计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330730
61(2): 307−323, 2024

基于分层表示和上下文增强的类摘要生成技术

陈豪伶　虞慧群　范贵生　李明辰　黄子杰
(华东理工大学计算机科学与工程系　上海　200237)
(hlchen@mail.ecust.edu.cn)

Class Summarization Generation Technology Based on Hierarchical Representation and Context Enhancement

Chen Haoling, Yu Huiqun, Fan Guisheng, Li Mingchen, and Huang Zijie
(*Department of Computer Science and Engineering*, *East China University of Science and Technology*, *Shanghai* 200237)

Abstract　Code summarization is a natural language description of source code, and high-quality code summaries help to improve developers' program understanding efficiency. In recent years, research on code summarization has focused on generating summaries for method-grained code snippet. However, in an object-oriented language such as Java, class is the basic programming unit. Due to the above problems, we propose a class summarization generation method based on hierarchical representation and context enhancement, called HRCE, as well as constructs a class summarization dataset containing 358 992 pairs of <Java class, content, summary>. HRCE uses code simplification strategy to remove non-critical code of class to shorten the code length. Then, HRCE models the class hierarchy, including class signature, attribute and method respectively, to obtain the semantic information and hierarchical structure information of the class. In addition, HRCE selects parent's class signature and class summary to describe the context that the class depends on in the project. Experiments show that a generative model for class summarization based on hierarchical representation and context enhancement is able to characterize the semantics and hierarchical structure of the code, and obtain information from both inside and outside of the target class. As a result, HRCE outperforms all baseline models on evaluation metrics such as BLEU, METEOR, ROUGE-L, etc.

Key words　automatic code summarization; hierarchical representation; context enhancement; deep learning; class summarization

摘　要　代码摘要是源代码的自然语言解释,高质量的代码摘要有助于提高开发人员程序理解效率.近年来,代码自动摘要的研究集中在为方法粒度的代码片段生成摘要.然而,对于面向对象的语言,例如Java,类才是项目的基本组成单元.基于上述问题,提出一种基于分层表示和上下文增强的类摘要生成方法HRCE(hierarchical representation and context enhancement),并构建了一个包含358 992个〈Java类,上下文,摘要〉数据对的类摘要数据集.HRCE使用代码精简策略去除类的非关键代码,从而缩短代码长度.然后,对类的层次结构,包括类签名、属性和方法分别进行建模,获得类的语义信息和层次结构信息.此外,从项目中抽取父类的签名及摘要来刻画类在项目中依赖的上下文.实验表明,基于分层表示和上下文增强的

收稿日期: 2023-09-11; **修回日期**: 2023-12-05
基金项目: 国家自然科学基金项目(62372174, 62276097);大数据流通与交易技术国家工程实验室课题研究计划、上海市促进高质量发展专项资金项目(2021-GYHLW-01007)
This work was supported by the National Natural Science Foundation of China (62372174, 62276097) and the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies, the Shanghai Municipal Special Fund Program for Promoting High Quality Development (2021-GYHLW-01007).
通信作者: 虞慧群(yhq@ecust.edu.cn)

类摘要生成模型能够表征代码的语义和层次结构,并可以从目标类的内部和外部获取信息. HRCE 在 BLEU, METEOR, ROUGE-L 等评估指标上超过了所有基准模型.

关键词 代码自动摘要;分层表示;上下文增强;深度学习;类摘要

中图法分类号 TP311.5

研究表明,在软件的开发和维护的过程中,软件工程师需要花费 59% 的时间在程序理解上[1]. 代码摘要可以帮助软件开发人员快速理解代码. 但是,由于项目工期紧,开发人员将主要精力花费在编程上,代码摘要编写工作往往被忽视. 代码自动摘要类似于机器翻译,可以为源码自动生成自然语言描述,节省手工编写摘要的时间,提高软件开发效率. 早期的代码自动摘要主要采用基于模板[2-7]和基于信息检索[8-11]的技术. 近年来,基于深度学习的方法被应用到代码自动摘要领域,并逐渐取代传统的方法.

然而,这些基于深度学习的代码自动摘要技术大都致力于为方法粒度的代码片段生成摘要[12]. 对面向对象的语言,例如 Java 语言,类才是项目的基本组成单元[13]. 与方法相比,类的结构更为复杂,代码量也更大. 因此,需要专门针对类摘要生成任务的深度学习模型. Li 等人[14]提出的 ClassSum 将深度学习的方法应用到类摘要生成任务中,并构建了一个包含十几万个样本的类摘要数据集. ClassSum 是一个基于多编码器的类摘要生成模型,使用 2 个编码器分别得到代码的 token 序列和 CSBT 序列的向量表征,再使用带有 Switch 网络的解码器在每个解码时间步动态地为 2 种代码表征分配权重,从而融合代码的语义和结构信息. 类的代码量较大,将整个类的源码作为输入对于编码器而言是一个极大的挑战. 因此,Li 等人[14]创造性地提出了源代码精简策略,只保留类签名、属性和方法签名等关键信息,并使用精简后的代码生成 CSBT 序列.

尽管 ClassSum 融合了类的结构和语义信息,但未充分利用类的层次结构. ClassSum 将抽象语法树(abstract syntax tree, AST)处理为 CSBT 序列,为保证编码效率对 CSBT 序列做截断处理,序列化和截断的操作一定程度上破坏了抽象语法树的结构,无法保留完整的层次结构. ClassSum 的代码精简策略可以有效控制类的代码文本的长度,但是使用单个编码器编码精简后的类,输入序列的长度依旧很长. 当输入序列过长时,循环神经网络很难处理长程依赖问题,从而丢失较早时间步的特征[15]. 此外,使用的语料库缺乏上下文,而在实际的摘要编写过程中,开发人员往往高度依赖程序的上下文信息. 最后,基于深度学习的代码摘要模型由数据驱动,数据集的质量和规模将直接影响生成摘要的质量. 目前的数据集规模较小,模型在数据集上学习到的知识有限,一定程度上影响生成摘要的质量.

针对这些问题,本文提出了基于分层表示和上下文增强的类摘要生成方法,主要工作内容有 3 方面.

1)类包含明显的层次结构,可以分为 3 个层次:类签名、属性和方法. 为了快速理解项目,软件开发人员往往只阅读类签名、属性名和方法签名,很少关注方法的具体实现. 受此代码阅读习惯的启发,提出一种基于分层的方法来表征类. 该方法首先对语料库中的源代码作精简处理[14],每个类只保留它的签名、属性名和方法签名. 编码阶段,对类的层次结构建模,根据各层次的特点设计不同的编码器得到向量表征. 解码阶段,通过注意力机制和 Switch 网络融合每个层次的关键信息生成摘要. 代码分层表示,一方面能对类的层次结构建模,在编码—解码的过程中显示地利用类的层次结构,学习对类摘要生成具有重要作用的信息在各层次的分布情况. 另一方面,可以缩短每个编码器输入序列的长度,缓解长程依赖问题.

2)在数据集构造方面,采用从开源网站下载公开代码仓库的方法获得原始数据,并使用多种静态分析工具从代码仓库中提取类的源代码及其摘要. 同时,分析类之间的继承关系,将父类的签名及父类的摘要作为类的上下文,最终构建了一个包含上下文信息的类摘要数据集. 数据集共有 358 992 个〈Java 类,上下文,摘要〉数据对.

3)数据集中的部分类继承了父类. 对于这些类,将父类的签名及父类的摘要作为上下文编码器的输入,从而引入类的上下文知识,使得模型可以同时从目标类的内部和外部获取信息,进一步提高这部分类的摘要生成质量. 对于没有父类的类,使用无实际意义的符号“〈pad〉”填充上下文序列,模型只学习类的内部知识.

1 相关工作

目前,深度学习已经成为代码摘要领域最主要

的方法，本节对基于深度学习的代码摘要方法进行系统的归纳和阐述，包括基于文本和抽象语法树的代码摘要方法、基于层次结构建模的代码摘要方法、融合上下文信息的代码摘要方法. 近年来，大语言模型逐渐被应用到代码生成、代码理解等代码相关任务上，本节也对相关的工作进行介绍.

1.1 基于文本和 AST 的代码摘要方法

此类方法通常将代码的文本序列或 AST 节点序列输入到深度学习模型中，得到向量表征，再解码生成摘要.

Iyer 等人[16] 率先将深度学习的方法引入代码自动摘要领域，使用带有注意力机制的长短期记忆（long short-term memory, LSTM）神经网络，根据代码语言的 token 序列生成自然语言描述. 之后，基于深度神经网络的技术逐渐成为代码自动摘要生成领域的主流技术. Allamanis 等人[17] 使用卷积神经网络和注意力机制从源代码的 token 序列中提取局部特征，生成类似函数名的摘要.

代码是结构化的语言，其抽象语法树蕴含着丰富的结构信息，对摘要的生成具有重要作用. Hu 等人[18] 提出了一种基于结构的遍历（structure-based traversal, SBT），把抽象语法树扁平化为序列，并将 SBT 序列作为模型输入，学习程序语言的语义和结构信息.

通过代码的文本和抽象语法树可以分别得到语义和结构信息. Leclair 等人[19] 使用 2 个双向的 GRU 编码器分别编码源代码 token 序列和 AST 节点序列，得到源代码的 2 种表示，再通过一个带有注意力机制的解码器生成摘要. 研究人员开始思考如何更有效地融合语义和结构信息. Hu 等人[20] 提出了一个多编码器模型 Hybrid-DeepCom，从源代码和抽象语法树中分别学习语义和结构信息，并使用一个混合注意力机制融合语义和结构信息. Zhou 等人[21] 提出的交换网络在每个解码时间步生成一个权重向量，从而有效地组合源代码和抽象语法树的表示.

此外，Ahmad 等人[22] 使用 Transformer 模型生成摘要. Alon 等人[23] 将源代码表示为 AST 中的一组路径，在解码时使用注意力机制选取相应的路径生成摘要. Zhou 等人[24] 将图神经网络应用到代码摘要生成任务，采用多头注意力机制学习 AST 节点的特征，并通过为其邻居节点分配不同的权重来聚合特征. 张世琨等人[25] 将源码中的签名和 API 调用作为关键字，在解码阶段使用关键字来优化注意力权重的分配.

总之，基于文本和抽象语法树的代码摘要在建模时忽略了代码的层次结构，无法获得代码的层次知识.

