J Intell Inf Syst (2018) 50:597–619
DOI 10.1007/s10844-017-0484-1

Profile based recommendation of code reviewers

Mikołaj Fejzer[1] · Piotr Przymus[1] · Krzysztof Stencel[2]

Received: 16 November 2016 / Revised: 31 July 2017 / Accepted: 6 August 2017 /
Published online: 15 August 2017
© The Author(s) 2017. This article is an open access publication

Abstract Code reviews consist in proof-reading proposed code changes in order to find their shortcomings such as bugs, insufficient test coverage or misused design patterns. Code reviews are conducted before merging submitted changes into the main development branch. The selection of suitable reviewers is crucial to obtain the high quality of reviews. In this article we present a new method of recommending reviewers for code changes. This method is based on profiles of individual programmers. For each developer we maintain his/her profile. It is the multiset of all file path segments from commits reviewed by him/her. It will get updated when he/she presents a new review. We employ a similarity function between such profiles and change proposals to be reviewed. The programmer whose profile matches the change most is recommended to become the reviewer. We performed an experimental comparison of our method against state-of-the-art techniques using four large open-source projects. We obtained improved results in terms of classification metrics (precision, recall and F-measure) and performance (we have lower time and space complexity).

Keywords Code review · Mining software repositories · Reviewer recommendation

✉ Mikołaj Fejzer
mfejzer@mat.umk.pl

Piotr Przymus
eror@mat.umk.pl

Krzysztof Stencel
stencel@mimuw.edu.pl

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland

[2] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland

1 Introduction

Software quality is one of the main concerns of software engineering. Numerous systematic approaches to assure this quality exist and are applied. Among industry accepted practices there are *peer code reviews*. They help to obtain a high quality code, better adjusted test coverage, consistent adherence to language standards and lower number of bugs. The knowledge about the project can be reinforced by code reviews (Nelson and Schumann 2004). However peer code reviews are time-consuming activities that require good cooperation of both the reviewer and the author of the reviewed code. Both participants will benefit if the time spent between a code change (a *commit*) and the start of the review process is decreased. Commits without assigned reviewers take the longest time to be merged due to the lack of both interest and proper domain knowledge (Drake et al. 1991; Jeong et al. 2009). Furthermore, some reviewers can be overburdened due to their specific focus on a sole aspect of a project. Moreover, there is also a problem of proper reviewer selection. Therefore, quick and accurate reviewer selection is the key success factor of peer code reviewing.

In this article, we focus on the automation of the code reviewer selection. Various tools and algorithms were proposed to solve this problem. Thongtanunam et al. (2015) propose measuring the similarity of commit file paths to match best reviewers, thus creating reviewer recommendation tool *Revfinder*. Balachandran designed *Review Bot* (Balachandran 2013) which uses repository annotations to assign authors of modified files as reviewers. Yu et al. (2014) and Yu et al. (2016) propose to assign reviewers via measuring the cosine similarity between the last reviews done by them and utilization of social network between reviewers sharing common interests. In this article we propose a new algorithm that avoids shortcomings related to previous methods, such as high complexity of file path comparison in *Revfinder* and low accuracy in *Review Bot*.

This article makes the following contributions:

- we propose a novel method to nominate reviewers in the peer reviewing process;
- we perform thorough experimental evaluation of this method;
- we compare this methods with state-of-the-art techniques to prove its quality.

The article is organized as follows. Section 2 addresses the related work. Section 3 defines the problem. Section 4 presents our algorithm. Section 5 describes dataset and metrics used for experimental evaluation. Section 6 shows the results of an experimental evaluation of the proposed method and describes significance of our findings. Section 7 concludes and rolls out possible future directions of research.

2 Background and related work

The practice of peer code reviews understood as reading and evaluating quality of code changes by other developers is known from 1960s (Bisant and Lyle 1989). Since that time it has played an important role in the life cycle of numerous projects. When done properly, it may help spreading knowledge among developers and encouraging the emergence of code standards for the project (Nelson and Schumann 2004). It also affects the final quality of the project as better code review coverage results in less post-release defects (McIntosh et al. 2014). Therefore, it is not surprising that the code review is commonly applied in top software projects, like Linux or Android operating systems, software frameworks such

as Akka, Mozilla, Qt or Spring, libraries like SciPy, software platforms like OpenStack or OpenShift, databases like PostgreSQL, standalone applications such as LibreOffice or projects maintained by open source foundations like Apache or Eclipse.

Advances in distributed version control systems encouraged easy adaptation of code reviews into the lifecycle of a project. A developer may work on a change set on his/her private repository and later ask for a review, before putting the change into the main branch of the project. There are various approaches when it comes to adapting code reviews into the lifecycle of a project. A well organized mailing list, as seen in Linux Kernel development, may be more than sufficient. Some may prefer automated code review systems, like Gerrit, GitLab, Phabricator, Review Board, Rietveld or Rhodecode, that can be configured to enable merging source code changes into the main repository only when the code review is successful.

Social coding platforms, like GitHub or Bitbucket, use a pull request model. In this model a contribution is done by: (1) forking the selected repository, (2) creating changes locally, (3) adding these changes to the remote fork, and finally (4) creating a pull request between the fork and the original repository. A pull request usually undergoes a code review before being accepted into the main repository. Due to the possibility to invite programmers to review code across all projects hosted on the platform, the number of potential participants is significantly higher than in a traditional code review system like Gerrit.

A code review process takes a noteworthy amount of time and effort of the reviewer. Moreover, it requires a good cooperation between the author and the reviewer. Both parties will benefit, if the overall code review time is reduced. However, commits without pre-assigned reviewers will take longer to be reviewed (Drake et al. 1991; Jeong et al. 2009). As a consequence, they will wait longer before being included into the main repository. Naïve solutions like random assignment of reviews or round-robin queuing of the reviewers also have certain limitations. If a reviewer nomination is inadequate (i.e. beyond expertise of the reviewer), the quality of the review may suffer from Parkinson's law of triviality. According to the law, a reviewer who scrutinizes a code lying outside his/her knowledge focuses on unimportant details (Rigby and Storey 2011). Furthermore, according to Nelson and Schumann (2004), it is difficult for all reviewers to spot performance problems, race conditions or synchronization issues. Fortunately, some automated tools may aid developers in such cases (Nelson and Schumann 2004). They provide fully automated solutions for obvious cases or semi-automatic solutions that annotate suspected code and/or suggest a domain expert who should review the code. As a consequence quick and accurate reviewer recommendation is an important code review aspect. This may be done manually, e.g. Linux kernel uses a mailing list where experts receive broadcast about reviews matching specific domain (Rigby and Storey 2011), or automatic. This paper focuses on the latter variant.

Mining software repositories is a relatively young research area focused mostly on mining pieces of information that possibly lead to better quality of software projects (Kagdi et al. 2007). In consequence, it has caused a new research trend which inspected the impact of those methods on various aspects of a software project lifecycle (Kagdi et al. 2007; Rigby and Storey 2011). Among a variety of the applications we may list bug prevention and prediction, development process guidance, team dynamics analysis, contribution detection, code reuse pattern analysis, coupling detection, architecture management and refactorisation support (Radjenovic et al. 2013; D'Ambros et al. 2010; Kim et al. 2008; Hassan 2008).

An important group of methods in this field focuses on measuring the similarity of entities stored in the repositories. Radjenovic et al. (2013) use code metrics in a system that focused on bug prevention and bug detection. When the techniques got enriched with the analysis of text data (D'Ambros et al. 2010), new possibilities of research appeared, such as combined approaches using both metrics and text processing (Kim et al. 2008). Due to the need of large dataset analysis it is common to use information retrieval and machine learning algorithms (Kagdi et al. 2007). One of the greatest challenges in software repository mining is addressing the problem of integrating multiple data sources, such as code repositories, issue trackers, bug databases and archived communications (Hassan 2008).

The validation of the research is usually conducted with standardized and preprocessed datasets containing information from repositories and code reviews systems. Here is a list of datasets known to the authors of this article:

- HTorrent (Gousios 2013) includes all events related to all projects hosted on Github platform, including interconnected data about pull requests, issues and users, captured from 2013 onwards.
- Hamasaki dataset (Hamasaki et al. 2013) includes data on commits and reviews from the following projects: Android, Chromium, OpenStack and Qt. This dataset includes logs of events created by Gerrit code review system, the complete description of event model and commit metadata.
- Thongtanunam dataset (Thongtanunam et al. 2015) is a modified version of Hamasaki et al. (2013). It contains LibreOffice instead of Chromium.

