软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(2):662−683 [doi: 10.13328/j.cnki.jos.007504] [CSTR: 32375.14.jos.007504]

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

基于依赖感知分层神经网络的代码注释增强方法*

张育博[1,2], 姚开春[1,3], 张立波[1,3], 武延军[1,3], 赵 琛[1,3]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 姚开春, E-mail: yaokaichun@outlook.com

摘 要: 作为软件工程领域的一项新兴技术, 源代码自动生成注释旨在为给定的代码片段生成自然语言描述. 目前最先进的代码注释技术采用编码器-解码器神经网络模型: 编码器提取源代码的语义表示, 而解码器则将其转换为人类可读的代码注释. 然而, 许多现有方法将输入的代码片段视为独立函数, 往往忽略了目标函数与其调用的子函数之间的上下文依赖关系. 忽视这些依赖关系可能导致关键语义信息的缺失, 从而降低生成注释的质量. 为此, 提出了一种函数依赖感知的分层代码注释神经网络模型 DHCS (dependency-aware hierarchical code summarization). DHCS 通过显式建模目标函数与其子函数之间的分层依赖关系, 旨在提高代码注释的质量. 采用了一个分层编码器, 包括子函数编码器和目标函数编码器, 使模型能够有效地捕捉局部和上下文的语义表示. 同时, 引入了一项自监督任务, 即掩码子函数预测, 以增强子函数的表示学习. 此外, 提出挖掘子函数的主题分布, 并将其与主题感知的复制机制相结合, 集成到注释解码器中. 因此, 它能够直接从子函数中提取关键信息, 从而更有效地生成目标函数的注释. 最后, 在针对 Python、Java 和 Go 语言构建的 3 个真实数据集上进行了大量实验, 结果充分验证了所提方法的有效性.

关键词: 代码注释生成; API 文档; 分层神经网络; 自监督任务

中图法分类号: TP311

中文引用格式: 张育博, 姚开春, 张立波, 武延军, 赵琛. 基于依赖感知分层神经网络的代码注释增强方法. 软件学报, 2026, 37(2): 662–683. http://www.jos.org.cn/1000-9825/7504.htm

英文引用格式: Zhang YB, Yao KC, Zhang LB, Wu YJ, Zhao C. Code Summarization Enhancing Method with Dependency-aware Hierarchical Neural Network. Ruan Jian Xue Bao/Journal of Software, 2026, 37(2): 662–683 (in Chinese). http://www.jos.org.cn/1000-9825/7504.htm

Code Summarization Enhancing Method with Dependency-aware Hierarchical Neural Network

ZHANG Yu-Bo[1,2], YAO Kai-Chun[1,3], ZHANG Li-Bo[1,3], WU Yan-Jun[1,3], ZHAO Chen[1,3]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: As an emerging technique in software engineering, automatic source code summarization aims to generate natural language descriptions for given code snippets. State-of-the-art code summarization techniques utilize encoder-decoder neural models. The encoder extracts the semantic representations of the source code, while the decoder translates them into human-readable code summaries. However, many existing approaches treat input code snippets as standalone functions, often overlooking the context dependencies between the target function and its invoked subfunctions. Ignoring these dependencies can result in the omission of crucial semantic information, potentially reducing the quality of the generated summaries. To this end, this study proposes a dependency-aware hierarchical code summarization

* 收稿时间: 2025-03-21; 修改时间: 2025-06-26; 采用时间: 2025-07-18; jos 在线出版时间: 2025-09-02
CNKI 网络首发时间: 2025-11-27

neural model, DHCS. DHCS is designed to improve code summarization by explicitly modeling the hierarchical dependencies between the target function and its subfunctions. The proposed approach employs a hierarchical encoder consisting of both a subfunction encoder and a target function encoder, allowing the model to capture both local and contextual semantic representations effectively. Meanwhile, a self-supervised task, namely the masked subfunction prediction, is introduced to enhance the representation learning of subfunctions. Furthermore, the topic distribution of subfunctions is mined and incorporated into a summary decoder with a topic-aware copy mechanism. Therefore, it enables the direct extraction of key information from subfunctions, facilitating more effective summary generation for the target function. Finally, extensive experiments are conducted on three real-world datasets constructed for Python, Java, and Go languages, which clearly validate the effectiveness of the proposed approach.

Key words: code summarization generation; API document; hierarchical neural network; self-supervised task

1 引 言

代码注释是对源代码的一种简洁的自然语言描述, 旨在以易于理解的方式阐明代码背后的逻辑或目的. 通常, 代码注释仅包含一个简短的句子, 能够为程序员提供快速理解代码片段功能的途径. 例如, 在浏览一个 Python 方法时, 程序员可能会看到类似“在 *a* 和 *b* 中查找最长匹配块”的 Pydocs 描述. 这种简洁的注释避免了程序员深入阅读代码, 从而提高效率和理解力. 虽然程序员可以从 Pydocs 和 JavaDocs 等软件文档工具中获得大量的帮助, 但手动创建注释的过程却是一个费时费力的工作. 尤其是在处理代码遗留问题时, 源代码通常缺乏完善的文档, 使得这一问题尤为突出. 因此, 软件工程领域的研究人员一直致力于自动生成代码注释, 以减轻程序员的负担.

近年来, 随着深度学习和自然语言处理的发展, 研究人员开始探索利用机器学习方法来实现自动生成代码注释[1−4]. 这些模型通常采用编码器-解码器的框架及其各种变体, 在这个框架中, 编码器接受代码片段作为输入并将其转化为潜在表示, 这些表示作为解码器生成自然语言描述的基础, 从而阐明代码片段的功能和目的. 例如, 受到神经机器翻译的启发[5,6], Hu 等人[2]和 Ahmad 等人[1]采用序列到序列架构进行代码注释生成. 在这种架构中, LSTM/Transformer 编码器从序列化的抽象语法树 (AST) 中学习潜在的代码表示, LSTM/Transformer 解码器则利用注意力机制来生成注释. 其他研究则设计了专门的编码器/解码器来处理特定形式的源代码输入, 如 AST 或 token 序列[7−10], 或者引入专门的学习目标来促进模型的训练[3,4,11]. 这些创新方法旨在通过深度学习技术提升代码注释的有效性和适应性. 此外, 随着 GPT 系列等超大语言模型的出现, 研究人员也在积极探索像 Code LLaMA[12]和 StarCoder[13]这样的解码器模型来处理多任务.

然而, 上述方法普遍存在一个共同的局限性: 它们将输入的代码片段视为独立的函数, 从而将代码中的依赖关系简化为单一字符串. 换句话说, 它们通常将整个函数或短代码片段作为输入, 供基于 Transformer 或 RNN 的编码器使用, 或者供解码器模型使用. 这种做法在许多常见代码注释数据集 (如 CodeSearchNet、Nematus 和 Funcom) 中也有所应用. 问题在于, 这种设置忽略了在准确生成代码注释时至关重要的函数依赖关系的信息, 而忽视这些依赖关系可能导致遗漏重要的语义信息. 如图 1 所示, 目标函数包含 5 个关键依赖关系 (子函数): iteration、collapse function preconditions、collapse snapshots、collapse function postconditions 和 decorate the function. 依赖关系指的是对项目中其他地方定义元素的调用 (例如 collapse_preconditions), 这些依赖关系对于程序的语义功能至关重要, 忽略它们可能导致对输入代码的理解偏差或理解错误.

本文通过引入一种依赖感知分层神经网络代码注释模型, 即 DHCS, 来解决上述挑战. DHCS 的核心是利用从被调用的依赖 (子函数) 中获得的信息, 以提高生成的代码注释的质量. 为此, 本文采取了多方面的策略. 首先, 构建了一个分层的 Transformer 编码器, 该编码器由目标函数编码器和子函数编码器组成. 子函数编码器用于学习子函数的语义特征, 而目标函数编码器则捕捉目标函数的上下文语义. 接着, 使用传统的 Transformer 解码器生成注释, 并基于层次化编码器编码的语义信息进行条件生成. 此外, 为了增强子函数的表示学习, 引入了一个自监督目标——掩码子函数预测 (masked subfunction prediction, MSP), 该目标增强了本文分层模型的预训练效果. 此外, 假设子函数的注释可以为描述目标函数提供关键线索. 基于此, 本文提出了一种主题感知复制机制, 能够有选择地从被调用的依赖项中提取重要信息. 本研究通过对包含 Python、Java 和 Go 这 3 种语言的新数据集进行全面评估, 经过一系列广泛实验, 最终证明利用被调用依赖项的信息可以显著提高代码注释的质量.

本文的主要贡献总结如下.

● 领域探索: 率先涉足依赖关系感知的代码注释研究领域, 为新的研究范式开辟了道路. 为支持这一方法, 精心构建了 3 个新的真实世界数据集, 分别针对 Python、Java 和 Go 这 3 种编程语言.

● 自监督任务: 提出了“掩码子函数预测”自监督任务, 并为子函数设计了主题感知复制机制, 两者都显著提升了代码注释的性能.

● 验证性研究: 通过大量实验和与最先进 baseline 方法的细致比较, 在代码注释生成任务中验证了模型的有效性.

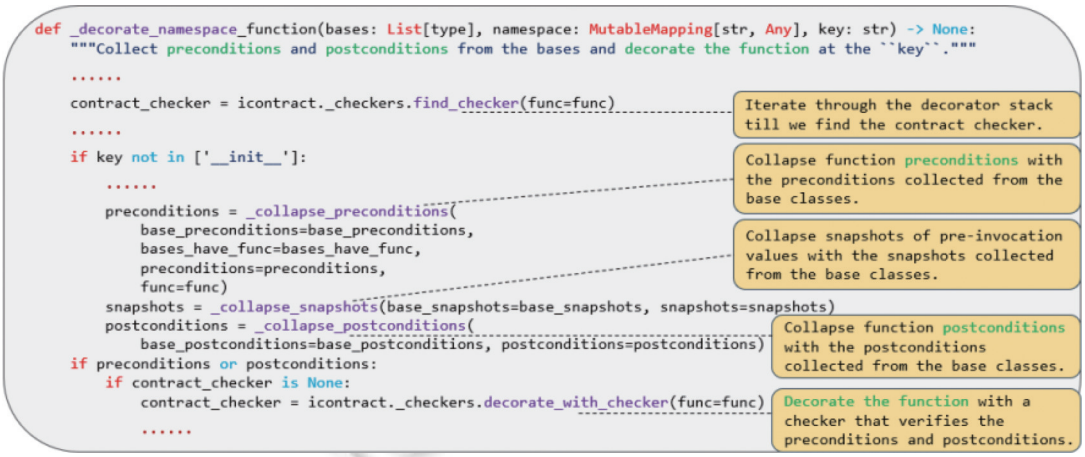


图 1 一个代码注释的示例

2 相关工作

本节讨论了与文本摘要、代码摘要以及基于代码的自监督预训练相关的研究工作.

2.1 文本摘要

文本摘要是从源文本中提取主要内容和关键信息的过程, 同时保留其原始语义. 它可以让人们快速获取文本中讨论的关键内容或主题. 文本摘要大致可分为两种主要类型: 提取式摘要和生成式摘要.

● 提取式摘要方法专注于摘要的相关性和句子的冗余性. 已有一些基于图结构的方法被提出, 如 TextRank[14]、LexPageRank[15]和 CoRank[16]. 这些方法将整个文本构建为一个拓扑图, 并为每个词分配重要性得分, 类似于 PageRank 算法; 然后, 从中提取得分较高的句子形成摘要. 随着深度学习技术的发展, Cao 等人[17]使用 CNN 进行文本分类; Singh 等人[18]使用 CNN 和 Bi-LSTM 提取文本特征; Nallapati 等人[19]采用 GRU 生成摘要.

● 生成式摘要方法通过理解原文的语义重新生成摘要. 基于 seq2seq 框架的神经网络在生成任务中展现出了巨大的潜力. Gu 等人[20]提出了 CopyNet, 生成摘要的同时保留了源文本中的重要信息. See 等人[21]提出了一种指针生成网络, 该网络可以根据词汇分布自动决定是从原文复制还是生成新词, 从而有效解决了罕见词 (OOV) 问题. Zhou 等人[22]在编码器中增加了选择性门控网络, 允许结合词级和语句级的隐藏状态. 随着预训练语言模型如 MASS[23]、BART[24]和 T5[25]的出现, Zhang 等人[26]结合预训练的 BERT 和 Transformer 提出了两阶段模型. Cao 等人[27]通过增强注意力机制, 提出了注意力头掩码技术, 聚焦于文本中的重要内容.

