软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006079]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

响应时间约束的代码评审人推荐方法*

胡渊喆[1,3], 王俊杰[1,2,3], 李守斌[1,3], 胡军[1], 王 青[1,2,3]

[1](中国科学院 软件研究所 互联网软件技术实验室,北京 100190)

[2](计算机科学国家重点实验室(中国科学院 软件研究所),北京 100190)

[3](中国科学院大学,北京 100049)

通讯作者: 王青, E-mail: wq@itechs.iscas.ac.cn

摘 要: 同行代码评审,即对提交代码进行人工评审,是减少软件缺陷和提高软件质量的有效手段,已被 Github 等开源社区以及很多软件开发组织广泛采用.在 GitHub 社区,代码评审是其 pull-based 软件开发模型的重要组成部分.开源项目往往存在成百上千个候选评审人员,为评审工作推荐合适的评审人员是一项很有价值且挑战性的工作.我们基于真实开源项目的数据分析发现,评审响应时间过长是普遍存在的问题,这会延长评审周期、降低参与人员积极性,而已有的代码评审人推荐工作均没有考虑响应时间这个因素.因此,本文提出响应时间约束的代码评审人推荐问题,即推荐的评审人能否在约定时间内进行评审;进而提出了基于多目标优化的代码评审人推荐方法(MOC2R),该方法通过最大化代码评审人经验、最大化在约定时间内的响应概率、最大化人员最近时间内的活跃性三个目标,使用多目标优化算法来推荐代码评审人员.我们基于 6 个开源项目的数据进行实验,结果表明,在不同时间窗约束下(2 小时,4 小时,8 小时), Top-1 准确率为 41.7%-61.5%,Top-5 准确率为 66.5%-77.7%,显著优于两条常用且业内领先的基线方法,且三个目标均对人员推荐有贡献,其中约定时间内的响应概率目标对于人员推荐的贡献最大.该方法能够进一步提升代码评审效率,提高开源社区的活跃性.

关键词: 代码评审; 响应时间约束; 多目标优化

中文引用格式: 胡渊喆,王俊杰,李守斌,胡军,王青.响应时间约束的代码评审人推荐方法.软件学报,2020. http://www.jos.org.cn/1000-9825/6079.htm

英文引用格式: Hu YZ, Wang JJ, Li SB, Hu J, Wang Q. Response Time constrained Code Reviewer Recommandation. Ruan Jian Xue Bao/Journal of Software, 2020 (in Chinese). http://www.jos.org.cn/1000-9825/6079.htm

Response Time constrained Code Reviewer Recommandation

HU Yuan-Zhe[1,3], WANG Jun-Jie[1], LI Shou-Bin[1,3], HU Jun[1], WANG Qing[1,2,3]

1(Laboratory for Internet Software Technologies, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

2(State Key Laboratory of Computer Sciences (Institute of Software, The Chinese Academy of Sciences), Beijing 100190, China)

3(University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Peer code review, or manual review of submitted code, which is an effective way to reduce defects and improve quality, has been widely adopted by open source communities and many software development organizations, such as Github. In the GitHub community, code reviews are an important part of its pull-based software development model. Open source projects often have hundreds or thousands of candidate reviewers, recommend suitable reviewers for code reivew is a very valuable and challenging work. Based on the data analysis of real open source projects, we found that the response time of review is a common problem, which will extend the review cycle and reduce the enthusiasm of participants. Existed work did not take the response time into account. Therefore, we proposed the code reviewer recommendation problem with response time constraint, and then proposed the code reviewer recommendation method

* 基金项目: 国家重点研发计划(2018YFB1403400)
收稿时间: 2020-02-05; 修改时间: 2020-03-23; 采用时间: 2020-05-07; jos 在线出版时间: 2020-05-26

(MOC2R) based on multi-objective optimization by maximizing the experience of code reviewers, maximizing the response probability within the time window, and maximizing the activity of staff within the latest time. We conducted experiments based on data from six open source projects, and the results showed that under different time window constraints (2h,4h,8h), Top-1 accuracy rate is 41.7%-61.5%,Top-5 accuracy rate is 66.5%-77.7%, significantly better than the two commonly used and industry-leading baseline methods, and all three objectives contributed to the recommendation among which the response probability within the time window contributes the most. The method can further enhance code review efficiency, improve the activity of the open source community.

Key words: code review; response time constrained; multi-objective optimization

1. 引言

同行代码评审,即对提交代码进行人工评审,已被证明是减少软件缺陷和提高软件质量的有效手段.近年来,现代代码评审(Modern Code Review)[1],一种轻量级的同行代码评审实践,已经被业界各公司(如微软、谷歌)和各大开源社区广泛采用.例如,GitHub 提供了一个在线评审流程：项目的贡献者（contributor）可以通过添加新特性或修复某些 bug 来对源代码进行一定程度的更改,但是更改不能直接提交给项目基线库,而是以合并请求（pull request）的形式提交给社区.每一个合并请求的创建就代表需要进行一次代码评审（code review）,每个贡献者都可以是潜在的评审人员,可以在代码风格、新特性的必要性、代码逻辑的正确性等方面对该合并请求留下评论（comment）.最后,项目集成人员在综合考虑所有评审人员的意见后,做出是否接受合并的决定,以确保来自贡献者的代码质量.该评审流程是 GitHub 社区 pull-based 软件开发模型的重要组成部分,确保了来自贡献者的代码质量[2,3].

对于一些受欢迎的 GitHub 项目,贡献者数量常常会达到千人以上规模,并因此提交了大量的合并请求,然而大多数合并请求不能及时得到评审[3].根据 Gousios 等人的调查结果,15%的贡献者抱怨合并请求没有在他们预期时间内被处理[4].进一步,Thongtanunam 等人的研究表明,没有找到合适代码评审人的合并请求比其他合并请求需要多 12 天才能被批准[5].此外,在一个项目中有成百上千个候选评审人员,这更加剧了代码评审人推荐问题的难度.因此,为了提高代码评审的质量和效率,迫切需要有效的评审人员推荐方法.

近年来,代码评审人推荐的研究备受关注.Balachandrany、Thongtanunam、 Hannebauer 等设计启发式规则,基于历史提交和评审信息抽取得到的评审人修改经验和评审经验来进行人员推荐[5,6,7];Jeoung、Lipcak 等通过抽取代码修改内容等提取特征,使用机器学习和信息检索方法来推荐评审人员[9,10];Yu 等基于开发者在代码提交和评论中的交互,通过社会网络方法进行人员推荐[11];Xia 等混合以上两种或多种方法,进一步优化评审人推荐的效果[15,16,17].以上研究均以能否找到合适的评审人为目标,忽略了评审响应时间这个重要因素.评审响应时间是指代码评审创建与评审人来参加评审（表现为在评审中留下评论）的时间间隔.

由于开源项目的社会自治性,实际的项目开发实践中,评审人的响应时间过长是制约项目进展和代码提交者积极性的重要因素[3,4,18].当一次提交经过相当长的时间后依然没有收到任何回复信息,该提交的贡献者会有更大的可能性脱离该项目贡献者团队.我们基于 6 个开源项目的数据统计发现,平均 15.3%-63.2%的合并请求在 72 小时内没有响应,平均 24%-68.4%的合并请求在 24 小时内没有响应.已有的仅以寻找合适评审人为目标的代码评审人推荐方法[5,6,7,9,10,11,15,16,17]并不能解决响应时间的问题.

为此,本文提出响应时间约束的代码评审人推荐问题,即推荐的评审人能否在约定时间内进行评审,目标是推荐合适的代码评审人在更短的时间内完成代码评审,从而进一步提升代码评审的效能,促进开源社区参与人员的积极性.“合适的代码评审人”是沿用已有研究[5,9,11,15]的说法,仅指该评审人是否进行代码评审,不考虑评审的响应时间.

