Applied Soft Computing Journal 100 (2021) 106908

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review

Moataz Chouchen [a], Ali Ouni [a,*], Mohamed Wiem Mkaouer [b], Raula Gaikovina Kula [c], Katsuro Inoue [d]

[a] ETS Montreal, University of Quebec, Montreal, QC, Canada
[b] Rochester Institute of Technology, Rochester, NY, USA
[c] Nara Institute of Science and Technology, Nara, Japan
[d] Osaka University, Osaka, Japan

ARTICLE INFO

Article history:
Received 10 April 2020
Received in revised form 2 September 2020
Accepted 11 November 2020
Available online 30 November 2020

Keywords:
Modern code review
Software quality
Code reviewers recommendation
Search-based software engineering

ABSTRACT

Contemporary software development is distributed and characterized by high dynamics with continuous and frequent changes to fix defects, add new user requirements or adapt to other environmental changes. To manage such changes and ensure software quality, modern code review is broadly adopted as a common and effective practice. Yet several open-source as well as commercial software projects have adopted peer code review as a crucial practice to ensure the quality of their software products using modern tool-based code review. Nevertheless, the selection of peer reviewers is still merely a manual and hard task especially with the growing size of distributed development teams. Indeed, it has been proven that inappropriate peer reviewers selection can consume more time and effort from both developers and reviewers and increase the development costs and time to market. To address this problem, we introduce a multi-objective search-based approach, named WhoReview, to find the optimal set of peer reviewers for code changes. We use the Indicator-Based Evolutionary Algorithm (IBEA) to find the best set of code reviewers that are (1) most experienced with the code change to be reviewed, while (2) considering their current workload, *i.e.*, the number of open code reviews they are working on. We conduct an empirical study on 4 long-lived open source software projects to evaluate our approach. The obtained results show that WhoReview outperforms state-of-the-art approach by an average precision of 68% and recall of 77%. Moreover, we deployed our approach in an industrial context and evaluated it qualitatively from developers perspective. Results show the effectiveness of our approach with a high acceptance ratio in identifying relevant reviewers.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

Modern Code Review (MCR) [1,2] is becoming a broadly adopted software engineering practice in both commercial and open-source software (OSS) projects. Code review [3,4] is defined as the process of reviewing other developers' code to ensure software quality, and find potential problems in their code changes before they are merged with the codebase. MCR derives from the formal and disciplined process of software inspection [5], which requires synchronous face-to-face meetings among developers to make a checklist-based code inspection and interactive discussion. Conversely, MCR provides practitioners with a convenient environment to read and discuss code changes and makes this activity lightweight, less formal and asynchronous through a tool specialized support for geographically distributed code review [1,6]. There is an increasing number of available MCR platforms including Gerrit [7], ReviewBoard [8], and Phabricator [9].

Although existing MCR platforms provide effective and valuable support to automate the review process, a considerable human effort is still involved. Hence, in typical software projects, developers who submit code changes face the regular challenge of identifying peer code reviewers to check their changes for quality assurance. Such selection is mainly based on the reviewers expertise with the specific modified files in the code change as well as previous review collaborations [3,10,11]. One of the main limitations in MCR tools is that reviewers assignment is still a manual task, and there are no awareness mechanisms to find and automatically assign adequate reviewers when code changes are submitted for review. Prior works demonstrated that bad reviewers assignment negatively impacts the review process and software quality [11–13].

Finding and assigning suitable code reviewers is a non-trivial decision-making task in software engineering involving several

* Corresponding author.
E-mail addresses: ali.ouni@etsmtl.ca, ouniaali@gmail.com (A. Ouni).

https://doi.org/10.1016/j.asoc.2020.106908
1568-4946/© 2020 Elsevier B.V. All rights reserved.

considerations. As code changes can affect many sub-modules and/or files in a software project, then typically the review involves several reviewers of each affected module or file. Moreover, in typical software projects, source code files are generally edited by several developers, and reviewed by several reviewers. At the same time, developers and reviewers could be assigned to different software engineering tasks with different workloads. It is thus more difficult to identify suitable peer reviewers especially when the number of changed modules/files is considerable and reviewers are becoming overloaded with different open code reviews.

Automating the code reviewers selection process is an essential task. It decreases the lead time for a review; and balances the workload onto a more comprehensive set of developers. An automated approach would also allow an increase in the number of potential reviewers.

Various approaches have recently been proposed to leverage the rich MCR data in order to ensure support to the code review process. GitHub Enterprise Cloud recently adopted a simplified round robin-based algorithm to assign reviewers [14] based on who has received the least recent review request regardless their expertise. Other existing approaches rely on the expertise with the code fragments being reviewed to identify potential reviewers [10,11,13,15–18]. Moreover, various empirical studies showed that reviewers complete code reviews faster and share quality feedback with the author when reviewers already have knowledge and experience with the code and the project [1,6,19]. Furthermore, since code review is, in essence, a human-centric process, the personal and socio-technical aspects play an extremely important role in identifying peer code reviewers as pointed out by several researchers [3,10,12,20,21]. However, most of these existing studies have tackled the problem of reviewers recommendation from a single perspective, *e.g.*, reviewers expertise and/or socio-technical aspects while ignoring an important factor which is the *reviewers workload*. That is, reviewers are assigned regardless of the number of outstanding reviews they currently have. As a consequence, such reviewers recommendation systems oftentimes involve a large number of reviews to a small set of reviewers (*e.g.*, 80% of the reviews in a project are reviewed by 20% of the developers) as pointed out by Asthana et al. [11] in their study with Microsoft. Thus, a better assignment of reviews across available reviewers is crucial for effective recommendation systems for MCR to balance the workload onto a more comprehensive set of reviewers.

To address this issue, we propose a novel approach, named WhoReview, that formulates the problem of code reviewers identification as a multi-objective search based problem to identify the set of appropriate developers while (1) maximizing the expertise of the recommended reviewers with the changed files, and (2) better managing the workload of reviewers by providing a better distribution of code reviews across available reviewers. Our proposed approach, WhoReview, adopts the indicator-based evolutionary algorithm (IBEA) [22] to find the best trade-off between the reviewers expertise and collaboration and the reviewers workload. Various parameters are involved when adopting any multi-objective evolutionary algorithms, such as IBEA, including the solution encoding, change operators rates such as crossover and mutation, the size of the initial population, the number of iterations, etc. In particular, we build on top of our previous research work [10] to formulate the code reviewers recommendation problem as a multi-objective search-based problem to incorporate the workload of reviewers as a new dimension in the recommendation process in addition to the reviewers expertise and their previous collaboration history.

We conduct an empirical study to evaluate WhoReview on a benchmark of four long-lived OSS projects from diverse domains, OpenStack [23], LibreOffice [24], Android [25], and Qt [26]. The empirical results show that WhoReview is efficient in recommending peer reviewers. The statistical analysis of the results shows that WhoReview performs better than four recent state-of-the-art approaches [10,13,16,27]. Moreover, we conduct a qualitative industrial case study to evaluate our approach with two large projects with our industrial partner. The results show that WhoReview provides more relevant reviewers recommendations compared to baseline techniques.

This paper extends our previous research work, RevRec [10], published in the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), in the following ways:

- We extend our single objective approach [10] into a novel multi-objective search-based formulation, namely WhoReview, that adopts indicator-based evolutionary algorithm (IBEA) to balance reviewers *expertise* and *workload*.
- We conduct an extensive empirical study on a benchmark[1] of 157,467 code reviews [28] from four OSS projects to evaluate the implementation of our WhoReview approach with different state-of-the-art techniques. The statistical analysis of the experimental results indicates that WhoReview can correctly recommend code reviewers wit an average precision of 68% and recall of 77%.
- We deploy our approach in an industrial context and qualitatively evaluate it from developer's perspective and compare it with baseline approaches. The obtained results show that our WhoReview approach is adequate with software developers needs and provides relevant peer reviewers recommendations.

The remainder of the paper is structured as follows. Section 2 gives an overview of MCR and summarizes the related work. Section 3 describes our approach, WhoReview, that formulate the code reviewers recommendation problem as a multi-objective search based optimization. Section 4 reports and discusses the results of our experiments to evaluate WhoReview. Section 5 describes potential threats to validity and Section 6 draws our conclusions.

2. Background and related work

This section describes the MCR process, and the reviews the related work.

2.1. Modern code review

Fig. 1 shows an abstraction of the key code review activities that take place in a Modern Code Review (*i.e.*, MCR) setting. Open Source Software (OSS) projects utilize online collaboration tooling such as Gerrit [7] to coordinate code review activities. For instance, OpenStack [23] has integrated Gerrit into their developer workflow [29]. Below, we highlight the three main steps in the code review process and the key activities.

1. **Request–Review:** First, the authors request for their new code changes (*i.e.*, patch) to be reviewed. The assignment is completed once an appropriate set of reviewers are assigned to the review.
2. **Review–Revision:** Once the review is assigned, reviewers will discuss and make comments on the submitted patch. In cases where there is rework needed, the feedback is sent back to the author. The author will then revise the submitted code change by making the appropriate changes based on the discussions of the reviewers. This cycle between review and revision may last several iterations.