2.2 代码摘要

代码摘要与文本摘要在本质上存在显著差异, 主要体现在它们独特的逻辑结构上. 与一般文本不同, 代码具有特有的组织方式, 这使得它在语义和结构信息方面更加丰富、更加复杂.

近年来, 已经有多种网络结构被用来提取源代码的语义和结构信息以生成摘要. Iyer 等人[28]提出了 Code-NN, 利用结合 LSTM 的注意力机制来捕捉语义信息. Allamanis 等人[29]提出了一种基于注意力的 CNN, 用于学习代码

的局部时序不变和长距离主题注意力特征. Ahmad 等人[1]将 Transformer 应用于建模代码 token 之间的成对关系. Mou 等人[30]设计了一种方法, 通过在程序的抽象语法树 (AST) 上使用卷积核来捕捉结构信息. Hu 等人[2]提出了 DeepCom, 实现了一种名为 SBT 的新遍历方法, 用于获取结构信息, 同时考虑了通过经典遍历方法获得的序列中信息丢失问题. Wan 等人[3]提出了一种 Hybrid2Seq 方法, 旨在通过在代码的 token 层次和结构层次上融合词汇和语法信息, 最大化代码序列和结构细节的利用. 类似地, LeClair 等人[8]提出了 ast-attendgru, 结合基于代码的词表示和基于 AST 的结构表示来独立学习结构信息.

随着代码理解领域中预训练和微调模型的流行, 出现了一些具有突破性的进展. 近年来, 一个有影响力的工作是 BERT[31], 它利用 Transformer 架构从前后上下文中学习. 基于 BERT 的成功, Kanade 等人[32]将其应用于源代码, 提出了 CuBERT, 通过为分词后的源代码生成预训练的上下文嵌入, 而无需显式建模代码特定的信息. 为了有效地融合多模态数据, CodeBERT[33]作为首个针对多种编程语言的双模态预训练模型被提出, 通过在训练过程中随机屏蔽某些词语来增强模型的理解能力. 然而, 这些方法主要将代码片段视为 token 序列, 忽略了代码内部的结构.

GraphCodeBERT[34]通过利用代码的语义结构并基于数据流学习代码表示来应对这一问题. 与传统的从左到右的代码生成任务不同, Fried 等人[35]提出了 InCoder, 用于处理包含大量随机性的代码编辑任务. 它采用 GPT 架构, 利用因果屏蔽填充任意左侧和右侧上下文模块, 实现双向上下文填充. Ahmad 等人[36]提出了 PLBART, 将 BERT 的双向编码器和 GPT 的单向解码器相结合, 能够在编码器和解码器两侧同时学习, 从而加速并提升下游任务的性能, 同时增强生成能力. 基于 T5[25]的基础, Wang 等人[37]提出了 CodeT5, 通过预训练时融入代码结构信息, 使得模型学习更丰富的代码表示, 尤其在理解标识符和关注预测有意义的代码词上有所提升. Gao 等人[38]提出了 SG-Trans 方法, 将代码的结构性信息融入 Transformer 模型中. 抽象语法树 (AST) 通常比源代码大得多, 但现有方法通过直接将整个 AST 提供给编码器, 忽视了这一大小限制. 为解决这个问题, Nagaraj 等人[39]提出 AST-MHSA, 利用多头自注意力机制从 AST 中提取重要的语义信息. 从整合外部知识的角度出发, Shahbazi 等人[40]提出了 APIContext2Com, 通过结合预定义的应用程序编程接口 (API) 上下文来提高生成摘要的效果. 详细的 API 信息阐明了代码片段的功能, 帮助更好地生成代码摘要. 关注代码中的额外动作词, Li 等人[41]开发了一种方法, 通过额外的动作词预测模块辅助代码摘要生成, 其中使用动作预测器来预测代码摘要中的主要动作, 并将其作为提示词增强摘要生成模型的表现. 为了生成类似人工编写的摘要并保留事实细节, Sun 等人[42]基于抽取与生成框架实现了一个代码摘要原型 EACS, 继承了抽取式和生成式方法的优点, 同时规避了它们各自的缺点.

近年来, 大型语言模型 (LLM) 广泛应用于各种与代码相关的任务. 例如, Wang 等人[43]提出了 CodeT5+, 它可以灵活地在仅编码器、仅解码器和编码器-解码器模式中运行, 以适应不同的下游代码应用任务. Meta 公司推出了基于 LLaMA 2 的代码生成模型 Code LLaMA[12], 它在开源框架内增强了填充能力, 扩展了长上下文输入能力, 并具备 zero-shot 编程任务能力. BigCode 社区推出了 StarCoder[13], 它在 The Stack 数据集上进行训练, 具备 8k 上下文长度、填充能力, 并通过多查询注意力实现了快速的大批量推理. 结果显示, 相较于其他代码大语言模型, StarCoder 表现出色. 在预训练阶段, DeepSeek[44]首次尝试通过结合项目级别的代码数据, 显著提升了跨文件的代码生成能力, 在开源和闭源模型中都实现了最先进的性能. 与这些方法不同, 本文的模型强调函数依赖关系, 而不是将其视为单纯的字符串, 从而使其能够学习代码的更深层次特征.

2.3 基于代码的自监督预训练

预训练范式通过利用大量可用的文本数据, 使语言模型能够捕捉语言的内在模式、结构和语义. 通过这种方式, 模型能够更深入地理解语法、句法和上下文. 这种方法在自然语言处理领域得到了广泛应用, 现在也越来越多地被应用于代码领域.

在预训练中, 一种常见的目标是掩码语言模型 (masked language modeling, MLM)[31], 该方法通过掩盖序列中的某些 token, 然后让模型根据上下文信息正确预测这些被掩盖的 token. 基于 MLM, 已经有一些积极的探索, 如掩盖连续片段[23], 在实体或短语级别进行掩码[45]. 与此不同, Clark 等人[46]提出了替换 token 检测 (replaced token detection, RTD) 预训练目标, 将 token 替换为合理的替代项, 而不是被掩盖. 预训练模型随后预测这些 token 是否

被替换, 而不是像 MLM 那样预测原始 token. Li 等人[47]随后提出了一种基于 RTD 的方法, 用于 few-shot 学习, 通过使用模板和标签描述词将输入转化为自然语言提示, 预训练模型预测最少被替换的标签描述词. AraELECTRA[48]和 DeBERTaV3[49]等其他工作也验证了 RTD 的有效性. Lachaux 等人[50]提出了一个新的目标, 称为标识符去混淆 (identifier deobfuscation, DOBF), 该方法专注于去混淆源代码中的标识符名称. 与 MLM 不同, 当选择一个标识符时, 该标识符的所有实例都被替换为相同的特殊 token. 这种方法旨在利用编程语言的结构信息, 通过预训练恢复被混淆的源代码. 对比学习 (contrastive learning, CL) 是另一种广泛使用的技术, 它通过训练模型最小化相似正样本之间的距离, 并最大化不同负样本之间的距离. 例如, Jain 等人[51]提出了 ContraCode, 利用对比学习聚焦于代码的功能, 而不是其形式. 该模型通过预训练识别功能上相似的程序变体. 然而, 程序的功能不仅依赖于代码序列, 不同的 token 和结构可以实现相同的功能. 为了解决这一问题, Ding 等人[52]提出了 BOOST, 通过对比学习使得功能相同的代码在表示空间中更接近. 与这些方法不同, 本文提出了一种在子函数级别进行自监督学习的目标, 旨在增强对函数依赖关系表示的学习.

3 问题定义

代码生成注释是指生成简洁、易于人类理解的自然语言描述或总结, 概括目标函数中所包含的功能和逻辑. 与之前的方法不同, 本文的任务重点关注目标函数中的上下文依赖关系, 以增强代码注释的效果. 具体而言, 设 X 表示目标函数的源代码, 它由 m 个元素组成, 即 $X=[x_1,x_2,\dots,x_m]$, 其中 x_i ($i\in[1,m]$) 表示源码中的 token, $\hat{X}_1,\dots,\hat{X}_j$ 表示 X 中的依赖关系 (即子函数, 下文统称为子函数), $\hat{X}_i$ ($i\in[1,j]$) 表示为一个包含 l 个 token 的序列 $[\hat{x}_1^i,\hat{x}_2^i,\dots,\hat{x}_l^i]$. 本文的任务是基于输入 X 生成目标序列 $Y=[y_1,y_2,\dots,y_n]$, 其中 Y 包含 n 个单词.

通常, 条件概率使用由参数 θ 特征化的参数化函数表示: $P(Y|X)=P(Y|X;\theta)$. 在训练过程中, 目的是识别最优的 θ 值, 以最大化训练数据集中 (X,Y) 对的条件概率. 一般而言, 模型旨在根据先前的单词预测注释中的下一个单词, 那么上述的条件概率可以被分解为一系列个别条件概率的乘积:

$$P(Y|X;\theta)=\prod_{i=1}^{n}P(y_i|\{y_1,\dots,y_{t-1}\},X;\theta) \tag{1}$$

最后, 本文使用负对数似然损失作为优化目标:

$$\mathcal{L}_{\text{nll}}=-\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log P\left(y_t^i|y_{<t}^i,X^i;\theta\right) \tag{2}$$

其中, N 是训练数据集中 (X^i,Y^i) 对的数量, T 是目标注释的长度.

4 方 法

本节详细介绍了 DHCS 模型, 该模型采用典型的神经编码器-解码器架构. DHCS 的整体框架如后文图 2 所示. 在此框架中, 设计了一个分层 Transformer 编码器, 以更好地捕捉目标函数和调用的子函数之间的结构化表示, 其中子函数编码器用于学习子函数的语义表示, 而目标函数编码器用于捕捉目标函数的上下文语义. 此外, 本文还提出了掩码子函数预测任务, 以增强子函数表示的学习. 最后, 本文使用主题感知复制解码器, 基于学到的上下文语义和子函数的主题分布生成代码注释.

4.1 分层编码器

4.1.1 子函数编码器

子函数编码器接收子函数的源代码作为输入. 给定子函数 $\hat{X}_i$ ($i\in[1,j]$), 本文首先使用字节对编码 (BPE) 分词器[53], 根据 Radford 等人[54]的方法将 $\hat{X}_i$ 分词为一系列 token: $[\hat{x}_1^i,\hat{x}_2^i,\dots,\hat{x}_l^i]$. 然后, 将该 token 序列传递到嵌入层, 得到嵌入表示 $e(\hat{X}_i)$. 最后, 这些嵌入被输入到子函数编码器中, 以编码子函数 $\hat{H}$ 的语义特征. 具体来说, 如下所示:

$$e(\hat{X}_i)=[e(\hat{x}_1^i),\dots,e(\hat{x}_l^i)]=EmbeddingLayer(\hat{X}_i) \tag{3}$$

$$\hat{H} = [\hat{h}_1, \ldots, \hat{h}_l] = SubEncoder(e(\hat{X}_i)) \tag{4}$$

其中, *SubEncoder* 表示子函数编码器的编码过程, 它采用 12 层的 Transformer 编码器, 且其参数由基于代码的预训练语言模型初始化. 此外, 本文在其输出层上添加了一个额外的均值操作层, 并将特征向量序列 $\hat{H}$ 转换为统一的向量表示 H', 以便它们符合目标函数编码器的表示, 其表示如公式 (5), 其中 H_i^* $(i = 1,3)$ 的解释见第 4.3.1 节.

