软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(3):1310−1329 [doi: 10.13328/j.cnki.jos.006690]

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

基于双重信息检索的 Bash 代码注释生成方法*

陈 翔[1,2], 于 池[1], 杨 光[1], 濮雪莲[3], 崔展齐[4]

[1](南通大学 信息科学技术学院, 江苏 南通 226019)

[2](信息安全国家重点实验室(中国科学院 信息工程研究所), 北京 100093)

[3](南通大学 经济与管理学院, 江苏 南通 226019)

[4](北京信息科技大学 计算机学院, 北京 100101)

通信作者: 陈翔, E-mail: xchencs@ntu.edu.cn; 濮雪莲, E-mail: pu.xl@ntu.edu.cn

摘 要: Bash 是 Linux 默认的 shell 命令语言. 它在 Linux 系统的开发和维护中起到重要作用. 对不熟悉 Bash 语言的开发人员来说, 理解 Bash 代码的目的和功能具有一定的挑战性. 针对 Bash 代码注释自动生成问题提出了一种基于双重信息检索的方法 ExplainBash. 该方法基于语义相似度和词法相似度进行双重检索, 从而生成高质量代码注释. 其中, 语义相似度基于 CodeBERT 和 BERT-whitening 操作训练出代码语义表示, 并基于欧式距离来实现; 词法相似度基于代码词元构成的集合, 并基于编辑距离来实现. 以 NL2Bash 研究中共享的语料库为基础, 进一步合并 NLC2CMD 竞赛共享的数据以构造高质量语料库. 随后, 选择了来自代码注释自动生成领域的 9 种基准方法, 这些基准方法覆盖了基于信息检索的方法和基于深度学习的方法. 实证研究和人本研究的结果验证了 ExplainBash 方法的有效性. 然后设计了消融实验, 对 ExplainBash 方法内设定(例如检索策略、BERT-whitening 操作等)的合理性进行了分析. 最后, 基于所提方法开发出一个浏览器插件, 以方便用户对 Bash 代码的理解.

关键词: 程序理解; Bash 代码; 代码注释生成; 信息检索; 代码语义; 代码词法

中图法分类号: TP311

中文引用格式: 陈翔, 于池, 杨光, 濮雪莲, 崔展齐. 基于双重信息检索的 Bash 代码注释生成方法. 软件学报, 2023, 34(3): 1310−1329. http://www.jos.org.cn/1000-9825/6690.htm

英文引用格式: Chen X, Yu C, Yang G, Pu XL, Cui ZQ. Bash Code Comment Generation Method Based on Dual Information Retrieval. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1310−1329 (in Chinese). http://www.jos.org.cn/1000-9825/ 6690.htm

Bash Code Comment Generation Method Based on Dual Information Retrieval

CHEN Xiang[1,2], YU Chi[1], YANG Guang[1], PU Xue-Lian[3], CUI Zhan-Qi[4]

[1](School of Information Science and Technology, Nantong University, Nantong 226019, China)

[2](State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences), Beijing 100093, China)

[3](Economics and Management School, Nantong University, Nantong 226019, China)

[4](School of Computer, Beijing Information Science and Technology University, Beijing 100101, China)

Abstract: Bash is the default shell command language for Linux, which plays an important role in the development and maintenance of Linux systems. Nevertheless, understanding the purpose and functionality of the Bash code is a challenging task. Therefore, an automatic method ExplainBash is proposed based on dual information retrieval for automatic Bash code comment generation. Specifically, the proposed method is based on semantic similarity and lexical similarity to perform dual information retrieval, which aims to generate high-quality code comments. For semantic similarity, CodeBERT and BERT-whitening operator are used to learn the code semantic

* 基金项目: 国家自然科学基金(61872263, 61702041, 61202006); 信息安全国家重点实验室开放课题(2020-MS-07); 江苏省前沿引领技术基础研究专项(BK20202001); 江苏省重点产业专利导航项目(DH20200072-10)

收稿时间: 2021-09-14; 修改时间: 2022-01-13; 采用时间: 2022-04-01

representation, and Euclidean distance is resorted to compute semantic similarity; while for lexical similarity, code is represented as a set of code tokens, then the edit distance is resorted to compute lexical similarity. A high-quality corpus is constructed based on the corpus shared in the NL2Bash study and the data shared in the NLC2CMD competition. After that, nine state-of-the-art baselines are selected from the automatic code comment generation domain, which cover the information retrieval-based methods and deep learning-based methods. Results of empirical study and human study verify the effectiveness of the proposed method. Ablation experiments are also designed to analyze the rationality of the settings (such as retrieval strategy, BERT-whitening operator) in the proposed method. Finally, a browser plug-in is developed based on the proposed method to facilitate the code comprehension of the Bash code.

Key words: program comprehension; Bash code; code comment generation; information retrieval; code semantic; code lexical

Shell 是开发人员和 Linux 操作系统之间进行交互的接口. 当前, Linux 操作系统支持不同类型的 shell, 其中, Bash 是 Linux 默认的 shell 命令语言, 并且在程序开发过程中应用较为广泛. Bash 作为一种脚本语言, 通常包括 3 个基本组件: 程序名(例如 find, mkdir, cd, grep)、选项(例如–*name*, –*i*)以及参数(例如“*.java”). 和传统的编程语言 C 语言、Java 语言、Python 语言等相比, Bash 语言的使用场景较少, 但是 Bash 语言在 Linux 系统的开发和维护过程中的作用仍然不可忽视.

Bash 语言具有语言灵活等特点[1], 因此对于不熟悉 Bash 语言的开发人员来说, 使用 Bash 语言在完成开发和维护任务时仍具有一定困难. 截至目前(即 2021 年 9 月), 在开发人员问答网站 Stack Overflow 上,累计有 84 195 条与“shell”关键词相关的问答帖子, 有 138 534 条与“bash”关键词相关的问答帖子. 以表 1 所示的 Stack Overflow 上与 Bash 代码相关问答帖子为例 (https://stackoverflow.com/questions/2443085/what-does-command-args-mean-in-the-shell), 该用户不能理解“<<”符号在 Bash 代码中的含义. 可以看出, 对于 Bash 语言不熟悉的开发人员来说, 在理解 Bash 代码的目的和实现功能上具有一定的挑战性. 因此, 针对 Bash 代码, 亟需可以生成相关代码注释的自动方法, 可以协助开发人员理解 Bash 代码.

表 1 Stack Overflow 网站上与 Bash 代码理解相关的帖子的示例

标题	What does “<<(command args)” mean in the shell?
内容	When looping recursively through folders with files containing spaces the shell script I use is of this form, copied from the Internet: **while** IFS=read –*r* –*d* $‘\0’ file; **do** dosomethingwith “$file” # **do** something with each file done<<(find /bar –*name* **foo** –*print*0) I think I understand the IFS bit, but I don’t understand what the ‘<<(...)’ characters mean. Obviously there’s some sort of piping going on here. It’s very hard to Google “<<”, you see.
标签	Bash, Shell

代码注释可以描述相关代码的设计意图和实现功能, 因此, 高质量的代码注释可以提高代码的可读性和可理解性, 并且在程序的开发和维护中发挥着重要作用. 近年来, 代码注释自动生成研究成为软件工程领域的一个研究热点, 我们的综述[2]对该问题的已有研究成果进行了系统梳理. 然而, 据我们所知, 大多数研究工作将研究对象设定为基于 Java 和 Python 语言的语料库, 而针对 Bash 代码的注释自动生成的研究则关注较少.

早期的代码注释生成方法借助信息检索技术来生成代码注释, 即复用语义相似代码的注释. 而最近的研究更多借鉴神经机器翻译的研究成果, 即将代码注释生成问题建模为机器翻译问题, 然后使用基于 Seq2Seq 架构[3,4]的神经网络来训练模型, 尝试学出代码与自然语言之间的隐含语义关系. 早期工作[5,6]探索了 Seq2Seq 架构, 随后, 研究人员[7−9]更多地从词法和语法的角度来学习代码的语义信息, 以生成更高质量的代码注释. 但这类方法也存在模型训练开销大、生成的注释中倾向于包含高频词, 而难以覆盖到一些低频词. 根据 Fu 和 Menzies 的建议[10], 在使用深度学习方法之前, 应该首先尝试使用计算开销小的轻量级简单方法. 此外, 基于深度学习的方法不能像基于信息检索的方法那样, 可以复用语义相似的代码注释. 因此, 本文首先考虑使用信息检索方法来解决 Bash 代码的注释自动生成问题.

和以往研究工作中提出的信息检索方法不同[11−15], 本文提出的 ExplainBash 方法基于双重信息检索. 由

于 Bash 命令的代码语法不规则，命令选项难以用语法树进行表示[1]，因此，针对 Bash 代码提取有效的语法结构信息具有一定的挑战性. 所以，ExplainBash 方法主要考虑源代码的语义信息以及词法信息，基于语义相似度(semantic similarity)和词法相似度(lexical similarity)在训练集中执行双重信息检索，找到与输入代码片段最为相似的代码片段. 具体来说，首先进行语义相似度检索，ExplainBash 使用 CodeBert[16]将 Bash 代码转换成语义向量，提取出语义特征；接着，通过 BERT-whitening 操作[17]对语义向量进行降维处理，以提高语义向量相似度的计算效果；随后，通过欧式距离计算两个代码片段之间的语义相似度，并在训练集中检索出与之语义最相似的 k 个代码片段；接着，进行词法相似度检索，受到 Yang 等人[18]和 Liu 等人[19]研究工作的启发，ExplainBash 方法将代码片段视为词元(token)集合，通过编辑距离计算两个代码片段之间的相似度. 从基于语义相似度检索出的 k 个代码片段集合中，进一步检索出词法相似度最高的代码片段，并复用该代码片段对应的代码注释.

为了评估基于双重信息检索的 ExplainBash 方法的性能，我们构造了一个高质量语料库，该语料库基于 NL2Bash[1]中共享的语料库以及 NLC2CMD 竞赛(https://eval.ai/web/challenges/challenge-page/674/leaderboard/1831)的官方数据. 我们将两者合并，并去除其中的重复数据以形成最终语料库. 与 NL2Bash[1]的研究工作相似，本文仅关注了 Linux 用户确定的 135 个最有用的实用程序[1]，也就是说，目标命令域只包含了这 135 个实用程序.

