面向程序员编程过程中重点代码修改识别研究

刘 雨[1,2],姜 瑛[1,2]
(1. 云南省计算机技术应用重点实验室,云南 昆明 650500;
2. 昆明理工大学 信息工程与自动化学院,云南 昆明 650500)

摘 要:软件的持续变化增加了维护的复杂性。为提高维护效率,该文提出了一种基于动态抽象语法树(Dynamic Abstract Syntax Tree,DAST)的重点代码修改识别方法。传统的修改识别方法主要依赖语法分析,忽略了程序语义的重要信息,导致识别的精度和粒度受到限制。通过 DAST 提取出代码的重点信息,结合语法结构和语义信息,以提高程序员在编程过程中重点代码修改识别的准确性。同时,引入了注意力机制,进一步突出重点代码的频繁修改区域。最后在版本变化和三个不同程序员的数据集上进行实验,结果表明,与传统的重点代码修改识别方法相比,该方法在准确性和稳定性方面均有提升,验证了在重点代码修改识别任务上的有效性,从而提升了软件维护效率。

关键词:软件维护;语法结构;语义信息;注意力机制;深度学习;重点代码修改
中图分类号:TP311.5 **文献标识码**:A **文章编号**:1673-629X(2025)01-0081-07
doi:10.20165/j.cnki.ISSN1673-629X.2024.0263

Research of Focused Code Modification Identification in Programmer-oriented Programming Processes

LIU Yu[1,2],JIANG Ying[1,2]
(1. Computer Technology Application Key Laboratory of Yunnan Province,Kunming 650500,China;
2. School of Information Engineering and Automation,Kunming University of Science and Technology,
Kunming 650500,China)

Abstract:Continuous changes in software increase the complexity of maintenance. To improve the maintenance efficiency,we propose a focused code modification identification method based on Dynamic Abstract Syntax Tree (DAST). The traditional modification identification method mainly relies on syntactic analysis and ignores the important information of program semantics,resulting in limited accuracy and granularity of identification. The key information of the code is extracted by DAST and the syntactic structure and semantic information is combined to improve the programmer´s key code modification identification accuracy in the programming process. Meanwhile,we introduce an attention mechanism to further highlight the frequently modified regions of focused code. Finally, experiments are conducted on version change and three different programmers´ datasets. It is showed that compared with the traditional focused code modification identification method,the proposed method improves both accuracy and stability,and verifies the effectiveness in the task of focused code modification identification,thus improving the software maintenance efficiency.

Key words:software maintenance;syntactic structure;semantic information;attention mechanisms;deep learning;focused code modification

0 引 言

在当前软件开发领域中,编程人员经常需要对代码进行迭代和修改,以添加新功能或提高代码质量。这些修改是软件系统持续演进的重要标志,也是确保系统健壮性和可维护性的关键。然而,在不断变化的开发需求中,准确识别和理解重点代码的修改是一个复杂的任务[1]。因此,为应对软件系统规模的扩大和频繁变更的需求,有效的识别重点代码修改变得尤为关键。

目前,多数研究人员在进行重点代码修改的识别

收稿日期:2024-05-27 **修回日期**:2024-09-27
基金项目:国家自然科学基金项目(62162038);国家重点研发计划项目(2018YFB1003904);云南省计算机技术应用重点实验室开放基金资助项目(2020101)
作者简介:刘 雨(1999-),男,硕士研究生,CCF 会员(N1503G),研究方向为智能软件工程、软件质量保障与测试;姜 瑛(1974-),女,博士,教授,博导,CCF 杰出会员(08959D),研究方向为软件质量保证与测试、云计算、大数据分析、智能软件工程。

时,主要基于抽象语法树(Abstract Syntax Tree,AST)进行建模。在数据预处理阶段对 AST 进行遍历,从而得到满足模型输入的 AST 节点序列。然而,仅利用 AST 节点序列会导致 AST 结构信息的丢失,从而忽视了代码中的语义信息,无法全面捕捉代码之间的依赖关系。因此,为了更准确地理解代码的演化,研究人员需要开发新的方法,这些方法不仅应关注语法结构的分析,还需要探讨代码的语义层面。

