软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(9):4056−4068 [doi: 10.13328/j.cnki.jos.006873]

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

TensorFlow 开源软件社区中贡献修订的实证研究*

李志星, 余 跃, 王 涛, 蔡孟栾, 王怀民

(国防科技大学 计算机学院, 湖南 长沙 410073)

通信作者: 余跃, E-mail: yuyue@nudt.edu.cn

摘 要: 人工智能 (artificial intelligence, AI) 的飞速发展得益于开源社区的开放协同, 大量的开发者通过提交 PR (pull-request) 为 AI 开源软件做贡献. 然而, 外部贡献者所提交的 PR 质量参差不齐, 开源项目管理团队需要对 PR 进行代码审查, 并要求贡献者根据审查意见对 PR 进行修订. PR 的修订过程对 AI 开源软件的质量有着重要的影响, 因此对该过程进行更加全面、深入的实证研究很有必要. 首先, 从 TensorFlow 开源软件社区中收集一组 PR 的修订历史, 通过对 PR 的代码提交信息以及审查评论进行定性分析, 归纳总结 PR 修订类型的分类体系. 其次, 根据此分类体系人工标注一组修订数据集, 并基于此数据集定量分析不同修订类型的频率分布、次序分布以及关联关系. 研究结果表明: TensorFlow 开源社区中的 PR 存在 3 大类共 11 种不同类型的修订, 其中完善类修订出现的频率最高; 此外, 相比于其他类修订和完善类修订, 修正类修订更常发生在 PR 的早期更新中; 与结构相关的修订更有可能与其他类型的修订同现或邻现, 配置修订以及变基修订有较大概率会接连出现. 实证研究结果可帮助 AI 开源实践者和研究者更好地理解 PR 的修订过程, 特别是有助于引导 PR 的审查和修订行为、提高开源群体协同效率.

关键词: 人工智能; 开源软件; 代码贡献; 代码审查; 代码修订

中图法分类号: TP311

中文引用格式: 李志星, 余跃, 王涛, 蔡孟栾, 王怀民. TensorFlow开源软件社区中贡献修订的实证研究. 软件学报, 2023, 34(9): 4056–4068. http://www.jos.org.cn/1000-9825/6873.htm

英文引用格式: Li ZX, Yu Y, Wang T, Cai ML, Wang HM. Empirical Study on Pull-request Revisions in Open Source Software Community of TensorFlow. Ruan Jian Xue Bao/Journal of Software, 2023, 34(9): 4056–4068 (in Chinese). http://www.jos.org.cn/1000-9825/6873.htm

Empirical Study on Pull-request Revisions in Open Source Software Community of TensorFlow

LI Zhi-Xing, YU Yue, WANG Tao, CAI Meng-Luan, WANG Huai-Min

(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: The recent boom in artificial intelligence (AI) benefits from the open collaboration of the open source software (OSS) community. An increasing number of OSS developers are contributing to AI projects by submitting pull requests (PRs). However, the PR quality submitted by external contributors varies, and the AI project management teams have to review PRs and ask contributors to revise them if necessary. Since the revision exerts a direct impact on the review efficiency and acceptance of PRs, it is important to achieve a better understanding of PR revisions. This study conducts an empirical study based on a set of PRs and their revision histories collected from the TensorFlow project. It first manually analyzes a sample of commit messages, reviews PR comments, and constructs a taxonomy of revision types. Then, according to the defined taxonomy, a large set of PR revisions are manually labeled. Based on the dataset, the frequency and order of each type of revision are explored. Additionally, this study also investigates the frequency distribution, order distribution, and correlation relationship between different types of revisions. The empirical findings show that there are 11 different types of revisions which can be classified into three categories. Evolvability revisions occur more frequently than other revision types, and

* 基金项目: 科技创新 2030—“新一代人工智能”重大项目 (2021ZD0112900); 国家自然科学基金 (62141209)
本文由“AI 软件系统工程化技术与规范”专题特约编辑张贺教授、夏鑫博士、蒋振鸣副教授、祝立明教授和李宣东教授推荐.
收稿时间: 2022-09-04; 修改时间: 2022-10-13, 2022-12-14; 采用时间: 2022-12-28; jos 在线出版时间: 2023-01-13
CNKI 网络首发时间: 2023-07-06

functional revisions are more likely to occur in the early PR updates than evolvability revisions and other types of revisions. Structure-related revisions have a high chance to co-occur or adj-occur with other revisions. Configuration-related revisions or rebasing revisions are more likely to appear in succession. The empirical results can help AI open source practitioners and researchers better understand the PR revision process, especially guide the review and revision behaviors of PRs and improve the collaborative efficiency of open source groups.

Key words: artificial intelligence (AI); open source software (OSS); pull-request (PR); code review; revision

近些年, 人工智能 (artificial intelligence, AI) 技术得到了迅猛发展, 在语音识别、图像识别等领域中取得了一系列重要突破[1,2], 正在加速融入医疗、教育、家居、交通等各行各业并发挥着重要作用[3,4]. 而 AI 的飞速发展在很大程度上得益于开源软件技术, 开放、共享的开源协同社区为 AI 软件的开发和维护聚集了大量的开发者, 为 AI 软件的高质量持续发展提供了创新源泉. 尤其是开源社区中的 PR (pull-request) 开发机制[5]简化了群体协同开发的过程, 降低了外部贡献者的参与门槛, 吸引了越来越多的开发者参与开源贡献. 同时, PR 开发机制具有较高的自动化和集成化水平, 能够减轻项目核心团队的管理负担, 提高其决策效率[6]. 因此, 主流的开源协同平台 (如国外的 GitHub (https://github.com/)、GitLab (https://about.gitlab.com/) 以及国内的 GitLink (https://www.gitlink.org.cn/)、Gitee (https://gitee.com/) 等) 都已陆续支持 PR 开发机制, 越来越多的开源项目开始采用这种协同机制, 例如 TensorFlow、scikit-learn、Keras、PyTorch 等 AI 框架都是在 GitHub 平台上开展基于 PR 的协同开发.

虽然 PR 开发机制提高了 AI 开源群体的协同效率, 但是其无法完全避免代码贡献中的质量问题, 开发者所提交的 PR 往往不具有可立即接受的初始状态. 尤其是随着大规模贡献者群体的涌入, 开发者之间的水平差异变得更加明显, AI 开源软件收到问题代码的风险持续增高. 因此, 为了保证开源软件的质量, 每一个提交的 PR 都需要经过项目管理团队的严格审查[7−9], 并根据项目的相关规范进行必要的修订, 直到符合标准后才能被正式接受. 由此可见, PR 的修订过程对于 PR 的最终状态以及 AI 软件的质量具有重要影响. 以往的研究工作对传统开发环境下软件代码的缺陷和修订类型进行了深入分析, 而 PR 开发机制是一种全新的协同开发模式, 其贡献修订过程还有待进一步探索.

