软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3265−3290 [doi: 10.13328/j.cnki.jos.007112]

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

代码审查自动化研究综述*

花子涵[1,2], 杨 立[1], 陆俊逸[1,2], 左 春[3]

[1](中国科学院 软件研究所 集成创新中心, 北京 100190)

[2](中国科学院大学 计算机科学与技术学院, 北京 100049)

[3](中科软科技股份有限公司, 北京 100190)

通信作者: 杨立, E-mail: yangli2017@iscas.ac.cn

摘 要: 随着现代软件规模的不断扩大, 协作开发成为软件开发的主流趋势, 代码审查成为现代化软件开发的重要工作流程. 但由于人工代码评审往往耗费审查者较大精力, 且存在审查者不匹配或审查者水平有限等问题, 人工代码评审的质量和效率难以保证, 且审查后的代码修复也十分费时费力. 因此, 亟需研究人员为代码审查流程进行改进, 提供自动化思路. 对代码审查自动化相关研究进行系统梳理和总结, 并重点介绍4种主要方向: 审查者推荐、代码变更质量评估、审查意见生成和代码自动修复. 整理了相关方向的 148 篇研究, 对每个方向的研究进行技术分类与分析. 随后, 整理了各方向研究任务的评估方法, 并整理出常用的数据集与开源工具. 最后, 对代码审查自动化领域面临的问题进行梳理, 并对未来研究进行展望.

关键词: 代码审查自动化; 审查者推荐; 代码变更质量评估; 审查意见生成; 代码自动修复

中图法分类号: TP311

中文引用格式: 花子涵, 杨立, 陆俊逸, 左春. 代码审查自动化研究综述. 软件学报, 2024, 35(7): 3265−3290. http://www.jos.org.cn/1000-9825/7112.htm

英文引用格式: Hua ZH, Yang L, Lu JY, Zuo C. Survey on Code Review Automation Research. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3265−3290 (in Chinese). http://www.jos.org.cn/1000-9825/7112.htm

Survey on Code Review Automation Research

HUA Zi-Han[1,2], YANG Li[1], LU Jun-Yi[1,2], ZUO Chun[3]

[1](Integrated Innovation Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China)

[3](Sinosoft Co. Ltd., Beijing 100190, China)

Abstract: During software development, collaborative development has become the mainstream trend for large-scale software development, and code review has become an important workflow of it. However, there are some problems in manual code review such as mismatch and knowledge limitations of reviewers, then the quality and efficiency of code review may be poor, and the code repair after review also takes time and effort for developers. It is urgently needed for researchers to improve the code review process and make it automated. This study provides a systematic survey of research related to code review automation, and focuses on 4 main directions: Reviewer recommendation, automated code quality estimation, review comment generation, and automated code refinement. The 148 high-quality publications related to this topic have been collected, and a technical classification and analysis have been carried out in this research field. Then, the evaluation methods of each task in directions are briefly summarized, and some benchmarks and open-source tools are listed. Finally, the key challenges and insights are proposed for future research.

* 基金项目: 中国科学院-东莞科技服务网络计划(202016002000032); 国家重点研发计划(2021YFC3340204); “一带一路”国际科学组织联盟联合研究合作专项计划(ANSO-CR-KP-2022-03)

本文由“面向复杂软件的缺陷检测与修复技术”专题特约编辑张路教授、刘辉教授、姜佳君副研究员、王博博士推荐.

收稿时间: 2023-09-11;修改时间: 2023-10-30; 采用时间:2023-12-14; jos 在线出版时间: 2024-01-05

CNKI 网络首发时间: 2024-03-09

Key words: code review automation; reviewer recommendation; code change quality estimation; review comment generation; automatic code refinement

现代软件工程领域数目众多的大规模软件和复杂系统一般需要多个开发者协作开发, 而代码审查(code review)作为协作开发中的必经流程, 能够有效提高合并入主分支的代码变更的质量. 该流程不仅可以减少软件缺陷, 还可以避免在后续出现更昂贵和复杂的修复工作, 从而节省时间和资源[1]. 然而, 进行代码审查需要耗费开发者大量时间和精力[2,3], 效率低下的同时, 代码审查的质量也难以保障[4,5]. 为了提高代码审查的效率与质量, 许多研究人员提出了自动化代码审查方法, 利用机器简化或替代代码审查的部分步骤. 在过去的 10 多年间, 有关代码审查自动化的研究已经成为软件工程领域的热点问题. 已有许多相关方向的研究工作在多个高水平会议或刊物上发表, 例如 ICSE、ESEC/FSE、ASE、SIGKDD、AAAI 等会议与 TSE、Proceedings of the IEEE 等高质量期刊.

目前已有研究人员对代码审查的部分研究成果进行总结, Çetin 等人[6]整理了 2009–2020 年间自动推荐审查者的 29 篇研究, 但审查者选择只是代码审查整个流程中的一部分, 代码审查的其他阶段并未被该研究考虑在内. Davila 等人[7]总结了现代代码审查的知识体系与辅助代码审查的方式, 但该研究偏向于描述代码审查的工作流程, 并未关注代码审查的自动化. Panichella 等人[8]从开发者角度总结了 2013–2019 年间促进现代代码审查的方法和工具, 但其综述偏向分析代码审查实例数据, 自动化代码审查只是他们总结工作的一部分, 没有被充分、全面地分析. 而且从 2020 年至今, 自动化代码审查的研究又有了很大进展, 并呈现出新的技术特征. 总体上, 目前还缺乏整合了代码审查整个流程的自动化相关研究的系统全面的综述. 鉴于此, 本文针对代码审查整体流程的自动化, 对审查者推荐、代码变更质量评估、审查意见生成与代码自动修复这 4 种面向代码审查的自动化任务进行介绍, 并通过细致分析相关研究, 进行自动化技术方法的总结, 旨在做出以下贡献:

(1) 对代码审查自动化进行系统介绍, 分类代码审查自动化相关任务;

(2) 针对代码审查自动化具体任务, 分析统计其技术方法与评估指标;

(3) 总结该领域常用的代码审查数据集以及开源工具, 协助未来的研究;

(4) 系统梳理代码审查自动化目前面临的问题, 提供未来的研究方向.

本文第 1 节介绍代码审查的背景知识, 对代码审查自动化任务方向进行分类并概述研究现状. 第 2–5 节分别对审查者推荐、代码变更质量评估、审查意见生成与代码自动修复这 4 类代码审查自动化具体任务进行技术分类, 并对相关研究进行分析与点评. 第 6 节介绍上述各个代码审查任务常见的评估方法. 第 7 节对代码审查自动化任务常用的数据集以及开源的代码审查自动化工具进行总结. 最后, 第 8 节根据研究现状总结该领域研究所面临的挑战以及对未来研究的启示.

1 背 景

1.1 研究背景

代码审查起源于 20 世纪 70 年代 Fagan 在美国 IBM 公司提出的代码检查(software inspection)会议[9], 并在随后的数十年中逐步发展为更为非正式的、有工具支持的、异步化的现代代码审查(modern code review)[10]. 目前已被应用于各类开源软件项目和商业软件项目, 成为被广泛认可的软件工程“最佳实践”之一[11]. 现代的代码审查首次由 Cohen 提出, 并随着 Bacchelli 和 Bird 的后续实证研究逐渐被学术界采纳, 它指的是除作者之外的开发人员, 即代码审查者(code reviewer)对代码更改进行评估以识别缺陷和质量问题, 并决定放弃这些更改或将其集成到主项目存储库中的过程[10,12]. 现代代码审查的主要特点是异步且有工具支持, 在开发人员的日常开发流程中定期发生[8].

代码审查的经典流程如图 1 所示, 整体上是开发人员和审查者的周期性交互流程. 一般情况下, 代码审查需要历经开发者对代码进行提交、审查者审查代码并对代码质量做出评判; 对不合格的代码发布评审意见; 最后, 开发者根据反馈进行修改并再次提交的一个循环. 在代码审查的整个循环中, 可以按照执行顺序分为

拉取请求提交、代码审查、代码修复这 3 个阶段.

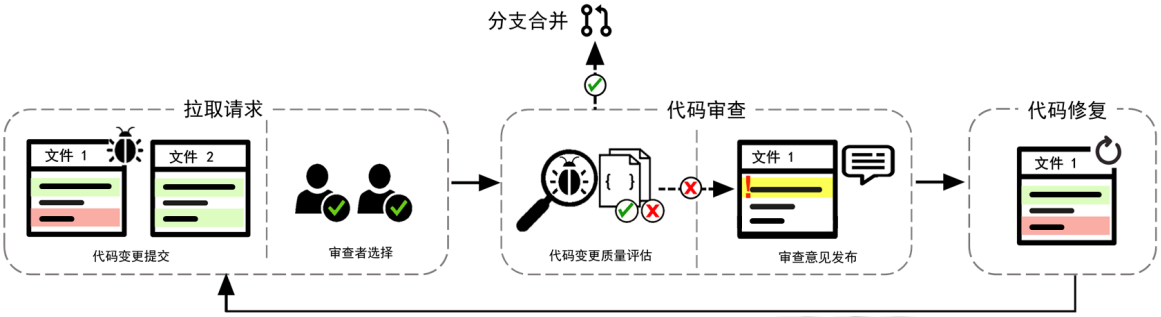

图 1 代码审查流程图

图 2 展示了一个典型的代码评审活动实例, 开发者通过拉取请求(pull request)发布对对代码文件的修改, 在本文中也被称作代码变更[13]. 而 GitHub 等众包协作平台可以自动提供代码变更的差异(diff)信息, 比如添加的代码行前使用符号“+”标记; 反之, 删除的代码行使用标记符“–”. 随后, 平台将通知相关审查者对变更文件进行评估, 若文件存在问题需要修改, 审查者便留下审查意见, 审查意见中通常包含审查者的身份、评论发布时间与评论内容等信息.

图 2 apache/calcite 项目中编号 3212 的代码审查片段

1.2 代码审查自动化

尽管代码审查极大地依赖人工的经验与知识, 但随着相关研究和应用工作的不断演进, 通过自动化技术辅助代码审查的工作取得了不少进展. 图 1 所示的代码审查的每一个环节中, 研究人员都提出了相应的自动化方法.

(1) 拉取请求提交阶段: 其自动化任务目标一般为协助拉取请求的生成, 主要的任务有审查者推荐、拉取请求分类[14]、拉取请求拆分与描述生成[15]、预测请求完成时间[16]等等.

(2) 代码审查阶段: 该阶段首先需对代码变更文件质量进行评估, 如果代码质量合格, 则合并入主分支; 若代码存在问题, 则审查者需要在有缺陷的行组下发表评论作为反馈. 下游的自动化任务包括代码变更质量自动化评估(能否合并分支)、审查意见生成、代码困惑检测(找出使代码审查者感到困惑的因素)[17]、从审查意见中检测人际冲突[18]、审查意见分类[19]、审查意见质量检测[20,21]等.

(3) 代码修复阶段: 该阶段需要开发人员对有问题代码进行修改, 因此, 相关任务为自动化代码修复, 生成质量合格的代码变更.

鉴于篇幅限制, 本文将针对若干自动化任务中的 4 个主要领域——审查者推荐、代码变更质量评估、审查意见生成与代码自动修复进行深入分析. 这 4 种下游任务不仅是代码审查流程中拥有丰富研究历史的关键

任务，且涵盖了代码审查的 3 个阶段，并分别属于推荐、分类与生成 3 种类型，因此具有较强的代表性.

1.3 文献资料的收集和整理

为了系统地总结代码审查自动化研究的主流技术，准确地描述本领域的最新进展，本文收集了自 2009 年至 2023 年 9 月的相关文献并进行梳理. 我们使用如下搜索规则(“Abstract: Code Review” OR “Abstract: Pull Request”) AND (“Abstract: Automatic” OR “Abstract: Automation”)并在 ACM Digital Library、IEEE Xplore、Springer Link、Google Scholar、Web of Science、Science Direct、中国知网等知名论文数据库中进行检索，并筛选出发表时间为 2009 年至今的结果. 在搜索出的 390 篇文献中，我们通过人工对论文标题与摘要部分进行分析，过滤掉无关论文和少于 5 页的短文. 最后，我们使用代码审查论文共享主页(https://naist-se.github.io/code-review/publications)中拥有自动化(automation)标签的论文对结果进行查漏补缺. 最终筛选出与代码审查自动化任务高度相关的研究 148 篇，本文选取 79 篇研究进行详细介绍.

图 3 展示了不同年份的论文发表数量并按照发表刊物所属 CCF 级别进行划分，不难看出：代码审查自动化的研究热度有逐年增加的趋势，尤其近两年增长迅猛，且发布在高质量会议和期刊上的研究数量不断增长，仅 2022 年就有 14 篇相关研究被 CCF A 类会议/期刊接受，这说明代码审查已经成为软件工程领域的热点. 图 4 展示了上述代码审查自动化 4 个方向及其不同自动化技术的研究工作占比，其中，审查者推荐作为早期的代码审查自动化任务，相关研究占比较大，且大部分研究采用相对传统的基于启发式的方法. 而随着深度神经网络技术的发展，代码审查中的代码自动修复等难度较大的任务开始被研究人员关注和探索，并逐渐成为该领域新兴的热门方向，深度学习也逐渐成为代码审查自动化领域目前的主流技术之一.