$$H' = \frac{1}{l}\sum_{i=1}^{l}\hat{h}_i \tag{5}$$

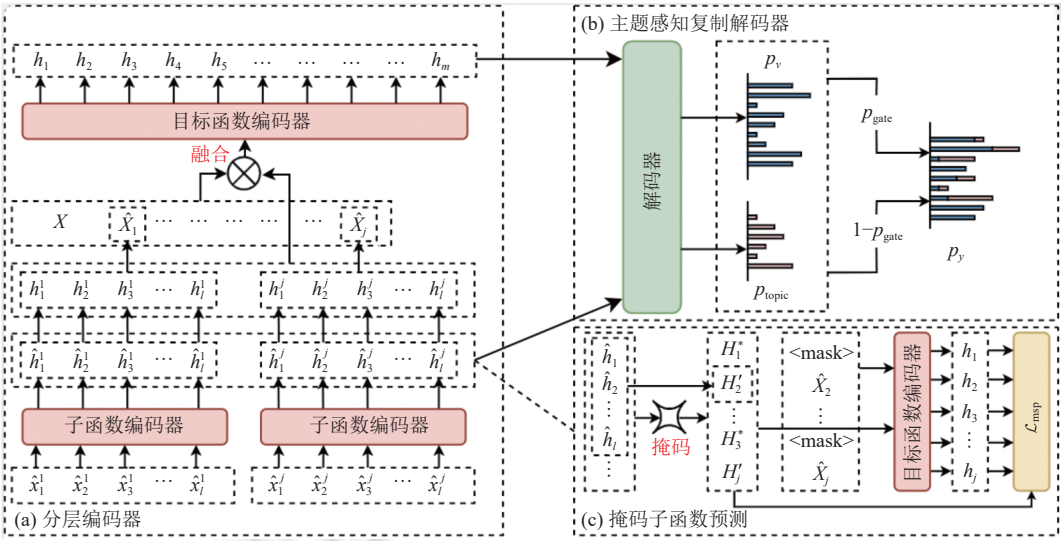


图 2 DHCS 的整体框架

4.1.2 目标函数编码器

目标函数编码器接收目标函数的源代码 X 作为输入. 它采用与子函数编码器相同的分词工具、结构和参数初始化. 因此, 它可以与子函数编码器共享相同的参数, 从而加速训练收敛. 具体来说, 给定目标函数代码片段 X, 在分词和添加特殊 token 后, 本文首先生成一个 token 序列 $[x_1, x_2, \ldots, x_m]$. 然后, 执行与子函数 $\hat{X}$ 编码器相同的操作, 通过嵌入层和编码器获得目标函数的上下文语义表示 H, 具体表示如下:

$$e(X) = [e(x_1), \ldots, e(x_m)] = EmbeddingLayer(X) \tag{6}$$

$$H = [h_1, \ldots, h_m] = TargetEncoder(e(X)) \tag{7}$$

其中, *TargetEncoder* 表示目标函数编码器的编码过程. 需要注意的是, 调用的子函数在 X 中仅以函数名称的字符串形式出现, 这无法完全体现子函数的逻辑或目的. 为了更有效地将子函数包含到目标函数中, 本文采用了以下两种特征融合方式.

● 合并融合 (Sum-fusion). 将子函数编码器学习的特征向量加到目标函数 X 中相应 token 的嵌入向量上. 以输入序列 $[x_1, x_2, \ldots, x_m]$ 为例, 如果 x_j 对应一个子函数, 那么 x_j 的嵌入将按公式 (8) 更新:

$$e(x_j) \leftarrow H' + e(x_j) \tag{8}$$

● 替换融合 (Replace-fusion). 直接将子函数名称对应的 token 嵌入替换为子函数的向量表示, 并将更新后的向量表示作为输入:

$$e(x_j) \leftarrow H' \tag{9}$$

合并融合的方法能够有效保持目标函数原始的 token 信息, 同时引入子函数的特征表示, 从而更全面地增强目标函数的表示能力. 相比之下, 替换融合的方法简化了处理流程, 但牺牲了目标函数原始信息的表达能力. 为了更全面地增强目标函数的表示能力, 最大限度地保留目标函数自身的关键特征 (即同时保持原始 token 信息和引

入子函数特征), 本文选择采用合并融合作为融合方式, 并将其应用于后续的实验与分析中.

值得注意的是, 子函数名称可能会被分词为多个连续的 token. 因此, 在融合过程中, 本文将每个 token 与学习的子函数表示进行求和或替换. 与之前的方法不同, 在这些方法中, 子函数名仅作为一个字符串处理. DHCS 采用了一种更合理的层次结构, 通过增强目标函数与调用的子函数之间的表示, 从而提升目标函数的表示能力.

4.2 主题感知复制解码器

本文同样采用 Transformer 作为解码器的骨架, 用于注释的生成. 解码器的输出是目标序列 $Y=[Y_1,Y_2,\ldots,Y_n]$. 在解码步骤 t 中, 解码器首先计算基于前面生成的 token: $[y_1,\ldots,y_{t-1}]$ 和上下文表示 H 的隐藏状态 h'_d, 然后通过 *Softmax* 操作计算每个注释 token 在词汇表上的生成概率 $P_v(t)$:

$$h'_d = Decoder(H,y_1,\ldots,y_{t-1}) \tag{10}$$

$$P_v(y_t|y_1,\ldots,y_{t-1}) = Softmax(h'_d) \tag{11}$$

直观上, 子函数的注释将有助于目标注释的生成. 受复制机制的启发[21,55], 本文设计了一个简单而有效的主题感知复制机制, 允许解码器在生成词汇时既能从词汇表中生成单词, 也能从子函数的主题中复制信息. 具体来说, 为了获得子函数的主题分布, 首先利用解码器根据子函数表示 $\hat{H}$ 生成子函数的注释 token 分布 $P'_v(y_1),P'_v(y_2),\ldots,P'_v(y_n)$, 然后计算注释 token 序列分布的均值. 公式如下:

$$h'_{d'} = Decoder\left(\hat{H},y_1,\ldots,y_{t-1}\right) \tag{12}$$

$$P_{v'}(y_t|y_1,\ldots,y_{t-1}) = Softmax(h'_{d'}) \tag{13}$$

$$P_s = \frac{1}{n}\sum_{t=1}^{n} P_{v'}(y_t) \tag{14}$$

其中, P_s 是一个子函数的主题分布. 最后, 本文通过计算每个子函数的主题分布 P_s 均值, 获得目标函数中所有子函数的最终主题分布 P_{topic}. 实际上, P_{topic} 可以视为词汇表中关键词的分布, 这些关键词可以部分反映子函数的逻辑和功能. 值得注意的是, 本文在计算主题分布和最终生成注释时使用相同的解码器.

一旦获得了主题分布, 本文设计了一种软选择机制来决定是从词汇表中生成单词, 还是从子函数的主题中复制信息. 具体来说, 在解码步骤 t 中, 根据解码器的状态 h'_d 计算选择概率 $P_{\text{gate}}\in[0,1]$, 如下所示:

$$P_{\text{gate}} = \sigma(MLP(h'_d)) \tag{15}$$

其中, *MLP* 表示一个多层感知机, 用于特征维度转换, σ 是 *Sigmoid* 函数. 最后, 通过联合概率计算在解码步骤 t 中生成 token y_t 的概率:

$$P_{y_t} = P_{\text{gate}}\times P_v + (1-P_{\text{gate}})\times P_{\text{topic}} \tag{16}$$

通过选择门 P_{gate}, 模型可以自适应地决定如何利用子函数的主题来辅助目标函数注释的生成.

4.3 掩码子函数预测

近年来, 预训练策略已经展示了它们显著提高下游任务性能的能力[33,34,37]. 在本文的 DHCS 中, 编码器和解码器可以使用基于代码的预训练语言模型 (如 CodeT5) 进行初始化, 从而继承编程语言中有价值的语义知识. 然而, 单靠这些语义知识可能不足以捕捉目标函数与其相关子函数之间深层的上下文层次关系. 为了解决这个问题, 本文精心设计了一种名为“掩码子函数预测”的自监督训练目标. 这一目标旨在通过深入挖掘子函数的上下文细节, 丰富其语义表示.

在掩码子函数预测任务中, 本文采用了动态的子函数名称掩码技术, 并按照以下步骤进行掩码子函数预测.

4.3.1 动态子函数名称掩码

根据第 3 节的定义, 目标函数 X 包含一个或多个子函数 $[\hat{X}_1,\hat{X}_2,\ldots,\hat{X}_j]$. 在通过子函数编码器处理后, 获得了学习到的表示序列 $\mathcal{H}=[H'_1,H'_2,\ldots,H'_j]$. 在训练阶段, 对于当前批次中的每个目标函数, 本文随机选择 k 个子函数 $[H'_1,\ldots,H'_k]$, 其中 $k<j$, 并用一个随机初始化的掩码子函数向量 H^* 替换这些子函数. 同时, 用特殊 token “<mask>”, 掩盖 X 中对应的子函数名称. 例如, 如果随机选择第 1 个和第 3 个子函数进行掩码, 那么掩码后的目标函数将变为

$X=[<\text{mask}>,\hat{X}_2,<\text{mask}>,\ldots,\hat{X}_j]$, 掩码后的子函数表示为 $H'=[H_1^*,H_2',H_3^*,\ldots,H_j']$. 这一动态采样过程在每次训练步骤中对批次中的每个目标函数进行迭代. 因此, 相同目标函数在不同步骤中可能会遇到不同的掩蔽子函数位置. 这种动态掩码策略使模型能够更有效地预测目标函数上下文中子函数的上下文语义.

4.3.2 掩码子函数预测

在掩码子函数预测的上下文中, 本文在一个多分类框架内进行操作. 具体而言, 将当前批次中的所有掩码子函数汇总到一个候选子函数池中. 模型的任务是从这个池中正确预测相关的子函数. 对于每个掩码的子函数位置, 当前上下文中占据该位置的原始子函数作为正样本. 相反, 在当前目标函数中共同掩码的子函数以及同一批次中其他目标函数中相应位置的子函数被视为负样本. 在训练阶段, 当掩码后的 X 和 $\mathcal{H}$ 输入目标函数编码器时, 得到上下文表示 $[h_1,\ldots,h_j]$. 然后, h_k 将用于预测原始子函数表示 H_k'. 对于一个批次中的 B 个掩码子函数, 其中预测的子函数表示 $h\in\mathcal{R}^{B\times D}$, 真实的子函数表示 $\mathcal{H}\in\mathcal{R}^{B\times D}$, 其中 D 表示特征维度, 可以为当前批次中的每一对掩码子函数计算相似度矩阵, 具体如下:

$$Sim(\mathcal{H},h)=Cosine\_Similarity(\mathcal{H},h) \tag{17}$$

其中, $Sim(\mathcal{H},h)$ 是第 j 个预测子函数表示和第 i 个原始子函数表示之间的预测相似度. 本文使用 *Softmax* 函数将相似度标准化为每个子函数的预测概率, 如公式 (18) 所示:

$$P(\mathcal{H},h)=\frac{\exp(Sim(h_j,\mathcal{H}_i))}{\sum_{r=1}^{B}\exp(Sim(h_j,\mathcal{H}_r))} \tag{18}$$

其中, $r\in[1,B],r\neq j$, 可以被视为 h_j 随机采样的负类. 最后, 本文可以对所有掩码子函数计算交叉熵损失:

$$\mathcal{L}_{\text{msp}}=-\frac{1}{B}\sum_{i=1}^{B}\log P(\mathcal{H}_i|h_j) \tag{19}$$

4.4 模型微调和推理

在初始的预训练阶段完成后, 本文将对 DHCS 进行微调, 以适应代码注释的下游任务. 微调过程仅通过优化交叉熵损失 (即公式 (2)) 来进行, 该损失函数衡量预测的注释生成概率与真实注释标签之间的差异. 需要注意的是, 在预训练阶段引入的任何子函数掩码都会在微调过程中移除, 以防止丢失关键的上下文代码信息. 一旦模型完成了预训练和微调, 便进入推理阶段, 生成目标函数的注释. 本文在算法 1 中提供了训练和推理过程的全面概述.

算法 1. 方法的训练和推理过程.

输入: 目标函数 X 和子函数 $[\hat{X}_i,\ldots,\hat{X}_j]$;
输出: 目标注释 Y.

1. 初始化子函数编码器、目标函数编码器和解码器参数;
2. 使用交叉熵损失 (公式 (19)) 对子函数编码器和目标函数编码器进行预训练;
3. 使用预训练获得的参数更新编码器;
4. 对模型的参数进行微调, 见公式 (2);
5. 使用最后一步学习的参数进行推断生成注释.

5 实验与分析

本节进行了广泛的实验, 以评估 DHCS 模型的有效性.

5.1 研究问题

本文的主要研究目标是评估 DHCS 在代码生成注释领域的提升效果. 此外, 本研究还希望验证两个以子函数为导向的创新对性能的影响, 即主题感知复制机制和掩码子函数预测任务. 为此, 本文提出了以下 4 个研究问题.

RQ1: 与一些最先进的 baseline 方法相比, DHCS 是否能提高代码注释的性能?

RQ2: 与直接解码方法相比, 主题感知复制机制对模型性能的影响如何?

RQ3: 与非自监督训练方法相比, 掩码子函数预测任务对模型性能的影响如何?

