软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(12):5501−5526 [doi: 10.13328/j.cnki.jos.006749]

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

代码变更表示学习及其应用研究进展*

刘忠鑫[1], 唐郅杰[1], 夏 鑫[2], 李善平[1]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](华为公司 软件工程应用技术实验室, 浙江 杭州 310007)

通信作者: 夏鑫, E-mail: xin.xia@acm.org

摘 要: 代码变更是软件演化过程中的关键行为, 其质量与软件质量密切相关. 对代码变更进行建模和表示是众多软件工程任务的基础, 例如即时缺陷预测、软件制品可追溯性恢复等. 近年来, 代码变更表示学习技术得到了广泛的关注与应用. 该类技术旨在学习将代码变更的语义信息表示为稠密低维实值向量, 即学习代码变更的分布式表示, 相比于传统的人工设计代码变更特征的方法具有自动学习、端到端训练和表示准确等优点. 但同时该领域目前也存在如结构信息利用困难、基准数据集缺失等挑战. 对近期代码变更表示学习技术的研究及应用进展进行了梳理和总结, 主要内容包括: (1) 介绍了代码变更表示学习及其应用的一般框架. (2) 梳理了现有的代码变更表示学习技术, 总结了不同技术的优缺点. (3) 总结并归类了代码变更表示学习技术的下游应用. (4) 归纳了代码变更表示学习技术现存的挑战和潜在的机遇, 展望了该类技术的未来发展方向.

关键词: 代码变更; 表示学习; 代码变更表示; 软件演化; 软件维护

中图法分类号: TP311

中文引用格式: 刘忠鑫, 唐郅杰, 夏鑫, 李善平. 代码变更表示学习及其应用研究进展. 软件学报, 2023, 34(12): 5501–5526. http://www.jos.org.cn/1000-9825/6749.htm

英文引用格式: Liu ZX, Tang ZJ, Xia X, Li SP. Research Progress of Code Change Representation Learning and Its Application. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5501–5526 (in Chinese). http://www.jos.org.cn/1000-9825/6749.htm

Research Progress of Code Change Representation Learning and Its Application

LIU Zhong-Xin[1], TANG Zhi-Jie[1], XIA Xin[2], LI Shan-Ping[1]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](Software Engineering Application Technology Lab, Huawei Technologies Co. Ltd., Hangzhou 310007, China)

Abstract: Code change is a kind of key behavior in software evolution, and its quality has a large impact on software quality. Modeling and representing code changes is the basis of many software engineering tasks, such as just-in-time defect prediction and recovery of software product traceability. The representation learning technologies for code changes have attracted extensive attention and have been applied to diverse applications in recent years. This type of technology targets at learning to represent the semantic information in code changes as low-dimensional dense real-valued vectors, namely, learning the distributed representation of code changes. Compared with the conventional methods of manually designing code change features, such technologies offers the advantages of automatic learning, end-to-end training, and accurate representation. However, this field is still faced with some challenges, such as great difficulties in utilizing structural information and the absence of benchmark datasets. This study surveys and summarizes the recent progress of studies and applications of representation learning technologies for code changes, and it mainly consists of the following four parts. (1) The study presents the general framework of representation learning of code changes and its application. (2) Subsequently, it reviews the currently available representation learning technologies for code changes and summarizes their respective advantages and disadvantages. (3) Then, the downstream applications of such technologies are summarized and classified. (4) Finally, this study discusses the challenges and

* 基金项目: 浙江大学教育基金会启真人才基金
收稿时间: 2021-12-23; 修改时间: 2022-04-21; 采用时间: 2022-07-25; jos 在线出版时间: 2022-10-27
CNKI 网络首发时间: 2023-02-09

potential opportunities ahead of representation learning technologies for code changes and suggests the directions for the future development of this type of technology.

Key words: code change; representation learning; code change representation; software evolution; software maintenance

1 引 言

代码变更 (code change), 也称代码编辑 (code edit), 是指对软件源代码的增加、删除和修改. 代码变更是软件演化过程中的关键行为, 被用以完成功能需求实现、软件缺陷修复和软件架构改进等重要任务[1,2]. 软件项目的代码库可被视为一系列代码变更的有序组合, 因此代码变更对于理解代码库和分析软件演化过程具有重要的意义. 由于代码变更在软件演化中的关键地位, 大量软件工程任务的进行都依赖于代码变更的分析和理解. 例如, 即时缺陷预测需要分析代码变更的质量以预测出现缺陷的概率[3−8], 软件制品可溯性恢复需要理解代码变更的内容和意图以构建代码与其他软件制品的关联[9,10]. 表示和建模代码变更是自动分析和理解代码变更的基础, 因此代码变更表示问题是影响众多软件工程任务的重要问题.

已有大量研究工作探究了代码变更的表示及其在下游软件工程任务上的应用[3−52]. 早期的代码变更表示技术主要通过人工设计的特征或特征提取规则将代码变更表示为特征向量[50,51]. 例如 Fluri 等人[53]通过人工定义抽象语法树 (abstract syntax tree, AST) 匹配规则来提取变更前后 AST 的最小编辑脚本, 并基于该编辑脚本来识别代码变更类型; Kamei 等人[54]基于代码变更的扩散 (diffusion)、规模 (size)、目标 (purpose)、历史 (history) 和开发者经验 (experience) 这 5 个维度定义了一系列特征来评估代码变更引入缺陷的可能性. 这些方法依赖于手工分析, 且通常只能提取到显式的、浅层的特征, 难以捕捉到代码变更语义等隐式的、深层的信息, 具有较大的应用局限性.

表示学习旨在将研究对象表示为低维稠密的实值向量[55]. 研究对象的语义信息被分散到向量的各个维度, 使得相似的对象能被映射到特征空间中距离相近的向量上. 近年来, 以深度学习[56]为代表的表示学习技术在计算机视觉和自然语言处理等领域取得了令人瞩目的突破[57,58]. 另一方面, 信息技术快速发展, 现代软件开发呈现快速演进、持续迭代和频繁交付的新形态. 这使得单一代码变更变得更轻量、更内聚, 极大地增加了代码变更的频率和数量, 从而使得利用表示学习模型从代码变更历史中学习代码变更的表示成为可能. 基于上述背景, 近年来, 代码变更表示学习得到了越来越广泛的关注[3,32,35]. 代码变更表示学习旨在将代码变更的语义信息表示为低维稠密的实值向量, 即学习代码变更的分布式表示[59]. 相比于传统的基于特征工程对代码变更进行表示和建模的方法, 代码变更表示学习技术具有以下优势.

- 自动学习: 此类技术利用表示学习模型从代码变更历史数据中自动学习代码变更的表示方法, 无需人工定义特征.
- 端到端训练: 代码变更表示学习模型可与下游任务模型一起进行端到端的训练, 简化了代码变更表示的训练和应用过程.
- 表示准确: 基于特别设计的表示学习模型、大规模代码变更数据以及下游任务提供的反馈信息, 此类技术能够更好地捕捉到更细粒度的[30,31]、包含语义信息[4,5]且与下游任务密切相关的高层次特征. 这些高层次特征有望更好地反映代码变更的内容与意图, 显著提升下游任务的性能.

由于上述优点, 代码变更表示学习技术已被应用到众多软件工程任务中, 包括代码提交日志生成[3,11−20,32,33,39]、即时缺陷预测[3−8]、即时注释更新[21,22]、安全漏洞补丁识别[25−27]和冲突合并[31,46]等. 但这一研究方向仍面临内在评估 (intrinsic evaluation) 方法缺乏和基准数据集 (benchmark datasets) 缺失等问题, 且目前尚无相关工作对代码变更表示学习的研究和应用进展进行系统性梳理. 鉴于此, 本文旨在对代码变更表示学习及其应用的研究进展进行系统性的梳理和归纳. 受限于算力和现有表示学习模型的能力, 现有代码变更表示学习相关研究考虑的最大代码变更粒度为代码提交 (commit), 尚未有面向更大粒度代码变更 (例如版本变更) 的表示学习方法. 因此, 本文聚焦于代码提交及更小粒度 (例如文件粒度, 块粒度) 的代码变更表示学习及其应用.

不同于表示学习在软件工程领域的常见研究对象, 例如代码[60,61]和代码文档[62], 代码变更包含两部分信息, 即变更前的代码和变更后的代码. 理解代码变更的关键在于结合上下文理解变更前后代码的差异, 而为了更好地捕

捉和表示“差异”, 代码变更表示学习技术需要比对与融合变更前后的代码中的信息. 此种比对与融合操作可以发生在代码变更被表示学习模型处理之前, 即先比对与融合变更前后的代码, 将代码变更转换为特殊的形式 (例如 diff), 再将转换后的代码变更输入到表示学习模型中获取其特征表示. 此种操作也可以发生在代码变更被表示学习模型处理之后, 即先将变更前后的代码输入表示学习模型以分别获取其特征表示, 再比对与融合二者的特征表示以获取代码变更的特征表示. 本文将前一类方法称为显式信息交互, 后一类方法称为隐式信息交互, 并在第 2.4 节中详细介绍这两大类方法. 后文将基于此将现有代码变更表示学习方法分为基于显式信息交互的方法和基于隐式信息交互的方法两类. 以此分类方法为出发点, 本文首先从输入表示形式和表示学习模型等维度对现有代码变更表示学习技术进行了梳理和归纳, 然后对此类技术的下游应用进行了归类总结, 最后探讨了代码变更表示学习领域目前存在的关键挑战和研究机遇, 展望了这一领域未来的研究趋势.

文献选取方式. 本文选取文献的标准为: 该文献针对代码变更表示学习提出了新理论或新技术, 或者该文献将代码变更表示学习相关理论和技术应用于软件工程任务中. 此外, 该文献应该公开发表在会议、期刊、技术报告或书籍中. 根据上述标准, 本文通过以下步骤对文献进行检索和选取.

(1) 选用主流的在线电子文献数据库, 包括 ACM Digital Library, IEEE Xplore 电子文献数据库, Springer Link 电子文献数据库, Google Scholar 学术搜索引擎, 中国知网等使用关键字进行原始搜索. 搜索的关键字包括: learning code changes、 learning code edits、 distributed representation of code changes 和 represent code changes 等. 同时在标题、摘要、关键词和索引中进行检索.

(2) 对软件工程领域主要的期刊和会议进行在线搜索, 具体包括 TSE、TOSEM、ICSE、FSE/ESEC、ASE、ISSTA、ICSME、SANER、MSR、软件学报、计算机学报、中国科学: 信息科学、计算机研究与发展等, 搜索近 5 年 (从 2017 年开始) 的相关文章.

(3) 由于代码变更表示学习在许多工作中仅作为所使用技术的一部分而非主题, 因此以上检索方式可能会遗漏相关工作. 为保证文献检索的完整性, 本文对上述步骤检索到的文章进行逐一查看, 并递归地从文章的引用和被引文献中, 进一步筛选出与代码变更表示学习相关的文献. 筛选方式具体为: 先查看文章的标题和摘要, 判断是否相关; 如果不能分辨, 则进一步查看文章中提出的方法是否满足以下两个要求: 1) 从数据中自动学习代码变更的分布式表示; 2) 将代码变更的分布式表示用于软件工程任务中.

基于以上检索方法和选取原则, 共计将 50 篇文献纳入到后文的文献总结中. 其中 46 篇文献在特定的软件工程任务下提出了代码变更表示学习的新模型和新算法, 另外 4 篇文献则是对已有的代码变更表示学习技术的实证研究或复现研究. 上述文献的分布情况在图 1 中展示, 其中发表较多的相关期刊和会议有: ASE 论文 6 篇, ICSE 论文 4 篇, TSE、FSE/ESEC、ACL 和 AAAI 论文各 3 篇, TOSEM、ICLR 和 MSR 论文各 2 篇. 剩余 22 篇相关文献发表在其他的会议和期刊上, 包括 ISSTA、IJCAI、EMNLP、SANER、JSS、ISSRE、CIKM、ESEM 和软件学报等.

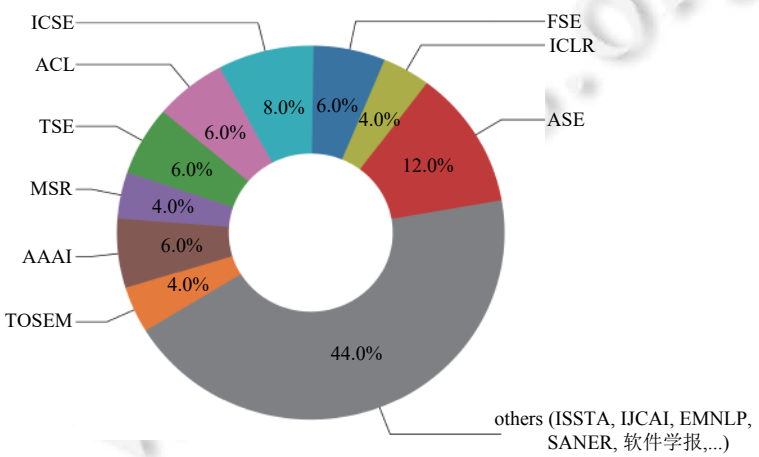


图 1 代码变更表示学习相关文献分布

2 代码变更表示学习相关概念和概况

2.1 代码变更表示学习相关概念

本文将用到的与代码变更表示学习相关的概念定义如下.

◆ 代码变更: 也被称为代码编辑或代码改动, 指对软件源代码的增加、删除和修改. 其包含两部分信息, 即变更前代码和变更后代码.

◆ 变更前代码/变更后代码: 发生变更前代码库中的所有代码和发生变更后代码库中的所有代码.

◆ 代码变更粒度: 代码的结构性使代码变更也具有层次性, 代码变更粒度按照从大到小包括: 代码提交 (commit/change)、文件 (file)、块 (hunk)、语句 (statement)、行 (line)、实体 (entity) 和符号 (token), 如图 2 所示.