针对上述问题,该文提出了一种基于动态抽象语法树(Dynamic Abstract Syntax Tree,DAST)[2]的方法,该方法不仅可以处理程序代码的语法结构,还可以理解代码的语义内容。首先阐述了如何利用 DAST 提取重点代码,并处理语法信息和语义输入的方法。接着,介绍了如何设计向量训练模型,以将节点信息有效地映射到高维向量空间。此外,为了提高模型对代码变化的敏感性和准确性,研究中引入了注意力机制,特别关注重点代码中修改频繁的区域。最后,通过对比实验,展示了该方法与其他几种基线方法在性能上的差异,验证了其有效性。为软件开发中代码修改的理解与管理提供了有力的技术支持。

1 相关工作

在代码修改识别领域,现有研究主要集中于利用 AST 的结构信息来识别代码变更。例如,Lozoya 等人[3]通过比对变更前后代码的 AST 路径集合,提取所有代码文件的前后版本,并基于 AST 表示转换为上下文,计算其对称差,以构建代码变更的结构化表示,从而完成代码变更识别任务。Yao 等人[4]采用树差分方法比对变更前后代码的 AST,通过 AST 的差分结果,提取变更前代码的最短 AST 编辑操作序列,作为代码变更的结构化方法。曹英魁等人[5]提出了一种结构信息增强的代码修改自动转换方法,通过引入结构信息,对代码修改进行自动化转换和优化,提高代码修改识别的效率和准确性。Malmi 等人[6]提出的代码文本编辑识别模型,通过学习如何编辑现有代码文本,识别代码修改的概率。该模型采用序列标记方法,为输入序列中的每个 token 标注分类标签。Brody 等人[7]通过分析代码的结构路径来识别上下文相关的代码变更,核心在于直接对结构性编辑进行表示,将编辑操作具体化为程序的 AST 路径序列,并将语法元素嵌入向量空间,建立 Pointer 网络模型来识别代码编辑的概率。曹炳豪等人[8]提出了一种基于改进代码属性图的图神经网络模型,通过图的方法捕获源代码中的语法和语义信息。此外,王青叶等人[9]探讨了代码审查过程中代码变更恢复的方法,结合深度学习和图神经网络的方法,可以在复杂的软件系统中有效地识别和分析代码修改。

传统表示技术将代码视为序列化文本进行处理。例如,Hu 等人[10]通过遍历 AST 节点序列表示代码文本。Zhang 等人[11]利用 word2vec 生成代码词向量,但未提取节点的树结构信息。Mou 等人[12]在训练 AST 节点词向量时,将子节点作为上下文信息,尽管考虑了一定的树形结构,但上下文信息仍不够全面。

上述研究大多侧重于分析代码的结构特征,而忽视了代码的语义层面,可能导致理解上的局限,因为单靠结构信息难以完全揭示代码意图。针对这一问题,该文将语法和语义特征相结合用于识别重点代码的修改。在训练节点词向量表示时,将父节点也作为上下文信息,以更全面提取代码结构信息。同时,引入注意力机制,使模型能够关注重点代码的频繁修改部分。

2 基于语法和语义的注意力增强重点代码修改识别方法

在软件开发过程中,由于软件需求的不断变化,程序员经常需要对代码进行修改。这些修改增加软件维护的难度。为了解决这个问题,该文提出了一种基于 DAST 的方法,结合了语法和语义信息,以提高重点代码修改点的识别精确性。并通过注意力机制,加强对编程中频繁修改的重点代码区域的关注。方法整体框架如图 1 所示。