[1] https://kin-y.github.io/miningReviewRepo.

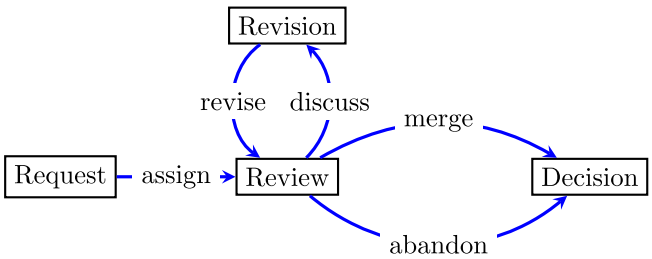

Fig. 1. The code review process showing key activities.

3. **Review–Decision:** Once there is a final consensus on the review and all discussions and revisions are satisfied, the review–decision is made. The end result is either a merged patch in the existing codebase or abandonment of the patch. During this step, there will be testing of the patch to ensure successful integration.

For instance, Fig. 2 shows an illustrative example of a code review from the Eclipse project[2] using the Gerrit code review platform. This code review involves a total of 9 changes files submitted by "Bob" and reviewed by four reviewers ("Paul", "Jonathan", "Andrew", and "Daniel") as well as an automated bot ("CI bot"). Finally, the reviewer "Jonathan" has voted "+2" to approve the code change and merge it into the code base.

In this study, we are interested in maintaining the efficiency of the code review process, especially targeting the assignment of reviewers, as shown in step 2. Finding the right reviewers is not trivial, especially in large diverse OSS teams that collaborate on a voluntary basis. The main reason is that usually the most expert reviewer is currently assigned to other open reviews that have not reached a decision, which significantly increases her/his workload.

2.2. Related work

Related work involves (*i*) factors influencing the code review process, and (*ii*) reviewer recommendation methods.

Peer code reviewers algorithms and implementations. Balachandran [27] was one of the first researchers to motivate the use of the ReviewBot tool, as a recommendation system to improve the review quality in an industrial context while considering the human effort, where formal in-person meetings are not possible. While Balachandran attempted to reduce the human effort, his approach considered mainly automated static code analysis and ignored the workload of reviewers. Patanamon et al. [13] designed RevFinder, to assist in the selection of reviewers in MCR based on the history of revised files to find the most experienced reviewer. Later, Zanjani et al. [16] introduced cHRev as an automated approach to recommend peer reviewers for MCR based on the expertise of the reviewers, extracted from historical review data. The built model leverages the number of review comments along with their time span. Xia et al. [15] proposed, Tie, a hybrid and incremental approach based on text mining and file location to find reviewers having experience with the files or similar files being reviewed. Fejzer et al. [17] proposed a selection method based on the profiles of individual developers. The profile corresponds to a multi-set of all file path tokens identified in the change commits that she/he reviewed which will be updated with new reviews. The approach uses a similar approach to [13] to calculate the similarity between available profiles and changed files in a submitted code change. Kim et al. [30] introduced an approach based on latent Dirichlet allocation based on reviewer expertise to identify reviewers. Reviewers expertise is based on the code change topics extracted from the project's review history. However, these existing techniques tend to be based mainly on the reviewers expertise while ignoring the workload.

Recently, Asthana et al. [11] used the concept of hybrid recommendation and load balancing. The authors have conducted an online evaluation at Microsoft and showed that efficient recommendation brings several advantages including the reduction of reviews completion time, the total number of reviewers per code review, and the reviewers active load. Ouni et al. [10] introduced RevRec, a search based approach using a single-objective genetic algorithm to identify peer reviewers based on their expertise and their social collaboration network. While Ouni et al.'s approach demonstrated that search-based techniques provide an efficient way to identify relevant reviewers, it only considers the review problem from a single perspective while ignoring the workload of the reviewers. Besides code review, other approaches focused on the recommendation of appropriate developers for different software engineering tasks. Anvik et al. [31] applied machine learning algorithms to the problem of developers assignment to bug reports. Later, Xia et al. [32] proposed an automated approach to assign developers to resolve bugs based on developers information. However, these recommendation methods have not been without critic, with Kovalenko et al. [6] arguing that the next generation of recommendation tools must include more user-centric approaches.

For the sake of clarity, we summarize in Table 1 the existent approaches for peer reviewer recommendation approaches.

Factors influencing the code review process. Ruangwan et al. [33] showed that human aspects are of crucial considerations in the review process. Baysal et al. conducted various studies to assess non-technical and human factors in MCR. Their key findings suggest that (1) organizational, personal and participation dimensions have an impact on the review process [12,34], and (2) the review process can be sensitive due to its innate nature to deal with people's characters and egos [35]. Moreover, other works advocate the importance of developers reputation to receive a faster first review response and feedback [3,36]. Other studies have shown that there is an interdependence between reviews, aiding in the assignment of the best reviewer [37]. Ebert et al. [38] later showed how human factors play a role in confusion during the review. More recent work observed a positive relationship between patch authors and their reviewers does play a role in the reviewer accepting a proposed patch [39]. It is widely accepted that code review is a hard process that involves different socio-personal aspects [3,12,34,38,40]. Indeed, historically, Fagan introduced the software inspection concept as a systematic and disciplined peer review process for software quality assurance [41]. Moreover, to better analysis MCR practices, in OSS projects in which developers basically participate in code review in a volunteer basis, a number of empirical studies found that a number of human and socio-technical factors do influence the peer review process in OSS, advocating the need for more automated support to the review process through reviewers recommendation systems [3,12,19,20,42–44].

3. Approach

This section describes our approach WhoReview for code reviewers recommendation using the indicator based evolutionary algorithm.

[2] https://git.eclipse.org/r/c/tracecompass/org.eclipse.tracecompass/+/78803, for privacy reasons, we have hidden some details from the interface, and used alias instead of the original developer names and emails.

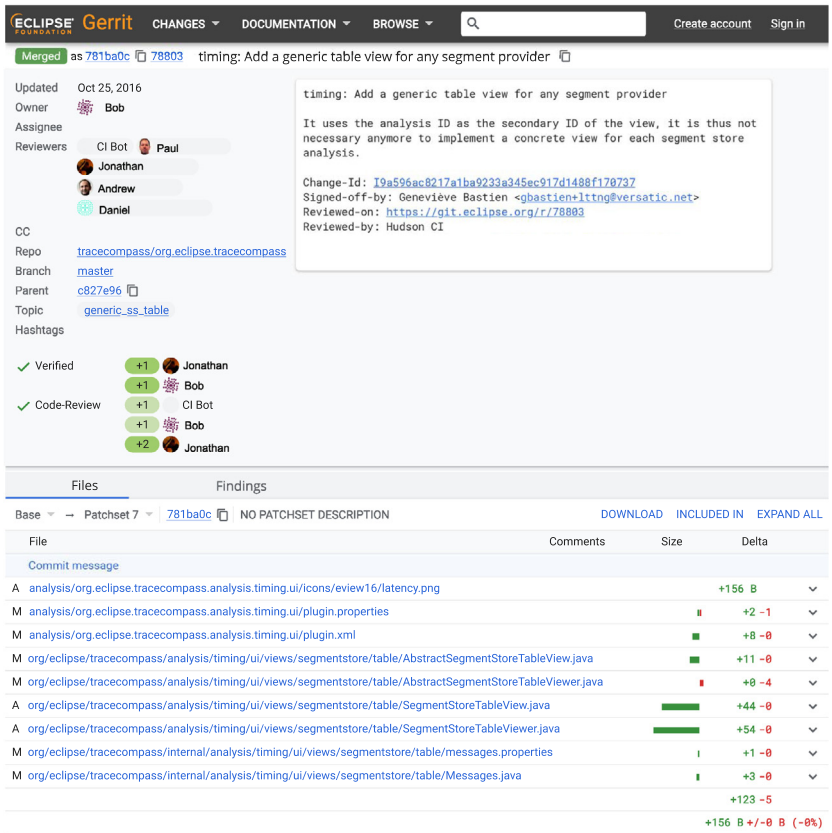

Fig. 2. An example of a code review from the Eclipse project using Gerrit.

Table 1
A summary of existent works for code reviewers recommendation.