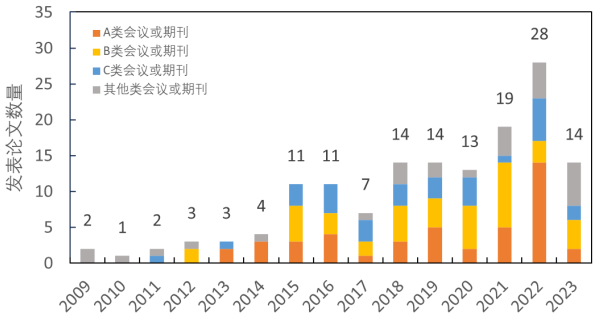

图 3 不同论文发表的年份分布

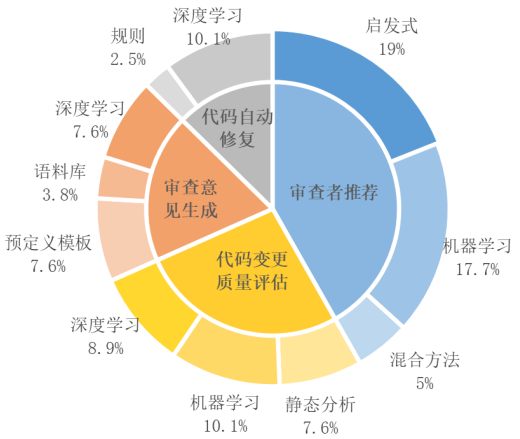

图 4 不同方向与技术研究工作占比

2 审查者推荐

在大型软件项目中，代码变更提交者通常根据代码文件的修订历史记录来选择审查者. 由于一个体量较大的拉取请求可能涉及较多数量的文件更改，那么为其分配合适的审查者至关重要. 代码审查者推荐需要对审查者候选集进行排序并返回排名靠前的数个结果. 针对此任务的研究，所使用的方法可以分为 3 类：基于启发式的方法、基于机器学习的方法和结合多种方法的混合方法.

2.1 基于启发式的方法

启发式(heuristic)的方法一般通过人工定义启发式规则来搜索合适的审查者，这些启发式规则通常利用图结构、审查者的信息、代码行变更历史、代码提交请求中的文本相似性以及文件路径相似性等属性来计算审查者评分，并据此生成推荐集合.

Balachandran 在 2013 年提出的 Review Bot[4]可以基于更改行的编辑历史，为对应的代码编辑者分配分数.

对于新增加的行, 该算法则使用源文件中最近的现有行作为替代. 同时, 对不同的文件类型分配不同的初始分数, 循环处理所有的行后, 根据得分对审查者候选集排序.

部分研究通过构建图来计算评分, 比如: Lee 等人[22]根据基于命名空间的树和代码编辑历史构建了开发者-源代码无向图, 根据开发者对源码的编辑提交次数赋予开发者-源代码边的权重, 计算源代码在命名空间树上的距离作为源代码-源代码边的权重; 随后, 使用随机游走算法输出推荐审查者的集合. Yu 等人[23]认为: 合适的审查者应与开发者有共同兴趣, 而开发者之间的兴趣可以通过拉取请求中提交者和审核者之间的评论互动直接反映. 他们据此构建了评论网络, 将开发者作为图的节点并通过评论发布时间、拉取请求个数等赋予边权重, 根据代码提交者在评论网络中的邻居的距离来推荐审查者. 而 Ying 等人[24]根据计算拉取请求的标题与描述之间的语义相似度计算边的权重. Kong 等人[25]使用了基于项目的评审关系网络, 以在同一公司的不同项目上根据评审次数作为边的权重. 然后, 使用社区检测方法 Louvain 法将网络划分为不同社区, 优先推荐同一社区的评审人. 当该社区的审查者数量不足时, 则基于术语进行推荐: 基于项目统计常用词作为术语, 为每个审查者生成术语的列表. 统计审查者术语列表与项目术语集合的重合项, 以推荐术语重合多的审查者. Sülün 等人[26,27]选择根据软件开发的整个生命周期绘制软件工件-团队人员的可追溯图. 软件工件可能包括需求、设计图、用例图、变更集、源代码文件、代码审查、测试用例、缺陷集、发布版本等. 团队成员则可能包括需求工程师、设计人员、开发人员、评审人员和测试人员. 而他们之间的关系可以包含测试、包含、评审、提交等, 并据此了解团队人员对软件工件的熟悉程度. 根据遍历不同软件工件图表得到开发人员与提交代码文件集合的边集加权之和, 得到该开发人员对此次拉取请求的了解程度得分. 在近期的研究中, Rong 等人[28]在 2022 年提出的 HGRec 基于超图进行推荐, 他们基于拉取请求构建了拉取请求-审查者超图, 因为同一个拉取请求可能有多个审查者参与评审, 普通的图的边无法描述这种关系, 既而用边可以连接任意个节点的超图实现. 根据拉取请求-审查者超图、拉取请求-开发者图、拉取请求-拉取请求相似度图综合计算评分.

而有的研究则通过挖掘审查者的信息, 构建审查者专业知识画像与建模评分函数. Zanjani 等人[29]根据审查者在拉取请求上投入的时间与发布的评价数量计算得分, 其中, 评审的发布还可能受到评论发布时间到最近的工作日的时间间隔影响. Asthana 等人的 WhoDo[30]更进一步, 除了考虑审查者对代码文件的评论发布次数、评审时长和时间间隔之外, 还考虑了这些指标在审查者对整个代码仓库的应用情况, 并且让使用者可以自定义打分函数中每一部分的权重. Mirsaeedi 等人[31]考虑到了软件开发过程中的人员流动, 而代码审查活动总是由小部分核心人员完成, 存在知识流失的风险. 因此, 他们开发的 GitHub 机器人 Sofia 可以有意地将部分代码审查推荐给从事审查活动较少的人员, 因为他们有更多的学习空间, 同时可以将核心人员囤积的大量审查下发, 以缓解他们的评审工作压力. Fejzer 等人[32]则将拉取请求中提到的专业技术整理为集合, 并通过评审经历维护一个审查者画像. 审查者每审查过一个拉取请求, 其包含的专业技术便加入画像, 并构成一个审查者技术集合. 最终, 使用 Tversky 指数计算审查者画像与拉取请求技术集合之间的匹配度作为评分.

通过计算相似度来为审查者打分, 也是一种有效的方法. 比如, Rahman 等人[33]开发的 GitHub 工具 CORRECT 根据拉取请求的共享库和所用的技术计算拉取请求之间的相似度. Jiang 等人[34]的主要思想为, 相似文件路径的文件的功能密切相关. 因此, 采用最长公共前缀 LCP (longest common prefix)指标评估文件路径的相似度, 辅以审查者活跃度、审查者社交关系特征, 通过最基础的向量空间模型(VSM)计算标题词向量的余弦相似度. Ye 等人[35]除了考虑到文件路径和请求标题的相似度, 还将审查者的社交关系、活跃度、工作时间段(通常在星期几活跃)与拉取请求中文件数目作为特征. 在计算标题相似度时, 还考虑了词频-逆文件频率(term frequency-inverse document frequency, TF-IDF)指标, 在多个项目上, 准确率相比 CoreDevRec 有 10%左右的提升.

2.2 基于机器学习的方法

基于机器学习的方法通常利用数据驱动技术进行审查者推荐. 相关研究通过挖掘审查数据的特征, 并使用统计分类模型、多目标优化、协同过滤、隐含狄利克雷分配等各种机器学习技术得到推荐结果.

Jeong 等人[36]最早在 2009 年就提出了用贝叶斯网络预测代码提交的实际审核者. Thongtanunam 等人[37]

也使用贝叶斯网络进行推荐. 他们最早定义了推荐文件路径相似度(file path similarity, FPS)算法, 该算法主要使用最长公共前缀LCP (longest common prefix)计算. 虽然简单却行之有效, 在多个OSS系统中达到了77.97%的准确率. 随后, Thongtanunam等人[38]进一步改进了他们的预测算法, 并将FLOSS和LibreOffice项目添加到他们的评估中. 提出的工具RevFinder使用了LCP以外的3种不同方法来计算文件路径相似性, 分别是最长公共后缀(longest common suffix)、最长公共子序列(longest common substring)和LCSubseq. LCSubseq计算两个文件路径中出现的具有相同相对顺序的路径分量的数量, 它的优点在于, 该技术的公共路径不必是连续的. Thongtanunam等人基于文件路径相似度的工作十分具有指导意义, 大多数研究在选取文件路径作为特征时通常采用FPS算法, 因此, RevFinder也成为业界常用的比较基准. Lipcak等人[39]就在大型数据集上评估了RevFinder和朴素贝叶斯网络, 在GitHub和Gerrit存储库中挖掘数据, 构建了包含51个项目的大型数据集, 包括29.3万个拉取请求、18万个提交者和15.7万个审查者. 他们发现: 两种方法在不同代码仓库上的表现不同, 并且利用额外的子项目细节可以改进推荐结果, 这在基于朴素贝叶斯的方法上结果显著. 2015年, Jiang等人[40]提出的CoreDevRec在挖掘了多个特征并用Spearman等级相关系数验证独立性后, 确定了文件路径、开发人员社交关系、审查者活跃程度这3个特征. 最后, 使用支持向量机(SVM)模型计算每个拉取请求分配给不同审查者的可能性. 而Júnior等人[14]探索了包括Jiang等人[40]应用的3种不同的特征集合, 并且使用了朴素贝叶斯、决策树、随机森林、*k*-近邻、支持向量机等多个不同的分类网络进行组合, 在32个开源项目仓库上实验, 发现不同项目适用的特征属性与分类网络的搭配各不相同.

考虑到评审者的多个特征因素存在互斥性, 部分研究使用进化算法求解. Chouchen等人[41]的研究定义了两个目标: 审稿人专业知识和审稿人工作量. 审查者的专业知识是根据代码更改与审查者以前的贡献之间的相似性来衡量审查者对代码更改的熟悉程度, 而审查者工作量是根据分配给审查者的代码审阅数量来衡量的审查者的忙碌程度. 他们利用多目标进化算法IBEA算法同时优化这两个目标, 并使用变异和交叉算子迭代改进候选解集合, 最后使用指标函数比较和选择解决方案. 与Chouchen等人类似, Al-Zubaidi等人[42]使用了非支配排序遗传算法NSGA-II, 但是他们设定的两个优化目标为: (1) 参与评审的机会, 主要根据审查者对代码变更的经验、项目的活跃度以及过去与提交者的合作来计算; (2) 工作量分布的偏度, 通过分配给每个审查者的开放审阅数量之间的差异来衡量. 同样, Rebai等人[43]也使用NSGA-II优化3个目标: 在Chouchen等人的审查者专业知识和审查者工作量指标的基础上加入了合作历史, 由开发者和审查者之前的合作审阅次数计算得出. 胡渊喆等人[44]则将响应时间作为约束条件, 推荐可以在约定时间内进行评审的审查者.

部分研究选择使用协同过滤进行推荐. Xia等人[45]认为: 大多数研究利用历史回顾数据中的各种数据, 并尝试提取显著特征以构建审稿人的专业知识模型. 然而, 这可能导致推荐模型忽略隐含的关系. 比如: 一个开发者没有审查过某一请求, 并不意味着该开发者不能胜任该请求, 他可能多次读取或调用提交代码相关的代码文件. 因此, 他们提出了基于协同过滤的推荐算法TIE, TIE使用潜在语义模型将审查者与请求映射到向量空间, 通过向量内积来得出不同请求对应不同审查者的值. Strand等人[46]结合了协同过滤和基于上下文的过滤, 以解决数据稀疏和冷启动问题. 比如: 在当新文件添加到代码存储库时, 可能导致发生冷启动问题, 即无法将新文件直接链接到潜在的审查者. 此外, 文件和审查者之间的交互矩阵是很稀疏的, 因为并非所有文件都被所有审查者修改或审阅. Strand等人利用了LightFM算法计算新文件的表示, 有效地解决了上述问题. 2020年, Chueshev等人[47]还是运用了这个经典的机器学习推荐算法构建了审查者评审矩阵和开发者提交矩阵, 同时使用交替最小二乘法(ALS)构建潜在语义空间, 并使用最相似的父级文件(通过最长公共子序列)替换新文件来解决冷启动问题.

这个任务也可以使用隐含狄利克雷分配(latent Dirichlet allocation)进行主题建模的方法, 这是一种在自然语言处理领域中用于主题建模的概率模型. LDA可以发现文本集合中隐藏的主题结构, 将文本数据表示为不同主题的混合. Kim等人[48]采用使用Apache Lucene 6.3.0处理自然语言文本, 采用空格分词器和多个过滤器处理源代码, 比如词语分隔符过滤器、停用词过滤器等. 他们用处理后的代码提交数据训练了一个主题分布模型, 接着统计审查者在数个主题上的代码审阅历史, 生成审查者的审阅主题向量, 最后计算新的请求代码

的在主题模型上的分布向量与归一化后的审阅主题向量的相似度作为推荐评分. 同样, Liao 等人[49]也使用LDA 挖掘主题, 并根据审查者的评审历史构建主题-审查者矩阵, 在推荐过程中, 计算拉取请求的主题分布向量与不同审查者在审查者-主题矩阵值之积进行打分.

2.3 混合方法

采用混合方法研究一般结合了两类方法进行审查者推荐.

Yang 等人[50]结合了启发式方法与机器学习方法构建了一个两层审稿人推荐模型 Revrec, 由启发式推荐算法和分类器组成: 第 1 层的启发式方法对所有审查者数据进行粗排, 选取前 10 名候选人进入第 2 层进一步排序; 第 2 层的得分决定审查人员在技术或管理方面参与审查过程的程度. 其使用了启发式方法中经典的代码文件相似度与文件路径相似度以及审查者对代码文件的评审历史这 3 项进行加权求和, 得到评估分数. 在第 2 层, 他们使用了 SVM 分类器, 并且使用了 3 个特征: 专业知识、技术评审数量、管理评审数量. 其中, 技术评审与管理评审是对评审评论是否包含代码的分类.

最后, 他们的实验表明, 混合方法的表现优于 Thongtanunam 等人[38]的 RevFinder.

与 Yang 等人不同, Xia 等人[51]并没有将两种模型使用在推荐的不同阶段, 而是分别训练了两种模型, 并在最终的推荐阶段进行模型融合, 加权了两个模型的输出分数, 并得到最终的评分结果. 他们选取基于相似度的启发式模型和基于朴素贝叶斯多项式网络的文本挖掘模型, 并且在有新的评审结果时更新模型. 而相比于其他机器学习方法, 朴素贝叶斯多项式分类器是基于概率的, 不需要在大量数据就上学习特征表示, 便于模型的更新. 类似地, Pandya 等人[52]也利用了基于相似度的模型和基于 SVM 的分类模型的加权融合, 并在最后阶段, 根据审查者的活跃度对模型结果进行过滤, 移除近 1 年都没有评审历史的候选者. 二者的方法都优于RevFinder, 其中, Xia 等人的工作比其之前的 TIE[45]在 top-1 指标上提升了 15.3%.