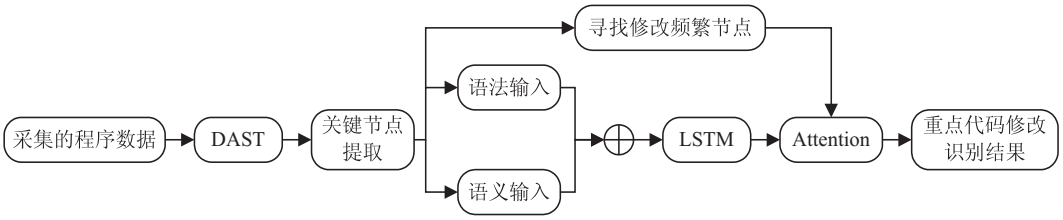

图 1 方法整体框架

2.1 重点代码的定义与提取

为了深入分析和理解代码结构,该文引入了重点代码的概念,以确保重点代码修改识别方法的有效性。重点代码在文中被定义为程序中对功能和结构具有关键影响的部分。根据 McConnell[13]在《Code Complete》中提出的标准,重点代码可以涵盖函数、类、异常处理、控制流和错误处理等关键程序结构。

为了能识别这些重点代码,该文采用 DAST 对重

点代码进行分析。DAST 能够更好地捕捉程序的动态变化和结构信息。通过分析 DAST 中的节点类型,可以识别函数定义、赋值操作、控制结构等重点代码元素,这些节点类型反映了程序中的重要结构和流程控制,如表 1 所示。

表 1　重点代码元素与 DAST 节点类型对应表

代码元素	节点类型
函数定义	FunctionDef
赋值操作	Assign
增量赋值操作	AugAssign
带类型注解的赋值操作	AnnAssign
全局变量声明	Global
异常处理	Try, Raise
输入输出	With, AsyncWith
控制流结构(条件判断)	If
控制流结构(循环)	For, AsyncFor, While
类定义	ClassDef

通过上述分析,为了提取出这些重点代码元素,该文提出了 DAST 重点代码追踪算法。通过向上追踪节点的祖先,判断是否包含重点代码类型,如算法 1 所示。如果一个节点的祖先中包含有关重点代码的节点类型,则该节点被视为重点代码的一部分。

算法 1:DAST 重点代码追踪算法
输入:DAST,重点代码元素 critical_elements
输出:重点代码节点列表 critical_node
1. 获取到 DAST
2. 从 DAST 中初始化出顶层节点 node
3. 初始化 ancestor_stack 栈,critical_node 列表
4. DFS(node)
5. 将 node 压入 ancestor_stack 栈中
6. if node in critical_elements
7. 将当前 node 添加到 critical_node 中
8. elif ancestor_stack 中任一元素在 critical_elements 中
9. 将当前 node 添加到 critical_node 中
10.　end if
11.　for child in node.children do
12.　　DFS(child)
13.　　将 ancestor_stack 栈中最后一个元素弹出
14. end for
15. return critical_node

2.2　语法与语义信息的提取和向量化

在利用 DAST 提取到重点代码节点后,若仅依赖这些节点的数据进行重点代码修改的识别,而忽略节点间的关系,会导致对代码的理解不全面。此外,如果节点数据未转换为易于处理的向量形式,将难以直接应用于修改识别方法。因此,该文通过提取并转换语法和语义信息为向量形式,以增强修改识别方法对代码结构含义的理解。

2.2.1　语法信息

为了有效捕获代码的语法结构并提取程序语句的结构信息,该文在分析 DAST 中的每个节点时,不仅需要考虑每个节点自身的信息,还要包括其上下文信息。即将节点的父节点和子节点信息与当前节点的信息结合,形成语法信息。这种方法能更准确地反映程序代码的结构特征,增强对代码复杂结构的理解能力。

通过上述分析,将语法信息输入序列记为 $M=(x_1+x_2+\cdots+x_n)$,其中 x_i 表示 DAST 中的一个节点与该节点的上下文信息,即 $x_i=(T_i,P_i,C_i)$,T_i 表示当前节点类型,P_i 表示当前节点的父节点,C_i 表示当前节点的子节点序列。这种表示方法记录了每个节点本身的信息,还包括与其直接相关的层级关系。

