

面向代码克隆检测的多维源代码表征学习方法研究

王丹

2024 年 5 月

面向代码克隆检测的多维源代码表征学习方法研究

王丹

北京理工大学

中图分类号：TQ028.1

UDC 分类号：540

面向代码克隆检测的多维源代码表征学习方法研究

作 者 姓 名	<u>王丹</u>
学 院 名 称	<u>计算机学院</u>
指 导 教 师	<u>马锐教授</u>
答辩委员会主席	<u>** 教授</u>
申 请 学 位	<u>工程硕士</u>
学 科 / 类 别	<u>计算机技术</u>
学位授予单位	<u>北京理工大学</u>
论文答辩日期	<u>2024 年 6 月 4 日</u>

Research on Multidimensional Source Code Representation Learning for Code Clone Detection

Candidate Name:	<u>Dan Wang</u>
School or Department:	<u>Beijing Institute of Technology</u>
Faculty Mentor:	<u>Associate Prof. Rui Ma</u>
Chair, Thesis Committee:	<u>Prof. **</u>
Degree Applied:	<u>Master of Science</u>
Major:	<u>Computer Technology</u>
Degree by:	<u>Beijing Institute of Technology</u>
The Date of Defence:	<u>June, 4th, 2024</u>

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下独立完成的研究成果。文中所撰写内容符合以下学术规范（请勾选）：

论文综述遵循“适当引用”的规范，全部引用的内容不超过 50%。

论文中的研究数据及结果不存在篡改、剽窃、抄袭、伪造等学术不端行为，并愿意承担因学术不端行为所带来的一切后果和法律责任。

文中依法引用他人的成果，均已做出明确标注或得到许可。

论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

与本人一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此声明。

签 名：

日 期：

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：

- ① 学校有权保管、并向有关部门送交学位论文的原件与复印件；
 - ② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；
 - ③ 学校可允许学位论文被查阅或借阅；
 - ④ 学校可以学术交流为目的，复制赠送和交换学位论文；
 - ⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

签 名： 日期：

导师签名： 日期：

摘要

代码克隆检测是软件工程领域的重要任务，如何对源代码进行表征学习决定了对源代码表征抽取的程度，进而影响下游任务所能检测的精度。作为代码克隆检测任务的核心技术和研究热点，现有的代码表征学习研究存在诸多不足，例如对代码结构信息和语义信息利用不充分，特征表达不够完善；表征模型对数据集、模型结构和优化算法等多方面因素的要求高等，这些不足导致代码克隆检测效率较低。本文提出了一种多维源代码表征学习方法，旨在通过构建三个不同维度的代码表征模型，将源代码的语义信息表示为稠密低维实值向量，以在低维空间中高效计算实体和关系的语义联系，并通过特征融合得到多维特征，实现对代码信息的充分利用，以更加全面准确与智能化的方式提高代码克隆测试效率。

关键词：代码克隆检测；深度学习；代码表征学习

Abstract

Code cloning detection is an important task in the field of software engineering. How to learn representation of source code determines the degree of source code representation extraction, which in turn affects the accuracy that downstream tasks can detect. As the core technology and research hotspot of code cloning detection task, existing research on code representation learning has many shortcomings, such as insufficient utilization of code structure and semantic information, and incomplete feature expression; The representation model has high requirements for various factors such as dataset, model structure, and optimization algorithms, which leads to low efficiency in code cloning detection. This project proposes a multi-dimensional source code representation learning method, aiming to construct three different dimensional code representation models to represent the semantic information of source code as dense low dimensional real value vectors, efficiently calculate the semantic connections of entities and relationships in low dimensional space, and obtain multi-dimensional features through feature fusion to fully utilize code information and improve the efficiency of code cloning testing in a more comprehensive, accurate, and intelligent way.

Key Words: Code cloning detections; Code representation learning; Deep learning

目录

第 1 章 绪论	1
1.1 研究背景与意义	1
1.2 研究现状与趋势	3
1.2.1 代码克隆检测技术	3
1.2.2 代码表征学习	4
1.3 研究内容	10
1.4 论文结构	11
第 2 章 多维源代码表征学习方法总体设计	14
2.1 代码表征学习面临的技术挑战	14
2.2 RLCCD 研究方案	15
2.2.1 研究思路及总体框架	15
2.2.2 代码预处理	17
2.2.3 多维源代码表征学习	17
2.2.4 克隆检测任务实现	19
2.3 RLCCD 定义描述	19
2.4 本章小结	21
第 3 章 基于预训练辅助模型的 Token 表征学习	22
3.1 研究动机	22
3.2 Token 表征方法设计	23
3.2.1 框架概述	23
3.2.2 预训练辅助词嵌入设计	24
3.2.3 Token 代码表征设计	25
3.3 Token 表征方法具体实现	28
3.4 实验验证	30
3.4.1 实验环境	30
3.4.2 实验数据集	30

3.4.3 评估指标	32
3.4.4 实验结果	33
3.5 本章小结	34
第 4 章 基于子树划分的抽象语法树表征学习	35
4.1 研究动机	35
4.2 AST 表征方法设计	36
4.2.1 框架概述	36
4.2.2 子树划分设计	37
4.2.3 抽象语法树表征学习设计	39
4.3 AST 表征方法具体实现	43
4.4 实验验证	45
4.4.1 实验设计	45
4.4.2 实验结果	46
4.5 本章小结	46
第 5 章 基于图过滤的程序依赖图表征学习	47
5.1 研究动机	47
5.2 PDG 表征方法方法设计	48
5.2.1 框架概述	48
5.2.2 图过滤设计	49
5.2.3 程序依赖图表征学习设计	52
5.3 PDG 表征方法具体实现	55
5.4 实验验证	56
5.4.1 实验设计	56
5.4.2 实验结果	56
5.5 本章小结	57
第 6 章 特征融合及 RLCCD 框架验证	58
6.1 特征融合	58

6.2 RLCCD 框架验证	59
6.2.1 实验设置	59
6.2.2 对比工具	59
6.2.3 RLCCD 性能评估实验结果	60
6.3 本章小结	60
结论	61
参考文献	62
攻读学位期间发表论文与研究成果清单	66
致谢	67

插图

图 1.1 2018-2023 年 Synopsys 审计代码库中的开源代码及漏洞占比示意图	1
图 1.2 2022 年 Synopsys 审计代码库中包含易受攻击组件的百分比示意图	2
图 1.3 代码克隆检测流程	3
图 1.4 论文组织结构	12
图 2.1 研究思路	16
图 2.2 RLCCD 总体框架	16
图 2.3 C 语言代码片段示例	19
图 2.4 面向代码克隆检测的多维源代码表征学习方法 RLCCD 架构图	20
图 3.1 基于预训练辅助模型的 Token 表征学习框架	23
图 3.2 Token 代码表征：AttBiLSTM 模型设计	26
图 3.3 LSTM 模型时序结构图	26
图 3.4 BiLSTM 模型结构图	28
图 3.5 示例源代码对应的 Token 序列	28
图 3.6 基于预训练辅助模型的 Token 表征学习方法实现	29
图 4.1 基于子树划分的抽象语法树表征学习框架	36
图 4.2 子树划分算法示例	38
图 4.3 抽象语法树表征模型设计	39
图 4.4 子树表征：树卷积设计	40
图 4.5 连续二叉树	41
图 4.6 GRU 模型时序结构图	42
图 4.7 树表征：BiGRU 模型设计	43
图 4.8 示例源代码对应的抽象语法树	44
图 4.9 基于子树划分的抽象语法树表征学习方法实现	44
图 5.1 基于图过滤的程序依赖图表征学习框架	48
图 5.2 程序依赖图 DOT 文件示例	49
图 5.3 代码片段 1 经过图结构化简后的图	50
图 5.4 程序依赖图表征模型设计	52

图 5.5 有向图矩阵示例	53
图 5.6 图卷积神经网络结构图	54
图 5.7 示例源代码对应的程序依赖图	55
图 5.8 基于图过滤的程序依赖图表征学习方法实现	55
图 6.1 特征融合方法	58

表格

表 3.1 实验环境配置	30
表 3.2 POJ104 数据集	31
表 3.3 本文预处理后的 POJ104 数据集正负样本数	32
表 3.4 分类问题的混淆矩阵	32
表 3.5 预训练语料库规模对实验结果的影响	33
表 3.6 Token 表征模型对实验结果的影响 (%)	34
表 4.1 抽象语法树子树划分实验结果	46
表 5.1 图过滤机制实验结果	57
表 6.1 RLCCD 实验结果	60

第1章 绪论

1.1 研究背景与意义

代码克隆，也叫代码复用，是指在软件系统中存在两个或两个以上的相似代码片段^[1]，是软件开发中的常见现象。随着互联网时代的发展，网络上各种开源项目越来越多样化，获取也更加便利。许多企业通过软件资源库、外部开源软件、软件产品线及开发框架等方式建立了多种多样的软件复用开发方法，同时开发人员自身也会通过多种方式大量复用已有的软件资源。在这些软件复用方法和资源的支持下，软件系统和软件产品大量引入了开源软件、网络资源、商业软件等第三方代码成分。这些第三方代码在多个软件系统中复制、传播和演化，给软件系统带来了软件质量的不确定性和风险，甚至导致漏洞的传播^[2]。

近年来，第三方代码中包含的漏洞数量呈现出快速增长的趋势。根据美国新思科技公司（Synopsys, Inc.）发布的《2024 年开源安全和风险分析报告》^[3]显示，在 2023 年审计的 1067 个代码库中，96% 的项目都包含开源代码，总审计代码库的 77% 代码源自开源软件。其中，84% 的代码库包含至少一个已知开源漏洞，有 74% 代码库中包含高风险漏洞，比 2023 年版的报告中增加了 54%。图1.1统计了 2018 至 2023 年 Synopsys 审计代码库中开源代码及漏洞占比，从图1.1中可以看出开源代码及漏洞数量整体呈上升趋势。

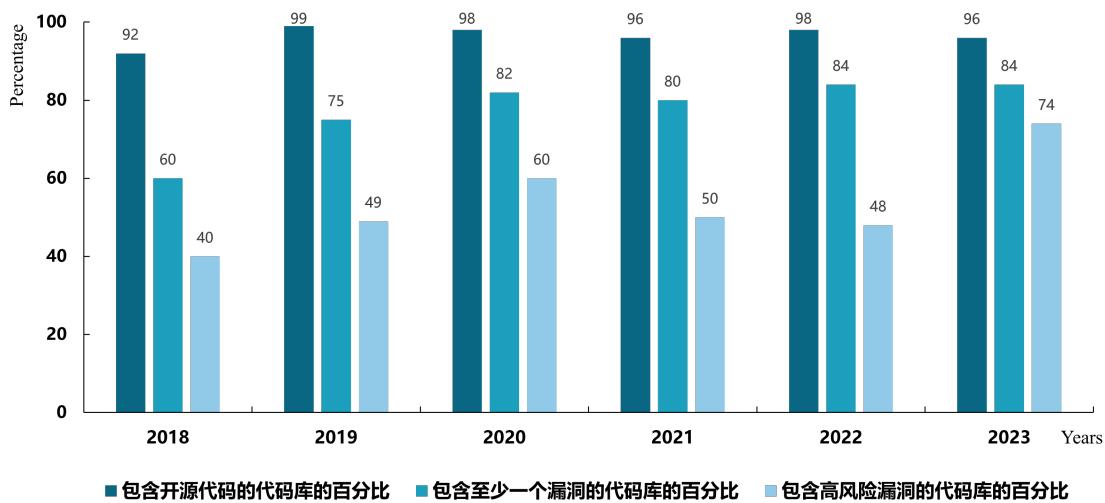


图 1.1 2018-2023 年 Synopsys 审计代码库中的开源代码及漏洞占比示意图

同时, Synopsys 按照行业划分, 统计了各行业包含开源代码的占比, 其中互联网等相关行业包含开源代码的代码库百分比均为 100%, 其余行业占比如图1.2所示。此外, 经过风险评估的代码库中, 14% 包含 10 年以上的漏洞, 漏洞的平均年龄为 2.8 年, 49% 包含 24 个月内未更新的组件。一旦这些组件出现安全问题, 通常会导致软件遭受供应链攻击。据 Gartner^[4]预测, 到 2025 年, 全球 45% 的组织将遭受软件供应链攻击, 比 2021 年增加三倍。因此, 准确地检测代码克隆对于软件开发和维护至关重要。

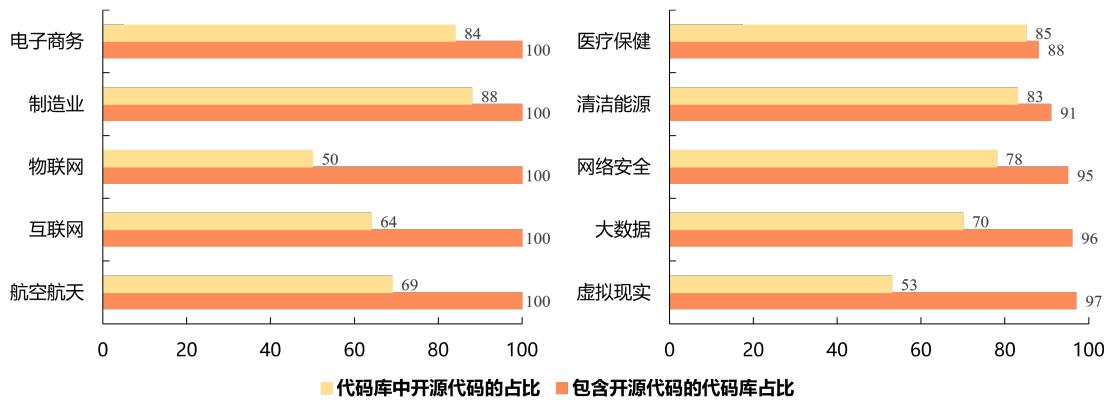


图 1.2 2022 年 Synopsys 审计代码库中包含易受攻击组件的百分比示意图

早期进行代码克隆检测通常采用人工检查并标注的方法, 通过收集整理大量的代码逐行检查语法、语义结构, 由人工复查筛选出正确的克隆代码并对其进行标注, 由此形成了早期的代码克隆数据样本, 例如, 2015 年 Svajlenko^[5]等三名克隆领域的专家评委花费了 216 小时通过人工验证方法构造了 BigcloneBench 数据集, 该数据集总数据量达到 800 万条, 其背后的人工花费巨大。这种利用人工的方法检测代码克隆效率低, 成本高, 并且无法保证准确率^[6], 因此, 有研究人员提出代码克隆检测技术, 目的在于自动化定位软件系统中的代码克隆, 并能够节约成本, 减少出错风险^[7]。

早期代码克隆检测技术通常将代码视为自然语言文本进行处理, 通过文本相似性判断代码相似程度; 随着编译技术的发展, 研究者们将编译原理中的词法分析技术运用到代码克隆检测领域; 近年来, 基于多维源代码表征学习的代码克隆检测技术引起了学者们广泛的兴趣, 有研究人员从代码克隆检测与代码表征学习技术相结合这一方面进行了探索, 试图从关键技术点入手, 找到合适的结合点, 以提高代码克隆检测技术的效率和智能化程度。

1.2 研究现状与趋势

1.2.1 代码克隆检测技术

代码克隆检测技术，旨在自动化定位软件系统中的代码克隆，节省成本，减少出错风险，从而更好地保证软件质量。目前已有的代码克隆方法大多需要对代码片段进行信息抽取，转换为中间表征，然后根据表征方式的不同，计算不同代码片段之间的相似度，完成克隆检测任务。其具体流程如图1.3所示。

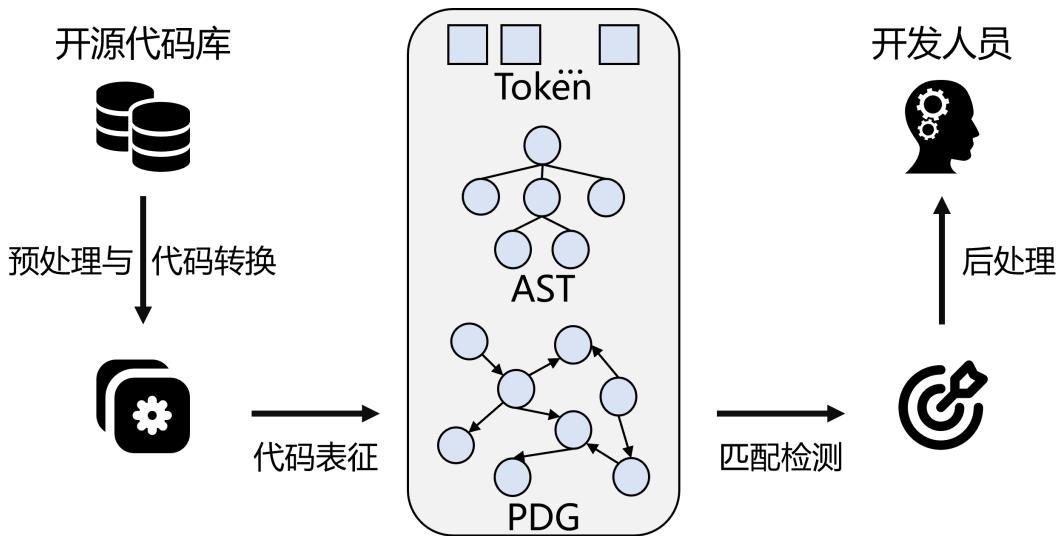


图 1.3 代码克隆检测流程

从图1.3可以看出一个完整的代码克隆检测过程通常包括预处理与转换、代码表征、匹配检测、后处理几个阶段。具体而言，一般的代码克隆检测从代码预处理与转换开始，首先删除与检测无关的空自行、注释、缩进等元素，并根据检测粒度将源代码划分为单独的片段，比如类、函数等；然后在代码表征步骤，将比较单元转换为词法单元（Token）、抽象语法树（abstract syntax tree, AST）、程序依赖图（Program dependency graph, PDG）等常见的中间表示；在匹配检测阶段，将根据得到不同的中间表示采用相应的匹配算法进行相似度计算，例如抽象语法树的比较通常采用子树匹配算法，程序依赖图的比较则采用子图同构算法。此阶段将代码片段两两对比，以查找相似代码源片段，得到代码克隆对。最后在后处理阶段，通常会通过人工检测或者算法过滤掉错误的代码克隆，并以适当的方式呈现给开发人员提供帮助。

在这些步骤中，代码表征方式决定了匹配检测方法的预处理方式、模型设计、部署方式、运行效率，并影响最终结果^[8]。比如将源代码表征为文本，其预处理过程主

要为去除噪声，如空格、注释等，其比较算法可以利用文本相似的一系列方法，能够检测到语法相似的克隆代码；而如果表征为抽象语法树，则其预处理过程需要解释器的参与，相似比较算法也更多地考虑了结构相似等，能够检测到语法层面相似的代码克隆。由此可见，代码表征方式是代码克隆检测的关键步骤。

1.2.2 代码表征学习

表征学习是指学习数据的表示，使其在构建分类器或其他预测因子时更容易提取有用信息^[9]。代码表征学习是对源代码的语义和语法信息进行表征，得到源代码的特征向量，并将其应用在不同的下游任务上。在代码克隆检测中，代码表征学习可以用来提取代码片段更高层次的抽象特征表示，这些特征表示能够捕捉代码的语法、语义以及结构信息。通过学习到的代码表征，可以更准确地比较和识别不同代码片段之间的相似性，从而实现克隆代码的检测和管理，提高检测的准确性和鲁棒性。因此，代码表征学习为代码克隆检测提供了重要的技术支持。

早在 100 多年前，基于传统机器学习的数据特征学习就被广泛提出^[10]，主成分分析 PCA(Principal Component Analysis)^[11]、线性判别分析 LDA (Linear Discriminant Analysis)^[12]都是经典的表征学习方法。随着神经网络的不断发展，基于深度学习的代码表征学习工作能够更有效地提取数据的特征，用于后续的分类或预测。2013 年，Y Bengio 等人^[9]发表了关于表征学习的经典综述。2016 年，Bengio 和 IGoodfellow 等人^[13]合著的《Deep Leanring》一书中为表征学习专著一章。近些年来，代码表征学习方法被用于代码克隆检测、代码推荐、代码剽窃等多个代码分析任务中，取得了一定的成就。根据源代码的抽象层次不同，现阶段代码表征学习工作可以分为基于 Token 的代码表征、基于树的代码表征、基于图的代码表征、基于语法和语义混合的代码表征四类。

(1) 基于 Token 的代码表征

基于 Token 的代码表征通常利用词法分析器将代码中的词汇单元（Token）划分出来。这些词汇单元通常包含关键字、数字、标识符等。将代码表示为词汇单元序列之后，对其进行建模，学习代码序列中所包含的有效信息，如功能语义信息、语法结构信息等，最后生成具有丰富代码信息的表征向量，应用于后续的代码克隆检测任务中。

著名的 CCFinder^[14]、CP-Miner^[15]等经典代码克隆检测工具都是基于 Token 级的，

可以很好地检测完全相同的代码对以及参数化后的代码对克隆问题。其中，CCFinder 将源代码中的每一行单独转换为 Token 序列，根据转化规则对 Token 进行修改，将类型名、变量名、常量的标识符替换为指定的特殊 Token，最后利用后缀树来查找相同的子序列并通过设置阈值来过滤克隆对。CP-Miner 增加了 Bug 检测，该工具的检测速度、检测精度相较于 CCFinder 有了很大的提高。

Jiang 等人^[16]首先使用神经网络在 Token 级别进行代码克隆检测，提出 CC Learner 方法。该方法使用 BigCloneBench^[5]作为训练样本，抽取了其中方法级别的 Token 序列，将保留字、类型标识符、方法标识符和变量标识符等 8 种符号表示为 8 种标记，随后将各种标记类型以及出现的频数作为代码的序列表示，并用于代码克隆检测。