1.2 基于层次结构建模的代码摘要方法

源代码及其抽象语法树都具有明显的层次特点，基于层次结构建模的代码摘要方法可以获取代码的层次结构知识.

Zhou 等人[26] 注意到代码的分层特点，即 token 组成语句，语句组成代码片段的层次. 为了建模代码的层次结构，将抽象语法树和代码片段按照语句的粒度切分. 编码阶段，每个语句先使用 token 编码器和内部注意力机制得到 token 层次的代码表征，再经过语句编码器得到语句层次的向量表征，最后将这 2 个层次的表征拼接起来. Wang 等人[27] 将代码、控制流图以及类型增强的 AST 作为输入，采用分层编码为 3 个模态的输入建立 token 层次和语句层次的代码表征. Lin 等人[28] 提出了一种基于块的 AST 分割方法，根据代码控制流图的支配树拆分代码，并为每个代码块生成一个 AST. 然后，使用 Tree-LSTM 建模 AST 子树，将根节点的输出作为 AST 的语法嵌入. 最后，将所有 AST 的嵌入向量经过一个平均池化层，得到最终的全局向量表示，并将该向量表示与每个 token 的词嵌入拼接，作为 Transformer 的输入，解码生成摘要. Shi 等人[29] 以分层的方式拆分代码的 AST，并根据子树的特点采用 2 阶段编码. 第 1 阶段使用基于树的递归神经网络和池化层得到每棵子树的向量表示，第 2 阶段使用另一个基于树的递归神经网络建模子树之间的层次关系，然后将抽象语法树和源码的表示一起输入到 Transformer 解码器中生成摘要.

总之，基于层次结构建模的代码摘要方法利用了代码的层次结构知识，一定程度上提升了摘要的质量，但是仅利用了目标代码的内部知识，未充分利用代码的上下文信息.

1.3 融合代码上下文信息的代码摘要方法

目前，大部分的代码摘要生成模型都在独立的代码摘要对上训练. 但在项目中，代码的功能需要通过项目本身以及它的上下文获取.

Mcburney 等人[30] 使用 PageRank 算法，并根据程序调用图确定函数的上下文，借助软件词语用例模型[31] 从上下文中提取关键字，然后使用模板生成摘要. Yu 等人[32] 使用目标函数的局部和全局信息生成摘要. 首先，使用局部编码器从函数中获取语义信息. 然后为了获得函数的全局信息，将目标函数与所在类的其他函数连接起来，构成类级别的上下文图并

使用图注意力网络得到目标函数类级别的表示. Wang 等人[33]使用类内上下文信息和类间上下文信息丰富函数的语义信息. 根据目标函数所在类的类名挖掘函数与类的关系, 通过 UML 类图得到所属类与其他类的关系, 最后将这些上下文信息、源码和 AST 一同作为模型的输入. Haque 等人[34]提出文件上下文的概念, 即把同一个文件内的其他函数当作上下文, 在编码阶段使用文件编码器获取函数的上下文信息. 此外, 他们将目标方法的方法名作为关键字单独编码, 并提出了动态词表机制缓解未登录词(out of vocabulary, OOV)问题. Bansal 等人[35]抽取目标函数所在项目中的文件作为上下文, 采用与 Haque 等人[34]相似的方式将文件中的方法转化为向量表示, 再使用 RNN 编码器得到方法的最终表示, 供解码阶段使用.

总之, 融合上下文信息的代码摘要方法能够从目标代码的外部获取信息, 提高生成摘要的质量.

1.4 基于大语言模型的代码相关任务

近年来, BERT[36], GPT[37-39]等预训练语言模型在自然语言处理任务上取得了极大的成功. 最近的一些工作将预训练语言模型应用到编程语言上. Feng 等人[40]提出的 CodeBERT 模型使用屏蔽语言建模和替换令牌检测 2 个任务进行预训练, 可以捕获编程语言和自然语言之间的联系, 并生成支持编程语言和自然语言理解任务的通用表示. Wang 等人[41]基于 Transformer 构建了一个预训练的编码器-解码器模型 CodeT5, 并提出一种新的标识符感知的预训练任务. Guo 等人[42]考虑到代码的结构特点提出了模型 GraphCodeBERT, 该模型引入 2 个结构感知的预训练任务, 分别从源码和数据流中学习代码的表示.

由 OpenAI 开发的人工智能内容生成模型 ChatGPT 在机器翻译、文本生成、摘要编写等语言理解和生成任务上展现出强大的能力. OpenAI 最新发布的 GPT-4[43]能够处理文本和图像的输入, 功能进一步增强. 近期也出现了一些针对代码相关任务的大模型. Chen 等人[44]使用 GitHub 上公开的代码数据微调了一个基于 GPT 的代码大模型 Codex. Meta 发布的 Code Llama[45]是基于 Llama 2 的开源大型代码语言模型, 可以执行各种代码相关任务, 例如代码生成、代码摘要生成. Code Llama 在代码生成任务上的表现逼近GPT-4.WizardLM 推出的WizardCoder[46]在CodeLlama 的基础上进行指令微调, 在 4 个广泛认可的数据集上表现出优秀的代码生成能力.

2 理论基础

2.1 编码器-解码器框架

编码器-解码器框架[47]是 NMT 领域的经典框架, 其包括一个编码器和一个带有注意力机制的解码器. 对于代码摘要生成任务而言, 编码器将输入的代码序列转换为向量表征, 解码器则根据这个中间表示解码生成摘要, 具体的执行过程有 2 个阶段:

1)编码阶段. 给定代码输入序列, 首先经过一个词嵌入层, 将输入序列转换为词嵌入向量序列 $x=(\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n)$. 再经过一个循环神经网络, 将词嵌入序列的向量表征编码为一组连续的表示, 得到代码的语义向量序列 $h=(\boldsymbol{h}_1, \boldsymbol{h}_2, \cdots, \boldsymbol{h}_n)$:

$$\boldsymbol{h}_t = f_{\text{enc}}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}), \tag{1}$$

其中 f_{enc} 是一个 RNN 编码器, 编码器的输出 $\boldsymbol{h}_t$ 由当前编码时间步的输入 $\boldsymbol{x}_t$ 和上一时刻的输出 $\boldsymbol{h}_{t-1}$ 共同决定.

2)解码阶段. 解码器根据中间表示 h 以及之前的解码结果预测目标序列下一个 token 的概率分布:

$$p(y_t|\boldsymbol{y}_{<t}, \boldsymbol{x}) = softmax(g(\boldsymbol{s}_t, \boldsymbol{c}_t)), \tag{2}$$

其中解码器通常为单向的 RNN, g 是一个概率估计函数, $\boldsymbol{s}_t$ 是解码器时刻 t 的隐藏状态. 解码器使用注意力机制计算出背景向量 $\boldsymbol{c}_t$:

$$\boldsymbol{c}_t = \sum_{i=1}^{n} \alpha_{ti} \boldsymbol{h}_i. \tag{3}$$

解码器当前状态 $\boldsymbol{s}_t$ 在编码器第 i 个输出 $\boldsymbol{h}_i$ 上的注意力权重 α_{ti} 计算公式为:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{n} \exp(e_{tk})}, \tag{4}$$

$$e_{ti} = a(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^{\mathrm{T}} \boldsymbol{W}_a \boldsymbol{h}_i, \tag{5}$$

其中 a 是对齐得分函数, 用于计算 $\boldsymbol{s}_t$ 和 $\boldsymbol{h}_i$ 的匹配程度.

2.2 LSTM 神经网络

LSTM 神经网络[48]引入记忆细胞的结构, 有效缓解了普通循环神经网络难以处理的长程依赖问题. LSTM 通过遗忘门决定上一个细胞状态中需要被遗忘的内容, 通过输入门决定当前时刻输入中需要被保留的内容, 并根据遗忘门和输入门更新记忆细胞状态. 最后, 通过细胞状态和输出门计算输出信号. LSTM 的结构如图 1 所示.

具体的计算过程为：

$$\boldsymbol{f}_t=\sigma(\boldsymbol{W}_f\boldsymbol{x}_t+\boldsymbol{U}_f\boldsymbol{h}_{t-1}+\boldsymbol{b}_f),\tag{6}$$

$$\boldsymbol{i}_t=\sigma(\boldsymbol{W}_i\boldsymbol{x}_t+\boldsymbol{U}_i\boldsymbol{h}_{t-1}+\boldsymbol{b}_i),\tag{7}$$

$$\widetilde{\boldsymbol{c}}_t=\tanh(\boldsymbol{W}_c\boldsymbol{x}_t+\boldsymbol{U}_c\boldsymbol{h}_{t-1}+\boldsymbol{b}_c),\tag{8}$$

$$\boldsymbol{c}_t=\boldsymbol{f}_t\odot\boldsymbol{c}_{t-1}+\boldsymbol{i}_t\odot\widetilde{\boldsymbol{c}}_t,\tag{9}$$

$$\boldsymbol{o}_t=\sigma(\boldsymbol{W}_o\boldsymbol{x}_t+\boldsymbol{U}_o\boldsymbol{h}_{t-1}+\boldsymbol{b}_o),\tag{10}$$

$$\boldsymbol{h}_t=\boldsymbol{o}_t\odot\tanh(\boldsymbol{c}_t),\tag{11}$$

其中，$\boldsymbol{x}_t$是当前时刻的输入，$\boldsymbol{h}_{t-1}$是上一时刻的输出，$\boldsymbol{f}_t$、$\boldsymbol{i}_t$、$\boldsymbol{o}_t$分别为遗忘门、输入门和输出门，$\boldsymbol{c}_{t-1}$、$\boldsymbol{c}_t$分别为上一时刻和当前时刻的细胞状态，σ是 sigmoid 激活函数.

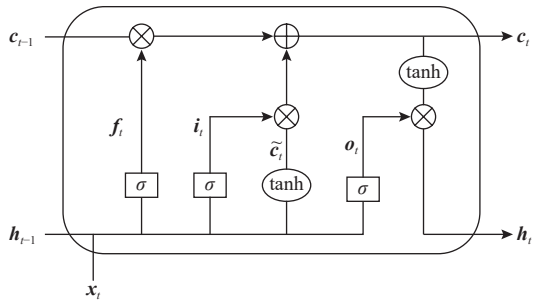


Fig. 1 The structure of a LSTM cell

图 1 LSTM 单元的结构

2.3 Switch 网络

Switch 网络由 Zhou 等人[21]提出，Switch 网络能有效融合代码的文本和抽象语法树的向量表征，提高代码摘要的生成质量. 考虑到解码器可能在每个解码时间步对不同模态的代码表征有偏好，Switch 网络学习自适应权重向量，为各代码表征动态地分配权重.