我们针对 Bash 代码的注释自动生成问题展开研究并提出了 ExplainBash 方法. 我们选择了以往代码注释自动生成领域中的最新基准方法进行了比较，这些基准方法包括基于深度学习的方法以及基于信息检索的方法. 实证研究和人本研究(humanstudy)的结果表明，在我们整理的语料库上，ExplainBash 方法相比最新的基准方法能够取得更好的性能. 除此之外，我们还设计了一系列消融实验，对 ExplainBash 方法内设定(例如检索策略、BERT-whitening 操作等)的合理性进行了分析和验证.

本文的主要贡献可总结如下.

- 提出了基于信息检索的 Bash 代码注释自动生成方法 ExplainBash，其借助基于语义相似度以及词法相似度的双重信息检索策略来生成 Bash 代码的注释.
- 整理并共享了针对 Bash 代码注释生成研究的高质量语料库，并在此语料库上展开实证研究和人本研究. 实验结果表明，本文提出的基于双重信息检索的注释生成方法 ExplainBash 的性能要优于来自代码注释自动生成领域中的基于信息检索以及深度学习的基准方法.
- 基于本文所提的 ExplainBash 方法，我们开发出一个基于 Chrome 浏览器的插件，以方便用户对 Bash 代码的理解.
- 为了方便其他研究人员重现我们的研究工作并提出更为新颖、有效的方法，我们将实验中涉及的语料库、实验代码、具体的实现结果和插件工具进行了共享(https://github.com/ExplainBash/explainbash).

本文第 1 节总结自然语言描述映射到 Bash 命令以及代码注释自动生成的相关工作，并突出本文相对已有工作的创新点. 第 2 节介绍 ExplainBash 方法的整体框架和实现细节. 第 3 节介绍实验设置内容. 第 4 节对实验结果进行深入分析. 第 5 节对 ExplainBash 方法中的一些影响因素进行讨论. 第 6 节分析对实证研究有效性构成的潜在威胁和应对策略. 第 7 节总结全文并对后续研究工作进行展望.

1 相关工作

这一节，我们首先分析了自然语言描述映射到 Bash 命令的相关工作；随后，我们重点分析了代码注释生成的已有工作；最后强调了我们研究工作的创新性.

1.1 自然语言描述映射到Bash命令

由于 Bash 代码的语法不规则，将自然语言描述映射到对应的 Bash 命令(即 NL2Bash 问题)是一个具有挑战性的研究问题，并吸引了研究人员的关注. Lin 等人[1]首次针对该问题展开了研究，并整理了 Bash 与对应自然语言描述的语料库. 具体来说，他们从开发人员问答论坛、技术教程、技术网站和课程材料等网站上搜集

了与 Bash 代码相关的数据以构建语料库, 经过数据清洗后, 共收集了超过 9 000 对数据, 覆盖了超过 100 个 Bash 实用程序. 随后, 他们考虑了 3 个神经机器翻译模型(即 Seq2Seq 模型[3,4]、CopyNet 模型[20]以及 Tellina 模型[21]), 并在 3 个不同大小的粒度上对这一任务进行评估. 实证研究结果验证了该问题具有较高的研究挑战性. Kan 等人[22]在 Lin 等人的研究基础上提出一种新的 GSAM (grid structure attention mechanism)机制, 作为 Seq2Seq 模型的一部分. 具体来说, 该机制使用双向 GRU (BidirectionalGRU)[23]作为特征提取模型, 然后将这些隐藏状态通过神经网络映射到网格结构中, 随后, 通过卷积神经网络来计算邻接特征. 他们还使用了拷贝机制[20]来缓解 OOV (outof vocabulary)问题. 与 Lin 等人提出的方法相比, 该方法在 BLEU-1/3 指标上分别提升了 7 个和 7.3 个百分点. 除此之外, D'Antoni 等人[24]开发出工具 NoFAQ, 通过学习修复命令行常见错误的示例, 构建一组规则来建议用户编写触发常见命令行错误的修复方法.

1.2 代码注释相关工作

在以往的代码注释研究工作中, 常见的方法可大致为基于信息检索的方法和基于深度学习的方法.

基于信息检索的方法是注释生成研究中最早关注的一类方法. 早期研究中, Haiduc 等人[11]最早提出了基于信息检索方法的文本摘要技术. 首次使用向量空间模型(vector space model)和潜在语义索引(latent semantic index)从语料库中检索出相关术语, 然后将检索出的术语重新组织以形成代码注释. Haiduc 等人[13]在后续研究中提出了基于代码的词法信息以及结构信息以生成代码注释, 根据潜在语义索引模型构建语料库, 然后计算出代码文本和语料库中的文本距离, 根据相似度大小检索出最相似的 5 个代码片段. Eddy 等人[12]使用 hPAM 主题模型, 从语料库中选择相关术语并将其组织到代码注释中. 由于代码复用[25,26]在大规模代码仓库中较为常见, 因此在代码复用较高的语料库中使用信息检索方法具有较好的表现. Wong 等人[27]尝试将开发人员问答网站(例如 StackOverflow)中的代码片段以及相关自然语言描述用来代码注释生成研究, 他们使用 SIM 工具(一个基于词元级别的代码克隆检测工具)来检测相似代码, 并将检测出的相似代码的注释作为最终注释. Rahman 等人[14]通过分析 9 016 个帖子和 Stack Overflow 网站上对应的答案以及评论, 他们发现, 22%的问题帖子和相关信息是有用的. 随后, 他们从 Stack Overflow 网站中提取了有效的注释, 然后基于 Stack Overflow 网站中的 292 个代码片段和相应的 5 039 个讨论, 验证了他们提出的方法的有效性. Wong 等人在后续研究[28]中提出了 CloCom 的方法, 具体来说, 他们使用基于词元的代码克隆检测工具从 Github 上检索相似代码, 并使用评论中的信息来生成注释. Liu 等人[15]提出了 NNGEN 方法, 这是一种基于最近邻的简单方法, 他们使用词袋模型(bag of words)将训练集中的代码变更(codechange)转换成向量, 随后, 基于余弦相似度选择出训练集中与代码变更最相似的 k 个代码变更. 计算出新代码变更中与这 k 个代码变更的 BLEU-4 指标评分, 将 BLEU-4 评分最高的代码变更的提交消息(commitmessage)作为新代码变更的提交消息.

为了提高模型的泛化能力, 深度学习方法被用到了最近的研究工作中. 一些工作使用 Seq2Seq 的架构将代码注释任务视为神经机器翻译(neural machine translation)[29]任务, 即将源代码作为输入、代码注释作为输出, 然后利用基于深度学习的方法来训练生成模型. Iyer 等人[7]首次提出了基于深度学习的 CODE-NN 方法, 将源代码视为词元序列, 使用 LSTM 和注意力机制构建编码器和解码器. Hu 等人[8]提出了 DeepCom 方法, 使用抽象语法树来分析 Java 方法的语义信息和结构信息, 然后提出了 SBT (structure-based traversal)方法对抽象语法树进行遍历, 并将抽象语法树转换为 AST 序列. Liang 等人[30]设计了一种新的递归神经网络 Code-RNN, 从源代码中提取语义特征, 然后使用 Code-GRU 作为解码器, 生成高质量代码注释. Ahmad 等人[31]使用 Transformer 模型来实现代码注释生成, Transformer 模型[32]是一种基于多头自注意力(multi-headed self-attention)机制的序列到序列模型, 可以有效捕获长程依赖(long-range dependencies). Yang 等人[9]提出一种基于 Transformer 的 ComFormer 方法, 该方法利用代码片段的词法信息和语法信息有效的学习代码语义; 除此之外, 他们使用 Byte-BPE 分词算法分割标识符. 最近的一些研究尝试结合信息检索技术与深度学习技术, 例如, Wei 等人[33]提出一个基于范例(exemplar)的生成框架, 该框架使用相似代码片段对应的注释作为范例来帮助神经网络模型生成目标代码的注释; Zhang 等人[34]提出一种基于检索的神经网络代码注释生成模型, 该模型使用检索到的最相似的代码片段来增强融合注意力机制的 Seq2seq 模型; Li 等人[35]则将基于信息检索获得的代

码注释视为原型(prototype), 并将原型中的模式信息与输入代码的语义信息相结合, 通过训练的神经网络自动编辑原型, 从而生成代码注释.

1.3 本文研究工作的创新性

我们针对 Bash 代码的注释自动生成问题展开研究, 而已有工作主要关注的是自然语言描述到 Bash 命令的映射. 虽然研究人员对代码注释生成展开了深入研究, 但他们关注的代码主要集中于 Java 和 Python 这两种高级编程语言. 为了对 Bash 代码的注释自动生成问题展开深入研究, 我们首先整理并构造了一个大规模的高质量语料库作为本文的实验对象; 其次, 我们提出了一种基于双重信息检索的 Bash 代码注释自动生成方法 ExplainBash, 该方法从语义相似度以及词法相似度这两个角度进行双重检索, 最终可以从代码历史库中检索出最为相似的代码片段, 然后复用该代码片段的注释. 实证研究和人本研究的结果表明, 我们所提的方法可以优于最新的基于信息检索和深度学习的基准方法. 同时, 消融实验也验证了双重信息检索策略的有效性.

2 方法介绍

本文提出的 ExplainBash 方法尝试借助信息检索方法来快速高效地检索出与输入代码功能描述相同的代码片段, 然后复用其对应的代码注释. 因此, 我们首先采用语义相似度检索技术过滤与输入代码语义不一致的代码片段, 然后采用词法相似度检索在语义相似的代码集合中检索出最相似代码片段, 从而保证检索结果的质量. 基于双重信息检索的 ExplainBash 方法的整体框架如图 1 所示, 该方法主要由两个模块构成: 语义相似度检索模块以及词法相似度检索模块. 具体来说, 首先根据语义相似度计算, 从训练集中检索出 k 个与输入代码片段语义最相似的代码片段集合; 然后, 通过词法相似度计算从上述集合中检索出相似度最高的代码片段, 将其对应的注释作为输入代码的注释. 接下来, 我们具体介绍每个模块的实现细节.

