软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(4):1601−1617 [doi: 10.13328/j.cnki.jos.007016]

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

基于多模态对比学习的代码表征增强预训练方法*

杨宏宇[1,2,4], 马建辉[1,2,4], 侯 旻[1,3,4], 沈双宏[1,3,4], 陈恩红[1,3,4]

[1](大数据分析与应用安徽省重点实验室(中国科学技术大学), 安徽 合肥 230027)

[2](中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230027)

[3](中国科学技术大学 大数据学院, 安徽 合肥 230027)

[4](认知智能全国重点实验室, 安徽 合肥 230088)

通信作者: 马建辉, E-mail: jianhui@ustc.edu.cn

摘 要: 代码表征旨在融合源代码的特征, 以获取其语义向量, 在基于深度学习的代码智能中扮演着重要角色. 传统基于手工的代码表征依赖领域专家的标注, 繁重耗时, 且无法灵活地复用于特定下游任务, 这与绿色低碳的发展理念极不相符. 因此, 近年来, 许多自监督学习的编程语言大规模预训练模型(如 CodeBERT)应运而生, 为获取通用代码表征提供了有效途径. 这些模型通过预训练获得通用的代码表征, 然后在具体任务上进行微调, 取得了显著成果. 但是, 要准确表示代码的语义信息, 需要融合所有抽象层次的特征(文本级、语义级、功能级和结构级). 然而, 现有模型将编程语言仅视为类似于自然语言的普通文本序列, 忽略了它的功能级和结构级特征. 因此, 旨在进一步提高代码表征的准确性, 提出了基于多模态对比学习的代码表征增强的预训练模型(representation enhanced contrastive multimodal pretraining, REcomp). REcomp 设计了新的语义级-结构级特征融合算法, 将它用于序列化抽象语法树, 并通过多模态对比学习的方法将该复合特征与编程语言的文本级和功能级特征相融合, 以实现更精准的语义建模. 最后, 在 3 个真实的公开数据集上进行了实验, 验证了 REcomp 在提高代码表征准确性方面的有效性.

关键词: 代码表征; 预训练模型; 多模态; 对比学习

中图法分类号: TP18

中文引用格式: 杨宏宇, 马建辉, 侯旻, 沈双宏, 陈恩红. 基于多模态对比学习的代码表征增强预训练方法. 软件学报, 2024, 35(4): 1601−1617. http://www.jos.org.cn/1000-9825/7016.htm

英文引用格式: Yang HY, Ma JH, Hou M, Shen SH, Chen EH. Pre-training Method for Enhanced Code Representation Based on Multimodal Contrastive Learning. Ruan Jian Xue Bao/Journal of Software, 2024, 35(4): 1601−1617 (in Chinese). http://www.jos.org.cn/1000-9825/7016.htm

Pre-training Method for Enhanced Code Representation Based on Multimodal Contrastive Learning

YANG Hong-Yu[1,2,4], MA Jian-Hui[1,2,4], HOU Min[1,3,4], SHEN Shuang-Hong[1,3,4], CHEN En-Hong[1,3,4]

[1](Anhui Province Key Laboratory of Big Data Analysis and Application (University of Science and Technology of China), Hefei 230027, China)

[2](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

[3](School of Data Science, University of Science and Technology of China, Hefei 230027, China)

[4](State Key Laboratory of Cognitive Intelligence, Hefei 230088, China)

Abstract: Code representation aims to extract the characteristics of source code to obtain its semantic embedding, playing a crucial role in

本文由“绿色低碳机器学习研究与应用”专题特约编辑封举富教授、俞扬教授、刘淇教授推荐.

收稿时间: 2023-05-15; 修改时间: 2023-07-07; 采用时间: 2023-08-24; jos 在线出版时间: 2023-09-11

CNKI 网络首发时间: 2023-11-28

deep learning-based code intelligence. Traditional handcrafted code representation methods mainly rely on domain expert annotations, which are time-consuming and labor-intensive. Moreover, the obtained code representations are task-specific and not easily reusable for specific downstream tasks, which contradicts the green and sustainable development concept. To this end, many large-scale pretraining models for source code representation have shown remarkable success in recent years. These methods utilize massive source code for self-supervised learning to obtain universal code representations, which are then easily fine-tuned for various downstream tasks. Based on the abstraction levels of programming languages, code representations have four level features: text level, semantic level, functional level, and structural level. Nevertheless, current models for code representation treat programming languages merely as ordinary text sequences resembling natural language. They overlook the functional-level and structural-level features, which bring performance inferior. To overcome this drawback, this study proposes a representation enhanced contrastive multimodal pretraining (REcomp) framework for code representation pretraining. REcomp has developed a novel semantic-level to structure-level feature fusion algorithm, which is employed for serializing abstract syntax trees. Through a multi-modal contrastive learning approach, this composite feature is integrated with both the textual and functional features of programming languages, enabling a more precise semantic modeling. Extensive experiments are conducted on three real-world public datasets. Experimental results clearly validate the superiority of REcomp.

Key words: code representation; pre-trained model; multimodal; contrastive learning

由于软件的无处不在和软件版本的快速迭代, GitHub, StackOverflow 等开源代码仓库中的源代码爆发式增长, 代码大数据时代[1]已经来临. 据 Evans Data Corporation[2]报告预计, 2024 年, 专业开发人员数会将增加至 2 870 万. 因此, 如何有效利用海量的源代码相关数据来提高专业开发人员的工作效率, 进而推动代码相关任务的发展(称作代码智能), 已经成为软件工程领域(SE)的研究热点. 要实现代码智能, 代码表征是关键. 代码表征[3]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, 例如理解型[4]任务(代码克隆检测、语义代码检索等). 最初, 传统的基于手工的方法[5]利用统计学原理表征代码[6]. 后来, 大量的机器学习算法[7]被运用到代码表征中, 但仍然需要手工从源代码中提取代码特征[8]. 无论是传统的手工方法还是基于机器学习的方法都依赖领域专家的标注, 繁重耗时, 且无法灵活地应用于特定下游任务, 这与绿色低碳的发展理念极不相符. 为解决这个问题, 近年来, 有学者开始将基于深度学习的模型, 如循环神经网络(recurrent neural network, RNN)[9]、卷积神经网络(convolutional neural network, CNN)[10−13]和图神经网络(graph neural network, GNN)[14,15]等引入到代码表征中, 试图表征隐藏在源代码中的更深层次的复杂特征, 进一步提高代码表征的准确性. 目前, 基于深度学习的方法已经广泛应用于各种代码智能任务.

根据源代码的抽象层次从低到高划分[16], 代码具有以下特征: 文本级特征是以自然语言方式概括函数功能的语句, 如代码注释; 语义级特征是经过词法分析的字符序列, 如代码字符; 功能级特征是以编程语言方式概括函数功能的序列, 如函数方法名和应用程序接口(application programming interface, API); 结构级特征是表示代码句法信息的树形或者图形结构, 如抽象语法树(abstract syntax tree, AST)、数据流图(data flow graph, DFG)、控制流图(control flow graph, CFG)和程序依赖图等. 由于抽象程度越高, 能够提取的信息越多, 所以大部分研究更关注结构级特征. 近年来, 对结构级特征的研究主要集中在 AST 上. 因为 AST 含有代码完整的结构信息, 而其他形式的结构表示, 如 DFG 和 CFG 都是从 AST 中提取的. 此外, 如果对代码只做了语义级的字符改变(如修改变量名称或者改变代码风格), 而不改变程序的执行结构, 那么修改后的代码与原始代码在结构信息上的表征是不变的. 因此, 可以利用这一点来实现跨语言表征, 有效避免不同编程语言的语法规则差异问题. 然而, AST 的树形结构不适用于通用的 Seq2Seq 模型, 因此, AST 的序列化算法成为研究 AST 的主要方向. 早期, TBCNN[10]使用基于树结构的 CNN[17]来表征抽象语法树. 为解决 CNN 的长序列学习依赖问题, Code2tree[18]则使用 GRU 编码器对 AST 序列进行编码, 并在解码阶段引入注意力机制以提高代码注释的质量. CDLH[19]框架引入了基于树结构的 LSTM[20]来挖掘词法信息和句法信息. Code2vec[21]将 AST 分解成一组路径, 并通过简单的全连接层学习如何聚合路径的原子表示. 后来, mrncs[22]提出了一种基于结构遍历的简化 AST 的方法, 保留关键句法信息的同时, 得到简化的 AST(称作 ISBT). 但是, 现有的大多数预训练模型对 AST 的提取并不充分, 存在一些问题, 如图 1 所示. 这些问题包括结点信息的冗余, 即所有结点均出现不止一次, 例如结点 A 出现了 2 次; 还有在序列化过程中额外开销的引入, 即序列中用大量的括号来表示一棵子树, 例如子树 B 的完整表示是“(B(D)***D***(E)***E***(F)***F***)”.