近年来, 随着深度学习的发展, 开始有研究逐渐转向利用神经网络模型的嵌入表示, 以替代传统机器学习方法挖掘的特征表示. Liu 等人[53]结合了基于图的启发式方法和基于深度学习的方法, 提出了一种新颖的注意力邻居嵌入传播方法(ANEP), 来解决数据稀疏问题. 他们首先构建了审查者为节点的审查者联系图, 审阅过相同代码的审查者可以被视为邻居, 并采用 Transformer 对代码和审查者进行嵌入表示. Transformer 模型是一种基于多头自注意力的序列到序列模型, 它可以有效地捕获长距离依赖(long-range dependencies). 审查者的表示可以由其审阅过的代码历史与其邻居聚合表示: 他们首先将代码序列拼接, 利用 Transformer 的注意力机制赋予重要的代码更改更高的权重, 学习到一个嵌入表示; 接着, 在审查者联系图中依次寻找 1 跳至 k 跳内的邻居, 得到每一层邻居的 Transformer 嵌入表示, 最后将整个表示集合进行加权聚合; 最后, 根据审查者嵌入表示与修改代码嵌入表示的相似度进行审查者推荐. 该方法首次将大型深度学习模型应用在该任务上, 为这个传统的推荐任务提供了新的方向.

2.4 小 结

现有的审查者推荐的方法都有各自的优点和不足, 具体见表 1.

可以看出:

- 基于启发式的方法通常需要预定义部分权重, 侧重于细粒度特征, 例如编辑历史、技术名词和发布时间等. 该方法虽然简单高效, 无须训练, 但受限于手工规则, 建模能力有限, 仅能够关注显式交互关系, 难以捕捉潜在的关系和处理复杂模式.
- 相反, 基于机器学习的方法更注重挖掘隐含的特征与关系, 例如通过主题建模生成的主题等. 然而, 尽管机器学习方法可以捕捉更复杂的特征, 但对数据质量要求较高, 需要解决数据稀疏等问题.
- 混合方法结合了人工经验知识与数据隐式特征, 融合了不同方法的优点, 通常表现优于单一方法[54], 但是模型复杂度较高.

尽管不同算法可能侧重不同特征, 文本相似性和文件路径相似性是被广泛采用的重要特征, 而且实验证明, 它们受训练数据的来源和大小影响较小[55,56]. 然而, 除了特征的选择, 该任务还需要解决推荐领域通常需

要面对的冷启动问题, 即如何在引入新文件时为其分配合适的审查者, 这也是代码审查者推荐任务中需要重点关注的问题之一.

表 1 审查者推荐方法对比

方法类型	技术选型	优点	缺点	相关文献
启发式	基于更改行编辑者	规则简单易懂, 易于实现	关注特征单一, 难以解决复杂的代码变更情况	[4]
	基于图关系	考虑多维度的交互信息, 如社交因素等	推荐结果受限于历史数据, 可拓展性差	[22−28]
	基于审查者专业知识画像	考虑审查者的多种行为和属性, 支持个性化推荐	过于依赖评审历史, 需要维护审查者画像更新	[29−32]
	基于相似度	计算简单高效, 易于实现	需要预定义关注特征难以捕捉隐含关系	[33−35]
机器学习	统计分类模型	考虑隐含特征, 具有较好的可解释性	需要领域知识和经验进行特征工程和模型选择依赖高质量的数据	[14,36−40]
	进化算法	综合多维度特征, 考虑审查者工作量、响应时间等效率因素	需要大量计算资源进行搜索, 依赖数据较多	[41−44]
	协同过滤	考虑隐含关系	存在交互矩阵稀疏、冷启动问题, 可扩展性差	[45−47]
	主题建模	考虑潜在主题, 发现代码之间的关联性	需要选择主题数需要大量计算资源建模	[48,49]
混合方法	启发式与机器学习	结合了两种方法的优点	模型复杂性高、训练时间长	[50−52]
	启发式与深度学习	学习了代码序列嵌入表示, 对代码内容理解度高, 适用性广泛	模型复杂性高、训练时间长, 需要大量训练数据与超参数调整, 模型可解释性差	[53]

3 代码变更质量评估

代码变更质量的自动化评判可以辅助审查者做决断, 甚至自动合并代码, 减少开放代码评审的数量. 对代码变更质量的评估, 本质上是代码审查人员对提交的代码是否合并进入主分支的判断, 即代码变更是否需要发布评审意见通知修改. 据此, 此任务属于一个二元的分类任务.

3.1 基于静态分析的方法

基于静态分析的方法一般根据人工制定的规则检验或者与代码库匹配来验证代码是否合格. 该方法通常使用静态分析工具对代码进行检测, 或者自行构建代码集进行问题代码的检索匹配.

大部分研究通过预定义规则, 通过词法分析、语法分析等检查代码, 并将规则集成为静态分析工具, 自动化地输出分析结果. 2009 年, Denney 等人[57]首次在该方向进行研究, 并实现了一款自动认证插件, 它支持输入一系列代码安全要求, 根据一个定义好的模板生成验证条件, 并自动验证代码是否满足这些要求. 其约束条件主要是基于数学规则, 实现对变量名称、函数的参数个数判断等模板化任务, 无法实现对代码的语义的分析. Balachandran[4]在 2013 年提出的工具 Review Bot 可以基于静态分析方法对代码更改进行检查并标记. Review Bot 集成了 3 个基于 Java 语言的静态分析工具: Checkstyle (http://checkstyle.sourceforge.net)、PMD (http://pmd.sourceforge.net)和 FindBugs (http://findbugs.sourceforge.net), 来实现对代码的正确性与规范性的检查, 其中, PMD 还可以使用 Java 或 XPath 编写新规则扩展. Mehrpour 等人[58]构建了静态分析工具 SAT, 它集合了 12 种静态分析工具, 包括代码坏味检测、内存泄漏检测、代码克隆检测等, 可以覆盖代码的规范性、安全性和质量检测. 同时, 他们的研究也表明: 静态分析工具无法识别代码中自然语言的问题(如注释、标识符等), 这部分的质量需要人工进行再次审核确认. Bosu 等人[59]的研究则建立在大规模的数据上, 他们分析了来自 10 个开源项目的 26 万多个代码审查请求, 并整理了 413 种易受攻击的代码漏洞. 这些规则可以被用于自动化漏洞检测, 以协助后续的研究. Hanam 等人[60]认为需要关注代码变更带来的语义变更, 而这些可能体现在更改区域之外的部分. 他们基于 AST 之间的差异定义了适用于 JavaScript 语言的 4 种语义变化影响, 并集

成为工具 SEMCIA.

部分研究采用基于模板匹配的分析. Zhang 等人[61]的研究提供了一个定制化的插件, 通过将通用模板与代码库进行匹配, 发现类似的更改并检测是否存在潜在的错误. 模板支持用户自定义, 包括更改代码的上下文或排除部分语句. 将更改代码片段转化为代码抽象语法树(abstract syntax tree, AST)后, 该工具便利用树搜索, 在整个代码库的 AST 中进行匹配. 该工具可以为审查者检测出代码更改在逻辑上的疏漏, 但是严重依赖代码仓库中的代码片段重复度, 因此不适合分析创新型代码提交. 类似的, Sharma 等人[62]则利用专业程序员支持论坛(例如 StackOverflow.com)提供的内容来识别给定代码片段的潜在缺陷. 由于代码论坛上一般都是对有缺陷代码的问题求助, 因此可以通过匹配代码片段是否出现在论坛中来识别其是否存在缺陷. 该工具从 StackOverflow 中挑选一组包含相似源码片段的讨论帖子, 相似度检测使用文档指纹技术的开源工具 MOSS (https://theory.stanford.edu/~aiken/moss/), 其被广泛用于检测代码抄袭. 最终在验证集上达到了超过 90%的准确率.

3.2 基于机器学习的方法

基于机器学习进行代码变更质量评估的相关研究一般使用传统的机器学习算法挖掘与提取拉取请求的相关特征, 包括代码、文本、审查者与项目的特征, 并通过分类算法进行二分类. 大部分研究选择基于机器学习中的经典分类模型, 如随机森林、贝叶斯网络等, 构建拉取请求分类器.

早在 2009 年, Jeong 等人[36]就基于对 Firefox 和 Mozilla core 代码仓库的评审活动分析, 发现超过半数的评审处于开放状态, 过多的代码审查请求堆积可能导致对开发进程阻塞. 因此, 他们整理了代码中关键词数量、代码更改相关的缺陷的发布时长、缺陷级别等特征, 并基于此训练了贝叶斯网络(Bayesian network)用于预测. 但是由于其将拉取请求的代码更改局限于修复缺陷, 无法评估完成其他任务(如添加新功能或者版本迭代)的拉取请求, 因此存在一定局限性. 而 Gousios 等人[63]的研究则分析了来自 GitHub 的 291 个项目, 提取了来自拉取请求、项目和审查者这 3 个方向的 15 个特征, 采用随机森林进行分类. 并评估了各个特征的有效性, 并发现其中最主要的特征是拉取请求是否修改了最近修改的代码. Fan 等人[64]受上述研究的启发, 构建了一个合并代码变更预测工具. 其首先从代码更改中提取 34 个特征, 这些特征分为 5 个维度: 代码、文件历史记录、所有者体验、协作网络和文本; 然后, 利用随机森林等机器学习技术来构建预测模型. 相比 Jeong 等人与 Gousios 等人的工作, 在准确率等指标上有提升, 但是性能方面存在不足. Jiang 等人[65]提出了 CTCPPre, 提取了代码特征、文本特征、贡献者特征和项目特征, 并将这些特征传递给 XGBoost 分类器, 分别计算拉取请求的接受概率和拒绝概率: 如果接受概率高于拒绝概率, 则预测拉取请求被接受; 否则, 预测拉取请求被拒绝. CTCPPre 不仅返回预测结果, 还提供拉取请求的接受概率和拒绝概率, 这可以帮助审查者做决策. 实验结果表明, CTCPPre 在 28 个开源项目上达到了 82%的准确率.

还有一部分工作聚焦于代码特征提取方面.

- Ochodek 等人[66]提出的拉取请求分类方法可以支持任何语言的代码. 该方法需要准备相关训练数据, 并在存在问题的代码行前进行标注; 随后, 该方法可以提取代码词级、行级和块级的特征, 包括使用字符级和 n 元的词袋模型以及部分预定义的特征. 由于训练样本通常较小, 该方法选择了基于决策树进行实现, 包括 CART、C5.0 决策树和随机森林. 由于其挖掘的特征达到了 120 项, 并在训练集上的准确率超过 99%, 因此可能存在过拟合的问题.
- Soltanifar 等人[67]测试了不同的特征和模型的组合, 他们使用了 3 种不同的机器学习算法: 逻辑回归、朴素贝叶斯和贝叶斯网络, 构建了 3 个模型. 而特征基本都从提交的代码和评审流程挖掘而来, 主要有代码行数、评审评论数量、修复的漏洞的优先级等(共 11 个特征). 最后, 贝叶斯网络在多个指标上都优于其余两个模型.
- 还有研究采用 n-gram 语言模型对比一个代码提交与整个代码文件的相似性, 据此分析代码风格是否一致. 如: Hellendoorn 等人[68]使用大量 Java 代码训练了一个 5-gram 的代码语言模型, 相比于神经网络技术, n-gram 语言模型的训练更为简便且具有一定的可解释性.

3.3 基于深度学习的方法

近年来, 随着深度学习模型的发展, 部分研究开始使用深度学习模型进行代码语义学习. 此类方法使用深度学习模型将代码映射到高维的向量空间, 学习代码的嵌入表示. 与机器学习提取到的特征相比, 这种表示通常包含更丰富的语义信息, 比如上下文等相关信息.

基于此方法的研究开始于南京大学 Lamda 实验室的 Shi 等人, 其研究成果于 2019 年在 AAAI 上发表. Shi 等人[69]认为: 需要评审的源代码与审核通过的修改后版本存在大量的相同部分, 为了判定提交的代码是否合格, 就不应该与以往的软件挖掘研究一样学习二者的相似性. 因此, 她首次提出了需要关注两个版本的代码文件之间的差异, 这种差异也就是代码经过修改的部分, 并认为需要对差异进行建模, 以学习修改特征. 由于该研究聚焦于代码的修改, 以及为了减少大量雷同的输入信息, 其选用的代码输入为修改行. 但这种操作也存在一定的问题: 首先, 修改行一般体量很小, 仅为几行代码, 这对于模型的学习是不够的; 其次, 同样的代码段在不同语境下的正确性也不同. 这样, 代码的上下文信息也需要被关注. 该文提出了模型 DACE, 为了较好地处理上下文信息, 该模型采用了一个上下文增强层处理输入, 它只选取了方法内的修改行, 将经过 word2vec 词嵌入后的方法整体作为输入, 经过 CNN (convolutional neural network)进行降维后, 输入进 LSTM (long short term memory)捕获周边代码对修改部分的影响, 按序列输入也可以有效维护代码的顺序属性. 最后输出的行数仅为修改行的行数, 但其已经包含了上下文信息和序列特性. 而修改特征学习层采用基于 RNN (recurrent neural network)的编码器-解码器结构, 但是在编码阶段, 如果简单地把一组数据进行拼接, 那么就可能浪费顺序语句的结构信息, 且存在忽略代码修改的版本顺序的问题. 因此, 需要体现出修改块内的语句结构和块间的变化方向, 就需要对两个版本的代码文件分别编码再融合. 因此, 编码器采用了一个自定义的可以接受成对输入的编码器结构. 最后, 模型通过一个全连接层输出代码通过审查的概率. 作为将深度学习应用到该任务上的开山之作, DACE 的模型设计为后续的研究提供了指导.