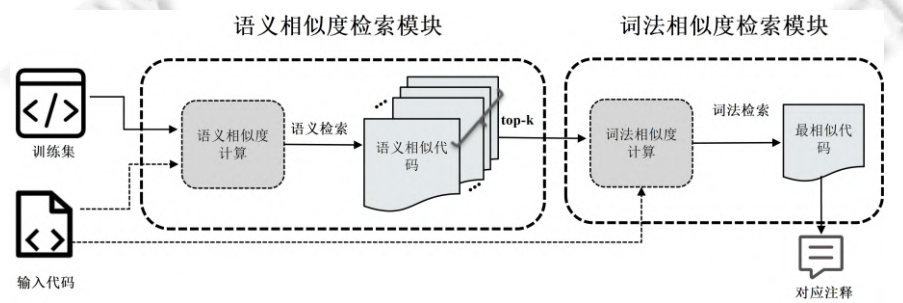

图 1 ExplainBash 的总体框架

2.1 语义相似度检索模块

在语义相似度检索模块, 鉴于 CodeBERT[16]在自然语言搜索以及代码文档生成任务中的优秀表现, 我们使用 CodeBERT 模型对代码片段进行语义提取, 并通过 BERT-whitening 操作[17]对提取的语义向量进行降维处理, 随后通过欧式距离计算语义相似度, 并根据语义相似度从训练集中检索出 k 个代码片段以组成集合. 当 k=8 时, 我们所提的方法在实验中可以取得最好的性能(超参数 k 的取值影响分析见第 5.1 节). 图 2 给出了语义相似度检索的过程.

CodeBERT[16]是基于编程语言和自然语言的双模态预训练模型, 该模型使用基于 Transformer 的神经网络构建而成, 它可以捕捉自然语言和编程语言之间的语义连接, 并输出可广泛支持自然语言-编程语言理解任务的通用表示. 研究人员使用混合目标函数来训练 CodeBERT, 包括掩码语言模型(masked language modeling)以及替换词元检测(replaced token detection).

其中, 掩码语言建模将自然语言(natural language, NL)-编程语言(program language, PL)对(x={w,c})作为输入, w 是一个 NL 单词序列, c 是一个 PL 的词元序列. 随机为 NL 和 PL 选择位置进行掩码(mask), 然后用特殊

的掩码词元进行替换. 掩码语言模型的目标是预测出被掩码的原始词元, 其损失函数 $L_{MLM}(\theta)$可定义如下:

$$L_{MLM}(\theta)=\sum_{i\in m^w\cup m^c}-\log p^{D_1}(x_i|\ w^{masked},c^{masked}) \tag{1}$$

其中, p^{D_1} 是从一个从词汇表中预测词元的区分器.

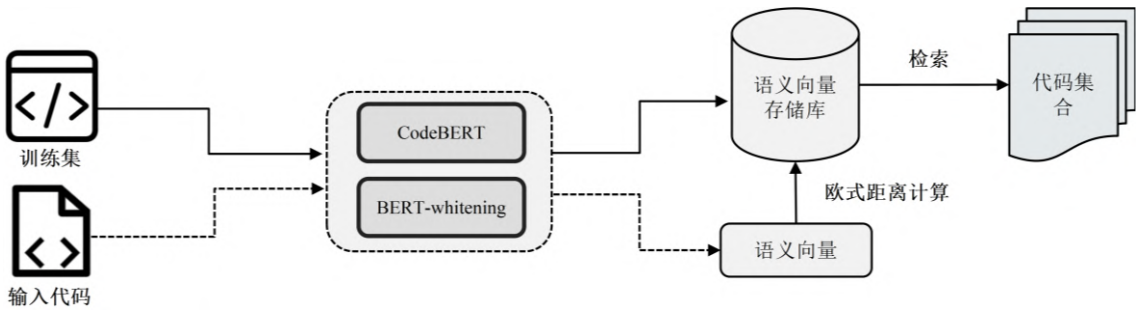

图 2 基于语义相似度的检索流程

替换词元检测首先分别用单模的自然语言和代码数据各自训练出一个 NL 生成器(NL generator) p^{G_w} 和一个 PL 生成器(code generator) p^{G_c}, 用于为随机掩码位置生成合理的备选方案, 对 PL 以及 NL 随机掩码后的 m^w 和 m^c 进行计算, 其中,

$$\hat{w}_i\sim p^{G_w}(w_i\mid w^{masked})\ \text{for}\ i\in m^w \tag{2}$$

$$\hat{c}_i\sim p^{G_c}(c_i\mid c^{masked})\ \text{for}\ i\in m^c \tag{3}$$

根据 $\hat{w}_i$ 和 $\hat{c}_i$, 可以求出 $w^{corrupt}$ 以及 $c^{corrupt}$, 其中, $w^{corrupt}$=*REPLACE*(w,m^w, $\hat{w}$), $c^{corrupt}$=*REPLACE*(c,m^c, $\hat{c}$), *REPLACE*(·)表示使用掩码词元替换所选位置:

$$x^{corrupt}=w^{corrupt}+c^{corrupt} \tag{4}$$

而判别器通过学习自然语言和代码之间的融合表示来检测一个词是否为原词. 判别器实际上是一个二元分类器, 如果生成器产生正确的词元, 则该词元的标签为真, 否则为假.

其损失函数 $L_{RTD}(\theta)$可定义如下:

$$L_{RTD}(\theta)=\sum_{i=1}^{|w|+|c|}(\delta(i)\log p^{D_2}(x^{corrupt},i)+(1-\delta(i))(1-\log p^{D_2}(x^{corrupt},i))) \tag{5}$$

其中, $\delta(i)=\begin{cases}1, & \text{if}\ x_i^{corrupt}=x_i\\ 0, & \text{otherwise}\end{cases}$ 表示指示函数, p^{D_2} 表示预测第 i 个单词为原词概率的判别器.

综上所述, CodeBERT 模型预训练的总目标为: $\min_\theta L_{MLM}(\theta)+L_{RTD}(\theta)$.

模型训练的最后一步是模型微调, 具体操作是在 NL-PL 任务中使用不同的 CodeBERT 设置. 例如: 自然语言代码搜索中, 会使用与预训练阶段相同的输入方式; 而在代码到文本的生成中, 使用编码器-解码器框架, 并使用 CodeBERT 初始化生成模型的编码器.

虽然 CodeBERT 只在 Go, Java, JavaScript, PHP, python 以及 Ruby 共 6 种语言进行预训练, 但在已有研究中, CodeBERT 已经被证明在新的编程语言上也具有较强的泛化能力[16,18], 因此根据迁移学习的思想, 本方法使用 CodeBERT 对 Bash 语句进行特征提取.

BERT-whitening[17]操作处理语义向量已移除数据的冗余信息, 希望获取更好的性能以及提高模型的检索速度. BERT-whitening 操作的目的是对向量集合 $\{x_i\}_{i=1}^N$ 进行线性变换, N 为集合中的向量数:

$$\tilde{x}_i=(x_i-\mu)W \tag{6}$$

使得 $\{\tilde{x}_i\}_{i=1}^N$ 是均值为 0, 协方差矩阵为单位矩阵. 其中, 可以根据文献[17]计算出 W 和μ.

语义相似度检索过程具体如下: 我们首先获取 Bash 代码序列的长度 M, 然后将其输入到 CodeBert 中, 提取输出的第 1 层的隐藏状态 h^0 和最后一层的隐藏状态 h^n, 并将其求和后进行平均, 其中, n 为隐藏层层数, 得

到代码的语义特征向量 X:

$$X=avg(h^0+h^n),\ X\in\mathbb{R}^d \tag{7}$$

其中, d 是 CodeBERT 中的 *hidden_size* 参数. 上述方法可以将训练集中的代码转换为向量集合 $\{X_i\}_{i=1}^N$, 此时, N 为训练集大小.

我们首先根据训练集得到向量集合 $\{X_i\}_{i=1}^N$, 然后通过 BERT-whitening 操作转换成相应的 $\{\tilde{X}_i\}_{i=1}^N$. 同样地, 对于测试集的 X_{test}, 我们使用上述的 W 和 μ, 计算出对应的 $\tilde{X}_{test}$.

最后, 我们通过欧式距离计算得到向量 $\tilde{X}_i$ 和向量 $\tilde{X}_{test}$ 的语义相似度 $semantic\_similarity(\tilde{X}_i,\tilde{X}_{test})$:

$$semantic\_similarity(\tilde{X}_i,\tilde{X}_{test})=\sqrt{\sum_{j=1}^{n}(\tilde{X}_i[j]-\tilde{X}_{test}[j])^2} \tag{8}$$

通过语义相似度计算, 可以在训练集中检索出 k 个与输入代码片段语义最相似的代码片段, 并在 k 个代码片段组成的集合内, 进行后续的词法相似度检索.

2.2 词法相似度检索模块

在词法相似度检索模块中, 首先遍历语义相似度检索模块输出的代码片段集合, 将遍历出的代码片段和输入代码片段, 基于编辑距离计算两者之间的相似度. 我们将代码片段视为词元构成的集合, 根据计算的文本编辑距离确定词法相似度. 图 3 显示了词法相似度的检索过程.

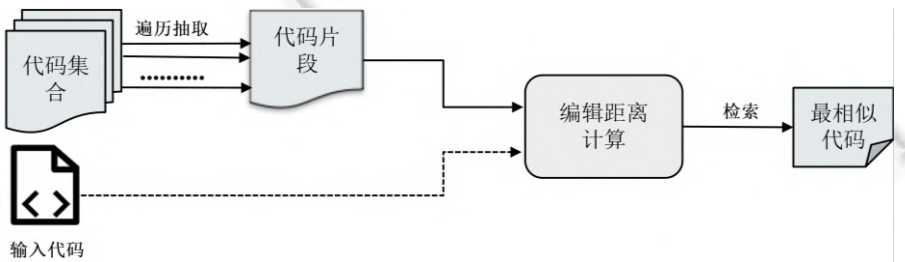

图 3 基于词法相似度的检索流程

编辑距离[36]又被称为 Levenshtein Distance, 是用来度量两个序列相似程度的指标. 具体来说, 编辑距离指的是将一个序列 w_1 转换成另一个序列 w_2 时, 所需要的最少编辑操作次数, 即用插入、删除和替换这 3 种操作将 w_1 转换到 w_2 的最少操作次数.

