Machine Translated by Google

2022 IEEE/ACM 第 44 届软件工程国际会议 (ICSE)

亲爱的:一种新的基于深度学习的自动程序修复方法

易立新
泽西研究所美国新泽西州技术学院
yl622@njit.edu

Shaohua Wang*新
泽西研究所。美国新泽西州技术学院
davidsw@njit.edu

Tien N. Nguyen德克
萨斯大学达拉斯分校 美国德克萨
斯州
tien.n.nguyen@utdallas.edu

抽象的

现有的基于深度学习 (DL) 的自动程序修复(APR) 模型在修复一般软件缺陷方面存在局限性。我们介绍了 DEAR,这是一种基于 DL 的方法,它支持修复一般错误,这些错误需要一次对一个或多个代码块中的一个或多个连续语句进行相关更改。我们首先设计了一种新的故障定位 (FL) 技术,用于结合传统的基于频谱 (SB) 的多块、多语句修复

具有深度学习和数据流分析的 FL。它获取 SBFL 模型返回的有缺陷的语句,检测有缺陷的 hunks并立即修复,并在 hunk 中扩展有缺陷的语句以包括周围的其他可疑语句。我们设计了一个两层的、基于树的 LSTM 模型,该模型结合了循环训练并使用分而治之的策略来学习适当的代码转换,以便在由周围子树组成的合适的固定上下文中固定多个语句。我们进行了几个实验来评估三个数据集上的 DEAR:Defects4J（395 个错误）、BigFix（+26k 个错误)和 CPatMiner（+44k 个错误）。在 Defects4J 数据集上,DEAR在仅使用 top-1 补丁的自动修复错误数量方面优于基线 42%–683% 。在 BigFix 数据集上,它修复的错误比现有的具有top-1 补丁的基于 DL 的 APR 模型多 31-145 个错误。在 CPatMiner 数据集上,在 667 个已修复的错误中,有 169 个 (25.3%) multi-hunk/multi-statement 错误。 DEAR 比最先进的基于 DL 的 APR 模型多修复了 71 和 164 个错误,包括 52 和 61 个多块/多语句错误。

CCS 概念

·软件及其工程→软件维护工具。

关键词

自动程序修复;深度学习;故障定位;

ACM 参考格式： Yi Li、
Shaohua Wang 和 Tien N. Nguyen。 2022. DEAR:一种基于深度学习的新型自动程序修复方法。第 44 届国际软件工程会议 (ICSE 22),2022 年 5 月 21-29 日,美国宾夕法尼亚州匹兹堡。 ACM,美国纽约州纽约市,13 页。 https://doi.org/10。1145/3510003.3510177

通讯作者

允许免费制作本作品的全部或部分的数字或硬拷贝供个人或课堂使用,前提是复制或分发不是为了盈利或商业利益,并且副本带有本通知和首页上的完整引用.必须尊重非 ACM 拥有的本作品组件的版权。允许使用信用抽象。要以其他方式复制或重新发布,以在服务器上发布或重新分发到列表,需要事先获得特定许可和/或付费。从 permissions@acm.org 请求权限。

ICSE 22,2022 年 5 月 21-29 日,美国宾夕法尼亚州匹兹堡 © 2022 计算机协会。
ACM ISBN 978–1–4503–9221–1/22/05。 15.00
美元 https://doi.org/10.1145/3510003.3510177

1 简介

研究人员提出了几种方法来帮助开发人员自动识别和修复软件中的缺陷。这种方法称为自动程序修复 (APR)。 APR方法一直在利用基于搜索的软件工程、软件挖掘、机器学习 (ML) 和深度学习 (DL) 领域的各种技术。

对于基于搜索的方法[9、10、24、30] ,搜索策略是在通过运算符对错误代码进行变异而产生的潜在解决方案空间中执行的。其他方法使用软件挖掘来从先前的错误修复[15、17、19、20、27]或类似代码[28、32]中挖掘和学习修复模式。修复模式位于源代码级别[19、20]或更改级别[13、16、40] 。机器学习已被用于挖掘修复模式,候选修复根据它们的可能性进行排名[21、22、33] 。虽然一些基于 DL 的 APR方法学习了类似的修复[11、41、42],但其他方法使用机器翻译或具有各种代码抽象的神经网络模型来生成补丁 [5、6、12、18、35、38、39] .

尽管他们取得了成功,但最先进的基于 DL 的 APR方法在修复一般缺陷方面仍然受到限制,这些缺陷涉及修复对文件相同或不同部分或不同文件中的多个语句的更改（它们是被称为帅哥）。现有的基于 DL 的方法都无法通过一次对多个块中的多个语句进行依赖更改来自动修复错误。他们只支持修复单个语句。如果我们在当前语句上使用这样的工具,该工具会将该语句视为不正确,而将其他语句视为正确。这不成立,因为要修复当前语句,不能将剩余的未修复语句视为正确代码。因此,当使用现有的基于 DL 的 APR 工具修复多块/多语句错误的单个语句时,它可能是不准确的。虽然 DL为修复学习提供了好处,但这种限制使得基于 DL 的APR 方法的能力不如支持多语句修复的其他方向（基于搜索和基于模式的 APR）。

在本文中,我们的目标是通过引入DEAR 来推进基于深度学习的 APR,DEAR 是一种基于 DL 的模型,支持修复一般错误,同时对属于一个或多个错误代码块的一个或多个错误语句进行相关更改.为此,我们做出了以下关键技术贡献。

首先,我们开发了一种针对多块、多语句错误的故障定位 (FL) 技术,该技术将传统的基于频谱的 FL (SBFL) 与 DL 和数据流分析相结合。DEAR 使用 SBFL 方法来识别可疑错误语句的排名列表。然后,它使用该错误语句列表来推导需要通过微调预训练的 BERT 模型[8]来修复在一起的错误块,以了解语句之间的修复关系。我们还设计了一个扩展算法,它接受一个错误的语句

in a hunk 作为种子,并扩展以包含周围其他可疑的连续语句。为此,我们使用 RNN模型将语句分类为错误或非错误,并使用数据流分析进行调整,然后形成错误块。

其次,在扩展步骤之后,我们已经识别出所有带有 buggy 语句的 buggy hunk(s)。我们开发了一种组合方法来学习,然后生成多块、多语句修复。在我们的方法中,从错误的语句中,我们使用分而治之的策略来学习抽象语法树 (AST) 中的每个子树转换。具体来说,我们使用基于 AST 的差分技术来推导训练数据中基于 AST 的细粒度更改以及错误代码和固定代码之间的映射。这些细粒度的子树映射帮助我们的模型避免错误和固定代码的不正确对齐,因此,在学习修复的多个 AST 子树转换时更加准确。

第三,我们增强并编排了一个基于树的两层长短期记忆 (LSTM) 模型[18],该模型具有一个注意层和一个循环训练,以帮助 DEAR 在合适的上下文中学习正确的代码修复更改周边代码。

对于我们的故障定位识别出的每个有缺陷的 AST 子树,我们将其编码为向量表示,并应用该 LSTM 模型来导出固定代码。在第一层,它学习修复上下文,即围绕有问题的 AST 子树的代码结构。在第二层,它学习代码转换以使用上下文作为附加权重来修复有问题的子树。

最后,可能存在多个有问题的子树。为了在训练中为每个有问题的子树构建周围的上下文,我们在其他有问题的子树（而不是那些有问题的子树本身)的修复之后包含 AST 子树。基本原理是修复后的子树实际上代表了 的正确周围代码。 （注意:在训练中,固定子树是已知的） 。