给定源代码文本和抽象语法树的背景向量$\boldsymbol{c}_t^{\text{lex}}$，$\boldsymbol{c}_t^{\text{AST}}$，Switch 网络根据解码器的隐藏状态$\boldsymbol{s}_t$计算一个权重向量$\boldsymbol{p}\in\mathbb{R}^d$：

$$\boldsymbol{p}=\delta(\boldsymbol{w}_p\boldsymbol{s}_t+\boldsymbol{b}_p),\tag{12}$$

其中，δ是激活函数，用于确保$\boldsymbol{p}$中每个维度的值在[0,1]区间内. 最终的背景向量表示$\boldsymbol{c}_t$由$\boldsymbol{c}_t^{\text{lex}}$和$\boldsymbol{c}_t^{\text{AST}}$的加权平均值经过一个非线性激活函数得到：

$$\boldsymbol{c}_t=\tanh(\boldsymbol{p}\otimes\boldsymbol{c}_t^{\text{lex}}+(1-\boldsymbol{p})\otimes\boldsymbol{c}_t^{\text{AST}}).\tag{13}$$

3 基于分层表示和上下文增强的类摘要

本节详细介绍基于分层表示和上下文增强的类摘要生成模型的架构 HRCE(hierarchical representation and context enhancement). HRCE 能有效表征类的代码文本，提取语义信息和层次结构信息，同时从父类及其相应的摘要中获取类的上下文语义. 利用类的内部和外部的知识，解码得到类功能的自然语言描述. HRCE 的整体架构如图 2 所示，它是一个多编码器-解码器模型，包括分层编码器、上下文编码器和联合解码器 3 部分.

模型的输入为类签名、属性名、方法签名和由

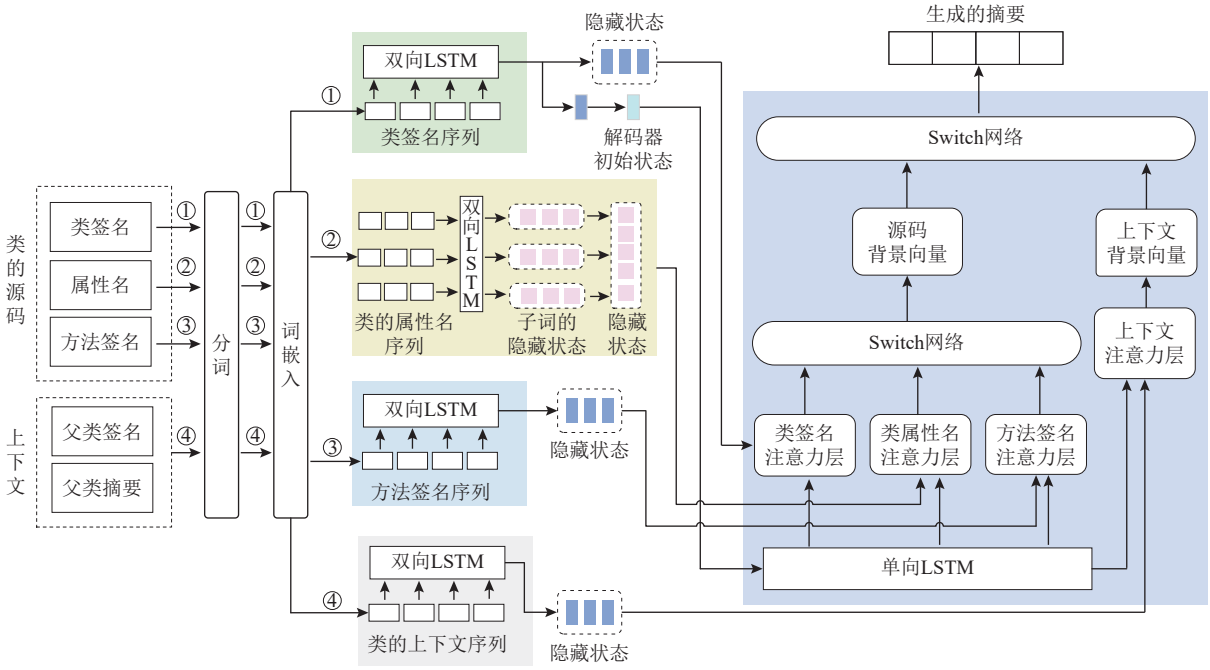


Fig. 2 Class summarization generation model HRCE

图 2 类摘要生成模型 HRCE

父类的签名及父类摘要组成的上下文,输出为生成的摘要. HRCE 采用导师驱动(teacher forcing)的训练方式,在模型训练阶段,解码器的输入为参考摘要,从而最大化摘要中真实 token 的概率,加快模型收敛速度. 测试阶段,模型无法看到参考摘要,解码器的输入为上一时刻的预测结果.

3.1 分层编码器

分层编码器由 3 个独立的子编码器组成:类签名编码器、属性名编码器和方法签名编码器,分别用于提取类签名、属性名和方法签名中的语义信息,并对类的层次结构建模.

3.1.1 类签名编码器

规范的类签名能反映一个类的大致功能,是编写摘要的重要依据[49]. 然而,以往的工作通常将整个代码段的 token 序列输入到编码器中,过长的输入使得神经网络难以处理长程依赖,从而遗忘前面输入的特征. 为了解决这个问题,本文用一个单独的编码器编码类签名,尽可能多地保留其语义特征.

类签名编码器使用一个双向的 LSTM.给定类签名 token 序列的嵌入向量序列 $x_{\mathrm{cs}}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_n)$. 在每个时刻 t 编码一个词向量 $\boldsymbol{x}_t$,得到隐藏状态:

$$\overrightarrow{\boldsymbol{h}}_t,\overleftarrow{\boldsymbol{h}}_t=f(\boldsymbol{x}_t,\boldsymbol{s}_{t-1}),\tag{14}$$

其中 $\overrightarrow{\boldsymbol{h}}_t$ 是前向隐藏状态,$\overleftarrow{\boldsymbol{h}}_t$ 是反向隐藏状态,将 2 个方向的输出连接得到最终的隐藏状态表示 $\boldsymbol{h}_t^{\mathrm{cs}}$:

$$\boldsymbol{h}_t^{\mathrm{cs}}=[\overrightarrow{\boldsymbol{h}}_t,\overleftarrow{\boldsymbol{h}}_t].\tag{15}$$

3.1.2 属性名编码器

属性名编码器由双向的 LSTM 和内部注意力机制组成. 给定一组属性名 $attrs=\{attr_1,attr_2,\ldots,attr_n\}$,经分词处理后,每个属性名都被拆分为若干个 token,即 $attr_i=\{token_1,token_2,\cdots,token_n\}$. $attr$ 经过词嵌入层转换为词嵌入向量序列 $x_i^{\mathrm{attr}}=(\boldsymbol{x}_{i1},\boldsymbol{x}_{i2},\cdots,\boldsymbol{x}_{il})$,再通过 LSTM 编码为隐藏表示序列 $h_i^{\mathrm{attr}}=(\boldsymbol{h}_{i1}^{\mathrm{attr}},\boldsymbol{h}_{i2}^{\mathrm{attr}},\cdots,\boldsymbol{h}_{il}^{\mathrm{attr}})$. 图 3 展示了属性名的编码过程.

考虑到每个 token 对属性名的表示贡献不同,使用一个注意力层提取关键信息,得到属性名的最终表示. 为了与解码器的注意力机制区分,称这个注意力为内部注意力[26]:

$$\boldsymbol{u}_{ij}=\tanh(\boldsymbol{w}_{\boldsymbol{u}}\boldsymbol{h}_{ij}^{\mathrm{attr}}+\boldsymbol{b}_{\boldsymbol{u}}),\tag{16}$$

$$\alpha_{ij}=\frac{\exp(\boldsymbol{u}_{ij}^{\mathrm{T}}\boldsymbol{u}_x)}{\sum_{j=1}^{l}\exp(\boldsymbol{u}_{ij}^{\mathrm{T}}\boldsymbol{u}_x)},\tag{17}$$

$$\boldsymbol{k}_i=\sum_{j=1}^{l}\alpha_{ij}\boldsymbol{h}_{ij}^{\mathrm{attr}},\tag{18}$$

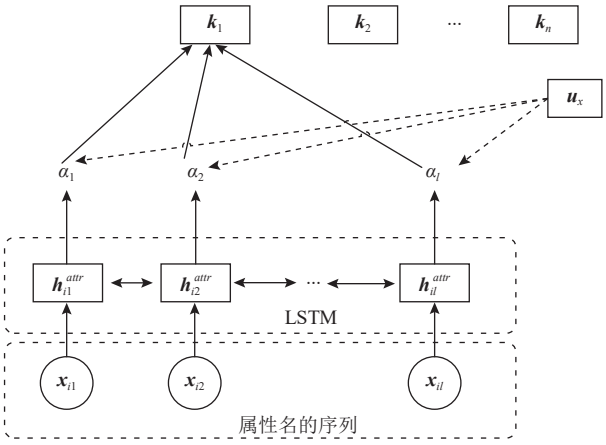


Fig. 3 The encoding process of attribute names

图 3 属性名的编码过程

其中,$\boldsymbol{w}_{\boldsymbol{u}}$ 和 $\boldsymbol{b}_{\boldsymbol{u}}$ 分别为参数矩阵和偏置值,α_{ij} 是归一化的权重. $\boldsymbol{u}_x$ 是通过联合学习得到的背景向量,用于选择 token. $\boldsymbol{k}_i$ 是属性名最终的表示,一组属性名的语义向量序列可表示为 $k^{\mathrm{attr}}=(\boldsymbol{k}_1,\boldsymbol{k}_2,\cdots,\boldsymbol{k}_n)$.