近年来, 大规模语言模型已成为自然语言处理及其相关领域的强大工具, 展现出惊艳的性能和多样的应用潜力. 这些模型的成功, 促使研究者开始思考: 是否能将其强大的表示学习能力应用于代码智能领域? 同时, 编程语言(programming language, PL)和自然语言(natural language, NL)有诸多相似之处: 都具有可重复并带有可预测的统计学规律, 所以自然语言处理领域中很多的方法在代码智能领域同样适用. 于是, 研究者尝试将自然语言处理中能够自监督学习的预训练模型, 如 Transformer[23], BERT[24], RoBERTa[25], GPT[26], BART[27]等应用到代码表征中. 为此, 编程语言预训练模型[28−32]被提出, 并逐渐统一了生成类、理解类和自回归任务,得到的代码的通用表示能在下游任务上复用. PLBART[28]是 BART[27]在编程语言方面的变体, 将文本级的注释或者语义级的代码字符单独作为代码的特征进行预训练. CodeBERT[29]是 BERT 在编程语言上的双模态扩展,将注释和代码字符融合在同一个序列中以表征代码. GraphCodeBERT[30]在 CodeBERT 基础上进一步添加了数据流图的变量序列作为代码结构级特征的补充表示, 并设计了数据流边预测和变量结点对齐等预训练任务.然而, 在从 AST 中提取数据流时, 只考虑了存在数值流动关系的代码字符, 这导致部分 AST 结构信息的遗漏.针对数据遗漏的现象, SynCoBERT[31]直接使用完整的 AST 作为结构信息的代表, 并将文本级、语义级和结构级特征融合在一个输入序列进行预训练, 同时设计了多模态对比学习、AST 边预测和标识符预测等 3 个预训练任务. 随后, UniXcoder[32]提出了新的融合 AST 和代码字符的一对一映射算法, 以引入结构级特征并缩短特征序列. 虽然这些编程语言预训练模型在代码智能领域取得了一定的效果, 但它们尚未完全利用代码的各个层次特征(文本级、语义级、功能级和结构级). 目前, 集成所有特征面临以下挑战: 如何减少不必要的结点引入以及如何有效缩短特征序列长度, 同时节省存储、传输和计算资源. 目前尚未有一个预训练模型能够集成所有特征, 现有模型各自涵盖的特征层次各不相同. 因此, 对于综合多个层次特征的编程语言预训练模型, 仍然需要进一步的研究和探索.

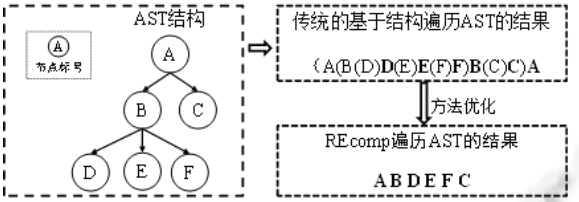


图 1 REcomp 优化的 AST 序列化算法的示例

为了充分挖掘代码各个层次的特征并更加准确地表征代码的语义信息, 本文提出了一个新的编程语言预训练模型, 称作 REcomp. 它旨在融合代码的文本级的注释、语义级的代码字符、功能级的函数名和结构级的 AST, 并通过设计多模态对比学习的预训练方法来学习这 4 个特征之间的语义等价性. 与其他领域有所不同, 在代码智能研究中, 将代码不同视角或层次的特征称为代码不同的模态, 因此, 多个特征的融合称为多模态表征. 之所以选择多模态对比学习的方法对代码进行建模, 是因为它可以通过共享学习的方式将代码的不同模态特征进行交叉融合, 从而进一步提高模型表征代码语义的能力. 这种共享学习的方式不仅减少了多个单一模态模型的训练和推理过程, 也减少了计算资源的开销. 此外, 本文还设计了两种新的数据增强方法, 即编程语言型(PL 型)和自然语言型(NL 型), 以此构建正负样本进行对比学习.

最后, 本文在公开基准 CodeXGLUE[2]数据集上进行了实验, 验证了所提模型的有效性. REcomp 很大程度地减少了资源的的消耗, 仅使用了极少的计算资源就达到了极快的训练速度, 并且在极短的时间内收敛. 以初始化模型 UniXcoder 为例, REcomp 仅使用了约 UniXcoder 算力资源的 1/20, 就将训练至收敛的时间约缩短为 UniXcoder 的 1/32. 在语义代码检索任务上, REcomp 在 CodeSeachNet[33]数据集上提升了约 1.9%的准确率; 在代码克隆检测任务上, REcomp 在 BigCloneBench[34]数据集和 POJ-104[35]数据集分别取得了 4%和 0.9%准确率的提升. 本文的主要贡献如下:

(1) 设计了一种优化的 AST 序列化方法, 它在保留 AST 中控制结点信息并且不破坏树结构的前提下, 按序地将非叶结点的结构信息和叶结点的语义信息融合, 得到语义级-结构级的复合特征. 相比

SBT[13], 此算法消除了结点冗余和括号重复的现象, 极大程度地缩短了 AST 特征序列, 更简短且精准地表示 AST;

(2) 设计了两种特别的数据增强方法: PL 型和 NL 型, 集成编程语言 4 个不同层级的特征. PL 型旨在学习符合编程语言特点的特征的语义等价性, NL 型可以弥合自然语言与编程语言的语义鸿沟;

(3) 提出了多模态对比学习预训练目标, 用于增强模型的代码表征能力, 从而提升模型的下游任务表现.

本文第 1 节介绍 REcomp 所涉及的相关工作, 包括预训练编程语言模型、多模态学习和对比学习. 第 2 节详细地阐述 REcomp 的模型架构. 第 3 节描述 REcomp 的有关实验. 在第 4 节总结全文工作.

1 相关工作

本文方法和预训练编程语言模型、多模态学习和对比学习的现有研究之间存在相关性, 下面对这些相关内容进行介绍.

1.1 预训练编程语言模型

预训练模型首先通过一个或者多个任务来预先学习相对泛化的知识, 然后在微调阶段, 利用学到的具体知识进行下游任务[3]. 本节将简要地介绍 CodeBERT[29], GraphCodeBERT[30], UniXcoder[32]. 这 3 个将分别作为 REcomp 预训练的初始化模型, 它们都是在开源基准数据集 CodeSearchNet[33]上进行预训练的.

CodeBERT 模型架构与 RoBERTa[25]相同, 利用多层双向 Transformer[23]进行自监督学习. CodeBERT 由 12 个相同的层组成, 每层的维度大小为 768. 模型参数总数达到 125M. CodeBERT 的掩码语言建模目标用的是带有函数注释的双峰数据, 然后替换令牌检测目标进一步训练在双峰和单峰样本上, 旨在学习一个代码字符是否是原始的. 在微调阶段, CodeBERT 在语义代码检索和源代码文档生成上的实验表明, 它优于监督学习的方法.

GraphCodeBERT 在 CodeBERT 基础上结合了代码的数据流, 在预训练阶段对变量之间“值从哪里来”的数值传递关系进行编码. 除了掩码语言建模, 它还引入了两个新的数据流结构感知的预训练任务:数据流图边预测和变量间结点对齐. GraphCodeBERT 利用 CodeSearchNet 的双峰数据进行预训练, 针对代码检索、克隆检测、代码翻译和代码细化等任务微调. 实验表明, 其微调结果超过了 CodeBERT.

UniXcoder 在 GraphCodeBERT 基础上将 AST 作为结构级特征, 是一种跨模态的编程语言预训练模型.它设计了一个前缀适配器, 这个适配器通过掩码注意力矩阵来控制模型的行为, 这统一了理解任务、生成任务和自回归任务. 它设计了一个 AST 和代码字符一对一的映射方法, 旨在将结构信息和语义信息融合. 此外, 它通过跨模态生成任务在不同编程语言之间的对齐表示来学习代码的通用表征.它在 9 个公开数据集上的 5 个代码相关的任务上进行评估, 结果表明, UniXcoder 在大多数任务上都达到了显著的效果.

CodeSearchNet 是含有 6 种编程语言的数据集, 分别包含双峰数据(注释-代码)约 213 万条和单峰数据(没有注释的代码)约 645 万条. CodeSearchNet 数据集在 6 种语言上的数据详情见表 1.

表 1 关于 CodeSearchNet 的数据统计

编程语言	Ruby	Java	JavaScript	PHP	Python	Go	All
单峰数据	164 048	1 569 889	1 857 835	977 821	1 156 085	726 768	6 452 446
双峰数据	52 905	500 754	143 252	662 907	458 219	319 256	2 137 293

1.2 多模态学习

多模态学习可以通过利用多模态数据之间的信息互补性, 从多源数据中学到更好的特征表示, 有利于下游任务的学习[36]. 因为编程语言含有多个抽象层次的特征, 所以用单个特征来表示编程语言是远远不够的, 很难覆盖所有视角. 并且不同的模态都是代码表示的平行语料, 具有语义等价性. DeepCS[37]是第一个用 RNN 融合功能级的 API 序列、函数名和语义级的代码字符等特征的代码表征模型. 然后, Comformer[38]开始关注结

构级特征 AST, 提出了一种简单的基于结构的 AST 遍历算法, 并将其序列化结果作为代码的特征, 这个特征被编码为 Transformer[23]的输入序列, 在代码摘要任务中表现良好. 后来, MMAN[14]为进一步充分挖掘结构信息, 将结构级的 AST, CFG 和语义级的代码字符结合. DeGraphCS[15]提出在编译后, 构造一个基于变量的流图, 以进一步提高代码语义表示的准确性, 并提出一种优化算法来去除变量流图中的冗余信息.