对于给定的两个代码片段, 首先对代码进行分词、词形还原等文本预处理操作, 然后对代码序列内的重复词元进行删除, 获取代码片段的词元集合 A 和 B, 最后计算出集合 A 和集合 B 的编辑距离 $dis_{A,B}$:

$$dis_{A,B}(i,j)=\begin{cases}\max(i,j), & \text{if } \min(i,j)=0\\ \min\begin{cases}dis_{A,B}(i-1,j)+1\\ dis_{A,B}(i,j-1)+1\\ dis_{A,B}(i-1,j-1)+1\end{cases}, & \text{otherwise}\end{cases} \tag{9}$$

其中, i, j 分别表示集合 A 和 B 的长度. 需要注意的是, 在计算编辑距离时, 我们将删除操作和插入操作视为一个操作次数, 而替换操作视为两个操作次数.

和 Liu 等人的方法[19]一样, 我们通过计算出 A 和 B 的编辑距离, 就可以进一步得到 A 和 B 的词法相似度 $lexical\_similarity(A,B)$:

$$lexical\_similarity(A,B)=1-\frac{dis_{A,B}}{\max(len(A),len(B))} \tag{10}$$

其中, $len(\cdot)$表示集合的规模. 根据计算, 可以在语义相似的 k 个代码片段的集合中检索出词法相似度最高的代码片段. 我们将经过上述双重信息检索方法检索出的 Bash 代码片段视为是输入代码片段的最相似代码, 并将

其对应的代码注释作为输入代码的注释.

3 实验设置

为了验证本文提出的基于双重信息检索的 Bash 代码注释生成方法 ExplainBash 的有效性, 我们设计了如下 3 个实验问题并分析了这些问题的设计动机.

- RQ1: 我们提出的 ExplainBash 方法的在 Bash 代码注释生成中的性能是否优于基准方法?

动机: 和以往研究使用的信息检索方法不同, 本文提出的 ExplainBash 方法考虑了 Bash 代码的语义信息和词法信息, 通过双重信息检索确定与输入代码片段最为相似的代码片段. 为了评估本文所提方法的有效性, 我们首先选择代码注释自动生成研究中基于信息检索的方法作为基准方法. 除此之外, 鉴于当前代码注释自动生成的研究工作主要集中于深度学习方法, 我们也将 ExplainBash 方法与基于深度学习的基准方法进行了比较. 我们针对该问题展开研究, 在该实验问题中主要选择来自代码注释生成领域的经典基准方法, 并分析这些方法在 Bash 代码注释生成问题中的性能. 为了自动评估这些方法生成的注释与语料库中参考注释的相似度, 我们主要考虑了 BLEU[37]、METEOR[38]和 ROUGR-L[39]这 3 个评估指标.

- RQ2: ExplainBash 方法生成的代码注释在人本研究中的表现如何?

动机: 虽然采用自动评测指标可以评估生成注释和参考注释间的差异, 但有时候也不一定能真实体现两者之间的语义相似度[39]. 因此在该实验问题中, 我们尝试通过人本研究(human study), 从语义角度来手工评测分析不同方法生成的注释质量.

- RQ3: ExplainBash 方法内使用的检索策略是否能提高生成注释的质量?

动机: ExplainBash 方法借助双重信息检索来选择最为相似的代码片段. 在该实验问题中, 我们设计了消融实验(ablationstudy)对本文考虑的双重信息检索方法的有效性进行分析, 从而评估语义相似度检索模块以及词法相似度检索模块对本文所提方法性能的影响. 除此之外, 我们还从双重信息检索策略的顺序对方法性能的影响进行了讨论.

3.1 语料库

为了验证所提出的方法, 我们构建了一个高质量的语料库. 语料库中, 单个数据由⟨Bash 代码,自然语言描述⟩构成. 据我们所知, NL2Bash[1]是首个将自然语言描述映射到 Linux 中 Bash 命令的研究工作, 该研究工作共享了一个高质量的 Bash 代码和对应自然语言描述的语料库, 后续针对 Bash 代码的相关研究工作[22,40]也均使用了该语料库作为他们的研究对象. 为了更好地评估方法的泛化能力, 我们还整合了 NLC2CMD 竞赛提供的数据来丰富语料库. 我们将两者进行合并, 然后去除其中重复的数据, 最终形成了一个规模更大的语料库, 该语料库共包含 10 592 个数据.

我们对该语料库中 Bash 代码以及对应注释的长度进行统计, 图 4 基于直方图给出了 Bash 代码段以及代码注释的长度统计信息.

基于该图我们可以发现, 语料库中, 大多数 Bash 代码的长度不超过 20, 长度主要集中在 8 左右, 而相对应的注释长度也主要集中在 10 左右.

表 2 和表 3 进一步给出了语料库中 Bash 代码长度和注释长度的统计信息.

由于信息检索的方法并不需要进行模型训练, 为了将 ExplainBash 方法和基于深度学习的方法进行对比, 验证 ExplainBash 方法的性能, 我们根据 8:1:1 的比例划分训练集、验证集以及测试集. 在信息检索模块中使用训练集以及测试集, 在深度学习模块使用训练集、验证集以及测试集.

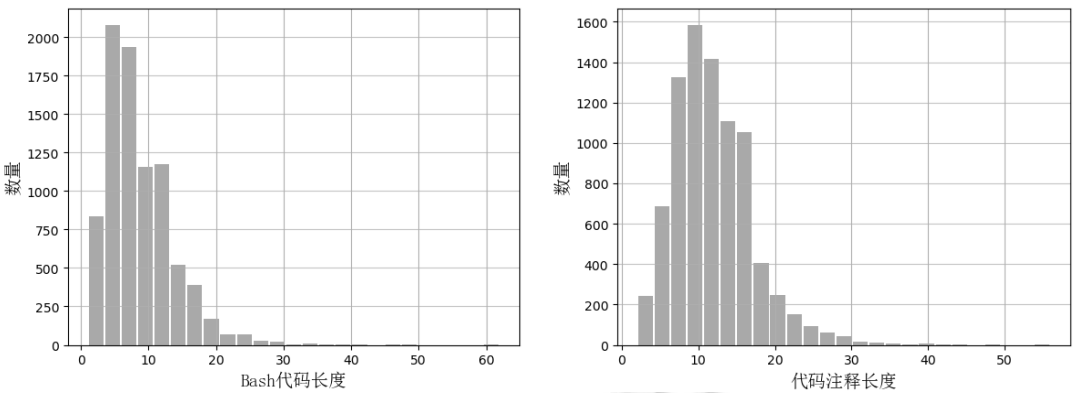

图 4 语料库中 Bash 代码和注释长度的统计信息

表 2 Bash 代码长度的统计信息

均值	众数	中位数	<16	<32	<48
8.528	4	7	0.908	0.997	0.999

表 3 代码注释长度的统计信息

均值	众数	中位数	<16	<32	<48
11.874	10	11	0.803	0.995	0.999

3.2 性能评估指标

为了评价我们提出方法的性能, 我们选择了 3 种性能指标, 包括 BLEU[37]、METEOR[38]以及 ROUGE-L[39]. 这些指标被广泛使用在机器翻译研究以及代码注释自动生成的相关研究中. 指标的详细说明如下.

- BLEU[37]: 其全称是 Bilingual Evaluation Understudy.

这是一种最早提出的机器翻译评价指标. BLEU 是一种基于精确度的相似性度量方法, 用于分析候选文本和参考文本中 *n* 元组(*n*-gram)共同出现的程度, 经常用来评估神经机器翻译模型的性能. 本质上, BLEU 是一个 *n*-gram 精确度的加权几何平均, 最后的结果是候选文本的统计和参考文本中,*n* 元组(*n*-gram)正确匹配次数与其中所有 *n* 元组出现次数的比值:

$$Count_{clip}=\min(Count,Max\_Ref\_Count) \tag{11}$$

其中, *Count* 是 *n* 元组在候选文本中出现次数, *Max_Ref_Count* 是该 *n* 元组在参考文本中出现的最大次数. 最终统计两者之间的最小值, 然后再将匹配结果除以候选文本中 *n* 元组的个数, 其计算 *n* 元组情况下的精度 p_n 的公式为:

$$p_n=\frac{\sum_{n\text{-}gram\in candidate}Count_{clip}(n\text{-}gram)}{\sum_{n\text{-}gram'\in candidate}Count_{clip}(n\text{-}gram')} \tag{12}$$

虽然上述改进可以使用 BLEU 有效地评估翻译质量, 然而 *n* 元组的匹配程度可能会随着句子长度的变短而变高, 因此可能会因为只翻译出参考文本中部分句子而导致匹配度很高. 为了避免这种偏向性, BLEU 在最后的评分中引入了长度惩罚因子(brevity penalty)来惩罚候选文本过短的问题, 其计算公式定义如下:

$$BP=\begin{cases}1, & \text{if } l_c>l_s\\ \mathrm{e}^{\left(1-\frac{l_s}{l_c}\right)}, & \text{if } l_c\leqslant l_s\end{cases} \tag{13}$$

其中, l_c 表示候选文本的长度, l_s 表示参考文本的长度. 因此, 最终的 BLEU 分数计算公式为:

$$BLEU=BP\cdot\exp\left(\sum_{n=1}^{N}W_n\log p_n\right) \tag{14}$$

n 的上限取值为 4, 即最多只能统计 4 元组的精度.

- METEOR[38]: 其全称是 Metric for Evaluation of Translation with Explicit Ordering.

METEOR 指标基于单精度的加权调和平均数和单字召回率, 目的是解决 BLEU 指标中固有的缺陷. 其使用 WordNet (https://wordnet.princeton.edu/)等知识源来扩充同义词集, 同时考虑了单词的词形. 在评价句子流畅度时, 使用了 chunk(即候选文本和参考文本能够对齐的、并且空间排列上连续的单词形成一个 chunk)的概念. chunk 的数目越少, 则意味着每个 chunk 的平均长度越长, 即候选文本和参考文本的语序越一致.

在计算 METEOR 指标值时, 首先找到候选文本和参考文本之间的最优匹配, 然后计算准确率 P 和召回率 R, 然后计算两者的参数化调和平均值 F_{mean}, 其计算公式定义如下:

$$F_{mean}=\frac{P\cdot R}{\alpha\cdot P+(1-\alpha)R} \tag{15}$$

其中, α为可调控参数. 接着计算惩罚因子 Pen:

$$Pen=\frac{\#chunks}{m} \tag{16}$$

这里的$\#chunks$ 表示 chunk 的数量, m 表示候选文本中能够匹配的一元组(1-gram)的数量. 则 METEOR 的计算公式可以表示为:

$$METEOR=(1-Pen)\times F_{mean} \tag{17}$$

- ROUGE-L[39]: 其全称是 Recall-Oriented Understudy for Gisting Evaluation.

