit. Figure 1a and 1b show examples of the information obtained from GitHub.

(a) Kafka PR #7430

```
number: 7430 state: merged
title: KAFKA-5609: Connect log4j should also log to a file
     by default (KIP-521)
description: Enable Kafka Connect to redirect log4j messages
     to a file by default, additionally to the redirection
     to standard output. The file-based log4j export is set
     to be daily and shares the same pattern with the stdout
     appender.
created at: 2019-10-02T03:13:26Z
updated at: 2019-10-02T22:21:30Z
closed at: 2019-10-02T22:04:11Z
merged at: 2019-10-02T22:04:11Z
files: [config/connect-log4j.properties]
author: <login>
reviewers: [<login>]
```

(b) Kafka commit 7041e76

```
id: 7041e76bd6ef0c28ec8de2d07f2208171745f936
description: MINOR: Some logging improvements for debugging
     delayed produce status (#4691). A few small logging
     improvements which help debugging replication issues.
committed at: 2018-03-19T11:08:12Z
files: [
  core/src/main/scala/kafka/server/DelayedProduce.scala,
  core/src/main/scala/kafka/server/LogOffsetMetadata.scala,
  core/src/main/scala/kafka/cluster/Partition.scala,
  core/src/main/scala/kafka/tools/DumpLogSegments.scala]
author: <login>
related pull request: 4691
```

(c) Android PR #31210

```
id: Ibc3325063e1b34e2eb71eb4643dc6fbfddf8ec96
number: 31210 state: merged
created at: 2012-01-18T12:04:51Z
updated at: 2012-01-18T16:26:15Z
description: Rename (IF_)LOGW(_IF) to (IF_)ALOGW(_IF)
project: platform/hardware/ril
files: [libril/ril.cpp]
author: <first name><last name>:<email>@example.com
reviewers: [<first name><last name>:<email>@example.com]
```

Fig. 1: Examples of data set entities (anonymized)

b) ***Gerrit data set:*** To compare our approach with the current state of the art, we use a second data set of four software projects using Gerrit: Android [25], Openstack [26], QT [27], and Libreoffice [28]. This data set was provided by Thongtanunam et al. [29], [37]. It contains a history of reviews, including related files and reviewers' identifiers. However, this data set is somewhat outdated and does not contain the development history used by our recommender system. To adapt it for our approach, we have updated the original data set to keep it up-to-date, and augmented it with the necessary data on commits and developers.

The original data set contains the reviews for over 42,000 pull requests across the four projects. Using the public Gerrit API [38], we have additionally extracted the more than 552,000 pull requests. In the same manner as for the GitHub projects, we have also extracted the 3,552,549 commits to all branches and the 2,367,210 related files. Figure 1c shows an example of a pull request obtained from Gerrit.

*c) **Identifying data set entities:*** To identify the GitHub contributors uniquely, we rely on their GitHub user logins. However, we have found that some commits of the GitHub data set do not contain logins of their developers, but instead only have the name and email obtained from the developer's local Git settings. Indeed, another contributor may act on behalf of the developer and submit his/her commits, opening a pull request. For the developers of commits without GitHub logins, we create surrogate logins by concatenating their local Git names and emails. For contributors whose profiles have been extracted from Gerrit, their internal Gerrit identifiers are used by default. If the developer of a commit does not have an identifier, for the same reasons as the developers of the GitHub data set, we create a surrogate login. Other entities, such as files, commits, and pull requests, are identified by their paths, commit IDs, and numbers, respectively.

C. Exploratory study

Based on our data sets, we have conducted an exploratory study to understand the needs for expanding the number of reviewers. The study is split into three parts. The first part concerns the roles of contributors and their distribution in the selected projects. The second part quantifies the effort offered by the contributors to perform reviews. The third part describes how contributors distribute their effort between development and review tasks.

Investigating the roles of contributors and their distribution: To study how contributors participate in the given projects, we identify the developers, i.e., the authors of commits, and the reviewers of pull requests. We count only those developers who have modified at least one file. Developers who only delete or add files are not taken into account by this study as, in the selected projects, they potentially represent contributors who only maintain the stability of the projects. To count reviewers, we process all available pull requests regardless of their state (opened, closed, etc.). Figure 2 illustrates that for all selected projects, most contributors follow a "plain developer" role. The rest participate as "plain reviewers" or combine both roles. Since our goal is to increase the number of reviewers from among the developers, our interest lies in expanding the "developer-reviewer" column from the contributors in the "plain developer". This part of the study shows that in all the projects except Openstack, the number of plain developers is 28-88% (on average, 55%) more than the number of all reviewers. In Openstack, the number of all reviewers exceeds that of plain developers and thus, its maintainers may be interested in recommending regular reviewers rather than expanding their set.

Quantifying the efforts offered to reviewing: Even when a project has a number of developer-reviewers, the degree to which they help reduce the overall reviewing burden depends on how many reviews they actually do. To investigate this, we study the corresponding cumulative distribution function for the number of reviews made for each project. Figure 3 presents the results obtained for the five projects (Beam, Kafka, Zookeeper, Android, and QT) selected as the most representative. Each graph illustrates how many developer-reviewers of the project have done at least x reviews, where one review corresponds to one reviewed pull request. A pull request with multiple reviewers is counted for each participating reviewer. Overall, all the graphs show a steeply sloping curve and a long tail tending to zero. The stronger the steepness of the curve, the more reviews are performed by a few contributors. The results show that each project has a certain imbalance in how developer-reviewers actually contribute to the set of reviews. The largest project Android demonstrates the worst trend, with one contributor performing over 28,000 reviews, and the others performing significantly fewer. The smallest Zookeeper project, in contrast, shows the best trend with a more balanced distribution of efforts to review. Beam and Kafka demonstrate similar distributions with a better trend than QT. Considering the projects that are not shown, the distribution of Flink and Libreoffice is respectively similar to that of Kafka and QT. Finally, Spark and Openstack present a trend worse than the previous projects but better than Android.