1.3 对比学习

传统的监督学习方法很大程度上依赖于大量的标注数据, 而标注数据往往需要领域的专家, 代价昂贵. 因此, 近年来, 无需外部监督信息的自监督学习方受到了研究者们的极大关注. 对比学习就是一种有效的自监督学习方法, 它通过最小化相似样本的语义向量的空间距离, 同时最大化不同样本之间的距离, 目的是帮助相近样本彼此更接近而不同样本彼此远离[39]. 它可以使模型学到样本的重要且具有区分度的内在隐藏特征, 进而将其应用于下游任务, 在计算机视觉和自然语言处理等领域得到了广泛研究, 如 SimCSE[40]是一个基于对比学习的方法来表示句子的通用嵌入的框架. 近年来, 有研究将对比学习应用到各种软件工程任务中,例如: Coder[41]是一种用于代码-代码检索、文本-代码检索和代码-文本摘要生成的对比学习方法; VarCLR[42]是基于不同的下游任务的对比学习模型, 旨在学习变量名称的语义表示. 最近, cpt-code[43]也证实了对比学习预训练可以得到代码更加丰富的语义表征. 在代码智能领域中, 设计源代码的数据增强方法成为应用对比学习的关键. ContraCode[44]通过源代码编译器在 JavaScript 上生成变体, 并进一步组合这些生成的变体作为数据增强的一种方法. MuCoS[45]应用一个名叫 JavaTransformer[46]的工具包, 里面包含 9 种对 Java 源代码进行数据增强的方法.

2 REcomp

本节将介绍 REcomp, 如图 2 所示.

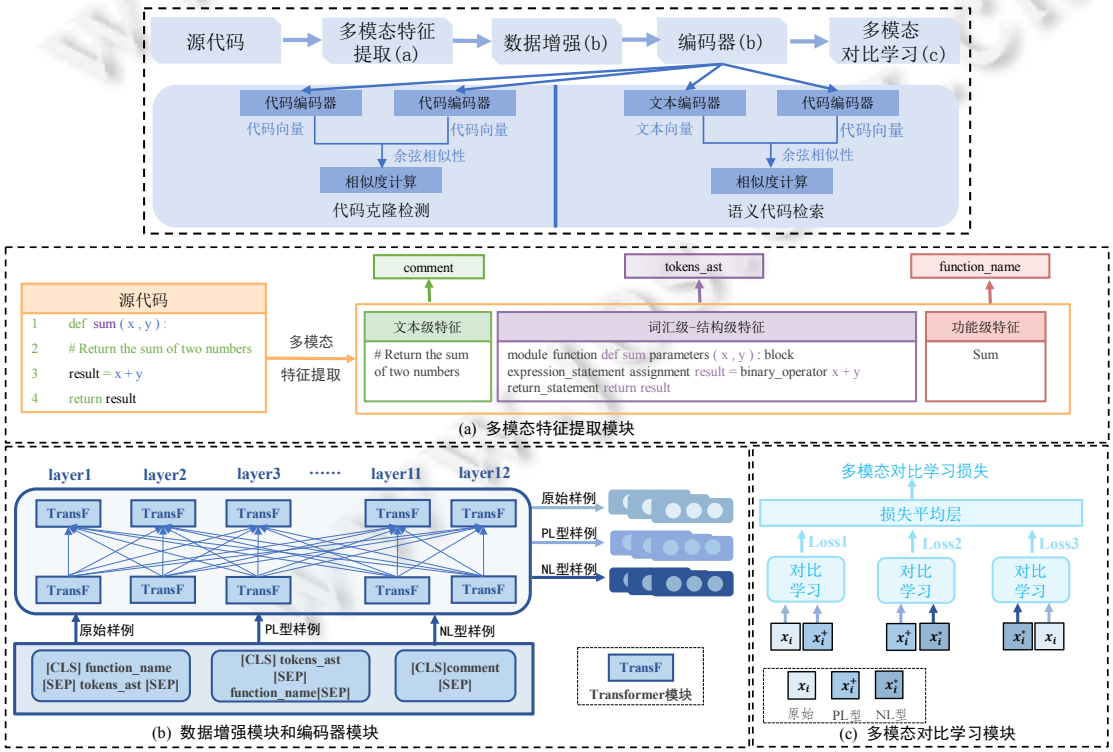


图 2 基于多模态对比学习的代码表征增强模型的框架图

REcomp 是一种基于多模态对比学习的代码表征增强的编程语言预训练模型, 是基于 Transformer[23]的, 能够以多种编程语言预训练模型(如 CodeBERT[29], GraphCodeBERT[30]和 UniXcoder[32])为初始化标准. 下面将依次介绍多模态特征提取、REcomp 的编码器、多模态对比学习的预训练任务(包含数据增强)和下游任务.

2.1 多模态特征提取

如图 2(a)多模态特征提取模块所示, 需要提取源代码 4 个层次的特征, 分别是文本级的代码注释、语义级-结构级的融合特征序列和功能级的函数名. 下面将详细阐述需要提取源代码多个模态特征的原因以及获取语义级-结构级融合的方法(见算法 1).

图 3 给出了一个带有注释和 AST 的 Python 代码示例. 注释“Return the sum of two numbers”高度概括了源代码的功能, 提供了关键的语义信息. 函数名称“sum”简明扼要的描述了源代码主体函数的功能, 也是表征源代码语义信息的另一重要而不可忽视的特征. 此外, 解析源代码后得到的 AST 包含丰富的句法相关的结构信息,例如: 非叶结点(控制结点)“return_statement”控制了其叶结点“return”和“result”整体行为; 非叶结点“parameters”指出来了“(x,y)”的类型是“参数”. 这里的“x”和“y”的结构是等价的. 这也是编程语言区别于自然语言的特殊的点: 编程语言中字符的语义和结构信息共同决定了它的表征, 存在同一字符表示不同语义和不同字符表示相同语义的现象. 因此, 不仅要关注语义级代码字符的信息, 也要关注隐藏的结构信息, 才能更加准确地表示源代码.

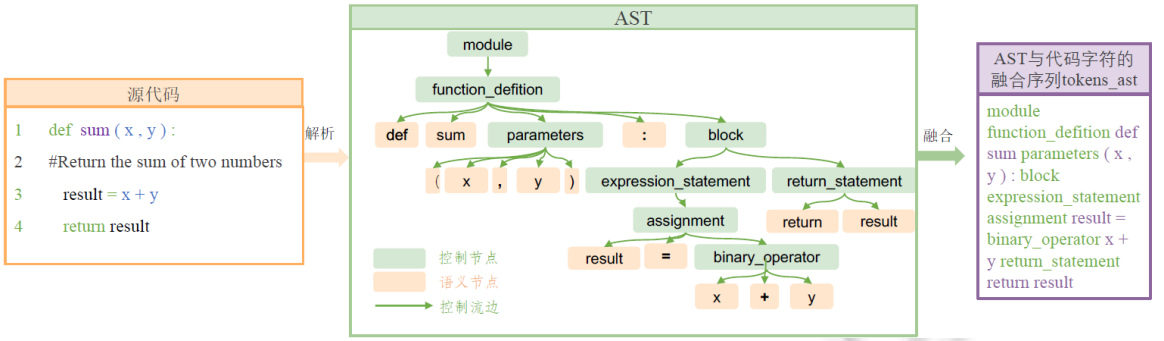


图 3 一个 Python 的 AST 与代码字符的融合序列示例

算法 1 描述了代码的语义级-结构级特征的融合的过程. 算法的输入是编程语言的类型 *Language*∈{php, java,python,ruby,go,javascript}, 以及程序 *Code* $S=(s_1,s_2,\ldots,s_i,\ldots,s_{|S|})$, 输出是所有语义级和结构级特征的集合 *TokensAST*. 将 *Code* ***S*** 相应的 *AST* 表示为六元组 $\tau=(N,N_{leaf},N_{nonleaf},c(\cdot),t(\cdot),r)$, N 是 *AST* 所有结点的集合, $N_{leaf}=(l_1,l_2,\ldots,l_{|leaf|})\subset N$ 是叶子结点的集合, $N_{nonleaf}=(nol_1,nol_2,\ldots,nol_{|nonleaf|})\subset N$ 是非叶子结点的集合, $r\in N$ 是 *AST* 的根结点, $c(\cdot)$是 *AST* 的 N_{leaf}与代码 $tokenss_i\in S$ 对齐的映射函数(第 18–27 行), $t(\cdot)$是 *AST* 的 $N_{nonleaf}$与其结点的 *type* 属性映射的函数(第 15 行).

