

MMCUP: 融合多模态信息的代码注释自动更新方法

刘诗凡¹⁾ 崔展齐¹⁾ 陈翔²⁾ 李莉¹⁾

¹⁾(北京信息科技大学计算机学院 北京 100101)

²⁾(南通大学信息科学技术学院 江苏 南通 226019)

摘 要 良好的代码注释对于程序维护有着重要价值. 但在实际开发过程中, 开发人员经常会在更改代码后忽略更新相应的代码注释, 导致更新后的代码和注释不一致, 对软件可维护性造成影响. 现有注释更新方法在进行注释更新时, 通常仅将代码视为普通文本进行处理, 忽视了代码结构信息. 为此, 本文提出了一种融合多模态信息的代码注释更新方法 MMCUP(Multi-Modal Comment UPdating). MMCUP 使用了旧代码注释、代码编辑序列和 AST 差异序列三种模态的信息来训练基于 Transformer 架构的模型, 以对注释进行更新. 实验结果表明, MMCUP 在 *Accuracy*、*Recall*@5 等指标上相较于 CUP 和 HatCUP 等方法至少提高了 5.8% 和 4.4%.

关键词 代码注释更新; 程序理解; 代码-注释共同演化; 深度学习; 序列到序列模型

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2024.00172

MMCUP: Updating Code Comments Based on Multi-Modal Information

LIU Shi-Fan¹⁾ CUI Zhan-Qi¹⁾ CHEN Xiang²⁾ LI Li¹⁾

¹⁾(School of Computer Science, Beijing Information Science and Technology University, Beijing 100101)

²⁾(School of Information Science and Technology, Nantong University, Nantong, Jiangsu 226019)

Abstract As the complexity of software continues to increase, program comprehension has become increasingly important in the process of software development. Code comments are one of the most important documents in program comprehension, and high-quality code comments are of great value for program maintenance. However, during software development developers often neglect to update corresponding comments after changing code. This could introduce inconsistent comments, which not only cause confusion in software development and maintenance but also have a negative impact on the robustness of the system. To address this problem, some research has attempted to automatically update corresponding comments when code changes occur. While code contains abundant and explicit structural information, existing approaches often treat code as plain text and ignore its structural information when updating comments. This can lead to many failures in comment updates. To address this issue, this paper proposes a code comment updating approach called MMCUP(Multi-Modal Comment UPdating) that integrates multi-modal information. MMCUP uses three modalities of information, which includes old code comment sequence, code edit sequence, and AST difference sequence. First, data processing is performed to construct comment sequences based on the original comment information. Then, Code edit sequences and AST difference sequences are constructed based on the code before and after changes. These sequences are combined with old code comments to form input sequences that are fed into the

收稿日期: 2023-01-17; 在线发布日期: 2023-07-11. 本课题得到江苏省前沿引领技术基础研究专项(BK202002001)、国家自然科学基金项目(61702041)、北京信息科技大学“勤信人才”培育计划项目(QXTCP C201906)资助. 刘诗凡, 硕士研究生, 中国计算机学会(CCF)学生会员, 主要研究方向为代码注释生成. E-mail: pawn2017@bistu.edu.cn. 崔展齐(通信作者), 博士, 教授, 中国计算机学会(CCF)高级会员, 主要研究领域为智能软件工程及可信人工智能技术. E-mail: czq@bistu.edu.cn. 陈翔, 博士, 副教授, 中国计算机学会(CCF)高级会员, 主要研究方向为软件测试、软件维护、经验软件工程和软件仓库挖掘. 李莉, 博士, 副教授, 中国计算机学会(CCF)会员, 主要研究方向为机器学习、文本分类、异常检测.

model. After that, the Transformer encoder is used to encode each token in the input sequence separately. During training, multi-modal information features are fused through a multi-head attention mechanism. Finally, the decoder in Transformer is used to decode the encoded multi-modal information features and update comments. Experimental results show that MMCUP has improved *Accuracy* by 5.8% compared to HatCUP and 4.4% compared to HebCUP. The *Recall@5* is 3.6% higher than HatCUP which achieves previous best performance. To determine whether the code edit sequence and AST difference sequence used in MMCUP can help improve the performance of comment update, we conducted ablation experiments. The experimental results show that both the code edit sequence and AST difference sequence can improve comment update performance. In addition, we conducted experimental analysis on the effect of MMCUP when facing more complex scenarios of code changes. The results show that compared with other approaches, for complex samples, MMCUP has shown the best performance among all comment update approaches. This indicates that MMCUP can learn different comment update situations to cope with more complex scenarios. To further validate the effectiveness of our method, we conducted a manual evaluation comparing MMCUP with HatCUP. The results of the manual evaluation also showed that the comments updated by MMCUP were more in line with the expectations of developers. Meanwhile, we discussed the reasons for failure cases of MMCUP and conducted an threats to validity analysis. In future research, we will further optimize the structured features extracted from code. For example, we will utilize control flow and data flow when extracting code features to increase the information obtained by the model. Additionally, we plan to explore other modalities of information that could be integrated into our model to further improve its performance.

Keywords code comment updating; program comprehension; code-comment co-evolution; deep learning; sequence-to-sequence model

1 引言

近年来,随着软件复杂度不断增加,程序理解在软件开发过程中变得越来越重要。在程序理解过程中,代码注释是最为重要的文档之一。然而在实际开发过程中,开发人员在更改源代码时常会忽略对注释进行相应更新^[1]。研究显示,有47%的受访从业者认为过时的注释是在软件开发过程中经常发生的问题^[2]。过时的注释不但会造成软件开发和维护混乱,还会对系统的健壮性产生负面影响。

为解决上述问题,有研究尝试在代码发生改变时使用机器学习方法自动更新相应注释。近年来,随着神经机器翻译(Neural Machine Translation, NMT)模型被成功应用于生成式阅读理解^[3]、诗词生成^[4]等任务,研究人员也尝试将 NMT 应用于更新过时注释。例如, Liu 等人^[5]提出了代码注释即时更新技术 CUP(Comment Updater),以解决广泛存在的过时注释问题。CUP 利用序列到序列模型,从旧注释和发生更改的代码中学习注释更新模式。由于 CUP

是基于深度学习的方法,在模型训练过程中需要大量的时间和计算资源,为提高注释更新效率, Lin 等人^[6]通过对 CUP 更新注释情况的分析,提出了一种基于启发式的注释更新方法 HebCUP,它将重点放在代码变更上,对注释做出单个令牌(token)级别的更新, HebCUP 不再需要进行模型训练,极大地缩短了更新注释所需的时间。Zhu 等人^[7]提出了一种基于 RNN(Recurrent Neural Network)的注释更新方法 HatCUP,该方法通过模仿开发者的编辑行为,生成一系列编辑行为对相应的代码注释进行更新,在 *Accuracy* 和 *Recall@5* 等指标上取得了比 HebCUP 更好的效果。

尽管上述方法在注释更新方面取得了较好的性能,但仍然存在一些局限性。例如,绝大多数被 CUP 成功更新注释的样本都只更改了单个令牌,而且 CUP 经常会被其他信息误导,即使代码的更改与注释更新之间存在明确关联时, CUP 也可能会忽略部分代码更改的信息,导致更新失败。而 HebCUP 则使用了启发式规则对注释进行单个令牌级别的更新,该方法虽然在简单场景下能够取得较好效果,但

在面对需要更改多处令牌,或者代码更改与注释更新之间没有直接关系时,HebCUP 将难以做出正确更新.

代码具有良好的结构属性^[8],现有方法在进行注释更新时,通常仅将代码视为普通文本而忽略了代码的结构信息.近年来,研究人员在代码注释自动生成^[9]、代码克隆检测^[10]等研究领域中使用抽象语法树(Abstract Syntax Tree,AST)来更好地表示源代码的结构化信息,以在这些任务中取得更好的性能.在代码注释更新任务中,由于需要表示更新前后代码的差异,如果直接使用两段代码的 AST 序列,会导致所构造的序列过长,不利于机器学习模型捕捉其中的关系.因此,需要对更改前后代码的 AST 进行有针对性的处理.

为此,本文提出了一种融合多模态信息的代码注释更新方法 MMCUP(Multi-Modal Comment UPdating).MMCUP 使用了三种模态的信息,包括旧代码注释序列、代码编辑序列和 AST 差异序列.首先,进行数据处理,根据原始注释信息构建相关注释序列,根据更改前后的代码构建代码编辑序列和 AST 差异序列,与旧代码注释组合构成输入序列,一同输入模型.然后,使用 Transformer 编码器对输入序列中的每个令牌分别进行编码,并在训练过程中通过多头注意力机制融合多模态信息特征.最后,使用 Transformer 中的解码器对被编码的多模态信息特征进行解码并更新注释.为验证 MMCUP 的有效性,我们在实验中使用了与 CUP 和 HebCUP 相同的数据集,该数据集包含 108k 代码-注释共同演化样本组,实验结果表明,MMCUP 在 *Accuracy*、*Recall@5* 指标上相较于 CUP 和 HatCUP 等方法至少提高了 5.8% 和 4.4%.

- 本文的主要贡献如下:
- (1) 提出了一种融合多模态信息的代码注释更新方法 MMCUP,该方法能够利用代码的语义信息和结构信息,并通过简化 AST 差异的表示方法,更精简地表示代码结构化差异.
 - (2) 为评估 MMCUP 方法的有效性,在含有 108k 代码-注释共同演化样本组的大规模数据集上进行了实验,从 *Accuracy*、*Recall@5* 等方面评估了 MMCUP 更新注释的性能.
 - (3) 为便于进行后续研究,在 GitHub 上开源了 MMCUP 工具^①的源码、实验数据集的代码和结构化信息以及实验原始结果.

本文第 2 节介绍相关工作;第 3 节通过一个示例来介绍本文工作的动机;第 4 节介绍融合多模态

信息的代码注释更新方法;第 5 节提出本文的研究问题,并通过实验对 MMCUP 的有效性进行评估;第 6 节讨论人工评价结果、部分注释更新失败的原因以及有效性威胁;第 7 节总结全文.

2 相关工作

本节介绍代码注释更新以及与之相关的代码注释自动生成和基于多模态的代码表征学习等工作,并在表 1 中对这三类工作进行了小结.

