计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(7):1−10 [doi: 10.15888/j.cnki.csa.009119]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于神经网络和信息检索的源代码注释生成①

沈　鑫[1,2], 周　宇[1,2]

[1](南京航空航天大学 计算机科学与技术学院, 南京 210016)
[2](南京航空航天大学 高安全系统的软件开发与验证技术工信部重点实验室, 南京 210016)
通信作者: 周　宇, E-mail: zhouyu@nuaa.edu.cn

摘　要: 源代码注释生成旨在为源代码生成精确的自然语言注释, 帮助开发者更好地理解和维护源代码. 传统的研究方法利用信息检索技术来生成源代码摘要, 从初始源代码选择相应的词或者改写相似代码段的摘要; 最近的研究采用机器翻译的方法, 选择编码器-解码器的神经网络模型生成代码段的摘要. 现有的注释生成方法主要存在两个问题: 一方面, 基于神经网络的方法对于代码段中出现的高频词更加友好, 但是往往会弱化低频词的处理; 另一方面, 编程语言是高度结构化的, 所以不能简单地将源代码作为序列化文本处理, 容易造成上下文结构信息丢失. 因此, 本文为了解决低频词问题提出了基于检索的神经机器翻译方法, 使用训练集中检索到的相似代码段来增强神经网络模型; 为了学习代码段的结构化语义信息, 本文提出结构化引导的 Transformer, 该模型通过注意力机制将代码结构信息进行编码. 经过实验, 结果证明该模型在低频词和结构化语义的处理上对比当下前沿的代码注释生成的深度学习模型具有显著的优势.

关键词: 代码注释生成; 抽象语法树; Transformer; 语义相似度; 自注意力机制; 程序理解

引用格式: 沈鑫,周宇.基于神经网络和信息检索的源代码注释生成.计算机系统应用,2023,32(7):1–10. http://www.c-s-a.org.cn/1003-3254/9119.html

Source Code Summarization Based on Neural Network and Information Retrieval

SHEN Xin[1,2], ZHOU Yu[1,2]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)
[2](Key Laboratory for Safety-critical Software Development and Verification, Ministry of Industry and Information Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract: Source code summarization is designed to automatically generate precise summarization for natural language, so as to help developers better understand and maintain source code. Traditional research methods generate source code summaries by using information retrieval techniques, which select corresponding words from the original source code or adapt summaries of similar code snippets; recent research adopts machine translation methods and generates summaries of code snippets by selecting the encoder-decoder neural network model. However, there are two main problems in existing summarization generation methods: on the one hand, the neural network-based method is more friendly to the high-frequency words appearing in the code snippets, but it tends to weaken the processing of low-frequency words; on the other hand, programming languages are highly structured, so source code cannot simply be treated as serialized text, or otherwise, it will lead to loss of contextual structure information. Therefore, in order to solve the problem of low-frequency words, a retrieval-based neural machine translation approach is proposed. Similar code snippets retrieved from the training set are used to enhance the neural network model. In addition, to learn the structured semantic information of

① 基金项目: 国家自然科学基金 (61972197); 江苏省自然科学基金 (BK20201292)
收稿时间: 2022-11-05; 修改时间: 2022-12-10; 采用时间: 2023-01-06; csa 在线出版时间: 2023-05-12
CNKI 网络首发时间: 2023-05-16

code snippets, this study proposes a structured-guided Transformer, which encodes structural information of codes through an attention mechanism. The experimental results show that the model has significant advantages over the deep learning model generated by the current cutting-edge code summarization in processing low-frequency words and structured semantics.

Key words: code summarization; abstract syntax tree (AST); Transformer; semantic similarity; self-attention mechanism; programming comprehension

随着软件规模和复杂度的不断提升, 导致开发者在复杂的软件开发周期中往往需要花费 90% 的精力在软件维护工作上 (如版本迭代和 bug 修复)[1–4]. 源代码注释以自然语言的形式, 在程序的理解和维护过程中发挥着至关重要的作用. 高质量的代码注释能够帮助开发人员快速有效的理解程序语义, 加速软件开发过程并保证程序的质量. 但是因为注释的书写总是需要花费开发人员一定的时间精力, 所以造成很多工程项目中代码注释缺失, 程序可读性较差. 作为软件工程领域基础及热点研究之一, 代码注释生成旨在为代码段自动生成简洁的自然语言描述, 从而促进程序理解和软件维护. 已有的研究采用机器翻译领域的神经网络模型为代码段构建语言模型, 同时考虑程序语言的强结构性, 学习代码结构信息和自然语言描述间的隐含关系. 但这些方法在提取和整合代码语法结构特征时采用无差别化统一处理的方式, 忽略了对信息的有效筛选, 这是其一; 另一方面, 基于神经网络的翻译方法对于代码段中高频词的处理更加友好, 但是弱化了对于低频词的处理. 因此, 对精确的源代码注释自动生成方法的研究势在必行.

注释生成领域开展的相关研究工作, 主要致力于为程序开发人员自动生成高质量的代码注释, 助力开发人员快速地进行程序语义理解和代码维护. 注释生成领域最初采用的神经网络的机器翻译模型仅将代码段作为序列化的文本进行处理, 忽略了程序语言强结构性的特征, 这也是程序语言区别于自然语言文本的主要方面, 因此导致了代码段上下文结构语义的丢失, 提取的特征并不能准确体现代码段的语义. 比如程序语言中大量出现的 if-else 分支语句, while 循环语句等, 其中上下文中可能多次出现同一标识符, 这时就要考虑它们在结构中的关系并对关系进行特征表示和编码. 对于训练数据集中的代码和注释对, 那些高频词汇在最后的注释生成中往往会有更高的概率出现, 但是低频词可能就只会有极小的概率, 而这些低频词有时候会使得生成的注释更准确更易于理解.