进一步,我们提出了基于多目标优化的代码评审人推荐算法（MOC2R）.该方法通过最大化代码评审人经验、最大化在约定时间内的响应概率、最大化人员最近时间内的活跃性三个目标,使用多目标优化算法来推荐代码评审人员.我们基于 6 个开源项目的数据进行实验验证,结果表明,在不同时间窗约束下（2 小时,4 小时,8 小时）,Top-1 准确率为 41.7%-61.5%,Top-3 准确率为 58.4%-74.9%,Top-5 准确率为 66.5%-77.7%,MRR 为

0.53-0.58.相比基线方法,平均 Top-1 准确性提高了 81%-143%,平均 MRR 提升了 56%-97%.此外,我们还实验评估了每个目标对于评审人推荐的贡献,发现三个目标缺一不可,其中约定时间内的响应概率目标对于人员推荐的贡献最大.

本文贡献在于:

1）我们基于真实开源项目数据分析了当前代码评审响应时间的现状,提出响应时间约束的代码评审人推荐问题,这是代码评审人推荐研究中第一个考虑响应时间的工作.

2）我们提出了基于多目标优化的代码评审人推荐算法, 可以最大化代码评审人经验、最大化在约定时间内的响应概率、最大化人员最近时间内的活跃性.

3）我们在 GitHub 的六个大型项目评估了方法的效果,并得到了很好结果.

本文结构组织如下:第二节进行了问题分析.第三节提出了基于多目标优化的代码评审人推荐算法.第四和第五节给出了相关实验的设计方法和实验结果分析,并在第六节做了进一步讨论.第七节回顾了代码评审人推荐的相关研究工作,最后在第八节进行总结.

2. 问题分析

本章基于真实的开源项目数据,分析当前代码评审响应时间的现状、以及代码评审的人员属性.

2.1 合并请求响应时间分析

我们基于 GitHub 六个流行的开源项目（详见 4.3 章节）,对历史合并请求的响应时间进行了统计,结果如下表所示:

Table 1 Pull Request Response Time

表 1 合并请求响应时间

Project	响应时间大于 2h	响应时间大于 4h	响应时间大于 8h	响应时间大于 16h	响应时间大于 24h	响应时间大于 72h
Fackbook/react	58.9%	51.7%	44.2%	35.4%	29.8%	20.2%
tensorflow/tensorflow	79.8%	72.8%	63.8%	52.7%	46.3%	32.1%
twbs/bootstrap	78.9%	68.5%	56.5%	44.4%	40.0%	26.8%
ohmyzsh/ohmyzsh	89.6%	85.9%	81.5%	75.8%	68.4%	63.2%
flutter/flutter	48.0%	42.3%	37.0%	30.4%	24.0%	15.3%
electron/electron	77.8%	69.8%	58.3%	44.0%	36.9%	20.7%

六个项目中,平均有 15.3%-63.2%的合并请求在 72 小时内没有响应,24%-68.4%的合并请求在 24 小时内没有响应.其中,flutter/flutter 是合并请求响应时间最快的项目,有 52%的合并请求在 2 小时内得到响应.而 ohmyzsh/ohmyzsh 是合并请求响应时间最慢的项目,超过 60%的合并请求在 24 小时内不能得到回复.进一步的,我们统计了 flutter/flutter 和 ohmyzsh/ohmyzsh 两个项目在 2015 年至 2019 年中每年 3 月 1 日的文件个数（NOF）和代码行数（LOC）,如表 2 所示.flutter/flutter 代码规模增长速度明显领先于 ohmyzsh/ohmyzsh.其在 2018 年 3 月至 2019 年 3 月代码规模增长了近一倍（33 万行代码增长至近 60 万行代码）,而 ohmyzsh/ohmyzsh 在 5 年时间内代码规模增长不足一倍（1.7 万行增长至 2.9 万行）.

Table 2 Code size in flutter and ohmyzsh

表 2 flutter 与 ohmyzsh 代码规模

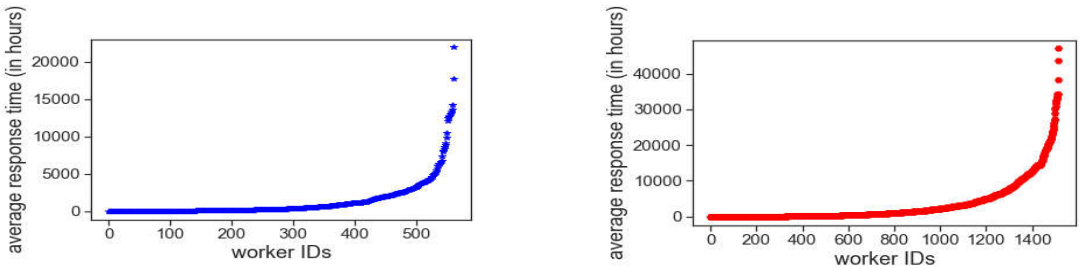
Project	2015.3		2016.3		2017.3		2018.3		2019.3	
	NOF	LOC	NOF	LOC	NOF	LOC	NOF	LOC	NOF	LOC
flutter/flutter	30	5508	598	93921	1105	197776	1572	331626	2235	599059
ohmyzsh/ohmyzsh	244	17518	283	21676	356	25881	381	26591	476	29566

根据谷歌工程实践文档,代码评审最长需要在一个工作日进行反馈[25]. 以上数据和分析说明,在开源社区的代码评审实践中,存在很多的合并请求不能及时得到响应的情况,这种情况存在的原因有很多,例如由于每日提交的合并请求数量过多,导致部分请求难以得到及时处理[8],像 flutter/flutter 项目平均每周接收到 85 个合并请求;再如由于不知道谁是合适的代码评审人,使得合并请求停留在评审人分派阶段,没有在预期时间内处理[5],

或者即使已经分派给某个代码评审人后,由于不合适的分派导致的评审时间过长,例如评审人经验不匹配、或评审人没有足够时间 [1,4].这延长了评审周期、降低了参与人员的积极性和项目的发展速度. 因此,本文提出响应时间约束的代码评审人推荐问题,即推荐的评审人能否在约定时间内进行评审,目标是为代码评审推荐合适且响应速度快的评审人,加速评审过程,提升社区活跃性.

2.2 人员响应时间分析

图 1 给出 flutter/flutter、ohmyzsh/ohmyzsh 两个项目的代码评审人的响应时间分布,其他项目的趋势和这两个项目相同,篇幅限制不再列出.



(a) flutter/flutter (b) ohmyzsh/ohmyzsh

Figure 1 Code reviewer average response time

图 1 代码评审人的平均响应时间

从上图看出,项目中不同代码评审人的平均响应时间存在很大差异.例如在 flutter/flutter 项目中,响应时间从 0.02 小时到 912.7 天;有 6.7%的评审人的平均响应时间小于 2 小时,有 10.9%的评审人的平均响应时间大于 3 个月.ohmyzsh/ohmyzsh 项目中,83.7%的评审人的平均响应时间大约 1 天.

已有研究[1,4]指出,快速响应是 pull-based 开发模型中支持贡献的关键组成部分,快速的响应能够很好的鼓励开源项目贡献者,有效的减少新贡献者加入项目的障碍,相反,代码贡献者往往都会对迟迟无法得到评审人响应而担忧,甚至影响他们后续对项目的参与和贡献.对于那些响应很慢的评审人,即便他们有经验、有能力,他们的评审结果对于待评审的代码,也是远水不解近渴,不能及时有效地帮助项目管理者了解代码质量,或者帮助开发人员改进质量.所以,需要将响应时间纳入评审人推荐问题中,考虑评审人在历史代码评审中的响应时间,以便推荐出响应速度快的代码评审人.

2.3 人员活跃性分析

我们取 2018 年 1 月-2019 年 1 月的评审数据,按照每半月为一个区间（作为横坐标的 24 个时间区间）,统计该区间上某代码评审人是否有评审活动,有评审活动记为 1,没有评审活动记为 0（作为纵坐标）.图 2 给出 flutter/flutter、ohmyzsh/ohmyzsh 两个项目上随机选取的 6 个评审人员的活动情况,其他项目的趋势和这两个项目相同,篇幅限制不再列出.

