基于自注意力机制神经机器翻译的软件缺陷自动修复方法

曹鹤玲[1,2,3],刘 昱[1,2],韩 栋[1,2]

(1. 粮食信息处理与控制教育部重点实验室(河南工业大学),河南郑州 450001;
2. 河南工业大学信息科学与工程学院,河南郑州 450001;
3. 河南工业大学河南省粮食信息处理国际联合实验室,河南郑州 450001)

摘 要: 循环神经网络对于代码序列数据有着良好的处理能力,软件缺陷修复的补丁生成模型大多采用循环神经网络实现. 然而,基于循环神经网络的补丁生成模型在处理代码序列中长距离依赖问题时仍然具有局限性,其修复成功率和修复效率较低. 针对此问题,提出一种基于自注意力神经机器翻译的软件缺陷自动修复方法(Self-attention Neural machine translation based automatic software Repair,SNRepair). 首先,为有效缓解源码中的未登录词问题,对数据集引入子词切分技术进行预处理;其次,为解决源代码中棘手的长距离依赖问题并更充分地利用局部信息,构建融合局部建模的Transformer程序补丁生成模型;然后,采用缺陷自动定位技术定位缺陷语句位置,利用参数优化后的Transformer补丁生成模型生成候选补丁;最后,运行测试用例验证候选补丁. 在具有395个真实Java软件缺陷的Defects4J缺陷库上实验评估,结果表明SNRepair方法与对比方法比较,修复成功率和修复效率更高.

关键词: 软件缺陷自动修复;神经机器翻译;自注意力机制;子词切分;局部建模

基金项目: 国家自然科学基金(No.61602154);河南省高等学校重点科研项目(No.22A520024);河南工业大学青年骨干教师培育项目(No.21420158);河南省重大公益专项(No.201300311200)

中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2024)03-0945-12

电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20220734

Self-Attention Neural Machine Translation for Automatic Software Repair

CAO He-ling[1,2,3], LIU Yu[1,2], HAN Dong[1,2]

(1. *Key Laboratory of Grain Information Processing and Control* (*Henan University of Technology*), *Ministry of Education*, *Zhengzhou*, *Henan* 450001, *China*;
2. *College of Information Science and Engineering*, *Henan University of Technology*, *Zhengzhou*, *Henan* 450001, *China*;
3. *Henan International Joint Laboratory of Grain Information Processing*, *Henan University of Technology*, *Zhengzhou*, *Henan* 450001, *China*)

Abstract: Recurrent neural network has good ability of the processing for code sequences, and the patch generation model is mostly implemented by it. However, recurrent neural network-based patch generation models still have some limitations when dealing with long-distance dependencies in code sequences, and their repair success rate and repair efficiency is low. To address the issue, we present SNRepair, an automatic software fault repair based on self-attention neural machine translation. First, the subword tokenization technology is introduced to preprocess the dataset to alleviate the problem of out of vocabulary. Second, a Transformer program patch generation model that integrates local modeling is constructed to alleviate the long-distance dependencies in the source code and make better use of local information. Third, the automatic fault localization technology is used to locate the possible fault position and the Transformer patch generation model through parameter optimization is adopted to generate candidate patches. Finally, the candidate patches are verified by test cases. On the 395 real Java software faults in the Defects4J, the results show that the SNRepair has higher repair success rate and repair efficiency than the compared ones.

收稿日期:2022-06-27;修回日期:2022-11-19;责任编辑:梅志强

Key words: automatic software repair; neural machine translation; self-attention mechanism; subword tokenization; localness modeling

Foundation Item(s): National Natural Science Foundation of China (No.61602154); Key Scientific Research Project of Colleges and Universities in Henan Province (No.22A520024); Cultivation Programme for Young Backbone Teachers in Henan University of Technology (No.21420158); Major Public Welfare Project of Henan Province (No.201300311200)

1 引言

随着软件规模和复杂度的上升，软件缺陷不可避免地存在且数量逐年上升．软件缺陷会导致软件出错，严重的软件缺陷不仅会给企业造成重大经济损失，还会直接威胁人们生命安全．及时成功地修复软件缺陷非常重要，但是修复缺陷的过程很耗时费力．开发人员修复软件缺陷时，也有可能引入新的缺陷，这使得修复软件缺陷过程变得非常困难．

软件缺陷自动修复针对不同修复场景，将整个修复过程自动化，旨在将开发人员从繁重修复任务中解脱出来[1]．Xin 等人[2]提出的基于代码搜索的 ssFix 修复方法，通过相似性对代码库进行搜索，获取索引代码段，从而产生程序补丁．Xuan 等人[3]提出的 Nopol 方法针对条件语句类型错误，利用约束求解器从语义的角度修复错误．Hua 等人[4]提出的 SketchFix 方法通过预定义修复模板来指导补丁生成过程．上述几类方法在缺陷修复方面取得了一定进展，但依赖于有限的人工定义的启发式规则或修复模板，修复成功率有限．为缓解上述局限性，我们将深度学习应用于软件缺陷自动修复中，利用海量的开源代码数据指导补丁生成过程，通过端到端补丁生成模型自动学习，难以通过人工捕获的缺陷程序语句和修复语句之间的复杂关系实现缺陷程序语句到修复语句的转换．

受到深度学习技术成功应用的启发，使用各种先进的深度神经网络进行缺陷修复取得了重大进展．Gupta 等人[5]通过一个多层神经网络来预测 C 语言中编译错误的缺陷位置并产生修复补丁．Chen 等人[6]针对 Java 语言缺陷提出 SequenceR，结合抽象上下文和复制机制来提升补丁的质量．Li 等人[7]提出两层深度网络模型 DLFix，一层用来学习程序缺陷修复上下文信息，另一层来实现缺陷程序到候选补丁的转换过程．虽然这类方法可处理多种不同类型的缺陷且不依赖于人工设计的启发式规则或修复模板，但目前该类方法仍存在如下问题：

（1）这类方法多采用基于循环神经网络（Recurrent Neural Network，RNN）的补丁生成模型，RNN 虽然对于序列数据尤其是变长序列数据有着良好的处理能力，但其顺序建模的方式使其难以实现并行化，且无法很好地处理长距离依赖问题．

（2）这类方法多工作在源代码单词级别，面临着比自然语言更加严重的未登录词问题，SequenceR[6]虽然尝试采用复制机制来解决此问题，但并未得到理想的效果．

为缓解上述问题，我们提出一种融合局部建模的自注意力神经机器翻译的软件缺陷自动修复方法．与基于 RNN 的修复方法相比，该方法能够直接建模序列内部任意代码单元之间的依赖关系，显著缓解了源代码中较为棘手的长距离依赖问题，并且能够实现 GPU（Graphics Processing Unit）上的并行化．此外，采用了基于一元语言模型（Unigram Language Mode，ULM）的子词切分技术预处理数据集，通过一个不太大的源代码子词词表中的子词去拼接代码单元，有效缓解了未登录词问题，增加了搜索空间中的正确补丁数．本文主要贡献包括：

（1）提出一种融合局部建模的自注意力 Transformer 补丁生成模型．该补丁生成模型能够在 GPU 上实现并行化，不仅可以有效处理源代码中棘手的长距离依赖关系问题，还能感知源代码的局部信息进而捕获有用的上下文信息．

（2）将一元语言模型子词切分技术应用于缺陷自动修复领域．该技术对数据集进行子词级别的预处理，有效地缓解了未登录词问题，缩小了搜索空间，增加了搜索空间中的正确补丁个数．

