软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(7):2142−2165 [doi: 10.13328/j.cnki.jos.006270]

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

基于指针生成网络的代码注释自动生成模型*

牛长安[1,2], 葛季栋[1,2], 唐 泽[1,2], 李传艺[1,2], 周 宇[3], 骆 斌[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 软件学院,江苏 南京 210093)

[3](南京航空航天大学 计算机科学与技术学院,江苏 南京 211106)

通讯作者: 李传艺, E-mail: lcy@nju.edu.cn

摘 要: 代码注释在软件质量保障中发挥着重要的作用,它可以提升代码的可读性,使代码更易理解、重用和维护.但是出于各种各样的原因,有时开发者并没有添加必要的注释,使得在软件维护的过程中,往往需要花费大量的时间来理解代码,大大降低了软件维护的效率.近年来,多项工作利用机器学习技术自动生成代码注释,这些方法从代码中提取出语义和结构化信息后,输入序列到序列的神经网络模型生成相应的注释,均取得了不错的效果.然而,当前最好的代码注释生成模型 Hybrid-DeepCom 仍然存在两方面的不足.一是其在预处理时可能破坏代码结构导致不同实例的输入信息不一致,使得模型学习效果欠佳;二是由于序列到序列模型的限制,其无法在注释中生成词库之外的单词(out-of-vocabulary word,简称 OOV word).例如在源代码中出现次数极少的变量名、方法名等标识符通常都为 OOV 词,缺少了它们,注释将难以理解.为解决上述问题,提出了一种新的代码注释生成模型 CodePtr.一方面,通过添加完整的源代码编码器解决代码结构被破坏的问题;另一方面,引入指针生成网络(pointer-generator network)模块,在解码的每一步实现生成词和复制词两种模式的自动切换,特别是遇到在输入中出现次数极少的标识符时模型可以直接将其复制到输出中,以此解决无法生成 OOV 词的问题.最后,在大型数据集上通过实验对比了 CodePtr 和 Hybrid-DeepCom 模型,结果表明,当词库大小为 30 000 时,CodePtr 的各项翻译效果指标平均提升 6%,同时,处理 OOV 词的效果提升近 50%,充分说明了 CodePtr 模型的有效性.

关键词: 软件质量保障;代码注释生成;神经网络;out-of-vocabulary word;指针生成网络

中图法分类号: TP311

中文引用格式: 牛长安,葛季栋,唐泽,李传艺,周宇,骆斌.基于指针生成网络的代码注释自动生成模型.软件学报,2021,32(7):2142−2165. http://www.jos.org.cn/1000-9825/6270.htm

英文引用格式: Niu CA, Ge JD, Tang Z, Li CY, Zhou Y, Luo B. Automatic generation of source code comments model based on pointer-generator network. Ruan Jian Xue Bao/Journal of Software, 2021,32(7):2142−2165 (in Chinese). http://www.jos.org.cn/1000-9825/6270.htm

Automatic Generation of Source Code Comments Model Based on Pointer-generator Network

NIU Chang-An[1,2], GE Ji-Dong[1,2], TANG Ze[1,2], LI Chuan-Yi[1,2], ZHOU Yu[3], LUO Bin[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Software Institute, Nanjing University, Nanjing 210093, China)

* 基金项目: 国家自然科学基金(61802167, 61972197, 61802095); 江苏省自然科学基金(BK20201250); 华为-南京大学下一代程序设计创新实验室合作协议子项目

Foundation item: National Natural Science Foundation of China (61802167, 61972197, 61802095); Natural Science Foundation of Jiangsu Province, China (BK20201250); Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming

本文由“面向非确定性的软件质量保障方法与技术”专题特约编辑陈俊洁副教授、汤恩义副教授、何啸副教授以及马晓星教授推荐.

收稿时间: 2020-09-15; 修改时间: 2020-10-26; 采用时间: 2020-12-14; jos 在线出版时间: 2021-01-22

[3](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

Abstract: Code comments plays an important role in software quality assurance, which can improve the readability of source code and make it easier to understand, reuse, and maintain. However, for various reasons, sometimes developers do not add the necessary comments, which make developers always waste a lot of time understanding the source code and greatly reduces the efficiency of software maintenance. In recent years, lots of work using machine learning to automatically generate corresponding comments for the source code. These methods extract such information as code sequence and structure, and then utilize sequence to sequence (seq2seq) neural model to generate the corresponding comments, which have achieved sound results. However, Hybrid-DeepCom, the state-of-the-art code comment generation model, is still deficient in two aspects. The first is that it may break the code structure during preprocessing, resulting in inconsistent input information of different instances, making the model learning effect poor; the second is that due to the limitations of the seq2seq model, it is not able to generate out-of-vocabulary word (OOV word) in the comment. For example, variable names, method names, and other identifiers that appear very infrequently in the source code are usually OOV words, without them, comments would be difficult to be understood. In order to solve this problem, the automatic comment generation model named CodePtr is proposed in this study. On the one hand, a complete source code encoder is added to solve the problem of code structure being broken; on the other hand, the pointer-generator network module is introduced to realize the automatic switch between the generated word mode and the copy word mode in each step of decoding, especially when encountering the identifier with few times in the input, the model can directly copy it to the output, so as to solve the problem of not being able to generate OOV word. Finally, this study compares the CodePtr and Hybrid-DeepCom models through experiments on large data sets. The results show that when the size of the vocabulary is 30 000, CodePtr is increased by 6% on average in translation performance metrics, and the effect of OOV word processing is improved by nearly 50%, which fully demonstrates the effectiveness of CodePtr model.

Key words: software quality assurance; source code comments generation; neural network; out-of-vocabulary word; pointer-generator network

可读性是软件源代码的重要质量属性之一,可读性还与其他质量属性,如可重用性、可维护性、可靠性、复杂性和可移植性等度量标准有显著的关系,早期研究结果表明,代码注释提高了银行家算法的可读性[1].Tashtoush 等人[2]研究量化了众多代码特性对源代码可读性的影响,发现其中注释(comment)对增强源代码的可读性的积极影响排名第三,仅次于有意义的名称(meaningful name)和一致性(consistency).可见,代码注释在帮助开发者理解源代码的过程中发挥着重要的作用,可以使开发人员或维护人员更快地理解源代码,从而更方便、更高效地重用和维护该段代码.因此,在一段高质量的代码中,相对应的注释是必不可少的.

虽然已有越来越多的公司和机构呼吁或要求开发人员在编写代码时遵循代码规范,为每一段代码添加注释.但是由于源代码编写者缺乏意识、缺少约束或时间问题等各种各样的原因,仍然有绝大部分的代码缺少相应的注释.研究表明,在软件开发和维护过程中,开发人员仍然会花费约 59%的时间在程序的理解上[3].此外,Fluri 等人[4]通过对 ArgoUML、Azureus 和 JDT 核心这 3 个开源系统中代码和注释随时间共同演化关系的研究,发现代码和注释很少共同演变,新添加的代码几乎没有被注释过,这表明,即使在拥有注释的源代码中,有很多注释仍然是过时的.可以看出,当前的很多源代码没有对应的正确的注释,如何为这些源代码自动生成注释成为一项非常有意义的工作.近年来,研究人员在不断尝试通过机器学习等方法实现源代码注释的自动生成.

代码注释自动生成又称为代码总结或代码摘要(code summary),是指对源代码进行的文字描述,关键在于将用编程语言编写的源代码翻译为自然语言,这不仅需要描述源代码的功能,还要描述开发人员写这段代码时的设计意图[5].Hu 等人[6]将代码注释自动生成任务表述为一种机器翻译任务.早期的代码注释自动生成大多采用信息检索(information retrieval,简称 IR)算法[7−10],近年来,随着深度学习的发展,采用深度神经网络的 NLP 算法[6,11−17]取得了较为突出的成绩,其中大多模型都是基于一种称为 seq2seq[18]的 RNN 框架.seq2seq 模型广泛用于机器翻译、对话系统、自动文摘等 NLP 领域,通常由编码器(encoder)和解码器(decoder)两个 RNN 组成,通过编码器对输入序列提取出一个定长的特征向量,再将特征向量输入到解码器中进行解码,输出概率最大的句子.

虽然基于深度神经网络的 NLP 算法取得了较为突出的成绩,但是这类算法都有一个难以避免的缺点:即需要在训练模型时基于训练数据集构建词库,在训练数据输入到神经网络之前,所有的单词都会转换为其在词库

中的索引,但是词库无法覆盖验证、测试和预测中出现的所有单词,并且由于时间、内存等限制,词库的大小通常是固定的,这就需要删除词库中在训练数据集中不常出现的单词,因此,在训练、验证、测试和使用模型进行预测时,会存在单词不在词库中的情况,从而导致了 OOV(out-of-vocabulary)问题.解决方法是将所有的 OOV 词都映射到词库中的一个表示未知单词的特殊符号“⟨UNK⟩”,而这种方法会在模型的输入端和输出端都造成影响.输入端的代价就是在编码时,模型无法知道该 OOV 词的真正含义,因此丢失了该词的语义信息;输出端的代价就是在解码时,模型选择的是词库中概率最高的单词作为当前时刻的输出,因此 OOV 词也无法在生成的语句中出现,这两方面的代价在一定程度上降低了模型的性能.OOV 问题对于机器翻译等输入输出都是自然语言的任务影响较小,因为自然语言中很少出现不常用的偏僻词汇.但是,对于注释自动生成任务的影响较大,源代码中会出现各种各样的变量名、方法名等缩写、简写或由多个单词拼接而成的单词,称为标识符(identifier),标识符通常也会出现在对应的注释中,但是,由于它们一般只会出现在一个样本中,所以出现次数极少,很有可能在词库中被裁剪掉,结果就是这些标识符会在词库中被表示为“⟨UNK⟩”,从而丢失了语义信息.并且,在生成的注释中也无法出现这些标识符.上述 Tashtoush 等人[4]的研究结果表明,在许多代码特征中,对于提升源代码可读性的积极性最高的就是有意义的名称.另一项研究表明,软件代码中 70%均为标识符[19].同时,注释中出现的标识符可以使开发者更好地定位注释当前所指的对象,因此无论是在源代码中还是在注释中,将标识符表示为“⟨UNK⟩”都会极大地影响源代码和注释的可读性.

Hybrid-DeepCom[6]是当前代码注释自动生成领域最先进的方法,它使用了两个编码器,分别将优化过的源代码序列和对应的序列化的抽象语法树作为模型的输入,提取出源代码的语义信息和结构信息,再对提取到的信息进行解码,从而生成对应的注释,其性能相对于之前基于神经网络的方法(即,基于 Attention[20]的 seq2seq[18]模型、DeepCom[9]和 CODE-NN[11])有了明显的提升[6],其中,CODE-NN 的性能优于基于 IR 的方法[11].然而,Hybrid-DeepCom 仍然存在不足,为了解决输入端的 OOV 问题,其对源代码序列进行的优化破坏了代码的语法结构特征,导致两个输入之间的不一致,从而削弱了模型的部分性能.本文首先通过额外添加完整的源代码编码器解决了 Hybrid-DeepCom 存在的不一致问题,提出了无指针生成网络的 CodePtr(CodePtr-PGN),实验结果表明,提升后的模型在大部分情况下的表现优于 Hybrid-DeepCom.同时,我们进一步针对输出端的 OOV 问题,引入了 See 等人[21]提出的指针生成网络(pointer-generator network),指针生成网络在解码阶段的每一步都会计算出一个概率用于进行软切换,即确定当前步骤是取词库中条件概率最大的单词作为当前时刻的输出,还是从输入序列中直接将指针指向的单词作为当前时刻的输出.

在实验验证环节,本文将在 Hybrid-DeepCom 提供的数据集上对 Hybrid-DeepCom、无指针生成网络的 CodePtr(CodePtr-PGN)和 CodePtr 进行对比实验,由于 OOV 问题对模型的影响与词库的大小密切相关,实验将在不同的词库大小下分别进行.实验中将使用广泛用于机器学习、文本摘要等领域的评分标准 BLEU(sentence-level BLEU 和 corpus-level BLEU)和 METEOR 对结果进行评估.结果表明,CodePtr-PGN 在大部分情况下优于 Hybrid-DeepCom,而 CodePtr 在不同的词库大小下,均比 Hybrid-DeepCom 表现得更为出色,在大多数情况下优于 CodePtr-PGN.此外,本文通过对比 3 个模型生成的注释,发现引入指针生成网络后,CodePtr 生成的注释中“⟨UNK⟩”标签的数量明显少于引入指针生成网络之前的两个模型(CodePtr 的源代码开源已在 https://github.com/NougatCA/CodePtr 中给出).

本文的主要贡献可总结如下:

(1) 针对 Hybrid-DeepCom 对源代码的预处理可能破坏语法结构特征这一不足,通过额外增加一个源代码编码器解决了输入之间的不一致问题.

(2) 为了解决输出端的 OOV 问题,将指针生成网络引入 CodePtr,这也是代码注释自动生成领域首次引入指针生成网络.