RQ4: 与最先进的基准方法相比, DHCS 的分层结构是否增加了计算复杂性?

本文提出 RQ1, 以评估所提出的 DHCS 模型在包含子函数调用的情况下, 是否优于其他最先进的 baseline 方法. 本文提出 RQ2 和 RQ3, 以评估 DHCS 模型中的每个组件. 同时还提出 RQ4, 以评估 DHCS 提出的分层结构是否增加了计算复杂性.

5.2 数据集

常用于代码生成注释任务的数据集主要包含 Nematus[56]、Funcom[8]和 CodeSearchNet[57], 它们包含 6 种编程语言 (Python、Java、JavaScript、Ruby、Go 和 PHP). 然而, 这些数据集中的目标函数被视为独立函数, 因此无法支持本文的研究问题.

为了解决代码生成注释中上下文依赖的问题, 本文引入了 3 个新的数据集, 分别针对 Python、Java 和 Go 语言. 具体来说, 在 CodeSearchNet 数据集中, 本文发现每个样本包含一个目标函数及其对应的注释, 并且在目标函数的函数体内, 往往会出现一个或多个子函数调用. 因此, 本文通过 CodeSearchNet 中提供的 url 信息, 收集和抓取了这些被调用的子函数. 在收集过程中, 选择保留包含一个或多个子函数调用的样本, 并过滤掉那些没有此类依赖关系的样本. 值得注意的是, 本文重点收集了 Python、Java 和 Go 语言的子函数数据, 因为这 3 种语言是目前最常用且最具代表性的编程语言. 本研究计划在未来的研究中继续对其他 3 种语言的数据集进行构建.

本文构建的 3 个新数据集分别包含了 Python、Java 和 Go 语言的 43 484、34 670 和 52 922 个样本. 本文将 Python 数据集分为 38 175 个训练集、2 441 个验证集和 2 868 个测试集; Java 数据集分为 29 670 个训练集、2 500 个验证集和 2 500 个测试集; Go 数据集分为 47 558 个训练集、2 332 个验证集和 3 032 个测试集.

在表 1 中, 本文展示了数据集的关键信息统计, 包括目标函数中的平均代码 token 数、子函数中的平均代码 token 数, 以及目标函数中平均的子函数调用次数. 值得注意的是, 本文的数据集每个样本中包含至少一个子函数的调用.

表 1 数据集的关键信息统计

编程语言	数据集	样例数量	子函数平均数	目标函数的平均token数	子函数的平均token数
Python	训练集	38 175	1.44	134.8	148.1
	验证集	2 441	1.40	134.9	153.3
	测试集	2 868	1.49	132.9	152.7
Java	训练集	29 670	1.36	131.0	138.7
	验证集	2 500	1.56	151.9	151.9
	测试集	2 500	1.43	132.1	140.0
Go	训练集	47 558	1.64	137.5	100.8
	验证集	2 332	1.76	120.5	94.4
	测试集	3 032	1.69	131.7	104.1

此外, 本文提供了基于子函数数量的样本分布的深入分析. 如后文图 3 所示, 外圈为训练集, 中圈为验证集, 内圈为测试集. 可以观察到, Python 语言中包含多个子函数的样本占比为 27.8%、25.7% 和 30.3% (训练集、验证集和测试集), 而 Java 语言和 Go 语言的比例则分别为 23.4%、36.7%、26.4% 和 35.0%、37.1%、38.6%.

5.3 评估指标

5.3.1 自动评估指标

在评估代码注释生成模型时, 本文采用了 3 种广泛认可的评估指标: *BLEU*[58]、*METEOR*[59]和 *ROUGE-L*[60]. 为了计算每个指标在测试集上的得分, 计算了所有生成注释的平均得分. 需要注意的是, 这些指标的得分越高, 表示模型的表现越好. 接下来, 将详细介绍这 3 种指标.

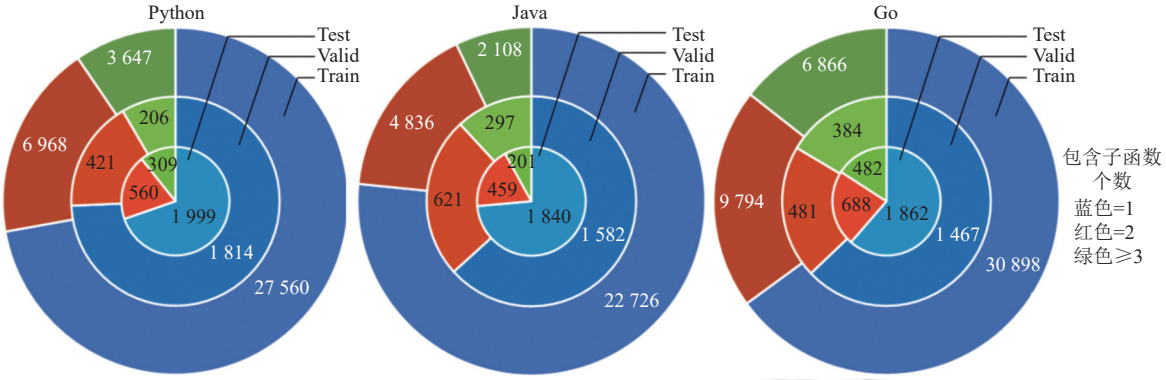


图 3　在 Python、Java 和 Go 数据集中包含不同数量的子函数

BLEU (bilingual evaluation understudy): 由 IBM 在 2002 年提出, *BLEU* 是自然语言处理 (NLP) 任务中广泛使用的评估指标, 如机器翻译和文本摘要. *BLEU* 主要基于精度的概念, *BLEU* 有多种变体, 按 *n*-gram 评估分类. 最常用的变体包括 *BLEU*-1、*BLEU*-2、*BLEU*-3 和 *BLEU*-4, 其中 *n*-gram 表示连续词汇的数量. 值得注意的是, *BLEU*-4 相比 *BLEU*-1 等指标更加注重句子的流畅度. *BLEU*-4 的计算公式为:

$$BLEU\text{-}4 = bp \cdot \exp\left(\sum_{i=0}^{n} w_n \log p_n\right) \tag{20}$$

其中, n 取值为 4, 表示 *n*-gram 的长度, w_n 是正权重, p_n 是精度得分. bp 是长度惩罚因子, 用于惩罚生成摘要长度与参考摘要长度不一致的情况. bp 的计算公式为:

$$bp = \begin{cases} 1, & c > r \\ e^{1-\frac{r}{c}}, & c \leqslant r \end{cases} \tag{21}$$

其中, r 表示参考摘要的长度, c 表示生成摘要的长度. *BLEU*-4 评分的范围在 0–100% 之间, 得分越高表示生成的摘要质量越高, 并且与参考摘要的匹配度越高. 近年来, *BLEU*-4 已成为研究中最主要的评估指标. 基于这一趋势, 本文同样采用 smoothed *BLEU*-4[61]作为评估指标.

METEOR (metric for evaluation of translation with explicit ordering): 作为一种综合评估指标, 涵盖了精确度和召回率, 旨在解决 *BLEU* 可能存在的局限性. *METEOR* 的计算涉及使用一个校准集 m, 该集源自 WordNet 词库. 它利用这个校准集, 通过比较最佳候选翻译与参考翻译, 计算精度 P 和召回率 R 的调和平均值, 计算公式如下:

$$F_{\text{mean}} = \frac{PR}{\alpha P + (1-\alpha)R} \tag{22}$$

$$Penalty = \gamma\left(\frac{\#chunks}{\#unigram\_matched}\right)^{\theta} \tag{23}$$

$$METEOR = F_{\text{mean}} \times (1 - Penalty) \tag{24}$$

其中, *Penalty* 是惩罚因子, *#chunks* 表示候选翻译与参考翻译之间连续匹配的片段数, *#unigram_matched* 指的是匹配的单个词的数量. *#chunks* 越小, 候选翻译与参考翻译的匹配程度越高. α, γ 和 θ 是用于评估的超参数, 通常在机器翻译评估中取值为 $\alpha = 0.9$, $\gamma = 0.5$, $\theta = 0.3$.

METEOR 是一种综合评估指标, 不仅考虑了整个语料库范围内的精确度和召回率, 还考虑了句子流畅性以及同义词对语义理解的影响. 然而, 需要注意的是, *METEOR* 对文本长度可能较为敏感, 并且可能依赖于外部资源来获取参考信息.

ROUGE-L (recall-oriented understudy for gisting evaluation): 是一种主要聚焦于评估召回率的度量. 它特别擅长衡量摘要之间的 *n*-gram 共现信息, 通常在生成包含多个句子或段落的长文本摘要时被使用. *ROUGE-L* 通过计算最长公共子序列的重叠度来量化其重合率, 具体如下:

$$R_{\text{lcs}} = \frac{L}{m} \tag{25}$$

$$P_{\text{lcs}} = \frac{L}{n} \tag{26}$$

$$ROUGE\text{-}L = \frac{(1+\beta^2)R_{\text{lcs}}P_{\text{lcs}}}{R_{\text{lcs}}+\beta^2 P_{\text{lcs}}} \tag{27}$$

其中, m 和 n 分别表示预测序列和参考序列的长度, L 表示两序列的最长公共子序列的长度, R_{lcs} 和 P_{lcs} 分别表示召回率和精确率.

ROUGE-L 评估句子中单词顺序的序列对齐情况. 然而, 它的一个局限性在于, 仅计算最长公共子序列, 而忽略了其他子序列可能的匹配, 这使得其评估结果可能不够全面.

值得注意的是, 尽管上述自动评估指标在代码注释生成任务中被广泛使用, 但它们确实存在固有的局限性. 例如, *BLEU* 和类似的指标可能无法可靠地评估一个高 *BLEU* 得分的方法是否一定能比低得分的方法带来更好的系统性能[62]. 这些评估指标基于这样一个假设: 生成摘要与参考摘要之间的文本相似度越高, 质量就越高. 然而, 这一假设忽略了两个关键因素. 首先, 参考摘要的质量可能较差或过时. 其次, 生成的摘要可能通过不同的措辞表达相同的语义. 因此, 单纯依赖基于相似度的评估指标存在局限性[63]. 在未来的工作中, 将计划探索更多专门为代码注释生成任务量身定制的更强大的评估指标.

5.3.2 人工评估指标

除了自动评估外, 本文还进行了额外的人工评估, 主要从以下 5 个方面进行评分: 流畅性、语义准确性、信息完整性、简洁性和风格与语域. 每个评估标准的详细描述如表 2 所示.

表 2 人工评估的 5 个参考指标

指标	描述
流畅性	摘要表达是否自然、流畅且符合语法
语义准确性	摘要是否正确理解了源代码的上下文, 并准确传达其意义
信息完整性	摘要是否包含了代码的所有重要信息
简洁性	摘要是否简明扼要地表达了代码的主要功能
风格与语域	摘要是否考虑了源代码的风格, 并在表达做出了相应的调整

在人工评估过程中, 为确保评估的客观性和准确性, 评估人员在打分时只能看到生成的注释内容, 而无法知道每个注释是由哪种方法生成的, 这样可以避免评估人员受到方法偏差的影响, 确保评估结果的公正性. 每个样本由 3 位评估人员进行评分, 以提高评分的可靠性. 本文随机选取了每种方法生成的 100 个注释, 请每位评估人员根据 1–5 的评分尺度 (1 表示“强烈不同意”, 5 表示“强烈同意”) 对 5 个评估指标进行评分, 每个注释的最终得分是该注释 3 位评估人员评分的平均值. 最后, 通过计算所有 100 个注释的平均得分, 得出每种方法的整体得分.

为了保证评估质量, 邀请的评估人员均是具有丰富编程经验的专家, 在数据统计时, 我们采用了去除极端分值的处理方法, 进一步减少评估人员的主观偏差. 同时, 为了量化评估人员间的一致性, 我们采用了 Conger’s 加权 Kappa 系数, 该系数适用于多名评估员对有序分类数据 (如 1–5 分) 的一致性评估. 我们对每个评估对象在上述 5 个指标上的平均得分进行计算得到的 Kappa 值为 0.672, 根据 Landis 和 Koch 的一致性标准, 该值处于 0.61–0.80 之间, 表明评估人员间具有高度的一致性.