2.1 Reviewer assignment problem

In general, the election of reviewers for documents of various kinds is known as the Reviewer Assignment Problem (Wang et al. 2008). Disparate models have been proposed in this area. They are based on information retrieval, natural language processing, graph analysis, recommender systems, supervised learning and clustering (Wang et al. 2008; Conry et al. 2009). In particular, Latent Semantic Indexing as a similarity function between potential reviewer biographies and documents was used by Dumais and Nielsen (1992). Yarowsky and Florian (1999) proposes the usage of cosine similarity between vectors of words obtained from papers published by candidate reviewers and from the paper be reviewed. He uses them to train the Naïve Bayes classifier. An approach based on collaborative filtering was proposed by Conry et al. (2009). He uses the cosine similarity of abstracts for manuscript-to-manuscript likeness and the number of co-authored papers for reviewer-to-reviewer correspondence. Another recommender system known as the Toronto Paper Matching System (Charlin and Zemel 2013) uses a similarity function defined as the dot product of topic vectors, obtained via Latent Dirichlet Allocation from a manuscript and candidate reviewers' publications. Insights on mathematical constraints and efficiency of recommender systems were discussed by Xie and Lui (2012). He concludes that heterogeneous review strategies result in similar or lower workload, but offer higher accuracy. Tang et al. (2010) creates a convex cost network. Edges of such a network describe various constraints, while nodes correspond to experts and queries. The selection of experts is based on finding the maximal flow between selected nodes. Bootstrapping can be done using traditional similarity measures. Liu et al. (2014) proposes a model using Random Walk with Restart to traverse the graph of potential reviewers. The information on authority,

expertise and diversity are used to build this network. This method outperforms models based on text similarity and topic similarity. In our method presented in this article we are not inspired directly by any of those approaches. However, we build the method from well-known elements (like user and item profiles) and adapt them to new contexts.

2.2 Automatic reviewer recommendation systems

Current approaches to the problem of finding appropriate reviewers are based on comparisons of various entities that quantify persons and the changed code.

The authors of "Who Should Review My Code" (Thongtanunam et al. 2015) suggest using the file paths of the project. They created a tool called *Revfinder* which uses (1) the information about modified file paths in a commit, and (2) string comparison functions to compute the file path similarity between a new commit and previous reviews of each of the candidate reviewers. The candidate with the maximum score is recommended to become the reviewer. This tool was tested on the Hasamaki dataset (Hamasaki et al. 2013) expanded to contain LibreOffice.

A different tool called *Review Bot* is used in VMware (Balachandran 2013). The proposed algorithm examines each modified line in a code change in a similar manner to `git blame` or `svn annotate` commands. Each author who worked on a given source code change is awarded points. Authors of more recent changes gain more points than authors of older changes. The final sum for each author is then used to elect top k authors recommended to become reviewers.

The authors of "Automatically Prioritizing Pull Requests" (van der Veen et al. 2015) try to solve a slightly different problem. However, their results are still significantly related to the recommendation of reviewers. The goal of (van der Veen et al. 2015) is to find which pull request is more worthy of reviewers' attention. It is based on the prediction which pull request shall receive the users' attention in a specific time window. Authors used 475 projects from GHTorrrent dataset (Gousios 2013). Three algorithms: Logistic Regression, Naïve Bayes and Random Forests were trained and evaluated using 90% training and 10% testing data with 10-fold cross-validation. Random Forests obtained 86% accuracy on average across all projects. It was the highest score among the evaluated methods.

In "Reviewer Recommender of Pull-Requests in GitHub" (Yu et al. 2014) the problem of reviewer selection was addressed in terms of similarity usage, an *expertise score* and interests shared between developers. The interest sharing information is stored in a *comment network*. A model based on the vector space is used to check the cosine similarity between pull requests. Then, each developer is assigned an *expertise score*. For each project a undirected graph called *comment network* of interaction between programmers is maintained. Its vertices represent programmers. An edge between two programmers exists, if they commented on the same pull request. Edges are weighted by the number of such commonly commented pull requests. All new pull requests are scored using the *expertise score* and then assigned to reviewers with similar interests as found in the *comment network*. The method was evaluated on GHTorrrent dataset (Gousios 2013) narrowed to 10 projects with more than 1000 pull requests. It has been revealed that an increase of the number of recommended reviewers lowers the precision and boosts the recall. If one reviewer is recommended, the precision is 0.71 and the recall equals 0.16. On the other hand, when 10 top reviewers are recommended, they the precision is 0.34 and the recall is 0.71.

A further extension of the results from Yu et al. (2014) was proposed in "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug

assignment?" (Yu et al. 2016). The authors utilize the information on developer interactions in the *comment network* (Yu et al. 2014) as a standalone method and also as a fusion with other methods. In order to select n reviewers, a breadth-first search is started on the *comment network* from the author of the pull request.

The authors tested their algorithm on 84 open-source projects against three other approaches: (1) state of the art FL-based system *Revfinder* (Thongtanunam et al. 2015), (2) an information retrieval algorithm based on the cosine similarity between vectors of words from each pull request, and (3) support vector machines trained to recognize pull requests reviewed in the past by a particular reviewer. The method based on comments networks had similar precision and recall as *Revfinder*. To improve the results authors also experimented with fusion models that combined their solution with other state-of-the-art techniques. As a result they selected the model that combined information retrieval techniques with comment networks. This model outperformed known models.

3 Problem statement

In a typical code review system a version control facility issues a list of commits to be scrutinized. There are various approaches on how to present this list of commits for a potential reviewer. The simplest approach is to display the list of all available commits together with some decision supporting tools (e.g. search and filter). The downside of this approach is that it lengthens the review process. This may lead to delaying the whole process or omitting some commits indefinitely (van der Veen et al. 2015).

In order to shorten the process, usually a review recommendation system is used. The idea of such a system is to prepare a sorted list of commits that fits the field of expertise of the reviewer, starting with best matching commits. Such systems may be based on human suggestions of who should review a commit and/or on an automatic system that learns about reviewers' preferences and their fields of expertise by analyzing the history of reviews. In this article we concentrate on automatic reviewer recommendation systems.

In such a system for each new (not yet reviewed) commit and for each prospective reviewer a matching score is computed. The higher is this score the better is the match between the reviewer and the commit. The input to the computation of this score is the commit at hand and the past work of the potential reviewer. When a new review appears in the system it is included in the reviewer's past work.

An automatic reviewer section system, being the subject of our research, nominates reviewers for incoming commits using the information on the reviewers' past work and on the commit itself.

3.1 Notations

Let C be a time ordered stream of commits. By C_t we will denote the commit that appears at the time t in the commit stream C. As a whole, each commit carries a source code diff and additional meta information. We are interested only in the list of names of modified files. By $f(C_t) = [f_1^t, f_2^t, \ldots, f_{l_t}^t]$ we will denote the list, where f_i^t is the name of the i-th file modified by the commit C_t for $i = 1, \ldots, l_t$, where l_t is the number of files in the commit C_t. More precisely, f_i^t is a path, i.e. the string representing the file location in the operating system. This string will also be considered as a sequence of words separated by a special character depending on the underlying operating system. Additionally, let $R = [R_1, \ldots, R_n]$ be the list of candidate reviewers.

Let $h(t)$ be the set of commits up to time t. Moreover, let us put $h_r(t)$ as the set of commits that at the time t have already been reviewed, and $h_r(t, R_j)$ as a set of commits that at the time t have already been reviewed by the reviewer R_j.

$$\begin{aligned} h(t) &:= \{C_i : i < t\} \\ h_r(t) &:= \{C_i : i < t \wedge C_i \text{ is already reviewed at } t\} \\ h_r(t, R_j) &:= \{C_i : i < t \wedge C_i \in h_r(t) \wedge C_i \text{ is reviewed by } R_j \text{ at time } t\} \end{aligned}$$

We aim to build a similarity function $s\colon C \times 2^C \to \mathbb{R}$. This function for a commit C_{t+1} and a reviewer's history $h_r(t, R_i)$ quantifies how this commit matches this reviewer's history. A larger value of $s\,(C_{t+1}, h_r\,(t, R_i))$ indicates a better match. In this article we assume that a commit is represented by the list of modified files, and that at any given time t for a reviewer R_k the set of his/her past reviews $h_r(t, R_k)$ is available. The similarity function must be practically computable even for large repositories with abundant streams of incoming commits.