Mikolov 等人^[17]利用 Word2vec、GloVe、BERT 进行 Token 的预训练，通过无标注样本训练深度网络结构，使用标注样本进行模型参数微调，从而提升模型性能。其中 BERT^[18]是双向 Transformer 的编码器，通过遮蔽语言模型和下一句预测 2 种预训练目标来调整模型参数。

Sajnani 等人^[19]提出了一种基于词袋模型的代码表征方法 SourcererCC，使用代码段 Token 的组成来度量两段代码块中词法粒度上的重复度，从而检测两段代码的相似性。这种方法相对于纯文本的代码表示形式实现了更高层次的代码分析。

Tao 等人^[20]提出了一种跨语言克隆检测模型 C4，通过对比学习的思想，采用预先训练的模型 CodeBert 提取代码特征，并转换为高维代码表示。此外，还通过一个可以有效识别克隆对和非克隆对的约束学习目标来微调 C4 模型。实验结果表明，C4 在准确率、召回率、F1 方面显著优于最先进的基线。

上述方法均在 Token 上进行代码的表征学习，力图充分提取代码中的属性信息。

(2) 基于树的代码表征

抽象语法树 AST 最早由 Yourdon 等人^[21]提出，是源代码的抽象语法结构的树状表示，可以有效地表示程序的语法及其结构，利用深度神经网络对抽象语法树进行建模得到其向量表示，根据该特征向量完成代码克隆检测任务，实现基于树的源代码表征。

White 等人^[22]提出了一种基于循环神经网络的代码表征方法，该方法将代码分为词汇以及句法两个层次。对于词汇级别的信息，该方法在代码的词汇单元序列上使用 RNN 神经网络进行建模。而对于代码的句法级别的信息，首先将代码转换为其对应的抽象语法树结构，之后将抽象语法树转换为其对应的满二叉树，最后将满二叉树转

换为橄榄树，并在其上使用另一个 RNN 神经网络进行建模。该方法将这两个特征相结合作为整个程序的特征向量，根据该向量进行代码克隆检测任务。

Mou 等人^[23]提出了一种基于树的卷积神经网络模型 TBCNN (Tree-based convolutional neural network)。该模型采用了“连续二叉树”的概念，直接在代码所对应的抽象语法树上进行卷积操作。在卷积操作之后获得了不同数目的 AST 结构特征向量，由于数目不同不能直接作为神经网络的输入，因此该方法还采用“动态池化”技术，最终将数目不同的特征向量转换为了一个向量。TBCNN 是一个通用的代码表征生成模型，所生成的向量能够包含代码片段中特有的代码模式，因而可以应用于不同的代码分析任务中。

Wei 等人^[24]提出了一种基于哈希的代码表征方法 CDLH(Clone Detection with Learning to Hash)。该方法使用 Word2Vec 模型学习标记嵌入以捕获词汇信息，然后训练基于抽象语法树的 LSTM 模型将这些嵌入组合成一个二进制向量来表示代码片段，最后通过计算哈希码的汉明距离来检测代码克隆。

Zhang 等人^[25]提出了一种基于抽象语法树的神经网络代码表征方法 ASTNN (A novel AST-based Neural Network)。该方法将完整的抽象语法树分割为多个语句子树。针对每个语句树，该方法设计了语句编码器用于将语句树转换为对应的语句表征向量，通过使用双向 GRU 神经网络对语句向量进行建模，对双向 GRU 层输出的隐含状态向量进行最大池化操作，以获得最显著的代码特征。该方法所生成的代码表征向量被应用于代码克隆检测任务中，在 POJ104^[23]和 BigcloneBench^[5]数据集上取得了当时最好的检测结果。

Yu 等人^[26]提出了一种基于树卷积的代码表征方法 TBCCD (Tree-Based Convolution for Clone Detection)。该方法提出了一种三角形卷积核对父节点和子节点卷积，通过自适应的参数编码树中的节点；同时考虑到抽象语法树的词法信息，通过位置相关的编码方式编码 Token 值，最后基于 CNN 进行克隆检测。

Ling 等人^[27]提出了一种树自动编码器架构 TAE。该方法使用无监督学习对大规模数据集的抽象语法树 AST 进行预训练，然后在下游任务代码克隆检测任务中对训练后的编码器进行微调，实现跨语言代码克隆检测。

Wu 等人^[28]提出了一种基于树的可扩展的语义代码克隆检测器 Amain，通过将抽象语法树转换为简单的马尔科夫链，并测量马尔科夫链中所有状态的距离，将距离值送入分类器中训练得到一个代码克隆检测器。

上述方法均在抽象语法树上进行代码的表征学习，力图充分提取代码中的结构信息。

(3) 基于图的代码表征

程序依赖图 PDG 最早由 Ferrante J 等人^[29]提出，是程序的一种图形表示，所含结构信息最多，能够表示程序的控制依赖，数据依赖以及地址依赖等关系，是一种带有标记的有向多重图。通过将程序表示为图的形式使得模型能够更好地理解代码中不同部分之间的依赖关系。

Allamanis 等人^[30]考虑到代码中的长依赖问题，如在代码中变量的定义位置与使用位置之间的距离问题，提出了基于图的代码表征方法，并介绍如何使用 GGNN (Gated Graph Neural Networks) 训练。首先将代码转换为对应的抽象语法树，之后通过不同的连接规则连接抽象语法树各个节点，获得了包含变量之间依赖关系在内的不同节点之间的关联关系；最后将构建好的代码图数据作为输入，输入到图神经网络中进行表征学习。

Lu 等人^[31]提出了一种用于程序分类的图网络模型 GGANN (Gated Graph Attention Neural Networks)。该方法从代码中提取数据流与函数调用信息，将其融合到抽象语法树中，从而将代码构建为一个包含丰富信息的图结构表示 FDA。在传统的 GGNN 模型上引入了注意力机制，用于获得图中每个节点的重要程度，进而获得更具有区分度的代码表征向量。

Brockschmidt 等人^[32]提出了一个生成代码模型，该模型利用部分生成程序的已知语义来指导生成过程。关键思想是在代码的抽象语法树上增加相应的边以构建代码图，之后使用图神经网络对已获得的部分程序的结构和数据流进行建模完成代码表征任务。这种表示有助于更好地指导生成过程的剩余部分。

Ben-Nun 等人^[33]提出了一种与语言以及平台无关的代码表征方法 inst2vec。该方法首先使用编译器对代码进行编译，得到代码的中间表示。但由于该中间表示并没有包含代码之中的数据流信息以及控制流信息，因此该方法将数据流和控制流也融合到该中间表示中，进而构建了代码上下文流图，最后在所构建的图上使用循环神经网络进行建模，获得代码的表征向量。该向量在程序分类实验中的准确率取得了当时最好的效果。

Wang 等人^[34]提出了一种称为流增强抽象语法树 FA-AST (Flow-Augmented Abstract Syntax Tree) 的程序图表示，考虑了仅仅使用代码的抽象语法树进行代码表征

建模实际上仍然有代码结构上的缺失这一问题，构建了代码抽象语法树的图形表示 FA-AST，通过将抽象语法树各个叶子结点相连构建出适合图神经网络处理的数据，然后应用两种不同类型的图神经网络 GNN 来检测克隆。

DeFreez 等人^[35]提出了一种学习嵌入方法 Fun2Vec，将每个函数映射到连续向量空间中的向量，以便使同义函数的向量非常接近。该方法采用随机游走算法，在程序的过程间控制流图上随机选择部分执行路径，捕获程序的层级结构，每条执行路径转换为一个标签序列，借助 Word2Vec 方法，把标签映射为连续实值向量，并通过神经网络训练函数的嵌入向量。

Kang 等人^[36]提出了一种针对代码补全问题的门控卷积网络模型 CC-CCNN。该方法通过从代码表示中获得有效的代码特征，提出了一种分类机制，通过使用已知的父节点对节点的表示进行分类，并在模型中构建训练图。实验结果表明，模型在数据集中最多优于最先进的方法 MRR 最多 9.2%，ACC 最多 11.4%。

上述方法均在图上进行代码的表征学习，力图充分提取代码中的语义信息。

(4) 基于语法和语义融合的代码表征

基于语法与语义融合的模型，结合抽象语法树 AST、数据流图 DFG、控制流图 CFG、词法单元 Token 序列，捕获程序的语法及语义结构信息。其中抽象语法树 AST 和 Token 序列反映了语法层面的信息，数据流图 DFG、控制流图 CFG 反映了语义层面的信息。

Tufano 等人^[37]采用四种不同的代码表征方法（即标识符 Token、抽象语法树 AST、字节码和控制流图 CFG）进行代码克隆检测，他们利用四种代码表示分别识别代码对的相似度，并计算平均值作为最终的相似度结果。

Saini 等人^[38]提出了一种基于多种度量的代码克隆检测框架 Oreo，该方法从程序的源代码中提取了包括被调用的外部方法的数量、变量的数量、语句的数量、循环的数量等 24 种度量，然后进一步从函数中抽取语义，并使用了基于哈希的方法进一步筛选，最后加入了深度学习的方法，将两个程序向量输入到孪生模型中来判断两个程序之间是否具有克隆关系。

Fang 等人^[39]结合抽象语法树，控制流图和调用图来学习代码特征，融合了语法和语义信息。首先，该方法从源码中分析出方法之间的调用图、每个方法的抽象语法树以及每个方法的控制流图；然后，用调用图找出每个功能的 AST 集合。通过 AST 集合抽取功能的语法信息；通过调用图组成每个功能的控制流图；通过控制流图抽取

功能的语义信息；最后，将抽取出来的语义信息送入前馈神经网络得到分类结果。

Hua 等人^[40]提出了一种使用注意力机制的功能代码克隆检测器 FCCA (Functional code clone detector using attention)，结合标记、抽象语法树和控制流图三种方式实现检测目标。该方法通过保留多个代码特征，包括非结构化（顺序令牌形式的代码）和结构化（抽象语法树和控制流图形式的代码）信息，在混合代码表示的基础上进行代码克隆检测。将多个代码特征融合到混合表示中，该混合表示配备有注意力机制，有助于最终检测精度的重要代码部分和特征。

Dong 等人^[41]提出了一种基于 Token 和 AST 的代码表征方式，提取数量特征如 AST 中的 AST 树的高度、节点数以及标记中操作数的个数、字符串的个数等作为神经网络的输入进行检测。

Feng 等^[42]提出了一种多模态的预训练模型，利用不同模态的信息互补作用，有效提升了模型的整体表征能力。CodeBERT 基于文档和代码，在自然语言和程序语言双模态下，利用 BERT 进行预训练，提取自然语言和程序语言之间的语义连接，为下游任务提供通用表示向量。

Duan 等人^[43]提出了一种无监督的程序代码表示学习技术 DEEPBINDIFF，依靠代码语义信息和程序控制流信息生成基本块嵌入，并且采用 k-HOP 贪婪匹配算法，利用基本块嵌入发现最优的相似性结果。通过大量二进制文件和真实的 OpenSSL 漏洞对原型进行评估，结果表明 DEEPBINDIFF 相比于最先进的工具，跨版本和交叉优化级别都更优。

Wu 等人^[44]提出了一种基于序列和基于图的软件功能克隆检测方法 SCDetector。针对给定的方法源代码，首先生成控制流图，然后应用社交网络中心性分析将图转换为某些语义标记（即具有图细节的标记）。最后，这些语义标记被馈送到 Siamese 网络中，以训练模型并使用它来检测代码克隆对。

总体而言，代码克隆检测是提高软件开发质量的重要手段，如何对代码进行合适的表征是代码克隆检测的关键问题^[45]。代码表征学习决定了对源代码信息抽取程度的上限，决定了检测技术的预处理方法、模型设计、部署方式、运行效率，并会影响最终结果。面向代码克隆检测这一下游任务，基于 Token 的方法将代码视为文本，保留了源代码本身存在的噪声，缺少代码的语义、结构信息；基于树的方法在一定程度上提取了代码的结构信息，但在转换 AST 的过程中会丢失部分信息，并且存在信息冗余；基于图的方法保留了语法和语义信息，但在生成图的过程中需要付出较高代价。

以上单个维度的代码表征方法各有利弊，但总体对代码信息利用不充分，因此，研究人员一方面通过对源代码进行充分利用，提出多维源代码表征方法，从而提高代码克隆检测能力；另一方面，通过研究更先进的算法来提高表征模型的适应性和泛化能力，也是目前重要的发展趋势。

1.3 研究内容

本文主要围绕如何将源代码表征学习技术应用到代码克隆检测领域的问题，通过不同维度对程序进行表征学习，并基于学习得到的语义特征进行克隆对的判定，充分发挥代码表征学习技术检测代码克隆的能力。针对现有代码表征学习方法存在的对代码结构信息和语义信息利用不充分的问题，本文提出面向代码克隆检测的多维源代码表征学习方法 RLCCD，旨在通过构建三个不同维度的代码表征模型，将源代码的语义信息表示为稠密低维实值向量，以在低维空间中高效计算实体和关系的语义联系，并通过特征融合得到多维特征，实现对代码信息的充分利用，以更加全面准确与智能化的方式提高代码克隆测试效率。

本文的主要工作包括：

(1) 提出面向代码克隆检测的多维源代码表征学习方法 RLCCD

本文提出了一种面向代码克隆检测的多维源代码表征学习方法 RLCCD，该框架主要针对代码表征提出三个关键技术点，从 Token 序列、抽象语法树 AST、程序依赖图 PDG 三种不同维度对代码特征表示进行优化，分别形成了基于预训练辅助模型的 Token 表征学习、基于子树划分的抽象语法树表征学习、基于图过滤的程序依赖图表征学习三种方法，然后通过特征融合将三种维度特征整合为一个多维特征，实现对代码信息的充分利用，以更加全面准确与智能化的方式提高代码克隆测试效率。

(2) 基于预训练辅助模型的 Token 表征学习

针对目前现有的基于 Token 的表征学习方法通常将代码表示为词汇单元，为了后续生成表征向量通常会将词汇单元规范化，丢失部分语法信息，出现在词汇表中不存在 Token 的难题，提出了一种基于预训练辅助模型的 Token 表征学习方法。该方法在模型训练之前，通过选取预训练辅助模型从代码语料库中学习基本单元的语法语义信息，以及这些单元之间的联系，最终给出一份单词-向量形式的词汇表，从而减少出现集外词问题的概率。在 POJ104 数据集上的消融实验评估表明，预训练辅助模型方法能够提高代码克隆检测准确率。

(3) 基于子树划分的抽象语法树表征学习

针对现有的基于树的表征学习方法通常将抽象语法树转换为完整二叉树，可能破坏源代码原有语法结构，增加 AST 高度，丢失长期上下文信息，削弱了神经网络模型捕捉更真实和复杂语义能力，导致梯度消失的难题，提出了一种基于子树划分的抽象语法树表征学习方法。该方法将每个大型的 AST 分割成小语句树序列，并通过捕获语句的词法和句法知识将每一个语句树都编码成一个向量，在得到一个语句向量序列后，将语句向量序列输入树卷积神经网络中生成代码片段的结构向量表示。在 POJ104 数据集上的消融实验评估表明，子树划分方法能够有效提取结构特征，提高代码克隆检测准确率。

(4) 基于图过滤的程序依赖图表征学习

针对现有的基于图的表征学习方法通常将程序表征为有向多重图，继而采用图匹配算法将图中的控制流和数据流编码为一个紧凑的语义特征矩阵，矩阵中每个元素都是高维系数特征向量，所消耗的时间、空间开销巨大的难题，提出了一种基于图过滤的程序依赖图表征学习方法。该方法通过收集 PDG 的简单特征来过滤掉明显不可能为克隆的 PDG 对。具体的，根据 PDG 的节点个数、控制边数、执行边数、数据边数、声明节点数、函数调用数、传入参数、传出参数等代表特征进行过滤，从而减少候选 PDG 对规模。在 POJ104 数据集上的消融实验评估表明，图过滤方法能够有效减少时间、空间开销，提高代码克隆检测准确率。

(5) 特征融合及实验验证

针对单个维度对代码信息利用不充分的问题，提出了基于多模态学习的特征融合方法，通过融合多个代码特征，包括非结构化(顺序 Token 形式的代码)和结构化(抽象语法树和程序依赖图形式的代码)信息，从多维数据中学到更好的特征表示。同时，选取了代码克隆检测领域常见的基准集 POJ104 进行实验验证，并与现有开源的 SourcererCC^[19]、ASTNN^[25]、SCDetector^[44]方法进行比较，主要通过精确度(Precision)、召回率(Recall)、F1 值三个指标评价实验结果，实验结果验证了 RLCCD 的可行性和有效性。

1.4 论文结构

本文总共分为六章，各章节的主要介绍内容如下，组织架构图如图1.4所示：

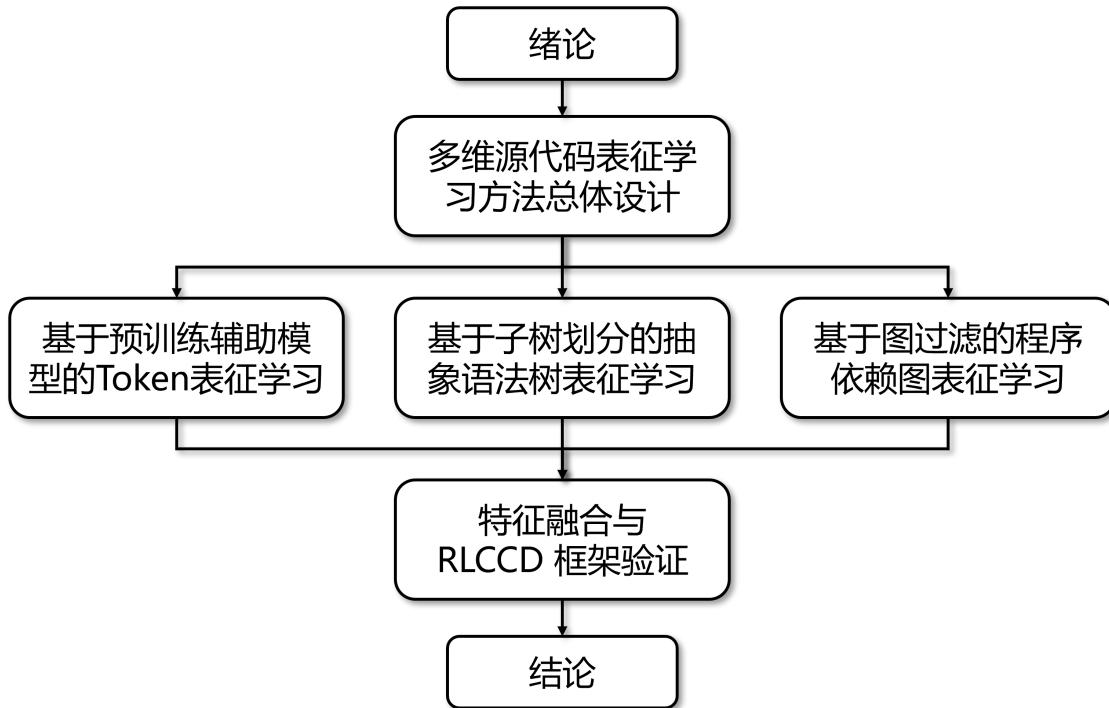


图 1.4 论文组织结构

第 1 章 绪论部分首先对本文的研究背景与意义进行了阐述，之后对代码克隆检测技术和代码表征学习技术的研究现状与趋势进行了分析总结，进而提出本文的主要研究内容，最后介绍了全文的组织结构。

第 2 章 分析了代码表征学习领域的关键技术挑战，基于此，提出了本文的面向代码克隆检测的多维源代码表征学习方法 RLCCD，并对该方法的整体框架进行了介绍，进而根据所提框架简要论述了本文研究的关键技术，即基于预训练辅助模型的 Token 表征学习方法、基于子树划分的抽象语法树表征学习方法、基于图过滤的程序依赖图表征学习方法，最后进行本章小结。

第 3 章 介绍基于预训练辅助模型的 Token 表征学习方法的设计与实现。首先，分析其研究动机，即目前 Token 表征学习面临的集外词问题，继而提出基于预训练辅助模型的方法设计，详细介绍该方法的设计思路和具体实现，最后，通过消融实验验证，评估本章提出的预训练辅助模型的有效性。

第 4 章 介绍基于子树划分的抽象语法树表征学习方法设计与实现。首先，分析其研究动机，即目前抽象语法树表征学习面临的梯度消失问题，继而提出基于子树划分的方法设计，详细介绍该方法的设计思路和具体实现，最后，对该方法的有效性进

行消融实验验证。

第5章 介绍基于图过滤的程序依赖图表征学习方法设计与实现。首先，分析其研究动机，即目前图表征学习面临的规模开销问题，继而提出基于图过滤机制的方法设计，详细介绍该方法的设计思路和具体实现，并给出了针对该方法有效性的消融实验验证。

第6章 介绍特征融合及本文研究框架 RLCCD 的实验验证。首先，针对特征融合的方法设计与具体实现进行了介绍。接着，对 RLCCD 框架的有效性进行了评估，通过与现有开源技术 SourcererCC^[19]、ASTNN^[25]、SCDetector^[44]进行实验对比，验证 RLCCD 方法的有效性。

结论 首先对全文的研究工作进行了总结，并讨论本文的主要贡献与创新之处，最后对下一步可开展的工作提出展望。

第 2 章 多维源代码表征学习方法总体设计

本章首先阐述了当前代码表征学习面临的一些关键技术挑战，进而针对现有的问题提出面向代码克隆检测的多维源代码表征方法 RLCCD。然后就该框架的总体架构和处理流程进行详细阐述，最后，给出了下游代码克隆检测任务的问题定义，针对 RLCCD 框架给出了形式化描述。

2.1 代码表征学习面临的技术挑战

目前已有的代码克隆检测方法大多遵循以下思路：(1) 首先对代码片段进行预处理；(2) 对处理好的代码片段进行代码表征，将其转换为中间表征；(3) 根据表征的方式不同计算不同代码片段之间的相似度，完成克隆检测任务。在代码克隆检测中，源代码表征方式决定了信息抽取的程度和粒度，进而影响了后续克隆检测的精度和效率。如何得到丰富且有效的源代码表征表示，是解决代码克隆检测任务的关键所在。从目前多种维度的代码表征方法来看，现有的代码表征方式存在以下技术挑战：