（3）在 Defects4J 缺陷库的 395 个真实 Java 软件缺陷上对 SNRepair 方法进行了实验评估，实验结果表明 SNRepair 比对比方法拥有更好的修复成功率和修复效率．

2 相关研究

近年来许多国内外学者研究软件缺陷自动修复，涌现出许多经典的软件缺陷自动修复方法．基于搜索的缺陷自动修复方法随机采用启发式算法来指导补丁生成过程．Wen 等人[8]利用细粒度的抽象语法树上下文信息来限制搜索空间．Villanueva 等人[9]采用新的适应度函数以防止搜索算法陷入局部最优解．而基于语义的缺陷自动修复方法，将补丁生成问题编码为公式或一个分析过程然后求解．Afzal 等人[10]从程序的执行路径中提取修复约束并在验证过程中动态更新程序规约．Gao 等人[11]对修复约束进行转换，同时扩展了程序合成方法．基于修复模板的缺陷自动修复方法利用人工预定义的修复策略或修复规则来生成程序补丁．Tian 等人[12]提出的修复模板可以对多种缺陷类型自动分类．Koyuncu 等人[13]提出的方法使用编辑语言自动

挖掘各类修复模板．相比于上述三类方法,SNRepair方法没有修复缺陷类型的限制．此外,SNRepair不需依赖于大量的领域知识以及人工制定的规则,可以自动学习缺陷代码和正确代码之间的复杂关系．

随着深度学习技术的发展,应用深度神经网络模型进行软件缺陷自动修复的方法取得了不错的修复效果．Chakrabotry等人[14]提出的CODIT方法通过在基于树的语法感知模型中编码代码结构来学习代码编辑以生成程序补丁．Tang等人[15]提出了基于语法规则转换的修复模型,利用双编码器分别提取程序的序列和语法结构特征以推理符合程序语法的修复补丁．而本文SNRepair方法更加注重学习缺陷语句与正确补丁之间的复杂关系来指导补丁生成．Tufano等人[16]实证研究了基于RNN的缺陷修复方法的可行性,从GitHub上获取真实缺陷用于训练．Chen等人[6]提出的SequenceR方法通过在修复模型中引入复制机制实现了部分代码的复用,一定程度上缓解了未登录词问题．Lutellier等人[17]提出的CoCoNuT方法使用了基于卷积神经网络的神经机器翻译模型,来自动修复软件中的缺陷．Li等人[7]提出一个基于RNN的双层深度学习模型以更好地利用上下文信息．而SNRepair方法采用自注意力机制来缓解代码序列中的长距离依赖问题;融合局部建模以捕获上下文信息;同时引入子词切分技术在子词级别进行数据预处理,有效缓解了未登录词问题．

由于当前最前沿的语言表征模型如GPT(Generative Pre-training Transformer)[18]和BERT(Bidirectional Encoder Representations from Transformers)[19]都基于Transformer模型,且取得了令人瞩目的成果,所以本文构建了融合局部建模的Transformer补丁生成模型．GPT通过上文预测下一个输出,而Transformer模型能更好地利用上下文信息预测输出;同时Transformer模型不需要像BERT花费高昂的代价训练庞大的语料．融合局部建模的Transformer补丁生成模型不仅有效缓解了RNN的长距离依赖问题,还使补丁生成模型具有良好的捕获局部信息的能力．因此,SNRepair方法具有更加优异的修复成功率和修复效率．

3　本文方法

本文提出的基于自注意力神经机器翻译的软件缺陷自动修复方法SNRepair分为3个阶段,如图1所示．阶段1是训练阶段:首先使用子词切分技术预处理数据集;然后将预处理后的数据送入补丁生成模型进行训练,不断调试优化得到最优模型．阶段2是推理阶段:首先使用GZoltar[20]工具中的Ochiai缺陷定位技术获取可能的缺陷位置信息;然后使用子词切分技术对缺陷语句进行预处理,通过补丁生成模型生成相应的候选补丁．阶段3是验证阶段:运行测试套件对候选补丁进行验证,若有补丁通过所有测试用例,则认为该补丁为合理补丁．

3.1　自注意力机制

自注意力机制[21]主要用于构建序列数据内部间的依赖关系,已成功应用于文本蕴涵和阅读理解等任务．

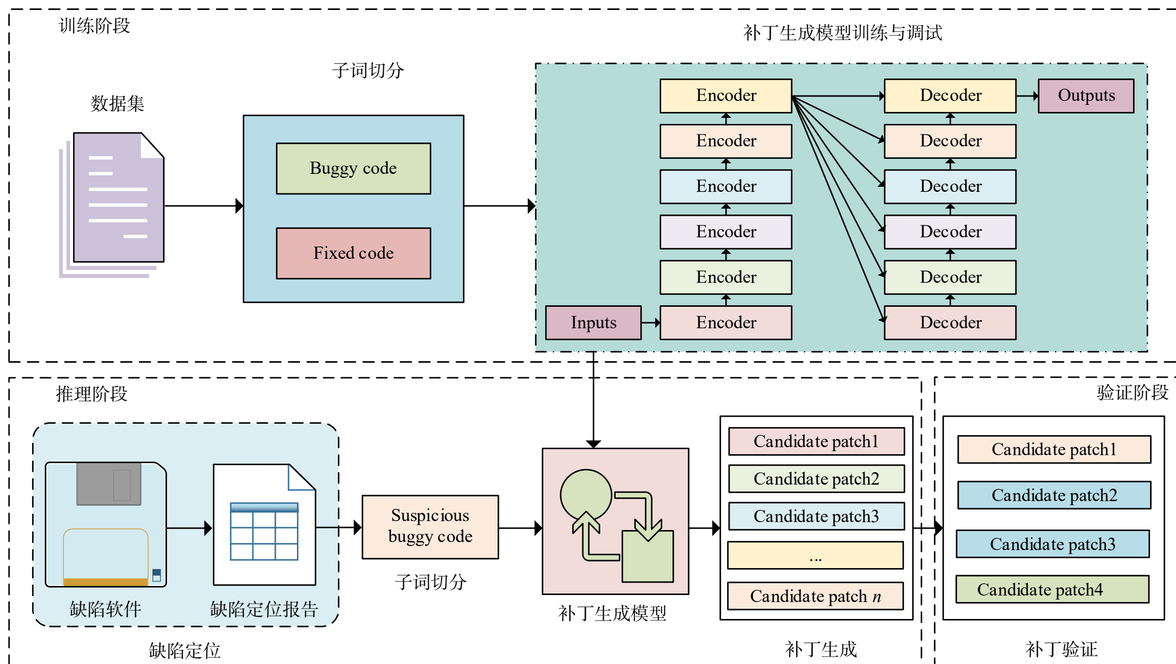

图1　基于自注意力神经机器翻译的软件缺陷自动修复框架图

给定代码$\{x_1, \cdots, x_a\}$序列，采用循环神经网络处理x_a时，依赖于信息x_{a-1}；处理信息x_{a-1}时，依赖于x_{a-2}；以此类推，如果要构建x_a和x_1的依赖关系，需要传递信息$a-1$次．代码序列越长，则传递距离越长，在传递过程中可能导致信息丢失．此外，顺序建模也会限制模型对序列的处理速度．