这是一个基于 *Recall* 召回率的度量指标, 用来计算候选文本和参考文本之间的最长公共子序列的长度. 长度越长, ROUGE-L 的分数就越高. 首先计算召回率 R_{lcs} 以及准确率 P_{lcs}:

$$R_{lcs}=\frac{LCS(X,Y)}{m} \tag{18}$$

$$P_{lcs}=\frac{LCS(X,Y)}{n} \tag{19}$$

其中, $LCS(X,Y)$是 X 和 Y 的最长公共子序列的长度, m,n 分别代表参考文本以及候选文本的长度. 则 ROUGE-L 的计算过程如下:

$$ROUGE\text{-}L=\frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs}+\beta^2P_{lcs}} \tag{20}$$

不难看出, BLEU、METOR 和 ROUGE 指标的取值范围介于 0–1 之间, 且经常以百分比的形式给出. 其取值越高, 表示方法的性能越好.

为了避免由于这些性能评估指标的不同实现版本而导致的结果差异, 在本文涉及的实验中, 我们均使用了 nlg-eval 库(https://github.com/Maluuba/nlg-eval)提供的计算方法来实现上述 3 种性能评估指标, 这可以确保实证研究结论的有效性.

3.3 工具实现

为了帮助开发人员能快速理解 Bash 代码的功能, 我们基于 ExplainBash 方法开发出一个浏览器插件, 并将该插件集成在 Chrome 浏览器中. 用户可以在浏览器中快速启动该插件, 并将需要生成注释的 Bash 代码复制到输入框中, 并随后点击生成注释的按钮, 最终可以生成对应的代码注释. 图 5 展示了该浏览器插件的界面截图.

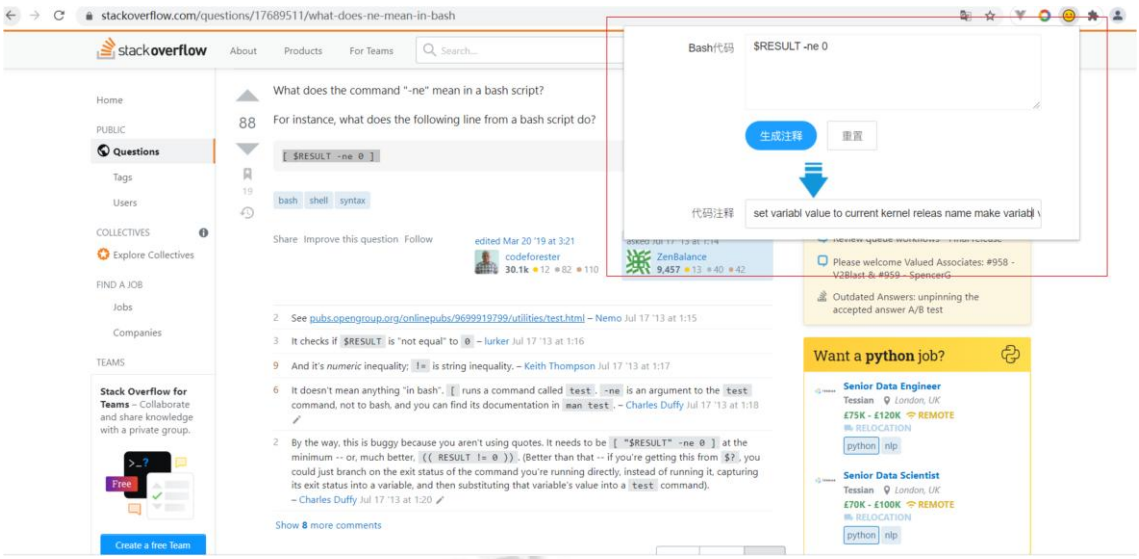

图 5　本文工具的界面截图

3.4 基准方法

我们针对 Bash 代码注释生成问题展开研究，但并没有找到该问题的基准方法可以进行比较. 为了评估本文所提的 ExplainBash 方法的性能，我们首先选择代码注释生成领域中基于信息检索的方法进行比较，包括 LSI[13]、VSM[11]和 NNGEN[15]方法. 除了这些基于信息检索的方法外，越来越多的代码注释研究使用深度学习模型，如: NL2Bash 研究[1]中表现最佳的 Copynet 模型[20]; Iyer 等人[7]首次提出了基于深度学习的 CODE-NN 方法; Ahmad 等人[31]利用 Transformer 模型来生成代码注释，并且证明了 Transformer 在代码注释生成研究中具有较好的性能. 为了更好地评估在 Bash 代码语料库中深度学习方法的表现，我们首先选择了 NL2Bash 研究中表现最佳的 CopyNet 模型作为我们的基准方法，然后也选择 Transformer[32]和 CODE-NN[7]作为比较对象. 我们尝试在 Transformer 模型训练过程中加入对抗训练方法 FGM (fast gradient method)[41]以提高模型的稳健性，并作为基准方法进行比较. 除此之外，我们还使用 Code BERT 构建的 Seq2seq 模型也作为基准进行比较. 除了传统的信息检索方法以及深度学习方法以外，我们还选择了同时利用信息检索以及深度学习的融合方法 Rencos[34]进行了对比实验. 接下来依次简要介绍论文考虑的 9 种基准方法.

- LSI[13]: 是最早的基于信息检索的代码注释自动生成方法. LSI 首先构建一个语料库，其中每个文档对应于一种方法. 然后计算方法文本与语料库中每个单词之间的距离，利用代码中的词汇和结构信息进行检索.
- VSM[11]: 将文本内容的处理简化为向量空间中的向量操作，并根据向量相似性计算语义相似性. 给定一个文档，它将根据单词的权重从中提取前 k 个单词，并将这些单词视为代码注释.
- NNGEN[15]: 基于最近邻算法生成提交消息. 该方法从余弦相似性和 BLEU-4 两个方面对训练集的差异进行排序，并选择训练中不同得分最高的提交消息.
- CopyNet[20]: 该方法将复制机制引入到基于编码器-解码器结构的模型，复制机制可以很好地选择输入序列中的子序列，并将它们放在输出序列的适当位置. 该模型也是当前 NL2Bash 研究中[1]性能表现最佳的模型.
- Transformer[32]: 基于多头自注意力机制(multi-head-self-attention mechanism)和位置编码(position encoding)的编码器-解码器框架，可以用来生成输入代码的代码注释.
- FGM[41]: 对抗训练是一种引入噪声的训练方式，在对抗训练过程中，对样本添加扰动. 语言模型领域中，增加对抗训练可以提高神经网络的稳健性以及泛化能力.

- CODE-NN[7]: 将源代码作为词元序列, 使用 LSTM 和注意力机制构建编码器和解码器, 这与 NL2Bash 研究中使用的 Seq2seq 模型的结构基本相同;
- CodeBERT[16]: CodeBERT 是基于编程语言和自然语言的双模态预训练模型, 在代码搜索以及代码文档生成等任务上具有良好表现[16]. 该方法使用 CodeBERT 模型作为编码器来构建 Seq2seq 模型;
- Rencos[34]: 该方法同时利用信息检索方法与深度学习方法的代码注释生成模型, 该模型使用检索到的最相似的代码片段来增强考虑了注意力机制的 Seq2seq 模型.

3.5 方法实现细节

研究中涉及的实验部分均基于 PyTorch 框架实现, 通过使用 Faiss 库(https://github.com/facebookresearch/faiss)、textdistance 库(https://github.com/life4/textdistance)以及 Transformer (https://github.com/huggingface/transformers)来实现我们提出的方法. 表 4 给出了实验涉及的超参数和具体的取值, 这些取值基于已有文献的推荐取值和我们实证研究中的实际性能. 除此之外, 我们根据基准方法的描述重新实现了这些方法, 并且这些方法的执行结果与原始文献中的结果接近. 实验运行的计算机的配置信息是: Inter 4210 CPU、24 GB 内存的 GeForceRTX3090 GPU、Windows 操作系统.

表 4 我们研究中涉及的超参数和对应取值

超参数名称	参数值
CodeBERT 的 embedding 维度 d	768
BERT-whitening 后的维度 d'	256
代码集合 k 的值	8

4 结果分析

4.1 RQ1结果分析

- RQ1: 我们提出的 ExplainBash 方法在 Bash 代码注释生成中的性能是否优于基准方法?

为了评估 ExplainBash 方法的性能, 我们选用了代码注释自动生成领域中基于信息检索的方法以及基于深度学习的方法作为我们的基准方法. 表 5 给出了 ExplainBash 方法和选择的基准方法的评估结果, 我们对最好的结果进行了加粗. 可以看到, 和所有的基准方法相比, 在 BLEU、METEOR 以及 ROUGR-L 指标下, ExplainBash 方法的性能均为最高, 这表示我们提出的 ExplainBash 方法可以针对 Bash 代码生成更高质量的代码注释.