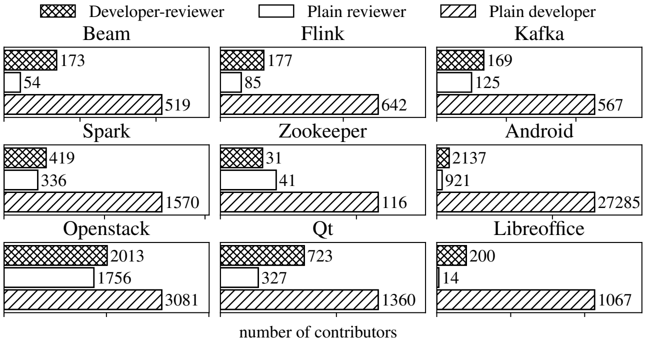

Fig. 2: Distribution of roles

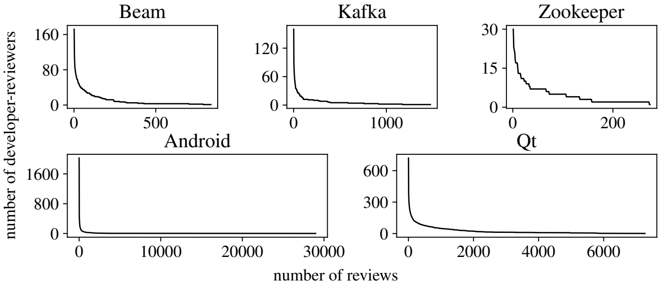

Fig. 3: Cumulative distribution of reviews made

Investigating the distribution of contributors' efforts between development and reviewing: To study how the effort offered to reviewing correlates with that offered to development, for each developer-reviewer we have plotted the joint distribution of commits and reviews. Figure 4 confirms that a large number of developer-reviewers perform a small number of reviews. For instance, 58%, 65%, 52%, 67%, and 53% of developer-reviewers respectively for Beam, Kafka, Zookeeper, Android, and QT have performed fewer than ten reviews. For the rest, i.e., Flink, Spark, Openstack, and Libreoffice, those values respectively are 62%, 68%, 55%, and 48%. However, among those contributors, there are developer-reviewers who have made a sufficient number of commits. Those developer-reviewers have enough expertise in the code to potentially contribute a higher number of reviews. Hereafter, we refer to them as low-intensity developer-reviewers. This part of the study shows that each project contains low-intensity developer-reviewers who can be recommended by our system to balance the efforts offered to reviewing.

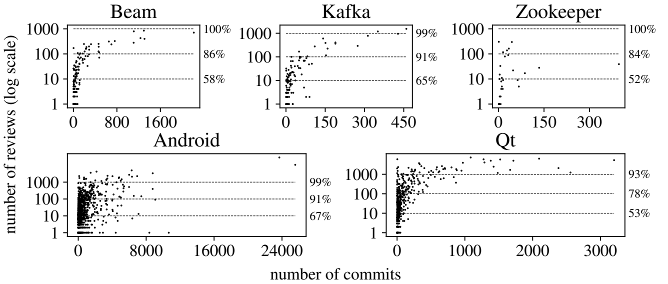

Fig. 4: The joint distribution of commits and reviews made by contributors

Summary: The results of this study reveal three key points. First, each project has an opportunity to increase the number of reviewers from among the plain developers. Even Openstack still has fewer developer-reviewers than plain developers. Second, despite the number of existing developer-reviewers, a few of them make most of the reviews, which leads to an imbalance of the overall efforts offered to reviewing. As the project grows, that trend intensifies, as illustrated by the largest project, Android, in Figure 3. Third, each project has a set of low-intensity developer-reviewers with sufficient experience in the project to perform more reviews.

III. Approach

In this section, we present our recommender-based approach. We base our recommendations on implicit feedback, specifically the existing reviewing and development history. The use of implicit feedback is necessary to satisfy the **R1** requirement of not requiring additional interactions from users. Fortunately, the history of reviews and commits already presents implicit feedback and is easily available. We first present the underlying approach of our recommender system. We then describe our recommendation algorithm itself and explain how the whole system works.

A. Collaborative filtering and matrix factorization

In this section, we present the underlying approach of our recommender system and indicate the requirements met.

To make recommendations, we rely on *collaborative filtering* (CF) [39], [40]. CF requires only information about a user's past behaviour and follows the idea that people who share a similar opinion on certain things may also have a similar opinion about other things. In our setting, we represent opinions in terms of development and reviewing activity. For instance, if a developer A works on files *F1* and *F2*, and another developer B works on files *F1*, *F2*, and *F3*, then A might also be aware of *F3*. Technically, CF deduces unknown relationships, searching for like-minded users with a similar history of activities without requiring additional knowledge about the domain. However, CF suffers from the *cold start* problem because it is unable to reason about items that are totally new to the system. On the other hand, having enough data, CF helps discover new user interests and various behaviour patterns. In our setting, to address the cold start problem, we rely on several heuristics to find similar data already known to the system.

To build a model of our recommender system, we use the Alternating Least Squares (ALS) algorithm [41], [42], [43]. ALS is a model-based CF algorithm belonging to the class of matrix factorization (MF) algorithms [42], [44], [45], [46]. MF algorithms have become widely known since the Netflix Prize Challenge, in predicting user ratings for films [16]. In general, MF algorithms demonstrate high accuracy in settings where the vast majority of items receive little or no feedback from users (*data sparsity* problem) and the total number of users and items continues growing (*scalability* problem). To address those challenges, MF algorithms project the initial data set, represented by the matrix M, into a lower-rank approximation. In the context of ALS, such an approximation is produced by an iterative optimization process that, unlike other MF algorithms, can be implemented in a parallel fashion [41], [47].

ALS, as every MF algorithm, produces two much smaller matrices P and Q that approximate the initial matrix M when multiplied together:

$$M = (r_{ui})_{\substack{u=\{1,\dots,n\} \\ i=\{1,\dots,m\}}} \approx PQ^T,$$

where

$r_{ui} \in \mathbb{R}$	– the number of interactions of the u-th user within the i-th item;
n	– the total number of users;
m	– the total number of items;
$P \in \mathbb{R}^{n \times f}$	– the user-factor matrix;
$Q \in \mathbb{R}^{m \times f}$	– the item-factor matrix;
f	– the number of factors (a predefined constant).