随后, Li 与 Shi 的研究团队又基于多实例学习的方法提出了模型 DeepReview[70], 实现一次拉取请求提交是否需要被审查的判定. 与传统的模型训练不同, 模型不是接收一组单独标记的实例, 而是学习一组带标记的被称为“包”的一组实例. 在本研究中, 修改块的二元组对应一个实例, 而包含多个修改块的一次拉取请求提交就是“包”. DeepReview 使用了拉取请求提交时的文本描述信息来增强输入, 并采用不同的 CNN 对代码输入和文本输入分别处理. 处理后的三元组{旧代码,新代码,文本描述}在融合层合为一个向量表示, 即一个实例, 而一次拉取请求中的所有实例最后输入全连接网络和最大池化层生成预测. 相比于单个代码变更的实例预测, 基于“包”的预测在有任意实例为负样本时, 整个“包”也应被模型分类为负. 因此, 多实例学习较符合真实场景下的拉取请求实例.

在 Li 等人的研究基础上, Wu 等人[71]利用抽象语法树抽象出语法特征, 提出了一种利用简化 AST 的代码质量评估方法 SimAST-GCN. 该模型使用了简化后的 AST 作为输入, 并生成相应的邻接矩阵和节点序列; 其次, SimAST-GCN 获取节点序列的词嵌入, 并使用双向门控循环单元(Bi-GRU)对语句的自然性进行建模, 得到一个隐藏层表示; 接着, 采用图卷积网络(GCN)来融合节点关系图表示和隐藏状态; 最后, 应用基于检索的注意力机制来导出代码的最终表示, 并计算修改代码与初始代码表示的差值, 经过最后一层的 Softmax 层进行分类. 与 Shi 等人的 DACE 相比, 拥有更好的性能, 不仅在多个评估指标上更优, 且耗费更少的训练时长.

尽管应用了长短期记忆神经网络或者通过 AST 来抽象代码特征, 代码元素之间的长距离依赖问题仍限制了模型能够处理的文本长度. Hellendoorn 等人[72]使用 Transformer 架构解决此问题, 并指出, 现有的研究存在数据集小且缺乏真实性的问题: 由于拉取请求中只有少部分会收到评审意见, 这将导致数据集的严重不平衡性; 而大多数工作通过数据清洗, 人为构造带有评审意见的平衡数据集, 可能导致训练结果在实践中严重失效. 此外, 他们将修改前与修改后的代码进行了对齐操作, 并发现无法对齐的差异块达到惊人的 23%, 这代表作者在评审后未在对应位置作出任何改动, 而这部分数据往往被其他研究排除在外. 为了扩大数据规模, Hellendoorn 等人在 GitHub 上爬取了 Java 和 Python 语言项目的 17.8 万余条拉取请求, 并尽可能减少人为的过滤限制, 将语料库扩充到 1Tb. 其模型采用了小型的 Transformer, 输入一个拉取请求中的所有差异块(最多 16

个), 并采用自注意力机制管理如此大的输入. 最后, 对每一个差异块进行是否合格的概率预测. 其结果表明: 在真实场景下的大规模数据集上, 数据的随机性可能对模型造成很大的影响, 基于监督学习的神经网络模型还有很大的提升空间.

随着大规模预训练模型迅猛发展, 许多研究开始选择使用代码变更数据集训练大模型, 并设计与代码审查高度相关的预训练任务, 使模型可以泛化到代码审查自动化相关的任务上.

文献[13]提出的模型 CodeReviewer 采用了近年来较热门的 Transformer 框架 T5, 并构建了 4 种预训练任务来服务于代码审查相关的自动化任务. 它收集了包括 Java、Python 等热门语言和 Php、Ruby 等稀有语言的大量拉取请求构建多语言数据集. 在预训练阶段, 文献[13]设计了差异块中修改行前标签("+"或者"–")预测、差异块遮蔽预测、评审内容遮蔽预测、差异块自然语言评审生成这 4 种任务. 与明确给出输入块标签不同, 这种预训练方法可以让不同任务目标相互辅佐, 训练模型解决多类下游任务. Sawant 等人[73]也在基于 Transformer 的框架上用大规模的数据集训练了 Policy2Code, 由于缺乏现有的代码评审相关的大型数据集, 他们选择了一些近似的数据集进行训练与评测, 并采用额外的训练策略弥补: 在 GitHub 的代码评论与代码修复数据集上进行训练, 并使用常见弱点枚举(common weaknesses enumeration)数据集与编码最佳实践(coding best practice)数据集进行评估. 他们采取了 Doc2BP 对代码评论数据分类, 筛选出符合代码最佳实践建议的评论. Policy2Code 训练阶段采用了多任务预训练与预微调的策略, Sawant 等人设计了段落来源判断(Doc)与代码-注释匹配任务(CC)作为预训练阶段任务, 并使用了代码审查中涉及代码修复的数据集进行微调. 由于该数据集体量较小, 因此适合在预训练后、具备一定代码理解能力的模型上进行迁移学习, 可以适用于代码评审中代码质量评估任务. 这类预训练模型虽然没有精准服务于特定任务, 在某些特定任务上的性能可能不如针对该任务训练的模型, 但是其具备了强大的代码变更及相关信息的语义理解能力, 在广泛应用和多任务环境中的实用性仍然很高.

3.4 小 结

现有的代码变更质量评估的方法都有各自的优点与不足, 具体见表 2.

表 2 代码变更质量评估方法对比

方法类型	技术选型	优点	缺点	相关文献
静态分析	基于预定义规则	支持自定义规则, 维护代码一致性	需要人工知识设计规则, 缺乏代码语义理解, 无法理解代码中的自然语言	[4,57–60]
	基于模板匹配	支持自定义模板, 简单高效	需要人工构建模板库, 依赖代码片段重复度与模板库质量	[61,62]
机器学习	统计分类模型	考虑拉取请求特征, 具有较好的可解释性	需要领域知识和经验进行特征工程和模型选择, 存在过拟合风险	[36,63–67]
	统计语言模型	维护代码一致性, 具有较好的可解释性	仅包含浅层语义信息, 如代码风格信息	[68]
深度学习	基于监督学习的神经网络模型	理解代码语义信息, 关注代码差异部分	数据集不平衡、难以构造, 仅建模代码内容特征	[69–72]
	预训练大模型	对输入代码限制较小, 理解代码语义信息, 泛化性强、支持多任务	模型复杂性高、训练时间长, 需要大量训练数据, 需要人工设计预训练任务, 模型可解释性差	[13,73]

针对此任务的早期研究一般都采用基于静态分析的方法, 该方法需要较高的人工成本, 但是具有较强的可解释性与可编辑性, 可以自定义其中的规则与模板, 且无须训练, 时间效率最高. 基于机器学习的方法与基于神经网络的方法整体模型运算类似黑盒, 可解释性较差, 且需要耗费较长时间与算力进行模型训练. 二者都着重与代码表征的学习, 相比之下, 基于神经网络的方法可以学习到更多的语义信息, 但是机器学习可以融入拉取请求的其他特征. 基于神经网络的方法早期应用的模型较为简单, 因此对数据的过滤以及处理更为复杂; 而后续基于大型的预训练模型的研究则通过大规模数据的预训练挖掘代码语义, 弱化了对单任务数据的操作. 此外, 由于该任务数据集普遍存在数据不平衡的问题, 这可能导致模型的过拟合或者模型结果的评估偏差, 这也是此任务下数据驱动方法需重视的因素之一.

4 审查意见生成

代码审查意见自动生成即针对拉取请求提交的代码, 由模型给出类似人工审查意见的自然语言评审. 模型接收代码形式输入, 生成自然语言的输出. 针对这个问题, 早期的研究主要是输出已有的评审语句, 输出预定义的模板或者从语料库中选择最为相关的评论进行推荐. 后续出现的直接生成评审意见的方法则将此任务建模为自然语言处理(natural language processing)领域的机器翻译问题.

4.1 基于预定义模板的方法

基于预定义模板的方法一般通过人工定义好的提示语句或者某些固定模板将代码变更与预定义或挖掘出的规则进行匹配, 并输出对应的预定义的模板语句.

预定义的规则一般由静态分析器集成, 大部分相关研究选择采用现有的静态分析器. Palvannan 等人[74]基于 Python 的静态分析器 Black 开发了可以在 GitHub 上的自助评审机器人 SUGGESTION BOT, 该工具将 Black 的输出格式化为评审语句形式的代码修改建议. 该工具已经被使用在真实开发流程中, 并在 GitHub 上得到了较高的评分. Balachandran[4]开发的 Review Bot 支持使用多个静态分析器对代码进行分析, 并通过解析器整合其输出, 并支持自选静态分析器. 但其需满足: (1) 可以在子进程中执行; (2) 输出为一个文件, 可以被输出解析器读取; (3) 输出应结构明确; (4) 报告的每个源代码问题都应包含源代码的行号与一条警告消息. 其实现思路较为简单, 仅仅是整合了不同的静态分析器的输出. 部分研究选择自行制定规则.

- Denney 等人[57]开发的插件可以根据预定义的规则检验代码, 并据此生成自然语言报告. 其报告内容包含代码与相匹配的验证条件的超链接, 并使用自然语言的文本模板将其描述为认证要求. 此外, 它支持用户自定义文本模板, 这种高结构化的信息可以辅助代码评审工作.
- Klinik 等人[75]为学生作业的评审开发了自动评审工具 Personal Prof, 其使用 Rascal 语言编码了作业的相关要求与规范(如类名、类之间的关系等). Rascal 具有强大的模式匹配功能, 可以对源码的各种数据类型与关系进行匹配, 包括访问修饰符(public, protected, private)和类层次结构等. 在 Personal Prof 执行期间, 该工具可以自动报出不匹配规则的异常消息作为评审的结果.

也有研究通过数据挖掘方法挖掘出关联规则作为验证规则. Chatley 等人[76]开发了基于 GitHub 的评审生成工具 Diggit, 可对代码的历史提交信息进行分析: 首先, 使用 Apriori 算法从给定的代码库的版本控制历史记录中挖掘代码文件之间的关联规则, 并依据对历史提交的统计分析, 对开发者可能忽略的部分进行提醒, 提醒开发者对关联代码文件的更改; 此外, Diggit 还可以分析代码质量趋势, 对例如代码复杂度、代码文件大小变化等改动等进行消息提醒.

4.2 基于语料库的方法

基于语料库的方法通常涉及构建存储{代码,评审}对的语料库, 并查找与语料库中相似的代码片段对应的评审语句, 通过计算代码与评审语句的相关性来进行评审语句的推荐. 在对代码文本与评审文本的相关性进行建模时, 一般分别处理代码与评审内容, 对其采用不同的编码方式或者标签化方法. 模型一般使用的训练数据也为{代码,评审}的二元组, 而一次拉取请求中产生的多个版本的代码可以处理为多条数据样本.

Gupta 等人[77]首次提出对代码文本与评审文本的相关性进行建模, 并使用神经网络进行人工代码审查意见的表示. 与 DACE[69]相同, Gupta 等人通过评审员评审的代码部位将代码分为上行、当前行和下行, 也就是需要修改的代码块与它的上下文, 并且通过 LSTM 抽取上下文信息. 数据处理方面, 代码部分使用标词器 Lexer8 进行标记; 而对于评审部分, 由于评审块带有链接以及审查者的名称以及“@”评注等信息, 与 Twitter 的推文格式较为相似, 因此该文使用了处理推文的标词器处理评审内容, 最后对两部分分别进行 word2vec 嵌入; 最后, 使用 4 种 LSTM 处理 3 个部分代码与评审块. 他们通过不同代码与评审组合训练了一个相关性评分函数, 为了生成负样本数据集, 他们不仅选取了其他代码块与评审内容组对, 还将代码修改后的版本与评审内容生成额外的负样本, 让模型不将评论应用于其他代码与修改后的代码块. 在生成阶段, 依据输入代码块与语料库中代码的相似度进行粗筛, 进而计算相关性评分给出推荐最符合的数条评审. 类似的, Siow 等人[78]

对数据进行了两个维度的嵌入: 单词维度(word-level)与字符维度(character-level). 在单词维度, 对评审内容里面的与项目特有词语进行符号化(比如函数名、哈希值、数值等), 对代码内容使用典型的解析器 Pygments 进行分词(tokenize). 字符维度的编码是为了解决 OOV (out of vocabulary)问题,因此使用了独热编码. 模型上选择双向 LSTM 进行两个维度的词嵌入并对向量进行融合, 并在此基础上考虑注意力机制. 注意力机制可以为序列中更为相关的词素(token)赋予更高的权重取值, 以进一步缓解长序列训练中存在的噪声输入问题. 最后, 合并基于注意力机制分别输出的代码向量和评审向量, 用于计算相关性评分.

Hong 等人[79]发现: 训练模型需耗费大量的时间与算力, 大模型的训练与验证时长可以达到 38 h 之多.因此, 他们认为轻量级的方法可能便于嵌入进代码审查的工作流程中, 进而提出了直接相似度匹配的方法 CommentFinder. CommentFinder 使用词袋模型处理代码, 并与代码库中带有审查意见的代码进行余弦相似度匹配, 筛选出排名前 10 的代码作为候选集; 接着, 基于源代码文本, 使用 Gestalt 模式匹配(GPM)方法重新评估相似度, 得到最终的排序结果. 该方法的最大特点是不需要进行训练, 但是构建带有评论的代码库也是该方法需要重点考虑的工作.

4.3 基于深度学习的方法

基于深度学习的方法通常根据输入的代码序列直接生成评审意见, 此方法将代码评审自动生成问题建模为文本到文本(即代码文本到自然语言文本)的神经机器翻译(neural machine translation, NMT)问题, 因此需要搜集大量代码与自然语言对应的语料库, 并训练出正确的字词映射, 甚至语法结构映射. 但是与一般的翻译问题不同, 代码评审并不是简单使用自然语义描述代码语义, 而需要发现代码中存在的问题并给出修改建议.