2.2.2　语义信息

在分析完代码的语法结构后,理解语句的含义也是至关重要,它传递了程序语句的具体含义,并为识别重点代码修改提供关键信息。因此,进行语义信息输入的目的就是增强模型对开发人员编程行为和意图的理解,提高在软件开发过程中重点代码修改识别的准确性。这使得模型不仅能够识别代码的结构变化,也能够理解这些变化背后的逻辑,为软件的维护提供更为全面和深入的支持。

根据 Hanam 等人[14]描述的控制依赖概念及其在程序分析中的应用,可以证明控制依赖分析在程序理解方面具有重要意义。因此,在理解程序代码的语义信息时,需要把握程序中的依赖关系,这些依赖关系揭示了代码中各语句间的相互联系。其中控制依赖是一种关键的依赖类型,描述了程序的控制结构。控制依赖性不仅反映了代码的结构,也揭示了语句间的语义关联。在 DAST 中,每个节点不仅对应于程序中的一条语句,而且还能反映该语句与其他语句之间的控制依赖关系。这些节点和它们的关系共同构成了语义信息的表示,为重点代码修改识别任务提供了必要的信息。

该文采用深度优先遍历法从 DAST 中提取节点的控制依赖关系,作为语义信息输入,具体算法流程如算法 2 所示。首先,初始化 DAST、语义信息栈和结果列表。在遍历过程中,根据节点类型记录语义信息并更新结果列表。对于控制结构节点,先将相关类型信息压入栈中,处理完子节点后再出栈。最终,得到包含所有节点语义信息的结果集。

算法 2:在 DAST 中寻找节点的控制依赖关系算法
输入:DAST
输出:DAST 中每个节点的语义信息 result
1. 获取到 DAST

```
2. 初始化 semantic_stack 栈,result=[ ]
3. 从 DAST 中初始化出顶层节点 node
4. DFS( node)
5.      if semantic_stack 不为空 then
6.        将当前节点 node 与 semantic_stack 添加到 result 中
7.      else
8.        将当前节点 node 与 None 添加到 result 中
9.      end if
10.     获取当前节点的类型 type
11.     if type 属于控制结构 then
12.       将 type 压入 semantic_stack 栈中
13.       sign 标记为 True
14.     else
15.       sign 标记为 False
16.     end if
17.     for child in node. children do
18.       DFS( child)
19.     if sign 为 True then
20.         将 semantic_stack 栈中最后一个元素弹出
21.     end if
22. end for
23. return result
```

通过上述分析,可以将语义信息输入序列记为 $N=(y_1,y_2+\cdots+y_n)$,其中 y_i 表示一个 DAST 节点与该节点的控制依赖信息,即 $y_i=(T_i,D_i)$,T_i 表示当前节点类型,D_i 表示当前节点的控制依赖语句的 DAST 节点序列。

2.2.3 基于树的词向量训练模型

在提取代码的语义和语法信息后,为了将代码的文本节点转化为易于处理的向量形式,该文采用了一种基于树的词向量训练模型,如图 2 所示。这种模型能够将文本节点有效的向量化,从而便于后续的代码分析和处理。

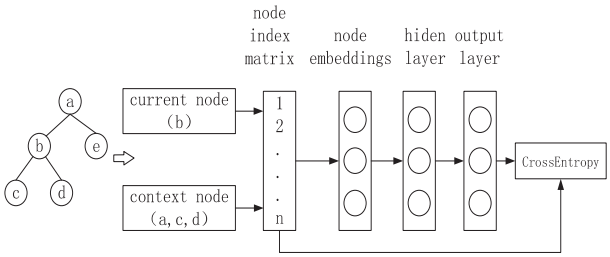

图 2 词向量训练模型

如图 2 所示,基于树的词向量训练模型[15]利用树结构学习节点表示。模型关注当前节点 b 时,通过整合其父节点 a 和子节点 c、d 的信息,形成节点 b 的上下文(a,c,d),以训练节点向量表示。这种方法通过考虑节点在树中的位置,增强了其结构信息,从而提升了节点的表达能力。