4 The proposed method

When a new commit arrives at a repository, the reviewer recommendation system must nominate the best candidates to review this commit. State-of-the-art systems (Thongtanunam et al. 2015; Balachandran 2013) read and analyze the whole history of the reviews and commits up to now. In our opinion, such approaches are impractical, because they consume too much time and system resources. Assume a commit C_{t+1}. For C_{t+1} we have to read and process $|h(t)| = t$ historical commits. Therefore, up to the time $t + 1$ the reviewer recommendation system has to process $O(|h(t)|^2) = O(t^2)$ commits in total. Even for large repositories it is a feasible effort from the point of view of a single repository. However, it could be a serious problem for repository hosting companies like `github` or `bitbucket` that store and serve millions of repositories. In order to overcome this problem, we propose a model based on profiles of reviewers. For each reviewer $R_i \in R$ we create his/her profile $P_i \in P$, where P is set of all available profiles. The profile P_i will be updated each time the reviewer R_i comments on a commit. Therefore, each time a new commit C_{t+1} is reviewed in the system, it will be added to the reviewer's profile.

Moreover, instead of the similarity function mentioned in Section 3.1, we rather use a similarity function $s\colon C \times P \to \mathbb{R}$ that for a commit C_{t+1} and a reviewer profile P_i quantifies how this commit fits this reviewer's history. In order to find best nominee to review a commit C_{t+1} from a set of reviewers R, we compute the value of this function for each candidate reviewer's profile. Therefore, we have to process $|R|$ reviewer's profiles but not t commits. At time $t + 1$, such a system needs to process $O(|R| \cdot |h(t)|)$ commits. This is a significant improvement as $|R|$ is usually notably smaller than $|C|$. For large open-source projects that we have evaluated, it holds that $|R| < 0.02 \cdot |C|$, see Section 5 Table 4. Therefore, to make the whole method feasible, we need (1) a data structure to store reviewers' profiles that has small memory footprint, (2) a suitable similarity function s that can be effectively computed, and (3) a fast profile updating mechanism to be applied whenever a reviewer comments on a commit. Fast updates of reviewer profiles are required as such system at time $t+1$ will have to perform $O(|h(t)|) = O(t)$ profile updates.[1] The remainder of this section is devoted to tackling those challenges.

[1]This will be usually close to t as most often there is only one review per commit, see Table 4 in Section 5.

4.1 The reviewer profile

In Section 3.1 we have introduced the notion of a commit as the list of modified files $f(C_t) = [f_1^t, f_2^t, \ldots, f_{l_t}^t]$, where l_t is the number of modified files in this commit. Each modified file path $f_i^t \in f(C_t)$ in a commit C_t is a string representing file location in the project. The file path may also be seen as a sequence of words separated by a special character. We can also drop the order of this words and treat their sequence as a multiset of words (or bag of words). We follow the semantics of multisets from Knuth (1981) and Singh et al. (2007).

By $m(f_i^t)$ we will denote the mapping that converts a file path f_i^t into the multiset of words (path segments) that occurs in f_i^t. Furthermore, we can compute the multiset-theoretic union of $m(f_i^t)$ for all paths in $f(C_t)$:

$$m(C_t) = \bigcup_{f_i^t \in f(C_t)} m(f_i^t)$$

We will treat $m(C_t)$ as the representation of the commit C_t used to construct the reviewers' profiles. We define the profile P_i for the reviewer R_i at the time t as the multiset-theoretic union of m for all commits previously reviewed by R_i, i.e.:

$$P_i(t) = \bigcup_{C_k \in h_r(t, R_i)} m(C_k)$$

In order to illustrate the above procedure, assume a repository containing commits C_1, C_2 and C_3 and an assignment of reviewers R_A and R_B as shown in Table 1.

The commit C_1 has the following representation as the multiset $m(C_1)$:

$$m(C_1) = \left\{ \begin{array}{r} (java : 3), (main : 2), (package1 : 2), (package2 : 1), (src : 3), \\ (test : 1), (SomeClass.java : 1), (DifferentClass.java : 1), \\ (someClassTest.java : 1) \end{array} \right\}$$

Since the commit C_1 is the first reviewed by R_A, $m(C_1)$ becomes the profile P_A of the reviewer R_A at time $t = 1$ as:

$$P_A(1) = \left\{ \begin{array}{r} (java : 3), (main : 2), (package1 : 2), (package2 : 1), (src : 3), \\ (test : 1), (SomeClass.java : 1), (DifferentClass.java : 1), \\ (someClassTest.java : 1) \end{array} \right\}$$

Table 1 Example repository and assignment of reviewers

Commit	List of reviewers	Paths (modified files)
C_1	R_A	"src/main/java/package1/SomeClass.java"
		"src/main/java/package2/DifferentClass.java"
		"src/test/java/package1/SomeClassTest.java"
C_2	R_A, R_B	"src/main/java/package2/DifferentClass.java"
		"src/test/java/package2/DifferentClassTest.java"
C_3	R_B	"src/main/java/package1/SomeClass.java"
		"src/main/java/package2/DifferentClass.java"
		"src/test/java/package1/SomeClassTest.java"

After the commit C_2 (at the time $t = 2$) this reviewer's profile P_A becomes $m(C_1) \cup m(C_2)$:

$$P_A(2) = \left\{ \begin{array}{r} (java : 5), (main : 3), (package1 : 2), (package2 : 3), (src : 5), \\ (test : 2), (SomeClass.java : 1), (DifferentClass.java : 1), \\ (someClassTest.java : 1) \end{array} \right\}.$$

4.2 The implementation of the reviewers' profiles

The proposed algorithm makes an extensive use of the reviewers' profiles. Therefore, they must be efficiently implemented. We have chosen hash tables for this purpose, since they provide average constant time of insertions and lookups. The hash function is used to map each word in a profile to its multiplicity. Hash tables are implemented in most popular programming languages.

In order to create the multiset-based representation $m(C_t)$ of a commit C_t, (1) all file paths modified in this commit are retrieved; (2) these paths are tokenized into words; and eventually (3) added to the hash table (by incrementing the occurrence counter). The computation of $m(C_t)$ performs $O(K)$ lookups in the hash table where K is the number of words (paths segments) in the file names modified by C_t. Since the expected time of a hash lookup is $O(1)$, the complexity of computation of $m(C_t)$ is $O(K)$.

An update of a reviewer's profile is the second most frequently performed operation in our reviewer recommendation algorithm. When a reviewer comments on a commit, the multiset-based representation of this commit gets added to his/her profile. This operation can be implemented as an iteration over items in the multiset-based representation of this commit. For each item in the commit, we appropriately increment the number of its occurrences in the profile. The time-complexity of such an update is equivalent to $O(S)$ hash lookups where S is the number of words in the multiset-based representation of the commit. Since the expected time of a hash lookup is $O(1)$, the complexity is $O(S)$.

A profile can contain only the words that are path segments (names of directories and files) in the repository. Therefore, a profile cannot be larger than a tree representation of all file paths in the directory structure of the repository. Table 8 in Section 5 shows memory footprint of such repository structures of analyzed projects. There are 47233 unique words for Android, 60521 for Libre Office, 9759 for Open stack and 98897 for Qt. Thus, the actual sizes of reviewer profiles are relatively small making the whole algorithm applicable in practice.

4.3 Computing the similarity between a profile and a pending commit

When a new commit arrives, our algorithm compares its multiset-based representation with profiles of all reviewers in order to elect the best candidate to review this commit. Various functions are commonly used to measure the similarity between source code entities (Terra et al. 2013). Most of them were first defined on sets and then adapted to the software engineering context via the transformation of source code entities into items of the function domain.

We selected two such functions, namely, the Jaccard coefficient and the Tversky index. The Jaccard coefficient is one of the most widely adopted similarity functions. It is defined for two sets X and Y as

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

It computes the fraction of overlapping items in the two sets.

On the other hand, the Tversky index is a generalized form of Tanimoto coefficient and is used to compare a variant to a prototype. For two sets X and Y it is defined as:

$$T(X, Y) = \frac{|X \cap Y|}{|X \cup Y| - \alpha|X - Y| - \beta|Y - X|}$$

The two parameters α (corresponding to the weight of a prototype) and β (corresponding to the weight of variant) need to be tuned for the specific application. This tuning for our method is described in Section 6.1.

We adapted set similarity functions such as Tversky index and Jaccard coefficient to work on multisets by replacing set operations such as union, intersection and relative complement to those defined on multisets (Petrovsky 2012; Singh et al. 2007; Knuth 1981). We decided to use Tversky index as the main similarity function due to the possibility of adjusting the importance ratio between a profile and a review. We use Jaccard coefficient for completeness as a baseline multiset similarity function in order to verify if our assumptions are correct.