(1) Token 维度代码表征存在集外词问题

基于 Token 的方法一般会利用词法分析器将源代码中的词汇单元 Token 划分出来，得到 Token 序列，并过滤掉无用的空格、注释、字符等，然后利用深度学习技术对其进行建模，生成具有丰富代码信息的表征向量，应用于下游代码任务。这类方法和自然语言处理（NLP）领域中常用来处理文本的方式很相似，产生一个规模巨大且稀疏的词汇表。但是，在大多数基于 Token 的代码克隆检测工具中，通常会将词法单元规范化，例如：将变量名用统一的标识符来代替。经过规范化 Token 产生的词汇表较小，导致模型学习能力有限，并且在训练过程中会出现未见过或未包含在词汇表中的词语。这些词语可能是用户自定义词、拼写错误、缩写、专有名词等。由于模型在训练阶段没有足够的信息来学习这些词语的表示，因此在实际应用中无法正确处理这些词语，从而导致模型的性能下降，这就是集外词（Out of vocabulary，简称 OOV）问题。集外词问题会对模型的性能和泛化能力造成影响，严重限制了代码表征的有限性。

(2) 树维度代码表征存在梯度消失问题

基于树的方法将代码通过语法解析转换成相应的抽象语法树，从而有效地表示代码的语法及其结构信息。与自然语言处理领域的长文本类似，当上下文序列很长的时候，基于树的神经网络模型容易出现梯度消失的问题，即梯度在训练过程中变得越来

越小，特别是当树非常深的时候，模型会面临梯度消失问题。目前大多数基于树的代码克隆检测方法为了简化或者提高效率，通常会将生成的抽象语法树转换为完整的二叉树，在转换的过程中，不仅破坏了源代码原有的语法结构，也会增加树的高度，进一步削弱模型捕捉复杂语义的能力，导致检测性能下降。

(3) 图维度代码表征存在规模开销问题

基于图的方法会将源代码表征为数据流图或者控制流图，数据流图代表了源代码中数据的走向，控制流图代表了代码中语句执行时的跳转流向。大多数基于图的代码克隆检测工具任务的核心是将图中的每个节点映射到一个低维、稠密的特征向量中，并将这些特征编码为特征矩阵，这一步通常需要大量空间开销。同时子图匹配算法是 NP 完全问题，计算成本过长，时间复杂度很高，因此图维度代码表征学习会存在算法计算开销大，可扩展性不好，检测结果召回率低等问题。

(4) 代码表征存在信息利用不充分问题

虽然目前代码表征在 Token、树、图等多种维度的研究已经取得了一定的进展，但还存在信息利用不充分的问题。代码不仅仅具有文本自然性，同时具有结构信息、语义信息。在现有的表征粒度中，Token 维度的代码表征通常只关注文本自然性，抽象语法树可以捕获程序的语法结构和模式，程序依赖图可以表达程序的部分语义信息，只使用单个特征来表示代码是远远不够的，很难覆盖所有信息，因此存在信息利用不充分，特征表达不完善的问题。

2.2 RLCCD 研究方案

2.2.1 研究思路及总体框架

针对2.1节提出的四个技术挑战，本文提出了如图2.1所示的研究思路。

具体地，本文主要针对代码表征学习的三个维度展开研究工作：(1) 针对 Token 序列特征挖掘，提出预训练增强辅助模型提取属性特征，从而解决传统基于 Token 序列的方法存在的集外词问题；(2) 针对抽象语法树 AST 特征挖掘，提出子树划分的改进方法提取结构特征，从而解决传统基于抽象语法树的方法存在的梯度消失问题；(3) 针对程序依赖图 PDG 特征挖掘，提出过滤机制提取语义特征，通过收集 PDG 的简单特征来过滤输入神经网络模型的输入，从而解决传统基于程序依赖图的方法存在的规模开销问题。

同时，考虑到经由不同表征方式处理所得到的信息通常具有互补性，且不同维度

的特征都是代码表示的平行语料，具有信息等价性，因此，本文提出基于多模态学习的特征融合方法，通过融合多个代码特征，包括非结构化（顺序 Token 形式的代码）和结构化（抽象语法树和程序依赖图形式的代码）信息，从多维数据中学到更好的特征表示，从而有利于提高下游代码克隆检测任务的检测精度。



图 2.1 研究思路

本文基于上述研究思路，设计了面向代码克隆检测的多维源代码表征方法 RL-CCD，框架如图2.2所示。由图2.2可见，本文提出的基本框架与1.2.1节提出的代码克隆检测的处理流程基本一致，并主要通过三个维度对代码表征学习环节进行改进，然后对三个维度得到的特征向量进行特征融合，得到的混合特征应用到下游代码克隆检测任务中。

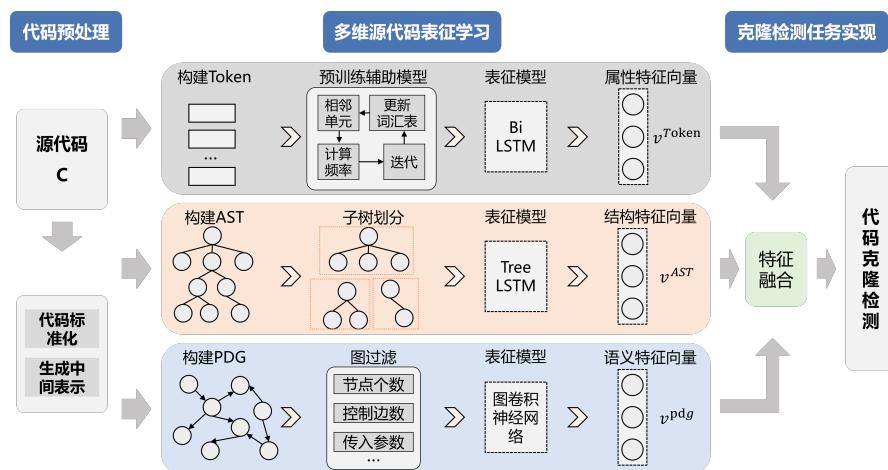


图 2.2 RLCCD 总体框架

2.2.2 代码预处理

代码预处理的目标是生成源代码片段对应的词法单元 Token 序列、抽象语法树 AST 和程序依赖图 PDG，主要包含 2 个流程：代码标准化、生成中间表示。

(1) 代码标准化

代码标准化的任务是去除与源代码无关的信息。首先是删除源代码片段中的注释、空行以及特殊符号，包括单行注释、多行注释、引入标准库的宏符号“#”、无关符号等。其次，由于代码本身是一段包含丰富信息的文本，开发者通常会通过个人命名习惯对常量等标识符进行命名，这些私人信息并没有大多实际意义，反而会降低后续代码处理的精确率。因此，本文定义了一定的转换规则，将代码中的某些标识符转换为对应的标记，在最大限度地保留原有重要信息的同时，减少缺失关键语义联系。

(2) 生成中间表示

基于标准化后的代码片段生成对应的中间表示：Token 序列、抽象语法树 AST 和程序依赖图 PDG。其中，词法单元 Token 序列可以通过词法分析器得到，词法分析器能够按照预定的语法规则将代码中的字符串分割为一个个词汇单元，这些词汇单元包含代码标准化处理后的标识符；抽象语法树 AST 以树状的形式抽象描述了程序语句的语法结构信息，生成抽象语法树时需要对源代码文本进行词法分析，然后依据语法规则分析整合 Token，得到树型结构；程序依赖图 PDG 能够表示源代码的控制依赖，数据依赖等关系，是一种带有标记的有向多重图。生成程序依赖图时，需要对程序进行语法分析，然后分析程序中变量的关联关系，根据这种关联关系描绘程序的数据依赖和控制依赖关系，形成图的描述。

2.2.3 多维源代码表征学习

RLCCD 框架的核心步骤是源代码表征学习，其目标是学习能够表示代码片段的连续向量，表现程序理解的认知层次，获取程序的语法、语义信息，创建程序更高抽象层次上的表示，它决定着对源代码信息抽取程度的上限，决定着检测方法的预处理方式、模型设计、部署方式、运行效率，并影响后续代码克隆检测任务所能检测的精度。本文提出的多维源代码表征学习方法包括 Token 序列、抽象语法树 AST、程序依赖图 PDG 三种不同维度。下面详细介绍研究方案并分析其优化改进。

(1) 词法单元 Token

传统的基于 Token 的表征学习方法通常将代码表示为词汇单元，为了后续生成表

征向量通常会将词汇单元规范化，丢失部分语法信息，出现在词汇表中不存在 Token 的难题。针对这样的问题，本文提出了一种预训练辅助模型的改进方法提取属性特征。具体来说，通过选取预训练辅助模型从代码语料库中学习基本单元的语法语义信息，以及这些单元之间的联系，最终给出一份单词-向量形式的词汇表，从而减少出现集外词问题的概率。

(2) 抽象语法树 AST

抽象语法树中包含了代码片段的结构信息，然而，传统的基于树的代码表征学习方法通常将抽象语法树转换为完整二叉树，可能破坏源代码原有的语法结构，增加 AST 高度，丢失长期上下文信息，削弱了神经网络模型捕捉更真实和复杂语义的能力，导致梯度消失的难题。针对这样的问题，本文提出子树划分的改进方法提取结构特征。具体来说，将每个大型的抽象语法树分解为小语句树序列，并通过捕获语句的词法和句法知识将每一个语句树都编码成一个向量。在得到一个语句向量序列后，将语句向量序列输入网络中生成代码片段的结构向量表示。这种细粒度的处理使得模型可以很好处理很深的抽象语法树，解决梯度消失问题。

(3) 程序依赖图 PDG

程序依赖图中包含代码片段的控制依赖，数据依赖等语义关系，然而，传统的基于图的代码表征学习方法通常采用图匹配算法将图中的控制流和数据流编码为一个紧凑的语义特征矩阵，矩阵中每个元素都是高维系数特征向量，面临消耗的时间、空间开销巨大的难题。针对这样的问题，本文提出了图过滤机制的改进方法提取语义特征。具体来说，根据 PDG 的节点个数、控制边数、执行边数、数据边数、声明节点数、函数调用数、传入参数、传出参数等代表特征进行过滤，减少模型的输入规模。

(4) 特征融合方法

特征融合的目标是将提取到的属性特性、结构特征、语义特征合并，得到一个更能代表代码信息的多维特征，更具有判别能力。具体来说，通过三个不同维度得到属性特性、结构特征、语义特征，然后将三个特征映射到相同的特征空间内，最简单的例子是对多个代码特征进行串联，除此之外，还可以使用神经网络、概率图模型等融合方法，将向量融合作为一个混合多维表征，包括非结构化(顺序 Token 形式的代码)和结构化(抽象语法树和程序依赖图形式的代码)信息。该多维特征能够在低维空间中高效计算实体和关系的语义联系，挖掘代码节点间更全面、层次更深的关系信息，从而提高后续下游的代码克隆检测任务的准确率。

2.2.4 克隆检测任务实现

克隆检测任务实现的核心任务是判断两个代码片段是否是真克隆对。经过2.2.2节代码预处理和2.2.3节多维源代码表征学习两个步骤，可以得到对应的混合特征向量表示作为输入，然后计算这两个向量之间的相似性判断是否存在代码克隆。目前，常见的计算向量相似性的方法包括计算距离度量、相似性度量两种，向量的距离越近相似度越大。一些研究倾向于把程序表征为向量形式，使用余弦相似度、Jaccard 相似度、欧几里得距离、汉明距离、曼哈顿距离等评估指标计算向量之间的相似度，当相似度大于某个固定阈值时则认为存在代码克隆。具体的，本文通过输入混合特征训练分类模型，以最小化模型损失为目标，完成代码克隆检测任务。

2.3 RLCCD 定义描述

为了更形象地描述代码克隆检测问题，本节首先给出示例源代码片段2.3，其中，图2.3(a)、图2.3(b)、图2.3(c)是三个简单的代码片段，函数的主要功能为：计算 *Data* 数组的元素之和，其中图2.3(a)的第 10-13 行采用 for 循环，图2.3(b)的第 9-13 行采用 while 循环，这两个代码片段属于真克隆对；图2.3(b)的第 9-14 行采用 for 循环，但在内部需要判断 *result* 的现有状态，因此图2.3(a)和图2.3(c)的代码片段属于假克隆对。

```

1  /* Count the sum of elements in an array */
2  #include <stdio.h>
3
4  int main()
5  {
6      int i = 0;//parameter one
7      int data[5]={1,2,3,4,5};//parameter two
8      int result = 0;//parameter three
9
10     for(;i<5;i++)
11     {
12         result += data[i];
13     }
14     printf("%d",result);
15     return 0;
16 }

1  /* Count the sum of elements in an array */
2  #include <stdio.h>
3
4  int main()
5  {
6      int i = 0;//parameter one
7      int data[5]={1,2,3,4,5};//parameter two
8      int result = 0;//parameter three
9
10     while(i<5)
11     {
12         result += data[i];
13         i++;
14     }
15     printf("%d",result);
16     return 0;
17 }

1  /* Count the sum of elements in an array */
2  #include <stdio.h>
3
4  int main()
5  {
6      int i = 0;//parameter one
7      int data[5]={1,2,3,4,5};//parameter two
8      int result = 0;//parameter three
9
10     for(;i<5;i++)
11     {
12         if(result < 10){
13             result += data[i];
14         }
15     }
16     printf("%d",result);
17     return 0;
18 }

```

(a) C 语言代码片段 1 (b) C 语言代码片段 2 (c) C 语言代码片段 3

图 2.3 C 语言代码片段示例

给定两个代码片段 C_a, C_b ，使用一个三元组 (C_a, C_b, y_{ab}) 的形式来表示一对代码片段，其中 y_{ab} 表示标签。如果 (C_a, C_b) 是一个克隆对，那么 y_{ab} 为 1，否则 y_{ab} 为 0。从 n 个代码片段中构建一个带有标签的训练集 $\{(C_a, C_b, y_{ab})|a, b \in n, a < b\}$ ，本文的

目的是训练一个深度学习模型来学习一个可以把代码 C 映射成一个特征向量 V 的函数 f 。对于任意代码片段，计算出他们的相似度分数 $S_{ab} = f(C_a, C_b)$ 并进行分类，使其分类结果尽可能接近已知的标签 y_{ab} 。在预测阶段，为了推断出两个代码片段是否是克隆对，在真假克隆对之间设置了一个阈值 k ，如果预测的相似度分数大于 k 值，那么认为两个代码片段是真克隆对，否则，认为它们是假克隆对。其中，训练一个深度学习模型将代码 C 映射成一个特征向量 V 的过程，可以视为代码表征学习的数值化过程，也是本文研究的重点。

面向代码克隆检测的多维源代码表征学习方法 RLCCD 架构如图2.4所示，采用 Siamese 架构。RLCCD 方法的输入是两个代码片段 C_a, C_b ，输出是一个 0 或 1 的标签，1 表示这两个代码片段是真克隆对，0 表示是假克隆对。Siamese 架构采用两个不同的输入，通过两个具有共享权值的相似子网络，输出一个编码来计算两个输入之间的相似度。整个方法的执行包括四个步骤，分别是：

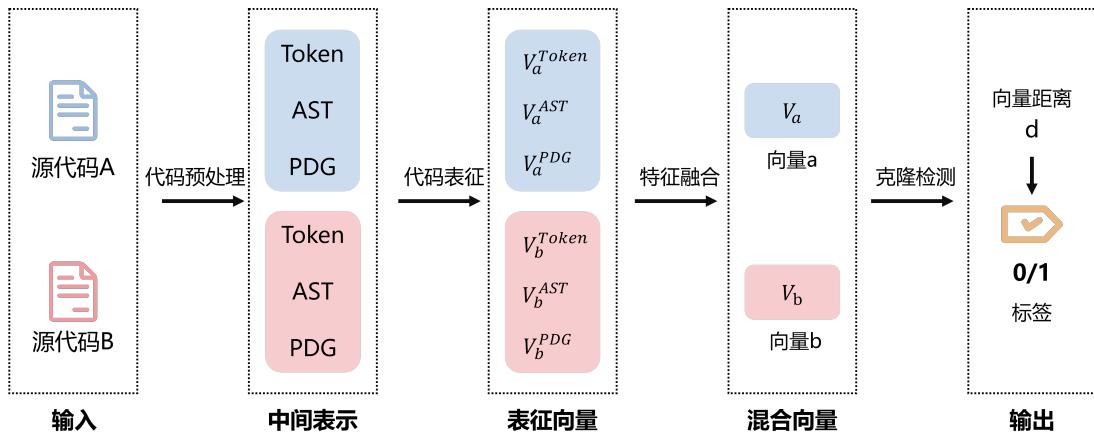


图 2.4 面向代码克隆检测的多维源代码表征学习方法 RLCCD 架构图

代码预处理： RLCCD 接受两个代码片段 C_a, C_b 作为输入，通过代码预处理阶段，得到对应的中间表示：词法单元 Token 序列、抽象语法树 AST、程序依赖图 PDG。

代码表征： 代码表征是本文的核心步骤，输入是代码片段 C_a, C_b 对应的中间表示，输出是对应的表征向量，分别记为代码片段 C_a 的属性特征 V_a^{Token} 、结构特征 V_a^{AST} 、语义特征 V_a^{PDG} ，代码片段 C_b 的属性特征 V_b^{Token} 、结构特征 V_b^{AST} 、语义特征 V_b^{PDG} 。

特征融合： 特征融合阶段的输入是代码片段 C_a, C_b 的特征向量，输出是对应的混合向量 V_a, V_b 。本文将提取到的属性特性、结构特征、语义特征合并，得到一个更能代表代码信息的多维特征。我们用公式2.1表示特征融合。

$$V = W_{dt} [v^{\text{Token}} \bullet v^{\text{AST}} \bullet v^{\text{PDG}}] + b_{dt} \quad (2.1)$$

其中 \bullet 表示特征融合方法， V 表示最终的多维混合代码表示。 W_{dt} 在生成的多维混合表示中平衡属性特征 V^{Token} 、结构特征 V^{AST} 、语义特征 V^{PDG} 的组成，而 b_{dt} 在训练模型时使模型偏向最终收敛。

克隆检测： 经过三个步骤，可以得到代码片段对应的向量表示 V_a, V_b 。由于代码克隆检测问题是一个二分类问题，即给定两个代码片段，需要输出 0 或 1，0 表示它们之间不相似，1 表示相似。因此通过公式2.2计算代码片段 C_a 对应的多维表征 V_a 与代码片段 C_b 对应的多维表征 V_b 之间的距离 d ，并将向量距离 d 映射到 0~1 之间，将输出值作为两个代码片段的相似度 S_{ab} 。

$$\begin{aligned} d &= |V_a - V_b| \\ S_{ab} &= \text{sigmoid}(d) \in [0, 1] \end{aligned} \quad (2.2)$$

并将损失函数定义为二元交叉熵，如公式2.3所示，训练模型的目标是最小化损失。

$$J(\Theta, S_{ab}, y_{ab}) = \sum (-(y_{ab} \cdot \log(S_{ab}) + (1 - y_{ab}) \cdot \log(1 - S_{ab}))) \quad (2.3)$$

其中， y_{ab} 表示两个代码片段的真实标签。

当所有参数都设置为最优化后，模型被存储起来。在预测阶段，通过公式2.4得到预测值。

$$\text{prediction} = \begin{cases} 1, & S_{ab} > k \\ 0, & S_{ab} \leq k \end{cases}, \quad (2.4)$$

2.4 本章小结

本章首先给出了代码克隆检测问题的定义，然后分析了源代码表征学习在代码克隆检测过程中所面临的关键技术挑战，主要表现为 Token 集外词问题、树梯度消失问题、图规模开销、单个表征维度对代码信息利用率低问题。针对四个问题，本文提出了面向代码克隆检测的多维源代码表征学习方法 RLCCD，在介绍了其整体框架后，对其中的关键技术点进行了简要的论述，最后详细描述了 RLCCD 的处理流程。

第3章 基于预训练辅助模型的 Token 表征学习

本章主要对本文提出的基于预训练辅助模型的 Token 表征学习方法进行详细介绍，首先介绍其研究动机，接着阐述其方法设计，以及具体的实现过程，最后介绍实验验证过程和结果。

3.1 研究动机

基于 Token 的代码表征方法本质上就是将源代码转换为一系列词法单元 Token 组成的序列，并对 Token 序列进行代码分析。基于 Token 的代码表征方法存在两种主要限制：

(1) 集外词问题：类似于自然语言技术处理文本，由于源代码中存在大量用户自己定义的标识符，不同用户的命名习惯不同，在对源代码的词法单元建模时，会产生一个规模巨大且稀疏的词汇表。该词汇表的规模会直接影响代码分析任务的效率，因此现有方法大多都对 Token 进行规范化，比如将变量名用统一的标识符来代替，从而降低词汇表的规模。但是后续神经网络模型训练过程中，当出现某个词汇在词汇表中没有出现过，那么神经网络模型就无法对齐建模，即出现了在词汇表中不存在的 Token，集外词（Out-of-vocabulary，简称 OOV）问题。有研究^[46]发现，针对代码表征中的集外词问题，在经典 BigcloneBench^[5]数据集中 OOV 比率高达 62.68%，在 OJClone 数据集^[23]中 OOV 比率达到了 16.82%。