表 5 我们所提的 ExplainBash 方法和基准方法的比较结果

类型	方法	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)
信息检索	LSI	30.18	18.07	12.48	9.40	18.30	28.82
	VSM	36.16	24.47	18.62	15.25	22.04	34.58
	NNGEN	50.62	38.75	32.11	27.85	27.69	45.88
深度学习	CopyNet	38.11	27.06	20.67	16.43	22.06	40.18
	Transformer	46.39	33.37	25.42	19.97	25.22	44.01
	Transformer+FGM	48.85	35.84	27.94	22.55	26.40	46.43
	CODE-NN	49.60	37.18	29.53	24.17	26.85	47.21
	CodeBERT	48.65	37.02	29.84	24.83	27.16	47.36
融合方法	Rencos	46.27	35.11	28.66	24.39	25.82	45.06
信息检索	ExplainBash	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**

与基于信息检索的方法相比, NNGEN 的方法在 Bash 代码注释自动生成上的性能与 LSI 方法和 VSM 方法相比具有显著提升. 这是由于 NNGEN 方法通过余弦相似度计算检索出最近邻代码, 然后通过计算 BLEU 值来选择出最相似代码, 这种方法比 LSI 以及 VSM 方法直接检索出目标代码的方式能够获取更好的性能. 和 NNGEN 方法相比, ExplainBash 方法在 BLEU-1/2/3/4 指标上分别提升了 1.19, 1.77, 1.85 和 1.77 个百分点, 在 METEOR 以及 ROUGE-L 指标上分别提升了 0.76 和 1.88 个百分点. 这表明我们提出的双重信息检索方法生成

的注释质量要好于基于信息检索的基准方法. 随后我们通过一些示例进行分析, 例如对于 Bash 代码“find . –*mindepth* 1 –*mmin* –60 | xargs –*r ls* –*ld*”, 该代码的参考注释为“list all files from the current directory tree that were modified less than 60 minutes ago”. ExplainBash 方法与 NNGEN 方法检索出的注释均和参考注释一致. 而对于长度较短的 Bash 代码来说, 例如“cat *my_script.py*”, 其参考注释为“print the contents of ‘*my_script.py*’”, ExplainBash 方法通过双重信息检索生成的注释为“display content of ‘*my_script.py*’”, 而 NNEGN 方法生成的注释为“save content of ‘*my_script.py*’ to variable”. 很明显, ExplainBash 方法可以生成更高质量的注释.

与基于深度学习的方法相比, 在 NL2Bash 研究中表现最佳的 CopyNet 模型在所选择的基于深度学习的基准方法中性能表现最差, 因此将 NL2Bash 研究中采用的方法直接进行序列的两端调换并不可行. 加入对抗训练方法 FGM 后的 Transformer 模型和 Transformer 模型相比性能有较大提升, BLEU-1/2/3/4 指标分别提升了 2.46, 2.47, 2.52 和 2.58 个百分点, 在 METEOR 和 ROUGE-L 指标上分别提升了 1.18 和 2.42 个百分点. 在基于深度学习的方法中, CODE-NN 方法以及使用 Code BERT 模型生成注释的方法性能最好. 和 CODE-NN 方法相比, ExplainBash 方法在 BLEU-1/2/3/4 指标上分别提升了 2.21, 3.34, 4.43 和 5.45 个百分点, 即分别提升了 3.4%, 7.4%, 13.1%和 20.2%的性能. 同时, 在 METEOR 和 ROUGE-L 指标上也具有较大的提升. 和 CodeBERT 方法相比, ExplainBash 方法在 BLEU-1/2/3/4 指标上分别提升了 3.16, 3.50, 4.12 和 4.79 个百分点, 即分别提升了 6.4%, 9.4%, 13.8%和 19.3%的性能. 同时, 在 METEOR 和 ROUGE-L 指标上也有提升. 与基于信息检索和深度学习的混合方法 Rencos 相比, ExplainBash 方法在 BLEU-1/2/3/4 指标上分别提升了 5.54, 5.41, 5.30 和 5.23 个百分点, 即分别提升了 11.9%, 15.4%, 18.4%和 21.4%的性能, 而在 METEOR 和 ROUGE-L 指标上也有明显提升.

随后, 我们通过一些示例进行分析. 例如 ExplainBash 方法对代码“*grep –v′*^\s*$′ *.py|wc*”生成的注释为“counts non-blank lines (lines with spaces are considered blank) in all *.*py* files in a current folder.” (该注释与参考注释一致), 而基于深度学习的方法 CODE-NN 训练后生成的注释为“count number of line in file and save result in variable”. 不难看出, 基于检索的 ExplainBash 方法生成的注释质量更高, 而深度学习方法生成的代码注释只能包含参考注释中的部分内容. 这是因为深度学习模型在训练时一般需要大量数据, 和现有代码注释研究中常见基于 Java 和 Python 编程语言的语料库相比, 我们使用的 Bash 代码注释生成的语料库规模还较小, 这是基于深度学习的方法在我们实证研究中性能并不理想的一个可能原因. 同时, 根据第 3.1 节中对语料库的统计信息分析来看, Bash 代码的平均长度为 8.528. 由于大部分 Bash 代码片段的长度较短, 基于深度学习的方法很难在短文本中学出语义信息. 而基于双重信息检索的 ExplainBash 方法则可以有效缓解该问题, 其可以通过语义相似度和词汇相似度从语料库中检索出相似代码, 从而生成高质量注释. 因此, 根据本文实证研究的结果, 对于本文关注的 Bash 代码注释生成问题, 本文所提的基于双重信息检索的方法是当前针对 Bash 代码注释自动生成问题的最好解决方案.

4.2 RQ2结果分析

- RQ2: ExplainBash 方法生成的代码注释在人本研究中的表现如何?

基于 RQ1 中的分析结果我们发现, 基于自动性能评估指标[37−39], ExplainBash 方法的性能要超过本文考虑的 9 种基准方法. 虽然采用自动评测指标可以评估生成注释和参考注释间的差异, 但有时候也不一定能真实体现两者之间的语义相似度[42]. 因此, 我们希望基于人本研究对生成的注释质量进行评估.

我们招募了两名计算机科学与技术专业的硕士生对 ExplainBash 方法生成的注释进行人本研究. 这两名学生均熟悉 shell 命令以及 Linux 操作系统开发, 评分过程采用盲审形式, 即招募的硕士生不会被告知代码注释对应的生成方法, 因此可以保证人本研究结论的可靠性.

我们的人本研究从注释的“流畅度”以及“正确性”这两个角度对代码注释进行评审, 其中:

- 流畅度是指注释需要使用流畅并且语法正确的自然语言去描述, 以方便开发人员对代码的理解.
- 正确性是指注释需要体现代码的设计意图和实现的主要功能.

我们将数据分成 3 组: 原始 Bash 代码片段以及对应的注释、CODE-NN 方法生成的注释以及 ExplainBash

方法生成的注释(因为 CODE-NN 是深度学习方法中性能最好的方法之一, 我们希望从人工角度分析信息检索方法和深度学习方法生成注释的不同). 根据原始代码片段的内容, 分别对 CODE-NN 方法以及 ExplainBash 方法生成的注释进行人工评分, 具体操作如下: 我们首先从测试集中随机抽取 150 条 Bash 代码以及对应注释作为对照组, 再从 CODE-NN 方法以及 ExplainBash 方法生成的注释中抽取出对应的注释作为研究的样本数据. 两名硕士生对相同的样本数据根据代码注释的质量进行评分, 共有 3 种评分: “不符合”、“基本符合”以及“完全符合”. 同时, 为了减少长时间标注对实验结果带来的误差, 本次人本研究要求参与人员每人半天仅标注 50 条数据. 由于“流畅度”评审只需要从语法以及是否能流畅阅读的角度进行评审, 为了方便参与人员更快地熟悉样本数据, 本次研究先进行“流畅度”角度评审再进行“正确性”角度的评审, 在研究期间, 参与人员可以在互联网上搜索相关信息以及不熟悉的概念内容. 最终, 本次人本研究共花费了 3 天时间.

表 6 总结了两位硕士生从“流畅度”角度对 CODE-NN 方法以及 ExplainBash 方法生成注释的评分, CODE-NN 方法生成的注释中“不符合”的比例占 33%左右, 而 ExplainBash 方法生成的代码注释中“不符合”的比例仅占 9%左右. 因此, 本文所提的 ExplainBash 方法生成的注释要显著优于基准方法 CODE-NN 生成的注释. 这是由于 ExplainBash 方法是基于双重信息检索的方式从语料库中检索出相似代码并复用其注释, 因此生成的注释的流畅度与语料库质量有关. 而我们实证研究中使用的语料库是由 NL2Bash[1]以及 NLC2CMD 竞赛数据共享的高质量语料库, 绝大部分代码注释由开发人员手工编写且符合语法规则, 因此本文所提方法生成的注释的流畅度相对较高.

表 6 代码注释流畅度评分(括号内为所占比例)

	方法	不符合	基本符合	完全符合
硕士 1	CODE-NN	31 (20.67%)	74 (49.33%)	45 (30%)
	ExplainBash	16 (10.67)	77 (51.33%)	57 (38%)
硕士 2	CODE-NN	35 (23.33%)	67 (44.67%)	48 (32%)
	ExplainBash	12 (8%)	71 (47.33%)	67 (44.67%)

表 7 总结了两位硕士生从“正确性”角度对 CODE-NN 方法以及 ExplainBash 方法生成注释的评分, CODE-NN 方法生成的注释中“不符合”的比例占 39%左右, 而 ExplainBash 方法生成的代码注释中“不符合”的比例占在 27%左右. 因此不难发现, 与 RQ1 中实证研究的结果相同, ExplainBash 方法在人本研究中, 其生成的注释可以更好地描述 Bash 代码的实现意图和实现功能.

表 7 代码注释正确性评分(括号内为所占比例)

	方法	不符合	基本符合	完全符合
硕士 1	CODE-NN	62 (41.33%)	65 (43.33%)	23 (15.33%)
	ExplainBash	39 (26%)	70 (46.67%)	41 (27.33%)
硕士 2	CODE-NN	57 (38%)	75 (50%)	18 (12%)
	ExplainBash	41 (27.33%)	77 (51.33%)	32 (21.33%)

我们参考以往工作[33,43]中人本研究的设定, 将参与人员的评分结果进行量化. 我们规定“不符合”为 0 分, “基本符合”为 1 分, “完全符合”为 2 分. 表 8 为最终 CODE-NN 方法以及 ExplainBash 方法的“流畅性”以及“正确性”得分. 基于该表格我们可以发现, 无论是“流畅性”角度还是“正确性”角度, ExplainBash 方法的得分均高于 CODE-NN 方法. 除此之外, 我们使用 Kappa 系数[44]来衡量这两名学生评估结果之间的一致性, 最终计算出的 Kappa 系数值均大于 0.85, 这表明了参与人员之间的评估结果具有一致性.

表 8 代码注释基于流畅度和正确性的得分(括号内为标准差)

	流畅性	正确性
CODE-NN	1.09 (0.72)	1.48 (0.68)
ExplainBash	**2.64 (0.63)**	**1.95 (0.71)**

4.3 RQ3结果分析

- RQ3: ExplainBash 使用的检索策略是否能提高生成注释的质量?