(3) 在大型的数据集上进行了实验.对比并分析了 CodePtr-PGN、CodePtr 和 Hybrid-DeepCom 的性能表现,证明了额外添加的编码器消除了不一致性以及指针生成网络的有效性.

(4) 分析了词库大小对模型性能的影响,提出了今后的研究方向.

本文第 1 节介绍代码注释自动生成的相关工作.第 2 节介绍本文提出的模型.第 3 介绍实验相关设置,进行实验,并对实验结果进行对比和分析,验证模型的有效性.最后总结全文,并对下一步工作进行展望.

1 相关工作

早期的代码注释自动生成任务大多基于信息检索(information retrieval,简称 IR)算法,近年来,由于深度神经网络在 NLP 领域突出的表现,相关研究大部分使用了神经网络(neural network,简称 NN),并取得了较好的成绩,除此之外,还存在很多其他领域的方法,由于也是对源代码进行处理,因此同样值得参考.

1.1 基于IR的代码注释自动生成

基于 IR 的代码注释自动生成算法一般采用基于向量空间模型(vector space model,简称 VSM)、潜在语义索引(latent semantic indexing,简称 LSI)、潜在 Dirichlet 分配(latent Dirichlet allocation,简称 LDA)等技术或代码克隆检测(code clone detection)等相关技术[5].

Kuhn 等人[8]利用 LSI 技术在源代码中寻找语言信息,例如,标识符、名称和注释等非正式语义信息,称这些信息为语言主题,他们认为这些语言主题同样反映了代码的意图,并根据主题对源代码进行了聚类.Haiduc 等人[9]使用 VSM 和 LSI 分析源代码文本,提取并生成类和方法的自然语言摘要.在使用 VSM 为源代码生成提取摘要时,根据所选权重选择源代码文档中最相关的术语.同时,还利用 LSI 技术计算语料库中每个词的向量与要总结的源代码文档的向量之间的余弦相似度,然后生成不出现在要总结的方法或类中,但出现在语料库中的高度相似的词.他们分析了 Java 项目中的方法和类源代码,并为其生成了简短而准确的文本描述.Movshovitz-Attias 等人[22]使用主题模型、LDA 和 *n*-gram 模型来预测 Java 源代码的注释.他们利用训练过的 *n*-gram 模型、LDA 模型和 link-LDA 模型,计算文档主题的后验概率,并据此进一步推断注释标记的概率,最后将概率较高的注释标记作为源代码文件的注释.Wong 等人[10]应用代码克隆检测技术(使用最长公共子串)来发现相似的代码段,并使用一些代码段中的注释来描述其他相似的代码段.

1.2 基于神经网络的代码注释自动生成

CODE-NN[11]是首个完全由数据驱动的代码自动生成方法,使用了基于 Attention 机制的神经网络,据我们所知,它是首个专门用于代码注释自动生成任务的神经网络模型.Iyer 等人将 CODE-NN 与基于语法的机器翻译系统 MOSES[23]和文本摘要模型 SUM-NN[24]进行了对比,CODE-NN 的 METEOR 和 BLEU-4 得分均高于两者.Tree-to-Sequence[25]认为将顺序序列直接输入到编码器中并不能得到很好的结果,因为输入的结构信息同样重要,因此 Tree-to-Sequence 使用了语法树作为编码器的输入,提出了一种新的模型,称为 Tree-to-sequence attention NMT model.新模型基于树,可以获取序列中的语法信息.实验结果表明,引入结构语法信息对机器翻译有明显的积极作用.

对于编程语言这种结构性强的语言,结构语法信息显得更加重要,因此 DeepCom[9]将 Java 代码转换为对应的抽象语法树(abstract syntax tree,简称 AST),为了将树形的 AST 作为顺序序列输入到 DeepCom 中,Haiduc 等人针对 AST 提出了一种新型的遍历方法 SBT(structure-based traversal),将 SBT 的遍历结果作为编码器的输入.SBT 可以在保留树的结构信息的同时将树转换为顺序输入,不同于先序遍历等传统遍历方法无法从遍历结果准确推出 AST,SBT 的遍历结果唯一对应一个 AST.同时,Haiduc 等人还针对输入端的 OOV 问题,提出如果 AST 中命名为“类型_变量名”的叶节点是 OOV 词,则将其名称替换为只有“类型”,保留一定信息的同时大大减少了 OOV 词的比例,但是由于“类型”的种类太少,同时标识符的名称里也存在很多有用的语义信息,这种方法依然无法避免地丢失了很多信息.实验结果表明,DeepCom 相对于 CODE-NN 在百分制 BLEU 上的提升约为 13 分,Haiduc 等人还通过实验证明了 SBT 的性能要优于先序遍历.

TL-CodeSum[13]同时将代码序列和 API 序列作为输入,分别对应两个编码器,然后将两个编码器的输出结合起来作为解码器的输入.Wan 等人[14]将称为 Actor-Critic 的深度强化学习框架引入到了代码自动生成任务,提出了 RL+Hybrid2Seq,其使用两个编码器将代码序列和 AST 作为输入,针对两个编码器的输出使用了混合注意力

层(hybrid attention).Wei 等人[15]将 CG(code generation)和 CS(code summary)两个任务作为对偶任务同时训练,使用代码序列作为 CS 模型的输入,使用注释作为 CG 模型的输入,两个模型相互提升.通过与 CODE-NN、DeepCom、RL+Hybrid2Seq 等模型的比较,证实了该模型在 CS 方面的优势.以上两种方法,RL+Hybrid2Seq 和 Wei 等人[15]提出的方法只利用了代码序列的信息,同时也未针对 OOV 问题作任何处理.Zhou 等人[26]强调了上下文信息对生成代码注释的重要性,提出的 ContextCC 通过解析 AST 提取出包括方法和方法之间依赖性等的上下文信息.此外,还基于预定义的模板和规则筛选代码和注释信息,以构建高质量的数据集,实验结果表明,ContextCC 的性能优于 DeepCom 等方法.ContextCC 为了解决 OOV 问题,将标识符替换为其上下文信息,例如将声明方法名的标识符替换为“⟨METHODNAME⟩”,这样的处理方式与 DeepCom[8]相似,会丢失标识符的所有信息.

Huang 等人[27]提出了一种基于强化学习的方法 RL-BlockCom,同时提出了一种组合学习模型,该模型将强化学习的 actor-critic 算法与编码器-解码器算法相结合.Wang 等人[28]提出了一种新的使用分层注意力网络(hierarchical attention network)的代码摘要方法,该方法通过合并多个代码功能(包括类型增强的 AST 和程序控制流)来实现,并将其引入到了深度增强学习框架中.LeClair 等人[29]提出的 ast-attendgru 使用了代码序列和 SBT 序列作为两个编码器的输入,实验结果表明,ast-attendgru 的表现优于 CODE-NN 和 DeepCom,但是 ast-attendgru 将两个编码器进行了串联,在效果上相当于只有一个编码器,其输入是 SBT 序列和代码序列拼接而成的序列,我们认为这样做会使输入序列变得非常长,长序列的输入会降低 RNN 模型的性能.Hybrid-DeepCom[6]是 DeepCom 的改进版,将优化的源代码序列和 SBT 序列输入到两个编码器中,与 ast-attendgru 不同的是,Hybrid-DeepCom 将两个编码器进行了并联,这样就有效地缩短了输入序列的长度.为了进一步解决输入序列中 OOV 词语义信息丢失的问题,对于源代码序列,Hu 等人根据 Java 的驼峰命名法对标识符进行了拆分,在保留了 OOV 词语义的同时大大减少了 OOV 词的比例.实验结果表明,Hybrid-DeepCom 的表现优于 DeepCom.Hybrid-DeepCom 分解标识符虽然在一定程度上解决了输入端的 OOV 问题,但是我们认为,分解标识符会破坏源代码的结构,从而造成两个输入之间出现不一致,本文将针对这一不足,通过额外增加一个编码器来解决这种不一致问题.

1.3 其他方法

除了代码注释自动生成领域之外,还存在很多其他领域的方法,这些方法提出的对源代码的处理、特征提取等思想,可统称为编程语言处理(programming language processing),对代码注释自动生成工作同样具有很大的参考价值.

Marcus 等人[7]利用 LSI 技术对源代码和外部文档进行了分析,并从中提取了语义信息,将其用于自动识别从系统文档到程序源代码的可追溯性链接.虽然并不是针对代码注释自动生成任务提出的方法,但是 Marcus 等人提出的方法仍可用于该任务.刘亚姝等人[30]在对恶意代码特征提取中使用 LDA 获得汇编指令中潜在的“文档-主题”“主题-词”的分布,设计恶意代码检测的工作框架,实现恶意样本的分类问题,并提出了困惑度的评价方法,从而解决了 LDA 模型主题数目需要预先指定的问题.高原等人[31]在函数名称推荐模型中使用了 IR 技术,从函数库中检索与输入函数相似的函数,首先对 Java 源代码的 AST 序列进行鹅卵石编码(shingles encoding)[32],接着使用 Jaccard 系数对节点进行相似度度量,将相似节点封装成节点对用以进行语义分析,然后使用语义信息计算每句代码和函数名称中词条的相关性,提取出特征代码,依据特征代码与词条的对应关系选出用于组成函数名称的词条,然后按照规则对词条进行重组推荐给用户.

黄袁等人[33]在代码提交的关键类判定过程中,从代码耦合特征、代码修改特征以及提交类型特征这 3 个方面提取了 21 种特征,然后提出了关键类识别的机器学习算法 ICC,建立特征向量与分类之间的映射关系,然后依据训练好的模型,使用随机森林判定提交代码中的关键类.CommentAdviser[34]致力于推荐注释点,同时为新手程序员提供注释建议.CommentAdviser 从两个维度——代码自特征(intra-structural context feature)和代码间特征(inter-structural context feature)提取了每个代码行的 11 种类型的结构上下文特征,同时,为了获取当前代码行的上下文语义信息,还使用词嵌入技术从代码中提取语义上下文特征.将这两种特征向量相连就生成了当前代

码行的特征向量,然后采用信息增益去除掉不相关的特征维度,之后使用一系列机器学习算法训练模型.林泽琦等人[35]在文档语义搜索方法中,从源代码的 AST 中提取源代码的 4 种代码元素和 11 种结构依赖关系构成代码结构图,然后将文档的语义表示为代码结构图上的一个带权子图,接着计算两者之间的相似度以度量源代码和文档之间的语义相似度.

Mou 等人[36]在程序功能自动识别方法中使用基于树的卷积神经网络(TBCNN),在程序的 AST 上设计了卷积内核以捕获结构信息.TBCNN 是用于编程语言处理的通用架构,实验结果表明,TBCNN 优于很多用于 NLP 的神经网络模型.Cao 等人[37]在为源代码与文档之间建立可追溯性的链接中,提出了一种基于决策树的方法,用于识别软件文档中的显著代码元素和上下文代码元素.不仅关注文档中代码段的词法特征,还关注源代码的语义特征,量化了软件文档中显著代码元素和上下文代码元素的 4 种文档相关功能和类型(type)、关系(relation)、距离(distance)这 3 种代码相关功能.API 使用代码推荐方法 DeepAPIRec[38]由语句预测的深层学习模型(称为语句模型)和参数具体化统计模型(称为参数模型)组成,其中,语句模型使用基于树的 LSTM(tree-LSTM)从 AST 和源代码的控制流中提取代码树,从而学习 API 使用和程序结构模式.Tai 等人[39]提出的 Tree-LSTM 用来解决自然语言表现出来的句法特性,即可以自然地将单词组合成短语,在预测两个句子的语义相关性和情感分类两项任务上,Tree-LSTM 的表现优于所有基线,包括 LSTM.

2 CodePtr

本文首先针对 Hybrid-DeepCom 在试图解决输入端 OOV 问题时带来的两个输入之间不一致的问题,通过增加一个编码器消除了其中的不一致,提出了 CodePtr-PGN,然后,为了进一步解决输出端的 OOV 词问题,本文通过引入指针生成网络,提出了 CodePtr.

CodePtr 的整体框架图如图 3 所示,与 Hybrid-DeepCom 和无指针生成网络的 CodePtr 之间的差异将在下方给出说明.