5.4 实验配置

本研究基于 Huggingface 的 CodeT5 PyTorch 实现构建 DHCS 模型. 与单一编码器不同, DHCS 采用了一个分层编码器, 其中子函数编码器和目标函数编码器共享相同的参数. 在预训练阶段, 首先使用基于代码的预训练模型 (如 CodeT5-multi-sum) 初始化 DHCS 的所有编码器和解码器. CodeT5 采用传统的编码器-解码器架构, 而 UniXcoder 采用了一个 Transformer 编码器, 但通过与编码器、解码器配置兼容的掩码注意力矩阵保持灵活性. 为了有效捕获目标函数与其调用的子函数之间的上下文层次关系, 选择分层编码器来进行表示学习. 与 UniXcoder 不同, CodeT5 的编码器架构更适合实现层次结构.

在超参数设置方面, 将源代码和目标序列的最大长度分别设置为 256 和 128. 如果输入序列的长度超过 256, 则需要对其进行截断. 此外, 将学习率设置为 5E–5, 采用层次化学习率衰减, 有助于模型的准确性. 使用 AdamW 优化器[64]来优化参数, 并采用了预热策略. 在推理阶段, 采用了 beam search 进行 token 生成, 并将 beam size 设为 6. DHCS 的实现基于 PyTorch 库, 所有实验均在 Linux 环境下运行, 并部署在两张 GeForce RTX 3090 GPU (24 GB 显存) 上. 值得注意的是, 由于 StarCoder 模型参数量庞大, 因此其训练运行在 4 张 A100 GPU 上, 并利用了 PEFT 加速技术, 以优化性能并提高训练效率.

5.5 Baseline

为了全面评估 DHCS 模型的有效性并确保对比的公平性与代表性, 本研究精心选取了 9 种最先进的代码注释生成方法作为 baseline. 这些 baseline 的选择主要基于以下考虑.

1) 时效性: 选取的模型均为近年来发表在顶级期刊或顶级会议 (ACL 2020–2024 和 ACM 2020–2024) 上的代表性工作, 并在代码理解和生成领域具有广泛的影响力 (如 CodeBERT 引用量>3000+, CodeT5 引用量>1700+, StarCoder 引用量>1000+).

2) 典型性: 选取的模型均是代码理解领域的典型方法, 涵盖了 Decoder-only (StarCoder)、Encoder-Decoder (UniXcoder、CodeT5、CodeT5+、EACS) 等不同架构, 同时包含了传统模型 (NeuralCodeSum) 与 LLM (GPT-3.5-turbo), 保证了技术路线的多样性.

基于以上原则, 本研究最终确定了以下 9 个基线模型进行对比.

● Baseline 1: RoBERTa[65]. 基于 BERT 的掩码语言模型 (MLM) 框架, 能够有效地捕捉来自训练数据的语言知识, 代表了 BERT 模型的一个先进版本.

● Baseline 2: CodeBERT[33]. 其代码预训练阶段采用了两个自监督训练任务, 包括掩码语言建模 (MLM) 和替换 token 检测 (RTD). 该模型是第 1 个大规模、跨多种编程语言和自然语言 (PL-NL) 的预训练模型.

● Baseline 3: UniXcoder[66]. 一个统一的预训练模型, 支持编码器-解码器、仅编码器和仅解码器的配置. 此外, 它明确结合了抽象语法树 (AST) 建模. 为了处理 AST 序列和代码序列之间的差异, UniXcoder 随机去除 50% 的非终结节点, 并利用跨模态内容融合, 结合从 AST 和代码注释中提取的语法和语义信息.

● Baseline 4: CodeT5[37]. 一个统一的预训练编码器-解码器模型, 设计时考虑了 token 类型信息, 并通过开发者指定的标识符传递代码的语义. 该模型假设开发者通常使用具有信息量的标识符来帮助理解代码, 因此这些标识符应保留有价值的代码语义.

● Baseline 5: NeuralCodeSum[1]. 基于 Transformer 模型, 利用自注意力机制来学习代码表示, 通过模拟代码 token 之间的成对关系来捕捉它们的长距离依赖.

● Baseline 6: GPT-3.5-turbo[67]. 由 OpenAI 提出的对话生成模型, 旨在理解上下文并生成连贯的回应. 通过在大规模语料库上的广泛预训练, 它能够学习各种语言知识和上下文. 本研究选择了 2021 年发布的 GPT-3.5-turbo 模型, 利用提示生成相应的注释.

● Baseline 7: EACS[42]. 一个统一的预训练编码器-解码器模型, 包含两个编码器: 一个从重要语句中提取信息, 另一个将整个代码片段转换为嵌入表示, 两个编码器的输出随后传递给解码器以生成预测的注释. 该模型从代码片段中提取重要语句, 以便理解并生成具有高自然性和信息量的注释.

● Baseline 8: CodeT5+[43]. CodeT5 系列的一个变种, 采用灵活的模块组合, 能够适应不同的下游代码任务. 该模型通过多种预训练任务从代码和文本数据中学习丰富的表示, 并弥合预训练和微调之间的差距.

● Baseline 9: StarCoder[13]. 一个仅包含解码器的 Transformer, 采用了多查询注意力 (multi-query-attention) 和学习的绝对位置嵌入. 通过广泛的评估, 该模型超越了其他代码大语言模型 (Code LLM). 此外, 它在对 Python 进行微调后的表现也超过了所有其他模型, 并且仍然保持在其他编程语言上的性能.

在这些 baseline 的基础上, 通过将本研究的模型与这些 baseline 进行比较来回答 RQ1. 此外, 还通过计算每个模型的训练时间和推理时间来回答 RQ4, 为了评估主题感知复制机制和掩蔽子函数预测任务对模型性能的影响,

还对 DHCS 的多个变体进行了比较, 以回答 RQ2 和 RQ3.

● DHCS w/o msp: DHCS 的变体 1. 不使用自监督训练, 即掩蔽子函数预测任务.

● DHCS w/o twc: DHCS 的变体 2. 不使用主题感知复制机制 (topic-aware copy mechanism, TWC), 在解码过程中, 解码器不直接从子函数的主题中提取 token.

● DHCS w/o msp&twc: DHCS 的变体 3. 同时去除了主题感知复制机制和掩蔽子函数预测任务, 因此, 该变体退化为一个分层结构的代码注释生成模型.

● DHCS w/o subfunc: DHCS 的变体 4. 不使用子函数增强 (subfunction enhancement), 因此, 该变体退化为一个类似 CodeT5 的代码注释生成模型.

5.6 实验结果

5.6.1 主要结果

为了回答 RQ1, 将 DHCS 模型与 9 种最先进的 baseline 方法进行比较, 结果如表 3 所示.

表 3 DHCS 与 baseline 模型在 Python、Java 和 Go 数据集上的主要实验结果 (%)

方法	*BLEU*-4			*METEOR*			*ROUGE-L*		
	Python	Java	Go	Python	Java	Go	Python	Java	Go
NeuralCodeSum	10.79	7.81	9.90	10.47	6.68	9.74	17.78	15.44	19.79
RoBERTa	10.37	10.98	13.53	8.11	10.17	13.82	17.10	23.40	30.12
CodeBERT	14.87	12.77	14.89	13.65	11.47	14.44	28.09	25.75	32.60
UniXcoder	17.22	15.61	15.81	15.19	13.48	15.84	30.51	30.18	34.79
CodeT5	17.69	16.22	17.28	17.19	14.93	17.04	34.03	32.00	36.75
GPT-3.5-turbo	15.56	13.15	12.51	16.87	15.84	16.82	27.17	22.84	22.89
CodeT5+	17.76	15.07	12.29	17.35	15.01	12.49	22.80	22.19	19.17
StarCoder	16.90	17.25	14.75	17.27	17.53	14.43	17.29	22.97	17.92
EACS	18.10	16.38	17.28	17.09	16.50	17.36	34.70	35.54	37.20
DHCS	18.62	18.59	18.05	17.63	16.60	17.92	34.99	35.69	37.42

从表 3 中可以看出, 在所有 baseline 中, NeuralCodeSum 表现最差, 因为本研究的模型与上述 baseline 一样, 依赖于预训练的编程语言模型. 利用这样的模型能够充分利用编程语言中固有的有价值的语义知识, 从而实现更优的性能. 与本文方案以及大多数 baseline 模型 (如 CodeBERT、UniXcoder、CodeT5 和 CodeT5+) 相比, RoBERTa 的表现较差. 这一差异可以归因于 RoBERTa 是一个在自然语言语料库上预训练的语言模型, 而其他 baseline 模型则是在大规模编程语言语料库上预训练的. 还可以看到, EACS 的表现优于其他 baseline 方法, 而 DHCS 在 3 个数据集上都取得了 *BLEU*-4、*METEOR* 和 *ROUGE-L* 分数上的最佳表现. 与其他方法相比, GPT-3.5-turbo 的表现较差, 甚至不如 CodeT5 和 UniXcoder. StarCoder 的表现稍好于 GPT-3.5-turbo, 但仍然逊色于本文方法. 这一差异大致归因于两个因素. 首先, GPT-3.5-turbo 采用的是无监督方法进行代码注释生成, 因此缺乏针对特定数据集的训练. 其次, GPT-3.5-turbo 和 StarCoder 在理解代码注释生成任务中的函数调用时遇到了困难. 与 EACS 相比, DHCS 在 Python 数据集上, 分别在 *BLEU*-4、*METEOR* 和 *ROUGE-L* 上取得了 0.52、0.36 和 0.29 的提升; 在 Java 数据集上, DHCS 在所有 3 个指标上比 CodeT5 分别提高了 2.21、0.1 和 0.15; 在 Go 数据集上, 得分分别提高了 0.77、0.56 和 0.22. 值得注意的是, 我们通过对 CodeT5 和 DHCS 之间的成对样本 Wilcoxon 符号秩检验计算得出 p 值. 在 Python 数据集上, *BLEU*-4 指标的 p 值低于 0.04, *METEOR* 指标的 p 值低于 0.01; 在 Java 数据集上, p 值均低于 0.01; 在 Go 数据集上 p 值约为 0.02. 这表明这些改进在统计学上具有显著性, 这些提升证明了 DHCS 的有效性. 表明目标函数中的子函数调用能够补充额外的信息, 从而提高代码注释的性能.

5.6.2 人工评估

本研究在 3 个数据集上进行了人类评估, 结果如图 4、图 5 所示. 图中蓝线为 Python 数据集上的评估结果; 橙线为 Java 数据集上的评估结果; 绿线为 Go 数据集上的评估结果. 参与此次评估的所有人员都具有丰富的领域知识, 至少拥有 5 年的软件工程专业背景和研究经验.