为更加深入和全面地理解 AI 开源软件社区中的贡献修订过程, 本文基于 TensorFlow 项目中的历史 PR 修订数据进行实证研究, 并重点探索了以下 4 个研究问题.

(1) RQ1: PR 会发生哪些类型的修订? 我们发现 TensorFlow 项目中的 PR 存在 11 种不同的修订类型, 可进一步分为 3 类, 包括完善类修订 (如文本和结构)、修订类修订 (如逻辑和检查) 以及其他类修订 (如配置和变基), 变基 (git rebase) 是将一个代码分支上的修改在另一个代码分支上进行重现的过程, 在基于 PR 的协同开发场景下常被用来同步主版本仓库中的最新代码. 此研究问题能够为项目管理团队设置 PR 审查清单和审查任务分工提供参考.

(2) RQ2: 不同类型的修订有怎样的频率分布? 总体来看, 完善类修订出现的频率明显多于修订类修订和其他类修订, 其比例约为 5:3:2. 具体来看, 其中较为普遍的修订类型包括结构和变基. 此研究问题有助于启发围绕常见修订类型的自动化工具设计以及协同过程优化.

(3) RQ3: 不同类型的修订所发生的次序有何特点? 相比较而言, 修正类修订所发生的次序比完善类修订与其他类修订更靠前一些, 例如接口修订往往发生在 PR 的前 3 轮更新中, 而变基修订往往发生在 5 轮以后的更新中. 此研究问题能够帮助项目管理者更加合理地规划自己的审查计划.

(4) RQ4: 不同类型的修订存在怎样的关联关系? 修订类型的同现概率矩阵与邻现概率矩阵显示, 结构修订与大多数修订类型均具有较高的同现及邻现概率. 除此之外, 当 PR 的某次更新中发生了配置修订或变基修订时, 其下一次更新中有较大的概率还会再次发生同类型的修订. 此研究问题有助于引导项目管理者充分发现 PR 中隐含的问题, 同时也能够帮助贡献者避免在修订 PR 时引入新的问题.

本文的主要贡献包括如下内容.

(1) 总结了 TensorFlow 开源社区中 PR 的修订类型并分析了每种修订类型的频率分布, 有助于 AI 开源实践者和研究者对 PR 的修订过程形成更为清晰和直观的认识, 优化协同模式和协同工具.

(2) 揭示了不同修订类型的次序分布以及它们之间的同现关系及邻现关系, 对于引导 AI 开源实践者的修订和

审查行为、增强代码缺陷预测模型具有现实指导意义.

本文第 1 节介绍研究背景. 第 2 节介绍相关研究工作. 第 3 节介绍实证研究方法, 主要包括数据采集、数据分类、数据标注以及数据分析. 第 4 节围绕 4 个研究问题报告实证研究结果. 第 5 节讨论有效性和研究启示. 第 6 节总结全文.

1 背景介绍

AI 框架往往以开源的形式存在, 全世界的 AI 开发者能够自由地在开源社区中进行协同. 当前, PR 开发机制已经逐渐成为开源群体协同的标准范式, 尤其是经过多年不断的发展与完善, GitHub 平台已经逐步将代码提交、代码审查、任务管理、进度跟踪等工具集成到 PR 机制中[5,10]. 由于 PR 能够显著降低参与门槛、提高协同效率, 越来越多的开源项目开始采用基于 PR 的协同模式. 如图 1 所示, 基于 PR 的协同过程主要包括以下步骤.

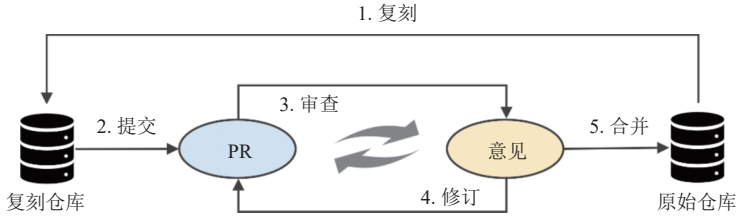

图 1 基于 PR 的协同开发过程

(1) 复刻 (fork): 对于一个开源软件项目, 任何一个感兴趣的开发者都可以通过复刻 (fork) 操作获取其源代码仓库的副本 (即复刻仓库), 由于复刻仓库中包含了与原始仓库相同的代码及提交记录, 开发者可以基于复刻仓库进行独立的代码开发.

(2) 提交 (summit): 当开发者在复刻仓库完成了具体的开发任务后, 在原始仓库的管理页面提交一个 PR 以通知原始仓库的管理者进行代码审查. 新创建的 PR 会包含一个简短的标题以及一段详细的描述, 同时 GitHub 也会根据贡献者指定的目标分支 (原始仓库中) 和工作分支 (复刻仓库中) 显示发生的代码变更.

(3) 审查 (review): 管理者会对发生变更的代码文件和代码片段进行人工审查, 并通过在 PR 中添加评论的方式发表修订意见. 另外, 一些开源项目会使用持续集成等自动化测试工具, 这些工具的测试结果也会被自动反馈到 PR 的管理页面.

(4) 修订 (revise): 当贡献者收到管理者的审查意见或自动工具的检查结果后需要对 PR 进行更新, 原始的讨论记录会被持续保存, 而新的代码更新会触发新一轮的审查. 通常, PR 的修订过程会迭代多次, 直到管理者同意接受该 PR 为止.

(5) 合并 (merge): 管理者将 PR 中的代码变更合并到原始仓库的目标分支中.

由以上流程可见, 贡献者提交的原始代码往往要经过一系列的修订才能被合并进开源软件的代码仓库, 修订过程是 PR 开发机制中的重要步骤, 其直接影响着开发者的协同效率以及软件的质量. 因此, 本文希望通过实证研究更加深入地理解开源软件项目, 尤其是 AI 开源软件中 PR 的修订过程.

2 相关研究

本节从以下两个方面对相关研究工作进行介绍.

2.1 PR 评估相关研究

PR 评估是软件协同开发过程中的一项重要活动, 确保了贡献者所提交的代码贡献不会引入任何质量问题. 当前有相关研究分析了开发者如何评估拉请求的质量, 如 Gousios 等人[11]的调研显示, 项目管理者比较关注拉请求代码与项目规范的一致性、源代码的可读性、测试覆盖率等因素. 另外, Dabbish 等人[12]指出管理者会根据拉请求提交者的历史贡献数据推断拉请求的质量. 为帮助项目管理者及时挑选合适的 PR 进行审查, 相关研究工作提

出了自动化的 PR 审查者推荐技术, 其中最常见的方法是度量新 PR 与历史 PR 之间的相似性[13], 然后将历史上最相似的 PR 的审查者推荐为新 PR 的审查者. 关于 PR 决策, Tsay 等人[14]通过回归分析发现项目管理者在做决策时会同时考虑技术性因素与社会化因素, 他们还通过对拉请求历史讨论数据的分析发现管理者的决策会受社区意见的影响[15]. Gousios 等人[11]通过对项目管理者的调研发现影响管理者决策的主要因素有拉请求质量、测试结果等. Ford 等人[16]通过眼动跟踪实验发现审查者在拉请求评估过程中会参考贡献者的历史开发活动以及个人主页信息.