自注意力机制可以对任意不同位置代码单元之间的关系直接建模．例如代码单元x_a，自注意力机制直接构建x_a与前$a-1$个代码单元之间的关系，即x_a与序列中其它代码单元的距离都是1．这种方式很好地解决了长距离依赖问题，并提升了模型并行处理能力．

3.2 训练阶段

（1） 数据提取

训练数据为代码中的缺陷行、围绕缺陷行的代码（用于表示上下文），以及缺陷行对应的正确修复补丁．图2为训练集上缺陷程序语句实例，粉色部分表示缺陷行，浅绿色部分表示缺陷行对应的正确行，浅蓝色部分表示缺陷行的上下文．上下文行提供了关于程序缺陷的相关语义信息，用于理解缺陷行为和预测修复补丁．为防止输入序列过长，我们对上下文行的大小进行了限制，若输入序列超过限制，SNRepair将会进行截断操作．

```
public class InvalidMatrixExceptionTest extends TestCase
 {
  public void testConstructorMessage ( )
      {
         String msg = "message" ;
       - InvalidMatrixException ex = new InvalidMatrixException ( msg , new Object [ 0 ] ) ;
       +InvalidMatrixException ex = new InvalidMatrixException ( msg , null ) ;
         assertEquals ( msg , ex . getMessage ( ) ) ;
      }
 }
```

图2 缺陷程序语句

（2） 子词切分技术

为缓解未登录词问题，本文从子词角度对数据集进行处理，使用基于ULM的子词切分技术[22]．主要原理是通过子词切分生成源代码子词词表，使用词表中的子词拼装代码单元．为此，采用Google开源工具包SentencePiece[23]中集成的ULM（Unigram Language Model）子词切分算法实现该技术．SentencePiece将句子视为Unicode字符序列，保留代码语句的所有信息，以实现代码语句和子词序列之间的无损可逆转换．

（3） 局部建模Transformer补丁生成模型

融合局部建模的Transformer补丁生成模型有效地缓解了长距离依赖问题，且能在GPU上实现并行化以提升模型效率．如图3所示，模型采用编码器-解码器（Encoder-Decoder）架构．局部建模融合到自注意力机制中用于增强对局部信息的建模能力，这将有助于补丁生成模型更有效地利用上下文信息．自注意力机制具体结构如图3右上角所示，融合局部建模的自注意力机制的具体结构如图3右下角所示．生成补丁时，缺陷程序语句的词嵌入融合位置编码作为编码器的输入，通过编码器的各个层抽象得到含有丰富上下文信息的向量表示，最后传递到解码器，根据前一时刻代码单元输出结果生成当前时刻的代码单元．

编码器：编码器由6个层组成，图3中蓝色框代表其中一层，缺陷程序语句的向量序列是每一层的输入．每个层由两个子层组成：第一个子层是由自注意力机制构成的多头注意力子层，用于对输入的向量序列进行新的表示；第二个子层是一个全连接的前馈神经网络，对输入向量序列进行进一步的转换．上述两个子层的周围都采用了残差连接和层归一化（Add&Layer-Norm）．残差连接可以使深层网络更加有效地传递信息；层归一化用于规范结果向量的取值范围．

解码器：解码器由6个相同的层组成，图3中红色框代表其中一层，每一层由三个子层组成：第一个子层是多头注意力层，加入了掩码机制来屏蔽未来信息以确保训练解码的一致．第二个子层是编码-解码注意力子层，帮助解码器生成目标修复语句的不同位置表示．第三个子层是一个全连接的前馈神经网络．此外，解码器的三个子层同样包含残差连接和层归一化．

位置编码：Transformer模型引入位置编码来表示代码单元之间的顺序关系．模型使用了不同频率的正余弦函数，如式（1）和式（2）所示．

$$\mathrm{PE}_{(\mathrm{pos}, 2a)} = \sin(\mathrm{pos}/10000^{2a/d}) \tag{1}$$

$$\mathrm{PE}_{(\mathrm{pos}, 2a+1)} = \cos(\mathrm{pos}/10000^{2a/d}) \tag{2}$$

其中，pos表示代码单元的位置，d表示每个位置隐藏层的大小，a代表位置编码向量中的第几维．正余弦函数的编码各占一半，因此，当位置编码的维度为512时，a的范围是[0,255]．在Transformer模型中，位置编码和词嵌入向量的维度均为d，将二者相加作为模型输入．正余弦函数是具有上下界的周期函数，用正余弦函数可将长度不同的序列的位置编码的范围都固定到[−1, 1]，这样在与词的编码进行相加时，不会产生太大差距．另外，位置编码的不同维度对应不同的正余弦曲线，为多维的表示空间赋予一定意义．

自注意力机制：自注意力结构如图3右上角所示，矩阵$\boldsymbol{Q}$（Query，查询）、$\boldsymbol{K}$（Key，键值）和$\boldsymbol{V}$（Value，值）由代码单元的表示向量组成的矩阵$\boldsymbol{X}$或上一层的输出进行线性变化得到，用于计算自注意力．$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$的计算如式（3）~式（5）．