我们在三个大型数据集上进行了实验来评估 DEAR :Defects4J [1] （395 个错误） 、BigFix [18] （+26k 个错误)和 CPat Miner 数据集[26] （+44k 个错误） 。基于 DL 的基线方法包括 DLFix [18]、 CoCoNuT [23]、 SequenceR [6]、Tufano19 [38]、 CODIT [5] 和 CURE [14]。 DEAR仅使用 Top-1 在所有三个数据集上修复的错误分别比性能最佳的基线 CURE 多31%（即 +11） 、5.6%（即 +41)和9.3%（即 +31)补丁,平均训练参数减少七倍。在Defects4J 上,它在修复错误的数量方面优于那些基线 42%–683% 。在 BigFix 上,它修复的错误比那些带有 top-1 补丁的基线多 31-145 个错误。在 CPatMiner 上,在DEAR 修复的 667 个错误中,有 169 个 (25.3%) multi-hunk/multi- statement 错误。与现有的基于 DL 的 APR 工具 CoCoNuT、DLFix 和 CURE相比,DEAR 修复了 71、164 和 41 个错误,包括52、61 和 40 个多块/多语句错误。我们还将DEAR 与 8 种最先进的基于模式的 APR 工具进行了比较。我们的结果表明,DEAR 生成的结果与基于模式的顶级 APR 工具具有可比性和互补性。在 Defects4J 上,DEAR 修复了 12 个错误（共 47 个） ,其中包括顶级基于模式的 APR 工具无法修复的7 个多块/多语句错误。

简而言之,本文的主要贡献包括A. Advancing based APR for general bugs with multi hunk/multi-statement fixes: DEAR advances based APR for general bug。我们表明,基于深度学习的 APR 可以实现与其他 APR 方向相当和互补的结果。

B. 先进的基于 DL 的 APR 技术:

```
1 public boolean verifyUserInfo(String UID, String password, String SSN) { 2 String retrieved_password =
     ; 3 字符串 retrieved_SSN =      ; 4 if (UID !=
null) { 5 - retrieved_password =
getPassword(UID); 6 +
retrieved_password = getPassword(toUpperCase(UID)); 7 + } 其
他 {

8 +          返回假;
9 + } 10
-布尔password_check= compare(password,retrieved_password); 11 + boolean password_check=
compare(passwordHash(password),retrieved_password);
12         if (password_check)
13           { retrieved_SSN = getSSN(UID);布尔
14           SSN_check = 比较（SSN,retrieved_SSN） ;如果（SSN_check){返回真;
15
16
17           }
18         }
19         返回假;
20 }
```

图 1:具有多个相关更改的一般修复

1) 一种用于多块、多语句修复的新型 FL 技术,将基于频谱的 FL 与 DL 和数据流分析相结合;

2) 采用分而治之策略的组合方法学习并生成多块、多语句修复;和

3) 双层 LSTM 模型的设计和编排,通过注意力层和循环训练进行增强。

C. 广泛的实证评估: 1)DEAR 优于现有的基于 DL 的 APR 工具; 2) DEAR 是第一个基于 DL 的 APR模型,在修复错误的数量方面与最先进的基于模式的工具处于同一水平,并生成互补的结果; 3) 我们的数据和工具是公开的 [2]。

2 动机

2.1 激励示例让我们展示一个错误修复

示例和我们对激励的观察。图 1 显示了verifyUserInfo中的错误示例,它根据数据库中的用户记录验证给定的用户 ID、密码和社会安全号码。这个错误体现在三个方面。首先,开发者忘记处理UID为null的情况。

因此,为了修复,他在第 7-9 行添加了一个else分支。二是开发者忘记对UID进行大写转换,导致错误,因为数据库中用户ID的记录都是大写字母。相应的错误修复更改是在第 6 行添加了对UID上的toUpperCase()的调用。第三,由于存储在数据库中的密码是通过散列编码的,因此用户输入的密码在比较之前需要进行散列反对数据库中的那个。因此,开发人员在第 11 行调用方法compare()之前添加了对passwordHash()的调用。从这个例子中,我们有以下观察结果:

观察 1 [A Fix with Dependent Changes to Multiple Statements]:此错误需要在同一个修复中同时对多个语句进行依赖修复更改:1) 添加带有return 语句的else分支（第 7-9 行） ,2) 添加toUpperCase在第 6 行,以及 3) 在第 11 行添加passwordHash。一次更改单个语句不会修复错误,因为给定的参数UID和密码都需要正确处理。

UID需要空检查和大写,密码需要散列。对多个语句的那些相关更改必须在同一个修复程序中同时发生,才能使程序通过测试用例。

Machine Translated by Google

最先进的基于 DL 的 APR 方法[6、18]一次修复一个单独的语句。在图1中,故障定位工具返回了两条错误行:第5行和第10行。假设使用这样一个基于DL的APR工具来修复第5行的语句。它将对第5行的语句进行修复更改(例如,修改第 5 行并添加第 7-9 行),但是,假设第 10 行和其他行的陈述是正确的。有了这个不正确的假设,这样的修复不会使代码通过测试用例,因为必须进行两个更改。因此,单个语句、基于 DL 的 APR 工具无法通过一次修复一个错误语句来修复此错误。通常,错误可能需要对同一个修复中的多个语句(可能是多个块)进行相关更改。

此外,基于模式的 APR 工具可能无法修复此缺陷,因为此示例中的代码是特定于项目的,可能与任何错误修复模式都不匹配。

观察 2 [多对多 AST 子树转换]:修复可能涉及对多个子树的更改。例如, if语句有一个新的else分支。调用getPassword()的参数被修改为调用toUpperCase()。

此修复还涉及多对多子树转换。在此示例中,修复将两个有问题的语句(第 5 行和第 10 行)转换为四个语句(具有新else分支的if语句、第 8 行的return语句、第 6 行带有toUpperCase 的修改语句以及修改后的在第 11 行带有passwordHash 的语句)。因此,一个修复可以分解为多个子树转换,如果使用具有分而治之策略的组合方法,我们可以学习各个转换。

观察 3 [正确修复上下文]:错误修复通常取决于周围代码的上下文。例如,要从给定的UID 获取密码,需要将 ID 大写,因此在正确的代码中,在调用getPassword方法时,很可能会调用toUpperCase方法。因此,建立正确的固定上下文很重要。在图 1 中,模型需要学习修复(第 5 行→第 6 行)wrt 周围的代码,这需要包括第 11 行的修复代码(而不是第 10 行,因为第 10 行有错误)。要修复第 5 行,正确的上下文必须包括第 11 行的passwordHash 。因此,修复错误语句的正确上下文必须包括另一个错误语句的固定代码 ,而不是 本身。

2.2 关键思想从观察

中,我们得出以下关键思想:关键思想1. A Fault Localization Method for Multi-hunk, Multi- statement Patches:根据观察 1,我们设计了一种新的 FL方法,它结合了传统的基于频谱的 FL(SBFL))与 DL 和数据流分析。我们使用 SBFL 来获得 can didate 语句的排名列表,以用它们的可疑分数来修复。我们在两个任务中扩展了 SBFL 的结果。首先,我们设计了一个 hunk检测算法,使用 DL 检测需要在同一个补丁中一起依赖更改的 hunk ,因为 SBFL 工具返回故障的可疑候选,但不一定要一起修复。其次,我们设计了一种扩展算法,该算法采用每个检测到的固定在一起的大块,并将其扩展为在大块中包含连续的可疑语句。

在图 1 中,SBFL 工具将第 5 行返回为可疑。在 hunk detection 之后,DEAR 通过变量retrieve_password使用数据依赖性来包含第 10 行的语句以进行修复。

关键思想2. 学习和生成多块、多语句修复的组合方法:学习多块/多语句修复中的分而治之策略。要使用多个语句自动修复错误,工具

需要对语句进行更改,即语句通常可能在修复后变成语句。一种天真的方法会让模型学习代码结构的变化,并在修复前后的代码之间进行对齐。由于修复涉及多个子树转换(观察 2),在训练期间,模型可能会错误地对齐修复前后的代码,从而导致错误地学习修复。例如,如果没有这一步,模型可能会将第 10 行的retrieved_password映射到第 6 行的相同变量(正确的映射是第 11 行)。因此,为了便于学习错误修复代码转换,在训练期间,我们使用分而治之的策略。我们将一个基于 AST 的细粒度更改检测模型集成到 DEAR 中,以映射修复前后的 AST。这样的映射使 DEAR 能够了解对子树的更多本地修复更改。例如,细粒度的AST变化检测可以推导出第4-5、7行的语句变成第4、6-9行的语句;第 10 行的语句变成第 11 行的语句。我们可以将它们分成两组,并对齐各自的 AST 子树供 DEAR 学习。