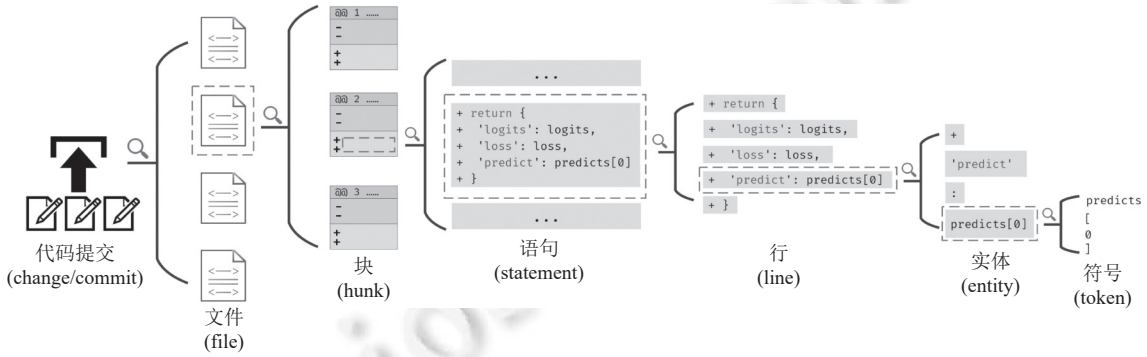


图 2 代码变更的层次性

◆ 代码变更的分布式表示: 分布式表示最初起源于自然语言处理 (natural language processing, NLP) 领域, 旨在使用低维向量中的所有元素来共同表示研究对象[59], 以克服传统的独热编码 (one-hot encoding) 存在的高维稀疏问题. 这一低维稠密向量被称为研究对象的分布式表示. 后来分布式表示概念和方法也被应用于其他领域, 将领域相关的实体表示为低维稠密实值向量, 以提升计算效率和下游应用的性能[63,64]. 类似的, 代码变更的分布式表示指表示代码变更的低维稠密实值向量.

◆ 代码变更表示学习: 基于表示学习的定义[55], 代码变更表示学习指利用机器学习模型将代码变更的语义信息表示为低维稠密实值向量. 本文将用于表示学习的机器学习模型称为表示学习模型.

2.2 代码变更表示学习应用的一般框架

与代码变更表示学习相关的软件工程任务大多遵循如图 3 所示的流程框架, 该框架包含以下环节.

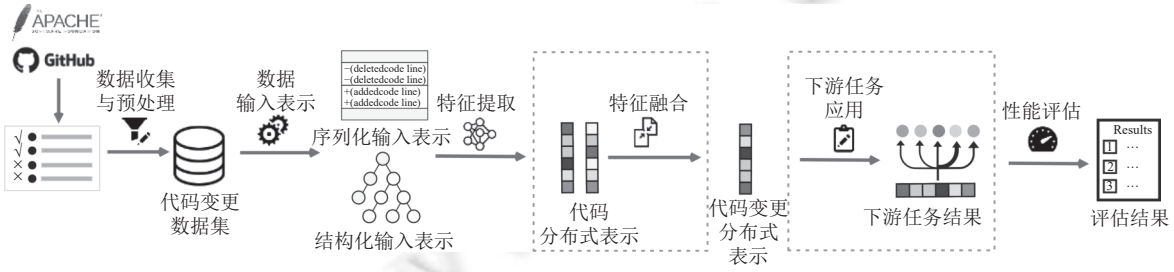


图 3 代码变更表示学习及其应用的一般流程框架

1) 数据收集和预处理. 代码变更数据通常是从主流的社交化开发平台 (例如 GitHub) 或开源软件基金会 (例如 Apache 基金会) 中流行的、高质量的开源项目的代码仓库中提取的. 从代码仓库直接提取的代码变更数据中可能存在着一些噪声数据或不符合要求的数据, 因此往往需要进行数据预处理来精化数据. 常用的代码变更数据预

处理方法包括: 过滤无关的代码变更 (例如非代码更改)[36]、移除规模过大或非常规代码变更 (例如 rollback 和 merge)[27]和使用占位符 (placeholder) 替代代码变更中的任务无关信息 (例如 commit ID)[13]等. 不同的任务有不同的信息需求, 因此其使用的数据预处理方式可能存在较大差异, 但目的都是提高数据质量, 提升表示学习模型的训练效果.

2) 数据输入表示. 这一阶段旨在将经过预处理的代码变更数据转换为表示学习模型能够处理的表示形式. 输入表示形式主要分为序列化输入表示和结构化输入表示两大类. 一些代码变更表示学习方法会在这一步中显式地比对与融合变更前后代码中的信息, 从而将代码变更中的两部分信息转化为单一的输入. 另一些方法则不会在这一步中进行信息比对与融合, 而是直接将变更前后代码分别转换为特定表示形式, 然后输入到表示学习模型中.

3) 特征提取. 这一阶段旨在利用表示学习模型提取代码变更或代码的特征表示. 对于已进行显式信息比对与融合的方法, 该阶段会直接生成单一的特征向量作为代码变更的分布式表示; 而对于尚未比对与融合变更前后代码的方法, 表示学习模型会分别提取变更前后代码的特征表示.

4) 特征融合. 这一阶段旨在将前一步骤中提取出的多个特征向量比对和融合成一个, 作为代码变更的分布式表示. 特征提取前已经显式比对了变更前后代码的方法并不需要这一步骤.

5) 下游任务应用. 这一阶段将学习到的代码变更分布式表示输入下游任务的模型以完成相关任务.

6) 性能评估. 这一阶段通过不同的性能指标量化地评估代码变更的分布式表示或下游任务的性能.

2.3 代码变更表示学习方式

根据表示学习模型在训练过程中是否需要额外的数据标签, 代码变更表示学习方法的学习方式包括监督学习和无监督学习两种.

监督学习. 大部分代码变更表示学习方法都采用监督学习的学习方式. 这些方法需要与下游任务共同训练. 训练时, 表示学习模型提取出的代码变更表示会被输入到下游任务中产生预测结果. 预测结果与数据标签之间的差异被用来指导代码变更表示学习模型的训练. 上述差异通过损失函数进行衡量. 下游任务类型不同, 其使用的损失函数也可能不同. 对于分类任务, 损失函数通常使用基于分类结果的分类交叉熵损失 (cross-entropy loss)[23−29,34,44]; 对于生成任务, 损失函数大多使用每一步生成结果的交叉熵损失的均值[12,13,16−20,41,46]; 对于排序任务或者部分二分类任务, 均方误差 (mean-square error) 通常被用作损失函数[9,36]. 此外, 还有部分研究工作在训练表示学习模型时使用了一些特殊的损失函数, 以适应特殊模型的训练或提升模型的性能. 例如 Gesi 等人[8]使用对比损失 (contrastive loss)[65]来训练 Siamese 网络[66]以引导模型学会辨别相同类样本和不同类样本.

无监督学习. 也有一些研究工作在训练代码变更表示学习模型时并不依赖额外的数据标签, 即进行无监督学习. 无监督学习的优点在于不受限于具体任务, 能够学到通用的、任务无关的代码变更表示, 有望适应不同的代码上下文和下游任务并获得较好的泛化性[11]. 但相比于监督学习方法, 无监督学习方法在相关工作中使用较少. Yin 等人[11]最早提出通过自编码器 (autoencoder) 来无监督学习代码变更的分布式表示. 他们先利用编码器提取代码变更表示, 再使用解码器根据代码变更表示和变更前的代码来重建变更后代码, 并基于重建损失来进行无监督学习[11,37]. Loyola 等人[6]则基于软件开发活动中的一系列代码变更之间修改内容的依赖关系, 构建一个代码变更间的状态转移图, 称为代码变更族谱 (code change genealogy)[1]. 他们进而基于从代码变更族谱中抽取的随机游走序列, 使用无监督词嵌入方法 (例如 Word2Vec[67]等) 来直接学习代码变更分布式表示. 特别地, 一些研究工作还使用了一种特殊的无监督学习方法——预训练 (pre-training)[68]. 该方法首先在大量无标签的代码变更数据上对表示学习模型进行预训练, 以获得代码变更相关的先验知识[32]. 在应用到下游任务时, 只需利用少量有标签的代码变更数据对预训练好的模型进行微调 (fine-tuning) 即可获得不错的性能. 许多监督学习任务都基于预训练的表示学习模型来学习代码变更的表示[10,16,27,31].

2.4 代码变更信息交互类型

不同于源代码和代码文档等软件工程中常见的表示学习研究对象, 代码变更包含变更前代码和变更后代码这两部分信息. 因此, 代码变更表示学习问题与其他软件制品的表示学习问题最大的区别在于此问题需要有效地比对与融合变更前后的代码中的信息. 根据第 2.2 节中一般流程框架, 本文基于上述对比与融合操作的时机将代码

变更表示学习领域的研究工作分为两类. 第 3 节和第 4 节将分别介绍与这两类研究工作:

1) 基于显式信息交互的代码变更表示学习: 此类技术在数据输入表示阶段进行显式地比对与融合. 具体而言, 此类技术通常会先通过 git diff 和 AST 差分[69]等方式显式地比对和提取代码变更的内容和结构信息, 并将其转换为特殊的数据形式 (例如 diff), 然后利用表示学习模型从此输入中提取特征向量作为代码变更的分布式表示. 即, “先交互, 再提取”.

2) 基于隐式信息交互的代码变更表示学习: 此类技术在特征融合阶段进行隐式地比对和融合. 具体而言, 此类技术会先利用表示学习模型分别提取变更前和变更后代码的特征向量, 然后通过不同的特征融合方法 (例如特征并联和相似度计算等) 将变更前代码和变更后代码的分布式表示对比融合为单一的特征向量作为代码变更的分布式表示. 即, “先提取, 再交互”.

3 基于显式信息交互的代码变更表示学习

基于显式信息交互的代码变更表示学习方法在数据输入表示阶段就会使用一系列方法比对与融合变更前和变更后代码的信息 (即, 显式信息交互), 以帮助表示学习模型更好地捕捉代码变更中的特征. 大部分显式信息交互方法都在显式信息交互之后将代码变更表示为单一的输入, 将其输入表示学习模型之后就能直接获得代码变更的分布式表示[11−16], 如图 4 所示. 特别地, 也存在极少数研究只通过显式信息交互提取变更代码的位置信息 (例如发生变更的代码块), 仍将变更前后的代码分别输入表示学习模型获得二者的特征表示, 然后利用变更位置信息从变更前后的代码特征中筛选出代码变更的特征表示[29,35]. 显式信息交互方式实质是由研究者根据领域知识和任务需求来为后续的表示学习模型提供先验知识, 从而帮助表示学习模型有效捕捉和表示代码变更中的信息. 这种信息交互方式支持定制信息交互方法以提高变更输入数据的质量、强化关键变更信息[17,32], 从而增强下游任务的性能, 在相关研究中得到了广泛的应用.

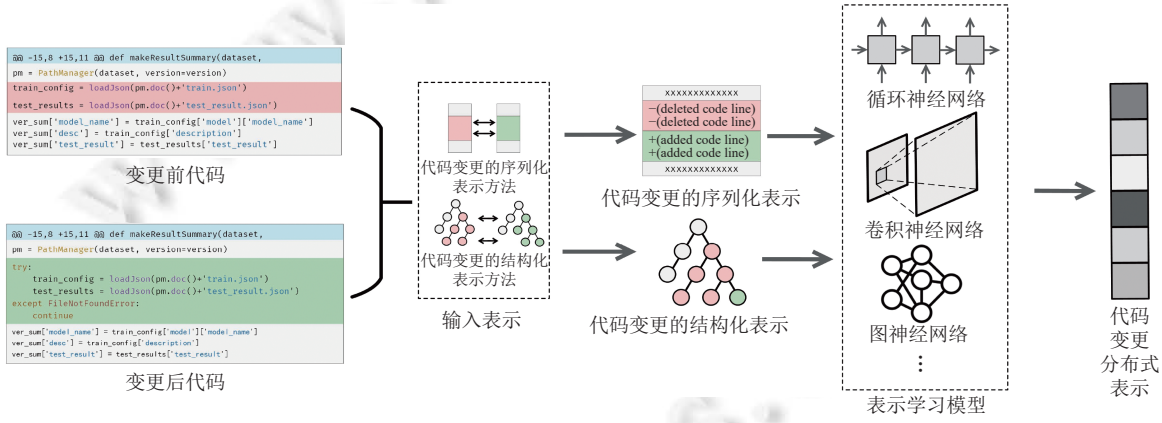


图 4 基于显式信息交互的代码变更表示提取流程

在基于显式信息交互的代码变更表示学习的研究中, 不同方法的输入表示方式——即将代码变更表示为何种形式以输入表示学习模型, 也不尽相同. 一些方法直接基于变更前后代码的文本构建代码变更的序列化表示, 例如将代码变更表示为 diff; 其他方法考虑了代码的结构性, 首先将变更前后的代码转化为树和图等形式, 再基于代码的结构化表示构建代码变更的结构化表示. 本节将基于显式信息交互的代码变更表示学习方法中, 输入表示阶段产生的是序列化表示的方法称为基于显式信息交互的序列化代码变更表示学习方法; 输入表示阶段产生的是结构化表示的方法称为基于显式信息交互的结构化代码变更表示学习方法.

3.1 基于显式信息交互的序列化代码变更表示学习

本节主要从输入表示形式和表示学习模型两方面来归纳基于显式信息交互的序列化代码变更表示学习的相关工作, 并总结在表 1 中.

表 1 基于显式信息交互的序列化代码变更表示学习相关工作

学习方式	输入表示形式	表示学习模型	对应文献	具体下游任务
监督学习	符号级变更序列	GRU	Panthaplackel等人[21]	即时注释更新
			Dinella等人[46]	冲突合并
			Panthaplackel等人[23]	即时代码-注释不一致检测
		LSTM	Liu等人[22]	即时注释更新
		Transformer	Zhao等人[40]	代码编辑预测
			Svyatkovskiy等人[31]	冲突合并
	diff (行级变更序列)	GRU	Jiang[18]	代码提交日志生成
			Liu等人[15]	代码提交日志生成
			Xu等人[13]	代码提交日志生成
		LSTM	Jiang等人[12]	代码提交日志生成
			Loyola等人[39]	代码提交日志生成
			Wang等人[20]	代码提交日志生成
			Loyola等人[45]	即时缺陷定位
			Siow等人[36]	代码评审意见推荐
			Ruan等人[9]	软件制品可追溯性恢复
		CNN	Hoang等人[4]	即时缺陷预测
			Ye等人[70]	代码评审人推荐
		Transformer	Jung[14]	代码提交日志生成
			Hellendoorn等人[38]	代码评审优先级排序
			Lin等人[10]	软件制品可追溯性修复
			Gao等人[24]	即时代码-注释不一致检测
			Nie等人[16]	代码提交日志生成
			Zhou等人[27]	安全漏洞补丁识别
			Ciborowska等人[52]	即时缺陷定位
		词嵌入	Loyola等人[33]	代码提交日志生成
	语句级变更序列	CNN+LSTM	Mi等人[35]	代码变更质量评估
			Shi等人[29]	代码评审结果预测
无监督学习	符号级变更序列	LSTM	Yin等人[11]	代码编辑迁移
			Pravilov等人[32]	代码编辑迁移、代码提交日志生成
			Panthaplackel等人[37]	代码编辑迁移
	diff (行级变更序列)	Word2Vec	Loyola等人[6]	即时缺陷检测
		DBN	Wang等人[5]	即时缺陷检测

3.1.1 输入表示形式

基于显式信息交互的序列化代码变更表示学习方法在将代码变更序列数据输入到表示学习模型之前, 都会通过一些比对和融合方法将其表示为合适的输入形式. 根据第 2.1 节所述的代码变更粒度, 不同的方法在比对和融合时会基于不同的对齐粒度——即在比对和融合变更前后代码时考虑的最小单元, 产生精细程度不同的输入表示形式.