基于代码到审查意见的神经机器翻译一般对两个版本的代码之间的差异性进行建模, 一般采用基于编码器-解码器结构(encode-decoder structure), 有时又被称为序列到序列模型(sequence to sequence model). 具体来说: 编码器的作用是将源代码编码为固定长度的向量表示; 解码器的作用是将源代码的向量表示进行解码, 并生成评审意见. 不同的编码器-解码器的主要区别在于代码的输入形式和神经网络的结构, 其使用的数据一般为{旧代码,新代码,评审}的三元组. 此外, 在这一部分使用的代码审查内容基本都需要经过过滤, 删除掉与代码修改建议无关的评论, 例如赞赏(“做的不错!”)、疑问(“为什么这样做?”)等. 这部分数据的过滤一般采取人工设计类别并标注小规模的样本, 训练分类模型进行分类.

Moshkin 等人[80]从简单的语言模型开始, 为 Python 语言开发了自动评审的工具. Moshkin 等人使用了 FastText 模型对源代码进行训练, 并使用基于 Transformer 架构的预训练神经网络语言模型生成自然语言评审, 并通过寻找相似的代码拉取请求生成推荐的代码块修改建议.

文献[13,81,82]均自行设计预训练任务构建大模型, 且均使用了 Text-to-Text Transfer Transformer (T5)框架. T5 框架代表着“预训练目标、结构性、未标记数据集、转化方法以及许多语言理解任务”等特性[83]. 其中, 文献[81]的模型 AUGER 仅针对审查意见生成任务, 而文献[13,82]均使用模型解决多个下游任务. 在数据处理上, T5 框架仅需要对数据进行分词. 文献[13,81]则使用了带“+”与“–”的原始差异块表示. 此外, AUGER 还引入了方法体的 AST 与对评审位置进行描述的“评审行标记(review tag)”, 并将其转化为词嵌入融入模型训练过程中, 从而进一步精化了数据集训练和生成任务, 因而取得了一定的性能提升. 文献[82]同样使用了方法粒度的输入. 在语料库方面, 文献[81,82]仅仅考虑针对 Java 语言的评审生成, 而文献[13]构建了一个基于 9 种语言的多语言数据集. T5 框架作为文本到文本 Transformer 探索迁移学习的极限, 往往需要通过多个预训练任务来理解一种语言: 文献[81]采用交叉预训练的方式对代码与评审文本进行掩码预测; 文献[13]设计了 3 种遮蔽预测和一个将代码转化为审查意见生成任务; 文献[82]单独构建了预训练数据集, 通过遮蔽预测任务让模型学习 Java 和技术英语(technical English)去理解这两种语言.

最近, Zhou 等人[84]与 Lin 等人[85]提出了直接微调代码预训练模型(code pre-training model)来实现此任务, 且均使用了 CodeT5 模型与 Tufano 等人的研究进行对比. 代码预训练模型如 CodeT5, CodeBERT 等均使用代码语言-自然语言对进行预训练, 以学习到代码的语义信息. 其中, GraphCodeBERT 将代码的 AST 作为输入, 并设计了 AST 序列相关的预训练任务, 以学习到代码的结构化信息. 二者的研究都表明: 不需要对此类模型

进行调整, 也可以得到与 Tufano 等人类似的表现, 甚至在某些数据集上有所提升. Lin 等人在分析了不同模型的表现后发现, 不同模型的输出结果是可以互补的, 继而提出了模型融合的思想.

4.4 小 结

现有的审查意见生成方法都有各自的优点和不足, 具体见表 3.

表 3 审查意见生成方法对比

方法类型	技术选型	优点	缺点	相关文献
预定义模板	预定义规则	标记问题代码, 维护代码一致性, 简单易懂, 支持自定义规则	需要人工预定义规则与模板语句, 缺乏针对性、灵活性	[4,57,74,75]
	数据挖掘规则	根据项目定制规则	需要人工预定义模板语句, 缺乏灵活性	[76]
语料库	神经网络模型表示	理解代码语义信息	需要构建高质量语料库, 缺乏灵活性	[77,78]
	词袋模型表示	无须训练, 简单高效	需要人工选择相似度算法组合, 需要构建高质量语料库, 缺乏灵活性	[79]
深度学习	预训练大模型	利用评审信息, 理解代码语义信息, 泛化性强	模型复杂性高、训练时间长, 需要大量训练数据, 需要人工设计预训练任务, 模型可解释性差	[13,80−82]
	微调代码预训练模型	无须设计模型与预训练任务, 模型基础强大	模型复杂、可解释性差	[84,85]

基于预定义模板的方法利用现有静态分析工具或自行设定的固定规则进行匹配, 并输出对应条目的提示语句. 然而, 该方法往往难以具备人工评审所具有的经验性与复杂性, 如评审人员往往可以发现某种代码不适用的特殊条件, 因此该方法存在一定的局限性. 基于语料库进行评审语句推荐的方法尽管使用了人工评审意见, 但是却强依赖于构建的语料库, 缺乏灵活变通能力. 基于深度学习直接生成的方法具有较强的鲁棒性, 并实现了真正意义上的审查意见生成, 取得了显著的进步, 但却仍然受制于训练数据集的大小与质量. 同时, 神经翻译模型的效果仍需要进一步提高. 另外, 模型的训练对机器硬件配置要求较高, 时间成本较大, 在效率上不如前两种方法. 特别地, 大型语言模型和代码专用大型语言模型在审查意见生成应用中的研究仍然处于起始阶段: 一方面, 通过增加训练数据量和复杂度, 可以进一步提高模型性能; 另一方面, 这类方法也带来了资源消耗大、参数调整复杂度高等问题. 因此, 如何利用低秩自适应、模型蒸馏等方法高效地对模型参数进行微调, 以及在不明显降低模型性能的前提下, 通过“上下文学习”等方式实现的“零样本”与“少样本”的实践应用, 是一个值得关注的领域. 由于代码审查意见可以是多元的, 面对相同的代码变更, 从不同方向提出代码审查意见都具有一定意义, 因此, 此任务的挑战还在于生成结果的有效性难以准确评估.

5 代码自动修复

代码审查过程中的代码自动修复问题与代码缺陷修复问题存在较大的相似性, 其主要区别在于:

- 代码缺陷聚焦于缺陷的修复, 即修复程序代码中存在的某种破坏正常运行能力, 甚至可能导致被攻击的部分.
- 代码审查所审查的代码通常是经过开发者自检、正常条件下可以运行的代码, 但可能在某些边缘条件下存在疏漏; 或者是违反编码规范等设计准则, 如代码风格不统一等代码坏味(code smell)问题[86].

因此, 代码审查是比代码缺陷检测更进一步的、提高可编译代码可读性、可维护性和设计质量的过程. 代码缺陷修复通常使用通用代码缺陷数据集, 而本文定义的代码自动修复任务是完全针对代码审查流程的, 相关实验均需构建特定于代码审查过程中的代码变更数据集. 不仅如此, 传统的缺陷自动修复往往需要进行缺陷定位, 而在面向代码审查的自动修复中, 代码修复一般发生在人工评审后, 即审查者已经标出了存在缺陷的代码段. 特别地, 部分面向代码审查的代码修复会引入评审意见数据, 审查意见可以提供代码修复的提示描述与推荐修改方案. 因此, 这两个任务通常是分开讨论的.

5.1 基于规则的方法

基于规则进行修复的方法一般针对代码风格的修复, 仅对代码中与格式相关的空白与缩进规范性进行检测, 具有较强的局限性. 同时, 此类方法也可以抽象出代码的编码规范, 具有较强的可解释性与可拓展性.

2014 年, Allamanis 等人[87]提出的工具 NATURALIZE 不仅可以给出代码样本是否与项目代码风格一致性的判断, 还可以生成代码修复片段. 其首先通过词法分析器将代码片段中的格式(如空格)进行标记, 再通过语法分析器转化为 AST, 最后利用编写好的语法解析器将语言模型输出的 AST 的格式还原. 它可以输出一个推荐代码序列, 并根据评分函数进行排名. NATURALIZE 在 GitHub 的 5 个流行开源项目上提交了 18 个补丁, 其中 12 个被接受.

Markovtsev 等人[88]则挖掘项目的编码风格, 并将其转化为编码规则. 开发的工具 STYLE-Analyzer 可以自动挖掘规则并给出修复结果. 该工具首先将代码的格式转化为原子级别的字符(如空格、换行等); 接着, 将格式字符使用独热编码生成代码的表示序列, 并使用无监督的随机森林算法进行训练. 随机森林算法包含了多个决策树分类器, 可以有效避免过拟合问题; 并且基于决策树的方法是依据规则的, 是具有可解释性的, 这与因不符合编码规范而产生的代码格式问题相匹配. 在生成修复的代码片段后, STYLE-Analyzer 还对修复前后代码片段的 AST 进行对比, 以保证产生的修复仅对格式进行了更改, 而不涉及代码本身的改动.

5.2 基于深度学习的方法

与审查意见生成任务类似, 在代码自动修复任务中, 由于基于深度学习的方法可以直接生成代码序列, 此任务可以被建模为代码到代码的神经翻译问题. 部分研究在输入中进一步加入了代码审查意见信息, 使用{代码,审查意见}的二元组作为输入.

2021 年, Tufano 等人[89]首次使用深度学习模型进行面向代码审查的代码修复任务, 并构建了相关的代码变更修复数据集. 在该研究中, Tufano 等人设计了两种实验: 一是直接输入源代码进行代码修复, 二是输入源代码与审查意见内容进行代码生成. 两种实验不仅对应于是否拥有审查意见这两种应用场景, 还可用于对比分析审查意见的效用. 研究方法上, 根据过程可分为 3 个模块: 数据挖掘、数据处理、模型搭建与训练. 他们在数据准备阶段做了大量的工作, 挖掘了 2 566 个 GitHub 与 6 388 个 Gerrit 平台的 Java 项目. 在处理代码数据方面, Tufano 等人认为: 经过代码抽象预处理, 再使用序列到序列模型生成代码时表现更优秀; 且代码抽象可以在有限的词典内生成代码表示, 可以起到降噪、减少输入大小的作用. 因此, 他们使用了代码缺陷修复任务中常用的代码抽象方法[90]: 构建所有标识符(变量名、方法名、链接等)的映射, 并将所有代码文件中相同变量名做相同映射, 最后生成的代码也需要经过该“字典”翻译. 此外, 经过过滤非 ascii 字符等处理, 最后将代码输入限制在 100 个字符数内. 最后, 符合要求的代码方法实例达到了 1.7 万余例. 接着, 在模型搭建方面, 两项实验对应着两种编码器-解码器模型: 直接输入源代码对应着一个编码器与一个解码器; 输入源代码与审查意见内容的实验则采用两个编码器与一个解码器. 二者都选择了包含多头注意力机制和前馈层的 Transformer 模型. 最后的实验结果也表明: 使用审查意见文本数据的代码生成有着更好的表现, 这肯定了代码审查意见对于后续自动修复任务的正向作用.

随后, Tufano 等人在 2022 年对先前的研究进行改进[82], 与 2021 年的研究进行对比, 其与先前研究的不同重点体现在使用了更大规模的数据与带有预训练任务的 T5 模型框架. Tufano 等人表示, 代码抽象工作存在限制: 其根据源代码构建的映射使得修复代码若出现新的变量则无法找到对应映射. 因此, 先前的工作只适用于非常简单的代码修改实例. 在新的工作中, 他们仅使用简单的分词操作处理代码, 并挖掘了更多项目, 将有效实例数扩大到 16.8 万多例, 将输入大小扩展到 512 个字符. 为了减轻噪声的影响, 其引入了庞大的预训练数据集, 训练模型学习 Java 语言与审查意见中常见的技术性英语. 在学习率的选择上, 他们也通过对照实验确定在下游任务中综合表现最好的 ST-LR 函数(slanted triangular learning rate). 相比于先前的研究[89], TufanoT5 模型不仅可以生成新的变量, 并且可以多生成 15%左右的完全正确的代码修复片段.

和文献[82]一样, 文献[13]也选择了 T5 模型框架, 并构建了规模更为庞大的多语言数据集. 其中, 带有审

查意见的实例多达 431 万余条. 其模型 CodeReviewer 可以支持包括 Ruby、Php 和 Go 等语言在内的 9 种语言.

同时, Thongtanunam 等人[91]在 TufanoTransformer[89]的基础上, 将其对代码抽象的预处理方式改为字节对编码(byte-pair-encoding), 以解决其无法处理新的字符的问题. 字节对编码可以进行动态词汇扩展, 动态地将词汇进行拆分和合并, 从而在处理文本时, 可以根据上下文来创建新的词汇, 可以很好地平衡数据压缩粒度且不存在 OOV 问题. 实验结果表明: 使用 BPE 作为预处理方式的 AutoTransform 模型, 可以在完美预测率指标上提升近 3 倍.

Pornprasit 等人[92]认为需要重点关注代码的差异块区域, 在先前研究的构建提交代码版本与最终被接受代码二元组的基础上引入代码的初始版本, 构成三元组进行训练. 该工作针对经过 BPE 编码后的代码词素(token)序列, 并在代码初始版本与修改版本的修改区域前后插入令牌级的特殊标记, 使神经翻译模型能够更好地关注更改方法中代码变更区域的词素. 相比于先前的 TufanoT5[82], 该方法在完美预测率指标上可以提升至少 62%, 比 AutoTransform 高至少 274%.

近期, 越来越多的代码预训练大模型被提出, 这些模型帮助研究者节省了大规模预训练的工作.

Zhou 等人[84]与 Lin 等人[85]使用并对比了代码预训练模型 CodeT5、CodeBERT 与 GraphCodeBERT. 尽管没有为该任务单独构建模型, 或者对模型进行重新训练, 这些大型预训练模型在相同的数据集上进行微调后, 仍可以取得与 TufanoT5[91]相似的结果, 在某些项目数据集上甚至表现更优. Yin 等人[93]则在使用代码预训练模型基础上加入了程序依赖图(program dependence graph)信息, 进一步学习代码的结构信息. 程序依赖图比 AST 包含更为丰富的信息, 它的有向边表示了不同节点之间的数据依赖与控制依赖关系; 并采用 PDG2Seq 算法将图转化为序列作为输入, 同代码文本经过不同的编码器, 结合 CodeT5 进行代码序列生成. PDG2Seq 算法是一种将程序依赖图转换为唯一的图序列编码的算法, 它可以无损地保留程序的结构信息和语义信息. 该算法的基本思想是: 利用最小深度优先搜索编码的规则, 从一个边开始扩展子图, 按照字典序的大小关系选择最小的编码序列作为图的表示. 该算法可以有效地将图结构转换为序列形式, 方便后续的深度学习模型进行特征提取和分析. Yin 等人的消融实验表明: 图序列信息的加入, 可使模型在 *BLEU* 指标上提升 1 个左右百分点. 这表明: 在基于“文本-文本”范式的预训练模型上融入代码结构信息的处理, 对于代码审查任务是有效的.