本文提出基于检索的方法对神经网络源代码注释生成任务进行优化. 信息检索技术在注释生成研究初期就已经广泛使用[5–7], 其主要是通过检索原始代码段中恰当的 term 来生成基于 term 的注释. 例如, 起初研究人员选择 LSI 和 VSM 方法从源代码中选择合适的 term 并生成源代码摘要[6,7], 后来又使用主题模型来进一步改进该方法. 此外, 除了使用这种基于 term 的注释生成方法, 还可以使用相似代码段的注释来辅助生成代码摘要. 由于代码复用在大规模的代码仓库中很常见[8,9], 而且代码克隆技术被用来从现有的代码仓库中检索相似的代码段, 考虑到现今代码的这种高度复用性, 相似代码段的注释也可以被改写来生成新的摘要.

具体地, 一方面, 本文首先基于训练集中的代码和注释训练一个融合注意力机制的编码器-解码器模型; 然后对于给定的一个测试代码段, 我们分别从语法和语义上检索训练集中最相似的两个代码段; 测试阶段对输入的代码段和两个检索到的代码段进行编码, 并且在解码时融合 3 个特征向量来预测摘要. 另一方面, 编程语言是高度结构化的, 因此不能简单地将源代码作为序列化文本处理, 容易造成上下文结构信息丢失. 为了学习代码段的结构化语义信息, 本文提出结构化引导的 Transformer, 该模型将序列化的代码输入和多视图的结构信息通过新提出的自注意力机制联合编码. 应用训练好的模型进行代码摘要生成测试, 实验结果证明该模型在低频词和结构化语义的处理上对比当下前沿的代码摘要生成的深度学习模型具有显著的优势, 增强注释生成的可读性和准确性. 此外, 本方法引入注意力机制, 增强神经网络模型训练的准确度, 利用模型对用户输入的目标代码段进行预测并生成精确的自然语言摘要来描述目标代码段的语义, 帮助开发人员更好地进行程序语义的理解.

本文的主要贡献如下.

1) 构建了一个结构化的表征源代码语义向量的 Transformer 深度学习训练模型.

2) 提出基于检索的方法来克服低频词问题, 增强注释生成的精确度.

3) 构建了一个高效的用于源代码注释生成的模型, 并在相关的真实数据集上进行验证, 结果证明本方法优于其他的注释生成模型.

1 基于神经网络和信息检索的源代码注释生成模型

基于信息检索 (IR) 的算法将源代码视为普通文本, 通过搜索源代码中的关键词、token 或者搜索相似代码段的注释来预测目标代码的注释, 其关注源代码的词汇语义, 但是忽略了源代码的结构信息、数据依赖和调用信息. 其次, 这类算法的有效性很大程度上取决于数据集中的相似代码, 如果数据集中没有出现类似的代码, 这类算法将无法为代码段生成准确的注释. 基于深度神经网络的注释生成算法, 大多使用基于 RNN 的编码器解码器框架并借助注意力机制将输入代码表示为向量, 通常将注释生成任务视为机器翻译任务. 注意力机制的作用在于调整源码中各 token 的权重分布, 当选择 CNN 构建注释生成系统时, 有必要结合注意力机制. 基于深度神经网络的注释生成算法是监督学习的一种, 因此这类算法需要高质量的数据集来训练和调整神经网络的参数, 以获取最终的生成模型. 本文提出一个基于检索的源代码注释自动生成模型, 其融合神经网络机器翻译模型和基于检索的方法, 旨在生成更加准确的源代码注释. 不同于已有的神经网络源代码机器翻译模型, 本方法在对检索代码库进行联合训练的时候不需要使用额外的编码器, 其包含一个基于注意力机制的编码器解码器模型, 两个基于相似度的代码段检索部分. 模型框架如图 1 所示.

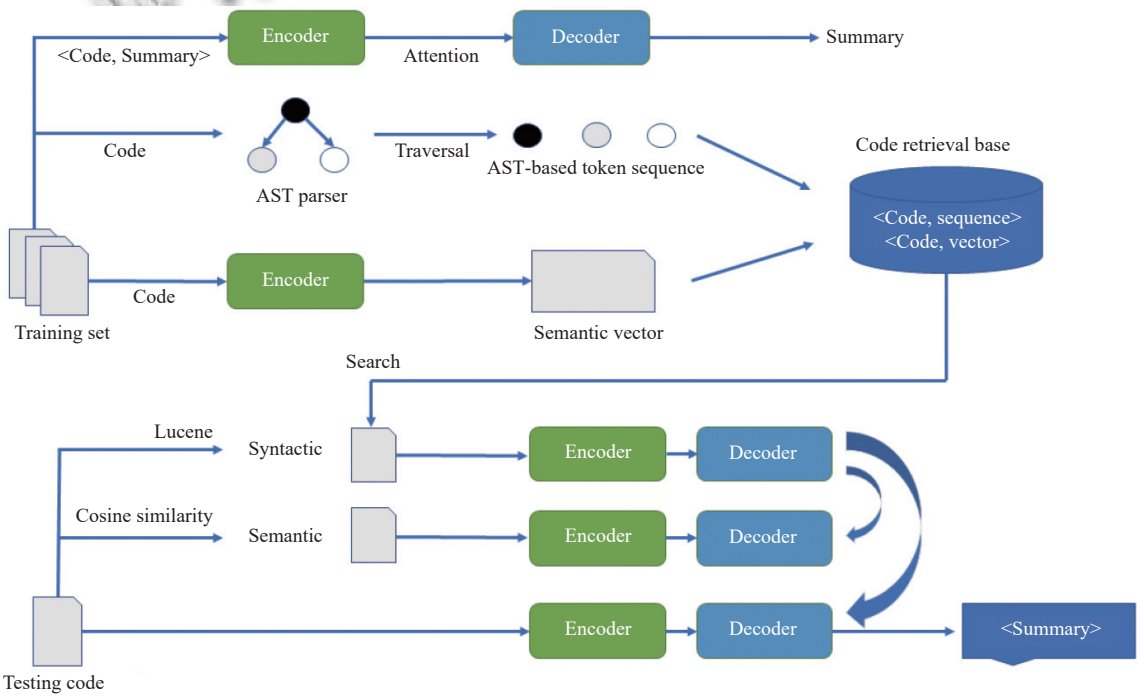

图 1 注释生成模型

1.1 基于注意力机制的 encoder-decoder 模型

类似于现有的神经网络源代码摘要生成方法[10–12], 我们创建并训练一个基于注意力机制的编码器-解码器模型. 对于编码器, 假设有一个包含一个单词序列的代码段 c, 首先使用一个嵌入层通过向量来初始化这些单词:

$$x_i = W_e^{\mathrm{T}} W_i,\ i \in [1,n] \tag{1}$$

其中, n 是代码段的长度, W_e 是嵌入矩阵, 然后我们使用 LSTM 单元来将向量序列 $X=x_1, x_2, \cdots, x_n$ 编码成隐藏层向量 $h_1, h_2, \cdots, h_t, \cdots, h_n$, 为了简化我们将 LSTM 单元表示如下:

$$h_t = \mathrm{LSTM}(x_t, h_{t-1}) \tag{2}$$

我们进一步地选择双向 LSTM 来捕获当前位置之

前和之后的语义信息. 当对代码段进行解码和生成第 i 个位置的注释单词时, 基于注意力的解码器首先通过隐藏层序列 $h_1, h_2, \cdots, h_t, \cdots, h_n$ 来计算隐含层状态序列的上下文向量 v_i, 计算公式如下:

$$v_i = \sum_{j=1}^{n} a_{ij} h_j \tag{3}$$

其中, a_{ij} 是 h_j 的注意力权重, 计算公式如下:

$$a_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{n} \exp\left(e_{ik}\right)} \tag{4}$$

$$e_{ij} = a\left(s_{j-1}, h_j\right) \tag{5}$$

其中, s_{j-1} 是解码器的最后一个隐藏层的状态, 式 (5) 中, 我们使用多层感知器 MLP 作为对齐模型 a, 在第 i 个时间步, 解码器的隐藏层状态 s_i 通过式 (6) 更新:

$$s_i = \text{LSTM}(s_{i-1}, y_{i-1}) \tag{6}$$

其中, y_{i-1} 是之前的输入向量, 为了将过去的对齐信息考虑在内, 使用输入反馈方法将 v_{i-1} 和 y_{i-1} 输入进行特征联合, 公式如下:

$$p(y_i|y_1, \cdots, y_{i-1}, c) = g(y_{i-1}, s_i, v_i) \tag{7}$$

其中, g 是生成器函数, 应用一个 MLP 层和 Softmax 函数. 我们采用 Beam Search 算法, 即在每一个时间步 t, 我们维护前 B 个最佳预测, B 就是 beam size. 训练这样一个基于注意力机制的编码器-解码器模型就是最小化损失函数:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \sum_{t=1}^{L} \log P\left(y_t^i | y_{<t}^i, c\right) \tag{8}$$

其中, θ 是训练参数, N 是训练实例的总数, L 是每一个目标序列的长度. 当使用代码段和对应的注释对训练后, 我们得到一个可以表示代码段的编码器, 和一个解码器, 通过最大化下一个词的条件概率来逐词地预测注释.

1.2 相似代码检索

根据现有的代码复用技术[13,14], 之前基于检索的源代码注释工作可以参照其相似代码段的注释[9,15]. 经验研究表明待预测注释的词序列 (包括低频词) 极有可能在相似代码注释中出现. 基于上述的编码器-解码器模型, 本文的方法可以结合训练集中相似代码段及其注释的知识更好地预测低频词. 为了尽可能多地包括有用的低频词汇, 我们打算从不同角度检索这样相似的代码段. 本工作中, 由于训练集非常大, 我们基于源代码的语法结构和嵌入语义设计出两个有效的源代码检索模块. 基于语法结构的源代码检索, 不同于纯文本, 源代码拥有自己特定的语法结构并且语法信息对于检索相似代码段至关重要. 之前的代码克隆检测任务已经在探索这样的语法知识, 不过需要代价高昂的基于树的匹配算法[16,17]. 最近关于源代码摘要的研究[10,18]同样考虑源代码的语法结构, 他们通过添加基于 AST 的 RNN 编码器来进行联合监督训练. 在我们的工作中, 因为训练集非常大且无监督方法是首选, 我们不进行训练, 而是基于 AST 的 token 序列来有效地计算代码段的语法相似性. 具体地, 给定一个来自测试集的输入代码片段和训练集中的任何一个, 我们将它们解析为两个 AST, 然后计算两个之间的相似度. 但是, 如果训练集非常大 (例如在我们的实验中规模 N 超过 50k), 基于树匹配算法 (如 tree edit distance[19]) 直接计算相似度会产生很多开销. 因为计算复杂度为 $O(N^2)$. 一个合适的折衷方案是通过树遍历将 AST 转换 token 序列. 因此, 我们将输入的代码片段以及所有的训练集解析为 AST, 并通过先序遍历进一步获得它们的序列表示. 基于这些序列, 我们使用现成且广泛使用的搜索引擎 Lucene 高效检索训练集中最相似的语法级别的代码片段. 对于语义级别的源代码检索, 最近基于神经网络的方法将源代码编码为语义向量, 已经展示了它们在捕获代码语义方面的优势. 但他们需要额外的计算代价来进行训练. 相比之下, 我们复用第 2.2 节中提到的已训练的编码器. 如上所述, 基于 Bi-LSTM 的编码器能够捕获源代码的序列信息并将语义嵌入到隐藏状态向量中.