当前关于 PR 评估的研究工作重点关注于 PR 的审查过程, 深入揭示了影响 PR 审查过程和决策结果的各种因素. 然而, 对于 PR 最终是否能合并以及 PR 的评估延迟, PR 修订过程同样扮演着重要的角色. 因此, 深入分析 PR 修订过程有利于完善对 PR 评估过程的理解, 进而提高 AI 开源软件项目的群体协同效率.

2.2 软件代码修订相关研究

软件代码缺陷及修订得到了研究学者的广泛关注. 在早期的研究工作中, Chillarege 等人[17]定义了 8 种软件缺陷类型, 进而对软件开发过程进行更细致的评估度量, 他们所定义的缺陷类型包括功能类 (functional)、接口类 (interface)、算法类 (algorithm) 等, 他们还进一步探索了不同修订类型在软件开发各个周期中的数量分布特征. El Emam 等人[18]基于一个企业软件项目的代码审查数据对文献 [17] 中所定义的分类模型进行测试, 他们发现该分类模型具有较高的再现性. 不过它们也观测到了少量不一致的数据, 并据此对该分类模型提出了相应的改进意见. Mäntylä 等人[19]通过对软件从业者与软件开发课程学生所接收到的代码审查评论进行分析, 建立了较为系统的代码缺陷分类体系, 他们将代码缺陷分为两大类, 即功能性缺陷和演化性缺陷 (evolvability defect), 他们的统计分析显示演化性缺陷比功能性缺陷更加常见, 其数量比例达到了 3:1. Beller 等人[20]基于开源软件代码审查数据对文献 [19] 中的研究发现进行了验证, 他们根据该模型人工标注了 1 400 多条代码修订数据, 结果显示开源软件中演化性修订与功能性修订间的数量比例也是 3:1. Panichella 等人[21]则通过开发者调研以及对 10 个开源项目历史审查数据的分析, 建立了更为详细的开源软件代码修订分类体系, 并将其与 Mäntylä 等人[19]建立的分类体系进行了融合. 最终, 他们新增了“其他类”这一主要类型来表示那些非源代码修订, 除此之外, 他们还针对每一种修订类型提出了自动化建议.

现有文献对代码的缺陷和修订类型进行了广泛和深入的分析, 但是不同修订类型的发生次序以及它们之间的关联关系还未得到进一步研究, 尤其是 AI 开源软件中的贡献修订实践还未得到专门研究. 此外, GitHub 作为当前最为流行的开源协同平台, 其所支持的基于 PR 的协同过程与工程化协同过程及传统的开源协同过程具有显著的不同, 因此有必要探索 PR 协同模式下是否存在与以往不同的修订类型以及修订的频率分布是否发生变化.

3 研究方法

如后文图 2 所示, 我们的研究方法主要包括 4 个步骤: 数据采集、数据分类、数据标注以及数据分析. 本节剩余内容介绍每个步骤的具体实施过程.

3.1 数据采集

遵循以往相关工作的研究策略[22−24], 我们将研究视角聚焦于流行的 AI 开源软件项目. 具体地, 本文选取 TensorFlow 软件项目展开实证研究, TensorFlow 是一款著名的开源 AI 框架, 在 GitHub 有超过 16.7 万的关注量 (即 star 数[25]), 历史上曾有 3 千多名开发者为其贡献过代码, 提交了超过 2 万多个 PR. 在数据采集阶段, 我们通过 GitHub 的官方 API (http://developer.github.com/v3/) 获取 TensorFlow 中 PR 的修订历史. 具体地, 我们通过调用接口“timeline”获取一个 PR 所发生的所有历史事件, 其中“commited”和“head_ref_force_pushed”事件表示 PR 的代码发生了变更, 前者表示贡献者又添加了新的代码提交 (commit), 后者表示贡献者直接更新了开发分支. 对于此两类事件, 我们进一步获取其对应的代码提交数据, 主要包括提交时间和提交信息 (commit message). 类似地, 我们通过接口“comments”获取 PR 的讨论历史, 包括一般性审查评论和代码级审查评论. 对于每一个评论, 我们解析出的信息包括评论者、评论时间以及评论文本内容.

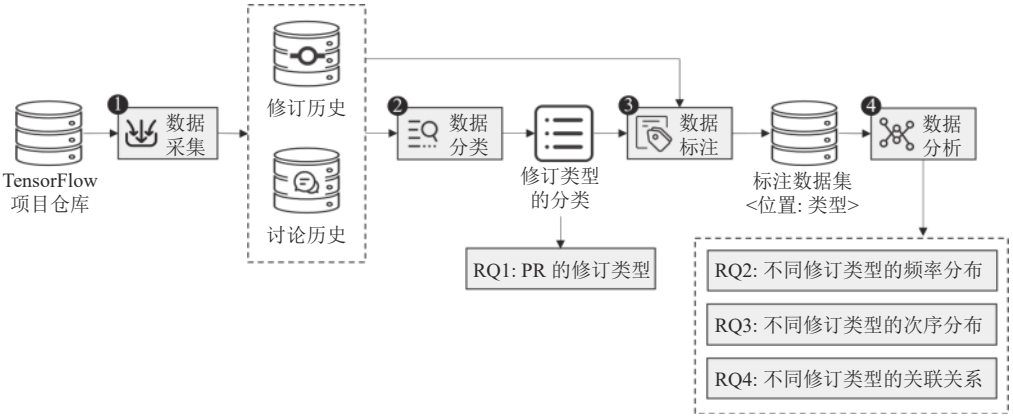

图 2　研究方法概览

3.2　数据分类

我们通过人工分析 PR 的修订历史和讨论历史, 对 PR 的修订类型进行分类. 为了能够充分挖掘 PR 的修订过程, 我们特别选取出那些经历了多轮更新的 PR, 并从它们历次的代码提交信息与审查评论中判断它们都发生了哪些类型的修订. 鉴于以往的研究工作已经对软件问题的类型做了较为深入的研究, 我们参照其中最新的一项研究成果[21]进行的数据分析. 具体地, 我们指派两名作者进行合作, 按照 Panichella 等人[21]提出的分类模型对 TensorFlow 中 PR 的修订数据进行联合分类, 当有些数据无法归类到已有的分类模型中时, 我们会创建新的类型. 此分析过程依照定性分析中经常采用的数据饱和原则[26,27], 即我们每次随机选择一项代码提交信息或者审查评论进行分类, 当我们连续分析了 N 个数据项都没发现新的类别时, 我们结束该分类过程. 为了提高分类过程的置信度, 我们将 N 设置为 5. 基于以上操作的结果, 我们回答第 1 个研究问题.