符号级变更序列. 一段代码可以被分词为一个符号序列 (token sequence). 为了在保留语义信息的同时尽可能减少代码变更的序列化表示中的冗余信息, 一些研究工作以符号为最小单位来比对和融合变更前后的代码, 并生成符号级变更序列作为表示学习模型的输入[21−23,31,40,46]. 例如一些研究在比对和融合变更前后代码时, 只在结果中保留发生了变更的符号和相应的变更操作 (例如替换、增加和删除等)[21]. 然而对于规模较小的代码变更 (例如变量重命名), 这种输入表示方法可能会导致输入到表示学习模型中的信息过少, 不利于代码提交日志生

成[3,11−20,32,33,39]等比较依赖上下文的下游任务. 因此, 另一些同样进行符号级显式信息交互的相关研究选择在结果中保留部分未变更的代码内容作为上下文. 例如 Liu 等人[22]将同时包含变更和未变更符号的序列输入到表示学习模型中, 并额外输入一个与输入的符号序列等长的“编辑操作序列”来标记每个符号的具体变更操作, 使得模型能够分辨出输入符号序列中的变更符号和未变更符号.

diff (行级变更序列). diff, 即行级变更序列, 是相关研究中较为常见的一种输入表示形式, 常基于 git diff 等工具对变更前后的代码进行逐行比对来生成[4,9,10,12−16,18,20,24,27,33,36,38,39,45,52,70]. 对于变更前后未发生修改的代码行, diff 会直接合并两者放入输入表示结果中; 而对于变更前后发生了修改的代码行, diff 会同时将两者放入输入表示结果中, 并在行首分别使用“–”和“+”来标识出变更前和变更后的代码行 (也称为删除行和添加行). 一些研究工作在构建 diff 时会保留变更代码行前后的部分未变更代码行作为变更上下文, 与其余变更代码行一同输入表示学习模型以获取代码变更的表示[9,10,12,13,15,16,18,20,24,38,39,52]. 不过, 这些未变更代码行中包含的信息并不一定都与变更代码行相关, 可能给表示学习模型的训练带来额外负担并影响模型效果. 鉴于此, 另一些研究在将代码变更转换为 diff 时选择只保留删除行和添加行作为表示学习模型的输入, 以减少输入中的冗余信息[4,14].

语句级变更序列. 除了符号级变更序列和行级变更序列, 也有研究以语句为最小单元对变更前后的代码进行比对与融合, 以生成语句级变更序列作为表示学习模型的输入. 相关工作中通常先逐行比对变更前后的代码以定位发生变更的代码行, 再基于变更的代码行识别对应的代码语句[29,35]. 例如 Shi 等人[29]首先通过比对变更前后的代码定位了发生变更的语句. 在将变更前后代码中的所有语句输入到变更表示学习模型后, 他们会根据定位结果选出变更语句对应的特征向量作为代码变更的表示. 考虑到一些代码语句可能会横跨多个代码行, 如图 2 所示, 因此与基于行级变更粒度的 diff 相比, 这种输入表示形式可能可以包含更准确的上下文信息.

3.1.2 表示学习模型

在获得代码变更的序列化表示之后, 大多数代码变更表示方法都首先使用一个嵌入层 (embedding layer) 来将序列化表示中的符号或字符转换为一个向量. 这个向量也称为词嵌入 (word embedding). 然后表示学习模型基于这些向量构建代码变更的分布式表示. 然而不同研究中使用的从代码变更的词嵌入序列中抽取代码变更分布式表示的模型却不尽相同.

RNN. 循环神经网络 (recurrent neural network, RNN) 是最常用的提取序列数据特征的表示学习模型, 能够建模代码变更的序列化表示中普遍存在的长期依赖, 例如函数的定义与调用和全局变量的声明与使用等. LSTM[71]和 GRU[72]是两种最常用的 RNN 变种, 例如 Jiang 等人[12]和 Wang 等人[20]都使用 LSTM 来提取代码变更的特征, 而 Panthaplackel 等人[23]和 Xu 等人[13]使用的则是 GRU. RNN 常常通过添加双向 (bidirectional) 序列遍历和注意力 (attention) 机制来进一步增强其建模长序列和抽取语义信息的能力, 例如 Jiang 等人[12]利用注意力机制从编码器 LSTM 输出的特征向量序列中更好地抽取代码变更中与当前解码时刻相关的语义信息, 而 Liu 等人[22]则使用了双向 LSTM 以更好地保留代码变更中上下文信息.

CNN. 卷积神经网络 (convolutional neural network, CNN) 能够有效地捕获文本序列中的局部信息且计算效率优于 RNN, 很早就被用于文本分类等任务[73]. 一些代码变更表示学习的相关工作也采用了 CNN 作为表示学习模型[4,70]. 例如 Ye 等人[70]使用一维 CNN 直接从 git diff 中学习代码变更的分布式表示, 而 Hoang 等人[4]则设计了一种层次化 CNN 结构, 分别使用一维和二维 CNN 从代码变更中抽取包含层次化信息的代码变更表示.

Transformer. 近年来基于自注意力的 Transformer 模型逐渐流行[74], 一些研究者也开始探索基于 Transformer 来构建以代码变更的序列化表示作为输入的表示学习模型[10,14,16,24,27,38,52]. 相比于传统的 CNN 和 RNN 模型, Transformer 能够通过注意力机制更好地捕获序列中的长期依赖, 从而有更强的表达能力, 但其同时在长序列上的计算复杂度比 RNN 和 CNN 更高[74]. 例如 Lin 等人[10]基于 Transformer 构建了 3 种不同的表示学习模型结构, 并在软件制品可追溯性恢复任务上取得了比 LSTM 和 GRU 等 RNN 模型更好的性能.

其他表示学习模型. 除此之外, 一些研究者还探究了其他模型在代码变更表示学习上的效果. 例如 Loyola 等人[33]直接使用注意力机制从代码变更序列化表示的词嵌入序列中提取代码变更表示; Wang 等人[5]通过深度置信网络 (deep belief network, DBN)[75]重建输入的方式来直接学习代码变更表示. 也有研究工作同时使用多种网络结

构来提取代码变更表示以结合多个表示学习模型的优点, 例如 Shi 等人[29]和 Mi 等人[35]都同时使用了 CNN 和 LSTM 以结合两者在局部特征提取和长期依赖捕捉上的优势.

3.2 基于显式信息交互的结构化代码变更表示学习

代码的自然性[76]使得很多为自然语言设计的表示学习方法也能被用于代码变更表示学习中. 但代码和自然语言之间也存在不容忽视的差异. 例如相比自然语言, 代码具有强结构性、长依赖、可执行性等特点[60]. 若直接利用自然语言的表示方法对代码变更进行表示学习, 可能会因为缺少对代码中的结构信息 (例如嵌套结构和条件控制结构等) 的建模或难以捕获标识符之间的长依赖 (例如标识符的声明和使用) 而无法有效应对相关下游任务. 另外, 对代码可执行性特点的忽略可能会导致代码变更相关的下游任务模型产生不符合语法的错误代码, 影响模型在代码生成任务中的实际性能.

鉴于此, 一些研究者尝试以显式信息交互的方式从变更前、后的代码中提取代码变更的结构化表示. 本文将这类方法称为基于显式信息交互的结构化代码变更表示学习方法. 这类方法通常将变更前后的代码解析成 AST 的形式, 以分析代码中元素的依赖和调用关系或作为结构化表示的重要组成部分等, 并进一步地将代码变更表示为变更树或变更图等结构化的输入形式以保留代码变更中的结构信息. 然后, 这些结构化的输入将被转化为表示学习模型能够接受的输入形式 (例如 AST 路径的序列)[17,25,30,41,43]或直接使用以结构化数据为输入的表示学习模型提取代码变更表示[11,23,37,42]. 相比于代码变更的序列化表示, 代码变更的结构化表示有望更好地捕获代码变更中的语义信息和依赖关系. 本节将从代码变更的输入表示形式和表示学习模型两方面来归纳基于显式信息交互的结构化代码变更表示学习相关研究, 并汇总在表 2 中.

表 2 基于显式信息交互的结构化代码变更表示学习相关工作

学习方式	输入表示形式	表示学习模型	对应文献	具体下游任务
监督学习	AST路径	LSTM	Lozoya等人[25]	安全漏洞补丁识别
			Liu等人[17]	代码提交日志生成
			Brody等人[41]	代码编辑预测
	AST编辑操作序列	LSTM	Yao等人[30]	代码编辑迁移
	AST变更子树	TBCNN	Ni等人[42]	缺陷引发原因分类
	代码变更图	GGNN	Yin等人[11]	代码编辑迁移
			Panthaplackel等人[37]	代码编辑迁移
			Panthaplackel等人[23]	即时代码-注释不一致检测
		GCN	Dong等人[51]	代码提交日志生成
		GCN+Transformer	曹英魁等人[50]	代码编辑迁移
	代码变更依赖图	CNN	Meng等人[43]	代码提交意图分类

3.2.1 输入表示形式

几乎所有基于显式信息交互的结构化代码变更表示学习技术都直接或间接地基于 AST 构造代码变更的结构化表示. 相关研究中使用或提出的代码变更的结构化表示形式包括以下几种.

(1) AST 路径. AST 的叶节点 (终止节点) 用于表示代码中的标识符和字面值, 非叶节点 (内部节点) 则用于表示代码中的语法结构. AST 路径 (AST path) 定义为 AST 中的两个终止节点之间的最短路径, 是一种常用的代码变更结构化表示形式[17,25,41]. AST 路径两端的终止节点分别对应代码中的一个标识符; 中间则是 AST 中的内部节点序列, 反映了两个标识符在代码语法结构上的关系. 一些研究通过比对变更前后代码的 AST 路径集合, 提取出 AST 路径集合之间的差异以构建代码变更的结构化表示[25]. 还有一些研究先使用基于树差分 (tree differencing) 算法的 AST 差分方法[69], 例如 ChangeDistiller[53]或 GumTree[77], 来比对变更前后代码的 AST 并定位其中发生了变更的终止节点, 再分别提取发生变更的终止节点之间的 AST 路径来构建代码变更的结构化表示[17,41].

(2) AST 编辑操作序列. 除 AST 路径外, 一些相关研究也使用基于树差分的 AST 差分方法来比对变更前后代

码的 AST. 基于 AST 差分结果, Yao 等人[30]利用动态规划等算法计算将变更前代码 AST 转换为变更后代码 AST 的最短 AST 编辑操作序列, 作为代码变更的结构化表示. 可以看到, 与 AST 路径相似, AST 编辑操作序列也是从变更前后代码的 AST 中抽取结构化信息, 再将结构化信息表示为序列以输入到表示学习模型中.

(3) AST 变更子树. 同样是使用 GumTree 等基于树差分的 AST 差分方法比对变更前后代码的 AST, Ni 等人[42]则直接将比对结果表示为 AST 变更子树以输入表示学习模型中. 该 AST 变更子树中包含了变更相关的代码元素 (例如标识符、常量和操作符等)、代码元素的类型 (例如类型声明和函数调用等) 和变更操作等. 相关研究中也常采用一些特殊的模型 (例如 TBCNN 等[78]) 作为表示学习模型以从 AST 变更子树这一类树状结构的输入中提取出代码变更分布式表示.

(4) 代码变更图. 图由于其能建立任意节点之间的连接来建模代码元素间复杂的关系, 也常被用作代码变更的结构化表示. 代码变更依赖图通过将变更前后代码的 AST 中部分节点进行连接而构成, 也是一种常用的代码变更的结构化表示. 现有的相关研究都是根据变更前后的代码序列的对齐结果, 匹配变更前后代码 AST 中的节点, 连接匹配的节点, 并添加相应的编辑操作 (例如增加、删除) 作为连接边的属性, 从而将两个 AST 融合为一个代码变更图以输入到表示学习模型中[11,23,37,50,51]. 这种结构化表示不仅保留了变更前后代码的结构信息, 还通过新增的边显式提供了 AST 节点的变更信息. 特别地, Dong 等人[51]还额外将同一行中的单词节点连接起来, 以保留代码的序列信息.

(5) 代码变更依赖图. 除了连接变更前后代码 AST 中的部分节点来构成图形式的输入以外, Meng 等人[43]尝试根据代码内部实体和语句之间的相互依赖关系来将代码变更转换为图形式. 具体而言, 他们首先利用 ChangeDistiller 作为 AST 差分方法定位代码中实体和语句的变更, 然后使用静态程序分析方法基于变更的语句和实体来分析代码内部实体之间和语句之间的依赖关系. 基于这些依赖关系, 他们以变更的实体作为节点、以实体之间的关系作为边构建了变更依赖图 (change dependency graph, CDG). 变更依赖图中的所有节点和边的特征向量表示都是由人工设计的. 相比于代码变更图, 变更依赖图显式提供了变更前后代码实体之间和语句之间的依赖关系, 能够帮助发现变更前后依赖关系的变化.

3.2.2 表示学习模型

基于显式信息交互的结构化代码变更表示学习的相关工作中常用的表示学习模型包括 LSTM、TBCNN、GGNN 和 CNN 这几类.

(1) LSTM. 将代码变更转换为 AST 路径和 AST 编辑操作序列的相关工作都采用 LSTM 等 RNN 模型作为表示学习模型[17,25,30,41]. 具体地, 对于 AST 路径, 大部分相关研究首先通过分词和词嵌入方法将 AST 路径两端的标识符表示为特征向量, 然后利用双向 LSTM 等处理序列数据的模型学习 AST 路径的中间节点的特征向量. 两类特征向量通过特征并联或全连接层进行融合以获得 AST 路径的表示[17,25,41]. 若模型输入中存在多条 AST 路径, 这些路径的表示常常会由额外的组件进行融合, 例如额外的 LSTM[41]和注意力机制[17,25]等, 以获得代码变更的分布式表示. 对于 AST 编辑操作序列, Yao 等人[30]将 AST 编辑操作的操作符 (添加、删除等) 的嵌入、AST 中变更节点的嵌入 (由 GGNN 模型[79]计算获得) 和变更 AST 节点的值的嵌入等特征进行并联后, 输入到全连接层中获取 AST 编辑操作的特征向量表示. 然后他们同样采用 LSTM 模型从 AST 编辑操作的向量序列中提取出代码变更的分布式表示.