(2) 预训练辅助模型训练代价大：为了解决 OOV 问题，本文打算使用预训练模型增加词汇表的大小。近期，研究人员在大规模语料库上预训练各种语言模型，在解决各种自然语言处理任务方面取得了良好进展^[47]。在表征学习领域，也有基于大规模预训练模型提升代码表征能力的方法被提出。InferCode^[48]将自然语言处理中的自监督学习思想引入到代码的抽象语法树的表示中，通过预测从 AST 上下文中自动识别的子树来训练代码表征，并且使用 AST 的子树作为训练标签，从而无需任何人工标记工作。该预训练 InferCode 模型可以应用于下游的无监督学习，例如代码聚类、代码克隆检测、跨语言代码搜索等。GraphCodeBERT^[49]方法提出了一个基于数据流的代码表征预训练模型，与抽象语法树不同，数据流包含代码变量间“值从哪里来”的语义特征，并不会带来深层次不必要的复杂信息，使用该特征可以更有效的生成代码表征。然而这些预训练模型通常存在参数规模庞大，训练及使用代价大的问题。

因此，针对集外词 OOV 问题，本文提出了一种轻量级的基于预训练辅助模型的 Token 表征学习方法，该方法在提升代码表征能力的同时，并不会引入过多参数，造成模型训练代价高的问题。

3.2 Token 表征方法设计

本节将主要介绍基于预训练辅助模型的 Token 表征学习方法的详细设计，首先介绍该方法的整体框架，并分别从预训练辅助词嵌入、Token 代码表征两方面介绍具体设计。

3.2.1 框架概述

本文提出的基于预训练辅助模型的 Token 表征学习方法整体框架如图3.1所示。该框架的输入是代码片段对应的 Token 序列，输出是对应的属性特征向量，主要包括预训练辅助词嵌入、Token 代码表征两个阶段。

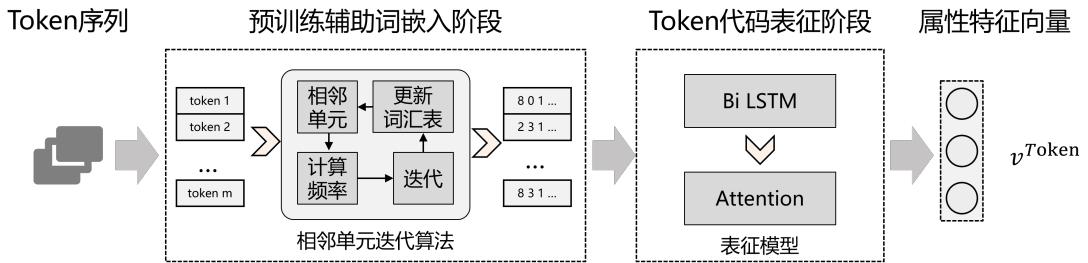


图 3.1 基于预训练辅助模型的 Token 表征学习框架

首先，预训练辅助词嵌入阶段以代码片段的 Token 序列作为训练数据，构建一个预训练辅助词嵌入模型。该模型的输入是代码片段对应的 Token 序列，输出为与输入对应的 Token 词嵌入向量。然后通过对模型的训练，使得该模型具有正确识别 Token 的能力，并将该模型保存下来。需要注意的是，为了减少集外词 OOV 问题，本文对模型采用相邻单元迭代算法，即，通过多次迭代来更新词汇表。

其次，Token 代码表征阶段以 Token 词嵌入向量作为训练数据，构建一个 Token 表征模型。该模型的输入是 Token 序列对应的词向量，输出为一个固定长度的密集向量用来表示代码的属性特征。需要注意的是，本文选用的 AttBiLSTM 神经网络，主要包含两个部分：双向长短时记忆部分 (BiLSTM) 和自注意力机制部分 (Attention)，前者主要目的是同时捕获序列的双向语义信息，后者的主要目的是总结序列的输入特征，并将每个代码片段缩减为一个单一的密集向量。

在上述框架中，本文的创新点主要体现在预训练辅助词嵌入阶段的相邻单元迭代算法、Token 代码表征阶段的模型设计两方面，下面将围绕这两个创新点来阐述本文的方法。

3.2.2 预训练辅助词嵌入设计

与自然语言处理类似，源代码的词法单元 Token 可以视为句子中的单词，Token 维度的代码表征方法第一步就是将词法单元 Token 转换为机器易于理解和操作的数值向量，即词嵌入技术。它将词汇表中的每个单词转换成一个低维度、连续的向量表示，这种向量通常被称为词向量。目前，使用预训练词嵌入模型训练词向量的研究工作发展迅速，Word2vec、GloVe、FastText、ELMo 和 BERT 等模型也相继被提出。其中，Word2vec 模型通过从大规模无标注文本数据集中学习上下文信息，自动捕捉并编码词汇的语义关系，生成的词嵌入向量作为下游任务的输入特征，可以显著提升神经网络模型的性能和泛化能力。因此，本文采用 Word2vec 模型进行词嵌入。

同时，为了减少集外词 OOV 问题出现的概率，本文提出了一种相邻单元迭代算法（Adjacent token iterative algorithm，简称 ATIA），该算法通过组合 Token 序列中相邻单元构造新的代码表示单元，然后统计新单元出现的频率，根据频率信息多次迭代，不断更新词汇表。下面对相邻单元迭代算法进行介绍。

具体地，首先将代码数据集中的源代码经过代码预处理，得到对应的 Token 序列作为初始语料库，代码预处理的工作详见2.2.2节，这里不展开阐述。然后按照代码块组合相邻 Token 并查找出出现次数最频繁的 Token 组合，将这个组合称为新的迭代单元（Adjacent gram，简称 Agram）。最后将这个 Agram 当作新的独立单元，加入到词汇表中，而原本组成这个 Agram 的原始 Token 单元并不会从词汇表中删除。在第一次迭代的时候，选取相邻的两两 Token 作为组合，把出现最频繁的 Token 组合确定为一个 Agram 单元之后再进行二次迭代，每次迭代在原来的基本单元上再组合一个新的邻近 Token 作为新的判断单元，不断的进行迭代，每次迭代都要对应的更新词汇表。

相邻单元迭代算法 ATIA 的伪代码如1所示。该算法有两个输入：待处理的初始 Token 语料库 *Corpus*、迭代次数 *N*，有两个输出：经过多次迭代后生成的语料库 *NewCorpus*、词汇表 *Vocab*，具体包括初始化 *AgramsSet*、计算单元 *Agram* 出现的次数、更新语料库 3 个步骤，每个步骤的作用如下：

(1) 初始化：首先初始化一个 *AgramsSet* 用来存放 Token 组合，初始化一个词汇

Algorithm 1 Iterative algorithm

Input: Origin Tokens Corpus:*Corpus*
Input: The number of iterations:*N*
Output: New Corpus: *New_Corpus*
Output: Vocabulary:*Vocab*

```

1: AgramsSet =  $\emptyset$ 
2: New_Corpus  $\leftarrow$  Corpus
3: Vocab = { }
4: for i  $\leftarrow$  1 to N do
5:   if i == 1 then
6:     for token in Corpus do
7:       add adjacent token to AgramsSet
8:     end for
9:   else
10:    numbers[] = {}
11:    for Agram in Agrams do
12:      Calculate the number of Agram in Corpus
13:      add number to numbers            $\triangleright$  step2: 计算单元 Agram 出现的次数
14:    end for
15:    n  $\leftarrow$  Max(numbers)
16:    Max_Agram  $\leftarrow$  Agram in Agrams whose number is n
17:    add (Max_Agram : n) to Vocab            $\triangleright$  step3: 更新词汇表
18:    add Agram to New_Corpus            $\triangleright$  step3: 更新语料库
19:  end if
20: end for
21: return New_Corpus, Vocab

```

表用来存放 Token 及其出现次数，并且将目前已有的 Token 语料库 *Corpus* 添加进 *NewCorpus*。

(2) 计算单元 Agram 出现的次数：多次迭代，在第一次迭代时，组合相邻 Token，得到 Agram 组合，并添加进 *AgramsSet* 集合中。遍历语料库，计算 *AgramsSet* 中所有的 Agram 出现的次数，并且把 Agram 组合当作 key，组合出现的次数当作对应 key 的 value 值保存到词汇表。

(3) 更新：遍历 *AramsSet* 中所有的 Agram 的 value 值，将频率最高的 Agram 合并到新语料库中。然后重复 (2)、(3) 步骤，直到迭代次数 N 减少为 0。

3.2.3 Token 代码表征设计

(1) 结构设计

为了提高 Token 维度代码表征能力，本文选用 AttBiLSTM 神经网络对上述得到

词向量进行建模，具体的模型设计如图3.2所示。该模型主要包括输入层、双向长短时记忆层（BiLSTM）和自注意力机制层（Attention）、输出层。

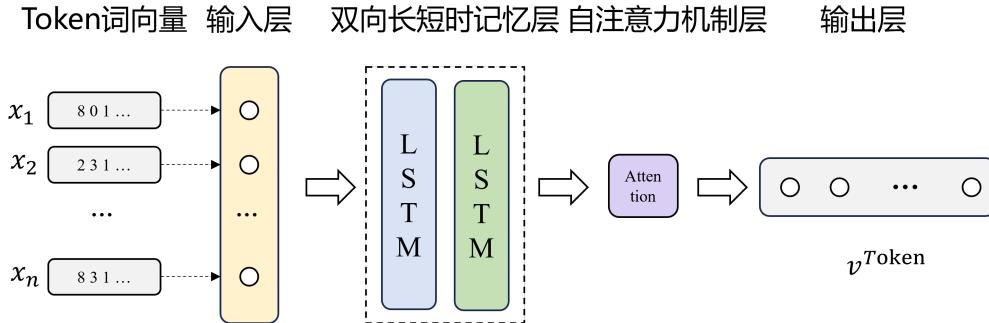


图 3.2 Token 代码表征：AttBiLSTM 模型设计

①输入层：输入层用于向模型输入训练数据，在本方法中模型的输入为经过预训练辅助嵌入模型得到的 Token 序列词向量，即每个词向量对应一个输入神经元。②双向长短时记忆层：由两层 LSTM 构成，同时捕获序列的双向语义信息。③自注意力机制层：本层的主要目的是总结序列的输入特征，并将每个代码片段缩减为一个单一的密集向量。④输出层：每个 Token 序列对应一个输出。

（2）模型选型

考虑到代码的 Token 序列是一种序列化的输入，并且前后词法单元可能存在依赖关系，目前常见的学习序列中长依赖关系的递归神经网络模型为长短期记忆网络 LSTM，其时序结构如下图3.3所示。每一个 LSTM 单元都有一个细胞状态和三个门（输入、输出、遗忘），通过三个门调节信息的流动，控制输入、自身状态、输出所占的比重，使网络能够在长时间内保留或遗忘信息。

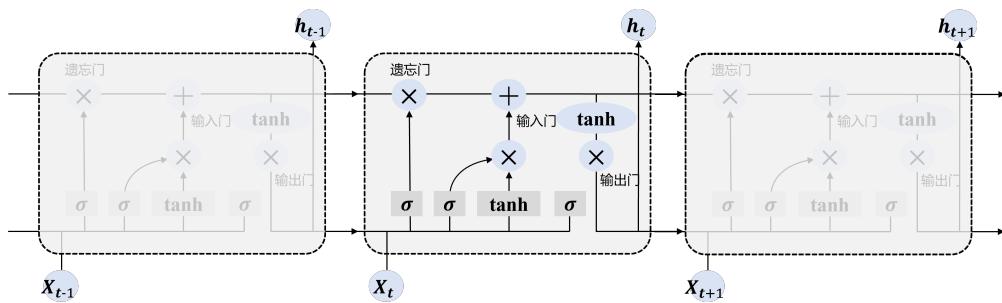


图 3.3 LSTM 模型时序结构图

具体来说，第一步需要接受上一个时间步的隐藏状态 h_{t-1} 和当前时间步的输入

数据 x_t , 输入遗忘门。遗忘门用来决定在当前时间步的存储状态中传递哪些前一个时间步的信息。具体地, 将合并的结果通过矩阵相乘、加偏置、非线性转换, 得到一个介于 0 到 1 之间的输出值 f_t , 通过公式3.1决定是否需要保留和遗忘多少信息, 其中 σ 表示为非线性激活 sigmoid 函数, 得出的值越接近 0 越有可能被丢弃, 越接近 1 越有可能被记住。

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (3.1)$$

输入门用来确定输入的 x_t 中哪些信息能够传递到当前时间步, 具体的计算公式如3.2, 同时依赖于 \tanh 和 sigmoid 函数, 得到当前时间步中需要记录的值 \tilde{C}_t 。

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \end{aligned} \quad (3.2)$$

为了更新新旧细胞的状态, 使用公式3.3计算当前时刻细胞的最终状态 C_t 。具体地, 用 f_t 来控制丢弃旧细胞信息, 同时加上新的重要信息。

$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t \quad (3.3)$$

输出门用来确定当前时间步的信息有多少能够传递到下一个时间步, 具体的计算公式如3.4。

$$\begin{aligned} out_t &= \sigma(W_{out} \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= out_t \otimes \tanh(C_t) \end{aligned} \quad (3.4)$$

通过上述三个门的时序操作, 最终得到长期状态 C_t 和短期记忆 h_t 。长期状态 C_t 从左到右经过 LSTM 单元, 通过遗忘门丢失一些前时间步信息, 通过输入门添加当前时间步信息, 最后通过输出门输出。但 LSTM 模型存在一个问题, 即无法获取从后往前的信息。

为了更好地提高模型对 Token 序列数据的表征能力, 本文选择了双向长短期记忆网络 BiLSTM 模型来进行 Token 代码表征, 模型的结构如图3.4所示。BiLSTM 模型由一个正向的 LSTM 模型和一个反向的 LSTM 模型组成, 主要思想是通过把序列向前、向后分别输入给两个独立的递归网络, 这两个子网络连接到一个输出层, 在每个词的输出部分把两个子网络的输出信息进行整合, 这样网络就同时拥有了序列中每个词的过去时刻信息和未来时刻信息, 更全面地捕捉序列信息, 从而提高模型的性能。BiLSTM 可以捕获到序列前后的关系依赖, 将代码片段的 Token 序列转换为可相互比较的向量。

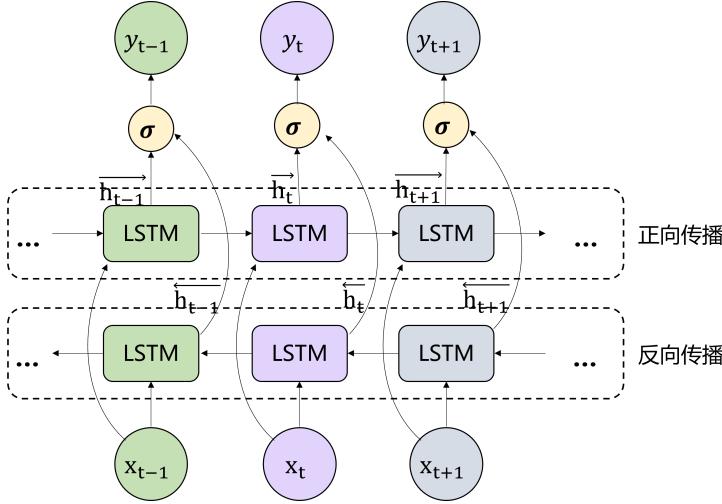


图 3.4 BiLSTM 模型结构图

令 BiLSTM 的 t 时刻的前向隐藏状态为 \vec{h}_t , 向前输出为 \vec{y}_t , 向后隐藏状态为 \overleftarrow{h}_t , 向后输出为 \overleftarrow{y}_t , 输出层参数是 W_0 与 b_0 , 则当前状态 \vec{y}_t 和 \overleftarrow{y}_t 可以通过公式3.5计算.

$$\begin{aligned}\vec{y}_t &= W_0 \cdot LSTM(x_t, \vec{h}_{t-1}) + b_0 \\ \overleftarrow{y}_t &= W_0 \cdot LSTM(x_t, \overleftarrow{h}_{t-1}) + b_0\end{aligned}\quad (3.5)$$

对当前状态 \vec{y}_t 和 \overleftarrow{y}_t 进行拼接可以得到当前 t 时刻的最终输出 y_t , 如公式3.6所示。

$$y_t = \vec{y}_t \oplus \overleftarrow{y}_t \quad (3.6)$$

3.3 Token 表征方法具体实现

在介绍具体实现之前, 本节首先给出 Token 表征方法的输入: 经过2.2.2小节的代码预处理阶段, 得到示例代码片段2.3对应的 Token 序列, 如图3.5所示。

```
[include', '<', 'stdio', 'h', '>', ←
'int', 'main', 'int', 'i', '=', '0', ←
'int', 'data', '5', '=', '1', '2', '3', ←
'4', '5', 'int', 'result', '=', '0', ←
for', 'i', '<', '5', 'i', 'result', '=', ←
'data', 'i', 'printf', '%', 'd', 'result', ←
'return', '0']←
```

(a) 代码片段 1 对应的 Token 序列

```
[include', '<', 'stdio', 'h', '>', ←
'int', 'main', 'int', 'i', '=', '0', ←
'int', 'data', '5', '=', '1', '2', '3', ←
'4', '5', 'int', 'result', '=', '0', ←
while', 'i', '<', '5', 'result', '=', ←
'data', 'i', 'i', 'printf', '%', 'd', ←
'result', 'return', '0']←
```

(b) 代码片段 2 对应的 Token 序列

```
[include', '<', 'stdio', 'h', '>', ←
'int', 'main', 'int', 'i', '=', '0', ←
'int', 'data', '5', '=', '1', '2', '3', ←
'4', '5', 'int', 'result', '=', '0', ←
for', 'i', '<', '5', 'i', 'if', 'result', ←
'<', '10', 'result', '=', 'data', 'i', ←
'printf', '%', 'd', 'result', ←
'return', '0']←
```

(c) 代码片段 3 对应的 Token 序列

图 3.5 示例源代码对应的 Token 序列

接下来，本章提出的基于预训练辅助模型的 Token 表征学习方法的实现如图3.6所示。该方法的输入是一对代码片段 C_a, C_b 对应的 Token 序列，表示为 $(w_1^a, w_2^a, \dots, w_m^a)$ 和 $(w_1^b, w_2^b, \dots, w_n^b)$ ，输出是 C_a, C_b 对应的属性特征向量 V_a^{Token}, V_b^{Token} ，整体采用 Siamese 架构，两个子网络共享权值，从下到上，主要包括预训练辅助词嵌入、Token 代码表征两个阶段。

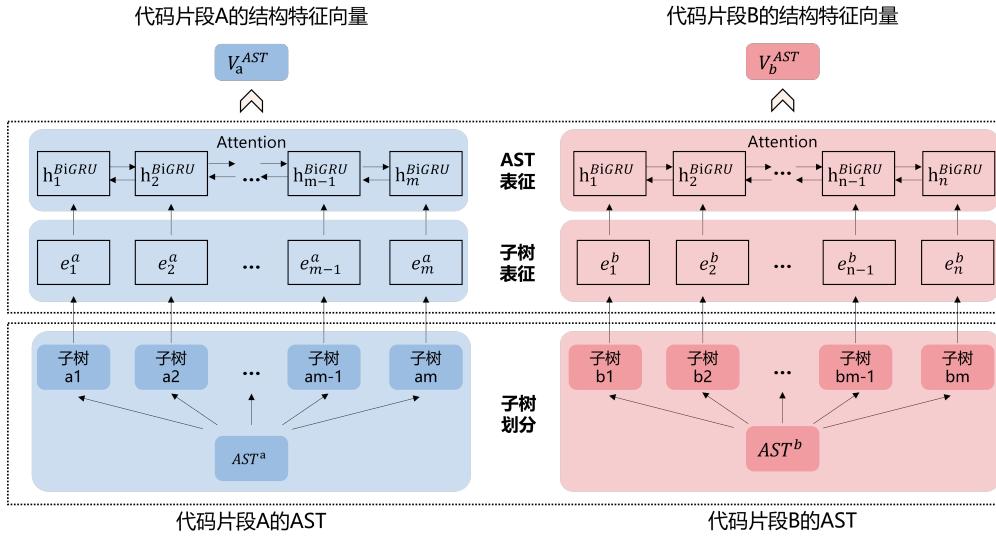


图 3.6 基于预训练辅助模型的 Token 表征学习方法实现

具体来说，在预训练辅助词嵌入阶段，对于代码片段 C_a ，经过分割得到的 Token 序列 $(w_1^a, w_2^a, \dots, w_m^a)$ ，(m 是代码片段 A 的长度)，然后在词汇表 E_w 中查找每个 Token 对应的向量，将所有的 token w_i^a ($i \in [1, m]$) 转化为密集向量。其中查找词汇表 E_w 已经经过代码语料库使用相邻单元迭代算法 ATIA 基于 Word2vec 模型进行预训练，在模型中是固定的。其查找过程可以表示为公式3.7:

$$x_i^a = \text{lookup_Token}(w_i^a, E_w) \quad (3.7)$$

经过嵌入阶段的预训练后，代码片段 C_a 生成了序列 $(x_1^a, x_2^a, \dots, x_m^a)$ 。使用同样的嵌入方式，可以得到对于 Token 序列 $(w_1^b, w_2^b, \dots, w_n^b)$ ，(n 是代码片段 B 的长度) 的向量表示 $(x_1^b, x_2^b, \dots, x_n^b)$ 。

在 Token 代码表征阶段，将代码向量的 Token 序列作为输入，使用 BiLSTM 模型进行编码。处理过程分为两个步骤，如公式3.8所示：

$$\begin{aligned} h_1^{aBiLSTM}, h_2^{aBiLSTM}, \dots, h_n^{aBiLSTM} &= BiLSTM(x_1^a, x_2^a, \dots, x_m^a) \\ V_a^{Token} &= \text{Attention}(h_1^{aBiLSTM}, h_2^{aBiLSTM}, \dots, h_n^{aBiLSTM}) \end{aligned} \quad (3.8)$$

经过嵌入阶段的预训练后，每一个 Token w_i^a 都被转换为一个密集向量 x_i^a ，经过 BiLSTM 编码后，生成了 $h_i^{aBiLSTM}$ ，然后经过 Attention 层的总结后，生成了 V_a^{Token} 作为代码片段 C_a 的最终 Token 表示，即属性特征向量。同样，可以使用相同的计算以 $(x_1^b, x_2^b, \dots, x_n^b)$ 作为输入为代码片段 C_b 计算 V_b^{Token} 。