$$\boldsymbol{Q} = \boldsymbol{X} \cdot \boldsymbol{W}^Q \tag{3}$$

$$\boldsymbol{K} = \boldsymbol{X} \cdot \boldsymbol{W}^K \tag{4}$$

$$\boldsymbol{V} = \boldsymbol{X} \cdot \boldsymbol{W}^V \tag{5}$$

其中，$\boldsymbol{W}^Q$、$\boldsymbol{W}^K$和$\boldsymbol{W}^V$表示三个不同权值矩阵．自注意

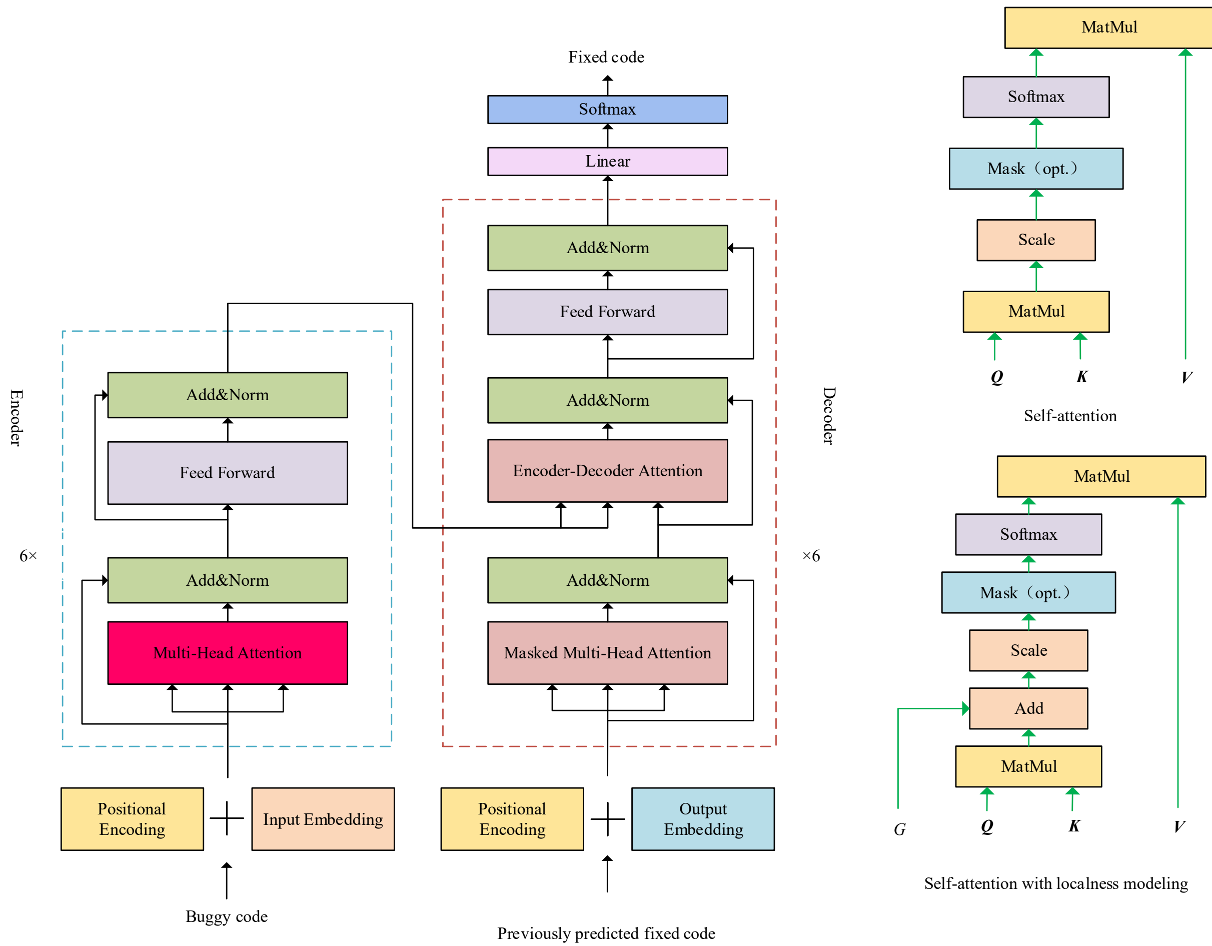

图3　局部建模Transformer补丁生成模型

力的计算如式(6),其中,softmax()为激活函数.

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{6}$$

多头注意力机制:多头注意力机制包含多个自注意力机制. 多头注意力机制允许补丁生成模型在不同表示空间中捕获信息,充分提取内部关系. 将矩阵$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$映射到不同表示空间,利用各自注意力机制独立计算得到不同的上下文向量,拼接和映射后获得最终结果. 多头注意力机制计算如式(7)和(8)所示.

$$\text{MultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{Concat}(\boldsymbol{h}_1,\ldots,\boldsymbol{h}_M)\boldsymbol{W}^{O} \tag{7}$$

$$\boldsymbol{h}_m=\text{Attention}(\boldsymbol{Q}\boldsymbol{W}_m^{Q},\boldsymbol{K}\boldsymbol{W}_m^{K},\boldsymbol{V}\boldsymbol{W}_m^{V}) \tag{8}$$

其中,M表示多头注意力机制的头数,$\boldsymbol{h}_m$表示第m个头获取的上下文向量,$\boldsymbol{W}_m^{Q}$、$\boldsymbol{W}_m^{K}$、$\boldsymbol{W}_m^{V}$和$\boldsymbol{W}^{O}$是参数矩阵.

局部建模:原始的自注意力机制通过加权平均操作来充分考虑所有输入信息,这样的操作分散了注意力的分布,忽略了相邻输入之间关系. 现有研究表明局部建模能够有效提升自注意力机制模型的性能[24]. 鉴于此,我们引入局部建模到Transformer补丁生成模型的自注意力机制中,如图3右下角所示. 在加入局部建模后,当代码单元x_i与代码单元x_j对齐时,自注意力机制能够将更多的注意力放到与x_j相邻的代码单元上,捕捉到有用的局部上下文信息,提高模型缺陷修复能力. 我们将局部建模转化为可学习的高斯偏差G(Gaussian Bias)[25],中心位置P_i和动态窗口D_i分别表示需要关注的区域的中心和范围;将学习到的高斯偏差纳入到原始的注意力分布中,形成一个考虑预期局部环境的修正分布. 加入高斯偏差G后,自注意力计算如式(9)所示.

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}+G\right)\boldsymbol{V} \tag{9}$$

其中,高斯偏差$G\in\mathbf{R}^{I\times I}$,序列长度用$I$表示. 高斯偏差中的任意元素$G_{i,j}\in[0,-\infty)$,测量代码单元$x_j$与中心位置$P_i$的紧密度. $G_{i,j}$的计算如式(10)所示.

$$G_{i,j}=-\frac{(j-P_i)^2}{2\sigma_i^{\,2}} \tag{10}$$

其中,σ_i表示标准差,$\sigma_i=\frac{D_i}{2}$,D_i为窗口大小,中心位置

P_i和窗口大小D_i可以用式(11)计算.

$$\begin{bmatrix} P_i \\ D_i \end{bmatrix} = I \cdot \mathrm{sigmoid}\left(\begin{bmatrix} p_i \\ z_i \end{bmatrix}\right) \tag{11}$$

其中,标量因子I用于将P_i和D_i调节为0和输入序列长度之间的实数.预测分别以两个标量p_i和z_i为条件.中心位置P_i的预测依赖于其对应的查询向量$\boldsymbol{Q_i}$,本文采用了一个前馈神经网络将$\boldsymbol{Q_i}$转化为位置隐藏状态,然后再通过线性投影$\boldsymbol{U}_{\mathrm{p}} \in \mathbf{R}^d$映射到标量$p_i$,具体的计算如式(12)所示.

$$p_i = \boldsymbol{U}_{\mathrm{p}}^{\mathrm{T}} \tanh(W_p \boldsymbol{Q_i}) \tag{12}$$

其中,$W_p \in \mathbf{R}^{d \times d}$,是一个模型参数.

由于Transformer补丁生成模型的多头注意力机制不是执行单一的注意力功能,而是采用M个参数不同的独立注意力模型联合注意来自不同位置的不同表示子空间的信息.因此,本文为每个注意力头分配一个不同的高斯偏差,并将式(11)重新定义为式(13).

$$\begin{bmatrix} P_i^m \\ D_i^m \end{bmatrix} = I \cdot \mathrm{sigmoid}\left(\begin{bmatrix} p_i^m \\ z_i^m \end{bmatrix}\right) \tag{13}$$

其中,p_i^m和z_i^m使用不同的参数进行训练,用以预测第m个注意力头的中心位置和窗口大小.

(4) Transformer补丁生成模型的训练调试

该模型的训练调试包括两个阶段.阶段1是采用预先设置好的参数,使用部分训练数据集训练Transformer补丁生成模型,旨在观察模型训练中损失函数的收敛效果,进而判断模型是否可以像学习自然语言一样学习程序代码语言中的信息.Transformer模型使用交叉熵损失函数.式(14)定义了交叉熵损失,其中$\hat{\boldsymbol{y}}$表示模型输出的分布,$\boldsymbol{y}$表示标准答案,$\hat{\boldsymbol{y}}[k]$和$\boldsymbol{y}[k]$分别表示向量$\hat{\boldsymbol{y}}$和$\boldsymbol{y}$的第k维,$|\boldsymbol{V}|$代表输出向量的维度.损失函数如式(15)所示,其中$\hat{\boldsymbol{Y}}$和$\boldsymbol{Y}$分别表示输出概率分布和标准答案分布,n代表训练样本个数.损失越小说明模型的预测越接近真实输出.

$$L_{\mathrm{ce}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{k=1}^{|\boldsymbol{V}|} \hat{\boldsymbol{y}}[k] \log(\boldsymbol{y}[k]) \tag{14}$$

$$L(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \sum_{j=1}^{n} L_{\mathrm{ce}}(\hat{\boldsymbol{y}}_j, \boldsymbol{y}_j) \tag{15}$$

阶段2在预设参数训练和调试后,将全部的训练集、验证集和测试集对Transformer补丁生成模型进行系统的训练、验证和测试.