Each of these two similarity measures can be efficiently implemented using hash tables. If the sizes of sets X and Y are M and N, the computation of their multiset union, difference and intersection requires at most $O(M+N)$ hash lookups which leads to the overall average time-complexity $O(M + N)$.

4.4 The reviewer recommendation model

In the proposed method, when a new commit C_{t+1} arrives at the repository, it is mapped to its multiset-based representation $m(C_{t+1})$ and compared to profiles of reviewers P. The similarity between $m(C_{t+1})$ and all profiles is computed (we use Tversky index and Jaccard coefficient for reference). Eventually, top n reviewers are selected. Table 2 summarizes the semantics of all necessary operations.

Using the values of the similarity function applied to profiles of reviewers with respect to the multiset-based representation of a commit, we order reviewers according to this similarity (the highest similarity first). We assume that a higher similarity implies better competences to review the pending commit. By this method we obtain the similarity values for all available reviewers. In order not to bother worse-matched reviewers, we select only n best fitting from the ordered list. When two or more reviewers obtain the same similarity, we use dates of their last review to break the ties. A reviewer who did most recent review takes precedence.

4.5 Extensions to reviewer recommendation model

In the model presented above, the passing time and the dynamics of teams are not reflected. A person who contributed to a particular functionality two years ago and left the project

Table 2 Operations in our method

Function	Definition	Description
$P_i(t)$	$\bigcup_{C_k \in h_r(t,R_i)} m(C_k)$	Construction of a profile
$m(C_t)$	$\bigcup_{f_i^t \in f(C_t)} m(f_i^t)$	Creation of multiset-based representation
$s(C_{t+1}, P_i(t))$	$T(P_i(t), m(C_{t+1}))$	Tversky index as the similarity function
$s(C_{t+1}, P_i(t))$	$J(P_i(t), m(C_{t+1}))$	Jaccard coefficient as the similarity function
$top(C_{t+1}, n)$	$\langle R_{i1}, R_{i2}, \ldots, R_{in} \rangle$	Computation of top-n

does not have to be the best candidate to review most recent changes. We address this issue by extinguishing older reviews in reviewer profiles. Extinguishing is done by multiplying frequencies of words in profiles by a factor lesser than one. The factor can be tuned to obtain different qualities of recommendations. This approach is similar to time-decaying Bloom filters (Cheng et al. 2005). The main differences are the method to calculate the factor and the data structure used. Some authors researched evolutionary rules in the context of membrane computing, and used multiset extinguishing to decrease factor of such rule activation (Arteta et al. 2011).

The extinguishing factor for a number of days (or the number of commits in between) d is computed by the formula:

$$ex(d) = (\sqrt[l]{f})^d$$

where f is the decaying factor and l the number of days (commits) for which the decaying factor is to be fully employed. The tuning of these two parameters for our algorithm is discussed in Section 6.1. For example, if $f = \frac{1}{2}$ and $l = 180$, then after half a year (or after 180 commits) each item in a reviewer profile is halved. Let $id(C_t)$ and $date(C_t)$ denote respectively the sequence number of the commit C_t and the number of day between commits C_1 and C_t.

We also consider another version of the selection of n top candidates to do the pending review when some profiles have exactly the same similarity with respect to the analyzed commit. In the *non tie-breaking* version of the algorithm we use the function

$$ntop(C_{t+1}, n) = \langle\langle R_{1,1}, \ldots, R_{1,m_1}\rangle, \ldots, \langle R_{n,1}, \ldots, R_{n,m_n}\rangle\rangle$$

where all reviewers on a single sublist (e.g. $\langle R_{1,1}, \ldots, R_{1,m_1}\rangle$) have the same similarity to the commit. Moreover, preceding lists contain reviewers with higher similarity to the commit than the following lists (Table 3).

4.6 Possible extensions to profile creation

We create reviewer profiles capturing a subset of features from a project history, namely the reviewed file paths. Other kinds of features obtained from different views on the same repository can also be aggregated into profiles. Such features can simply be already present in a repository, e.g. the author of each line. They can also be computed by a separate algorithm, e.g. a topic model of commit metadata via Latent Dirichlet Allocation. The data preprocessing method can also be adjusted to cater the needs of a specific project. In certain projects

Table 3 Modified operations in our method

Function	Definition	Description
$P_i^{id}(t)$	$\bigcup_{C_k \in h_r(t,R_i)} m(C_k) \cdot ex(id(C_k) - id(C_{k-1}))$	Construction of profile extinguished by id
$P_i^{date}(t)$	$\bigcup_{C_k \in h_r(t,R_i)} m(C_k) \cdot ex(date(C_k) - date(C_{k-1}))$	Construction of profile extinguished by date
$s(C_{t+1}, P_i^{id}(t))$	$T(P_i^{id}(t), m(C_{t+1}))$	Similarity extinguished by id
$s(C_{t+1}, P_i^{date}(t))$	$T(P_i^{date}(t), m(C_{t+1}))$	Similarity extinguished by date
$ntop(C_{t+1}, n)$	$\langle\langle R_{1,1}, \ldots, R_{1,m_1}\rangle, \ldots, \langle R_{n,1}, \ldots, R_{n,m_n}\rangle\rangle$	Non tie-breaking top n computation

the analysis of merge/maintenance commits might introduce unnecessary noise. However, other projects can benefit from it.

In a typical software development environment the roles of the code reviewer and the code author often belong to the same set of individuals. Thus, the information on the code authorship can be aggregated to build authors' profiles. Assume two distinct possible profiles for each person, i.e. his/her reviewer profile and author profile. Given these two distinct profiles, we can consider at least the four following scenarios to use them: (1) recommendation via reviewer profile only, (2) recommendation via authorship profile only, (3) combining the similarity score from both profiles, (4) combining both profiles into one author/reviewer profile. The third scenario requires post-processing of similarity scores to create one unified list of reviewers possibly using different similarity functions for each kind of profile. The fourth scenario can be implemented via obtaining authorship information on file path level and adding those file paths to existing reviewer profile. It seems easier to implement, but it lacks the flexibility of the third scenario. We did not have a dataset with authorship information. Therefore, we implemented only the first scenario. However, we hope to analyze multiple views (at least reviewers' and authors') on the same repository in future work.

5 Empirical evaluation

In this section we detailed the empirical evaluation of the proposed method. We examined its accuracy, performance and memory footprint. We used the Thongtanunam dataset (Thongtanunam et al. 2015) that is based on mature and well-known open-source projects with long development history (see Table 4). We compared the obtained results with the state-of-the-art methods, i.e. ReviewBot (Balachandran 2013) and Revfinder (Thongtanunam et al. 2015). Additionally, we have reimplemented Revfinder. In the remainder of this article our implementation of Revfinder will be referenced as Revfinder*. The original implementation of Revfinder is not available. However, we wanted to compute more quality metrics of their results than they had published. They showed only the recall, while we needed also the

Table 4 Statistics of processed projects

	Projects			
Statistic	Android	LibreOffice	OpenStack	Qt
Commits	5126	6523	6586	23810
(duplicated)	0	1	0	1
Reviewers	94	63	82	202
(only one review)	7	10	1	7
Reviews	5467	6593	9499	25487
First review	2010-07-13	2012-03-06	2011-07-18	2011-05-18
Last review	2012-01-27	2014-06-17	2012-05-30	2012-05-25
Average review distance				
seconds	9492	11032	4160	1354
ids	5.951814	1.499617	1.207865	1.140529
Sample size (MB)	3.8	5.5	4.2	19

precision and F-measure. We also wanted to assess the efficiency of their method that had not been mentioned in their article. We did not create a reimplementation of ReviewBot, since its quality had been verified to be inferior to Revfinder.

5.1 The experimental setup

We have run all experiments on a computer with two Six-Core AMD Opteron™ processors, 32 GB RAM, Intel®RAID Controller RS2BL040 set in RAID5, 4 drives Seagate Constellation ES ST2000NM0011 2000 GB. There was no other load on this machine during the experiments.

The time consumption was measured using Python's *timeit* module. For our method we took the average of 10 runs on each project. In case of Revfinder* (for LibreOffice, OpenStack and Android), we also executed 10 runs and took their average execution time. For Revfinder* on Qt we performed only one run, since the run-time exceeded 5 days and consumed excessive memory.

The memory footprint was measured using Python's *psutil* for the whole program and *sys.getsizeof* function for selected data structures, such as lists of reviewer profiles and arrays containing all distances. Our sequential reimplementation of Revfinder had lower memory consumption than the parallel version. However, it was more time and CPU consuming.

We published the sources of our implementation in order to aid replicating our results (Fejzer and Przymus 2016).