3.1.3 方法签名编码器

与类签名编码器类似,方法签名编码器也是一个双向的 LSTM.给定方法签名 token 序列的词嵌入向量序列 $x_{\mathrm{ms}}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_n)$. 每个时刻 t,编码一个词向量 $\boldsymbol{x}_t$,得到其隐藏状态:

$$\overrightarrow{\boldsymbol{h}}_t,\overleftarrow{\boldsymbol{h}}_t=f(\boldsymbol{x}_t,\boldsymbol{s}_{t-1}),\tag{19}$$

其中 $\overrightarrow{\boldsymbol{h}}_t$ 是前向隐藏状态,$\overleftarrow{\boldsymbol{h}}_t$ 是反向隐藏状态,将 2 个方向的输出连接,得到方法签名的最终的隐藏状态表示 $\boldsymbol{h}_t^{\mathrm{ms}}$:

$$\boldsymbol{h}_t^{\mathrm{ms}}=[\overrightarrow{\boldsymbol{h}}_t,\overleftarrow{\boldsymbol{h}}_t].\tag{20}$$

3.2 上下文编码器

本文将父类的类签名及父类的摘要作为上下文,挖掘父类的类签名及父类的摘要中包含的语义信息,上下文编码器是一个双向的 LSTM.给定上下文符号序列的嵌入向量序列 $x_{\mathrm{cnt}}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_n)$,经神经网络编码后得到隐藏表示序列 $h^{\mathrm{cnt}}=(\boldsymbol{h}_1^{\mathrm{cnt}},\boldsymbol{h}_2^{\mathrm{cnt}},\cdots,\boldsymbol{h}_n^{\mathrm{cnt}})$.

3.3 联合解码器

联合解码器在解码过程中同时关注源代码表示和上下文表示,由单向 LSTM、注意力机制和 Switch 网络[21]组成.

类签名中包含丰富的语义信息,因此使用类签名编码器的语义向量初始化解码器. 在每一个解码时间步,计算源代码的环境向量(context vector)和上下文的环境向量.

源代码经分层编码器编码后得到类签名、属性名和方法签名的隐藏状态序列 h^{cs}、k^{attr}、h^{ms}. 使用注

意力机制分别从这 3 部分中提取关键信息. 公式有：

$$\boldsymbol{c}_t^{\mathrm{cs}} = \sum_{j=1}^{L} \alpha_{tj}^{\mathrm{cs}} \boldsymbol{h}_j^{\mathrm{cs}}, \tag{21}$$

$$\boldsymbol{c}_t^{\mathrm{attr}} = \sum_{j=1}^{L} \alpha_{tj}^{\mathrm{attr}} \boldsymbol{k}_j^{\mathrm{attr}}, \tag{22}$$

$$\boldsymbol{c}_t^{\mathrm{ms}} = \sum_{j=1}^{L} \alpha_{tj}^{\mathrm{ms}} \boldsymbol{h}_j^{\mathrm{ms}}, \tag{23}$$

其中，$\alpha_{tj}^{\mathrm{cs}}$，$\alpha_{tj}^{attr}$，$\alpha_{tj}^{\mathrm{cs}}$分别是编码器隐藏状态$\boldsymbol{h}_j^{\mathrm{cs}}$，$\boldsymbol{k}_j^{\mathrm{attr}}$，$\boldsymbol{h}_j^{\mathrm{ms}}$在时刻$t$的注意力权重系数，计算公式有：

$$\alpha_{tj}^{\mathrm{cs}} = \frac{\exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{cs}} \boldsymbol{h}_j^{\mathrm{cs}}\right)}{\sum_{j=1}^{L} \exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{cs}} \boldsymbol{h}_j^{\mathrm{cs}}\right)}, \tag{24}$$

$$\alpha_{tj}^{\mathrm{attr}} = \frac{\exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{attr}} \boldsymbol{k}_j^{\mathrm{attr}}\right)}{\sum_{j=1}^{L} \exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{attr}} \boldsymbol{k}_j^{\mathrm{attr}}\right)}, \tag{25}$$

$$\alpha_{tj}^{\mathrm{ms}} = \frac{\exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{ms}} \boldsymbol{h}_j^{\mathrm{ms}}\right)}{\sum_{j=1}^{L} \exp\left(\boldsymbol{s}_t \boldsymbol{W}^{\mathrm{ms}} \boldsymbol{h}_j^{\mathrm{ms}}\right)}, \tag{26}$$

其中$\boldsymbol{c}_t^{\mathrm{cs}}$、$\boldsymbol{c}_t^{\mathrm{attr}}$和$\boldsymbol{c}_t^{\mathrm{ms}}$的维度不同，通过线性变换将它们转换为相同的维度：

$$\widetilde{\boldsymbol{c}_t^{\mathrm{cs}}} = \boldsymbol{W}_l^{\mathrm{cs}} \boldsymbol{c}_t^{\mathrm{cs}} + \boldsymbol{b}_{\mathrm{cs}}, \tag{27}$$

$$\widetilde{\boldsymbol{c}_t^{\mathrm{attr}}} = \boldsymbol{W}_l^{\mathrm{attr}} \boldsymbol{c}_t^{\mathrm{attr}} + \boldsymbol{b}_{\mathrm{attr}}, \tag{28}$$

$$\widetilde{\boldsymbol{c}_t^{\mathrm{ms}}} = \boldsymbol{W}_l^{\mathrm{ms}} \boldsymbol{c}_t^{\mathrm{ms}} + \boldsymbol{b}_{\mathrm{ms}}. \tag{29}$$

对 Switch 网络[21]稍做改动，使其能够为 3 个环境向量生成注意力权重. 首先，根据解码器的隐藏状态$\boldsymbol{s}_t$计算概率向量$\boldsymbol{p}^{\mathrm{cs}} \in \mathbb{R}^d$：

$$\boldsymbol{p}^{\mathrm{cs}} = \delta(\boldsymbol{W}_{\mathrm{cs}} \boldsymbol{s}_t + \boldsymbol{b}_{\mathrm{cs}}), \tag{30}$$

其中$\boldsymbol{p}^{\mathrm{cs}}$是环境向量$\widetilde{\boldsymbol{c}_t^{\mathrm{cs}}}$的权重，$\delta$是一个激活函数，确保向量$\boldsymbol{p}^{\mathrm{cs}}$的每一个维度的值都在[0,1]区间内.

向量$\boldsymbol{p}$为环境向量$\widetilde{\boldsymbol{c}_t^{\mathrm{attr}}}$和$\widetilde{\boldsymbol{c}_t^{\mathrm{ms}}}$的权重向量之和：

$$\boldsymbol{p} = 1 - \boldsymbol{p}^{\mathrm{cs}}. \tag{31}$$

根据$\boldsymbol{p}$和$\boldsymbol{s}_t$分别计算环境向量$\widetilde{\boldsymbol{c}_t^{\mathrm{attr}}}$的权重$\boldsymbol{p}^{\mathrm{attr}}$和环境向量$\widetilde{\boldsymbol{c}_t^{\mathrm{ms}}}$的权重$\boldsymbol{p}^{\mathrm{ms}}$：

$$\boldsymbol{w}_{\mathrm{attr}} = \delta(\boldsymbol{W}_{\mathrm{attr}} \boldsymbol{s}_t + \boldsymbol{b}_{\mathrm{attr}}), \tag{32}$$

$$\boldsymbol{p}^{\mathrm{attr}} = \boldsymbol{w}^{\mathrm{attr}} \boldsymbol{p}, \tag{33}$$

$$\boldsymbol{p}^{\mathrm{ms}} = \boldsymbol{p} - \boldsymbol{p}^{\mathrm{attr}}. \tag{34}$$

源代码环境向量为$\widetilde{\boldsymbol{c}_t^{\mathrm{cs}}}$、$\widetilde{\boldsymbol{c}_t^{\mathrm{attr}}}$和$\widetilde{\boldsymbol{c}_t^{\mathrm{ms}}}$的加权和：

$$\boldsymbol{c}_t^{\mathrm{src}} = \tanh\left(\boldsymbol{p}^{\mathrm{cs}} \otimes \widetilde{\boldsymbol{c}_t^{\mathrm{cs}}} + \boldsymbol{p}^{\mathrm{attr}} \otimes \widetilde{\boldsymbol{c}_t^{\mathrm{attr}}} + \boldsymbol{p}^{\mathrm{ms}} \otimes \widetilde{\boldsymbol{c}_t^{\mathrm{ms}}}\right). \tag{35}$$

上下文环境向量$\boldsymbol{c}_t^{\mathrm{cnt}}$的计算与式 (21) 和式 (24) 相同. 通过 Switch 网络得到源代码环境向量的权重$\boldsymbol{p}^{\mathrm{src}}$和上下文环境向量的权重$\boldsymbol{p}^{\mathrm{cnt}}$. 最终的环境向量为：

$$\boldsymbol{c}_t = \tanh\left(\boldsymbol{p}^{\mathrm{src}} \otimes \boldsymbol{c}_t^{\mathrm{src}} + \boldsymbol{p}^{\mathrm{cnt}} \otimes \boldsymbol{c}_t^{\mathrm{cnt}}\right). \tag{36}$$

最后，根据解码器的隐藏状态$\boldsymbol{s}_t$和环境向量$\boldsymbol{c}_t$，计算下一个单词的概率分布：

$$p(y_t \mid \boldsymbol{y}_{<t}, \boldsymbol{x}) = softmax(\boldsymbol{W}_s \tanh(\boldsymbol{W}_t [\boldsymbol{c}_t; \boldsymbol{s}_t])). \tag{37}$$

4 数据集

4.1 数据集的构建

从 CodeSearchNet 数据集中选取在 Github 上开源的 Java 代码仓库地址，成功下载 4 188 个项目，采用与 Li 等人[14]相似的规则过滤项目，最终保留 3 955 个 Java 项目. 对于每个 Java 项目，使用 SrcML 工具抽取项目中的类及其对应的 Javadoc 注释.Javadoc 注释一般分为 3 个部分：第 1 部分简要描述类的功能，是一个摘要性的句子；第 2 部分用一句或者多句话详细描述类的作用；第 3 部分是特殊说明，包括版本说明、参数说明等. 因此，选取 Javadoc 的第一句话作为类的摘要. 此外，分析类之间的继承关系，将父类及其相应的摘要作为类的上下文. 最终得到 358 992 个〈Java 类，上下文，摘要〉数据对.

4.2 数据预处理与数据集划分

对于数据集中的每个样本，删除类的属性类型和方法体，只保留类的签名、属性名和方法签名. 具体有 3 个步骤：

1）用 SrcML 将 Java 类转换为 XML 文件.

2）以深度优先搜索的方式遍历 XML 文件. 当遍历到方法或嵌套类时，根据〈body〉标签删除其内部元素. 当遍历到属性定义时，根据〈type〉标签删除属性类型.