Study	Year	Method	User study
Balachandran [27]	2013	ReviewBot: static analysis to find reviewers who previously modified or reviewed the submitted sections of code.	Yes
Patanamon et al. [13]	2015	RevFinder: rank reviewers based on the history of revised files to find the most experienced reviewer.	No
Zanjani et al. [16]	2015	cHRev : find reviewers based on their expertise by leveraging the number of review comments along with their time span.	No
Xia et al. [15]	2015	Tie: a hybrid and incremental approach based on text mining and file location to find reviewers having experience with the files or similar files being reviewed.	No
Ouni et al. [10]	2016	RevRec: a search based approach to find reviewers based on their experience and social network.	No
Fejzer et al. [17]	2018	A selection method based on the profiles of individual developers by leveraging a multi-set of all file path tokens identified in the reviewer's change commits.	No
Kim et al. [30]	2018	A latent Dirichlet allocation approach based on reviewer expertise to identify reviewers based on the code change topics extracted from the project's review history.	No
Asthana et al. [11]	2019	WhoDo: A hybrid approach with scoring function to prioritize reviewers and balance load	Yes

3.1. Approach overview

Fig. 3 depicts an overview of WhoReview. The input is a code change that is being submitted for review, which is composed of a number of changed source code files (*i.e.*, patch) committed by a developer. It takes as input also the history of previous and current code reviews collected from the review platform in which the project is hosted, *e.g.*, Gerrit.[3] As output, WhoReview returns a set of most suitable reviewers to be assigned to review a code change. To search for the best set of candidate reviewers, our approach uses the indicator-based evolutionary algorithm (IBEA) [22]. The search algorithm evolves towards finding the optimal trade-off between two objectives (1) the reviewers *expertise* based on their experience with the source code files in previous code reviews and the reviewers collaboration with the developer who submitted the code change, and (2) the workload of the currently recommended reviewers. In the next subsections, we present the necessary details on the problem formulation and explain how we adapted IBEA for the reviewers recommendation problem in MCR.

3.2. Problem formulation

A typical software project S consists of a set of n developers, $D = \{d_1, \ldots, d_n\}$, and set of m reviewers $R = \{r_1, \ldots, r_m\}$, and a set of q source code files, $F = \{f_1, \ldots, f_q\}$. A source code change C is performed by a developer $d \in D$. A code change comprises a set of t changed files $F_c = \{f_{c_1}, \ldots, f_{c_t}\}$.

Let $R_c = \{r_1, \ldots, r_k\}$ be a set of candidate reviewers for the code change C. Each reviewer r_i has (1) his own expertise

[3] https://www.gerritcodereview.com/.

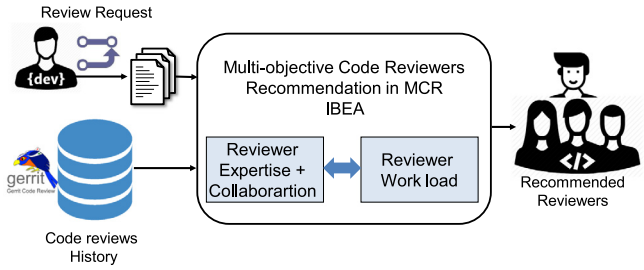

Fig. 3. An overview of the WhoReview approach.

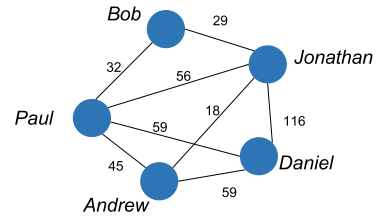

Fig. 4. An illustrative example of the reviewer collaboration (RC) model.

with the files F_p, (2) his history of review collaborations with the developer d and the reviewers $R_c \setminus r_i$, and (3) his current workload, *i.e.*, the number of open code reviews. In particular, we define three models for the code review recommendation problem (1) reviewers expertise model, (2) reviewers collaboration model, and (3) reviewers workload model.

3.2.1. Reviewer Expertise (RE)

In a software project, each individual reviewer r_i has a level of expertise with each of the changed files $F_c = \{f_{c_1}, \ldots, f_{c_t}\}$. The expertise of a given reviewer r_i with the code change files F_c is denoted as $E(r_i, F_c)$ and calculated as follows:

$$E(r_i, F_c) = \sum_{\forall f_{c_j} \in F_c} Exp(r_i, f_{c_j}) \tag{1}$$

where the function $Exp(r_i, f_{c_j})$ computes the expertise level of a given reviewer r_i with the file $f_{c_j} \in F_c$. The value of $Exp(r_i, f_{c_j})$ is based on the history of previous reviews accomplished by r_i based on the number of similar files that are reviewed using the file path similarity [13,16,45]. For a pair of files, we consider the Jaccard similarity between their path tokens, higher than a threshold of 40%, obtained using the camel case splitter.

We build our expertise model, RE, based on the review comments (1) *frequency*, and (2) *recency*. The more and recent the comments of the reviewers r_i on the file f the more expertise he has with [10,13,16]. Indeed, as expertise may change over time, we consider both reviews frequency and recency as primary factors to capture the reviewers expertise. We thus define the $Exp(r_i, f)$ as follows:

$$Exp(r_i, f) = \sum_{\forall f_i \in Jaccard(f)} frequency(r_i, f) \times recency(r_i, f) \tag{2}$$

where $frequency(r_i, f)$ calculates the number of review comments by the reviewer r_i for the file f as well as the set its similar files, if any, having a Jaccard similarity [46] higher than a specific threshold value. The function $recency(r_i, f)$ reflects the freshness of the review comments and is calculated as the complement of the number of days since the last comment of the reviewer r_i to the file f, normalized with respect to the total number of calendar days from the project's creation.

To illustrate the expertise model, let us consider a simplified example of a developer, "Jean", who is a candidate reviewer to review the code change illustrated in Fig. 2. For sake of simplicity, we suppose that (1) the code change involves only the file `org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/SegmentStoreTableView.java`, denoted as F, and (2) the project has started one year ago, *i.e.*, 365 days. Let "Jean" a candidate reviewer who has a review experience with various files in the project, as shown in Table 2.

The changed file, F, has a Jaccard similarity equals or higher than 40% with 3 out of the 8 files from the review history of Jean, $F5$, $F6$ and $F8$, based on Table 2. For example, Jean's review recency with the file F5, `SegmentStoreFatory.java`, $recency(Jean, F5) = 1 - 8/365 = 0.98$, while the review frequency, $frequency(Jean, F5) = 27$. Then, based on Eq. (2), the expertise of Jean with the file F is equal to the sum of frequency multiplied by recency of the 3 files F5, F6 and F8, $Exp(Jean, F) = 27 \times 0.98 + 2 \times 0.69 + 9 \times 0.99 = 173.37$. Following the same way, the experience could be calculated for multiple files involved in a given code change.

Finally, the expertise, RE, of a (set of) reviewer(s) R_c with the code change files F_c is calculated as follows:

$$RE(R_c, F_c) = \frac{\sum_{\forall r_i \in R_c} E(r_i, F_c)}{|R_c|} \tag{3}$$

where $E(r_i, F_c)$ is provided by Eq. (1).

3.2.2. Reviewer Collaboration (RC)

In a software project, each individual reviewer r_i have its own review collaboration (RC) record with (i) the developer who submitted the code change C, and/or (ii) the rest of reviewers $R_c \setminus r_i$ based on the reviews history. The review collaboration forms a social network among all contributors in the project which could be represented as a weighted, undirected graph $G = (V, E)$. The vertices V represent the contributors (*i.e.*, code developers and reviewers), while the edges E express the strength of collaboration among them as a count of the number of interactions in terms of the frequency of review comments during their co-reviews history. For a code change C, the developer and potential reviewers can be represented as a sub-graph $G_c = (V_c, E_c)$ [10]. Our collaboration model is built on G_c based on two measures, (1) the sub-graph connectivity, and (2) the sum of weights on the edges (*i.e.*, comments count). For a developer d, the RC score is calculated as follows:

$$RC(d, R_c) = \frac{|E_c|}{|V_c| \times (|V_c| - 1)/2} \times \sum_{\forall e_{c_i} \in E_c} e_{c_i} \tag{4}$$

where the coefficient $\frac{|E_c|}{|V_c| \times (|V_c|-1)/2} \in [0, 1]$ reflects the connectivity of the sub-graph (a value of 1 indicates higher connectivity, if the graph is complete), and e_{c_i} is the weight on the edge e_i, *i.e.*, the total number of review comments exchanged among each pair of contributors.

To illustrate the RC model, we consider the example shown in Fig. 2, with the four reviewers, Paul, Jonathan, Andrew and Daniel, denoted as R_c. The developer Bob, denoted as d, who submitted the code change has prior collaboration with only Jonathan and Paul. The collaboration among them forms the graph shown in Fig. 4, where the edge weights indicate the total number of review comments between two contributors. Then, the review collaboration between Bob and the four reviewers is calculated based on Eq. (4) as follows, $RC(d, R_c) = \frac{8}{5 \times (5-1)/2} \times (32 + 29 + 56 + 59 + 18 + 45 + 116 + 59) = 331.2$. The more the graph is connected and the weights on the edges are high, the more the RC score is high.

Table 2
A simplified example of the review experience of the reviewer "Jean".