(2) TBCNN. TBCNN 即基于树的卷积神经网络 (tree-based convolutional neural network)[78], 在 Ni 等人[42]的研究中被用于从 AST 变更子树中提取代码变更的分布式表示. 与传统 CNN 的不同之处在于, TBCNN 的滑动窗口定义在输入的树结构 (即 AST 变更子树) 的子树上. 具体地, TBCNN 先通过编码层 (coding layer) 基于每个节点的特征向量编码其对应的父节点的特征向量, 然后通过树卷积层 (tree convolutional layer) 使用固定子树深度的特征检测器 (feature detectors) 滑动到树中的每一个子树上以提取子树的特征向量作为输出, 最后通过池化操作基于树中节点的特征向量提取出代码变更的分布式表示.

(3) GGNN 和 GCN. 在数据输入表示阶段将代码变更转换为代码变更图的相关研究通常使用图神经网络, 例如门控图神经网络 (gated graph neural networks, GGNN)[79]和图卷积网络 (graph convolutional network, GCN)[80]从图中学习和提取代码变更的分布式表示. 相关研究通常先使用图神经网络学习代码变更图中节点的分布式表示, 再使用一种加权平均算法[81]融合所有节点的表示来输出代码变更表示[11,23,37]或直接将所有节点的表示提供给下

游组件[51].

(4) GCN+Transformer. 曹英魁等人[50]使用了一种特别的编码器来学习代码变更表示. 他们的编码器首先利用GCN来捕捉代码变更图中的结构信息, 接着利用Transformer中的self-attention机制捕获代码变更中的长程依赖关系, 最后利用门机制(gating)在图节点的表示向量中引入该节点的单词字符信息和该节点对应的代码产生式的字符信息.

(5) CNN. Meng等人[43]将代码变更转换为变更依赖图后, 使用CNN作为表示学习模型. 具体而言, 他们首先利用启发式规则和广度优先遍历从变更依赖图中选出部分重要的节点和边, 然后利用CNN模型从选出的节点和边中抽取特征, 最后并联节点和边的特征作为代码变更表示.

4 基于隐式信息交互的代码变更表示学习

基于隐式信息交互的代码变更表示学习方法在将变更前后代码分别输入到表示学习模型之后, 才会对变更前后代码中的信息进行比对和融合(即, 隐式信息交互), 如图5所示.

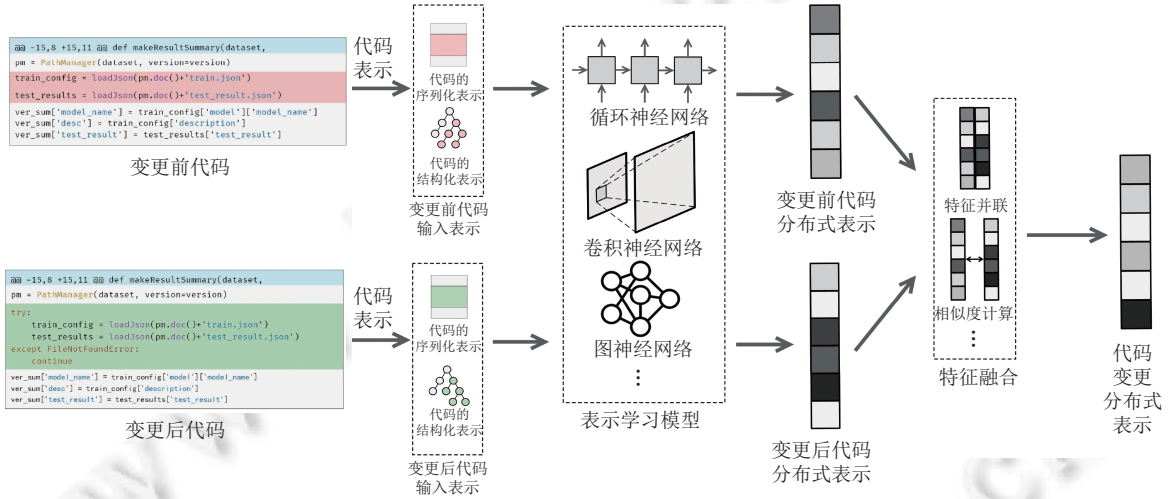


图5 基于隐式信息交互的代码变更表示提取流程

与基于显式信息交互的相关工作相比, 基于隐式信息交互的代码变更表示学习相关研究数量较少. 不同研究之间的差异主要体现在所用的表示学习模型和特征融合方法上. 因此, 本节也将从这两个维度出发对相关研究进行归类并总结, 并汇总在表3中.

表3 基于隐式信息交互的代码变更表示学习相关工作

学习方式 输入表示类型	表示学习模型	对应文献	具体下游任务
监督学习 序列化表示	CNN	Hoang等人[34]	缺陷补丁识别
		Li等人[28]	代码评审结果预测
	LSTM	Loyola等人[39]	代码提交日志生成
		Zhou等人[26]	安全漏洞补丁识别
		Tian等人[49]	补丁正确性评估
	GRU	Hoang等人[3]	代码提交日志生成、缺陷修复补丁识别、即时缺陷检测
	Transformer	Bai等人[19]	缺陷自动修复、代码提交日志生成
		Tian等人[49]	补丁正确性评估
	Doc2Vec	Tian等人[49]	补丁正确性评估
	LSTM+RNN (GRU或LSTM)	Le等人[44]	安全漏洞严重性预测

4.1 表示学习模型

通用模型. 目前, 绝大多数基于隐式信息交互的代码变更表示学习方法都使用代码的序列化表示作为表示学习模型的输入, 并未将变更前后代码分别表示为 AST 和图等结构. 在第 3.1.1 节介绍的以序列化数据作为输入的表示学习模型都可以不加修改地被用于基于隐式信息交互的表示学习方法中, 包括 CNN、RNN 和 Transformer 等. 本节将这些模型称为通用模型. 例如 Hoang 等人[34]和 Li 等人[28]都使用 CNN 来提取代码的特征表示, 而 Loyola 等人[39]和 Bai 等人[19]则分别使用了双向 LSTM 和多层的 Transformer 结构.

定制模型. 另外一些相关工作则基于代码和代码变更的特征以及下游任务的需求对通用的表示学习模型 (CNN、RNN 和 Transformer 等) 进行了改进. 例如 Hoang 等人[3]使用了层次化的双向 GRU 模型来保留代码中“符号-行-块-文件”的层次化变更信息. Zhou 等人[26]则将代码表示为符号序列并在代码语句之间插入特殊符号作为锚点, 将序列输入双向 LSTM 以后提取出锚点位置对应的输出作为相应语句的特征向量, 从而使得语句级代码表示中包含符号级的上下文信息. 一些工作[49]将代码变更 (例如 diff) 视为文本段, 并利用文本的分布式表示学习方法, 例如 Doc2Vec[82]来学习代码变更表示. 还有一些相关工作同时使用了多种神经网络模型来增强模型的表示能力, 例如 Le 等人[44]既利用 CNN 提取代码局部特征, 还利用 RNN 建立代码内的长期依赖, 以更好地提取代码的特征表示.

4.2 特征融合

特征融合旨在比对与融合表示学习模型输出的变更前后代码的特征表示, 以构建代码变更的分布式表示. 其中特征融合的方法会直接影响代码变更表示的质量, 进而影响下游任务的性能.

基于特征并联的特征融合方法. 相关研究中最常见的特征融合方法是特征并联 (concatenation), 即直接并联变更前后代码的特征向量作为代码变更的分布式表示. 特征并联产生的代码变更表示能够包含变更前后代码的特征, 同时由于简单有效, 在相关研究中得到了广泛的使用[3,28,34,44].

基于算术运算的特征融合方法. 一些特征融合方法利用简单的算术运算来比对和融合变更前后代码的特征表示, 例如向量间的余弦相似度[3,39,49]、向量的求和与求差等线性组合运算[3,19,49], 以获取代码变更的分布式表示. 这一类特征融合方法与基于特征并联的特征融合方法一样, 并不会引入需要学习的参数.

基于神经网络的特征融合方法. 还有一些相关工作则使用了包含参数的神经网络学习如何融合变更前后代码的特征表示, 且这些用于特征融合的神经网络会与表示学习模型一同进行训练. 例如 Zhou 等人[26]和 Hoang 等人[3]分别使用 CNN 和视觉问答任务 (visual question answering, VQA) 中常用的 NTN (neural tensor networks) 模型[83]来进行特征融合. 相比于简单的算术运算, 这些基于神经网络的特征融合方法具有更强的特征提取能力, 但同时也会引入额外的训练开销和过拟合风险.

基于多种特征融合方法. 此外, 也有相关工作同时使用多种特征融合方法比对融合变更前后的代码表示. 例如 Hoang 等人[3]在文件级代码特征上同时使用了并联、余弦相似度和线性组合等多种特征融合方法, 并将所有方法产生的特征向量并联来组成代码变更的分布式表示.

5 代码变更表示学习的应用

表示学习模型产生的代码变更表示可以被用于多种与代码变更相关的软件工程任务. 不同的下游任务可能侧重于代码变更中的不同信息, 所用的性能评估指标也可能存在差异. 本节将代码变更表示学习的下游任务分类为生成任务、分类任务和排序任务这 3 种, 分别对这 3 类任务进行了总结, 并汇总在表 4 中.

表 4 代码变更表示学习相关工作下游任务

任务类型	具体下游任务	相关文献	性能评价指标
生成任务	代码提交日志生成	Hoang等人[3], Jiang等人[12], Xu等人[13], Jung[14], Liu等人[15], Nie等人[16], Liu等人[17], Jiang [18], Bai等人[19], Wang等人[20], Pravilov 等人[32], Loyola等人[33], Loyola等人[39], Dong等人[51]	*BLEU*, *ROUGE*, *METEOR*, 完美匹配正确率

表 4 代码变更表示学习相关工作下游任务 (续)

任务类型	具体下游任务	相关文献	性能评价指标
	即时注释更新	Panthaplackel等人[21], Liu等人[22]	
	代码编辑迁移	Yin等人[11], Yao等人[30], Pravilov等人[32], Panthaplackel等人[37], 曹英魁等人[50]	
	代码编辑预测	Zhao等人[40], Brody等人[41]	
	冲突合并	Svyatkovskiy等人[31], Dinella等人[46]	
分类任务	即时缺陷预测	Hoang等人[3, 4], Wang等人[5], Loyola等人[6], Zeng等人[7], Gesi等人[8]	*Accuracy*, *Precision*, *Recall*, *F*-score, *AUC*, *CostEffort@L*, *PofB*20
	缺陷引发原因分类	Ni等人[42]	
	安全漏洞严重性预测	Le等人[44]	
	安全漏洞补丁识别	Lozoya等人[25], Zhou等人[26], Zhou等人[27]	
	缺陷修复补丁识别	Hoang等人[3, 34]	
	即时代码-注释不一致检测	Panthaplackel等人[23], Gao等人[24]	
	代码评审结果预测	Li等人[28], Shi等人[29]	
	代码提交意图分类	Meng等人[43]	
	代码变更质量评估	Mi等人[35]	
	补丁正确性评估	Tian等人[49]	
排序任务	软件制品可追溯性恢复	Ruan等人[9], Lin等人[10]	*Precision@K*, *Accuracy@K*, *MRR*, *MAP*
	即时缺陷定位	Loyola等人[45], Ciborowska等人[52]	
	代码评审意见推荐	Siow等人[36]	
	代码评审人推荐	Ye等人[70]	
	代码评审优先级排序	Hellendoorn等人[38]	

5.1 生成任务

代码变更的分布式表示可以被用于生成多种类型的文本. 根据生成的文本的类型, 可以将此类任务进一步分为自然语言 (NL) 生成任务和代码 (PL) 生成任务. 本节将分别介绍这两类生成任务和常用的评估指标.

5.1.1 自然语言生成任务

代码提交日志生成 (commit message generation) 旨在自动为版本管理系统中的代码提交生成描述其内容和意图的自然语言语句, 以减轻开发者手动编写代码提交日志的负担. 该任务的输入是代码提交中的代码变更, 输出是相应的代码提交日志. 研究者们已经提出了大量基于代码变更表示学习的方法来应对此任务[3,11−20,32,33,39,51]. 例如, Jiang 等人[12]提出的代码提交日志生成方法先将代码变更表示为 diff, 然后利用一个基于 LSTM 的编码器来学习代码变更的分布式表示, 最后使用基于 LSTM 的解码器来自动根据学到的分布式表示生成代码提交日志.

即时注释更新 (just-in-time comment update) 旨在帮助开发者在进行代码变更后自动对过时 (obsolete) 注释进行更新, 以减少过时注释的引入, 从而提升软件系统的鲁棒性和可维护性[21,22]. 其输入中不仅包含代码变更, 还包含变更前的代码注释, 其输出是变更后的代码注释. 例如 Liu 等人[22]基于双向 LSTM 模型和共同注意力机制构建能同时学习代码变更表示和变更前注释表示的模型, 并将两种表示同时输入到下游的 LSTM 解码器中自动生成变更后的注释.

5.1.2 代码生成任务

代码编辑迁移是研究较多的一种与代码变更表示学习相关的代码生成任务[11,30,32,37,50]. 该任务的输入包含作为模板的代码变更的分布式表示和需要应用该编辑的代码. 作为模板的代码变更表示是利用代码变更表示学习模型从相似的代码变更中提取的, 包含通用的变更模式 (例如变量名重构和 API 迁移等) 信息; 需要应用该编辑的代码的分布式表示则由一个专门的编码器从输入代码中提取. 基于代码变更表示和代码表示, 下游模型自动生成修改后的代码以实现代码编辑的迁移. 例如 Yin 等人[11]先利用自编码器从已有的代码变更中学习出代码变更表示,

然后分别采用了“序列到序列”和“图到树”两种解码器, 根据代码变更表示和需要应用该编辑的代码的表示来生成编辑后的代码序列或代码 AST. 代码编辑迁移有望实现代码变更模式的自动提取和应用, 从而降低开发者手动实现代码编辑规则或重复进行相似代码编辑的负担.

代码编辑预测旨在根据历史代码变更预测下一步可能的编辑操作[40,41], 类似于代码补全 (code completion) 任务[84]. 该任务的输入是某段代码的历史编辑序列, 例如开发者编写某段代码时逐个插入字符的编辑操作序列. 预测方法需要从历史变更序列中学习编辑操作的规律并输出对同一段代码的下一次编辑操作的预测. 例如 Brody 等人[41]将同一段代码的历史变更表示为 AST 路径的序列, 然后利用 LSTM 模型从 AST 路径序列中捕捉已执行的变更的特征表示, 最后基于此特征表示从待选的 AST 路径中选择一个作为下一次编辑操作的预测.