5.2 Experimental dataset

We used the Thongtanunam dataset (Thongtanunam et al. 2015) that contains data on code reviews of the following projects: Android (a mobile operating system), LibreOffice (an office suite), OpenStack (a cloud computing platform) and Qt (an application framework). A summary of this dataset is presented in Table 4. This dataset contains contributors' activity recorded by the Gerrit code review systems. Gerrit works with Git repositories. By default it allows a commit to enter a repository only if this commit has at least one positive review and it has been blocked by no reviewer. The dataset contains data on reviews. The features of each review are its unique identifier, the list of reviewers, the list of files, the creation date, the completion date, the name of the project and the information whether this code review has been merged.

It should be noted that it is impossible to obtain 100% accurate reviewer recommendations, because the authors of some reviews have no previous review history. Knowing that we can identify the highest possible accuracy of recommendation as: 98% (Android) and 99% (LibreOffice, OpenStack and Qt), we performed initial analysis of reviewers' activity in order to understand the dataset better.

The left column of Fig. 1 shows the number of reviews per a single reviewer. For all four projects the diagram conforms to an exponential distribution. Most reviewers create less than 20 reviews for Android and LibreOffice and less than 60 reviews for OpenStack and Qt. In our opinion, these numbers result from a reviewer's focus on a specific bug or a feature request.

The middle column of Fig. 1 shows the durations of individual reviewers' activity. In case of Android and LibreOffice they are significantly longer than for Qt and OpenStack. It is probably caused by designated maintainers working for companies contributing to these projects.

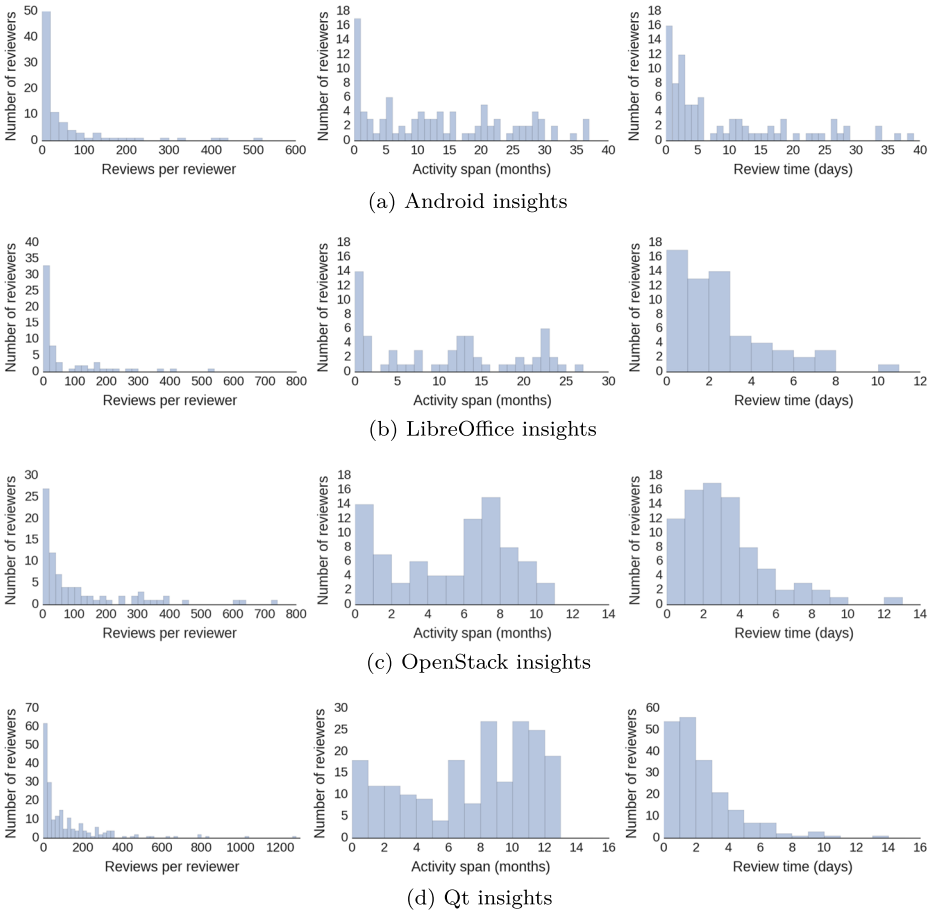

Fig. 1 Evaluated projects insights

The right column of Fig. 1 depicts the durations of individual reviews. For all projects the majority of reviews are completed up to three days for LibreOffice and OpenStack projects, up to two days for Qt and up to six days for Android. The longest review time and existence of outliers in case of Android suggests that a reviewer recommendation system can aid prioritizing the process (Yu et al. 2014).

5.3 Applied metrics

We computed the precision, the recall and F-measure in order to assess the quality of our solution. These metrics are commonly used in data mining and in particular in bug detection and reviewer recommendation (Lee et al. 2013; Jeong et al. 2009). We assume that $actual(C_{t+1}, n)$ is the actual n reviewers of the commit C_{t+1}. We define the metrics for our experimental evaluation using operations from Section 4.4. Therefore, $actual(C_{t+1}, n)$ is the set of n persons recommended at time t to review the commit C_{t+1}.

$$precision(t+1, n) = \frac{|top(C_{t+1}, n) \cap actual(C_{t+1}, n)|}{|top(C_{t+1}, n)|}$$

$$recall(t+1,n) = \frac{|top(C_{t+1},n) \cap actual(C_{t+1},n)|}{|actual(C_{t+1},n)|}$$

$$F\text{-}measure(t+1,n) = \frac{2 \cdot precision(t+1,n) \cdot recall(t+1,n)}{precision(t+1,n) + recall(t+1,n)}$$

The authors of Revfinder compared their tool to ReviewBot (Thongtanunam et al. 2015; Balachandran 2013) using only top-n recall and the mean reciprocal rank (*MRR*). We compare their results to ours with respect to those metrics. The mean reciprocal rank is the metric commonly used in information retrieval (Voorhees 1999). It aids evaluating any process that produces a list of possible responses to a series of queries, ordered by the probability of their correctness.

For the problem of reviewer recommendation we assume that $rank\colon R \to \mathbb{N}$ is the index of the first actual reviewer in the prediction. Then we define mean reciprocal rank as follows:

$$MRR = |t+1| \sum_{t=1}^{t+1} \frac{1}{rank(top(C_{t+1}, 10))}$$

5.4 Parameter selection

Tversky index (see Section 4.3) has two parameters α and β. They are weights applied to sizes of multiset differences. The value α weighs the difference between a profile and a commit, while β weighs the reverse difference. We used an exhaustive search to discover appropriate values of these parameters. We found out that $\alpha = 0$ and $\beta = 1$ gave best results for the problem of reviewer recommendation. We checked possible values of those parameters on 10% of dataset in range between 0 and 1, by step 0.01, every time satisfying the equation $\alpha + \beta = 1$. These values of parameters mean that we use only the commit-to-profile difference, the intersection of a profile and a commit and their union.

We also tuned the extinguishing of profiles' content by date. We assumed that the impact of a review is halved after half a year. Therefore, we put $l = 183$, $f = 0.5$.

For the extinguishing of profiles' content by id, we used $l = 2500$, $f = 0.5$. The number 2500 is slightly less than the half the number of reviews for Android, LibreOffice and OpenStack. Our results show that the method based on Tversky index enhanced by the extinguishing is not a significant improvement. The method with extinguishing has better accuracy on the mean reciprocal rank than standard Tversky index on OpenStack and Qt projects since those project have the larger number of reviews and smallest average time between their submission (see Table 4). Thus, older reviews are more likely to have significant impact.

6 Results and discussion

6.1 Recommendation system accuracy

We tested the method presented in Section 4 against state-of-the-art methods using the dataset described in Section 5.2.

We experimented with five implementations. Three of them are based on Tversky index as the similarity function. The first variant denoted by *Tversky No Ext* has no extinguishing mechanism. The other two use extinguishing by id (denoted by *Tversky Ext id*) and by date (*Tversky Ext date*) as described in Section 4.3. The fourth implementation used the Jaccard coefficient as the similarity function (denoted by *Jaccard*). The fifth variant was

our reimplementation of Revfinder denoted by *Revfinder**. In the tables we also present the results of *Revfinder* and *ReviewBot* as published in Thongtanunam et al. (2015) and Balachandran (2013).

Figure 2 summarizes the results. We can see that *Tversky No Ext* obtains better precision-to-recall ratio and higher F-measure than all other methods. Results gathered in Table 5 show a detailed comparison of the methods listed above in terms of the following metrics: top-n recall for $n \in \{1, 3, 5, 10\}$ and the mean reciprocal rank. Furthermore, for reference we include results published in articles on Revfinder (Thongtanunam et al. 2015) and ReviewBot (Balachandran 2013). We conclude that the methods proposed in this article obtain the higher recall and the mean reciprocal rank on all projects.