表 1 相关工作对比

研究类型	相关工作	主要特点
代码注释更新	Liu 等人的工作 ^[5]	提出基于深度学习的方法 CUP,使用基于 LSTM 的序列到序列模型,并引入了多种定制的功能,如使用特殊的分词器对复合词进行拆分等.
	Lin 等人的工作 ^[6]	通过分析 CUP 的结果,提出基于启发式方法的 HebCUP,缩短了注释更新所需时间.
	Zhu 等人的工作 ^[7]	提出了注释更新方法 HatCUP,以一种新的编辑机制来模拟人类更新行为.
	Yang 等人的工作 ^[11]	结合了深度学习和启发式方法,提出了一种混合注释更新方法 CBS.
代码注释自动生成	Kuhn 等人的工作 ^[13]	使用 LSI 方法提取语义信息,从主题中抽取出的关键词,作为代码摘要.
	Liu 等人的工作 ^[14]	使用词袋模型表示注释,并基于余弦相似度从训练集中选出最为相似的 k 个样本.
	Wong 等人的工作 ^[15]	应用代码克隆检测技术,将相似代码的注释作为目标代码的注释.
	Iyer 等人的工作 ^[16]	提出了一种基于 LSTM 的注释生成模型 CODE-NN,为 C# 和 SQL 生成自然语言注释.
	Hu 等人的工作 ^[9]	提出了一种使用 AST 来分析 Java 程序方法结构和语义信息的模型 Deep-Com.
	Ahmad 等人的工作 ^[17]	首次使用 Transformer 模型进行代码注释生成.
	Wan 等人的工作 ^[18]	使用强化学习进行代码注释生成.
基于多模态的代码表征学习	Tang 等人的工作 ^[19]	提出了注释生成方法 AST-Trans,对 AST 进行更有效地编码以利用代码结构信息.
	Chakraborty 等人的工作 ^[20]	提出了一种基于神经机器翻译模型的多模态自动代码编辑器 Modit,该方法使用了三种模态信息:编辑位置、编辑代码上下文、提交消息.
	Gao 等人的工作 ^[21]	将多模态学习运用于代码注释生成任务中,设计了 M2TS 方法,该方法在全局和局部多个层次上提取代码的 AST 结构信息.
	Yang 等人的工作 ^[22]	提出了一种基于 Transformer 的多模态智能合约代码摘要方法 MMTrans,该方法使用 AST 的 SBT 序列和图两种模态.

① <https://github.com/rookieLiu2018/MMCUP>

2.1 代码注释更新

代码注释更新技术可在代码进行更改时,对注释进行相应更新,以避免因代码更新造成注释过时,从而误导开发人员。Liu 等人^[5]对代码注释更新问题展开研究,提出了注释更新方法 CUP,该方法基于 LSTM(Long Short-Term Memory)引入了多种定制的增强功能以提升更新效果。此外,他们构建了包含 108k 组代码-注释共同演化样本组的数据集,这是注释更新任务的首个大规模数据集。

随后, Lin 等人^[6]分析了 CUP 方法成功和失败的情况。此外,他们还分析了数据集对于 CUP 方法的影响,并提出了一种基于启发式的方法 HebCUP,将重点放在代码变更上,对注释做出令牌级别的更新,该方法在多项评价指标上取得了比 CUP 更好的性能,并极大地缩短了注释更新所需要的时间。CUP 的更新结果为 HebCUP 制定规则提供了参考,HebCUP 在过滤后的部分数据集上的性能优于 CUP。但是,HebCUP 需要对注释更新的结果进行分类,总结成功更新样本的特点,并人工制定相应的更新规则。该方法可以对代码中修改单个令牌的注释进行有效更新,但面对其他较为复杂的代码变更时,缺乏相应规则对注释进行更新。

Zhu 等人^[7]提出了注释更新方法 HatCUP,该方法引入了一种结合代码变化图分析和数据流依赖分析的结构引导注意机制,通过基于 RNN 的编码器-解码器架构,以一种编辑机制来模拟开发人员的更新行为。实验结果表明 HatCUP 优于当时最先进的基于深度学习的方法 CUP,与基于启发式的方法 HebCUP 相比,HatCUP 也显示出更好的整体性能。

Yang 等人^[11]则结合了深度学习和启发式方法,提出了一种混合方法 CBS。该方法在进行更新前首先将代码分为两类,即倾向于使用启发式方法进行注释更新和使用其他方法进行注释更新,再根据分类结果选择使用 HebCUP 或 CUP 进行更新。实验结果表明,CBS 提高了注释更新的成功率。

以上工作为代码注释更新任务构建了较大规模的实验数据集。CUP 和 HebCUP 仅将代码视为文本进行处理,在面对复杂代码变更时注释更新效果较差,而基于启发式规则的 HebCUP 只能更新特定类型的样本。HatCUP 则需要更多额外信息,且其基于 RNN 的架构存在无法准确捕捉长期依赖关系的问题。CBS 作为一种分类方法,其重点在于将待更新的样本进行分类,综合使用基于深度学习的 CUP 和基于启发式规则的 HebCUP 两种方法以达到更

好的效果。本文关注提高基于深度学习的代码注释更新方法性能,可在 CBS 中替代 CUP 模块,以提升 CBS 更新注释的性能。与上述方法不同,MMCUP 使用了基于 Transformer 的架构,并引入了代码的结构化差异信息。此外,MMCUP 还对更改前后代码的 AST 差异进行了简化,与其他信息进行融合以达到更好的效果。在此前工作中,我们对引入代码的结构化信息进行了初步探讨^[12],本文在此基础上对 AST 差异序列的使用进行了改进,重新设计了模型的输入序列,并使用人工评价的方法补充自动化评价指标的不足。

2.2 代码注释自动生成

注释自动生成技术可直接根据代码片段生成相应注释。在早期工作中,研究人员主要使用基于信息检索的方法生成注释。近年来,由于深度学习技术在许多软件工程任务中的出色表现,越来越多的研究人员开始尝试采用深度学习技术解决注释生成任务,并取得了良好效果。

使用信息检索生成注释的方法通过在带有注释的代码数据集中计算目标代码与数据集中代码的相似度,返回一段或多段相匹配的代码,并利用相对应的注释来为目标代码生成注释。基于信息检索生成注释的方法一般采用基于向量空间模型(Vector Space Models)、潜在语义索引(Latent Semantic Indexing,LSI)、主题模型等技术。例如 Kuhn 等人^[13]认为代码命名中隐藏了开发者的意图,他们首先利用 LSI 方法在方法名、变量名等标识符中提取语义信息,并对代码进行主题聚类,然后从主题中抽取相应的关键词,作为代码的摘要。Liu 等人^[14]提出了一种基于 K 最邻近(K -Nearest Neighbor)的注释生成方法 NNGen,该方法使用词袋模型,将训练集中的代码和目标代码转化为向量表示,随后基于余弦相似度从训练集中选出最为相似的 k 个样本,计算目标代码与相似代码的 BLEU-4 评分,选择评分最高的相似代码对应注释作为目标代码的注释。实验结果表明,NNGen 相比深度学习方法可以提速 2600 倍,并且在 BLEU 指标上提升了 21%。Wong 等人^[15]应用代码克隆检测技术在开源代码网站中搜索与目标代码相似的代码段,并使用这些相似代码段的注释为目标代码生成注释。

基于深度学习的方法常将代码注释生成问题建模为机器翻译问题。Iyer 等人^[16]首先使用此类方法尝试为代码生成注释,他们提出了一种基于 LSTM 的注释生成模型 CODE-NN,为 C# 和 SQL 生成自

然语言注释. Hu 等人^[9]提出了一种使用 AST 来分析 Java 方法结构和语义信息的模型 DeepCom, 该模型使用代码的 AST 序列作为神经网络输入, 通过训练来学习代码的语义和结构信息. 实验结果表明, DeepCom 生成了比 CODE-NN 更为准确的注释. Ahmad 等人^[17]首次利用 Transformer 模型进行代码注释生成, 实验结果表明该方法可以有效地捕获长距离依赖, 且性能优于 CODE-NN 和 DeepCom. Wan 等人^[18]利用强化学习进行代码注释生成, 他们使用 LSTM 模型来捕获代码的语义信息, 使用另一个基于 AST 的 LSTM 模型来捕获代码的结构信息. 此外, 他们还使用强化学习框架来解决解码时面临的曝光偏差 (exposure bias) 问题, 以获取更好效果. Tang 等人^[19]提出了注释生成方法 AST-Trans, 更有效地编码对 AST 进行以利用代码结构信息, 它使用了 AST 中的两种节点关系: 祖先-后代和兄弟关系, 应用树结构的注意力机制为相关节点动态分配权重, 并根据这两种关系排除不相关节点. 实验结果表明 AST-Trans 效果显著优于其他方法.

在软件开发过程中, 代码注释对于程序的理解和维护至关重要, 高质量的代码注释有助于提高开发和维护效率. 代码注释生成可以为缺乏注释的代码生成相应注释, 而代码注释更新则是对过时的注释进行更新, 以减少不一致注释, 两类工作都可以帮助开发人员节省时间和精力, 提高软件的可维护性. 此外, 代码注释自动生成与代码注释更新都属于序列生成类任务, 涉及到对源代码和注释的处理以及使用基于深度学习的模型进行生成, 两类任务具有一定的相似性, 注释生成任务中的相关工作为我们的研究提供了参考.

但是, 与代码注释自动生成不同, 代码注释更新的目标并不是从头开始生成注释, 而是在原先注释的基础上进行修改, 在大部分情况下所更新出的注释与原注释较为相似.

2.3 基于多模态的代码表征学习

为了更充分地利用源代码中的信息, 研究人员试图从不同模态来学习源代码. Chakraborty 等人^[20]提出了一种基于神经机器翻译模型的多模态自动代码编辑器 Modit, 该方法利用三种信息模式: 编辑位置、编辑代码上下文、提交消息, 使用神经机器翻译模型自动生成代码编辑补丁, 实验结果表明, 编辑位置作为一种输入方式可以缩小补丁的搜索空间, 所生成的代码编辑也优于此前最先进的基线模型 CODIT.

