软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2085−2102 [doi: 10.13328/j.cnki.jos.007458] [CSTR: 32375.14.jos.007458]

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

基于图 Transformer 的变更集错误定位方法*

刘 浩[1], 杜军威[1], 李玉莹[1], 房敏营[1], 贾学海[2]

[1](青岛科技大学 数据科学学院, 山东 青岛 266061)

[2](青岛科技大学 信息科学技术学院, 山东 青岛 266061)

通信作者: 杜军威, E-mail: dujunwei@qust.edu.cn; 李玉莹, E-mail: Liyy@qust.edu.cn

摘 要: 错误定位是软件维护过程中的关键环节, 如何提升自动化故障定位的有效性和效率是软件工程领域的研究焦点之一. 随着开源软件数量激增且软件热更新需求增多, 面向变更集的自动化错误定位成为软件质量保障的重要手段. 传统基于信息检索的错误定位方法只能表征自身文本信息, 未能充分考虑变更集中的结构和语义变化, 无法直接应用于变更集的错误定位任务. 因此, 提出一种基于图 Transformer 的变更集错误定位方法, 使用基于变更信息抽象语法树表征代码结构变化信息, 并从局部和全局角度表征变更代码和错误报告的语义信息, 进而实现变更集中错误信息的匹配和定位. 对来自 6 个错误诱发变更集的错误报告和变更进行测试, 与最先进模型相比, *MAP* 和 *MRR* 指标分别提升 11.4% 和 12.9%, 证明了该方法的有效性.

关键词: 错误定位; 图表示; 错误诱发变更集; 抽象语法树; 错误报告

中图法分类号: TP311

中文引用格式: 刘浩, 杜军威, 李玉莹, 房敏营, 贾学海. 基于图Transformer的变更集错误定位方法. 软件学报, 2026, 37(5): 2085–2102. http://www.jos.org.cn/1000-9825/7458.htm

英文引用格式: Liu H, Du JW, Li YY, Fang MY, Jia XH. Change Set Bug Localization Method Based on Graph Transformer. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2085–2102 (in Chinese). http://www.jos.org.cn/1000-9825/7458.htm

Change Set Bug Localization Method Based on Graph Transformer

LIU Hao[1], DU Jun-Wei[1], LI Yu-Ying[1], FANG Min-Ying[1], JIA Xue-Hai[2]

[1](College of Data Science, Qingdao University of Science & Technology, Qingdao 266061, China)

[2](College of Information Science and Technology, Qingdao University of Science & Technology, Qingdao 266061, China)

Abstract: Bug localization is a critical aspect of software maintenance, and improving the effectiveness and efficiency of automated fault localization has become a central research focus in software engineering. With the surge in open-source software and the increasing demand for software hot updates, automated bug localization focused on change sets has become a key tool for software quality assurance. Traditional bug localization methods based on information retrieval can only represent textual information and fail to fully account for structural and semantic changes within change sets, making them unsuitable for direct application in change set bug localization tasks. Therefore, this study proposes a method for change set bug localization based on graph Transformer, which uses an abstract syntax tree to represent change information and capture code structure changes. The method represents both local and global semantic information of the changed code and bug reports, enabling the matching and localization of bug information within change sets. It is tested on bug reports and changes from six sets of bug-inducing change sets. Compared to the state-of-the-art models, the proposed method demonstrates improvements of 11.4% and 12.9% in *MAP* and *MRR* metrics, respectively, validating the efficacy of the proposed approach.

Key words: bug localization; graph representation; bug-inducing change set; abstract syntax tree (AST); bug report

---

* 基金项目: 国家自然科学基金 (62202253, 6217072142); 山东省自然科学基金 (ZR2024QF199, ZR2021MF092, ZR2022MF326, ZR2021QF074)
收稿时间: 2024-12-09; 修改时间: 2025-01-27, 2025-03-27; 采用时间: 2025-04-29; jos 在线出版时间: 2025-09-10
CNKI 网络首发时间: 2025-09-11

近年来, 软件应用市场竞争激烈, 快速响应市场需求至关重要. 为满足用户的多样化需求并实现软件功能的迅速迭代, 开发人员需要进行频繁的代码变更和维护. 这种变更不仅增加了版本发布的数量, 还引发了大量错误, 活跃项目每日的提交次数可高达数百次, 伴随的错误数量甚至超过提交量[1]. 这些错误不仅影响系统正常功能, 还可能引发数据泄露和系统崩溃等严重问题[2]. 因此, 准确定位错误产生的变更版本成为维护软件系统稳定性和安全性的核心任务.

随着软件系统规模的不断扩展和更新迭代速度的加快, 软件的变更变得愈加复杂. 继续依赖传统的手动定位方法, 显然无法满足快速、精准定位错误的需求, 从而可能导致软件发布的延迟和质量的下降, 进而影响用户体验. 研究表明, 现代软件系统平均每 2–3 周发布一个新版本, 而像 Linux 内核这样的大型开源项目, 其每日的变更次数甚至可达数百次. 频繁的版本变更已经成为软件开发的常态, 但这一趋势也使得错误定位变得愈加困难. 尤其是在大量变更中, 产生的错误数量往往超过提交的变更数, 且这些错误不仅可能影响系统的基本功能, 还可能引发数据泄露、系统崩溃等严重后果, 威胁到软件系统的稳定性和安全性. 因此, 如何准确地将故障归因于具体的变更集, 已经成为软件工程领域亟待解决的核心问题[3]. 在此背景下, 面向变更集的自动化错误定位技术应运而生, 成为软件工程研究中的长期挑战, 并且被认为是保障软件快速变更和高质量迭代的重要手段[4].

基于文本信息检索的方法将变更序列视为文本信息, 通过计算错误报告与变更序列的文本相似度进行匹配. Locus[5]首次在基于信息检索 (IR) 的错误定位技术中使用变更集构建文本语料库. 该研究在语料库中检索文档, 与错误报告进行匹配, 进而实现错误定位. FBL-BERT[6]将 BERT 模型应用于错误变更集的定位, 将错误报告与代码的局部信息匹配进行错误定位. 这些研究取得了一定进展, 推动了自动化错误定位研究的发展. 然而, 代码通过清晰的函数划分、模块化和注释等, 展现出显著的结构特征, 仅提取语义信息无法进行准确表征, 这从很大程度上影响了错误定位的准确性. 为了克服这一局限, 基于图结构的方法受到广泛关注. 该研究将代码变更集表示为图, 进而实现更精准的代码表示学习. Du 等人[7]提出了语义流图, 使用程序元素之间的数据流和控制流以及程序元素的具体角色以捕捉代码的语义, 并设计了 SemanticCodeBERT, 用于学习深层代码结构的表示. 这些方法引入了程序元素之间的数据依赖和控制依赖关联, 也带来了冗余节点, 无法有效聚焦变更特征, 即代码图结构中的局部变化节点. 而这些局部变化节点中包含了代码的变化信息, 是变更的关键内容, 却极易在规模庞大的代码信息中被忽略[8].

为了定位错误, 现有研究分别从局部和全局角度评估错误报告与变更集的语义相关性. 一部分研究[6,9,10]从局部细节出发, 计算错误报告与变更集局部信息的最大余弦相似度, 进行错误匹配. 另一部分研究[5,7]从全局视角出发, 将错误报告作为整体, 与变更集的整体嵌入计算相似度. 尽管这些方法在各自的视角上取得了进展, 但未能同时综合局部细节和全局信息, 限制了错误定位的准确性.

近年来, 基于图的 Transformer 模型因其学习代码图节点复杂关系的优秀能力而成为研究热点[11−14]. 该模型通过注意力机制捕捉图节点间的长距离依赖关系, 进而有效表征变更代码的整体结构和局部节点特征[15−17]. 基于此, 本文提出了一种基于图 Transformer 的代码变更集错误定位方法 GTCL, 包含 3 个主要模块: (1) 基于变更的语法树构建模块. 首先为每个变更块构建一个基于变更的抽象语法树, 记录代码变更前后的信息并标注变更内容. (2) 图 Transformer 模块, 从全局和局部角度提取和学习变更代码与错误报告的语义信息. (3) 错误定位模块, 依据代码和错误报告相关性进行排序, 完成错误定位任务.

为了评估 GTCL 的有效性, 本文对来自 6 个错误诱发变更集的错误报告和变更进行测试. 实验结果表明: GTCL 能有效地定位错误诱发变更集, 与最先进模型相比, *MAP* 和 *MRR* 指标分别提升 11.4% 和 12.9%. 实验还证明了 GTCL 的主要组件 (全局和局部特征表示模块、相似度注意力计算模块) 以及变更信息 (变更节点标签、变更的抽象语法树) 对定位结果的贡献. 另外, 本文对 GTCL 主要参数 (模型层数、隐藏状态维度和相关性判断的阈值) 的影响进行了讨论.

1 研究背景

1.1 基于变更的抽象语法树

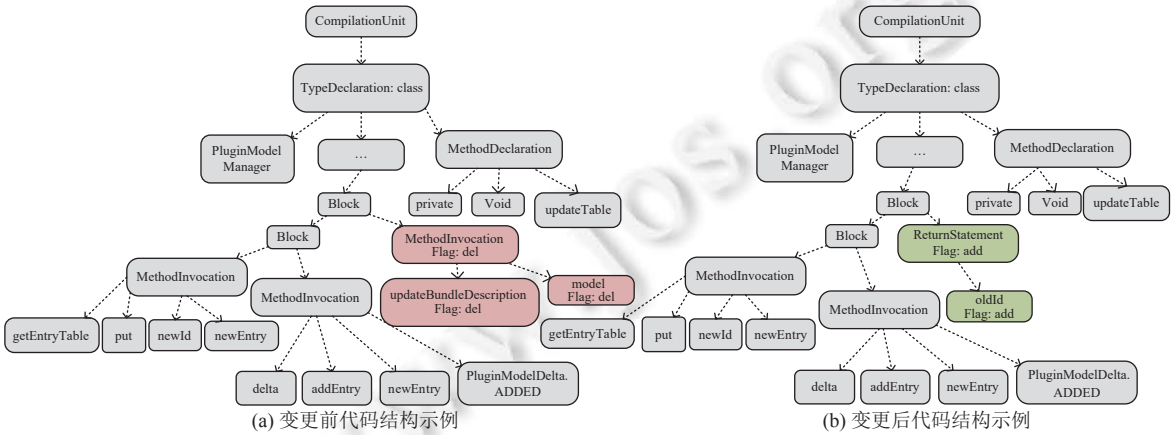
在版本控制系统中, 每一次的变更提交均涵盖了代码修改前后的状态. 具体而言, 更改的代码在行前面标有“+”

或“–”, 用以区分新增与删除的内容, 而未更改的代码保持不变且仅显示一次. 代码变更块包含一组连续的变更行, 同时还包括上下文未发生变化的相关代码行. 值得注意的是, 开发者提交的更改可能涉及多个文件, 每个文件中的修改可以位于一个或多个位置, 每个位置称为一个块或增量. 基于变更提交这种特殊形式, 本文将每个代码块抽象为一个抽象语法树, 旨在捕捉并结构化地展现变更代码块的结构信息, 每个变更的 AST 节点对应于代码的一个语法构造或组成部分 (如操作符、变量、函数调用等). 该语法树能够清晰地描绘代码的语法结构, 其中每个节点对应于代码的一个语法构造或组成部分, 比如操作符、变量、函数调用等. 代码的变更信息通过抽象语法树节点的变化直观体现. 这种节点级别的变化能够准确反映代码修改的具体内容和位置, 从而为代码变更的分析和理解提供了清晰的视角.