ID	File	# of comments	# of days since last change
F1	org/eclipse/tracecompass/internal/segmentstore/core/segmentHistoryTree/SegmentHistoryTree.java	6	20
F2	org/eclipse/tracecompass/internal/provisional/segmentstore/core/BasicSegment2.java	21	65
F3	org/eclipse/tracecompass/internal/provisional/segmentstore/core/ISegment2.java	32	2
F4	org/eclipse/tracecompass/internal/segmentstore/core/segmentHistoryTree/SegmentTreeNode.java	43	37
F5	org/eclipse/tracecompass/segmentstore/core/SegmentStoreFactory.java	27	8
F6	org/eclipse/tracecompass/segmentstore/core/ISegmentStore.java	2	112
F7	org.eclipse.jgit.junit/src/org/eclipse/jgit/junit/MockSystemReader.java	15	14
F8	org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/AbstractSegmentStoreTableView.java	9	2

Table 3
A simplified example of the review workload model.

File	Average review time
analysis/org.eclipse.tracecompass.analysis.timing.ui/icons/eview16/latency.png	1.2
analysis/org.eclipse.tracecompass.analysis.timing.ui/plugin.properties	7.8
analysis/org.eclipse.tracecompass.analysis.timing.ui/plugin.xml	3.5
org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/AbstractSegmentStoreTableView.java	4.9
org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/AbstractSegmentStoreTableViewer.java	13.4
org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/SegmentStoreTableView.java	7.2
org/eclipse/tracecompass/analysis/timing/ui/views/segmentstore/table/SegmentStoreTableViewer.java	4.5
org/eclipse/tracecompass/internal/analysis/timing/ui/views/segmentstore/table/messages.properties	2.8
org/eclipse/tracecompass/internal/analysis/timing/ui/views/segmentstore/table/Messages.java	4.1

3.2.3. Reviewer Workload (RW)

In addition to developers expertise and collaboration, each reviewer in a project has her/his own workload that typically changes overtime based on the number of open reviews s/he is working on. Our RW model is primarily based on the number of current open reviews. However, since not all code changes have the same difficulty to review them, we also consider the notion of code change size to better estimate the workload of reviewers. That is, the larger the code change size, the higher is the reviewer workload. Let, r a reviewer who has n open reviews for the code changes $C = \{c_1, \ldots, c_n\}$ assigned to her/him for review with the status "open". The reviewer workload $RW(r)$ is calculated as follows :

$$RW(r) = \sum_{\forall c_i \in C} size(c_i) \times load(c_i) \tag{5}$$

where $size(c_i)$ returns the size of the code change c_i in terms of the number of files modified. Since different files may require different workloads, we consider the history of code changes by adding the weight function $load(c_i)$ that estimates the workload required to review the code change c_i based on the average time spent to review each file in c_i in the project's review history. If a file has no change history, it is then assigned a default weight equals to 1. Let $F = \{f_1, \ldots, f_m\}$ be the set of files modified in the code change c_i, the function $load(c)$ is calculated as follows.

$$load(c) = \frac{1}{\mid F \mid} \sum_{\forall f_i \in F} \frac{hist_time(f_i)}{max_time} \tag{6}$$

where $hist_time(f_i)$ returns the average time, in terms of calendar days, taken in all previously reviewed code changes involving the file f_i, and max_time is the time taken for the longest code review in the project history, after removing outlier values using a boxplot-based technique.

To illustrate the reviewer workload (RW) model, we consider a reviewer, Paul, who has 1 open code change c_1 to be reviewed, as shown in Fig. 2 from the Eclipse project. The size of code change $size(c_1) = 9$ (*i.e.*, the total number of changed files). Suppose that each file f_i had an average review time in its history $hist_time(f_i)$ as shown in Table 3, and that the longest review took 48 days. Then, $load(c_1) = \frac{1}{9} \times (\frac{1.2+7.8+3.5+4.9+13.4+7.2+4.5+2.8+4.1}{48}) = 0.11$. Hence, based on Eq. (5), the workload of the reviewer Paul, $RW(Paul) = 9 \times 0.11 = 0.99$. Similarly to $load(c_1)$, if the reviewer Paul has more than one open review, then his workload $RW(Paul)$ corresponds to the sum of the load of all open reviews.

The three models, *RE*, *RC*, and *RW*, aim at providing a general picture on the code review process while shedding the light on different aspects. Having reviewers with sufficient expertise with the code change being reviewed is of paramount importance for quality review. The socio-technical factor and social network among developers and reviewers are also crucial [4,10,36,47]. Moreover, balancing the reviewers workload helps to better distribute the number of reviews across all available reviewers within a project. It also averts assigning an unfairly high workload to a few number of active and experienced developers, who have actively participated in multiple code reviews and regularly committed to different parts of the project's codebase [10,11].

3.3. IBEA adaptation

The generic nature of evolutionary algorithms, such as IBEA, requires a specific adaptation to turn it into a domain-specific algorithm. The adaptation is composed of 3 main elements: (*i*) solution representation, (*ii*) solution evaluation, and (*iii*) solution evolution.

3.3.1. Solution representation

A candidate solution to the reviewer recommendation problem, is a subset of reviewers, selected from all developers, currently actively contributing to the project. We encode the set of reviewers using a vector of length n. Each dimension of the vector contains a selected potential reviewer, through a unique numerical ID, identifying each reviewer. The vector length is variable, as it represents the number of selected subset of reviewers. The length of a vector can change during the solution's evolution, within an upper threshold *maxSize*, previously determined as part of the algorithm tuning, or chosen by the decision maker. The reviewers order in the vector representation is not important.

3.3.2. Solution evaluation

Each candidate solution should be quantitatively evaluated via a fitness function. The fitness function is composed of one or many objective functions that aim at quantifying how good is each individual, *i.e.*, reviewers recommendation list, in a population of solutions. To evaluate how good is each candidate solution, *i.e.*, its fitness, we employ two objective functions to be optimized (1) the reviewers expertise and collaboration, and (2) the reviewers current workload. An optimal solution is expected to provide the optimal trade-off between both objective functions.

A. Maximize reviewers expertise and collaboration (REC). Inspired from [10], we use an expertise and collaboration model that combine both the history of previous reviews and the history of collaboration between developers. Prior work has shown that there is a high correlation and complementary between both reviewers expertise and collaboration [10], so they should be aggregated into one single objective function. For a candidate solution, R, that consists of a set of n reviewers $R = \{r_1, \ldots, r_n\}$ to review a code change C submitted by a developer d, the reviewers expertise and collaboration is calculated as follows:

$$REC(R, C) = \alpha \times RE(R, F) + \beta \times RC(d, R) \tag{7}$$

where the functions $RE(R, F)$, and $RC(d, R)$ represent the reviewers expertise and collaboration, as given by Eqs. (3) and (4), respectively, and the weights $\alpha + \beta = 1$. This objective function should be maximized

B. Minimize reviewers work load. For a candidate solution, $R = \{r_1, \ldots, r_n\}$, that consists of a set of n reviewers, the solution's workload corresponds to the average workload value for each individual reviewer r_i. The workload is calculated as follows:

$$Workload(R) = \frac{\sum_{\forall r_i \in R} RW(r_i)}{n} \tag{8}$$

where $RW(r_i)$ calculates the current workload of the reviewer r_i and provided by Eq. (5). This objective function should be minimized to assign more developers with low workload. Evolving towards the near-optimal trade-off between both reviewers expertise and workload are the main drivers of the search process.

Reviewers ranking list. Since IBEA returns a set of reviewer lists, in the Pareto front, without raking, we employ a ranking technique that takes advantage of the optimal (*i.e.*, non-dominated) solutions returned by the algorithm. The final ranking of the optimal recommended reviewers corresponds to the number of occurrences of the reviewer ID in all the near-optimal solutions returned in the Pareto front. The more a reviewer appears in the non-dominated solutions, the more relevant s/he is.

3.3.3. Solution evolution

Any evolutionary algorithm relies on change operators to evolve its solution(s). For instance, IBEA deploys the crossover and mutation, to transition the solution's phenotype into a preferably enhanced one. Practically, IBEA randomly initiate a population of several vectors of random sizes. This population evolves through crossover and mutation into a newer population whose solutions are close to a near-optimal state, in comparison with the solution of the previous population. The genetic operators, used in this process, are detailed below:

Crossover is a binary recombination operator that takes as input two parent solutions, and cross them over to produce two offspring solutions. For the case of our problem adaptation, the single random cut-point crossover is deployed. The crossover process is initiated by randomly splitting each parent into two sub-vectors. Afterward, the first offspring is constructed by concatenating the left sub-vector of one parent with the right sub-vector of the second parent, and vice versa for the second offspring. To ensure the consistency of each solution, there are pre and post conditions that need to be maintained. These conditions are domain-specific. For instance, the two offsprings are analyzed for consistency, and if a solution becomes inconsistent, by containing two identical identification numbers, of the same reviewer, issued from parent solutions, a repair operator removes the duplicated reviewers.

