软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(8):3824−3842 [doi: 10.13328/j.cnki.jos.006947]

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

基于混合图表示的软件变更预测方法*

杨馨悦[1], 刘 安[1], 赵 雷[1], 陈 林[2], 章晓芳[1,2]

[1](苏州大学 计算机科学与技术学院, 江苏 苏州 215006)

[2](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210023)

通信作者: 章晓芳, E-mail: xfzhang@suda.edu.cn

摘 要: 软件变更预测旨在识别出具有变更倾向的模块, 可以帮助软件管理者和开发人员有效地分配资源, 降低维护开销. 从代码中提取有效的特征在构建准确的预测模型中起着重要作用. 近年来, 研究人员从利用传统的手工特征进行预测转向具有强大表示能力的语义特征, 他们从抽象语法树 (abstract syntax tree, AST) 的节点序列中提取语义特征构建模型. 但已有研究忽略了 AST 的结构信息以及代码中丰富的语义信息, 如何提取代码的语义特征仍然是一个具有挑战性的问题. 为此, 提出一种基于混合图表示的变更预测方法. 该模型首先结合 AST、控制流图 (control flow graph, CFG)、数据流图 (data flow graph, DFG) 等结构信息构建代码的程序图表示, 接着利用图神经网络学习出程序图的语义特征, 根据该特征预测变更倾向性. 所提模型能够融合各种语义信息以更好地表征代码. 在多组变更数据集上开展与最新变更预测方法的对比实验, 验证了所提方法的有效性.

关键词: 软件变更预测; 图神经网络; AST; 控制流图; 数据流图

中图法分类号: TP311

中文引用格式: 杨馨悦, 刘安, 赵雷, 陈林, 章晓芳. 基于混合图表示的软件变更预测方法. 软件学报, 2024, 35(8): 3824–3842. http://www.jos.org.cn/1000-9825/6947.htm

英文引用格式: Yang XY, Liu A, Zhao L, Chen L, Zhang XF. Software Change Prediction Based on Hybrid Graph Representation. Ruan Jian Xue Bao/Journal of Software, 2024, 35(8): 3824–3842 (in Chinese). http://www.jos.org.cn/1000-9825/6947.htm

Software Change Prediction Based on Hybrid Graph Representation

YANG Xin-Yue[1], LIU An[1], ZHAO Lei[1], CHEN Lin[2], ZHANG Xiao-Fang[1,2]

[1](School of Computer Science & Technology, Soochow University, Suzhou 215006, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Software change prediction, aimed at identifying change-prone modules, can help software managers and developers allocate resources efficiently and reduce maintenance overhead. Extracting effective features from the code plays a vital role in the construction of accurate prediction models. In recent years, researchers have shifted from traditional hand-crafted features to semantic features with powerful representation capabilities for prediction. They extracted semantic features from abstract syntax tree (AST) node sequences to build models. However, existing studies have ignored the structural information in the AST and the rich semantic information in the code. How to extract the semantic features of the code is still a challenging problem. For this reason, the study proposes a change prediction method based on hybrid graph representation. To start with, the model combines AST, control flow graph (CFG), data flow graph (DFG), and other structural information to construct the program graph representation of the code. Then, it uses the graph neural network to learn the semantic features of the program graph and the features obtained to predict change-proneness. The model can integrate various semantic information to represent the code better. The effectiveness of the proposed method is verified by comparing it with the latest change prediction methods on various change datasets.

Key words: software change prediction; graph neural network (GNN); abstract syntax tree (AST); control flow graph (CFG); data flow graph (DFG)

* 基金项目: 国家自然科学基金 (62172202, 61872177); 江苏省自然科学基金 (BK20211307); 江苏省高等学校基础科学研究重大项目 (19KJA610002, 22KJA520008); 软件新技术与产业化协同创新中心资助项目; 江苏高校优势学科建设工程
收稿时间: 2022-08-21; 修改时间: 2022-11-17, 2023-02-13; 采用时间: 2023-04-09; jos 在线出版时间: 2023-09-13
CNKI 网络首发时间: 2023-09-14

软件维护是工作量最大、时间跨度最长的阶段, 维护工作占整个软件生命周期的 60% 到 70%, 维护的成本在总的开销中占到了 90%[1], 如何降低维护开销是软件工程中一个至关重要的问题. 软件变更预测旨在识别出具有变更倾向的模块, 即在将来会被频繁修改的软件模块. 这些模块是软件系统中的薄弱部分, 存在不合理的设计结构、代码质量低或缺陷等潜在问题, 导致软件在维护时期需要花费大量的资源[2]. 软件变更预测技术能够在早期阶段识别出变更倾向的模块, 可以帮助开发人员提前发现并解决潜在的问题, 从而提高软件产品的质量; 并且在熟悉这些代码的阶段对其进行修改, 与维护时期相比投入的精力和时间成本更低. 另一方面, 变更预测技术可以为维护人员提供关于维护工作的建议, 例如进行改正性维护工作, 解决在测试阶段未发现的错误; 以及预防性维护工作, 进行重构、同行评审、测试等, 进一步提高软件质量, 以降低后期维护的工作量和开销. 软件变更预测技术是降低维护开销的一个有效手段[3].

软件变更预测技术已经引起了大量研究人员的关注. 他们首先设计用于衡量代码模块特性的度量元, 目前, 广泛被使用的度量元是基于代码特征和基于软件开发过程的度量元. 基于代码特征的度量元主要包括代码行数、面向对象指标 (继承、内聚和耦合等)、圈复杂度指标和代码异味等[4−7]; 基于开发过程的度量元主要包括代码改变的次数、改变的时间、改变的频率和开发者信息[2,8]. 然后基于这些传统特征利用机器学习方法构建模型, 例如, 逻辑回归[2−4,7], 随机森林[9], 支持向量机[3,10]等. 然而, 这些方法无法捕获代码的语义信息, 具有不同语义的代码它们的传统特征值可能是相同的[11], 传统的模型在训练预测过程中无法准确地区分不同的代码. 因此, 利用语义特征可以提高现有变更预测方法的性能.

已有研究者从抽象语法树 (abstract syntax tree, AST) 中抽取语义特征, 他们遍历 AST 得到节点序列并利用词嵌入技术学习出语义特征[11,12]. 这些方法没有利用 AST 中的结构信息, 此外, AST 中不包含控制流、数据流等语义信息. 为了得到强有力的代码表示, 研究者们关注于代码的控制流图 (control flow graph, CFG) 或数据流图 (data flow graph, DFG). 例如, 一些研究者们从 AST 和控制流图中提取代码特征用于克隆检测[13,14]; Guo 等人[15]引入数据流图信息用于预训练模型, 在多个下游任务中获得了最先进的性能. CFG 表示了程序执行过程中所有路径, 图中节点是代码语句块, 从 CFG 中提取的特征只包含语句块之间的控制流信息, 无法表达语句块内部的代码特征. 其次, 与 AST 相比 CFG 中边是较少的, 在特征学习过程中, 较少的边意味着节点之间有较少的信息交互, 很难学习到代码潜在的特征表示[16]. 而 DFG 从数据传递和加工角度表达程序的逻辑功能, 利用 DFG 只能捕获代码中变量的使用情况. 由于各种图结构只能表示单一的代码特征, 将多种结构信息融合可以更准确地建模代码来提升预测模型性能.

鉴于上述分析的两种流程图的局限性以及对于一些开发语言 (例如 Java) 来说, 提取 CFG 和 DFG 是困难的[16], 目前, 在 AST 的基础上添加控制流和数据流信息是一种新颖并且有效的方法, 从而将多种结构信息融合[17]. AST 不仅能表示 CFG 中语句块内的代码特征, 也包含了 DFG 中的变量信息, 在 AST 中添加控制流和数据流相关的边可以对代码丰富的语法结构和各种语义信息建模更好地表征源代码.

此外, 目前对于软件变更预测的研究均是文件级别的粒度[3,4], 其对应的 AST 结构是巨大的, 存在大量的中间节点用于以树的形式表示程序的语法. 其中, 包括一些不重要的中间节点, 例如, name 和 value 等, 这些节点可能会降低关键节点的重要程度, 删除这些节点不仅可以降低模型学习的难度还可以减少训练的时间开销.

针对上述 2 个问题, 本文提出了一种基于混合图表示的软件变更预测方法 (hybrid graph representation based software change prediction, HGSCP). 首先, 我们简化 AST, 删除其中对于语法表达不重要的中间节点并进行重构得到简化的 AST, 它保留了关键的语法结构. 接着, 在简化的 AST 上增加表示控制流、数据流信息的边, 将 3 种代码图结构结合起来得到代码混合图表示. 最后, 我们利用图神经网络学习混合图中每个节点的嵌入, 再通过全局注意力机制得到整个代码的特征表示并用于变更预测. 经过多组对比实验验证了本文所提方法在变更预测问题上的有效性.

本文的贡献主要包括以下 3 点.

(1) 提出一种程序的混合图表示方法, 将代码的语法结构信息与控制流、数据流等语义信息结合来表征代码, 从而提取出有效的代码特征. 我们的表示方法完全基于 AST, 可以扩展到其他编程语言.

(2) 本文是第 1 个将图神经网络应用于变更预测的工作, 我们利用图神经网络学习混合图的向量表示并用于变更预测.

(3) 我们在 8 个开源项目上开展了严格的实验来验证所提方法的有效性, 并且开源了所有的数据和代码.

本文第 1 节介绍软件变更预测和代码图表示的相关研究现状. 第 2 节介绍本文所提出的基于混合图表示的软件变更预测方法的设计与实现. 第 3 节是实验设计, 介绍研究问题、实验数据集、评价指标、实验环境与参数设置. 第 4 节通过对比实验验证所提模型的有效性. 第 5 节对方法中的细节进行讨论并分析有效性的威胁. 最后总结全文.

1 相关工作

1.1 软件变更预测

现有的软件变更预测模型大多数都基于传统度量元并利用机器学习方法构建模型. Lu 等人[18]分析了程序的面向对象特征与变更倾向性之间的相关性, 他们分析了 62 种面向对象指标, 包括类的大小、内聚、耦合和继承指标, 结果表明类的大小指标在区分是否易于变更方面效果优于耦合、内聚指标, 继承类的指标与变更倾向性之间相关性最小.

Zhu 等人[3]利用圈复杂度和代码中 token 频率等指标进行变更预测, 采用不平衡学习方法 bagging 和重采样方法的组合解决变更预测中类不平衡问题.