5.3 小 结

现有的代码自动修复论文方法都有各自的优点和不足, 具体见表 4.

表 4 代码自动修复方法对比

方法类型	技术选型	优点	缺点	相关文献
规则	预定义规则	维护代码风格一致性, 简单易懂	需要人工预定义规则, 仅针对代码风格修复, 局限性强	[87]
	数据挖掘规则	维护代码风格一致性, 根据项目定制规则, 可解释性强	仅针对代码风格修复, 局限性强	[88]
深度学习	预训练大模型	丰富的预训练任务、针对性较强, 理解代码语义信息	模型复杂性高、训练时间长, 需要大量数据准备工作, 需要人工设计预训练任务, 模型可解释性差	[13,82,89,91,92]
	微调代码预训练模型	无须设计模型与预训练, 模型基础强大	单任务上效果欠佳, 模型复杂, 可解释性差	[84,85,93]

综合来看, 基于规则的修复方法仅针对代码风格问题进行修复, 但优点在于具有强大的可解释性, 以及针对项目实施不同的编码规范, 但是不能解决代码中的逻辑或语义错误. 而基于深度学习的生成方法虽然没有功能上的限制, 但模型的重点在于深度理解代码语义以及不同版本代码变更之间的差异. 然而, 代码坏味带来的代码修复数据可能在一定程度上成为学习代码语义差异的噪声. 当下, 由于大型预训练模型在深度理解代码语义方面表现出色, 基于该方法的创新成为这个任务的研究热点. 但是大模型需要消耗一定时间和资源, 同时也依赖大量的高质量训练数据, 这是一个不容忽视的挑战之一. 未来的研究需要解决在如何有限的数据和资源下训练出更高效模型的问题, 以便更好地适应实际场景下的软件开发需求.

6 代码审查自动化任务的评估方法

在基于代码审查自动化的 4 个任务中, 不同任务对应着不同的评估方法. 但审查者推荐、代码变更质量评估任务都有着丰富的先例可以借鉴, 而代码审查自动生成任务与代码修复任务两类生成任务的评估方式在目前看来仍是一个需探索的开放性问题.

6.1 审查者推荐的评估方法

审查者推荐任务返回最有可能的 K 个审查者候选集, 其中, K 值一般取 1, 3, 5[7], 相关研究常用的评估指标有在 R 个推荐集合中计算前 K 个推荐结果的准确率(top-K accuracy)、平均精确率(mean *Precision@K*)、平均召回率(mean *Recall@K*)和平均 *F*1 值(mean *F-Score@K*).

- 准确率统计推荐集合里实际审查者在长为 K 的推荐审查列表中的比例, !!代表若存在则返回 1, 反之为 0.

$$\text{Top\_}K\ Accuracy=\frac{1}{|R|}\sum_{r\in R}!!recommendations_{actual\cap made} \tag{1}$$

- 精确率是指推荐的 Top K 个审查者中实际审查者所占的比例.

$$\text{Mean}\ Precision@k(r)=\frac{1}{|R|}\sum_{r\in R}\frac{recommendations_{actual\cap made}}{recommendations_{made}} \tag{2}$$

- 召回率是指推荐的 Top K 个审查者中的实际审查者占所有应当被推荐的审查者的比例.

$$\text{Mean}\ Recall@k(r)=\frac{1}{|R|}\sum_{r\in R}\frac{recommendations_{actual\cap made}}{recommendations_{actual}} \tag{3}$$

- 平均 *F*1 值是根据精确率和召回率计算得到, 计算方法如下.

$$\text{Mean}\ F\_Score@k(r)=2\times\frac{\text{Mean}\ Precision@k(r)\times\text{Mean}\ Recall@k(r)}{\text{Mean}\ Precision@k(r)+\text{Mean}\ Recall@k(r)} \tag{4}$$

部分研究还使用了平均倒数排名(*MRR*)来评估正确的审查者在推荐结果中的排名, 计算公式如下.

$$MRR=\frac{1}{|R|}\sum_{r\in R}\frac{1}{rank(r)} \tag{5}$$

其中, 如果实际审查者不在推荐列表中, 则排名 *rank*(*r*)返回正无穷. 这样, *MRR* 会趋近于 0:

6.2 代码变更质量评估的评估方法

基于代码审查的代码质量评估模型本质是一个二分类器, 但特别的是: 由于往往只有很少数的拉取请求被拒绝, 数据集存在严重的不平衡性. 因此, 有关研究一般采用精确率和召回率的调和值, 即基于 *F*1 值的评价体系. 部分研究也使用 *AUC* 值与 *MCC* 值等二分类模型评估指标.

AUC (area under the receiver operating characteristic)指的是 ROC (receiver operating characteristic)曲线下的面积. ROC 曲线用来衡量当分类阈值变化时, 分类器系统的判别能力, 且 ROC 曲线对样本是否平衡不敏感, 适用于样本类别不均衡时对分类器性能的评价. 比如: Hellendoorn 等人[72]的研究是输入拉取请求中的一组差异块, 因此其更关注高精度/低召回率的区域, 确保对拉取请求整体预测的准确性. 分类器的 *AUC* 等于在随机选择一个正例和一个负例的情况下, 分类器将正例的得分高于负例的概率. 因此, *AUC* 值越接近 1, 代表分类器性能越好.

MCC (matthews correlation coefficient)是一种用于度量分类器性能的统计指标, 它考虑了分类器的真正例(*TP*)、真负例(*TN*)、假正例(*FP*)和假负例(*FN*), 其中, *TP* 代表判断结果中有多少是判断正确的正例, *TN* 是判断错误的正例, *FP* 是判断正确的反例, *FN* 是判断错误的反例. MCC 提供了对分类性能的综合评估, 尤其适用于样本类别不平衡的情况, 计算公式如下.

$$MCC=\frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{6}$$

6.3 审查意见生成的评估方法

对自动生成的审查意见进行评估极具挑战, 因为代码审查内容本身就存在极大的复杂性, 不同的评审人员对同样的提交有着不同方面的见解. 目前的研究往往采取两种方式结合: 第 1 种评估方式是请有经验的开发人员进行人工评估; 第 2 种评估方式是使用来自机器翻译领域的评测指标来辅助评估.

在人工评估时, 研究者通常会邀请经验丰富的开发人员(通常要求具有至少 5 年的编程经验)阅读拉取请求内容与生成的代码审查意见, 一般采用打分或分类的方式进行评估. 在打分的方式中, 评估志愿者需要从信息量(评审评论内容给出建议具体程度)和相关性(评审评论与代码部分的相关程度)这两个维度评估, 给出 1–5 的评分. 在分类的方式中, Tufano 等人[89]将审查意见有效性分为 3 类: 语义上相同、其他的解法、不相关或错误. 由于同一段代码的修改方向也是多样的, 如果评审评论可以提出与基线不同的具体建议, 那么也可以看作一个成功的案例.

在自动评估的方法中, 评测指标主要来自机器翻译和文本总结等研究领域, 可以评估候选文本和参考文本的相似度[94]. 常用的有 *BLEU* (bilingual evaluation understudy)、*ROUGE* (recall-oriented understudy for gisting evaluation)指标. *BLEU* 指标[95]的思想是比较候选文本和参考文本里 *n* 元词组(*n*-gram)的重合程度. *BLEU* 可表示为 *n*-gram 加权和与惩罚项 *BP* 的乘积, *c* 代表模型输出文本的长度, *r* 表示参考文本的长度.

$$BLEU = BP \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{7}$$

$$BP = \begin{cases} 1, & c \geqslant r \\ \mathrm{e}^{1-\frac{r}{c}}, & c < r \end{cases} \tag{8}$$

BLEU 指标更偏重查准率, 而忽略了查全率. 所以整体上来看, *BLEU* 评测指标更偏向于评估较短的审查文本; 而 *ROGUE*[96]指标面向的是查全率, 特别是 *ROUGE-L* 指标, 针对的是最长公共子序列的重合率, 因此偏向处理较长文本. 除了上述两种通用评估方法, Lin 等人[85]还提出了编辑进度(edit progress)的评估方法, 通过计算公共序列, 来得到生成结果与参考文本之间需要修改的步数.

6.4 代码自动修复的评估方法

针对代码自动修复任务常见的评估指标为完美预测率(perfect prediction), 表示模型生成的代码文本与参考代码参考完全相同的百分比. 还有上文所述的 *BLEU* 与 *ROUGE* 指标. 特别地, 有研究认为传统的用于评价自然语言的指标忽略了代码的语法和语义特征, 并不适合评估代码, 因而使用传统指标的一些变体如 *CodeBLEU* 来针对代码序列评估[82]. 其不仅比较生成代码与参考代码 *n* 元语法的相似度, 还考虑到抽象语法树 AST 和数据流的相似性. 在公式(9)中, $BLEU_{\text{weight}}$ 表示加权计算的 *n*-gram 匹配; $Match_{\text{ast}}$ 表示 AST 匹配; $Match_{\text{df}}$ 表示语义数据流匹配:

$$CodeBLEU=\alpha\times BLEU+\beta\times BLEU_{\text{weight}}+\gamma\times Match_{\text{ast}}+\delta\times Match_{\text{df}} \tag{9}$$

此外, 部分研究也考虑了正确预测在生成结果中的排名, 采用平均倒数排名(*MRR*)作为评估指标.

7 代码审查自动化数据集和开源工具总结

7.1 代码审查自动化常用数据集

尽管代码审查自动化有多种下游任务, 但是其本质都是对代码审查工作流程的数据进行挖掘与学习. 因此, 相关研究的数据基本都来源于 GitHub、Gerrit 等众包协作平台的开源项目仓库. 研究者一般通过这些大型开源平台提供的接口, 爬取开源项目的拉取请求的相关信息, 在预处理后自行构建数据集. 部分研究还提供开源项目仓库数据自动爬取工具, 如: Heumüller 等人[97]提供的工具 ETCR 可以自动爬取项目拉取请求, 并构建成关系型数据库, 协助后续研究获取数据并自定义处理分析. 表 5 列出了近年来代码审查自动化相关研究构造的常用数据集及其数据属性与共享网址.

表 5 代码审查自动化常用数据集

来源文献	包含信息	编程语言	数据量	年份	共享网址
[98]	评审意见、代码修复	多语言	144k	2018	https://crop-repo.github.io/
[89]	代码修复	Java	630k	2019	https://sites.google.com/view/learning-codechanges/data
[33]	评审意见、代码修复、审查者信息	多语言	82k	2020	https://zenodo.org/record/3678551#.YxW70C5BwQ8
[97]	评审意见、代码修复	Java	231k	2021	https://zenodo.org/record/5079076#.YxWsYC5BwQ8
[99]	代码修复	多语言	98k	2021	https://zenodo.org/record/7029359#.YxG7ey5BwQ8
[82]	评审意见、代码修复	Java	383k	2022	https://zenodo.org/record/5387856#.YxVG_i5BwQ9
[13]	评审意见、代码修复	多语言	7 933k	2022	https://zenodo.org/record/6900648#.YxF2ES5BwQ9
[91]	代码修复	Java	147k	2022	https://github.com/awsm-research/AutoTransform-Replication
[100]	评审意见、代码修复	多语言	10k	2022	https://zenodo.org/record/5832080#.YxVKiy5BwQ8
[52]	评审意见、审查者信息	多语言	27k	2022	https://figshare.com/articles/dataset/CORMS_DATASET_A_GitHub_and_Gerrit_based_Hybrid_Code_Reviewer_Recommendation_Approach_for_Modern_Code_Review/20493042
[28]	评审意见、审查者信息	多语言	87k	2022	https://figshare.com/articles/dataset/Open_Data/19199981/1

7.2 开源代码审查自动化工具

工具是软件工程领域相关研究的落地实现, 同时也是验证算法有效性的重要支撑, 可以为其他研究人员学习其方法以及对比新技术提供方便. 在工业界, 已经有部分帮助代码审查的工具问世, 并被投入实际生产开发流程, 部分研究工作也被成功转化为自动化工具. 表6列出了代码审查自动化领域的部分知名工具, 包括其发表时间、适用的任务以及访问链接. 本文使用 RR (reviewer recommendation)、QE (quality estimation)、RCG (review comment generation)和 CR (code refinement)分别代表审查者推荐、代码质量自动化评估、审查意见生成与代码自动修复任务. 得益于众包协作平台的发展, 工具逐渐从早期的独立集成演化为轻量级的插件, 更便于嵌入工作流程, 部分工具也支持用户自定义推荐规则、评审规则. 特别地, WasmEdge 提供的机器人模板支持用户输入 OpenAI 的接口密钥来调用目前最先进的通用大模型 GPT4 协助代码评审自动化.