给定一个代码片段 c, 我们通过 Bi-LSTM 编码器对其进行编码, 并得到一系列隐藏状态向量$[h_1, \cdots, h_n] \in R^{n \times 2k}$, 其中 k 是向量维度, n 是代码片段 c 的长度. 与文献 [16] 类似, 我们在向量序列上应用全局最大池化操作来获得语义表示$r_c \in R^{n \times 2k}$, 公式如下:

$$r_c = \left[\max\left(h_i^1\right), \cdots, \max\left(h_i^{2k}\right)\right],\ i = 1, \cdots, n \tag{9}$$

对于一个测试代码段 c_{test} 和训练集中任意一个代码段 c_i, 它们的余弦相似度计算如下:

$$\cos\left(\overrightarrow{r_{\text{test}}}, \overrightarrow{r_i}\right) = \frac{\overrightarrow{r_{\text{test}}} \cdot \overrightarrow{r_i}}{\left\|\overrightarrow{r_{\text{test}}}\right\| \left\|\overrightarrow{r_i}\right\|} \in [-1, 1], i \in [1, N] \tag{10}$$

其中, N 是训练集的大小. 最后, 获得最高分的代码段被选为训练集中最相似的结果.

基于两个源代码检索组件, 由于训练集通常很大,

我们可以在测试前离线准备一个代码检索库, 它存储源代码和基于 AST 的 token 序列对以及源代码和语义向量对. 给定一个新的在线测试代码段, 我们基于高效的 Lucene 搜索引擎检索语法级别上的相似代码片段, 并且通过快速计算向量的简单余弦相似度来搜索语义级别的相似代码段.

选择语法或语义层面最相似的代码片段 (比如 top-1) 并非易事. 一方面, 更多的 top-k (k >1) 相似的候选者可能是有噪声的, 尤其是当他们的相似度较低 时. 另一方面, 如果相似度阈值 T 被设置为过滤相似度相对较低的代码片段, 它同样也许会剔除有用的低频词. 因为它很难找到这样一个静态阈值来区分一个代码段是否包含有用的低频词. 我们发现选择没有阈值的最相似的代码片段 (如 k=1 和 T=0) 会在包含有用的低频词和噪声信息之间有一个较好的权衡.

1.3 基于检索的源代码注释生成

在训练我们的基于注意力的编码器-解码器模型并从训练集中检索两个最相似的代码片段之后, 我们将在线为测试集中的代码片段预测和生成注释语句. 直观地, 我们使用检索到的相似代码段 (用于低频词) 增强了基于注意力的编码器-解码器模型 (用于高频词). 一个简单的解决方案是通过相似度来增强相似代码片段中所有单词的概率, 但它可能包含干扰词. Zhang 等[20] 试图基于源词和目标词对齐使用翻译片段过滤这些干扰词. 与此同时, 除了代码片段的相似性, 我们还在解码时考虑了每个单词的条件概率以帮助剔除可能的通常条件概率较低的干扰词. 基于第 2.1 节中我们训练模型的编码器和解码器, 基于检索的神经摘要生成器同时编码每个测试代码片段及其两个最相似的代码段, 通过注意力机制获取它们的上下文向量并融合条件概率和相似性对其进行解码以预测摘要.

具体地, 对于一个测试代码片段 c_{test}, 我们从训练集中搜索并检索其两个在语法和语义上最相似的代码段. 然后, 我们通过训练的基于注意力的编码器-解码器模型 (第 2.1 节) 并行编码这 3 个代码片段. 基于得到的 3 个隐藏状态序列H_{test}, H_{syn}和H_{sem}用于c_{test}, c_{syn}和c_{sem}, 在解码过程中的每个时间步 t, 我们可以根据式 (3) 注意力权重以生成上下文向量, 然后计算条件概率并通过式 (7) 预测下一个单词. 为简化起见, 我们分别将这些条件概率表示为$P_{\text{test}}(y_t|y_{<t})$, $P_{\text{syn}}(y_t|y_{<t})$, $P_{\text{sem}}(y_t|y_{<t})$.

为了增强最初的基于注意力的编码器-解码器模型的预测性能, 我们利用检索到的代码片段并融合所有这 3 个条件概率. 一种直接的方法是将它们简单地相加. 然而, 当c_{test}和c_{syn} (或c_{sem}) 之间的相似度太低 (甚至c_{syn}是训练集中检索到的最相似的结果), $P_{\text{syn}}(y_t|y_{<t})$可能对预测正确的摘要词产生负面影响. 因此, 我们也应该考虑融合的相似性值. 直接重用代码检索中的相似性 (如式 (10)) 中的结果是不合理的, 因为我们使用不同方法并且从不同方面计算了这些相似度: 一个在语法层面, 另一个在语义层面.

为解决这个问题, 我们借助动态编程[21] 根据文本编辑距离$d(c_{\text{test}}, c_{\text{syn}})$和 $d(c_{\text{test}}, c_{\text{sem}})$对相似度进行归一化, 使它们具有可比性:

$$Sim(c_{\text{test}}, c_{\text{ret}}) = 1 - \frac{d(c_{\text{test}}, c_{\text{ret}})}{\max(|c_{\text{test}}|, |c_{\text{ret}}|)} \tag{11}$$

其中, c_{ret} 表示任何检索到的代码片段. 有了这些归一化的相似度, 我们结合条件概率得到最终的条件分布如下:

$$\begin{aligned} P_{\text{final}}(y_t|y_{<t}) =& P_{\text{test}}(y_t|y_{<t}) + \lambda \cdot Sim(c_{\text{test}}, c_{\text{ret}}) P_{\text{syn}}(y_t|y_{<t}) \\ &+ \lambda \cdot Sim(c_{\text{test}}, c_{\text{sem}}) P_{\text{sem}}(y_t|y_{<t}) \end{aligned} \tag{12}$$

其中, λ 是一个可以手动调整的超参数.

基于以上的最终条件分布, 我们可以依次预测下一个单词, 最后生成整个摘要语句, 它融合了检索到的代码片段中的知识.

2 实验评估

在这部分中, 我们进行实验来验证所提出方法的有效性, 并将它和几个软件工程领域以及自然语言处理领域的方法进行对比.