3）从修改后的 XML 文件中获取类的签名、属性名和方法签名.

图 4 以一个 Java 类为例，展示代码的精简过程. 首先，将类的源码转换为 XML 文件. XML 的结构较为复杂，为了清楚地展示类的精简过程，此处不改变 XML 文件的层级结构，但删减和修改了部分内容. 如图 4 (b)所示，〈decl_stmt〉标签的内容是属性定义，包括属性的类型和名称，它根据〈type〉标签删除属性类型.〈function〉标签内是方法的定义，通过〈body〉标签删除方法体. 当删除类的所有的属性类型和方法体后，得到一个新的 XML 文件，从此文件生成代码精简后的类.



(a) 类的源码 (b) 类的XML文件

(c) 去除属性类型和方法体的XML文件 (d) 代码精简后的类

Fig. 4 The process of class code reduction

图 4 类代码精简过程

统计代码精简后的数据集中类的属性名和方法签名的个数分布.结果如表 1 和表 2 所示，85% 的类的属性名个数不超过 5，88% 的类的方法签名个数不超过 15. 方法签名编码器和属性名编码器的输入序列分别由类的方法签名和属性名拼接而成，考虑到编码效率为属性名个数超过 5 的类，随机保留 5 个属性；为方法签名个数超过 15 的类，随机保留 15 个方法签名. 随机保留机制确保类中的每个属性名和方法签名都有同等被选中的概率，是综合考虑编码效率和语义保留后采取的方法. 数据集中的少量样本会因为舍弃部分属性和方法签名丢失语义信息，但若不对这部分样本做处理，编码器同样会因为输入序列过长丢失语义信息，导致编码效率下降. 例如：数据集中类最多包含 1 787 个方法，而方法签名的平均长度为 8 个 token，拼接而成的序列长度近 2 万个 token.

Table 1 Statistics on the Number of Attribute

表 1 属性名个数统计

序号	个数	属性名
1	最大	1 684
2	平均	3
3	<1	50%
4	<3	75%
5	<5	85%

Table 2 Statistics on the Number of Method Signature

表 2 方法签名个数统计

序号	个数	属性名
1	最大	1 787
2	平均	8
3	<4	50%
4	<8	75%
5	<15	88%

对类各层次的代码片段以及类摘要作分词和截断处理. 根据驼峰和蛇形命名规则拆分代码中的组合词，并将所有的 token 转化为小写，从而降低未登录词的比例. 统计分词后的类签名、属性名、方法

签名和上下文的序列长度，结果如表 3 所示. 考虑到编码效率，对类各层次的代码序列作截断处理，使得类签名的长度不超过 20 个 token，每个属性名的长度不超过 5 个 token，每个方法签名的长度不超过 15 个 token. 借助 NLTK 对类摘要做分词处理，根据表 3 的长度统计结果，将长度超过 20 个 token 的摘要序列截断.

Table 3 Sequence Length Statistics

表 3 序列长度统计

序列名称	最大长度	平均长度	25% 长度	50% 长度	75% 长度
类签名	1 105	10	6	9	14
属性	25	4	3	4	4
方法签名	968	13	8	11	15
上下文	648	0	0	0	15
摘要	50	10	5	8	13

对由父类的签名和摘要拼接而成的上下文序列也做截断处理，确保上下文序列的长度不超过 20 个 token.对于没有父类的类，使用无实际意义的符号〈pad〉填充上下文序列.

将预处理后的数据集按照 7∶1∶1∶1 的比例划分为训练集、验证集、混项目测试集和跨项目测试集. 首先，随机选取一部分项目，使得这些项目包含的样本量占数据总量的 10%，并把这部分数据作为跨项目测试集. 剩余的数据按照既定的混项目比例随机划分为训练集、验证集和混项目测试集. 数据集的划分情况如表 4 所示.

Table 4 Splits of Dataset

表 4 数据集划分

数据集	样本数	项目数
混项目测试集	35 900	2 697
跨项目测试集	35 186	187
验证集	35 900	2 687
训练集	252 006	3 677
总数	358 992	3 955

5 实验设置

5.1 基准模型

本文选用 7 个基准模型与 HRCE 做效果对比. 其中 4 个是经典的方法摘要生成模型，选用这 4 个模型的原因有 2 点：1）这些基准模型将代码文本的 token 序列或抽象语法树节点序列输入到编码器-解码器框架，生成代码的自然语言描述. 类的源码同样可以处理为 token 序列和抽象语法树节点序列，因此这些模型也能生成类摘要. 2）HRCE 致力于学习类的语义和结构信息以及缓解长程依赖问题，这些基准模型的设计也是为了解决这些问题，可以检验 HRCE 的有效性. 基准模型 ClassSum 是首个基于深度学习的类摘要生成模型. 近年来，大语言模型表现出了优秀的代码理解能力，本文选用 CodeBERT 和 Code Llama 这 2 种不同类型的大语言模型来评估它们在类摘要生成任务上的表现. 下面详细介绍 7 种基准模型的原理：

1）CodeNN[16]. CodeNN 是一个端到端的数据驱动的代码摘要生成模型，使用带有注意力机制的 LSTM 解码器，直接从源代码 token 序列的词嵌入向量生成摘要.

2）SBT[18]. SBT 是一种抽象语法树遍历方法. 在对抽象语法树进行前序遍历时，为每个子树添加一对括号. 该遍历方式保证可以从 SBT 序列恢复到原始的抽象语法树，能够较好地保留代码的结构信息.

3）AST-attendgru[19]. AST-attengru 是一个基于 GRU 的编码器-解码器模型，该模型使用 2 个编码器分别编码代码的 token 序列和 SBT 序列. 解码时使用注意力机制得到 token 序列和 SBT 序列的上下文向量，并通过拼接操作结合 2 个上下文向量同时学习到代码的语义和结构信息.

4）Transformer[50]. Transformer 是一个基于多头注意力机制的模型，该模型能够有效捕获长程依赖，因此将它作为本文的一个基准模型.

5）ClassSum[14]. ClassSum 是首个基于深度学习技术的类摘要生成模型，该模型通过精简代码去除类的非关键信息，只保留类签名、属性和方法签名. 从精简后的代码以及 CSBT 序列中获取语义和结构信息，解码时通过门控网络融合这 2 部分信息，为 Java 类生成摘要.

6）CodeBERT[40]. CodeBERT 是基于 Transformer 的预训练代码语言模型，该模型使用掩码语言建模任务和替换令牌检测任务进行预训练，能够生成文本的通用表示. 经过微调后，在代码文档生成等下游任务上表现出色.

7）Code Llama[45]. Code Llama 是基于 Llama 2 的大型代码语言模型，在代码生成任务上的一次通过率超越了 ChatGPT，逼近 GPT-4.由于 Code Llama 在代码相关任务上的能力不逊色于 GPT-4，且是一个被广泛使用的开源免费的模型，因此将其作为本文的基准

模型.

5.2 评估指标

BLEU[51]、METEOR[52]和ROUGE-L[53]是机器翻译和代码摘要领域广泛使用的自动评估指标,本文使用这3种指标评估生成的类摘要的质量.

1)BLEU. BLEU根据n-gram可以划分为多个评价指标.常见的有BLUE-1、BLUE-2、BLUE-3、BLUE-4,其中的数字表示连续单词的个数.给定一个候选摘要,BLEU计算其与参考摘要的n-gram匹配精度的加权几何平均值,并且对短摘要施加惩罚.

2)METEOR. BLEU是一种基于精确度的评估指标,METEOR在BLEU的基础上,综合考虑精确度、召回率和词序,并进行同义词匹配.

3)ROUGE-L. ROUGE-L是ROUGE指标的一种,它根据候选摘要和参考摘要最长公共子序列的长度衡量二者的相似度.

5.3 参数设置

1)HRCE的参数设置为:

①编码器和解码器的词嵌入维度设置为256.编码器采用双向LSTM,每个方向的维度都为512,层数设置为2.解码器采用单向LSTM,隐藏层维度设置为512,层数设置为2.嵌入层和循环层都采用随机失活策略,将失活率设为0.2.

②代码和摘要的词表都未设置固定大小,而是根据单词在数据集中出现的频率决定是否加入词表.设置词频为2,并将在数据集中出现次数大于等于2的单词加入词表.

③模型使用Adam[54]优化器训练,初始学习率设为0.001,以最大限度地减少在训练集上的损失.将批量大小设置为32.

④为了防止模型过拟合,在训练时采用早停策略,当模型的性能连续4个epoch都没有提升则停止训练.此外,限定模型训练次数的上限为100个epoch.

⑤模型在验证和测试阶段均使用集束搜索[55],并将束宽设为4.

2)基准模型的参数设置为:

①AST-attendgru、SBT、ClassSum和CodeNN的隐藏层维度都设置为256.

②Transformer以及CodeBERT的参数配置分别与Vaswani等人[50]和Feng等人[40]的工作保持一致.

6 实验结果与分析

本节将从整体性能、消融实验以及数据集划分方式3个方面分析HRCE的效果.

6.1 整体性能

表5显示了HRCE和基准模型的实验数据,从整体上看,HRCE在所有的评估指标上都优于基准模型.

Table 5 Performance of HRCE and Baseline Models on Hybrid-Projects Test Set and Cross-Projects Test Set

表5 HRCE和基准模型在混项目测试集和跨项目测试集上的表现

模型	混项目测试集				跨项目测试集			
	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L
Code Llama	0.40	1.90	11.47	14.54	0.55	2.05	11.72	15.67
SBT	5.01	11.60	10.97	22.61	0.35	2.66	4.04	9.17
CodeNN	13.78	20.32	17.06	43.18	0.36	4.02	6.73	15.72
Transformer	19.09	26.22	21.94	43.18	1.21	5.61	9.83	20.75
AST-attendgru	24.18	31.15	24.27	47.51	1.31	7.01	11.26	24.60
CodeBERT	24.06	32.27	24.92	48.59	2.24	9.19	13.19	**28.51**
ClassSum	28.13	35.38	26.84	50.96	2.83	8.90	13.02	27.18
HRCE	**30.32**	**37.01**	**28.06**	**52.18**	**3.81**	**9.42**	**13.74**	27.20

注:加粗数值表示最优结果.