如图 1 所示为代码变更的一个示例, 其中包含删除代码行 (以红色高亮显示)、新增代码行 (以绿色高亮显示) 以及未更改代码行. 为了分析这些代码变更, 我们利用基于变更的抽象语法树, 图 2 直观展示了变更前后抽象语法树的对比情况, 图 2(a) 的树表示代码变更前的状态, 其中删除的代码行所对应的节点以红色标出; 图 2(b) 的树则反映变更后的代码结构, 新增代码行的节点以绿色节点标出. 未更改的代码则对应于灰色的节点. 基于变更的抽象语法树突出显示了变化的节点, 提供了代码结构层面的详细变更信息. 同时, 基于变更的抽象语法树, 可以从全局角度获得其整体表示, 而在局部专注其变化的节点. 将局部信息和全局信息结合起来, 我们可以获得更全面的视角, 从而更准确地定位错误. 利用局部信息确定变化点, 局部的变更代码让我们把注意力聚焦到第 7 行的 updateBundleDescription(model) 语句和第 8 行的 return oldId 语句. 结合全局信息, 可以查看包含这个返回语句的类和方法的定义, 了解它们的用途和调用关系.

```
1  public class PluginModelManager {
2      private String handleIdChange(String newId, IPluginModelBase model, ModelEntry newEntry, PluginModelDelta delta) {
3          //...省略部分代码
4          if (newEntry!=null) {
5              getEntryTable().put(newId, newEntry);
6              delta.addEntry(newEntry, PluginModelDelta.ADDED);
7          }
-          updateBundleDescription(model);
8  +       return oldId;
9      }
10
11     private void updateTable(){
12
13     }
14 }
```

图 1　代码变更块实例



(a) 变更前代码结构示例　　(b) 变更后代码结构示例

图 2　变更的抽象语法树

通过比较变更前后的抽象语法树, 我们可以深入分析代码变更的具体影响: 分析被删除节点的功能和作用, 可以帮助判断这些删除是否会引发功能缺失或错误. 分析新增代码节点是否满足需求并实现了预期的功能, 同时不

会引入新的错误.

此外, 先前的工作[5,6]指出, 将整段代码视为一个整体进行信息表征会掩盖更细致的代码变更块之间的关键细节. 这种粗粒度的表示方式增加了从其中提取与变更紧密相关特征的难度. 为了克服这一挑战, 特别是在代码更改涉及多个独立的代码变更块时, 本研究提出了一种新的方法, 即在代码变更块的级别上构建抽象语法树, 保留更详尽的信息, 以便更准确地理解和分析代码在不同变更块中的变化情况. 通过这种方法, 我们能够捕捉更精细的代码变更细节, 为后续的错误信息匹配和定位提供强有力的支持.

1.2 基于图的 Transformer

图 Transformer 的整体架构如图 3 所示. 给定初始的邻接矩阵 A, 节点嵌入 X_v, 图 Transformer 旨在学习图中代码节点的嵌入表示. 对于图 G 的邻接矩阵 $A \in \{0,1\}^{N\times N}$, 表示节点和节点之间是否存在有向边.

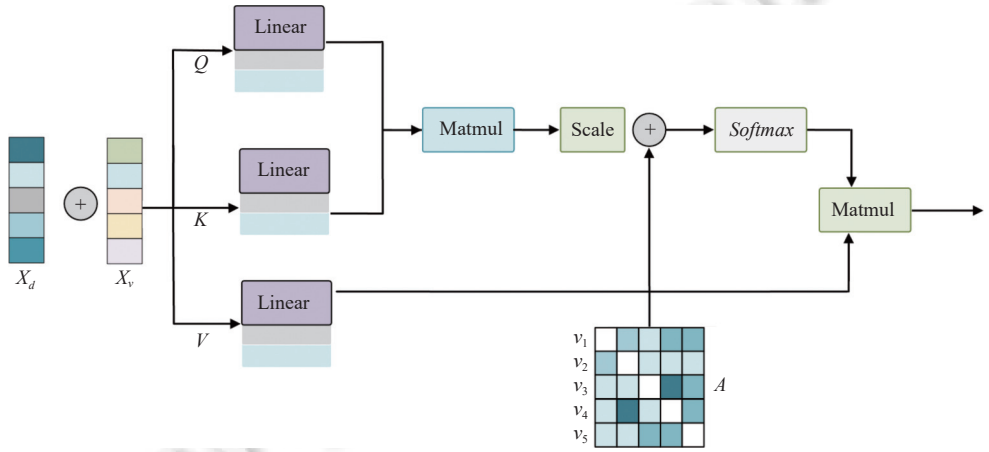


图 3 图 Transformer 的整体架构

本文采用了一种双向信息传播方法, 该方法结合了自底向上和自顶向下的信息传播策略, 旨在全面考虑上下文信息, 从而显著提升节点 token 表示的质量. 将节点嵌入 X_v 与深度编码 X_d 作为图 Transformer 的初始输入. 在有向无环图的结构中, 深度编码包含节点在图中的位置信息, 不仅映射了节点在有向无环图的层级结构位置, 而且为代码更改的图结构分析和处理奠定了基础. 在本文框架中, 节点深度 v 可以通过以下方式定义:

$$depth(v)=\begin{cases}0, & \text{if } v \text{ is a source node}\\ 1+\max_{(u,v)\in E} depth(u), & \text{otherwise}\end{cases} \tag{1}$$

没有前驱节点的所有节点都是源节点, 源节点的深度为 0. 对于其他节点, 其深度由其所有前驱节点的最大深度决定. 具体的, 节点的深度是其所有前驱节点 u 的最大值加 1. 根据公式 (1), 有向无环图的深度编码定义如下:

$$DE_{(v,\,2n)}=\sin\left(\frac{depth(v)}{10000^{\frac{2n}{d}}}\right),\ DE_{(v,\,2n+1)}=\cos\left(\frac{depth(v)}{10000^{\frac{2n}{d}}}\right) \tag{2}$$

其中, d 是节点特征维度, n 是所考虑的维度的索引. 对于有向无环图 G, 定义矩阵 $M\in\mathbb{R}^{|V|\times|V|}$ 来衡量图 G 中节点 v_i 和 v_j 之间的空间关系. 矩阵 M 可以通过节点之间的连接来得到. 在本文中如果两个节点相连, 定义 M_{ij} 是 v_i 和 v_j 之间的最短路径的距离, 如果没有, 将其输出设置为特殊的值, 即−1. 由于图 Transformer 需要图结构作为输入, 而文本本身没有天然的图结构, 因此我们构建了一个虚拟的全连接图, 即每个词节点与其他所有词节点都有连接. 将词嵌入后的节点特征和虚拟图的边信息作为图 Transformer 模型的输入:

$$\begin{cases}attention(Q,K,V)=Softmax\left(\dfrac{QK^{\mathrm{T}}}{\sqrt{d}}+W_m M_{ij}\right)V\\ M_{ij}=\begin{cases}SPD(v_i,v_j), & \text{if } v_i\leftarrow v_j\\ -1, & \text{otherwise}\end{cases}\end{cases} \tag{3}$$

其中, W_m 为 M_{ij} 矩阵的可学习矩阵, 并在所有的层共享. 通过使用空间关系矩阵 M_{ij}, 单个 Transformer 层中的每个

节点可以根据图结构信息自适应地关注所有其他节点. 然后, 将初始化向量输入图 Transformer, 并按照以下的方式进行注意力头 *head* 和隐藏状态 *h* 的学习:

$$\begin{cases} h^{(t)} = FFN(concat(head_1,\ldots,head_t)W^h) \\ head_a = attention(h^{(t-1)}W_a^Q, h^{(t-1)}W_a^K, h^{(t-1)}W_a^V) \end{cases} \tag{4}$$

其中, *FFN* 为前馈神经网络, *a* 为注意力头的数目, W_a^Q、W_a^K、W_a^V 是模型中可学习的权重矩阵. 为简化表述, 公式 (4) 省略了残差连接, 退出策略和归一化策略. 本文使用图 Transformer 对错误报告和代码变更块进行全局和局部的特征表示.

2 基于图 Transformer 的变更集错误定位方法 GTCL

在本节中, 我们详细阐述提出的基于图 Transformer 的代码变更-错误报告定位模型 GTCL. 模型的整体架构如图 4 所示. 总体而言, 本模型分为 3 个模块: (1) 基于变更的语法树构建模块, 为历史提交中的变更块构建基于变更的抽象语法树, 记录代码变更前后的信息并标注变更内容. (2) 图 Transformer 模块, 在这一模块中, 我们从局部和全局两个角度分别提取错误报告与变更集的信息, 并使用图 Transformer 学习其嵌入表示. (3) 错误定位模块, 它分别计算错误报告与变更集之间的局部和全局相似度, 并使用注意力机制调节全局和局部的相似度贡献程度, 用以判断错误报告和变更集的相关性.