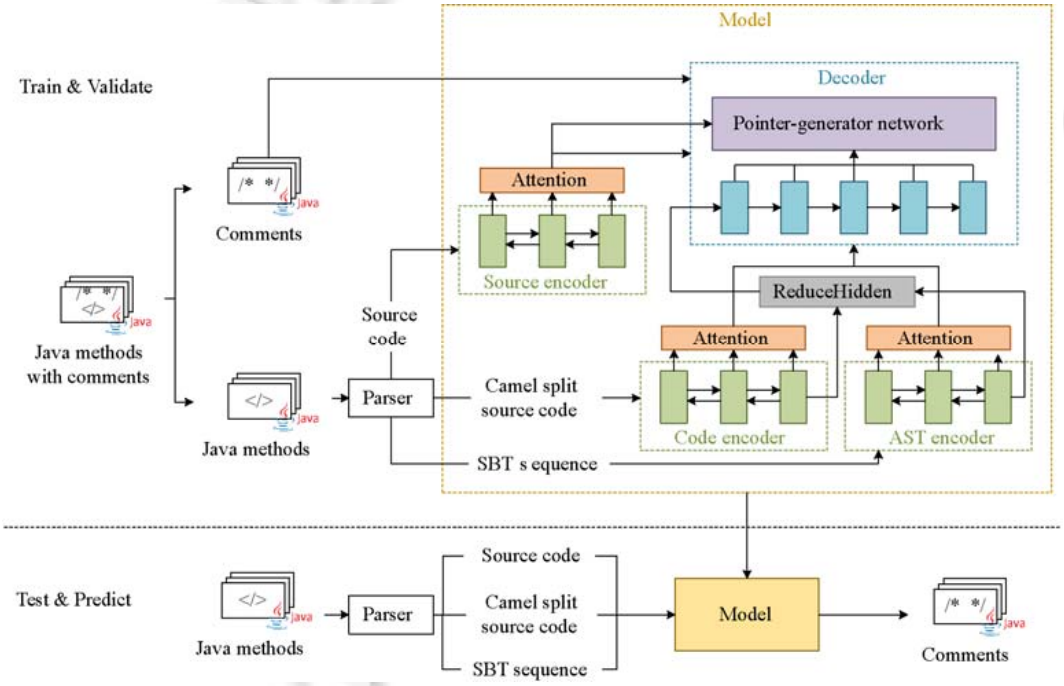


Fig.1 Overall framework of CodePtr

图 1 CodePtr 的总体结构

其中,将“Pointer-generator network”模块以及相关的连线删去后就得到了 CodePtr-PGN,CodePtr 中,Source

Encoder 只用于传递给 Attention 机制使其匹配学习到的 AST 信息,而对于引入了指针生成网络的 CodePtr, Source Encoder 的 Attention 权重还作为指针生成网络的输入.

从图中可以看出,CodePtr 大致上由以下几部分组成.

(1) 对 Java 源代码进行处理的 Parser;

(2) 3 个编码器:

a) Source Encoder,对未经驼峰式分解的源代码序列进行编码,是专门为指针生成网络添加的编码器;

b) Code Encoder,对经过驼峰式分解的源代码序列进行编码,目的是提取源代码序列的语义信息;

c) AST Encoder,对 Java 代码对应的 AST 的 SBT 序列进行编码,目的是提取结构信息;

(3) 维度映射层 ReduceHidden,使多个编码器的隐藏状态输出可以输入到解码器中;

(4) 加入了 Attention 机制和指针生成网络的解码器.

2.1 Parser

我们对 Java 方法的源代码进行了一系列处理,分别得到了输入到 Source Encoder、Code Encoder 和 AST Encoder 的输入序列.对相应的注释只进行了小写处理.

对于 Source Encoder,由于是专门用来学习指针生成的网络,保持源代码就是最好的选择.同时,为了减少冗余,只做了一些必要的处理.首先将源代码全部转换为小写,然后将具体的数字和字符串分别替换成了“⟨NUM⟩”和“⟨STR⟩”标签,由此就得到了 Source Encoder 的输入序列.

对于 Code Encoder,由于源代码中大量的标识符均为 OOV 词,同时,这些名称都是根据 Java 的命名规范——驼峰命名法进行命名的,由多个单词拼接而成,其中隐藏着大量的信息,这些有意义的名称对于理解源代码有着很大的帮助.因此,为了减少 OOV 比例,学习到更多的信息,我们对源代码中用驼峰命名法命名的名称进行了分解,例如将变量名“invocationTime”分解为“invocation”和“time”,将方法名“copyMutationMappingFile”分解为“copy”“mutation”“mapping”和“file”,将类名“RFFilter”分解为“rf”和“filter”.根据驼峰命名法进行分解后,训练集上只出现过 1 次的单词数量从 203 581 减少到了 8 226,未经裁剪的词库大小从 476 654 减少到了 43 480.同时,裁剪掉的 OOV 词数量占裁剪前词库的比例大为降低,下一节的实验中,在训练集大小为 432 674、词库大小为 30 000 的情况下,分解前后的 OOV 比例分别为 93.70%和 31.00%,而这一数字在词库大小为 50 000 的情况下为 89.51%和 0%,可见对标识符进行分解大大降低了词库的大小.对源代码进行分解后,再经过与 Source Encoder 相同的处理,就得到了 Code Encoder 的输入序列.值得注意的是,这里对标识符进行分解的操作是为了解决输入序列中 OOV 词丢失语义信息而提出的,与本节最开始的说明相同,这一步处理对本文解决的输出序列中生成 OOV 词并没有关系.

对于 AST Encoder,首先使用 Eclipse 的开发工具 JDT(http://www.eclipse.org/jdt/)将 Java 代码转换为对应的 AST,再使用文献[12]中提出的 SBT 遍历方法生成 AST 的遍历序列,由于已有另外的编码器对源代码中变量名称进行编码,因此,在 SBT 遍历结果中,只保留 AST 节点的类型信息,删掉了节点的名称.由此,就得到了 AST Encoder 的输入序列.

2.2 编码器

第 1.2 节最后介绍的 Hybrid-DeepCom 对源代码序列中的标识符进行了拆分,可以大大减少 OOV 比例,同时丰富源代码的语义信息,但是对于源代码这类结构性、语法性强的语言,如果将一个词拆分为多个词,很有可能破坏了源代码的结构.由于同时使用了 SBT 序列使模型提取到了隐藏的结构信息,分解后的源代码序列并不能完全对应 SBT 中的每个节点,我们认为这样使模型并不能完全利用学习到的结构信息.

换一个角度来看,就是在只有 Code Encoder 和 AST Encoder 的 Hybrid-DeepCom 中,本文认为 Code Encoder 的任务和作用有两个,一是提取源代码序列中的语义信息,另一个是与 AST Encoder 提取到的结构信息进行匹配.本文认为,第 1 个作用主要体现在 Code Encoder 的最终隐藏状态上,最终隐藏状态向量蕴含了提取到的隐藏特征;而第 2 个作用主要体现在解码时对 Code Encoder 输入的 Attention 权重上,依据提取到的结构信息选择对

源代码不同位置分配不同的 Attention 权重.这两个作用对输入是未经过分解的源代码来说是可行的,因为此时的输入与 AST Encoder 的输入 AST 是完全对应的.但是,Hybrid-DeepCom 为了丰富语义信息对标识符进行了分解,破坏了源代码的结构,使得 Code Encoder 和 AST Encoder 的输入不再完全对应,在增强了 Code Encoder 第 1 个作用的同时,不可避免地削弱了第 2 个作用.在一定程度上,这两个任务是互斥的,因为要丰富语义信息就需要对标识符进行分解,这样就破坏了语法结构信息;而保留语法结构就会将大量的标识符变成“⟨UNK⟩”标签,丢失了语义信息.因此,Code Encoder 无法同时在两个任务上达到最佳性能,只能在训练中找到两者中间的平衡点.

为了同时发挥两个任务的最优性能,CodePtr 引入了一个新的编码器将这两个作用分开,将第 2 个作用通过 Source Encoder 实现,Source Encoder 将分解前的源代码序列作为输入,与 AST Encoder 的输入完全对应,使结构信息发挥更大的作用,同时也使 Code Encoder 更好、更专注地发挥提取语义信息的作用.由于第 2 个作用主要通过 Attention 权重实现,并且,由于分解后的语义信息包含分解前的语义信息,为了避免冗余以及过多“⟨UNK⟩”标签的干扰,我们将不把 Source Encoder 的最终隐藏状态传给解码器,而只在解码阶段使用其对应的 Attention 权重.

因此,CodePtr 有 3 个编码器:输入为源代码序列的源代码编码器(source encoder)、输入为分解后源代码的代码编码器(code encoder)和输入为 SBT 序列的 AST 编码器(ast encoder).

Source Encoder 用于对未分解的源代码进行编码,一方面是由于上述原因,另一方面是为指针生成网络提供 Attention 权重,因为分解过后的源代码序列中标识符已被分解,无法提供整个单词的 Attention 权重.对于长度为 T_s 未分解的源代码序列 $X^s=x_1^s,x_2^s,...,x_{T_s}^s$,编码器将其编码为隐藏状态,对于时刻 t,Source Encoder 接受当前时刻的未分解的源代码序列输入 x_t^s,将上一时刻的隐藏状态 h_{t-1}^s 更新为 h_t^s,即

$$h_t^s=f_s(x_t^s,h_{t-1}^s) \tag{1}$$

其中,f_s 在本文中采用了 GRU 单元,由此可以得到 Source Encoder 的所有隐藏状态 $output_s=[h_1^s,h_2^s,...,h_{T_s}^s]$.

分解过后的源代码序列相对于分解前的代码序列而言,大大降低了 OOV 词的比例,可以使编码器学习到更丰富的语义信息,因此,Code Encoder 十分必要,对于长度为 T_c 未分解的源代码序列 $X^c=x_1^c,x_2^c,...,x_{T_c}^c$,在 t 时刻,Code Encoder 接受分解后的代码序列输入 x_t^c,将上一时刻的隐藏状态 h_{t-1}^c 更新为 h_t^c,公式如下:

$$h_t^c=f_c(x_t^c,h_{t-1}^c) \tag{2}$$

其中,f_c 同样采用了 GRU 单元,得到 Code Encoder 的隐藏状态 $output_c=[h_1^c,h_2^c,...,h_{T_c}^c]$.

对于结构性强的语言,源代码的结构信息同样重要,Java 代码对应的 AST 可以反映源代码的结构信息,而 Hu 等人在文献[12]中提出的 SBT 遍历方法可以无损地将 AST 树形结构转换为序列.因此,AST Encoder 将源代码对应的 AST 的 SBT 序列作为输入,使 CodePtr 可以提取到源代码的结构信息.对于长度为 T_a 的 SBT 序列 $X^a=x_1^a,x_2^a,...,x_{T_a}^a$,AST Encoder 在 t 时刻接受 SBT 序列输入 x_a^t,将上一时刻的隐藏状态 h_{t-1}^a 更新为 h_t^a,即

$$h_t^a=f_a(x_t^a,h_{t-1}^a) \tag{3}$$

其中,f_a 为 GRU 单元,得到 AST Encoder 的隐藏状态 $output_a=[h_1^a,h_2^a,...,h_{T_a}^a]$.

3 个编码器中,Source Encoder 是专门为指针生成网络添加的编码器,Code Encoder 和 AST Encoder 可以同时学习到源代码的语义信息和结构信息.对于多编码器的最终隐藏状态,CodePtr 在传入解码器之前使用了一个维度映射层 ReduceHidden.

2.3 维度映射层ReduceHidden

在经典的 seq2seq 网络[18]中,编码器最后时刻的隐藏状态作为解码器的初始隐藏状态,这样可以使编码器提取到的信息传递给解码器.而对于多编码器的情况,由于编码器和解码器的隐藏维度大小一般是相同的,因此 CodePtr 使用了维度映射层 ReduceHidden,使多个输出的隐藏状态可以输入到解码器中.

上一小节中提到了我们将不会把 Source Encoder 的最终隐藏状态向量传递给解码器,原因有两个:一是因为 Source Encoder 的作用是在解码时匹配 AST 信息,并且在指针生成网络开启时为其提供源代码的 Attention 权重,这都是通过解码时提供 Attention 权重来实现的;二是因为最终隐藏状态向量中蕴含的是编码器提取到的

信息,对于 Source Encoder 和 Code Encoder 来说,提取到的都是源代码的语义信息,而且由于 Code Encoder 使用分解后的源代码序列,因此 Code Encoder 提取到的语义信息包括了 Source Encoder 提取到的语义信息,并且要比其丰富.综合以上原因,为了减少冗余,降低干扰,我们将不传递 Source Encoder 的最终隐藏状态向量.

ReduceHidden 接受 Code Encoder 和 AST Encoder 的隐藏状态 $h_{T_c}^c$ 和 $h_{T_a}^a$, 分别简写为 h_c 和 h_a,并通过一个线性层进行维度映射,由此得到输入到解码器的初始状态 s_0,具体公式如下:

$$s_0 = \text{ReLU}(W_h[h_c; h_a] + b_h) \tag{4}$$

其中,W_h 和 b_h 为可学习参数,ReLU(rectified linear unit)为线性整流函数,这里用作激活函数.Reduce Hidden 的输出作为解码器的初始隐藏状态,可以同时将 Code Encoder 提取到的语义序列信息和 AST Encoder 提取到的结构信息传递给解码器.

2.4 解码器

CodePtr 使用的是加入了 Attention 机制和指针生成网络的解码器,在介绍 Attention 和指针生成网络前,首先介绍没有添加 Attention 机制和指针生成网络的解码器.