Next we have assessed whether the observed improvements in terms of used metrics are statistically significant. We have compared the ratio of successful recommendations to all recommendations, defined as $|top(C_{t+1}, n) \cap actual(C_{t+1}, n)|$. Then, we have examined results of top-n for all methods, all four projects and all $n \in \{1, 2, ..., 10\}$. We used Levene test for equality of variances (Levene 1960) to ensure that the variance of the results is not equal. Then we used the Kruskal-Wallis H-test (Kruskal and Wallis 1952) with null hypothesis "Median coefficients specified for each method are equal.", and p-value threshold equal to 0.05. We were able to reject this hypothesis for all tests. Then we used Student's t-test for independent samples with the hypothesis "Two method results have the same average values" with the same p-value threshold. We were able to reject this hypothesis for most pairs of results, namely 362 out of 400. The exceptions were (1) *Tversky No Ext* and *Revfinder** on OpenStack for $n \in \{6, 7, 8, 9, 10\}$, (2) *Tversky Ext id* and *Tversky Ext date* for Android and LibreOffice for all n.

Thus we conclude that, in majority of cases the methods proposed in this article show statistically significant improvement of the precision, recall and F-measure metrics.

6.1.1 Non tie-breaking approach

The experiments presented above concern a tie-breaking version of the proposed model (see Section 4.5). However, the non-tie-breaking version is an interesting alternative. If we

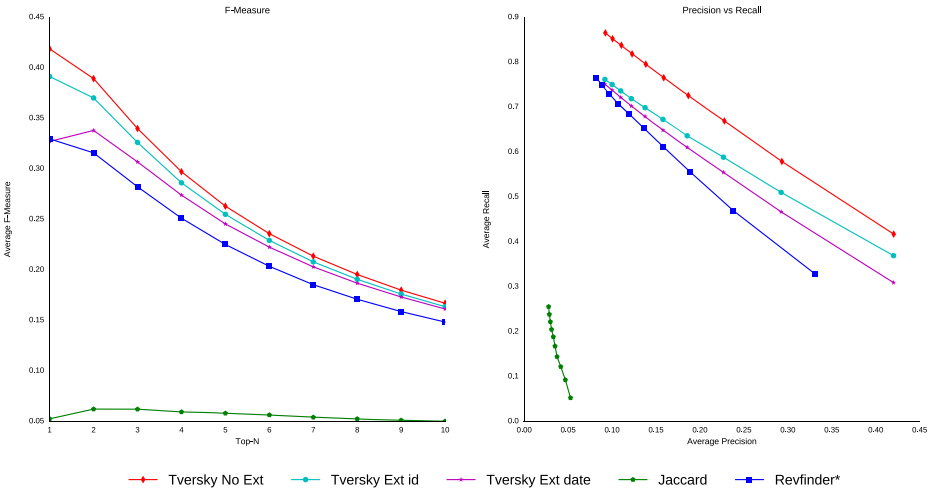

Fig. 2 Metrics comparison

Table 5 The recall and MRR for all methods and four projects

		Recall				MRR
	Method	Top1	Top3	Top5	Top10	
Android	Tversky No Ext	0.5492	0.8034	0.8591	0.9066	0.7301
	Tversky Ext id	0.5138	0.7467	0.8004	0.8469	0.7290
	Tversky Ext date	0.4684	0.7315	0.7913	0.8460	0.6985
	Jaccard	0.0788	0.1859	0.2544	0.3798	0.4070
	Revfinder*	0.4686	0.7218	0.8098	0.8898	0.6640
	Revfinder	0.46	0.71	0.79	0.86	0.60
	ReviewBot	0.21	0.29	0.29	0.29	0.25
LibreOffice	Tversky No Ext	0.3353	0.5390	0.6571	0.8106	0.5799
	Tversky Ext id	0.3225	0.5216	0.6329	0.7872	0.5763
	Tversky Ext date	0.2692	0.4917	0.6097	0.7748	0.5340
	Jaccard	0.0239	0.0524	0.0747	0.1367	0.3559
	Revfinder*	0.2816	0.5183	0.6428	0.7972	0.5417
	Revfinder	0.24	0.47	0.59	0.74	0.40
	ReviewBot	0.06	0.09	0.09	0.10	0.07
OpenStack	Tversky No Ext	0.4177	0.6985	0.7967	0.8892	0.5500
	Tversky Ext id	0.2916	0.4829	0.5537	0.6157	0.5530
	Tversky Ext date	0.2260	0.4360	0.5176	0.6055	0.4924
	Jaccard	0.0835	0.1937	0.2609	0.3884	0.3931
	Revfinder*	0.3987	0.6846	0.7903	0.8854	0.5390
	Revfinder	0.38	0.66	0.77	0.87	0.55
	ReviewBot	0.23	0.35	0.39	0.41	0.30
Qt	Tversky No Ext	0.3655	0.6351	0.7480	0.8540	0.5973
	Tversky Ext id	0.3479	0.6014	0.7024	0.7962	0.6054
	Tversky Ext date	0.2720	0.5586	0.6746	0.7771	0.5500
	Jaccard	0.0226	0.0530	0.0785	0.1154	0.3926
	Revfinder*	0.1899	0.3403	0.4184	0.5356	0.5290
	Revfinder	0.2	0.34	0.41	0.69	0.31
	ReviewBot	0.19	0.26	0.27	0.28	0.22

consider top-n groups of reviewers with the same score, we can observe the following. Figure 3 shows histograms of group size for non-tie-breaking *Tversky No Ext* top-1. For all projects singleton groups dominate. The number of smaller groups gradually decreases. This indicates that our methods usually find a small number of matching reviewers. Thus, a valid question arises: Do we really need to break the ties? For large and long-term projects it may be fairly common that more than one person is responsible for specific parts of the project. Therefore, more than one reviewer has appropriate knowledge to perform a review. Under that assumption we investigate the predictive performance of this solution. We observe that top-n groups yield a notably high recall rates. Table 6 shows the recall of top-n groups for several values of n. We conclude that this approach is also applicable, especially when a project's maintenance is naturally divided among working groups. It happens in e.g. subsystems in Linux kernel and FreeBSD operating system (Rigby and Storey 2011). In such systems, having the high accuracy of group based recommendation may be desirable.

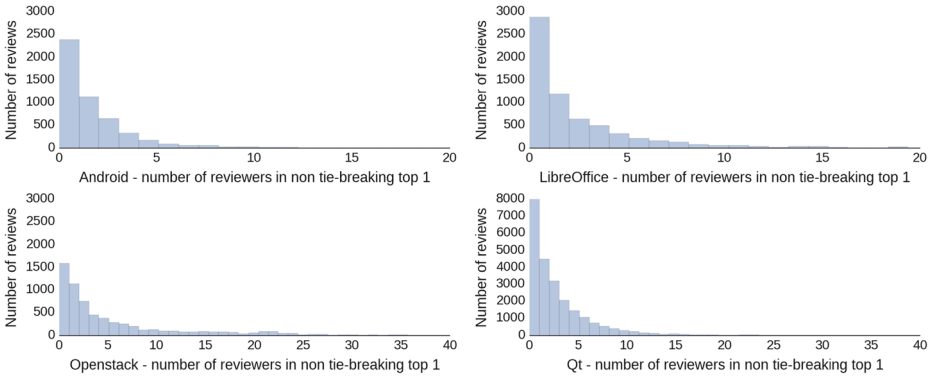

Fig. 3 The number of recommended reviewers in top-1 Tversky No Ext without tie-breaking. The horizontal axis shows the number of reviewers recommended. The vertical axis presents the number of commits for which this number of reviewers was recommended by this method

6.2 Performance evaluation

We measured the efficiency of three implementations, namely *Tversky No Ext*, *Jaccard* and *Revfinder**. We split the processing into three phases corresponding to operations in our model (see Table 2). They are the transformation of a commit into a multiset, the actual update of a profile and the similarity calculation in order to recommend the reviewers. Table 7 presents the results. The times for the three phases are total running times for all operations per a single project. The rightmost column presents average processing time per a review in a project.