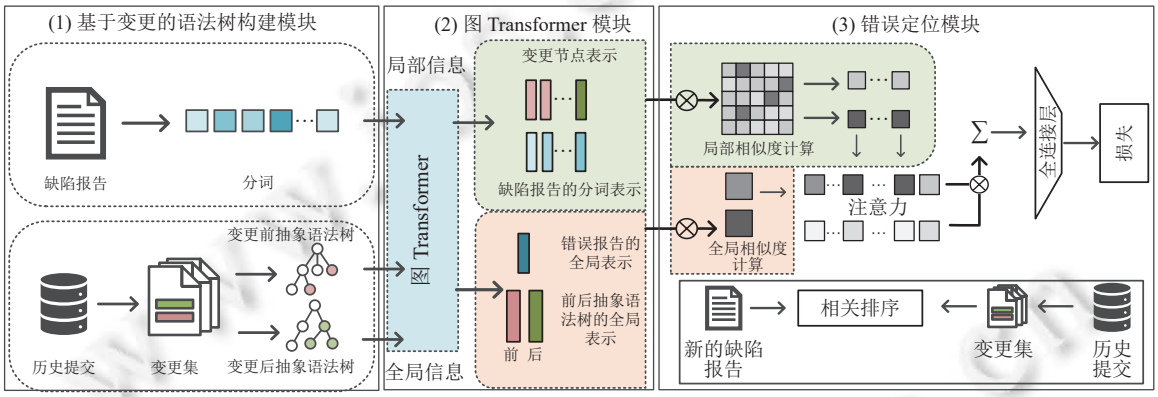


图 4 模型结构图

2.1 全局和局部特征表示

首先, 针对给定代码变更块进行解析并构建其变更前后的抽象语法树. 具体来说, 对于一个代码变更块, 获得其变更代码 ($code_{old}$, $code_{new}$), 然后获得这对变更代码的抽象语法树. 将更改之前的代码表示成 AST_{old}, 更改之后的代码表示成 AST_{new}, 并使用抽象语法树工具 GumTree[18,19]计算更改前后语法树节点的变化以识别两棵树之间的差异. 具体的, 我们比较了单个代码变更块提交之前和之后的 AST 树中的变更节点, 将这些变更节点分为 4 种类型[20–25], 分别如下.

- V_{add}. 如果代码节点 *v* 存在于 AST_{new} 中但不存在于 AST_{old} 中, 则 *v* 是新增的节点, 为其添加编辑标签 add.
- V_{del}. 如果代码节点 *v* 存在于 AST_{old} 中但不存在于 AST_{new} 中, 则 *v* 是删除的节点, 为其添加编辑标签 del.
- V_{move}. 如果代码节点 *v* 同时存在于 AST_{old} 和 AST_{new} 中, 并且其位置和子树的位置发生了变化, 则 *v* 是移动的节点, 在 AST_{new} 中为其和其子树添加编辑标签 move.
- V_{update}. 如果代码节点 *v* 同时存在于 AST_{old} 和 AST_{new} 中, 并且其值发生更新, 则 *v* 是更新的节点, 在 AST_{new} 中为其添加编辑标签 update.

如果代码节点 *v* 同时存在于 AST_{new} 和 AST_{old} 中, 并且其值和位置都保持不变, 为其添加标签 match. 本研究认为全局信息足以表达未更改信息. 为了模型能够理解这两个版本之间的差异, 并学习代码的变更信息, 仅保留变更

语法树的差异类型节点. 通过分析变更语法树的变更节点, 突出显示细粒度代码变更的差异, 使得模型可以学习 AST_{old} 到 AST_{new} 的转换过程, 从而学习 AST_{old} 和 AST_{new} 之间的差异. 通过为原始代码节点添加编辑标签, 将编辑信息融入抽象语法树中, 得到带有编辑信息的抽象语法树, 其本质上是一组带有每个代码变更块基本语义和编辑信息的有向无环图[26].

为了将上述有向无环图作为图 Transformer 的输入, 我们定义有向无环图 $G=(V,E,X)$, 其中节点集为 V, 图中的每个节点 $v_i \in V$ 有 *value* 和 *flag* 两个属性, *value* 属性表示节点 v_i 的语义信息, *flag* 属性标明节点 v_i 的编辑类型信息. E 为有向边集合. $X_v \in \mathbb{R}^{N\times d}$ 表示代码图节点 v 的初始节点特征向量, 特征维数为 $N\times d$, $X_e \in \mathbb{R}^{N\times d}$ 表示节点编辑标签的初始嵌入向量, 特征维数为 $N\times d$. 编辑标签仅在训练局部变更节点特征时使用. 由于深度编码存在于每个节点中, 我们只需将其做点积并将输出添加到节点特征中. 定义节点 v 的初始化向量表示为:

$$h_v^{(0)} = [h_v^x | h_v^d | h_v^e] \tag{5}$$

其中, 上标为 x、d、e 的 h 向量分别表示节点 v 的初始 token 嵌入、节点的深度嵌入、节点的编辑类型嵌入. $h_v^{(0)}$ 由 3 个嵌入拼接获得. 将初始嵌入输入图 Transformer 模型中, 经过训练, 得到局部变更节点的特征向量表示:

$$h_v = GF(h_v^{(0)}) \tag{6}$$

同时, 对于图表示任务, *READOUT* 函数旨在将最终迭代的节点特征聚合到整个图 G 的表示中, 获得变更代码块的整体嵌入 H_{old} 和 H_{new}:

$$H_{\text{old}}, H_{\text{new}} = READOUT\left(\left\{h_i^{(L)} | v_i \in G\right\}\right) \tag{7}$$

READOUT 函数为平均池化操作. 其次, 错误报告由自然语言描述组成, 在处理给定的错误报告时, 将问题的描述分解为 L 个单词. 这些单词随后被输入到图 Transformer 网络中. 可以将第 t 个单词的特征向量 q_t 表示为:

$$q_t = GF(X_t) \tag{8}$$

同样的, 使用 *READOUT* 函数获得错误报告的整体嵌入 Q_g.

2.2 相似度计算

在得到变更和错误报告的整体和局部特征表示后, 本文采用多维度向量相似度方法, 通过在多个维度上描述和比较对象特性, 能够更全面地捕捉代码与文本间的复杂关系. 我们定义两个向量 $x \in \mathbb{R}^d$ 和 $y \in \mathbb{R}^d$, 它们之间的相似度函数定义为:

$$f_{\text{sim}}(x,y,W) = \frac{W|x-y|^2}{\|W|x-y|^2\|_2} \tag{9}$$

其中, $|\cdot|^2$ 表示元素级方差, $\|\cdot\|_2$ 表示 $L2$ 正则化手段, $W \in \mathbb{R}^{m\times d}$ 是 m 维相似向量的可学习参数矩阵.

2.2.1 全局相似度表示

在全局相似度表示中, 使用全局特征 H_{old}, H_{new} 和 Q_g 依据公式 (8) 相似性计算函数计算代码片段和错误报告的全局相似度表示:

$$\begin{cases} S_{\text{old}}^g = f_{\text{sim}}(H_{\text{old}}, Q_g, W_{\text{old}}^g) \\ S_{\text{new}}^g = f_{\text{sim}}(H_{\text{new}}, Q_g, W_{\text{new}}^g) \end{cases} \tag{10}$$

其中, $W_{\text{old}}^g \in \mathbb{R}^{m\times d}$、$W_{\text{new}}^g \in \mathbb{R}^{m\times d}$ 旨在学习全局相似性表示.

2.2.2 局部相似度表示

在局部相似度表示中, 本文关注匹配的细节. 具体而言, 精准匹配的内容对于定位代码中的错误部分具有极高的价值, 因为这些匹配部分往往直接关联代码中的潜在问题. 相反, 不匹配的内容则可能意味着该部分的代码变更与代码错误之间的关联性较弱, 甚至完全无关. 值得注意的是, 我们并不主张简单地弱化或忽略这些不匹配的更改部分, 而是应该正视其可能存在的负面作用[27]. 具体而言, 首先计算出所有代码节点和错误词语的语义相关性分数矩阵 $S_{i,t}$:

$$S_{i,t} = \frac{h_i^{\text{T}} q_t}{|h_i||q_t^{\text{T}}|},\ i \in [1,n], t \in [1,L] \tag{11}$$

其中, h_i 为第 i 个变更节点向量, q_t 为错误报告中第 t 个词向量, n 为变更节点数, L 为词序列长度. 得到的相关性分数矩阵后, 进一步利用这一矩阵来计算代码节点与错误词语之间的正负相关性分数, 有相关和不相关两个分支. 在不相关分支中, 我们的目标是准确有效地利用不相关的更改, 使它们降低代码更改-错误报告文本对的整体相似度. 在文本模态中, 一个词语与代码节点的最大跨模态相似性反映了它相关或者不相关的程度. 因此, 计算每个错误报告词语 q_t 和所有代码更改抽象语法树节点之间的最大池化相似度:

$$K_t^{\text{neg}} = \max\left(\left\{K - S_{i,t}^{(c)}\right\}_{i=1}^{n}\right) \tag{12}$$

其中, K 为边界超参数. 代码更改-错误报告文本对中, 第 t 个词语和代码节点的负相关性可以通过以下方式计算:

$$S_t^{\text{neg}} = S_{:,t}^{(c)} \odot Mask_{\text{neg}}(S_{i,t}^{(c)}) \tag{13}$$

其中, $Mask_{\text{neg}}(\cdot)$ 是一个掩码, 当输入为负时, 等于 1, 否则为 0; $\odot$ 表示点乘; K 为相似性边界, 设置为超参数. 根据 S_t^{neg} 我们计算出代码变更节点对 q_t 的相似度表示:

$$S_{\text{sim}}^{\text{neg},t} = f_{\text{sim}}!(h_i^{\text{neg}}, q_t, W_t),\ t \in [1, L] \tag{14}$$

其中, h_i 为 $S_{\text{sim}}^{\text{neg},t}$ 所对应的变更节点的表示, 即 K_t^{neg} 对应的最负相关的 s 节点向量. $W_t \in \mathbb{R}^{m\times d}$ 旨在学习局部相似表示, L 代表文本的序列化标记后的长度.