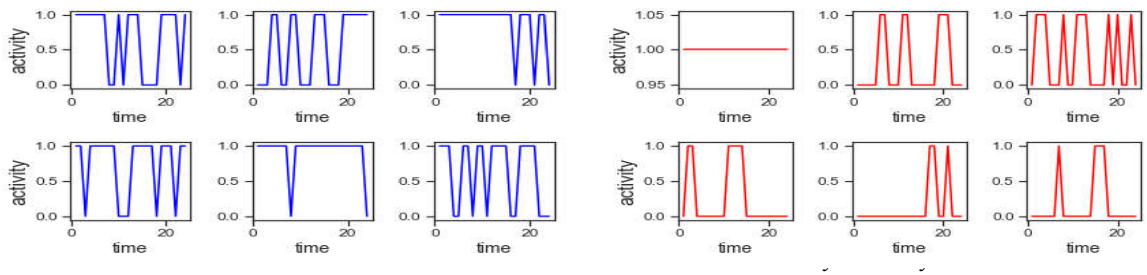


Figure 2 Code reviewer activity distribution

图 2 代码评审人的评审活动分布

从上图可以看出,绝大多数代码评审人并不是一直活跃的,他们会有很多不活跃的时间.如果在这些不活跃

的时间推荐给他们评审任务,则他们进行评审的响应时间会非常长、或者说响应的概率很小.因此,考虑评审人在最近时间内的活跃情况,有助于找到合适且响应速度快的代码评审人.

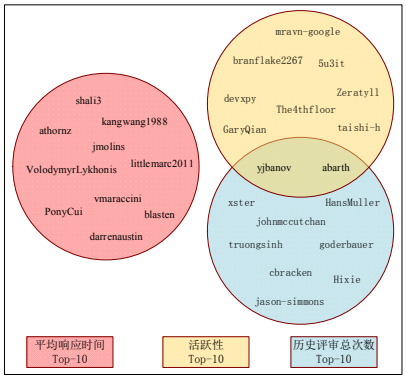
2.4 多特征对比分析

在上面两节分析中,我们发现,在进行评审人推荐时,需要考虑人员响应时间和活跃性两个特征;同时,已有研究指出人员经验会影响评审人推荐的效果.本节进一步分析基于各自特征得到的代码评审人是否存在重合,从而启发我们的方法设计.

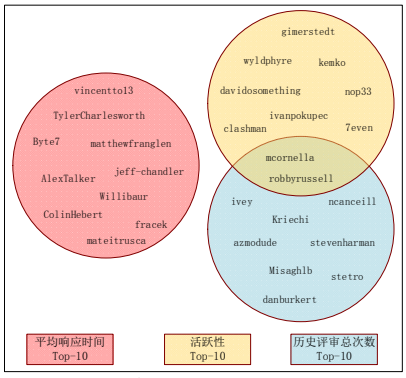
我们取 flutter/flutter 和 ohmyzsh/ohmyzsh 两个项目中全部数据,分别基于平均响应时间、活跃性(即最近一次参加评审的时间)和历史评审总次数(表征人员经验)三个特征进行统计,并得到基于各自特征的评审人排序.图 3 给出了不同特征下前 10 个评审人的交叠情况.

从图中可以看出,基于各自特征推荐的代码评审人基本不存在重叠.具体来说,平均响应时间最快的 10 个评审人与其他两个特征的前 10 名均未有交叠.在活跃性最高和评审总次数最多的统计结果中,flutter/flutter 和 onmyzsh/ohmyzsh 仅有两个评审人交叠.

这说明,三个特征需要同时进行考虑,才能推荐出既合适又能在更短时间内完成评审的代码评审人,因此我们将采用多目标优化方法进行推荐.



(a) flutter/flutter



(b) ohmyzsh/ohmyzsh

Figure 3 Overlap of three attributes

图 3 基于各自特征的人员交叠图

3. MOC2R:基于多目标优化的代码评审人推荐

前面的分析和实验表明,代码评审人在历史代码评审中的响应速度、最近时间内的活跃性均会影响他们是否及时进行代码评审,历史响应速度表征评审人的工作习惯,响应速度快的评审人未来也可能快速完成评审;活跃性表征评审人参加待评审任务的可能性,近期活跃的评审人更有可能快速完成评审.已有研究已经指出评审人对于待评审任务的经验会影响代码推荐效果[5,6,7].因此,这三个方面需要同时考虑,以便找到合适且响应速度快的代码评审人.

本文提出的基于多目标优化的代码评审人推荐方法 MOC2R(Multi-Objective Code Reviewer Recomendation),以最大化代码评审人的经验、最大化代码评审人在约定时间内的响应概率、最大化所推荐人员最近时间内的活跃性为优化目标,将代码评审人推荐问题建模成多目标优化问题,建立基于多目标优化的推荐模型.MOC2R 方法框架如图 4 所示.

3.1 节定义了代码评审相关的基本元素.由于多目标优化方法最重要的是优化目标的度量,3.2 节给出了优化目标的度量方法,3.3 节对 MOC2R 中如何使用多目标优化进行了介绍.

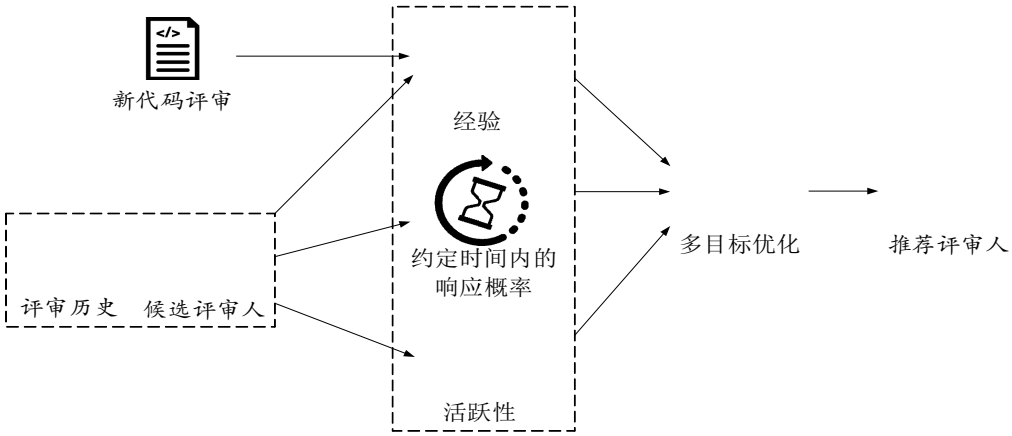


Figure 4 MOC2R Overview

图 4 MOC2R 方法框架

3.1 基本元素和定义

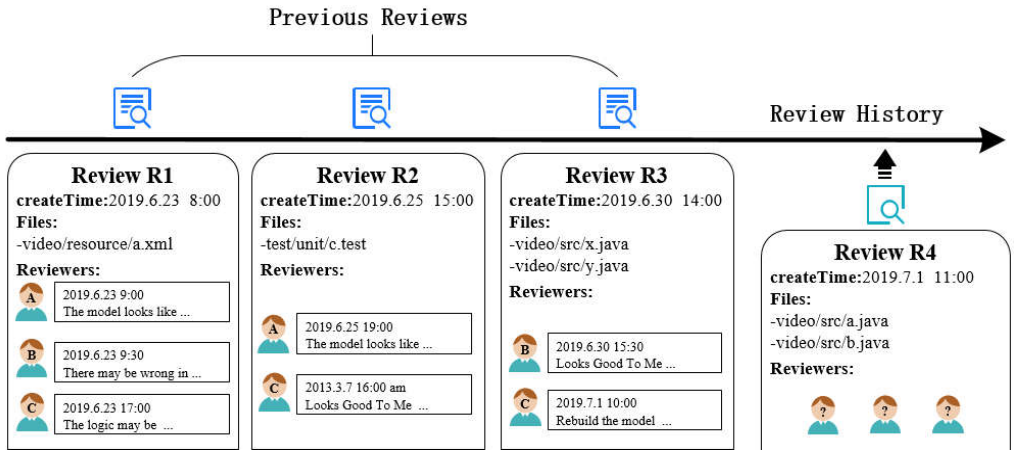


Figure 5 Diagram for code review

图 5 代码评审示意图

图 5 给出评审历史、候选评审人、新代码评审的示意图.