2.1 数据准备和实验设置

我们在两个公开的大规模数据集 (Python 和 Java) 上进行实验. Python 数据集由 Barone 等[22] 提供, 其中包含来自 GitHub 开源仓库的 Python 函数及其注释. 它使用文档字符串作为自然语言描述, 我们称之为注释. 这个数据集包括 108 726 个代码注释对, 并已被用于文献中的训练和评估. Java 数据集是由 Hu 等[12] 从开源仓库中收集的包含 Java 方法和注释的数据集, 其注释是从其 Javadoc 中提取的第 1 个句子. 原始数据集包括 API 序列摘要和代码摘要两部分, 我们选择后者, 将 69 708 个代码和注释对作为我们的 Java 数据集. 为了公平比较, 我们将 Python 数据集分为训练集、验证集和测试集, 占比分别为 60%、20% 和 20%, 并以

8:1:1 的比例拆分 Java 数据集以和基准[12,23] 保持相同的拆分比例. 为了使训练集和测试集不相交, 我们从测试集中删除了重复的样本. 我们设置了代码段和摘要的长度限制 (如 Python 数据集分别为 100 和 50, Java 数据集分别为 300 和 30), 因为这样的设置可以覆盖原始序列长度的大部分. 这两个数据集的统计数据如表 1 所示, 其中 MaxL、AvgL 和 UniT 是最大长度, 平均长度, 以及唯一 token 的总数. 我们同样考虑摘要中的低频词频率不超过 10 和 100 的情况. NumW 是低频词的数量, NumS 是至少包含一个低频词的摘要数量. 摘要中所有独特词中的低频词的百分比, 以及包含至少一个低频词的摘要的百分比也显示在括号中.

表 1 数据集统计信息

数据集	源代码长度(#words)			注释长度(#words)			词频≤10		词频≤100	
	MaxL	AvgL	UniT	MaxL	AvgL	UniT	NumW	NumS	NumW	NumS
Python	157 116	133	481 756	333	9.9	37 111	32 093 (86.5%)	46 481 (42.8%)	36 003 (97.0%)	87 636 (80.6%)
Java	4 842	99.9	230 336	670	17.1	35 535	30 342 (85.4%)	34 207 (41.1%)	34 223 (96.3%)	63 954 (77.3%)

为了标记源代码, 我们使用库 tokenize 和 javalang 分别获取 Python 和 Java 代码段的标记. 我们进一步通过蛇形命名法或者驼峰命名法将代码标记拆分为子标记以减少数据稀疏性. 同时, 我们使用 ast 库和 javalang 来获取它们对应的 AST. 为了标记化摘要, 我们利用 NLTK 工具包的标记化模块[23]. 此外, 我们限制了源代码和注释的最大词汇量为 50k, 因为太大的词汇量可能会导致更坏的性能. 词汇表外的单词被 UNK 替换. 为了更好地比较, 我们还将这种标记化应用于其他方法以避免潜在的影响.

我们基于开源系统 openNMT 实施本文方法. 在训练期间, 将嵌入大小设置为 256 并将 LSTM 中隐藏状态的维度设置为 512. 批处理大小设置为 32, 最大迭代次数为 100k. 我们采用广泛使用的 Adam[24] 作为具有学习率 0.001 的优化器用于训练我们的模型. 束尺寸设置为 5, 并且超参数 λ 设置为 3. 以上所有超参数都是基于验证集通过在一些替代方案中选择最佳参数确定. 所有实验均在一台 Ubuntu 20.04 服务器, 16 核 2.4 GHz CPU, 128 GB RAM 和配置 64 GB 内存的 RTX2080Ti GPU 上进行.

2.2 度量指标

与现有工作[10−12] 类似, 我们使用通用指标评估不同方法的性能, 包括 *BLEU*、*METEOR*、*ROUGE-L* 和 *CIDER*. 这些指标在机器翻译、文本摘要、和图像字幕等领域应用也很广泛.

给定生成的摘要 X 和真实值 Y, *BLEU* 通过计算 X 和 Y 之间的 n-gram 的重叠率和对较短的翻译假设应用简洁惩罚来度量 n-gram 精确度. *BLEU*-1/2/3/4 对应的分数分别是 unigram、2-gram、3-gram 和 4-gram. 计算 *BLEU-N* (N = 1, 2, 3, 4) 的公式是:

$$BLEU\text{-}N = BP \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n$$

其中, p_n 是候选句和参考句的 n-gram 匹配的精确度评分. *BP* 是过短惩罚, ω_n 是均匀权重, 取 $1/N$.

对于要比较的一对句子, *METEOR* 创建一个它们之间的单词对齐并通过以下方式计算相似度分数:

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R}$$

其中, P 和 R 是 unigram 精度和召回率, *frag* 是碎片分数. α、β 和 γ 是 3 个惩罚参数其默认值分别为 0.9、3.0 和 0.5.

ROUGE-L 广泛用于文本摘要并且提供基于最长公共子序列 (*LCS*) 的 F1-score. 假设 X 和 Y 的长度分别为 m 和 n, 则:

$$\begin{cases} P_{\text{lcs}} = \dfrac{LCS(X,Y)}{m} \\ R_{\text{lcs}} = \dfrac{LCS(X,Y)}{n} \\ F_{\text{lcs}} = \dfrac{(1+\beta^2) P_{\text{lcs}} \cdot R_{\text{lcs}}}{R_{\text{lcs}} + \beta^2 P_{\text{lcs}}} \end{cases}$$

其中, $\beta = P_{\text{lcs}}/R_{\text{lcs}}$, F_{lcs} 即 *ROUGE-L* 的值.

CIDER 通常用于衡量图像字幕的质量, 它通过计算每个 n-gram 的 TF-IDF 权重来考虑参考语句中 n-gram 的频率. n-gram 的 *CIDER* 分数使用平均余弦来计算候选句和参考句之间的相似度. 最终结果是通过组合不同的 n-gram 分数来计算的.

注意 *BLEU*、*METEOR* 和 *ROUGE-L* 的分数范围为 [0, 1], 通常以百分比记录. 但是 *CIDER* 是不在 0 和 1 之间, 因此以实际值记录.

2.3 对比模型

我们将本方法与现有的源代码摘要工作进行比较.

它们可以分为两组: 基于检索的和基于 NMT 的方法. 此外, 我们还比较了一个组合上述两种方法的效果最好的算法, 尽管它最初是为自然语言翻译提出. 本文使用这些方法的默认设置, 除非另有说明.