冲突合并是一种定义在协同开发场景下的代码生成任务[31,46]. 在协同开发的场景下, 不同的开发者可能会同时对相同的代码库进行不同的更改, 从而造成代码冲突. 这一任务旨在自动解决此类冲突并生成冲突合并后的代码. 此任务的输入包含 3 段不同的代码, 即一段变更前的代码 (O) 和两段不同的变更后的代码 (A,B), 输出为两段变更后代码合并之后的代码. 相关工作中通常先将输入中的 3 段代码组合成两个代码变更, 接着利用代码变更表示模型分别获取它们的分布式表示, 然后同时输入到下游模型中生成冲突合并结果. 为了简化该任务, 相关工作常常会预定义若干个冲突合并模式, 并通过逐步预测最恰当的冲突合并模式来生成冲突合并结果. 例如 Svyatkovskiy 等人[31]定义了 9 种不同的冲突合并模式 (例如保留 A 中的符号、保留 B 中的符号等) 以合并产生冲突的符号, 并通过对每一个产生冲突的符号进行冲突合并模式的预测来逐步生成冲突合并结果. 值得注意的是, Svyatkovskiy 等人[31]指出, 他们定义的 9 种冲突合并模式只能覆盖大约 85% 的冲突合并情况. 这意味着通过预测这 9 种冲突合并模式无法完美解决所有情况下的代码冲突.

5.1.3 常用评估指标

● *BLEU*[85] 是机器翻译领域常用的性能评估指标[86], 主要通过评估机器翻译方法生成的序列与参考序列之间的子序列相似度来量化评估翻译质量. 该指标被定义为两个序列之间的 n-gram 匹配精准率 (*Precision*) 的几何平均值, 同时使用惩罚因子 (*BP*) 来避免短序列由于更容易匹配而获得偏高的分数, 如下:

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

其中, w_n 为加权权重, p_n 为 n-gram 的匹配精准率. *BP* 的定义如下:

$$BP = \begin{cases} 1, & \text{if } c > r \\ \mathrm{e}^{1-r/c}, & \text{if } c \leqslant r \end{cases} \tag{2}$$

其中, c 是生成的序列的长度, r 是参考序列的长度. *BLEU* 的取值范围为 0–1, 值越大说明生成的序列越接近参考序列. *BLEU* 被广泛用于代码提交日志生成[3,12−20,32,33,39]和即时注释更新[21,22]等任务的性能评估.

● *ROUGE*[87]是文本摘要领域常用的一类性能评估指标[88]. 相比于 *BLEU*, *ROUGE* 依赖于生成序列和参考序列之间 n-gram、词序列和词对的共现, 且更关注于召回率指标. 其中 *ROUGE-N* ($N = 1,2$) 和 *ROUGE-L* 是最常用的 *ROUGE* 指标, 分别依赖于 n-gram 的共现和最长公共序列 (LCS). *ROUGE* 主要被用于评估代码提交日志生成任务的性能[15−17,19,20].

● *METEOR*[89]是另一种机器翻译常用的性能评估指标. 相比于 *BLEU*, *METEOR* 考虑了完全匹配、词干 (stem) 匹配、同义词 (synonym) 匹配和释义 (paraphrase) 匹配等多种匹配方法来精确匹配机器翻译方法生成的序列和参考序列. *METEOR* 通过计算 F-score 来考虑召回率并同时使用了一个惩罚函数来惩罚错误的词序. 与 *BLEU* 类似, *METEOR* 也被广泛用于评估代码提交日志生成[13,16,17,20,33,39]和即时注释更新[21]等任务.

● 完美匹配正确率衡量了生成的序列跟参考序列完全一致的样本的比例, 在一些代码变更相关的生成任务中, 特别是代码编辑迁移和代码编辑预测等代码生成任务中被频繁使用[11,32,37,40,41,46]. 相比于 *BLEU* 等性能指标, 完美匹配正确率更加严格, 不考虑不完全匹配的生成序列 (例如代码生成任务中生成的存在语法错误的代码序列), 能够评估方法的性能下界, 帮助了解方法的实用性.

5.2 分类任务

代码变更表示学习的下游分类任务可以按照其主题被分为 3 大类: 代码缺陷和漏洞相关的分类任务、代码注释相关的分类任务和其他分类任务. 本节将会依次介绍这 3 大类代码变更相关分类任务和常用的评估指标.

5.2.1 代码缺陷与漏洞相关的分类任务

即时缺陷预测 (just-in-time defeat prediction) 旨在预测一个代码变更中是否引入了缺陷[3−8]. 即时缺陷预测工具能够帮助开发者将代码审查的精力花费在更可能引入缺陷的代码变更上. 在该任务中, 即时缺陷预测方法先利用代码变更表示学习模型获取代码变更的分布式表示, 然后结合从代码提交日志中提取出的特征表示分析代码变更中存在缺陷的可能性. 例如 Hoang 等人[4]分别使用两个 CNN 模型来提取代码变更的分布式表示和代码提交日志的分布式表示, 然后将两个特征向量并联后输入到下游分类器中预测代码提交中是否存在缺陷.

缺陷引发原因分类旨在基于缺陷补丁中的代码变更对引发该缺陷的原因 (例如算法错误和模块间接口错误等) 进行分类[42]. 在开发者修复缺陷时, 缺陷引发原因分类方法可以帮助开发者检索不同类型的缺陷来寻找最佳的缺陷解决方案. 该任务一般利用表示学习模型 (例如 Ni 等人[42]使用的 TBCNN 模型[78]) 从缺陷对应的修复补丁中提取代码变更的分布式表示并输入到下游分类器以预测该缺陷补丁对应的缺陷引发原因.

安全漏洞严重性预测旨在预测一个安全漏洞的机密性、完整性和严重等级等安全漏洞严重性指标, 以帮助开发者在检测到安全漏洞之后规划安全漏洞修复的优先顺序[44]. 该任务以引入安全漏洞的代码提交作为输入, 一般利用表示学习模型 (例如 Le 等人[44]使用的 CNN 和 GRU 模型) 提取该代码提交中代码变更的分布式表示, 然后将其输入到下游的多个分类器中以同时预测多个安全漏洞严重性指标.

安全漏洞补丁识别, 也被称为“安全相关代码提交识别 (security-relevant commit identification)”, 旨在从一系列代码提交中识别出与漏洞修复或安全问题修复相关的代码提交 (补丁), 以提醒软件使用者关注和及时应用这些补丁, 保证系统安全[25−27]. 在此任务中, 输入一般同时包含代码变更和代码提交中的文本信息 (例如代码提交日志). 通常该任务在利用表示学习模型提取出代码变更表示之后, 将其代码提交中文本信息的特征一并输入到下游任务的分类器中, 以判断代码提交是否包含安全漏洞补丁或与安全问题相关. 例如 Zhou 等人[26]使用了两个同时包含了 CNN 和 LSTM 的神经网络模型来分别提取代码变更和代码提交日志的分布式表示, 并通过集成学习模型结合两种特征的分类结果以预测代码提交中是否包含安全漏洞补丁.

缺陷补丁识别与安全漏洞补丁识别任务十分相似, 仅在任务目标上稍有区别. 具体而言, 缺陷补丁识别任务旨在判断代码提交中是否包含修复软件缺陷的补丁, 而非是否与安全问题相关[3,34]. 例如 Hoang 等人[34]先利用层次化 CNN 模型作为表示学习模型提取 Linux 内核代码提交的代码变更表示, 然后利用下游的线性分类器基于代码变更表示预测该代码提交中是否包含缺陷补丁. 他们进而根据缺陷补丁识别结果来判断该代码提交是否应该及时推送给 Linux 稳定版用户 (即是否是 Linux 稳定版补丁), 以帮助提升系统安全性.

补丁正确性评估旨在预测自动程序修复工具生成的、能通过所有测试用例的似真补丁 (plausible patch) 能否正确修复软件缺陷, 以对补丁进行筛选和排序. 例如, Tian 等人[49]提出基于 BERT[68], CC2Vec[3]和 Doc2Vec[82]等表示学习模型学习补丁中代码变更的分布式表示, 然后利用下游的分类器预测补丁的正确性.

5.2.2 代码注释相关的分类任务

即时代码-注释不一致检测 (just-in-time code-comment inconsistency detection) 旨在基于代码变更预测代码注释是否“过时”, 例如由参数增删或返回值重命名导致的注释跟代码不匹配[23,24]. 给定代码变更和对应代码的变更前注释 (原注释), 该任务一般利用代码变更表示学习技术提取代码变更的分布式表示, 将其与原注释中提取的特征信息一同输入到分类模型中预测在代码变更后原注释是否已经过时. 该任务与基于代码变更的其他分类任务的显著不同在于, 模型不仅需要捕获代码变更中的信息, 还需要将它们与原注释中的信息进行“比较”以判断原注释是否已经过时. 除了面向常规注释的即时代码-注释不一致检测[23], 还有一些研究工作专门针对检测 TODO 注释是否“过时”以清除不必要的 TODO 注释[24].

5.2.3 其他分类任务

代码评审结果预测的目标是在开发者提交代码变更到版本管理系统时, 根据代码提交中的信息来预测其是否

能够被接受并合并, 从而为开发者提供关于代码提交质量的即时反馈[28,29]. 相关研究工作也是将代码变更表示学习模型与下游分类器结合起来进行预测. 例如 Shi 等人[29]同时使用 CNN 和 LSTM 模型从需要审核的代码提交中提取出变更前后代码的特征表示, 然后将其输入到自编码器中学习代码变更的分布式表示, 最后使用分类器基于该表示输出代码评审结果.

代码提交意图分类以代码提交中的代码变更为输入, 旨在预测代码提交的意图 (例如缺陷修复或功能添加等) 以帮助开发者理解代码提交并促进代码评审等协同开发任务[43]. 例如, 为了预测代码提交意图, Meng 等人[43]先利用程序分析技术将代码变更转化为变更依赖图的形式, 然后使用两个 CNN 分别提取图中的节点和边的特征, 最后将两个 CNN 输出的特征并联后输入到下游的全连接层网络进行意图预测.

代码变更质量评估旨在预测输入的代码变更的质量. 相关研究工作同样通过结合代码变更的分布式表示和下游预测模型来应对此任务[35]. 例如 Mi 等人[35]先使用一个同时包含 CNN 和 LSTM 模型的表示学习模型抽取变更前后代码的特征表示, 然后使用一个自编码器对变更前后代码的特征表示加以转换, 并根据转换后两个代码特征表示的距离来输出代码变更质量的预测. 值得一提的是, 该研究中用到代码变更的质量标签并非手动标注的, 而基于 Stack Overflow 等问答平台中包含的代码变更及其得票数自动标注的.

5.2.4 常用评估指标

常见的分类任务性能评估指标, 例如准确率 (*Accuracy*)、精准率 (*Precision*)、召回率 (*Recall*) 和 *F*-score 等也适用于代码变更表示学习的分类任务, 此处不再赘述. 本节主要关注相关工作中用到的其他分类指标.

- *AUC* 定义为 ROC 曲线 (receiver operating characteristic curve) 与坐标轴围成的面积, 是一种阈值无关的评估指标, 因此被应用在很多数据标签不平衡的分类任务上, 例如即时缺陷预测[3,4,7,8]和代码评审结果预测[28,29]等.

除此之外, 一些相关工作还提出和使用了一些更贴近应用场景的“代价感知 (effort-aware)”评估指标. 这些指标会评价方法在限定额度的代价条件下 (例如需要检查的代码行数) 的表现和价值. 相关工作中用到的代价感知指标包括:

- *CostEffort@L* 定义为最多审查 *L* 行代码的情况下能够找到的正例样本的比例. 相关工作在计算该指标之前会按照模型输出的分类置信度排序样本, 然后根据置信度从高到低来遍历样本直到总计 *L* 行代码被审查. 通过以审查的代码行数为代价, 该指标隐含地提高了模型在一些较大规模代码改动上的预测成本. 这一指标已被用于评估安全漏洞补丁识别任务[27], 指标定义中的正例即对应安全漏洞补丁.
- *PofB*20 与 *CostEffort@L* 类似, 定义为根据模型输出的分类置信度从高到低的顺序人工检查 20% 总代码行数的情况下能够检查出的缺陷的比例. 这一指标已在即时缺陷预测任务中得到了使用[5].

5.3 排序任务

排序任务通常需要预测代码变更与多个其他类型的实体 (例如缺陷报告) 的相关度, 并按照相关度从高到低排序其他类型的实体. 本节将会介绍与代码变更表示学习相关的排序任务及其常用的评价指标.

5.3.1 相关工作中的排序任务

软件制品可追溯性恢复 (software artifact traceability recovery) 旨在恢复软件开发过程中软件制品之间缺失的链接. 其中, commit-issue 链接恢复是一种常见的子任务, 通常被视为排序任务[9,10]. 该任务的输入包含同一个项目中的代码提交 (commit) 和 issue. commit-issue 链接恢复方法通常利用表示学习模型提取代码提交中代码变更的分布式表示, 通过计算代码变更表示与备选 issue 的特征表示之间的相关度从高到低排序备选 issue, 以帮助恢复代码提交与 issue 之间的链接关系. 例如 Lin 等人[10]基于 BERT[68]构建了表示学习模型并分别提取出代码提交中代码变更的特征表示和 issue 中自然语言描述的特征表示, 然后将两者通过特征并联等方式融合为一个特征向量以输入到分类器中预测两者的相关度.

即时缺陷定位旨在预测缺陷报告与代码变更之间的相关度以定位最有可能引入缺陷报告中所描述缺陷的代码变更[45,52]. 该任务的输入包含缺陷报告和可能引入该缺陷的代码变更. 跟其他分类任务类似, 相关研究通常使用表示学习模型提取出代码变更和缺陷报告的特征表示, 然后将二者融合为一个特征向量以输入到下游的相关度预

测模型中. 基于预测的相关度即可将备选代码变更进行排序, 以定位最有可能引入了相应缺陷的代码变更. 例如 Loyola 等人[45]使用代码变更族谱和双向 LSTM 抽取代码变更分布式表示, 使用另一个双向 LSTM 抽取缺陷报告的特征, 两者通过特征并联和特征向量求差等方法融合为一个单一的特征向量, 然而输入到下游的 RankNet[90]模型中输出相关度预测.