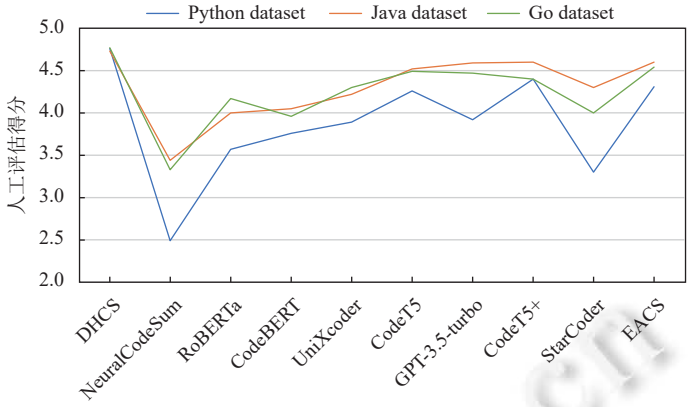
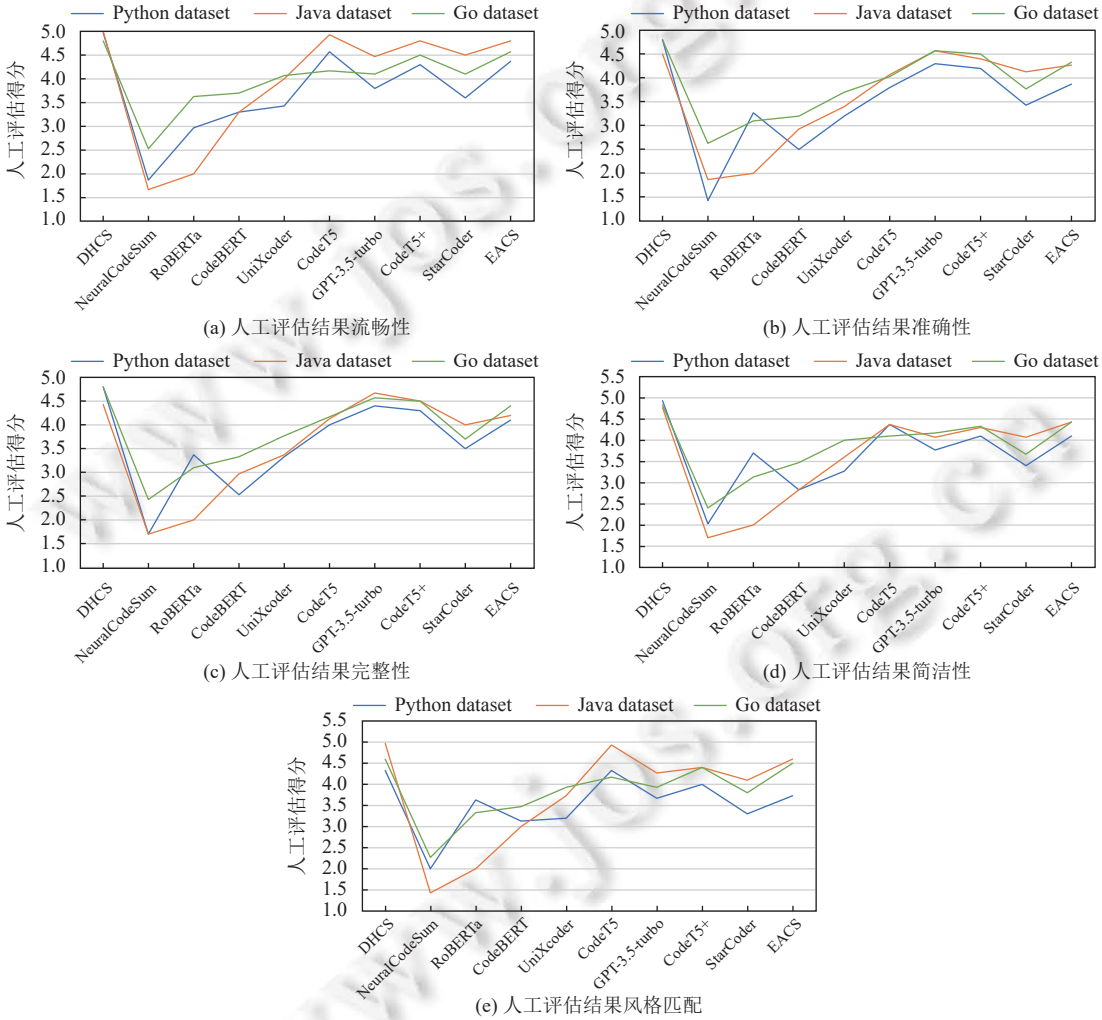


图 4　3 种数据集上的人工评估结果



(a) 人工评估结果流畅性　(b) 人工评估结果准确性

(c) 人工评估结果完整性　(d) 人工评估结果简洁性

(e) 人工评估结果风格匹配

图 5　3 种数据集上 5 种指标的人工评估结果

从图 5 中可以观察到, 基于大型语言模型的方法表现不如预期, 与 CodeT5、CodeT5+和 EACS 相比稍有不足, 这是因为生成的注释与参考的注释之间存在不一致的问题. 相比之下, 本文方法在人工评估中获得了最高的分数,

在准确性、完整性、简洁性和整体注释质量方面均表现突出. 子函数调用在实际代码中经常出现, 捕捉函数调用中代码结构的语义和关系可以显著提高生成注释的质量.

5.6.3 消融实验

为了回答 RQ2 和 RQ3, 本研究还进行了消融实验, 以评估 DHCS 中两个关键组件的影响, 即主题感知复制机制和掩码子函数预测任务. 实验结果如表 4 所示.

表 4 消融实验结果 (%)

不同变体方法	*BLEU*-4			*METEOR*			*ROUGE-L*		
	Python	Java	Go	Python	Java	Go	Python	Java	Go
DHCS	18.62	18.59	18.05	17.63	16.60	17.92	34.99	35.69	37.42
DHCS w/o msp	18.25	18.34	17.84	17.60	16.59	17.73	34.83	35.02	37.02
DHCS w/o twc	18.40	18.37	17.89	17.60	16.54	17.69	34.84	35.63	37.01
DHCS w/o msp&twc	18.13	18.08	17.62	17.37	16.42	17.55	34.14	34.84	36.79
DHCS w/o subfunc	17.69	16.22	17.28	17.19	14.93	17.04	34.03	32.00	36.75

从表 4 中可以看到, 在去除各个模块后, DHCS 的性能均有所下降. 特别是去除掩码子函数预测任务 (msp) 和主题感知复制机制 (twc) 后, 性能下降最为明显, 这验证了本文的自监督训练目标在分层编码器中的重要性和有效性. 此外, 还发现, 去除子函数增强 (subfunc) 时, DHCS 仍然略优于没有使用 msp 和 twc 的模型, 表明子函数的调用确实对代码注释的生成有增益作用.

5.6.4 训练/推理时间

为了回答 RQ4, 首先报告了本文方法和所有 baseline 模型的参数规模, 结果如表 5 所示.

表 5 各模型的参数规模统计 (M)

方法	参数量
NeuralCodeSum	86
RoBERTa	173
CodeBERT	173
UniXcoder	127
CodeT5	223
GPT-3.5-turbo	—
CodeT5+	223
StarCoder	15 517
EACS	223

从表 5 中可以看出, NeuralCodeSum 的参数量较小, 表现也较差. 其他 baseline 模型的参数量相似, 平衡了性能和计算效率. 值得注意的是, StarCoder 的参数量显著较大, 达到了 15 517M, 这使得它的计算资源需求较高, 训练时间更长, 推理速度较慢. 相比之下, 本文模型虽然较小, 但在计算资源上更高效, 能在保持竞争力的同时提供更快的部署速度和更高的训练效率.

后文图 6 展示了各个模型在 3 个数据集上的训练时间和推理时间. 横轴表示各模型, 纵轴表示每个 epoch 的训练或推理步骤的时间 (以 min 为单位). 可以观察到, StarCoder 的训练和推理时间最长, 大约是其他方法的 10 倍. 本文模型尽管引入了分层结构和新的解码机制, 但性能提升的同时, 计算复杂度的增加非常小.

5.7 讨 论

5.7.1 依赖关系对基准模型的影响

本文方法采用了分层结构来捕捉上下文依赖关系, 从而提高了注释的质量. 与此不同, 其他 baseline 模型未能做到这一点. 为了评估函数依赖关系对这些模型的影响, 通过将子函数信息直接集成到 baseline 模型的输入中, 进行了一次对比实验. 在训练和测试过程中, 这些 baseline 模型的输入包括了子函数信息, 结果如表 6 所示. 其中, “_wsf”表示包含子函数的输入.

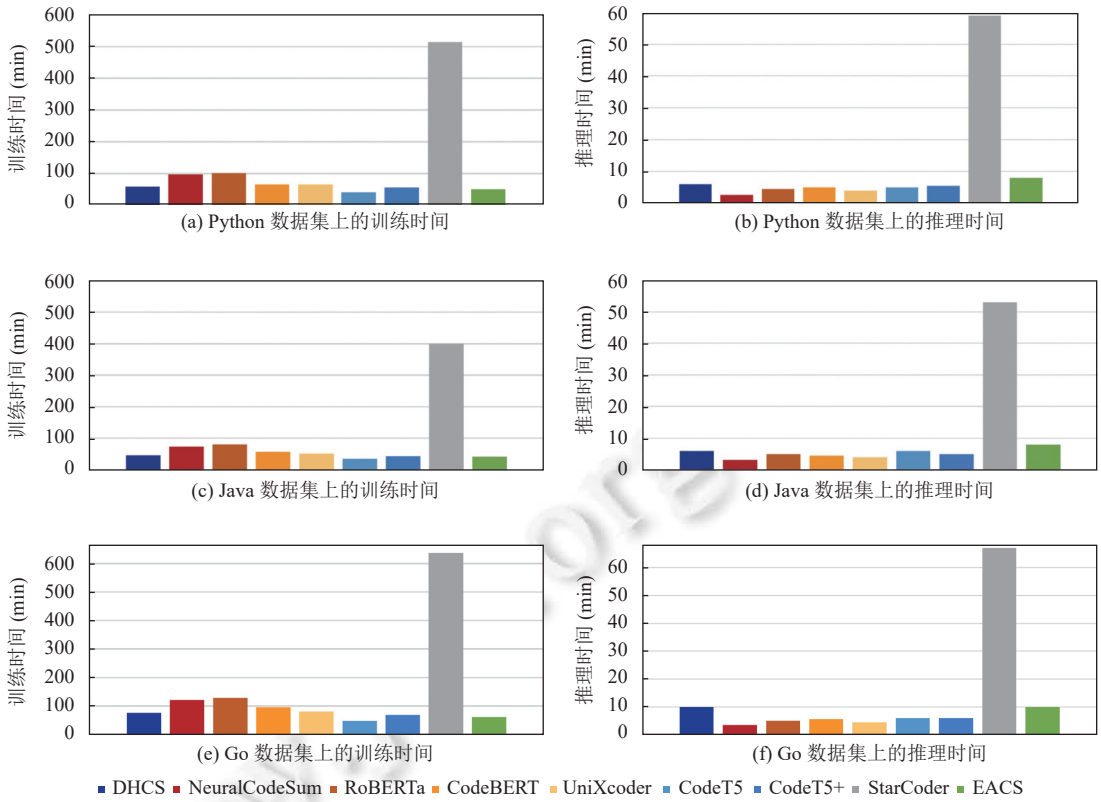


图 6　3 个数据集上的训练时间 (每个 epoch) 和推理时间

如表 6 所示, 当将子函数信息作为额外输入时, 大多数 baseline 模型在 3 个指标上的得分略有下降. 这些方法仅将子函数信息视为目标函数的附加输入, 并未有效地捕捉子函数和目标函数之间的依赖关系. 虽然 StarCoder 的表现略有改善, 但整体表现并未显著提升, 仍然不如本文方法. 这表明, 虽然大语言模型确实能够捕捉到更长的上下文, 但它们并未完全理解子函数和目标函数之间的依赖关系. 这也验证了本文方法的有效性, 能够有效捕捉代码中的上下文依赖关系.

表 6　含子函数信息的输入对 DHCS 和 baseline 模型的影响 (%)

方法	*BLEU-4*			*METEOR*			*ROUGE-L*		
	Python	Java	Go	Python	Java	Go	Python	Java	Go
NeuralCodeSum	10.79	7.81	9.90	10.47	6.68	9.74	17.78	15.44	19.79
NeuralCodeSum_wsf	10.51	7.27	8.72	9.68	6.49	8.67	17.56	12.20	17.77
RoBERTa	10.37	10.98	13.53	8.11	10.17	13.82	17.10	23.40	30.12
RoBERTa_wsf	8.25	10.15	7.67	6.61	7.82	6.48	15.60	16.88	14.37
CodeBERT	14.87	12.77	14.89	13.65	11.47	14.44	28.09	25.75	32.60
CodeBERT_wsf	14.36	12.45	13.01	13.53	11.37	11.76	26.96	25.06	26.50
UniXcoder	17.22	15.61	15.81	15.19	13.48	15.84	30.51	30.18	34.79
UniXcoder_wsf	16.80	14.93	15.66	14.96	13.26	15.52	29.45	28.94	33.48
CodeT5	17.69	16.22	17.28	17.19	14.93	17.04	34.03	32.00	36.75
CodeT5_wsf	17.30	16.15	17.16	17.05	14.16	16.27	33.62	31.30	36.22
GPT-3.5-turbo	15.56	13.15	12.51	16.87	15.84	16.82	27.17	22.84	22.89
GPT-3.5-turbo_wsf	13.87	13.41	12.18	16.20	16.68	16.37	22.52	23.64	22.26
CodeT5+	17.76	15.07	12.29	17.35	15.01	12.49	22.80	22.19	19.17
CodeT5+_wsf	17.79	14.94	15.58	18.19	15.00	17.51	22.98	22.28	30.87

表 6 含子函数信息的输入对 DHCS 和 baseline 模型的影响 (%)(续)

方法	*BLEU*-4			*METEOR*			*ROUGE-L*		
	Python	Java	Go	Python	Java	Go	Python	Java	Go
StarCoder	16.90	17.25	14.75	17.27	17.53	14.43	17.29	22.97	17.92
StarCoder_wsf	16.83	17.45	14.62	17.07	17.44	14.34	16.97	18.43	17.65
EACS	18.10	16.38	17.28	17.09	16.50	17.36	34.70	35.54	37.20
EACS_wsf	17.89	16.29	17.27	16.92	15.04	17.12	33.40	32.27	37.00
DHCS	18.62	18.59	18.05	17.63	16.60	17.92	34.99	35.69	37.42

5.7.2 函数名称对任务的影响

在编写代码时, 程序员通常会为函数命名以增强对代码功能的理解. 因此, 函数名称本身可能部分地反映了函数的目的. 例如, 名为“searchAccession”的函数意味着其功能可能与搜索 Accession 有关, 并可能涉及其他操作, 按照驼峰命名法的惯例. 然而, 这种做法可能会使代码注释生成模型过分关注函数的名称, 而忽略代码的实际逻辑和功能. 因此, 本文进行实证研究, 以了解函数名称对代码注释性能的影响.