By reducing the dimensionality of our data, we have the effect of representing each user u by an f-dimensional dense vector. As a side effect, this reduction of dimension forces a generalization of the input data, which leads to a better understanding of the data set. The MF process creates a *latent semantic* space where the hidden structure of the data is exposed. Thus, factorization can be viewed as a tool revealing the semantic meaning of the input data through a set of factors. Conceptually, each factor can be associated with an abstract topic. In this way, P contains users' "tastes" in those revealed topics, while the values in Q describe how much of the "topic" each item has.

To make predictions regarding items and their relevance to users, we take a dot product:

$$\hat{r}_{ui} = p_u \cdot q_i, \tag{1}$$

where

$p_u \in \mathbb{R}^f$ – the u-th row of P (user profile);
$q_i \in \mathbb{R}^f$ – the i-th row of Q (item profile).

The meaning of those prediction scores $\hat{r}_{ui}$ depends on the type of feedback used to provide recommendations. In the context of implicit feedback, Equation (1) predicts user interest in items, where higher $\hat{r}_{ui}$ values mean higher interest.

The standard ALS algorithm [42] has been designed for explicit feedback presented by ratings that a user may give to items, e.g., Netflix star ratings, YouTube thumbs-up/down. The use of implicit feedback requires improvements to ALS to consider the nature of zero values that, compared to an explicit zero, do not give strong negative ratings, but simply reflect a lack of work for various reasons: interest, skills, experience, lack of awareness of open pull requests, etc. To adjust ALS for our implicit feedback, we have applied the suggestions of Hu et al. [41]. First, we define a set of binary variables b_{ui}, which in our setting show whether implicit feedback is associated with the u-contributor for the i-file:

$$b_{ui} = \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases},$$

Second, we use the *log scheme* also proposed by Hu et al. [41] to measure the *confidence* of knowledge for the u-th contributor in the i-th file:

$$c_{ui} = 1 + \alpha\, log(1 + r_{ui}/\epsilon). \tag{2}$$

Each contributor receives some minimal confidence for the files where he/she does not have implicit feedback ($r_{ui} = 0$). Indeed, a contributor may not have commits or reviews for a file ($b_{ui} = 0$) but know its purpose, just because the contributor has already read the file, calls its methods somewhere or has reviewed related files. On the other hand, the more the contributor works within certain files ($r_{ui} > 0$), the more the contributor knows the content of those files. Thus, we get higher values of c_{ui} in $b_{ui} = 1$ as soon as we observe an increasing amount of implicit feedback ($r_{ui} \gg 0$) related to the u-th contributor and the i-th file. Using the b_{ui} values, we distinguish the contributors who have some possible knowledge of the file ($b_{ui} = 0$) from those who have actually worked on this file ($b_{ui} = 1$). To control the increase in the confidence of knowledge depending on the amount of implicit feedback, Equation (2) provides hyperparameters α and ϵ, which must be tuned at the validation step.

Taking into account the b_{ui} indicators and the notion of confidence, the ALS cost function used to get a lower-rank approximation of M and, therefore, to produce a latent semantic space is the following [41]:

$$f(P,Q) = \sum_{u,i} c_{ui}(b_{ui} - \hat{r}_{ui})^2 + \lambda\left(\sum_{u} ||p_u||^2 + \sum_{i} ||q_i||^2\right). \tag{3}$$

The iterative optimization process of ALS implies that each iteration gets closer to a factorized representation of the initial matrix M. First, ALS fixes the user-factor matrix P and solves for Q by minimizing (3). Then, it fixes the item-factor matrix Q and solves for P by minimizing (3) similarly. To prevent overfitting and, therefore, to make our model general, ALS also introduces *L2 regularization*, $\lambda(\sum_u ||p_u||^2 + \sum_i ||q_i||^2)$, where λ is a hyperparameter tuned at the validation step [48].

Compliance with the requirements: Using CF as an underlying approach meets the **R2** requirement (development language independence). To provide recommendations, CF only uses a history of activities, such as a history of reviews and development. Thus, a new development language or code structure will not require additional engineering for the system.

The notion of confidence meets the **R3** requirement (exploit contributors' knowledge about related parts of the project to which explicit contribution was not previously made). Low confidence for a file where $b_{ui} = 0$ still leaves the possibility that the recommender system will consider the contributor's experience sufficient to recommend him/her for this or a related file in the future. The final decision depends on how the contributor's history of previous work is similar to that of other contributors.

B. Recommendation algorithm

In this section, we describe our recommendation algorithm that addresses two key points. The first key point is the recommendations of regular reviewers. The second is identifying the appropriate plain developers who may be recommended as new possible reviewers. To address the first key point, we rely on the history of previous reviews. To address the second key point, one possible solution is to use the developer's previous development history. However, a developer who has a lot of commits in a set of files is not necessarily suitable for reviewing them. Reviewing changes in those files may require additional skills, knowledge, experience, etc. Another strategy may be to find a developer with development preferences similar to those of an actual reviewer.

*a) **General approach**:* Technically, to make recommendations of regular and new possible reviewers, our recommender system relies on two matrices M^{rev} and M^{dev} of implicit feedback. M^{rev} contains the number of reviews r^{rev}_{ui} made by the u-th reviewer within the i-th file. M^{dev} contains the number of commits r^{dev}_{ui} made by the u-th developer within the i-th file. By minimizing the ALS cost function (3) over M^{dev}, we respectively produce the reviewer and file profiles: $p^{rev}_u \in P^{rev}$ and $q^{rev}_i \in Q^{rev}$. In the same manner, by minimizing the ALS cost function (3) over M^{dev}, we get the developer and file profiles: $p^{dev}_u \in P^{dev}$ and $q^{dev}_i \in Q^{dev}$. Those p^{rev}_u and p^{dev}_u profiles respectively represent the reviewing and development preferences of the u-contributor. Since plain reviewers do not have development experience, their p^{dev}_u profiles remain empty. Similarly, due to a lack of reviewing experience, plain developers obtain empty p^{rev}_u profiles.