表 6 开源代码审查自动化工具

工具名称	年份	支持功能	下载地址
Veracode	2006	QE、CR	https://www.veracode.com/
JArchitect	2009	QE	https://www.jarchitect.com/
Rhodecode	2010	RR	https://rhodecode.com/
Codacy	2012	QE、CR	https://www.codacy.com/
Sonar	2013	QE、RCG	http://www.sonarsource.org/
CodeScene	2015	QE、RCG	https://codescene.com/
Getty	2017	QE、CR	https://github.com/thegetty
CoderFactor.io	2018	RR、RCG	https://www.codefactor.io/
CFar	2018	QE	https://github.com/human-se/cfar-survey
Diggit	2018	RCG	https://github.com/lawrencejones/diggit
Embold	2019	QE、CR	https://embold.io/
STYLE-analyzer	2019	CR	https://freesoft.dev/program/139746338
Sofia	2020	RR	https://github.com/apps/sofiarec
Visual Expert	2021	QE、CR	https://www.visual-expert.com/
RSTrace+	2021	RR	https://github.com/facebookarchive/mention-bot#algorithm
Code Review Bot	2022	QE、RCG	https://github.com/codereviewbot/code-review-bot
WasmEdge	2023	QE、RCG、CR	https://github.com/flows-network/github-pr-summary
SUGGESTION BOT	2023	RE、RCG	https://github.com/suggestionsbot

8 总结与展望

针对代码审查过程的自动化不仅有助于项目合作人员对代码的审核和评审, 还提高了开发人员对代码质量检验与提升的效率. 这可以极大地缩短合作项目的开发时间, 并且保证每一次更改的规范性与效用. 在软

件工程领域, 大量的实证研究已经肯定了代码审查的作用[10], 但代码审查流程自动化的研究仍是一个崭新的方向, 针对代码审查自动化任务的挖掘还在不断地发展. 本文针对代码审查自动化进行了系统性综述, 并对相关研究中使用的技术方法进行总结, 以推动研究者对该方向更深入的研究. 本文将代码审查自动化分为 3 个阶段: 拉取请求提交阶段、代码审查阶段与代码修复阶段, 并分别对其中的 4 类具体任务——审查者推荐、代码变更质量评估、审查意见生成与代码自动修复的相关研究进行分析与点评, 最后对其使用的评估方法进行整理. 尽管对代码审查的相关研究已经取得一定高质量的成果, 但是仍有可以改进的方向, 值得研究人员进一步关注.

(1) 统一且高质量数据集的搭建

目前, 代码审查相关的研究使用的数据集基本均为自行爬取, 且来源项目、使用语言、数据量大小多有不同. 但是真实环境下的评审相关的数据有效性却难以保证, 且存在项目间的差异. 比如在审查者推荐任务中, 完全根据历史数据进行审查者推荐并不一定是理想的[101]. 在基于 Gerrit 的评审调查中, 74%的非推荐评审人认为他们可以轻松地审核其未参与的拉取请求[102]. 同时, 在代码评审中可能存在人际关系冲突[18], 由此产生的审查意见可能是完全没有意义的. 同时, 有研究表明, 自动化工具在专有项目上的表现普遍比开源项目要差[56], 这也与目前研究使用的数据可得性有关. 使用统一且高质量的数据集不仅可以减轻研究人员前期的数据准备工作并去除部分噪声, 还可以方便研究过程中使用目前主流的评测指标进行评估, 从而避免造成数据的不同对研究结论有效性的影响. 通过搭建统一的多语言、多项目的大型公开实验数据库, 有助于研究人员不断补充优化实验数据. 同时, 方便研究人员在实验评估阶段, 将新提出的方法与已有基准方法进行更公平和全面的对比.

(2) 融合与辅助其他软件工程任务

尽管目前针对代码审查及其相关自动化的研究仍然相对有限, 然而在软件工程领域, 仍有一些其他的研究方向, 有着与代码审查自动化相关任务交叉应用的潜力. 代码审查研究中, 针对代码差异的对比学习的思想可以应用于版本更替自动化的相关研究, 比如代码发布说明(release note)生成任务等. 而代码变更质量评估与代码自动修复任务都需要对代码风格与代码缺陷进行检查与自动修复, 现有部分研究便采取了整合多个代码静态检查工具的策略进行实现. 同时, 代码审查中的代码变更也可以为代码坏味修复与研究基础广泛的代码缺陷自动修复[103,104]相关研究带来新的数据来源与借鉴.

(3) 代码审查自动化任务评估的维度统一与类人工自动化评估

目前, 针对代码审查自动化的研究在具体任务上仍存在评估方法复杂、维度不统一的问题. 由于代码审查自动化任务最终需要融入实际开发流程中以实现其更大价值, 因此, 收集真实环境下的用户反馈、实际测试和广泛讨论是该任务不可忽视的重点. 部分面向代码审查自动化工具的研究工作采取真实开发环境下的使用反馈作为方法有效性的评估, 比如统计工具给出建议被接受的条目、收集用户问卷等, 而大部分面向方法创新的研究则通常关注测试集上自动评估指标的数值表现, 仅有小部分研究考虑了时间效率[84,85,91], 对模型训练与运行时间进行针对性的评估. 特别地, 审查意见与代码自动修复这两类生成任务的生成结果存在多义性、语义复杂等特点, 因而此类任务需引入人工评估. 但由于人工评估的代价较大, 且存在着评判标准模糊、评价指标不统一等问题, 现有的研究人工评估样本数据量小, 且存在人工筛选样本的可能, 整体缺乏说服力, 无法全面反映模型的整体质量. 当前, ChatGPT 作为与人类高度对齐的一个模型, 为客观准确评估代码审查自动化任务提供了一个新的思路, 目前已有利用类 GPT3.5 模型来辅助人工评估[105]的工作. 本文认为: 通过引入 ChatGPT 等大模型技术进行类人工自动化评估, 是未来值得深入研究和探索的方向之一.

(4) 针对众包协作平台的工具开发

由于代码审查的行为与软件开发工作流程密切相关, 且主要发生在众包协作平台, 因此要落实研究成果, 就需要将其转化为实用性的工具. 因此, 一种可行的方式是利用众包平台官方提供 API 提供自动化交互插件, 在开发人员创建拉取请求时推荐审查者, 提交代码时进行自动评估, 如存在问题, 则给出相应的解决方案; 在审查阶段, 生成提示信息辅助审查人员生成代码审查; 最后, 根据评审内容, 为开发者提供相应的代码修

复解决方案. 目前, 部分相关研究提供自行构建的插件工具, 展示研究成果的使用方案, 如 Mirsaeedi 等人的审查者推荐机器人 Sofia[31]、Chatley 等人的评审生成工具 Diggit[76]等. 但是大多数研究往往只专注于一到两个任务, 目前还没有一个可以涵盖整个代码评审流程的自动化工具. 因此, 未来的自动化工具可以集合多个单任务研究, 实现一款适用于多种任务的、全方面辅助代码审查流程的实用工具. 同时, 自动化工具的开发也代表着相关研究需要关注模型的应用效率与实用性. 工具返回结果的时间耗费需要保证在用户感知的范围内(不到 1 s)[55]. 如果工具生成的是推荐列表, 那么也需要在候选集的大小与准确率之间做好权衡, 避免给用户带来过多的干扰[102].

我们希望以上提出的研究发展方向可以为相关研究人员提供新的思路, 对未来的研究具有指导意义. 最终提高代码审查流程的效率, 方便大型项目的协作开发与更新维护工作.

References:

[1] Czerwonka J, Greiler M, Tilford J. Code reviews do not find bugs—How the current code review best practice slows us down. In: Proc. of the 37th ACM/IEEE Int'l Conf. on Software Engineering. 2015. 27−28.

[2] Bosu A, Greiler M, Bird C. Characteristics of useful code reviews: An empirical study at Microsoft. In: Proc. of the 12th Working Conf. on Mining Software Repositories. 2015. 146−156.

[3] Kononenko O, Baysal O, Godfrey MW. Code review quality: How developers see it. In: Proc. of the 38th ACM/IEEE Int'l Conf. on Software Engineering. 2016. 1028−1038.

[4] Balachandran V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proc. of the 35th ACM/IEEE Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 931−940.

[5] Morales R, McIntosh S, Khomh F. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. 2015. 171−180.

[6] Çetin HA, Doğan E, Tüzün E. A review of code reviewer recommendation studies: Challenges and future directions. Science of Computer Programming, 2021, 208: 102652.

[7] Davila N, Nunes I. A systematic literature review and taxonomy of modern code review. Journal of Systems and Software, 2021, 177: 110951.

[8] 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.

[9] Michael F. A History of Software Inspections. Berlin, Heidelberg: Springer, 2002. 562−573.

[10] Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A. Modern code review: A case study at google. In: Proc. of the 40th Int'l Conf. on Software Engineering: Software Engineering in Practice. New York: ACM, 2018. 181−190.

[11] Denney E, Fischer B. Generating code review documentation for auto-generated mission-critical software. In: Proc. of the 3rd IEEE Int'l Conf. on Space Mission Challenges for Information Technology. 2009. 394−401.

[12] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review. In: Proc. of the 35th ACM/IEEE Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 712−721.

[13] Li ZY, Lu S, Guo D, Duan N, Jannu S, Jenks G, Majumder D, Green J, Svyatkovskiy A, Fu S, Sundaresan N. Automating code review activities by large-scale pre-training. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2022. 1035−1047.

[14] De Lima Júnior ML, Soares DM, Plastino A, Murta L. Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes. Journal of Systems and Software, 2018, 144: 181−196.

[15] Wang M, Lin Z, Zou Y, Xie B. CoRA: Decomposing and describing tangled code changes for reviewer. In: Proc. of the 34th Int'l Conf. on Automated Software Engineering. 2019. 1050−1061.

[16] Chouchen M, Ouni A, Olongo J, Mkaouer MW. Learning to predict code review completion time in modern code review. Empirical Software Engineering, 2023, 28(4): 82.

[17] Ebert F, Castor F, Novielli N, Serebrenik A. An exploratory study on confusion in code reviews. Empirical Software Engineering, 2021, 26(1): 12.

[18] Qiu HS, Vasilescu B, Kästner C, Egelman C, Jaspan C, Murphy-Hill E. Detecting interpersonal conflict in issues and code review: Cross pollinating open- and closed-source approaches. In: Proc. of the 44th Int'l Conf. on Software Engineering: Software Engineering in Society. New York: ACM, 2022. 41−55.

[19] Zampetti F, Mudbhari S, Arnaoudova V, Di Penta M, Panichella S, Antoniol G. Using code reviews to automatically configure static analysis tools. Empirical Software Engineering, 2021, 27(1): 28.

[20] Hijazi H, Cruz J, Castelhano J, Couceiro R, Castelo-Branco M, De Carvalho P, Madeira H. IReview: An intelligent code review evaluation tool using biofeedback. In: Proc. of the 32nd Int'l Symp. on Software Reliability Engineering. 2021. 476−485.

[21] Hijazi H, Duraes J, Couceiro R, Castelhano J, Barbosa R, Medeiros J, Castelo-Branco M, De Carvalho P, Madeira H. Quality evaluation of modern code reviews through intelligent biometric program comprehension. IEEE Trans. on Software Engineering, 2023, 49(2): 626−645. [doi: 10.1109/TSE.2022.3158543]

[22] Lee JB, Ihara A, Monden A, Matsumoto K. Patch reviewer recommendation in OSS projects. In: Proc. of the 20th Asia-Pacific Software Engineering Conf. 2013. 1−6.

[23] Yu Y, Wang H, 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]

[24] Ying H, Chen L, Liang T, Wu J. EARec: Leveraging expertise and authority for pull-request reviewer recommendation in GitHub. In: Proc. of the 3rd Int'l Workshop on CrowdSourcing in Software Engineering. New York: ACM, 2016. 29−35.

[25] Kong D, Chen Q, Bao L, Sun C, Xia X, Li S. Recommending code reviewers for proprietary software projects: A large scale study. In: Proc. of the 29th IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. 2022. 630−640.

[26] Sülün E, Tüzün E, Doğrusöz U. RSTrace+: Reviewer suggestion using software artifact traceability graphs. Information and Software Technology, 2021, 130: 106455. [doi: 10.1016/j.infsof.2020.106455]

[27] Sülün E, Tüzün E, Doğrusöz U. Reviewer recommendation using software artifact traceability graphs. In: Proc. of the 15th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. New York: ACM, 2019: 66−75.

[28] Rong G, Zhang Y, Yang L, Zhang F, Kuang H, Zhang H. Modeling review history for reviewer recommendation: A hypergraph approach. In: Proc. of the 44th Int'l Conf. on Software Engineering. New York: ACM, 2022. 1381−1392.

[29] Zanjani MB, Kagdi H, Bird C. Automatically recommending peer reviewers in modern code review. IEEE Trans. on Software Engineering, 2016, 42(6): 530−543. [doi: 10.1109/TSE.2015.2500238]

[30] Asthana S, Kumar R, Bhagwan R, Bird C, Bansal C, Maddila C, Mehta S, Ashok B. WhoDo: Automating reviewer suggestions at scale. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2019. 937−945.

[31] Mirsaeedi E, Rigby PC. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. New York: ACM, 2020. 1183−1195.

[32] Fejzer M, Przymus P, Stencel K. Profile based recommendation of code reviewers. Journal of Intelligent Information Systems, 2018, 50(3): 597−619. [doi: 10.1007/s10844-017-0484-1]

[33] Rahman MM, Roy CK, Collins JA. CORRECT: Code reviewer recommendation in GitHub based on cross-project and technology experience. In: Proc. of the 38th Int'l Conf. on Software Engineering Companion. 2016. 222−231.

[34] 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. Information and Software Technology, 2017, 84: 48−62. [doi: 10.1016/j.infsof.2016.10.006]

[35] Ye X. Learning to Rank reviewers for pull requests. IEEE Access, 2019, 7: 85382−85391. [doi: 10.1109/ACCESS.2019.2925560]

[36] Jeong G, Kim S, Zimmermann T, Yi K. Improving code review by predicting reviewers and acceptance of patches. Research on Software Analysis for Error-free Computing Center Tech-Memo, 2009, 6(1): 1−18.

[37] Thongtanunam P, Kula RG, Cruz AEC, Yoshida N, Iida H. Improving code review effectiveness through reviewer recommendations. In: Proc. of the 7th Int'l Workshop on Cooperative and Human Aspects of Software Engineering. New York: ACM, 2014. 119−122.

[38] Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto K. Who should review my code?—A file location-based code-reviewer recommendation approach for modern code review. In: Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering. 2015. 141−150.

[39] Lipcak J, Rossi B. A large-scale study on source code reviewer recommendation. In: Proc. of 44th Euromicro Conf. on Software Engineering and Advanced Applications. 2018. 378−387.