本文采用了两种策略来操控目标函数中的函数名称: 掩码 (masking) 和随机替换 (random replacement). 在掩码策略中, 使用特殊 token “<mask>”来隐藏函数名称; 而在随机替换策略中, 从数据集中的其他目标函数中随机选择一个函数名称并将其替换为原始函数名称. 这两种策略帮助本研究代码注释生成模型在处理函数名称变化时的鲁棒性.

从表 7 中可以看出, 所有模型在函数名称进行掩码或替换时都受到了影响. 在 baseline 模型中, CodeT5 对函数名称变化的鲁棒性较强, 这得益于其标识符感知的预训练机制. 与此类似, 本文方法也展现了较好的表现.

表 7 函数名称对代码注释生成任务的影响 (%)

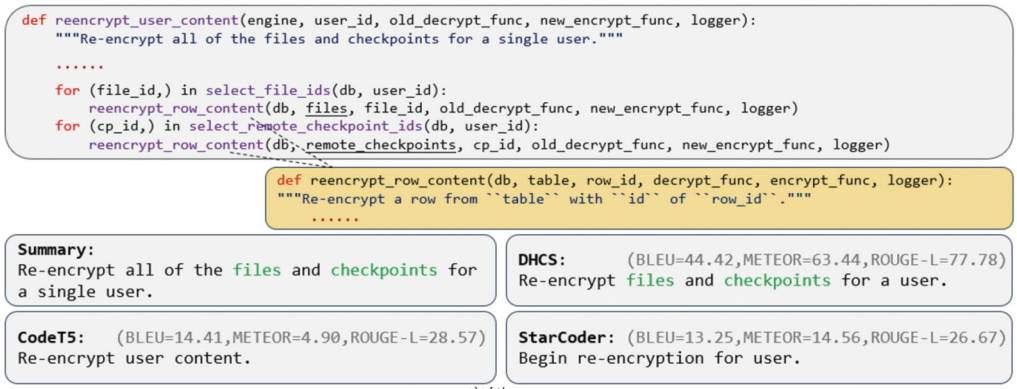
不同策略下的方法	*BLEU*-4		*METEOR*		*ROUGE-L*	
	Python	Java	Python	Java	Python	Java
RoBERTa	10.37	10.98	8.11	10.17	17.10	23.40
RoBERTa (mfn)	10.37	8.97	8.11	6.83	17.10	17.11
RoBERTa (rfn)	10.36	9.03	8.11	7.71	17.09	18.74
CodeBERT	14.87	12.77	13.65	11.47	28.09	25.75
CodeBERT (mfn)	13.41	11.45	12.24	10.27	24.84	23.44
CodeBERT (rfn)	11.79	11.51	9.62	10.94	21.06	24.33
UniXcoder	17.22	15.61	15.19	13.48	30.51	30.18
UniXcoder (mfn)	14.50	13.69	12.98	12.29	25.47	27.63
UniXcoder (rfn)	13.95	14.30	12.77	12.37	24.96	28.14
CodeT5	17.69	16.22	17.19	14.93	34.03	32.00
CodeT5 (mfn)	15.67	14.54	15.26	13.74	30.04	29.85
CodeT5 (rfn)	15.84	14.46	15.33	13.70	30.28	29.77
DHCS	18.62	18.59	17.63	16.60	34.99	35.69
DHCS (mfn)	16.36	16.39	15.65	15.36	30.63	32.73
DHCS (rfn)	16.37	16.35	15.77	15.33	30.64	32.62

注: “mfn”表示掩码策略, “rfn”表示随机替换策略

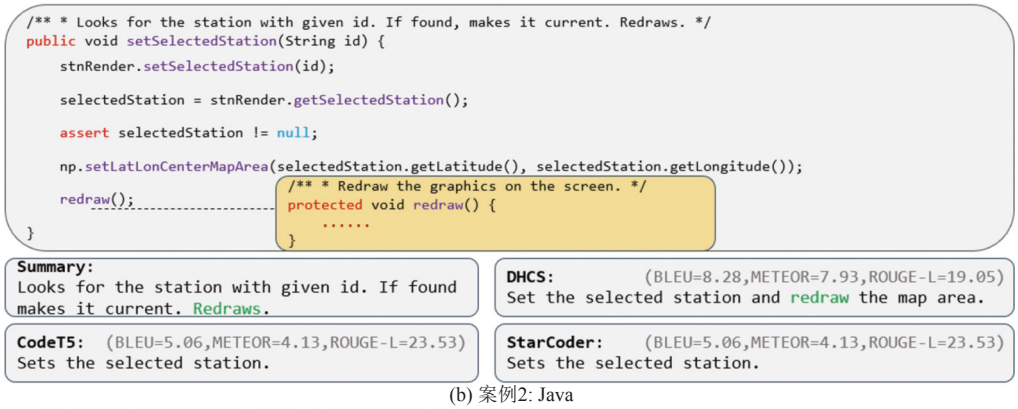
5.8 案例研究

为了进一步突出 DHCS 的有效性, 如图 7 所示, 本文提供了 4 个代码注释的示例案例. 案例 1 关于 Python 语言, 案例 2 和案例 3 关于 Java 语言, 案例 4 关于 Go 语言. 值得注意的是, 案例 1、2 和案例 4 包含了一个子函数的调用, 而案例 3 包含了 3 个子函数的调用. 在 *BLEU*-4、*METEOR* 和 *ROUGE-L* 这 3 个关键指标上, DHCS 表现出显著优于 baseline 模型 CodeT5 和 StarCoder 的优势.

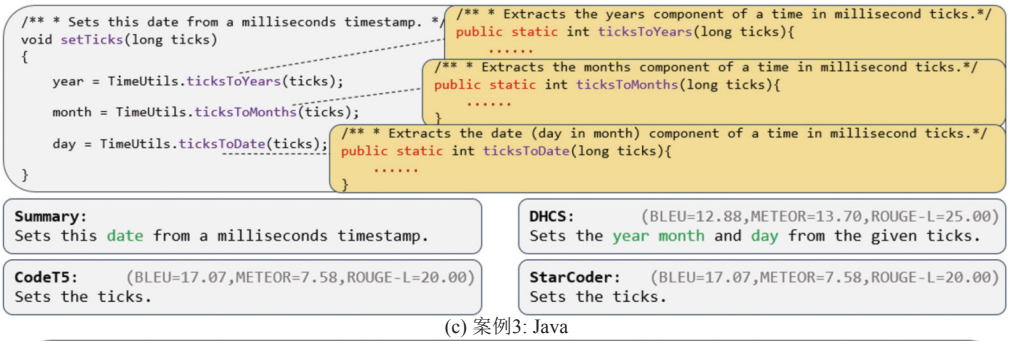
通常, 子函数是目标函数功能的主要驱动力. 例如, 在图 7 中的案例 1 中, 目标函数“reencrypt_user_content”调用了子函数“reencrypt_row_content”. 通过引入子函数, 可以看到关键信息位于第 2 个参数中. 因此, 可以推断出, 两个 for 循环使得子函数可以分别对文件和检查点进行重新加密. 因此, 生成的注释更加准确, 更好地反映了目标函数的目的. 相比之下, 由于没有包含子函数信息, 两个 baseline 模型生成的注释更为笼统.



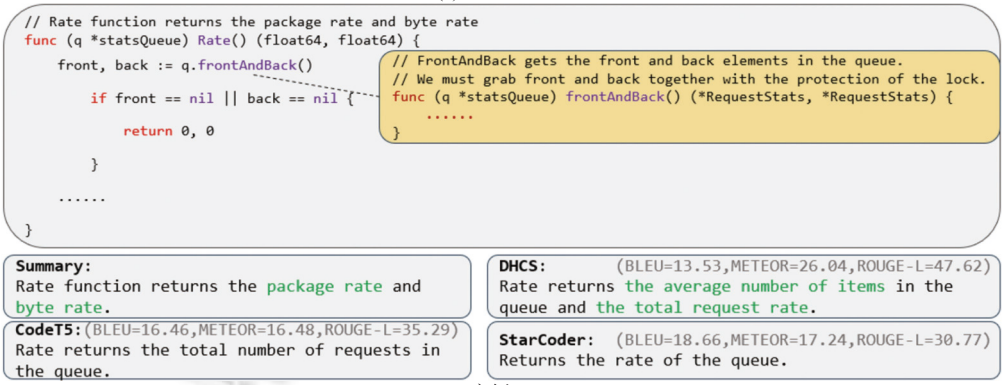
(a) 案例1: Python



(b) 案例2: Java



(c) 案例3: Java



(d) 案例4: Go

图 7 案例研究

在案例 2 中, 子函数提供了重绘操作的关键信息, 而两个 baseline 模型都忽略了这一点. 案例 3 中, 目标函数包含了 3 个子函数, 它们分别从时间戳中提取年、月和日期信息. 如果没有考虑这些子函数, 很难进行正确的分析并生成准确的注释, 如两个 baseline 模型所示. 案例 4 中, 子函数提供了返回队列中两个元素这一关键信息, 与参考注释中返回 package rate 和 byte rate 相契合. DHCS 生成的注释符合参考注释的并列结构, 与代码中实际返回两个速率值的逻辑一致, 这种结构相似性直接提升了 *ROUGE-L* 的分数. 虽然 DHCS 的用词不完全准确, 但保留了“rate”这一核心术语, 与参考注释高度相关. 而其他两种方法生成的注释则只关注到一个元素, 并且核心内容也不贴切. 这进一步证明了本文模型通过整合子函数调用来增强代码注释性能.

6 总 结

本文提出了一种依赖感知的分层代码注释生成模型, 旨在通过建模目标函数与其调用的函数依赖之间的层次结构, 增强代码注释生成任务的效果. 具体来说, 本文首先设计了一个分层编码器, 由子函数编码器和目标函数编码器组成, 用于捕捉局部和上下文的语义表示. 此外, 引入了一种自监督目标——掩码子函数预测, 以增强子函数的表示学习. 进一步考虑到子函数的关键作用, 还挖掘了被调用子函数的主题分布, 并设计了一个具有主题感知复制机制的解码器, 直接提取子函数中的关键信息, 促进目标函数的注释生成. 最后, 为本研究构建了 3 个真实世界数据集, 并通过广泛的实验验证了所提出方法的有效性. 本文的工作不仅推进了代码注释生成领域的发展, 还解决了关于代码依赖关系处理的问题, 为未来更全面、准确的代码注释生成技术奠定了基础.

References

[1] Ahmad W, Chakraborty S, Ray B, Chang KW. A Transformer-based approach for source code summarization. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. ACL, 2020. 4998–5007. [doi: 10.18653/v1/2020.acl-main.449]

[2] Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. In: Proc. of the 26th Conf. on Program Comprehension. Gothenburg: ACM, 2018. 200–210. [doi: 10.1145/3196321.3196334]

[3] Wan Y, Zhao Z, Yang M, Xu GD, Ying HC, Wu J, Yu PS. Improving automatic source code summarization via deep reinforcement learning. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 397–407. [doi: 10.1145/3238147.3238206]

[4] Wei BL, Li G, Xia X, Fu ZY, Jin Z. Code generation as a dual task of code summarization. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 589.