Gao 等人^[21]将多模态学习运用于代码注释生

成任务中, 设计了 M2TS 方法, 该方法在全局和局部多个层次上提取代码的 AST 结构信息, 同时为了补充 AST 中缺失的语义信息, 还获取了代码的令牌特征, 并使用多模态融合的方法将它们与提取的 AST 特征进一步结合, 该方法不仅融合了源代码的结构和上下文语义信息, 还更为准确地表示了每种模态的关键特征. 在两个 Java 数据集和一个 Python 数据集上的实验结果表明 M2TS 优于此前最先进的注释生成方法.

Yang 等人^[22]提出了一种基于 Transformer 的多模态智能合约代码摘要方法 MMTrans, 该方法使用 AST 的两种模态, 即基于结构的遍历 (Structure-Based Traversal, SBT) 序列和图形来表示源代码的全局和局部语义信息, 然后使用两个编码器、一个联合解码器和多头注意结构来捕获代码标记之间的远程依赖关系. 在所构建数据集上的实验结果表明, MMTrans 取得了比 Hybrid-DeepCom 更好的性能, 在实际测试中生成了更高质量的注释.

此前的研究探索了基于不同代码表征技术的多模态学习, 验证了融合代码不同模态信息对提升模型性能的有效性. 在上述工作基础上, MMCUP 验证了多模态学习在代码注释更新任务中的可行性.

3 研究动机

此前的研究中, 主要将代码视为普通文本进行处理. 这类方法通过比对更改前后的代码令牌, 明确在每个令牌上进行的细粒度操作, 以对代码更改进行表示. 这类表示方法忽略了代码具有的良好结构属性, 在分析代码更改的过程中, 不同位置的令牌更改在代码结构中所对应的重要程度也不同, 仅通过比对令牌变化难以准确捕获代码更改与注释更新之间的依赖关系, 容易导致注释更新失败.

表 2 为一个代码注释更新的示例, 表中 Code Change 是代码的更改情况, 其中红色表示删除的代码, 绿色为新添加的代码, Old Comment 是更新前的代码注释, New Comment 是人工更新的代码注释, 其他行则对应不同方法的更新结果. 从表 2 可以看出, 注释的更新与代码的变化是高度相关的, 开发人员将代码中的 “Collections.unmodifiable List (completers)” 修改为 “completer”, 并在人工更新的注释中删除了相应描述. 然而, 表中 CUP、HebCUP 和 HatCUP 等三种注释更新方法均没有捕捉到修改代码的相应意图, 错误地进行了更新.

表 2 代码注释更新示例 1

Code Change:
- public Collection <Completer> getCompleters() { - return Collections.unmodifiableList(completers); + public Completer getCompleter() { + return completer;

以基于深度学习的注释更新方法 CUP 为例，图 1 为 CUP 对示例 1 中代码更改的表示，虽然其对新版本和旧版本中的每个代码令牌进行了细粒度描述，但由于代码中存在着多处变更，仅通过此类方法进行表示容易导致难以判断注释更改与哪一部分代码相关。在此示例中，CUP 错误地将“an unmodi-

fiable list”更改为了“a collection”，而 HebCUP 仅将“completers”替换为“Completer”，HatCUP 将“list”替换为“completer”，均未能正确地更新注释。

图 2 为示例 1 中 getCompleters() 方法中代码差异部分对应的 AST，图中粉色节点为旧代码 AST 中被删除的节点，蓝色节点为旧代码 AST 中进行了更新的节点。从图中可以看出，旧代码中与注释更改相关的“Collection.unmodifiableList”在 AST 中体现为子树删除，这对应了代码注释中的相关更改。在此示例中，使用 AST 信息可以更加准确地建立起代码与注释更新之间的联系。通过加入代码的结构化信息有助于提高更新注释的性能。

为此，本文提出了一种融合多模态信息的代码注释自动更新方法，通过结合代码语义信息和结构化信息，可以更有效地表示代码更改，使得模型能够更为准确地捕获到代码更改与注释更新间的重要关系，提升模型进行注释更新的准确率。

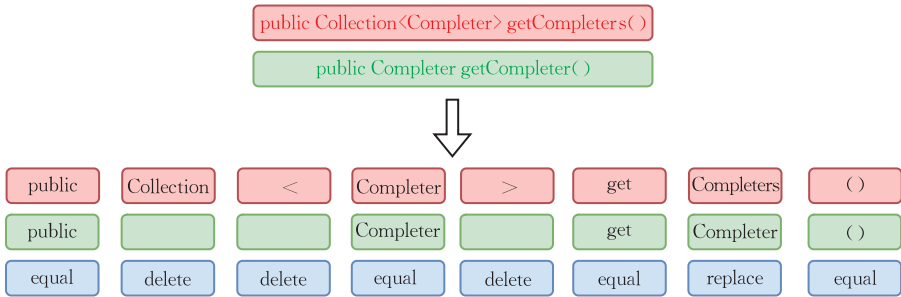


图 1 CUP 的代码更改表示示例

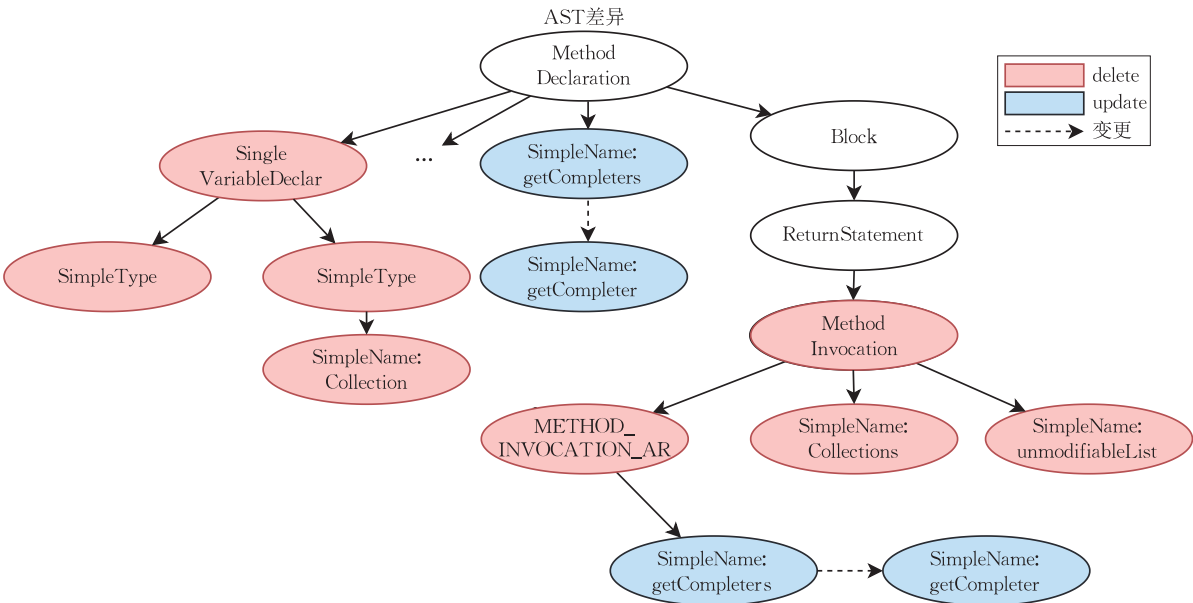


图 2 示例代码差异部分的 AST

4 MMCUP 方法

本节将详细介绍代码注释自动更新方法 MMCUP,图 3 为 MMCUP 的流程图,包括三个阶段:数据处理、模型训练、注释更新.在数据处理阶段,对数据集中的原始数据进行清洗和过滤,并通过代码变更和相应注释变更构建后续阶段中需要使用的相关序列.模型训练阶段,MMCUP 使用了基于 Transformer 的编码器-解码器架构,处理了三种模态的信息:代

码 AST 差异信息、代码的文本差异信息和旧注释信息,并从中捕获到与新注释之间的依赖关系.在注释更新阶段,MMCUP 会根据此前学习到的依赖关系对旧注释进行更新.具体步骤为在数据处理阶段分析更改前后代码的文本差异和结构差异,分别表示为代码编辑序列和 AST 差异序列,并与原注释序列结合生成输入序列.然后,使用处理好的数据训练 MMCUP 模型.最后,将待更新样本的代码编辑序列、结构化差异序列以及旧注释输入到训练好的模型中,以生成更新注释.在本节中将详细介绍上述步骤.

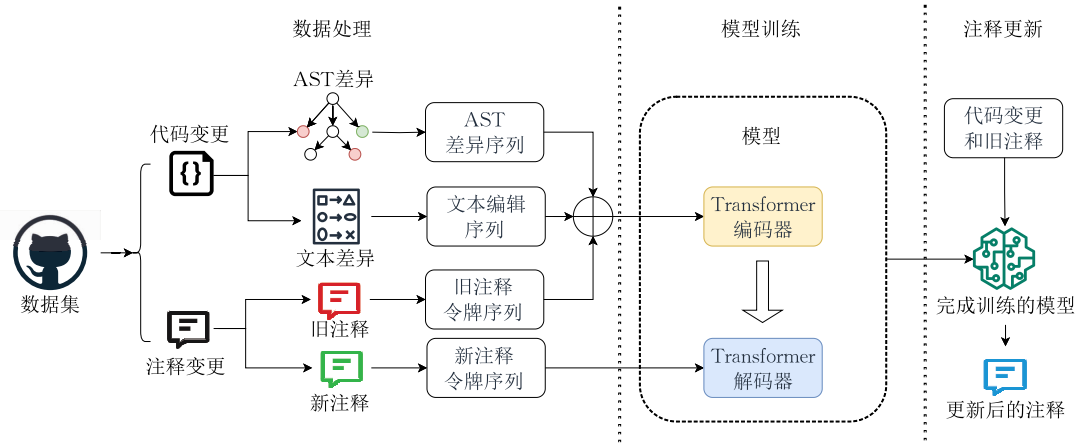


图 3 MMCUP 方法流程图

4.1 数据处理阶段

在此阶段中,MMCUP 对代码更改以及注释的原始信息进行处理.首先,根据代码更改情况构建代码编辑序列,根据注释的原始信息构建模型能够接收的注释序列;然后,根据 AST 的变化情况构建代码变更的结构化序列;最后,为了更好地融合多模态信息,需要将上述序列进行合并,以构建输入序列.