相关分支旨在衡量代码更改-错误报告配对的相似程度. 本文专注于关注跨模态的共享语义, 我们首先获取查询词; 然后, 根据查询词聚合对应的代码变更节点; 最后, 基于融合结果衡量相关更改片段的相似程度. 具体而言, 通过公式 (15) 计算跨模态注意力权重:

$$W_{i,t}^{\text{inter}} = Softmax_{\lambda}\left(Mask_{\text{pos}}\left(\{S_{i,t} - k\}_{t=1}^{n}\right)\right) \tag{15}$$

其中, $W_{i,t}^{\text{inter}}$ 表示第 t 个单词 q_t 和第 i 个代码更改节点之间的语义关联. $Mask_{\text{pos}}(\cdot)$ 表示一个掩码, 当输入为正时等于输入值, 否则为负无穷. 在这种情况下, 不相关的代码更改节点权重, 即 $\{S_{i,t} - k\} < 0$ 被清零. 对于第 i 个单词, 代码更改节点中与之对应的共享语义可以聚合为:

$$\widehat{h_t} = \sum\nolimits_{i=1}^{n} W_{i,t}^{\text{inter}} h_i \tag{16}$$

基于这个加权的代码更改节点特征, 单词 q_t 的相似度可以表示为:

$$S_t^{\text{pos}} = f_{\text{sim}}\left(\widehat{h_t}, q_t, W_t\right),\ t \in [1, L] \tag{17}$$

最后, 代码更改节点-错误报告 (v_i, q_t) 的局部相似度可以通过不相关更改和相关更改共同表示, 得到的相似度表示将作为代码变更块和错误报告之间局部相似度的量化指标.

2.2.3 得分计算

根据全局和局部的相似性表示 $S = (S_{\text{old}}^{g}, S_{\text{new}}^{g}, S_t^{\text{neg}}, S_t^{\text{pos}})$ 来计算每个表示的聚集权重. 我们不采用定值权重给全局和局部信息, 使用定值权重无法根据具体的代码更改进行灵活调整, 会导致在某些更改下过度依赖局部信息而忽略了全局上下文, 或者在另一些更改下过度关注全局信息而忽略了局部的关键细节, 从而影响错误定位的准确性. 通过批量归一化 BN 处理权重, 再经线性变换以求得规范化的权重 β_p. 随后, 使用这些权重聚合相似性表示, 并将结果输入全连接层, 预测代码和错误报告的最终相似度为:

$$\beta_p = \frac{\delta(BN(W_f S))}{\sum_{z\in N^{\delta}} BN(W_f S)} \tag{18}$$

其中, δ 是 Sigmoid 函数, $W_f \in \mathbb{R}^{m\times 1}$ 表示线性变换.

$$score = FFN\left(\sum\nolimits_{z\in N^{\delta}} \beta_p \cdot S\right) \tag{19}$$

在所有步骤完成后, 将得到的表示向量输入前馈神经网络计算最终得分, 这一得分将作为代码与问题描述之间相似度的量化指标, 为后续的匹配、排序等任务提供数据支撑和决策依据.

2.2.4 训 练

在本文中, 我们采用双向排序损失函数作为训练的目标函数, 该函数要求匹配的代码变更-错误报告对的相似

度要高于不匹配对的相似度, 并且有一个固定的边际, 强制模型在预测正负样本时保持一定的间隔, 从而在区分正负样本时更为准确.

给定匹配的代码变更-错误报告对 (c, p), 以及 mini-batch 内对应的最难不匹配代码 c' 和最难负文本 q', 我们计算双向排名损失如下:

$$L = \min \sum_{(c,q)} [r - f_{\text{sim}}(c,q,W) + f_{\text{sim}}(c',q,W)] + [r - f_{\text{sim}}(c,q,W) + f_{\text{sim}}(c,q',W)] + \lambda|\theta|^2 \tag{20}$$

其中, 最难不匹配代码 c' 和最难负文本 q' 通过公式 (18) 选取. c' 表示与错误报告相似度得分最高, 但与错误报告无关的变更代码, q' 表示与变更代码相似度得分最高的无关错误报告文本. r 是一个边界参数, $f_{\text{sim}}(\cdot,\cdot,\cdot)$ 表示用相似度计算模块实现的相似性预测函数.

3 实 验

本节介绍实验设置, 包括模型所用到的数据集, 评估指标, 基线方法和超参数配置. 我们在 6 个软件更改项目数据集上进行了广泛实验, 以回答以下研究问题.

RQ1: 相较于其他方法, GTCL 的性能如何? 本文与 7 种不同类型的基线方法进行对比, 比较了在 6 个变更数据集上的错误定位效果, 以验证方法性能.

RQ2: GTCL 能否捕获代码的变化信息? 为此设计了两项实验: 一是删除编辑节点标签以评估其对模型性能的影响; 二是仅使用单一语法树, 不考虑变化信息, 评估变化信息对模型性能的影响.

RQ3: GTCL 各子模块对本方法总体性能的贡献如何? 本文分别对变更语法树、全局和局部表示进行了消融实验, 研究变更语法树构建和图 Transformer 两个模块对本方法总体性能的影响.

RQ4: GTCL 的主要参数 (模型层数 L, 隐藏状态维度 d 和相关性判断阈值 k) 会对定位结果产生怎样的影响?

3.1 实验数据

为了评估本文错误定位技术的有效性, 采用了 Ciborowska 等人[6]构建的错误数据集. 该数据集基于 Wen 等人[5]所提供数据集进行分离和手动验证. 包括 AspectJ、JDT、PDE、SWT、Tomcat 和 ZXing 这 6 个软件项目, 其中的错误报告数目和变更块数目如表 1 所示, 每个项目的数据集都包括版本更新记录和错误报告, 这些信息对于分析代码中的错误和验证错误定位技术的有效性至关重要.

表 1 实验数据集

数据集	错误报告	更改次数	代码变更块	变更文件	变更集
AspectJ	200	5539	23446	14030	2939
JDT	94	16582	150630	58619	13860
PDE	60	10834	100373	42303	9419
SWT	90	17688	69833	25666	10206
Tomcat	193	15381	72134	30866	10034
ZXing	20	853	6165	2846	843

为了对每个项目进行模型训练和评估, 我们首先根据收集到的错误报告和诱发变更集按照时间顺序排列, 在确保训练集早于验证集和测试集的前提下, 随机选择 60% 的数据用作训练集, 20% 用于验证集, 剩余的 20% 作为测试集. 在划分过程中, 确保训练集和测试集之间不存在重复的错误报告和诱发变更集. 在负样本的生成上, 通过选择不属于诱发变更集的代码更改来构建负样本, 形成包含错误报告、诱发变更集和非诱发变更集的三元组, 然而, 由于生成负样本的方法计算成本较高, 并且通过选择句法相似且不诱导错误的变更集未能显著提升检索准确性[6], 本文最终决定采用随机采样策略来生成负样本, 并选择与正样本数量相同的负样本数量, 以保障数据集的平衡性. 需要指出的是, 尽管训练集并未涵盖所有可用的代码变更, 但在错误定位过程中, 模型会对特定项目中所有可用的代码变更执行检索.

3.2 评价指标及参数设置

3.2.1 评价指标

本文旨在从代码变更集合中检索与错误报告相关的代码片段, 追求算法的高精确率. 为了评估模型的性能, 我们采用了一组常用的用于评估信息检索系统性能的指标.

Top@N: 用于衡量软件开发中错误检测工具的有效性. 它反映了通过检查可疑代码实体列表中的前 N 项, 能够发现缺陷实体的缺陷比例. 该指标越高, 开发人员定位缺陷所需的工作量就越少, 性能也就越好.

Precision@K ($P@K$): 用于评估排名中排名靠前的变更集与错误报告相关的数量. $P@K$ 的值等于排名中位于前 K 位置的相关变更集数量 $\left|Rel_{B_i}\right|$, 在 B 个错误报告中取平均值:

$$P@K = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{\left|Rel_{B_i}\right|}{K} \tag{21}$$

平均倒数排名 (MRR): 量化了模型将第 1 个相关变更集定位到错误报告的能力. 该指标通过计算在 B 个错误报告中的倒数排名的平均值得出, 其中错误报告 B_i 的倒数排名等于排名中第 1 个相关变更集的倒数排名:

$$MRR = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{rank_{B_i}^{(1)}} \tag{22}$$

其中, $rank_{B_i}^{(1)}$ 表示在第 i 个错误报告中, 第 1 个相关变更集在返回结果中的排名.

平均精确率 (MAP): 衡量模型定位与错误报告相关的所有变更集的能力, MAP 被计算为 B 个错误报告的平均精度值 ($AvgP$) 的平均值, 而错误报告 B_i 的平均精度值 $AvgP_{B_i}$ 是基于排名中所有相关变更集的位置来计算的:

$$\begin{aligned} AvgP &= \sum_{j=1}^{M}\frac{P@j\times pos(j)}{N} \\ MAP &= \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{AvgP_{B_i}} \end{aligned} \tag{23}$$

其中, j 是排名, M 是检索到的变更集数量, $pos(j)$ 表示第 j 个变更集是否与该错误报告相关, N 是与变更集相关的错误报告总数, $P@j$ 是在检索排名中的前 j 个位置的精确率, B_i 是第 i 个错误报告.

基于上述评价指标, 我们使用 Wilcoxon 符号秩检验来验证两个模型在预测性能上是否有显著差异, 当 p 值小于 0.05 时, 就认为模型间的差异是不可忽视的.

3.2.2 实验设置

实验在一台配备 12 核 3.2 GHz Intel 处理器的服务器上进行, 使用了一块 80 GB 显存的 NVIDIA A100 显卡, 并运行 CUDA 11.7. 模型实现采用了 PyTorch v2.0.1. 由于预训练任务计算资源需求高且数据集规模庞大, 我们选择了现成的 Graphformer 预训练模型进行实验. 具体设置方面, 我们将错误报告最大嵌入长度限制设定为 256, 语法树节点数量设置最大为 50, 使用 12 层 Graphformer, 同时将代码和文本的隐藏状态维度设置为 512, 批次大小设置为 256, 多头自注意力机制中的头数设为 2. 相关性判断的阈值 k 被设置为 0.35, 边缘参数 γ 设为 0.2. 在训练过程中, 使用 Adam 作为优化器, 并将超参数设置为 1E–8, 我们使用了 0.4 的 dropout 层以防止过拟合. Epoch 设置为 20 轮, 当 MRR 在 10 个 epoch 内没有改善时, 停止训练过程.