ReduceHidden 的输出作为解码器的初始状态输入到解码器中,解码器在每个时刻将上一时刻目标序列的单词向量和上一时刻的隐藏状态作为输入,计算出当前时刻的输出,即

$$s_t = f_{dec}(\hat{y}_{t-1}, s_{t-1}) \tag{5}$$

其中, f_{dec} 在本文中使用了 GRU 单元, $\hat{y}_{t-1}$ 表示目标序列在上一时刻的单词向量,s_{t-1} 为解码器上一时刻的隐藏状态,s_t 表示 t 时刻解码器的隐藏状态.编码器在每一时刻,将当前时刻的隐藏状态 s_t 通过线性变换和 softmax 层后,计算出词库中每一个单词的概率,其中概率最高的单词即为当前时刻的输出,即

$$p(y_t \mid y < t, s_t) = \text{softmax}(W_s s_t + b_s) \tag{6}$$

其中, $p(y_t|y<t,s_t)$表示在 t 时刻之前所有输出单词和当前隐藏状态的条件下,输出词库中每个词的概率,W_s 和 b_s 为可学习参数,softmax 表示 softmax 激活函数.解码器每一步都会挑选当前时刻概率最高的单词作为当前时刻的输出,直到在某个时刻输出停止符号才停止.

2.5 Attention机制

编码器在对输入序列进行编码为固定大小的状态向量时本质上是一个有损压缩的过程,输入序列越长,由压缩造成的损失就越大,较长的时间序列使得状态向量丢失了很多信息.同时,将不固定长度的输入序列压缩为一个固定长度的向量,也丢失了输入序列的序列信息.最后,由于解码器得到的仅仅是一个固定长度的状态向量,其无法直接关注到输入序列中的各种细节信息.出于以上考虑,CodePtr 引入了 Attention 机制.

NLP 领域的 Attention 机制首先由 Bahdanau 等人[40]提出,之后,Luong 等人[20]对其进行了优化,去掉了部分冗余,简化了计算,提升了速度.Luong 等人提出了两种类型的 Attention 机制:Global Attention 和 Local Attention,分别可以使 Attention 机制在整个输入序列中进行,或者在输入序列的局部进行,本文采用了前者.

由于我们的模型有 3 个编码器,需要使用 3 次 Attention 机制,这里,我们先将 Source Encoder 作为例子.在解码器生成注释的过程中,Attention 机制首先将上一时刻的解码器状态与 Source Encoder 中所有的隐藏状态进行对比和对齐,得到一个表示 t 时刻对 Source Encoder 的注意力分配权重、长度等于 Source Encoder 输入序列长度的向量 a_t^s, 即

$$a_t^s(s_{t-1}) = \text{align}(s_{t-1}, output_s) \tag{7}$$

其中,s_{t-1} 表示解码器在上一时刻的状态,$output_s$ 为 Source Encoder 在编码时每一步的隐藏状态,align 为对齐函数,对于长度为输入序列长度的向量 a_t^s, 其元素值 a_{ti}^s 表示在当前时刻解码器对 Source Encoder 的第 i 个隐藏状态的注意力权重,计算方法为

$$a_{ti}^s = align(s_{t-1}, output_{si}) = \frac{\exp(score(s_{t-1}, output_{si}))}{\sum_{j=1}^{T_x} \exp(score(s_{t-1}, output_{sj}))} \tag{8}$$

其中,T_x表示 Source Encoder 的输入序列的长度,$output_{si}$、$output_{sj}$分别表示 Source Encoder 在 i 和 j 时刻的隐藏状态,score 函数的计算方法使用了文献[20]中的 concat 方法,即

$$score(s_{t-1}, output_{si}) = W_a[s_{t-1}^T; output_{si}] \tag{9}$$

$[s_{t-1}^T; output_{si}]$表示将 s_{t-1}^T 和 $output_{si}$ 进行拼接,W_a 为可学习参数.将式(7)中的 $output_s$ 分别替换为 $output_c$ 和 $output_a$,可以分别得到对于 Code Encoder 和 AST Encoder 的输入序列的注意力权重矩阵 a_t^c 和 a_t^a.然后将对应的编码器的全部隐藏状态的注意力权重加权平均,得到各自的上下文向量,然后再将这些上下文向量相加,得到最终的上下文向量 c_t,即

$$c_t = \sum_{i=1}^{T_s} a_{ti}^s h_i^s + \sum_{i=1}^{T_c} a_{ti}^c h_i^c + \sum_{i=1}^{T_a} a_{ti}^a h_i^a \tag{10}$$

其中,T_s、T_c、T_a分别表示 Source Encoder、Code Encoder 和 AST Encoder 的输入序列的长度,h_i^s、h_i^c、h_i^a分别为 3 个编码器在 i 时刻的隐藏状态.通过将 c_t与当前解码器的状态 s_t拼接起来,再经过维度映射和激活之后,就得到了 Attention 机制下的解码器状态 $\hat{s}_t$.

$$\hat{s}_t = \tanh(W_c[c_t; s_t]) \tag{11}$$

其中,W_c为可学习参数,$[c_t;s_t]$为两个向量的拼接,tanh 为双曲正切激活函数.得到 $\hat{s}_t$ 后,将公式(6)中的 s_t 替换为 $\hat{s}_t$,即可得到 Attention 机制下每个单词的输出概率.

引入 Attention 机制后的 seq2seq 模型虽然可以使解码器关注到输入序列中的每个元素,但是仍然没有解决输出序列中的 OOV 问题,图 2 所示为在只有 seq2seq 和 Attention 机制下进行代码注释自动生成任务的示意图,这里,为了表示简便,只显示 Source Encoder 一个编码器.可以看到,解码器当前正在生成“the”之后的输出,由于词库大小的限制,变量名“whitelabelwallet”(已预处理为小写)在词库中被裁剪掉了,因此生成注释时,对于目标序列中“whitelabelwallet”单词出现的地方,尽管 Attention 机制对于源代码中的单词“whitelabelwallet”分配的注意力权重很大,但是由于其是 OOV 词,模型只能生成“〈UNK〉”标签,这样就极大地降低了注释的准确性和可读性,针对这一问题,我们引入了指针生成网络.

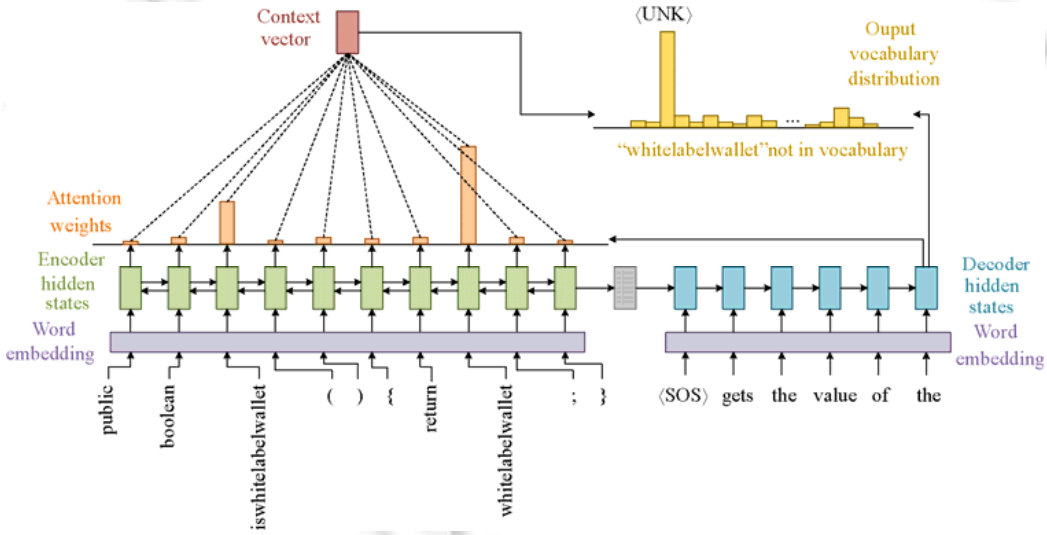


Fig.2 Seq2seq model with Attention is still unable to generate OOV words

图 2 引入了 Attention 机制的 seq2seq 模型仍然无法生成 OOV 词

2.6 指针生成网络

为了解决生成式文本摘要(abstractive text summarization)任务中,模型无法从源文本中复制内容的缺点,See

等人[21]提出了指针生成网络(pointer-generator network).指针生成网络是 Ptr-Net[41]和传统 seq2seq 模型的混合,Ptr-Net 由 Vinyals 等人[41]提出,利用 Attention 机制得到的每一步对于输入序列元素的注意力分配,生成一个指针指向注意力分配最高的元素,直接从输入序列中复制对应的元素作为输出,指针生成网络将其与生成式模型 seq2seq 进行结合,相当于在解码的每一步都有两种模式:一种是根据解码器的输出从词库中选择某个单词作为当前的输出,一种是根据 Attention 机制得到的指针直接从输入中复制作为当前的输出.指针生成网络通过一个可学习的概率 p_{gen} 在两个模式之间进行软切换(soft switch),t 时刻的 p_{gen} 由当前时刻的 Attention 机制得出的上下文向量 c_t、输入解码器的隐藏状态 s_t 和解码器的输入 y_t 共同计算而出,具体公式如下:

$$p_{\text{gen}} = \text{sigmoid}(W_c c_t + W_s s_t + W_y y_t + b_{\text{gen}}) \tag{12}$$

其中,W_c、W_s、W_y 和 b_{gen} 均为权重矩阵或偏置矩阵,均为可学习的参数,sigmoid 为 sigmoid 激活函数.

为了使模型可以知道哪些单词可能需要被复制,我们对于每一个样本,定义扩展词库(extended vocabulary)为输出端的词库再加上一些 OOV 词,在本文中,最可能被复制的单词就是在代码和注释中都出现的标识符,因此我们要求这些 OOV 词同时出现在源代码和注释中,并且都不在两者的词库中,扩展词库可以使这些共同的 OOV 词和“⟨UNK⟩”标签区别开来,以让模型学习这种复制关系,解码阶段的每个时刻,对于扩展词库中的每一个词 w,其条件概率 $P(w)$为

$$P(w) = p_{\text{gen}} P_{vocab}(w) + (1 - p_{\text{gen}}) \sum_{i:w_i = w} a_{ti} \tag{13}$$

其中,$P_{vocab}(w)$表示 w 在注释词库中的概率,即公式(6)得出的概率,a_{ti} 表示在当前 t 时刻对于输入序列中第 i 个单词的 Attention 权重,i 为在词库中 w 单词的下标,大多数情况下只有一个值.注意到,如果 w 是扩展出的 OOV 词,那么 $P_{vocab}(w)$则为 0,而 w 如果在注释词库中但没有出现在源代码词库中,那么 $\sum_{i:w_i = w} a_{ti}$ 为 0.由此得出的 $P(w)$代替公式(6)得出的 $P_{vocab}(w)$,成为解码器在每一步进行输出选择的依据.

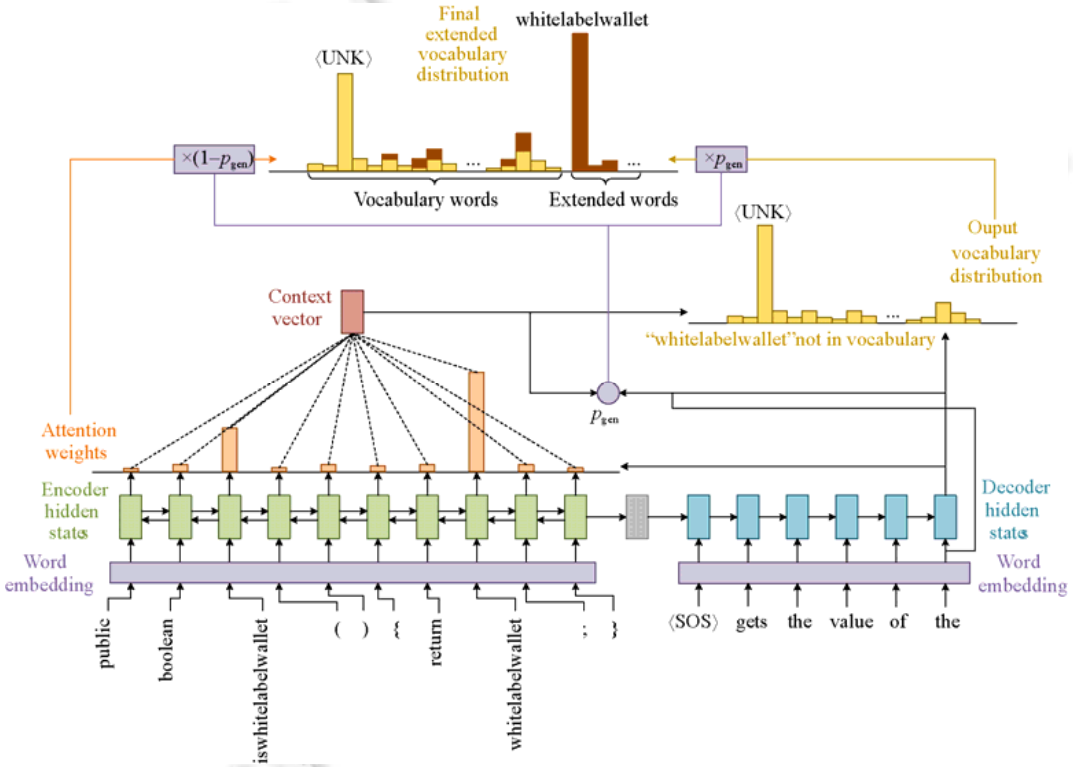


Fig.3 The model in Fig.2 is able to generate OOV words after introduced pointer-generator network

图 3 在图 2 中的模型引入指针生成网络后,可以生成 OOV 词

图 3 显示了将指针生成网络的模型加入到图 2 中的模型后,在相同时刻下进行解码的示意图,从图中可以看出,加入指针生成网络之后,引入了最终扩展词库概率分布(final extended vocabulary distribution),最终扩展词库概率分布由两部分组成,一部分是在输出词库中的词(vocabulary word),这些词的概率乘以 p_{gen} 后得到最终概率分布,如果其也在输入序列中出现的话,还会加上输入序列中该词的注意力权重乘以$(1-p_{\text{gen}})$的概率;另一部分是扩展出的同时出现在输入输出序列中的 OOV 词(extended word),由于它们不在输出词库中,最终的概率分布仅为对应的注意力权重乘以$(1-p_{\text{gen}})$.因此,尽管图 2 所示的输出词库的概率分布与图 1 中相同,但在指针生成网络将其与注意力权重加权相加后得到的最终扩展词库概率分布中,OOV 词“whitelabelwallet”的概率最高,因此,当前时刻的输出是“whitelabelwallet”而不是“⟨UNK⟩”.由此看出,可以在解码阶段生成 OOV 词是指针生成网络的一个主要的优势.