固定多个子树的组合方法。我们通过增强基于树的 LSTM模型[18]的设计和编排以添加注意力层和循环训练(第 3.3 节)来支持在一个或多个 hunk 中具有多个语句的修复。

虽然该模型一次修复一个子树,但我们需要对其进行增强以一次修复多个 AST 子树。

具体来说,我们修改了它在两层中的操作,以同时考虑多个有问题的子树。例如,在训练期间,我们标记第 5 行语句的每个 AST 子树和

修复前的第 10 行是错误的。在第一层,对于错误语句的每个子树,我们将其替换为伪节点,并将新的 AST 及其(伪)节点视为错误语句的修复上下文。伪节点是通过嵌入技术计算的,以捕获错误语句的结构(第3.2 节)。在第二层,DEAR 学习从第 5 行语句的子树到固定子树的转换

第 6-9 行的语句。从第一层学习的固定上下文向量用作第二层代码转换学习中的权重。我们对每个错误语句重复相同的过程。为了修复,我们一次对所有错误语句执行修复转换的组合。

关键思想3. 使用正确的周围固定上下文进行转换学习:要学习正确的上下文来修复语句,我们需要使用其他错误语句的固定版本来训练模型(观察 3)。例如,对于训练,为了学习对第 5 行语句的修复,模型需要集成其他错误行的修复版本,即第 11 行的代码,以passwordHash作为修复上下文(而不是越野车线 10)。如果使用修复前的周围代码(即第 10 行),模型将学习错误的上下文来修复第 5 行。

2.3 方法概述2.3.1训练过程。训练输入

包括修复前后的源代码(图 2),它被解析为 AST。

Machine Translated by Google

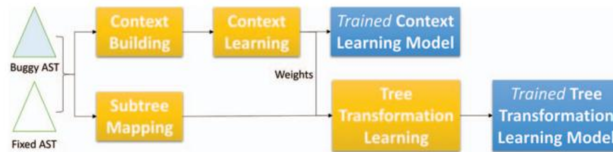

图 2:训练过程概览

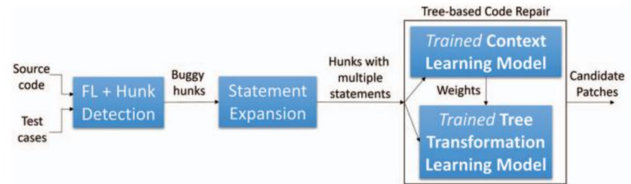

图 3:修复过程概览

输出包括用于上下文学习和树转换学习（修复)的两个训练模型。上下文学习模型（CTL)旨在学习权重（代表上下文的影响)以对树变换学习结果进行调整。树转换学习模型 (TTL) 旨在学习代码转换以修复有问题的 AST 子树。

情境学习（第 3.2-3.3 节）。第一步是构建训练前/后修复上下文。通过分而治之的策略,我们使用 CPatMiner [26]来导出更改、插入和删除的子树（关键思想 2）。结果,错误语句的 AST 子树被映射到相应的固定子树。对于每个错误的子树和各自的固定子树,我们构建了整个方法的两个 AST 作为上下文,一个在修复之前,一个在修复之后,并将它们都用于基于树的 LSTM 上下文的输入层和输出层的训练学习模型（第 3.3 节）。

为了为每个有问题的子树构建正确的上下文,我们利用了关键思想 3：我们使用其他有问题的子树的固定版本来训练我们的模型。最后,从该学习中计算出的向量用作树变换学习中的权重。

树转换学习（第 3.4 节）。我们首先使用 CPat Miner [26]来导出子树映射。为了学习错误修复树转换,每个错误子树本身及其修复后的固定子树在第二个基于树的 LSTM 的输入层和输出层用于训练。此外,表示上下文的权重在上下文学习模型中计算为向量,在此步骤中用作附加输入。

2.3.2固定过程。图 3 说明了固定过程。输入包括有缺陷的源代码和测试用例集。

故障定位和 Buggy-Hunk 检测（第 4.1 节）。从关键思想 1 开始,我们首先使用 SBFL 工具来定位具有可疑分数的错误语句。 Hunk 检测算法使用这些语句来导出需要一起修复的错误 hunk。

多语句扩展（第 4.2 节）。因为 SBFL 可能会为一个 hunk 返回一个语句,我们的目标是扩展到可能包括更多连续的错误语句。为此,我们结合了RNN [7]和数据流分析来检测更多错误语句。

基于树的代码修复（第 4.3 节）。对于从多语句扩展中检测到的错误语句,我们使用关键思想 2 来同时对多个错误子树进行修复。对于有问题的子树,我们将方法的 AST 构建为上下文,并将其用作经过训练的上下文学习模型 ,(CTL) 的输入,以产生表示上下文影响的权重。 buggy 子树用作训练树转换模型 (TTL) 的输入,以生成上下文无关的固定子树。最后,该权重用于调整到候选补丁的固定子树。

我们对当前候选代码应用语法规则和程序分析来生成固定代码。我们以与 DLFix [18] 中相同的方式使用测试用例重新排名和验证固定代码。

3 训练过程

3.1 配对错误和固定子树训练数据包含修复前后方法源代码

对。请注意,修复可能涉及多种方法。我们没有将整个错误方法与固定方法配对,而是使用分而治之的策略来帮助模型更好地学习适当上下文中的固定转换。首先,我们使用 CPatMiner 工具 [26] 来导出修复更改。

如果一个子树对应一个语句,我们称它为语句子树。根据 CPatMiner 的结果,我们使用以下规则将错误子树与相应的固定子树配对： 1. 错误子树 (-subtree) 是具有更新或删除的子树。

2. 如果一个子树被删除,我们将它与一棵空树配对。

3. 如果一个有缺陷的子树被标记为更新（即,它被更新或者它的子节点可以被插入、删除或更新）,我们将这个有缺陷的子树与它对应的固定子树配对。

4. 如果插入一个 -subtree并且它的父节点是另一个 -subtree,我们将它与那个父 -subtree 配对。如果父节点不是 -子树,我们将空树与相应插入的 -子树配对。

3.2 上下文构建图 4 说明了我们的

上下文构建过程。对于每对错误的 AST 1和固定的 AST 1 （第 3.1 节）,我们对变量执行 alpha 重命名。在第 1 步中,我们使用词嵌入模型 GloVe [29] （捕获良好的代码结构)对每个 AST 节点进行编码,将语句节点视为一个句子,将每个代码标记视为一个词。我们使用这些向量来

标记1和1中的 AST 节点。 此步骤之后的 AST 是修复前后的向量化 AST 2和2。

在步骤 2 中,我们处理2中的每一对 buggy-subtree和 2 中对应的 fixed-subtree。首先,我们使用 TreeCaps [4] 对 和 执行节点汇总,分别捕获和 的树结构到 和 中。其次,对于每个其他有问题的子树,例如 ,以及它们对应的固定子树,例如,我们处理如下。因为是我们在修复之前在生成的上下文3的构建中替换为的固定版本（关键思想 3）。也就是说,我们用它的固定版本替换每个其他有问题的子树。然而,为了在固定后构建结果上下文3 ,我们保留它,因为它是固定的子树,因此提供了正确的上下文。

生成的 AST 3 用作错误子树的修复前上下文,并在上下文学习模型 (CTL)的编码器输入层使用。生成的 AST 3 用作CTL 中解码器输出层的修复后上下文并使用（图 4）。最后,向量和将用作后面树变换学习的加权输入。