3.2.3 对比方法

综合已有文献研究和相关开源代码的现状, 我们选取了错误检测性能最优的 3 个基于深度学习的方法和 4 个基于错误变更集的错误检测方法, 在所选数据集上进行了充分的实验与分析.

对于利用深度神经网络特征进行错误定位的方法, 许多相关研究 (如 Np-CNN[28]、LS-CNN[29]、DeepLoc[30] 等) 未公开其代码, 使得其复现过程具有一定的挑战性[31]. 为确保已有工作复现的准确性, 我们对现有的开源工作进行了全面的调研和筛选, 并从中选择了 3 种具有良好复现结果的优秀定位方法, 作为基于深度模型特征的错误定位基线方法. 这一选择过程旨在提高实验结果的可靠性和可比性.

BLIA[32]: 整合了之前提出的错误相关特征, 通过组合这些特征的方式提高基于信息检索的错误定位性能.

DNNLOC[33]: 利用深度神经网络来进行错误定位, 结合了卷积神经网络 (CNN) 和长短期记忆网络 (LSTM) 来处理代码和错误报告的序列数据, 从而提取高层次的特征表示. 该模型在处理代码的上下文信息和长范围依赖关系方面表现优异, 能够有效地识别和定位代码中的错误.

DreamLoc[34]: 采用 wide and deep 模型架构来提升错误特征的学习效果. 综合考虑了错误报告和代码的信息, 通过引入注意力机制和门控机制, 旨在提高错误报告与代码文件之间的关联匹配精度.

Locus[5]: 利用代码块级别的粒度和向量空间模型, 根据在错误报告、代码块和日志消息之间获得的最大相似度得分来定位相关的变更集.

Bug2Commit[8]: 通过构建基于图的神经网络来对代码提交与错误报告进行匹配, 它将错误报告和代码提交映射到一个共享的嵌入空间中, 并利用图神经网络 (GNN) 来捕捉错误报告与代码提交之间的复杂关系.

FBL-BERT[6]: 使用 BERT 作为预训练模型, 将变化的标签加入训练中, 通过对错误报告和代码变更集的文本进行词嵌入, 将它们映射到 BERT 生成的上下文敏感的嵌入空间中. 通过对最大余弦相似度的求和来综合评分, 从而提高错误定位的精度. 该方法强调了使用 BERT 的上下文信息来增强对代码和错误报告之间语义关联的捕捉能力, 从而有效提升错误定位的准确性.

SemanticCodeBERT[7]: 该模型使用语义流图 (SFG) 来捕捉代码语义, 基于 SFG 设计和训练改进的 CodeBERT, 并设计了一种新颖的分层动量对比错误定位技术.

我们在相同的错误定位场景下, 对比了以上基准方法与本文提出的 GTCL 方法在所设计研究问题下的实验结果和相关指标, 以验证 GTCL 方法的有效性. GTCL 方法不依赖于日志消息等其他信息检索 (IR) 特征, 因为我们旨在探索从错误报告中的自然语言到代码更改之间的映射关系. 尽管高质量的日志消息可以通过提高某些变更集的评分对结果产生积极影响, 但并非所有相关的代码更改都伴随有高质量的日志消息.

3.3 实验结果与分析

RQ1: 相较于其他方法, GTCL 的性能如何?

为了验证这个问题, 我们将 GTCL 模型的定位性能与其他 7 种不同类型的基线方法在 6 个变更数据集上进行对比, 具体结果详见表 2.

表 2 展示了 GTCL 与其他对比方法在 6 组数据集上的定位性能指标, 其中每个指标的粗体值表示所有方法中最佳的结果. 在最优指标方面, 相较于次优基线方法 SemanticCodeBERT, GTCL 的 *MAP* 平均提升 5.8%, *MRR* 平均提升 7.2%; 在 *P*@1 指标上, 有着最为显著的提升, 达到 12.2%. 特别是在 SWT 数据集中, *P*@1 的值达到了 0.478, 这意味着在仅定位一个错误诱发变更集以应对错误报告时, GTCL 的精确率接近 50%. 在 Tomcat 数据集上, GTCL 的 Top@5 达到 0.907, Top@10 达到 0.954, 该结果证明了本模型定位错误变更集性能的先进性. 根据 Wilcoxon 符号秩检验的结果 GTCL 与其他模型在预测性能上具有显著差异.

进一步分析, 3 种传统深度模型在变更数据集上的表现较差, 主要由于其泛化能力有限, 导致其性能指标均低于定位错误诱发变更集的 3 种基线模型. 其中, BLIA 和 DNNLOC 由于仅依赖序列信息而未考虑代码的结构特征, 导致错误报告与错误代码的语义相似度较低, 从而使其性能指标最低. 而 DreamLoc 尽管属于传统深度模型, 其宽度与深度架构以及代码切片技术在提升错误特征学习效果方面有所改进, 相较于前两种方法有所提升. 除早期的 Locus 技术外, 基于错误诱发变更集的基线模型在定位性能上皆优于 3 种传统深度模型, 其中, SemanticCodeBERT 在与 Bug2Commit 和 FBL-BERT 两类模型的比较中表现出更高的性能. 这一优越性源于 SemanticCodeBERT 利用语义流图来捕捉代码的语义特征, 相较于仅使用代码的序列信息, 结构信息提供了更为丰富和深刻的上下文理解, 进而更有效地捕捉代码的语法和语义特征, 从而提升错误报告与变更块匹配的准确性. 此外, GTCL 在定位结果上展现出比基线模型更有效且稳定的性能, 其在所有 6 个数据集上的各项指标均优于基准模型. 与 SemanticCodeBERT 相比, 基于变更的抽象语法树在捕捉变更块的变化方面表现更为出色. 相较于语义流图, 基于变更的抽象语法树提供了更为精确的全局与局部视角, 从而更敏锐地捕捉了图节点的变更信息, 并有效减少了无关冗余节点的干扰.

表 2 GTCL 与跨项目和项目间错误定位方法的性能比较

数据集	基线模型	*P*@1	*P*@3	*P*@5	Top@1	Top@5	Top@10	*MAP*	*MRR*
AspectJ	BLIA	0.267	0.199	0.167	0.267	0.542	0.621	0.110	0.315
	DNNLOC	0.258	0.210	0.171	0.258	0.556	0.627	0.121	0.314
	DreamLoc	0.268	0.212	0.169	0.268	0.572	0.654	0.144	0.319
	Locus	0.260	0.221	0.154	0.260	0.566	0.639	0.121	0.318
	Bug2Commit	0.284	0.237	0.205	0.284	0.574	0.669	0.155	0.324
	FBL-BERT	0.297	0.255	0.211	0.297	0.581	0.674	0.148	0.338
	SemanticCodeBERT	0.356	0.268	0.224	0.356	0.601	0.720	0.164	0.380
	GTCL	**0.407**	**0.301**	**0.249**	**0.407**	**0.638**	**0.766**	**0.173**	**0.424**
JDT	BLIA	0.224	0.195	0.104	0.224	0.455	0.532	0.014	0.325
	DNNLOC	0.226	0.191	0.106	0.226	0.459	0.537	0.016	0.339
	DreamLoc	0.254	0.200	0.118	0.254	0.515	0.592	0.046	0.384
	Locus	0.259	0.199	0.115	0.259	0.513	0.552	0.044	0.356
	Bug2Commit	0.277	0.212	0.125	0.277	0.563	0.645	0.064	0.424
	FBL-BERT	0.289	0.228	0.139	0.289	0.587	0.673	0.082	0.479
	SemanticCodeBERT	0.311	0.241	0.158	0.311	0.631	0.723	0.127	0.539
	GTCL	**0.394**	**0.269**	**0.175**	**0.394**	**0.799**	**0.918**	**0.146**	**0.587**
SWT	BLIA	0.171	0.121	0.111	0.171	0.348	0.400	0.051	0.411
	DNNLOC	0.244	0.142	0.144	0.244	0.496	0.570	0.064	0.424
	DreamLoc	0.299	0.151	0.141	0.299	0.608	0.698	0.054	0.438
	Locus	0.245	0.141	0.138	0.245	0.498	0.571	0.052	0.421
	Bug2Commit	0.330	0.172	0.166	0.330	0.576	0.662	0.069	0.429
	FBL-BERT	0.390	0.194	0.175	0.390	0.603	0.691	0.098	0.443
	SemanticCodeBERT	0.452	0.205	0.181	0.452	0.722	0.826	0.149	0.558
	GTCL	**0.478**	**0.233**	**0.190**	**0.478**	**0.824**	**0.948**	**0.151**	**0.607**
PDE	BLIA	0.223	0.112	0.061	0.223	0.453	0.520	0.064	0.201
	DNNLOC	0.243	0.139	0.062	0.243	0.495	0.568	0.072	0.214
	DreamLoc	0.259	0.143	0.131	0.259	0.526	0.601	0.062	0.228
	Locus	0.245	0.121	0.108	0.245	0.498	0.671	0.061	0.218
	Bug2Commit	0.264	0.168	0.145	0.264	0.536	0.613	0.069	0.229
	FBL-BERT	0.267	0.175	0.151	0.267	0.542	0.621	0.098	0.243
	SemanticCodeBERT	0.274	0.181	0.156	0.274	0.557	0.638	0.114	0.298
	GTCL	**0.307**	**0.211**	**0.179**	**0.307**	**0.624**	**0.715**	**0.121**	**0.327**
ZXing	BLIA	0.204	0.182	0.101	0.204	0.445	0.532	0.144	0.203
	DNNLOC	0.234	0.212	0.112	0.234	0.477	0.548	0.154	0.210
	DreamLoc	0.259	0.253	0.121	0.259	0.512	0.607	0.162	0.328
	Locus	0.245	0.242	0.118	0.245	0.498	0.571	0.154	0.219
	Bug2Commit	0.267	0.268	0.125	0.267	0.535	0.622	0.189	0.371
	FBL-BERT	0.297	0.275	0.141	0.297	0.603	0.691	0.198	0.374
	SemanticCodeBERT	0.351	0.284	0.150	0.351	0.672	0.770	0.214	0.398
	GTCL	**0.407**	**0.291**	**0.169**	**0.407**	**0.705**	**0.805**	**0.221**	**0.407**
Tomcat	BLIA	0.213	0.152	0.102	0.213	0.432	0.496	0.092	0.298
	DNNLOC	0.242	0.162	0.110	0.242	0.491	0.564	0.095	0.301
	DreamLoc	0.269	0.183	0.122	0.269	0.547	0.628	0.099	0.325
	Locus	0.325	0.202	0.111	0.325	0.661	0.757	0.094	0.307
	Bug2Commit	0.374	0.218	0.119	0.374	0.753	0.872	0.102	0.391
	FBL-BERT	0.397	0.225	0.133	0.397	0.804	0.924	0.135	0.399
	SemanticCodeBERT	0.431	0.264	0.140	0.431	0.874	0.932	0.142	0.476
	GTCL	**0.447**	**0.293**	**0.158**	**0.447**	**0.907**	**0.954**	**0.151**	**0.487**