最后,通过将 $P_{vocab}(w)$替换为 $P(w)$,可以使指针生成网络相关参数参与到损失函数的计算中,从而在之后的优化步骤中对相关参数进行优化.

3 实 验

本节将加入指针生成网络前后的 CodePtr 和 Hybrid-DeepCom 在大型数据集上进行了实验,我们将主要关注以下 3 个研究问题.

- RQ1:加入 Source Encoder 是否可以提升模型的性能?
- RQ2:指针生成网络能否解决输出端的 OOV 问题?
- RQ3:词库大小对模型性能的影响有什么规律?

我们首先在第 3.1 节介绍实验所使用的数据集以及对数据进行预处理,第 3.2 节介绍实验的模型、环境以及参数等设置,第 3.3 节介绍评估方法,第 3.4 节给出实验结果并对研究问题进行回答.

3.1 数据集

本文采用了 Hu 等人[6]创建的数据集(https://github.com/xing-hu/EMSE-DeepCom),该数据集由 Hu 等人从 GitHub 上收集到的创建于 2015 年~2016 年的 Java 项目组成,为了保证数据集的质量,过滤掉了 10 星以下的项目.然后从项目中提取到 Java 方法以及对应的 Javadoc,他们依据 Javadoc 的编写规范,提取了其中第 1 句作为该 Java 方法的注释.接着过滤掉了 getter、setter 以及有@Test 等标注的测试方法,因为 Hu 等人认为并没有必要为这些方法添加注释.同时,为了避免冗余,过滤掉了重载方法,因为重载方法往往都实现了相似的功能.在至此得到的数据集中,我们分别随机选取了 20 000 份样本作为验证集和测试集,剩下的 445 812 份样本作为训练集.

最后,在输入模型前,为了使 RNN 发挥出更好的性能,需要限制数据集的最大长度,由于源代码在分解前的长度一定小于等于分解后的长度,因此过滤掉分解后源代码序列长度超过 200 的样本,并且过滤掉注释长度大于 30 的样本.同时,由于在对结果进行评分时,使用的 BLEU 评分标准需要使用 4-gram,因此也过滤掉了注释长度小于 4 的样本.

3.2 实验设置

为了对比 CodePtr 和 Hybrid-DeepCom 以及指针生成网络的有效性,我们对 Hybrid-DeepCom、无指针生成网络的 CodePtr(CodePtr-PGN)和 CodePtr 进行了实验,3 个模型的异同在第 2 节开始既已进行了说明.

考虑到 OOV 问题与词库大小密切相关,我们分别将 3 个模型在词库大小为 5 000、10 000、30 000、50 000 下进行了实验.实验中,最大训练轮数(epoch)为 30 轮,模型将在每 5 000 个 batch 和每轮结束时进行验证,同时使用了早停法(early stop),最低验证损失如果连续 20 次没有被刷新,则终止训练.验证集上的损失用来表示模型的泛化能力,训练结束后,验证损失最低的模型为最佳模型,最佳模型将在测试集上进行测试,测试时使用 corpus-level BLEU、sentence-level BLEU 和 METEOR 这 3 个常用于机器翻译、文本摘要等领域的评分标准进行评估,评估方法将在第 3.3 节进行介绍.

模型使用开源的 Python 机器学习框架 PyTorch 实现,实验在 NVIDIA GeForce RTX 2080 Ti 上进行,模型相

关的超参数如下所示.

- 编码器使用维度为 256 的双向 GRU,解码器使用相同维度的单向 GRU,词嵌入的维度同样为 256;
- 批处理大小(batch size)为 32,每轮都对数据集进行了打乱处理;
- 优化器为 Adam[42],学习率为 0.001,虽然 Adam 已经实现自动调整学习率,但是 Loshchilov 等人[43]的实验结果表明,在 Adam 上使用学习率衰减(learning rate decay)可以改善 Adam 的表现,因此使用了学习率衰减;
- 为了防止曝光误差(exposure bias),训练时使用了 teacher forcing,比率为 0.5;
- 使用了梯度裁剪,梯度最大值设置为 5;
- 使用了集束搜索(beam search),宽度(beam width)为 5.

3.3 评估方法

我们使用了机器翻译领域常用的评价指标作为本次实验的评估方法,这些指标在代码注释自动生成领域也经常被广泛使用,分别为 corpus-level BLEU、sentence-level BLEU 和 METEOR.

3.3.1 BLEU

BLEU(bilingual evaluation understudy,双语评估替补),这项指标最初是 Papineni 等人[44]为翻译而发明的,但渐渐广泛用于评估各种 NLP 任务生成的文本.BLEU 可以评估模型生成的句子,也称为候选(candidate)和参考(reference)句子之间的相似度,取值范围在 0 和 1 之间,1 表示两者完全匹配,0 表示完全不匹配.BLEU 作为 NLP 领域的评价指标,具有计算复杂度小、与语言无关等优点,因此被广泛采用.

BLEU 通过计算 candidate 和 reference 的 *n*-gram 模型,然后统计其匹配的个数来计算得出.例如,对于目标句子“sets the gravity for the drawer”和候选句子“sets the gravity for the pixels”.使用 1-gram 时,查看候选句子中的每个单词是否出现在目标句子中,可以看到,候选句子中的“sets”“the”“gravity”“for”和“the”这 5 个词出现在了目标句子中,而候选句子的长度为 6,因此,1-gram 的精度分数为 5/6.再以 4-gram 举例,如图 4 所示.

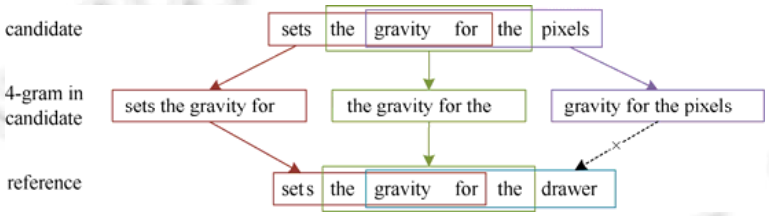


Fig.4 Example for 4-gram precision score for BLEU

图 4 BLEU 的 4-gram 精度分数的例子

从图中可以看出,候选句子共可分为 3 个 4-gram 词组,其中 2 个在目标句子中出现,因此,该例子的 4-gram 精度分数为 2/3.然而,这种计算方法存在一种缺陷,例如,若将上述例子中的候选句子变为“the the the the the”,则根据上述计算方法,可以得到 1-gram 的精度为 1,这显然是不合理的.因此,BLEU 在对 *n*-gram 匹配次数计数时设置了上限,即

$$Count_{clip}=\min(Count,\text{Max}\_Ref\_Count) \tag{14}$$

其中,*Count* 是 *n*-gram 在候选句子中出现的次数,Max_*Ref*_*Count* 是该 *n*-gram 在目标句子中最大的出现次数,该例子中“the”在目标句子中出现了 2 次,因此“the”的最大计数为 2,可以得出该例子的 1-gram 精度分数为 2/5.形式化地说,计算 *n*-gram 的精度分数 P_n 的公式为

$$P_n=\frac{\sum_{n-gram\in candidate}Count_{clip}(n-gram)}{\sum_{n-gram'\in candidate}Count(n-gram')} \tag{15}$$

其中,P_n 中的 *n* 表示 *n*-gram 下的精度,而 candidate 表示候选句子.因此可以看出,该式的分母为 candidate 中 *n*-gram 的个数,分子为 candidate 中所有 *n*-gram 出现在 reference 中的个数.

BLEU 还存在一个问题,就是 *n*-gram 的匹配度可能会随着句子长度的变短而变好,因此 BLEU 在最后的评分结果中引入了长度惩罚因子(brevity penalty,简称 BP),对于长度为 c 的候选句子和有效长度为 r 的目标句子,惩罚因子 BP 的计算方法如下.

$$\mathrm{BP}=\begin{cases}1, & \text{if } c>r\\ \mathrm{e}^{\left(1-\frac{r}{c}\right)}, & \text{if } c\leqslant r\end{cases} \tag{16}$$

因此,最终的 BLEU 分数为

$$BLEU=BP\cdot\exp\left(\sum_{n=1}^{N}W_n\log p_n\right) \tag{17}$$

其中,exp 为指数函数,p_n 为公式(15)得到的结果,对于 *n*-gram,W_n=1/*n*,对于 BLEU-4,N=4.

本次实验使用了 corpus-level BLEU 和 sentence-level BLEU,前者则是在整个语料库中进行的 BLEU 评分,也就是在测试完成后,对所有生成的注释进行评分;后者是对每一对候选句子和参考句子进行 BLEU 评分,最后得出平均值.评分通常使用累积 *n*-gram 得分,即最终的得分由各 *n*-gram 加权平均得到,本次实验采用的 BLEU-4 为 1-gram、2-gram、3-gram、4-gram 下的得分进行加权平均,权重为 BLEU-4 所设定的默认值,均为 0.25.

3.3.2 METEOR

METEOR 由 Denkowski 等人[45]提出,基于单精度的加权调和平均数和单字召回率,其目的是解决一些 BLEU 标准中固有的缺陷,其最大的特点就是引入了同义词匹配,本实验的同义词匹配基于 wordnet(https://wordnet.princeton.edu/).

METEOR 首先找到候选句子和参考句子之间的一个最优匹配,然后计算准确率(precision)P 和召回率(recall)R.本次实验中,准确率为候选句子中存在匹配的单词数量除以候选句子的总单词数量,召回率为候选句子中匹配的单词数量除以参考句子的总单词数量,得到 P 和 R 后,接着计算两者的参数化调和平均值 F_{mean},公式为

$$F_{mean}=\frac{P\cdot R}{\alpha\cdot P+(1-\alpha)R} \tag{18}$$

其中,α为可设定的参数.单词之外,为了考虑短语的匹配程度,METEOR 引入了 chunks,表示在参考句子中相邻的若干个单词,其在候选句子中匹配的单词也相邻的短语,使用 c 表示 chunks 的数量,m 表示匹配的单词数量,则惩罚因子 Pen 为

$$Pen=\gamma\left(\frac{c}{m}\right)^{\beta} \tag{19}$$

其中,β和γ为可设定的参数.可以看出,当候选句子和参考句子没有短语匹配,均为单个词匹配时,c=m,惩罚因子最大,当 chunk 变长时,c 变小,惩罚因子变小,因此 METEOR 得分为

$$Score=(1-Pen)\cdot F_{mean} \tag{20}$$

实验中我们对每一个生成的注释进行 METEOR 评估,最终取平均值作为 METEOR 得分.参数方面,根据文献[45]中基于数据实验得到的参数设定,我们取α=0.9,β=3,γ=0.5.

3.4 实验结果及分析

本节首先在第 3.4.1 节直观罗列不同方案最终的实验结果,从词库规模和评估方式两个维度对最终的结果进行对比,并且对比了不同方案的参数规模和训练开销,第 3.4.2 节~第 3.4.4 节深入分析了实验结果并分别回答了 3 个研究问题.

3.4.1 实验结果

模型在不同词库大小下的运行时间大约为 20~25 小时,训练阶段均是通过早停法结束.最终得分见表 1.表中,C-BLEU 表示 corpus-level BLEU,S-BLEU 表示 sentence-level BLEU,得分采用百分制.