As for the *mutation* operator, it is an unary operator that generates an offspring by slightly altering a parent solution. Mutation aims to explore mutated solutions of a given one, through a slight change, unlike recombination operators which tend to generate completely new solutions. For our problem, the mutation operator randomly selects one dimension of the vector, *i.e.*, a reviewer, and substitutes its ID with another one from the given search space. Practically, the mutation replaces one reviewer with randomly chosen one from the pool of available reviewers in the project. Similarly to crossover conditions, the chosen reviewer should not belong to the current reviewers in the solution to avoid redundancy.

4. Empirical evaluation

In this section, we report our experimental study to evaluate the efficiency of WhoReview.

4.1. Research questions

Our experimental study aims at addressing the three main research questions.

- **RQ1. (SBSE validation).** How effective is our WhoReview approach in identifying appropriate reviewers?
- **RQ2. (State-of-the-art comparison).** How does WhoReview compared to state-of-the-art code reviewers recommendation approaches?
- **RQ3. (Industrial validation).** How do software developers evaluate our WhoReview in practice?

4.2. Studied projects

We conducted our empirical study on an existing benchmark[4] of four large and long-lived OSS projects that were actively studied in the recent code review research literature [10,13,15,16,21,28,48], Android [25], OpenStack [23], Qt [26], and LibreOffice [24]. Android is a popular and free software stack for mobile devices developed by Google. OpenStack is a cloud computing software platform for controlling large pools of computing power, storage, and networking resources throughout a data-center, and mostly deployed as an infrastructure-as-a-service (IaaS). Qt is a widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms. Table 4 presents the statistics of the four systems in terms of number of reviewed code changes, number of reviewers, and number of files. All collected reviews are closed, *i.e.*, having a status either "*Merged*" or "*Abandoned*", and contain at least one file.

[4] https://kin-y.github.io/miningReviewRepo.

Table 4
Studied systems statistics.

System	#Reviews	#Reviewers	#Files	Av. files per review
Android	5126	94	26,840	8.26
OpenStack	6586	82	16,953	6.04
Qt	23,810	202	78,401	10.64
LibreOffice	6523	63	35,273	11.14

4.3. Evaluation method and metrics

The ultimate goal of **RQ1** is to first evaluate the formulation of our approach from an SBSE perspective by following Harman and Jones [49] guidelines. We compare the performance of IBEA with random search (RS) [50] to assess whether there is a need for an intelligent search method to explore the search space of possible sets of reviewers among the available reviewers. Indeed, RS is the simplest form of search algorithms, which is known as direct-search or derivative-free search. RS is unguided and often fails to find globally optimal solutions as it does not take advantage of the use of genetic operators to evolve the current population [49,51–54].

Moreover, we use common performance metrics to compare the performance of IBEA against three widely-used multi-objective evolutionary search algorithms (MOEA), including the non-dominated sorting genetic algorithm (NSGA-II) [55], the Strength Pareto Evolutionary Algorithm (SPEA2) [56], and the multi-objective evolutionary algorithm based on decomposition (MOEA/D) [57]. Our performance analysis is based on widely-used metrics in MOEAs as defined by Zitzler et al. [58]. These performance measures assess basically three different aspects (*i*) the quality of the obtained Pareto fronts, (*ii*) the convergence towards the Pareto front, and (*iii*) the diversity of the identified optimal solutions. In particular, we study the four following performance measures:

- *Hypervolume (HV)* [59]: it calculates the hypervolume of the multi-dimensional objective space enclosed by a given front and a fixed reference point r, usually the anti-optimal point in space. HV evaluates how good is the approximate front in achieving the optimization objectives. It captures both convergence and diversity. Higher HV values are desirable.
- *Generational Distance (GD)* [60]: it calculates how far is the approximation set S from the reference set RS, which represents a measure of error. GD is the mean value of the Euclidean distance, in an n-dimensional space, between each point in S and its nearest neighboring point in RS for a given n objective space. $GD = 0$ indicates that all the elements generated are in the Pareto-optimal set, *i.e.*, the lower the value of the GD, the better.
- *Spacing (SP)* [61]: it measures how well the non-dominated solutions are distributed along the approximation front, as the variance in the distance of the neighbors vectors in the non-dominated vectors. The higher SP, the better.
- *Contribution (IC)* [62]: It measures the percentage of non-dominated solutions that are in the Reference Front (RF). While IC depends on the number of obtained non-dominated solutions, it penalizes an algorithm if it generates 'few but excellent' solutions. The higher IC, the better.

After evaluating our approach in terms of MOEAs performance, it is important to evaluate its efficiency in solving the problem at hand, *i.e.*, peer code reviewers recommendation. We employ three widely-used performance evaluation metrics, precision, recall, and mean reciprocal rank (MRR) that are common in the evaluation of code review and software engineering recommendation systems [10,13,16,27,63].

For each code change c in the studied projects, let us consider the following possible outcomes, True Positive (TP) represents the top-k recommended reviewers by the tool that correctly belong to the list of actual reviewers; False Positive (FP) represents the number of top-k recommended reviewers that do not belong the list of actual reviewers; False Negative (FN) is the number of actual reviewers, that do not belong to the list of top-k actual reviewers. Based on these outcomes, precision, recall and MRR are computed as follows:

$$precision@k = \frac{TP}{TP + FP} \qquad recall@k = \frac{TP}{TP + FN} \tag{9}$$

We calculate both precision and recall measures with different k values, $k = \{1, 3, 5, 10\}$. Moreover, we evaluate our approach in terms of Mean Reciprocal Rank (MRR). Given a code change c, the reciprocal rank corresponds to the multiplicative inverse of the rank of the first true positive, *i.e.*, correct, reviewer recommended in a ranked list produced by a code reviewer recommendation technique. Then, MRR corresponds to the average of the reciprocal ranks of a set of recommendations for a given code change. Let R a reviewers recommendation list, then MRR is computed as follows:

$$MMR = \frac{\sum_{\forall r \in R} rank(r)}{|R|} \tag{10}$$

where $rank(r)$ refers to the rank of the first correct peer reviewer in the recommendation list. Higher MRR values indicate better performance of a peer reviewers recommendation approach.

For the evaluation procedure, we consider the test review T_r as the most recent 1000 reviews, for each project, in their chronological order. Then, for each T_{ri}, we consider the list of actual reviewers as the ground truth. Then we calculate our performance measures, precision, recall and MRR.

To answer **RQ2**, we measure the efficiency of WhoReview as compared with recent state-of-the-art techniques, RevRec [10], cHRev [16], RevFinder [13], and ReviewBot [27]. Our comparison is based on the precision@k, recall@k and MRR as defined by Eqs. (9) and (10), respectively. RevRec [10] uses a mono-objective search based approach to identify reviewers that are the most experienced and have previous collaborations. cHRev [16] uses reviewers expertise from previous reviews based on the review comments frequency and recency. RevFinder [13] used an expertise model based on similar file paths of previous reviews. ReviewBot [27] constructs an expertise model using static analysis based on the source code change history.

To answer **RQ3**, we study the usefulness of our approach from developers perspective with our industrial partner, a large company producing digital document products, services and printers. We evaluate WhoReview during a period of two weeks, *i.e.*, ten working days on two large software systems developed by our industrial partner using MCR. We denote both projects as *Project-1* and *Project-2* in this paper for confidentiality reasons. During the study period, for each of the submitted code changes, in their chronological order, we generate the top-1 reviewer to be invited to perform the review using four reviewers recommendation techniques, WhoReview, RevRec, cHRev, and a baseline round-robin approach. The round-robin approach is based on the reviewer who received the least recent review request, and alternates between all senior team members regardless of the number of their current outstanding reviews. In total, we evaluated 390 code change requests, distributed as follows, 184 and 208 code change requests, for Project-1 and Project-2, respectively. Thus, each approach was evaluated 46 times in Project-1 and 52 times for Project-2. To get the reviewers opinion on the review requests suggestions, they were asked to provide their willingness

Table 5
The achieved results by each of the MOEAs being compared over 31 independent runs in terms of HV, GD, SP, IC, Precision, Recall, and MRR.

	HV		GD		SP		IC		Precision@3		Recall@3		MRR	
	Median	*p*-value (d)	Median	*p*-value (d)	Median	*p*-value (d)	Median	*p*-value (d)	Median	*p*-value (d)	Median	*p*-value (d)	Median	*p*-value (d)
IBEA	**0.89**	–	**0.04**	–	**0.12**	–	**0.23**	–	**0.52**	–	**0.61**	–	**0.69**	–
NSGA-II	0.85	<0.05 (M)	**0.04**	No. Diff	0.08	<0.05 (S)	0.19	<0.05 (M)	0.51	No. Diff	0.6	<0.05 (S)	0.67	<0.05 (M)
SPEA2	0.72	<0.05 (L)	0.13	<0.05 (L)	0.05	<0.05 (L)	0.11	<0.05 (L)	0.49	<0.05 (M)	0.57	<0.05 (L)	0.65	<0.05 (L)
MOEA/D	0.71	<0.05 (L)	0.15	<0.05 (L)	0.05	<0.05 (L)	0.12	<0.05 (L)	0.51	No. Diff	0.58	<0.05 (M)	0.65	<0.05 (L)
RS	0.46	<0.05 (L)	0.32	<0.05 (L)	0.02	<0.05 (L)	0.03	<0.05 (L)	0.21	<0.05 (L)	0.29	<0.05 (L)	0.19	<0.05 (L)

The columns *p*-value (d) report the statistical difference (*p*-value) based on the Mann–Whitney test and effect-size (*d*) between IBEA and the algorithm in the current row.
The effect-size (*d*) is N : Negligible — S : Small — M : Medium — L : Large.

to accept or decline the request following using a 5 point Likert scale [64] ranging between "*Strongly Disagree*" (1) to "*Strongly Agree*" (5). To avoid potential biases in the experiment, the individual reviewers are not aware of the particular experiment objectives nor the specific approaches being compared.