基于 RQ1 中实证研究的结果我们可以发现, Explain Bash 方法的性能要超过传统的基于信息检索的方法以及基于深度学习的方法. 该方法中主要考虑了语义信息以及词法信息, 为了验证这种基于双重信息检索策略的有效性, 我们设计了消融实验对其有效性进行验证. 在保证运行环境以及其余参数设置不变的情况下, 我们只使用语义相似度检索的策略(即 ExplainBash-Semantic)、只使用词法相似度检索的策略(即 ExplainBash-Lexical)以及先进行词法相似度检索再进行语义相似度检索的方法(即 ExplainBash-Reverse)对同一个训练集进行实验, 并将这 3 种方法设置为对照组.

表 9 给出了 ExplainBash 方法和仅使用单检索策略方法的比较结果, 并对最好的结果进行了加粗, 其中, Time 为对整个测试集进行检索使用的时间. 可以看到, 和仅使用单检索策略的两种方法相比, ExplainBash 方法生成的注释在各项指标上均可以获取更好的结果. ExplainBash 方法和 ExplainBash-Lexical (只使用词法相似度检索)方法相比, 基于双重信息检索的 ExplainBash 方法在 BLEU-1/2/3/4 指标分别提升了 3.7%, 5.3%, 6.8%以及 7.9%, 在 METEOR 以及 ROUGE-L 指标上分别提升了 4.3%和 3.8%. 除了可以生成更高质量的代码注释, ExplainBash 方法在测试机上的检索时间也缩短了 72.4%. ExplainBash 方法和 ExplainBash-Semantic (只使用语义相似度检索)方法相比, 各项评估指标值均能有提升. 不难看出, 由于 ExplainBash 方法在语义相似度检索的基础上还需进行词法相似度检索, 因此针对实证研究中的测试集, 其检索时间仅需多花 2.1 s, 但相比于注释质量的提升, 该额外需要消耗的时间成本是可以容忍的.

表 9 ExplainBash 方法和基于其他检索策略方法的比较结果

方法	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)	Time (s)
ExplainBash-Semantic	51.00	39.64	33.13	28.90	27.77	47.07	**15.29**
ExplainBash-Lexical	49.96	38.45	31.79	27.45	27.27	46.01	63.10
ExplainBash-Reverse	51.50	40.34	33.85	**29.62**	28.19	47.59	275.47
ExplainBash	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**	17.39

除此之外, 我们可以看到, ExplainBash-Semantic(只使用语义相似度检索)方法的各项评估指标值均高于 ExplainBash-Lexical(只使用词法相似度检索)方法, BLEU-1/2/3/4 指标分别提升了 1.04, 1.19, 1.34 和 1.45 个百分点, 即性能分别提升了 2.08%, 3.09%, 4.21%以及 5.28%, METEOR 以及 ROUGE-L 指标分别提升了 1.83%和 2.30%. 因此我们认为, 只考虑词法信息有时并不能检索出更为相似的代码片段, 需要首先考虑语义信息.

为了分析检索策略的合理性, 我们还对 ExplainBash 方法的检索顺序进行了讨论, 我们将 ExplainBash 方法和先进行词法相似度检索再进行语义相似度检索的方法 ExplainBash-Reverse 进行了对比, 可以看到, ExplainBash-Reverse 的各项评估指标值虽然低于 ExplainBash 方法, 但仍比上述基于单策略的方法有明显提升, 这说明双重信息检索的策略可以有效提升生成注释的质量. 但相较于其他检索策略, ExplainBash- Reverse 方法却需要更多的时间开销, 即面对相同数量的 Bash 代码, 需要的检索时间是 ExplainBash 的 15.8 倍. 由于首先进行词法相似度检索, 只能保证代码之间的词法相似, 因此可能会过滤掉与输入代码语义相似的代码片段, 从而导致检索质量的下降.

5 讨 论

在本节中, 我们对 ExplainBash 方法的内部设置的合理性进行讨论, 包括语义相似度检索过程中超参数 k 的取值、语义相似度计算中使用的 BERT-whitening 操作以及词法相似度检索中使用的编辑距离对实验结果的影响等.

5.1 超参数*k*对ExplainBash方法性能的影响

我们将超参数 k 的取值分别设置为 4, 5, 6, 7, 8, 9, 10, 15, 20, 25 和 30. 表 10 给出了 k 的不同取值对于性能的影响. 实验结果表明, 当 k 取值为 8 时, Explainbash 方法在各项评价指标下的表现均达到最佳; 而当 k 的取值过大之后, 会导致语义相似度检索过程中, 将语义相似度并不高的代码检索到 top-k 的代码集合中, 从而降低词法相似度检索的精度, 并最终影响到生成注释的质量.

表 10 超参数 k 的不同的取值对 ExplainBash 方法性能的影响

k 的取值	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)
5	51.53	40.26	33.81	29.54	28.07	47.23
6	51.29	39.95	33.41	29.06	28.15	47.32
7	51.09	40.03	33.63	29.36	28.12	46.95
9	51.07	39.96	33.52	29.25	27.96	47.22
10	51.37	40.14	33.71	29.46	28.17	47.07
15	50.76	39.82	33.50	29.32	27.91	47.19
20	51.50	40.49	34.07	29.77	28.38	47.58
25	50.90	39.70	33.27	29.01	37.98	47.00
30	50.70	39.53	33.13	28.96	27.89	46.59
ExplainBash (k=8)	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**

5.2 使用BERT-whitening操作对ExplainBash方法性能的影响

使用 BERT-whitening 操作可以对学出的代码语义向量进行降维操作, 从而提高语义相似度计算的效果. 为了验证使用 BERT-whitening 操作可以提升 ExplainBash 方法的性能, 我们设置了一组实验对该问题进行讨论. 表 11 给出了该组实验的比较结果, 其中, without-whitening 表示在语义相似度计算时不使用 BERT-whitening 操作. 结果表明, 在 ExplainBash 方法中, 使用 BERT-whitening 操作可以在各项性能评估指标下均有性能提升. 除此之外, 由于使用 BERT-whitening 操作可以对学出的语义向量进行降维处理, 因此最终可以提高在测试集上的检索效率. 基于对检索整个测试集所需时间的统计我们发现, 使用 BERT-whitening 操作可以减少 20%的检索时间.

表 11 BERT-whitening 操作对 ExplainBash 方法性能的影响

方法	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)	Time (s)
without-whitening	51.29	39.98	33.51	29.27	28.10	47.17	21.81
ExplainBash	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**	**17.39**

5.3 编辑距离对对ExplainBash方法性能的影响

在计算词法相似度时, 编辑距离是一种常用方法. Liu 等人[19]在他们针对代码注释自动生成的研究工作中, 也使用了编辑距离来检索出最相似的代码片段. 为了分析使用不同文本距离计算方法对 ExplainBash 方法性能的影响, 我们在计算词法相似度时也考虑了其他不同的计算方式作为对比方法.

除了编辑距离计算方法之外, 计算文本距离的方法还包括基于词元的计算方法、基于序列的计算方法以及基于压缩的计算方法等. 对此, 我们从各种方法中取出最为经典的 Jaccard 距离[45]、LCS(最长公共子序列)距离[46]以及 Entropy 熵距离[47]作为编辑距离的基准方法. 表 12 给出了使用不同方法计算词法相似度的结果, 我们可以发现, 使用编辑距离的 ExplainBash 方法在各项评价指标下均能取得更好的性能. 因此我们认为, 针对 Bash 代码注释自动生成问题, 使用编辑距离计算相似度是 ExplainBash 方法的最佳选择.

表 12 不同词法相似度计算方法对 ExplainBash 性能的影响

方法	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)
Jaccard	51.23	39.91	33.38	29.11	28.11	47.39
LCS	50.38	38.99	32.48	28.21	27.57	46.61
Entropy	51.02	39.32	32.59	28.20	27.90	46.72
ExplainBash	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**

5.4 考虑Bash代码片段的结构特征对ExplainBash方法性能的影响

这一节, 我们讨论除了 Bash 代码的语义信息以及词法信息外, 加入 Bash 代码的语法结构特征作为检索依据. 由于 Bash 代码的语法不规则, 命令选项难以用语法树表示[1]. 因此, 针对 Bash 代码片段提取有效的结构信息是一项具有挑战性的任务. Bharadwaj 等人[48]最近提出了一种将 Bash 代码转换成抽象语法树的方法, 我们借助该方法获取到代码片段的抽象语法树信息, 并将其转换成词元序列, 该序列同时包含了 Bash 代码片段的语法结构信息以及词法信息. 我们将获取的序列替换 ExplainBash 方法词法相似度模块中的仅包含词法

信息的词元序列, 构建出一种考虑了 Bash 代码语法结构信息的 ExplainBash-Syntax 检索方法.

表 13 给出了考虑了语法结构信息的 ExplainBash-Syntax 方法在数据集中的实验结果. 可以看出, 在大部分评价指标下, 我们提出的 ExplainBash 方法的性能均优于加入代码片段的语法结构信息后的检索方法, 虽然各个指标的提升不大, 但加入代码片段的语法结构信息后的检索方法相较于 ExplainBash 方法多花费了 79.62 秒的检索时间(即增加了 457.84%的时间成本). 因此我们认为, 本文提出的 ExplainBash 方法仍是当前解决 Bash 代码注释生成问题的最好解决方案.

表 13 加入语法结构信息对 ExplainBash 性能的影响

方法	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)	Time (s)
ExplainBash-Syntax	51.34	40.21	33.78	29.52	28.15	46.77	97.01
ExplainBash	**51.81**	**40.52**	**33.96**	**29.62**	**28.45**	**47.76**	**17.39**

6 有效性威胁

在本节中, 我们将分析对我们研究有效性可能构成的潜在威胁和应对措施.

6.1 内部有效性威胁

第 1 个内部有效性的威胁因素是 ExplainBash 方法在实现过程中的潜在缺陷, 为了避免这些缺陷, 我们使用了比较成熟的框架, 例如使用 Faiss 库进行信息检索, 使用 PyTorch 框架(https://github.com/pytorch/pytorch)实现本文所提方法和基准方法. 第 2 个是选择的基准方法是否合理. 我们主要选择了来自代码注释生成领域的经典基准方法. 由于许多基于深度学习的代码注释生成方法[8,9]需要利用抽象语法树来提取代码片段的语法信息, 而由于 Bash 代码的语法存在不规则的问题[1], 因此很难通过抽象语法树来提取出相关语法信息. 因此, 选择基准方法和 ExplainBash 进行比较时, 我们选择了经典的 CODE-NN 方法以及 Transformer 方法进行比较, 这些基准方法均不涉及代码片段的语法信息, 且在代码注释研究中具有较好的性能.