Machine Translated by Google

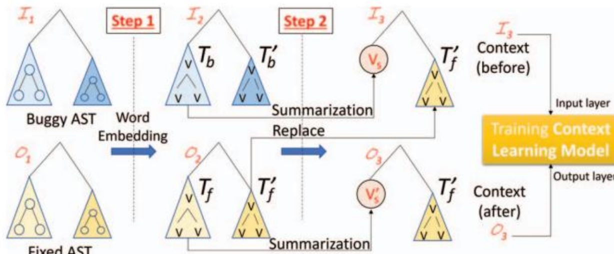

图 4:构建上下文以训练上下文学习模型

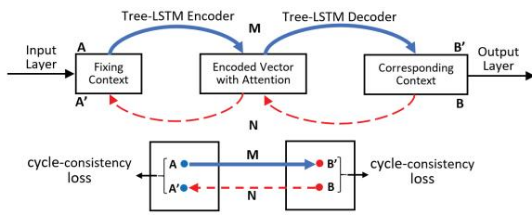

图 5:基于注意力的 Tree-based LSTM 中的循环训练

3.3通过带有注意力层和循环训练的基于树的 LSTM 进行上下文学习

对于上下文学习和树转换学习,我们通过注意层和循环训练增强了 DLFix [18]中的两层、基于树的 LSTM 模型。我们在该模型中添加了一个注意力层,该模型现在有 3 层:编码器层、解码器层和注意力层（图 5）。对于编码器和解码器,为了学习用 AST 表示的固定上下文,我们使用基于 Child-Sum Tree 的LSTM [36]。与针对每个时间步循环的常规 LSTM 不同,此模型针对每个子树进行循环以捕获结构。

我们还使用循环训练[45]来进一步改进。循环训练旨在通过持续训练和重新训练以强调输入和输出之间的映射来帮助模型更好地学习输入和输出之间的映射。这在可以通过多种方式将错误代码修复为不同的固定代码,或者可以将多个错误代码修复为一个固定代码的情况下很有用。这使得常规的基于树的 LSTM 不太准确。通过循环训练,强调输入和最可能的输出对,以减少这种一对多或多对一关系的噪声。

循环训练发生在编码器和解码器之间。我们使用正向映射：→表示过程,反向映射：→表示过程（图 5）。 →
→ ,
→ →

我们对两者应用对抗性损失并得到两个损失函数( )和, ,)。(( ))和(( ))和( )之间的差异用于生成循环一致性损失,以确保学习的映射函数是循环一致的。(,)和,)。使用激励循环一致性损失(,),整体损失函数计算如下：

,

在数学上,我们有两个损失函数 , ,
(, ,

(,) = ~ ( ) [|| (( )) − ||1] (( ))
+ ~ ( ) [|| − ||1] (1)

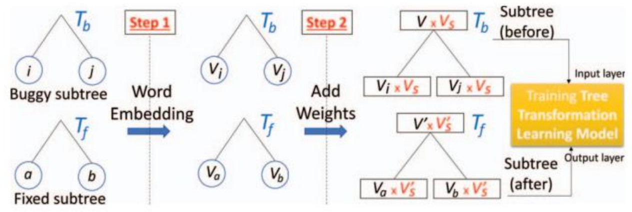

图 6:树转换学习 (, 在图 4 中)

(, , ,) = ( ) , , ,) + (, , ,)
+ (, (2)

其中( )是整个循环训练的损失函数;和 是映射到 和 的映射函数,旨在区分预测结果到； (( ))和真实结果;旨在区分预测结果(( ))和真实结果 ;是运行函数的循环一致性损失函数是激励循环一致性损失;是控制两个目标相对重要性的参数。

, ;

3.4 树变换学习图6说明了树变换学习过程。我们使用与第 3.3 节相同的具有注意层和循环训练的基于树的 LSTM 模型来学习每个错误子树的代码转换。

在第 1 步中,我们为第 3.2 节中的所有代码标记构建词嵌入。越野车子树 ( ) 和固定子树 ( ) 中的每个 AST 节点都标有其向量表示（图 6）。接下来,我们使用从图 4 中的上下文学习中计算得出的汇总向量作为权重,并分别对 buggy-subtree 中节点的每个向量和 fixed-subtree 中的每个向量执行叉积。

叉积后得到的两个子树用于基于树的 LSTM 模型的输入和输出层,用于树变换学习。我们使用叉积是因为我们的目标是将向量作为节点的标签,并将其用作表示上下文的权重,以学习用于错误修复的代码转换。

4 固定过程

4.1 Fixing-together Hunk Detection Algorithm修复多块、多语句错误的第一步是我们的FL 方法检测在同一补丁中固定在一起的错误块。为此,我们微调了谷歌的预训练 BERT模型[8],以使用 BERT 的句子对分类任务学习语句之间的固定关系。然后,我们在算法中使用微调的 BERT 模型来检测固定在一起的大块头。让我们详细解释一下我们的 hunk 检测算法。

4.1.1微调 BERT 以学习语句之间的固定关系。我们首先微调 BERT 以了解是否需要将两个语句固定在一起。让我们成为一组为错误而固定在一起的大块头。训练过程的输入是训练集中所有错误的所有集合。

步骤1. 对于一对帅哥和 in 语句和 ,我们从每个大块头中取出 , 每一对,并构建向量

Machine Translated by Google

与伯特。我们考虑在同一个补丁中固定在一起的语句对 ,) 作为（以微调 BERT。

步骤2. 对所有对和中的语句 (,) s 的所有对重复步骤 1。我们还对所有 s 重复步骤 1。我们使用它们来微调 BERT 模型,以学习所有大块对中任意两个语句之间的固定关系。

4.1.2使用 Fine-tuned BERT 进行 Hunk Detection。在获得微调的 BERT 后,我们用它来确定是否需要将代码块固定在一起。此过程的输入是微调的 BERT 模型、错误代码和测试用例。输出是需要固定在一起的大块组。该过程按以下步骤进行。 ,

步骤1. 我们使用基于频谱的 FL 工具（在我们的实验中,我们使用 Ochiai [3])在给定的源代码和测试用例上运行。它返回错误语句和可疑分数的列表。

第 2步。将 FL 工具返回的方法中的连续语句组合在一起,形成块1､2 、 ...､ .

第3 步。为了决定是否需要将一对 hunks ( ) 固定在一起,我们使用经过微调的 BERT 模型。具体来说,对于每对语句 ( ),我们使用微调的 BERT 来衡量(,) 的固定关系得分。和之间的固定在一起分数分别是和中所有语句对分数的平均值。如果所有语句的平均分数)根据需要配对高于阈值,我们考虑（被固定在一起。从检测到的帅哥对中,我们构建了固定在一起帅哥的组。帅哥组有落合可疑度得分最高的任何语句将排在第一位并固定。理由是这样的组包含最可疑的语句,因此应该首先固定。 ,

4.2 多语句扩展算法从 4.1 节的算法中检测到的 buggy hunk 可能只包含一个语句,因为这些可疑语句中的每一个最初都是由 SBFL 工具派生的,它并不专注于检测 hunk 中连续的 buggy 语句。因此,在这一步中,我们从 hunk 检测算法中获取结果,并将其扩展以在 hunk 中包含更多潜在的语句。

关键思想。我们的想法是将深度学习与数据流分析相结合。我们首先使用 GRU 单元[7] （将在第 4.2.2 节中解释)训练一个 RNN 模型来学习判断一个语句是否有错误。我们从真实的错误陈述中收集该模型的训练数据。然后我们使用数据流分析来调整结果。具体来说,如果一条语句被 RNN模型标记为 buggy,则不需要进行任何调整。然而,即使 RNN模型将给定语句确定为无错误,并且如果数据依赖于错误语句,我们仍将其标记为错误。

4.2.1扩展算法。 Multi-Statement Expansion算法的输入是buggy语句 buggyS,即hunk的种子语句。输出是一大堆有问题的连续语句。

首先,它通过包含buggyS之前和之后的语句（第 2 行的 Expand2N CandidatesList)来生成 buggy 语句的候选列表。在当前的实现中,=5。然后，

算法 1多语句扩展算法

```
1:函数MultiStatementsExpansion( 2: 3: 4: 5:
        =                              )        )
        =                    (        ()
          =                       (          ,            )
    返回
6:函数DataDepAnalysis( 7: 8: 9: 10:     ,        ,          )
        =                      (          )
                      ,        ,
                 ((     ,        ,       ((        ))))
    返回
11:函数DDExpandHunk(对于每个( 12: 13: if     ,        ,
    14: 15: 16:        ∈         ) & ( ∉ )然后∪       )) 做
                     (     ,
                     =
         否则打破
    返回
```