Liu 等人[6]认为代码异味指标可以提高模型性能, 他们比较了 5 种代码异味与 CK 指标在变更预测中的性能, 结果表明代码异味指标与变更倾向性之间存在显著相关性, 并且代码异味指标在预测性能上优于基线指标, Catolino 等人[7]也证实了他们的研究.

此外, Catolino 等人[2]也研究了开发者相关的指标与代码变更之间是否存在相关性, 他们基于开发者指标 (例如, 开发者数量) 构建模型, 结果表明, 开发者指标在变更预测中表现良好, 且开发者指标与基于代码特征的指标和开发过程指标相结合可以获得更优的性能表现.

软件系统的变更历史中包含丰富的演化信息, Elish 等人[8]认为软件产品最近的变更情况可以用来预测将来是否会发生变更, 他们提出了 16 个演化指标描述软件的更改情况. 结果表明, 所提出的演化指标可以有效地预测变更倾向的模块.

Malhotra 等人[19]通过研究特征降维技术来解决特征维度高、包含不相关和重复特征的问题. 他们基于 86 个代码指标研究了 11 种特征降维技术在变更预测中的效果. 研究发现特征降维技术对于去除冗余和不相关的特征至关重要, 线性判别分析 (linear discriminant analysis, LDA)、自编码器和基于封装式特征选择的方法获得了较好的性能, 他们也指出面向对象指标和基于图的指标与变更倾向性是高度相关的.

Zhu 等人[20]利用深度学习技术进行变更预测, 提出了一种基于卷积神经网络的变更预测方法, 并结合重采样方法来解决类不平衡问题. 研究表明, 基于卷积神经网络和重采样结合的方法获得了最优的性能表现.

已有工作使用传统特征进行变更预测, 而传统度量元无法表达程序的语义信息, 我们利用语义特征提高模型的性能.

1.2 基于 AST 的语义特征

与自然语言不同, 程序中包含丰富、明确和复杂的结构信息, 自然语言模型可能不适用于程序设计语言. Mou 等人[21]提出了基于树的卷积神经网络 TBCNN 用于处理编程语言, 通过卷积操作来捕获 AST 的结构特征. 为了从 AST 中提取深度语义特征, Wei 等人[22]提出了 CDLH 端到端特征学习框架, 利用长短期记忆 (long short-term memory, LSTM) 网络学习 AST 中语义特征, 在 LSTM 中增加了多个状态单元, 以自下而上的方式计算 AST 每个节点单元状态, 最后将根节点的状态输出作为代码片段的表示. Zhang 等人[12]指出当 AST 深度很深时, 上述树神经网络会存在梯度消失和无法捕获长期依赖的上下文信息的问题, 为此, 他们提出了一种新颖的代码表示方法 (ASTNN), 首先将 AST 拆分成一系列语句级别的小树, 先对每个语句树进行神经嵌入表示, 再使用循环神经网络编码语句之间的顺序和依赖关系得到整个代码的特征表示. 此外, Yang 等人[23]也认为较大的 AST 结构会使得模

型很难学习到代码中关键的语法知识, 他们对 AST 进行简化, 删除了其中冗余的节点, 在软件变更预测任务中验证了方法的有效性. Alon 等人[24]提出了一个神经网络模型用于学习代码嵌入, 他们将 AST 分解为路径集合, 利用神经网络学习每条路径的原子表示, 通过聚合所有路径的表示得到代码的特征向量, 通过预测方法函数名的任务验证了所提模型的有效性.

基于 AST 的语义特征在多个研究中证实了它的有效性, 我们的方法也以 AST 为基础提取代码特征.

1.3 基于图的语义特征

为了利用代码的控制流和数据流信息, Wang 等人[16]构建了流增强的抽象语法树 (FA-AST) 程序图表示, 在 AST 上增加控制流和数据流边; 应用两种不同类型的图神经网络从程序图中提取特征, 他们的工作首次将图神经网络应用到克隆检测领域. 在检测“功能性”代码克隆方面, 现有技术不能有效地从源代码中提取语法和语义信息, Fang 等人[14]提出了一种新的联合代码表示方法, 利用嵌入技术分别获得 AST 和控制流图两种结构特征再将它们结合用于克隆检测任务. Hua 等人[13]提出了 FCCA, 一种基于深度学习的代码克隆检测方法, 从源代码中抽取多种特征, 包括非结构化 (代码的 token) 和结构化 (AST 和控制流图) 信息. 将这些代码特征融合成一个混合表示, 并且在此过程中使用了注意力机制.

编程语言的预训练模型在代码搜索、代码补全等任务上取得了显著的改进. 然而, 现有的预训练模型将代码视为文本, 忽略了代码固有的结构. Guo 等人[15]提出了 GraphCodeBERT 模型, 他们在预训练阶段引入了数据流特征. 研究结果表明, 在代码搜索、克隆检测等 4 个下游任务上实现了最先进的性能.

在漏洞检测领域中, Wang 等人[25]提出了一个基于图的漏洞检测模型 FUNDED, 结合了 AST、程序控制图和程序依赖图来构建程序图, 利用图神经网络学习推理出更好的代码表示用于漏洞检测.

为获得更好的变更预测性能, 本文也利用控制流和数据流信息增强代码表示, 同时我们会删除 AST 中重复的无关紧要的中间节点以学习得到更准确的代码特征, 这也极大地降低了模型学习的难度.

2 基于混合图表示的软件变更预测方法

本文提出的基于混合图表示的软件变更预测方法框架如图 1 所示. 该方法的输入是目标程序的源代码. 首先, 结合 AST 和控制流、数据流等信息构建程序图表征代码, 用于提取语义结构特征. 其中, 第 1 步简化 AST 会删除其中与语法结构无关的中间节点并进行重构得到简化的 AST; 在简化 AST 上显式地增加表示结构语义信息的边以更好地建模复杂的代码结构. 最后利用图神经网络学习程序图的嵌入表示以捕获和推理程序中语法、控制流和数据流关系, 代码嵌入被传递到下游神经网络最终进行预测.

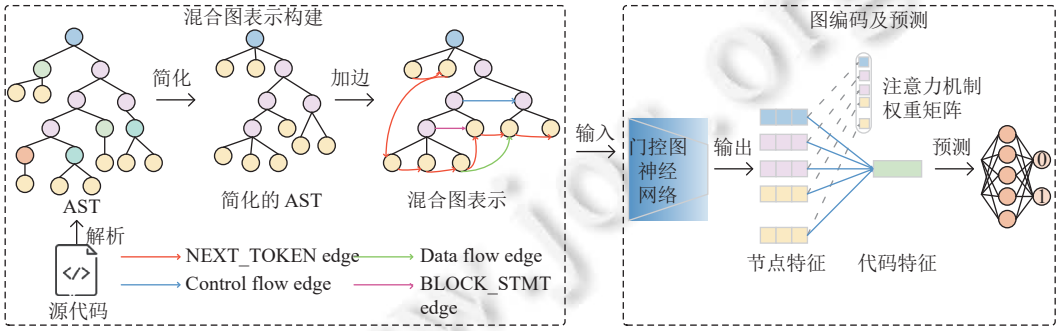


图 1 基于混合图表示的软件变更预测方法框架

2.1 简化 AST

简化 AST 操作将删除与语法结构不相关的中间节点, 仅保留表示语法信息的关键节点. 我们先前的工作[23]总结了关键节点的集合, 见表 1. 在简化过程中会保留 3 种类型的节点: 表 1 中的 AST 关键节点、所有的终端节点和所有与数据流、控制流结构相关的节点, 将所有保留的 3 类节点记为 nodelist.

表 1 AST 关键节点

节点类型	节点名称
声明节点	ClassDeclaration, MethodDeclaration, EnumDeclaration, TypeDeclaration
基本节点	AssertStatement, CatchClauseParameter, CatchClause, ReferenceType, ReturnStatement, MethodInvocation, StatementExpression, TryResource, TryStatement
控制流节点	ForControl, ForStatement, EnhancedForControl, IfStatement, WhileStatement, BreakStatement, ContinueStatement, BlockStatement, SwitchStatement, SwitchStatementCase, DoStatement, ThrowStatement, SynchronizedStatement

简化过程的实现步骤如算法 1 所示. 按深度优先遍历算法遍历 AST 并新建一棵简化的 AST. 算法的输入是原始 AST 的根节点 root 和保留的节点列表 nodelist; 输出是简化后 AST 的根节点 newroot. 值得注意的是, 我们使用 Python 的第三方库 anytree (https://github.com/c0fec0de/anytree) 来构建简化的 AST, 其节点是 anytree 中 AnyNode 类型, newroot 和新建的简化 AST 节点都是该类型的实例. 该类型节点包含 parent 变量用于保存父亲节点, 在简化过程中实现删除节点后对语法树进行重构操作. 算法 1 的第 2, 3 行先获得当前 root 节点的 token 和所有的孩子节点. 第 4 行判断该节点是否在保留列表 nodelist 中, 如果是保留的节点, 接着第 5 行再判断该 AST 节点是否为简化后 AST 的根节点, 如果是简化 AST 的根节点, 则将当前节点信息存储到 newroot 中, 并更新 parent 变量用于重构简化 AST 实现与所有孩子节点相连, 见算法 1 第 6–8 行; 否则新建一个 AnyNode 节点存储原始 AST 节点信息并更新 parent 值, 见算法 1 第 10, 11 行. 如果节点不在保留列表中则直接递归地处理该节点的所有孩子节点, 见算法 1 第 12, 13 行, 直到遍历结束, 将得到新建的简化 AST 并返回其根节点.

算法 1. 简化 AST.