注: 本文将GTCL与7种基线方法分别进行成对t检验, p值均小于等于0.01

值得注意的是, SemanticCodeBERT 的结果与其原文 (文献 [7]) 中同项目上的结果有较大差异, 在此给出分析: 本文实验中对数据集进行预处理, 包括删除了更改代码中的注释以及日志信息. 我们查看了 SemanticCodeBERT 的数据集, 其数据集中保留了代码中的注释和日志信息, 这些内容虽然对于代码的实际运行没有直接作用, 但可能包含了一些有助于模型理解代码语义的额外信息. 删除这些信息后, 模型失去了这部分辅助理解的线索, 使得模型在学习代码语义时出现偏差, 进而影响最终的结果. 为了保证比较结果的公平性, 使用处理的数据集在本文实验设备上对 SemanticCodeBERT 进行了重新训练. 在实验中发现, 在不同设备上进行实验, 硬件性能的差异导致模型收敛的速度和程度不同, 本实验使用的是一台配备 12 核 3.2 GHz Intel 处理器的服务器, 使用了一块 80 GB 显存的 NVIDIA A100 显卡, 并运行 CUDA 11.7. 而文献 [7] 是在 Ubuntu 系统上使用 128 GB RAM 的 NVIDIA Tesla A100 进行预训练. 本文实验设备与文献 [7] 使用的设备可能不同, 这会影响模型训练的速度和稳定性. 在训练过程中, 硬件性能的差异导致模型收敛的速度和程度不同, 从而使得模型学到的参数存在差异, 最终影响实验结果.

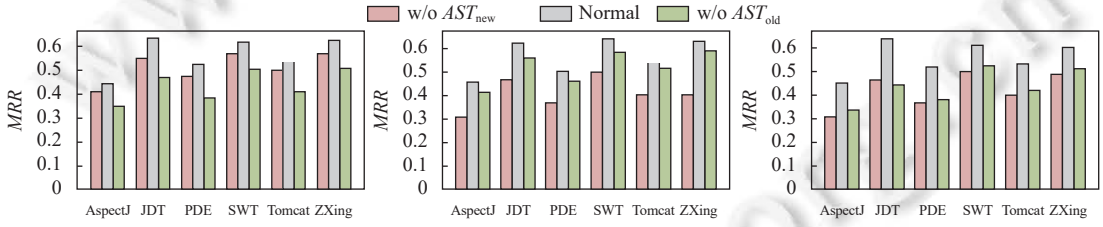
综上所述, GTCL 模型在软件错误定位任务中展现出卓越的性能. 基于变更的抽象语法树在捕捉变更块变化方面的优势, 以及全局与局部视角的结合, 使得 GTCL 在多个关键性能指标上均优于其他对比方法. 这一成果不仅为软件错误定位任务提供了新的思路和方法, 也为软件维护和质量保障领域的研究和应用提供了新的可能性.

RQ2: GTCL 能否捕获代码的变化信息?

为了回答这个问题, 我们设计了两个实验: 一是删除编辑节点标签以考察其对模型性能的影响, 二是仅使用单一变更语法树来分析其对模型效果的作用.

(1) 前后变更语法树与变更类型的相关性分析

为了深入探讨前后变更语法树与变更类型之间的关系, 我们进行了一系列关键实验. 首先, 将实验数据集按照变更类型划分为 3 个子集: 仅包含删除操作的 Delete 数据集, 仅包含新增操作的 Add 数据集, 以及其他操作的数据集 Other. 接着, 对 GTCL 模型进行消融实验, 分别在变更数据集的旧语法树和新语法树的基础上进行了实验, 具体步骤包括: 替换旧语法树, 并将其复制成两棵新语法树; 替换新语法树, 并将其复制成两棵旧语法树. 实验结果如图 5 所示.



(a) Delete 数据集的 *MRR* 指标 (b) Add 数据集的 *MRR* 指标 (c) Other 数据集的 *MRR* 指标

图 5 替换新旧抽象语法树对 *MRR* 指标的影响分析

在 3 个数据集上, 替换新旧语法树的操作均导致了模型 *MRR* 指标的下降. 这一现象可以归因于替换操作导致了代码变更部分信息的丢失. 在 Delete 数据集上, 替换语法树造成了性能的显著下降. 进一步分析表明, 替换旧语法树对结果的影响最为显著, *MRR* 指标较基本模型降低了 28.8%. 这一情况可能与旧语法树中主要包含删除节点信息有关. 相对而言, 在 Add 数据集上, 替换新语法树对模型性能的影响最为显著, *MRR* 指标较基本模型降低了 33.7%. 这一结果可能与新语法树中主要包含新增节点信息有关. 在 Other 数据集上, 虽然替换新旧语法树的操作没有显著差异, 但均导致了性能的下降, *MRR* 指标较基本模型降低了 30.3%. 综上所述, 保留新旧语法树对于模型捕捉代码变更信息至关重要, 因为新旧语法树反映了代码的前后变化差异, 包括具体的删除、增加和更新操作, 这对提升模型性能具有重要意义, 证明前后变更语法树体现了代码块的变化信息.

(2) 变更节点标签对模型的性能有何影响

除了对变更语法树的结构分析, 我们还进一步研究了变更节点标签对模型性能的影响. 如图 6 所示, 在图

Transformer 模型中, 系统地探讨了保留变更节点标签信息 (如图 6 中 Change 部分) 和去除变更节点 (如图 6 中 w/o Change 部分) 对模型在精确率 *P@K* 评估指标下的影响. 实验结果显示保留标签信息的模型在精确率 *P@K* 方面表现显著优于去除标签信息的模型. 保留标签信息较去除标签信息在 *P*@1、*P*@3 和 *P*@5 指标上分别提升了 15.8%、18.0% 和 22.4%. 对这一结果的解释是变更节点的标签明确反映了旧版本和新版本代码之间的差异, 这些标签提供了关键的变更信息. 在 *P@K* 指标上, JDT 数据集中保留标签信息的效果提升最为显著, 达到了 21.7%. 这一提升主要归因于 JDT 数据集中与错误报告相关的变更块中包含更多变更节点, 以及这些变更节点所含有的丰富变更标签信息. 通过保留这些信息, GTCL 能够有效地捕捉和利用变更节点间的语义关系, 从而提高模型的精确性和鲁棒性.

(a) *P*@1 (b) *P*@3 (c) *P*@5

图 6 有无变更节点的标签信息在 *P*@1、*P*@3 和 *P*@5 上的性能比较

综上所述, 变更节点标签对于图 Transformer 模型在软件错误定位任务中的性能提升至关重要. 通过保留和利用这些信息, 模型能够更有效地捕捉和利用代码变更与错误报告之间的语义关系, 从而提高定位的精确性和鲁棒性. 这一发现不仅为 GTCL 模型的设计提供了有力支持, 也为未来的软件错误定位研究提供了新的思路和方法.

RQ3: GTCL 各子模块对本方法总体性能的贡献如何?

本研究进一步分析了模型各组成部分对整体性能的影响, 通过进行消融实验来评估变更语法树的使用、全局相似度表示以及局部相似度分别对模型性能的贡献. 具体而言, 表 3 详细对比了未使用变更语法树直接进行相似度计算的情形与仅依赖全局相似度表示或仅依赖局部相似度表示的性能.

表 3 消融实验分析

数据集	消融实验	*P*@1	*P*@3	*P*@5	Top@1	Top@5	Top@10	*MAP*	*MRR*
AspectJ	w/o AST_{change}	0.167	0.121	0.114	0.167	0.214	0.255	0.087	0.159
	w/o Global	0.251	0.165	0.147	0.251	0.444	0.536	0.121	0.294
	w/o Local	0.311	0.212	0.176	0.311	0.541	0.614	0.144	0.372
	GTCL	**0.407**	**0.301**	**0.249**	**0.407**	**0.638**	**0.766**	**0.173**	**0.424**
JDT	w/o AST_{change}	0.154	0.116	0.101	0.154	0.212	0.251	0.075	0.219
	w/o Global	0.251	0.205	0.112	0.251	0.437	0.523	0.103	0.351
	w/o Local	0.305	0.284	0.144	0.305	0.534	0.655	0.121	0.541
	GTCL	**0.394**	**0.269**	**0.175**	**0.394**	**0.799**	**0.918**	**0.146**	**0.587**
SWT	w/o AST_{change}	0.254	0.144	0.105	0.254	0.439	0.517	0.078	0.208
	w/o Global	0.330	0.198	0.124	0.330	0.571	0.674	0.112	0.413
	w/o Local	0.394	0.214	0.155	0.394	0.681	0.784	0.125	0.526
	GTCL	**0.478**	**0.233**	**0.190**	**0.478**	**0.824**	**0.948**	**0.151**	**0.607**
PDE	w/o AST_{change}	0.157	0.108	0.085	0.157	0.282	0.318	0.069	0.164
	w/o Global	0.214	0.147	0.122	0.214	0.370	0.433	0.086	0.225
	w/o Local	0.254	0.175	0.146	0.254	0.439	0.512	0.102	0.256
	GTCL	**0.307**	**0.211**	**0.179**	**0.307**	**0.624**	**0.715**	**0.121**	**0.327**

表 3 消融实验分析 (续)