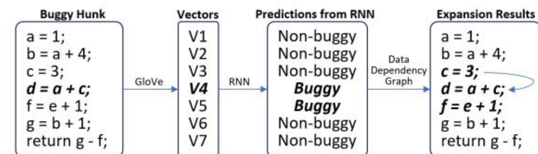

图 7:多语句扩展示例

它使用 RNN 模型作为分类器来预测候选列表中的每个语句（ buggyS 除外）是否存在错误（第 3 行的 RN NClassifier）。为了训练该 RNN 模型,我们在训练数据的 buggy hunks 中使用了 buggy语句（参见第 4.2.2 节）。

TreeCaps [4] 用于对语句进行编码。

在 DataDepAnalysis（第 4 行)中,为了调整 RNN 模型的结果,我们获得了围绕错误语句buggyS 的 buggyHunk,其中包含buggyS之前和之后的语句,这些语句被 RNN 模型预测为错误（第 7 行）。然后,我们从候选列表中的中央错误语句（第 8 行,通过 TopHalf)向上和向下（第 9 行,通过 BotHalf）逐条检查语句。在 DDExpandHunk 中,我们继续（向上或向下)扩展当前的 buggy hunk bug gyHunk 以包含被 RNN 模型视为错误或与中心 buggy 语句buggyS 具有数据依赖性的语句（第 13-14 行）。如果我们遇到一个没有 buggyS 的数据依赖性的非错误语句,或者我们用尽了列表（第 15 行）,我们将停止该过程（向上或向下） 。最后,返回包含连续 buggy 语句的buggy hunk 。

在图 7 中,SBFL 工具在第 4 行返回错误语句。所有语句都通过 GloVe [29]编码到向量集中,并由 RNN 模型分类。我们从第 4 行的语句向上扩展以包括第 3 行（即使 RNN 模型预测它没有错误） ,因为第 3 行通过变量c 与第 4 行的错误语句具有数据依赖性。我们包括第 5 行,因为 RNN 模型预测第 5 行有错误。此时,我们停止向上和向下方向,因为我们在第 2 行和第 6 行遇到与第 4 行没有数据依赖性的非错误语句。即排除第 1–2 行和第 6–7 行。最终结果包括第 3-5 行的语句作为 buggy hunk。

4.2.2使用 RNN 进行错误语句预测。我们介绍了如何使用基于 GRU 的 RNN 模型[7]来预测错误语句。

训练。为了训练 RNN 模型,我们在训练数据集中的所有块中使用有缺陷/无缺陷的语句。我们使用 GloVe [29]

Machine Translated by Google

对语句中的每个标记进行编码,以便语句由一系列标记向量表示。我们使用基于 GRU 的 RNN 模型[7]的神经架构来使用与错误/非错误标签关联的语句的 GloVe 向量。

RNN 模型以时间步长运行。在时间步长(>=1),在输入层,GRU 消耗的 GloVe 向量被标记为 1,如果它是错误的语句,否则为 0。除了时间步长 + 1 的输入,我们将时间步陈述 。在输出层, 长的输出馈送到 GRU。

预言。经过训练的基于 GRU 的 RNN 模型在第 3 行的扩展算法中用于预测 hunk 中的语句是否有错误。该模型采用 GloVe 标记向量形式的语句。它采用 hunk 中所有语句的向量,并以多时间步长的方式将它们标记为错误或非错误。

4.3 基于树的代码修复图 8 说明

了这个过程。在导出方法中的错误块后,DEAR使用经过训练的 LSTM 模型一次对所有块中的所有错误语句执行代码修复。

基于树的代码修复按以下步骤进行:

第 1 步。识别有缺陷的 S-子树。对于每个块,我们将代码解析为 AST,并识别与派生的错误语句对应的错误子树。在图 8 中,子树1和2被标识为有问题。如果一个 buggy-subtree 是另一个更大的 buggy-subtree 的一部分,我们只需要对更大的 buggy-subtree 进行修复

-subtree 因为该修复还修复了较小的 -subtree。

第 2 步。嵌入和总结。我们使用 GloVe [29]执行词嵌入,并使用 TreeCaps [4]在所有有问题的子树上执行树摘要以获得上下文。例如,在图 8 中, 1和2被汇总为两个向量1和2。

第 3 步。预测上下文。我们使用经过训练的上下文学习模型(CTL) 在具有 AST 节点的上下文上运行,其中还包括用于预测上下文的汇总节点。在生成的 AST 中,结构与输入上下文的 AST 相同,只是汇总节点变为新节点。例如,上下文中的1和2在步骤 3 之后变为 and。

第 4 步。添加权重。来自步骤 2 的权重1和2用于与越野车子树1和2 中的向量的乘积。1和2中的每个节点由节点的原始向量与权重向量之间的乘法向量表示

或者 2 .

第 5 步。预测转换。我们使用经过训练的树转换学习模型来预测子树和修复。

第 6 步。去除重量。我们从第 4 步中删除权重以获得有缺陷子树的候选固定子树。例如,我们消除 和 获得候选固定子树 和 2.

然而,因为我们知道叉积和一个向量,我们可以获得无限数量的解决方案。因此,为了产生一个单一的解决方案,我们假设 2、 对于每个节点 和 2、 和 与节点向量1垂直 和二合一 2.

然后,我们可以得到未加权的节点向量 合1 如下: _

$$= \frac{\times 1}{1\,1} \quad (3)$$

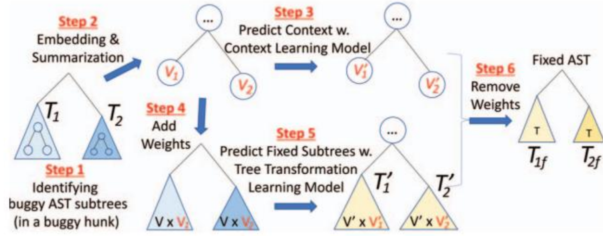

图 8:基于树的代码修复

在获得固定 S 子树中每个节点的向量后,我们基于词嵌入生成了 1, 候选补丁。对于每个节点,我们计算了它与向量列表中所有标记的每个向量之间的余弦相似度得分。要1生成候选补丁,我们选择列表中的向量 令牌。固定子树中一个节点的令牌。

有它的相似性分数通过将所有标记的所有相加,我们得到了候选人的总分。我们为每个人选出前 5 名候选人节点生成候选人并根据

4.4 后处理天真的方法在形成候选修

复时会面临组合爆炸,因为对于子树中的每个节点,我们维护前 5 个候选。但是,当我们将固定子树中所有节点的候选者组合在一起时,许多候选者对项目中的当前方法无效。因此,当我们通过组合节点的所有候选者形成候选者时,我们应用一组过滤器以与 DLFix [18] 中相同的方式验证程序语义。这使我们能够立即消除无效的候选日期。具体来说,我们使用 alpha 重命名过滤器使用包含范围内所有有效名称的字典将名称更改回正常的 Java 代码,使用语法检查过滤器来删除有语法错误的候选者,以及名称验证过滤器来检查变量、方法和类的有效性。此外,为了进一步改进,我们使用集束搜索来仅维护排名靠前的候选修复。因此,我们在形成陈述时并未穷尽所有成分。这有助于维持可管理的候选人数量。

应用所有过滤器后,我们还在候选补丁上使用了 DLFix [18] 的重新排序方案。然后我们使用测试用例对这些候选者进行补丁验证。我们从上到下验证每个补丁,直到识别出正确的补丁并且补丁验证结束。如果固定位置的所有候选人都不能通过所有测试用例,我们选择下一个位置重复该过程。

5 实证评估

5.1 研究问题为了评估 DEAR,我们寻求回答

以下问题: RQ1。在 Defects4J 基准上与基于深度学习的 APR 模型进行比较研究。与 Defects4J 上现有的基于 DL 的 APR 模型相比, DEAR 的表现如何?

RQ2。与基于深度学习的 APR 模型在大型错误数据集上的比较研究。与基于深度学习的 APR 模型相比,DEAR 在大规模错误数据集上的表现如何?

Machine Translated by Google

RQ3。基于模式的 APR 方法对Defects4J 的比较研究。与最先进的基于模式的 APR 方法相比,DEAR 的表现如何?

RQ4。 DEAR 的敏感性分析。各种因素如何影响 DEAR 在 APR 中的整体表现?
RQ5。时间复杂度和模型的训练参数。
什么是时间复杂度和训练参数的数量?