在 GitHub 的开源项目中,贡献者（contributor）每一次提交代码都需要创建一个合并请求（pull request）,每一个合并请求的创建就代表需要进行一次代码评审（code review）.任何一个在合并请求中添加了评论信息（comment）的贡献者,我们都视为是这次代码评审的评审人员（reviewer）.

代码评审历史是指过去已经完成的一系列代码评审的合集,表示为$R=\{r_1,r_2,\dots,r_n\}$,其中 r_i 表示已经完成的第 i 次代码评审.

对于一次已完成的**代码评审** r,包括其创建时间、一组被评审的变更文件、以及所有参加过这次代码评审的评审人,表示为$r=\{createTime(r),Files(r),Reviewers(r)\}$,其中 $createTime(r)$表示代码评审 r 的创建时间,$Files(r)=\{f_1,f_2,\dots f_n\}$表示被评审的变更文件列表, $Reviewers(r)=\{(w_1,t_1),(w_2,t_2),\dots(w_n,t_n)\}$表示参加评审的评审人列表,$w_i$ 表示代码评审人,t_i 表示该评审人参加本次代码评审并留下第一条评论的时间.某个代码评审人可能会在一个代码评审中有多次评论,本文只记录第一条评论的时间用于目标的建模.对于后续评论的发生时间对于建模和推荐效果的影响,将在后续工作中研究.

候选评审人是开源项目中所有的贡献者,表示为 $W=\{w_1, w_2, \dots w_n\}$.

一次**新代码评审**包括创建时间和一组待评审的变更文件,表示为 $nr=\{createTime(nr), Files(nr)\}$.

我们定义代码评审人推荐问题为：给定一个新代码评审 *nr*、一组候选人员 *W*、历史代码评审集合 *R* 以及响应时间窗 φ（即在 φ 时间内需要进行响应）,推荐目标是从候选人员 *W* 中选择出一个包含 *k* 个人员的子集 *S*,使得选择出来的集合 *S* 中的人员能够最大化代码评审人经验、最大化在约定时间内的响应概率、最大化人员最近时间内的活跃性.

3.2 优化目标度量

MOC2R 定义了如下三个优化目标：

i）目标一：代码评审人的经验

我们基于文件路径相似算法[5]设计代码评审人相关经验的度量方法.它的基本思想是相似路径中的文件具有相似的功能,如果代码评审人在过去的评审历史中积累了评审相似代码的经验,他们更有可能是待评审任务的合适评审人.

给定两个文件 f_i 和 f_j,其相似度定义为两个文件路径的相似程度,计算公式如下所示：

$$Similarity(f_i, f_j) = \frac{Pat\square Comparison(f_i, f_j)}{Max\{Lengt\square(f_i), Lengt\square(f_j)\}} \quad (1)$$

其中,$Length(f_i)$表示文件 f_i 目录层级个数, $PathComparison(f_i, f_j)$是两个文件相同的目录层级个数.具体来说,首先将每个文件按照目录层级拆分成两个序列,然后计算两个序列的最长公共子序列.例如 f_i 的文件路径为 *src/layout/android/settings/utils.java*, f_j 的文件路径为 *src/com/android/settings/location/utils.java*, $Length(f_i)$的返回结果为 *5*,$Length(f_j)$的返回结果为 *6*.$PathComparison(f_i, f_j)$返回结果为 *4*（即 *src*、*android*、*settings*、*utils.java*）.

基于文件相似性,对于新评审 *nr*,其和历史评审 r_i 的相似度定义为两个评审包含文件集合（*Files(nr)*和 $Files(r_i)$）之间的相似性,表示为：

$$ReviewSimilarity(r_i, nr) = \frac{\sum_{f_i \in Files(r_i), nf_j \in Files(nr)} Similarity(f_i, nf_j)}{|Files(r_i)| * |Files(nr)|} \quad (2)$$

两次代码评审相似度越高,表示 r_i 中的评审人员对新评审 *nr* 中的变更文件越有经验.

对于一个评审候选人 *w*,其相对于待评审任务 *nr* 的经验 *ExpertiseScore*, 定义为其历史评审中与待评审任务相似度的累积值,计算如公式（3）：

$$ExpertiseScore(w, nr) = \sum_{r_i \in R_w} ReviewSimilarity(r_i, nr) \quad (3)$$

其中,R_w 表示候选人 *w* 所完成的评审任务的集合.

ii）目标二：代码评审人在约定时间内的响应概率

MOC2R 方法中,基于代码评审人在历史评审中的响应速度来度量其在约定时间内进行响应的概率.我们假设响应时间窗 φ（即约定的响应时间）,代码评审人 *w* 在 φ 时间内响应的概率即为评审人 *w* 参加过的所有代码评审 R_w 中,其响应时间在 φ 内的评审次数占全部评审任务的百分比.计算公式如下所示：

$$ResopnseScore = \frac{\sum_{r_i \in R_w} response@\varphi}{|R_w|} \quad (4)$$

$$response@\varphi = \begin{cases} 1, \ if \ FastestResponseTime（w, ri） - createTime(r_i) < \varphi \\ 0, \ if \ FastestResponseTime（w, ri） - createTime(r_i) \geq \varphi \end{cases} \quad (5)$$

其中在代码评审人 *w* 最快响应时间小于响应时间窗 φ 时,$response@\varphi$ 返回 *1*.否则,返回 *0*.$FastestResponseTime（w, r_i）$则计算了代码评审人 *w* 在 r_i 中的首次响应时间,即 *w* 留下第一条评论的时间与 *ri* 创建时间的差值. $createTime（ri）$是代码评审 *ri* 的创建时间.

iii）目标三：代码评审人最近时间内的活跃性

活跃性度量候选人在最近时间内的活跃程度,即在最近时间 t_{thres} 内,候选人参与过的代码评审的累计次数.

累计次数越高,代表该候选人在最近时间 t_{thres} 内的表现越活跃.活跃性得分计算公式如下所示：

$$ActiveScore = \sum_{r_i \in R_w} active@t_{t\_res} \quad (6)$$

$$active@t_{t\_res} = \begin{cases} 1, & if\ Now - t_{t\_res} \le ti < Now \\ 0, & if\ ti < Now - t_{t\_res} \end{cases} \quad (7)$$

其中 *Now* 表示当前时间,$active@t_{thres}$ 表示该候选人在代码评审 r_i 中添加评论的时间 *ti* 是否在时间范围 t_{thres} 内,时间范围 t_{thres} 是方法的输入参数.

3.3 多目标优化方法

前面提到,MOC2R 方法需要优化三个目标.很显然,同时在所有目标上达到最优是很困难的.MOC2R 方法希望获得帕累托边界（Pareto front）[31]上的一组解（solution）,这组解能够支配其他的解,也就是说比其他解优越.

MOC2R 使用 NSGA-II 算法（Non-dominated Sorting Genetic Algorithm-II）[32]优化前面提到的三个目标.NSGA-II 是在软件工程和其他领域均被广泛使用的多目标优化方法.根据 Harman 等人的研究[30],有超过65%的软件分析相关的优化技术是基于遗传算法（对于单目标优化）和 NSGA-II（对于多目标优化）.