To make recommendations regarding files that may interest developer-reviewers and plain reviewers to review, our recommender system uses Equation (1). The calculations involve the corresponding p^{rev}_u and q^{rev}_u profiles and output $\hat{r}^{rev}_{ui}$ scores, where higher values give more desirable results. To estimate a possible prediction score that a plain developer could get as a reviewer, our recommender system relies on developer-reviewers who bridge the gap between M^{rev} and M^{dev}. Knowing the similarity between development preferences of the v-th developer-reviewer and u-th plain developer, we can estimate for the u-th plain developer and i-th file a possible prediction score similar to (1) as follows:

$$\hat{r}^{rev}_{ui} = \frac{\sum_{v \in U} sim(p^{dev}_u, p^{dev}_v)\ \hat{r}^{rev}_{vi}}{\sum_{v \in U} sim(p^{dev}_u, p^{dev}_v)}, \tag{4}$$

where

U – the set of developer-reviewers;
sim – the similarity function (cosine, Jaccard, etc.).

Equation (4) formalizes our hypothesis (contributors with similar developer preferences may have similar reviewing preferences). Consequently, if the similarity between development preferences of a developer-reviewer and plain developer is high, then a plain developer is likely to have the same prediction score $\hat{r}^{rev}_{ui}$ as a developer-reviewer to review the file. Finally, for a given pull request pr including various files $i \in I_{pr}$, we can calculate an overall prediction score, where higher values indicate a greater relevance of the u-th contributor for pr-th pull request:

$$\hat{r}_{ut} = \sum_{i \in I_{pr}} \hat{r}^{rev}_{ui} \tag{5}$$

b) ***Addressing the cold start problem:*** Making recommendations, our system may encounter a cold start problem when a developer creates a new file and then adds it for reviewing as part of a pull request. This newly created file will not be present in the model and, therefore, it will receive no reviewing recommendations. To avoid this problem, our system replaces the missing files with the ones present in the model using the similarity of file paths. We rely on two criteria for comparing file paths, following Thongtanunam et al. [29]: the *longest common prefix* (LCP) and the *longest common subsequence* (LCSubseq). For instance, if we have two files "doc/source/index.rst" and "doc/source/devref/index.rst", then LCP is "doc/source" and LCSubseq is "doc/source/index.rst". In our context, LCP has a priority over LCSubseq, meaning that our system first compares LCPs and only if they have equal values it involves LCSubseq. In such a way, we primarily replace the missing files with files from the same parent directory.

c) ***Workflow:*** Overall, we obtain a workflow that makes it possible to recommend both regular and new possible reviewers, which meets the **R4** requirement.
First step: Using Equation (1), for each file $i \in I_{pr}$, our recommender system suggests K' reviewers with the highest prediction $\hat{r}^{rev}_{ui}$ scores. If each reviewer has been recommended for only one file within the top-K', we have a maximum of $K'|I_{pr}|$ candidates from among the regular reviewers.
Second step: Our system searches for developer-reviewers in the set of candidates obtained in the first step. If those contributors are not found, the system is only able to recommend regular reviewers but not identify new possible reviewers. Otherwise, for each found developer-reviewer, the system extracts K' plain developers with similar development preferences. To estimate the similarity of development preferences, the system uses cosine similarity regarding the p^{dev}_{ui} profiles.

TABLE II: Statistics on the evaluation data sets

Project	Time Period	# Rev.	# Dev.	# PR	# Commits	# Files
Beam	01-2016 - 01-2020	227	664	10,466	23,663	29,406
Flink	01-2016 - 01-2020	248	648	9,231	11,761	22,246
Kafka	01-2016 - 01-2020	285	599	7,156	5,082	4,349
Spark	01-2016 - 01-2020	572	1,174	16,494	11,843	10,410
Zookeeper	01-2016 - 01-2020	69	140	1,144	651	3,022
Android	10-2008 - 01-2012	94	8,989	5,126	836,324	836,990
Openstack	07-2011 - 05-2012	82	372	6,586	12,628	12,275
QT	05-2011 - 05-2012	202	477	23,810	22,816	92,859
Libreoffice	03-2012 - 06-2014	64	614	6,523	68,976	76,205

Notes: Rev. - reviewers; Dev. - developers

Third step (optional): For identifying new possible reviewers, our system estimates the scores of plain developers extracted in the second step using Equation (4),
Fourth step: Using Equation (5), our system calculates the overall prediction scores of the candidates (i.e., regular reviewers, plain developers) and sorts them in descending order. Now, the first K ($K <= K'$) candidates may be taken to provide the top-K recommendations for the pr-th pull request.

IV. Experimental study

This section presents the evaluation of our approach, including the methodology and the obtained results.

A. Methodology

In this section, we describe the methodology and its key elements used to build the evaluation pipeline. To compare the accuracy of recommendations provided by our system, we have selected several baseline approaches: RevFinder by Thongtanunam et al. [29] and Tie by Xia et al. [30]. These approaches also rely on the history of previously reviewed files and demonstrate high accuracy of recommending regular reviewers.

a) ***Goal and research questions:*** The goal of our study is to evaluate how well our system recommends both regular and new possible reviewers (**R4**), given the remaining requirements **R1**-**R3**. To achieve this goal, we base our evaluation methodology on two research questions (*RQ*).
RQ1: *How well does our system learn the contributors' reviewing preferences and how well is it able to predict regular reviewers on this basis compared to existing solutions?*
RQ2: *Is our system able to use the similarity of development preferences between developer-reviewers and plain developers to expand the set of actual reviewers?*

b) ***Studied projects:*** We have selected the projects studied in Section II-B. From each project, we extract a subset of the data to build an evaluation data set. The subset of the data for the GitHub projects covers the last four years of development, during which those projects used the pull-based model. The subset for Gerrit projects matches in the reviewing history the data set from the study of Thongtanunam et al. [29] and Xia et al. [30]. We use this subset to compare our system with the baseline approaches RevFinder and Tie. Table II provides complete statistics on the constructed evaluation data set.