5.2 数据收集

我们对三个数据集进行了实证评估:1) Defects4J v1.2.0 [1] 有 395 个错误和测试用例; 2) BigFix [18] 在 +180 万个错误方法中有 +26k 个错误; 3) CPatMiner [26] 有来自 5,832 个 Java 项目的 +44k 个错误。

所有实验均在配备 8 核Intel CPU 和单个 GTX Titan GPU 的工作站上进行。

5.3 实验方法5.3.1 RQ1。与 Defects4J 上基于深度学习的 APR 的比较。

比较基线。我们将 DEAR 与五个最先进的基于 DL 的 APR 模型进行了比较: DLFix [18]、 CoCoNuT [23]、 SequenceR [6]、 Tufano19 [38]、 CODIT [5] 和CURE [14]。

程序和设置。我们复制了所有基于 DL 的 APR,但CURE 除外,它不可用。我们按照他们论文中的细节重新实现了 CURE 。我们针对 CPatMiner 数据集中的错误和修复训练了所有 DL 方法,并针对Defects4J 中的所有 395 个错误(两个数据集之间没有重叠)对其进行了测试。所有 DL 方法都应用相同的故障定位工具 Ochiai [3],并使用 Defects4J 中的测试用例进行补丁验证。根据之前的实验[13、18],我们为补丁生成和验证工具设置了 5 小时的运行时间限制。

我们使用beam-search 使用以下关键超参数调整 DEAR:(1) 用于 hunk 检测的 BERT:epoch 大小 (e-size) (2, 3, 4, 5),批量大小 (b-size) (8 , 16, 32, 64),和学习率 (l-rate)
(3 -4, 1 -4, 5 -5, 3 -5, 1 -5); (2) LSTM for Multi-Statement Expansion and code repair: e-size (100, 150, 200, 250), b-size (32, 64, 128, 256), l-rate (0.0001, 0.0005, 0.001) , 0.003, 0.005); (3) GloVe for representation vectors: vector size (v-size) (100, 150, 200, 250), l-rate (0.001, 0.003, 0.005, 0.01), b-size (32, 64, 128, 256)),和电子尺寸(100, 150, 200, 250)。使用其他默认参数。

DEAR 的最佳设置是 (1) e-size=4, b-size=32, l-rate=1 for BERT; −4 (2) e-size=200, l-rate=0.003, b-size=128 for LSTM; (3)对于 GloVe,v 大小=200,l-rate=0.001,b-size=64,e-size=200。对于其他模型,我们调整了他们论文中的参数,例如 word2vec 的向量长度、学习率和 epoch 大小,以找到每个数据集的最佳参数。我们在同一 CPatMiner 数据集上使用上述参数调整了所有方法以获得最佳性能。一旦我们获得了每个模型的最佳参数,我们就将它们用于以后的实验。

定量分析。我们报告的错误数量模型可以自动修复以下错误位置类型:

类型 1。 One-Hunk, One-Statement:修复了一个错误,只涉及一个语句。

类型 2。 One-Hunk,Multi-Statements:修复的错误仅涉及一个带有多个语句的 hunk。

类型 3。 Multi-Hunks, One-Statement:修复的错误涉及多个帅哥;每个大块头都有一个固定的陈述。

类型 4。 Multi-Hunks,Multi-Statements:修复的错误涉及多个帅哥;每个大块头都有多个语句。

类型 5。 Multi-Hunks,Mix-Statements:修复涉及多个 hunks 的错误,一些 hunks 有一个语句,而其他 hunks 有多个语句。

评估指标。我们使用排名靠前的候选补丁报告可以正确修复的错误数量和合理补丁的数量(即通过所有测试用例,但未通过实际修复)。

5.3.2 RQ2。与大型数据集上基于 DL 的 APR 的比较。

比较基线。我们在两个大型数据集:BigFix 和 CPatMiner 上将DEAR 与 RQ1 中的相同基线进行比较。

程序和设置。首先,我们在 BigFix 和 CPatMiner 上评估了所有基于 DL 的 APR 模型。在 DLFix 和 Sequencer 之后,我们将数据随机分成 80%/10%/10% 用于训练、调优和测试。其次,我们有跨数据集评估:在 CPatMiner 上训练基于 DL 的方法并在 BigFix 上测试,反之亦然。与 Defects4J 不同,BigFix 和 CPatMiner 数据集没有测试用例。

没有测试用例,我们就无法对所有 DL 方法使用故障定位和补丁验证。因此,我们将实际的 bug 位置输入到 DL 模型中,包括 buggy hunk 和语句上的位置。基于 DL 的基线不区分块,而是一次处理每个错误语句。我们使用开发人员的实际修复作为基本事实来评估基于 DL 的方法。

评估指标。我们使用 top 度量,定义为正确补丁在排名靠前的候选列表中的次数与错误总数之比。

5.3.3 RQ3。与 Defects4J 上基于模式的 APR 的比较。

比较基线。我们将 DEAR 与 Defects4J 上最先进的基于模式的 APR 工具进行比较: Elixir [33]、 ssFix [43]、 CapGen [40]、 FixMiner [16]、 Avatar [19]、 Hercules [34]、 Sim Fix [13]和Tbar [20]。我们能够在相同的计算环境下复制以下基于模式的基线: Elixir、ssFix、FixMiner、SimFix、TBar 。我们将工具的时间限制设置为 5 小时。对于其他基线,由于代码不可用,我们使用他们论文中报告的结果,因为它们在同一数据集上运行。我们使用相同的设置和评估指标。

5.3.4 RQ4。敏感性分析。我们评估了不同因素对 DEAR 绩效的影响。我们考虑以下几点: (1)大块头检测(Hunk); (2) 多语句扩展(Expansion); (3) 多语句树模型和循环训练; (4)数据拆分方案。我们对每个因素都使用留一法策略。

我们在 Defects4J 上评估前三个因素,在CPatMiner上评估最后一个因素,因为我们需要更大的数据集来进行各种拆分。

5.3.5模型训练的时间复杂度和参数个数。我们测量模型的训练和修复时间及其在数据集上进行模型训练的参数数量。

6 实证结果

6.1 问题 1。与基于深度学习的 APR 模型在 Defects4J 上的比较结果

6.1.1故障定位。我们首先评估 APR 模型

当与故障定位工具 Ochiai [3] 一起使用时。表 1 和表 2 显示了 DEAR 和基线模型之间的比较结果。

Machine Translated by Google

表 1:RQ1。与基于深度学习的 APR 模型在具有故障定位的 Defects4J上的比较

项目	图表关闭Lang Math Mockito时间总计6/9 15/19 3/5 6/12 6/8 14/18						
音序器	3/3	4/5	2/2	30/55	0/0	0/0	
它被编码	1/2	2/5	0/0		0/0	0/0	
龙卷风19	3/4	3/5	1/1		0/0	0/0	
DL修复	5/12 6/10 5/12 12/18				1/1	1/2	
椰子 6/11		6/9	5/13 13/21 6/13		2/2	1/1	33/57
治愈	6/10 5/14 16/23				2/2	1/2	36/71
X/Y:分别	8/16	7/11	8/15 20/33		1/2	3/6 47/91亲爱的	

是正确和似是而非的补丁的数量。

表 2:RQ1。在具有故障定位的 Defects4J 上与基于深度学习的 APR 模型进行详细比较

Bug 类型类型	DLFix CoCoNUT CURE 亲爱的			
1. One-Hunk One-Stmt Type 2. One-	30	33	36	29
Hunk Multi-Stmts Type 3. Multi-Hunks	0	0	0	4个
One-Stmt Type 4. Multi-Hunks Multi-	0	0	0	11
Stmts Type 5. Multi-Hunks Mix-Stmts Total	0	0	0	1
	0	0	0	2个
	30	33	36	47