[40] Jiang J, He JH, Chen XY. CoreDevRec: Automatic core member recommendation for contribution evaluation. Journal of Computer Science and Technology, 2015, 30(5): 998−1016. [doi: 10.1007/s11390-015-1577-3]

[41] Chouchen M, Ouni A, Mkaouer MW, Kula RG, Inoue K. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. Applied Soft Computing, 2021, 100: 106908.

[42] Al-Zubaidi WHA, Thongtanunam P, Dam HK, Tantithamthavorn C, Ghose A. Workload-aware reviewer recommendation using a multi-objective search-based approach. In: Proc. of the 16th ACM Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. New York: ACM, 2020. 21−30.

[43] Rebai S, Amich A, Molaei S, Kessentini M, Kazman R. Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations. Automated Software Engineering, 2020, 27(3): 301−328.

[44] Hu YZ, Wang JJ, Li SB, Hu J, Wang Q. Response time constrained code reviewer recommendation. Ruan Jian Xue Bao/Journal of Software, 2021, 32(11): 3372−3387 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6079.htm [doi: 10.13328/j.cnki.jos.006079]

[45] Xia X, Lo D, Wang X, Yang X. Who should review this change?—Putting text and file location analyses together for more accurate recommendations. In: Proc. of the 31st Int'l Conf. on Software Maintenance and Evolution. 2015. 261−270.

[46] Strand A, Gunnarson M, Britto R, Usman M. Using a context-aware approach to recommend code reviewers: Findings from an industrial case study. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering: Software Engineering in Practice. New York: ACM, 2020. 1−10.

[47] Chueshev A, Lawall J, Bendraou R, Ziadi T. Expanding the number of reviewers in open-source projects by recommending appropriate developers. In: Proc. of the 36th Int'l Conf. on Software Maintenance and Evolution. 2020. 499−510.

[48] Kim J, Lee E. Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation. Symmetry, 2018, 10(4): 114.

[49] Liao Z, Wu Z, Li Y, Zhang Y, Fan X, Wu J. Core-reviewer recommendation based on pull request topic model and collaborator social network. Soft Computing, 2020, 24(8): 5683−5693.

[50] Yang C, Zhang X, Lingbin Z, Fan Q, Wang T, Yu Y, Yin G, Wang H. RevRec: A two-layer reviewer recommendation algorithm in pull-based development model. Journal of Central South University, 2018, 25: 1129−1143.

[51] Xia Z, Sun H, Jiang J, Wang X, Liu X. A hybrid approach to code reviewer recommendation with collaborative filtering. In: Proc. of the 6th Int'l Workshop on Software Mining. 2017. 24−31.

[52] Pandya P, Tiwari S. CORMS: A GitHub and Gerrit based hybrid code reviewer recommendation approach for modern code review. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2022. 546−557.

[53] Liu J, Deng A, Xie Q, Yue G. A code reviewer recommendation approach based on attentive neighbor embedding propagation. Electronics, Multidisciplinary Digital Publishing Institute, 2023, 12(9): 2113.

[54] Pejić N, Radivojević Z, Cvetanović M. Helping pull request reviewer recommendation systems to focus. IEEE Access, 2023, 11: 71013−71025.

[55] Hu Y, Wang J, Hou J, Li S, Wang Q. Is there a "golden" rule for code reviewer recommendation? —An experimental evaluation. In: Proc. of 20th Int'l Conf. on Software Quality, Reliability and Security. 2020. 497−508.

[56] Chen Q, Kong D, Bao L, Sun C, Xia X, Li S. Code reviewer recommendation in tencent: Practice, challenge, and direction. In: Proc. of 44th Int'l Conf. on Software Engineering: Software Engineering in Practice. 2022. 115−124.

[57] Denney E, Fischer B. Generating code review documentation for auto-generated mission-critical software. In: Proc. of the 3rd IEEE Int'l Conf. on Space Mission Challenges for Information Technology. 2009. 394−401.

[58] Mehrpour S, LaToza TD. Can static analysis tools find more defects? Empirical Software Engineering, 2022, 28(1): 5.

[59] Bosu A, Carver JC, Hafiz M, Hilley P, Janni D. Identifying the characteristics of vulnerable code changes: An empirical study. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM, 2014. 257−268.

[60] Hanam Q, Mesbah A, Holmes R. Aiding code change understanding with semantic change impact analysis. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution. 2019. 202−212.

[61] Zhang T, Song M, Pinedo J, Kim M. Interactive code review for systematic changes. In: Proc. of the 37th IEEE Int'l Conf. on Software Engineering, Vol.1. 2015. 111−122.

[62] Sharma S, Sodhi B. Using stack overflow content to assist in code review. Software: Practice and Experience, 2019, 49(8): 1255−1277.

[63] 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. New York: ACM, 2014. 345−355.

[64] Fan Y, Xia X, Lo D, Li S. Early prediction of merged code changes to prioritize reviewing tasks. Empirical Software Engineering, 2018, 23(6): 3346−3393.

[65] Jiang J, Zheng J, Yang Y, Zhang L. CTCPPre: A prediction method for accepted pull requests in GitHub. Journal of Central South University, 2020, 27: 449−468.

[66] Ochodek M, Hebig R, Meding W, Frost G, Staron M. Recognizing lines of code violating company-specific coding guidelines using machine learning—A method and its evaluation. Empirical Software Engineering, 2020, 25(1): 220−265.

[67] Soltanifar B, Erdem A, Bener A. Predicting defectiveness of software patches. In: Proc. of the 10th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. New York: ACM, 2016. 1−10.

[68] Hellendoorn VJ, Devanbu PT, Bacchelli A. Will they like this? Evaluating code contributions with language models. In: Proc. of the 12th Working Conf. on Mining Software Repositories. 2015. 157−167.

[69] Shi ST, Li M, Lo D, Thung F, Huo X. Automatic code review by learning the revision of source code. In: Proc. of the AAAI Conf. on Artificial Intelligence, 2019, 33(1): 4910−4917.

[70] Li HY, Shi ST, Thung F, Huo X, Xu B, Li M, Lo D. DeepReview: Automatic code review using deep multi-instance learning. In: Yang Q, Zhou ZH, Gong Z, Zhang ML, Huang SJ, eds. Proc. of the Advances in Knowledge Discovery and Data Mining. Cham: Springer, 2019. 318−330.

[71] Wu B, Liang B, Zhang X. Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network. Knowledge-based Systems, 2022, 252: 109450.

[72] Hellendoorn VJ, Tsay J, Mukherjee M, Hirzel M. Towards automating code review at scale. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2021. 1479−1482.

[73] Sawant N, Sengamedu S. Code compliance assessment as a learning problem. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). 2022. 445−454.

[74] Palvannan N, Brown C. Suggestion bot: Analyzing the impact of automated suggested changes on code reviews. arXiv:2305.06328, 2023.

[75] Klinik M, Koopman P, Van der Wal R. Personal prof: Automatic code review for Java assignments. In: Proc. of the 10th Computer Science Education Research Conf. New York: ACM, 2022. 31−38.

[76] Chatley R, Jones L. Diggit: Automated code review via software repository mining. In: Proc. of the 25th Int'l Conf. on Software Analysis, Evolution and Reengineering. 2018. 567−571.

[77] Gupta A. Intelligent code reviews using deep learning. In: Proc. of the 24th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. 2018.

[78] Siow JK, Gao C, Fan L, Chen S, Liu Y. CORE: Automating review recommendation for code changes. In: Proc. of the 27th Int'l Conf. on Software Analysis, Evolution and Reengineering. 2020. 284−295.

[79] Hong Y, Tantithamthavorn C, Thongtanunam P, Aleti A. CommentFinder: A simpler, faster, more accurate code review comments recommendation. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2022. 507−519.

[80] Moshkin V, Kalachev V, Zarubin A. Automation of program code analysis using machine learning methods. In: Proc. of the 2022 Int'l Russian Automation Conf. (RusAutoCon). IEEE, 2022. 404−408.

[81] Li LW, Yang L, Jiang H, Yan J, Luo T, Hua ZH, Liang G, Zuo C. AUGER: Automatically generating review comments with pre-training models. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2022. 1009−1021.

[82] Tufano R, Masiero S, Mastropaolo A, Pascarella L, Poshyvanyk D, Bavota G. Using pre-trained models to boost code review automation. In: Proc. of the 44th Int'l Conf. on Software Engineering. New York: ACM, 2022. 2291−2302.

[83] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research, 2020, 21(1): 5485−5551.

[84] Zhou X, Kim K, Xu B, Han D, He J, Lo D. Generation-based code review automation: How far are we? arXiv:2303.07221, 2023.

[85] Lin H, Thongtanunam P. Towards automated code reviews: Does learning code structure help? In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. New York: ACM, 2023. 703−707.

[86] Tian YC, Li KJ, Wang TM, Jiao QQ, Li GJ, Zhang YX, Liu H. Survey on code smells. Ruan Jian Xue Bao/Journal of Software, 2023, 34(1): 150−170 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6431.htm [doi: 10.13328/j.cnki.jos.006431]

[87] Allamanis M, Barr ET, Bird C, Sutton C. Learning natural coding conventions. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM, 2014. 281−293.

[88] Markovtsev V, Long W, Mougard H, Slavnov K, Bulychev E. STYLE-analyzer: Fixing code style inconsistencies with interpretable unsupervised algorithms. In: Proc. of the 16th Int'l Conf. on Mining Software Repositories. Montreal: IEEE, 2019. 468−478.

[89] Tufano R, Pascarella L, Tufano M, Poshyvanyk D, Bavota G. Towards automating code review activities. In: Proc. of the 43rd Int'l Conf. on Software Engineering. 2021. 163−174.

[90] Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D. On learning meaningful code changes via neural machine translation. In: Proc. of the 41st Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 25−36.

[91] Thongtanunam P, Pornprasit C, Tantithamthavorn C. AutoTransform: Automated code transformation to support modern code review process. In: Proc. of the 44th Int'l Conf. on Software Engineering. 2022. 237−248.

[92] Pornprasit C, Tantithamthavorn C, Thongtanunam P, Chen C. D-ACT: Towards diff-aware code transformation for code review under a time-wise evaluation. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. 2023. 296−307.

[93] Yin Y, Zhao Y, Sun Y, Chen C. Automatic code review by learning the structure information of code graph. Sensors, Multidisciplinary Digital Publishing Institute, 2023, 23(5): 2551.

[94] Chen X, Yang G, Cui ZQ, Meng GZ, Wang Z. Survey of state-of-the-art automatic code comment generation. Ruan Jian Xue Bao/Journal of Software, 2021, 32(7): 2118−2141 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6258.htm [doi: 10.13328/j.cnki.jos.006258]

[95] Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting on Association for Computational Linguistics. ACL, 2002. 311−318.

[96] Lin CY. Rouge: A package for automatic evaluation of summaries. In: Proc. of the Text Summarization Branches Out. 2004. 74−81.

[97] Heumüller R, Nielebock S, Ortmeier F. Exploit those code reviews! Bigger data for deeper learning. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2021. 1505−1509.

[98] Paixao M, Krinke J, Han D, Harman M. CROP: Linking code reviews to source code changes. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. 2018. 46−49.

[99] Bhandari G, Naseer A, Moonen L. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In: Proc. of the 17th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. New York: ACM, 2021. 30−39.

[100] Hong Y, Tantithamthavorn CK, Thongtanunam PP. Where should I look at? Recommending lines that reviewers should pay attention to. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. 2022. 1034–1045.

[101] Turzo AK. Towards improving code review effectiveness through task automation. In: Proc. of the 37th Int'l Conf. on Automated Software Engineering. New York: ACM, 2023. 1–5.

[102] Gauthier IX, Lamothe M, Mussbacher G, McIntosh S. Is historical data an appropriate benchmark for reviewer recommendation systems?—A case study of the Gerrit community. In: Proc. of the 36th Int'l Conf. on Automated Software Engineering. 2021. 30–41.

[103] Deng X, Ye W, Xie R, Zhang SK. Survey of source code bug detection based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 625–654 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/ j.cnki.jos.006696]

[104] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2665–2690 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

[105] Yang ZZ, Chen SR, Gao CY, Li ZH, Li G, LYU MRT. Deep learning based code generation methods: Literature review. Ruan Jian Xue Bao/Journal of Software (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6981.htm [doi: 10.13328/j.cnki.jos.006981]

附中文参考文献:

[44] 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. 软件学报, 2021, 32(11): 3372–3387. http://www.jos.org.cn/1000-9825/6079.htm [doi: 10.13328/j.cnki.jos.006079]

[86] 田迎晨, 李柯君, 王太明, 焦青青, 李光杰, 张宇霞, 刘辉. 代码坏味研究综述. 软件学报, 2023, 34(1): 150–170. http://www.jos.org.cn/1000-9825/6431.htm [doi: 10.13328/j.cnki.jos.006431]

[94] 陈翔, 杨光, 崔展齐, 孟国柱, 王赞. 代码注释自动生成方法综述. 软件学报, 2021, 32(7): 2118–2141. http://www.jos.org.cn/1000-9825/6258.htm [doi: 10.13328/j.cnki.jos.006258]

[103] 邓枭, 叶蔚, 谢睿, 张世琨. 基于深度学习的源代码缺陷检测研究综述. 软件学报, 2023, 34(2): 625–654. http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/j.cnki.jos.006696]

[104] 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665–2690. http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

[105] 杨泽洲, 陈思榕, 高翠芸, 李振昊, 李戈, 吕荣聪. 基于深度学习的代码生成方法研究进展. 软件学报, 2024, 35(2): 604–628. http://www.jos.org.cn/1000-9825/6981.htm [doi: 10.13328/j.cnki.jos.006981]

花子涵(2000－), 女, 硕士生, CCF 学生会员, 主要研究领域为智能软件工程.

陆俊逸(1999－), 男, 博士生, 主要研究领域为智能软件工程.

杨立(1978－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为智能化软件开发方法及质量保障.

左春(1959－), 男, 研究员, 博士生导师, 主要研究领域为智能化软件系统工程.