*c) **Experiment setup**:* As a part of our methodology, we make a simulation similar to the one described by Thongtanunam et al. [29] and Xia et al. [30]. In this setup, we use our system over a selected time interval and check how accurately it recommends the actual reviewers for incoming pull requests. We start by sorting all available pull requests and commits for a selected time interval in the order they appear in the project. Data before a pre-selected date bp initialize the initial training data set D^{train}, the rest belongs to D^{valid}. As for RevFinder and Tie, D^{train} contains information for the first 100 days of the data set. We first feed D^{train} into the ALS model of the recommender system to train it initially. For each pull request of D^{valid}, our system then makes recommendations and collects a given pull request and related commits to update the model later. Every seven days, starting with bp, we fully re-train the model using D^{train}, plus any new collected commits and reviews prior to that time. In this way, we keep our system suitable for doing accurate predictions. In addition, every 180 days, we optimize the model by tuning its hyperparameters.

*d) **Model training and hyperparameter tuning**:* Each time we train the ALS model of our recommender system, we split the initial matrix of implicit feedback M (M^{rev} or M^{dev}) into two parts M^{train} and M^{test} by masking a certain percentage of the $r_{ui} \in M$. For each contributor u who has reviewed or modified more than x files, we randomly select $n\%$ of the r_{ui} values and move them to M^{test}. The matrix M^{train} stores the remaining values. We set n to 15%. To choose x we first rank the contributors by the number of files they have touched, and then set x to the minimum number of files touched by a contributor in the top 25%.

To estimate the quality of model training, we evaluate a scenario where we generate for each contributor an ordered list of the files he/she had not previously reviewed or modified. The list is sorted from the one predicted to be most preferred to the one predicted to be least preferred. The list may contain both files masked in M and those never affected by the developer's commits.

Let us denote by $rank_{ui}$ the percentile-ranking of file i within the ordered list of all files prepared for contributor u. $rank_{ui} = 0\%$ would mean that file i is predicted to be the most suitable for contributor u, thus preceding all other files in the list. On the other hand, $rank_{ui} = 100\%$ indicates that the file i is predicted to be the least preferred for a contributor u, thus placed at the end of the list. Thus, our basic metric for estimating the quality of model training is the *expected percentile ranking* of the file in the test set, which is [41]:

$$\overline{rank} = \frac{\sum_{u,i} r^t_{ui}\, rank_{ui}}{\sum_{u,i} r^t_{ui}}.$$

The lower values of $\overline{rank}$ give more desirable results and represent actual preferences masked in M closer to the top of the recommendation lists. Since for random recommendations the expected value of $rank_{ui}$ is 50%, $\overline{rank} \geq 50\%$ represents a model not better than random. To get lower values of $\overline{rank}$, we need to tune the hyperparameters provided by our model: λ, f, α, and ϵ. In our experiment, we tune these hyperparameters using Bayesian optimization [49], [50] and cross-validation [51], [52].

*e) **Evaluation metrics**:* To estimate the accuracy of recommending regular reviewers, we rely on the *top@k* prediction accuracy and the Mean Reciprocal Rank (*MRR*). Those metrics are commonly used in the current state of the art and make it possible to compare our system with the baseline approaches.

Top@k prediction accuracy calculates the proportion of pull requests for which the recommender system proposed at least one actual reviewer in the top-k results. For example, an accuracy value of 0.75 in the top-10 indicates that for 75% of the pull requests at least one actual reviewer is found in the top 10 results. Given a pull request pr, if at least one of its top-k reviewers is an actual reviewer, we consider that the reviewers were recommended correctly, and set the value of $isRecomm(pr, top\text{-}k)$ to 1; otherwise, we set $isRecomm(pr, top\text{-}k)$ to 0. For a set of pull requests R, the top@k prediction accuracy is computed as:

$$top@k(R) = \frac{\sum_{pr \in R} isRecomm(pr, top\text{-}k)}{|R|} \qquad (6)$$

To compare the accuracy of recommendations with those of RevFinder and Tie, we choose k to be 1, 3, 5 and 10.

Mean Reciprocal Rank (*MRR*) [53] calculates the average of the reciprocal ranks of the actual reviewers in the list of recommendations. Given a pull request pr, its reciprocal rank is the multiplicative inverse of the rank of the first actual reviewer in the list of recommendations. MRR is the average of the reciprocal ranks of pull requests in a set of pull requests R:

$$MRR = \frac{1}{|R|} \sum_{pr \in R} \frac{1}{rank(pr)} \qquad (7)$$

In the above equation, $rank(pr)$ refers to the rank of the first correctly recommended reviewer for the pr-th pull request.

B. Results

In this section, we present the results of our experimental study regarding the research questions. For each research question, we present its approach and results.

RQ1: *How well does our system learn the contributors' reviewing preferences and how well is it able to predict regular reviewers on this basis compared to existing solutions?*

Approach. To answer **RQ1**, we launch the presented evaluation pipeline and iteratively estimate the top@k prediction accuracy and MRR over D^{valid}. For comparison with RevFinder and Tie, we rely on the metrics presented in their latest studies [29], [30].

Results. Table III shows the top@k accuracy and MRR obtained by our recommender system with the last pull request of D^{valid}. Since RevFinder and Tie do not involve GitHub projects in their studies, the corresponding values in Table III

TABLE III: Top@k prediction accuracy and MRR of the recommender system (RS) compared with Tie and RevFinder (Rev.)