如表 1 所示,DEAR 可以自动修复最多数量的错误(47) 并生成最多数量的合理补丁 (91),这些补丁通过了 Defects4J 上的所有测试用例。特别是,DEAR 可以自动修复 32、41、33、17、14和 11 个错误,分别比 Sequencer、CODIT、Tufano19、DLFix、 CoCoNuT 和 CURE 多（即 213%、683%、236%、57%） 、42% 和 31% 的相对改进）。与 Defects4J 上的那些工具相比,DEAR 可以自动修复这些工具分别遗漏的 35、34、41、18、31 和 18 个错误。通过DEAR 的结果与基线组合的结果之间的重叠分析， DEAR 可以修复他们遗漏的 18 个独特的错误。

表 2 显示了 DEAR 和基于深度学习的顶级基线（DLFix、CoCoNuT、CURE)与不同错误类型之间的比较。

对于单块错误（类型 1-2） ,DEAR 修复了 33 个错误,包括其他工具遗漏的 4 个独特的单一大块错误。

对于多块错误（类型 3-5） ,DEAR 可以修复DLFix、CoCoNuT 和 CURE 无法修复的14 个错误。现有的基于 DL 的APR 模型无法修复这些错误,因为一次修复一个语句的机制不适用于需要同时修复多个语句的相关更改的错误。因此,他们不会为这些情况生成正确的补丁。

对于multi-hunk 或 multi-statement错误（类型 2-5） ,亲爱的修复其中 18 个（在 47 个修复错误中,即修复错误总数的 38.3%）。

6.1.2无故障定位。我们还在不受第三方 FL 工具影响的情况下,将 DEAR 与其他工具的修复能力进行了比较。所比较的所有工具（表 3)都指向正确的修复位置并执行修复。如图所示,如果已知修复位置,DEAR 的修复能力也高于那些基线（53 个错误对 44、40 和 48） 。

重要的是,它可以修复 20 个 multi-hunk/multi-statement 错误（占总共 53 个已修复错误的37.7% ） ,而 CoCoNuT、DLFix 和 CURE 只能修复 7、5 和 10 个此类错误。

DEAR 比现有的基于 DL 的模型更通用,因为它可以支持具有 multi-hunk 或 multi-statements 的依赖修复。

重要的是,它显着改进了这些基于 DL 的模型和

表 3:RQ1。与没有故障定位（即正确定位)的 Defects4J上基于深度学习的 APR 模型的比较

Bug 类型类型	DLFix CoCoNUT CURE 亲爱的			
1. One-Hunk One-Stmt Type 2. One-	35	37	38	33
Hunk Multi-Stmts Type 3. Multi-Hunks	1个	3个	3个	4个
One-Stmt Type 4. Multi-Hunks Multi-	4个	3	6	13
Stmts Type 5. Multi-Hunks Mix-Stmts Total	0	0	0	1
	0	1个	1个	2个
	40	44	48岁	53

表 4:RQ2。与大型数据集上的 DL APR 比较

	CPatMiner			BigFix		
	（4,415 个测试错误）（2,594 个测试错误）					
工具/数据集 Top-1 Top-3 Top-5 Top-1 Top-3 Top-5 Sequencer 7.8% 10.8% CODIT 4.5%						
9.1% Tufano19		8.9%	10.3%	8.5%	9.1%	
		7.4%	9.2%	3.9%	6.3%	
	8.6%	9.3%	11.2%	7.7%	8.8%	9.6%
DL修复	11.4%	12.3% 13.1%		11.2%	11.9% 12.5%	
椰子	13.5%	14.7% 15.3%		12.2%	13.6% 14.3%	
治愈	14.2%	15.1% 15.5%		12.9%	14.2% 14.1%	
亲爱的	15.1% 15.6% 16.8% 14.1% 15.4% 16.3%					

表 5:RQ2。与跨数据集上的 DL APR 比较

工具/数据集	CPatMiner(Train)/BigFix BigFix(Train)/CPatMiner Top-1 Top-3 Top-5					
	Top-1 Top-3 Top-5 5.4% 5.8% 6.1% 7.2%					
音序器			6.2%	5.3%		
它被编码	2.5%	4.0%	4.4%	3.2%	5.2%	6.4%
龙卷风19	4.5%	5.4%	5.7%	5.9%	6.3%	7.6%
DL修复	6.3%	6.9%	7.3%	8.2%	8.7%	9.2%
椰子	6.7%	7.4%	8.1%	8.3%	9.6%	10.7%
治愈	7.1%	7.7%	8.2%	8.7%	9.9%	10.9%
亲爱的	7.5%	8.1% 8.6%		9.6% 10.2% 11.3%		

将 DL 方向提升到与其他 APR 方向（基于搜索和基于模式)相同的级别,可以处理多语句错误。此外,DEAR 是完全数据驱动的,不需要像基于模式的 APR 模型那样定义固定模式。

6.2 RQ2。在大型数据集上与基于 DL 的 APR 模型的比较结果表 4 显示,DEAR 可以修复比两个大型数据集上任何基于 DL 的 APR 基线更多的错误。使用 top-1 补丁,DEAR可以修复 CPatMiner 中 4,415 个错误中的 15.1%。它修复了比带有 top-1 补丁的基线多40-322个错误。在 BigFix 上,它可以使用 top-1 补丁修复 2,594 个错误中的 14.1%。它可以比使用 top-1 补丁的基线多修复31-145 个错误。

表 5 显示 DEAR 在交叉数据集设置中也优于基线,在该设置中我们在 CPatMiner 上训练模型并在 BigFix 上测试它们,反之亦然。

表 6 显示了 CPatMiner 与不同错误类型的详细比较结果。如图所示，DEAR 可以在两个大型数据集上的每种类型的错误位置上自动修复更多错误。在 667 个修复的错误中,DEAR 修复了 169 个类型 2-5 的 multi-hunk 或 multi-stmt 错误（即,占修复错误总数的 25.33%）。 DEAR 修复了比基线 CoCoNuT、DLFix 和 CURE更多的错误（71、164 和 41 个） ,并且修复了每种错误类型中更多的错误。

Machine Translated by Google

表 6:RQ2。详细分析。在 CPatMiner 数据集上与基于 DL 的 APR 模型进行 Top-1 结果比较

类型(#bugs)	椰子	治愈	DL修复	亲爱的
	#固定的	#固定的	#固定的	#固定的
类型 1 (1,668)	28.7% (479)	29.8%(497)	23.7% (395)	29.9% (499)
2 型 (530)	1.3% (7)	2.1% (11)	0.6% (3)	4.2% (22)
3 型 (879)	12.4% (109)	13.2% (116)	11.8% (104)	13.7% (120)
类型 4 (1,089)	0% (0)	0% (0)	0% (0)	2.0% (22)
5 型 (249)	0.4% (1)	0.8% (2)	0.4% (1)	2.0% (5)
总计 (4,415)	13.5% (596)	14.2% (626)	11.4% (503)	15.1% (667)

表 7:RQ3。与基于模式的 APR 模型的比较

项目 ssFix	图表关闭Lang Math Mockito时间总计 20/60 21/25 25/31					
3/7	2/11	5/12 10/26		0/0	0/4	26/41
CapGen 4/4	0/0	5/5	12/16	0/0	0/0	27/53
修复矿工 5/8	5/5	2/3	12/14	0/0	1/1	34/56
长生不老药 4/7 8/12 12/19	0/0			0/0	2/3	43/81
阿凡达 5/12 5/11 6/13	8/12			2/2	1/3	
模拟修复 9/13 14/268	6/8			0/0	1/1	
Tbar 5/13 19/36 9/14	8/12			1/2	1/3	
赫拉克勒斯 8/10 8/13 10/15 20/29 0/0 49/72					3/5	
亲爱的8/16 7/11 8/15 20/41 1/2 47/91					3/6	

X/Y:是正确且合理的补丁的数量;数据集:Defects4J

DEAR 比 CoCoNuT、DLFix 和 CURE 多修复了 52、61 和 40 个 multi-hunk/multi-stmt 错误,以及 20、104 和 2 个 one-hunk/one-stmt 错误。对于其他工具修复的多语句错误(类型 2 和 5) ,修复的语句是独立的。此结果表明一次修复每个单独的语句是行不通的。