Table 1 Scores of each model under different vocabulary size

表 1 不同词库大小下各模型的得分

词库大小	评估方法	Hybrid-DeepCom	CodePtr-PGN	CodePtr
5 000	C-BLEU	37.14	36.68	**40.11**
	S-BLEU	35.47	35.18	**39.24**
	METEOR	48.06	48.01	**50.86**
10 000	C-BLEU	40.27	40.93	**43.14**
	S-BLEU	38.98	39.49	**42.41**
	METEOR	50.85	51.51	**53.33**
30 000	C-BLEU	41.73	44.30	**47.94**
	S-BLEU	40.92	43.72	**47.76**
	METEOR	52.21	54.30	**57.00**
50 000	C-BLEU	40.45	**42.74**	42.02
	S-BLEU	39.49	**42.27**	41.50
	METEOR	51.42	**53.37**	52.84

可以看到,在词库大小为 5 000~30 000 时,CodePtr 的得分明显高于前两个模型,在词库大小为 50 000 时与 CodePtr-PGN 接近,两者均高于 Hybrid-DeepCom.CodePtr-PGN 与 Hybrid-DeepCom 相比,两者在词库大小为 5 000 时得分相近,在 10 000~50 000 时前者高于后者.

Hybrid-DeepCom、无指针生成网络的 CodePtr 和 CodePtr 的参数数量分别为 25.64M、34.24M 和 39.37M.在模型的运行开销方面,在词库大小为 30 000 的情况下,Hybrid-DeepCom 在第 18 轮由于达到了早停条件而终止训练,总运行时长约为 23.5 小时;CodePtr-PGN 在第 17 轮终止了训练,总运行时长约为 27.5 小时;CodePtr 则在第 17 轮终止了训练,总运行时长约为 26.5 小时.我们将在下面的小节中回答所提出的 3 个研究问题.

3.4.2 RQ1:加入 Source Encoder 是否可以提升模型的性能?

之前我们提出了 Hybrid-DeepCom 的缺点,就是 Code Encoder 的输入和 AST Encoder 的输入之间存在不一致,我们认为这是 Code Encoder 在 Hybrid-DeepCom 中担任两个角色造成的.其两个角色中,第 1 个是提取代码序列特征的角色,需要通过 RNN 提取出代码序列的语义信息;第 2 个是提供注意力的角色,Attention 机制需要通过给 Code Encoder 的输入的不同权重来决定当前的输出.我们认为,Hybrid-DeepCom 通过分解标识符增强了其作为第 1 个角色的作用,但却破坏了代码结构,削弱了其作为第 2 个角色的作用,这两个作用在这里是无法同时提升的.因此我们在此基础上再增加一个输入,即分解标识符之前的代码序列,分担 Code Encoder 的第 2 个角色,提出了 CodePtr-PGN,将 Hybrid-DeepCom 中 Code Encoder 的第 2 个任务由过新的编码器 Source Encoder 来承担,我们认为这样可以使模型中的每个编码器只有一个角色,并专注于一个任务,从而在一定程度上提高了模型的表现.

我们将 Code-PGN 与 Hybrid-DeepCom 在不同词库大小下的表现进行对比,分析不同词库大小下,加入 Source Encoder 是否会提升模型的表现,如果带来了提升,我们将进一步分析提升的原因.

表 1 中的实验结果表明,在词库大小为 5 000 时两者得分相近,在 10000~50000 时 CodePtr-PGN 高于 Hybrid-DeepCom.对于两者在词库大小为 5 000 时得分相近的原因,我们认为是上述 Source Encoder 为 CodePtr-PGN 带来的优势在较小的词库大小下并没有产生作用,因为对于含有较多 OOV 词的分解前的源代码,在较小的词库大小下,Source Encoder 的输入包含了大量的“⟨UNK⟩”标签,这些大比例的无意义信息使得 Source Encoder 无法给模型带来实际的提升.这也是当词库大小为 30 000 和 50 000 时,CodePtr-PGN 与 Hybrid-DeepCom 的分差比在 10 000 时要大的原因.

表 2 显示了在词库大小为 30 000 时 CodePtr-PGN 和 Hybrid-DeepCom 在所示样本上的输出,其中加粗的单词表示 Source Encoder 和 Code Encoder 输入之间的差异.

从表中可以看出,Code Encoder 的输入将类名“NinePatchBorder”根据驼峰命名法分解为了“nine”“patch”和“border”.而对于两个模型的预测输出,Hybrid-DeepCom 和 CodePtr 分别输出了“instantiates the nine setting.”和“instantiates a new nine patch border.”.通过参考注释,我们可以看出,两个模型都正确地判断出了源代码的意图,就是实例化(“instantiates”)一个类.而对于类的名称,在两个模型都没有指针生成网络的情况下,Hybrid-DeepCom 生成的注释中仅有“nine”,而 CodePtr-PGN 则生成了该类名分解后的所有单词.图 5 显示了两个模型

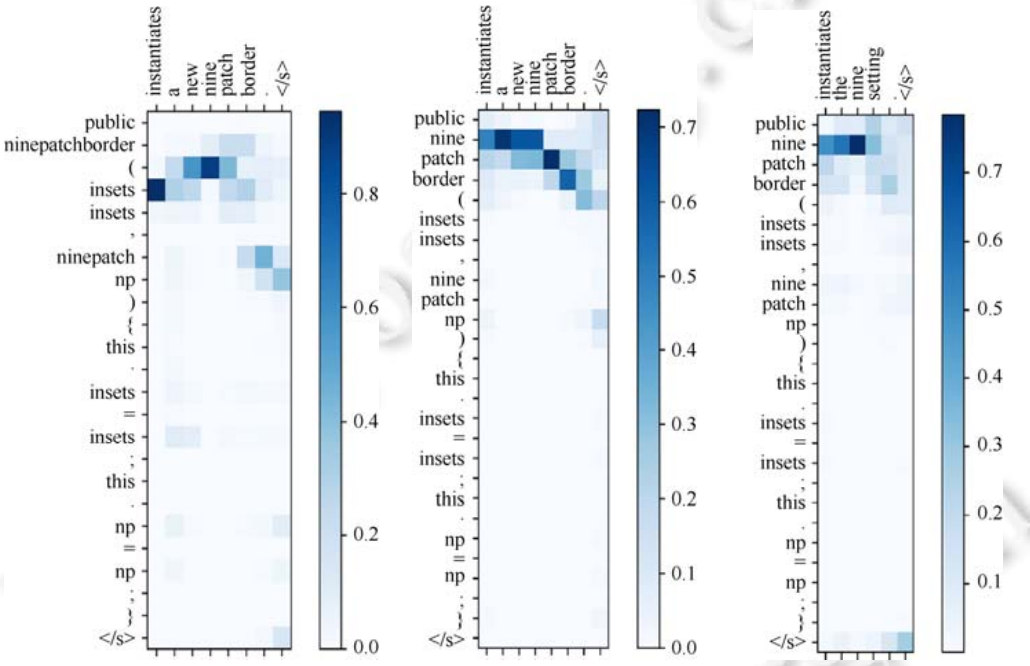
在解码的每一个时刻对分解前后的输入序列的 Attention 权重.

Table 2 A sample from test dataset and predictions of two models

表 2 测试集中的一个样本以及两个模型的预测输出

源代码	public NinePatchBorder(Insets insets, NinePatch np){
	this.insets=insets;
	this.np=np;
	}

Source Encoder 输入	public **ninepatchborder** (insets insets, ninepatch np) {this. insets=insets; this. np=np;}
Code Encoder 输入	public **nine patch border** (insets insets, nine patch np) {this. insets=insets; this. np=np;}
参考注释	instantiates a new nine patch border.
Hybrid-DeepCom 输出	instantiates the nine setting.
CodePtr-PGN 输出	instantiates a new nine patch border.



(a) Attention weights of source input in CodePtr (without PGN)

(b) Attention weights of code input in CodePtr (without PGN)

(c) Attention weights of code input in Hybrid-DeepCom

Fig.5 Each step's Attention weights of input sequence in CodePtr-PGN and Hybrid-DeepCom when decoding

图 5 CodePtr-PGN 和 Hybrid-DeepCom 在解码时每一时刻对输入序列的 Attention 权重分布

图 5(a)和图 5(b)分别为 CodePtr-PGN 中 Source Encoder 和 Code Encoder 对应的输入序列在解码时的 Attention 权重分布,图 5(c)为 Hybrid-DeepCom 中 Code Encoder 输入序列的权重分布,横坐标表示解码时生成的每一个单词的时刻,纵坐标表示输入序列的每个单词.从图 5(c)可以看出,Hybrid-DeepCom 在生成“nine”时,正确地关注到了输入序列中的“nine”,而在下一时刻,模型仍然较多地关注了“nine”,而不是输入序列中的下一个单词“patch”,从而生成了看起来没有相关性的“setting”.而对于图 5(b),CodePtr-PGN 在生成“nine”“patch”和“border”时都正确地关注到了输入序列中对应的单词.

我们认为这是 CodePtr-PGN 的 Source Encoder 分担了 Hybrid-DeepCom 中 Code Encoder 的匹配 AST 信息任务带来的效果.因为对标识符“NinePatchBorder”进行分解破坏了源代码的语法结构,使得分解后的“nine”代替了原来的类名的位置,而解码器在解码时根据 AST 提取到的信息,类的名称本应指向“NinePatchBorder”出现的位置,分解后该位置变成了“nine”,而且 Hybrid-DeepCom 中的 Code Encoder 需要在提取语义信息的同时匹配

AST 信息,使其在生成类名时只关注到了 AST 信息中指向的类名“nine”.而 CodePtr-PGN 则将匹配 AST 信息的任务分给了 Source Encoder,使 Code Encoder 可以专注于提取语义信息,而不需要在一定程度上互斥的两个任务之间加以权衡.从图 5(b)可以看出,这一改进使 Code Encoder 更加准确地注意到了类“NinePatchBorder”的完整名称.此外,通过观察图 5(a)和图 5(c)中 Attention 权重的局部分布特征,可以发现两者在某些局部的分布有一定的相似性,并且这些特征是图 5(b)中所没有的,例如上方和右下方,因此我们有理由相信,CodePtr-PGN 的 Source Encoder 的确分担了 Hybrid-DeepCom 中 Code Encoder 的一部分任务.

本小节举例分析了 CodePtr-PGN 相较于 Hybrid-DeepCom 的优势,证明了引入的 Source Encoder 在大部分情况下对模型的表现带来了提升,并具体分析了带来提升的原因.

3.4.3 RQ2:指针生成网络能否解决输出端的 OOV 问题?

指针生成网络可以帮助 CodePtr 在注释中生成 OOV 词,我们对比分析了 CodePtr 与 CodePtr-PGN 和 Hybrid-DeepCom 在测试集上生成的注释,统计出了与 OOV 相关的信息,具体结果见表 3.表中 OOV 比例是指在训练阶段,词库中被裁剪掉的 OOV 词占裁剪前词库大小的比例,Source 是指未经驼峰分解的词库,Code 是指经驼峰分解的词库,Comment 是指根据注释构建的词库,SBT 序列由于只保留了类型,加上括号和一些特殊符号,对应的词库大小仅为 58,不需要裁剪.生成的注释中“⟨UNK⟩”数量是指两个模型针对 19 431 个样本的测试集上生成的注释中“⟨UNK⟩”标签的总数,CodePtr 生成的 OOV 词数量等于 CodePtr 在解码过程中输出 OOV 词的总次数,Hybrid-DeepCom 和无指针生成网络的 CodePtr 无法在注释中生成 OOV 词,因此其生成的 OOV 词数量为 0.

Table 3 OOV related statistics on test dataset of Hybrid-DeepCom and CodePtr

表 3 Hybrid-DeepCom 与 CodePtr 在测试集上的 OOV 相关数据

词库大小	OOV 比例(%)			生成的注释中“⟨UNK⟩”的数量			CodePtr 生成的
	Source	Code	Comment	Hybrid-DeepCom	CodePtr-PGN	CodePtr	OOV 词数量
5 000	98.95	88.50	90.70	5 329	6 126	4 53	1 093
10 000	97.90	77.00	81.41	2 756	3 272	18 17	801
30 000	93.70	31.00	44.22	670	896	397	232
50 000	89.51	0.00	7.04	92	107	12	24

从上表可以看出,相比于 Hybrid-DeepCom 和 CodePtr-PGN,CodePtr 生成的注释中“⟨UNK⟩”的数量有明显的降低,并且随着词库大小的减少,OOV 比例的升高,CodePtr 生成的 OOV 词就越多.同时我们也发现,CodePtr-PGN 生成的注释中“⟨UNK⟩”的数量比 Hybrid-DeepCom 要多,我们分析是因其加入的 Source Encoder 的输入为分解前的源代码序列,从表中“OOV 比例”栏目下“Source”一列可以看出,其中含有大量的“⟨UNK⟩”标签,对 CodePtr-PGN 造成了干扰,因此可以看到,随着词库大小的增大,两者生成“⟨UNK⟩”的数量差距在逐渐缩小.