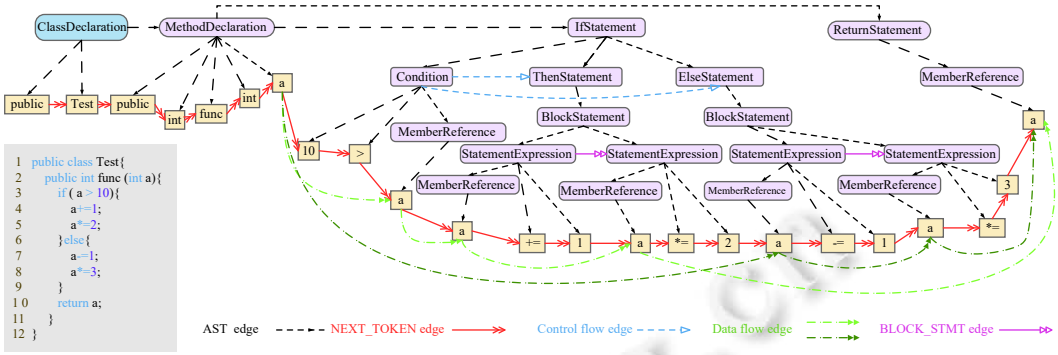
输入: AST 根节点 root, 保留的节点列表 nodelist;
输出: 简化的 AST 根节点 newroot.

```
1. function Simplify(newroot, root, nodelist, parent=None)
2.     token=get_token(root) //得到当前节点 root 的 token
3.     children=get_child(root) //得到当前节点 root 的所有孩子
4.     if token in nodelist //判断是否删除 root, 删除不在 nodelist 中的节点
5.         if newroot is None //判断简化 AST 的根节点是否为空
6.             newroot.token=token //如果为空, 则当前是根节点
7.             newroot.data=root
8.             parent=newroot //更新父节点指针为当前节点
9.         else //不为空的情况, 则新建节点
10.            newnode=AnyNode(token=token, data=root, parent=parent) //新建简化 AST 的节点 AnyNode
11.            parent=newnode
12.    for child in children //递归地简化每个孩子节点
13.        Simplify(newroot, child, nodelist, parent)
14. Simplify(newroot, root, nodelist)
15. Return newroot //返回简化的 AST 根节点
```

2.2 程序图构建

为了更好地建模复杂的代码结构, 我们结合了 AST、控制流和数据流信息构建了程序的混合图表示. 我们在简化的 AST 上添加了 4 种额外类型边. 通过显式地增加控制流、数据流边来提高模型学习程序表示的能力, 从而提升下游变更预测任务的性能.

我们新增的 4 种类型边分别是: 表示代码自然顺序的 NEXT_TOKEN 边、表示代码顺序执行的 BLOCK_STMT 边、数据流边以及控制流边. 我们结合 1 个具体的例子来详细地说明新增的 4 种类型边, 如图 2 所示.




```
1  public class Test{
2    public int func (int a){
3      if ( a > 10){
4        a+=1;
5        a*=2;
6      }else{
7        a-=1;
8        a*=3;
9      }
10     return a;
11   }
12 }
```

图 2 代码混合图示例

图 2 的示例中给出了 1 个代码片段及对应的程序图表示, 用 4 种不同的颜色表示增加的不同类型的边. 我们以基于 Python 开发的开源工具包 javalang 解析的 AST 结构为例进行详细描述.

NEXT_TOKEN 边: 将 AST 的所有终端节点按顺序连接起来, 对应于图 2 中橙色的有向边. 终端节点是源代码中的标识符、变量等, NEXT_TOKEN 边可以捕获代码 token 的自然顺序. 这种关系可以帮助模型识别代码 token 之间的差异.

BLOCK_STMT 边: 将 AST 的 BlockStatement 节点块中的语句按顺序连接起来, 如图 2 中粉色有向边所示, 从左到右按序将 StatementExpression 节点连接起来. BlockStatement 节点对应于代码的循环体、条件控制语句块等域对象, 因此 BLOCK_STMT 边可以捕获代码块中语句执行的顺序信息; BlockStatement 节点的子节点数量是任意的, 特别地, 当拥有多个子节点时, BLOCK_STMT 边可以捕获了较远的语句之间的依赖关系, 这对于理解代码块表达的语义起到关键的作用.

数据流边: 将每个变量节点与下一次使用的节点相连, 在 AST 上增加了变量使用逻辑关系, 如图 2 中绿色有向边所示, 颜色的深浅表示不同的数据流分支. 对于常用的提取 AST 特征的方法包括序列化 AST 成节点序列[11]和利用基于树神经网络的方法[22]都无法捕获远距离相同变量之间的依赖关系, 通过在 AST 上显示地增加数据流边可以将远距离的相同变量联系起来, 从而能够有效地学习变量的使用情况以及相同变量间的依赖关系, 推理程序的数据流知识, 更准确地建模代码中的语义信息, 赋予模型从数据流的角度判断程序变更倾向性的能力.

控制流边: 我们关注于 4 种基本的控制结构, 分别是 If 条件分支、For 循环、While 循环以及 DoWhile 循环. 这些控制结构在各类语言中是普遍存在的, 例如 Switch 分支在 Python 语言中并不支持. 下面我们详细地说明如何增加控制流边.

首先是条件分支 IfStatement 节点, 该节点有 3 个孩子节点, 分别是 Condition、ThenStatement 和 ElseStatement. 这 3 个孩子节点之间是没有关联的, 我们将 Condition 和 ThenStatement 节点连接起来, 表示条件判断为真时代码执行路径; 将 Condition 和 ElseStatement 节点连接起来, 表示条件判断为假时代码执行路径, 如图 2 中蓝色有向边所示. 对于 For 循环 ForStatement 节点, 该节点有两个孩子节点, 分别是 ForControl 和 Body 节点. 将这两个孩子节点连接起来, 表示循环的执行顺序. While 和 DoWhile 循环分别对应于 WhileStatement 和 DoStatement 节点, 它们都有两个孩子节点 Condition 和 Body. 同样地, 将这两个节点相连表示循环的执行顺序. 通过增加控制流边, 模型能够理解代码的执行顺序, 以及学习出分支和循环控制结构的潜在模式, 从而深入地理解代码, 以提取有效的代码特征.

结合代码的 AST、控制流和数据流等语义信息构建出程序的混合图表示, 对代码的语法结构以及丰富的语义关系建模, 可以更好地表征代码理解程序的语义, 获得有效的代码表示, 从而提高下游变更预测任务的性能.

2.3 图编码及预测

图神经网络的快速发展使得处理非欧式空间数据成为可能. 图神经网络在社交网络[26]、知识图谱[27]等领域中取得了惊人的成功. 门控图神经网络 (gated graph neural network, GGNN)[28]在类型预测、克隆检测中表现出了强大的推理和表示能力, 我们利用 GGNN 学习程序图的语义嵌入, 用于预测变更倾向性. 在实验部分我们也会探索不同的图模型的性能. GGNN 具体的学习过程如下.

增强 AST 的程序图是一个有向图 $G=(V,E)$ 它包含简化 AST 节点集合 V 和多种类型边集合 E, 表示两个节点之间存在的关系. 对于每条边, 我们还添加相应的反向边使边数加倍, 这些反向边有助于节点间传递信息. 每个节点维持一个状态 h, 与所有图神经网络一样, 通过邻域聚合方式来更新节点向量表示. 节点通过将其当前状态 (即, 嵌入向量) 作为消息发送给相连的所有邻居来交换信息, 每个节点汇聚所有的邻居消息更新自己的向量状态, 用于下一次迭代. 更新节点状态的过程重复地进行固定次数之后, 通过读出函数将节点状态聚合得到整个图的向量表示.

GGNN 使用多层感知机 (multi-layer perceptron, *MLP*) 作为消息传递函数, 利用门控神经单元 (gate recurrent unit, GRU)[29]更新节点特征. GGNN 的传播过程如下:

$$m_i^t=\sum_{j\in N(i)} MLP(h_i^{t-1},h_j^{t-1},e_{ij}),\forall(i,j)\in E \tag{1}$$

$$z_i^t=\sigma(W^z m_i^t+U^z h_i^{t-1}) \tag{2}$$

$$r_i^t=\sigma(W^r m_i^t+U^r h_i^{t-1}) \tag{3}$$

$$\widetilde{h_i^t}=\tanh(W m_i^t+U(r_i^t\odot h_i^{t-1})) \tag{4}$$

$$h_i^t=(1-z_i^t)\odot h_i^{t-1}+z_i^t\odot\widetilde{h_i^t} \tag{5}$$

其中, 公式 (1) 表示节点 i 在 t 时刻聚合邻居节点的信息, $N(i)$ 是 i 的所有邻居节点, e_{ij} 是该边的嵌入. 根据式 (1) 可计算出 t 时刻节点 i 的消息 m_i^t. 公式 (2)–公式 (5) 是 GRU 更新过程, 利用了 t 时刻的邻居节点消息 m_i^t 和 t–1 时刻节点状态 h_i^{t-1} 计算得到 t 时刻状态. z 和 r 分别是更新门和重置门, $\sigma(x)=1/(1+e^{-x})$ 是逻辑 Sigmoid 函数. W^z,U^z,W^r,U^r,U 是权重矩阵. $\widetilde{h_i^t}$ 是候选状态, 结合状态 h_i^{t-1} 以确定当前状态 h_i^t, tanh 是双曲正切函数. $\odot$ 表示逐元素乘法, Li 等人[28]证实了这种类似 GRU 的传播步骤更有效.

在消息传递阶段, 上述更新过程重复运行 T 步. 在读出阶段, 通过以下读出函数聚合节点状态得到整个图的向量表示[28].

$$h_G=MLP_G\left(\sum_{i\in V}\sigma(MLP_{\text{gate}}(h_i^T))\odot MLP(h_i^T)\right) \tag{6}$$

其中, $\sigma(MLP_{\text{gate}}(h_i^T))$ 充当软注意机制, 决定哪些节点与当前图级任务相关. 在得到全局图表示 h_G 后, 将其传递给标准的全连接网络以使用 Softmax 函数进行分类, 预测其变更倾向性.

3 实验设计

3.1 研究问题

本文提出的 HGSCP 方法, 新颖之处在于程序图表示的构建和利用 GGNN 模型提取代码语义特征. 程序图表示是在简化 AST 上增加了 4 种类型的语义边, 对代码中丰富的语义信息进行建模. 为了说明 HGSCP 方法的有效性, 我们设计了以下 3 个研究问题 (research question, RQ) 详细地讨论 HGSCP 方法的性能表现.

RQ1: 与现有方法相比, 我们提出的 HGSCP 表现如何?

RQ2: 程序图中增加的控制流、数据流等语义信息的效果如何? 哪些语义信息的效果最好?

RQ3: 不同的图神经网络模型对我们的方法性能会产生什么样的影响?

3.2 实验数据

目前, 软件变更预测领域中缺少公开数据集, 研究者们需要构建用于实验的数据集. 我们遵循现有工作[2−4,30], 从开源软件仓库 GitHub 中选择了 8 个开源 Java 项目作为实证研究数据集. 这些项目包括许多演化版本并且涵盖

了各个领域和不同规模的应用, 因此可以全方面地衡量我们方法的性能.