在我们的评审人推荐场景中,帕累托边界表示 NSGA-II 得到的在三个目标之间的最佳权衡.管理人员可以通过检查帕累托边界发现最合适的权衡解.

MOC2R 的多目标优化方法包括以下四个步骤：

1）表示

像其他选择问题一样[19,22,24],我们将每个候选评审人编码为一个二进制变量.如果评审人被选中,值为 1;否则,值为 0.MOC2R 得到的解是一个二进制变量的向量,向量的长度是所有的候选评审人.评审人推荐问题的解空间是所有候选评审人是否被选中的组合.

2）初始化.

MOC2R 随机产生初始种群,也就是说在解空间中随机选择 K 个解,根据 Harman[30]的建议,我们将 K 设置为200.

3）遗传算子.

对于二进制编码的解的演化,我们使用标准遗传算子[20].我们使用单点交叉、按位变异的方式产生下一代.我们使用二进制比赛方式进行选择操作,也就是说随机选择两个解进行交叉和变异,然后选择适应性最强的两个解进入下一代.

4）适应性函数.

由于我们希望优化三个目标,每个候选解都用 3.2 章节描述的三个目标函数进行衡量.对于每个目标,我们首先得到每个解中评审人的目标值,然后将该解中所有选中人员的目标值求和.

4. 实验设计

4.1 研究问题

我们通过以下三个问题,从方法的性能、各个目标的作用、参数的影响三个方面验证方法的效果.

RQ1：MOC2R 方法在评审人推荐方面的效果如何？

RQ2：MOC2R 方法的三个优化目标对于评审人推荐的贡献是怎样的？

RQ3：MOC2R 在不同参数下的敏感度如何？

本方法主要希望推荐可以快速响应的合适的评审人,一个工作日内反馈对开发者和项目管理者都是非常有益的[25],所以对于方法中的响应时间窗 φ,我们分别取 2 小时、4 小时、8 小时进行实验,并给出对比结果和分析.

4.2 Ground Truth和基线方法

响应时间约束的代码评审人推荐的 Ground Truth 是基于历史代码评审活动中评审人的参与记录得到的.具体来说,我们首先获取测试集中每个代码评审实际参与的评审人,并计算他们留下第一条评论的时间与该代码评审创建时间之间的间隔,此时间间隔即为该评审人在本次评审中的响应时间.

本次实验中,我们选择了 ReviewFinder[5]和 IR+CN[11]两种代码评审人推荐算法作为基线.这两个算法均是业内领先的代码评审人推荐算法,并被很多研究所引用.

ReviewFinder 的主要思想是假设大多数大型项目中,目录结构组织良好,具有相似文件路径的文件实现的功能是相似的或紧密关联的.该方法通过计算文件相似度获得两次代码评审的相似度得分,对于一个新的代码评审任务,计算每个历史评审的相似度得分并将得分分配给参与过相应历史评审的候选人,候选人中按累积分数高低推荐给新的代码评审任务.

IR+CN 是一个结合了信息检索和社交网络的混合推荐方法,一方面该方法从历史评论记录中提取每一位候选评审人的兴趣信息,对于一个新的代码评审,与这个代码评审发起者具有相同兴趣的评审人被认为是合适的代码评审人;同时结合信息检索算法,对两个合并请求的相似度进行比较,得到最终的推荐结果.

4.3 实验数据采集和预处理

为了采集实验数据,我们在 GitHub 中综合考虑 Star 个数和合并请求个数选择了六个开源项目,包括 facebook/react, tensorflow/tensorflow, twbs/bootstrap, ohmyzsh/ohmyzsh,flutter/flutter 和 electron/electron.这六个开源项目已关闭的合并请求个数均在 5000 以上,平均每个项目获得了 5K+的 Watch,114.35K+的 Star 和 35.7K+的 Fork.其中 Watch 表示关注,只要项目发生变动,关注的人都会收到通知消息;Star 表示点赞,表示对项目的支持和喜欢;Fork 表示创建项目代码库的分支,并拷贝到自己账号中.三个指标从不同角度反映了项目的受欢迎程度,值越大说明项目越受欢迎.此外,这些开源项目中的每一个都得到了广泛的应用,并在各自的技术领域发挥着重要的作用.我们通过 GitHub REST API v3（https://developer.github.com/v3/）抓取这六个开源项目的相关数据.采集完成后数据信息如表 3 所示.

Table 3 Data Set

表 3 数据集

项目名称	合并请求	贡献者	开始时间	结束时间	文件个数	主要语言	Watch	Star	Fork
Fackbook/react	8361	1348	2013/5/29	2019/8/1	1646	JavaScript	6.3K	141K	27K
tensorflow/tensorflow	11815	2343	2015/11/9	2019/8/1	19116	C++	8.2K	139K	79K
twbs/bootstrap	9603	1103	2011/8/19	2019/9/16	457	JavaScript	7K	138K	67K
ohmyzsh/ohmyzsh	5106	1476	2010/8/31	2019/8/1	814	Zsh	2.5K	101K	18K
flutter/flutter	14300	501	2015/11/10	2019/9/16	3812	Dart	2.3K	82K	11K
electron/electron	8618	907	2013/6/19	2019/9/16	1762	C++	2.6K	80K	10K

在本实验中,对收集到的合并请求进行了如下过滤:

a）去掉了少于两个评审人员参与的合并请求.因为当一次代码评审过程中评审人数大于 2 次时,这次代码评审才是可信的[26,27].

b）按基线算法 IR+CN[11]的数据要求,对合并请求中标题和描述信息进行了去除停留词和词干还原,并将标题和描述信息中单词总数少于 5 个词的合并请求去掉.

和已有评审推荐方法[5,16]的实验设置类似,我们取各项目中最新的 1000 个合并请求作为测试集,其他合并请求作为训练集.

对于方法中的活跃性时间范围参数 t_{thres},我们在训练集上采用十折交叉验证的方式,分别评估 t_{thres} 为 1 天,3 天,7 天,15 天,30 天的推荐效果,结果发现 t_{thres} 为 3 天时,各个项目均能取得较好的效果,因此时间范围参数 t_{thres}

设置为 3 天.

4.4 评价指标

为了评估我们的方法,我们使用*时间窗约束的 Top-k 准确率*和*时间窗约束的平均倒数排名*（Mean Reciprocal Rank,MRR）两个度量指标.这两个度量指标经常被应用于软件工程领域推荐系统的评估工作中[6,29,30].

1）时间窗约束的 Top-k 准确率计算的是方法推荐的 Top-k 个评审人,至少一人在约定时间内出现在实际评审人列表中的合并请求的个数和合并请求总个数的百分比.给出一系列能够在响应时间阈值 φ 内响应的合并请求(PR@φ), 时间窗约束的 Top-k 准确率可以根据公式 8 进行计算.

$$\text{时间窗约束的 Top-k 准确率} = \frac{\sum_{pr \in PR@\varphi} isCorrect(pr@\varphi, Top-k)}{|PR@\varphi|} \times 100\% \quad (8)$$

$$isCorrect(pr@\varphi, Top-k) = \begin{cases} 1, & \text{if 推荐k个评审人中至少一人在约定时间}\varphi\text{内进行响应} \\ 0, & \text{if 推荐k个评审人中没有人在约定时间}\varphi\text{内进行响应} \end{cases} \quad (9)$$

其中 isCorrect(pr@φ,Top-k)函数在至少有一个 Top-k 中的评审人员参与了该合并请求的代码评审且其最快响应时间小于 φ 的情况下返回 1,否则返回 0;

2）时间窗约束的 MRR 计算的是代码评审人推荐列表中正确推荐的评审人倒数排名的平均值.给出一系列能够在响应时间窗 φ 内响应的合并请求(PR@φ),MRR 可以根据如下公式进行计算.

$$MRR = \frac{1}{|PR@\varphi|} \sum_{pr \in PR@\varphi} \frac{1}{rank(Reviewers(pr))} \quad (10)$$

其中,rank(Reviewers(pr))返回推荐列表中第一个实际参与了该合并请求的评审人的排名.当推荐列表中的评审人没有任何一个参与了该合并请求的实际评审时,返回 0.理想情况下,一个方法能够完美推荐代码评审人时,MRR 值为 1.