代码评审意见推荐 (code review comment recommendation) 是一种代码协同开发场景下的排序任务, 旨在为提交到版本管理系统中的代码变更推荐最合适的评审意见, 以帮助开发者在实际代码评审之前获得一些参考意见以即时修复部分问题, 从而加快代码评审过程[36]. 该任务以代码变更和大量备选评审意见作为输入, 相关方法使用表示学习模型分别提取代码变更和评审意见的分布式表示, 然后融合二者的特征表示并利用下游模型预测相关度. 例如 Siow 等人[36]同时使用了注意力机制和双向 LSTM 模型提取代码变更和评审意见的分布式表示, 然后将两者并联后输入到下游模型中预测相关度, 并根据相关度推荐合适的评审意见.

代码评审人推荐旨在预测不同的备选评审人评审给定代码变更的适合程度, 并根据适合程度推荐最合适的候选人[70]. 该任务的输入只包含代码变更, 通常在使用表示学习模型提取代码变更的分布式表示后直接将其输入到下游模型中生成待选评审人员的适合程度 (相关度), 并根据适合程度排序备选评审人员. 也有一些相关工作引入了额外的启发信息帮助模型训练. 例如 Ye 等人[70]在损失函数中引入了额外的惩罚项, 以考虑备选评审人是否修改过代码变更中某些文件及是否审核过变更提交者提交的其他代码变更.

代码评审优先级排序旨在评估代码变更的评审优先级, 并根据优先级结果对所有代码变更进行排序以帮助评审人员定位并优先检查最需要评审的代码块, 提升代码评审效率[38]. 该任务跟代码评审人推荐任务类似, 核心也是利用预测模型基于代码变更的分布式表示对评审优先级进行预测. 例如 Hellendoorn 等人[38]将一次代码变更分割为多个变更代码块并分别预测每个变更代码块的评审优先级, 然后根据评审优先级排序并定位最需要被评审的变更代码块.

5.3.2 常用评估指标

排序任务相比于分类任务通常更加关注相关实体在相关度排序结果中的具体排名, 并且基于排名来计算各项性能评估指标. 常用的排序任务相关评估指标包括:

- *Precision@K* 主要衡量排序结果的前 K 位 (Top-K) 的精准率, 即, 与请求相关的对象在排序结果前 K 位中出现的比例. *Precision@K* 值越高, 代表排序结果前列中越有可能出现与请求相关的对象. 该指标在一些排序相关任务中被用于评估排序结果, 例如软件制品可追溯性恢复[10]等.
- *Accuracy@K* 在 Ye 等人[70]的工作中, 被定义为“至少存在一个相关对象位于排序结果前 K 位的请求在所有请求中所占的比例”, 以评估代码评审人推荐任务的排序结果质量.
- *MRR*, 即“平均排名倒数 (mean rank reciprocal)”, 是一种专门统计排序任务的指标. 定义为:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{Rank_{x_i}} \tag{3}$$

其中, $Rank_{x_i}$ 表示正例样本点 x_i 的排名. 该指标关注所有正例样本的相关度排名, 越高的 *MRR* 值代表模型能够越好地将正例样本排到前列. 该指标已被用于评估软件制品可溯性恢复[9,10]和代码评审人推荐[70]等任务.

- *MAP* 即“平均精确率均值 (mean average precision)”. 对于一个排序请求, 其排序结果中所有实体的排名精确率的平均值被定义为精确率均值 (average precision, AP). *MAP* 则定义为测试集中所有请求的精确率均值的均值. 相比于 *MRR* 只关注正例样本, *MAP* 关注所有正负例实体样本的排序结果. 与 *MRR* 一样, *MAP* 也常被用于评估软件制品可溯性恢复[10]和代码评审人推荐[70]等任务的性能.

6 讨 论

本节对比基于显式信息交互和隐式信息交互的代码变更表示学习, 并对代码变更表示学习和代码表示学习的关系进行讨论.

6.1 基于显式信息交互和基于隐式信息交互的代码变更表示学习的对比

表 5 总结了基于显式信息交互和基于隐式信息交互的代码变更表示学习的差异和优缺点. 在处理过程方面, 二者主要有 3 点不同: 1) 输入表示形式不同: 基于显式信息交互的方法将代码变更转换为特殊形式的单一输入 (例如 diff) 供表示学习模型编码, 而基于隐式信息交互的方法则将代码变更分割为变更前代码及变更后代码这两个输入. 2) 信息交互发生的阶段不同: 对于显式信息交互, 变更前后代码之间的比对和融合发生在数据输入表示阶段; 而对于隐式信息交互, 比对和融合则发生在特征融合阶段. 3) 信息交互的方式不同: 显式信息交互利用基于领域知识和任务需求人工设计的规则和算法来对变更前后代码中的信息进行比对和融合, 而隐式信息交互则基于神经网络组件和下游任务的反馈信息从训练数据中自动学习应该如何比对和融合变更前后代码中的信息.

表 5 基于显式信息交互和隐式信息交互的代码变更表示学习的对比

方法	处理过程			优点	缺点
	输入表示形式	信息交互阶段	信息交互方式		
基于显式信息交互的代码变更表示学习	代码变更的单一表示(例如diff)	数据输入表示阶段	基于人工设计的规则和算法	能有效抽取显式特征(例如变更的标识符)	在特征难以显式提取的任务和大规模数据集上泛化性受限
基于隐式信息交互的代码变更表示学习	变更前代码和变更后代码	特征融合阶段	基于学习	能自动学习隐式特征(例如语义信息)	对数据质量、数据规模和模型设计更敏感

在实施效果方面, 仅有极个别代码变更表示学习相关研究对比了这两类方法. 例如 Li 等人[28]在基于 CNN 模型和隐式信息交互进行代码评审结果预测时, 以将代码变更直接表示为 diff 形式并输入模型进行预测的显式信息交互方法作为基线, 其实验结果表明基于隐式信息交互的方法在 *F*-score 和 *AUC* 指标上较基于显式信息交互的方法有明显优势. 然而在一项有关代码提交日志生成的研究中, Loyola 等人[39]既采用了将代码变更表示为 diff 形式的显式信息交互方法, 也尝试了利用 LSTM 分别提取变更前后的代码特征并基于余弦相似度进行融合比对的隐式信息交互方法, 其结果表明基于隐式信息交互的方法的整体性能落后于基于显式信息交互的方法. 这表明基于显式信息交互和基于隐式信息交互的方法并无绝对的优劣之分, 其实施效果主要受下游任务的特点及算法设计的影响.

相比之下, 基于显式信息交互的方法由于利用了领域知识和人工设计的规则, 能有效抽取代码变更中的显式特征 (例如被变更的标识符), 在规模较小的数据集或具有明显的可抽取特征的任务上有望取得更好的效果, 但常过于聚焦显式特征, 在特征难以显式抽取的任务和大规模数据集上可能泛化性稍差; 基于隐式信息交互的方法由于主要从数据中自动学习如何比对和融合代码变更中的信息, 因此对数据质量、数据规模和模型设计更加敏感, 但具备学习隐式特征 (例如语义特征) 的能力, 有望在特征难以显式提取的任务和大规模数据集上取得更好的效果.

从微观层面, 代码变更的显式信息交互与隐式信息交互, 难以兼容. 如表 5 所示, 二者的处理过程存在较大差异: 前者接收单一输入, 而后者同时接收变更前后的代码; 前者的表示学习模型直接输出代码变更表示, 而后者还需额外的特征融合操作. 因此二者难以融合为一种信息交互方式. 然而从宏观层面, 研究者可以对代码变更同时应用这两种信息交互方式以抽取两种不同的代码变更表示. 融合这两种代码变更表示有望进一步提升下游任务的性能.

此外, 从第 3 节和第 4 节可以看出, 基于显示信息交互和基于隐式信息交互的代码变更表示学习的具体任务有所区别. 这一现象可能的原因包括: 1) 沿用早期工作中的做法. 例如, 代码提交日志生成任务的早期研究者将该任务建模为“将 diff 翻译为代码提交日志”的“序列到序列 (sequence-to-sequence)”任务, 他们构建的公开数据集也都将代码变更存储为 diff 形式. 因此该任务的研究工作大都采用基于显式信息交互的方法来学习代码变更表示[12−16]. 2) 适应不同任务的不同信息偏好. 例如, 代码编辑迁移任务需要代码变更表示学习方法有效捕捉细粒度、具体的代码变更, 因此基于显式信息交互的方法更为适用[11,30,37,50]. 而安全漏洞严重性预测任务往往要对变更前后代码的语义信息进行对比, 因此该任务的研究者偏向于利用基于隐式信息交互的方法从数据中学习如何比对和融合信息[44]. 不过, 当前绝大多数代码变更表示学习相关任务只采用了两类方法中的一类, 缺乏对两类方法优劣的

系统性比较. 因此, 基于显式信息交互和基于隐式信息交互的方法的详细对比在未来仍需进一步探索和梳理.

6.2 代码变更表示学习与代码表示学习的关系

由于代码变更可被视为变更前代码和变更后代码的组合, 代码变更表示学习与代码表示学习既密切关联也存在明显差异: 在输入输出方面, 代码表示学习旨在学习单一代码片段的分布式表示, 以捕捉代码的句法、结构和语义信息, 代码变更表示学习旨在学习两段代码之间差异的分布式表示, 以捕捉两段代码在句法、结构和语义信息上的不同. 二者输出相似, 但输入不同, 学习的侧重点也有区别. 在方法方面, 代码表示学习方法可被视为代码变更表示学习方法的基础. 基于显式信息交互的代码变更表示学习方法往往会借鉴代码表示学习方法, 并基于代码变更的特点 (例如包含两段代码、侧重代码差异等) 对数据输入表示和表示学习模型进行改进, 以适配代码变更表示学习的需求. 例如, Liu 等人[17]基于代码表示学习中将代码表示为 AST 路径的思想, 提出将代码变更也表示为 AST 路径. 不过由于代码变更表示学习侧重于代码中变更的内容, 他们仅提取被增加/被删除的叶节点之间的 AST 路径. 基于隐式信息交互的代码变更表示学习方法则直接利用代码表示学习模型分别学习变更前后代码的分布式表示, 然后对二者的分布式表示进行特征融合.

7 代码变更表示学习的挑战与机遇

本节主要从代码变更表示学习的关键挑战和潜在研究机遇两方面出发, 展望了此领域未来的研究趋势.

7.1 关键挑战

7.1.1 结构信息利用困难

目前大多数代码变更表示学习的相关研究都基于代码变更中的序列化信息, 对提取和利用结构化信息的研究相对较少. 代码变更中的结构化信息有利于表示学习方法捕捉变更前后代码内部的依赖关系和代码间的匹配关系, 从而提升学到的代码变更表示的质量与下游任务的性能. 但目前代码变更中结构化信息的抽取仍存在困难, 包括: 1) 结构化信息提取工具的鲁棒性有限[25]. 一些重要软件工程任务需要对可能存在编译错误的代码或者不完整的代码片段进行处理, 例如即时缺陷预测. 而目前能准确解析编译错误的代码或不完整代码片段的、鲁棒的 AST 构建工具相对较少. 2) 基于结构化信息的表示学习模型计算复杂度较高. 例如, 在将代码变更表示为图之后, 现有研究工作主要使用 GGNN 等图模型来提取图整体的特征作为代码变更的分布式表示[11,23,37], 然而 GGNN 等模型的计算复杂度较高, 其计算开销将会随图中的点的个数增加而快速上升, 难以适用于大规模的图表示. 这些困难使得利用代码变更中的结构化信息面临较大的挑战.

7.1.2 内在评估方法缺乏

现有研究工作通常通过下游任务的性能指标来间接地评估学习到的代码变更表示的质量, 这种评估方式被称为外在评估 (extrinsic evaluation). 外在评估与具体任务高度耦合, 最终的结果跟任务类型、任务目标、下游任务模型和使用的数据集等多种因素密切相关, 难以直接反映出代码变更表示学习技术的性能. 目前, 尚未有研究工作不依赖下游任务直接评估代码变更分布式表示的质量, 即进行内在评估 (intrinsic evaluation). 内在评估方法的缺乏使得研究者不同的代码变更表示学习技术难以被公平比较, 也令研究者在应用代码变更表示学习技术时难以准确定位性能瓶颈. 因此, 未来的研究工作需要进一步探究代码变更表示学习的内在评估指标与方法.

7.1.3 基准数据集缺失

基准数据集能够促进新技术的开发, 帮助快速、全面、公平地比较不同技术的优缺点, 为技术的应用提供重要的参考. 例如, ImageNet[91]极大地推进了计算机视觉领域的发展, GLUE 基准数据集[92]对于自然语言处理领域大规模预训练模型的提出和发展功不可没. 在软件工程领域, Husain 等人[93]也提出了 CodeSearchNet 这一基准数据集, 促进了代码表示学习技术的发展. 但这一基准数据集主要着眼于代码和自然语言的语义匹配, 而非代码变更表示学习. 代码变更表示学习的下游任务多样, 不同任务的信息需求也各有侧重. 因此, 目前大多数研究都自行收集并构建任务相关的数据集. 这些数据集大多规模较小、质量参差不齐、缺乏维护, 随着时间推移难免过时[47]. 大规模、高质量、多任务、持续维护的基准数据集的缺失仍是代码变更表示学习领域的一个严峻的挑战.

7.1.4 代码变更数据的长尾分布

在与代码变更表示学习相关的软件工程任务中, 代码变更数据分布通常是高度不平衡的. 一方面, 代码变更数据在任务相关的数据类别上存在长尾分布. 例如对于即时缺陷预测[4,5,8]、安全漏洞补丁识别[26,27]等任务, 正样本(即引入缺陷、与安全相关的代码提交) 只占数据集的很小一部分, 大部分代码变更都是负样本. 如果不加以处理, 这种数据类别上的长尾分布问题会导致模型产生对负例样本的偏置, 例如模型始终预测样本是负样本也能降低训练损失. 另一方面, Gesi 等人[8]还发现代码变更在度量元 (metric) 层面也存在长尾分布. 具体而言, 他们按照代码变更改动行数和代码变更改动文件数等多个度量元来划分数据, 发现代码变更数据在这些度量元上也存在长尾分布, 且这一长尾分布现象会对 DeepJIT[4]和 CC2Vec[3]这两个即时缺陷预测模型的训练产生负面影响. 目前, 上述长尾分布问题仍未得到充分地研究, 还是代码变更表示学习领域的重要挑战.