The results presented in Table 7 are averages of running times for a number of executions. We executed 10 consecutive runs of the sequential version of our algorithm, both *Tversky No Ext* and *Jaccard*. In case of *Revfinder**, we used 10 consecutive runs only on Android, LibreOffice and OpenStack. We decided to evaluate *Revfinder** on Qt using only 1 run. This is due to the excessive consumption of resources for that project. One run took five days to compute. In all tests our method was faster by an order of magnitude. The coefficient of variation did not exceed 1% for all repeated runs. We used Mann-Whitney U two-sample test $n_1 = n_2 = 10, U = 100, p < 0.05$ with null hypothesis "Median execution time for *Tversky No Ext* and *Revfinder** are equal on the same project." on Android, LibreOffice and Openstack projects. Median times of 10 consecutive runs for *Tversky No Ext* on those projects were 248.84, 713.79 and 105.50 respectively. Median times of 10 consecutive runs for *Revfinder** on those projects were 15770.17, 24875.84 and 17235.29 respectively. We rejected null hypothesis on all tested projects.

Table 6 The recall and MRR of Tversky No Ext without tie-breaking

	Recall				MRR
Project	Top1	Top3	Top5	Top10	
Android	0.7052	0.9058	0.9243	0.9317	0.8499
LibreOffice	0.4486	0.8013	0.8896	0.9402	0.6706
OpenStack	0.7486	0.9140	0.9417	0.9589	0.8289
Qt	0.6078	0.8798	0.9339	0.9633	0.7618

Table 7 Results of the performance evaluation

Project	Method	Total time (in seconds)			Average
		Multiset transf-ormation	Profile update	Similarity calculation	single review processing
Android	Tversky No Ext	0.4232	5.3811	241.8630	0.0483
Android	Jaccard	0.4147	5.7651	132.3110	0.0270
Android	Revfinder*	–	0.0181	15797.9000	3.0819
LibreOffice	Tversky No Ext	0.6813	39.9458	668.6240	0.1087
LibreOffice	Jaccard	0.6741	39.9517	356.9950	0.0610
LibreOffice	Revfinder*	–	0.0249	24875.2000	3.8135
OpenStack	Tversky No Ext	0.3989	3.3080	101.1660	0.0159
OpenStack	Jaccard	0.3964	3.3920	56.6723	0.0092
OpenStack	Revfinder*	–	0.0254	17227.2000	2.6157
Qt	Tversky No Ext	2.3738	68.4784	4795.6800	0.2044
Qt	Jaccard	2.3844	68.2125	2651.9200	0.1143
Qt	Revfinder*	–	2.0000	432000.0000	18.1437

The memory footprint of profiles created by our algorithm is smaller than *Revfinder** (which uses 10 GB of RAM for Android, 12 GB for LibreOffice, 11 GB for OpenStack, and more than 32 GB for Qt). It is evidenced in Table 8.

This experimental observations confirm that our method uses significantly less computing power (Fig. 4) and memory than state-of-the-art methods. Thus, it is feasible to apply our methods in large repositories such as Github.

6.3 Discussion

The methodology We created a new approach to the problem of reviewer recommendation. We started by defining two distinct phases of processing: (1) the aggregation of reviews into profiles, and (2) the calculation of profile-to-review similarity. Later we analysed typical lifetime of a repository in terms of the number of occurrences of these two phases. That led us to the proposed solution. It minimizes the cost of the second phase by spending more time in the first phase. Both Revfinder and ReviewBot do it reversely. They have the low complexity of the first phase and the high complexity of the second phase.

Table 8 Memory footprint of Tversky No Ext (in MB)

Project	Virtual memory	Multiset profile size	
		All	Average
Android	396.6328	3.2235	0.0343
LibreOffice	411.6172	5.2938	0.0840
OpenStack	400.1172	1.8546	0.0226
Qt	530.0469	12.5634	0.0622

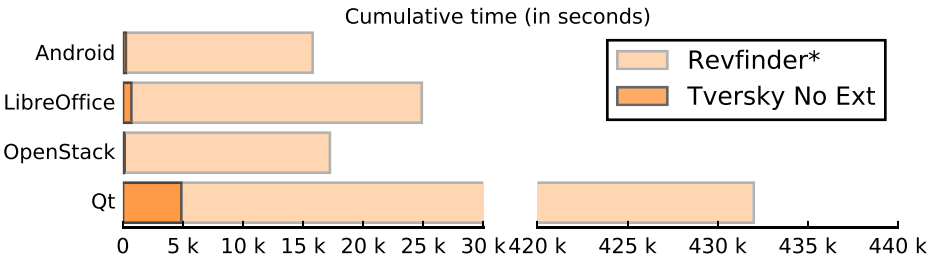

Fig. 4 Performance difference between methods

Complexity comparison We accelerated the calculation of similarity in an analogous manner as a database index enables more efficient access to records. However, a database index adds storage and processing penalty. We are aware that, due to Amdahl law, the optimization of the most common and costly operation shall give us better improvement. We estimated that from two proposed phases the similarity computation shall be done notably more often and shall determine the complexity of the whole algorithm. Usually, the number of reviewers is significantly lower than the number of commits/reviews. As shown in Table 7 the computation of similarity is more expensive than other phases. Thus, it attests our assumptions about the overall complexity.

Empirical comparison Our two phase approach allowed using various similarity functions that yielded varying results. That enabled understanding the data better and selecting a function maximizing the similarity between aggregated profiles and reviews. By adapting two set difference functions (Tversky index and Jaccard coefficient) to multisets we discovered that the most important feature for calculating similarity was the difference between multisets of a commit and a profile. Our two phase approach also enabled adding optimizations during the aggregation phase, such as extinguishing of older reviews. The evaluation of our method using Tversky index against state-of-the art methods shown that in most cases we obtained statistically significant improvements for all metrics. Furthermore, we achieved significantly lower time complexity. We have also proven that methods extinguishing past reviews are comparable with the state-of-the-art (see Table 5).

Profile construction We build reviewers profiles based on reviewed file paths only, due to the fact that previous studies had shown that file path solutions like Revfinder obtain better accuracy than methods using code authorship like ReviewBot. Additionally, we were limited in our selection of available features by dataset constraints. The dataset does not contain detailed and complete information about code authorship. However, it is possible to expand our model, for instance via the construction of separate profiles based on code authorship and a redefinition of the similarity function including both profiles.

7 Conclusions and future work

The ability to elect competent reviewers for a commit in a large, industrial system is extremely important. However, its automation can be undermined by limits created by both the computing power and the available memory. In this article we introduced reviewers' profiles that aggregate their expertise. We presented a novel method to suggest reviewers

for code based on such profiles. This method has significantly higher accuracy and lower computational complexity than state-of-the-art methods. We also showed the statistical significance of the improvements of the precision, the recall and F-measure.

We also analysed the methods proposed in this article in terms of performance. We concluded that they have smaller complexity and outperform the state-of-the-art methods. Excessive experimental evaluation presented in this article proves that our methods can be applied in large industrial system.

There are several possible future areas of research. Firstly, the usage of other data structures for reviewer profiles is worth considering, e.g. locality-sensitive hashing can yield further performance improvements. Secondly, the prediction accuracy is also an issue. Due to the small number of developers compared to the quantity of file paths in all commits, we did not adopted collaborative filtering. In our opinion it can improve the accuracy of the methods. Thirdly, recent studies (Yu et al. 2016) show that fusion methods can achieve similar performance to the state-of-the-art. By introducing the social factors to our method as a reviewer ordering mechanism we should be able to confirm or disprove this fact. Another possible approach based on fusion is achievable by adding the whole repository to the dataset. This possibly enables constructing more detailed profiles, based on available source code contained in each commit, like code authorship information.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Arteta, A., Blas, N.G., & de Mingo López, L.F. (2011). Solving complex problems with a bioinspired model. *Engineering and Applications of AI*, *24*(6), 919–927. https://doi.org/10.1016/j.engappai.2011.03.007.

Balachandran, V. (2013). Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In Notkin, D., Cheng, B.H.C., & Pohl, K. (Eds.) *35th international conference on software engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013* (pp. 931–940). IEEE / ACM. http://dl.acm.org/citation.cfm?id=2486915.

Bisant, D.B., & Lyle, J.R. (1989). A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, *15*(10), 1294–1304. http://doi.org/10.1109/TSE.1989.559782.

Charlin, L., & Zemel, R. (2013). The toronto paper matching system: an automated paper-reviewer assignment system.

Cheng, K., Xiang, L., Iwaihara, M., Xu, H., & Mohania, M.K. (2005). Time-decaying bloom filters for data streams with skewed distributions. In *15th international workshop on research issues in data engineering (RIDE-SDMA 2005), stream data mining and applications, 3-7 April 2005, Tokyo, Japan* (pp. 63–69). IEEE Computer Society. https://doi.org/10.1109/RIDE.2005.15.