算法 1 的核心是 *_SemanticsStructureFusion* 函数(第 7–17 行), 它将 $t(N_{nonleaf})$与 $c(N_{leaf})$依次添加到集合 *TokensAST*, 即 *ca*(见公式(1))中. 整个算法自顶向下的遍历 *AST*, 融合了代码的语义信息和结构信息.

算法 1. 语义级-结构级特征融合(*SemanticsStructureFusion*).

输入: 编程语言 *Language*, 源代码 *Code*;

输出: 所有语义级和结构级特征的集合 *TokensAST*.

1. Function *SemanticsStructureFusion*(*Langauge*,*Code*)
2. *TokensAST*←∅
3. get the *AST parser* of *Code* depending on different *Language*
4. parse the *Code* to *AST* then get the *root* of the *AST*

5. _*SemanticsStructureFusion*(*Code*,the *root* of the *AST* parsed by *Code*)
6. Return *TokensAST*
7. Function _*SemanticsStructureFusion*(*Code*,*root*)
8. *TokensAST*←∅
9. **If** *root.Children*=∅ **then** Return
10. **For** *child* **in** *root.children*
11. **If** *child.type*="identifier" and *child.children*≠∅ **then**
12. *tokens*←*AlignNodeToTokens*(*child* is one leaf node N_{leaf} of the *AST*,*Code*)
13. add the *tokens* to the *TokensAST*
14. **Else**
15. *ast*←*child.type* and add the *ast* to the *TokensAST*
16. _*SemanticsStructureFusion*(*Code*,*child*)
17. Return *TokensAST*
18. Function *AlignNodeToTokens*(N_{leaf},*Code*)
19. *start*←N_{leaf}.*start_node*,*end*←N_{leaf}.*end_node*
20. *tokens*←∅ and *tokens*⊂*Code*
21. **If** *start*[0]=*end*[0] **then** *tokens*←*Code*[*start*[0]][*start*[1]:*end*[1]]
22. **Else**
23. *tokens*←*Code*[*start*[0]][*start*[1]:]
24. **For** *i* **in** [*start*[0]+1,…,*end*[0]]
25. concatenate *Code*[*i*] to *tokens*
26. concatenate *Code*[*end*[0]][:*end*[1]] to *tokens*
27. Return *tokens*

注意: *AlignNodeToTokens* 函数(第 18–27 行)中, 根据 N_{leaf}的 *start_node* 和 *end_node* 属性值,在源代码中有唯一确定的 *tokens*⊂*S* 与 N_{leaf}对齐. 并且 *start_node* 是一个元组, 表示为(行号,序号). 例如图 3 所示: 源代码中的字符"sum"位于第 1 行, 包含第 5 个字符到第 7 个字符, 因此它的 *start_node* 值为(0,4); 同理,它的 *end_point* 的值为(0,6).

Code ***S*** 的函数名和自然语言注释形式化为 *f* 和 *n*, 其中, *J* 和 *L* 是 *f* 和 *n* 应用 RoBERTa[25]分词器划分的长度:

$$ca=(t_1,t_2,...,c_1,...,t_{|N_{nonleaf}|},...,c_{|N_{leaf}|}),\ t_j=t(N^j_{nonleaf}),\ c_i=(N^i_{leaf}) \tag{1}$$

$$f=(f_1,f_2,\ldots,f_j,\ldots,f_J),\ 1\leqslant j\leqslant J,\ n=(n_1,n_2,\ldots,n_l\ldots,n_L),\ 1\leqslant l\leqslant L \tag{2}$$

将公式(1)的 *ca* 和公式(2)的 *f* 进行拼接, 且使用特殊符号[*CLS*]表示是一个位于段序列最前端的特殊开始字符, [*SEP*]是分隔不同模态数据的特殊字符, 源代码的输入 *code* 表示如下:

$$code=([CLS]\oplus f\oplus[SEP]\oplus ca\oplus[SEP]) \tag{3}$$

2.2 编码器

REcomp 分别以 CodeBERT, GraphCodeBERT 和 UniXcoder 为初始化模型[3], 其编码向量函数表示为 *Embedding*(·), 结合多模态输入 ***code*****,** 如公式(3)所示, 其中间表示 e_{code} 表示在公式(4):

$$e_{code}=Embedding(code) \tag{4}$$

如图 2(b)编码器模块所示: REcomp 由 *N*(*N*=12)层 Transformer[23]组成, 由中间形式 *e* 作为模型输入生成每一层的隐藏状态 H^N, 每一层 Transformer 都含有一个架构相同的转换器, 该转换器使用多头注意力机制(multi-headedself-attention), 通过一个前馈神经网络(FeedForwardLayer)覆盖上一层的输出, 初始化时 $H^0=e_{code}$, 对于

第 I 层的 Transformer, 多头注意力机制的输出通过如下方式计算:

$$H^N=\{h_0^N,h_1^N,...,h_{n-1}^N\},\ E^n=LN(MultiAtt(H^{n-1})+H^{n-1}),\ H^n=LN(FFN(E^n)+E^n) \tag{5}$$

其中, $MultiAtt$ 是多头注意力机制, FFN 是前馈神经网络, LN 是正则化操作, 最终的输出 E^N 由下列公式得到:

$$\begin{aligned} &Q_i=H^{l-1}W_i^Q,\ K_i=H^{l-1}W_i^K,\ V_i=H^{l-1}W_i^V,\ head_i=softmax\left(\frac{Q_iK_i^T}{\sqrt{d_k}}\right)V_i \\ &E^N=[head_1;...;head_N]W_N^O \end{aligned} \tag{6}$$

其中, 前一层的输出 $H^{(l-1)}\in R^{(n*dh)}$被线性映射为〈查询,键,值〉, 即〈Q,K,V〉, dk 是 $head$ 的大小. $W_i^Q,W_i^K,W_i^V\in R^{(dh*dk)}$, 其初始化 W_1^Q, W_1^K, W_1^V, 是随机初始化的参数. $W_N^O\in R^{(dh*dk)}$ 是模型的输出参数.

2.3 多模态对比学习

对比学习鼓励原始样本的表示更加接近“正”增强样本的表示, 同时远离“负”样本的表示, 使得模型在不同数据之间能学习到更加均匀的决策边界点. 最近的几项工作[42,45]试图比较相似和不同的源代码, 然而它们只比较了单个模态的特征, 却忽略了编程语言的多模态特性. 为此, 本文设计了多模态对比学习的预训练目标来探索不同模态之间交互信息的最大潜力, 鼓励模型学习编程语言不同模态的联系和语义等价性. 对比学习的关键是构造正负样本, 因此, 本文提出了两种新的数据增强的方法, 分别称作 NL 型和 PL 型, 如图 2(b)数据增强模块所示.

对于正样本的设计, 本文针对 CodeSearchNet[33]中双模态(PL-NL)数据进行构造.

(1) NL 型: 为了弥合编程语言与自然语言的语义鸿沟, 本文将文本级的自然语言注释单独视为一个正样本, 将它表示为 $code^*$, 如公式(7)所示:

$$code^*=([CLS]\oplus comment\oplus[SEP]) \tag{7}$$

(2) PL 型: 因为功能级的函数名、语义级的代码字符和结构级的 AST 都具有编程语言的特点, 所以为了学习这 3 个特征的语义等价性, REcomp 交换功能级的函数名 f 和语义级-结构级特征的融合序列 ca 在输入序列中的位置, 将此作为一种数据增强的方法, 得到的另一个正样本, 将它表示为 $code^+$, 如公式(8)所示:

$$code^+=([CLS]\oplus ca\oplus[SEP]\oplus f\oplus[SEP]) \tag{8}$$

为了让模型最大程度地学习编程语言 4 个模态特征的语义等价性, 本文将原始样本、PL 型正样本和 NL 型正样本两两组合, 得到($code$,$code^*$), ($code^*$,$code^+$)和($code^+$,$code$)这 3 个组合.

对于负样本的设计, 采用的损失函数是交叉熵损失函数. 对于原始样本 x_i, 将原始样本 batch 里其余 $x_j(j\neq i)$ 作为负样本, 简称为 In-batch, 并且将正样本 batch 里不同对 $x_j^+(j\neq i)$ 也作为负样本, 简称为 cross-batch. 因此, 以($code$,$code^*$)为例, 如果一个 batch 里有 N 组($code$,$code^*$), 那么对于 1 个 $code$ 就有 2N–2 个负样本, 分别是: N–1 个 $code^-$和 N–1 个 $code^{-*}$. 因此, 对比学习的损失函数 $Loss_{MCL}$ 表示在公式(10), 它的具体工作流程如图 2(c)多模态对比学习模块所示:

$$l(x_i,x_i^+)=-\ln\frac{\exp(v_i,v_i^+)}{\exp(v_i,v_i^+)+\sum_1^{2N-2}\exp(v_i,v_i^-)} \tag{9}$$

其中, v_i 是 x_i 对应的语义向量, 由公式(7)得到; v_i^+ 是 x_i 正样本的语义向量; v_i^- 是 x_i 负样本的语义向量:

$$Loss_{MCL}=\sum_1^N[l(x_i,x_i^+)+l(x_i,x_i^*)+l(x_i^*,x_i^+)] \tag{10}$$

其中, x_i 是原始样本, x_i^+ 是 NL 型的正样本, x_i^* 是 PL 型的正样本.