对于数据集的构造, 我们首先确定每个项目的观察时期, 从中选择多个版本作为实验对象, 为其标记变更倾向性的标签. 具体地, 通过分析两个版本之间项目的变更历史为前一版本打标签. 例如, Ambari 项目的观察周期是 2013 年 2 月–2016 年 9 月, 从中选出了 3 个版本作为研究对象, 2013 年 2 月是 1.2.0 版本的发布时间, 利用 1.2.0 版本到 2.1.0 版本之间项目的变更历史为 1.2.0 版本的项目标记变更倾向性, 观察 2.4.1.0 版本到 2016.9 的变更历史为 2.4.1.0 版本标记变更倾向性. 统计观察周期内每个代码文件被修改的次数和累计被修改的代码行数, 结合这两个指标使用先前工作提出的变更标签定义算法[23], 为每个代码文件标记变更倾向性的标签. 由于被标记为变更倾向的样本是较少的, 图神经网络很难学到其中潜在的特征表示, 会极大地影响模型的性能. 因此, 我们利用随机上采样方法增加变更倾向的文件的比例. 表 2 显示了所有选择的项目的详细信息, 包括选择的版本数量、时间窗口、样本数量和对应的具有变更倾向的样本的比例. 我们开源了所有实验的代码和数据 (https://github.com/xylitea/HGSCP).

表 2 实验数据集

项目	版本个数	版本号	时期	样本数量	变更倾向的样本比例 (%)
Ambari	3	1.2.0, 2.1.0, 2.4.1.0	2013.2–2016.9	4721	37
Ant	8	1.6.1, 1.6.5, 1.7.1, 1.8.1, 1.8.4, 1.9.2, 1.9.11, 1.10.1	2004.2–2017.2	7666	39
Jenkins	6	1.312, 1.349, 1.417, 1.503, 1.580.2, 1.599	2009.6–2014.4	6005	35
Argouml	3	0.23.2, 0.25.3, 0.28	2006.9–2009.3	4619	26
Poi	5	1.5.0, 3.1, 3.7, 3.15, 4.0.1	2007.6–2018.11	8680	41
JMeter	5	2.5, 2.13, 3.1, 3.3, 5.0	2011.11–2018.9	5794	42
Hibernate	4	3.6.0, 4.1.1, 5.0.2, 5.0.10	2010.10–2016.11	17208	50
Lucene	2	2.2.0, 6.3.0	2007.6–2016.11	4608	33

3.3 评价指标

在本文中, 我们选取了 3 种使用最为广泛的评价指标来评估变更预测模型的性能: $F1$ 度量 ($F1$-measure)、AUC 和马修斯相关系数 (Matthews correlation coefficient, MCC)[31,32], 这些指标能够评估模型的总体性能.

$F1$ 度量是查准率和查全率的调和平均值, 它是一个重要的分类指标反映模型的整体性, 定义如下:

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{9}$$

其中, TP 是预测模型正确分类的变更倾向模块的数量, TN 是模型正确分类的不具有变更倾向模块的数量, FP 和 FN 是指模型错误地将样本数据分类为变更倾向 (FP) 或非变更倾向 (FN), 未能正确识别出模块变化倾向性的数量.

AUC 是受试者工作特征曲线下的面积, 其中 y 轴代表真阳性率 (TPR), x 轴表示假阳性率 (FPR). AUC 不受样本分布的影响, 能够客观地衡量模型本身的性能. AUC 值越接近 1, 模型的性能越好; 当值为 0.5 时, 模型遵循随机策略.

MCC 也是评估二分类模型性能的指标, 该指标综合考虑了真阳性、真阴性、假阳性和假阴性, 因此 MCC 是一个比较均衡的指标, 当样本分布差距较大时同样适用, 不会出现较大的偏差. 它的取值范围为 [−1, 1], 取值越大表示模型性能越好; 取值为−1 则预测结果与实际分类完全相反. 其定义如下:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{10}$$

3.4 假设检验

假设检验用于衡量两个模型在预测性能上是否存在显着性差异. 我们使用 Wilcoxon 符号秩检验[33]和 Cliff's δ

检验[34]分析我们的方法和基线方法的性能差异.

Wilcoxon 符号秩检验是一种非参数统计假设检验, 不要求数据服从严格的分布, 例如, 正态分布. 在 95% 的置信水平下, 如果 p 值小于 0.05, 则表明两组实验结果之间的差异具有统计学意义, 否则不具有统计学意义.

Cliff's δ 检验是一种效应值大小度量, 用于量化两组数据之间的差异的重要性. 我们使用它来评估两组实验结果之间的差异在实际应用中是否重要, 它被视为对 Wilcoxon 符号秩检验的补充分析. 表 3 显示了 Cliff's δ 值与有效性水平之间的映射.

表 3 Cliff's δ 值与其效应值之间的映射

Cliff's δ值	效应值
$\lvert\delta\rvert < 0.147$	Negligible
$0.147 \leqslant \lvert\delta\rvert < 0.33$	Small
$0.33 \leqslant \lvert\delta\rvert < 0.474$	Medium
$0.474 \leqslant \lvert\delta\rvert$	Large

"Win/Tie/Loss"指标[35]是分析假设检验结果的一种常用手段, 该指标的值计算方法如下: 给定基线方法 M, 如果 HGSCP 优于方法 M, 意味着 Wilcoxon 符号秩的 p 值小于 0.05 且 Cliff's δ 值大于或等于 0.147, 两个模型的实验结果具有显著性并且它们之间的差异是不忽略的. 此时, 将 HGSCP 方法标记为"Win". 相反, 如果方法 M 优于我们的方法, p 值小于 0.05 并且 Cliff's δ 值大于或等于 0.147, 则 HGSCP 将被标记为"Loss". 在其他情况下, 将被标记为"Tie". 通过计算"Win/Tie/Loss"指标评估 HGSCP 和基线方法在所有指标下的性能差异.

3.5 对比方法

为了验证我们的 HGSCP 方法的有效性, 我们与典型的变更预测模型 SASTNN 以及一个新颖的基于 AST 的深度代码表示方法 ASTNN 进行了对比实验.

SASTNN 方法[23]首次将代码语义特征运用到变更预测任务中. 从代码的抽象语法树中提取语义特征, 通过深度优先遍历算法遍历 AST, 挑选出能够表达代码功能和语法等关键 AST 节点, 得到对应的语料库; 利用词嵌入技术学习出每个节点蕴含的语义特征向量, 最后将这些特征输入到门控循环神经单元对代码深度语义特征进行建模, 并预测软件的变更倾向性, 获得了最先进的性能.

ASTNN 方法[12]被提出用于解决树神经网络存在的梯度消失和长期依赖问题, ASTNN 也是一种从抽象语法树中提取有效特征的方法. 它将 AST 按照代码语句的粒度, 将 AST 分割为一系列的语句树, 然后对每个语句树进行编码得到语句级的语法特征向量, 接着利用循环神经网络对所有的语句树向量的自然性进行建模, 得到整个代码的特征表示, 在克隆检测和代码分类任务上都获得了最优的效果.

3.6 参数设置

我们使用的深度学习建模工具 PyTorch 及其扩展库 PyTorch Geometric[36]实现 HGSCP 模型, 所有实验都在 24 核 3.8 GHz CPU 和一个 NVIDIA GeForce RTX 3090 GPU 服务器上进行. 我们使用语法分析工具 javalang 解析代码得到 AST. HGSCP 方法中, 我们遵循现有工作[16], 将图神经网络输出的向量和节点向量的维度设置为 100, 我们使用随机向量初始化节点, 通过前期实验发现随机初始化的效果优于利用 Word2Vec 模型[37]训练得到的节点嵌入. 图神经网络进行 4 次信息交换. 使用 Adam 优化器[38]训练我们的模型, 学习率为 0.01, 批量大小为 32, 每次训练为 10 轮.

4 实验过程与结果分析

本节将详细介绍每个研究问题中的对比方法及相应的实验结果分析.

4.1 针对 RQ1 的结果分析

RQ1 的目标是为了验证 HGSCP 方法的性能. 我们与最新的变更预测模型 SASTNN[23]和一个经典的代码表征模型 ASTNN[12]进行了对比实验, 在 3 个评价指标及多个任务上的实验结果如图 3−图 5 所示.

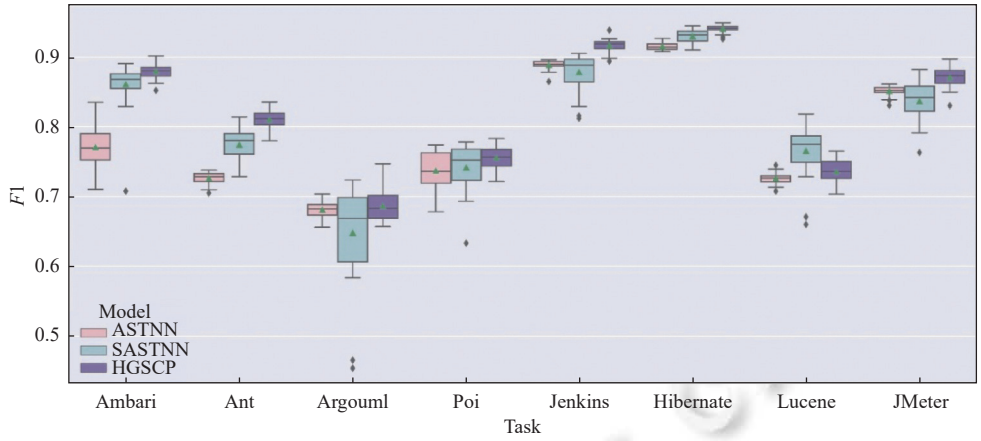


图 3　变更倾向性预测模型性能比较箱线图 (*F*1)