具体来看,专门为类摘要生成任务设计的模型ClassSum相较于其他基准模型更具竞争力.ClassSum采用代码精简策略去除类源码中的非关键代码,使用2个双向LSTM编码器分别编码精简后的代码token序列和CSBT节点序列,解码时利用门控网络动态地生成权重向量来融合代码的语义和结构信息,使得每一个解码时间步可以对不同模态的向量表征有偏好.AST-attendgru使用2个GRU编码器分别编码代码的token序列和SBT节点序列,以此学习代码的语义和结构信息.但是,AST-attendgru直接编码原

始代码文本的 token 序列，由于类的源码中包含许多非关键信息，编码器很难提取到有用特征. 解码阶段，AST-attendgru 通过拼接操作结合 2 个编码器编码结果的注意力向量，导致不同模态的向量表征在每个解码时间步的贡献都相同. 因此，AST-attendgru 在各个指标上的表现都远不及 ClassSum.

Transformer 的输入也是类的源码，但 Transformer 是基于多头注意力机制的模型，能够有效捕获长程依赖. Transformer 与 AST-attendgru 的效果差异主要来自于模型的输入.AST-attendgru 将类源码的 token 序列和 SBT 节点序列作为输入，从中获取语义和结构信息，而 Transformer 的输入不包含 SBT 节点序列，无法学习到类的结构信息，生成的摘要质量不高. CodeNN 直接从类源码 token 序列的词嵌入向量中产生摘要，无法很好地捕获语义特征，生成的摘要质量较差.

SBT 与其他模型的性能差距源于 SBT 丢失了语义信息. 相对于结构信息，语义对摘要的生成更为重要. SBT 将源码转化为抽象语法树，并将抽象语法树叶节点的类型和值通过符号“_”连接起来作为一个 token，然后使用基于结构的遍历方式将抽象语法树扁平化为序列作为模型的输入. 这种处理方式能够很大程度上保留抽象语法树的结构信息，但是没有构建源码中组合词的子词嵌入表示，丢失了大量的语义信息. 此外，类的抽象语法树规模很大，为了保证编码器效率，需要对抽象语法树结点序列做截断处理，也会丢失部分结构信息. 综上，对类摘要生成任务而言，SBT 丢失语义信息也无法完整地保留结构信息，生成的类摘要质量较差，在 BLEU-4, BLEU-4(s), METEOR 和 ROUGE-L 上的值分别为 5.01, 11.60, 10.97, 22.61.

微调后的 CodeBERT 在类摘要生成任务上的表现优于大部分的基准模型，但不及 ClassSum 和 HRCE. CodeBERT 在 6 种编程语言上进行预训练，能够捕获自然语言和编程语言之间的联系，表现出强大的语义理解能力. 因此即使只输入类的源码，CodeBERT 的性能也能接近双模态输入的 AST-attendgru. 但是，CodeBERT 的预训练语料都是函数级的代码，而类的结构更为复杂，在语义理解上面临着更大的挑战. 因此 CodeBERT 的性能不及专门为类摘要生成任务设计的模型.

Code Llama 根据提示信息生成摘要，类摘要生成任务的提示模板如图 5 所示. 设计 2 个提示模板，比较只输入类的源码和同时输入源码与上下文这 2 种情况下，Code Llama 在类摘要生成任务上的表现.结果如表 6 所示，在 2 个测试集上，添加上下文信息后生成的摘要在各个衡量指标上的表现都优于只根据源码生成的摘要，表明外部信息对类摘要生成的有用性.

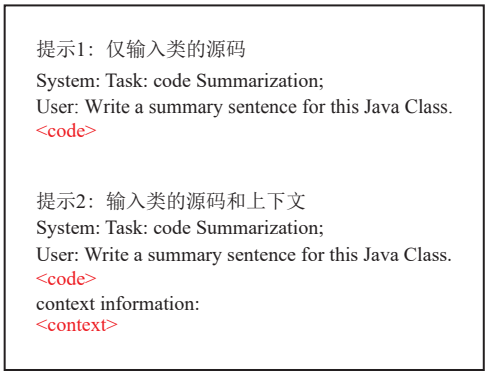


Fig. 5 The prompt template of Code Llama

图 5 Code Llama 的提示模板

Table 6 Performance of Code Llama on Hybrid-Projects Test Set and Cross-Projects Test Set with and Without Context

表 6 Code Llama 有无上下文时在混项目测试集和跨项目测试集上的表现

模型输入	混项目测试集				跨项目测试集			
	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L
无上下文	0.19	1.28	9.86	11.24	0.26	1.57	11.08	13.87
有上下文	**0.40**	**1.90**	**11.47**	**14.54**	**0.55**	**2.05**	**11.72**	**15.67**

注：加粗数值表示最优结果.

从表 5 的结果来看，与其他模型相比，Code Llama 生成的摘要与参考摘要的相似度最低，主要有 2 个原因. 1）Code Llama 没有在类摘要生成任务上进行微调，无法学习类摘要的编写模式. 如表 7 的例 1 所示，Code Llama 生成的摘要在形式上与参考摘要存在较大差异. 参考摘要简洁明了、概括性较强，而 Code Llama 生成的摘要冗长，包含大量非必要的描述，无法达到通过摘要快速掌握类的功能的目的. 2）大语言模型虽然能够执行广泛的文本理解和生成任务，但是也存在一些局限性，例如“幻觉”[56]，即生成无意

Table 7 Class Summarization Generated by Code Llama

表 7 Code Llama 生成的类摘要

例子编号	代码	生成的类摘要

1	public class IPv4Generator implements Generator<String> {
	private static final String IP_ADDRESS_FORMAT =
	"%s. %s. %s. %s";
	private final Generator<Integer> octetGenerator;
	public IPv4Generator() {
	this (new RandomPositiveIntegerGenerator (0, 256));
	}
	public IPv4Generator (Generator<Integer> delegate) {
	octetGenerator = delegate;
	}
	@Override
	public String next() {
	return format (IP_ADDRESS_FORMAT, octetGenerator.next(),
	octetGenerator. next(), octetGenerator. next(),
	octetGenerator. next());
	}
	}
		参考摘要：class generates random valid ipv addresses
		
		Code Llama：The IPv4Generator class implements the Generator interface and generates IPv4 addresses by combining four octets, each generated by a separate Generator instance. The class uses the IP_ADDRESS_FORMAT string to format the generated octets into a valid IPv4 address.

2	public class ChartImageManager {
	
	private final Chart chartModel;
	⋮
	public ChartImageManager (Chart chartModel, String
	outputFormat) throws Exception {
	this (chartModel, outputFormat, null, null, null, null);
	}
	⋮
	}
		参考摘要：represents a list of identifiers in square brackets
		
		Code Llama： ter) deviceRenderer).
		get ImageMap(); } } catch
		(Exception ex) { throw new
		ChartException (ChartEnginePlugin. ID,
		ChartException. GENERATION, ex); }
		finally { if (deviceRenderer != null) {
		deviceRenderer. dispose(); } }

义或不忠实于输入的文本. Code Llama 在执行类摘要生成任务的过程中，也存在“幻觉”问题. 如表 7 的第 2 个例子，Code Llama 没有根据提示模板的要求生成摘要，而是生成了与模型输入无关的代码. 经统计，所有的测试样本中约五分之一的摘要生成结果未忠实于输入. “幻觉”问题以及大量的冗余描述导致 Code Llama 在类摘要生成任务上的表现不佳.

HRCE 的 BLEU-4 得分为 30.32、BLEU-4(s) 得分为 37.01、METEOR 得分为 28.06、ROUGE-L 得分为 52.18，与最好的基准模型 ClassSum 相比，分别获得了 7.79%、4.61%、4.55%、2.39% 的性能提升. HRCE 与 ClassSum 一样，对类的源码做精简处理，只保留类的签名、属性名和方法签名这些关键代码. 与 ClassSum 不同的是，HRCE 采用分层表示的方法对类的关键代码进行层次结构建模，能从类的代码文本中同时学习语义和层次结构信息，无需输入代表结构信息的 CSBT 序列. 且分层表示的设计缩短了每个编码器输入序列的长度，有效缓解了模型记忆长序列的挑战. 此外，HRCE 考虑到类的继承关系，使用上下文编码器学习父类的语义信息，使得有父类的类能够同时获取内部和外部的信息，进一步提高这些类的摘要生成质量. HRCE 在所有衡量指标上的表现，证明分层表示和上下文增强的方法可以为类生成高质量的摘要.

6.2 消融实验

为了检验 HRCE 各模块的有效性，设计了一组消融实验，通过去除模型的某一部分得到 3 个变体模型：

1）HRCE-NON-SIG，表示 HRCE 去除分层编码中的类签名编码器或方法签名编码器. 该模型将类签名 token 序列和方法签名 token 序列拼接起来，输入到一个双向 LSTM 编码器. 由于类签名编码器和方法签名编码器都由双向 LSTM 构成，因此该模型可同时表示去除类签名编码器或方法签名编码器.

2）HRCE-NON-ATTR，是由 HRCE 去除属性编码器得到的变体. 为了保留属性名的语义信息，将所有的属性名拼接为一个 token 序列，与方法签名一同输入到方法签名编码器中. 该处理方式将属性名视为普通的 token 序列，不做特殊的编码处理，以此检验属性编码器的有效性.

3）HRCE-NON-INHER，是 HRCE 去除上下文编码器得到的变体. 该模型只对类进行分层表示，不添加上下文信息，模型解码时只能根据类本身的信息生成摘要而无法获得外部知识.

表 8 给出 HRCE 和 3 种变体模型在各个评估指

Table 8 Ablation Experiment Results on Hybrid-Projects Test Set and Cross-Projects Test Set

表 8 在混项目测试集和跨项目测试集上的消融实验结果

模型	混项目测试集				跨项目测试集			
	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L	BLEU-4	BLEU-4(s)	METEOR	ROUGE-L
HRCE-NON-SIG	29.13	35.82	27.34	51.12	3.21	8.68	12.92	25.89
HRCE-NON-ATTR	29.64	36.42	27.68	51.92	3.62	**9.49**	**13.99**	**28.26**
HRCE-NON-INHER	28.22	34.72	26.55	49.58	3.49	9.00	13.21	26.56
HRCE	**30.32**	**37.01**	**28.06**	**52.18**	**3.81**	9.42	13.74	27.20

注：加粗数值表示最优结果.