3.3　数据标注

根据上一步定义的 PR 修订分类模型, 我们进一步标注了一组数据集. 首先, 我们随机挑选出 20 个 PR, 并按照事件发生的先后顺序逐个遍历 PR 的所有事件, 当一个修订事件 (即第 3.1 节中所提到的两种代码变更事件) 伴随着一个非修订事件时, 我们则判定 PR 发生了一次更新. 将所有的更新按序编号后, 会形成一组更新集合 $<u_1,u_2,u_3,\ldots,u_n>$, 对于其中的某一次更新 u_i, 我们对 u_{i-1} 与 u_i 之间发生的所有审查讨论, 以及 u_i 所关联的代码提交信息进行分析, 以标注 u_i 都包含了哪些类型的修订. 对于其中任何一类修订 r, 我们生成一个标记数据项 $<i,r>$. 为减轻人工标注过程的主观性, 我们仍然指派两名作者进行联合标注, 当两人意见不一致时, 邀请第三人进行额外的独立标注, 并通过投票策略 (即少数服从多数) 来确定最终的标注结果. 在该过程中我们共标记了 193 次 PR 更新中的 384 次修订.

3.4　数据分析

基于标注数据集, 我们分别做了以下 3 方面的数据分析.

3.4.1　频率分析

我们首先计算数据集中所有修订实例的总数量, 然后我们计算每一种修订类型的实例的数量, 进而得到每一种修订类型发生的频率. 值得注意的是, 在 PR 的一次更新中, 有可能会出现多处代码发生同类修订的情形, 我们的处理策略是: 一种修订类型在一次更新中只会被统计 1 次.

3.4.2　次序分析

如算法 1 所示, 我们迭代处理 PR 的每一次更新 (第 3 行), 并从中解析出两个参数, 即当前更新是第几轮 (第 4 行), 以及当前更新都发生了哪些类型的修订 (第 5 行). 修订轮数的值会被依次加入每一种修订类型对应的修订轮数集合中 (第 6–8 行). 最终, 我们就获得了每一种修订类型的次序分布 (第 10 行).

算法 1. RevisionPositions.

输入: PR 修订数据集 (*revisions*);
输出: 每种修订类型发生的次序集合 (*change_itrs*).

过程:

```
1. Let change_pos be a map
2. Initialize change_poss as <revision type:list of revision round>
3. for revision in revisions
4.     pos ← revision.whichRound
5.     changes ← revision.changesSet
6.     for change in changes:
7.         add pos to change_pos[change]
8.     end
9. end
10. return change_pos
```

3.4.3 关联分析

我们对不同修订类型间的关联关系进行了分析. 具体地, 对于给定的一种修订类型 r_j, 我们计算它与修订类型 r_j 同时出现在同一次 PR 更新中的概率, 以及它与 r_j 先后出现在相邻的两次 PR 更新中的概率, 其计算过程分别如公式 (1) 和公式 (2) 所示.

$$co\_prob\left(r_i, r_j\right) = \frac{co\_count\left(r_i, r_j\right)}{count\left(r_i\right)},\ 1 \leqslant i, j \leqslant 12 \text{ 且 } i \neq j \tag{1}$$

$$adj\_prob\left(r_i, r_j\right) = \frac{adj\_count\left(r_i, r_j\right)}{count\left(r_i\right)},\ 1 \leqslant i, j \leqslant 12 \tag{2}$$

对于同现概率的计算公式 (co_prob), 函数 co_count 返回的是我们数据集中两种修订类型同时出现在同一次 PR 更新中的次数, 函数 $count$ 返回的是在所有的 PR 更新中某种修订类型发生的总次数. 对于相邻概率的计算公式 (adj_prob), 函数 adj_count 返回的是两种修订类型先后出现在相邻的两次 PR 更新中的次数.

4 研究结果

4.1 PR 修订类型

如表 1 所示, 我们发现 PR 所发生的修订主要包括 11 种, 这些修订类型可进一步分为 3 组. 对于前两组修订类型 (即完善和修正), 以往的研究工作已经进行了充分的介绍, 而对于第 3 组修订类型 (即其他), 本文新发现了两种类型, 即变基和撤销. 在接下来的章节中, 我们会基于实例对每种类型的修订进行详细解释.

4.1.1 完善类修订

完善类修订是指 PR 的代码本身并不存在功能性缺陷 (既不影响软件的正常运行), 修订目的主要是提高代码的质量, 以增加软件项目的可维护性. 完善类修订主要包括 3 种类型: 文本、风格以及结构.

● 文本: 该类型的修订能够提高代码的可读性, 代表性的修订包括优化缺失或不准确的代码注释 (“This doesn’t match with the comment saying but change the op name. Change the comment?”)、规范变量的命名 (“We eschew ‘unusual’ or ‘bespoke’ abbreviations inside of Google code, so tgt_id is not a good name. Perhaps simply intrinsic or intrin (the latter being a relatively common abbreviation)”) 等.

● 风格: 此类型的修订主要是解决代码风格一致性方面的问题, 例如代码缩进标准 (“Please keep the

indentation at 2 spaces, here and later”)、空格/空行的规范使用 (“No newline (although you don’t have to change this, it’ll be fixed when we import the PR)”)、特殊标号的使用习惯 (“Please add () to make it clear”) 等. 目前, GitHub 项目通常使用持续集成服务[28]自动检测并报告风格类的代码问题, 这在一定程度上提高了 PR 的审查和修订效率.

● 结构: 该类型的修订包括对代码组织结构的调整, 如将某个代码片段移动到更合适的位置 (“Could you put all const Tensor** types close to each other?”)、处理重复代码 (“This line seems redundant with the two lines below. Remove?”) 等. 此外, 该类型的修订还包括对代码实现细节的调整, 如剔除不可达的代码片段 (“Unnecessary, as this is unreachable code”)、补充测试用例 (“Can you write a test which fails before this PR and passes after?”) 等.

表 1　PR 修订类型的分类模型及描述

分组	类别	描述
完善	文本	修订文本型缺陷, 如缺少注释、命名不规范等
	风格	修订代码风格问题, 如多余的空格或空行等
	结构	修订代码组织或实现方案方面的问题
修正	资源	修订数据、变量等资源处理相关的错误
	逻辑	修订控制流、对比、计算等操作中存在的错误
	接口	修订模块间交互时的错误
	检查	修订检查变量值、函数返回值、用户输入等操作中的错误
	缺陷	修订程序完整性、兼容性等方面的重大缺陷
其他	配置	修订配置中的问题
	变基*	执行变基操作
	撤销*	撤销不必要的代码变动

注: *表示本文新发现的类别

4.1.2　修正类修订

修正类修订解决的是导致软件无法正常运行或者在软件运行时导致非预期表现的功能性错误, 其主要包括 5 种具体的修订类型: 资源、逻辑、接口、检查以及缺陷.