表 4 中,我们挑选了测试集上的一些样本,并给出了 Hybrid-DeepCom、CodePtr-PGN 和 CodePtr 两个模型在词库大小为 30 000 情况下生成的注释,受篇幅所限,代码中的具体字符串使用“⟨STR⟩”替代,注释中加粗的单词为 OOV 词.

可以看到,通过引入指针生成网络,CodePtr 可以在注释中生成 OOV 词,绝大部分均为属性名、方法名、类名等标识符,大大提高了注释的可读性.例子 2 中,CodePtr 生成了比参考句子更正确的注释,虽然出现了重复,但是可读性更高,Hybrid-DeepCom 出现了两处重复,可读性并不高,CodePtr without PGN 虽然没有出现重复,但却缺失了关键的标识符名称和“list”关键词.例子 8 中,相比于参考注释中的变量名,CodePtr 生成了正确的变量名,Hybrid-DeepCom 生成了看似不相关的注释,CodePtr-PGN 虽然正确地捕捉到了程序的意图,但在变量名处生成了“⟨UNK⟩”标签.我们还注意到例子 2 中的 CodePtr 和 Hybrid-DeepCom,例子 5 中的 CodePtr without PGN 和 CodePtr,都生成了重复的一个或多个单词,这是在基于 Attention 机制的 seq2seq 模型中普遍存在的问题[46,47].在解码过程中,由于 Attention 机制,生成的单词很大程度上依赖于输入序列的相关部分,并且输入序列中的单词参与到了每个输出单词的生成过程中,因此会出现过度翻译(over-translation)和翻译不足(under-translation)的问题[46],在输出结果中出现重复就是出现了过度翻译的问题,即对输入序列中的某个单词或片段过度关注[47].

Table 4 Examples of generated comment by models

表 4 模型生成的注释例子

序号	例子

	protected void forwardReply(AbstractMRListener client, AbstractMRReply r){
	((Dcc4PcListener)client).reply((Dcc4PcReply)r);
	}

1	**Human-written**: forward a dcc pcreply to all registered dcc pcinterface listeners.
	Hybrid-DeepCom: forward a ⟨UNK⟩ to all registered easydccinterface listeners.
	CodePtr-PGN: forward a ⟨UNK⟩ to all registered ecosinterface listeners.
	CodePtr: forward a **dcc4pcreply** to all registered nceinterface listeners.
	public void addBloatControl(BloatControlFunction bloat_control){
	this.bloatControl.add(bloat_control);
	}

2	**Human-written**: set new bloat control function
	Hybrid-DeepCom: add a control control to the list of control control list.
	CodePtr-PGN: add a control to this control.
	CodePtr: adds a **bloatcontrolfunction** control control to the control list.
	public RootPaneNoFrameState(){
	super();
	}

3	**Human-written**: creates a new **rootpanenoframestate** object.
	Hybrid-DeepCom: creates a new ⟨UNK⟩
	CodePtr-PGN: creates a new ⟨UNK⟩ object.
	CodePtr: creates a new **rootpanenoframestate** object.
	public Boolean isWhiteLabelWallet(){
	return whiteLabelWallet;
	}

4	**Human-written**: gets the value of the **whitelabelwallet** property.
	Hybrid-DeepCom: gets the value of the ⟨UNK⟩ property.
	CodePtr-PGN: gets the value of the ⟨UNK⟩ property.
	CodePtr: gets the value of the **whitelabelwallet** property.
	public CombinatorialNameGenerator build(final boolean usesMiddles) throws IllegalStateException{
	checkState(usesMiddles);
	return new CombinatorialNameGenerator(this, usesMiddles);
	}

5	**Human-written**: uses the builder to construct a new **combinatorialnamegenerator**.
	Hybrid-DeepCom: creates a new ⟨UNK⟩ with the specified options for this instance of this builder.
	CodePtr-PGN: builds the name using the provided builder using the provided builder.
	CodePtr: builds the **combinatorialnamegenerator** with the specified builder builder.
	private static boolean CallNonvirtualBooleanMethodV(JNIEnvironment env,
	int objJREF,
	int classJREF,
	int methodID,
	Address argAddress) throws Exception{
	if (traceJNI)
	VM.sysWrite(⟨STR⟩);
	RuntimeEntrypoints.checkJNICountDownToGC();
	try{
	Object obj=env.getJNIRef(objJREF);
	Object returnObj=JNIHelpers.invokeWithVarArg(obj, methodID, argAddress, TypeReference.Boolean, True);
	return Reflection.unwrapBoolean(returnObj);
	}

6	catch (Throwable unexpected){
	if (traceJNI)
	unexpected.printStackTrace(System.err);
	env.recordException(unexpected);
	return False;
	}
	}
	Human-written: **callnonvirtualbooleanmethodv** invoke a virtual method that returns a boolean value
	Hybrid-DeepCom: ⟨UNK⟩ invoke a virtual method that returns a boolean value
	CodePtr-PGN: ⟨UNK⟩ invoke a virtual method that returns a boolean value
	CodePtr: **callnonvirtualbooleanmethodv** invoke a virtual method that returns a boolean value

Table 4 Examples of generated comment by models (Continued)

表 4 模型生成的注释例子(续)

序号	例子

	PersistentRegistrarImpl(String[] configArgs, LifeCycle lifeCycle) throws Exception{
	super(configArgs, null, True, lifeCycle);
	}

7	**Human-written**: constructs a non activatable **persistentregistrarimpl** based on a configuration obtained using the provided arguments.
	Hybrid-DeepCom: create an instance registrar based based on the configuration.
	CodePtr-PGN: creates a new instance based on the given arguments.
	CodePtr: constructs an activatable **persistentregistrarimpl** using the provided obtained from the using the provided configuration.
	public Boolean isVirtualICH7MPresent(){
	return virtualICH7MPresent;
	}

8	**Human-written**: gets the value of the **virtualich mpresent** property.
	Hybrid-DeepCom: returns indicator image data is enabled.
	CodePtr-PGN: gets the value of the ⟨UNK⟩ property.
	CodePtr: gets the value of the **virtualich7mpresent** property.
	public boolean isSetBonk(){
	return this.bonk !=null;
	}

9	**Human-written**: returns true if field **bonk** is set has been assigned a value and false otherwise **Hybrid-DeepCom**: returns true if field ⟨UNK⟩ is set has been assigned a value and false otherwise **CodePtr-PGN**: returns true if field ⟨UNK⟩ is set has been assigned a value and false otherwise **CodePtr**: returns true if field **bonk** is set has been assigned a value and false otherwise

我们同时分析了 CodePtr 在解码例子 9 的过程中,在生成每一个单词时的 Attention 权重分布图和 p_{gen} 的取值,由于 Source Encoder 专门用于使指针生成网络发挥作用,因此这里只考虑 Source Encoder 的输入,如图 6 所示.

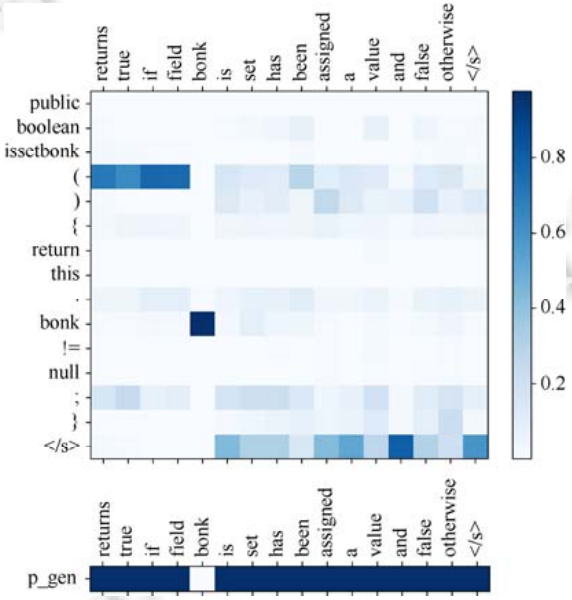


Fig.6 Attention weights distribution and p_{gen} values of CodePtr when decoding Example 9

图 6 CodePtr 解码例子 9 时的 Attention 权重分布和 p_{gen} 取值

图 6 中上方为 Attention 权重分布,横轴为时间轴,表示解码时的每个时刻,纵轴为 Source Encoder 的输入,图中的色块表示在生成每一个单词时解码器对 Source Encoder 输入序列中每个单词的 Attention 权重,颜色越深

表示权重越高,下方表示解码器在每个时刻的 p_{gen} 值.因此可以看到,在生成 OOV 单词“bonk”时,解码器对输入中的“bonk”的 Attention 权重最高,而输入序列中的其他单词几乎为 0,并且此时的 p_{gen} 值几乎为 0,p_{gen} 表示的是解码器在两个模式间软切换中生成模式所占的比重,因此,此时的 p_{gen} 表示模型倾向于完全切换到复制模式,又因为此时输入序列中“bonk”的 Attention 权重接近 1,所以解码器直接复制输入序列中的“bonk”作为当前时刻的输出.

本小节证明了指针生成网络可以减少生成注释中“⟨UNK⟩”的数量,提高注释的可读性,并且举例分析了作用原理,从多方面证明了指针生成网络可以在一定程度上解决输出端的 OOV 问题.

3.4.4 RQ3:词库大小对模型性能的影响有什么规律?

图 7 显示了不同词库大小下 3 个模型的平均得分折线图.首先可以看出,在词库大小为 5000~30000 时,由于引入了指针生成网络,CodePtr 的得分高于前两个模型,而在 50 000 时与无指针生成网络下的 CodePtr 接近,两者均高于 Hybrid-DeepCom.

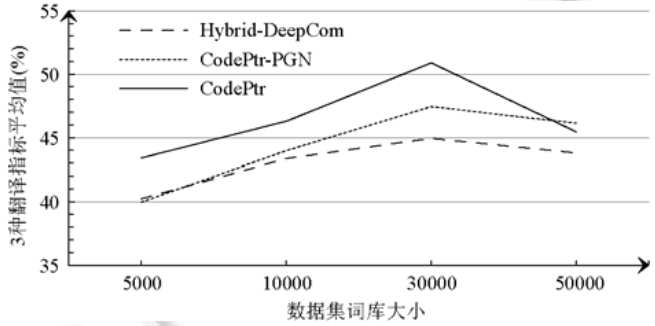


Fig.7 Average score of models under different vocabulary size, the horizontal coordinate indicates the different vocabulary sizes and the vertical coordinate is the average score of the model on the three evaluation metrics

图 7 不同词库大小下模型的平均得分,横坐标是不同的词库大小,
纵坐标是模型在 3 个评估标准下的平均得分

当词库大小为 50 000 时,CodePtr 与无指针生成网络的 CodePtr 得分相近.因为由表 2 可知,在词库大小为 50 000 的情况下,由于词库已足够大,指针生成网络在注释中生成的 OOV 词仅有 24 个,同时也可以推断出在训练阶段,由于缺乏可用于训练指针生成网络的样本,对指针生成网络的训练也存在不足,在这种情况下,指针生成网络的作用并不大,反而会因为训练不足而对最终结果产生干扰,因此前者的得分略低于后者.

由以上分析可以看出,对于 CodePtr,词库大小并不是越大越好,对其他模型也是如此,从图中还可以看出,在词库大小为 5000~30000 时,模型的得分随着词库大小的升高而升高,而词库大小为 50 000 时,3 个模型的得分均有所下降,我们认为,此时由于噪声过多,对模型的性能造成了负面的影响.从表 2 可以得知,在词库大小为 50 000 时,分解后的源代码词库的 OOV 比例为 0,这表示 50 000 大小的词库对于本次实验进行的数据集而言已足够大.我们分析是因为过大的词库引入了很多并不需要的单词,这些单词仅在训练集中出现很少次数,因为很多变量名、方法名等标识符都是简写或拼接而来的,在其他样本中再次出现的概率很低,即使是分解过后的变量名,也存在很多出现次数极少的单词,例如简写、自定义的单词等等.将这些单词也进行编码,将其映射到词向量空间等训练步骤,我们认为是没有必要的,甚至适得其反,因为要使模型的泛化能力提升,需要使其学习到该问题的一般性规律,在这方面,我们认为那些出现次数极少的单词对于模型来说是噪声,因为极低的出现概率,模型进行测试和预测时很少再遇到相同的单词,Gong 等人[48]通过研究发现,在进行词嵌入时,语义相近的词本应拥有相似的词向量表示,但是这对于出现次数极少的单词而言并不成立,他们认为这是出现次数极少的单词对应的词向量很少被更新造成的.此外,Gong 等人还发现高频单词和低频单词常常分布在词向量空间的不同区域,这就导致了对于低频单词而言,词向量是无效的.因此,过大的词库反而会对模型造成负面的影响.