2.3 Semantic-ATT-LSTM

通过词向量训练模型生成每个节点的输出向量后,将语法与语义信息转换为向量并拼接,形成完整的输入向量。该向量同时反映代码修改的语法和语义特征,为后续的重点代码修改识别任务提供更全面的信息。

为了消除尺度的影响,需对语法和语义向量进行归一化处理。然后将归一化后的向量逐元素拼接,形成新的输入向量。记 DAST 节点序列的语法向量为 $G=(g_1,g_2,\cdots,g_n)$,其中 g_i 表示一个 DAST 节点的语法信息的嵌入表示,语义向量为 $S=(s_1,s_2,\cdots,s_n)$,其中 s_i 表示为一个 DAST 节点的语义节点信息的嵌入表示,则拼接后的向量 $C=\{(t_1,g_1,s_1),(t_2,g_2,s_2),\cdots,(t_n,g_n,s_n)\}$,其中 t_i 表示为一个 DAST 节点类型的嵌入表示。

LSTM[16]的记忆单元可以保持先前节点的信息,这对处理代码上下文依赖的节点至关重要。例如,在修改变量引用时,模型须掌握变量的声明位置和作用域。LSTM 不仅能够存储和回忆这些关键信息,还在处理长距离依赖上表现良好,能有效识别 DAST 中跨越多个层级的远距离相关节点。因此,将 LSTM 网络应用于重点代码修改识别任务,如图 3 所示。

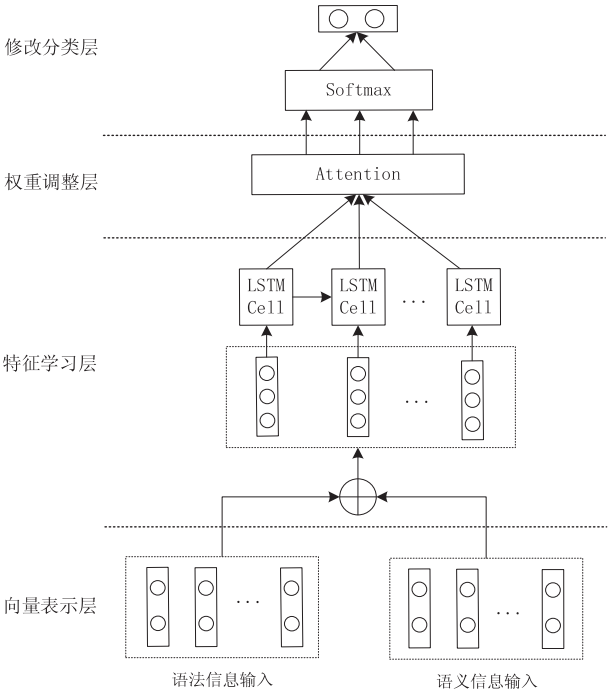

图 3 Semantic-ATT-LSTM 模型

2.4 注意力机制在重点代码修改识别中的应用

尽管 LSTM 网络在处理长距离依赖方面有优势,但在准确识别重点代码修改上仍显不足。该文将注意力机制融入现有方法,以增强代码的语法和语义理解。注意力机制通过聚焦关键信息,赋予重要节点更多权重,提升判断准确度。该方法中注意力机制评估输入节点间的关注和影响程度。

注意力函数[17]本质上是一种由多个查询(Q)和

相应的键值对($K-V$)组成的映射函数。其计算过程可以分为三个步骤:

(1)计算相似度权重:计算查询(Q)与每个键(K)的相似度权重,使用公式 1 表示。

$$f(Q,K)=QK^t \tag{1}$$

(2)使权重归一化:函数对相似度权重进行归一化,得到权重向量 a ,使用公式 2 表示。

$$a=\mathrm{softmax}(f(Q,K)) \tag{2}$$

(3)注意力值计算:将归一化后的权重与相应的值(V)进行加权求和,得到最终的注意力值,使用公式 3 表示。

$$\mathrm{Attention}(Q,K,V)=\sum a_i V \tag{3}$$

根据上述注意力函数的计算过程,结合 DAST 的修改频繁节点, Q 、K 和 V 的构建方式如下:

(1) Q 的构建:根据 DAST 的修改频繁节点,并将这些节点的嵌入表示构建查询向量,反映代码变更部分。

(2)键值对的构建:为 DAST 中的每个节点生成 K 和 V 。节点的嵌入表示作为 K ,代表节点特征;同样的嵌入表示作为 V ,用于生成最终特征表示。

(3)映射函数的计算:针对每个节点,利用公式 1 计算查询向量与 K 的点积,并通过公式 2 对结果进行归一化,生成注意力权重,量化每个节点与查询向量之间的相关性。

(4)特征提取:利用公式 1 计算查询向量与 K 的点积,通过公式 2 对结果归一化,生成注意力权重,量化每个节点与查询向量的相关性。

将上述注意力机制的输出应用于重点修改节点,可以更有效地识别重点代码修改部分。

3　实验及分析

该文旨在构建一种识别程序员编程过程中重点代码修改的方法,提高软件开发的维护效率和健壮性。为评估该方法的有效性,通过宋超等人[18]提出捕获工具收集数据,并在重点代码修改识别任务上进行了一系列实验。

3.1　实验数据

实验从两个维度验证数据。首先,记录了半年内四个项目版本的代码更新,展示代码的持续更新和迭代特性。其次,分析了三名程序员在不同编程任务上的修改记录,以了解其编程风格和习惯差异。程序员 1 在编写代码时几乎不进行修改,注重一次性完成;程序员 2 倾向于在编写代码时进行较少的修改;程序员 3 则倾向于频繁优化代码,反映了迭代式开发的方法。最后,根据收集的原始代码数据,利用算法 1 提取重点代码节点,结果如表 2 所示。

表 2　提取重点代码前后的节点数对比

数据集	原始代码节点数	重点代码节点数
版本变化数据集	10 956	6 243
程序员 1 数据集	6 345	4 963
程序员 2 数据集	32 777	19 900
程序员 3 数据集	58 057	34 569

3.2　实验评估与分析

为了客观评估提出的修改识别方法的性能,采用了国际通用的评价标准:准确率(Precision)、召回率(Recall)和 F1 值(F1-measure)[19]。在修改识别中,准确率指正确预测的修改节点占所有预测为修改节点的比例,是衡量模型预测准确性的重要指标;召回率则表示在所有实际修改的节点中,模型成功预测为修改的节点比例;F1 值是一个综合考虑准确率和召回率的指标。

为验证文中方法在识别重点代码修改方面的有效性,与 Malmi 等人[6]的 LaserTagger 方法和 Brody 等人[7]的 C3PO 方法进行对比实验,并进行了消融实验。实验结果如表 3 所示,展示了在持续编程过程中,各对比方法在提取重点代码后的版本修改分析数据集上的表现。

表 3　持续编程过程的版本修改实验结果　%

数据集	版本变化数据集		
评价标准	准确率	召回率	F1 值
LaserTagger	72.38	68.83	68.89
C3PO	77.18	95.14	78.75
LSTM	76.94	91.89	77.98
ATT-LSTM	77.26	92.79	78.39
Semantic-ATT-LSTM	78.86	92.44	79.15

表 3 对比了未融合注意力机制的 LSTM 模型与加入注意力机制的 ATT-LSTM 模型的实验结果。LSTM 的准确率为 76.94%,召回率为 91.89%,F1 值为 77.08%。引入注意力机制后,ATT-LSTM 的准确率提高到 77.26%,召回率提高到 92.79%,F1 值提高到 78.39%。这一对比结果显示了准确率、召回率和 F1