5. 结果分析

5.1 RQ1. 方法性能

表 4 和表 5 分别列出 MOC2R 以及基线方法在 6 个实验项目上在不同时间窗约束下的 Top-k 准确率和 MRR 结果.

Table 4 Result of Top-k accuracy in different time window

表 4 时间窗约束的 Top-k 准确率结果

Project	Approch	时间窗约束=2h			时间窗约束=4h			时间窗约束=8h		
		Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Fackbook/react	ReviewFinder	31.0%	47.8%	55.8%	35.6%	49.5%	59.8%	44.6%	54.6%	60.6%
	IR+CN	34.9%	55.4%	63.6%	43.1%	54.7%	67.5%	53.9%	61.7%	68.3%
	MOC2R	**61.5%**	**74.9%**	**77.7%**	**63.7%**	**72.3%**	**76.5%**	**64.2%**	**71.4%**	**73.8%**
tensorflow/tensorflow	ReviewFinder	22.1%	37.4%	45.6%	20.6%	34.6%	47.9%	38.3%	60.3%	65.8%
	IR+CN	32.0%	48.3%	56.1%	33.5%	49.0%	58.0%	44.4%	60.8%	**75.9%**
	MOC2R	**52.7%**	**69.2%**	**73.5%**	**55.9%**	**68.1%**	**74.1%**	**58.0%**	**70.9%**	75.1%
twbs/bootstrap	ReviewFinder	14.5%	24.6%	34.0%	19.4%	40.4%	48.2%	24.7%	45.3%	57.6%
	IR+CN	27.3%	44.2%	56.2%	29.6%	45.5%	54.6%	35.0%	52.0%	62.2%
	MOC2R	**53.0%**	**65.0%**	**71.2%**	**46.8%**	**61.2%**	**69.6%**	**50.5%**	**66.2%**	**72.7%**
ohmyzsh/ohmyzsh	ReviewFinder	13.8%	23.7%	39.7%	17.0%	29.2%	39.7%	24.4%	34.7%	43.1%
	IR+CN	17.1%	31.4%	45.2%	23.1%	41.4%	50.6%	29.3%	44.2%	58.7%
	MOC2R	**52.9%**	**68.9%**	**73.8%**	**47.1%**	**62.8%**	**70.2%**	**47.0%**	**66.1%**	**71.1%**
flutter/flutter	ReviewFinder	26.0%	50.9%	55.3%	37.1%	46.5%	60.8%	41.6%	58.4%	**73.3%**
	IR+CN	34.3%	53.0%	59.7%	37.6%	46.2%	58.7%	39.5%	51.5%	64.7%
	MOC2R	**44.0%**	**63.1%**	**66.5%**	**49.4%**	**65.4%**	**70.5%**	**55.2%**	**67.4%**	73.0%
electron/electron	ReviewFinder	18.9%	29.8%	35.3%	20.1%	32.5%	43.3%	28.9%	40.5%	54.8%
	IR+CN	23.7%	34.0%	38.8%	27.1%	39.7%	49.5%	35.7%	53.3%	69.1%
	MOC2R	**41.7%**	**58.4%**	**69.6%**	**46.7%**	**61.7%**	**70.4%**	**52.7%**	**62.5%**	**71.8%**

mean	ReviewFinder	21.0%	35.7%	44.3%	25.0%	38.8%	49.9%	33.7%	49.0%	59.2%
	IR+CN	28.2%	44.4%	53.3%	32.3%	46.1%	56.5%	39.6%	53.9%	66.5%
	MOC2R	**51.0%**	**66.6%**	**72.1%**	**51.6%**	**65.2%**	**71.9%**	**54.6%**	**67.4%**	**72.9%**

Table 5 result of MRR in different time window

表 5 时间窗约束的 MRR 结果

Project	Approch	MRR		
		时间窗约束=2h	时间窗约束=4h	时间窗约束=8h
Fackbook/react	ReviewFinder	0.40	0.46	0.50
	IR+CN	0.47	0.50	0.58
	MOC2R	0.68	0.69	0.68
tensorflow/tensorflow	ReviewFinder	0.31	0.30	0.49
	IR+CN	0.42	0.44	0.57
	MOC2R	0.63	0.61	0.64
twbs/bootstrap	ReviewFinder	0.26	0.30	0.32
	IR+CN	0.39	0.38	0.46
	MOC2R	0.62	0.62	0.59
ohmyzsh/ohmyzsh	ReviewFinder	0.22	0.26	0.31
	IR+CN	0.28	0.34	0.38
	MOC2R	0.61	0.55	0.56
flutter/flutter	ReviewFinder	0.41	0.47	0.51
	IR+CN	0.44	0.46	0.46
	MOC2R	0.57	0.60	0.62
electron/electron	ReviewFinder	0.27	0.29	0.37
	IR+CN	0.32	0.36	0.45
	MOC2R	0.53	0.58	0.59
mean	ReviewFinder	0.31	0.35	0.42
	IR+CN	0.39	0.41	0.48
	MOC2R	0.61	0.61	0.61

由表 4 和表 5 可以看出,在不同的时间窗约束下,MOC2R 都能得到较好的效果： Top-1 准确率为 41.7%-61.5%,Top-3 准确率为 58.4%-74.9%,Top-5 准确率为 66.5%-77.7%,MRR 为 0.53-0.58;平均 Top-1 准确率为 51%,平均 Top-3 准确率为 66.6%,平均 Top-5 准确率为 72.1%,平均 MRR 为 0.61.这说明 MOC2R 可以很好地在多目标优化中推荐在时间约束下（约定时间窗内响应）最合适的评审人.准确率随着时间窗加大而增加的项目,是由于其评审人的响应速度较慢,倾向于在较长的时间中响应;而准确率随着时间窗加大而减少的项目,是由于该项目的评审人的响应速度较快,倾向于在较短的时间内响应.

我们可以看到,相对于 ReviewFinder 和 IR+CN 两种推荐方法,时间窗约束越强的窗口,MOC2R 方法的优势越明显.当时间窗约束为 2 小时时,相比 ReviewFinder 方法,MOC2R 方法的 Top-1 准确率比 ReviewFinder 平均提高了 143%（(51%-21%)/21%）,MOC2R 方法的 Top-5 准确率平均提高了 63%（(72.1%-44.3%)/44.3%）,MRR 平均提高了 97%（(0.61-0.31)/0.31）.相比 IR+CN 方法,MOC2R 方法的 Top-1 准确率平均提高了 81%（(51%-28.2%)/28.2%）,Top-5 准确率平均提高了 35%（(72.1%-53.3%)/53.3%）,MRR 提升了 56%（(0.61-0.39)/0.39）.并且,IR+CN 方法在推荐效果上优于 ReviewFinder 方法,这是因为 IR+CN 在考虑两次代码评审的相似度基础上,还通过社交网络方式考虑了两个评审人的相似度.进一步的,在整体响应时间较慢的项目（例如 ohmyzsh/ohmyzsh）,我们方法的 MRR 指标能够 2 倍以上（(0.61-0.28）/0.28）优于基线算法,这说明,我们方法推荐的第一个正确的评审人的排序位置更靠前.

5.2 RQ2. 各个目标的贡献

为了清晰的比较各个目标对评审人推荐性能的影响,我们对 MOC2R 的三个目标分别进行了实验分析.我们将 EXP（ExpertiseScore）、RSP（ResponseScore）和 ACT（ActivScore）分别以单目标方式和双目标方式进行评审人推荐.对于单目标方式,我们按照该目标的值进行排序;对于双目标方式,我们基于两个目标运行多目

标优化算法.图 6、图 7、图 8 分别列出不同组合方式在不同时间窗约束下的 Top-k 准确率和 MRR.