Projects	**Top-1**			**Top-3**			**Top-5**			**Top-10**			**MRR**		
	RS	**Tie**	**Rev.**	**RS**	**Tie**	**Rev.**	**RS**	**Tie**	**Rev.**	**RS**	**Tie**	**Rev.**	**RS**	**Tie**	**Rev.**
Beam	0.21	-	-	0.45	-	-	0.60	-	-	0.77	-	-	0.32	-	-
Flink	0.28	-	-	0.50	-	-	0.61	-	-	0.74	-	-	0.34	-	-
Kafka	0.30	-	-	0.59	-	-	0.72	-	-	0.88	-	-	0.34	-	-
Spark	0.23	-	-	0.51	-	-	0.66	-	-	0.83	-	-	0.29	-	-
Zookeeper	0.36	-	-	0.67	-	-	0.77	-	-	0.85	-	-	0.42	-	-
Android	0.42	**0.57**	0.46	0.60	**0.81**	0.71	0.68	**0.87**	0.79	0.73	**0.92**	0.86	0.52	**0.70**	0.60
Openstack	0.36	**0.43**	0.38	0.66	**0.73**	0.66	0.78	**0.83**	0.77	0.87	**0.91**	0.87	0.44	**0.60**	0.55
QT	**0.37**	0.30	0.20	**0.63**	0.45	0.34	**0.71**	0.52	0.41	**0.78**	0.62	0.69	**0.51**	0.41	0.31
Libreoffice	0.20	**0.76**	0.24	0.36	**0.91**	0.47	0.44	**0.93**	0.59	0.58	**0.96**	0.74	0.31	**0.84**	0.40

are omitted. Our recommender system provides accuracy over 70% in the top-10 across all the projects. Although Tie generally has better accuracy in the top-10 than our recommender system, Tie has a noticeable decrease in accuracy for QT. This may occur due to the increasing number of reviewers and pull requests. In this sense, QT is the largest among the four Gerrit projects in our experiment.

The worst results obtained by our system are for Libreoffice. At the beginning of our experiment, $\overline{rank}$ is almost 50%, which indicates that the model suffers from lack of data and cannot infer the reviewing preferences of actual reviewers. Over time, the amount of data iteratively grows, $\overline{rank}$ reaches 24% and, as a result, the accuracy of recommendations increases. Another reason for the low accuracy for Libreoffice concerns the data set provided by Thongtanunam et al. [29]. Their data set appears to have fewer pull requests and actual reviewers compared to the same subset of our data set presented in Section II-B. Overall, our system shows lower results than Tie and RevFinder. However, compared to those solutions, our system has both regular and new possible reviewers in the list of recommendations, to expand the number of reviewers.

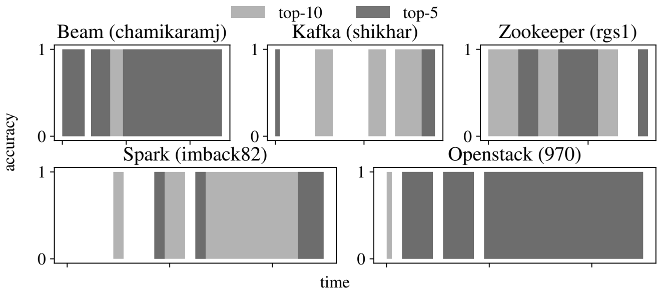

Fig. 5: Tracking developer-reviewers

RQ2: *Is our system able to use the similarity of development preferences between developer-reviewers and plain developers to expand the set of actual reviewers?*

Approach. To answer **RQ2**, we use our evaluation pipeline to track plain developers who over time become developer-reviewers. We estimate how many of them are recommended for their first review, based on the similarity of development preferences. We then continue to track these developer-reviewers and estimate how often they appear in the top-5 and top-10 recommendations for their actual pull requests.

Results. Our recommender system has found 466 plain developers who over time become developer-reviewers. Figure 5 shows the developer-reviewers of five projects selected as the most representative: Beam, Kafka, Zookeeper, Spark, and Openstack. Each developer-reviewer, except for Spark, has been recommended for the first review based on the similarity of development preferences. Once a developer-reviewer makes the first review, our system then relies on his/her reviewing preferences to make recommendations. Figure 5 illustrates that with the increasing number of made reviews, the developer-reviewers begin to appear more in the top-10 and top-5. Overall, the recommender system suggested 255, 179, 83, and 356 plain developers for Android, Openstack, QT, and Libreoffice. For Beam, Flink, Kafka, Spark, and Zookeeper, the recommender system suggested 223, 282, 127, 138, and 87 plain developers.

V. Related work

The problem of finding an appropriate code reviewer has been considered by many studies. Lipcak et al. [54] have categorized the most relevant into four groups based on the features and techniques they use to recommend reviewers: heuristic-based, machine learning-based, social network-based, and hybrid approaches. *Heuristic-based* approaches process historical data and use various heuristics to find relevant reviewers [55], [29], [30], [56], [57], [58]. *Machine learning-based* approaches use machine learning techniques and require building a model based on a training set [59]. *Social network-based* approaches identify various relationships among contributors to suggest reviewers [60], [61], [62]. *Hybrid* approaches rely on various combinations of techniques [63], [64], [65], [66].

Most of the presented heuristic-based approaches rely only on the history of previous reviews and, therefore, do not include plain developers in the list of recommendations. For instance, RevFinder [29] relies on the history of previously reviewed file paths and their similarity. Tie [30] extends RevFinder, additionally processing the textual information of previously completed reviews. cHRev [56] and WRC [58] build a reviewer's expertise based on the past review contributions.

CORRECT [57] is a heuristic solution based on the idea that if a previous pull request used some similar external library or technology as the current pull request, then the reviewers of the past pull request are also good candidates for the current one. Compared to the other solutions, CORRECT allows inviting reviewers from outside the target project. However, CORRECT does not involve plain developers of the target project. Another heuristic solution ReviewBot [55] suggests experts who in the past have already modified or reviewed changed lines of source code. This solution may potentially be adapted to include plain developers.

Social network-based approaches analyze relationships among all contributors and provide recommendations, potentially including data on plain developers. For instance, Yu et al. [60], [61] provide several recommender systems built around analyzing a network of contributors and their comments on pull requests. In contrast, Liao et al. [62] construct a network of collaborators and pull request topics.

Xia et al. [65] propose a hybrid solution similar to our recommender system in terms of the used techniques. To propose recommendations, they combine latent factor models and neighbourhood methods, capturing implicit relationships among reviewers. However, their model contains only the history of previous reviews and, therefore, cannot extend the set of actual reviewers.