3.4 实验验证

为了验证基于预训练辅助模型的 Token 表征方法的有效性，本节展开实验验证。首先，介绍了实验整体的环境配置、数据集，以及实验评估指标；接着，对基于预训练模型的 Token 表征方法进行了消融实验。

3.4.1 实验环境

本章的实验验证均运行于 Linux 系统下，其系统硬件配置如表3.1所示。

表 3.1 实验环境配置

环境	配置
操作系统	Ubuntu 20.04
处理器	Intel Core i9-12900KF × 24
内存	31.1G
显卡	NVIDIA
磁盘	1TB

3.4.2 实验数据集

为了验证预训练辅助模型在 Token 层面上表征学习的有效性，本文面向代码克隆检测任务对预训练辅助模型进行分析和评估，选取的实验数据为 POJ104 数据集。如表3.2所示，POJ104 数据集是一个基于 C 语言所构建的大型数据集。OJ 系统是一个以编程教学为目的公开评判系统，共存在 104 个编程问题，针对每个编程问题，学生们通过在线提交自己的代码来尝试解决，同时 OJ 系统将自动判断提交源代码的正确性和有效性。对于 OJ 系统来说，同一个问题的不同答案可以看作一对真克隆对，不同

问题的答案可以视为一对假克隆对。POJ104 数据集针对每一个编程问题，均提供 500 个学生提交源代码，即共有 52000 个样本。

表 3.2 POJ104 数据集

代码	属性
Dataset	POJ104 数据集
Language	C
Program	52000
Classes	104
Max tokens	8737
Avg tokens	245

得到 POJ104 数据集后，本文首先对数据集进行初步筛选，去掉其中包含乱码的样本，共得到 51485 个源代码样本。然后对源代码进行预处理，通过深入研究并比较现有标记转换规则，发现 CC Learner^[16]对转换规则进行了实验评估，验证各个标记的有效性，因此本文参考 CC Learner^[16]这篇文献的转换规则，对标识符进行了归一化处理。具体地，保留 IF、For 等关键词，数字常量值用 NUM 替代，字符串用 String 替代等。删除样本中包含的空白行和注释等多余代码，并将数据集保存到一个 program.pkl 文件中，program.pkl 文件中一共包含 51485 行 \times 3 列的数据集，每一行数据代表一个源代码样本，第一列为源代码 id，第二列保存源代码样本 code，第三列为源代码的标签 label，即属于哪一个编程问题。

接着，本文随机两两组合同一标签 label 的源代码，组成 5200 个真克隆对，随机组合不同标签的源代码组成 44800 个假克隆对，一共给包含 50000 个克隆对，并将其保存到 oj_clone_ids.pkl 文件中，oj_clone_ids.pkl 文件中一共包含 50000 行 \times 3 列的数据集，每一行数据代表一个克隆对样本，第一列为源代码 id1，第二列为源代码 id2，第三列为克隆对的标签 label，真克隆对标签为 1，假克隆对标签为 0。最后，依据随机种子将数据集按照 3: 1: 1 划分为训练集、测试集、验证集，其中的正负样本数如下表3.3所示。

表 3.3 本文预处理后的 POJ104 数据集正负样本数

数据集	真克隆对	假克隆对	克隆对数
训练集 train	3162	26838	30000
测试集 test	1022	8978	10000
验证集 dev	1016	8984	10000
总计	5200	44800	50000

3.4.3 评估指标

代码克隆检测问题是二分类问题，因此本文采用精确率（Precision）、召回率（Recall）、F1 值三个评估指标来度量实验结果，其中使用了混淆矩阵中的 TP、FN、FP、TN，如表3.4所示。

表 3.4 分类问题的混淆矩阵

实际值	预测值	
	正样本 (P)	负样本 (N)
正样本 (P)	TP	FN
负样本 (N)	FP	TN

其中，混淆矩阵中的真阳性、假阳性、真阴性、假阴性代表的含义如下：

真阳性 (True Positive, TP): 样本实际为正样本，并且被模型预测为正样本，即实际上标记为真克隆对并且被检测出来为真克隆对的代码对。

假阳性 (False Positive, FP): 样本实际为负样本，但是被模型预测为正样本，即实际上标记为假克隆对但是被检测出来为真克隆对的代码对。

假阴性 (False Negative, FN): 样本实际为正样本，但是被模型预测为负样本，即实际上标记为真克隆对但是被检测出来为假克隆对的代码对。

真阴性 (True Negative, TN): 样本实际为负样本，并且被模型预测为负样本，即实际上标记为假克隆对并且被检测出来为假克隆对的代码对。

精确率（Precision）表示正确检测到的代码克隆数量占全部预测为代码克隆的比例，计算公式如3.9所示：

$$Precision = \frac{TP}{TP + FP} \quad (3.9)$$

召回率 (Recall) 表示正确检测到的代码克隆数量占总体实际代码克隆数量的比

例，计算公式如3.10所示：

$$Recall = \frac{TP}{TP + FN} \quad (3.10)$$

精确率 (Precision) 和召回率 (Recall) 指标有时候会出现的矛盾的情况，这样就需要综合考虑两者的表现，最常见的方法就是 F1，精确率和召回率的加权调和平均，用于评价分类模型的好坏。计算公式如3.11。

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (3.11)$$

3.4.4 实验结果

(1) 预训练语料库规模的实验结果

为了探究预训练语料库规模对 Token 表征实验结果的影响，本文选取 POJ104 数据集中部分数据进行预训练，基于相邻单元迭代算法构造预训练语料库。具体地，在3.4.2数据库中随机选取了 100 对、200 对、500 对代码克隆对作为预训练数据集，构造预训练语料库，对 Token 表征模型进行预训练。具体实验结果如表3.5所示。

表 3.5 预训练语料库规模对实验结果的影响

预训练数据集规模	词表长度	OJClone		
		准确率 (%)	召回率 (%)	F1 值 (%)
100 对	240	89.74	87.84	88.78
300 对	463	89.76	90.24	90.00
500 对	593	89.78	100	94.62

从表3.5可以看出，随着预训练语料库的增大，代码克隆检测任务的 F1 值呈现上升趋势，即对代码克隆的检测效果会不断增强。考虑到需要预留出足够的答案样本用来构造训练集和测试集，最终选择 500 对作为 C 语言实验中的预训练数据集大小。

(2) Token 表征模型实验结果

为了探究 Token 表征模型 AttBiLSTM 对实验结果的影响，本文基于预训练辅助的基础上，将 AttBiLSTM 模型与 LSTM 与 BiLSTM 模型进行对比，结果如表3.6所示。其中 LSTM、BiLSTM 模型基于 Pytorch1.10 实现，其参数设置为：①词嵌入层：将大小为 600 的词汇表映射到 128 维的向量空间。②双向 LSTM 层：双向的 LSTM 单元堆叠而成，输入特征的维度与嵌入层匹配为 104 层，隐藏层维度设置 104，由于设置为

双向，因此输出维度为 $104*2=208$ 层。③全连接层：208 层的输入映射到 1 维的输出中。④sigmoid 层：激活函数，将全连接层的输出转换到 0 到 1 之间的概率值。模型的优化器为 Adam 优化器，学习率设置为 0.0001，Dropout 为 0.1，Epochs 为 50，参数的确定是通过多次调试后选择最优参数作为最后的结果。

表 3.6 Token 表征模型对实验结果的影响 (%)

表征模型	准确率 (%)	召回率 (%)	F1 值 (%)
RNN	xxx	xxx	xxx
LSTM	xxx	xxx	xxx
AttBiLSTM	89.78	100	94.62

从表3.5可以看出，随着预训练语料库的增大，克隆检测任务的 F1 值呈现上升趋势，即对代码克隆的检测效果会不断增强。考虑到需要预留出足够的答案样本用来构造训练集和测试集，最终选择 AttBiLSTM 模型作为 Token 表征模型。

3.5 本章小结

本章主要对 RLCCD 中基于预训练辅助模型的 Token 表征学习方法的设计与实现进行详细阐述。首先介绍了 Token 维度的研究动机，其次介绍了 Token 表征学习的方法设计，具体论述了其整体框架、预训练辅助模型、表征学习，接着开展实验证，结果表明了此方法的有效性和模型的准确性。

第4章 基于子树划分的抽象语法树表征学习

本章主要对本文提出的基于子树划分的抽象语法树表征学习方法进行详细介绍，首先介绍其研究动机，其次阐述其具体方法设计与实现，最后进行实验验证。

4.1 研究动机

基于 Token 的代码表征通常将代码视为自然语言文本，根据程序员的开发风格不同，代码的命名方式、上下文组织方式也不一样，因此在代码表征过程中通常会产生很多噪声，并且遗漏源代码的结构信息。为了提高代码表征能力，研究人员提出了基于抽象语法树的代码表征方法。抽象语法树是源代码语法结构的一种抽象表现形式，以操作数、操作符、声明节点、语句节点等作为树节点，以树的形式包含了源代码中的语法信息和语法结构。基于树的方法利用了树本身的结构化特征，利用深度神经网络对树进行建模得到其向量表示，根据该特征向量完成下游代码任务，在一定程度上可以消除源代码本身的噪声，有效地提取程序的结构信息。

基于树的代码表征方法存在两种主要限制：

(1) 树转化导致高度增加：由于抽象语法树的大小和深度对神经网络性能有显著影响，因此现有的基于树的方法通常将抽象语法树转化为二叉树，通过将父节点的第三个或者更多子节点移动到新的子树中进行简化。然而，在转换的过程中会改变源代码原有的语义，从而难以捕捉远程依赖关系，甚至丢失一些上下文信息。有研究发现^[30]，在抽象语法树中，具有高度关联的两个节点可能相距甚远，例如：函数调用的形参节点与实参节点存在数据依赖关系，但两者可能在转化过程中划分到不同的子树中，降低对程序的结构语法捕获能力。因此，由于转化会增加抽象语法树高度，从而加重了梯度消失问题，削弱神经模型捕捉更真实和复杂语义的能力。

(2) 神经网络梯度消失：在使用神经网络对抽象语法树建模的过程中，梯度是通过树型拓扑结构的反向传播来计算的。由于源程序结构的复杂性，因此抽象语法树通常规模过大、节点很深，此时会出现梯度消失问题。具体地，在反向传播过程中，神经网络根据设定好的损失函数指导权重值的更新优化。而梯度（即损失函数对模型参数的导数）经过多层网络传递时，如果激活函数的导数接近于 0，参数更新就会变得不稳定，导致模型发散或者训练不收敛，影响神经网络模型的训练效率和稳定性。

因此，针对上述问题，本文提出了一种基于子树划分的抽象语法树表征方法，该

方法将大型抽象语法树分割为小型语句树序列，在减小抽象语法树大小和深度的同时，通过捕获语句树的特征，提高树维度的代码表征能力。

4.2 AST 表征方法设计

本节将介绍基于子树划分的抽象语法树表征学习方法的详细设计，首先介绍该方法的整体框架，并从子树划分、抽象语法树表征学习两方面介绍具体设计。

4.2.1 框架概述

本文提出的基于子树划分的抽象语法树表征学习方法整体框架如图4.1所示。该框架的输入是代码片段对应的抽象语法树，输出是对应的结构特征向量，主要包括子树划分、树表征两个阶段。

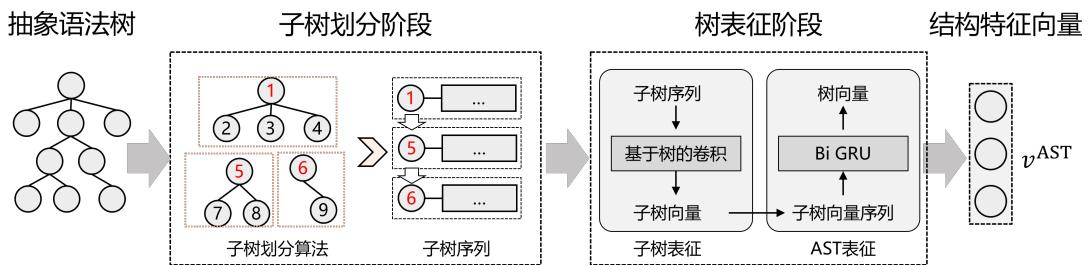


图 4.1 基于子树划分的抽象语法树表征学习框架

首先，子树划分阶段以代码片段对应的抽象语法树作为训练数据，以抽象语法树的声明节点和语句节点作为切割粒度，设计一个基于先序遍历的子树划分算法，将抽象语法树切分成若干子树，得到对应的子树序列。

其次，树表征阶段以子树序列作为输入，输出代码片段对应的结构特征向量。该阶段分为两个部分，首先针对每一个子树，构建一个基于树的卷积神经网络将子树编码成向量，实现对细粒度语义的捕捉；其次，构建一个基于双向门控循环单元(BiGRU)的神经网络模型，通过最大池化层对子树特征进行压缩，输出一个固定长度的密集向量用来表示代码的结构特征。需要注意的是，在树表征阶段，本文使用两种神经网络模型，前者在语句粒度上对代码信息进行特征提取，后者引入在GRU模型的基础上引入双向结构，从而更好地捕捉序列数据地双向依赖关系。

在上述框架中，本文的创新点主要体现在子树划分阶段的子树划分算法、树表征阶段的树卷积、BiGRU模型设计两方面，下面将围绕这两个创新点来阐述本文的方法。

4.2.2 子树划分设计

抽象语法树是源代码语法结构的一种抽象表现形式，以树的形式包含了源代码中的语法信息和语法结构，不同的节点类型代表了源代码中的不同元素。其中，声明节点和语句节点是两种关键的节点类型。

声明节点用来定义代码中变量、函数、类等元素的声明，不仅包含了声明的标识符名称，还包含了该标识符的类型信息、作用域以及可能的初始值等。例如，在编译时，编译器可以通过检查声明节点来确保变量在使用前已经被正确声明，并且其类型与使用场景相匹配。

语句节点则用来定义代码的执行语句，如赋值语句、条件语句、循环语句、函数调用等。在 AST 中，语句节点通常包含了语句的类型、操作数以及可能的控制流信息。例如，在编译条件语句时，编译器可以通过分析语句节点来确定条件表达式的求值结果，并据此决定程序的执行路径。

抽象语法树存在因为树规模过大、高度过深导致的梯度消失问题，本文设计了一个基于先序遍历的子树划分算法，通过在语句粒度上对 AST 进行划分，在减少树高度的同时，实现对细粒度语义的捕捉。下面首先对本文提出的子树划分算法涉及到的基本元素进行定义，接着详细介绍整个算法流程。

定义 4.1. 抽象语法树语句节点集合：对于一个代码片段的 AST，假设该 AST 的节点集合为 $T = \{T_1, T_2, \dots, T_n\}$ ，影响子树划分标准的语句节点包括：块语句、FOR 循环语句、While 循环语句、条件语句、返回语句，因此本文定义这些不同类型语句节点构成集合 $S = \{BlockStatement, IfStatement, WhileStatement, ForStatement, ReturnStatement\}$ 。

具体来说，子树划分算法需要寻找一个子树节点集合，表示为 $SubT = \{SubT_i | SubT \in T, SubT \in S, i = 1, 2, \dots, k (k < n)\}$ ，其中集合 $SubT$ 通过先序遍历 AST 节点获得， n 表示 AST 中节点的总数量。

子树划分 *split_AST* 算法的伪代码如2所示，该算法有两个输入：根节点 *Root_Node*、语句节点集合 *S*，输出为子树序列 *SubTrees*。该算法的目的是按照先序遍历抽象语法树，根据给定的语句节点集合 *S*，从根节点 *Root_Node* 开始，判断每个子节点 *child_node*，是否需要将其作为新的子树进行切分（算法第 4 行），如果子节点属于语句节点集合，则直接加入子树节点集合中（算法第 5 行），否则需要继续遍历，找到包含子节点的子树，进行递归切分（算法第 11 行），最终得到子树节点列表，即子树

序列 $SubTrees$ 。

Algorithm 2 Subtree partitioning algorithm ($split_AST$)

Input: Root node of abstract syntax tree: $Root_Node$

Input: Node set: S

Output: SubTree set: $SubTrees$

```

1:  $SubTrees = \{\}$ 
2:  $SubTrees.append(Root\_Node)$ 
3: for child_node in Root_Node.children do
4:   if child_node ∈ S then
5:     SubTrees.append(child_node)
6:   else
7:     for subtree in SubTrees do
8:       if child_node ∈ SubTrees.descendant then
9:         SubTrees.append(child_node)
10:      else
11:        split_AST (subtree)
12:      end if
13:    end for
14:   end if
15: end for
16: return SubTrees

```

具体地，如图4.2所示，(a) 为示例代码片段，(b) 为利用代码分析工具 Joern 生成得到抽象语法树 T ，(c) 为利用子树划分算法将其切分为一系列子树 $SubT$ 。其中，以 Method 为根节点，其名称为 main，body 节点则代表了内部代码块，该块中包含三棵变量声明子树 (Local)、一棵 FOR 循环条件子树 (ForStatement)、一棵函数声明子树 (Function) 以及返回语句子树 (ReturnStatement)，而 FOR 循环条件子树内部包含一棵 If 条件句法子树 (IfStatement)。

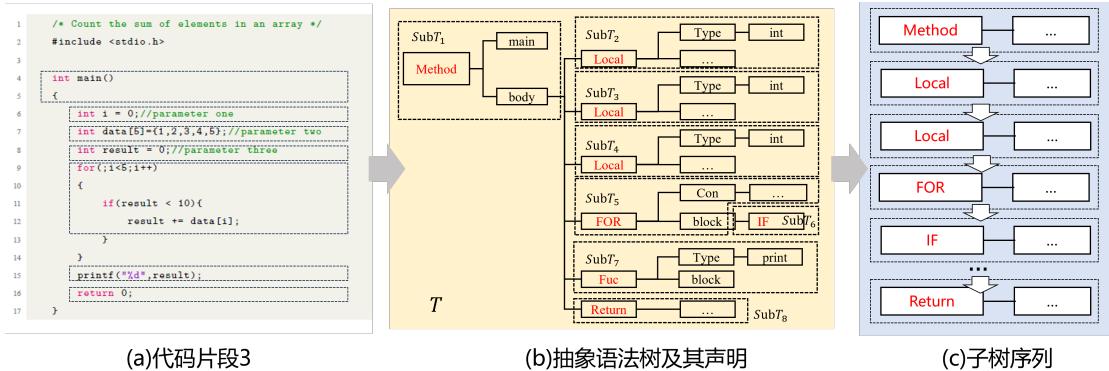


图 4.2 子树划分算法示例

4.2.3 抽象语法树表征学习设计

(1) 结构设计

为了提高抽象语法树维度代码表征能力，本文选取树卷积网络对上述得到子树序列进行建模，使用双向门控循环单元 (BiGRU) 对整个树进行建模。具体的模型设计如图4.3所示。该模型主要包括输入层、子树卷积层、双向GRU层、最大池化层、输出层。

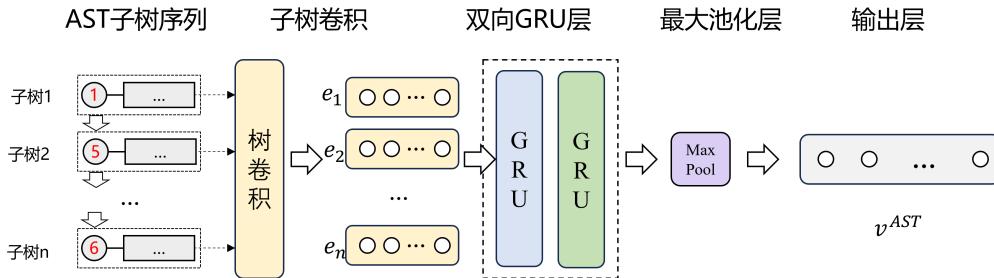


图 4.3 抽象语法树表征模型设计

①输入层：输入层用于向模型输入训练数据，在本方法中模型的输入为经过子树划分得到的子树序列。②子树卷积层：针对子树序列中的每一个子树，通过基于树的卷积，捕获语句粒度语义信息。③双向门控循环层：由两层 GRU 构成，同时捕获序列的双向语义信息。④最大池化层：总结子树序列的输入特征，并将其缩减为一个单一的密集向量。⑤输出层：每个抽象语法树对应一个输出。

(2) 模型选型

树表征阶段包含两个步骤：子树表征、整树表征。因此本文设计了两个神经网络模型：前者的输入是子树序列，针对每个子树进行基于树的卷积，在子树粒度上对代码进行信息提取；后者的输入是子树向量序列，针对所有子树的特征进行双向特征提取，并整合整棵树的信息，得到一个结构特征向量。

(2.1) 子树表征模型

卷积神经网络（Convolutional Neural Network, CNN）是一种深度学习模型，其架构包括多个卷积层、池化层和全连接层。其中，卷积层负责从输入数据中提取特征，池化层用于降低数据维度，全连接层则用于将前面各层的特征映射到输出空间。其中，卷积思想是指将一个固定大小的窗口（通常被称为卷积核）在输入数据上按照一定的步长进行滑动，并对每个窗口中的局部片段进行特征提取，最后得到一系列特征，每

一个特征对应一个卷积核提取的特征。它能够将输入数据从底层到高层逐步抽象化，形成层次化的特征表示。

常见的卷积核通常都是正方形的，但树形结构通常是不规则，因此有研究^[26]提出了基于树的卷积神经网络。该网络的核心思想是将卷积操作扩展到树形结构上。通过定义在树上的卷积核，可以捕获树的节点与其邻居之间的局部特征。这种局部感知的方式与传统的卷积神经网络类似，但不同的是，其卷积核是三角形的，能够在不规则的树形结构上进行。基于树的卷积如下图4.4所示。