4.1.1 注释序列及代码编辑序列的构建

在构建注释序列时,首先根据驼峰命名法对注释中的标识符进行拆分,对拆分后的结果再使用匈牙利命名法进行检查,然后对注释中含有的特殊标点符号进行过滤,并使用空格替换注释中的标点符号,最后分别生成更改前后的注释序列.

在构建代码编辑序列时,首先将代码中的标识符根据驼峰命名法、匈牙利命名法将其拆分为多个令牌,以减少 OOV(Out of Vocabulary)单词,然后根据拆分之后的结果构建代码序列.

随后,在得到更改前后的代码序列后,将序列中的令牌统一为小写,并使用文本差异化对比工具来

提取两段代码序列之间的差异,通过差异分析结果,可以将旧代码变更为新代码时的操作进行分类.在对比工具分析结果的基础上,我们将编辑操作分为三种类型:INSERT、DEL、UPDATE.其中,INSERT 是指新插入旧代码中的部分,DEL 是指旧代码中被删除的部分,UPDATE 是指旧代码中的令牌被新代码中的令牌替换.需要注意的是,UPDATE 必须包含更新前后的代码内容,以明确指出旧注释中的哪些令牌将被新令牌替换.在更改前后的代码令牌完全相同时不对该令牌添加额外描述,在代码发生变化时,使用<操作>[令牌]…[令牌]<操作_END>的形式对代码更改情况进行描述.图 4 描述了根据代码变更构建代码编辑序列的过程.图中代码变更部分红色的代码是被删除的内容,绿色的代码是新增加的内容,代码编辑序列是最终构建出的用于表示代码文本变化的编辑序列.例如:“collection<DEL_END>”表示删除了“collection<”,“<UPDATEFROM> completers<UPDATETO> completer<UPDATE_END>”表示该段代码从“completers”更新为“completer”.

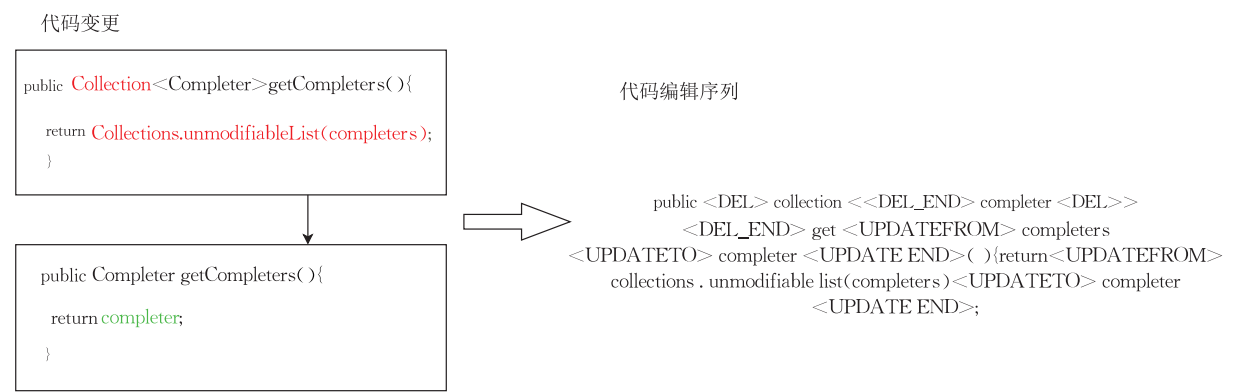


图 4 编辑序列生成过程

4.1.2 代码变更的结构化信息序列构建

此前的工作表明,代码的 AST 序列可以直观地描述代码相关的结构化信息^[9,23],因此本文选择通过构建更改前后代码的 AST 差异序列来描述代码之间的结构化差异.需要注意的是,更改前后代码所对应 AST 的差异序列中包含了代码的节点类型和节点属性值,如果对 AST 进行遍历并使用类似于处理文本的方式构建序列会导致序列长度过长,不利于模型捕捉重要依赖关系^[24].因此,需要对 AST 的差异情况进行简化.

具体来说,在生成 AST 差异序列时,首先对更改前后代码所对应的 AST 进行差异对比,其次,保留有差异并且有属性值的节点,舍去其他节点.在

得到差异节点之后,根据对节点的操作不同,可将节点分为四类:分别是 UPDATE、MOVE、INSERT、DELETE.其中,UPDATE 代表旧代码 AST 节点属性值发生变化. MOVE 代表旧代码中的节点移动到新代码 AST 中的另一个位置. INSERT 为新节点插入到旧代码的 AST 中. DELETE 代表旧代码中的 AST 节点被删除.在保留下这些有差异的节点后,分别对每个节点进行描述,最终每个节点的结构化信息由<操作,节点类型,节点属性值>组成,将每个差异节点的信息进行连接后生成代码的 AST 差异序列.图 5 描述了使用动机示例中的代码变更来构建 AST 差异序列的过程.

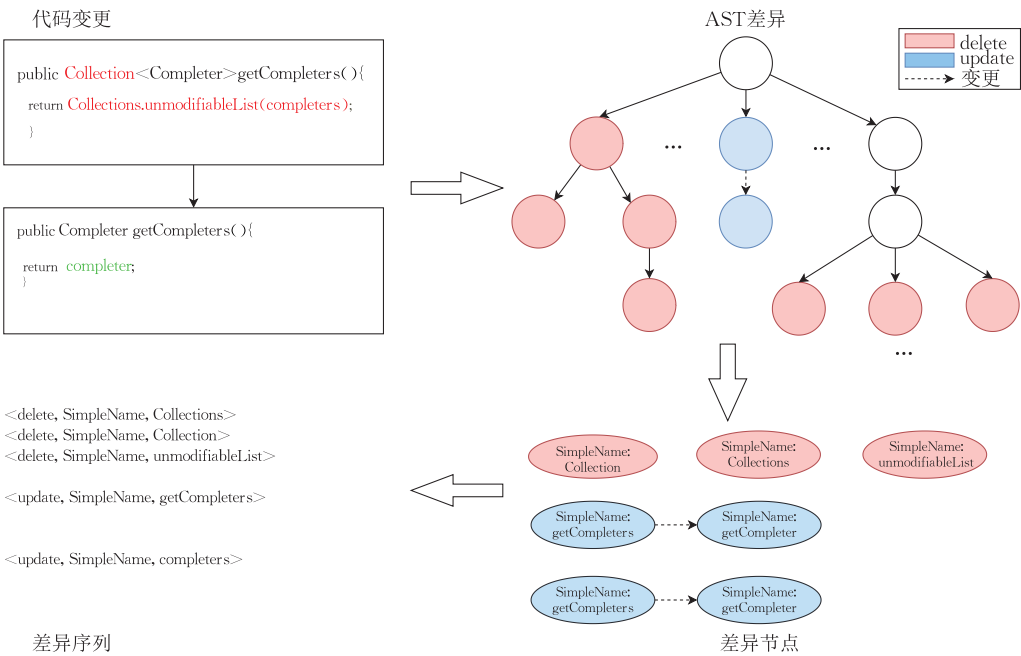


图 5 AST 差异序列生成过程示例

4.1.3 构建组合序列

此前的研究表明,在多模态生成任务中,使用单个编码器比为每个模态使用单独的编码器具有

更好的性能^[20].这是因为使用同一个编码器对不同模态的信息进行编码可以使模型更好地捕获这些信息之间的关系,从而在生成序列时达到更好效

果. 因此, MMCUP 对经过上述处理后得到的序列进行整合, 以构建多模态信息组合序列, 并输入到同一个编码器中进行特征提取. 因此, MMCUP 需要对经过上述处理后得到的序列进行整合, 以构建多模态信息组合序列. 在所生成的序列中, 前半部分为旧注释序列, 中间部分为代码编辑序列, 最后为 AST 差异序列, 三者通过特殊符号〈s〉进行分隔, 以区分不同的信息来源.

4.2 模型训练阶段

MMCUP 使用基于 Transformer 的编码器-解码器模型^[25], 进行模型训练时, MMCUP 首先将组合序列和新注释序列进行向量化, 随后将向量化后的组合序列输入到模型编码器, 将向量化后的新注释序列作为参考答案输入到模型解码器.

对于输入编码器中的向量序列 $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_m)$, 在编码器中使用多头注意力机制来学习每个向量与其他向量之间的关系. 多头注意力机制建立在缩放点注意力机制上, 通过使每个向量 $\mathbf{x}_i$ 作为查询(query) Q 到一系列键值对的映射(Key-Value)进行操作, 缩放点积注意力的计算方法如式(1)所示. 其中 d_k 是 Q 和 K 的维度.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

对于每个输入序列中的令牌, 编码器将这些令牌所对应的 Q, K, V 映射到不同子空间中, 通过计算不同子空间的注意力结果来学习不同范围内的依赖关系(例如, 短距离依赖和长距离依赖关系), 然后对多个子空间的注意力计算结果进行拼接, 产生最终输出, 其中每个子空间被称为一个注意力头(head). 式(2)为多头注意力的计算方法, $\mathbf{W}^O$ 为可学习的参数矩阵, $head_i$ 为第 i 个注意力头缩放点积注意力的计算结果, n 是注意力头的数量. 通过多头注意力机制, 模型可以学习序列内部的相关特征.

$$\text{MultiHead}(Q, K, V) =$$

$$\text{Concat}(head_1, head_2, \dots, head_i, \dots, head_n)\mathbf{W}^O \quad (2)$$

在此过程中每个令牌可以学习到与序列中其他令牌的依赖关系, 由于不同的模态信息在同一个序列中, 所以每个令牌的最终编码信息都包含了三种模态的信息.

经过编码器提取的相关特征会输入到解码器中, 与编码器相比, 解码器增加了带掩蔽的多头注意力(Masked Multi-Head Attention), 这是由于在训练过程中, 指定时刻的输出应取决于之前的输出, 因

此需要防止解码器看到未来时刻的信息. 在带掩蔽的多头注意力中, MMCUP 会生成上三角矩阵, 通过将此矩阵与每个输入序列相乘, 达到隐藏指定时刻之后信息的作用. 在解码时, 首先会输入新注释序列的起始令牌, 在通过带掩蔽的多头注意力后, 会与编码器的输出进行交互, 使模型能够学到两者之间的联系, 得到该令牌的最终表示.