标上的表现. 结果表明, HRCE-NON-INHER 的表现最差, 在各个指标上的分数都低于其他变体. 这是因为 HRCE-NON-INHER 只利用类的内部语义和结构信息生成摘要, 而类的作用往往是由其本身以及项目中与它有关联的上下文共同决定的. 换言之, 摘要中的内容可能并不直接来自类的内部代码, 而是出现在类的上下文中. 表 8 的结果证明在类摘要生成模型中添加上下文信息的有用性. 值得注意的是, 结合表 5 和表 8 的实验结果, 可以看到 HRCE-NON-INHER 和 ClassSum 在 BLEU-4 上的分数分别为 28.22 和 28.13. 这 2 个模型都只利用类的内部信息生成摘要, 区别在于 ClassSum 通过 SBT 序列获取类的结构信息, 而 HRCE-NON-INHER 的输入中并不包含能够表征代码结构的 SBT 序列, 但二者在 BLEU 和 METEOR 指标上的表现几乎相同. 这一结果证明对类进行分层表示的有效性, 不仅能够获取类的语义信息, 还能够获取类的层次结构信息.

HRCE-NON-SIG 略好于 HRCE-NON-INHER, 但与完整的模型 HRCE 相比, BLEU-4 的分值从 30.32 下降到了 29.13, 这表明为类签名单独编码的必要性. 当循环神经网络的输入序列较长时, 容易产生长程依赖问题, 从而忘记较早编码时间步的特征. 与类的其他部分相比, 类签名的语义信息更为重要, 有时只需阅读类的签名就可以了解类的功能. 根据 4.2 节的统计, 类签名的 token 序列往往很短, 使用单独的类签名编码器, 可以最大程度地保留其语义信息.

HRCE-NON-ATTR 的表现也不及完整的模型 HRCE, 这表明对每个属性单独编码并使用内部注意力机制得到每个属性名的向量表示是适合属性名的编码方式, 可以得到更有利于摘要生成的属性名向量表征.

6.3 不同数据集划分方式

表 5 和表 8 分别展示了 HRCE 和基准模型在混项目测试集和跨项目测试集上的得分. 与混项目测试相比, HRCE 跨项目测试时, 在 BLEU-4, BLEU-4(s), METEOR 和 ROUGE-L 上的性能分别下降了 87%, 75%, 51% 和 48%.

这是因为同一个项目中通常会存在许多相似代码和复用代码. 数据集混项目随机划分时, 这些相似或复用的代码片段会被划分到训练集、验证集和测试集中. 因此, 训练集和测试集的样本差异不会很大, 模型在测试集上也能得到与训练集一致的表现. 数据集跨项目划分时, 测试集中的项目都不会出现在训练集中. 由于缺少相似的代码片段, 训练集和测试集的样本差异巨大, 因此在各个衡量指标上的得分都大幅度降低.

值得注意的是, Code Llama 在混项目测试集和跨项目测试集上的表现几乎一致. 因为 Code Llama 通过提示生成摘要, 无需使用类摘要数据从头训练或微调, 所有的测试数据对模型而言都是跨项目的数据.

6.4 输出示例

表 9 展示 HRCE 和基准模型生成的摘要, 与基准模型相比, HRCE 生成的摘要与参考摘要最相似. Code Llama 生成的类摘要实例已在 6.1 节中进行分析, 本节不再重复说明.

第 1 个例子中, 摘要的信息主要来自类签名, 包含“constant”“expression”“number”等关键字, HRCE 采用分层表示, 使用单独的编码器编码类签名, 缓解了长程依赖问题, 最大程度地保留了类签名的语义信息, 因此能够准确捕获这些关键字. ClassSum 面临记忆长序列的挑战, 生成的摘要虽然也包含了所有关键字, 但后半段摘要“number constant number and num”无法准确描述类的功能. CodeBERT 只在函数级的编程语言上进行预训练, 在理解内容更为复杂的 Java 类上面临挑战, 生成的摘要对类功能描述不够准确. AST-attendgru 和 Transformer 都未采用代码精简策略, 难以提取有效特征, 因此摘要只包含了部分关键词. CodeNN 和 SBT 都无法很好地学习语义信息, 生成的摘要与代码的相关性较低.

Table 9 Output Examples of Different Models

表 9 不同模型的输出示例

编号	代码文本	生成的摘要

1	上下文：
	public abstract class NumberExpression implements ComparableExpression<Number>
	marker interface for constant expressions ;
	public interface ConstantExpression <T> extends Expression<T>
	abstract base class for all number expressions
	类的代码：
	class NumberConstant extends NumberExpression implements ConstantExpression<Number> {
	private final Number number ;
	public NumberConstant (Number number) {
	this. number = number ;
	}
	public Number evaluate (Object o) {
	return number ;
	}
	}
		参考摘要：expression that represents a constant number value
		
		HRCE: expression that represents a constant number value
		ClassSum: expression that represents number constant number and num
		CodeBERT: represents the constant number value
		AST-attendgru: represents a constant number
		Transformer: represents a constant expression
		CodeNN: the bpmn expression element
		SBT: represents an integer range

2	上下文：
	public interface Generator<T>
	generates instance of lt t gt
	
	类的代码：
	public class IPv4Generator implements Generator<String> {
	private static final String IP_ADDRESS_FORMAT = ″%s. %s. %s. %s″ ;
	private final Generator<Integer> octetGenerator ;
	public IPv4Generator () {
	this (new RandomPositiveIntegerGenerator (0, 256)) ;
	}
	public IPv4Generator (Generator<Integer> delegate) {
	octetGenerator = delegate ;
	}
	@Override
	public String next () {
	return format (IP_ADDRESS_FORMAT, octetGenerator. next () , octetGenerator. next () , octetGenerator. next () , octetGenerator. next ()) ;
	}
	}
		参考摘要：class generates random valid ipv addresses
		
		HRCE: class generates random valid ipv addresses
		ClassSum: link generator for link <unk>
		CodeBERT: the generator for address
		AST-attendgru: default generator
		Transformer: ipv address generator
		CodeNN: a generator for type
		SBT: generates a random integer value

3	上下文：
	public abstract class AbstractStreamExporterImpl extends AssignableBase<Archive<?>> implements StreamExporter
	base support for i o stream based exporters ;
	public interface TarExporter extends StreamExporter
	exporter used to represent an link assignable in tar format
	
	类的代码：
	public class TarExporterImpl extends AbstractStreamExporterImpl implements TarExporter {
	public TarExporterImpl (final Archive<?> archive) {
	super (archive) ;
	}
	@Override
	public InputStream exportAsInputStream () {
	final AbstractExporterDelegate<InputStream> exportDelegate = new TarExporterDelegate (this. getArchive ()) ;
	return exportDelegate. export () ;
	}
	}
		参考摘要：implementation of link tar exporter used to export an archive as a tar format
		
		HRCE: implementation of link tar exporter used to export an archive as a zip format
		ClassSum: implementation of link tar exporter
		CodeBERT: implementation of link tar exporter
		AST-attendgru: implementation of link tar exporter
		Transformer: implementation of link tar file exporter for tar files
		CodeNN: implementation of code amazon tar
		SBT: implementation of a html table foot

第2个例子，摘要的信息来自类签名和方法签名. HRCE的分层表示可以提取各层次的特征，Switch网络在每个解码时间步动态地选择并融合各层次的信息，因此可以从类签名和方法签名中获取“class”“generator”“random”“ipv”“address”等关键信息组成摘要. 其他基准模型由于未能充分表示类的复杂结构，生成的摘要与参考摘要的相似性较低.

第3个例子，摘要的部分信息来自父类. HRCE使用上下文编码器引入父类信息，因此生成的摘要中包含了“used”“format”等来自父类的关键字. ClassSum, AST-attendgru和CodeBERT能从类的源码中获取关键词生成摘要“implementation of link tar exporter”，但由于遗漏父类信息，此摘要无法完整描述类的功能. Transformer也能获取到源码的关键信息，但是生成了一些无关的描述. CodeNN和SBT只能获取源码的部分关键信息，生成的摘要包含大量无关的单词.

7 总　　结

本文针对类摘要自动生成任务，提出了一种基于分层表示和上下文增强的方法HRCE.实验结果表明，HRCE能够为类生成高质量的注释，在所有的评估指标上超越了最先进的基准模型. 主要原因有2点：

1）HRCE根据类的层次特点，采用分层表示的方法得到类的签名、属性名和方法签名的代码表征. 分层表示的方法能够学习类的语义信息并对类的层次结构建模以获取结构信息，同时缩短了编码器输入序列的长度，缓解了长程依赖问题.

2）HRCE将父类的签名和父类的摘要作为类的上下文，在解码过程中融合上下文语义信息，进一步提高有父类的类的摘要生成质量.

此外，本文还构建了一个包含上下文信息的类摘要语料库①，为后续的类摘要研究提供数据基础.

类具有丰富的上下文，而本文只使用了父类的相关信息，下一步的工作将从项目中提取不同类型的代码上下文，比较不同上下文对摘要生成的贡献. 此外，HRCE具备跨语言训练的能力，后续将尝试使用此模型为其他面向对象的语言生成摘要.

作者贡献声明：陈豪伶构思和实现模型，完成实验并撰写论文；虞慧群和范贵生指导实验方案并修改论文；李明辰指导实验和处理数据；黄子杰指导论文写作.