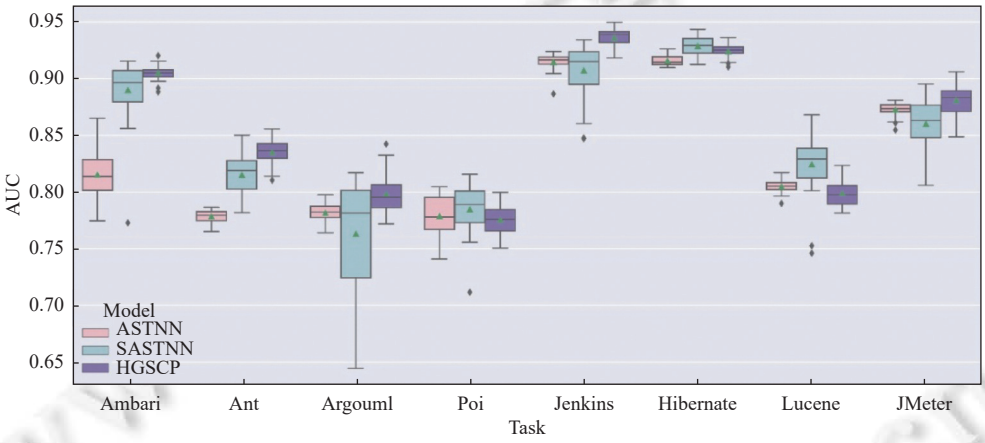


图 4　变更倾向性预测模型性能比较箱线图 (AUC)

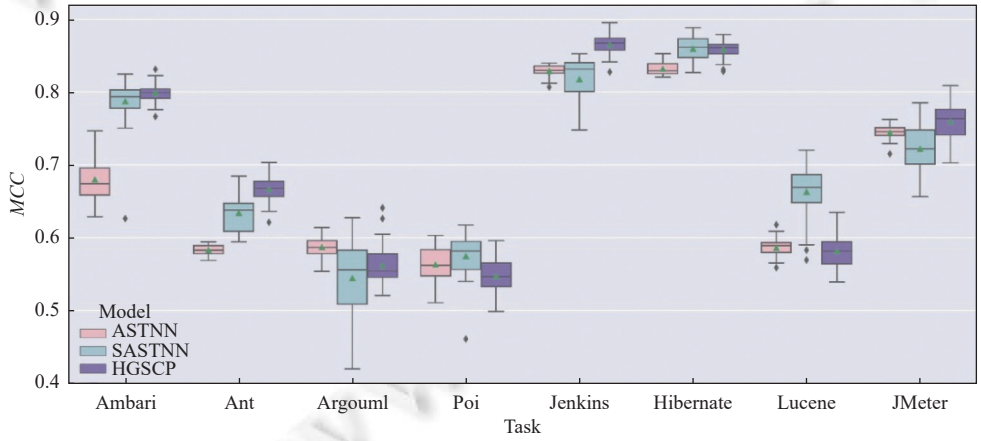


图 5　变更倾向性预测模型性能比较箱线图 (*MCC*)

图 3–图 5 分别给出了 HGSCP 方法与两个对比方法在 *F*1、AUC、*MCC* 度量指标上的箱线图. 箱线图显示了在每个项目上重复 30 次实验的结果, 其中水平线代表中位数, 三角形表示均值. 不在上下边缘线范围内的点为异常值. 由图 3 可以看出在 *F*1 度量上, 我们的方法有着巨大的优势, 在 7 个项目上都获得了最佳的效果; 同样地, 在

AUC 和 *MCC* 指标上 HGSCP 方法也在绝大多数的项目上优于两个对比方法. 通过计算所有项目的平均结果发现在 *F*1 度量、AUC 和 *MCC* 指标上, 我们的方法较 ASTNN 方法分别提高了 3.7%、2.4% 和 2.9%; 较 SASTNN 方法分别提高了 2.0%、1.0% 和 0.5%. 充分地验证了 HGSCP 方法的有效性.

我们进一步分析实验结果之间的显著性, 表 4 和表 5 给出了 HGSCP 与两个基线方法之间 Wilcoxon 符号秩检验以及 Cliff's δ 检验结果. 其中, 每一列显示了 Wilcoxon 符号秩检验的 p 值和 Cliff's δ 值, p 值不小于 0.05 时, 显示原始值, 否则, 将其替换为"<0.05". 我们将 Cliff's δ 值替换为表 3 所示的效应值大小, 同时在效应值前面添加"+"或"–"以分别区分正负 Cliff's δ 值. 例如, 给定 Ambari 项目, HGSCP 与 ASTNN 相比在 *F*1 指标上, p 值小于 0.05 且 Cliff's δ 值大于 0.474, 因此表中内容为"<0.05 (+Large)". 观察两个表中的 Win/Tie/Loss 比较结果, HGSCP 均获得了更多的"Win", 与 ASTNN 相比, 对应于 *F*1、AUC、*MCC* 指标分别在 7、6、5 个项目上显著地优于 ASTNN 方法; 与 SASTNN 相比, 分别在 7、5、3 个项目上具有显著的差异, 这表明 HGSCP 方法与对比方法相比具有显著的性能优势.

表 4 HGSCP vs. ASTNN 假设检验结果

Task	$p(\delta)$		
	*F*1	AUC	*MCC*
Ambari	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
Ant	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
Argouml	0.572 (+Negligible)	<0.05 (+Large)	<0.05 (–Large)
Hibernate	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
Jenkins	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
JMeter	<0.05 (+Large)	<0.05 (+Medium)	<0.05 (+Medium)
Lucene	<0.05 (+Medium)	<0.05 (–Medium)	0.329 (–Small)
Poi	<0.05 (+Medium)	0.271 (–Negligible)	<0.05 (–Medium)
Win/Tie/Loss	7/1/0	6/1/1	5/1/2

表 5 HGSCP vs. SASTNN 假设检验结果

Task	$p(\delta)$		
	*F*1	AUC	*MCC*
Ambari	<0.05 (+Large)	<0.05 (+Medium)	0.153 (+Small)
Ant	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
Argouml	<0.05 (+Small)	<0.05 (+Medium)	0.199 (+Small)
Hibernate	<0.05 (+Large)	<0.05 (–Small)	0.959 (–Negligible)
Jenkins	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
JMeter	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
Lucene	<0.05 (–Large)	<0.05 (–Large)	<0.05 (–Large)
Poi	<0.05 (+Small)	<0.05 (–Medium)	<0.05 (–Large)
Win/Tie/Loss	7/0/1	5/0/3	3/3/2

上述分析证实了 HGSCP 方法在变更预测中可以获得更优的性能表现. 已有方法提取 AST 的语法信息用于变更预测, 他们都将 AST 序列化为 token 列表没有利用 AST 的结构信息, HGSCP 方法不仅充分利用 AST 结构特征, 还融合了代码的语义关系以建模代码复杂的结构特征. 实验结果表明结合 AST 语法结构、控制流和数据流等语义信息可以有效地提高变更预测模型的性能. 综上所述, 与已有工作相比, 我们提出的 HGSCP 方法提取强有力的特征表示用于变更预测, 获得了最先进的效果.

4.2 针对 RQ2 的结果分析

为了探讨增加控制流、数据流等语义信息对模型预测性能的影响, 基于简化 AST, 我们对比了额外增加语义信息 (HGSCP 方法) 和不增加这些语义信息模型的效果. 将仅基于简化 AST 的模型称为 SAST-GGNN, 直接利用 GGNN 学习简化 AST 的特征表示并预测代码的变更倾向性. 实验结果如图 6 所示, 其中 3 个子图分别展示了对比

方法在不同项目上的 *F*1、AUC、*MCC* 指标值. 我们将在后面章节中详细地分析简化 AST 的意义.

图 6 的实验结果表明了增加控制流和数据流信息的 HGSCP 模型获得了更高的性能, 从均值可以看出 HGSCP 方法在 7 个项目上所有的指标均优于 SAST-GGNN 方法. 考虑所有项目的平均结果, HGSCP 方法在 *F*1、AUC、*MCC* 指标上较 SAST-GGNN 方法分别提升了 0.2%、0.4%、0.4%. 表 6 给出了两个模型的统计检验结果, 根据 Win/Tie/Loss 指标两个方法在一半的项目中没有显著性差距, 在其他的项目中, HGSCP 方法获得了更多的“Win”, 这说明了增加的 4 种类型边可以进一步提高变更预测性能.

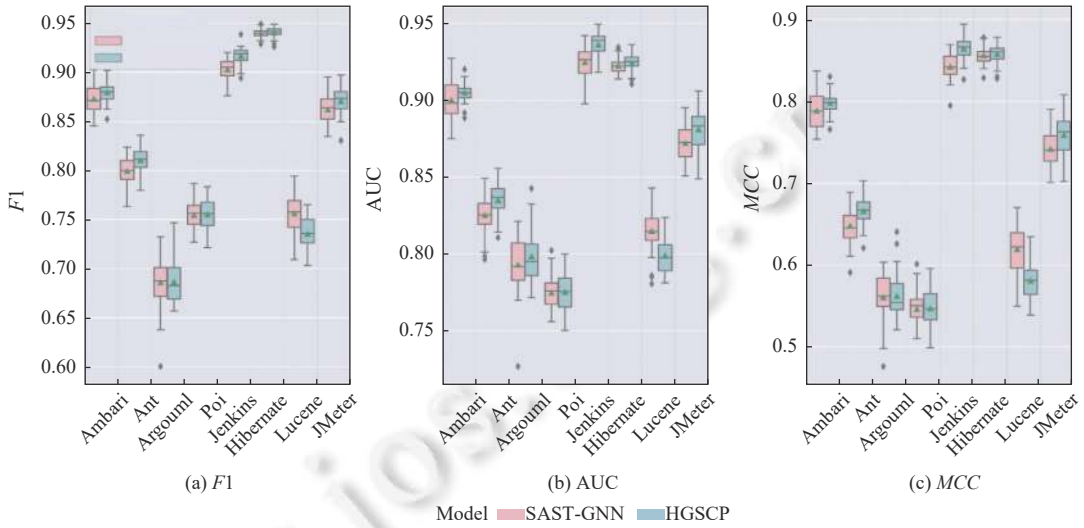


图 6 SAST-GGNN 和 HGSCP 模型性能比较箱线图

表 6 HGSCP vs. SAST-GGNN 假设检验结果

Task	$p(\delta)$		
	*F*1	AUC	*MCC*
Ambari	0.08 (+Small)	0.057 (+Small)	0.057 (+Small)
Ant	<0.05 (+Medium)	<0.05 (+Medium)	<0.05 (+Large)
Argouml	0.829 (–Negligible)	0.349 (+Negligible)	0.829 (–Negligible)
Hibernate	0.339 (+Small)	0.221 (+Small)	0.600 (+Small)
Jenkins	<0.05 (+Large)	<0.05 (+Large)	<0.05 (+Large)
JMeter	<0.05 (+Medium)	<0.05 (+Medium)	<0.05 (+Medium)
Lucene	<0.05 (–Large)	<0.05 (–Large)	<0.05 (–Large)
Poi	0.781 (+Negligible)	0.943 (+Negligible)	0.877 (+Negligible)
Win/Tie/Loss	3/4/1	3/4/1	3/4/1

同时, 也能看出 SAST-GGNN 在 Lucene 项目上显著地优于我们的方法, 对于这一结果我们推测如下: 由统计结果得知该项目平均每个文件有 932 个 AST 节点, 数量是最多的, 说明该项目中的文件的代码较复杂. 此外, 我们也进行了充分的实验证实了基于简化的 AST 构建程序图表示, 模型最终能获得更优的表现 (见第 5.1 节). 对复杂的 AST 进行简化能有效地提升模型的性能, 我们的 HGSCP 方法又在简化的 AST 上增加了多条边, 虽然这些边都表达了各种语义信息, 但也使得程序图表示变得复杂, 模型很难准确地学习到增加的语义特征, 反而导致了性能的下降. 另一个原因是 Lucene 项目的样本数量较少, 当样本数量充足时模型能够学习出代码潜在语义特征并且具有更好的泛化能力.