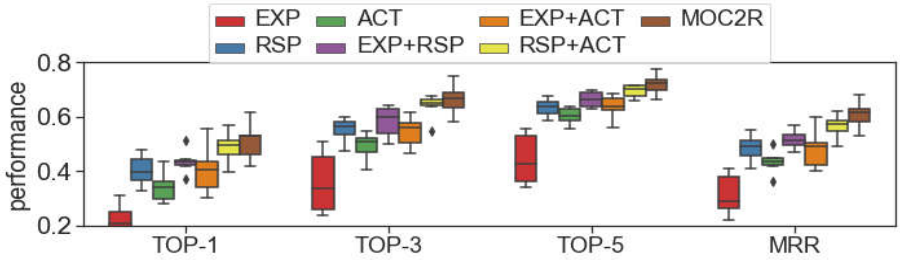


Figure 6 Top-k&MRR in different objective, φ=2h

图 6 时间窗约束为 2 小时,各目标 Top-k 准确率和 MRR

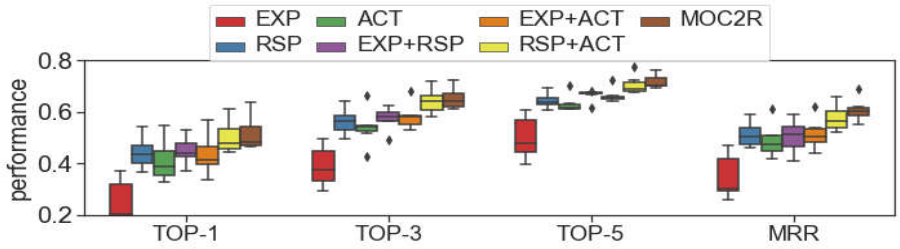


Figure 7 Top-k&MRR in different objective, φ=4h

图 7 时间窗约束为 4 小时,各目标 Top-k 准确率和 MRR

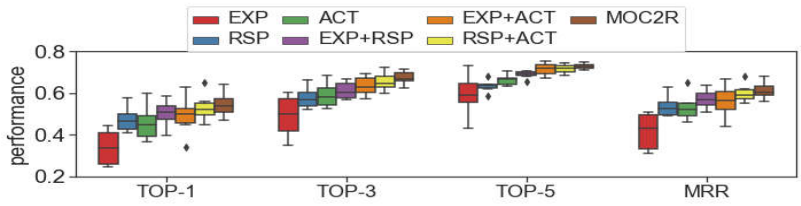


Figure 8 Top-k&MRR in different objective, φ=8h

图 8 时间窗约束为 8 小时,各目标 Top-k 准确率和 MRR

从图中可以看出,在时间窗约束为 2 小时时,单个目标的 Top-1 准确率平均为 21%-40%, MRR 平均为 0.31-0.49;使用两个目标的 Top-1 准确率平均为 44%-49%, MRR 平均为 0.48-0.57,而 MOC2R 方法的平均 Top-1 准确率为 51%,MRR 为 0.61,高于单目标和双目标的情况,这说明,MOC2R 方法中运用的经验、响应概率、活跃性三个目标缺一不可,共同对于评审人推荐发挥作用.

对于采用两个目标的情况,我们看到,去掉约定时间内的响应概率目标,评审人推荐效果最差,这说明该目标对于考虑响应时间的评审人推荐的贡献最大.并且随着时间窗约束的增大,评审人推荐效果下降的幅度逐渐缩小,表明对于评审人推荐的贡献变小.这很好理解,时间窗约束得越严格,越要求评审人的响应时间足够快,而评审人约定时间内的响应概率就是评审人该方面的度量.

对于采用单个目标,我们可以看到,只采用人员经验的评审人推荐效果最低,这意味着仅仅考虑评审人员经

验是不够的,有经验的人员如果不能及时响应评审任务,无疑会对项目和社区代码贡献者产生负面影响.这也说明考虑响应时间的评审人推荐的重要性.基于响应概率或活跃性指标得到的推荐效果优于基于人员经验的推荐效果,这说明在考虑响应时间的评审人推荐问题中,人员过去活动这方面的信息是需要纳入考虑的重要因素.

5.3 RQ3. 方法的敏感性

活跃性指标中的时间范围 t_{thres} 是 MOC2R 方法的参数,为了分析其对于算法的影响,我们对时间范围设定了一系列实验值 t_{thres}={1 天,3 天,7 天,15 天,30 天},并对 MOC2R 在不同参数的性能进行了分析.表 6 列出时间窗约束为 4 小时的 Top-3 准确率,在其他时间窗下的准确率趋势相似.

Table 6 Top-3 accuracy of MOC2R in different t_{thres} (φ=4h)

表 6 MOC2R 时间窗约束为 4 小时,在不同时间范围 t_{thres} 中 Top-3 准确率对比分析

Project	MOC2R Top-3 accuracy@φ=4h				
	t_{thres}=1 天	t_{thres}=3 天	t_{thres}=7 天	t_{thres}=15 天	t_{thres}=30 天
Fackbook/react	73.4%	74%	72.8%	72.3%	69.5%
tensorflow/tensorflow	72.2%	70.7%	68.9%	68.1%	65.3%
twbs/bootstrap	63.5%	64.1%	62.5%	61.2%	59.4%
ohmyzsh/ohmyzsh	59.7%	60.4%	62%	62.8%	65.2%
flutter/flutter	67.1%	68.3%	69.6%	65,4%	63.2%
electron/electron	64%	64.5%	63.7%	61.7%	58.6%

除 ohmyzsh/ohmyzsh 项目外,在 t_{thres} 取值 1 天或 3 天时,MOC2R 的 Top-3 准确率能得到对比实验中的最大值,t_{thres} 取值 7 天、15 天和 30 天时,Top-3 准确率小幅下降,降幅小于 5%.我们认为,一般来说,当 t_{thres} 设置较大值时,能够潜在的将更多可能的人员包含进来,但同时也引入噪声,可能会影响推荐效果;而当 t_{thres} 设置较小值时,能够过滤掉噪声对于推荐效果的影响,得到较好的推荐效果.对于六个项目中贡献者活跃性最小的 ohmyzsh/ohmyzsh,较长时间的活跃性时间范围 t_{thres} 反而能够更加准确的帮助我们推荐合适的代码评审人.

6. 讨论

特定算法的性能因项目而异.如表 3 和表 4 所示,MOC2R 在不同的项目中具有不同的 Top-k 准确度和不同的 MRR 表现,其他算法也是如此.我们仔细的分析了六个开源项目的数据和其在 GitHub 上的组织管理方式,除平均响应时间最慢的 ohmyzsh/ohmyzsh 项目外,其他五个开源项目均配置了 facebook 的机器人 Robot(https://github.com/facebookarchive/mention-bot),项目中所有的合并请求都会在第一时间被 Robot 进行统一处理,分配一个或多个标签进行分类,比如缺陷、改进、新特性、新版本等标签.并且在执行了自动化测试和静态检查的项目中,Robot 还会根据自动化测试与静态检查结果判断是否提醒核心开发人员关注该合并请求状态.在此基础上,我们对 flutter/flutter 项目进行了统计.该项目未使用 Robot 预处理合并请求之前,其合并请求的平均最快响应时间约为 7.5 小时,而在使用了 Robot 之后最近一年的合并请求的平均最快响应时间保持在 4 小时以内.Robot 的使用可以帮助贡献者和核心开发人员更好的理解合并请求,并帮助他们选择要查看的合并请求,缩短了响应时间.但现有的 Robot 背后的机制是基于贡献者提交的变更代码行寻找潜在的评审人,如果 Robot 可以使用本文提出的 MOC2R 方法,自动为合并请求推荐评审人,那么对提升代码评审效率、提高开源社区的活跃性将会起到很好的促进作用.