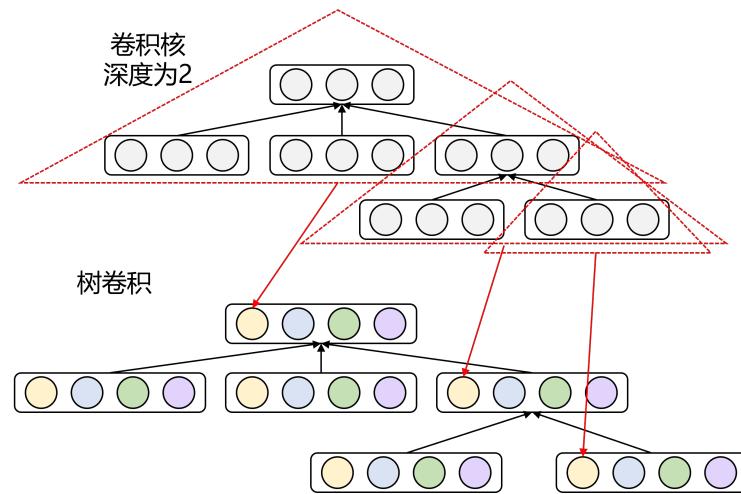


图 4.4 子树表征：树卷积设计

在基于树的卷积神经网络中，每个节点都与其邻居节点相连，形成一个局部邻域。卷积核在这个局部邻域上进行滑动，计算节点与其邻居的加权和，从而提取出特征。用数学公式表示，卷积核的滑动尺寸是 d ，它表示每个滑动窗口能够包括的树的层数。在这个窗口内包含 n 个节点 (x_1, x_2, \dots, x_n) ，其中， $x_i \in \mathbb{R}^{N_f}$ ， N_f 表示每个节点建模后的向量大小。使用公式4.1可以对滑动窗口内 n 个节点进行卷积得到基于树的卷积神经网络输出。

$$y = \tanh \left(\sum_{i=1}^n W_{\text{conv},i} \cdot x_i + b_{\text{conv}} \right) \quad (4.1)$$

其中， $y, b_{\text{conv}} \in \mathbb{R}^{N_f}$ ， $W_{\text{conv},i} \in \mathbb{R}^{N_c \times N_f}$ ， N_c 为最后卷积得到的向量长度。具体地，在上图4.4的例子中，红色三角形表示卷积核，固定深度 d 为 2，最后输出的卷积向量长度 N_c 为 4。树结构在卷积前后保持相同的形状，而每个节点向量的维数由原来的 3 维变为 4 维。

基于树的卷积网络模型存在一个限制：其子节点的数量被限制为两个。但实际上，

AST 理论上可以存在无限数量的子节点，每次滑动窗口中的节点数 n 难以固定，从而导致无法确定权重矩阵的数量，即公式4.9中的 $W_{conv,i}$ 。现有研究的解决方法是预先定义一组规则将抽象语法树转化为二叉树，然后再进行树卷积。这种处理方式，会改变源代码原有的语义，从而难以捕捉远程依赖关系，甚至丢失一些上下文信息，因此本文采用研究^[26]中提出的连续二叉树概念，将 AST 的每一个子树都看作二叉树，而不管它的形状和尺寸。图4.5展示了连续二叉树的定义。

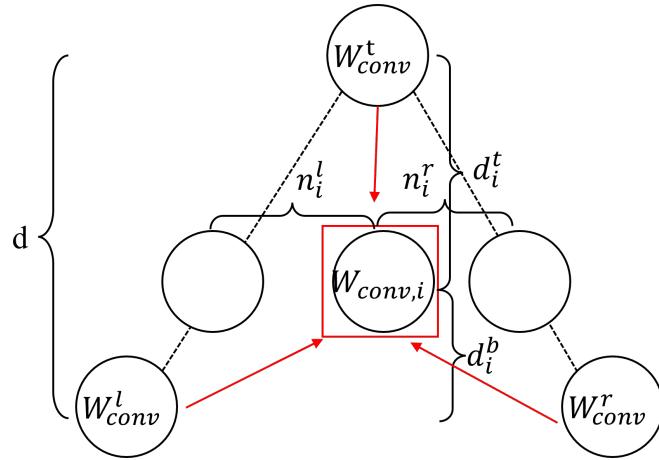


图 4.5 连续二叉树

具体地，针对滑动窗口中的节点，需要设置三个变量： $W_{conv}^t, W_{conv}^l, W_{conv}^r$ 分别代表当前节点距离树形上、左、右三个方向，其卷积矩阵 $W_{conv,i}$ 是这三个变量的线性组合，其系数根据节点在当前滑动窗口中的相对位置计算，具体公式为4.2：

$$W_{conv,i} = \frac{d_i^b}{d_i^b + d_i^t} W_{conv}^t + \frac{d_i^t}{d_i^b + d_i^t} W_{conv,i}^b \quad (4.2)$$

其中， d_i^t 表示当前节点距离树形顶点的距离， d_i^b 表示当前节点距离树形左右顶点构成的边的距离，两者相加为 d ，即 $d_i^t + d_i^b = d$ 。上式中 $W_{conv,i}^b$ 有如下定义：

$$W_{conv,i}^b = \begin{cases} \frac{n_i^r}{n_i^r + n_i^l} W_{conv}^l + \frac{n_i^l}{n_i^r + n_i^l} W_{conv}^r & n_i^l \geq 1 \text{ or } n_i^r \geq 1, \\ \frac{1}{2} W_{conv}^l + \frac{1}{2} W_{conv}^r & n_i^l = n_i^r = 0. \end{cases} \quad (4.3)$$

其中， n 表示树的同一层左右两个方向的兄弟节点个数。通过上述公式，可以计算在一个三角形的卷积窗口中，当节点越接近顶点，其 W_{conv}^t 的权重值越大，当节点越接近树形结构左下节点，其 W_{conv}^l 的权重值越大，当节点越接近树形节点结构右下节点，其 W_{conv}^r 的权重值越大。在具体实验中，本文设置滑动窗口的 d 值为 2。

(2.2) 整树表征模型

3.2.3小节中提到长短期记忆网络LSTM模型针对长依赖问题效果明显，解决了传统RNN模型中存在的梯度消失问题，但是LSTM模型同样存在缺陷，其LSTM单元内包含三个门，机制相对复杂，计算成本会有所提高，因此，有研究提出了门控循环单元GRU，将遗忘门和输入门进行组装，形成一个新的更新门，这种改进使得GRU在模型参数和计算效率上通常优于LSTM。GRU单元的时序结构如下如4.6所示，每个GRU单元都包含两个门：重置门(Reset Gate)和更新门(Update Gate)，通过重置门和更新门来直接控制信息的流动和隐藏状态的信息更新。

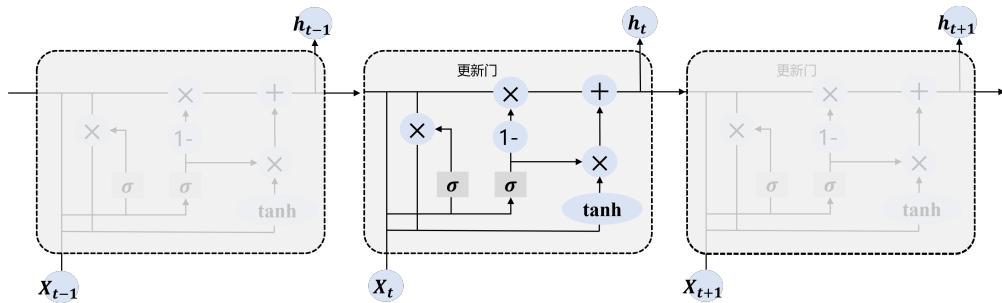


图 4.6 GRU 模型时序结构图

具体来说，首先会根据当前时间步的输入 x_t 和上一时间步的隐藏状态 h_{t-1} 计算出重置门和更新门的值，具体的计算公式见4.4。

$$\begin{aligned} z_t &= \sigma(W_u \cdot [h_{t-1}, x_t] + b_z) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \end{aligned} \quad (4.4)$$

然后，输入重置门，对上一个时间步的隐藏状态进行处理，得到一个新的候选隐藏状态 \tilde{h}_t 。其中，重置门用来控制上一个时间步的信息有多少应该被遗忘。具体地，通过公式4.5计算一个介于-1到1之间的值，决定上一个时间步的隐藏状态有多少信息应该被保留或遗忘，得出的值越接近0越有可能被丢弃，越接近1越有可能被记住。

$$\tilde{h}_t = \tanh(W_x \cdot x_t \otimes U_h \cdot (r_t \cdot h_{t-1}) + b_h) \quad (4.5)$$

接着，输入更新门，将新的候选隐藏状态 \tilde{h}_t 和上一个时间步的隐藏状态 h_{t-1} 进行加权组合，得到当前时刻的隐藏状态 h_t ，如公式4.6所示。其中，更新门用来控制新输入的信息和上一个时间步的信息应该如何结合。具体地，通过公式4.7决定当前时刻的隐藏状态应该有多少信息来自上一时刻的隐藏状态，以及有多少信息来自当前时刻的输入和重置门处理后的结果。通过更新门，GRU能够平衡新旧信息的影响，从而

有效地捕捉序列数据中的长期依赖关系。最后，当前时刻的隐藏状态会被输出。

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes \tilde{h}_t \quad (4.6)$$

同样地，在抽象语法树中，某一个节点的状态不仅和前面层的节点状态有关，也可能和后一层的节点状态有关。为了更好地提高模型对 AST 子序列的表征能力，本文选择了双向 BiGRU 模型来进行树代码表征，模型的结构如下图4.7所示。BiGRU 模型结构结合了双向 RNN（双向循环神经网络）和 GRU（门控循环单元）的优点，通过其双向结构能够更有效地学习并保持长期的信息流，从而使得当前时刻的输出与前后时刻的状态都产生联系，在处理长序列数据时表现出色，在具有较快收敛速度、较强学习能力等优点的同时，还能减少梯度消失问题。

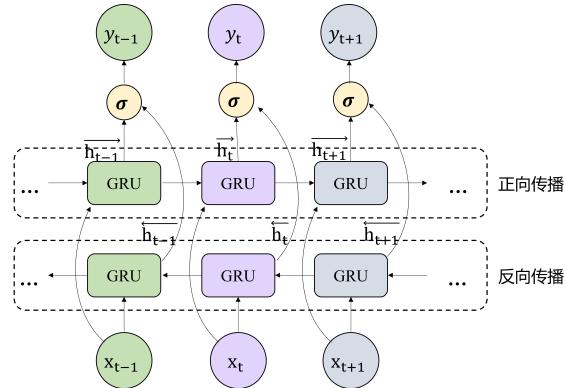


图 4.7 树表征：BiGRU 模型设计

和3.2.3提出的双向长短时记忆模型相比，GRU 结构简单，参数简单，因此通常可以更快收敛到最优解，从而节省计算资源和时间。整个模型，除了子单元有所改变外，架构不变。这里不展开介绍，仅给出公式4.7。

$$\begin{aligned} \vec{y}_t &= W_0 \cdot GRU(x_t, \vec{h}_{t-1}) + b_0 \\ \overleftarrow{y}_t &= W_0 \cdot GRU(x_t, \overleftarrow{h}_{t-1}) + b_0 \\ y_t &= \vec{y}_t \oplus \overleftarrow{y}_t \end{aligned} \quad (4.7)$$

4.3 AST 表征方法具体实现

在介绍具体实现之前，本节首先给出 AST 表征方法的输入：经过2.2.2小节的代码预处理阶段，得到示例代码片段2.3对应的抽象语法树，如图4.8所示。仔细分析可

以看出代码片段4.8(a)对应的抽象语法树在FOR循环内部一共有13个子节点，子树高度为4；代码片段4.8(b)对应的抽象语法树在WHILE循环内部也包含13个子节点，子树的高读为4，两者子节点个数相同；而代码片段4.8(c)对应的抽象语法树在FOR循环中共有18个子节点，子树高度为6，与代码片段4.8(a)对应的抽象语法树在根节点的下一层、下两层中1-9号节点架构相似。

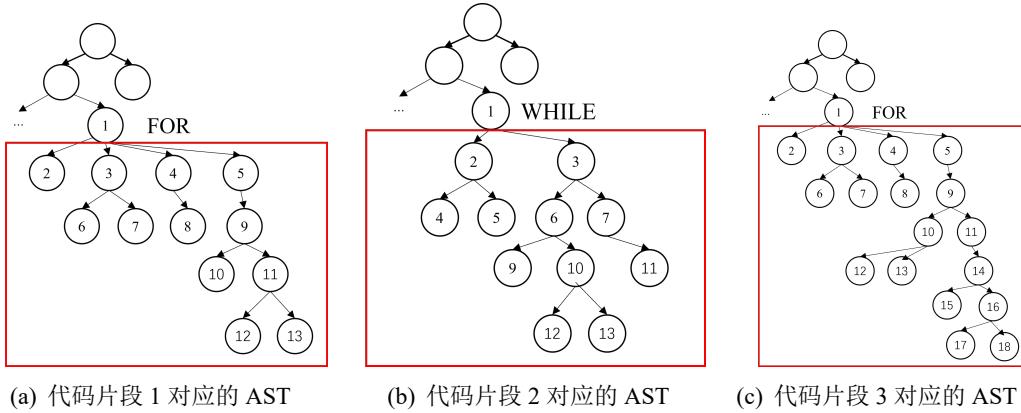


图 4.8 示例源代码对应的抽象语法树

接下来，本章提出的基于子树划分的抽象语法树表征学习方法的实现如图4.9所示。该方法的输入是一对代码片段 C_a, C_b 对应的抽象语法树，表示为 AST_a, AST_b ，输出是 C_a, C_b 对应的结构特征向量 V_a^{AST}, V_b^{AST} ，整体采用 Siamese 架构，两个子网络共享权值，从下到上，主要包括子树划分、子树表征、树表征三个阶段。

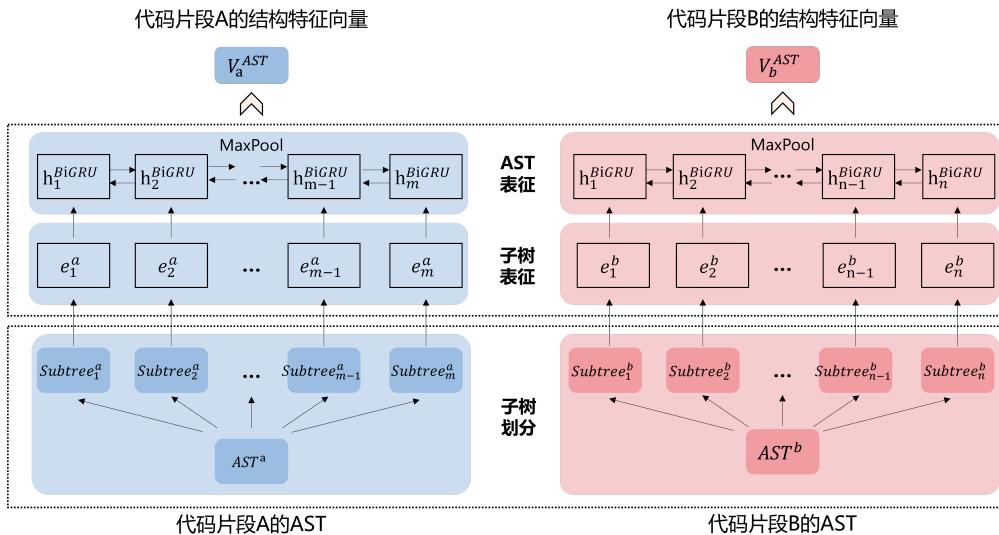


图 4.9 基于子树划分的抽象语法树表征学习方法实现

具体来说，在子树划分阶段，对于代码片段 C_a ，经过代码分析工具 Joern 生成得到抽象语法树 AST_a ，通过子树划分算法，可以得到子树序列，用 $(SubTree_1^a, SubTree_2^a, \dots, SubTree_m^a)$ 表示， m 是子树序列的长度。其子树划分过程可以表示为公式4.8：

$$(SubTree_1^a, SubTree_2^a, \dots, SubTree_m^a) = split_AST(AST_a) \quad (4.8)$$

经过子树划分后，代码片段 C_a 生成了子树序列 $(SubTree_1^a, SubTree_2^a, \dots, SubTree_m^a)$ 。使用同样的划分方法，可以得到代码片段 C_b 生成了子树序列 $(SubTree_1^b, SubTree_2^b, \dots, SubTree_n^b)$ ， n 是代码片段 C_b 对应的子树序列长度。

在树表征阶段，首先将代码片段的子树序列作为输入，使用基于树的卷积神经网络模型进行编码，得到每个子树对应的语句向量 $(e_1^a, e_2^a, \dots, e_m^a)$ 。然后，使用双向门控循环单元 (BiGRU) 来模拟语句的自然性，通过最大池化层将 BiGRU 的隐藏状态采样到单个固定长度的向量 V_a^{AST} 中，作为最终的抽象语法树表示，即结构特征向量。具体的处理过程如公式4.9

$$\begin{aligned} h_1^{aBiGRU}, h_2^{aBiGRU}, \dots, h_n^{aBiGRU} &= BiGRU(e_1^a, e_2^a, \dots, e_m^a) \\ V_a^{AST} &= MaxPool(h_1^{aBiGRU}, h_2^{aBiGRU}, \dots, h_n^{aBiGRU}) \end{aligned} \quad (4.9)$$

同样，可以使用相同的计算以子树序列 $(SubTree_1^b, SubTree_2^b, \dots, SubTree_n^b)$ 作为输入为代码片段 C_b 计算 V_b^{AST} 。

4.4 实验验证

为了验证基于子树划分的抽象语法树表征学习方法的有效性，本节开展实验验证。首先，介绍了实验的具体设计，接着对子树划分、树表征模型进行消融实验。

4.4.1 实验设计

本节使用与3.4节中同样的实验环境和数据集对基于子树划分的抽象语法树表征学习方法进行对比实验。使用代码分析工具 Joern 获取数据集中代码片段的抽象语法树，然后使用基于树的卷积神经网络训练子树嵌入，并将嵌入向量大小设置为 128。模型的 batch_size 为 32，优化器为 Adam 优化器，学习率为 0.002，克隆检测的阈值设置为 0.5。同样选取常用的精确率 (Precision)、召回率 (Recall)、F1 值作为评估指标。

表 4.1 抽象语法树子树划分实验结果

对比	P	R	F1
基于 AST 的 Tree-LSTM	89.78	87.88	88.82
基于 AST 的 + 子树划分的 Tree-LSTM	92.7	87.6	90.0

4.4.2 实验结果

消融对比实验：体现 AST 子树划分的有效性

基于 AST 的 Tree-LSTM

基于 AST 的 + 子树划分的 Tree-LSTM

4.5 本章小结

本章主要对 RLCCD 中基于子树划分的抽象语法树表征学习方法的设计与实现进行详细阐述。首先介绍了抽象语法树维度的研究动机，其次介绍了抽象语法树表征学习的方法设计，具体论述了其整体框架、子树划分、树表征学习，接着开展实验验证，结果表明了此方法的有效性和模型的准确性。

第 5 章 基于图过滤的程序依赖图表征学习

本章主要对本文提出的基于图过滤的程序依赖图表征学习方法进行详细介绍，首先介绍其基本思想，其次阐述其具体方法设计与实现，最后进行实验验证。

5.1 研究动机

程序依赖图 PDG 是代码的一种图形表示，所含结构信息最多，能够表示程序的控制依赖，数据依赖等关系，是一种带有标记的有向多重图。程序依赖图 PDG 结点代表语句，边代表依赖关系，依赖关系包括数据依赖和控制依赖。基于图的代码表征方法首先使用代码分析工具构建包含代码语法结构、调用关系、数据流等信息的程序依赖图，然后通过子图匹配的方法，将 PDG 图中的控制流和数据流编码为一个紧凑的语义特征矩阵，其中每个元素都是一个高维的稀疏二值特征向量。通过将代码表示为图的形式使得模型能够更好地理解代码中不同部分之间的依赖关系，更适合研究代码内的丰富语义信息。

基于图的代码表征方法存在两种主要限制：

(1) 规模开销较高：基于图的表征学习方法通常需要构建代码的结构图或控制流图作为分析的基础，对于具有复杂控制流或数据流的代码片段，构建准确的图表示是一个不小的挑战。特别地，当代码片段中具有循环、递归或异常处理机制时，图的构建过程更加困难。这些复杂结构不仅增加图构建的复杂性，还有可能导致图表示的精度下降；即使在成功生成图之后，图表征学习方法的计算成本也很高。例如，对图进行子图同构等操作时，往往需要借助复杂的图算法来实现，并将生成的程序依赖图两两匹配，对于包含 n 个代码片段的数据集，需要进行 n^2 次匹配检测。而其中包含大量无用的匹配，会浪费大量的时间和计算资源。这些算法不仅计算成本大，而且随着代码库规模的扩大，处理时间也会显著增加，导致开销高。

(2) 对代码修改的敏感性：在实际开发过程中，代码通常会经过各种微小的更改，例如变量名更改、代码格式化、添加或删除注释等，这些修改可能会导致图的表示发生显著变化。具体来说，当代码中的变量名被更改时，图的节点和边可能会受到影响，因为变量名通常作为图中的一个重要特征被考虑在内。同样，代码格式的调整，如缩进、换行或空格的变化，虽然不影响代码的逻辑功能，但也可能导致图的拓扑结构发生变化。此外，添加或删除注释虽然对代码的执行没有影响，但在构建代码图时，这

些注释也可能被当作图的一部分，从而影响到图的表示。因此，基于图的克隆检测方法可能无法准确地检测出这些轻微修改过的克隆代码。更进一步，随着代码库的不断增长和变化，基于图的克隆检测方法可能需要不断地更新和调整以适应新的代码结构。这意味着方法的实现和维护成本可能会相对较高，因为开发者需要定期更新和调整方法以适应代码的变化。这种持续的更新和调整不仅增加了工作负担，还可能影响到方法的长期有效性。