4.3 注释更新阶段

MMCUP 中的解码器会从左至右预测令牌, 直到序列结束. 在预测过程中, MMCUP 使用集束搜索^[26](Beam Search)方法生成序列, 集束搜索的方法在序列生成中被广泛使用, 特别是在文本生成任务上取得了良好效果. 在本文中, MMCUP 通过使用集束搜索方法来提高代码注释更新的准确性. 集束搜索使用广度优先搜索来构建搜索树, 在树的每一层会生成当前层状态的所有后继, 并按照启发式规则对这些状态进行升序排序, 保留固定数量的候选后继节点, 并舍弃掉其他后继. MMCUP 会在每层选取最有可能成为正确注释的令牌组成更新注释序列.

5 实验及评估

为评估所提出代码注释更新方法 MMCUP 的有效性, 本文提出以下 4 个研究问题并分析了这些问题的设计动机:

RQ1: 与其他注释更新方法相比, MMCUP 表现如何?

动机: MMCUP 融合了多模态信息进行代码注释更新, 为分析 MMCUP 与其他基准方法相比性能如何, 即 MMCUP 相比其他方法在注释更新的效果上是否取得了有效提升, 在这个 RQ 中, 我们选择注释更新方法 CUP、HebCUP 和 HatCUP 作为对比对象. 为评估这些方法更新注释的质量, 我们使用了 *Accuracy*、*Recall@5* 等 6 个自动化评估指标.

RQ2: MMCUP 的关键组件对注释更新结果的影响如何?

动机: MMCUP 使用代码编辑序列和 AST 差异序列以提高注释更新性能, 在这个 RQ 中, 为研究 MMCUP 中使用的代码编辑序列和 AST 差异序列是否有助于提升注释更新性能, 我们进行了消融实验.

RQ3: MMCUP 在处理 AST 差异较大的样本时,

注释更新性能如何?

动机: 在这个 RQ 中, 我们将 AST 中出现增加或删除子树的情况称为 AST 差异较大的样本, 该类样本容易导致注释更新方法无法捕获到重要的依赖关系, 并做出错误更新, 为研究 MMCUP 在面对此类样本时的表现, 分析 MMCUP 在面对此类场景时的性能如何, 是否能够有效地处理这类代码更改, 提出了本研究问题.

RQ4: MMCUP 在面对代码的更改较为复杂的场景时, 注释更新性能如何?

动机: 在这个 RQ 中, 我们沿用 HatCUP 的定义, 将基于启发式规则的代码注释更新方法 HebCUP 无法正确更新的情况视为复杂场景, 为研究 MMCUP 在面对此类场景时的表现, 提出了本研究问题.

5.1 实验设计

5.1.1 数据集

在实验中, 我们选择了 CUP 创建, 并在 HebCUP 和 HatCUP 等工作中广泛使用的数据集^①. 该数据集最早由 Liu 等人^[5]根据 GitHub 上 1496 个 java 项目中的 48 007 次提交构建, 并按照提交顺序将前 80% 作为训练集, 剩余的 20% 打乱顺序, 平均分为验证集和测试集. 通过这种方式, 可确保训练集中的所有注释更新都发生在验证集和测试集中的注释更新之前. 随后, Lin 等人^[6]根据特定规则过滤了该数据集中仅对语言表达进行优化的样本, 例如修改错别词汇等. 过滤后的数据集总共丢弃了 6183 个样本, 最终包含 80 591、8827 和 9204 个代码-注释共同演化样本组, 分别用于训练、验证和测试, 本文在 MMCUP、CUP、HebCUP 和 HatCUP 上的实验均在该过滤后的数据集上进行.

为分析 MMCUP 在不同场景下的性能如何, 我们对测试集中的样本进行了划分. 首先, 当代码更改前后的 AST 中有增加或删除子树的操作时, 我们认为属于 AST 有较大差异的场景. 测试集中有 4697 个样本满足 AST 有较大差异的标准, 占 51%. 然后, 我们沿用了 HatCUP 的设置, 即将 HebCUP 无法正确处理的样本作为代码更改较为复杂的场景. 这是由于 HebCUP 很难为代码更改与注释更新没有直接关系的样本进行正确更新, 因此, 可以合理地假设 HebCUP 无法正确处理的样本中代码更改更为复杂^[6]. 测试集中有 6844 个样本满足复杂样本的标准, 占 74.3%. 数据集中代码长度、注释长度、不同样本类别样本数量和占比统计信息如表 3 所示.

表 3 数据集的统计信息

代码长度统计			
Avg.	<150	<200	<250
91.8	79.7%	87.1%	92.6%
注释长度统计			
Avg.	<20	<40	<60
13.2	84.0%	99.8%	100%
AST 有较大差异的样本		复杂样本	
数量	占比(测试集)	数量	占比(测试集)
4697	51.0%	6844	74.3%

5.1.2 评价指标

根据所提出的研究问题, 实验使用 *Accuracy*、*Recall@5*、*AED* (Average Edit Distance)、*RED* (Relative Edit Distance)、*GLEU*^[27] 和 *SARI*^[28] 共 6 个指标来评估代码注释更新方法的性能.

Accuracy 用来评价方法是否正确地更新了注释. 我们使用数据集中开发人员更新后的注释作为参考注释, 若代码注释更新方法生成的注释和参考注释相同则认为更新成功. *Accuracy* 是指模型生成的推荐注释列表中排名第一的注释和参考注释相同的数量所占百分比.

Recall@5 与 *Accuracy* 类似, 用来衡量推荐注释列表中排名前五的注释是否有和参考注释相同的结果, 如果存在相同的结果则认为更新成功.

AED 用来衡量方法生成的注释与参考注释差异的平均值, 它使用单词级别的 Levenstein 距离^[29]评价不同方法生成的更新注释和参考注释之间的差距. 其中, Levenstein 距离用于测量两个字符串序列之间的差异. 两个字符串之间的 Levenstein 距离是指将一个字符串更改为另一个字符串所需的最少编辑操作次数. *AED* 的值越小, 代表方法生成的注释与参考注释越为接近.

RED 用于衡量相对字符编辑距离的平均值, 相对字符编辑距离是指不同方法生成的更新注释和参考注释之间差距与旧注释和参考注释之间差距的比值. 形式上, 给定一个有 N 个样本的测试集, *AED* 和 *RED* 的计算方法分别如式(3)和式(4)所示. 其中 *edit_distance* 是单词级别的 Levenstein 距离. $\hat{y}$ 和 y 分别是方法产生的注释和参考注释, x 为旧注释.

$$AED = \frac{1}{N} \sum_{k=1}^N \text{edit_distance}(\hat{y}^{(k)}, y^{(k)}) \quad (3)$$

$$RED = \frac{1}{N} \sum_{k=1}^N \frac{\text{edit_distance}(\hat{y}^{(k)}, y^{(k)})}{\text{edit_distance}(x^{(k)}, y^{(k)})} \quad (4)$$

GLEU 最初被用于评价语法纠错 (Grammatical

① <https://github.com/HATCUP0/hatcup>

Error Correction)方法,该指标通过计算生成的句子与参考句子集合的 n -gram 重叠进行评估, $GLEU$ 分数越高,生成的注释与参考注释越接近.

$SARI$ 指标最早被用于评价文本摘要任务,该指标将生成的摘要与源文本和参考摘要进行比较,衡量不同方法保留、删除和添加某些单词对句子带来的影响,该指标得分越高,表示输出摘要的质量越高.

5.1.3 基准方法

在实验中,我们将CUP、HebCUP和HatCUP作为比较对象.其中,CUP是基于深度学习的方法,其基于序列到序列(Seq2Seq)框架,在编码器中使用双向LSTM,以使用注意力机制来更好地捕获代码变更和注释之间的关系,并引入了多项定制的增强功能,以有效处理代码注释更新任务的特征. HebCUP则是一种启发式注释更新的方法,该方法将重点放在代码的变更上,对注释做出令牌级别的更新,实验表明其在多项评价指标上取得了超过CUP的性能,并极大地缩短了注释更新需要的时间. HatCUP是一种基于深度学习的方法,通过基于RNN的编码器-

解码器架构,以一种新的编辑机制来模拟开发人员的更新行为,实验表明, HatCUP表现出比CUP、HebCUP等注释更新方法更好的整体性能.

5.1.4 实现细节

本文使用Pytorch和Transformers来实现的方法MMCUP,使用GumTree来提取AST的差异.其中,模型的词嵌入尺寸被设置为768,注意力层的数量设置为12.使用Adafactor优化所有参数,初始学习率设置为0.0005.在训练期间,批量大小设置为30.编码器和解码器的最大长度分别设置为512和32.所有实验都在具有32GB RAM、AMD 5700X 8-Core Processor和英伟达RTX 3070 GPU的计算机上完成.

5.2 实验结果与分析

5.2.1 RQ1:与其他注释更新方法相比,MMCUP表现如何?

为回答这个问题,我们与CUP、HebCUP和HatCUP在所选择的数据集上进行了对比实验,实验结果如表4所示.

表 4 不同注释更新方法的性能对比

方法名	Accuracy / %	Recall@5 / %	AED	RED	GLEU	SARI
CUP	15.8	26.8	3.62	0.960	50.30	38.62
HebCUP	25.6	27.6	3.52	0.896	54.14	41.29
HatCUP	24.2	34.9	3.43	0.860	56.46	45.94
MMCUP	30.0	38.5	3.31	0.878	59.80	47.79

从表中可以看出,在 $Accuracy$ 指标上,MMCUP取得了最佳效果,比基于深度学习的方法CUP和HatCUP分别高14.2%和5.8%.此外,对比此前取得最佳效果的HebCUP,MMCUP也提高了4.4%.对于 $Recall@5$ 指标,MMCUP也比之前性能最好的方法HatCUP高3.6%.这表明MMCUP能够更好地进行注释更新,其使用的编辑序列和AST序列能够使MMCUP能够更好地排除其他无关信息的干扰,更为明确地学习到代码更改与注释更新之间的依赖关系.

对于 AED 指标,MMCUP较此前效果最好的方法HatCUP下降了0.12,这表明MMCUP所更新的注释最接近开发人员人工更新的参考注释.对于 RED 指标,MMCUP略高于HatCUP.分析发现,这是由于 RED 需要与原注释进行比较,而MMCUP倾向于在序列中更改更多单词,这样的更改方式在出现更新错误时,容易导致 RED 偏大.总体上看,MMCUP的 RED 值低于CUP和HebCUP,仅比HatCUP高0.018.此外,MMCUP在 $GLEU$ 和 $SARI$ 这两项指标上分别比此前最好的方法HatCUP高

3.34和1.85,这表明MMCUP更新的注释在流畅度和质量上更优.上述实验结果表明MMCUP所更新的注释与参考注释更为接近,可以有效减轻开发人员的工作量.