Conry, D., Koren, Y., & Ramakrishnan, N. (2009). Recommender systems for the conference paper assignment problem. In Bergman, L.D., Tuzhilin, A., Burke, R.D., Felfernig, A., & Schmidt-Thieme, L. (Eds.) *Proceedings of the 2009 ACM conference on recommender systems, RecSys 2009, New York, NY, USA, October 23-25, 2009* (pp. 357–360). ACM. https://doi.org/10.1145/1639714.1639787.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In Whitehead, J., & Zimmermann, T. (Eds.) *Proceedings of the 7th international working conference on mining software repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings* (pp. 31–41). IEEE Computer Society. https://doi.org/10.1109/MSR.2010.5463279.

Drake, J., Mashayekhi, V., Riedl, J., & Tsai, W.T. (1991). A distributed collaborative software inspection tool: Design, prototype, and early trial. In *Proceedings of the 30th aerospace sciences conference*.

Dumais, S.T., & Nielsen, J. (1992). Automating the assignment of submitted manuscripts to reviewers. In Belkin, N.J., Ingwersen, P., & Pejtersen, A.M. (Eds.) *Proceedings of the 15th annual international ACM*

SIGIR conference on research and development in information retrieval. Copenhagen, Denmark, June 21-24, 1992 (pp. 233–244). ACM. https://doi.org/10.1145/133160.133205.

Fejzer, M., & Przymus, P. (2016). Implementation code repository. http://www-users.mat.umk.pl/~mfejzer/reviewers_recommendation.

Gousios, G. (2013). The ghtorent dataset and tool suite. In *Zimmermann et al.* (Zimmermann et al. 2013) (pp. 233–236). https://doi.org/10.1109/MSR.2013.6624034.

Hamasaki, K., Kula, R.G., Yoshida, N., Cruz, A.E.C., Fujiwara, K., & Iida, H. (2013). Who does what during a code review? Datasets of OSS peer review repositories. In *Zimmermann et al.* (Zimmermann et al. 2013) (pp. 49–52). https://doi.org/10.1109/MSR.2013.6624003.

Hassan, A.E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008* (pp. 48–57). IEEE.

Jeong, G., Kim, S., Zimmermann, T., & Yi, K. (2009). 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)* (pp. 1–18).

Kagdi, H.H., Collard, M.L., & Maletic, J.I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, *19*(2), 77–131. https://doi.org/10.1002/smr.344.

Kim, S., Whitehead, E.J., Jr., & Zhang, Y. (2008). Classifying software changes: Clean or buggy?. *IEEE Transactions on Software Engineering*, *34*(2), 181–196. https://doi.org/10.1109/TSE.2007.70773.

Knuth, D.E. (1981). *The art of computer programming, volume II: seminumerical algorithms*, 2nd Edn. Addison-Wesley.

Kruskal, W.H., & Wallis, W.A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, *47*(260), 583–621.

Lee, J.B., Ihara, A., Monden, A., & Matsumoto, K. (2013). Patch reviewer recommendation in OSS projects. In Muenchaisri, P., & Rothermel, G. (Eds.) *20th Asia-Pacific software engineering conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2* (pp. 1–6). IEEE Computer Society. https://doi.org/10.1109/APSEC.2013.103.

Levene, H. (1960). Robust tests for equality of variances1. *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, *2*, 278–292.

Liu, X., Suel, T., & Memon, N.D. (2014). A robust model for paper reviewer assignment. In Kobsa, A., Zhou, M.X., Ester, M., & Koren, Y. (Eds.) *Eighth ACM conference on recommender systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014* (pp. 25–32). ACM. http://doi.org/10.1145/2645710.2645749.

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A.E. (2014). The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In Devanbu, P.T., Kim, S., & Pinzger, M. (Eds.) *Proceedings of the 11th working conference on mining software repositories, MSR 2014, May 31 - June 1, 2014, Hyderabad, India* (pp. 192–201). ACM. http://doi.org/10.1145/2597073.2597076.

Nelson, S.D., & Schumann, J. (2004). What makes a code review trustworthy? In *37th Hawaii international conference on system sciences (HICSS-37 2004), CD-ROM / Abstracts Proceedings, 5-8 January 2004, Big Island, HI, USA*. IEEE Computer Society. https://doi.org/10.1109/HICSS.2004.1265711.

Petrovsky, A.B. (2012). An axiomatic approach to metrization of multiset space. In *Multiple criteria decision making: proceedings of the 10th international conference: expand and enrich the domains of thinking and application* (p. 129). Springer.

Radjenovic, D., Hericko, M., Torkar, R., & Zivkovic, A. (2013). Software fault prediction metrics: A systematic literature review. *Information & Software Technology*, *55*(8), 1397–1418. http://doi.org/10.1016/j.infsof.2013.02.009.

Rigby, P.C., & Storey, M.D. (2011). Understanding broadcast based peer review on open source software projects. In Taylor, R.N., Gall, H.C., & Medvidovic, N. (Eds.) *Proceedings of the 33rd international conference on software engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011* (pp. 541–550). ACM. https://doi.org/10.1145/1985793.1985867.

Singh, D., Ibrahim, A., Yohanna, T., & Singh, J. (2007). An overview of the applications of multisets. *Novi Sad Journal of Mathematics*, *37*(3), 73–92.

Tang, W., Tang, J., & Tan, C. (2010). Expertise matching via constraint-based optimization. In Huang, J.X., King, I., Raghavan, V.V., & Rueger, S. (Eds.) *2010 IEEE/WIC/ACM international conference on web intelligence, WI 2010, Toronto, Canada, August 31 - September 3, 2010, main conference proceedings* (pp. 34–41). IEEE Computer Society. https://doi.org/10.1109/WI-IAT.2010.133.

Terra, R., Brunet, J., Miranda, L.F., Valente, M.T., Serey, D., Castilho, D., & da Silva Bigonha, R. (2013). Measuring the structural similarity between source code entities (S). In *The 25th international conference on software engineering and knowledge engineering, Boston, MA, USA, June 27-29, 2013* (pp. 753–758). Knowledge Systems Institute Graduate School.

Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., & Matsumoto, K. (2015). Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In Guéhéneuc, Y., Adams, B., & Serebrenik, A. (Eds.) *22nd IEEE international conference on software analysis, evolution, and reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015* (pp. 141–150). IEEE. https://doi.org/10.1109/SANER.2015.7081824.

van der Veen, E., Gousios, G., & Zaidman, A. (2015). Automatically prioritizing pull requests. In *12th IEEE/ACM working conference on mining software repositories, MSR 2015, Florence, Italy, May 16-17, 2015* (pp. 357–361). IEEE. https://doi.org/10.1109/MSR.2015.40.

Voorhees, E.M. (1999). The TREC-8 question answering track report. In Voorhees, E.M., & Harman, D.K. (Eds.) *Proceedings of the eighth text retrieval conference, TREC 1999, Gaithersburg, Maryland, USA, November 17-19, 1999, vol. special publication 500-246. National institute of standards and technology (NIST)*. http://trec.nist.gov/pubs/trec8/papers/qa_report.pdf.

Wang, F., Chen, B., & Miao, Z. (2008). A survey on reviewer assignment problem. In Nguyen, N.T., Borzemski, L., Grzech, A., & Ali, M. (Eds.) *Proceedings of the new frontiers in applied artificial intelligence, 21st international conference on industrial, engineering and other applications of applied intelligent systems, IEA/AIE 2008, Wroclaw, Poland, June 18-20, 2008, lecture notes in computer science* (Vol. 5027, pp. 718–727). Springer. https://doi.org/10.1007/978-3-540-69052-8_75.

Xie, H., & Lui, J.C.S. (2012). Mathematical modeling of competitive group recommendation systems with application to peer review systems. arXiv:1204.1832.

Yarowsky, D., & Florian, R. (1999). Taking the load off the conference chairs: towards a digital paper-routing assistant.

Yu, Y., Wang, H., Yin, G., & Ling, C.X. (2014). Reviewer recommender of pull-requests in github. In *30th IEEE international conference on software maintenance and evolution, Victoria, BC, Canada, September 29 - October 3, 2014* (pp. 609–612). IEEE Computer Society. https://doi.org/10.1109/ICSME.2014.107.

Yu, Y., Wang, H., Yin, G., & Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information & Software Technology*, *74*, 204–218. https://doi.org/10.1016/j.infsof.2016.01.004.

Zimmermann, T., Penta, M.D., & Kim, S. (Eds.) (2013). *Proceedings of the 10th working conference on mining software repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. IEEE Computer Society. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6597024.