2.4 下游任务

本文将经过多模态对比学习预训练后的编码器形式化为 Encoder. 为探究模型是否增强了代码表征的能力, 本文选取了两个典型的理解型下游任务(代码克隆检测和语义代码检索)对模型的有效性进行评估.

2.4.1 代码克隆检测

代码克隆检测是一个通过测量两个源代码之间的相似性来识别代码克隆问题是否存在的任务, 这个问题形式化定义为: 给定一个源代码 c_1 和一个待检测的代码 c_2, 根据 c_1 和 c_2 的相似性分数, 返回 0/1 标签, 标签 0 表示 c_2 不是 c_1 的克隆体, 标签 1 则表示 c_2 是 c_1 的克隆体. 本文选择余弦相似性(记作 *cosSim*)作为相似性分数的计算函数, 来对它们进行相似度分数计算, 计算表示如公式(11):

$$e_{c_1}=Encoder(c_1),\ e_{c_2}=Encoder(c_2),\ score=cosSim(e_{c_1},e_{c_2}) \tag{11}$$

2.4.2 语义代码检索

语义代码检索是一个根据用户的查询意图, 从代码语料库中匹配最具相关性的源代码的任务. 这个问题形式化定义为: 给定一个自然语言形式的查询语句 n, 与代码语料库 $U_C=\{c_1,c_2,c_3,\ldots,c_i,\ldots,c_I\}$ 中的代码 $c_i\in U_C$ 进行相似度分数计算(I 表示 U_C 中代码的总数量), 并且从 U_C 中返回具有最高分数的源代码 $\hat{c}_\tau$, 计算表示如公式(12):

$$e_{c_i}=Encoder(c_i),\ e_n=Encoder(n),\ \hat{c}_\tau=\arg\max_{c_i} cosSim(e_{c_i},e_n) \tag{12}$$

3 实 验

3.1 数据集

REcomp 使用的是一个代码智能领域的公开基准数据集 CodeXGLUE[2]. 本文选取代码克隆检测任务的两个数据集——BigCloneBench[34]和 POJ-104[35]以及代码检索任务的 CodeSearchNet[33]数据集.

BigCloneBench 是一种广泛被使用的大型代码克隆基准, 包含约 6 000 000 真的克隆对和 260 000 个假的克隆对, 这些克隆对来自 10 个不同功能的克隆对语料库. 它们[34]通过对没有任何标记的真假克隆对的源代码进行过滤, 最后将数据集划分成 901 208 / 415 415 / 415 416 个示例, 分别用于训练、验证和测试;

POJ-104 数据集来自一个教学的开放编程平台, 它包含 104 个问题, 每个问题包含 500 个学生编写的 C/C++程序. 它被划分成 32 000 / 8 000 / 12 000 条数据, 分别用于训练、验证和测试;

CodeSearchNet 中每个示例都包含一个与文档配对的函数, 它们[33]选取文档的第 1 段作为代码功能描述的注释, 通过删除满足一定条件的示例来对其进行数据过滤, 进而提高数据集的质量. 上述数据过滤的条件分别是: 去除源代码存在编译错误且无法解析成为抽象语法树的示例; 去除源代码的文档(即自然语言注释)字符数量小于 3 或者大于 256 的示例; 去除源代码的文档中含有特殊表示和一些与功能无关的内容, 如指向外部资源的链接或者 HTML 标签等示例; 去除源代码的文档里含有非英文字符的示例. 过滤后的 CodeSearchNet 数据集统计信息见表 2.

表 2 代码检索任务中 CodeSearchNet 的数据统计

编程语言	Ruby	Java	JavaScript	PHP	Python	Go
训练集	24 927	164 923	58 025	241 241	251 251	167 288
验证集	1 400	5 183	3 885	12 982	13 914	317 325
测试集	1 261	10 955	3 291	14 014	14 918	8 122

3.2 预训练设置

为了公平比较, 本文同样在 CodeSearchNet[33]数据集上预训练 REcomp. REcomp(C/G/U)表示分别以 CodeBERT, GraphCodeBERT, UniXcoder 初始化的 REcomp, 他们的训练集大小分别为 48/60/64, 学习率为 1e−5, 优化器采用 AdmW, 原始样本和 PL 型正样本输入大小为 300, NL 型正样本的输入大小为 64. 然后, 用 1 个 NVIDIA Tesla A100(大小为 40 GB)分别训练 28k, 22k 和 21k 训练步.

3.3 评价指标

3.3.1 代码克隆检测

代码克隆检测的评价指标有平均精度(mean average precision, MAP@R)、召回率(recall)、准确率(precision)、$F1$-值($F1$-score). MAP@R 用于评价给定查询在集合中检索出的 R 个最相似样本的结果, 其中, R 设置为 499, 其计算方式如公式(13):

$$\text{MAP@R}=\frac{\sum_{q=1}^{Q} AvgPrecision(q)}{Q} \tag{13}$$

其中, Q 是数据集大小. 余下评估指标的计算方法如式(14):

$$Precision=\frac{tp}{tp+fp},\ Recall=\frac{tp}{tp+fn},\ F1=\frac{2*(Precision*Recall)}{Precision+Recall} \tag{14}$$

3.3.2 语义代码检索

语义代码检索任务的评价指标[47]有: 最佳命中率(FRank)和平均排序倒数(mean reciprocal rank, MRR). FRank 是指第 1 个命中结果在结果列表中的排名, 是计算 MRR 的基础. MRR 是利用查询接收到的结果集中在第 1 个正确答案的位置来评估模型的性能, 如果正确答案越靠前, 则表示模型效果越好:

$$MRR=\frac{1}{|Q|}\sum_{q=1}^{Q}\sigma(FRank_q\leqslant k) \tag{15}$$

其中, Q 是查询语句的集合. $\delta(\cdot)$是特征函数, 满足$(\cdot)$时, $\delta(\cdot)=1$; 否则, $\delta(\cdot)=0$.

3.4 实验结果分析

本节分为以下 6 个小节: 第 3.4.1 节分析 REcomp 的整体性能; 第 3.4.2 节验证 REcomp 中单个组件的有效性; 第 3.4.3 节对比 REcomp 与基准模型[29−32]的预训练资源使用情况; 第 3.4.4 节验证优化的 AST 序列化算法的有效性; 第 3.4.5 节阐述批量大小与模型性能的关系; 第 3.4.6 节进行案例分析, 通过对部分案例的语义向量可视化,来验证 REcomp 的有效性.

3.4.1 主要实验结果

主要实验旨在测试基于多模态对比学习的代码表征增强模型的整体效果, 验证多模态对比学习预训练目标的有效性. 本文将在两个代码理解型的下游任务(代码克隆检测和语义代码检索)上比较 REcomp 和 10 个基准方法的实验结果. 这 10 个基准方法分为两类: 第一类是从头到尾都是在评估任务上进行训练; 第二类是模型先在未标记的语料库进行预训练, 然后在评估任务上进行评估. 这 10 个基准方法分别是 TextCNN[12], BiLSTM[48], Transformer[23], RoBERTa[25], RoBERTa(code), PLBART[28], CodeBERT[29], GraphCodeBERT[30], SynCoBERT[31], UniXcoder[32]. 为了公平起见, 在下游任务评估时, 与 UniXcoder 保持一致, 本文只用语义级特征代码字符来表征源代码. 除学习率外, REcomp 在下游任务上微调的参数分别与其初始化模型保持一致. REcomp 在 CodeSearchNet[33], BigCloneBench[34]和 POJ-104[35]数据集上的学习率分别为 8e−06, 2e−5, 1e−5.

就代码克隆检测而言, 其结果见表 3: 在 POJ-104 数据集上, REcomp(C/G/U)分别比初始化模型 CodeBERT, GraphCodeBERT 和 UniXcoder 提升了约 7.2%, 3.5%和 1.4%的 MAP@R. REcomp(C)不仅超过了 GraphCodeBERT 约 4.7%, 还超过了 REcomp(G), 表明 CodeBERT 在代码理解任务上的提升空间很大. 在 BigCloneBench 数据集上, REcomp(C)相比于 CodeBERT 提升了约 0.6%的准确率; 在召回率上, REcomp(U)超过了 UniXCoder 近 2.1%.