表5为表2中示例使用不同方法生成更新注释的结果.表中New Comment为参考注释,其他行则对应不同方法的更新结果,CUP正确地删除了注释中的“list”,但错误地将“unmodifiable”修改为“collection”. HebCUP根据启发式的规则识别出代码中被修改过的令牌后,将代码中的更改对应到注

表 5 不同方法对示例1的更新结果

Code Change:
- public Collection <Completer> getCompleters() {
- return Collections.unmodifiableList(completers);
+ public Completer getCompleter() {
+ return completer;
}
Old Comment: Returns an unmodifiable list of all the completers.
New Comment: Returns the completer.
CUP: Returns a collection of all the completer.
HebCUP: Returns an unmodifiable list of all the Completer.
HatCUP: Returns an unmodifiable completer of all the completers.
MMCUP: Returns the completer.

释中,将“completers”修改为“completer”,但没有删除注释中其他应删除的部分. HatCUP 将“list”错误地修改为“completer”. 只有 MMCUP 对注释进行了多处修改,生成了正确的更新注释.

对 RQ1 的回答:对于 *Accuracy* 和 *Recall@5* 指标,MMCUP 的性能优于 HatCUP 等其他三种方法,其中,*Accuracy* 相比此前表现最好的 HebCUP 提高了 4.4%,*Recall@5* 指标高出此前表现最好的 HatCUP 3.6%. *AED* 指标相比此前表现最好的 HatCUP 降低了 0.12,这说明 MMCUP 生成的注释在整体上最接近参考注释. 总体来说,MMCUP 对注释进行更新的性能优于其它对比方法.

表 6 MMCUP 与停用部分组件时的性能对比

方法名	<i>Accuracy</i> / %	<i>Recall@5</i> / %	<i>AED</i>	<i>RED</i>	<i>GLEU</i>	<i>SARI</i>
MMCUP	30.0	38.5	3.31	0.878	59.80	47.79
-w/o AST	28.4	32.4	3.42	0.894	58.37	47.12
-w/o edit	27.6	30.3	3.45	0.914	57.79	46.42

从表 6 中可以看出,MMCUP 在各项指标上均优于 MMCUP w/o AST 和 MMCUP w/o edit. 其中 MMCUP w/o edit 的效果最差,相较于 MMCUP, *Accuracy* 下降了 2.4%,*Recall@5* 下降了 8.2%,*AED* 和 *RED* 分别上升了 0.14 和 0.036,*GLEU* 和 *SARI* 分别下降了 2.01 和 1.37. 这是由于未经处理的代码序列长度较长,并且更改前后代码中有关联的部分在序列中相距较远,不利于模型捕捉其中的相关性,导致性能下降. 此外,MMCUP w/o AST 的 *Accuracy* 相比于 MMCUP 下降了 1.6%,*Recall@5* 下降了 6.1%,*AED* 和 *RED* 分别上升了 0.11 和 0.016,*GLEU* 和 *SARI* 分别下降了 1.43 和 0.67,也表明了 MMCUP 构建 AST 差异序列的必要性.

对 RQ2 的回答:不使用编辑序列和不使用 AST 差异序列的 MMCUP 在各个指标上相比完整模型均有所下降,这表明 MMCUP 中所使用的代码文本和代码结构表示方式都能够帮助模型更为准确地捕捉到代码更改与注释更新之间的联系,以提升模型性能.

5.2.3 RQ3:MMCUP 在处理 AST 差异较大的样本时,注释更新性能如何?

当更改前后的代码文本差异较小但 AST 差异较大时,可能会对导致未关注结构化信息的方法性能下降. 为了研究注释更新方法在更改前后代码所对应的 AST 有较大差异的样本上效果如何,对不同方法面对此类样本的处理情况进行了实验. 使用不同的注释更新方法在这些样本上进行了实验,结果如表 7 所示.

5.2.2 RQ2:MMCUP 的关键组件对注释更新结果的影响如何?

为更好地表示代码更改情况与注释更新之间的关系,我们在 MMCUP 中引入了编辑序列和 AST 差异序列. 为分析这两个部分对正确更新注释的贡献如何,我们将 MMCUP 与 MMCUP w/o AST 和 MMCUP w/o edit 两个变体进行了对比实验. 其中,MMCUP w/o AST 是在进行模型训练时没有使用 AST 差异序列,MMCUP w/o edit 是指在进行模型训练时直接将更改前后的代码序列合并后输入模型,而不构建相应的代码编辑序列. 实验结果如表 6 所示.

表 7 不同注释更新方法对代码更新前后的 AST 差异较大样本的性能对比

方法名	<i>Accuracy</i> / %	<i>Recall@5</i> / %	<i>AED</i>	<i>RED</i>
CUP	7.3	13.2	4.41	0.994
HebCUP	9.2	14.1	4.38	0.983
HatCUP	11.8	20.2	4.35	0.975
MMCUP	17.9	26.8	4.22	0.979

从表 7 中可以看出,MMCUP 在 *Accuracy* 和 *Recall@5* 上均取得了最好的效果,相比于表中其他方法至少提高了 6.1% 和 6.6%. 相对于在整体数据集上的表现,MMCUP 的性能有了更大幅度的提升. 这说明 MMCUP 在处理 AST 差异较大的样本时,表现出了更好的性能. 对于 *AED*,MMCUP 也较其他方法至少降低了 0.13. 但由于我们方法在更新时倾向于更改更多单词的特点,在 *RED* 指标上还是略高于 HatCUP. 总体上看,MMCUP 是面对更新前后 AST 差异较大时表现最好的方法,且优势更加明显.

表 8 中的示例更直观地说明了 MMCUP 在代码更新前后 AST 差异较大样本上的优势,表中红色部分为更改前的代码,绿色为更改后的代码. 从表 8 中可以看出,旧代码删除了方法中的变量声明“Method method”,在方法内部删除了方法调用中的参数“method”,这在 AST 中被视为子树的删除. 参考注释中进行了相应的更改,删除了“Method”. 在各种方法中,只有 MMCUP 在注释中删除了“Method”,生成了正确的更新注释.

表 8 代码注释更新示例2

Code Change:
-public boolean canCallListenerInterface(Component component, Method method){ -return isEnabled(component) && component.canCallListenerInterface(method); +public boolean canCallListenerInterface(Component component){ +return isEnabled(component) && component.canCallListenerInterface(); }
Old comment: For further information please read the javadoc on {@link Component#canCallListenerInterface(Method)}, this method has the same semantics.
New comment: For further information please read the javadoc on {@link Component#canCallListenerInterface()}, this method has the same semantics.
CUP: For further information please read the javadoc on {@link Component#canCallListenerInterface(Method)}, this method has the same semantics.
HebCUP: For further information please read the javadoc on {@link Component#canCallListenerInterface(Method)}, this method has the same semantics.
HatCUP: For further information please read the javadoc on {@link Component#canCallListenerInterface(Method)}, this method has the same semantics.
MMCUP: For further information please read the javadoc on {@link Component#canCallListenerInterface()}, this method has the same semantics.

对 RQ3 的回答:对于更新前后 AST 差异较大的样本,MMCUP 比其他对比方法在 *Accuracy* 和 *Recall*@5 指标上至少提高了 6.1% 和 6.6%。相对于在全量数据集上的结果,MMCUP 展现出更大的性能优势,这表明 MMCUP 能够更好地捕捉到更改前后代码结构差异的信息,并用于更新注释。

5.2.4 RQ4:MMCUP 在面对代码更改较为复杂的场景时,注释更新性能如何?

我们沿用 HatCUP 中的设置,将基于启发式规则的代码注释更新方法 HebCUP 无法正确更新的情况视为复杂场景。HebCUP 只能根据启发式规则提取代码变更,并对注释进行单个令牌级别的更新,对于那些更新内容未出现在代码更改内容中的评论,很难手动定义注释更新模板。因此,可以合理地假设 HebCUP 无法正确处理的场景通常更为复杂。为了研究注释更新方法面对代码更改较为复杂场景

时更新效果如何,对不同方法面对此类场景时的更新注释情况进行了实验,结果如表 9 所示。

表 9 不同方法在代码更改较为复杂场景的更新性能对比				
方法名	<i>Accuracy</i> / %	<i>Recall</i> @5 / %	<i>AED</i>	<i>RED</i>
CUP	3.2	8.8	4.36	0.978
HebCUP	0.0	1.1	4.41	0.992
HatCUP	8.1	16.2	4.27	0.969
MMCUP	13.7	21.6	4.23	0.973

从表 9 中可以看出,MMCUP 相比于其他的方法取得了更好的性能,这表明其不仅能够处理类似于单个令牌替换的代码更新,也能够根据复杂的代码变化和注释更新情况正确地学习到依赖关系以更新不同样本。

在面对代码的更改较为复杂的场景时,MMCUP 相比于其他注释更新方法可以更为准确地捕捉到代码更改与注释更新之间的联系。例如表 10 中的示

表 10 代码注释更新示例3

Code Change:
- @Deprecated + public Builder pingBeforeActivateConnection (boolean pingBeforeActivateConnection) { + public Builder protocolVersion (ProtocolVersion protocolVersion) { + LettuceAssert.notNull(protocolVersion, "ProtocolVersion must not be null"); - this.pingBeforeActivateConnection = pingBeforeActivateConnection; + this.protocolVersion = protocolVersion; return this; }
Old comment: See {@link #DEFAULT_PING_BEFORE_ACTIVATE_CONNECTION}.
New comment: See {@link #DEFAULT_PROTOCOL_VERSION}
CUP: See {@link #protocolVersion PING_BEFORE_Version}.
HebCUP: See {@link #DEFAULT_protocol_BEFORE_ACTIVATE_CONNECTION}.
HatCUP: See {@link #DEFAULT_PING_BEFORE_ACTIVATE_CONNECTION}.
MMCUP: See {@link #DEFAULT_PROTOCOL_VERSION}

例,在方法名和方法内部均将“pingBeforeActivateConnection”修改为“protocolVersion”,注释中的修改也对应了代码中的修改,在预处理时,“pingBeforeActivateConnection”和“protocolVersion”会被拆分为多个令牌,因此在更新时需要对这些令牌做出修改,这就要求注释更新方法更为准确地捕捉到其中的联系.在这个示例中,CUP 错误地将“DEFAULT”修改为“protocolVersion”;HebCUP 根据修改规则,只修改了第一个令牌,即将“PING”修改为“protocol”;HatCUP 则没有对旧注释做出任何修改,输出了原注释;只有 MMCUP 进行了正确的注释更新.