因此, 在绝大多数的项目上, 增加语义边都能有效地提高预测准确性. 总的来说, 在简化 AST 基础上增加控制流、数据流和代码中 token 自然顺序等表达语义信息的边能进一步提高模型的性能.

接下来, 我们进一步探究各种类型边的作用. 首先对于每个项目我们统计了程序图中各类边的平均数量, 结果见表 7.

表 7 每个项目的平均增加的边的数量

Task	astedge	block (E1)	control (E2)	nexttoken (E3)	nextuse (E4)
Ambari	647	24	12	384	37
Ant	723	26	20	422	57
Argouml	365	13	9	207	23
Hibernate	710	22	15	412	50
Jenkins	568	12	12	336	31
JMeter	623	17	12	369	36
Lucene	715	28	16	447	65
Poi	574	15	10	353	44

表 7 中的 astedge 代表原始 AST 的边; block 表示新增 BLOCKSTATEMNET 中顺序执行的边; control 表示新增 4 种控制结构的执行顺序; nexttoken 表示新增代码中 token 的自然顺序; nextuse 表示新增变量的使用逻辑. 为了简化增加的 4 种边, 我们分别用 E1、E2、E3 和 E4 代替. 从表 7 可以看出构造的程序图中 AST 边是最多的, 其次是 nexttoken 边, 表示控制流信息的 control 边是最少的.

我们的 HGSCP 方法在简化 AST 的基础上增加了这 4 类信息边从而提取出复杂的结构语义特征, 在 HGSCP 方法中依次删除每种类型边观察其性能变化用于确定对于模型性能贡献最大的语义信息, 实验结果见表 8, 每行最优的结果以加粗显示.

表 8 不同类型的边效果

Task	All	All-E1	All-E2	All-E3	All-E4	E1-E2	E1	E2
Ambari	**0.880**	0.872	0.872	0.873	0.874	0.870	0.875	0.875
Ant	**0.810**	0.785	0.792	0.796	0.788	0.796	0.792	0.786
Argouml	**0.687**	0.648	0.650	0.655	0.662	0.653	0.652	0.645
Hibernate	0.941	0.932	0.934	0.942	0.935	**0.943**	0.940	0.939
Jenkins	**0.917**	0.906	0.906	0.903	0.908	0.906	0.906	0.900
JMeter	**0.871**	0.863	0.858	0.869	0.861	0.864	0.867	0.869
Lucene	**0.736**	0.724	0.726	0.729	0.727	0.729	0.734	0.724
Poi	**0.755**	0.729	0.723	0.729	0.723	0.731	0.735	0.728
Average	**0.824**	0.807	0.808	0.812	0.810	0.811	0.813	0.808

表 8 的第 1 列是 HGSCP 方法的 $F1$ 度量, 使用了所有边 (All) 的效果; 第 2–5 列分别表示依次删除 block、control、nexttoken 和 nextuse 边使用其他 3 种类型边进行实验的 $F1$ 值. 观察前 5 列发现使用了所有边的模型在 7 个项目中获得了最高的 $F1$ 值; 从均值也可以看出, 删除任意类型的语义边都会导致模型性能下降. 值得注意的是删除 block 边 (All-E1) 和 control 边 (All-E2) 之后模型性能降低最多, 分别降低了 1.7 和 1.6 个百分点, 说明 block 和 control 边在预测中的贡献最大, 由表 7 可知 block 和 control 边的数量是最少的, 这意味着在变更预测中控制流信息是至关重要的.

最后, 我们探究仅使用贡献最大的 block 边和 control 边的预测模型性能如何. 利用 block 边和 control 边的结合以及分别单独使用它们进行实验, 结果如表 8 第 6–8 列所示. 在这 3 组实验中, 可以看出仅利用两种控制流信息不能获得最高的性能, 说明多种结构信息的融合是提升性能的一种有效手段. 值得注意的是仅利用 block 边的模型获得了 0.813 的 $F1$ 值, 结果仅次于我们的 HGSCP 方法. 从表 7 可以看出 block 边的数量也是较少的, 因此构建的程序图不会过于复杂, 模型训练需要更少的时间开销. 对于模型性能要求不高时, 在简化 AST 的基础上, 仅新增利用 block 边不仅可以获得仅次于 HGSCP 的预测性能, 且时间开销较之 HGSCP 更少.

综上所述, 我们在简化 AST 的基础上增加 4 种类型的语义边获得了最优的预测性能, 在所有语义信息中新增的 block 边即循环体、条件控制语句块等域对象中语句执行顺序信息对于模型性能的贡献最大.

4.3 针对 RQ3 的结果分析

本节探讨不同的图神经网络模型对我们方法性能的影响. 表 9 展示了使用不同图神经网络模型的性能表现,

包括图卷积网络 (graph convolutional network, GCN)[39,40]、关系图卷积网络 (relational graph convolutional network, RGCN)[41]以及图注意网络 (graph attention network, GAT)[42].

表 9 不同图神经网络的效果

Task	GGNN (ours)			GCN			RGCN			GAT		
	*F*1	AUC	*MCC*	*F*1	AUC	*MCC*	*F*1	AUC	*MCC*	*F*1	AUC	*MCC*
Ambari	0.880	0.905	0.799	0.669	0.770	0.558	0.872	0.897	0.786	0.866	0.897	0.780
Ant	0.810	0.835	0.666	0.804	0.829	0.655	0.803	0.829	0.656	0.771	0.808	0.633
Argouml	0.687	0.798	0.562	0.660	0.767	0.535	0.633	0.750	0.503	0.684	0.788	0.559
Hibernate	0.941	0.924	0.858	0.947	0.929	0.871	0.945	0.928	0.868	0.944	0.925	0.865
Jenkins	0.917	0.936	0.865	0.914	0.937	0.862	0.917	0.932	0.866	0.921	0.940	0.869
JMeter	0.871	0.881	0.759	0.874	0.882	0.761	0.857	0.869	0.739	0.869	0.875	0.749
Lucene	0.736	0.799	0.581	0.742	0.803	0.587	0.738	0.807	0.587	0.706	0.774	0.535
Poi	0.755	0.775	0.547	0.724	0.762	0.522	0.753	0.770	0.538	0.757	0.781	0.558
Average	**0.824**	**0.857**	**0.705**	0.792	0.835	0.669	0.815	0.848	0.693	0.815	0.848	0.694

GCN 利用卷积操作从图数据中提取特征, 它的输入是节点特征矩阵和邻接矩阵, 邻接矩阵是共享的, 通过迭代不断更新节点特征. RGCN 是为了对关系数据建模而开发的, 并且扩展了 GCN 模型, 能获得更好的鲁棒性. 在计算图节点特征时, 先获得相邻节点的向量, 然后针对每种关系类型进行单独转换, 最后累积所有类型的特征并通过激活函数更新该图节点. GAT 与其他图神经网络一样融合邻居节点特征得到当前节点的特征表示, 但它认为不同的邻居节点对节点特征的更新起着不同的作用, GAT 通过自注意力机制来对邻居结点进行聚合, 对不同的邻居自适应地分配不同的权值, 从而提高模型的性能.

首先, 从均值可以看出 RGCN 的实验结果在 3 个指标上均优于 GCN, 这也证实了 Schlichtkrull 等人[40]的工作, 扩展后的模型 RGCN 可以获得比 GCN 更好的性能. 其次, RGCN 和 GAT 的实验结果较相近, 在 *F*1 度量和 AUC 上获得了相同的指标值; GAT 在 *MCC* 指标上比 RGCN 高 0.1%. 这两种模型在聚合邻居结点时与普通的图神经网络不同, RGCN 考虑了节点之间不同的关系, GAT 为邻居节点分配不同的权值, 实验结果表明这些模型的改进是有效性. 最后, 将这 3 种图神经网络作为编码器提取语义特征用于预测, 最终都能获得了较好的预测结果, 这说明我们的 HGSCP 方法不依赖于特定的图模型. HGSCP 在 *F*1、AUC 和 *MCC* 指标上的平均结果分别是 0.842、0.857 和 0.705, 与这 3 个模型相比, 我们使用 GGNN 作为图编码器的 HGSCP 方法获得了最优表现. GGNN 在聚合邻居结点信息时利用了 GRU 的门控更新方式, 其中忘记门会过滤对节点更新不重要的信息从而学习出准确的特征表示, 结果表明利用 GGNN 模型提取代码语义特征能更准确地预测代码的变更倾向性.

综合而言, 我们的方法在不同的图编码器下都能获得较好的性能, 基于 GGNN 的 HGSCP 方法获得了最优的性能表现.

4.4 实验小结

本节我们开展了多组对比实验来验证所提方法 HGSCP 的有效性. 首先, RQ1 与最新的变更预测方法、经典的代码表征模型进行对比, 在 *F*1 指标上分别提高了 2 和 3.7 个百分比, 获得了最先进的性能, 充分说明了我们方法的有效性. 接着, RQ2 分析了 HGSCP 方法中增加语义信息的意义. RQ2 实验结果显示: block 边即代码块中语句执行顺序对于模型性能的贡献最大, 多种语义信息的结合能获得更高的性能. 最后, RQ3 对比了 4 种不同图模型的效果. 所有的图模型应用到我们的方法中都能获得较好的性能, 本文所提的 HGSCP 方法使用 GGNN 在 *F*1、AUC、*MCC* 评价指标上的结果分别为 0.824、0.857、0.705, 在所对比的图网络模型中获得了最优的性能表现. 上述 3 个研究问题充分验证了本文 HGSCP 方法的有效性.