● 资源: 该类型的修订主要是解决与数据处理和变量操作相关的错误, 如变量的定义和初始化 (“Use int64 here, shape is a vector of int64” “Undefined variable ‘_Normalize’”) 等.

● 逻辑: 此类型的修订修复的是代码逻辑方面的问题, 包括错误的对比值 (“Should not be DISABLE?”)、算法性能问题 (“I think something about this change is indeed causing timeouts. Possibly the new summarization code is too slow and is executed in some of the tests that are timing out.”) 等.

● 接口: 该类型的修订解决的是模块间交互时错误, 如函数调用出错 (“slice_t.eval() doesn’t work in eager mode. Please use self.evaluate instead.”)、参数传递出错 (“evaluate_generator() is missing the class_weight param, which cause test failure. Please check the test log”) 等.

● 检查: 此类型的修订主要是修复检查、验证相关的问题, 包括检查函数调用结果 (“Please add a check if the results are correct”)、检查变量取值 (“Please also check if node is not null”) 等.

● 缺陷: 该类型的修订解决的是影响较大的问题, 如方案完成度问题 (“support dynamic shaped input for time_major rnn and other minor changes”)、兼容性问题 (“backward compatability. You might be ok if none is really using the old version”) 等.

4.1.3　其他类修订

除了以上两类修订之外, 还有一部分修订往往不涉及对具体源代码文件的编辑, 包括: 配置、变基以及撤销.

● 配置: 软件项目除了源代码文件外, 还有诸多配置文件, 此类型的修订就是修改与这些配置相关的内容, 如配置代码编译信息 (“You also need to create this BUILD file. Or we can simply check in a bazel BUILD file in MKL-DNN

repository and not need this?”).

● 变基: PR 的审查过程较长, 往往能够持续数周甚至数月, 而在此过程中, 项目的主版本库有可能会陆续合并了新的代码, 因此开发者通常需要使用变基操作 (rebase), 将项目主版本库中最新的代码同步到 PR 所对应的开发分支中, 例如: “apparently recent changes in MLIR has changed the implementation quite a bit. let me rebase and adjust to the new codes.”

● 撤销: 有些修订内容是不必要的, 因此项目管理者会让贡献者撤销不必要的修订, 例如: “Unnecessary change, please revert”“Since a quick search over the non-contrib part of your pull request didn’t find any usages of these new symbols I think it’s more prudent to drop them”.

4.2 不同修订类型的频率分布

图 3 所示为各修订类型的频率分布. 总的来看, 完善类修订比修正类修订和其他类修订出现的频率更高一些. 具体地, 有 48% 左右的修订是完善类, 29% 左右的修订是修正类, 剩余 22% 的修订是其他类. 与以往研究所报告的数据进行对比, 我们发现 3 类修订的频率并未发生明显变化, 只是其他类修订的数量相对多了一些.

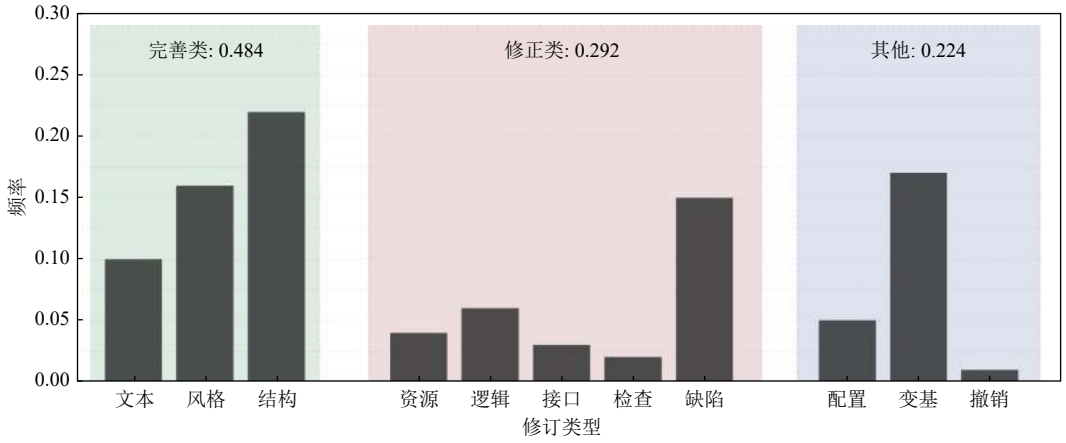

图 3 不同修订类型的频率分布

对于完善类修订, 与结构修订的数量最多, 而风格修订数量最少. 由于开源项目往往会配置静态代码分析工具 (如 Checkstyle (https://heckstyle.org/)、Pylint (https://pylint.pycqa.org/en/latest/)、Clang-format (https://lang.llvm.org/docslangFormat.html) 等), 并且通常会请求贡献者在提交 PR 之前, 先在本地运行相关脚本对代码更改进行质量检测, 这在一定程度上减少了 PR 提交后代码风格方面的修订. 而对于代码结构方面的问题, 由于它们不易被贡献者提前发觉并加以避免, 因此此类型的修订往往由项目管理者的审查驱动. 鉴于结构修订出现的频率最高, 如果当前的代码推荐等自动化工具能够被有效地集成到 PR 开发模型中, 这将有助于减少 PR 的修订次数, 进而提高开源开发者的协同效率.

对于修正类修订, 缺陷修订比其他几种类型的修订有更多的数量. 以往的研究工作指出, 超过 80% 的贡献者在提交 PR 前, 会检查他们的代码更改是否能够通过测试[29], 这导致大多数的 PR 较少会发生资源操作、运行逻辑和接口调用方面的问题. 而对于解决方案不完整以及与以往版本不兼容等问题, 本地测试不一定能够完全发现它们, 从而导致此类型的修订在 PR 的人工审查过程中出现的次数相对较多一些.

对于其他类修订, 变基修订有着极为显著的数量优势, 其出现的频率远远超过其他几个类型的修订. 变基是 PR 协同模式下分布式异步协同流程中的必要操作, 如图 4 所示, 在一个 PR 尚处于审查状态时, 项目主开发分支可能已经合并了新的代码提交, 为了更精确地判断 PR 是否依然能够被安全合并, 项目管理者通常要求贡献者将项目最新的代码更改同步到 PR 中, 而这一般由变基操作完成. 由于软件的快速演化和不可避免的审查延迟, 对 PR 执行变基操通常是难以避免的. 不过, 由于变基操作有时仅是无变更的更新操作 (即 PR 对应分支和项目的主

开发分支不存在合并冲突), 协同平台若能支持 PR 的自动变基, 则 PR 的审查延迟问题能够得到部分缓解. 此外, 能够自动解决合并冲突的工具有助于进一步提高变基修订的效率.