2.3.1 基于检索的方法

LSI 是一种文本检索技术, 用于分析文档的潜在含义或概念. 在文献 [6] 中用于选择代码片段中最重要的术语作为基于术语的摘要. 为了生成人类可读的摘要语句, 对于任何的测试代码片段, 我们使用 LSI 从训练集中检索最相似的语句并取其注释作为结果. 相似度的计算基于 LSI 减少的向量和余弦距离, 我们设置向量维度为 500.

VSM 是向量空间模型[25] 的缩写形式. 一个 VSM 的经典例子是 TF-IDF, 它被一些自动源代码摘要工作所采用[16]. 不同于 LSI, 它将代码片段索引到基于词频和文档频率的权重向量. 一旦获得向量表示, 我们通过余弦距离来检索最相似代码片段的摘要.

NNGen 是一个简单却有效的最近邻算法, 其为代码变更提供提交信息[17]. 在基于词袋和词频构建代码变更的向量之后, 它通过向量的余弦相似度和 *BLEU*-4 分数检索代码的最近邻. 然后 NNGen 直接复用最近邻的提交消息. 我们通过替换代码变更为代码段在我们的任务中复现这样的算法.

2.3.2 基于 NMT 的方法

CODE-NN 是第 1 个基于深度学习生成源代码摘要的神经网络方法[26]. 它是一个 LSTM 编码器-解码器神经网络, 将代码片段编码为具有注意机制的文本向量并生成摘要.

TL-CodeSum 是一种多编码器神经网络模型, 可将 API 序列与代码 token 序列一起编码, 并且使用转移的 API 知识从源代码生成摘要[12]. 它首先使用外部数据集训练 API 序列编码器. 然后学习到的 API 表示应用于源代码摘要任务以辅助摘要生成.

Hybrid-DRL 是一种先进的混合神经方法, 其使用混合代码表示和深度强化学习. 基础模型架构也是多编码器 NMT 通过编码 AST 和序列信息来学习结构和序列信息. 类似的合并 AST 的工作也由文献 [18,21] 完成, 但 Hybrid-DRL 进一步在解码阶段使用强化学习解决曝光误差问题并获得更好的性能.

此外, 正如提到的, 我们还包括一种方法, 即在 NLP 中使用检索到的翻译片段指导 NMT 翻译 (GRNMT)[20]. 翻译片段是检索到的目标句子的 *n*-gram, 也通过字对齐匹配输入语句和检索到的源语句中的常用词. 在解码过程中, GRNMT 借助翻译片段提高单词预测的准确性.

2.4 实验结果

表 2 展示了在我们的评估标准上不同注释生成方法的性能. 我们使用 Hybrid-DRL 提供的脚本计算这些指标的值. 对于这些指标, 值越大越好, 每一个指标的最优值标记为黑体. 从表 2 中, 我们可以看到基于检索的方法 LSI, VSM 和 NNGen 产生了很好的结果. 特别是, VSM 获得这 3 种技术中最好的性能. 有点令人困惑的是 LSI 似乎比 VSM 更差, 因为 LSI 考虑术语相关性, 通常来说比 VSM (TF-IDF) 在捕获纯文本的语义上表现更好[27]. 但如文献 [27] 中所述, 如果 TF-IDF 的维度 (如词汇量) 相比 LSI (即 500) 要大得多, 它可能会产生更好的性能. 在我们的案例中, 源代码语料库的词汇量很大, 在 TF-IDF 中超过 50k, 因为标识符任意命名导致源代码的唯一 token 要比纯文本中的多得多. Haiduc 等[7] 的工作中也观察到了这种现象. 将 VSM 与 NNGen 进行比较, VSM 更好, 因为它考虑了文档频率, 但 NNGen 没有.

表 2 源代码摘要方法比较

方法	Python				Java			
	BLEU-4 (%)	*METEOR* (%)	*ROUGE-L* (%)	*CIDER*	*BLEU*-4 (%)	*METEOR* (%)	*ROUGE-L* (%)	*CIDER*
LSI	17.6	17.2	40.2	1.982	17.4	14.4	34.8	1.803
VSM	19.3	19.0	42.7	2.216	19.0	15.4	36.6	1.983
NNGen	17.4	17.1	40.2	1.967	18.7	15.0	36.3	1.933
CODE-NN	8.1	13.4	35.1	1.229	6.3	9.1	28.9	0.978
TL-CodeSum	10.4	13.6	35.3	1.335	16.1	13.7	33.2	1.66
Hybrid-DRL	15.0	17.9	42.2	2.042	13.3	13.5	36.5	1.656
GRNMT	15.8	18.5	43.4	1.978	15.0	15.0	37.6	1.732
RBNMT	**20.8**	**20.9**	**47.2**	**2.453**	**20.6**	**17.2**	**41.7**	**2.196**

3 种基于 NMT 的方法中, CODE-NN, TL-CodeSum 和 Hybrid-DRL, CODE-NN 的表现比其他两个神经网络模型差, 因为它只依赖于 token 的嵌入 (如词法级别) 来理解代码段的语义. 相比之下, TL-CodeSum 更好, 因为它借助学习到的 API 序列知识捕获了更多的源代码语义. 我们还使用了原始 Java 数据集中提供的外部数据集[12] 为 Java 数据集预训练 API序列编码器, 但 Python 没有这样的外部数据集. 因此 TL-CodeSum 在 Java 数据集上比在 Python 上更有效. 对于 Hybrid-DRL, 它在 Python 数据集上比 TL-CodeSum性能更好, 因为包含了基于 AST 的源代码结构信息并且暴露偏差问题得到解决. 但在 Java 数据集上, 对于 *BLEU*、*METEOR* 和 *CIDER* 等指标, Hybrid-DRL 比 TL-CodeSum 差可能是因为来自原始 Java 数据集的外部数据集上的 API 序列知识主导性能. 此外, 我们发现 CODE-NN 和 TL-CodeSum 总体上性能差于基于检索的方法, 这不足为奇, 这是因为上文描述的低频词问题.