7.1.5 跨项目和跨语言学习

代码变更表示学习技术需要在训练集上训练, 在测试集上评估. 常见的划分训练集和测试集的方法包括混合所有软件项目的代码变更数据, 然后随机或根据时间戳划分, 以及将每个项目中的一部分数据划分到训练集, 另一部分划分到测试集. 一些研究者也考虑了跨项目 (cross-project) 的实验设定[5,38,48], 即将一些项目的数据全部划分到训练集, 另一些项目划分到测试集. 这样的实验设定更加注重学到的代码变更表示的泛化性和可迁移性. 但实验显示代码变更表示模型在跨项目设定下的效果总是不如混合所有项目或同项目设定下的效果[48], 这表明跨项目的代码变更表示学习存在更大的困难与挑战.

另一方面, 现有的大多数代码变更表示学习工作都只在单一编程语言上评估了所提出技术的性能, 忽略了技术在其他编程语言上的泛化能力. 一些研究工作通过在跨语言设定下开展实验, 发现学到的代码变更分布式表示在不同编程语言上存在一定的可迁移性, 但现有代码变更表示学习技术在跨语言场景下的表现相比于同语言场景还存在较大差距[31,48]. 因此, 如何在跨语言场景下有效学习代码变更的分布式表示也是代码变更表示学习的一大挑战.

7.1.6 基于显式信息交互的代码变更表示学习尚存挑战

当前, 基于显式信息交互的代码变更表示学习方法面临的挑战主要包括: 1) 过于聚焦变更的代码, 对代码变更的整体语义和变更代码及其上下文间关系的学习不够. 这一方面是由于此类方法将代码变更处理为单一的输入表示形式后, 会利用标记或特殊符号强调变更的代码, 人为地引入先验知识. 另一方面是因为前人工作中常用的表示模型 (例如 LSTM 和 GNN) 在捕捉长程依赖和全局上下文方面存在局限. 2) 缺乏对不同输入表示形式和不同表示学习模型的系统性比较.

7.1.7 基于隐式信息交互的代码变更表示学习尚存挑战

现有基于隐式信息交互的代码变更表示学习方法面临的挑战主要包括: 1) 难以聚焦细粒度的变更信息. 现有相关工作通常将变更前代码和变更后代码分别编码为单一的分布式表示, 然后采用算术运算比对和融合代码表示, 难以识别和区分变更前后代码之间的细粒度差异. 2) 缺乏对代码和代码变更的结构信息的利用. 如何在特征融合阶段对代码和代码变更的结构信息加以利用尚未被充分探索.

7.2 研究机遇

7.2.1 代码变更的预训练

预训练是近几年非常热门的表示学习技术. 该技术先利用自监督学习在大规模无标签数据上对模型进行预训练 (pre-training), 以获得良好的模型初始参数. 在被应用于下游任务时, 预训练好的模型仅需在少量有标签数据上继续训练 (fine-tuning) 即可获得较好的性能[94,95]. 预训练能够利用易于收集的大规模无标签数据来提高模型在数据量较少的目标任务上的性能. 大多数与代码变更相关的软件工程任务都难以收集到大量有标签的代码变更 (例如安全漏洞补丁), 而开源项目的代码仓库中包含丰富的无标签代码变更数据. 这意味着预训练技术有望提升代码变更表示学习方法的性能. 一些 NLP 预训练模型 (例如 BERT[68]) 已经被应用在很多代码变更表示学习的相关研究中并且取得了不错的效果[14,27,31,49], 研究者们也提出了一些面向源代码的预训练模型 (例如 CodeBERT[96]). 相关研究的实验证明, 预训练有效提升了模型效果[14,16,31,93]. 然而, 目前尚缺乏为代码变更定制的、更能够有效学习代

码变更先验知识的预训练任务[16,32]和面向代码变更的预训练模型. 进一步探索预训练模型在代码变更表示学习中的应用或能有效提升现有代码变更学习技术的性能.

7.2.2 多任务代码变更表示学习

现有的代码变更表示学习研究大多只关注单一的软件工程任务. 不同的软件工程任务对代码变更的信息需求既存在差异, 例如即时代码缺陷更关注代码变更的质量, 而代码提交日志生成更关注代码变更的内容和意图; 也存在相似之处, 例如即时缺陷预测和软件制品可追溯性恢复都关注代码提交与软件缺陷的关联. 在自然语言处理等领域, 多任务学习 (multi-task learning) 的有效性已经反复得以验证[97]. 多任务学习能帮助代码变更表示学习模型分辨不同任务相似却不同的信息需求, 帮助模型捕捉到代码变更中多种维度的特征. 一些研究者已开始尝试通过将多任务学习应用到代码变更表示学习领域. 例如 Bai 等人[19]提出将缺陷修复模型和代码提交日志生成模型串联起来实施联合训练, 提升了模型性能. 在未来, 探究更多代码变更相关的任务之间的关联关系并设计更加先进有效的多任务学习框架有望成为领域的研究趋势之一.

7.2.3 代码变更层次信息的利用

代码变更中的信息是层次化的, 例如一次代码提交可能改动多个文件, 一个被改动的文件内可能包含多个变更的代码块等. 这些层次化信息有望提升代码变更表示学习的性能并帮助设计出适用于多种代码变更粒度的通用技术. 然而, 目前少有研究工作考虑大粒度的代码变更 (例如包含多个文件改动的代码提交), 对层次信息加以利用的研究工作也并不多见. 探索利用代码变更层次信息的新方法和利用大粒度代码变更特征来增强代码变更表示学习仍然是有待探索的研究方向.

7.2.4 基于显式信息交互的代码变更表示学习研究机遇

基于显式信息交互的代码变更表示学习研究机遇可概括为: 1) 设计更先进的输入表示形式和表示学习模型使代码变更表示能兼顾变更代码和未变更代码中的重要信息. 例如, 可尝试利用 Transformer 模型并设计特殊的掩码机制, 使变更的代码和未变更的代码能够显式地交互. 2) 识别和利用代码变更中的依赖关系以更好地学习其中的语义信息和上下文信息. 目前, 仅有 Meng 等人[43]在学习代码变更表示时对代码变更中的依赖关系加以利用, 取得了良好的效果. 这一方向需要进一步的研究和探索. 3) 对不同输入表示形式和不同表示学习模型进行系统性的比较, 推动代码变更表示学习领域后续的研究和应用工作.

7.2.5 基于隐式信息交互的代码变更表示学习研究机遇

基于隐式信息交互的代码变更表示学习研究机遇可以概括为: 1) 有效利用代码和代码变更的结构信息. 如第 4.1 节所述, 基于代码变更中的结构化信息进行隐式信息交互的研究目前仍是空白. 现有利用结构化信息的代码变更表示学习技术虽然是基于显式信息交互的, 但是其提取代码变更分布式表示的流程与图 5 中基于隐式信息交互的方法非常相似[17]. 这意味着后续研究工作能够参考现有基于显式信息交互的结构化代码变更表示学习技术来构建基于隐式信息交互的结构化代码变更表示学习技术, 以填补相应技术的空白. 另一方面, 设计和利用考虑代码结构信息的特征融合模型对变更前后的代码特征进行“结构化”的特征融合, 有望在保留代码变更整体语义的同时聚焦细粒度的变更信息, 提升现有方法的性能. 2) 结合代码预训练技术. 代码预训练技术使代码表示学习领域取得了长足的进步. 基于隐式信息交互的代码变更表示学习方法需要利用代码表示学习模型来编码变更前后的代码. 集成代码预训练模型来学习变更前后代码的分布式表示能够为后续的特征融合阶段奠定良好的基础, 有望有效提升现有方法的性能.

8 总 结

近年来, 代码变更表示学习技术在很多软件工程任务中得到了广泛的应用, 并作为相关算法中的关键组成部分不断促进相关任务取得新的进展和突破. 尽管已有较多工作研究和使用了软件变更表示学习技术, 但目前尚无对相关技术和下游应用的系统性归纳和总结, 且代码变更表示学习领域仍存在结构信息利用困难、基准数据集缺失等挑战. 鉴于此, 本文基于现有技术对变更前后代码的信息进行比对和融合的时机对现有技术进行了归类, 从学习方式、输入表示方式、表示学习模型和特征融合方法等维度对现有技术进行了梳理, 并归类和总结了代码变更

表示学习的下游任务和常用的性能评估指标, 最后通过总结当前的代码变更表示学习领域存在的关键挑战和潜在机遇, 对这一领域的未来发展进行了展望.

References:

[1] Brudaru II, Zeller A. What is the long-term impact of changes? In: Proc. of the 2008 Int'l Workshop on Recommendation Systems for Software Engineering. Atlanta: ACM Press, 2008. 30–32. [doi: 10.1145/1454247.1454257]

[2] Sommerville I. Software Engineering. 9th ed., Boston: Pearson, 2011.

[3] Hoang T, Kang HJ, Lo D, Lawall J. CC2Vec: Distributed representations of code changes. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 518–529.

[4] Hoang T, Khanh Dam H, Kamei Y, Lo D, Ubayashi N. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Montreal: IEEE, 2019. 34–45. [doi: 10.1109/MSR.2019.00016]

[5] Wang S, Liu TY, Nam J, Tan L. Deep semantic feature learning for software defect prediction. IEEE Trans. on Software Engineering, 2020, 46(12): 1267–1293. [doi: 10.1109/TSE.2018.2877612]

[6] Loyola P, Matsuo Y. Learning feature representations from change dependency graphs for defect prediction. In: Proc. of the 28th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Toulouse: IEEE, 2017. 361–372. [doi: 10.1109/ISSRE.2017.30]

[7] Zeng ZR, Zhang YQ, Zhang HT, Zhang LM. Deep just-in-time defect prediction: How far are we? In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2021. 427–438. [doi: 10.1145/3460319.3464819]

[8] Gesi J, Li JW, Ahmed I. An empirical examination of the impact of bias on just-in-time defect prediction. In: Proc. of the 15th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Bari: ACM, 2021. 7. [doi: 10.1145/3475716.3475791]

[9] Ruan H, Chen BH, Peng X, Zhao WY. $D_{EEP}L_{INK}$: Recovering issue-commit links based on deep learning. Journal of Systems and Software, 2019, 158: 110406. [doi: 10.1016/j.jss.2019.110406]

[10] Lin JF, Liu YL, Zeng QK, Jiang M, Cleland-Huang J. Traceability transformed: Generating more accurate links with pre-trained BERT models. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 324–335. [doi: 10.1109/ICSE43902.2021.00040]

[11] Yin PC, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL. Learning to represent edits. In: Proc. of the 7th Int'l Conf. on Learning Representations. New Orleans: OpenReview.net, 2019.

[12] Jiang SY, Armaly A, McMillan C. Automatically generating commit messages from diffs using neural machine translation. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 135–146. [doi: 10.1109/ASE.2017.8115626]

[13] Xu SB, Yao Y, Xu F, Gu TX, Tong HH, Lu J. Commit message generation for source code changes. In: Proc. of the 28th Int'l Joint Conf. on Artificial Intelligence. Macao: AAAI Press, 2019. 3975–3981.

[14] Jung TH. CommitBERT: Commit message generation using pre-trained programming language model. In: Proc. of the 1st Workshop on Natural Language Processing for Programming. Association for Computational Linguistics, 2021. 26–33. [doi: 10.18653/v1/2021.nlp4prog-1.3]

[15] Liu Q, Liu ZH, Zhu HM, Fan HF, Du BW, Qian Y. Generating commit messages from diffs using pointer-generator network. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Montreal: IEEE, 2019. 299–309. [doi: 10.1109/MSR.2019.00056]

[16] Nie LY, Gao CY, Zhong ZC, Lam W, Liu Y, Xu ZL. CoreGen: Contextualized code representation learning for commit message generation. Neurocomputing, 2021, 459: 97–107. [doi: 10.1016/j.neucom.2021.05.039]

[17] Liu SQ, Gao CY, Chen S, Nie LY, Liu Y. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans. on Software Engineering, 2022, 48(5): 1800–1817. [doi: 10.1109/TSE.2020.3038681]

[18] Jiang SY. Boosting neural commit message generation with code semantic analysis. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 1280–1282. [doi: 10.1109/ASE.2019.00162]

[19] Bai JQ, Zhou L, Blanco A, Liu SJ, Wei FR, Zhou M, Li ZJ. Jointly learning to repair code and generate commit message. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 9784–9795. [doi: 10.18653/v1/2021.emnlp-main.771]

[20] Wang HY, Xia X, Lo D, He Q, Wang XY, Grundy J. Context-aware retrieval-based deep commit message generation. ACM Trans. on Software Engineering and Methodology, 2021, 30(4): 56. [doi: 10.1145/3464689]

[21] Panthaplackel S, Nie PY, Gligoric M, Li JJ, Mooney R. Learning to update natural language comments based on code changes. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. ACL, 2020. 1853–1868. [doi: 10.18653/v1/2020.acl-main.168]

[22] Liu ZX, Xia X, Yan M, Li SP. Automating just-in-time comment updating. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2020. 585–597.

[23] Panthaplackel S, Li JJ, Gligoric M, Mooney RJ. Deep just-in-time inconsistency detection between comments and source code. Proc. of the AAAI Conf. on Artificial Intelligence, 2021, 35(1): 427–435. [doi: 10.1609/aaai.v35i1.16119]

[24] Gao ZP, Xia X, Lo D, Grundy J, Zimmermann T. Automating the removal of obsolete TODO comments. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 218–229. [doi: 10.1145/3468264.3468553]

[25] Lozoya RC, Baumann A, Sabetta A, Bezzi M. Commit2Vec: Learning distributed representations of code changes. SN Computer Science, 2021, 2(3): 150. [doi: 10.1007/s42979-021-00566-z]

[26] Zhou YQ, Siow JK, Wang CY, Liu SQ, Liu Y. SPI: Automated identification of security patches via commits. ACM Trans. on Software Engineering and Methodology, 2022, 31(1): 13. [doi: 10.1145/3468854]

[27] Zhou JY, Pacheco M, Wan ZY, Xia X, Lo D, Wang Y, Hassan AE. Finding A needle in a haystack: Automated mining of silent vulnerability fixes. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2021. 705–716. [doi: 10.1109/ASE51524.2021.9678720]

[28] Li HY, Shi ST, Thung F, Huo X, Xu BW, Li M, Lo D. DeepReview: Automatic code review using deep multi-instance learning. In: Proc. of the 23rd Pacific-Asia Conf. on Knowledge Discovery and Data Mining. Macao: Springer, 2019. 318–330. [doi: 10.1007/978-3-030-16145-3_25]

[29] Shi ST, Li M, Lo D, Thung F, Huo X. Automatic code review by learning the revision of source code. Proc. of the AAAI Conf. on Artificial Intelligence, 2019, 33(1): 4910–4917. [doi: 10.1609/aaai.v33i01.33014910]

[30] Yao ZY, Xu FF, Yin PC, Sun H, Neubig G. Learning structural edits via incremental tree transformations. In: Proc. of the 9th Int'l Conf. on Learning Representations. OpenReview.net, 2021.