5 讨 论

5.1 简化 AST 的作用

在本节, 我们讨论 HGSCP 方法中简化 AST 的意义. 为了回答该问题, 我们分别在原始 AST 和简化后的 AST

上进行对比实验, 利用 GGNN 学习两种 AST 的语法结构特征来预测变更倾向性. 为了便于比较, 将这两个模型分别记为 AST-GGNN 和 SAST-GGNN. 仅使用 AST 信息, 没有增加额外的语义边. 实验结果如图 7 所示.

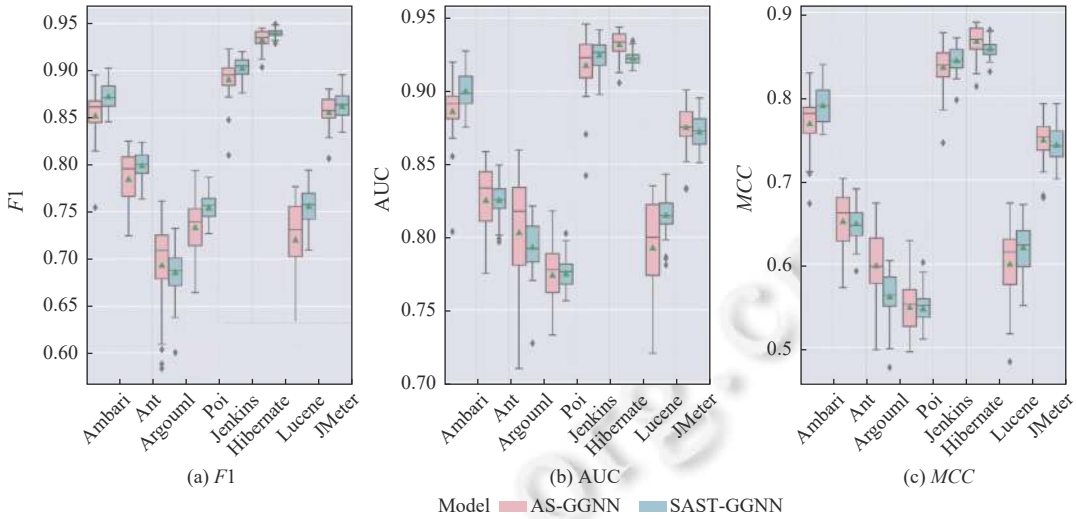


图 7 AST-GGNN 和 SAST-GGNN 模型性能比较箱线图

从图 7 可以看出, SAST-GGNN 模型在 *F*1、AUC 和 *MCC* 指标上分别在 7、5 和 3 个项目上优于 AST-GGNN 模型, 在 *F*1 和 AUC 指标上所有项目上的均值较对比模型分别提高了 1.4%、0.2%; 在 *MCC* 指标上基于 AST 的方法的性能则较优, 但较之 SAST-GGNN 仅提高了 0.1%. 箱线图中箱子的长度表示了数据的变化范围. 从图 7 中可以看出: 与 AST-GGNN 方法相比, SAST-GGNN 方法的实验结果箱子长度更短, 说明基于简化 AST 的方法实验结果波动更小, 模型更稳定. 未简化的 AST 中重复冗余的节点对模型稳定性有较大的影响.

从表 10 的统计检验结果中可以看出, 与 AST-GGNN 方法相比, SAST-GGNN 在 *F*1 和 AUC 指标上获得了更多的“Win”, 这说明 SAST-GGNN 方法是显著地优于基于 AST 的方法. 在 MCC 指标上两个方法都获得了相同的“Win”, 两个方法的性能差距不具有显著性. 因此, 可以得出结论: 基于简化 AST 的方法 SAST-GGNN 可以进一步提升基于原始 AST 方法的性能. AST 的树状结构表示了代码的语法特征, 其中存在重复的对于语法结构的表达不重要的节点, 这些节点增大了提取准确语法特征的困难, 删除这些节点可以获得更好的性能. 总的来说, 我们的简化 AST 操作是有效的, 可以进一步提高预测性能.

表 10 SAST-GGNN vs. AST-GGNN 假设检验结果

Task	$p(\delta)$		
	*F*1	AUC	*MCC*
Ambari	<0.05 (+Large)	<0.05 (+Medium)	<0.05 (+Medium)
Ant	0.094 (+Small)	0.975 (–Negligible)	0.629 (–Negligible)
Argouml	0.237 (–Small)	0.197 (–Small)	<0.05 (–Large)
Hibernate	<0.05 (+Medium)	<0.05 (–Large)	<0.05 (–Medium)
Jenkins	<0.05 (+Medium)	0.197 (+Small)	0.329 (+Small)
JMeter	0.360 (+Small)	0.299 (–Small)	0.262 (–Small)
Lucene	<0.05 (+Large)	<0.05 (+Medium)	<0.05 (+Small)
Poi	<0.05 (+Large)	0.976 (–Negligible)	0.704 (–Negligible)
Win/Tie/Loss	5/3/0	2/5/1	2/4/2

为了验证简化 AST 操作能否降低模型的时间开销, 我们首先统计了每个项目在简化前后的 AST 节点数量以及简化的比例如表 11 所示, 表中数量是项目中所有代码文件的平均节点数量. 同时, 表 11 也列出了 AST-GGNN 和 SAST-GGNN 模型的时间比较.

表 11 AST 与 SAST 节点数量和对应模型时间开销比较

Task	节点数量			时间开销 (min)	
	AST	SAST	Rate (%)	AST	SAST
Ambari	780	649	17	4.57	4.07
Ant	896	725	19	7.96	7.23
Argouml	459	367	20	3.66	3.23
Hibernate	763	711	7	17.02	14.66
Jenkins	702	570	19	5.62	4.99
JMeter	763	624	18	5.75	5.16
Lucene	932	717	23	5.65	7.39
Poi	721	576	20	8.35	7.30
Average	752	617	18	7.32	6.75

从表 11 可以看出, 对于原始的 AST 平均会删除其中 18% 的节点, 相应的程序图复杂度会降低, 从而模型需要更少的训练测试时间, 表 11 的结果也佐证了这一点. SAST-GGNN 在 7 个项目上的所需的时间开销均低于 AST-GGNN, 从平均结果可以看出 SAST-GGNN 比基于原始 AST 的方法少了 0.57 min. 降低的时间花费与简化 AST 节点成正比, 删除的节点越多, 模型需要的时间越少, 但删除节点的同时也会造成语法结构信息的丢失, 平衡简化操作和其带来的时间效益是一个重要的问题. 我们在简化过程中保留了表 1 中的节点, 使简化后的 AST 能保留其中关键的结构信息, 实验结果证实了我们简化 AST 的有效性, 同时能够降低模型的时间开销.

值得注意的是, 对于大型的项目来说, 简化操作可以极大地降低模型的时间开销, 其中 Hibernate 项目的样本数最多, 简化后模型的时间花费减少了 2.36 min. 但在 Lucene 项目上 SAST-GGNN 需要更多的时间, 我们发现该项目中平均每个文件的节点个数为 932, 是所有项目中最多的, 简化操作是一个递归的过程, 较长的代码文件需要更多的时间, 虽然简化后的训练测试时间减少了, 仍然不能弥补简化操作所需要的大量时间. 简化操作带来的时间效益取决于简化 AST 操作增加的时间成本和减少的训练测试时间. 在绝大多数的项目上, 简化后的 AST 都可以减小模型的时间成本.

综合而言, 简化后的 AST 尽量保留了重要的语法结构信息, 可以获得比原始 AST 更好的性能表现, 验证了简化操作的有效性, 同时也降低了模型的时间开销, 基于简化 AST 构建程序图表示能进一步提高模型的性能.

5.2 有效性威胁

5.2.1 内部威胁

内部有效性威胁源于我们实验的局限性, 实验环境的正确性对内部有效性具有重要意义. 在本研究中, 我们使用 PyTorch 深度学习框架构建 HGSCP 模型, 利用 javalang 将源代码转换为相应的 AST, 使用 PyTorch Geometric 库来实现图神经网络. 这些工具被广泛应用于各种研究, 因此它们是可靠的. 另一方面, 代码的实现可能包含潜在的错误, 这会影响方法的准确性. 为了降低该内部威胁, 我们对源代码进行了严格的代码审查, 并进行了充分的白盒测试.

5.2.2 外部威胁

外部有效性是指模型的泛化能力, 我们选取了 8 个开源项目进行数据集的构建, 然而这些项目并不能代表所有的情况. 对于其他项目, 我们提出的方法可能会产生更好或更差的结果. 为了减少对数据集构建的影响, 我们尽量选择来自不同应用领域、不同规模的项目, 这些项目具有不同的特征, 如文件数量、变更比例等, 尽可能保证数据的多样性, 并且这些项目维护的时间长, 为研究提供了足够的数据. 此外, 我们的方法仅在 Java 语言中进行了评估. 应用于其他编程语言时, 模型的效果无法保证. 在其他数据集上的实验将有助于进一步验证我们方法的有效性.

5.2.3 构造威胁

本研究仅关注了模型在机器学习领域提出的性能指标上的表现, 这些指标无法衡量模型在实际应用中的效果, 为了能够评估模型在实际应用中性能表现, 研究者们提出了工作量感知 (effort-aware or cost-effectiveness) 指标

对软件工程任务的预测技术进行性能评估. 工作量感知指标是指当开发者根据预测模型的预测结果进行代码审查时, 审查一定数量的代码 (即工作量) 所能检查到的目标问题的数量或者比例, 该指标能够衡量模型在实际应用中的效果. 在未来的工作中引入工作量感知指标来评估变更预测模型的实际应用效果有助于全方面地验证我们方法的有效性.

6 总 结

软件变更预测旨在预测出具有变更倾向的代码模块, 有目的地关注这些变更倾向的模块, 提前解决其中潜在的问题, 可以有效地提高软件质量并降低维护的开销. 针对现有的工作基于代码的传统度量或从序列化的 AST 中抽取特征预测变更倾向性, 他们不能很好地表征代码, 也未利用代码中丰富的结构语义信息, 本文提出了一种基于混合程序图表示的变更预测方法 HGSCP.