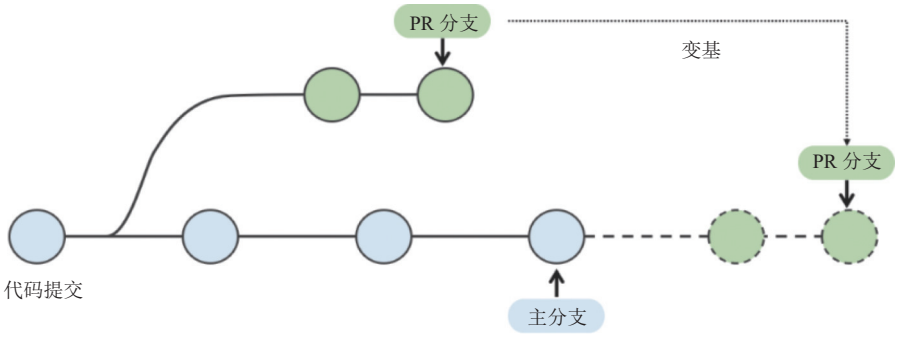

图 4 PR 的变基操作示意图

4.3 不同修订类型的执行次序

图 5 所示为不同修订类型执行次序 (即发生在 PR 的第几轮更新) 的分布情况. 我们用盒图的形式展示了每一种修订类型中所有修订实例的执行次序的分布, 并另外用 3 个彩色盒图展示了 3 组修订类型的总体分布情况. 可以看出, 修正类修订的执行次序比完善类修订和其他类修订更小一些 (中位值大小分别为: 4, 5, 5), 即发生的时间更早一些, 并且我们通过 Mann-Whitney-U 检验[30]发现这种差异性是显著的 (P<0.05). 此现象与实际的代码审查实践活动相一致, 如 Gousios 等人[11]指出, 项目管理者优先关注 PR 的正确性和质量, 即在审查的早期会重点检查代码是否存在技术性错误, 以及是否可以进一步优化. 而当管理者基本完成了质量审查后并即将接受 PR 时, 会着重检查 PR 是否符合项目的贡献合并规范, 因此风格修订所发生的时间通常较晚. 对于变基和撤销修订, 它们往往是当审查过程进行到一定程度后才更有可能发生的修订, 因此它们的执行次序也相对较为靠后.

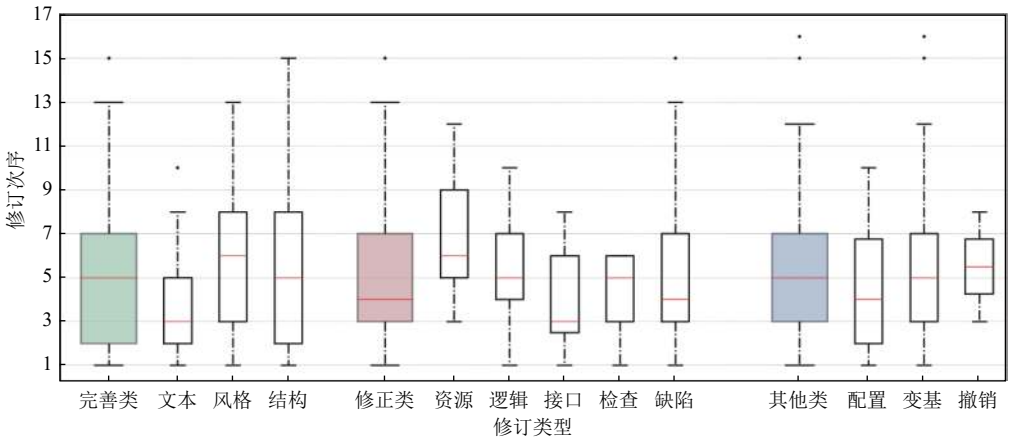

图 5 不同修订类型的执行次序的分布

4.4 不同修订类型的关联关系

表 2 和表 3 分别是不同类型的修订同时出现和先后相邻出现的概率矩阵. 关于不同类型的修订同时出现在 PR 的一次更新中的概率, 我们从表 2 中的数据可以看到, 对于所有的修订类型, 结构修订都具有最大的同现概率, 其中与检查修订同现的概率最大. 而当 PR 的某次更新中包含了结构修订时, 最有可能与其同现的修订是风格修订. 此外, 相比较而言, 文本修订和风格修订也具有相对较高的概率和其余类型修订同现, 这很大程度上应该是由于这些修订类型本身具有较高的出现频率 (如图 3 所示). 值得注意的是, 配置和变基修订较少和其他类型的修订同现.

表 2 不同类型的修订同时出现的概率矩阵

类型	文本	风格	结构	资源	逻辑	接口	检查	缺陷	配置	变基	撤销
文本	—	0.36	0.48	0.08	0.16	0.04	0.04	0.28	0.04	0.16	0.04
风格	0.22	—	0.46	0.10	0.07	0.07	0.07	0.20	0.02	0.10	0.02
结构	0.22	0.35	—	0.09	0.11	0.05	0.05	0.22	0.02	0.09	0.02
资源	0.22	0.44	0.56	—	0.00	0.11	0.00	0.33	0.00	0.11	0.00
逻辑	0.29	0.21	0.43	0.00	—	0.07	0.07	0.21	0.00	0.07	0.07
接口	0.14	0.43	0.43	0.14	0.14	—	0.14	0.43	0.00	0.00	0.14
检查	0.20	0.60	0.60	0.00	0.20	0.20	—	0.40	0.00	0.00	0.20
缺陷	0.18	0.21	0.32	0.08	0.08	0.08	0.05	—	0.00	0.05	0.03
配置	0.08	0.08	0.08	0.00	0.00	0.00	0.00	0.00	—	0.08	0.00
变基	0.10	0.10	0.12	0.02	0.02	0.00	0.00	0.05	0.02	—	0.00
撤销	0.50	0.50	0.50	0.00	0.50	0.50	0.50	0.50	0.00	0.00	—

表 3 不同类型的修订先后相邻出现的概率矩阵

类型	文本	风格	结构	资源	逻辑	接口	检查	缺陷	配置	变基	撤销
文本	0.41	0.32	0.41	0.09	0.05	0.00	0.05	0.32	0.09	0.23	0.00
风格	0.16	0.32	0.32	0.05	0.05	0.08	0.03	0.24	0.03	0.14	0.03
结构	0.16	0.30	0.34	0.08	0.08	0.06	0.02	0.22	0.04	0.22	0.00
资源	0.00	0.12	0.12	0.00	0.00	0.00	0.00	0.38	0.12	0.25	0.00
逻辑	0.25	0.00	0.17	0.17	0.17	0.00	0.00	0.25	0.08	0.25	0.00
接口	0.14	0.57	0.29	0.00	0.29	0.00	0.00	0.43	0.00	0.00	0.00
检查	0.25	0.25	0.25	0.00	0.25	0.25	0.00	0.75	0.00	0.25	0.00
缺陷	0.12	0.12	0.35	0.03	0.15	0.03	0.06	0.29	0.03	0.12	0.00
配置	0.18	0.09	0.27	0.00	0.09	0.00	0.00	0.00	0.64	0.18	0.00
变基	0.17	0.14	0.17	0.06	0.00	0.00	0.00	0.22	0.03	0.44	0.03
撤销	0.50	0.50	0.00	0.00	0.50	0.00	0.00	0.50	0.00	0.00	0.00

关于不同类型的修订先后出现在 PR 两次相邻的更新中的概率, 我们从表 3 中的数据可以看出, 结构修订仍然与大部分类型的修订具有最高的相邻出现的概率, 尤其是当 PR 的前一次更新中包含文本修订时, 下一次更新有 40% 的概率会发生结构修订. 有意思的是, 当 PR 发生了配置修订或变基修订时, 其下一次更新中有较大概率会再次发生同类修订, 其概率分别为 64% 和 44%.