这也解释了词库大小从 30 000 增加到 50 000 时,CodePtr 的两个模型比 Hybrid-DeepCom 下降幅度更大的现象,CodePtr 为了引入指针生成网络,增加了一个编码器用于接受未分解的源代码,表 5 统计了训练集上分解前的源代码词库中单词出现次数的分布情况.从表中可以得知,在词库大小为 50 000 的情况下,大量出现次数为 5~10 次的单词被添加到了词库中,这些单词在大小为 476 654 的训练集上出现的次数极少,其词向量很少被训练和更新,因此可以称得上是噪声,引入大量的噪声单词对 CodePtr 造成了较大的影响.

Table 5 Statistics of occurrences of words in source vocabulary

表 5 分解前源代码词库中单词出现次数统计

词库大小	出现次数	单词数量	占比(%)	累计占比(%)	剩余
476 654	1	203 581	42.71	42.71	273 073
	2	98 266	20.62	63.33	174 807
	3	45 974	9.65	72.97	128 833
	4	31 737	6.66	79.63	97 096
	5~10	56 526	11.86	91.49	40 570
	11~100	35 975	7.55	99.04	4 595
	101~	4 595	42.27	100	0

4 总结与展望

本文针对当前表现最佳模型的两个输入不完全匹配的不足,通过增加一个输入对其进行了改进,首先提出了无指针生成网络的 CodePtr.然后针对现有的代码注释自动生成方法普遍存在的局限性,即注释中无法生成 OOV 词的问题,引入了指针生成网络,提出了 CodePtr.实验结果表明,无指针生成网络的 CodePtr 在大多数情况下的性能优于 Hybrid-DeepCom,而 CodePtr 在实验中的所有情况都优于 Hybrid-DeepCom,在大多数情况下优于无指针生成网络的 CodePtr.

在未来的工作中,我们试图解决在第 3.4.4 节中提出的较大词库大小情况下模型得分下降的问题.此外,考虑到驼峰分解前后的源代码序列存在较多冗余,我们将尝试把两个输入融合,在减少冗余的同时也可以加快模型的训练速度.另外,针对第 3.4.3 节中提到的输出序列中出现重复的问题,可以考虑使用文献[46,47]中的方法加以解决.

References:

[1] Tenny T. Procedures and comments vs. the banker's algorithm. ACM SIGCSE Bulletin, 1985,17(3):44–53. [doi: 10.1145/382208.382523]

[2] Tashtoush Y, Odat Z, Alsmadi IM, Yatim M. Impact of programming features on code readability. Int'l Journal of Software Engineering and Its Applications, 2013,7(6):441–458. [doi: 10.14257/ijseia.2013.7.6.38]

[3] Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. on Software Engineering, 2017,44(10):951–976. [doi: 10.1109/TSE.2017.2734091]

[4] Fluri B, Wursch M, Gall HC. Do code and comments co-evolve? on the relation between source code and comment changes. In: Proc. of the 14th Working Conf. on Reverse Engineering (WCRE 2007). IEEE, 2007. 70–79. [doi: 10.1109/WCRE.2007.21]

[5] Song X, Sun H, Wang X, Yan J. A survey of automatic generation of source code comments: Algorithms and techniques. IEEE Access, 2019,7:111411–111428. [doi: 10.1109/ACCESS.2019.2931579]

[6] Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering, 2019, 1–39. [doi: 10.1007/s10664-019-09730-9]

[7] Marcus A, Maletic JI. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proc. of the 25th Int'l Conf. on Software Engineering. IEEE, 2003. 125–135. [doi: 10.1109/ICSE.2003.1201194]

[8] Kuhn A, Ducasse S, Gírba T. Semantic clustering: Identifying topics in source code. Information and Software Technology, 2007, 49(3):230–243. [doi: 10.1016/j.infsof.2006.10.017]

[9] Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text summarization techniques for summarizing source code. In: Proc. of the 17th Working Conf. on Reverse Engineering. IEEE, 2010. 35–44. [doi: 10.1109/WCRE.2010.13]

[10] Wong E, Liu T, Tan L. Clocom: Mining existing source code for automatic comment generation. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2015. 380–389. [doi: 10.1109/SANER.2015.7081848]

[11] 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 (Volume 1: Long Papers). 2016. 2073–2083. [doi: 10.18653/v1/P16-1195]

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

[13] 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. 2018. 2269–2275. [doi: 10.5555/3304889.3304975]

[14] Wan Y, Zhao Z, Yang M, Xu G, Ying H, 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. 2018. 397–407. [doi: 10.1145/3238147.3238206]

[15] Wei B, Li G, Xia X, Fu Z, Jin Z. Code generation as a dual task of code summarization. In: Advances in Neural Information Processing Systems. 2019. 6559–6569.

[16] Wang X, Pollock L, Vijay-Shanker K. Automatically generating natural language descriptions for object-related statement sequences. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2017. 205–216. [doi: 10.1109/SANER.2017.7884622]

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

[18] Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems. 2014. 3104–3112. [doi: 10.5555/2969033.2969173]

[19] Deissenboeck F, Pizka M. Concise and consistent naming. Software Quality Journal, 2006,14(3):261–282. [doi: 10.1007/s11219-006-9219-1]

[20] Luong MT, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. In: Proc. of the 2015 Conf. on Empirical Methods in Natural Language Processing. 2015. 1412–1421.

[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 (Volume 1: Long Papers). 2017. 1073–1083. [doi: 10.18653/v1/P17-1099]

[22] Movshovitz-Attias D, Cohen W. Natural language models for predicting programming comments. In: Proc. of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). 2013. 35–40.

[23] Koehn P, Hoang H, Birch A, Callison-Burch C, Federico M, Bertoldi N, Cowan B, Shen W, Moran C, Zens R, Dyer C, Bojar O, Constantin A, Herbst E. Moses: Open source toolkit for statistical machine translation. In: Proc. of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions. 2007. 177–180. [doi: 10.5555/1557769.1557821]

[24] Rush AM, Chopra S, Weston J. A neural attention model for abstractive sentence summarization. In: Proc. of the 2015 Conf. on Empirical Methods in Natural Language Processing. 2015. 379–389. [doi: 10.18653/v1/D15-1044]

[25] Eriguchi A, Hashimoto K, Tsuruoka Y. Tree-to-sequence attentional neural machine translation. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016. 823–833. [doi: 10.18653/v1/P16-1078]

[26] Zhou Y, Yan X, Yang W, Chen T, Huang Z. Augmenting Java method comments generation with context information based on neural networks. Journal of Systems and Software, 2019,156:328–340. [doi: 10.1016/j.jss.2019.07.087]

[27] Huang Y, Huang S, Chen H, Chen X, Zheng Z, Luo X, Jia N, Hu X, Zhou X. Towards automatically generating block comments for code snippets. Information and Software Technology, 2020, 106373. [doi: 10.1016/j.infsof.2020.106373]

[28] Wang W, Zhang Y, Sui Y, Wan Y, Zhao Z, Wu J, Yu P, Xu G. Reinforcement-learning-guided source code summarization via hierarchical attention. IEEE Trans. on Software Engineering, 2020. [doi: 10.1109/TSE.2020.2979701]

[29] LeClair A, Jiang S, 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). IEEE, 2019. 795–806. [doi: 10.1109/ICSE.2019.00087]

[30] Yashu L, Zhihai W, Yueran H, Hanbing Y. A method of extracting malware features based on probabilistic topic model. Journal of Computer Research and Development, 2019,56(11):2339 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20190393]

[31] Gao Y, Liu H, Fan XZ, Niu ZD. Method name recommendation based on source code depository and feature matching. Ruan Jian Xue Bao/Journal of Software, 2015,26(12):3062–3074 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4817.htm [doi: 10.13328/j.cnki.jos.004817]

[32] Broder AZ. On the resemblance and containment of documents. In: Proc. of the Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171). IEEE, 1997. 21–29. [doi: 10.1109/SEQUEN.1997.666900]

[33] Huang Y, Liu ZY, Chen XP, Xiong YF, Luo XN. Auxiliary method for code commit comprehension based on core-class identification. Ruan Jian Xue Bao/Journal of Software, 2017,28(6):1418–1434 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5225.htm [doi: 10.13328/j.cnki.jos.005225]

[34] Huang Y, Jia N, Zhou Q, Chen XP, Xiong YF, Luo XN. Method combining structural and semantic features to support code commenting decision. Ruan Jian Xue Bao/Journal of Software, 2018,29(8):2226–2242 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5528.htm [doi: 10.13328/j.cnki.jos.005528]

[35] Lin ZQ, Zou YZ, Zhao JF, Cao YK, Xie B. Software text semantic search approach based on code structure knowledge. Ruan Jian Xue Bao/Journal of Software, 2019,30(12):3714–3729 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5609.htm [doi: 10.13328/j.cnki.jos.005609]

[36] Mou L, 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. 2016. 1287–1293. [doi: 10.5555/3015812.3016002]

[37] Cao Y, Zou Y, Luo Y, Xie B, Zhao J. Toward accurate link between code and software documentation. Science China Information Sciences, 2018,61(5):050105. [doi: 10.1007/s11432-017-9402-3]

[38] Chen C, Peng X, Sun J, Xing Z, Wang X, Zhao Y, Zhang H, Zhao W. Generative API usage code recommendation with parameter concretization. Science China Information Sciences, 2019,62(9):192103. [doi: 10.1007/s11432-018-9821-9]

[39] Tai KS, Socher R, Manning CD. Improved semantic representations from tree-structured long short-term memory networks. In: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th Int'l Joint Conf. on Natural Language Processing (Volume 1: Long Papers). 2015. 1556–1566. [doi: 10.3115/v1/P15-1150]

[40] 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, ICLR 2015. 2015.

[41] Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems—Volume 2. 2015. 2692–2700. [doi: 10.5555/2969442.2969540]

[42] Kingma DP, Ba J. ADAM: A method for stochastic optimization. In: Proc. of the 3rd Int'l Conf. on Learning Representations, ICLR 2015. 2015.

[43] Loshchilov I, Hutter F. Decoupled weight decay regularization. In: Proc. of the Int'l Conf. on Learning Representations. 2018.

[44] Papineni K, Roukos S, Ward T, *et al*. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, 2002. 311–318. [doi: 10.3115/1073083.1073135]

[45] Denkowski M, Lavie A. Meteor universal: Language specific translation evaluation for any target language. In: Proc. of the 9th Workshop on Statistical Machine Translation. 2014. 376–380. [doi: 10.3115/v1/W14-3348]

[46] Tu Z, Lu Z, Liu Y, Liu X, Li H. Modeling coverage for neural machine translation. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016. 76–85. [doi: 10.18653/v1/P16-1008]

[47] Mi H, Sankaran B, Wang Z, Ittycheriah A. Coverage embedding models for neural machine translation. In: Proc. of the 2016 Conf. on Empirical Methods in Natural Language Processing. 2016. 955–960. [doi: 10.18653/v1/D16-1096]

[48] Gong C, He D, Tan X, Qin T, Wang L, Liu T. Y. Frage: Frequency-agnostic word representation. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. 2018. 1341–1352. [doi: 10.5555/3326943.3327066]

附中文参考文献:

[30] 刘亚姝,王志海,侯跃然,严寒冰.一种基于概率主题模型的恶意代码特征提取方法.计算机研究与发展,2019,56(11):2339–2348. [doi: 10.7544/issn1000-1239.2019.20190393]

[31] 高原,刘辉,樊孝忠,牛振东.基于代码库和特征匹配的函数名称推荐方法.软件学报,2015,26(12):3062–3074. http://www.jos.org.cn/1000-9825/4817.htm [doi: 10.13328/j.cnki.jos.004817]

[33] 黄袁,刘志勇,陈湘萍,熊英飞,罗笑南.基于关键类判定的代码提交理解辅助方法.软件学报,2017,28(6):1418–1434. http://www.jos.org.cn/1000-9825/5225.htm [doi: 10.13328/j.cnki.jos.005225]

[34] 黄袁,贾楠,周强,陈湘萍,熊英飞,罗笑南.融合结构与语义特征的代码注释决策支持方法.软件学报,2018,29(8):2226–2242. http://www.jos.org.cn/1000-9825/5528.htm [doi: 10.13328/j.cnki.jos.005528]

[35] 林泽琦,邹艳珍,赵俊峰,曹英魁,谢冰.基于代码结构知识的软件文档语义搜索方法.软件学报,2019,30(12):3714–3729. http://www.jos.org.cn/1000-9825/5609.htm [doi: 10.13328/j.cnki.jos.005609]

牛长安(1997－),男,学士,CCF 学生会员,主要研究领域为软件工程,自然语言处理.

李传艺(1991－),男,博士,助理研究员,CCF 专业会员,主要研究领域为软件工程,业务过程管理,自然语言处理.

葛季栋(1978－),男,博士,副教授,CCF 高级会员,主要研究领域为软件工程,分布式计算与边缘计算,业务过程管理,自然语言处理.

周宇(1981－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为智能化软件技术,云计算,大数据.

唐泽(1994－),男,硕士,主要研究领域为代码摘要,API 补全.

骆斌(1967－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件工程,人工智能.