3.3 推理阶段

(1)缺陷定位:训练好Transformer补丁生成模型后,进入推理阶段,使用补丁生成模型生成程序补丁.在本文SNRepair方法中,借助GZoltar[20]工具中的Ochiai技术定位缺陷,Ochiai计算如式(16)所示,参数s表示程序缺陷行,参数$n_{\mathrm{ep}}(s)$、$n_{\mathrm{ef}}(s)$分别表示覆盖缺陷行s的成功测试用例总数和失败测试用例总数,参数n_{f}为通过缺陷行s的所有失败测试用例总数.

$$\mathrm{Ochiai}(s) = \frac{n_{\mathrm{ef}}(s)}{\sqrt{n_{\mathrm{f}} \times (n_{\mathrm{ef}}(s) + n_{\mathrm{ep}}(s))}} \tag{16}$$

(2)补丁生成阶段:首先,从缺陷报告中获取怀疑值最大缺陷行的位置信息,由SNRepair自动从缺陷软件提取包含缺陷行及上下文在内的缺陷程序语句.使用SentencePiece子词切分技术[23]对缺陷程序语句进行子词切分,进而得到缺陷程序语句子词序列.将子词序列输入到补丁生成模型中,采用束搜索解码策略来生成程序修复语句.束搜索是自然语言处理领域常见的搜索策略,包含缺陷行及上下文在内的缺陷程序语句通过束搜索解码器被预测,该解码器可以找到近似最大化已训练的补丁生成模型条件概率的输出.对于每次迭代,束搜索算法检查B个最可能的单词(B对应于波束宽度),并根据接下来S个预测步骤的总似然得分(S对应于搜索深度)对它们进行排名.最后,束搜索算法输出根据每个序列的似然度排序的前B个最有可能的序列.补丁生成模型输出的是子词序列,需要将生成修复语句子词序列转化为成源代码样式的修复语句.缺陷定位后的缺陷程序通过补丁生成模型生成的束搜索结果如图4所示,缺陷行用粉色表示,缺陷行的上下文用浅蓝色表示.

```
public class MetagunDesktop {
    public static void main ( String [ ] argv ) {
        -new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 , false ) ;
} }

①new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 , true , false ) ;
②new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 , false , false ) ;
③new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 ) ;
④new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 , false , true ) ;
⑤new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( ) ;
…
…
```

图4 补丁生成

3.4 验证阶段

通过运行测试套件来验证模型生成的多个候选补丁.如果一个候选补丁通过了所有的测试用例,那么它就被视为合理的补丁.如果所有的候选补丁都不能通过验证,则在缺陷位置报告中选择下一个可疑的缺陷位置,通过补丁生成模型再次生成候选补丁进行验证,直到有补丁通过验证.图5是图4对应的候选补丁中最终通过全部测试用例的合理补丁.

```
+new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( new Metagun ( ) , "Metagun" , 320 , 240 ) ;
```

图5补丁验证

4 实验结果及分析

4.1 数据集

本文选取了公开数据集Bugs2Fix[16]及CodRep[26].

这两个数据集均来自Java开源项目. CodRep数据集中包含单行缺陷及其修复, Bugs2Fix数据集中既有单行缺陷又有多行缺陷及其修复. 本文主要研究生成单行补丁来修复缺陷,所以舍弃了Bugs2Fix中的多行缺陷修复数据. 最终所使用的数据集共有40 250对样本. 其中90%设定为训练集,10%为测试集,验证集由训练集再划分,占训练数据的5%.

为评估SNRepair修复效果,需要选取高质量的缺陷库. 本文选取公开缺陷库Defects4J [27],它广泛应用于各类缺陷自动修复方法的评估中,具体缺陷库信息如表1所示.

表1　Defects4J(v1.2.0)缺陷库

项目名称	项目简称	缺陷数量	测试用例数	代码行数
Chart	C	26	2 205	96 000
Closure	Clo	133	7 927	90 000
Lang	L	65	2 245	22 000
Math	M	106	3 602	85 000
Mockito	Moc	38	1 457	11 000
Time	T	27	4 130	28 000
总计	—	395	21 566	332 000

4.2　实验设置

4.2.1　实验环境

模型的设计与实现基于深度学习框架PyTorch(v1.6.0). SNRepair使用Google开源工具包SentencePiece[23]实现ULM子词切分技术;使用GZoltar[20]工具中Ochiai技术进行缺陷定位. 实验运行环境为Ubuntu18.04,12×Xeon E5-2678 v3 CPU,64 GB内存,2×NVIDIA GeForce RTX 2080,16 GB显存.

4.2.2　参数设置

SentencePiece工具包的参数设置如下:character_coverage:0.999 5;model_type: unigram; vocab_size: 8 000;补丁生成模型参数设置如下:batch size: 32; learning rate: 0.003; epoch: 20; dropout: 0.3; label smoothing: 0.1;编码器与解码器层数:6;注意力头:8;隐藏层与词向量维度:256;束搜索宽度:50;每个缺陷修复时间:120 min.

4.3　研究问题

问题1　在修复成功率上,SNRepair与对比方法相比表现如何?

问题2　在修复效率上,SNRepair与对比方法相比表现如何?

问题3　SNRepair学习了哪些修复操作?

为研究上述问题,本文将通过测试用例的补丁定义为合理补丁,如果合理补丁与Defects4J提供的正确补丁相同或语义上相同,那么认为该合理补丁为正确补丁.

4.4　实验结果与分析

为了进一步评估SNRepair,本文选取了10种该领域先进的软件缺陷自动修复方法HDRepair[28]、ssFix[2]、ACS[29]、CapGen[8]、LSRepair[30]、Nopol[3]、SketchFix[4]、CODIT[14]、SequenceR[6]和jGenProg[31]进行对比. 其中, CODIT和SequenceR为基于深度学习的修复方法; HDRepair、ssFix、ACS、CapGen、LSRepair和jGenProg为基于搜索的修复方法;Nopol为基于语义的修复方法; SketchFix为基于模板的修复方法.

4.4.1　研究问题1

在修复成功率方面,首先将SNRepair与不属于深度学习类的方法进行对比,再和同属深度学习类的方法对比,然后分析SNRepair在修复成功率上的优势.

表2是SNRepair与不属于深度学习类的修复方法在Defects4J六个项目上的修复结果. 括号内的数字表示拥有合理补丁缺陷的数量. 例如,SNRepair在Chart项目上的修复结果为5(7),表示SNRepair在Chart项目上可正确修复的缺陷数量是5,拥有合理补丁缺陷数量是7. 表2显示SNRepair成功修复27个缺陷,并为46个缺陷生成了合理补丁,修复成功率为6.84%.