Overall, those studies that potentially include data on plain developers are primarily focused on recommending regular reviewers. At the same time, Kovalenko et al. [67] demonstrate that existing approaches do not provide helpful information for most code change authors, as these approaches suggest reviewers who are obvious to authors in advance. Our approach provides a machine learning model that recommends both regular and new possible reviewers and thus, is able to promote knowledge sharing and potentially balance the distribution of efforts offered to reviewing.

VI. Discussion

a) ***Integration in the software development process:*** We consider how our recommender system could be integrated into a software development process. One strategy is to link our system with existing tools such as GitHub or Gerrit. In our experimental study (Sec. IV), we presented a pipeline that could be adapted to a production-ready solution. Once our system is installed, it could help authors of pull requests. When an author opens a pull request, our system may offer a list of regular and new potential reviewers. Even if some of the potential reviewers do not yet have write permissions to the repository, they can still take part in the conversation and suggest changes. Another possible use of our system is to increase the number of reviewers in an already open pull request. The assigned reviewers may need to find another expert in order to get more feedback on the change. Our recommender system could propose such a contributor.

b) ***Threats to validity:*** A threat to the *internal validity* of our results is related to the data set obtained through the GitHub and Gerrit APIs. We found that developers could potentially use several accounts, causing our recommender system to recognize them as different persons. In addition, some projects use service bots to automate various development processes, and our system treats them the same as real contributors, if they make changes to files. Overall, this could have some impact on the quality of recommendations. However, due to CF, the impact should be minimal if there is enough data from other contributors.

A threat to *external validity* relates to the generalization of our results. Although we use nine projects of different sizes and characteristics in our evaluation, future work may involve more projects to better identify the impact of various factors, such as the number of files, contributors, on the accuracy of recommendations in general.

A threat to *construct validity* relates to the suitability of our evaluation metrics. Following past studies, we use the top@k prediction accuracy and MRR. Our goal, however, is not to improve on these specific numbers, but rather to validate our expanded set of reviewers. However, according to Kovalenko et al. [67], in practice, this can add value to the recommender system over the current state of the art. Future work should be focused on estimating the impact of recommending new possible reviewers.

VII. Conclusion

In this paper, we have proposed a recommender-based approach to help open-source projects expand the number of reviewers from among the appropriate developers. The key advance of our system over existing work is the ability to recommend both regular reviewers and new possible reviewers based on their past review and commit history. By recommending new possible reviewers, our system could potentially reduce the burden on the few contributors that typically do most of the reviews. In addition, this promotes knowledge sharing among the contributors.

In Section II-C, we present an exploratory study demonstrating the relevance of our work. The evaluation (Sec. IV) shows that all nine projects could benefit from our system. It can be integrated into the ongoing development processes, iteratively trained and used for making online recommendations.

In future work, we plan to extend the number of studied projects in order to generalize the results. At the same time, we will enhance the evaluation of our system and investigate the impact of expanding the number of reviewers in practice by an additional empirical study. We will also compare our approach with the approach proposed contemporaneously by Mirsaeedi et al. [68], which focuses on recommending additional reviewers in order to distribute knowledge about the code base and to mitigate developer turnover.

Availability: The data set is available on Zenodo,[1] The recommender system is archived by Software Heritage[2] and is also hosted on GitHub;[3]the authors welcome any contributions.

[1] https://doi.org/10.5281/zenodo.3998437

[2] https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/lawall/icsme2020.git

[3] https://github.com/alexchueshev/icsme2020

References

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 712–721.

[3] "Gerrit code review," https://www.gerritcodereview.com/, 2020, accessed: 2020-05-07.

[4] Github, https://github.com/, 2020, accessed: 2020-05-07.

[5] CodeFlow, https://www.getcodeflow.com/, 2020, accessed: 2020-05-07.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *11th Working Conference on Mining Software Repositories*, ser. MSR, 2014, pp. 202–211.

[7] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18, 2018, pp. 181–190.

[8] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects," in *11th Working Conference on Mining Software Repositories*, ser. MSR, 2014, pp. 192–201.

[9] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 541–550.

[10] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1028–1038.

[11] D. Vetter, "Intel Open Source Technology Center – Patch Review," https://blog.ffwll.ch/slides/review-training-public.pdf, 2014, version: 1.3. Accessed: 2019-08-22.

[12] Apache Spark , https://spark.apache.org/contributing.html, 2020, accessed: 2020-05-07.

[13] Apache Kafka , https://kafka.apache.org, 2020, accessed: 2020-05-07.

[14] I. Steinmacher, G. Robles, B. Fitzgerald, and A. Wasserman, "Free and open source software development: the end of the teenage years," *Journal of Internet Services and Applications*, vol. 8, no. 1, p. 17, Dec 2017.

[15] L. F. Dias, I. Steinmacher, and G. Pinto, "Who drives company-owned OSS projects: internal or external members?" *Journal of the Brazilian Computer Society*, vol. 24, no. 1, pp. 16:1–16:17, Dec 2018.

[16] J. Bennett and S. Lanning, "The Netflix prize," in *KDD Cup Workshop 2007*. New York: ACM, Aug. 2007, pp. 3–6. [Online]. Available: http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf

[17] J. Y. Chung and M. J. Kim, "Music recommendation model by analysis of listener's musical preference factor of K-pop," in *2018 International Conference on Information Science and System*, ser. ICISS '18. New York, NY, USA: ACM, 2018, pp. 8–11. [Online]. Available: http://doi.acm.org/10.1145/3209914.3209932

[18] Q. Han, M. Ji, I. Martinez de Rituerto de Troya, M. Gaur, and L. Zejnilovic, "A hybrid recommender system for patient-doctor matchmaking in primary care," in *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, 2018, pp. 481–490.

[19] W. Guo, H. Gao, J. Shi, B. Long, L. Zhang, B.-C. Chen, and D. Agarwal, "Deep natural language processing for search and recommender systems," in *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19, 2019, pp. 3199–3200.