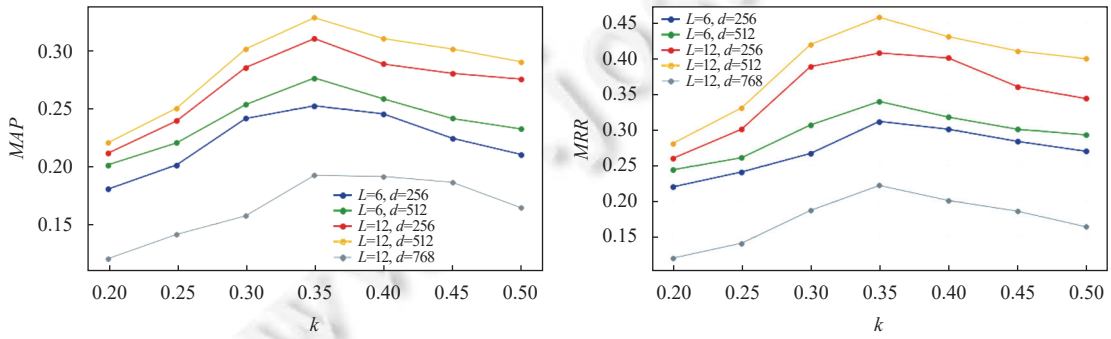
数据集	消融实验	*P*@1	*P*@3	*P*@5	Top@1	Top@5	Top@10	*MAP*	*MRR*
ZXing	w/o AST_{change}	0.223	0.160	0.090	0.223	0.386	0.440	0.110	0.221
	w/o Global	0.265	0.190	0.110	0.265	0.458	0.522	0.130	0.269
	w/o Local	0.354	0.250	0.140	0.354	0.612	0.698	0.180	0.346
	GTCL	**0.407**	**0.291**	**0.169**	**0.407**	**0.705**	**0.805**	**0.221**	**0.407**
Tomcat	w/o AST_{change}	0.254	0.167	0.090	0.254	0.516	0.542	0.086	0.277
	w/o Global	0.319	0.209	0.113	0.319	0.647	0.681	0.108	0.348
	w/o Local	0.376	0.247	0.133	0.376	0.763	0.802	0.127	0.409
	GTCL	**0.447**	**0.293**	**0.158**	**0.447**	**0.907**	**0.954**	**0.151**	**0.487**

实验结果显示, 在 GTCL 模型的实验中, 未使用变更语法树对模型性能的影响最为显著. 具体而言, 去除变更语法树后, 6 个数据集的平均 *MAP* (表 3 中 w/o AST_{change} 的 *MAP* 均值) 为 0.084; GTCL 的平均 *MAP* (表 3 中 GTCL 的 *MAP* 均值) 为 0.161. 相较于 GTCL, 未使用变更语法树的平均 *MAP* 下降 47.8%. 同理, *MRR* 指标下降 52.4%. *P*@*K* 与 Top@*N* 指标也有明显下降. 这一结果验证了变更语法树在捕捉代码变更信息方面的重要性, 以及其在提升模型性能方面的关键作用. 相比之下, 仅使用局部相似度表示或全局相似度表示时, 模型性能的衰退幅度相对较小. 具体而言, 相较于局部相似度表示, 全局相似度表示更能在性能上为模型带来显著的提升, 使得 *MAP* 平均值提升了 16.0%, 而 *MRR* 指标也提升了 36%. 与局部相似度表示相比, 全局相似度表示不仅包含了变更节点的详细信息, 还囊括了整个变更的语义特性, 因此能够更好地保留代码和错误报告的上下文信息. 这种全面的信息表示方式有助于模型更准确地捕捉代码变更与错误报告之间的语义匹配关系, 从而提升定位的准确性. 实验结果还表明, 全局相似度表示在预测过程中发挥了更为关键的作用. 通过增加对变更代码和错误报告间最终相似度得分的权重, 全局相似度表示能够更有效地指导模型的预测过程, 提高定位的准确性. 同样地, 局部相似度表示也在模型中发挥了积极作用. 它通过对局部变更节点微妙变化地捕捉, 增强了模型在识别变化细节上的敏感性和准确性. 这种局部与全局信息的结合, 使得 GTCL 模型在软件错误定位任务中表现出卓越的性能.

综上所述, GTCL 模型中全局相似度表示和局部相似度表示的联合运用是至关重要的. 两者相辅相成, 共同促进了模型在软件错误定位任务中的表现. 全局相似度表示提供了全面的上下文信息, 有助于模型更准确地捕捉代码变更与错误报告之间的语义匹配关系; 而局部相似度表示则增强了模型在识别变化细节上的能力, 提高了定位的准确性和鲁棒性. 这一发现不仅为 GTCL 模型的设计提供了有力支持, 也为未来的软件错误定位研究提供了新的思路和方法.

RQ4: GTCL 的主要参数会对定位结果产生怎样的影响?

本实验旨在优化深度学习模型的超参数设置, 以提高错误定位任务中的平均精确率. 我们评估不同的模型层数 *L*、隐藏状态维度 *d* 和相关性判断阈值 *k* 在 AspectJ 数据集上对 *MRR* 和 *MAP* 指标的影响. 实验结果如图 7 所示.



(a) 不同参数下的*MAP* 指标 (b) 不同参数下的*MRR* 指标

图 7 AspectJ 数据集上不同超参数设置下的 *MAP* 和 *MRR* 指标

根据图 7 显示, 当相关性判断阈值 k 较小时, 本文提出的错误定位方法的实验结果表现显著较差. 随着 k 值的增加, *MRR* 和 *MAP* 这两个指标逐步上升. 这是由于更高的阈值使得更多的不相关代码被排除, 从而提高了实验结果的准确性. 然而, 当 k 值超过 0.35 并持续增大时, 相关代码可能被误判为不相关代码, 这会导致训练模型包含更多的无关更改, 从而降低实际匹配代码的得分. 随着不相关代码比例的增加, 这种误判对最终的错误定位结果的准确性产生负面影响. 图中后半部分显示了 *MAP* 和 *MRR* 值的缓慢下降趋势, 进一步证实了这一点.

此外, 从实验结果中可以观察到, 当使用较小的模型层数 (L=6) 和隐藏状态维度 (d=256) 时, 本文所用错误定位方法的表现显著不佳. 相反, 使用较大的模型层数 (L=12) 和隐藏状态维度 (d=512) 时, 实验效果显著提升. 这一现象可以归因于以下几个因素: 首先, 较大的模型层数 L 增加了模型的深度, 使其能够逐层抽象出输入数据的特征, 捕捉到更加复杂和高级的图结构特征. 在图结构数据中, 深层模型有助于学习节点间更细致和深层次的关系. 其次, 增加隐藏状态维度 d 为模型提供了更大的表示空间, 使其能够编码更多的信息, 提升了模型区分图结构和节点属性的能力. 这种高维的表示有助于模型在处理复杂图数据时提高精度和泛化能力.

然而, 需要指出的是, 较大的模型层数 (L=12) 和隐藏状态维度 (d=768) 也可能导致训练和计算开销的显著增加. 过大的计算开销可能迫使我们减少训练批次大小, 从而导致 L=12 和 d=768 组合下的训练效果不如预期. 因此, 综合考虑模型的性能与计算资源的平衡, 本文选择了 L=12, d=512 和 k=0.35 作为最终的训练配置.

4 相关工作

4.1 基于图的代码表示

程序语言相较于自然语言, 其结构性与层次性更为明显. 这种特性使得基于图的代码表示学习方法在捕捉代码复杂结构和语义信息方面具有显著优势. 基于图的代码表示具有抽象语法树、控制流和数据流等多种维度的表示模式, 这些信息在错误定位过程中发挥了关键作用, 能够提高错误定位的精确性和效率. code2vec[35]通过抽取并编码抽象语法树中的路径, 捕捉代码的结构和语义, 但其并没有表征变更代码的变化信息. Li 等人[36]提出了基于卷积神经网络的 DP-CNN 模型, 该模型针对文件级错误检测, 并将抽象语法树作为代码原始表征以获取更多的结构信息, 但错误检测粒度仅在文件级别. Ma 等人[37]采用了一种特别设计的结构引导注意力来全局捕获控制流图中的信息, 以便更好地从控制流图中提取特征, 但没有综合考虑局部信息. Du 等人[7]提出了语义流图 (SFG) 来捕捉代码语义, 并进一步提出了 SemanticCodeBERT, 用于学习考虑深层代码结构的代码表示. 还有许多工作[38−40]将多种维度的表示模式结合, 使用多重融合图表示代码结构. 例如, Wang 等人[41]提出了 FUNDED, 它在抽象语法树的基础上, 人工定义了 8 种关系, 以将抽象语法树扩增为一个有向多图. 然而, 尽管多重融合图表示代码结构能够显式表征更多维度的代码特征, 但这种方法也存在一些问题. 例如, 相同的节点往往会被合并, 导致部分信息丢失; 代码语言的自然顺序也会受到影响; 同时, 还可能增加无关的冗余节点, 对模型的性能产生负面影响.

为了克服这些问题, 本文构建了基于变更的抽象语法树, 旨在更有效地表征代码的结构信息, 特别是显式表征变更节点, 从而学习错误定位中变更集代码的变更信息. 通过关注代码变更部分, 我们能够更准确地捕捉与错误相关的特征, 提高错误定位的精确性和效率.

4.2 面向变更集的错误定位研究

面向变更集的错误定位是根据软件项目的历史版本信息和错误跟踪器中的错误报告, 通过分析相关的代码变更, 找出最初引入错误的变更集的过程[42]. 这个过程的主要目标是快速且准确地定位导致错误出现的具体代码变更, 以便开发者能够有效地修复问题. 在版本控制系统 (version control system, VCS) 中, 每次提交 (commit) 通常包含更改前和更改后的代码快照. 这种表示方式使得开发者能够清晰地理解代码的增量和变更历史, 从而更容易地追踪和定位错误.

在基于变更集的错误定位研究中, Locus[5]是第 1 个在基于 IR 的错误定位技术中利用代码变更块 (code change hunk) 创建文本语料库的研究者. Bug2Commit[8]模型通过构建基于图的神经网络来对代码提交与错误报告进行匹配. 它将错误报告和代码提交映射到一个共享的嵌入空间中, 并利用图神经网络 (GNN) 来捕捉错误报告与代码提

交之间的复杂关系. FBL-BERT[6]将变化的标签加入训练中, 但是没有利用代码的结构信息, 而是简单地使用代码的文本信息. SemanticCodeBERT[7]在代码预训练的基础上, 使用语义流图来表示代码变更块, 更充分地捕捉了代码语义, 但是没有体现代码的变更.