6.2 外部有效性威胁

本文主要的外部威胁在于语料库的选择, 我们选择了 NL2Bash[1]提供的语料库, 因为当前针对 Bash 代码的相关研究[22,40]的语料库均来自于 NL2Bash 研究工作. 除此之外, 我们还使用了 NeurIPS 2020 举办的 NLC2CMD Challenge 的比赛数据, 将两者合并, 并经过去重等处理后形成更大规模的高质量语料库, 从而保证实证研究的结论具有一般性. 在未来工作中, 我们将通过挖掘 GitHub 和 StackOverflow 来进一步丰富我们的语料库.

6.3 构造有效性威胁

构造有效性威胁是指实证研究中使用的评价指标是否真实反映了方法的性能. 在我们的研究中, 我们考虑了 3 种性能评估指标, 包括 BLEU[37]、METEOR[38]以及 ROUGE-L[39]. 这些指标已经被广泛应用于机器翻译领域和代码注释自动生成领域[9,31], 因此具有一定的代表性.

7 总结与展望

本文针对 Bash 代码的注释自动生成问题展开研究, 并提出了一种基于语义信息和词法信息的双重信息检索的 ExplainBash 方法. 基于实证研究和人本研究的结果表明, 在我们整理的语料库上, 基于双重信息检索的 ExplainBash 方法的性能要优于来自代码注释自动生成领域的基于信息检索的方法以及基于深度学习的方法. 除此之外, 我们也通过设计一系列消融实验, 验证了本文所提方法内的设定的合理性. 最后, 我们基于本文所提的 ExplainBash 方法开发出一个基于 Chrome 浏览器的插件, 以方便用户对 Bash 代码的理解.

在后续研究工作中, 我们将首先尝试从 GitHub 和 StackOverflow 中继续搜索更多 Bash 代码数据来丰富我们的语料库, 可以更加充分地覆盖更多类型的 Bash 命令; 其次, 虽然在当前数据集下, 我们发现本文提出的

基于信息检索的 ExplainBash 方法的性能要优于深度学习方法, 我们在未来工作中打算将信息检索方法和深度学习方法进行有效融合, 使得模型可以在信息检索生成的代码注释以及深度学习模型生成注释之间自动完成最优选择.

References:

[1] Lin XV, Wang C, Zettlemoyer L, *et al*. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In: Proc. of the 11th Int'l Conf. on Language Resources and Evaluation (LREC 2018). 2018. 3107−3118.

[2] Chen X, Yang G, Cui ZQ, Meng GZ, Wang Z. Survey of state-of-the-art automatic code comment generation. Ruan Jian Xue Bao/ Journal of Software, 2021, 32(7): 2118−2141 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6258.htm [doi: 10.13328/j.cnki.jos.006258]

[3] Cho K, van Merrienboer B, Gulcehre C, *et al*. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proc. of the Int'l Conf. on Empirical Methods in Natural Language Processing (EMNLP 2014). 2014. 1724−1734.

[4] Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. In: Proc. of the 27th Int'l Conf. on Neural Information Processing Systems, Vol.2. 2014. 3104−3112.

[5] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473, 2014.

[6] Luong T, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. In: Proc. of the EMNLP. 2015. 1412−1421.

[7] Iyer S, Konstas I, Cheung A, *et al*. Summarizing source code using a neural attention model. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol.1: Long Papers). 2016. 2073−2083.

[8] Hu X, Li G, Xia X, *et al*. Deep code comment generation. In: Proc. of the IEEE/ACM 26th Int'l Conf. on Program Comprehension (ICPC). IEEE, 2018. 200−210.

[9] Yang G, Chen X, Cao J, *et al*. ComFormer: Code comment generation via transformer and fusion method-based hybrid code representation. In: Proc. of the 8th Int'l Conf. on Dependable Systems and Their Applications (DSA). IEEE, 2021. 30−41.

[10] Fu W, Menzies T. Easy over hard: A case study on deep learning. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. 2017. 49−60.

[11] Haiduc S, Aponte J, Moreno L, *et al*. 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.

[12] Eddy BP, Robinson JA, Kraft NA, *et al*. Evaluating source code summarization techniques: Replication and expansion. In: Proc. of the 21st Int'l Conf. on Program Comprehension (ICPC). IEEE, 2013. 13−22.

[13] Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. In: Proc. of the ACM/ IEEE 32nd Int'l Conf. on Software Engineering, Vol.2. IEEE, 2010. 223−226.

[14] Rahman MM, Roy CK, Keivanloo I. Recommending insightful comments for source code using crowdsourced knowledge. In: Proc. of the IEEE 15th Int'l Working Conf. on Source Code Analysis and Manipulation. 2015. 81−90.

[15] Liu Z, Xia X, Hassan AE, *et al*. Neural-machine-translation-based commit message generation: How far are we? In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. 2018. 373−384.

[16] Feng Z, Guo D, Tang D, *et al*. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the Conf. on Empirical Methods in Natural Language Processing: Findings. 2020. 1536−1547.

[17] Su J, Cao J, Liu W, *et al*. Whitening sentence representations for better semantics and faster retrieval. arXiv:2103.15316, 2021.

[18] Yang G, Liu K, Chen X, *et al*. CCGIR: Information retrieval-based code comment generation method for smart contracts. Knowledge-based Systems, 2022, 237: Article No.107858.

[19] Liu S, Chen Y, Xie X, *et al*. Retrieval-augmented generation for code summarization via hybrid GNN. In: Proc. of the Int'l Conf. on Learning Representations. 2020. 1−16.

[20] Gu J, Lu Z, Li H, *et al*. Incorporating copying mechanism in sequence-to-sequence learning. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol.1: Long Papers). 2016. 1631−1640.

[21] Lin XV, Wang C, Pang D, *et al*. Program synthesis from natural language using recurrent neural networks. Technical Report, UW-CSE-17-03-01, Seattle: Department of Computer Science and Engineering, University of Washington, 2017.

[22] Kan JW, Chien WC, Wang SD. Grid structure attention for natural language interface to Bash commands. In: Proc. of the Int'l Computer Symp. (ICS). IEEE, 2020. 67−72.

[23] Schuster M, Paliwal KK. Bidirectional recurrent neural networks. IEEE Trans. on Signal Processing, 1997, 45(11): 2673−2681.

[24] D'Antoni L, Singh R, Vaughn M. NoFAQ: Synthesizing command repairs from examples. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. 2017. 582−592.

[25] Kim M, Sazawal V, Notkin D, *et al*. An empirical study of code clone genealogies. In: Proc. of the 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2005. 187−196.

[26] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, 2002, 28(7): 654−670.

[27] Wong E, Yang J, Tan L. Autocomment: Mining question and answer sites for automatic comment generation. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2013. 562−567.

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

[29] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473, 2014.

[30] Liang Y, Zhu K. Automatic generation of text descriptive comments for code blocks. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2018. 5229−5236.

[31] Ahmad W, Chakraborty S, Ray B, *et al*. A transformer-based approach for source code summarization. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. 2020. 4998−5007.

[32] Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. In: Proc. of the Advances in Neural Information Processing Systems. 2017. 5998−6008.

[33] Wei B, Li Y, Li G, *et al*. Retrieve and refine: Exemplar-based neural comment generation. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2020. 349−360.

[34] Zhang J, Wang X, Zhang H, *et al*. Retrieval-based neural source code summarization. In: Proc. of the IEEE/ACM 42nd Int'l Conf. on Software Engineering (ICSE). IEEE, 2020. 1385−1397.

[35] Li J, Li YM, Li G, Hu X, Xia X, Jin Z. EditSum: A retrieve-and-edit framework for source code summarization. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2021). 2021. 155−166.

[36] Yujian L, Bo L. A normalized levenshtein distance metric. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2007, 29(6): 1091−1095.

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

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

[39] Lin CY. Rouge: A package for automatic evaluation of summaries. In: Proc. of the Workshop on Text Summarization Branches Out, Post Conf. Workshop of ACL 2004. 2004. 74−81.

[40] Trizna D. Shell language processing: Unix command parsing for machine learning. arXiv:2107.02438, 2021.

[41] 41Miyato T, Dai AM, Goodfellow I. Adversarial training methods for semi-supervised text classification. arXiv:1605.07725, 2016.

[42] Stapleton S, Gambhir Y, LeClair A, *et al*. A human study of comprehension and code summarization. In: Proc. of the 28th Int'l Conf. on Program Comprehension. 2020. 5−17.

[43] Hu X, Gao ZP, Xia X, Lo D, Yang XH. Automating user notice generation for smart contract functions. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2021). 2021.

[44] Fleiss JL. Measuring nominal scale agreement among many raters. Psychological Bulletin, 1971, 76(5): 378−382.

[45] Jaccard P. The distribution of the flora in the alpine zone. New Phytologist, 1912, 11(2): 37−50.

[46] Maier D. The complexity of some problems on subsequences and supersequences. Journal of the ACM (JACM), 1978, 25(2): 322−336.

[47] Shannon CE. A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review, 2001, 5(1): 3−55.

[48] Bharadwaj S, Shevade S. Explainable natural language to Bash translation using abstract syntax tree. In: Proc. of the 25th Conf. on Computational Natural Language Learning. 2021. 258–267.

附中文参考文献:

[2] 陈翔, 杨光, 崔展齐, 孟国柱, 王赞. 代码注释自动生成方法综述. 软件学报, 2021, 32(7): 2118−2141. http://www.jos.org.cn/1000-9825/6258.htm [doi: 10.13328/j.cnki.jos.006258]

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为智能软件工程, 软件仓库挖掘, 经验软件工程.

于池(1997－), 男, 硕士生, 主要研究领域为代码注释自动生成.

杨光(1997－), 男, 硕士生, 主要研究领域为智能化软件工程, 代码注释自动生成, 代码自动生成.

濮雪莲(1979－), 女, 副教授, 主要研究领域为智能软件工程.

崔展齐(1984－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件测试及分析技术.