5 讨 论

5.1 有效性分析

影响我们实证研究结论有效性的一个主要因素来自数据集. 我们的实证分析基于 TensorFlow 开源项目, 虽然它是具有代表性的 AI 开源软件, 但是在 GitHub 以及其他开源平台上存在着数量巨大的 AI 开源软件, 我们无法保证本文的实证分析结果能够推广到其他任何一个开源项目. 此外, 我们也无法保证文中的发现是否只特定于人工智能社区. 因此, 未来的工作可以考虑软件在编程语言以及应用领域上的多样性, 并且将国内开源社区和国产 AI 框架纳入研究范围, 在更大规模的数据集上对本文研究结果作进一步验证.

人工分析过程也会影响实证研究结论的有效性. 本文的数据分类和数据标注都是人工分析过程, 而人工分析有可能会产生误差或者引入主观偏见, 为了缓解这一问题, 我们指派多个作者联合执行这些过程, 当他们出现意见不一致的情况时, 会进行讨论以达成共识, 必要时会邀请另外作者参与讨论.

5.2 启 示

图 3 的结果显示有近 17% 的修订是变基修订, 如前所述, 变基操作是开发分支之间的同步行为, 其目的是将

PR 更新到可审查的状态, 并不涉及对代码的实质性改进, 因此此类修订应尽量避免. 当一个项目中频繁出现变基修订时, 项目管理团队应该评估导致该现象出现的原因是项目演化过快还是审查延迟过长. 如果是后者, 管理团队应该及时改进其 PR 审查过程, 或者采用适当的推荐工具[31]来辅助项目中的任务调度工作.

为了提高代码的可维护性, 开源软件项目一般都有代码风格要求, 这往往也是贡献指南中会重点突出的元素[32]. 除此之外, 一些软件还会提供代码风格自动规整工具以减轻贡献者的负担. 然而我们发现有近 16% 的修订与代码风格相关 (如图 3 所示) , 这有可能是由于一次性贡献者或偶尔贡献者的群体规模逐渐增大的原因, 因为这些贡献者在风格规范化等非核心工作上花费额外精力的意愿往往不强烈[33,34]. 目前 CI 工具已被广泛应用于自动发现代码风格问题, 这有利于减少管理者的审查负担, 开源协同平台可以设计面向代码风格的自动修复工具, 这有助于进一步减少 PR 的审查延迟. 此外, 开源协同平台可以面向一次性贡献以及偶尔贡献场景设计相应的协同辅助工具, 提高该场景下的协同效率.

图 5 的结果显示, 不同修订类型所发生的次序存在差异. 由于并不是所有的 PR 都能够被接受, 为了防止不必要的修订 (如贡献者根据审查意见修复了很多代码风格问题, 并执行了多次变基操作后, 被告知因其所建议的新功能不符合项目发展方向, PR 最终被拒绝了), 管理者应该注意设置合理的审查关注点顺序. 例如, 一些开源项目要求 PR 审查过程应先关注宏观重点问题再关注细节问题 (“Upfront architectural or larger changes should be resolved first before nits”), https://github.com/kubernetes/community/blob/master/contributors/guide/review-guidelines.md

我们发现一些修订类型存在明显的同现和邻现关系. 同现关系能够启发管理者实现更加全面和针对性的审查, 例如当管理者发现一个 PR 需要接口方面的修订时, 由于风格修订与接口修订有较高的同现概率, 管理者则需要重点审查该 PR 是否也需要风格修订. 修订类型间的邻现关系有利于引导贡献者的修订过程, 例如当管理者要求贡献者对其拉请求进行配置相关的修订时, 由于该类型的修订有较大的概率会接连出现, 因此贡献者在更新 PR 时需要特别注意是否能够一次性完美解决配置相关的问题. 此外, 对于自动缺陷预测工具, 相关研究可以考虑利用修订类型间的关联关系来增强预测效果, 例如将 PR 在历史上发生的修订类型作为特征变量加入模型的输入中.

6 总结与展望

AI 开源软件的开发与维护依赖于开源社区的开放与协同, 然而社区贡献者提交的 PR 质量参差不齐, 因此 PR 的审查和修订过程对 AI 软件的质量至关重要. 本文通过定性分析和定量分析相结合的方式对 TensorFlow 中 PR 的修订过程进行了实证研究. 我们首先基于 PR 的代码提交信息和审查评论信息归纳总结了 PR 的修订类型. 其次, 我们基于人工标注数据集量化分析了不同修订类型的频率分布、次序分布以及关联关系. 未来的研究工作可进一步探索修订类型与开发者特征之间的关系, 如分析哪些开发者所提交的 PR 更容易发生某种特定类型的修订. 其外, 从贡献者视角分析不同修订类型的执行难度也将会是一项有意义的工作, 这有助于引导审查者发表更具备执行性的修订意见.

References:

[1] He KM, Zhang XY, Ren SQ, Sun J. Deep residual learning for image recognition. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 770–778. [doi: 10.1109/CVPR.2016.90]

[2] Abadi M, Barham P, Chen JM, Chen ZF, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng XQ. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 265–283.