对 RQ4 的回答:对于复杂样本,MMCUP 在所有注释更新方法中表现出了最好的性能,成功更新了 13.7% 的复杂样本,比此前效果最好的方法 HatCUP 高 5.6%,这表明 MMCUP 可以学习到不同的注释更新情况,以应对更为复杂的场景.

6 讨 论

在本节中将讨论 MMCUP 在人工评价下的表现、可能失败的情况及有效性分析.

6.1 人工评价

由于 *Accuracy* 等自动化评价指标着重于评价待测序列与参考序列之间的文本相似度,而代码注释作为一类主观性较强的文本,仅使用相似度难以准确评价其质量,因此我们还通过人工评价的方式对不同方法更新的注释进行对比.

在人工评价中,我们沿用了 Yang 等人^[30]的方法.具体来说,由于人工分析所有样本所耗费的成本巨大,我们根据一种广泛使用的抽样方法^[31]选择了最小数量的随机样本,如式(5)所示.其中, n_0 取决于选定的置信水平和期望的误差幅度, $size$ 为测试集中样本的数量,根据式(5),我们计算出人工评估中需要对 192 个样本进行评价.

$$MIN = \frac{n_0}{1 + \frac{n_0 - 1}{size}} \tag{5}$$

为了进行人工评估,我们向 16 名参与者发起了问卷调查.其中,6 名参与者是计算机专业的硕士研究生,拥有至少 3 年以上的 Java 项目开发经验.另外,我们还邀请了 10 名不同年龄段、具备软件开发经验的工程师参与了调查.其中 4 名工程师的年龄小于 25 岁,4 名工程师的年龄为 26 至 30 岁,2 名工程师的年龄为 31 至 40 岁.这些软件工程师中,3 名开发经验为 1 至 3 年,3 名开发经验为 3 至 5 年,4

名开发经验为 5 至 7 年.由于 HatCUP^[7]是此前代码注释更新中总体表现最好的方法,因此我们主要对 MMCUP 和 HatCUP 的注释更新情况进行人工评价.

我们从测试集中随机选择了 192 个样本,对应问卷中的 192 道问题,每个问题包括更新前的注释、参考注释、MMCUP 和 HatCUP 分别更新后的注释.我们为每位参与者制作了一份问卷,样本页面如图 6 所示.为了保证评价的公平性,这两种方法更新的注释在问卷中进行了随机排序,我们要求参与者仔细阅读注释更新样本,然后对每个样本的更新情况进行评价,评价标准如下:

- (1) 优秀(3 分):此注释与参考注释所表达的意思完全相同,可以直接作为该段更新后代码的注释使用.
- (2) 良好(2 分):此注释可以描述更新后的代码或与参考注释有一定的相似,但存在少量问题(例如不完整、重复或语法问题),只需要稍作修改即可.
- (3) 普通(1 分):此注释未对原注释进行了更改或此注释对原注释进行了更改,但它缺乏必要的关键内容或进行了无意义的更改.
- (4) 差(0 分):此注释对原注释进行了更改,但更新后的注释会错误地引导开发人员.

Answer question for the following information:

Code Change:

```
- public Collection <Completer> getCompleters() {  
-     return Collections.unmodifiableList(completers);  
+ public Completer getCompleter() {  
+     return completer;  
}
```

Old Comment: Returns an unmodifiable list of all the completers.

New Comment: Returns the completer.

Comment1: Returns an unmodifiable completer of all the completers.
Score:

Comment2: Returns the completer.
Score:

图 6 人工评价调查问卷

通过比较平均得分和各类得分的比例,可以验证 MMCUP 是否比 HatCUP 更具有优势.

在人工评价的过程中,每个问题都由 16 位参与者分别进行评价.为提高人工评价的可信度,我们制定了以下规则对结果中违反评分标准,存在明显错误的样本进行了过滤,包括:

- (1) 候选注释完全相同时,给出了不同的分数.
- (2) 候选注释与参考注释完全相同时,未给出“优秀”的评价.
- (3) 候选注释与更改前注释完全相同时,未给出“普通”的评价.

随后,使用组内相关系数^[32] (Intraclass Correlation Coefficient, ICC)来衡量这 16 位参与者评价结果的一致性. 16 位参与者人工评价结果的 ICC 值计算结果为 0.784,表明 16 位参与者之间的答案较为一致. 图 7 直观地展示了 MMCUP 和 HatCUP 两种方法所生成更新注释的人工评价等级统计结果,其中左侧为 MMCUP,右侧为 HatCUP. 从图 7 可以看出,对于 MMCUP 更新的注释,有 55.3% 的人工评价分数大于 1,即注释质量较好,无需进行重大修改,该项数值比 HatCUP 高 8.1%. 此外,对于评价为“差”的注释,MMCUP 比 HatCUP 减少了 2.1%. 表 11 为人工评价的平均得分统计,MMCUP 的人工评价平均得分为 1.91,表明其更新的注释在整体上基本接近于“良好”等级,相较于 HatCUP 高 0.19 分,提高了约 11%. 从人工评价的结果可以看出,与 HatCUP 相比,MMCUP 所生成的更新注释质量更高.

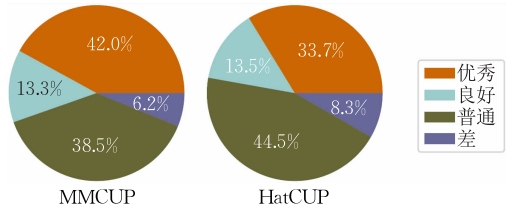


图 7 不同方法的各类评价占比图

表 11 不同方法进行人工评价的平均得分对比

方法名	平均分
HatCUP	1.72
MMCUP	1.91

6.2 方法存在的不足

虽然 MMCUP 在自动化评价指标和人工评价中均取得良好的结果,但仍然存在某些无法成功更新注释的情况. 为分析我们方法主要会在哪种情况下失败,我们随机检查了 200 个 MMCUP 注释更新失败的样本,分析了其中失败的原因,并总结了两类典型情况.

一类典型情况是代码更改与注释更新之间的相关性较弱. 尽管此前 HebCUP 已经根据相关规则清洗过干扰样本,但在数据集中仍存在一些样本在更新注释时仅优化了语言表达或更正了语法错误. 如表 12 所示的示例,该样本代码变更部分较多,涉及到多处标识符和结构变更,但在更新注释时,仅加入了冠词“a”,由于此类样本的代码更改与注释更新之间不存在较强的相关性,因此 MMCUP 及其他方法均无法根据代码更改进行准确的注释更新. 在这个示例中,MMCUP 加入了单词“name”,对应着代码中一处标识符的更改,与参考注释不一致.

表 12 由于相关性较弱导致注释更新失败的示例

Code Change:
<pre>- byte[] depPkg(final Dependency dep) { + String depPkg(final PkgDep dep) { - final TokenSet instVers = new TokenSet(); - for(final byte[] nextPkg : repo.pkgDict()) { - if(nextPkg != null && startsWith(nextPkg, dep.pkg)) instVers.add(version(nextPkg)); + final HashSet<String> instVers = new HashSet<>(); + for(final String nextPkg : repo.pkgDict().keySet()) { + if(nextPkg.startsWith(dep.name)) instVers.add(version(nextPkg)); } - final byte[] version = availVersion(dep, instVers); - return version == null ? null : dep.name(version); + final String version = availVersion(dep, instVers); + return version == null ? null : dep.id(version); }</pre>
Old comment: Checks if secondary package, i.e.
New comment: Checks if a secondary package, i.e.
MMCUP: Checks if secondary package names , i.e.

此外,表 13 中的示例也属于此类情况,MMCUP 根据代码中的修改,将旧注释中的“response scheduler”替换为了“token”,但参考注释中还加入了额外的单词. 这种情况下无法仅通过代码变更做出与参考注释完全一致的更新,MMCUP 能够正确更新

关键词,但无法加入额外的形容词. 另一类典型情况是开发人员在修改代码时几乎重写了整个方法,并且重新编写了该方法的注释. 这种情况下 MMCUP 无法根据训练结果对此类样本进行更新. 如表 14 所示,开发人员对该方法进行了重

表 13 由于相关性较弱导致注释更新失败的示例

Code Change:
<pre>- public Scheduler getResponseScheduler() { - return responseScheduler; + public String getToken() { + return token; }</pre>
Old comment: Returns the defined response scheduler .
New comment: Returns the currently configured token .
MMCUP: Returns the defined token .

表 14 由于重写整个方法导致注释更新失败的示例

Code Change:
<pre>_ public static RouterOptions create() { - return builder().build(); } + public Builder mutate() { + Builder builder = new Builder(); + builder.setToken(getToken()) + .setReactorResources(getReactorResources()) + .setExchangeStrategies(getExchangeStrategies()) + .setGlobalRateLimiter(getGlobalRateLimiter()); + getResponseTransformers().forEach(builder::onClientResponse); + return builder; + }</pre>
Old comment: Returns a new {@link RouterOptions} with default settings.
New comment: Create a {@link Builder} using the options configured in this instance.
MMCUP: Returns a new {@link RouterOptions} with default settings.

写,从表中可以看出更改前后的代码之间变化较大,这种变化使得 MMCUP 无法对旧注释进行更新,仅直接输出了原注释.

6.3 有效性分析

本小节从内部有效性、外部有效性和构造有效性 3 个方面对所提方法进行有效性分析.