本文提出了将变更信息显式融入抽象语法树的方法. 通过关注代码变更部分, 并将其与基于变更的抽象语法树相结合, 我们能够更准确地捕捉变更集与错误报告之间的联系. 这种方法不仅利用了代码的文本信息, 还结合了代码的结构信息和变更信息, 从而提高了错误定位的准确性和效率.

综上所述, 定位变更集是软件错误定位任务中的一个重要方面. 通过结合代码的历史版本信息、错误报告以及代码的结构和变更信息, 我们能够更有效地定位导致错误出现的具体代码变更. 这将有助于开发者更快地修复问题, 提高软件的质量和稳定性.

5 有效性威胁

本文的内部有效性威胁主要来自技术实现及实验实施. 为了缓解威胁, 实验使用的图 Transformer 以及抽象语法树构造均采用其官方或开源实现, 并在第 3.2.2 节所描述的系统环境下进行调试. 本文实验中已有研究方法的复现全部基于其作者本人开源的代码和数据, 并保留原作者对内部相关参数的各项设定.

本文的构建威胁主要来自实验采用的评估指标. 为了缓解威胁, 实验采用先前工作广泛使用的指标. 这些指标在前人的研究中已被广泛使用, 并被证明在类似实验中具有较高的有效性和可靠性. 通过使用这些经过充分验证的评估指标, 我们旨在确保实验结果的可靠性和可重复性, 从而有效降低评估过程中的潜在偏差和误差.

本文的外部有效性威胁主要来自实验对象程序. 为了缓解威胁, 实验采用定位诱发错误变更集广泛使用的基准数据集. 因其是基于 Java 的错误数据集, 尽管其在软件工程研究中具有极为广泛的应用, 但是仍然不能够轻易地将实验结论推广到其他程序语言, 这也是本文研究的局限性. 在未来, 我们会拓展实验场景, 在更多的程序语言(例如, C/C++与 Python) 上比对诱发错误变更集定位效果, 从而得到更为一般的结论. 未来工作应该在更为广泛的数据集上进行比对研究.

6 总结与展望

本文提出了基于图 Transformer 的变更集错误定位方法. 该方法从变更前后的代码出发, 首先使用抽象语法树工具生成变更前后代码的抽象语法树, 将变更前后代码之间的变化体现在抽象语法树上并标注变更节点的变更类型; 再分别获得错误报告和历史提交的全局和局部两个视角的代码变更的特征; 然后计算错误报告和历史提交的全局相似度, 并采用相关性匹配计算变更节点与错误报告的局部相似度; 最后, 利用注意力系数聚合全局和局部的相似度计算错误报告与历史提交的相似度进行错误定位.

本文在 6 个提交数据集上进行了充分实验, 并且对实验结果进行了分析与总结. 由于故障定位方法本身就是相当复杂的问题, 本文还有一些需要改进的地方, 后续会对模型进行完善. 尽管基于现有的变更提交模型已经能够取得不错的定位效果, 本文所提出的方法仍然难以理解仅有词级别的差异而语义不同的代码更改. 因此, 后续研究可以考虑利用大模型理解代码的语义信息作为补充, 将两者结合, 从而实现更为全面的定位效果.

References

[1] Nguyen AT, Nguyen TT, Al-Kofahi J, Nguyen HV, Nguyen TN. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lawrence: IEEE, 2011. 263–272. [doi: 10.1109/ase.2011.6100062]

[2] Zhou J, Zhang HY, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 14–24. [doi: 10.1109/ICSE.2012.6227210]

[3] Ma X, Huang P, Jin XX, Wang P, Park S, Shen DC, Zhou YY, Saul LK, Voelker GM. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In: Proc. of the 10th USENIX Conf. on Networked Systems Design and Implementation (NSDI). Lombard: USENIX Association, 2013. 57–70.

[4] 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]

[5] Wen M, Wu RX, Cheung SC. Locus: Locating bugs from software changes. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 262–273.

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

[7] Du YL, Yu ZX. Pre-training code representation with semantic flow graph for effective bug localization. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 579–591. [doi: 10.1145/3611643.3616338]

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

[9] Murali V, Gross L, Qian R, Chandra S. Industry-scale IR-based bug localization: A perspective from Facebook. In: Proc. of the 43rd Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid: IEEE, 2021. 188–197. [doi: 10.1109/ICSE-SEIP52600.2021.00028]

[10] Zhou X, Xu BW, Han DG, Yang Z, He JD, Lo D. CCBERT: Self-supervised code change representation learning. In: Proc. of the 2023 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Bogotá: IEEE, 2023. 182–193. [doi: 10.1109/ICSME58846.2023.00028]

[11] 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]

[12] Jung TH. CommitBERT: Commit message generation using pre-trained programming language model. arXiv:2105.14242, 2021.

[13] 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]

[14] 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]

[15] Luo YK, Thost V, Shi L. Transformers over directed acyclic graphs. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2023. 2070.

[16] Wang WH, Zhang KC, Li G, Liu SQ, Li AR, Jin Z, Liu Y. Learning program representations with a tree-structured Transformer. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Taipa: IEEE, 2023. 248–259. [doi: 10.1109/SANER56733.2023.00032]

[17] Ying CX, Cai TL, Luo SJ, Zheng SX, Ke GL, He D, Shen YM, Liu TY. Do Transformers really perform bad for graph representation? In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 2212.

[18] 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]

[19] Falleri JR, Martinez M. Fine-grained, accurate, and scalable source code differencing. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 2856–2867. [doi: 10.1145/3597503.3639148]

[20] An GB, Hong JG, Kim N, Yoo S. Fonte: Finding bug inducing commits from failures. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 589–601. [doi: 10.1109/ICSE48619.2023.00059]

[21] Yin PC, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL. Learning to represent edits. arXiv:1810.13337, 2018.

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

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

[24] 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]

[25] 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 Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 970–981. [doi: 10.1145/3510003.3510069]

[26] Tarjan R. Depth-first search and linear graph algorithms. In: Proc. of the 12th Annual Symp. on Switching and Automata Theory (SWAT 1971). East Lansing: IEEE, 1971. 114–121. [doi: 10.1109/SWAT.1971.10]

[27] Zhang K, Mao ZD, Wang Q, Zhang YD. Negative-aware attention framework for image-text matching. In: Proc. of the 2022 IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR). New Orleans: IEEE, 2022. 15640–15649. [doi: 10.1109/CVPR52688.2022.01521]

[28] Huo X, Li M, Zhou ZH. Learning unified features from natural and programming languages for locating buggy source code. In: Proc. of the 25th Int'l Conf. on Artificial Intelligence. New York: AAAI, 2016. 1606–1612.

[29] Huo X, Li M. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. Melbourne, Australia: AAAI, 2017. 1909–1915. [doi: 10.24963/ijcai.2017/265]

[30] Xiao Y, Keung J, Bennin KE, Mi Q. Improving bug localization with word embedding and enhanced convolutional neural networks. Information and Software Technology, 2019, 105: 17–29. [doi: 10.1016/j.infsof.2018.08.002]

[31] Shen ZW, Niu FF, Li CY, Chen X, Li Q, Ge JD, Luo B. Software bug location method combining information retrieval and deep model features. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3245–3264 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7111.htm [doi: 10.13328/j.cnki.jos.007111]

[32] Youm KC, Ahn J, Lee E. Improved bug localization based on code change histories and bug reports. Information and Software Technology, 2017, 82: 177–192. [doi: 10.1016/j.infsof.2016.11.002]

[33] Lam AN, Nguyen AT, Nguyen HA, Nguyen TN. Bug localization with combination of deep learning and information retrieval. In: Proc. of the 25th Int'l Conf. on Program Comprehension (ICPC). Buenos Aires: IEEE, 2017. 218–229. [doi: 10.1109/ICPC.2017.24]

[34] Qi BH, Sun HL, Yuan W, Zhang HY, Meng XX. DreamLoc: A deep relevance matching-based framework for bug localization. IEEE Trans. on Reliability, 2022, 71(1): 235–249. [doi: 10.1109/tr.2021.3104728]

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

[36] Li J, He PJ, Zhu JM, Lyu MR. Software defect prediction via convolutional neural network. In: Proc. of the 2017 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). Prague: IEEE, 2017. 318–328. [doi: 10.1109/QRS.2017.42]

[37] Ma YF, Du YL, Li M. Capturing the long-distance dependency in the control flow graph via structural-guided attention for bug localization. In: Proc. of the 32nd Int'l Joint Conf. on Artificial Intelligence (IJCAI). Macao: IJCAI, 2023. 2242–2250. [doi: 10.24963/ijcai.2023/249]

[38] Peng Y, Wang CZ, Wang WX, Gao CY, Lyu MR. Generative type inference for Python. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 988–999. [doi: 10.1109/ASE56229.2023.00031]

[39] Tang XZ, Tian HY, Chen ZH, Pian W, Ezzini S, Kabore AK, Habib A, Klein J, Bissyande TF. Learning to represent patches. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. (ICSE-Companion). Lisbon: ACM, 2024. 396–397. [doi: 10.1145/3639478.3643521]

[40] Zhang YW, Li G, Jin Z, Xing Y. Neural program repair with program dependence analysis and effective filter mechanism. arXiv: 2305.09315, 2023.

[41] 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]

[42] 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 with English abstract). http://www.jos.org.cn/1000-9825/6749.htm [doi: 10.13328/j.cnki.jos.006749]

附中文参考文献

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

[24] 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]

[31] 申宗汶, 牛菲菲, 李传艺, 陈翔, 李奇, 葛季栋, 骆斌. 融合信息检索和深度模型特征的软件缺陷定位方法. 软件学报, 2024, 35(7): 3245–3264. http://www.jos.org.cn/1000-9825/7111.htm [doi: 10.13328/j.cnki.jos.007111]

[42] 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 [doi: 10.13328/j.cnki.jos.006749]

作者简介

刘浩, 硕士生, 主要研究领域为软件工程, 软件缺陷检测, 代码表示.

杜军威, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程, 智能化工安全, 图神经网络, 推荐算法.

李玉莹, 博士, 讲师, 主要研究领域为智能软件工程, 软件测试.

房敏营, 博士生, CCF 学生会员, 主要研究领域为软件工程, 代码分析, 代码表示.

贾学海, 硕士生, 主要研究领域为软件工程, 代码漏洞检测, 代码表示.