作为基于检索和基于 NMT 结合的方法, GRNMT 使用检索到的翻译片段指导一个简单的编码器解码器模型. 它在所有指标上都优于 CODE-NN, 在 *METEOR* 和 *ROUGE-L* 等一些指标上优于 TL-CodeSum 和 Hybrid-DRL, 这意味着检索信息实际上有助于注释生成. 此外, GRNMT 在 *ROUGE-L* 指标上优于所有基于检索的方法, 这表明编码器-解码器神经网络模型也有助于提高性能.

最后, 从表 2 中我们可以看到本方法实现了所有评估指标的最佳性能. 原因是我们在语法级别和语义级别分别检索最相似的代码片段作为我们的基于注意力机制的编码器解码器模型的附加上下文. 我们还通过相似度和条件概率的融合来预测下一个词. 与我们的方法相比, 与自然语言翻译任务不同, GRNMT 性能较差, 因为它很难精确匹配代码段中包含相应元素的摘要的 *n*-gram 并获得高质量的翻译片段. 由于大规模的训练集和测试阶段的在线代码检索, 我们的方法可能会比单个 NMT 模型花费更多时间来生成注释. 但是, 我们在实验的两个数据集上平均只需要 89 ms 为每个测试代码片段生成摘要, 因为高效的搜索引擎 Lucene 和余弦相似度的快速计算.

3 相关工作

源代码注释生成一直是软件工程领域研究的一个重要问题, 目前已有的源代码注释生成算法可以分成 2 类: 基于关键词模板的算法、基于神经网络的算法[13,28].

3.1 基于检索的注释生成算法

基于关键词模板的方法, 对于给定的代码注释对应的源码数据集, 和一段缺少注释的源代码, 其首先计算目标代码段和数据集中代码的相关性, 然后返回一段或者多段与目标代码段匹配的源码, 同时它们的注释会被用于生成目标代码的注释.

这类信息检索算法一般包括 VSM、LSI、LDA 以及其他一些类似代码克隆检测的技术. 基于 LSI 或 VSM 的检索算法通常将代码段表示为向量、矩阵或者数组, 向量中的每个元素都表示源代码中某个单词的权重, 计算 VSM 中项权重的方法主要是 TF-IDF. LSI 则利用 SVD 识别术语与概念主题之间的关联度, 同时提取文本的概念主题, 注释生成系统根据术语的权重或者查询文本和源码文本之间的文本相似度来确定该术语是否出现在源代码注释中, 具有较高相似度的术语代表了和代码片段或查询主题有更高的关联, 因此注释系统将推荐这些术语作为关键词来构建目标代码段的注释. Haiduc 等[7] 使用 VSM 和 LSI 分析源代码文本并生成类和方法的抽取式抽象摘要, 他们首先将源代码文档和程序包转换成语料库, 然后以矩阵形式表示包含在标识符中的术语以及语料库中源码和文档的注释. 当使用 VSM 从源码中生成抽取式摘要时, 源码文档中关联度最高的术语将根据权重被挑选出来, 同时使用 LSI 计算语料库中各术语向量和需要生成摘要的源代码向量的余弦相似度, 生成那些出现在语料库中, 但没有出现在目标代码摘要中的高相似度的术语. 通过这种方法, 他们分析了 Java 项目中的类和方法的源码, 并为其生成了简短准确的自然语言描述.

3.2 基于神经网络的注释生成算法

基于深度神经网络的注释生成算法主要包括两类: 基于 RNN 的算法和基于其他神经网络的算法. 当应用神经网络解决注释生成问题时, encoder-decoder 框架和注意力机制是至关重要的结构. encoder-decoder 框架又被称为 Seq2Seq 模型, 在该框架中, 编码器的作用是将源代码编码为固定大小的向量, 解码器负责解码源代码向量, 并对源代码的注释做出预测. 众多编码器和解码器结构的不同源于输入的形式和神经网络的类型, 一般而言, 编码器解码器的内部结构会选择 RNN、CNN 以及 RNN 的变体, 如 GRU 和 LSTM. 注意力机

制通常被引入 encoder-decoder 框架中, 负责动态分配较高的权重给那些编码器输入序列中关联程度更高的 token, 经验证可以显著提高长序列的性能.

基于 RNN 的注释生成算法, 一般根据编码器中使用的 RNN 数量, 将其分成两类: 基于单编码器的注释生成算法和基于多编码器的注释生成算法.

在基于单编码器的注释生成算法中, 编码器由一个 RNN 结构组成, Iyer 等[26] 提出一个基于 LSTM 的注释生成模型 CODE-NN, 该模型使用带有注意力机制的 LSTM 生成 C#代码段和 SQL 语句的自然语言描述, CODE-NN 基于从 Stack Overflow 中收集的包含标题代码段对的数据集训练, 在此期间, 注意力机制会关注源码中相关的重要 token, 完成对相关内容的选择, 而 LSTM 则提供所有单词的上下文. GRU 也用于注释生成模型, Zheng 等[2] 选择 GRU 作为编码器解码器框架的基本结构并结合全局的注意力机制, 他们将代码段中的嵌入标志 (如标识符) 作为可学习的先验权重以评估输入序列不同部分的重要性, 再根据出现顺序将代码段中的标识符分类后, 最后将源码中的 token 编码为嵌入式 token. 这样一来, 注意力机制就可以专注于程序中的这些特定部分, 即更好地理解代码结构, 最终提升了注释生成系统的准确性.

对于多编码器的注释生成算法中, 每一个编码器表示并抽取来自源代码的不同类型的信息, 继而带来准确率的提升. Hu 等[10] 将从自动 API 摘要任务中获得的知识迁移到注释生成任务中, 因为他们的注释生成系统配有两个编码器, API 编码器和源代码编码器, 在已学习的迁移的 API 知识的辅助下, RNN 解码器结合来自两个编码器中的注意力信息生成目标代码的摘要.