表2　SNRepair与不属于深度学习类的修复方法的修复情况对比

Tools	Chart	Closure	Lang	Math	Time	Mockito	Total	P/%
	26	133	65	106	38	27	395	
HDRepair	0 (2)	0 (7)	2 (6)	4 (7)	0 (1)	0 (0)	6 (23)	1.52
ssFix	3 (7)	2 (11)	5 (12)	10 (26)	0 (4)	0 (0)	20 (60)	5.06
ACS	2 (2)	0 (0)	3 (4)	12 (16)	1 (1)	0 (0)	18 (23)	4.56
CapGen	4 (4)	0 (0)	5 (5)	12 (16)	0 (0)	0 (0)	21 (25)	5.32
LSRepair	3 (8)	0 (0)	8 (14)	7 (14)	1 (1)	0 (0)	19 (37)	4.81
Nopol	1 (6)	0 (0)	3 (7)	1(21)	0 (1)	0 (0)	5 (35)	1.27
SketchFix	6 (8)	3 (5)	3 (4)	7 (8)	0 (0)	0 (1)	19 (26)	4.81
jGenProg	0 (7)	0 (0)	0 (0)	5 (18)	0 (2)	0 (0)	5 (27)	1.27
SNRepair	5 (7)	7 (14)	4(8)	9 (15)	1 (1)	1 (1)	27 (46)	6.84

相比与不属于深度学习类的修复方法,SNRepair在修复成功率上有显著优势. 此外,相比于HDRepair、ssFix、LSRepair、Nopol和jGenProg, SNRepair在拥有合理补丁的缺陷被正确修复的概率上亦有显著优势.

SNRepair方法与深度学习修复方法CODIT[14]和SequenceR[6]在Defects4J上的缺陷修复对比如表3所示. 表中展示了每种方法可正确修复的缺陷数量以及每个被正确修复的缺陷的ID. 使用深度学习的缺陷修复方法CODIT和SequenceR在设计时不使用缺陷定位工具,直接提供缺陷正确位置. 为对比公平性,我们也采用直接给出缺陷正确位置进行修复. 从表3中可以看出,在直接给定缺陷正确位置信息后,SNRepair正确修复了32个缺陷,比直接使用Ochiai缺陷定位多修复

表3 SNRepair与CODIT、SequenceR方法缺陷修复情况对比

项目	缺陷总量	可修复的缺陷		
		SNRepair	CODIT	SequenceR
Chart	26	C1, C8, C11, C13, C20	C8, C10, C11, C12	C1, C9, C11
Closure	133	Clo10, Clo18, Clo38, Clo52, Clo73, Clo86, Clo93, Clo113, Clo125	Clo86,Clo92,Clo93	Clo18, Clo73, Clo86
Math	106	M30, M33, M41, M57, M 59, M63, M69, M70, M75, M80, M85, M96	M30, M49, M57, M59, M70, M98	M30, M57, M58, M75, M82, M85
Lang	65	L6, L21, L29, L59	L6, L21, L26	L6, L59
Time	38	T4	—	—
Mockito	27	Moc8	—	—
总计	395	32 (↑5)	16	14
成功率	—	8.10% (↑18.5%)	4.05%	3.54%

了5个缺陷,修复成功率提高了18.5%.

本文SNRepair与深度学习修复方法CODIT和SequenceR修复缺陷数量对比图6所示. 从表3和图6综合分析,本文方法比CODIT和SequenceR方法多修复了16和18个缺陷,且本文方法在Defects4J的6个项目上的修复数量均高于对比方法. 在这6个项目上,本文方法比CODIT方法多修复了1、6、6、1、1和1个缺陷,比SequenceR方法多修复了2、6、6、2、1和1个缺陷.

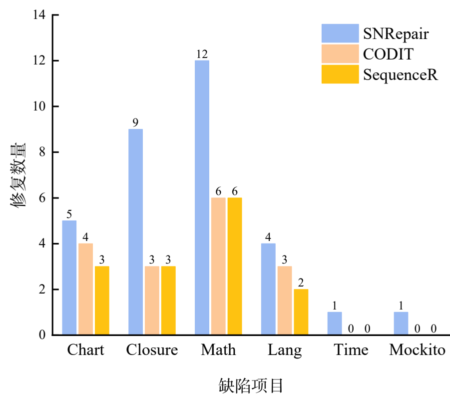

图6 SNRepair与CODIT、SequenceR修复数量对比

本文SNRepair方法与CODIT、SequenceR修复情况对比维恩图如图7所示. 从图中可以看出,SNRepair方法在Chart项目上修复了CODIT和SequenceR均不能修复的缺陷C13和C20;SNRepair方法在Closure项目上修复了CODIT和SequenceR均不能修复的缺陷Clo10、Clo38、Clo52、Clo113和Clo125;SNRepair方法在Math项目上修复了CODIT和SequenceR均不能修复的缺陷M33、M41、M63、M69、M80和M96;SNRepair方法在Lang项目上修复了CODIT和SequenceR均不能修复的缺陷L29;SNRepair方法在Time项目上修复了CODIT和SequenceR均不能修复的缺陷T4;SNRepair方法在Mockito项目上修复了CODIT和SequenceR均不能修复的缺陷Moc8.

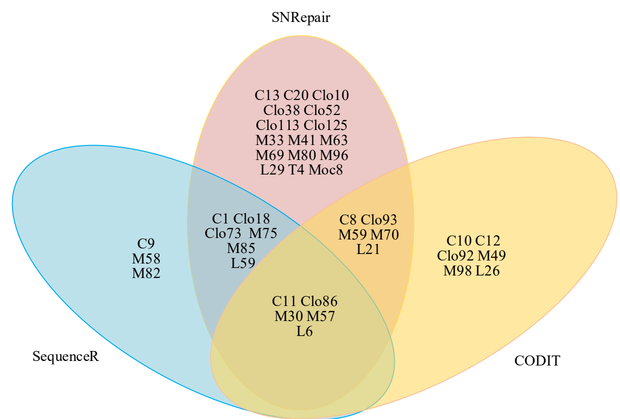

图7 SNRepair与CODIT、SequenceR修复情况对比维恩图

综合分析表2、表3、图6和图7,可以看出SNRepair方法在修复成功率上更具优势,主要原因是子词切分技术的使用有效缓解了未登录词问题,增加了搜索空间中正确的补丁个数. 此外,构建并使用了融合局部建模的Transformer补丁生成模型能够保持捕获长距离依赖能力,同时也拥有良好的对局部源代码信息建模的能力,具有良好的预测准确性.

为了进一步探究SNRepair对于缺陷修复的适应性,我们对正确修复的32个缺陷进行分类统计. 根据程序自身的定义和特性,Defects4J上的缺陷共可分为赋值类错误(Assignment)、条件表达式错误(Conditional)、循环错误(Loop)、方法调用错误(Method Call)、变量错误(Variable)、返回表达式错误(Return)、对象实例化错误(Object Instantiation)、方法定义错误(Method Definition)、异常(Exception)、类型错误(Type)10类[32]. 表4显示了SNRepair正确修复缺陷的类型,从表中可以看出SNRepair所修复的缺陷种类覆盖了10类缺陷种的8类,说明SNRepair具备良好的缺陷修复适应性. 此外,SNRepir所修复的方法调用错误类缺陷多达18个,变量错误类缺陷达12个,亦可说明SNRepair在修复这两类缺陷上具有很大优势.