本方法的内部有效性影响主要在于所实现 MMCUP 中的缺陷. 为了避免这种威胁,我们在实现 MMCUP 时使用了 PyTorch^①、Transformers^②、Gumtree^③ 等被广泛使用的第三方库和工具. 为确定 Gumtree 分析结果的正确性,我们随机检查了 100 个样本的分析结果,在抽查的样本中未发现有误的分析结果. 另外,为确保对照方法的正确性,我们使用了 CUP、HebCUP 作者发布的在测试集上生成的更新注释,由于 HatCUP 没有发布所生成的更新注释,我们使用其作者发布的源码并按照论文中的设置进行了训练,在相同的实验环境下进行了复现并与论文中发布的结果进行对比,以达到相似的性能. 另外,所编写的代码经过了多次的测试和检

查,尽可能减少内部有效性的影响.

本方法的外部有效性影响主要在于所使用的数据集. 为了方便对比,我们选择了与 HatCUP 相同的数据集,Zhu 等人^[7]对 HebCUP 的数据集进行了清洗和过滤,该数据集由 Java 代码和注释构成,具有一定的代表性,此前已被应用于注释更新的任务中. 此外,用于分析 AST 差异的 Gumtree 支持 Python、C 等多种编程语言,使得 MMCUP 不依赖于某种特定的编程语言和注释类型,可应用于其他编程语言上进行注释更新.

本方法的构造有效性影响主要在于评价指标的有效性. 本文采用了在代码注释更新任务中常用的指标来评价方法的性能. 其中 *Accuracy* 和 *Recall* @5 可以直观地反映出方法是否能够成功地更新注释,*AED*、*RED* 可以在一定程度上反映出更新后的注释和参考注释之间的相似程度.

① <https://pytorch.org/>

② <https://github.com/huggingface/transformers>

③ <https://github.com/GumTreeDiff>

7 总 结

本文提出了一种利用多模态信息的注释更新方法 MMCUP, 用于在软件演化过程中更新代码注释. 为了更好地表示更改前后的代码差异, MMCUP 引入了代码的 AST 差异信息, 但由于更改前后代码的 AST 差异序列较长, 不利于模型捕捉代码更改差异, 因此需要对该序列进行简化, 最终得到 AST 差异序列. 在进行注释更新时, MMCUP 会使用三种模态的信息: 旧代码注释、代码编辑序列和 AST 差异序列, 然后使用基于 Transformer 架构的模型捕捉代码更改和注释更新之间的依赖关系. 实验结果表明, MMCUP 优于 CUP、HebCUP、HatCUP 等方法, 取得了最好的注释更新效果. 在未来研究中, 我们将对提取结构化特征的部分进行进一步优化, 例如在提取代码特征时, 对控制流和数据流进行分析, 以增加模型获取的信息. 针对代码 AST 为树形结构的特点, 考虑在进行向量化后, 使用图神经网络代替 Transformer 编码器, 以使代码注释更新模型达到更好的效果.

参 考 文 献

- [1] Wen F, Nagy C, Bavota G, et al. A large-scale empirical study on code-comment inconsistencies//Proceedings of the International Conference on Program Comprehension. Montreal, Canada, 2019: 53-64
- [2] Hu X, Xia X, Lo D, et al. Practitioners' expectations on automated code comment generation//Proceedings of the International Conference on Software Engineering. Pittsburgh, USA, 2022: 1693-1705
- [3] Liu Hao, Hong Yu, Zhu Qiao-Ming. Unsupervised domain adaptive machine reading comprehension method. Chinese Journal of Computers, 2022, 45(10): 2133-2150(in Chinese) (刘皓, 洪宇, 朱巧明. 无监督的领域自适应机器阅读理解方法. 计算机学报, 2022, 45(10): 2133-2150)
- [4] Qin Chuan, Wang Meng, Si Guang-Wen, Yao Heng. Constructive information hiding with Chinese quatrain generation. Chinese Journal of Computers, 2021, 44(4): 773-785 (in Chinese) (秦川, 王萌, 司广文, 姚恒. 基于绝句生成的构造式信息隐藏算法. 计算机学报, 2021, 44(4): 773-785)
- [5] Liu Z, Xia X, Yan M, et al. Automating just-in-time comment updating//Proceedings of the International Conference on Automated Software Engineering. Melbourne, Australia, 2020: 585-597
- [6] Lin B, Wang S, Liu K, et al. Automated comment update: How far are we?//Proceedings of the International Conference on Program Comprehension. Madrid, Spain, 2021: 36-46
- [7] Zhu H, He X, Xu L. HatCUP: Hybrid analysis and attention based just-in-time comment updating//Proceedings of the International Conference on Program Comprehension. Pittsburgh, USA, 2022: 619-630
- [8] Mou L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing//Proceedings of the AAAI Conference on Artificial Intelligence. Phoenix, USA, 2016: 1287-1293
- [9] Hu X, Li G, Xia X, et al. Deep code comment generation//Proceedings of the International Conference on Program Comprehension. Gothenburg, Sweden, 2018: 200-210
- [10] Büch L, Andrzejak A. Learning-based recursive aggregation of abstract syntax trees for code clone detection//Proceedings of the International Conference on Software Analysis, Evolution and Reengineering. Hangzhou, China, 2019: 95-104
- [11] Yang Z, Keung J, Yu X, et al. On the significance of category prediction for code-comment synchronization. ACM Transactions on Software Engineering and Methodology, 2023, 32(2): 1-41
- [12] Liu S, Cui Z, Xie R. SICUP: A comment updating approach based on structural information//Proceedings of the International Conference on Software Analysis, Evolution and Reengineering. Macao, China, 2023: 970-971
- [13] Kuhn A, Ducasse S, Girba T. Semantic clustering: Identifying topics in source code. Information and Software Technology, 2007, 49(3): 230-243
- [14] Liu Z, Xia X, Hassan A, et al. Neural-machine-translation-based commit message generation: How far are we?//Proceedings of the International Conference on Automated Software Engineering. Montpellier, France, 2018: 373-384
- [15] Wong E, Liu T, Tan L. CloCom: Mining existing source code for automatic comment generation//Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering. Montreal, Canada, 2015: 380-389
- [16] Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model//Proceedings of the Annual Meeting of the Association for Computational Linguistics. Berlin, Germany, 2016: 2073-2083
- [17] Ahmad W, Chakraborty S, Ray B, et al. A transformer-based approach for source code summarization//Proceedings of the Annual Meeting of the Association for Computational Linguistics. Online, 2020: 4998-5007
- [18] Wan Y, Zhao Z, Yang M, et al. Improving automatic source code summarization via deep reinforcement learning//Proceedings of the International Conference on Automated Software Engineering. Montpellier, France, 2018: 397-407
- [19] Tang Z, Shen X, Li C, et al. AST-Trans: Code summarization with efficient tree-structured attention//Proceedings of the International Conference on Software Engineering. Pittsburgh, USA, 2022: 150-162
- [20] Chakraborty S, Ray B. On multi-modal learning of editing source code//Proceedings of the International Conference on Automated Software Engineering. Melbourne, Australia, 2021: 443-455

[21] Gao Y, Lyu C. M2TS: Multi-scale multi-modal approach based on transformer for source code summarization//Proceedings of the International Conference on Program Comprehension. Pittsburgh, USA, 2021: 24-35

[22] Yang Z, Keung J, Yu X, et al. A multi-modal transformer-based code summarization approach for smart contracts//Proceedings of the International Conference on Program Comprehension. Madrid, Spain, 2021: 1-12

[23] Hu X, Li G, Xia X, et al. Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering, 2020, 25: 2179-2217

[24] Dong J, Lou Y, Zhu Q, et al. FIRA: Fine-grained graph-based code change representation for automated commit message generation//Proceedings of the International Conference on Software Engineering. Pittsburgh, USA, 2022: 970-981

[25] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the International Conference on Neural Information Processing Systems. Long Beach, USA, 2017: 6000-6010

[26] Freitag M, Al-Onaizan Y. Beam search strategies for neural machine translation. arXiv preprint arXiv:1702.01806, 2017

[27] Napoles C, Sakaguchi K, Post M, et al. Ground truth for grammatical error correction metrics//Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing. Beijing, China, 2015: 588-593

[28] Xu W, Napoles C, Pavlick E, et al. Optimizing statistical machine translation for text simplification. Transactions of the Association for Computational Linguistics, 2016, 4: 401-415

[29] Levenshtein V I. Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR, 1965, 163(4): 845-848

[30] Yang G, Chen X, Zhou Y, et al. DualSC: Automatic generation and summarization of shellcode via transformer and dual learning//Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering. Hawaii, USA, 2022: 361-372

[31] Singh R, Mangat N S. Elements of Survey Sampling. Dordrecht, Netherlands: Springer Science & Business Media, 2013

[32] Young M, Dank G R, Roper R, et al. Inter-observer reliability. Behaviour, 1979, 69(3-4): 303-315



LIU Shi-Fan, M. S. candidate. His research interests focus on code comment generation.

CUI Zhan-Qi, Ph. D. , professor. His research interests include intelligent software engineering and trustworthy artificial intelligence.

CHEN Xiang, Ph. D. , associate professor. His research interests include software testing, software maintenance, empirical software engineering, and mining software repository.

LI Li, Ph. D. , associate professor. Her current research interests include machine learning, text classification, and outlier detection.

Background

The research of this paper belongs to code comment update in software engineering tasks. In the development process, developers often neglect to update the corresponding comments after changing the code, which results in inconsistent comments and affects the maintainability of software. In recent years, with the increasing frequency of software updates, inconsistent comments are inevitably introduced.

In previous work, the researchers attempted to use machine learning methods to automatically update the corresponding comments when code changes. In this process, they have built a large data set for experiments. The results show that these works have achieved good results, but there are still have limitations in some situations. In previous comment updating work, code changes are typically treated as ordinary text and the structural information of the code are ignored. However, the code contains abundant and explicit structural information, In our preliminary research, it has been verified

that the structured information of the code can effectively improve the accuracy of comment updating.

On the basis of the previous research, we propose a comment update approach MMCUP based on multi-modal information. MMCUP trains the Transformer model with multi-modal features which include old code comments, code edit sequences and AST differencing sequences. Abstract Syntax Tree (AST) can describe the structured information of code and has been widely used in previous relevant research. Therefore, MMCUP describes the structural differences by using the AST of the codes before and after modification. Because the length of AST differencing sequences is too long, which is not conducive for the model to capture important dependencies. Therefore, we need to simplify representation of the AST difference. Experiments show that our method improves the accuracy of comment updating and achieves good results even in the case of complex code modification.