另一方面,代码评审人推荐准确率越高并不代表推荐的评审人越能够找到更多的问题.针对代码评审质量的判别方法、判断依据的相关研究还比较少.因为我们进行代码评审的直接目的还是希望能够进一步提升软件产品的质量,那么如何评估一次代码评审做的好不好、什么样的代码评审人在评审过程中能够发现更多的问题,也将会成为我们下一个阶段的研究方向.

7. 相关工作

近年来,国内外出现了许多关于代码评审人推荐的研究和算法.我们根据它们考虑的主要特征和用于推荐代码评审人的主要技术分为四组：i)启发式方法,ii)基于机器学习的方法,iii)基于社交网络的方法,iv)混合方法.

7.1 启发式方法

传统的推荐方法分析以往的提交和评审信息,如文件修改历史记录、文件路径历史信息、代码行修改历史、历史评审中评审人评论次数等,并基于评审人的修改经验或评审经验来寻找最相关的代码评审人员.主要的算法有 ReviewBot、RevFinder、WRC 和 CORRECT 等.ReviewBot 是 Balachandrany 于 2013 年提出并实现的一款工具[6],被广泛认为是最早的代码评审人推荐研究之一.它的推荐依据是基于如下假设：在合并请求中更改的代码行应该由先前讨论或修改过相同代码行的代码评审人来评审.Thongtanunam 等人于 2015 年提出 ReviewFinder [5],也是目前被广泛引用的推荐算法.该方法采用的是合并请求中包含的文件路径相似度的计算与排序的方法 FPS（File Path Similarity）.Hannebauer 等人于 2016 年提出 FPS 的优化算法 WRC[7].该方法在 FPS 的基础上增加了前置步骤,通过加权评论数的方式将评审经验引入 FPS 的计算过程中,以便得到更精确的推荐结果.Rahman 等人于 2016 年提出代码评审人推荐算法 CORRECT [8],使用过相似技术或引用过相似的代码库的评审人是合适的候补代码评审人.

7.2 基于机器学习和信息检索的方法

这组方法使用不同的机器学习算法或者信息检索方法来推荐代码评审人员.它们不同于前一组的主要区别在与它们首先需要建立一个基于训练集的模型.Pred.Rev.是 Jeoung 等人于 2009 年提出的[9],基于补丁元数据、补丁内容和 bug 报告信息等一系列特征,利用贝叶斯网络预测代码评审人员和补丁接受度的方法.同样使用朴素贝叶斯算法预测评审人的还有 Lipcak [10]和 Jiang 等人[13,14]. Lu 等人于 2016 年提出基于时间和影响力因子的信息检索评审人推荐算法[12].

7.3 基于社交网络的方法

这组方法中,社交网络被用来确定开发人员之间评审交流的相似性,这就为代码评审人员的推荐提供了更多相似的候选对象.*CN*(Comment Network)方法是 Yu 等人通过分析贡献者和开发者之间的社会关系,提出的一种代码评审人推荐方法[11].其假设开发者的兴趣可以从他们的评论中提取出来,与合并请求的发起者共享共同兴趣的开发人员被认为是合适的代码评审人员.

7.4 混合方法

其他一些研究中,研究人员采用了不同的方法进行混合,来推荐代码评审人员.例如 CoreDevRec、TIE、IR+CN、PR+CF 等[11,15,16,17].

以上方法的评价指标均为能够推荐合适的代码评审人,而 2.1 节的数据分析表明,在开源社区的代码评审实践中,存在很多的合并请求不能及时得到响应的情况,这延长了评审周期、降低了参与人员的积极性和项目的发展.上述提到的基于准确性的代码评审人推荐方法并不能解决响应时间的问题.因此,我们提出响应时间约束的代码评审人推荐问题和基于多目标优化的代码评审人推荐方法,能够推荐合适且响应速度快的评审人,提高代码评审的响应速度和社区活跃性.

8. 结论

本文提出了响应时间约束的代码评审人推荐问题和基于多目标优化的代码评审人推荐算法 MOC2R,通过最大化代码评审人员的经验、最大化推荐人员在约定时间内的响应概率、最大化人员的活跃性来为代码评审推荐人员.实验结果表明 MOC2R 能够取得很好的推荐效果.在未来,我们计划探索更多的优化目标,有助于选择更加合适的代码评审人.同时寻求机会企业内部部署 MOC2R,以便更好地评估它在实践中的表现.还将研究如

何评估代码评审质量,以便进一步优化评审人推荐方法.

References:

[1] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 international conference on software engineering, pages 712–721. IEEE Press, 2013.

[2] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In Proceedings of the 36th International Conference on Software Engineering, pages 345–355. ACM, 2014.

[3] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 358–368. IEEE Press, 2015.

[4] Georgios Gousios, Margaret Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. Delft University of Technology Software Engineering Research Group, 1, 2014.

[5] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, pages 141–150. IEEE, 2015.

[6] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. ICSE'13, 2013, pp. 931–940.

[7] C. Hannebauer. Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Comparison. ASE'16.

[8] M. M. Rahman, C. K. Roy, and J. A. Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on. IEEE, 2016, pp. 222–231.

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

[10] J. Lipcak, B. Rossi. A Large-Scale Study on Source Code Reviewer Recommendation. 2018 IEEE DOI 10.1109/SEAA.2018.00068

[11] Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?. Information and Software Technology, vol. 74, pp. 204–218, 2016.

[12] 卢 松, 杨 达, 胡 军, 张 潇. 基于时间和影响力因子的 Github Pull Request 评审人推荐.计算机系统应用第 25 卷第 12 期,2016.

[13] Jiang J, Yang Y, He J, Blanc X, Zhang L. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. Inf. Softw. Technol. 84, 48–62, 2017.

[14] Jiang J, David L, Zhang L. Who should make decision on this pull request? Analyzing time-decaying relationships and file similarities for integrator prediction. The Journal of Systems and Software, 2019.

[15] J. Jiang, J.-H. He, and X.-Y. Chen. Coredevrec: Automatic core member recommendation for contribution evaluation. Journal of Computer Science and Technology, vol. 30, no. 5, pp. 998–1016, 2015.

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

[17] XIA Z, SUN H, JIANG J, WANG X, LIU X. A hybrid approach to code reviewer recommendation with collaborative filtering. 2017 6th International Workshop on Software Mining (Software Mining). IEEE, 2017: 24-31.

[18] YANG C, ZHANG X, ZENGL, FAN Q, YIN G, WANG H, An empirical study of reviewer recommendation in pull-based development model. Preceedings of the 9th Asia-PacificSymposium on Internetware.ACM, 2017:14.

[19] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In ISSTA'15, pages 234–245.

[20] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. JSS, 2010, 83(12):2416–2430.

[21] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In ICST, 2015, pages 1–10.

[22] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza. A systematic review on search-based mutation testing. IST, 2017, 81:19–35.

[23] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In ISSTA, 2007, pages 140–150.

[24] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In FSE, 2009, pages 111–120, 2009.

[25] https://google.github.io/eng-practices/review/reviewer/speed.html

[26] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4):35, 2014.

[27] Chris Sauer, D Ross Jeffery, Lesley Land, and Philip Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. IEEE Transactions on Software Engineering, 26(1):1–14, 2000.

[28] C. Tantithamthavorn, R. Teekavanich, A. Ihara, and K.-i. Matsumoto. Mining A Change History to Quickly Identify Bug Locations : A Case Study of the EclipseProject. In ISSREW'13, 2013, pp. 108–113.

[29] C. Tantithamthavorn, A. Ihara, and K.-I. Matsumoto. Using Co-change Histories to Improve Bug Localization Performance. In SNPD'13, Jul. 2013, pp. 543–548.

[30] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR), vol. 45, no. 1, p. 11, 2012.

[31] Jürgen Teich. Pareto-Front Exploration with Uncertain Objectives. EMO 2001: 314-328

[32] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, T. Meyarivan. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II. PPSN 2000: 849-858