表4　SNRepair修复缺陷类型统计

缺陷类型	缺陷ID	总计
Assignment	C11, C13, Clo93, M41, M69, M80, M96	7
Conditional	C1, Clo18, Clo38, Clo73, Clo113, Clo125, M85, Moc8	8
Loop	M41	1
Method Call	C8, C13, C20, Clo10, Clo52, Clo93, Clo113, Clo125, M33, M63,M69, M70, M75, M96, L6, L21, L59, T4	18
Variable	C8, C11, C20, M30, M33, M41, M57, M59, L6, L21, L59, T4	12
Return	Clo10, Clo52, Clo86, M59, M63, M70, M75, L21	8
Object Instantiation	C13, T4	2
Method Definition	L29	1
Exception	—	0
Type	—	0

4.4.2　研究问题2

在缺陷自动修复问题中,生成候选补丁所需的时间远远小于补丁验证所需的时间,这是因为对候选补丁进行验证需要重复地运行全部的测试用例,测试用例的运行是十分耗时的. 在为一个缺陷找到合理补丁时,如果验证的候选补丁数越少,则运行测试用例花费的时间越少,进而修复效率也相对越高. 因此,在此问题中,我们采用了在为缺陷找到合理补丁时,共验证了多少个候选补丁这一标准[33]来检验和比较SNRepair方法的效率,通过效率可以衡量SNRepair的方法在不浪费计算资源、时间和精力的情况下生成有效补丁的能力. 这一标准最早由Qi[33]等人提出来,且近年来亦被JAID[34]和PraPR[35]等研究所采用.

针对研究问题2,由于基于深度学习的缺陷自动修复方法在补丁生成方式上与基于搜索的、语义的和模板的方法本质上不同,因此在修复效率上,本文只与基于深度学习的方法进行对比. 本文复现了SequenceR[6],并记录了其在Defects4J上的详细修复情况. 在生成候选补丁时,我们将束宽度B统一设置为50. 为避免偶然因素的影响,我们多次运行了SequenceR和SNRepair,所展示的实验数据均为多次运行后的平均值.

选取了Defects4J数据集上SNRepair与SequenceR均可产生合理补丁的缺陷共17个. 统计了SNRepair方法与SequenceR方法在为这17个缺陷找到合理补丁时,验证候选补丁数量如图8所示. 从图8中可以看出,在为这17个缺陷找到合理补丁时,SNRepair平均验证了8个候选补丁,而SequenceR平均验证了12个候选补丁,SNRepair比SequenceR少验证了4个补丁. 为了更加全面的对SequenceR和SNRepair的修复效率进行比较,统计了两种方法在Defects4J的每个项目上的候选补丁验证数量平均值,如图9所示. 从图9可以看出,SNRepair在Chart、Closure、Math、Lang项目上的候选补丁验证数量平均值分别比SequenceR少5、3、1、2个. 综合图8和图9,可以看出SNRepair比SequenceR在修复效率上更具优势.

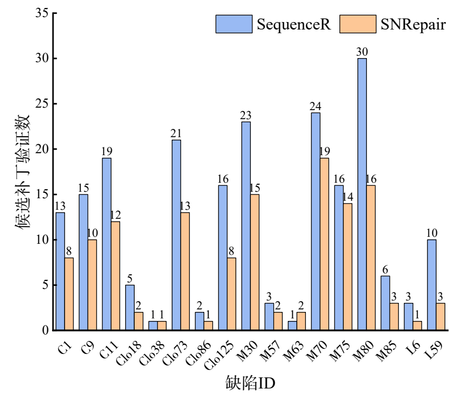

图8　SequenceR和SNRepair在17个缺陷上的候选补丁验证数量对比

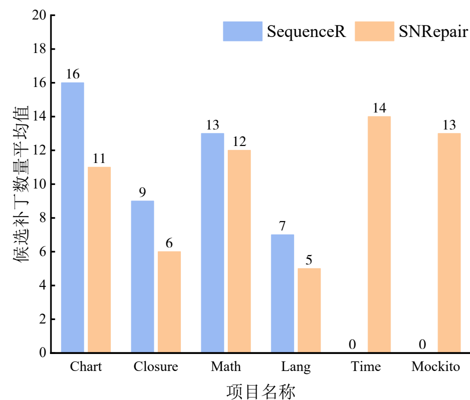

图9　在Defects4J每个项目上SequenceR和SNRepair生成候选补丁验证数量平均值对比

4.4.3　研究问题3

我们采用和文献[6,36]一样的方式,对SNRepair学习到的修复操作(即SNRepair模拟了哪种类型的修复操作来生成程序补丁)进行深入研究,通过修复操作可以判断SNRepair可修复的具体缺陷类型. 为此,我

们手动分析了SNRepair在Defects4J(v1.2.0)上生成的32个正确补丁,并对生成正确补丁所使用的不同的修复操作进行了统计.

从图10可以看出,SNRepair所使用的修复操作类型达13种. 其中,变量修改操作(Var. mod)、返回表达式修改操作(Ret. E. mod)、对象实例化修改操作(O. I. mod)、方法定义修改操作(M. D. mod)、方法调用修改操作(M. C. mod)、方法调用删除操作(M. C. rem)、方法调用增加操作(M. C. add)、循环修改操作(Lp. mod)、条件表达式修改操作(Con. E. mod)、条件表达式减少操作(Con. E. red)、条件表达式扩展操作(Con. E. exp)、条件分支增加操作(Con. B. add)和赋值语句更改操作(Ass. mod)各使用了12,8,2,1,15,2,7,2,3,1,3,1,7次. 使用次数最多的是方法调用修改操作,共15次,使用次数最少的是方法定义修改、条件表达式减少和条件分支增加操作,各1次. 多种修复操作的使用证明SNRepair学到了修复缺陷的不同方式. 此外,SNRepair每生成一个正确补丁平均需要执行2个修复操作,多于SequenceR[6]和ENCORE[36]. 上述结果不仅说明了SNRepair可以修复多种类型的软件缺陷,还说明了SNRepair比SequenceR和ENCORE更易修复较为复杂的缺陷.

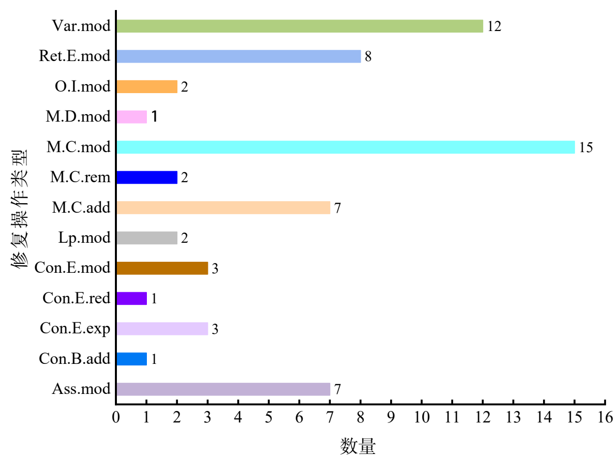

图10 SNRepair在Defects4J上生成正确补丁所使用的修复操作统计

5 结束语

本文提出一种基于自注意力神经机器翻译的软件缺陷自动修复方法SNRepair. 首先,为解决未登录词问题,引入子词切分算法对数据集预处理,从而提升搜索空间中的正确补丁数. 然后,构建并使用了融合局部建模的Transformer补丁生成模型,很好地缓解了源代码中的长距离依赖问题,且局部建模的引入提高了补丁生成模型对源代码局部信息的感知. 该方法有训练、推理和验证三个阶段:对数据集进行子词切分、对Transformer模型进行调试训练;采用Transformer补丁生成模型生成子词形式补丁,并转化为源代码形式补丁;运行测试套件对模型所生成的补丁进行检验. 在Defects4J缺陷库上进行验证,SNRepair方法在修复成功率和效率上优于对比方法.