[3] Hamet P, Tremblay J. Artificial intelligence in medicine. Metabolism, 2017, 69: S36–S40. [doi: 10.1016/j.metabol.2017.01.011]

[4] Chassignol M, Khoroshavin A, Klimova A, Bilyatdinova A. Artificial intelligence trends in education: A narrative overview. Procedia Computer Science, 2018, 136: 16–24. [doi: 10.1016/j.procs.2018.08.233]

[5] Gousios G, Pinzger M, van Deursen A. An exploratory study of the pull-based software development model. In: Proc. of the 36th Int’l Conf. on Software Engineering. Hyderabad: ACM, 2014. 345–355. [doi: 10.1145/2568225.2568260]

[6] Zhu JX, Zhou MH, Mockus A. Effectiveness of code contribution: From patch-based to pull-request-based tools. In: Proc. of the 24th

ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 871–882. [doi: 10.1145/2950290.2950364]

[7] Vasilescu B, Yu Y, Wang HM, Devanbu P, Filkov V. Quality and productivity outcomes relating to continuous integration in GitHub. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 805–816. [doi: 10.1145/2786805.2786850]

[8] Yu Y, Wang HM, Yin G, Wang T. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Information and Software Technology, 2016, 74: 204–218. [doi: 10.1016/j.infsof.2016.01.004]

[9] Nadri R, Rodriguezperez G, Nagappan M. On the relationship between the developer's perceptible race and ethnicity and the evaluation of contributions in OSS. IEEE Trans. on Software Engineering, 2021, 48(8): 2955–2968. [doi: 10.1109/TSE.2021.3073773]

[10] Chacon S, Straub B. Pro Git. 2nd ed., Berkeley: Apress, 2014. [doi: 10.1007/978-1-4842-0076-6]

[11] Gousios G, Zaidman A, Storey MA, van Deursen A. Work practices and challenges in pull-based development: The integrator's perspective. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 358–368. [doi: 10.1109/ICSE.2015.55]

[12] Dabbish L, Stuart C, Tsay J, Herbsleb J. Social coding in GitHub: Transparency and collaboration in an open software repository. In: Proc. of the 2012 ACM Conf. Computer Supported Cooperative Work. Seattle: ACM, 2012. 1277–1286. [doi: 10.1145/2145204.2145396]

[13] Jiang J, Yang Y, He JH, Blanc X, Zhang L. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. Information and Software Technology, 2017, 84: 48–62. [doi: 10.1016/j.infsof.2016.10.006]

[14] Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 356–366. [doi: 10.1145/2568225.2568315]

[15] Tsay J, Dabbish L, Herbsleb J. Let's talk about it: Evaluating contributions through discussion in GitHub. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 144–154. [doi: 10.1145/2635868.2635882]

[16] Ford D, Behroozi M, Serebrenik A, Parnin C. Beyond the code itself: How programmers really look at pull requests. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Society. Montreal: IEEE, 2019. 51–60. [doi: 10.1109/ICSE-SEIS.2019.00014]

[17] Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY. Orthogonal defect classification—A concept for in-process measurements. IEEE Trans. on Software Engineering, 1992, 18(11): 943–956. [doi: 10.1109/32.177364]

[18] El Emam K, Wieczorek I. The repeatability of code defect classifications. In: Proc. of the 9th Int'l Symp. on Software Reliability Engineering. Paderborn: IEEE, 1998. 322–333. [doi: 10.1109/ISSRE.1998.730897]

[19] Mäntylä MV, Lassenius C. What types of defects are really discovered in code reviews? IEEE Trans. on Software Engineering, 2009, 35(3): 430–448. [doi: 10.1109/TSE.2008.71]

[20] Beller M, Bacchelli A, Zaidman A, Juergens E. Modern code reviews in open-source projects: Which problems do they fix? In: Proc. of the 11th Working Conf. on Mining Software Repositories. Hyderabad: ACM, 2014. 202–211. [doi: 10.1145/2597073.2597082]

[21] Panichella S, Zaugg N. An empirical investigation of relevant changes and automation needs in modern code review. Empirical Software Engineering, 2020, 25(6): 4833–4872. [doi: 10.1007/s10664-020-09870-3]

[22] Tan X, Zhou MH, Sun ZY. A first look at good first issues on GitHub. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Virtual Event: ACM, 2020. 398–409. [doi: 10.1145/3368089.3409746]

[23] Jiang J, Lo D, Ma XY, Feng FL, Zhang L. Understanding inactive yet available assignees in GitHub. Information and Software Technology, 2017, 91: 44–55. [doi: 10.1016/j.infsof.2017.06.005]

[24] Kalliamvakou E, Gousios G, Blincoe K, Singer L. The promises and perils of mining GitHub. In: Proc. of the 11th Working Conf. on Mining Software Repositories. Hyderabad: ACM, 2014. 92–101. [doi: 10.1145/2597073.2597074]

[25] Borges H, Hora A, Valente MT. Understanding the factors that impact the popularity of GitHub repositories. In: Proc. of the 2016 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Raleigh: IEEE, 2016. 334–344. [doi: 10.1109/ICSME.2016.31]

[26] Fusch PI, Ness LR. Are we there yet? Data saturation in qualitative research. The Qualitative Report, 2015, 20(9): 1408–1416.

[27] Li ZX, Yu Y, Wang T, Yin G, Li SS, Wang HM. Are you still working on this? An empirical study on pull request abandonment. IEEE Trans. on Software Engineering, 2022, 48(6): 2173–2188. [doi: 10.1109/TSE.2021.3053403]

[28] Yu Y, Yin G, Wang T, Yang C, Wang HM. Determinants of pull-based development in the context of continuous integration. Science China Information Sciences, 2016, 59(8): 080104. [doi: 10.1007/s11432-016-5595-8]

[29] Gousios G, Storey MA, Bacchelli A. Work practices and challenges in pull-based development: The contributor's perspective. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Austin: IEEE, 2016. 285–296. [doi: 10.1145/2884781.2884826]

[30] Wilcoxon F. Individual comparisons by ranking methods. In: Kotz S, Johnson NL, eds. Breakthroughs in Statistics. New York: Springer, 1992. 196–202. [doi: 10.1007/978-1-4612-4380-9_16]

[31] van der Veen E, Gousios G, Zaidman A. Automatically prioritizing pull requests. In: Proc. of the 12th IEEE/ACM Working Conf. on Mining Software Repositories. Florence: IEEE, 2015. 357–361. [doi: 10.1109/MSR.2015.40]

[32] Elazhary O, Storey MA, Ernst N, Zaidman A. Do as i do, not as i say: Do contribution guidelines match the GitHub contribution process? In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Cleveland: IEEE, 2019. 286–290. [doi: 10.1109/ICSME.2019.00043]

[33] Pinto G, Steinmacher I, Gerosa MA. More common than you think: An in-depth study of casual contributors. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Osaka: IEEE, 2016. 112–123. [doi: 10.1109/SANER.2016.68]

[34] Steinmacher I, Pinto G, Wiese IS, Gerosa MA. Almost there: A study on quasi-contributors in open-source software projects. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Gothenburg: IEEE, 2018. 256–266. [doi: 10.1145/3180155.3180208]

李志星(1992－), 男, 博士, 助理研究员, 主要研究领域为群体协同, 开源软件, 实证软件工程.

蔡孟栾(1998－), 男, 硕士生, CCF 学生会员, 主要研究领域为智能化软件工程, 数据挖掘.

余跃(1988－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为实证软件工程, 群体化开发, 开源软件生态.

王怀民(1962－), 男, 博士, 教授, CCF 会士, 主要研究领域为可信计算, 群体智能, 分布式计算.

王涛(1984－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为软件仓库挖掘, 基于群体智能的软件工程, 开源软件生态.