值均有提升,证实了注意力机制在提高重点代码修改识别性能方面的有效性。

进一步对比未融合语义信息的 ATT-LSTM 模型与加入语义信息的 Semantic-ATT-LSTM 模型的实验结果。ATT-LSTM 的准确率为 77.26%,召回率为 92.79%,F1 值为 78.39%。引入语义信息后,Semantic-ATT-LSTM 的准确率提升至 78.86%,召回率为 92.44%,F1 值为 79.15%。表明语义信息有效提升了对重要代码修改的识别能力。

LaserTagger 在识别版本修改方面的表现为:准确率 72.38%,召回率 68.83%,F1 值 68.89%。这表明序列标记方法在预测准确性上表现良好,但在全面识别正类样本方面有局限性。C3PO 的准确率为 77.18%,召回率为 95.14%,F1 值为 78.75%,在各项指标上优于 LaserTagger,尤其是在召回率方面,表明其在识别正类样本方面表现更好。同时 Semantic-ATT-LSTM 在所有评价指标上均优于 LaserTagger。尽管其召回率略低于 C3PO,但准确率和 F1 值均高于 C3PO,这表明文中方法在整体性能上更为可靠。

在不同程序员的编程修改分析中,首先是程序员 1 的编程修改数据,文中方法与其他对比方法在该数据集上的实验结果如表 4 所示。

表 4 程序员 1 的编程修改识别实验结果 %

数据集	程序员 1 数据集		
评价标准	准确率	召回率	F1 值
LaserTagger	87.00	34.97	43.69
C3PO	87.08	34.97	43.84
LSTM	86.29	13.66	22.32
ATT-LSTM	87.08	35.52	44.22
Semantic-ATT-LSTM	90.70	91.85	74.12

从表 4 的实验结果可以看出,尽管程序员 1 很少进行修改,Semantic-ATT-LSTM 的召回率达到了 91.85%,高于其他四种方法。这表明该模型在识别修改样本方面表现良好,能够捕捉少量修改实例,展现出较强的泛化能力。同时,Semantic-ATT-LSTM 的 F1 值达到了 74.12%,均高于其他方法,说明它在保持较高准确性的同时,最大限度地覆盖了程序员 1 的修改情况,实现了准确率和召回率的良好平衡。

为了进一步验证文中方法的有效性,利用程序员 2 和 3 的编程修改数据,将文中方法与其他对比方法在该数据集上进行了实验,结果见表 5 和表 6。

表 5 程序员 2 的编程修改识别实验结果 %

数据集	程序员 2 数据集		
评价标准	准确率	召回率	F1 值
LaserTagger	74.99	69.86	70.36
C3PO	80.87	96.35	81.06
LSTM	79.66	95.65	80.03
ATT-LSTM	80.67	96.00	80.84
Semantic-ATT-LSTM	83.42	96.77	83.25

表 6 程序员 3 的编程修改识别实验结果 %

数据集	程序员 3 数据集		
评价标准	准确率	召回率	F1 值
LaserTagger	74.14	70.56	71.89
C3PO	80.29	96.17	82.05
LSTM	80.11	95.40	81.79
ATT-LSTM	80.17	95.86	81.92
Semantic-ATT-LSTM	83.03	98.15	84.42

从表 5 和表 6 的实验结果可以看出,提出的 Semantic-ATT-LSTM 方法在程序员 2 和程序员 3 上的数据中,准确率、召回率和 F1 值均高于其他四种方法,表明文中方法具有较好的识别准确性,能够更全面地理解代码的结构和语义信息,更好地捕捉程序员的修改行为。

综上所述,通过比较四种方法在不同实验数据集上的结果可以看出,Semantic-ATT-LSTM 在版本修改和三个程序员的实验中都展现出了较好的性能,证明了结合语义信息和注意力机制在处理重点代码修改任务的有效性。

4　结束语

该文探讨了在软件开发中有效识别重点代码修改识别的问题。通过结合语法和语义信息,可以更全面考虑代码的信息,增强对软件变更影响的洞察力。采用基于树结构的词向量训练技术,可以有效捕捉代码节点的结构位置。通过注意力机制聚焦 DAST 中的频繁修改节点,提高了代码变更的捕获能力。最后实验结果表明,该方法在准确性和稳定性方面均有提升,验证了其在重点代码修改识别任务上的有效性。因此,该方法可用于代码审查和质量保证,提升软件可靠性和可维护性。