4.4. Statistical test methods used

Due to the stochastic nature of the MOEAs being evaluated, we compare the performance of our algorithms by running each of them 31 independent runs for each code change, for each project. The obtained results are then statistically analyzed using the Mann–Whitney test [65,66], a non-parametric statistical test method, with a 95% confidence level ($\alpha = 0.05$). Moreover, we adjusted the p-values using Bonferroni correction [67,68]. Moreover, we assess the effect size to evaluate the algorithms difference magnitude using the non-parametric effect Cliff's delta (d) [69]. The effect size is statistically considered *negligible* if $|d| < 0.147$, *small* if $0.147 \leq |d| < 0.33$, *medium* if $0.33 \leq |d| < 0.474$, or *high* if $|d| \geq 0.474$.

4.5. Results and discussions

Results for RQ1. (SBSE Validation). Table 5 reports the results for RQ1. The statistical analysis of the achieved results by IBEA compared to NSGA-II, SPEA2, MOEA/D, and RS show a compelling superiority of IBEA in terms of the four considered MOEA performance metrics. The best median hypervolume (HV) score is 0.89 achieved by IBEA compared to NSGA-II which is the second best algorithm with medium and large effect sizes. In terms of the generational distance (GD), both IBEA and NSGA-II achieved the best performance, 0.04, with no statistical differences. Both algorithms outperform SPEA2, MOEA/D and RS with large effect size. In terms of spacing (SP), we observe that IBEA achieves the highest performance with a median score of 0.23 with statistical different large effect size compared to the competing algorithms (except for NSGA-II, where the effect size is small). Similar superior results were also achieved by IBEA in terms of contribution (IC) with medium and large effect size.

To get a more qualitative sense of the obtained results, we show the Pareto front of each algorithm in Fig. 5 with a randomly picked review from the OpenStack project. We observe that IBEA tends to evolve more near-optimal solutions in the middle region of the identified Pareto front with a good spread of solutions along the front, pushing it outwards towards the ideal point (*i.e.*, high expertise score and low workload score). We also observe that NSGA-II has less non-dominated solutions in the middle of the Pareto front. However, for both extremes of the Pareto front we observe that both IBEA and NSGA-II reach similar regions of the search space. On the other hand, we observe that both SPEA2 and MOEA/D achieve less interesting solutions in their Pareto fronts. For the peer reviewers recommendation problem, near-optimal solutions within the extreme edges of the Pareto front are typically less desirable than solutions in the middle region. Indeed, solutions in the middle region provide the optimal compromise between both objective functions (expertise and workload). Whereas, extreme edge region solutions represent reviewers recommendations with either high workload and high expertise or low workload and low expertise.

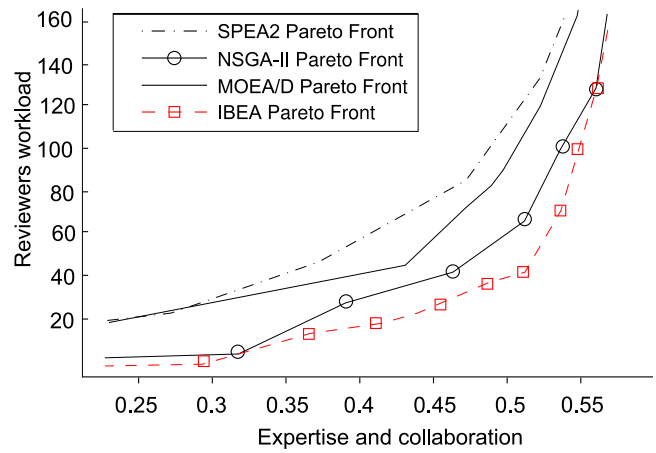

Fig. 5. The Pareto fronts obtained by WhoReview with the different MOEAs.

In terms of precision, recall and MRR, we observe that the results are inlined with the MOEA performance metrics. As reported in Table 5, IBEA achieves the highest results in terms of median scores for the three metrics precision, recall and MRR with small to large effect sizes, except for the precision score compared to NSGA-II and MOEA/D for which the results are not statistically different. We also observe that random search turns out to be the worst algorithm in all the studied metrics. These findings confirm the suitability of our formulation for the peer reviewer recommendation problem.

Results for RQ2. (State-of-the-art comparison). Table 6 reports the average precision and recall results achieved by each of the studied approaches, WhoReview, RevRec, cHRev, RevFinder and ReviewBot. Clearly, WhoReview achieves the best precision among the studies approaches in the four systems. In particular, for the top-1, it reaches the highest precision score at 0.68. We observe relatively stable results across the four projects with a precision@1 ranging from 0.58 to 0.68, for Qt and Android, respectively. Similar results were achieved in terms of recall where WhoReview provides a higher superiority in the top-10 recommended reviewers with a score ranging from 0.68 to 0.77 for LibreOffice and OpenStack, respectively. Upon close inspection of the studied projects, we observe a considerable difference in terms of the projects team sizes and the total number of reviewers available, *e.g.*, 202 reviewers are available for Qt, while only 62 are available for LibreOffice (31%). This finding indicates the good scalability of our approach with respect to the total number of available reviewers in a project.

Table 6
The average precision and recall scores achieved by WhoReview, RevRec, cHRev, RevFinder, and ReviewBot.

Project	k	Precision@k					Recall@k				
		WhoReview	RevRec	cHRev	RevFinder	ReviewBot	WhoReview	RevRec	cHRev	RevFinder	ReviewBot
Android	1	**0.68**	0.58	0.5	0.34	0.21	0.45	0.38	0.27	0.18	0.11
	3	0.62	0.47	0.35	0.25	0.17	0.52	0.51	0.5	0.39	0.19
	5	0.53	0.39	0.3	0.22	0.12	0.65	0.61	0.61	0.48	0.29
	10	0.48	0.34	0.26	0.18	0.09	**0.73**	0.71	0.65	0.54	0.38
OpenStack	1	**0.62**	0.59	0.48	0.32	0.24	0.46	0.41	0.31	0.15	0.12
	3	0.59	0.52	0.42	0.27	0.2	0.59	0.54	0.39	0.29	0.2
	5	0.54	0.43	0.38	0.25	0.16	0.62	0.61	0.52	0.37	0.32
	10	0.48	0.36	0.31	0.21	0.11	**0.77**	0.74	0.66	0.46	0.39
Qt	1	**0.58**	0.49	0.45	0.3	0.22	0.45	0.41	0.33	0.14	0.09
	3	0.55	0.45	0.4	0.27	0.19	0.58	0.5	0.47	0.27	0.16
	5	0.49	0.41	0.37	0.21	0.13	0.63	0.59	0.52	0.35	0.24
	10	0.43	0.34	0.34	0.16	0.09	**0.7**	0.65	0.6	0.43	0.3
LibreOffice	1	**0.61**	0.52	0.5	0.48	0.38	0.48	0.34	0.32	0.32	0.18
	2	0.56	0.45	0.42	0.4	0.36	0.55	0.48	0.42	0.38	0.22
	5	0.53	0.41	0.4	0.32	0.32	0.61	0.57	0.48	0.42	0.31
	10	0.46	0.39	0.35	0.3	0.23	**0.68**	0.59	0.51	0.49	0.38

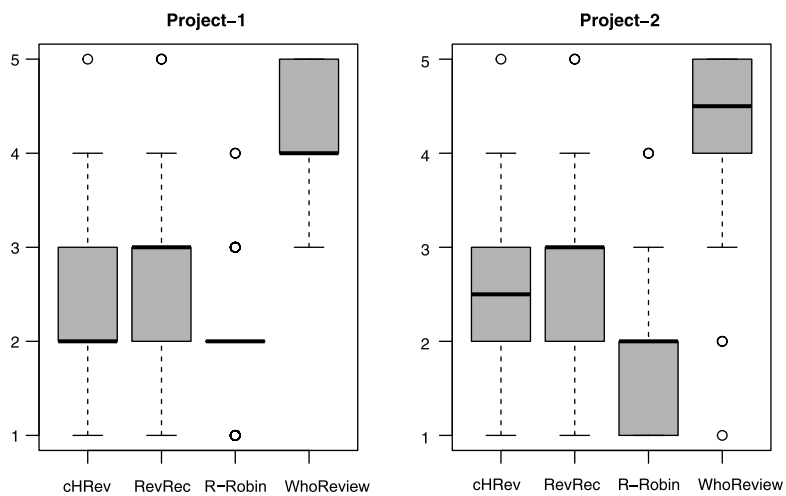

Fig. 6. Boxplots of the developers evaluation of WhoReview, RevRec, cHRev and round-robin.