参考文献

[1] 李斌, 贺也平, 马恒太. 程序自动修复: 关键问题及技术[J]. 软件学报, 2019, 30(2): 244-265.
LI B, HE Y P, MA H T. Automatic program repair: Key problems and technologies[J]. Journal of Software, 2019, 30(2): 244-265. (in Chinese)

[2] XIN Q, REISS S P. Leveraging syntax-related code for automated program repair[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE, 2017: 660-670.

[3] XUAN J F, MARTINEZ M, DEMARCO F, et al. Nopol: Automatic repair of conditional statement bugs in Java programs[J]. IEEE Transactions on Software Engineering, 2017, 43(1): 34-55.

[4] HUA J R, ZHANG M S, WANG K Y, et al. Towards practical program repair with on-demand candidate generation [C]//Proceedings of the 40th International Conference on Software Engineering. New York: ACM, 2018: 12-23.

[5] GUPTA R, PAL S, KANADE A, et al. DeepFix: Fixing common C language errors by deep learning[C]//Proceedings of the 31st AAAI Conference on Artificial Intelligence. San Francisco: AAAI Press, 2017: 1345-1351.

[6] CHEN Z M, KOMMRUSCH S J, TUFANO M, et al. Sequencer: Sequence-to-sequence learning for end-to-end program repair[J]. IEEE Transactions on Software Engineering, 2021, 47(9): 1943-1959.

[7] LI Y, WANG S H, NGUYEN T N. DLFix: Context-based code transformation learning for automated program repair [C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York: ACM, 2020: 602-614.

[8] WEN M, CHEN J J, WU R X, et al. Context-aware patch generation for better automated program repair[C]//Proceedings of the 40th International Conference on Software Engineering. New York: ACM, 2018: 1-11.

[9] VILLANUEVA O M, TRUJILLO L, HERNANDEZ D E. Novelty search for automatic bug repair[C]//Proceedings of the 2020 Genetic and Evolutionary Computation Conference. New York: ACM, 2020: 1021-1028.

[10] AFZAL A, MOTWANI M, STOLEE K T, et al. SOSRepair: Expressive semantic search for real-world program

repair[J]. IEEE Transactions on Software Engineering, 2021, 47(10): 2162-2181.

[11] GAO X, WANG B, DUCK G J, et al. Beyond tests: Program vulnerability repair via crash constraint extraction [J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(2): 1-27.

[12] TIAN Y C, RAY B. Automatically diagnosing and repairing error handling bugs in C[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017: 752-762.

[13] KOYUNCU A, LIU K, BISSYANDÉ T F, et al. FixMiner: Mining relevant fix patterns for automated program repair[J]. Empirical Software Engineering, 2020, 25(3): 1980-2024.

[14] CHAKRABORTY S, DING Y, ALLAMANIS M, et al. CODIT: Code editing with tree-based neural models[J]. IEEE Transactions on Software Engineering, 2022, 48(4): 1385-1399.

[15] TANG Y, ZHOU L, BLANCO A, et al. Grammar-based patches generation for automated program repair[C]// Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. Stroudsburg: Association for Computational Linguistics, 2021: 1300-1305.

[16] TUFANO M, WATSON C, BAVOTA G, et al. An empirical study on learning bug-fixing patches in the wild via neural machine translation[J]. ACM Transactions on Software Engineering and Methodology, 2019, 28(4): 1-29.

[17] LUTELLIER T, PHAM H V, PANG L, et al. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2020: 101-114.

[18] BROWN T B, MANN B, RYDER N, et al. Language models are few-shot learners[C]//Proceedings of the 34th International Conference on Neural Information Processing Systems. New York: ACM, 2020: 1877-1901.

[19] DEVLIN J, CHANG M, LEE K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding[EB/OL]. (2018-10-11) [2022-05-01]. https://arxiv. org /abs/1810.04805.

[20] RIBOIRA A, ABREU R. The GZoltar project: A graphical debugger interface[C]//Proceedings of the International Academic and Industrial Conference on Practice and Research Techniques. Berlin: Springer-Verlag, 2010: 215-218.

[21] LIN Z H, FENG M W, SANTOS C N, et al. A structured self-attentive sentence embedding[EB/OL]. (2017-05-29) [2022-05-01]. https://arxiv.org/abs/1703.03130.

[22] KUDO T. Subword regularization: improving neural network translation models with multiple subword candidates [C]//Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics. Stroudsburg: Association for Computational Linguistics, 2018: 66-75.

[23] KUDO T, RICHARDSON J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing[C]//Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Stroudsburg: Association for Computational Linguistics, 2018: 66-71.

[24] LUONG T, PHAM H, MANNING C D. Effective approaches to attention-based neural machine translation [C]//Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Stroudsburg: Association for Computational Linguistics, 2015: 1412-1421.

[25] YANG B S, TU Z P, WONG D F, et al. Modeling localness for self-attention networks[C]//Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Stroudsburg: Association for Computational Linguistics, 2018: 4449-4458.

[26] CHEN Z, MONPERRUS M. The CodRep machine learning on source code competition[EB/OL]. (2018-07-05) [2022-05-01]. https://arxiv.org/abs/1807.03200.

[27] JUST R, JALALI D, ERNST M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York: ACM, 2014: 437-440.

[28] LE X B D, LO D, LE GOUES C. History driven program repair[C]//2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Piscataway: IEEE, 2016: 213-224.

[29] XIONG Y F, WANG J, YAN R F, et al. Precise condition synthesis for program repair[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2017: 416-426.

[30] LIU K, KOYUNCU A, KIM K, et al. LSRepair: Live search of fix ingredients for automated program repair [C]//2018 25th Asia-Pacific Software Engineering Conference (APSEC). Piscataway: IEEE, 2018: 658-662.

[31] MARTINEZ M, DURIEUX T, SOMMERARD R, et al. Automatic repair of real bugs in Java: A large-scale exper-

iment on the defects4j dataset[J]. Empirical Software Engineering, 2017, 22(4): 1936-1964.
[32] SOBREIRA V, DURIEUX T, MADEIRAL F, et al. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). Piscataway: IEEE, 2018: 130-140.
[33] QI Y H, MAO X G, LEI Y, et al. Using automated program repair for evaluating the effectiveness of fault localization techniques[C]//Proceedings of the 2013 International Symposium on Software Testing and Analysis. New York: ACM, 2013: 191-201.
[34] CHEN L S, PEI Y, FURIA C A. Contract-based program repair without the contracts[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE, 2017: 637-647.
[35] GHANBARI A, BENTON S, ZHANG L M. Practical program repair via bytecode mutation[C]//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2019: 19-30.
[36] LUTELLIER T, PANG L, PHAM V H, et al. ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair[EB/OL]. (2019-06-20)[2022-05-01]. https://arxiv.org/abs/1906.08691.

作者简介

曹鹤玲 女,1980年5月出生于河南南阳.河南工业大学信息科学与工程学院副教授,硕士生导师,CCF高级会员.主要研究领域为软件分析与测试、深度学习技术.
E-mail: caohl@haut.edu.cn

刘 昱 男,1997年3月出生于河南郑州.河南工业大学信息科学与工程学院硕士研究生,CCF会员.主要研究领域为软件分析与测试.
E-mail: zzhautly@163.com

韩 栋 男,1997年1月出生于河南驻马店.河南工业大学信息科学与工程学院硕士研究生.主要研究领域为软件分析与测试.
E-mail: handong531@163.com