就语义代码检索任务而言,其实验结果见表 4, REcomp(C/G/U)分别比初始化模型 CodeBERT, GraphCodeBERT 和 UniXcoder 提升了平均 MRR 约 2.2%, 2.7%和 0.7%. 在 Ruby 上, 模型表现显著, 最高提升了 5%的准确率. 在 JavaScript 和 Go 上, REcomp(C)的性能超过了 GrapCodeBERT. REcomp 在不同语言的数据集上性能提升各有不同, 是因为不同编程语言的自身特点区别鲜明. 例如, REcomp(U)在 Ruby 数据集上的 MRR 增加了 2.4%, 而 Java 数据集的 MRR 只增加了 0.2%. 造成这种现象的可能原因是, 它们的语法和表达性

不同: Ruby 语言的语法简洁, 它注重代码的可读性和易于理解, 尽量使用更少的语法结构和标点符号, 使得代码看起来更接近自然语言, 因此, REcomp 极大地拉近了它与自然语言的语义鸿沟; 相比之下, Java 语言的语法更为严格, 需要使用更多的关键字和符号来定义代码结构, 因此在拉近代码和查询之间的语义距离还需要更多的结构信息. 根据 Go 和 Python 数据集进一步验证上述猜想, 它们的 MRR 分别增加了 0.4%和 0.6%, 并且 Go 和 Python 都注重代码的简洁性和可读性, 相比 Java 其结构更加简单. 它们都采用了清晰简洁的语法和规范, 使得代码更易于编写、理解和维护, 因此它们 MRR 的提升都高于语法结构更加复杂的 Java.

表 3 代码克隆检测任务的评价结果 (%)

模型	POJ-104	BigCloneBench		
	MAP@R	Recall	Precision	*F*1-score
RoBERTa	76.67	95.1	87.8	91.3
SynCoBERT	88.24	–	–	–
PLBART	86.7	94.8	92.5	93.6
CodeBERT	82.67	94.7	93.4	94.1
REcomp(C)	89.89	94.0	**94.0**	94.0
GraphCodeBERT	85.16	94.8	95.2	95.0
REcomp(G)	88.63	94.0	94.0	94.0
UniXcoder	90.52	92.9	97.6	95.2
REcomp(U)	91.91	**95.0**	95.0	95.0

表 4 6 种编程语言的代码检索任务的评价结果 (%)

模型	Ruby	JavaScript	Go	Python	Java	PHP	平均
CNN	27.6	22.4	68.0	24.2	26.3	26.0	32.4
BiLSTM	21.3	19.3	68.8	29.0	30.4	33.8	33.8
Transformer	27.5	28.7	72.3	39.8	40.4	42.6	41.9
RoBERTa	58.7	51.7	85.0	58.7	59.9	56.0	61.7
RoBERTa(code)	62.8	56.2	85.9	61.0	62.0	57.9	64.3
SynCoBERT	72.2	67.7	91.3	72.4	72.3	67.8	74.0
PLBART	67.5	61.6	88.7	66.3	66.3	61.1	68.5
CodeBERT	67.9	62.0	88.2	67.2	67.6	62.8	69.3
REcomp(C)	71.5	64.6	90.4	68.5	69.8	64.1	**71.5**
GraphCodeBERT	70.3	64.4	89.7	69.2	69.1	64.9	71.3
REcomp(G)	75.3	67.5	91.5	71.4	71.8	66.5	**74.0**
UniXcoder	74.0	68.4	91.5	72.0	72.6	67.6	74.4
REcomp(U)	76.4	68.9	91.9	72.6	72.8	67.8	**75.1**

综上所述, REcomp 提高了代码表征的准确性, 从而增强了模型在代码理解型任务上的性能. 这有助于减少代码调试和优化的需求, 提高了代码的运行效率, 促进了代码的重用和模块化开发. 这样有益于节约能源、减少计算机软硬件资源的使用, 并减少对相关电力和碳排放的依赖.

3.4.2 消融实验

本小节将验证 3 个问题.

(1) PL 型正样本能否让模型学习到具有编程语言特点的特征的语义等价性?

(2) NL 型正样本能否弥合自然语言和编程语言的语义鸿沟?

(3) PL 与 NL 的组合型是否增强了模型代码表征的能力?

本文用-w/o 定义了“删除”操作符, 用来检测单个组件的有效性, 则有:

- -w/o PL: 表示删除 PL 型正样本, 进行多模态对比学习预训练任务时只使用 NL 型正样本;
- -w/o NL: 表示删除 NL 型正样本, 进行多模态对比学习预训练任务时只使用 PL 型正样本;
- -w/o MCL: 表示没有经过多模态对比学习预训练任务, 直接用初始化模型对下游任务进行评估.

(1) NL 型的有效性

就克隆检测任务而言, 从表 5 观察到: REcomp(C/G/U)去掉 NL 型, 比 NL+PL 型下降了约 0.14%, 0.4%和 0.9%, 高出基准方法约 7.1%, 3.1%和 0.54%. 在语义代码检索任务上, 仅用 NL 型作为正样本预训练, 即 REcomp(C/G/U, -w/o PL), 在 MRR 上高出其基准分别约 2.1%, 2.6%和 0.9. REcomp(C/G, -w/o NL)相比 REcomp(C/G)下降了 1.8%, 1.4%和 0.7%, 可以得出, REcomp(C/G/U)是 NL 型敏感的, 即它们的性能会受到具

有自然语言特点的代码特征的影响. 此外还可以发现, REcomp(U)是强 NL 型敏感的, 因为 REcomp(U,NL)在 *MRR* 上比 REcomp(U)高了 0.2%.

(2) PL 型的有效性

在代码克隆检测任务上, 以 POJ-104 数据集为例, 其结果见表 5, 仅使用 PL 型进行预训练, 即 REcomp (C/G,-w/o NL), 在 MAP@R 上分别高出基准模型 CodeBERT, GraphCodeBERT 和 UniXcoder 约 7.1%, 3.1%和 0.54%. 并且去掉 PL 型后, REcomp(C/G)的准确率有所下降. 因此可以得出, CodeBERT 和 GraphCodeBERT 是 PL 型敏感的, 即它们会受到具有编程语言特点的代码特征的影响.

表 5 语义代码检索任务上的消融实验的结果 (%)

模型	Ruby	JavaScript	Go	Python	Java	PHP	平均
REcomp(c)	71.5	64.6	90.4	68.5	69.8	64.1	71.5
-w/o PL	71.7	64.8	90.3	68.4	69.6	63.8	71.4
-w/o NL	67.7	61.9	89.9	67.5	68.1	63.1	69.7
-w/o MCL	67.9	62.0	88.2	67.2	67.6	62.8	69.3
REcomp(G)	75.3	67.5	91.5	71.4	71.8	66.5	74.0
-w/o PL	75.0	67.5	91.2	71.4	71.7	66.5	73.9
-w/o NL	71.7	65.8	90.8	70.7	70.4	66.0	72.6
-w/o MCL	70.3	64.4	89.7	69.2	69.1	64.9	71.3
REcomp(U)	76.4	68.9	91.9	72.6	72.8	67.8	75.1
-w/o PL	76.6	69.9	92.1	72.4	73.2	67.6	75.3
-w/o NL	74.2	68.4	91.5	72.4	72.3	67.6	74.4
-w/o MCL	74.0	68.4	91.5	72.0	72.6	67.6	74.4

在语义代码检索任务上, REcomp 去掉 NL 型正样本的结果如表 6 所示, 即 REcomp 只使用 PL 型正样本进行预训练 REcomp(C/G/, -w/o NL), 其性能仍然高出基准的 CodeBERT 和 GraphCodeBERT 约 0.4%和 1.3%. 从 NL 型与 PL+NL 组合型的结果比较的角度来验证, 去掉 PL 型组件后, REcomp(C/G)总体性能下降了约 0.1%.

表 6 代码克隆检测(POJ-104)消融实验的 MAP@R(%)结果

REcomp(G)	-w/o PL	-w/o NL	-w/o MCL
88.63	88.54	88.24	85.16
REcomp(U)	-w/o PL	-w/o NL	-w/o MCL
91.91	92.92	91.06	90.52
REcomp(C)	-w/o PL	-w/o NL	-w/o MCL
89.89	89.80	89.75	82.67

(3) PL 与 NL 组合型的有效性

PL 与 NL 组合型的有效性就是本文第 3.4.1 节主要实验的内容, 在第 3.4.1 节中, 已经证实了多模态对比学习模块(PL 与 NL 组合型)的有效性.

总结: 模型对 PL 型的敏感程度不及 NL 型, NL 型起到了主要作用, 特别是在 REcomp(C/U)中, 去掉 PL 型后, 模型在 Ruby, JavaScript, Go 和 Java 上进一步提升了, 但在 Python 和 PHP 上, 仍是 PL+NL 强于 NL. 总体来说, 增加 PL 型的 REcomp 的性能都是优于其初始化模型的. 因为初始化模型、编程语言和数据集的不同, REcomp 对 NL 型和 PL 型正样本的敏感程度不一样, 所以存在少数 PL 型或者 NL 型优于 PL 与 NL 组合型的现象.

3.4.3 预训练资源对比实验

本文罗列了基准方法中预训练模型在预训练阶段的计算资源的使用详情, 见表 7. 从表 7 中可以看出, REcomp 仅使用极少的计算资源和小批量训练集就在较短的时间迅速收敛. 因此可以看出, 多模态对比学习是一种环保、低碳的学习方法, 它通过共享学习和特征融合等方式来减少模型推理和训练的过程, 进而降低能源消耗和环境压力, 为可持续发展提供了一种新的解决方案.