While our search-based approach has shown better performance than state-of-the-art by combining both workload and expertise, the highest precision and recall scores can reach 0.68 and 0.77, respectively. Indeed, one of the limitations of the evaluation of the reviewer recommendation systems, it that it is assumed that the people who performed the review are the best people to do so, and those who were not invited to review are not appropriate as pointed out by prior studies [10,11,18,70]. Moreover, we observe from Table 7, that WhoReview achieves the highest MRR score ranging from 0.61 to 0.72 for the four studied projects. On average, WhoReview achieves an MRR score of 0.66, outperforming the state-of-the-art approaches, RevRec, cHRev, RevFinder, and ReviewBot achieving an average of 0.60, 0.52, 0.47 and 0.21, respectively. These results indicate that our approach can identify appropriate reviewers in the first few ranks. To better evaluate our approach, it is crucial to assess its applicability with developers to assess whether a recommended reviewer would accept or decline the review request in real-world context. This motivates us to deploy WhoReview in an industrial setting and evaluate it from developers view (RQ3).

Results for RQ3. (Industrial validation). Fig. 6 reports the developers evaluation recorded for our approach, WhoReview, compared to RevRec, cHRev, and round robin for a set of 390 code review requests across both projects with our industrial partner. In total, both projects involve 134 contributors who are globally distributed and adopt MCR. We observe from Fig. 6 a clear superiority of the average score achieved by WhoReview across both projects. WhoReview achieved an average of 4.3 and 4.2, for Project-1 and Project-2, respectively. The second best approach is RevRec with an average of 2.8 and 2.9. Whereas, cHRev achieved 2.5 and 2.6, and round-robin achieved 2 and 1.9. The statistical analysis using the Mann–Whitney test indicates that WhoReview is statistically different from RevRec, cHRev and round robin (see Fig. 6).

This feedback confirms the effectiveness of WhoReview by identifying several more valid reviewers recommendations than the expertise-based approaches (RevRec, and cHRev) and the baseline round-robin method that iterates between reviewers regardless their expertise or workload. Upon a closer discussion with some developers after completing the experimental study, most of the reviewers who accepted the review requests indicate that they feel more comfortable with the code reviews being assigned to them as they come with a good matching with their current workload and expertise. Reviewers also indicate that a round-robin approach is not appropriate since not all code reviews require the same effort and time to be properly completed as some code change require several iterations with the code owner before it is merged with the code base. We also found

Table 7
The achieved MRR scores by each approach.

	WhoReview	RevRec	cHRev	RevFinder	ReviewBot
Android	**0.72**	0.69	0.65	0.60	0.25
OpenStack	**0.69**	0.63	0.58	0.55	0.30
Qt	**0.63**	0.54	0.43	0.31	0.22
LibreOffice	**0.61**	0.52	0.45	0.40	0.07
Average	**0.66**	0.60	0.52	0.47	0.21

some cases in which, code change owners manually added other reviewers. We thus excluded all these cases from our study.

5. Threats to validity

This section discusses the potential threats to validity that may affect our study.

5.1. Threats to construct validity

Threats to construct validity could be related to the performance measures. We mainly used standard performance metrics such as precision, recall and MRR that are widely used in recommendation systems and peer code review literature [10,13,15,16]. Another potential threat could be related to the selection of MOEA techniques. Although we use the IBEA, NSGA-II, SPEA2, and MOEA/D which are widely used and have shown high performance software engineering [52,54,71–74], there are other techniques. As part of our future work, we plan to conduct a larger comparative study with other search techniques. Moreover, unlike voluntary developers in OSS projects, industrial developers decision to accept or decline a code review invitation could be influenced by contractual obligations/commitments. To mitigate this concern, we alternated between WhoReview and three baseline techniques.

5.2. Threats to internal validity

Threats to internal validity relate to experiments bias. While we used a longitudinal data benchmark setup to evaluate WhoReview, it is assumed in this benchmark that reviewers who are involved in the code review were the right ones to review the change and those who did not participate in the review were not appropriate reviewers. To mitigate this issue, we evaluated our approach in an industrial context with the original developers from two different projects and teams.

5.3. Threats to external validity

Threats to external validity relate to the generalizability of our results. We have analyzed a total of 157,467 code reviews from OSS projects, and studied 2 industrial projects with different team sizes and programming languages. In the future, we plan to further apply our approach to more projects from both industrial and OSS worlds.

Other potential threats to validity of our work concern some of the measurements we used to estimate the developer's expertise and workload. For instance, the $size(c_i)$ may under- or overestimate the workload and could be better improved to consider the complexity and other quality metrics to better estimate the required effort instead of simple count of the number of files. Similarly, the Jaccard similarity score used to identify similar files may have some limitations. While it is one of the fast and efficient similarity measures used in prior works [10,46,75] that requires relatively low computational resources given that we are already using an evolutionary search which is known by its high computational cost, this measure may not be the most accurate measurement. To mitigate such issues, we plan on extending our experiments by performing a comparative study between state-of-the art similarity models [76,77].

6. Conclusion and future work

We introduced in this paper, WhoReview, a multi-objective search-based approach to find and recommend relevant peer reviewers for MCR. We adopt IBEA as a multi-objective search method to find the optimal set of reviewers that provide the best trade-off between two conflicting objectives, (1) the reviewers expertise and collaboration and (2) the reviewers workload. To evaluate our approach, we conducted an empirical evaluation on a benchmark of four large scale and long-lived software projects. Results show our approach outperforms state-of-the-art approach by an average precision of 0.68 and recall of 0.77. We also conducted an industrial qualitative evaluation with WhoReview to assess the relevance of our approach from developers perspective on two large projects from our industrial partner. The obtained results show the effectiveness of our approach in identifying relevant reviewers in practice.

As future work, we plan to evaluate our approach in a larger set of industrial and OSS projects. We want to extend the expertise, collaboration and workload models to consider additional social and technical aspects to improve the recommendation accuracy. We also would like to extend WhoReview with an interactive component to record and learn from the decisions of the invited reviewers to better personalize future recommendations.

CRediT authorship contribution statement

Moataz Chouchen: Conceptualization, Formal analysis, Methodology, Software, Validation, Investigation, Writing - original draft. **Ali Ouni:** Conceptualization, Methodology, Validation, Supervision, Resources, Writing - review & editing, Funding acquisition, Project administration. **Mohamed Wiem Mkaouer:** Methodology, Validation, Writing - review & editing. **Raula Gaikovina Kula:** Data curation, Validation, Writing - review & editing. **Katsuro Inoue:** Methodology, Validation, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grant RGPIN-2018-05960, and Japanese Society for the Promotion of Science (JSPS) KAKENHI Grant Numbers 18H04094, 20K19774 and 20H05706.

References

[1] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: 35th International Conference on Software Engineering (ICSE), Piscataway, NJ, USA, 2013, pp. 712–721.
[2] M. Beller, A. Bacchelli, A. Zaidman, E. Juergens, Modern code reviews in open-source projects: Which problems do they fix?, in: 11th Working Conference on Mining Software Repositories (MSR), in: MSR 2014, 2014, pp. 202–211.
[3] A. Bosu, J.C. Carver, Impact of peer code review on peer impression formation: A survey, in: International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 133–142.