因此，针对上述问题，本文提出了一种基于图过滤的程序依赖图表征学习方法，该方法通过预处理图过滤，减少候选 PDG 对集合的规模。

5.2 PDG 表征方法方法设计

本节将介绍基于图过滤的程序依赖图表征学习方法设计与实现，首先介绍该方法的整体框架，并从图过滤、程旭依赖图表征学习两方面介绍具体设计。

5.2.1 框架概述

本文提出的基于图过滤的程序依赖图表征学习方法整体框架如图5.1所示。该框架的输入是代码片段对应的程序依赖图，输出是对应的语义特征向量，主要包括图过滤、图表征两个阶段。

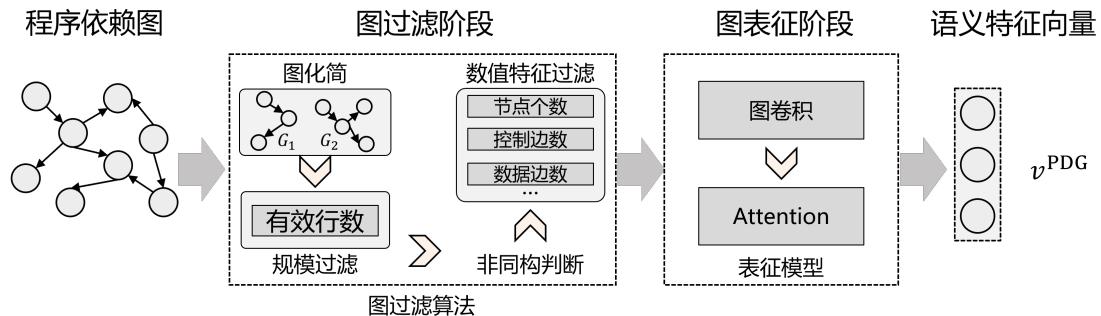


图 5.1 基于图过滤的程序依赖图表征学习框架

首先，图过滤阶段以代码片段对应的程序依赖图作为训练数据，设计一个基于候选程序依赖图对的图过滤算法，通过减少候选图对的规模，提高后续模型的训练能力。需要注意的是，为了更好地减少图规模，本文设计了一些有效的过滤策略，通过对 PDG 的边进行约简、对 PDG 对集合进行过滤的方式，只保留克隆可能性较大地候选 PDG 对，从而有效地加速检测过程，减少后续模型输入规模。

其次，图代码表征阶段以过滤后的程序依赖图作为训练数据，构建一个图表征模型。该模型的输入是程序依赖图，输出为一个固定长度的密集向量用来表示代码的语言特征。需要注意的是，本文选用的图卷积神经网络，主要包含两个部分：图卷积部分和自注意力机制部分，前者主要目的是通过在图上的卷积操作来捕捉节点的特征以及节点之间的关系。后者的主要目的是总结图节点特征，并将每个代码片段缩减为一个单一的密集向量。

在上述框架中，本文的创新点主要体现在图过滤阶段的图过滤算法、图代码表征阶段的模型设计两方面，下面将围绕这两个创新点来阐述本文的方法。

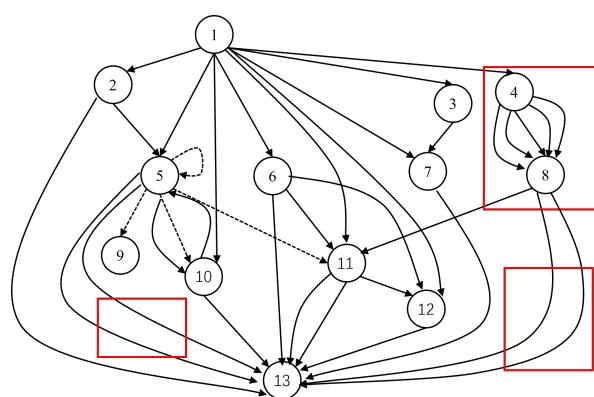
5.2.2 图过滤设计

本文使用代码分析工具 Joern 生成程序依赖图。其中，Joern 通过静态分析源代码，生成关键的图结构信息，反映代码中的依赖关系和函数调用层次，同时 Joern 提供查询和可视化功能，用户可以通过命令将分析结果导出为多种格式，从而更好地理解代码的逻辑和流程。使用 Joern 工具，生成代码片段2.3(a)对应的程序依赖图，并导出 DOT 文件如5.2(a)所示，使用开源图形可视化工具 Graphviz 对 DOT 文件进行转化，可以得到如5.2(b)的图形。

```

1  digraph "main" {
2      *9* [label = <<METHOD_main></SUB>>]
3      *46* [label = <<METHOD_RETURN,int></SUB>>]
4      *12* [label = <<@lt;operator@gt;.assignment,i + 0></SUB>>]
5      *16* [label = <<@lt;operator@gt;.assignment,data[5]=(1,2,3,4,5)></SUB>>]
6      ...
7      *25* [label = <<@lt;operator@gt;.assignment,result = 0></SUB>>]
8      *41* [label = <<pxintf.printf(@quot;@dquot;,result)></SUB>>]
9      *44* [label = <<RETURN,return 0,></SUB>>]
10     *30* [label = <<@lt;operator@gt;.lessThan,i1lt;5></SUB>>]
11     *33* [label = <<@lt;operator@gt;.postIncrement,i++></SUB>>]
12     *45* [label = <<LITERAL_0,return 0></SUB>>]
13     *18* [label = <<@lt;operator@gt;.arrayInitializer,(1,2,3,4,5)></SUB>>]
14     *36* [label = <<@lt;operator@gt;.assignmentPlus,result += data[i]></SUB>>]
15     *38* [label = <<@lt;operator@gt;.indirectIndexAccess,data[i]></SUB>>]
16     *44* -> *46* [label = *DDG: @lt;RET@gt;]
17     *12* -> *46* [label = *DDG: i = 0*]
18     *16* -> *46* [label = *DDG: data*]
19     *16* -> *46* [label = *DDG: (1,2,3,4,5)*]
20     *16* -> *46* [label = *DDG: data[5]=(1,2,3,4,5)*]
21     *25* -> *46* [label = *DDG: result = 0*]
22     *30* -> *46* [label = *DDG: i*]
23     *30* -> *46* [label = *DDG: i1lt;5*]
24     ...
25     *30* -> *30* [label = *CDC: *]
26     *30* -> *38* [label = *CDC: *]
27     *30* -> *33* [label = *CDC: *]
28     *30* -> *36* [label = *CDC: *]
29 }

```



(a) 代码片段 1 对应的 PDG DOT 格式

(b) 代码片段 1 对应的 PDG 可视化

图 5.2 程序依赖图 DOT 文件示例

分析上图5.2(a)，程序依赖图 DOT 文件的描述包含两个部分：使用"number "[label = <function, name>] 来描述 PDG 点的特征，包括节点编号、节点的标签，其中标签内

还包括源代码语句中的变量名称、变量属性等信息。使用 "number" → "number" [label = "CDG/DDG:data"] 来描述 PDG 边的特征，包括边的起始点编号、边的终点标号、边代表的依赖关系（控制依赖用 CDG 表示、数据依赖用 DDG 表示）。使用开源图形可视化工具 Graphviz 得到的图 5.2(b) 中包含 13 个顶点，用实线箭头表示数据依赖边，虚线箭头表示控制依赖边。通过 DOT 描述语言存储生成的 PDG 能够很好的表现出代码片段的语法信息，表示出不同的节点和边的各种特征和属性，为下一步的过滤算法打好基础。

如果代码片段功能复杂，那么对应的程序依赖图规模也会很大，同时包含很多冗余边。例如上图 5.2(b) 中红框中的边，它们起始节点、终止节点均相同。右上角的红框对应 5.2(a) 中的 data 数组，数组内包含 5 个元素，对应含有 5 条边。实际上，这些边只有数值不同，属性相同，因此需要进行适当的优化来使得程序程序依赖图的结构精简，又不会丢失语义信息。

针对上述问题，本文设计了一种基于候选图对集合的图过滤算法，算法的伪代码如 3 所示。该算法有多个输入：候选程序依赖图集合 G 、PDG 有效行数阈值 L 、程序依赖图对规模比率 T ，输出为：经过过滤后的 PDG 对集合 R ，初该算法分为四个步骤：PDG 图结构化简、规模过滤、非同构判断、数值特征过滤，每个步骤的作用如下：

(1) PDG 图结构化简：首先对候选程序依赖图集合中的图进行化简，对起始节点、终止节点均相同的边进行化简，仅保留一条边，从而减少图的大小以便后续图匹配操作。下图 5.3 是化简后的结果。

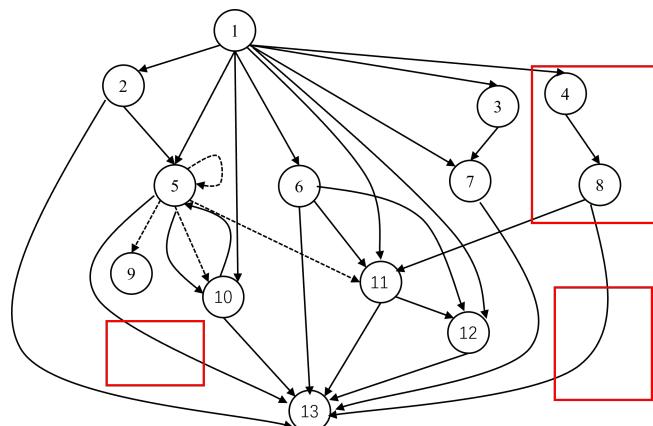


图 5.3 代码片段 1 经过图结构化简后的图

(2) 规模过滤：其次对图的规模进行判断，针对代码克隆检测，有研究证明，有

Algorithm 3 Graph filter algorithm (*filter_PDG*)

Input: PDG pairs: G
Input: The threshold: L
Input: The threshold of PDG pair's scale ratio: T
Input: The threshold of CV's string numerical similarity: G_s
Output: Candidate PDG pairs: R

```

1: initialization
2: for each PDG paris  $G_1, G_2$  in  $G$  do
3:   deleteSelfLoops( $G_1$ )
4:   deleteSelfLoops( $G_2$ )
5:   if sizeof( $G_1$ ) < L or sizeof( $G_2$ ) < L then
6:     PDG pair( $G_1, G_2$ ) is filtered
7:   else
8:     if min( $G_1, G_2$ ) / max( $G_1, G_2$ ) < T then
9:       if there is subgraph between ( $G_1, G_2$ ) then
10:         $R \leftarrow R \cup (G_1, G_2)$ 
11:       else
12:         PDG pair( $G_1, G_2$ ) is filtered
13:       end if
14:     else
15:       if number similarity of  $G_1, G_2 > G_s$  then
16:          $R \leftarrow R \cup (G_1, G_2)$ 
17:       else
18:         PDG pair( $G_1, G_2$ ) is filtered
19:       end if
20:     end if
21:   end if
22: end for
23: return  $R$ 

```

效行数小于 6 行的克隆程序本身并没有任何意义，并且会被绝大多数的代码克隆检测工具忽略。因为小于 6 行有效行的函数多为空函数，没有太大的参考意义。因此，本文也设置一个阈值，如果 PDG1 和 PDG2 中有任意一个 PDG 的有效行数小于阈值，那么就需要过滤掉这对 PDG。

(3) 非同构判断：接着对 PDG 对中是否存在子图同构进行判断。假设一对 PDG 对中，规模较大的图为 G_1 ，规模较小的图为 G_2 ，根据子图同构的概念，只要 G_1 和 G_2 满足以下两个条件之一：

$$\begin{aligned} \max(O(G_1)) &< \max(O(G_2)) \\ \min(I(G_1)) &< \min(I(G_2)) \end{aligned} \tag{5.1}$$

那么这两个图就不可能存在子图同构关系。其中 $O(G)$ 表示程序依赖图的节点出

度, $I(G)$ 表示程序依赖图的节点入度。如果这对 PDG 不存在子图同构, 但节点总数相差很大, 那么就需要过滤掉这对 PDG。

(4) 数值特征过滤: 最后通过收集 PDG 的简单特征来过滤掉明显不可能为克隆的 PDG 对。具体的, 根据 PDG 的节点个数、控制边数、执行边数、数据边数、声明节点数、函数调用数、传入参数、传出参数等代表特征进行过滤, 这些特征包含了代码一定的语义和语法信息。通过过滤掉差异很大的 PDG 对, 从而减少候选 PDG 对集合的规模。

5.2.3 程序依赖图表征学习设计

(1) 结构设计

为了提高程序依赖图维度代码表征能力, 本文选取图卷积网络对上述得到程序依赖图进行建模。具体的模型设计如图5.4所示。该模型主要包括输入层、多层卷积层、自注意力机制层、输出层。

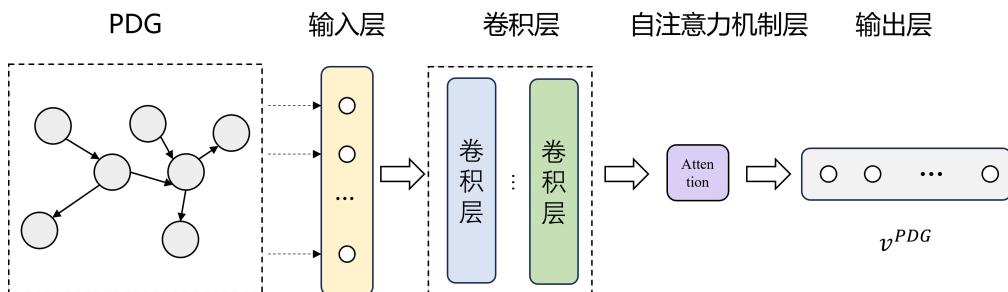


图 5.4 程序依赖图表征模型设计

①输入层: 输入层用于向模型输入训练数据, 在本方法中模型的输入为经过过滤算法判断得到的程序依赖图。②卷积层: 由多层卷积层堆叠构成, 通过卷积操作提取节点的特征, 同时考虑到节点之间的关联。③自注意力机制层: 本层的主要目的是捕捉序列中的长距离依赖关系, 通过直接计算任意两个位置的依赖关系, 并将每个代码片段缩减为一个单一的密集向量。④输出层: 每个程序依赖图对应一个输出。

(2) 模型选型

传统卷积神经网络中的卷积操作, 是指利用滑动窗口在输入数据上平移, 每次计算窗口内数据的有用特征, 从而达到特征提取的目的。然而, 这种操作只能处理图片、语言等结构规则的数据。现实中, 像图结构这种不规则的数据普遍存在, 其每个节点都拥有独特的邻接环境, 无法通过传统 CNN、RNN 的方法计算。因此, 有研究^[50]提

出图卷积神经网络。

在图结构中，每个节点都受到其邻居节点的影响，这种影响随着关系的紧密程度而变化，最终使节点达到稳定状态。图卷积神经网络（GCN）正是基于这样的思想，通过卷积操作来提取图中节点的特征。在 GCN 中，每个节点的特征表示不仅包含其自身的特征信息，还会考虑其所有邻居的特征信息，并通过学习得到的参数来更新。通过多层 GCN 的堆叠，可以逐步传播全局信息，实现对整个图的信息聚合和表示学习。相较于表真的 CNN，GCN 能够很好地处理不规则和非欧氏空间的数据，因此在社交网络、化学分子结构、交通网络等领域有广泛应用。值得一提的是，图卷积操作具有置换不变性，即不依赖于节点的具体排序，这符合图数据的本质特性。

图卷积神经网络是一种特殊的前馈神经网络结构，为减少网络中参数个数，用卷积层来代替传统的全连接层，提高神经网络的训练效率，卷积神经网络可以提取信息最多的数据特征，生成一个固定大小的向量表示结构。这样的设计有助于深入挖掘数据的语法和语义信息，使得图卷积神经网络在代码克隆检测等任务中表现出色。

因此，本文选用图卷积神经网络来构建程序依赖图的表征学习模型，该模型通过聚合节点自身的信息和邻居节点的信息，实现节点表示的更新，通过多层图卷积堆叠，实现全图信息聚合，进而提升程序依赖图的特征提取能力。

首先给出 GCN 的符号定义：对于图 $G = (V, E)$ ，其中 V 表示节点集，节点个数为 N ，对于每个节点 $v_i \in V$ ，均有其特征向量 x_i ，节点的特征向量特征用矩阵 $X \in R^{N \times C}$ 表示，其中 C 表示每个特征向量的维度； E 表示边的集合，如果节点 v_i 和 v_j 之间存在边，那么 $(v_i, v_j) \in E$ 表示。在卷积过程中涉及到三种矩阵表示，因此本文给出下图5.5示例。

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	1
C	0	0	0	1	0
D	0	1	0	0	0
E	0	0	0	0	0

	A	B	C	D	E
A	1	0	0	0	0
B	0	2	0	0	0
C	0	0	1	0	0
D	0	0	0	1	0
E	0	0	0	0	0

	A	B	C	D	E
A	1	-1	0	0	0
B	0	2	-1	0	-1
C	0	0	1	-1	0
D	0	-1	0	1	0
E	0	0	0	0	0

图 5.5 有向图矩阵示例

(1) 邻接矩阵 $A \in R^{N \times N}$: 用来表示图中节点之间关系的一种矩阵表示方法，如果

存在从节点 v_i 指向节点 v_j 的边, 那么 $A[i][j] = 1$, 如果不存在边, 则 $A[i][j] = 0$ 。(2) 度矩阵 $D \in R^{N \times N}$: 对角矩阵, 用于表示图中每个节点的度(即与该节点相连的边的数量)。出度是从该节点出发的边的数量, 入度是指向该节点的边的数量。度矩阵只有对角线上有值, 对于节点 v_i 有 $D[i][i] = \sum_j A_{ij}$, 其他位置的元素都是 0。对于有向图, 可以分别构建入度矩阵 D_{in} 和出度矩阵 D_{out} 。(3) 拉普拉斯矩阵 $L \in R^{N \times N} = D_{out} - A$: 通过矩阵的形式描述了图的拓扑结构和性质, 在图的传播和扩散问题中起到关键作用, 通过特征值分解得到图的特征向量, 有助于图的划分和聚类。

通过引入邻接矩阵和度矩阵, 图卷积神经网络能够捕获图的拓扑结构信息, 这是处理图数据的关键。该模型的输入 $X \in R^{N \times C}$ 为节点特征矩阵、 $A \in R^{N \times N}$ 为邻接矩阵, 输出是一个特征矩阵 $Z \in R^{N \times F}$, 表示学习到的每个节点的特征表示, F 为模型自己定义的输出向量维度。下图5.6为图卷积神经网络结构图。

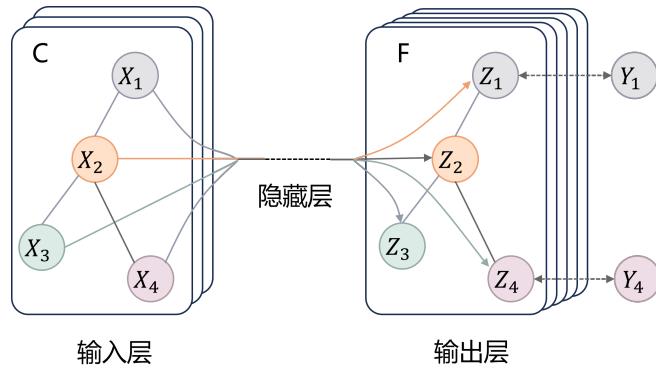


图 5.6 图卷积神经网络结构图

中间每一个图卷积层的输入都是邻接矩阵 A 和节点特征 X , 层与层之间的传播公式都可以写成5.2;

$$X^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X^l W^l) \quad (5.2)$$

其中: X^l 表示第 l 层的节点特征矩阵。 W^l 是第 l 层的参数, 即可学习的权重矩阵。 σ 是激活函数, 如 ReLU。 $\tilde{A} = A + I$, 其中 I 是单位矩阵, 表示给每个节点添加一条到自身的边, 聚合时可以结合自身的特征消息。 \tilde{D} 是 \tilde{A} 的度矩阵, 其公式为 $D[\tilde{i}][i] = \sum_j \tilde{A}_{ij}$ 。对邻接矩阵 \tilde{A} 进行归一化操作 $\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$ 是为了信息传递的过程中保持特征矩阵 X 的原有分布, 那些具有较低度的节点会对其邻居有较大的影响, 而具有高度的节点会产生更小的影响因为他们的影响会被分散给很多的邻居节点。

5.3 PDG 表征方法具体实现

在介绍具体实现之前，本节首先给出 PDG 表征方法的输入：经过2.2.2小节的代码预处理阶段，得到示例代码片段2.3对应的程序依赖图，如图5.7所示。图中的实线表示节点之间的控制依赖，虚线表示节点之间的数据依赖。仔细分析三张图，其中图5.7(a)和图5.7(b)红框中节点 11、12 虽然位置有所改变，但是其边依赖均相同，因此可以视为完全相同的同构图；而图5.7(c)因为增加了一个 if 语句，图中也增加了一个节点，同时添加的红色的线表示新增的数据依赖、控制依赖。

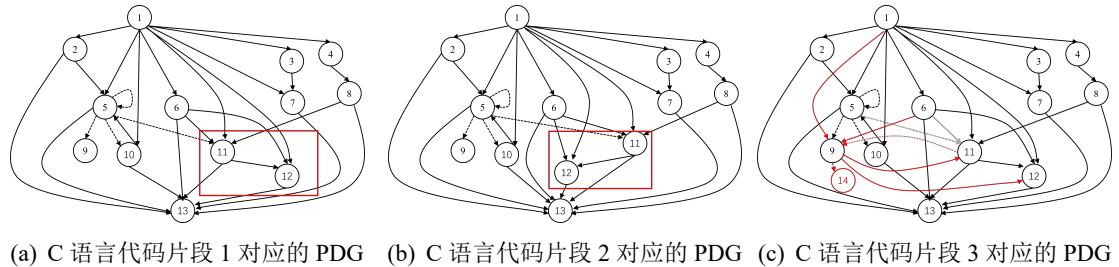


图 5.7 示例源代码对应的程序依赖图

接下来，本章提出的基于图过滤的程序依赖图表征学习方法的实现如图5.8所示。该方法的输入是一对代码片段 C_a, C_b 对应的程序依赖图，表示为 PDG_a, PDG_b ，输出是 C_a, C_b 对应的语义特征向量 V_a^{PDG}, V_b^{PDG} ，整体采用 Siamese 架构，两个子网络共享权值，从下到上，主要包括图过滤判断、图表征三个阶段。