参　考　文　献

[1] Xia Xin, Bao Lingfeng, Lo D, et al. Measuring program comprehension: A large-scale field study with professionals[J]. IEEE Transactions on Software Engineering, 2017, 44(10): 951−976

[2] Haiduc S, Aponte J, Moreno L, et al. On the use of automated text summarization techniques for summarizing source code[C] //Proc of the 17th Working Conf on Reverse Engineering. Piscataway, NJ: IEEE, 2010: 35−44

[3] Haiduc S, Aponte J, Marcus A, et al. Supporting program comprehension with source code summarization[C] //Proc of the 32nd ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2010: 223−226

[4] Eddy B P, Robinson J A, Kraft N A, et al. Evaluating source code summarization techniques: Replication and expansion[C] //Proc of the 21st Int Conf on Program Comprehension. Piscataway, NJ: IEEE, 2013: 13−22

[5] McBurney P W, Liu C, McMillan C, et al. Improving topic model source code summarization[C] //Pro of the 22nd Int Conf on Program Comprehension. New York: ACM, 2014: 291−294

[6] Movshovitz-Attias D, Cohen W. Natural language models for predicting programming comments[C] //Proc of the 51st Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2013: 35−40

[7] Wong E, Yang Jinqiu, Tan Lin. Autocomment: Mining question and answer sites for automatic comment generation[C] //Proc of the 28th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2013: 562−567

[8] Sridhara G, Hill E, Muppaneni D, et al. Towards automatically generating summary comments for Java methods[C] //Proc of the 25th IEEE/ACM Int Conf on Automated Software Engineering. New York: ACM, 2010: 43−52

[9] Sridhara G, Pollock L, Vijay-Shanker K. Generating parameter comments and integrating with method summaries[C] //Proc of the 19th Int Conf on Program Comprehension. Piscataway, NJ: IEEE, 2011: 71−80

[10] Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods[C] //Proc of the 33rd Int Conf on Software Engineering. New York: ACM, 2011: 101−110

[11] Abid N J, Dragan N, Collard M L, et al. Using stereotypes in the automatic generation of natural language summaries for C++ methods[C] //Proc of Int Conf on Software Maintenance and Evolution. Piscataway, NJ: IEEE, 2015: 561−565

[12] Song Xiaotao, Sun Hailong, Wang Xu, et al. A survey of automatic generation of source code comments: Algorithms and techniques[J]. IEEE Access, 2019, 7: 111411−111428

[13] Moreno L, Aponte J, Sridhara G, et al. Automatic generation of natural language summaries for Java classes[C] //Proc of the 21st Int

① https://github.com/Haohaoi123/HRCE

Conf on Program Comprehension. Piscataway, NJ: IEEE, 2013: 23−32
[14] Li Mingchen, Yu Huiqun, Fan Guisheng, et al. ClassSum: A deep learning model for class-level code summarization[J]. Neural Computing and Applications, 2023, 35(4): 3373−3393
[15] Li Zheng, Wu Yonghao, Peng Bin, et al. Setransformer: A transformer-based code semantic parser for code comment generation[J]. IEEE Transactions on Reliability, 2022, 72(1): 258−273
[16] Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model[C] //Proc of the 54th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2016: 2073−2083
[17] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code[C] //Proc of the 33rd Int Conf on Machine Learning. New York: ACM, 2016: 2091−2100
[18] Hu Xing, Li Ge, Xia Xin, et al. Deep code comment generation [C] //Proc of the 26th Conf on Program Comprehension. New York: ACM, 2018: 200−210
[19] Leclair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines[C] //Proc of the 41st Int Conf on Software Engineering. New York: ACM, 2019: 795−806
[20] Hu Xing, Li Ge, Xia Xin, et al. Deep code comment generation with hybrid lexical and syntactical information[J]. Empirical Software Engineering, 2019, 25(3): 2179−2217
[21] Zhou Ziyi, Yu Huiqun, Fan Guisheng. Effective approaches to combining lexical and syntactical information for code summarization[J]. Software: Practice and Experience, 2020, 50(12): 2313−2336
[22] Ahmad W U, Chakraborty S, Ray B, et al. A transformer-based approach for source code summarization[J]. arXiv preprint, arXiv: 2005. 00653, 2020
[23] Alon U, Brody S, Levy O, et al. Code2Seq: Generating sequences from structured representations of code[C/OL] //Proc of the 7th Int Conf on Learning Representations. Amherst, MA: OpenReview. Net, 2019 [2023-11-30].https://openreview.net/forum?id=H1gKYo09tX
[24] Zhou Yu, Shen Juanjuan, Zhang Xiaoqing, et al. Automatic source code summarization with graph attention networks[J]. Journal of Systems and Software, 2022, 188: 111257
[25] Zhang Shikun, Xie Rui, Ye Wei, et al. Keyword-based source code summarization[J]. Journal of Computer Research and Development, 2020, 57(9): 1987−2000 (in Chinese)
(张世琨, 谢睿, 叶蔚, 等. 基于关键词的代码自动摘要 [J]. 计算机研究与发展, 2020, 57(9): 1987−2000)
[26] Zhou Ziyi, Yu Huiqun, Fan Guisheng, et al. Summarizing source code with hierarchical code representation[J]. Information and Software Technology, 2022, 143: 106761
[27] Wang Wenhua, Zhang Yuqun, Sui Yulei, et al. Reinforcement-learning-guided source code summarization using hierarchical attention[J]. IEEE Transactions on Software Engineering, 2020, 48(1): 102−119
[28] Lin Chen, Ouyang Zhichao, Zhuang Junqing, et al. Improving code summarization with block-wise abstract syntax tree splitting[C] //Proc of the 2021 IEEE/ACM 29th Int Conf on Program Comprehension. Los Alamitos, CA: IEEE Computer Society, 2021: 184−195
[29] Shi Ensheng, Wang Youlin, Du Lun, et al. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees[C] //Proc of the 2021 Conf on Empirical Methods in Natural Language Processing. Stroudsburg, PA: ACL, 2021: 4053−4062
[30] Mcburney P W, McMillan C. Automatic source code summarization of context for Java methods[J]. IEEE Transactions on Software Engineering, 2016, 42(2): 103−119
[31] Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of NL-queries for software maintenance and reuse [C] //Proc of the 31st Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2009: 232−242
[32] Yu Xiaohan, Huang Quzhe, Wang Zheng, et al. Towards context-aware code comment generation[C] //Proc of Findings of the Association for Computational Linguistics: EMNLP. Stroudsburg, PA: ACL, 2020: 3938−3947
[33] Wang Yanlin, Shi Ensheng, Du Lun, et al. Cocosum: Contextual code summarization with multi-relational graph neural network[J]. arXiv preprint, arXiv: 2107. 01933, 2021
[34] Haque S, Leclair A, Wu Lingfei, et al. Improved automatic summarization of subroutines via attention to file context[C] //Proc of the 17th Int Conf on Mining Software Repositories. New York: ACM, 2020: 300−310
[35] Bansal A, Haque S, McMillan C. Project-level encoding for neural source code summarization of subroutines[C] //Proc of the 29th Int Conf on Program Comprehension. Piscataway, NJ: IEEE, 2021: 253−264
[36] Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint, arXiv: 1810. 04805, 2018
[37] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners[J]. OpenAI Blog, 2019, 1(8): 9
[38] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[J]. Advances in Neural Information Processing Systems, 2020, 33: 1877−1901
[39] Ouyang Long, Wu J, Jiang Xu, et al. Training language models to follow instructions with human feedback[J]. Advances in Neural Information Processing Systems, 2022, 35: 27730−27744
[40] Feng Zhangyin, Guo Daya, Tang Duyu, et al. CodeBERT: A pre-trained model for programming and natural languages[J]. arXiv preprint, arXiv: 2002. 08155, 2020
[41] Wang Yue, Wang Weishi, Joty S, et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation[J]. arXiv preprint, arXiv: 2109. 00859, 2021
[42] Guo Daya, Ren Shuo, Lu Shuai, et al. GraphCodeBERT: Pre-training code representations with data flow[J]. arXiv preprint, arXiv: 2009. 08366, 2020
[43] OpenAI. GPT-4 technical report[J]. arXiv preprint, arXiv: 2303. 08774, 2023
[44] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[J]. arXiv preprint, arXiv: 2107. 03374, 2021
[45] Roziere B, Gehring J, Gloeckle F, et al. Code Llama: Open foundation models for code[J]. arXiv preprint, arXiv: 2308. 12950, 2023
[46] Luo Ziyang, Xu Can, Zhao Pu, et al. WizardCoder: Empowering code large language models with evol-Instruct[J]. arXiv preprint, arXiv: 2306. 08568, 2023
[47] Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with

neural networks[J]. Advances in Neural Information Processing Systems, 2014, 27: 3104−3112

[48] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural Computation, 1997, 9(8): 1735−1780

[49] Ge Fan, Kuang Li. Keywords guided method name generation [C] //Proc of the 29th Int Conf on Program Comprehension. Piscataway, NJ: IEEE, 2021: 196−206

[50] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C] // Proc of the Annual Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2017: 5998−6008

[51] Papineni K, Roukos S, Ward T, et al. BLEU: A method for automatic evaluation of machine translation[C] //Proc of the 40th Annual Meeting of the Association for Computational Linguistics. New York: ACM, 2002: 311−318

[52] Denkowski M, Lavie A. METEOR universal: Language specific translation evaluation for any target language[C] //Proc of the 9th Workshop on Statistical Machine Translation. Stroudsburg, PA: ACL 2014: 376−380

[53] Lin C Y. ROUGE: A package for automatic evaluation of summaries[C] // Proc of the Workshop on Text Summarization Branches Out. Stroudsburg, PA: ACL, 2004: 74−81

[54] Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint, arXiv: 1412. 6980, 2014

[55] Wu Yonghui, Schuster M, Chen Zhifeng, et al. Google's neural machine translation system: Bridging the gap between human and machine translation[J]. arXiv preprint, arXiv: 1609. 08144, 2014

[56] Ji Ziwei, Lee N, Frieske R, et al. Survey of hallucination in natural language generation[J]. ACM Computing Surveys, 2023, 55(12): 1−38



Chen Haoling, born in 1999. Master candidate. Student member of CCF. Her main research interests include automatic code summarization and program comprehension.

陈豪伶, 1999 年生. 硕士研究生. CCF 学生会员. 主要研究方向为代码自动摘要、程序理解.



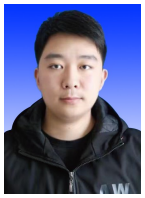
Yu Huiqun, born in 1967. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include software engineering, trusted computing, cloud computing, and formal methods.

虞慧群, 1967 年生. 博士, 教授, 博士生导师. CCF 高级会员. 主要研究方向为软件工程、可信计算、云计算、形式化方法.



Fan Guisheng, born in 1980. PhD, associate research fellow, PhD supervisor. Member of CCF. His main research interests include software engineering, service computing, and software architecture analysis techniques.

范贵生, 1980 年生. 博士, 副研究员, 博士生导师. CCF 会员. 主要研究方向为软件工程、服务计算、软件架构分析技术.



Li Mingchen, born in 1998. PhD candidate. His main research interests include automatic code summarization and program comprehension.

李明辰, 1998 年生. 博士研究生. 主要研究方向为代码自动摘要、程序理解.



Huang Zijie, born in 1994. PhD candidate. Student member of CCF. His main research interests include code smell, software quality assurance, program comprehension, and empirical software engineering.

黄子杰, 1994 年生. 博士研究生. CCF 学生会员. 主要研究方向为代码异味、软件质量保障、程序理解、实证软件工程.