[4] A. Bosu, J.C. Carver, C. Bird, J. Orbeck, C. Chockley, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, IEEE Trans. Softw. Eng. 43 (1) (2016) 56–75.
[5] M. Fagan, Design and code inspections to reduce errors in program development, in: Software Pioneers, Springer, 2002, pp. 575–607.
[6] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, A. Bacchelli, Does reviewer recommendation help developers?, IEEE Trans. Softw. Eng. (2018).
[7] Available at : https://www.gerritcodereview.com/ (Gerrit code review).
[8] Available at : https://www.reviewboard.org (ReviewBoard).
[9] Available at : https://www.phacility.com/phabricator (Phabricator).
[10] A. Ouni, R.G. Kula, K. Inoue, Search-based peer reviewers recommendation in modern code review, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 367–377.
[11] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, B. Ashok, Whodo: automating reviewer suggestions at scale, in: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 937–945.
[12] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M.W. Godfrey, Investigating code review quality: Do people and participation matter?, in: International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 111–120.
[13] T. Patanamon, T. Chakkrit, G.K. Raula, Y. Norihiro, I. Hajimu, M. Kenichi, Who should review my code? A file location-based code-reviewer recommendation approach for modern code review, in: 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015.
[14] GitHub Enterprise Cloud, Code reviewers assignment, 2020, https://help.github.com/en/github/setting-up-and-managing-organizations-and-teams/managing-code-review-assignment-for-your-team, accessed: 2020-01-19.
[15] X. Xia, D. Lo, X. Wang, X. Yang, Who should review this change?: Putting text and file location analyses together for more accurate recommendations, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 261–270.
[16] M.B. Zanjani, H. Kagdi, C. Bird, Automatically recommending peer reviewers in modern code review, IEEE Trans. Softw. Eng. 42 (6) (2015) 530–543.
[17] M. Fejzer, P. Przymus, K. Stencel, Profile based recommendation of code reviewers, J. Intell. Inf. Syst. 50 (3) (2018) 597–619.
[18] C. Hannebauer, M. Patalas, S. Stünkel, V. Gruhn, Automatically recommending code reviewers based on their expertise: An empirical comparison, in: IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 99–110.
[19] P.C. Rigby, M.-A. Storey, Understanding broadcast based peer review on open source software projects, in: 33rd International Conference on Software Engineering (ICSE), New York, New York, USA, 2011, p. 541.
[20] A. Bosu, J.C. Carver, How do social interaction networks influence peer impressions formation? A case study, in: Open Source Software: Mobile Open Source Technologies - 10th International Conference on Open Source Systems (OSS), 2014, pp. 31–40.
[21] M. Chouchen, A. Ouni, M.W. Mkaouer, R.G. Kula, K. Inoue, Recommending peer reviewers in modern code review: a multi-objective search-based approach, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO), 2020, pp. 307–308.
[22] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: Parallel Problem Solving from Nature-PPSN VIII, Springer, 2004, pp. 832–842.
[23] Available at : http://www.openstack.org/ (OpenStack).
[24] Available at : https://www.libreoffice.org/ (LibreOffice).
[25] Available at : https://source.android.com/ (Android).
[26] Available at : http://qt-project.org/ (Qt).
[27] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: International Conference on Software Engineering (ICSE), 2013, pp. 931–940.
[28] X. Yang, R. Gaikovina Kula, N. Yoshida, H. Iida, Mining the modern code review repositories: A dataset of people, process and product, in: 13th Working Conference on Mining Software Repositories (MSR), 2016.
[29] Available at : https://docs.openstack.org/infra/manual/developers.html (OpenStack MCR manual).
[30] J. Kim, E. Lee, Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation, Symmetry 10 (4) (2018) 114.
[31] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug?, in: 28th International Conference on Software Engineering (ICSE), 2006, pp. 361–370.
[32] X. Xia, D. Lo, X. Wang, B. Zhou, Accurate developer recommendation for bug resolution, in: 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 72–81.
[33] S. Ruangwan, K. Matsumoto, P. Thongtanunam, A. Ihara, The impact of human factors on the participation decision of reviewers in modern code review, Empir. Softw. Eng. (2019).
[34] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The influence of non-technical factors on code review, in: 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 122–131.
[35] O. Baysal, R. Holmes, A Qualitative Study of Mozillas Process Management Practices, Tech. Rep. CS-2012-10, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, 2012.
[36] A. Bosu, J.C. Carver, Impact of developer reputation on code review outcomes in oss projects: An empirical investigation, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2014, pp. 33:1–33:10.
[37] T. Hirao, S. McIntosh, A. Ihara, K. Matsumoto, The review linkage graph for code review analytics: a recovery approach and empirical study, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 578–589.
[38] F. Ebert, F. Castor, N. Novielli, A. Serebrenik, Confusion in code reviews: Reasons, impacts, and coping strategies, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 49–60.
[39] P. Thongtanunam, A.E. Hassan, Review dynamics and their impact on software quality, in: IEEE Transaction on Software Engineering (TSE), 2020.
[40] J. Cohen, E. Brown, B. DuRette, S. Teleki, Best Kept Secrets of Peer Code Review, Smart Bear, 2006.
[41] M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Syst. J. 15 (3) (1976) 182–211.
[42] P.C. Rigby, D.M. German, M.-A. Storey, Open source software peer review practices: a case study of the apache server, in: 30th International Conference on Software Engineering, 2008, pp. 541–550.
[43] C. Bird, D. Pattison, R. D'Souza, V. Filkov, P. Devanbu, Latent social structure in open source projects, in: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: SIGSOFT '08/FSE-16, 2008, pp. 24–35.
[44] X. Yang, N. Yoshida, R.G. Kula, H. Iida, Peer review social network (person) in open source projects, IEICE Trans. Inf. Syst. 99-D (3) (2016) 661–670.
[45] A. Ouni, M. Kessentini, H. Sahraoui, M.S. Hamdi, Search-based refactoring: Towards semantics preservation, in: 28th International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 347–356.
[46] S. Niwattanakul, J. Singthongchai, E. Naenudorn, S. Wanapu, Using of jaccard coefficient for keywords similarity, in: International Multiconference of Engineers and Computer Scientists, Vol. 1, 2013, pp. 380–384.
[47] X. Yang, N. Yoshida, R.G. Kula, H. Iida, Peer review social network (person) in open source projects, IEICE Trans. Inf. Syst. 99 (3) (2016) 661–670.
[48] S. McIntosh, Y. Kamei, B. Adams, 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 (MSR), 2014, pp. 192–201.
[49] M. Harman, B.F. Jones, Search-based software engineering, Inf. Softw. Technol. 43 (14) (2001) 833–839.
[50] D.C. Karnopp, Random search techniques for optimization problems, Automatica 1 (2) (1963) 111–121.
[51] M. Harman, The current state and future of search based software engineering, in: Future of Software Engineering (FOSE), 2007, pp. 342–357.
[52] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Comput. Surv. 45 (1) (2012) 11.
[53] A. Ouni, Search-based software engineering: Challenges, opportunities and recent applications, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2020, pp. 1114–1146.
[54] I. Saidani, A. Ouni, M. Chouchen, M.W. Mkaouer, Predicting continuous integration build failures using evolutionary search, Inf. Softw. Technol. 128 (2020) 106392.
[55] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[56] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the Strength Pareto Evolutionary Algorithm, TIK-report 103, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001.
[57] Q. Zhang, H. Li, Moea/D: a multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731.
[58] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. Da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132.
[59] D. Brockhoff, T. Friedrich, F. Neumann, Analyzing hypervolume indicator based algorithms, in: International Conference on Parallel Problem Solving from Nature, Springer, 2008, pp. 651–660.
[60] C.A.C. Coello, N.C. Cortés, Solving multiobjective optimization problems using an artificial immune system, Genet. Program. Evol. Mach. 6 (2) (2005) 163–190.

[61] J.R. Schott, Fault Tolerant Design using Single and Multicriteria Genetic Algorithm Optimization, Tech. rep., Air force inst of tech Wright-Patterson afb OH, 1995.
[62] H. Meunier, E.-G. Talbi, P. Reininger, A multiobjective genetic algorithm for radio network optimization, in: Proceedings of the 2000 Congress on Evolutionary Computation, Vol. 1, 2000, pp. 317–324.
[63] I. Avazpour, T. Pitakrat, L. Grunske, J. Grundy, Dimensions and metrics for evaluating recommendation systems, in: Recommendation Systems in Software Engineering, Springer Berlin Heidelberg, 2014, pp. 245–273.
[64] R. Likert, A technique for the measurement of attitudes., Arch. Psychol. (1932).
[65] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1–10.
[66] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: Search Based Software Engineering, Springer, 2011, pp. 33–47.
[67] Y. Hochberg, A sharper Bonferroni procedure for multiple tests of significance, Biometrika 75 (4) (1988) 800–802.
[68] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Softw. Test. Verif. Reliab. 24 (3) (2014) 219–250.
[69] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions., Psychol. Bull. 114 (3) (1993) 494.
[70] E. Doğan, E. Tüzün, K.A. Tecimer, H.A. Güvenir, Investigating the validity of ground truth in code reviewer recommendation studies, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019, pp. 1–6.
[71] N. Almarimi, A. Ouni, S. Bouktif, M.W. Mkaouer, R.G. Kula, M.A. Saied, Web service api recommendation for automated mashup creation using multi-objective evolutionary search, Appl. Soft Comput. 85 (2019) 105830.
[72] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (3) (2016) 1–53.
[73] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using nsga-iii, ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3) (2015) 1–45.
[74] N. Almarimi, A. Ouni, M. Chouchen, I. Saidani, M.W. Mkaouer, On the detection of community smells using genetic programming-based ensemble classifier chain, in: 15th International Conference on Global Software Engineering (ICGSE), 2020, pp. 43–54.
[75] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.
[76] D. Kartchner, T. Christensen, J. Humpherys, S. Wade, Code2vec: Embedding and clustering medical diagnosis data, in: 2017 IEEE International Conference on Healthcare Informatics (ICHI), IEEE, 2017, pp. 386–390.
[77] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning distributed representations of code, Proc. ACM Program. Lang. 3 (POPL) (2019) 1–29.