[31] Svyatkovskiy A, Mytcowicz T, Ghorbani N, Fakhoury S, Dinella E, Bird C, Sundaresan N, Lahiri S. MergeBERT: Program merge conflict resolution via neural transformers. arXiv:2109.00084, 2022.

[32] Pravilov M, Bogomolov E, Golubev Y, Bryksin T. Unsupervised learning of general-purpose embeddings for code changes. In: Proc. of the 5th Int'l Workshop on Machine Learning Techniques for Software Quality Evolution. Athens: ACM, 2021. 7–12. [doi: 10.1145/3472674.3473979]

[33] Loyola P, Marrese-Taylor E, Matsuo Y. A neural architecture for generating natural language descriptions from source code changes. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics. Vancouver: Association for Computational Linguistics, 2017. 287–292. [doi: 10.18653/v1/P17-2045]

[34] Hoang T, Lawall J, Tian Y, Oentaryo RJ, Lo D. PatchNet: Hierarchical deep learning-based stable patch identification for the Linux kernel. IEEE Trans. on Software Engineering, 2021, 47(11): 2471–2486. [doi: 10.1109/TSE.2019.2952614]

[35] Mi JW, Shi ST, Li M. Learning code changes by exploiting bidirectional converting deviation. In: Proc. of the 12th Asian Conf. on Machine Learning. Bangkok: PMLR, 2020. 481–496.

[36] Siow JK, Gao CY, Fan LL, Chen S, Liu Y. CORE: Automating review recommendation for code changes. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 284–295. [doi: 10.1109/SANER48275.2020.9054794]

[37] Panthaplackel S, Allamanis M, Brockschmidt M. Copy that! Editing sequences by copying spans. Proc. of the AAAI Conf. on Artificial Intelligence, 2021, 35(15): 13622–13630. [doi: 10.1609/aaai.v35i15.17606]

[38] 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. Athens: ACM, 2021. 1479–1482. [doi: 10.1145/3468264.3473134]

[39] Loyola P, Marrese-Taylor E, Balazs J, Matsuo Y, Satoh F. Content aware source code change description generation. In: Proc. of the 11th Int'l Conf. on Natural Language Generation. Tilburg University: Association for Computational Linguistics, 2018. 119–128. [doi: 10.18653/v1/W18-6513]

[40] Zhao R, Bieber D, Swersky K, Tarlow D. Neural networks for modeling source code edits. arXiv:1904.02818, 2019.

[41] Brody S, Alon U, Yahav E. A structural model for contextual code changes. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 215. [doi: 10.1145/3428283]

[42] Ni Z, Li B, Sun XB, Chen TH, Tang B, Shi XC. Analyzing bug fix for automatic bug cause classification. Journal of Systems and Software, 2020, 163: 110538. [doi: 10.1016/j.jss.2020.110538]

[43] Meng N, Jiang ZJ, Zhong H. Classifying code commits with convolutional neural networks. In: Proc. of the 2021 Int'l Joint Conf. on Neural Networks (IJCNN). Shenzhen: IEEE, 2021. 1–8. [doi: 10.1109/IJCNN52387.2021.9533534]

[44] Le THM, Hin D, Croft R, Babar MA. DeepCVA: Automated commit-level vulnerability assessment with deep multi-task learning. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2021. 717–729. [doi: 10.1109/ASE51524.2021.9678622]

[45] Loyola P, Gajananan K, Satoh F. Bug localization by learning to rank and represent bug inducing changes. In: Proc. of the 27th ACM Int'l Conf. on Information and Knowledge Management. Torino: ACM, 2018. 657–665. [doi: 10.1145/3269206.3271811]

[46] Dinella E, Mytkowicz T, Svyatkovskiy A, Bird C, Naik M, Lahiri SK. DeepMerge: Learning to merge programs. IEEE Trans. on Software Engineering, 2022. [doi: 10.1109/TSE.2022.3183955]

[47] Liu ZX, Xia X, Hassan AE, Lo D, Xing ZC, Wang XY. Neural-machine-translation-based commit message generation: How far are we? In: Proc. of the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering. Montpellier: IEEE, 2018. 373–384. [doi: 10.1145/3238147.3238190]

[48] Tao W, Wang YL, Shi ES, Du L, Han S, Zhang HY, Zhang DM, Zhang WQ. On the evaluation of commit message generation models: An experimental study. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Luxembourg: IEEE, 2021. 126–136. [doi: 10.1109/ICSME52107.2021.00018]

[49] Tian HY, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF. Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2020. 981–992.

[50] Cao YK, Sun ZY, Zou YZ, Xie B. Structurally-enhanced approach for automatic code change transformation. Ruan Jian Xue Bao/Journal of Software, 2021, 32(4): 1006−1022 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6227.htm [doi: 10.13328/j.cnki.jos.006227]

[51] Dong JH, Lou YL, Zhu QH, Sun ZY, Li ZL, Zhang WJ, Hao D. FIRA: Fine-grained graph-based code change representation for automated commit message generation. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Pittsburgh: IEEE, 2022. 970–981. [doi: 10.1145/3510003.3510069]

[52] Ciborowska A, Damevski K. Fast changeset-based bug localization with BERT. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Pittsburgh: IEEE, 2022. 946–957. [doi: 10.1145/3510003.3510042]

[53] Fluri B, Wursch M, Pinzger M, Gall H. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. on Software Engineering, 2007, 33(11): 725–743. [doi: 10.1109/TSE.2007.70731]

[54] Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. on Software Engineering, 2013, 39(6): 757–773. [doi: 10.1109/TSE.2012.70]

[55] Liu ZY, Sun MS, Lin YK, Xie RB. Knowledge representation learning: A review. Journal of Computer Research and Development, 2016, 53(2): 247–261 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20160020]

[56] LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521(7553): 436–444. [doi: 10.1038/nature14539]

[57] Hamilton WL, Ying R, Leskovec J. Representation learning on graphs: Methods and applications. IEEE Data Engineering Bulletin, 2017, 40(3): 52–74.

[58] Kolesnikov A, Zhai XH, Beyer L. Revisiting self-supervised visual representation learning. In: Proc. of the 2019 IEEE/CVF Conf. on Computer Vision and Pattern Recognition. Long Beach: IEEE, 2019. 1920–1929. [doi: 10.1109/CVPR.2019.00202]

[59] Hinton GE, McClelland JL, Rumelhart DE. Distributed representations. In: Rumelhart DE, McClelland JL, eds. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. Cambridge: MIT Press, 1986. 77–109.

[60] Liu F, Li G, Hu X, Jin Z. Program comprehension based on deep learning. Journal of Computer Research and Development, 2019, 56(8): 1605−1620 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20190185]

[61] Hu X, Li G, Liu F, Jin Z. Program generation and code completion techniques based on deep learning: Literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1206−1223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[62] Lin ZQ, Zou YZ, Zhao JF, Cao YK, Xie B. Software text semantic search approach based on code structure knowledge. Ruan Jian Xue Bao/Journal of Software, 2019, 30(12): 3714−3729 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5609.htm [doi: 10.13328/j.cnki.jos.005609]

[63] Han X. Distributed representation of knowledge graphs [Ph.D. Thesis]. Beijing: Beijing University of Posts and Telecommunications,

2019 (in Chinese with English abstract).

[64] Wang T. Research on the advertisement click-through-rate prediction algorithm based on distributed representation [MS. Thesis]. Wuhan: Huazhong University of Science and Technology, 2019 (in Chinese with English abstract).

[65] Weinberger KQ, Saul LK. Distance metric learning for large margin nearest neighbor classification. The Journal of Machine Learning Research, 2009, 10: 207–244.

[66] Koch G, Zemel R, Salakhutdinov R. Siamese neural networks for one-shot image recognition. In: Proc. of the 32nd Int'l Conf. on Machine Learning. Lille: JMLR, 2015. 1–8.

[67] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv:1301.3781, 2013.

[68] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: ACL, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

[69] Yang W. Identifying syntactic differences between two programs. Software: Practice and Experience, 1991, 21(7): 739–755. [doi: 10.1002/spe.4380210706]

[70] Ye X, Zheng YJ, Aljedaani W, Mkaouer MW. Recommending pull request reviewers based on code changes. Soft Computing, 2021, 25(7): 5619–5632. [doi: 10.1007/s00500-020-05559-3]

[71] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735–1780. [doi: 10.1162/neco.1997.9.8.1735]

[72] Cho K, van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proc. of the 2014 Conf. on Empirical Methods in Natural Language Processing (EMNLP). Doha: Association for Computational Linguistics, 2014. 1724–1734. [doi: 10.3115/v1/D14-1179]

[73] Kim Y. Convolutional neural networks for sentence classification. In: Proc. of the 2014 Conf. on Empirical Methods in Natural Language Processing (EMNLP). Doha: Association for Computational Linguistics, 2014. 1746–1751. [doi: 10.3115/v1/D14-1181]

[74] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[75] Hinton GE, Osindero S, Teh YW. A fast learning algorithm for deep belief nets. Neural Computation, 2006, 18(7): 1527–1554. [doi: 10.1162/neco.2006.18.7.1527]

[76] Hindle A, Barr ET, Gabel M, Su ZD, Devanbu P. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122–131. [doi: 10.1145/2902362]

[77] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 313–324. [doi: 10.1145/2642937.2642982]

[78] Mou LL, Li G, Jin Z, Zhang L, Wang T. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv:1409.5718, 2015.

[79] Li YJ, Tarlow D, Brockschmidt M, Zemel RS. Gated graph sequence neural networks. In: Proc. of the 4th Int'l Conf. on Learning Representations. San Juan, 2016.

[80] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: Proc. of the 5th Int'l Conf. on Learning Representations. Toulon: OpenReview.net, 2017.

[81] Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE. Neural message passing for Quantum chemistry. In: Proc. of the 34th Int'l Conf. on Machine Learning. Sydney: JMLR.org, 2017. 1263–1272.

[82] Le Q, Mikolov T. Distributed representations of sentences and documents. In: Proc. of the 31st Int'l Conf. on Machine Learning. Beijing: JMLR.org, 2014. II-1188–II-1196.

[83] Socher R, Perelygin A, Wu J, Chuang J, Manning CD, Ng A, Potts C. Recursive deep models for semantic compositionality over a sentiment treebank. In: Proc. of the 2013 Conf. on Empirical Methods in Natural Language Processing. Seattle: ACL, 2013. 1631–1642.

[84] Li J, Wang Y, Lyu MR, King I. Code completion with neural attention and pointer networks. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. Stockholm: AAAI Press, 2018. 4159–4125.

[85] Papineni K, Roukos S, Ward T, Zhu WJ. Bleu: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia: ACL, 2002. 311–318. [doi: 10.3115/1073083.1073135]

[86] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: Proc. of the 3rd Int'l Conf. on Learning Representations. San Diego, 2015.

[87] Lin CY. ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. Barcelona: ACL, 2004. 74–81.

[88] Kryscinski W, Keskar NS, McCann B, Xiong CM, Socher R. Neural text summarization: A critical evaluation. In: Proc. of the 2019 Conf.

on Empirical Methods in Natural Language Processing and the 9th Int'l Joint Conf. on Natural Language Processing. Hong Kong: ACL, 2019. 540–551. [doi: 10.18653/v1/D19-1051]

[89] Banerjee S, Lavie A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proc. of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. Ann Arbor: ACL, 2005. 65–72.

[90] Burges C, Shaked T, Renshaw E, Lazier A, Deeds M, Hamilton N, Hullender G. Learning to rank using gradient descent. In: Proc. of the 22nd Int'l Conf. on Machine Learning. Bonn: ACM, 2005. 89–96. [doi: 10.1145/1102351.1102363]

[91] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Proc. of the 25th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: Curran Associates Inc., 2012. 1097–1105.

[92] Wang A, Singh A, Michael J, Hill F, Levy O, Bowman S. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In: Proc. of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. Brussels: ACL, 2018. 353–355. [doi: 10.18653/v1/W18-5446]

[93] Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436, 2020.

[94] Sun C, Qiu XP, Xu YG, Huang XJ. How to fine-tune BERT for text classification? In: Proc. of the 18th China National Conf. on Chinese Computational Linguistics. Kunming: Springer, 2019. 194–206. [doi: 10.1007/978-3-030-32381-3_16]

[95] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018. https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf

[96] Feng ZY, Guo DY, Tang D, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the Findings of the Association for Computational Linguistics: EMNLP 2020. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[97] Zhang Y, Yang Q. An overview of multi-task learning. National Science Review, 2018, 5(1): 30–43. [doi: 10.1093/nsr/nwx105]

附中文参考文献:

[50] 曹英魁, 孙泽宇, 邹艳珍, 谢冰. 一种结构信息增强的代码修改自动转换方法. 软件学报, 2021, 32(4): 1006–1022. http://www.jos.org.cn/1000-9825/6227.htm [doi: 10.13328/j.cnki.jos.006227]

[55] 刘知远, 孙茂松, 林衍凯, 谢若冰. 知识表示学习研究进展. 计算机研究与发展, 2016, 53(2): 247–261. [doi: 10.7544/issn1000-1239.2016.20160020]

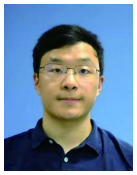
[60] 刘芳, 李戈, 胡星, 金芝. 基于深度学习的程序理解研究进展. 计算机研究与发展, 2019, 56(8): 1605–1620. [doi: 10.7544/issn1000-1239.2019.20190185]

[61] 胡星, 李戈, 刘芳, 金芝. 基于深度学习的程序生成与补全技术研究进展. 软件学报, 2019, 30(5): 1206−1223. http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[62] 林泽琦, 邹艳珍, 赵俊峰, 曹英魁, 谢冰. 基于代码结构知识的软件文档语义搜索方法. 软件学报, 2019, 30(12): 3714−3729. http://www.jos.org.cn/1000-9825/5609.htm [doi: 10.13328/j.cnki.jos.005609]

[63] 韩笑. 知识图谱分布式表示研究 [博士学位论文]. 北京: 北京邮电大学, 2019.

[64] 王涛. 基于分布表示的广告点击率预估算法研究 [硕士学位论文]. 武汉: 华中科技大学, 2019.



刘忠鑫(1994－), 男, 博士, 特聘研究员, CCF 专业会员, 主要研究领域为智能软件工程, 软件仓库挖掘.

夏鑫(1986－), 男, 博士, CCF 专业会员, 主要研究领域为软件仓库挖掘, 经验软件工程.

唐郅杰(1999－), 男, 硕士生, 主要研究领域为智能软件工程.

李善平(1963－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为分布式计算, 软件工程, Linux 内核.