[5] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: Proc. of the 3rd Int'l Conf. on Learning Representations. San Diego: OpenReview.net, 2015. [doi: 10.48550/arXiv.1409.0473]

[6] Clement C, Drain D, Timcheck J, Svyatkovskiy A, Sundaresan N. PyMT5: Multi-mode translation of natural language and Python code with Transformers. In: Proc. of the 2020 Conf. on Empirical Methods in Natural Language Processing (EMNLP). ACL, 2020. 9052–9065. [doi: 10.18653/v1/2020.emnlp-main.728]

[7] LeClair A, Haque S, Wu LF, McMillan C. Improved code summarization via a graph neural network. In: Proc. of the 28th Int'l Conf. on Program Comprehension. Seoul: ACM, 2020. 184–195. [doi: 10.1145/3387904.3389268]

[8] LeClair A, Jiang SY, McMillan C. A neural model for generating natural language summaries of program subroutines. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 795–806. [doi: 10.1109/ICSE.2019.00087]

[9] Shi ES, Wang YL, Du L, Zhang HY, Han S, Zhang DM, Sun HB. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 4053–4062. [doi: 10.18653/v1/2021.emnlp-main.332]

[10] Wu HQ, Zhao H, Zhang M. Code summarization with structure-induced Transformer. In: Proc. of the 2021 Findings of the Association for Computational Linguistics (ACL-IJCNLP 2021). ACL, 2021. 1078–1090. [doi: 10.18653/v1/2021.findings-acl.93]

[11] Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing source code with transferred API knowledge. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. Stockholm: ijcai.org, 2018. 2269–2275. [doi: 10.24963/ijcai.2018/314]

[12] Rozière B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu JY, Remez T, Rapin J, Kozhevnikov A, Evtimov I, Bitton J, Bhatt M, Ferrer CC, Grattafiori A, Xiong WH, Défossez A, Copet J, Azhar F, Touvron H, Martin L, Usunier N, Scialom T, Synnaeve G. Code LLaMA: Open foundation models for code. arXiv:2308.12950, 2023.

[13] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! arXiv:2305.06161, 2023.

[14] Mihalcea R, Tarau P. TextRank: Bringing order into text. In: Proc. of the 2004 Conf. on Empirical Methods in Natural Language Processing. Barcelona: ACL, 2004. 404–411.

[15] Erkan G, Radev DR. LexPageRank: Prestige in multi-document text summarization. In: Proc. of the 2004 Conf. on Empirical Methods in Natural Language Processing. Barcelona: ACL, 2004. 365–371.

[16] Fang CJ, Mu DJ, Deng ZH, Wu ZA. Word-sentence co-ranking for automatic extractive text summarization. Expert Systems with Applications, 2017, 72: 189–195. [doi: 10.1016/j.eswa.2016.12.021]

[17] Cao ZQ, Li WJ, Li SJ, Wei FR. Improving multi-document summarization via text classification. In: Proc. of the 31st AAAI Conf. on Artificial Intelligence. San Francisco: AAAI, 2017. 3053–3059. [doi: 10.1609/aaai.v31i1.10955]

[18] Singh A, Gupta M, Varma V. Unity in diversity: Learning distributed heterogeneous sentence representation for extractive summarization. In: Proc. of the 32nd AAAI Conf. on Artificial Intelligence. New Orleans: AAAI, 2018. 5473–5480. [doi: 10.1609/aaai.v32i1.11994]

[19] Nallapati R, Zhai FF, Zhou BW. SummaRuNNer: A recurrent neural network based sequence model for extractive summarization of documents. In: Proc. of the 31st AAAI Conf. on Artificial Intelligence. San Francisco: AAAI, 2017. 3075–3081. [doi: 10.1609/aaai.v31i1.10958]

[20] Gu JT, Lu ZD, Li H, Li VOK. Incorporating copying mechanism in sequence-to-sequence learning. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Berlin: ACL, 2016. 1631–1640. [doi: 10.18653/v1/P16-1154]

[21] See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Vancouver: ACL, 2017. 1073–1083. [doi: 10.18653/v1/P17-1099]

[22] Zhou QY, Yang N, Wei FR, Zhou M. Selective encoding for abstractive sentence summarization. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Vancouver: ACL, 2017. 1095–1104. [doi: 10.18653/v1/P17-1101]

[23] Song KT, Tan X, Qin T, Lu JF, Liu TY. MASS: Masked sequence to sequence pre-training for language generation. In: Proc. of the 36th Int'l Conf. on Machine Learning. Long Beach: PMLR, 2019. 5926–5936.

[24] Lewis M, Liu YH, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. ACL, 2019. 7871–7880. [doi: 10.18653/v1/2020.acl-main.703]

[25] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text Transformer. The Journal of Machine Learning Research, 2020, 21(1): 140.

[26] Zhang HY, Xu JJ, Wang J. Pretraining-based natural language generation for text summarization. In: Proc. of the 23rd Conf. on Computational Natural Language Learning (CoNLL). Hong Kong: ACL, 2019. 789–797. [doi: 10.18653/v1/K19-1074]

[27] Cao SY, Wang L. Attention head masking for inference time content selection in abstractive summarization. In: Proc. of the 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. ACL, 2021. 5008–5016. [doi: 10.18653/v1/2021.naacl-main.397]

[28] Iyer S, Konstas I, Cheung A, Zettlemoyer L. Summarizing source code using a neural attention model. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. Berlin: ACL, 2016. 2073–2083. [doi: 10.18653/v1/P16-1195]

[29] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proc. of the 33rd Int'l Conf. on Machine Learning. New York City: PMLR, 2016. 2091–2100.

[30] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence. Phoenix: AAAI, 2016. 1287–1293. [doi: 10.1609/aaai.v30i1.10139]

[31] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional Transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Vol. 1: Long and Short Papers). Minneapolis: ACL, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

[32] Kanade A, Maniatis P, Balakrishnan G, Shi KS. Pre-trained contextual embedding of source code. arXiv:2001.00059, 2019.

[33] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the 2020 Findings of the Association for Computational Linguistics: EMNLP 2020. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[34] Guo DY, Ren S, Lu S, Feng ZY, Tang DY, Liu SJ, Zhou L, Duan N, Svyatkovskiy A, Fu SY, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang DX, Zhou M. GraphCodeBERT: Pre-training code representations with data flow. In: Proc. of the 9th Int'l Conf. on Learning Representations. OpenReview.net, 2021.

[35] Fried D, Aghajanyan A, Lin J, Wang SD, Wallace E, Shi F, Zhong RQ, Yih WT, Zettlemoyer L, Lewis M. InCoder: A generative model

for code infilling and synthesis. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[36] Ahmad W, Chakraborty S, Ray B, Chang KW. Unified pre-training for program understanding and generation. In: Proc. of the 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. ACL, 2021. 2655–2668. [doi: 10.18653/v1/2021.naacl-main.211]

[37] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[38] Gao SZ, Gao CY, He YL, Zeng JC, Nie L, Xia X, Lyu M. Code structure guided Transformer for source code summarization. ACM Trans. on Software Engineering and Methodology, 2023, 32(1): 23. [doi: 10.1145/3522674]

[39] Nagaraj Y, Gupta U. AST-MHSA: Code summarization using multi-head self-attention. arXiv:2308.05646, 2023.

[40] Shahbazi R, Fard F. APIContext2Com: Code comment generation by incorporating pre-defined API documentation. In: 2023 IEEE/ACM 31st Int'l Conf. on Program Comprehension. Melbourne: IEEE, 2023. 13–24. [doi: 10.1109/ICPC58990.2023.00012]

[41] Li MC, Yu HQ, Fan GS, Zhou ZY, Huang ZJ. Enhancing code summarization with action word prediction. Neurocomputing, 2024, 563: 126777. [doi: 10.1016/j.neucom.2023.126777]

[42] Sun WS, Fang CR, Chen YC, Zhang QJ, Tao GH, You YD, Han TX, Ge YF, Hu YL, Luo B, Chen ZY. An extractive-and-abstractive framework for source code summarization. ACM Trans. on Software Engineering and Methodology, 2024, 33(3): 75. [doi: 10.1145/3632742]

[43] Wang Y, Le H, Gotmare A, Bui N, Li JN, Hoi S. CodeT5+: Open code large language models for code understanding and generation. In: Proc. of the 2023 Conf. on Empirical Methods in Natural Language Processing. Singapore: ACL, 2023. 1069–1088. [doi: 10.18653/v1/2023.emnlp-main.68]

[44] Guo DY, Zhu QH, Yang DJ, Xie ZD, Dong K, Zhang WT, Chen GT, Bi X, Wu U, Li YK, Luo FL, Xiong YF, Liang WF. DeepSeek-coder: When the large language model meets programming—The rise of code intelligence. arXiv:2401.14196, 2024.

[45] Sun Y, Wang SH, Li YK, Feng SK, Chen XY, Zhang H, Tian X, Zhu DX, Tian H, Wu H. ERNIE: Enhanced representation through knowledge integration. arXiv:1904.09223, 2019.

[46] Clark K, Luong MT, Le QV, Manning CD. ELECTRA: Pre-training text encoders as discriminators rather than generators. arXiv:2003.10555, 2020

[47] Li ZC, Li SS, Zhou GD. Pre-trained token-replaced detection model as few-shot learner. In: Proc. of the 29th Int'l Conf. on Computational Linguistics. Gyeongju: ACL, 2022. 3274–3284.

[48] Antoun W, Baly F, Hajj H. AraELECTRA: Pre-training text discriminators for Arabic language understanding. In: Proc. of the 6th Arabic Natural Language Processing Workshop. Kyiv: ACL, 2021. 191–195.

[49] He PC, Gao JF, Chen WZ. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReivew.net, 2023.

[50] Lachaux MA, Roziere B, Szafraniec M, Lample G. DOBF: A deobfuscation pre-training objective for programming languages. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 1147.

[51] Jain P, Jain A, Zhang TJ, Abbeel P, Gonzalez J, Stoica I. Contrastive code representation learning. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 5954–5971. [doi: 10.18653/v1/2021.emnlp-main.482]

[52] Ding YRB, Buratti L, Pujar S, Morari A, Ray B, Chakraborty S. Contrastive learning for source code with structural and functional properties. arXiv:2110.03868, 2021.

[53] Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Berlin: ACL, 2016. 1715–1725. [doi: 10.18653/v1/P16-1162]

[54] Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I. Language models are unsupervised multitask learners. OpenAI Blog, 2019, 1(8): 9.

[55] Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proc. of the 29th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 2692–2700.

[56] Barone AVM, Sennrich R. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. In: Proc. of the 8th Int'l Joint Conf. on Natural Language Processing (Vol. 2: Short Papers). Taipei: ACL, 2017. 314–319.

[57] Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436, 2019.

[58] Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia: ACL, 2002. 311–318. [doi: 10.3115/1073083.1073135]

[59] Banerjee S, Lavie A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proc. of the 2005 ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. Ann Arbor: ACL, 2005. 65–72.

[60] Lin CY. ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. Barcelona: ACL, 2004. 74–81.

[61] Lin CY, Och FJ. ORANGE: A method for evaluating automatic evaluation metrics for machine translation. In: Proc. of the 20th Int'l Conf. on Computational Linguistics. Geneva: ACL, 2004. 501–507.

[62] Roy D, Fakhoury S, Arnaoudova V. Reassessing automatic evaluation metrics for code summarization tasks. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 1105–1116. [doi: 10.1145/3468264.3468588]

[63] Mastropaolo A, Ciniselli M, Di Penta M, Bavota G. Evaluating code summarization techniques: A new metric and an empirical characterization. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 218. [doi: 10.1145/3597503.3639174]

[64] Loshchilov I, Hutter F. Decoupled weight decay regularization. In: Proc. of the 7th Int'l Conf. on Learning Representations. New Orleans: OpenReview.net, 2019.

[65] Liu YH, Ott M, Goyal N, Du JF, Joshi M, Chen DQ, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692, 2019.

[66] Guo DY, Lu S, Duan N, Wang YL, Zhou M, Yin J. UniXcoder: Unified cross-modal pre-training for code representation. In: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Dublin: ACL, 2022. 7212–7225. [doi: 10.18653/v1/2022.acl-long.499]

[67] OpenAI. ChatGPT. 2023. https://chat-gpt.org/

作者简介

张育博, 博士, 主要研究领域为自然语言处理, 计算机视觉, 神经机器翻译, 文本生成.

姚开春, 博士, 助理研究员, CCF 专业会员, 主要研究领域为自然语言处理, 大语言模型及其应用.

张立波, 博士, 副研究员, CCF 专业会员, 主要研究领域为图像处理, 模式识别.

武延军, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 开源软件供应链, RISC-V 基础软件.

赵琛, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为编译技术, 自动测试, 操作系统, 网络软件.