但该方法在 DAST 的构建和注意力机制的实现可能会增加计算复杂度,导致在大规模项目中的应用受到限制。未来的研究可以针对这些问题进行优化,例如通过改进算法减少计算开销,以进一步提高识别的效率和适用范围。

参考文献:

[1] BRUDARU I I,ZELLER A. What is the long-term impact of changes[C]//Proceedings of the 2008 international workshop on recommendation systems for software engineering (RSSE ' 08). Atlanta:Association for Computing Machinery,2008:30-32.

[2] YAO W,JIANG Y,YANG Y. The metric for automatic code generation based on dynamic abstract syntax tree[J]. International Journal of Digital Crime and Forensics,2023,15(1):1-20.

[3] LOZOYA R C, BAUMANN A, SABETTA A, et al. Commit2vec:learning distributed representations of code changes[J]. SN Computer Science,2021,2(3):150.

[4] YAO Z,XU F F,YIN P,et al. Learning structural edits via incremental tree transformations[J]. arXiv:2101.12087,2021.

[5] 曹英魁,孙泽宇,邹艳珍,等. 一种结构信息增强的代码修改自动转换方法[J]. 软件学报,2021,32(4):1006-1022.

[6] MALMI E,KRAUSE S,ROTHE S,et al. Encode,tag,realize:high-precision text editing[J]. arXiv:1909.01187,2019.

[7] BRODY S,ALON U,YAHAV E. A structural model for contextual code changes[C]//Proceedings of the ACM on programming languages. [s.l.]:ACM,2020:1-28.

[8] 曹炳豪,汪智超,朱二周. 面向软件漏洞检测的改进代码属性图的图神经网络[J]. 微电子学与计算机,2024,41(1):74-82.

[9] 王青叶,万志远,李善平,等. 代码审查中代码变更恢复的经验研究[J]. 软件学报,2022,33(7):2581-2598.

[10] HU X,LI G,XIA X,et al. Deep code comment generation with hybrid lexical and syntactical information[J]. Empirical Software Engineering,2020,25:2179-2217.

[11] ZHANG Jian,WANG Xu,ZHANG Hongyu,et al. A novel neural source code representation based on abstract syntax tree[C]//Proceedings of the 41st international conference on software engineering. Montreal:IEEE,2019:783-794.

[12] MOU Lili,LI Ge,ZHANG Lu,et al. Convolutional neural networks over tree structures for programming language processing[C]//Proceedings of the thirtieth AAAI conference on artificial intelligence. Phoenix:AAAI,2016:1287-1293.

[13] MCCONNELL S. Code complete[M]. Boston:Pearson Education,2004.

[14] HANAM Q,MESBAH A,HOLMES R. Aiding code change understanding with semantic change impact analysis[C]//Proceedings of the IEEE international conference on software maintenance and evolution (ICSME ' 19). Cleveland:IEEE,2019:202-212.

[15] HENKEL J,LAHIRI S K,LIBLIT B,et al. Code vectors:understanding programs through embedded abstracted symbolic traces[C]//Proceedings of the 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. Lake Buena Vista:ACM,2018:163-174.

[16] HOCHREITER S,SCHMIDHUBER J. Long short-term memory[J]. Neural Computation,1997,9(8):1735-1780.

[17] 吴小华,陈　莉,魏甜甜,等. 基于 Self-Attention 和 Bi-LSTM 的中文短文本情感分析[J]. 中文信息学报,2019,33(6):100-107.

[18] 宋　超,姜　瑛,杨　扬. 应用代码自动生成与补全的编程贡献分析[J]. 信息技术,2022,46(12):29-38.

[19] KIM Y,DENTON C,HOANG L,et al. Structured attention networks[J]. arXiv:1702.00887,2017.