其他基于神经网络的注释生成算法, 如 CNN, 它的作用在于它可以提取输入序列的层次特性, Allamanis 等[21] 将 CNN 引入注意力机制中, 为源代码片段生成了简短的描述性摘要. 注释系统采用的是编码器解码器框架和注意力机制, 解码器的基本单元是 GRU. 卷积注意力模块将卷积应用在输入源代码片段上, 以检测应该关注的重要 token 序列, CNN 模型一 般不常用于为时序序列建模, 通常会选择 CNN 和其他辅助模型来解决时序序列问题.

4 结论与展望

本文结合神经网络和信息检索的方法来生成源代码注释, 给定一个测试代码段, 从训练集中检索相似的代码段并进行编码, 融合语法和语义信息辅助生成代码注释, 解决低频词问题, 并通过实验结果对比证实了基于检索的神经网络源代码注释生成方法的效果和潜力.

但是由于注释生成系统内部的复杂性和现有技术的局限性, 这仍然是一项挑战性的工作, 未来的研究工作应该继续探索神经网络模型和其他模型之间的协同作用, 更高效地表征源代码的结构信息和语义信息, 此外测试数据集和注释质量评估模型的统一和改进也是一个有价值的研究方向.

参考文献

1 Wan Y, Zhao Z, Yang M, *et al*. Improving automatic source code summarization via deep reinforcement learning. Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering. Montpellier: IEEE, 2018. 397–407.

2 Zheng WH, Zhou HY, Li M, *et al*. Code attention: Translating code to comments by exploiting domain features. arXiv:1709.07642, 2017.

3 Ko AJ, Myers BA, Coblenz MJ, *et al*. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering, 2006, 32(12): 971–987.

4 LaToza TD, Venolia G, DeLine R. Maintaining mental models: A study of developer work habits. Proceedings of the 28th International Conference on Software Engineering. Shanghai: ACM, 2006. 492–501.

5 Eddy BP, Robinson JA, Kraft NA, *et al*. Evaluating source code summarization techniques: Replication and expansion. Proceedings of the 21st International Conference on Program Comprehension (ICPC). San Francisco: IEEE, 2013. 13–22.

6 Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. Cape Town: IEEE, 2010. 223–226.

7 Haiduc S, Aponte J, Moreno L, *et al*. On the use of automated text summarization techniques for summarizing source code. Proceedings of the 17th Working Conference on Reverse Engineering. Beverly: IEEE, 2010. 35–44.

8 Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 2002, 28(7): 654–670. [doi: 10.1109/TSE.2002.1019480]

9 Wong E, Liu TY, Tan L. CloCom: Mining existing source code for automatic comment generation. Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). Montreal: IEEE, 2015. 380–389.
10 Hu X, Li G, Xia X, *et al*. Deep code comment generation. Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension. Gothenburg: IEEE, 2018. 200–210.
11 Hu X, Li G, Lo D, *et al*. Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering, 2019, 25(3): 2179–2217.
12 Hu X, Li G, Xia X, *et al*. Summarizing source code with transferred API knowledge. Proceedings of the 27th International Joint Conference on Artificial Intelligence. Stockholm: AAAI Press, 2018. 2269–2275.
13 彭斌, 李征, 刘勇, 等. 基于卷积神经网络的代码注释自动生成方法. 计算机科学, 2021, 48(12): 117–124. [doi: 10.11896/jsjkx.201100090]
14 Kim M, Sazawal V, Notkin D, *et al*. An empirical study of code clone genealogies. Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Lisbon: ACM, 2005. 187–196.
15 Wong E, Yang JQ, Tan L. Autocomment: Mining question and answer sites for automatic comment generation. Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). Silicon Valley: IEEE, 2013. 562–567.
16 Rush AM, Chopra S, Weston J. A neural attention model for abstractive sentence summarization. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Lisbon: ACL, 2015. 379–389.
17 Liu ZX, Xia X, Hassan AE, *et al*. Neural-machine-translation-based commit message generation: How far are we? Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering. Montpellier: IEEE, 2018. 373–384.
18 LeClair A, Jiang SY, McMillan C. A neural model for generating natural language summaries of program subroutines. Proceedings of the 41st International Conference on Software Engineering. Quebec: IEEE, 2019. 795–806.
19 Mou LL, Li G, Zhang L, *et al*. Convolutional neural networks over tree structures for programming language processing. Proceedings of the 30th AAAI Conference on Artificial Intelligence. Phoenix: AAAI Press, 2016. 1287–1293.
20 Zhang JY, Utiyama M, Sumita E, *et al*. Guiding neural machine translation with retrieved translation pieces. Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). New Orleans: Association for Computational Linguistics, 2018. 1325–1335.
21 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. Proceedings of the 33rd International Conference on Machine Learning. New York: PMLR, 2016. 2091–2100.
22 Barone AVM, Sennrich R. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. Proceedings of the 8th International Joint Conference on Natural Language Processing (Volume 2: Short Papers). Taipei: ACL, 2017. 314–319.
23 Bird S, Loper E. NLTK: The natural language toolkit. Proceedings of the 2004 ACL Interactive Poster and Demonstration Sessions. Barcelona: ACL, 2004. 214–217.
24 Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv:1412.6980, 2014.
25 Salton G, Wong A, Yang CS. A vector space model for automatic indexing. Communications of the ACM, 1975, 18(11): 613–620. [doi: 10.1145/361219.361220]
26 Iyer S, Konstas I, Cheung A, *et al*. Summarizing source code using a neural attention model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin: ACL, 2016. 2073–2083.
27 Zhang W, Yoshida T, Tang XJ. A comparative study of TF*IDF, LSI and multi-words for text classification. Expert Systems with Applications, 2011, 38(3): 2758–2765. [doi: 10.1016/j.eswa.2010.08.066]
28 Song XT, Sun HL, Wang X, *et al*. A survey of automatic generation of source code comments: Algorithms and techniques. IEEE Access, 2019, 7: 111411–111428. [doi: 10.1109/ACCESS.2019.2931579]

(校对责编: 孙君艳)