[20] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec++: An enhanced tag recommendation system for software information sites," *Empirical Software Engineering*, vol. 23, no. 2, pp. 800–832, Apr 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9533-1

[21] J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides, "Resdec: Online management tool for implementation components selection in software product lines using recommender systems," in *23rd International Systems and Software Product Line Conference - Volume B*, ser. SPLC '19, 2019, pp. 63:1–63:4.

[22] Apache Beam, https://beam.apache.org/, 2020, accessed: 2020-05-07.

[23] Apache Flink , https://flink.apache.org/, 2020, accessed: 2020-05-07.

[24] Apache Zookeeper, https://zookeeper.apache.org/, 2020, accessed: 2020-05-07.

[25] Android, https://android-review.googlesource.com/, 2020, accessed: 2020-05-01.

[26] Openstack, https://review.opendev.org/, 2020, accessed: 2020-05-01.

[27] QT, https://codereview.qt-project.org/, 2020, accessed: 2020-05-01.

[28] Libreoffice, https://gerrit.libreoffice.org/, 2020, accessed: 2020-05-01.

[29] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 141–150.

[30] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 261–270.

[31] Apache Software Foundation, "Statisics," https://projects.apache.org/statistics.html, 2020, accessed: 2020-01-05.

[32] Apache Hadoop , https://hadoop.apache.org/, 2020, accessed: 2020-05-07.

[33] Apache HBase, https://hbase.apache.org/, 2020, accessed: 2020-05-07.

[34] Apache Hive, https://hive.apache.org/, 2020, accessed: 2020-05-07.

[35] GitHub Help, "About pull requests," https://help.github.com/en/articles/about-pull-requests, 2020, accessed: 2020-05-07.

[36] GitHub Developer, "Graphql api v4," https://developer.github.com/v4/, 2020, accessed: 2020-05-07.

[37] RevFinder, "Replication data," https://github.com/patanamon/revfinder, 2020, accessed: 2020-05-01.

[38] Gerrit API, "Gerrit code review - /changes/ rest api," https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html.

[39] P. Resnick and H. R. Varian, "Recommender systems," *Commun. ACM*, vol. 40, no. 3, pp. 56–58, Mar. 1997. [Online]. Available: https://doi.org/10.1145/245108.245121

[40] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2010.

[41] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *2008 Eighth IEEE International Conference on Data Mining*, Dec. 2008, pp. 263–272.

[42] A. Cichocki, R. Zdunek, A.-H. Phan, and S.-i. Amari, *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation*, 10 2009.

[43] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," in *4th International Conference on Algorithmic Aspects in Information and Management*, ser. AAIM '08, 2008, pp. 337–348.

[44] L. Baltrunas, B. Ludwig, and F. Ricci, "Matrix factorization techniques for context aware recommendation," in *Fifth ACM Conference on Recommender Systems*, ser. RecSys '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 301–304. [Online]. Available: https://doi.org/10.1145/2043932.2043988

[45] J. Kawale, H. Bui, B. Kveton, L. T. Thanh, and S. Chawla, "Efficient thompson sampling for online matrix-factorization recommendation," in *28th International Conference on Neural Information Processing Systems – Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1297–1305.

[46] B. Yi, X. Shen, H. Liu, Z. Zhang, W. Zhang, S. Liu, and N. Xiong, "Deep matrix factorization with implicit feedback embedding for recommendation system," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 8, pp. 4591–4601, Aug 2019.

[47] M. Winlaw, M. B. Hynes, A. Caterini, and H. D. Sterck, "Algorithmic acceleration of parallel als for collaborative filtering: Speeding up distributed big data recommendation in Spark," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 682–691.

[48] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," in *Twenty-First International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 78. [Online]. Available: https://doi.org/10.1145/1015330.1015435

[49] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, ser. NIPS'11. Red Hook, NY, USA: Curran Associates Inc., 2011, pp. 2546–2554.

[50] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. H. Hoos, and K. Leyton-brown, "Towards an empirical foundation for assessing Bayesian optimization of hyperparameters," in *In NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.

[51] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence – Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[52] D. I. Ignatov, J. Poelmans, G. Dedene, and S. Viaene, "A new cross-validation technique to evaluate quality of recommender systems," in *Perception and Machine Intelligence*, M. K. Kundu, S. Mitra, D. Mazumdar, and S. K. Pal, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 195–202.

[53] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[54] J. Lipcak and B. Rossi, "A large-scale study on source code reviewer recommendation," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 378–387.

[55] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 931–940. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486915

[56] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, June 2016.

[57] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: Code reviewer recommendation in GitHub based on cross-project and technology experience," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 222–231.

[58] C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn, "Automatically recommending code reviewers based on their expertise: An empirical comparison," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 99–110. [Online]. Available: https://doi.org/10.1145/2970276.2970306

[59] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," in *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006)*, 2009, pp. 1–18.

[60] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer recommender of pull-requests in GitHub," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 609–612.

[61] ——, "Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1, 2014, pp. 335–342.

[62] Z. Liao, Z. Wu, Y. Li, X. Fan, and J. Wu, "Core-reviewer recommendation based on pull request topic model and collaborator social network," *Soft Computing*, 07 2019.

[63] J. Jiang, J.-H. He, and X.-Y. Chen, "Coredevrec: Automatic core member recommendation for contribution evaluation," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 998–1016, Sep 2015. [Online]. Available: https://doi.org/10.1007/s11390-015-1577-3

[64] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, 01 2016.

[65] Z. Xia, H. Sun, J. Jiang, X. Wang, and X. Liu, "A hybrid approach to code reviewer recommendation with collaborative filtering," in *2017 6th International Workshop on Software Mining (SoftwareMining)*, 2017, pp. 24–31.

[66] C. Yang, X.-h. Zhang, L.-b. Zeng, Q. Fan, T. Wang, Y. Yu, G. Yin, and H.-m. Wang, "Revrec: A two-layer reviewer recommendation algorithm in pull-based development model," *Journal of Central South University*, vol. 25, no. 5, pp. 1129–1143, May 2018. [Online]. Available: https://doi.org/10.1007/s11771-018-3812-x

[67] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?" *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 710–731, 2020.

[68] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1183–1195.