本文首先构造出一种新的程序图表示, 从代码中抽取语法特征以及控制流、数据流等语义信息, 将它们结合来表征源代码. 然后, 利用门控图神经网络从程序图表示中学习代码的特征向量表示, 最后根据学习的语义特征预测代码模块的变更倾向性. 通过在大量变更数据集的多组实验以及假设检验比较, 验证了本文所提出的基于混合程序图表示的变更预测方法可以获得更高的预测性能, 同时也验证了我们的程序图构造的有效性.

目前, 软件变更预测的研究主要集中于有监督的方法, 而对于新项目或历史信息有限的项目无法获得标签数据, 这些方法将不再适用. 在未来的工作中, 我们将利用自监督学习方法进行变更预测, 而无需标签信息. 此外, 面向编程语言的预训练模型很少引入控制流图、数据流图等语义信息, 我们将加入语义信息进行预训练, 以提高模型对代码的表征能力.

References:

[1] Schneidewind NF. Measuring and evaluating maintenance process using reliability, risk, and test metrics. IEEE Trans. on Software Engineering, 1999, 25(6): 769–781. [doi: 10.1109/32.824387]

[2] Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A. Enhancing change prediction models using developer-related factors. Journal of Systems and Software, 2018, 143: 14–28. [doi: 10.1016/j.jss.2018.05.003]

[3] Zhu XY, He YY, Cheng L, Jia XL, Zhu L. Software change-proneness prediction through combination of bagging and resampling methods. Journal of Software: Evolution and Process, 2018, 30(12): e2111. [doi: 10.1002/smr.2111]

[4] Yan M, Zhang XH, Liu C, Xu L, Yang MN, Yang D. Automated change-prone class prediction on unlabeled dataset using unsupervised method. Information and Software Technology, 2017, 92: 1–16. [doi: 10.1016/j.infsof.2017.07.003]

[5] Zhou YM, Leung H, Xu BW. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Trans. on Software Engineering, 2009, 35(5): 607–623. [doi: 10.1109/TSE.2009.32]

[6] Liu HH, Yu YJ, Li BX, Yang YB, Jia R. Are smell-based metrics actually useful in effort-aware structural change-proneness prediction? An empirical study. In: Proc. of the 25th Asia-Pacific Software Engineering Conf. (APSEC). Nara: IEEE, 2018. 315–324. [doi: 10.1109/APSEC.2018.00046]

[7] Catolino G, Palomba F, Fontana FA, De Lucia A, Zaidman A, Ferrucci F. Improving change prediction models with code smell-related information. Empirical Software Engineering, 2020, 25(1): 49–95. [doi: 10.1007/s10664-019-09739-0]

[8] Elish MO, Al-Khiaty MAR. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. Journal of Software: Evolution and Process, 2013, 25(5): 407–437. [doi: 10.1002/smr.1549]

[9] Catolino G, Ferrucci F. An extensive evaluation of ensemble techniques for software change prediction. Journal of Software: Evolution and Process, 2019, 31(9): e2156. [doi: 10.1002/smr.2156]

[10] Malhotra R, Khanna M. Dynamic selection of fitness function for software change prediction using particle swarm optimization. Information and Software Technology, 2019, 112: 51–67. [doi: 10.1016/j.infsof.2019.04.007]

[11] 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]

[12] Zhang J, Wang X, Zhang HY, Sun HL, Wang KX, Liu XD. A novel neural source code representation based on abstract syntax tree. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]

[13] Hua W, Sui YL, Wan Y, Liu GZ, Xu GD. FCCA: Hybrid code representation for functional clone detection using attention networks. IEEE Trans. on Reliability, 2021, 70(1): 304–318. [doi: 10.1109/TR.2020.3001918]

[14] Fang CR, Liu ZX, Shi YY, Huang J, Shi QK. Functional code clone detection with syntax and semantics fusion learning. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 516–527. [doi: 10.5281/zenodo.3895414]

[15] Guo DY, Ren S, Lu S, Feng ZY, Tang DY, Liu SJ, Zhou L, Duan N, Svyatkovskiy A, Fu SY, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang DX, Zhou M. Graphcodebert: Pre-training code representations with data flow. In: Proc. of the 9th Int'l Conf. on Learning Representations. ICLR, 2020.

[16] Wang WH, Li G, Ma B, Xia X, Jin Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 261–271. [doi: 10.1109/SANER48275.2020.9054857]

[17] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: ICLR, 2018.

[18] Lu HM, Zhou YM, Xu BW, Leung H, Chen L. The ability of object-oriented metrics to predict change-proneness: A meta-analysis. Empirical Software Engineering, 2012, 17(3): 200–242. [doi: 10.1007/s10664-011-9170-z]

[19] Malhotra R, Kapoor R, Aggarwal D, Garg P. Comparative study of feature reduction techniques in software change prediction. In: Proc. of the 18th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Madrid: IEEE, 2021. 18–28. [doi: 10.1109/MSR52588.2021.00015]

[20] Zhu XY, Li N, Wang Y. Software change-proneness prediction based on deep learning. Journal of Software: Evolution and Process, 2022, 34(4): e2434. [doi: 10.1002/smr.2434]

[21] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence. Phoenix: AAAI Press, 2016. 1287–1293.

[22] Wei HH, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. Melbourne: AAAI Press, 2017. 3034–3040.

[23] Yang XY, Zhang XF, Tong Y. Simplified abstract syntax tree based semantic features learning for software change prediction. Journal of Software: Evolution and Process, 2022, 34(4): e2445. [doi: 10.1002/smr.2445]

[24] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 3: 40. [doi: 10.1145/3290353]

[25] Wang HT, Ye GX, Tang ZY, Tan SH, Huang SF, Fang DY, Feng YS, Bian LZ, Wang Z. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. on Information Forensics and Security, 2021, 16: 1943–1958. [doi: 10.1109/TIFS.2020.3044773]

[26] Gao HY, Wang ZY, Ji SW. Large-scale learnable graph convolutional networks. In: Proc. of the 24th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining. London: ACM, 2018. 1416–1424. [doi: 10.1145/3219819.3219947]

[27] Wu YT, Liu X, Feng YS, Wang Z, Zhao DY. Jointly learning entity and relation representations for entity alignment. 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. 240–249. [doi: 10.18653/v1/D19-1023]

[28] 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: ICLR, 2015.

[29] Cho K, van Merriënboer B, Bahdanau D, Bengio Y. On the properties of neural machine translation: Encoder-decoder approaches. In: Proc. of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation. Doha: ACL, 2014. 103–111. [doi: 10.3115/v1/W14-4012]

[30] Zhuang WY, Wang H, Zhang XF. Just-in-time defect prediction based on AST change embedding. Knowledge-based Systems, 2022, 248: 108852. [doi: 10.1016/j.knosys.2022.108852]

[31] Baeza-Yates R, Ribeiro-Neto B. Modern Information Retrieval. New York: ACM Press, 1999.

[32] Baldi P, Brunak S, Chauvin Y, Andersen CAF, Nielsen H. Assessing the accuracy of prediction algorithms for classification: An overview. Bioinformatics, 2000, 16(5): 412–424. [doi: 10.1093/bioinformatics/16.5.412]

[33] Xing Y, Qian XM, Guan Y, Zhang SH, Zhao MC, Lin WT. Cross-project defect prediction method using adversarial learning. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2097–2112 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6571.htm [doi: 10.13328/j.cnki.jos.006571]

[34] Jia XY, Zhang WZ, Li WW, Huang ZQ. Feature representation method for heterogeneous defect prediction based on variational autoencoders. Ruan Jian Xue Bao/Journal of Software, 2021, 32(7): 2204–2218 (in Chinese with English abstract). http://www.jos.org.cn/

1000-9825/6257.htm [doi: 10.13328/j.cnki.jos.006257]
[35] Wang H, Zhuang WY, Zhang XF. Software defect prediction based on gated hierarchical LSTMs. IEEE Trans. on Reliability, 2021, 70(2): 711–727. [doi: 10.1109/TR.2020.3047396]
[36] Fey M, Lenssen JE. Fast graph representation learning with PyTorch Geometric. arXiv:1903.02428, 2019.
[37] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 26th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: Curran Associates Inc., 2013. 3111–3119.
[38] Kingma DP, Ba J. Adam: A method for stochastic optimization. In: Proc. of the 3rd Int'l Conf. on Learning Representations. San Diego: ICLR, 2014.
[39] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: Proc. of the 5th Int'l Conf. on Learning Representations. Toulon: ICLR, 2016.
[40] Zhao G, Wang QG, Yao F, Zhang YF, Yu G. Survey on large-scale graph neural network systems. Ruan Jian Xue Bao/Journal of Software, 2022, 33(1): 150–170 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6311.htm [doi: 10.13328/j.cnki.jos.006311]
[41] Schlichtkrull M, Kipf TN, Bloem P, van den Berg R, Titov I, Welling M. Modeling relational data with graph convolutional networks. In: Proc. of the 15th European Semantic Web Conf. Heraklion: Springer, 2018. 593–607. [doi: 10.1007/978-3-319-93417-4_38]
[42] Veličković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. arXiv:1710.10903, 2017.

附中文参考文献:

[33] 邢颖, 钱晓萌, 管宇, 章世豪, 赵梦赐, 林婉婷. 一种采用对抗学习的跨项目缺陷预测方法. 软件学报, 2022, 33(6): 2097–2112. http://www.jos.org.cn/1000-9825/6571.htm [doi: 10.13328/j.cnki.jos.006571]
[34] 贾修一, 张文舟, 李伟漳, 黄志球. 基于变分自编码器的异构缺陷预测特征表示方法. 软件学报, 2021, 32(7): 2204–2218. http://www.jos.org.cn/1000-9825/6257.htm [doi: 10.13328/j.cnki.jos.006257]
[40] 赵港, 王千阁, 姚烽, 张岩峰, 于戈. 大规模图神经网络系统综述. 软件学报, 2022, 33(1): 150–170. http://www.jos.org.cn/1000-9825/6311.htm [doi: 10.13328/j.cnki.jos.006311]

杨馨悦(1997－), 女, 硕士生, CCF 学生会员, 主要研究领域为软件工程, 软件变更预测.

陈林(1979－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件分析与测试.

刘安(1981－), 男, 博士, 教授, CCF 专业会员, 主要研究领域为数据库与工业软件, 众包计算, 数据安全与隐私.

章晓芳(1980－), 女, 博士, 教授, CCF 专业会员, 主要研究领域为智能软件工程, 软件分析、测试与维护.

赵雷(1972－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为数据管理与数据分析, 数据智能与软件, 知识工程, 机器学习.