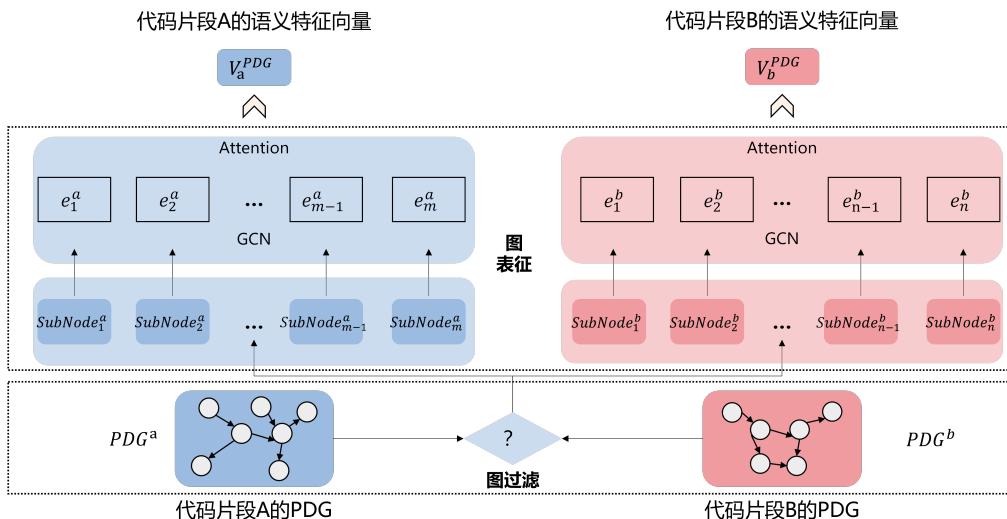


图 5.8 基于图过滤的程序依赖图表征学习方法实现

在图过滤阶段，将输入的 PDG 对进行过滤判定。首先对程序依赖图进行图结构约简，然后判断其有效行数是否大于 6，是否存在子图同构，最后再判断两者的数值特征相似度是否大于阈值。如果顺利上述四个步骤，则该 PDG 对输入图表征模型。其过滤过程可以表达为公式5.3:

$$(x_1^a, x_2^a, \dots, x_m^a) = filter_PDG(PDG_a) \quad (5.3)$$

在图表征模型，将 PDG 图作为输入，使用图卷积神经网络进行编码。处理过程分为两个步骤，如公式5.4所示。

$$\begin{aligned} e_1^{aGCN}, e_2^{aGCN}, \dots, e_n^{aGCN} &= GCN(x_1^a, x_2^a, \dots, x_m^a) \\ V_a^{PDG} &= Attention(e_1^{aGCN}, e_2^{aGCN}, \dots, e_n^{aGCN}) \end{aligned} \quad (5.4)$$

经过图卷积编码后，生成了 e_i^{aGCN} ，然后经过 Attention 层的总结后，生成了 V_a^{PDG} 作为代码片段 C_a 的最终图表示，即语义特征向量。同样，可以使用相同的计算以通过图过滤判断的 PDG 作为输入为代码片段 C_b 计算 V_b^{PDG} 。

5.4 实验验证

为了验证基于图过滤的程序依赖图表征学习方法的有效性，本节开展实验验证。首先，介绍了实验的具体设计，接着对子树划分、树表征模型进行消融实验。

5.4.1 实验设计

本节使用与3.4节中同样的实验环境和数据集对基于图过滤的程序依赖图表征学习方法进行对比实验。使用代码分析工具 Joern 获取数据集中代码片段的程序依赖图，然后使用图卷积神经网络训练图嵌入，并将嵌入向量大小设置为 128。模型的 batch_size 为 32，优化器为 Adam 优化器，学习率为 0.002，克隆检测的阈值设置为 0.5。同样选取常用的精确率（Precision）、召回率（Recall）、F1 值作为评估指标。

5.4.2 实验结果

消融对比实验：体现图过滤机制的有效性

基于 PDG 的 GCN

基于 PDG 的 + 图过滤的 GCN

表 5.1 图过滤机制实验结果

对比	P	R	F1
基于 PDG 的 GCN	0.xx	0.xx	0.xx
基于 PDG 的 + 图过滤的 GCN	0.xx	0.xx	0.xx

5.5 本章小结

本章主要对 RLCCD 中基于图过滤的程序依赖图表征学习方法的设计与实现进行详细阐述。首先介绍了程序依赖图维度的研究动机，其次介绍了程序依赖图表征学习的方法设计，具体论述了其整体框架、图过滤、图表征学习，接着开展实验验证，结果表明了此方法的有效性和模型的准确性。

第 6 章 特征融合及 RLCCD 框架验证

本章主要对面向代码克隆检测的多维源代码表征方法整体框架 RLCCD 进行介绍，同时进行实验评估及验证。具体地，RLCCD 是由上述基于预训练辅助模型的 Token 表征学习、基于子树划分的抽象语法树表征学习、基于图过滤的程序依赖图表征学习方法三种维度融合形成并进行实现的表征方法，最后通过与 SourcererCC、ASTNN、SCDetector 进行对比实验以验证该框架的有效性。

6.1 特征融合

特征融合方法是指将不同来源或不同层次的特征进行组合，合并成一个比输入特征更具有判别能力的特征，该多维特征能够在低维空间中高效计算实体和关系的语义联系，提高特征的表达能力和分类效果，有利于下游代码克隆检测任务的学习。

因此，经过第3、4、5章的表征学习，得到三种维度的特征向量：属性特性 V^{Token} 、结构特征 V^{AST} 、语义特征 V^{PDG} 。上述三种表征方式得到的特征向量通常具有信息互补性，且不同维度的特征是代码表示的平行语料，具有信息等价性，因此，本文提出了基于多模态学习的特征融合方法，将三个维度的特征向量通过特征融合生成多维表示。然后将多维表示送入单层线性网络。本文设计了两个特征融合方式：特征连接 (Feature concatenation) 和特征加法 (Feature addition)，如下图6.1所示。

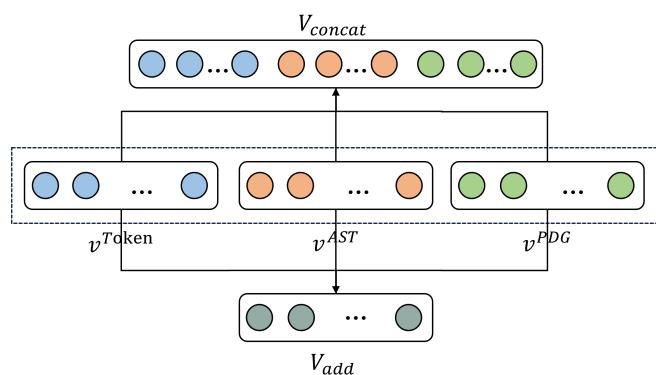


图 6.1 特征融合方法

其中特征连接是指直接将两个特征张量在某个维度上连接在一起，生成一个更大的张量。例如两个输入特征 x 和 y 的维数若为 p 和 q ，输出特征 z 的维数为 $p+q$ 。在连接时，采取串联策略，两个张量的维度必须保持一致。而特征加法是指将两个特征

张量按照元素相加在一起，生成一个新的张量。采取并行策略，特征向量本身的维度并没有增加。因此可以用公式6.1表示：

$$\begin{aligned}\tilde{V}_{concat} &= concat(V^{\text{Token}}, V^{\text{AST}}, V^{\text{PDG}}) \\ \tilde{V}_{add} &= add(V^{\text{Token}}, V^{\text{AST}}, V^{\text{PDG}}) \\ V &= W_{dt} \cdot \tilde{V} + b_{dt}\end{aligned}\tag{6.1}$$

其中 V 表示最终的多维混合代码表示, W_{dt} 在生成的多维混合表示中平衡属性特征、结构特征、语义特征的组成，而 b_{dt} 在训练模型时使模型偏向最终收敛。

6.2 RLCCD 框架验证

本文的实验设计主要围绕以下 4 个方面的研究问题：

- RQ1：本文提出的预训练辅助模型策略是否优于基线方法？（见 3.3 节）
- RQ2：本文提出的子树划分策略是否优于基线方法？（见 4.3 节）
- RQ3：本文提出的图过滤策略能否优于基线方法？（见 5.3 节）
- RQ4：与现有代码克隆检测工具相比，RLCCD 表现如何？

6.2.1 实验设置

(1) 系统环境

本章实验均在 Ubuntu 16.04 LTS (64 位) 系统下进行。

(2) 实验数据集

本章实验部分采用的数据集为代码克隆检测领域常见的基准集 POJ104，该数据集是一个基于 C 语言构建的大型数据集，包含了 104 个编程问题以及学生提交的对应问题的不同 C 语言源代码，在该数据集中，针对同一问题的不同源解法的代码被视为一个克隆对。

6.2.2 对比工具

在对比整个方法的效果时，本文选取开源的 SourcererCC、ASTNN、SCDetector 方法进行比较。

SourcererCC: SourcererCC 是一种相对较新的基于 token 的克隆检测工具。该工具通过词袋模型，把收集的数据全部编码成词频信息，后将代码行转换成一个由词频构成的向量，通过向量的比较获取相似度。

表 6.1 RLCDD 实验结果

对比	P	R	F1
SourcererCC	0.11	0.43	0.17
ASTNN	0.87	0.95	0.91
SCDetector	0.97	0.81	0.82
RLCDD	0.xx	0.xx	0.xx

ASTNN: 一种基于神经网络的源代码表示方法。它将整个抽象语法树 AST 分解成一系列小型语句树，并通过捕获语句的词法和语法信息将语句树分别编码为向量，最后采用了 RNN 模型生成代码片段的向量表示。ASTNN 方法完整保留了抽象语法树的结构信息，能够检测到所有类型的代码克隆。

SCDetector: 是基于令牌和基于图的方法的结合。给定一个方法源代码，我们首先生成 CFG，然后应用中心性分析将图转换为某些语义标记（即具有图细节的标记）。最后，这些语义标记被馈送到 Siamese 网络中，以训练模型并使用它来检测代码克隆对。

6.2.3 RLCDD 性能评估实验结果

对比实验：体现框架的有效性

6.3 本章小结

本章节主要对 RLCDD 框架进行了实验验证，以验证 RLCDD 的 XXX 能力。

结论

本文采用……。（结论作为学位论文正文的最后部分单独排写，但不加章号。结论是对整个论文主要结果的总结。在结论中应明确指出本研究的创新点，对其应用前景和社会、经济价值等加以预测和评价，并指出今后进一步在本研究方向进行研究工作的展望与设想。结论部分的撰写应简明扼要，突出创新性。）

参考文献

- [1] 乐乔艺, 刘建勋, 孙晓平, 等. 代码克隆检测研究进展综述[J]. 计算机科学, 2021, 48(S02): 14.
- [2] 孙祥杰, 魏强, 王奕森, 等. 代码相似性检测技术综述[J]. 计算机应用, 2023: 1-14.
- [3] Synopsys, Inc. 2024 Open Source Security and Risk Analysis Report[Z]. <https://www.synopsys.com/zh-cn/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html>. 2024.
- [4] Moore S. 7 Top Trends in Cybersecurity for 2022[Z]. <https://www.gartner.com/en/articles/7-top-trends-in-cybersecurity-for-2022>. April 13, 2022.
- [5] Svajlenko J, Roy C K. Evaluating clone detection tools with BigCloneBench[C]. 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2015: 131-140.
- [6] Dang Y, Zhang D, Ge S, et al. Transferring Code-Clone Detection and Analysis to Practice[C]. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). 2017: 53-62.
- [7] Yang J, Hotta K, Higo Y, et al. Classification model for code clones based on machine learning[J]. Empirical Software Engineering, 2015, 20: 1095-1125.
- [8] 陈秋远, 李善平, 鄢萌, 等. 代码克隆检测研究进展[J]. 软件学报, 2019, 30(4): 19.
- [9] Bengio, Yoshua, Courville, et al. Representation Learning: A Review and New Perspectives[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2013, 35(8): 1798-1828.
- [10] Wikipedia. Feature learning[Z]. https://en.wikipedia.org/wiki/Feature_learning.
- [11] Pearson K. On lines and planes of closest fit to systems of points in space.[J]. PHILOSOPHICAL MAGAZINE, 1901, 2(7-12): 559-572.
- [12] Fisher R A. THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS[J]. Annals of Human Genetics, 2012, 7(7): 179-188.
- [13] Goodfellow I, Bengio Y, Courville A. Deep Learning[M]. MIT Press, 2016.
- [14] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilingual token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.
- [15] Li Z, Lu S, Myagmar S, et al. CP-Miner: finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on Software Engineering, 2006, 32(3): 176-192.
- [16] Jiang L, Su Z, Chiu E. Context-based detection of clone-related bugs[C]. ESEC-FSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007: 55-64.
- [17] Pennington J, Socher R, Manning C. GloVe: Global Vectors for Word Representation[C]. Moschitti A, Pang B, Daelemans W. Proceedings of the 2014 Conference on Empirical Methods in Natural Lan-

- guage Processing (EMNLP). Doha, Qatar: Association for Computational Linguistics, 2014: 1532-1543.
- [18] Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[C]. Burstein J, Doran C, Solorio T. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Minnesota: Association for Computational Linguistics, 2019: 4171-4186.
- [19] Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: Scaling Code Clone Detection to Big-Code[C]. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 2016: 1157-1168.
- [20] Tao C, Zhan Q, Hu X, et al. C4: Contrastive Cross-Language Code Clone Detection[C]. 2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC). 2022: 413-424.
- [21] Yourdon E. Structured programming and structured design as art forms[C]. AFIPS '75. Anaheim, California: Association for Computing Machinery, 1975: 277.
- [22] White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection[J]. 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016: 87-98.
- [23] Mou L, Li G, Zhang L, et al. Convolutional Neural Networks over Tree Structures for Programming Language Processing[C]. AAAI Conference on Artificial Intelligence: THIRTIETH AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE. 2016: 1287-1293.
- [24] Wei H H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code[C]. IJCAI'17. Melbourne, Australia: AAAI Press, 2017: 3034-3040.
- [25] Zhang J, Wang X, Zhang H, et al. A Novel Neural Source Code Representation Based on Abstract Syntax Tree[C]. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019: 783-794.
- [26] Yu H, Lam W, Chen L, et al. Neural Detection of Semantic Code Clones Via Tree-Based Convolution[C]. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). 2019: 70-80.
- [27] Ling H, Zhang A, Yin C, et al. Improve Representation for Cross-Language Clone Detection by Pretrain Using Tree Autoencoder[J]. Intelligent Automation & Soft Computing, 2022.
- [28] Wu Y, Feng S, Zou D, et al. Detecting Semantic Code Clones by Building AST-based Markov Chains Model[C]. ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. <conf-loc>, <city>Rochester</city>, <state>MI</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023.

- [29] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[J]. ACM Trans. Program. Lang. Syst., 1987, 9(3): 319-349.
- [30] Allamanis M, Brockschmidt M, Khademi M. Learning to Represent Programs with Graphs[J]. ArXiv, 2017, abs/1711.00740.
- [31] Lu M, Tan D, Xiong N N, et al. Program Classification Using Gated Graph Attention Neural Network for Online Programming Service[J]. ArXiv, 2019, abs/1903.03804.
- [32] Brockschmidt M, Allamanis M, Gaunt A L, et al. Generative Code Modeling with Graphs[J]. ArXiv, 2018, abs/1805.08490.
- [33] Ben-Nun T, Jakobovits A S, Hoefler T. Neural code comprehension: a learnable representation of code semantics[C]. NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems. Montréal, Canada: Curran Associates Inc., 2018: 3589-3601.
- [34] Wang W, Li G, Ma B, et al. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree[C]. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2020: 261-271.
- [35] DeFreez D, Thakur A V, Rubio-González C. Path-based function embedding and its application to error-handling specification mining[C]. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018: 423-433.
- [36] Yang K, Yu H, Fan G, et al. A graph sequence neural architecture for code completion with semantic structure features[J]. Journal of Software: Evolution and Process, 2021, 34.
- [37] Tufano M, Watson C, Bavota G, et al. Deep Learning Similarities from Different Representations of Source Code[J]. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), 2018: 542-553.
- [38] Saini V, Farmahinifarahani F, Lu Y, et al. Oreo: detection of clones in the twilight zone[C]. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018: 354-365.
- [39] Fang C, Liu Z, Shi Y, et al. Functional code clone detection with syntax and semantics fusion learning[J]. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020.
- [40] Hua W, Sui Y, Wan Y, et al. FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks[J]. IEEE Transactions on Reliability, 2020, 70: 304-318.
- [41] Dong W, Feng Z, Wei H, et al. A Novel Code Stylometry-based Code Clone Detection Strategy[C]. 2020 International Wireless Communications and Mobile Computing (IWCMC). 2020: 1516-1521.
- [42] Feng Z, Guo D, Tang D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages[J]. ArXiv, 2020, abs/2002.08155.

- [43] Duan Y, Li X, Wang J, et al. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing[C]. 27TH ANNUAL NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM (NDSS 2020). 2020.
- [44] Wu Y, Zou D, Dou S, et al. SCDetector: software functional clone detection based on semantic tokens analysis[C]. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021: 821-833.
- [45] 吕泉润, 谢春丽, 万泽轩, 等. 基于对比学习的跨语言代码克隆检测方法[J]. 计算机应用研究, 2024: 1-7.
- [46] 冷林珊, 刘爽, 田承霖, 等. 预训练增强的代码克隆检测技术[J]. 软件学报, 2022, 33: 1758-1773.
- [47] Zhao W X, Zhou K, Li J, et al. A Survey of Large Language Models[Z]. 2023.
- [48] Bui N D Q, Yu Y, Jiang L. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees[C]. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021: 1186-1197.
- [49] Guo D, Ren S, Lu S, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow[Z]. 2021.
- [50] Kipf T N, Welling M. Semi-Supervised Classification with Graph Convolutional Networks[Z]. 2017.
- [51] IJaDataset2.0. Ambient Software Evolution Group[Z]. <http://secold.org/projects/seclone>. January 2013.
- [52] Rumelhart D E, Hinton G E, Williams R J. Learning Representations by Back Propagating Errors[J]. Nature, 1986, 323(6088): 533-536.
- [53] Hinton G E, Osindero S, Teh Y W. A Fast Learning Algorithm for Deep Belief Nets[J]. Neural Computation, 2006, 18(7): 1527-1554.
- [54] Chen L, Ye W, Zhang S. Capturing source code semantics via tree-based convolution over API-enhanced AST[C]. CF '19: Proceedings of the 16th ACM International Conference on Computing Frontiers. Alghero, Italy: Association for Computing Machinery, 2019: 174-182.
- [55] 王亚芳. 基于图像相似度检测代码克隆[D]. 内蒙古师范大学, 2020.
- [56] Alomari H W, Stephan M. Clone Detection through srcClone: A Program Slicing Based Approach[J]. Journal of Systems and Software, 2022, 184: 111115.

攻读学位期间发表论文与研究成果清单

- [1] Doe J, **Zhang S.** The Book with Title[M]. Dummy Publisher, 2000. (已出版.)
- [2] 张三, 李杰, 罗运军. 交联型与线形水性聚氨酯的形状记忆性能比较[J]. 化工进展, 2006(01): 78-81. (EI 收录, 检索号:06129773469. 已刊出.)
- [3] 李杰, 张三, 罗运军. 交联型与线形水性聚氨酯的形状记忆性能比较[J]. 化工进展, 2007(01): 78-81. (EI 收录, 检索号:06129773469. 已刊出.)
- [4] **Zhang S.** The Book without Title[M]. Dummy Publisher, 2100. (已出版.)

致谢

时间飞逝，转眼临近研究生毕业。当这篇文章正式落笔，就意味着我在北京理工大学的时光走到了尾声。回顾这段时间的工作学习，在课堂上学到的理论知识被强化、加深、理解，从而提高了我的专业知识素养，成为我人生的重要财富。在这里，我要向所有帮助过我的老师同学表示衷心的感谢。

首先，非常真挚地感谢我的导师马锐。马老师，从毕业设选题、撰写开题报告、资料整理、进度反馈、论文修改的过程中，一直耐心指导，不厌其烦地帮助我克服困难。马老师也腾出了很多个人时间，给我指导，使我可以顺利完成我的毕业论文。在这里，我真诚地祝愿马老师工作顺利、身体健康、万事顺意。

感谢北京理工大学对我的培养教育，感谢它为我创造了一个环境优美、治学严谨、学风浓厚的校园氛围。在我迷茫时，可以去图书馆寻找知识解答疑惑；当我伤心时，可以去中心公园散步平复心情得到治愈；当我开心时，可以去食堂选一份美食来庆祝。我在北京理工大学度过了愉快、幸福的三年，在这里，也希望北理可以不断进步，未来我可以重回校园，为母校的发展感到无比自豪。

感谢父母对我的支持和照顾，感谢他们多年来无微不至的照顾和坚定不移的支持。不管我在哪里，他们永远是我最坚强的后盾。当我遇到困难时，给我最温柔的建议和最坚定的鼓励。无论是在物质上，还是精神上，他们的支持与付出，鼓励着我坚定求学的决心，无畏地继续奔向美好地未来。

感谢同学对我的鼓励陪伴是大家让我收获到了友谊。我们在外出求学的生涯里相互扶持，用真挚的心换来坚定的情。虽然我们来自五湖四海，但我们在北京理工大学相遇，北京理工大学也见证了我们一起走过的四年。

最后，再次向各位表示真挚的谢意，感谢在我前进的路上，有你们的参与