表 7 各模型预训练使用的资源统计

模型	输入长度	显卡资源	批量大小	时间(h)	训练步数(k)
CodeBERT	512	16 块 NVIDIA Tesla V100	2 048	6	100
GraphCodeBERT	640	32 块 NVIDIA Tesla V100	1 024	83	200
SynCoBERT	512	8 块 NVIDIA Tesla V100	128	80	110
UniXcoder	1 024	64 块 NVIDIA Tesla V100	1 024	192	800
REcomp(C/G/U)	300	3 块 NVIDIA Tesla V100	64/60/48	7/5.5/5.25	28/22/21

3.4.4 融合算法有效性分析

语义级-结构级融合算法旨在验证 REcomp 优化的序列化 AST 方法能否有效缩短模型特征序列的长度. 因为在基准方法的预训练模型里, 只有 SynCoBERT 和 UniXcoder 用 AST 作为代码表征的一部分, 所以本文将 SynCoBERT, UniXcoder 和 REcomp 对 CodeSearchNet[33]中 6 种编程语言的训练集的 AST 序列化后的平均长度进行统计, 其结果如图 4 所示. 它们分别是 244, 530 和 139. 由此得出: REcomp 相比于 UniXcoder 和 SynCoBERT, 其 AST 的序列化结果缩短约 381%和 176%. 因此, REcomp 中优化的序列化 AST 方法, 极大程度地减少了代码的冗余特征, 并且缩短了模型的特征序列的长度, 进而减少了数据存储和传输中的能量消耗, 最终节省了计算资源.

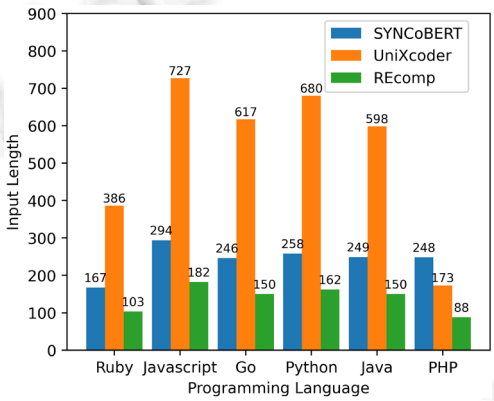


图 4 模型在 AST 序列长度上的对比

3.4.5 参数分析

REcomp 作为预训练模型, 其性能受到参数的影响. 因此, 本小节就实验中重要的几个参数之一的训练批量大小进行讨论. 本文选择语义代码检索任务里的随机一种编程语言, 本次实验的随机结果是 Ruby 语言, 以此进行实验, 其余参数与微调时相同, 训练轮数设置为 2, 实验结果如图 5 所示. 从图 5 中可以看出, batch size 越大, 检索任务的准确性越高. 因此, batch size 对检索结果起正向作用. 但是 batch size 过大, 会导致训练时占用显卡资源过多; 而且, 当 batch size 达到某个阈值时, 模型在性能上的增长趋于平缓, 但资源上的消耗却更大了. 因此, 考虑到这一点, 本文将 batch size 设置为 64.

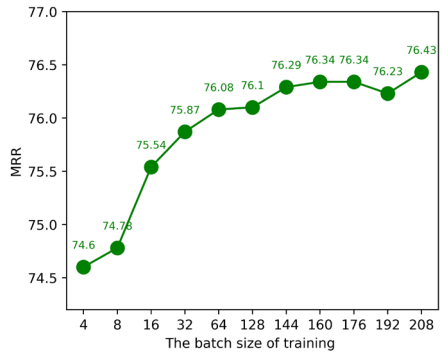
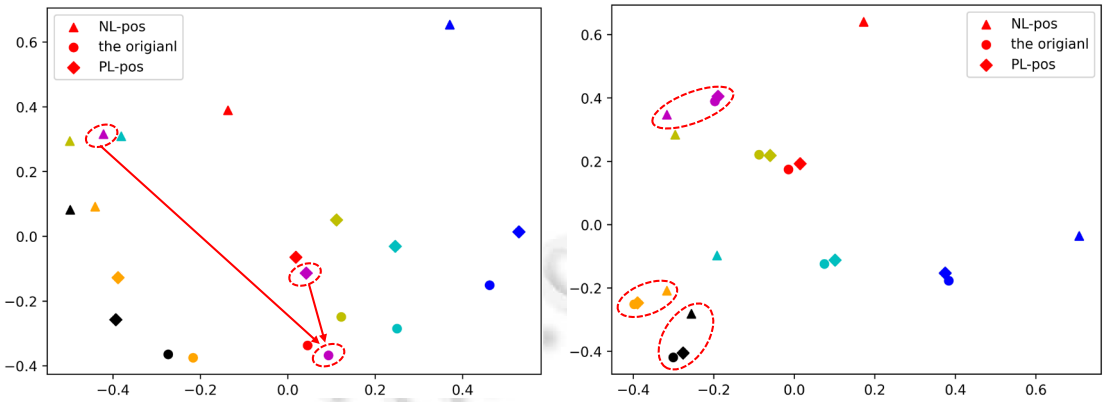


图 5 批量大小与准确性的相关性

3.4.6 案例分析

DeepCS[37]提出: 在向量空间中, 两个向量语义越接近, 其距离越小. 基于这个前提, 本文从 CodeSearchNet[33]中随机选取 7 个功能不同的源代码, 作为 7 个类别. 如图 6 示: 用 7 种不同颜色表示 7 个类别, 相同颜色的不同符号具有语义等价性, 圆形、菱形和三角形分别表示原始样本、PL 型正样本和 NL 型正

样本. 在图 6(a)中, 向量空间的所有点是通过初始化模型 UniXcoder[32]编码得到的, 相同颜色不同符号的点彼此距离较远, 如红色虚线区域的紫色向量. 图 6(b)中, REcomp(U)表征的语义向量, 呈现出相同颜色不同符号的点相互聚拢的趋势, 如红色虚线覆盖区域所示. 因为 PL 型跟原始样本都是编程语言, 而 NL 型是自然语言, 所以圆形点与菱形点的距离比其与三角形点的距离更为接近.



(a) UniXcoder 表征的语义向量　　(b) REcomp(U)表征的语义向量

图 6　不同表征方法下的语义向量类别的可视化

根据上述现象得出结论: 经过多模态对比学习预训练任务后, 模型有效地学习到了编程语言 4 个不同抽象层次的特征的语义等价性. 相比 NL 型(文本级的自然语言注释), 具有编程语言特点的 PL 型与原始样本的空间距离更为接近, 侧面验证了编程语言与自然语言之间存在语义鸿沟.

3.5 实验环境

研究中涉及的所有实验均基于 PyTorch 框架实现, 通过使用 tree-sitter (https://github.com/tree-sitter/tree-sitter)、apex 库(https://github.com/NVIDIA/apex)以及 Transformer (https://github.com/huggingface/transformers)来实现 REcomp. 表 8 给出了微调时, 代码克隆检测和语义代码搜索的超参数和具体的取值, 这些取值基于已有文献的推荐取值和我们实证研究中的实际性能. 在代码克隆检测任务中输入 1 是源代码, 输入 2 是待检测代码; 在语义代码检索任务中, 输入 1 是代码, 输入 2 是自然语言. 优化器采用 AdmW, 并且采用了 FP16 混合精度来加速训练. 此外, 我们根据基准方法的描述重现了这些方法, 并且这些方法的执行结果与原始文献中的结果接近. 实验运行的计算机的配置信息是: Inter i5-8265U CPU、32 GB 内存的 Tesla V100-SXM2、Linux 操作系统.

表 8　REcomp 涉及的超参数和对应取值

任务(数据集)	模型	训练集大小	测试集大小	输入长度 1	输入长度 2	学习率	训练轮次
代码克隆检测 (POJ-104)	REcomp (C/G/U)	8	16	400	400	1e−5	1
代码克隆检测 (BigCloneBench)	REcomp(C/G)	16	32	512	512	2e−5	1
	REcomp(U)	16	32	512	512	5e−5	1
语义代码检索 (CodeSearchNet)	REcomp(C)	64	64	256	128	2e−5	8
	REcomp(G)	32	64	256	128	2e−5	10
	REcomp(U)	64	64	300	64	8e−6	10

4 总　结

大规模编程语言模型的发展, 给代码智能领域带来了机遇和挑战. 本文提出了基于多模态对比学习的预训练方法(REcomp), 旨在增强模型代码表征的能力, 并且在公开基准数据集 CodeXGLUE[2]上验证了其在下游任务——代码克隆检测和语义代码检索上的有效性. REcomp 利用对比学习对融合了源代码所有抽象层次

特征(包括文本级代码注释、语义级代码字符、功能级函数名和结构级的 AST)的语义向量建模, 充分挖掘了各个模态特征的隐藏信息, 提高了代码表征的准确性. 除此之外, REcomp 优化了序列化 AST 的方法, 在保留完整的非叶结点的控制信息和叶子结点的语义信息的同时, 避免了结点冗余和序列过长等问题, 极大程度缩短了输入的特征序列. 更短的特征序列促使模型仅使用极少的计算资源和小批量训练集, 就在较短的时间迅速收敛, 并在公开数据集上达到了不错的效果.

References:

[1] Rey SJ. Big code. Geographical Analysis, 2023, 55(2): 211−224.

[2] Lu S, Guo D, Ren S, *et al*. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102. 04664, 2021.

[3] Cheng SQ, Liu JX, Peng ZL, *et al*. CodeBERT based code classification method. Computer Engineering and Applications, 2023, 59(24): 277−288 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.2209-0402]

[4] Jiang Y, Li M, Zhou ZH. Software defect detection with Rocus. Journal of Computer Science and Technology, 2011, 26(2): 328−342. [doi: 10.1007/s11390-011-1135-6]

[5] Jiang L, Misherghi G, Su Z, *et al*. Deckard: Scalable and accurate tree-based detection of code clones. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE 2007). IEEE, 2007. 96−105.

[6] Russell R, Kim L, Hamilton L, *et al*. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications (ICMLA). IEEE, 2018. 757−762.

[7] Zhou ZH, Chen SF. Neural network ensemble. Chinese Journal of Computer, 2002, 25(1): 1−8 (in Chinese with English abstract).

[8] Hindle A, Barr ET, Gabel M, *et al*. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122−131

[9] Nachmani E, Marciano E, Burshtein D, *et al*. RNN decoding of linear block codes. arXiv:1702.07560, 2017.

[10] Mou L, Li G, Jin Z, *et al*. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv:1409. 5718, 2014.

[11] Shuai J, Xu L, Liu C, *et al*. Improving code search with co-attentive representation learning. In: Proc. of the 28th Int'l Conf. on Program Comprehension. 2020. 196−207.

[12] Kim Y. Convolutional neural network for sentence classification [MS. Thesis]. University of Waterloo. arXiv:1408.5882v2, 2014.

[13] Li Z, Wu Y, Peng B, *et al*. SeCNN: A semantic CNN parser for code comment generation. Journal of Systems and Software, 2021, 181: 111036.

[14] Wan Y, Shu JD, Sui YL, *et al*. Multi-modal attention network learning for semantic source code retrieval. In: Proc. of the 2019 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 13−25.

[15] Zeng C, Yu Y, Li S, *et al*. DeGraphCS: Embedding variable-based flow graph for neural code search. ACM Trans. on Software Engineering and Methodology, 2023, 32(2): 1−27.

[16] Xie CL, Liang Y, Wang X. Survey of deep learning applied in code representation. Computer Engineering and Applications, 2021, 57(20): 53−63 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.2106-0368]

[17] Hu X, Li G, Xia X, *et al*. Deep code comment generation. In: Proc. of the 26th Conf. on Program Comprehension. 2018. 200−210.

[18] Wen W, Chu J, Zhao T, *et al*. Code2tree: A method for automatically generating code comments. Hindawi Scientific Programming, 2022. https://doi.org/10.1155/2022/6350686

[19] Wei H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the IJCAI. 2017. 3034−3040.

[20] Graves A. Supervised Sequence Labelling with Recurrent Neural Networks. Springer, 2012. 37−45.

[21] Alon U, Zilberstein M, Levy O, *et al*. Code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 1−29.

[22] Gu J, Chen Z, Monperrus M. Multimodal representation for neural code search. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2021. 483−494.

[23] Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. In: Proc. of the Advances in Neural Information Processing Systems. 2017. 30

[24] Devlin J, Chang MW, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810. 04805, 2018.

[25] Liu Y, Ott M, Goyal N, *et al*. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692, 2019.

[26] Radford A, Narasimhan K, Salimans T, *et al*. Improving language understanding by generative pre-training. OpenAI, 2018.

[27] Lewis M, Liu Y, Goyal N, *et al*. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv:1910.13461, 2019.

[28] Ahmad WU, Chakraborty S, Ray B, *et al*. Unified pre-training for program understanding and generation. arXiv:2103.06333, 2021.

[29] Feng Z, Guo D, Tang D, *et al*. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155, 2020.

[30] Guo D, Ren S, Lu S, *et al*. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366, 2020.

[31] Wang X, Wang Y, Mi F, *et al*. SynCoBERT: Syntax-guided multi-modal contrastive pretraining for code representation. arXiv: 2108.04556, 2021.

[32] Guo D, Lu S, Duan N, *et al*. UniXcoder: Unified cross-modal pre-training for code representation. arXiv:2203. 03850, 2022.

[33] Husain H, Wu HH, Gazit T, *et al*. CodeSearchNet challenge: Evaluating the state of semantic code search . arXiv:1909.09436, 2019.

[34] Svajlenko J, Islam JF, Keivanloo I, *et al*. Towards a big data curated benchmark of inter-project code clones. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 476−480.

[35] Mou L, Li G, Zhang L, *et al*. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence (AAAI-16). 2016.

[36] Lü TG, Hong RC, He J, *et al*. Multimodal-guided local feature selection for few-shot learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2068−2082 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6771.htm [doi: 10.13328/j.cnki.jos.006771]

[37] Gu X, Zhang H, Kim S. Deep code search. In: Proc. of the 40th Int'l Conf. on Software Engineering. 2018.

[38] Yang G, Chen X, Cao J, *et al*. Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In: Proc. of the 8th Int'l Conf. on Dependable Systems and Their Applications (DSA). 2021.

[39] Liu B, Li RL, Feng JF. A brief introduction to deep metric learning. CAAI Trans. on in Telligent Systems, 2019, 14(6): 1064−1072 (in Chinese with English abstract).

[40] Gao T, Yao X, Chen D. SimCSE: Simple contrastive learning of sentence embeddings. arXiv:2104.08821, 2021.

[41] Bui ND, Yu Y, Jiang L. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proc. of the 44th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval. 2021. 511−521.

[42] Chen Q, Lacomis J, Schwartz EJ, *et al*. VarCLR: Variable semantic representation pre-training via contrastive learning. In: Proc. of the 44th Int'l Conf. on Software Engineering. 2022. 2327−2339.

[43] Neelakantan A, Xu T, Puri R, *et al*. Text and code embeddings by contrastive pre-training. arXiv:2201.10005, 2022.

[44] Jain P, Jain A, Zhang T, *et al*. Contrastive code representation learning. arXiv:2007.04973, 2020.

[45] Du L, Shi X, Wang Y, *et al*. Is a single model enough? MuCoS: A multi-model ensemble learning for semantic code search. arXiv:2107.04773, 2021.

[46] Rabin MR, Bui ND, Wang K, *et al*. On the generalizability of neural program models with respect to semantic-preserving program transformations. Information and Software Technology, 2021, 135: 106552.

[47] Wei M, Zhang LP. Research progress of code search methods. Application Research of Computers, 2021, 38(11): 3215−3221, 3230 (in Chinese with English abstract). [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

[48] Cho K, Van Merriënboer B, Bahdanau D, *et al*. On the properties of neural machine translation: Encoder-decoder approaches. arXiv:1409.1259, 2014.

附中文参考文献:

[3] 成思强, 刘建勋, 彭珍连, 等. 以 CodeBERT 为基础的代码分类研究. 计算机工程与应用, 2023, 59(24): 277−288. [doi: 10.3778/j.issn.1002-8331.2209-0402].

[7] 周志华, 陈世福. 神经网络集成. 计算机学报, 2002, 25(1): 1−8.

[16] 王霞, 梁瑶, 谢春丽. 深度学习在代码表征中的应用综述. 计算机工程与应用, 2021, 57(20): 53−63. [doi: 10.3778/j.issn.1002-8331.2106-0368]

[36] 吕天根, 洪日昌, 何军, 等. 多模态引导的局部特征选择小样本学习方法. 软件学报, 2023, 34(5): 2068−2082. http://www.jos.org.cn/1000-9825/6771.htm [doi: 10.13328/j.cnki.jos.006771]

[39] 刘冰, 李瑞麟, 封举富. 深度度量学习综述. 智能系统学报, 2019, 14(6): 1064−1072.

[47] 魏敏, 张丽萍. 语义代码检索方法研究进展. 计算机应用研究, 2021, 38(11): 3215–3221, 3230. [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

杨宏宇(1999－), 女, 硕士生, CCF 学生会员, 主要研究领域为代码数据挖掘, 代码智能.

马建辉(1975－), 男, 博士, 副教授, 主要研究领域为数据挖掘, 机器学习, 区块链.

侯旻(1995－), 女, 博士, 讲师, 主要研究领域为推荐系统, 时间序列分析, 金融数据挖掘.

沈双宏(1995－), 男, 博士生, CCF 学生会员, 主要研究领域为教育数据挖掘, 用户建模.

陈恩红(1968－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域是机器学习, 数据挖掘, 社会网络, 个性化系统推荐.