6.3 问题 3。与基于模式的 APR 模型的比较结果如表 7 所示,DEAR 在错误数量方面与顶级基于模式的工具 Hercules 和 Tbar处于同一水平。

表 8 显示了不同错误类型的比较细节。如图所示,DEAR 修复了 Hercules 遗漏的 7 个多/混合语句错误(类型2、4-5)。进一步调查,我们发现Hercules 旨在修复复制的错误,即 hunks 必须有类似的语句。这 7 个 bug 是不可复制的,即buggy hunks 有不同的 buggy statements 或者 buggy hunk 有多个不相似的 buggy statements。对于类型 1 和类型 3,DEAR修复的单语句错误比 Hercules 少 9 个,因为它的修复不正确。 DEAR 总共修复了 Hercules 遗漏的 12 个错误:图表7、16、20、24;时间-7;关闭 - 6,10,40;朗10;数学 41,50,91。

与 Tbar 相比,DEAR 修复了 15 个多的 multi-hunk/multi-stmt错误。 Tbar 的设计目的不是像 DEAR 那样一次修复多语句。相反,它一次修复一个语句,因此,当这 15 个错误需要对多个语句进行依赖修复时,它就无法正常工作。Tbar 可以修复的 3 个类型 2 错误是对单个语句的修复是独立的。同样的原因也适用于 SimFix。 Tbar 修复了 11 个更正确的 one-hunk/one-statement 错误。

简而言之,我们将基于 DL 的模型 DEAR 提高到可比的以及与那些基于模式的 APR 模型的互补水平。

6.4 RQ4。敏感性分析6.4.1 Fixing-together Hunk 检测的影响。如表 9 所示,在没有 hunk 检测的情况下,DEAR 可以自动修复 35 个错误。通过 hunk 检测,DEAR 可以修复 14 个以上的 multi-hunk 错误(类型 3-5)。它

表 8:RQ3。与基于模式的 APR 的详细比较

Bug 类型类	SimFix Tbar Hercules 亲爱的			
型 1. One-Hunk One-Stmt	30	40	34	29
Type 2. One-Hunk Multi-Stmts	1	3	0	4
Type 3. Multi-Hunks One-Stmt	3个	0	15	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	
Type 5. Multi-Hunks Mix-Stmts	0	0	0	1 2
Total	34	43	49	47

表 9:RQ4。缺陷敏感性分析4J

变体没有	没有	没有		亲爱的
	大块头	扩展 30	注意周期	
1型26	31			29
2 型 40 2				4
Type-3	0	13	9	11
类型 4 00 1				
5型00 2				1 2
总计 40	35	43		47

由于不正确的 hunk 检测,修复了两个较少的 Type-1 错误。简而言之,块检测很有用,因为多块/多语句错误需要同时对多个块进行依赖修复。

6.4.2 多语句扩展的影响。如表 9 所示,在没有扩展的情况下,DEAR 修复了 Defects4J 中的 43 个错误。通过扩展,它修复了类型 2、4、5 中的 7 个多 stmt 错误,同时修复了 2 个 Type-3 错误和 1 个 Type-1 错误。在这两种类型中修复较少错误的原因是多语句扩展可能通过将单语句错误视为多语句错误来错误地扩展错误块。即便如此,DEAR 仍然可以修复更多的错误,可见多语句展开的用处。

为了比较 Hunk Detection 和 Multi-Statement Expansion 的影响,让我们注意到没有 Hunk Detection的 DEAR 变体错过了所有 14 个 multi-hunk 错误(类型 3、4、5)。没有扩展的变体错过了所有 7 个多语句错误(类型 2、4、5)。
但是,让我们考虑一下修复它们的难度有多大。在使用 Hunk-Detection 修复的 14 个多块错误中,有 11 个错误属于 Type-3(多块/单语句) ,其中一些方法(例如, Hercules)可以通过一次修复一个语句来处理。只有 3 个错误属于类型 4-5。相比之下,Expansion 修复的所有 7 个错误都是多语句错误(类型 2、4、5) ,现有的基于 DL 的 APR 方法无法修复这些错误。因此,Expansion 有助于处理比 Hunk-Detection 更具挑战性的错误。

6.4.3 具有注意力和循环训练的基于树的 LSTM 模型的影响。 (注意力周期)为了衡量注意力和周期训练的影响,我们从 DEAR 中删除了这两种机制以生成基线。我们的结果表明,在 Defects4J 中,DEAR在所有错误类型上比基线多修复了 7 个错误(增加 17.5%)。

该结果表明这两种机制的有用性。

6.4.4 训练数据大小的影响。表 10 显示训练数据的大小对 DEAR 的性能有影响。如表 10 所示,训练数据越多,DEAR 的准确性越高。

这是预期的,因为 DEAR 是一种数据驱动的方法。但即使训练数据较少(70%/30%) ,DEAR 也达到了 11.7% 的 top-1 结果,仍然高于 DLFix(11.4% in top-1)和 Sequencer(7.7% in top-1) ;两者都有更多的训练数据(90%/10% 分裂) 。

表 10:训练数据大小的影响

CPatMiner 数据集的拆分方案 90%/10% 80%/20% 70%/30% 13.8%			
Top-1 的错误总数百分比	15.1%		11.7%

```
1 public void excludeRoot( String path) { 2 —
String url = toUrl(path) ; 3 -
findOrCreateContentRoot(url) 。 addExcludeFolder( 网址 ) ; 4 +
网址 url = toUrl(路径); 5 +
findOrCreateContentRoot(url).addExcludeFolder(url.getUrl()); 6 } 7

public void useModuleOutput(String production, String test )
        { modifiableRootModel.inheritCompilerOutputPath( false ) ;
9 - modifiableRootModel.setCompilerOutputPath(toUrl(production)); 10 -
modifiableRootModel.setCompilerOutputPathForTests( toUrl( test ) ) ; 11 +
modifiableRootModel.setCompilerOutputPath(toUrl(production).getUrl()); 12 +
modifiableRootModel.setCompilerOutputPathForTests(toUrl(test).getUrl()); 13 }
```

图 9:CPatMiner 中的多块/多语句修复

6.5 RQ5。时间复杂度和参数DEAR 在 CPatMiner 上的训练时间为

+22 小时,在 CPatMiner 上预测每个候选补丁需要 2.4-3.1 秒。在 BigFix 上训练DEAR 花费了 18-19 小时,在 BigFix 上预测每个候选人花费了 3.6-4.2 秒。由于数据集小得多,对候选人的Defects4J 预测只用了 2.1 秒。

每个测试用例的测试执行时间为 +1 秒。对所有错误修复的测试用例进行测试验证需要 2-20 分钟。

最佳基线 CURE [14]比 DEAR（RQ1和 RQ2)修复的错误更少,并且在 CPatMiner 和 BigFix 上分别需要比 DEAR 多 7 倍和 7.3 倍的训练参数。具体来说, DEAR 和 CURE 在CPatMiner 上需要 0.39M 和 3.1M 训练参数,在 BigFix 上需要 0.42M 和 3.5M 参数。因此,DEAR比 CURE 更简单,同时取得了更好的结果。

有效性的威胁。我们测试了 Java 代码。 DEAR 中的关键模块是独立于语言的,除了第三方 FL 和带有程序分析的后处理。基于模式的 APR工具需要带有测试用例的数据集,因此,我们仅在 Defects4J 上比较了它们。我们尽力重新实现基于模式的APR 基线和 CURE 以进行公平比较。

说明性示例。图 9 显示了来自 DEAR 的正确修复。它正确地检测到两个越野车大块头;每个都有多个语句。
DEAR 利用同一方法中存在的变量名称（第 8 行的 modifiableRootModel）编写第 11-12 行的固定代码。基于 DL 的基线,Sequencer [6]和 CoCoNuT [23],将代码视为序列,并且不能很好地得出此修复程序的结构更改。 DLFix 一次修复一个语句,因此不起作用（第 2 行和第 3 行的修复相互依赖）。对于基于模式的 APR [13,34],没有针对此错误的修复模板。

限制。亲爱的有以下限制。首先,与 ML方法一样,修复罕见或词汇外的名称具有挑战性。有了更多的训练数据,DEAR 就有更高的机会遇到生成新名称的成分。其次,我们只关注导致测试失败的错误。安全性、漏洞和非失败测试错误仍然是它的局限性。第三,我们不能通过添加几个新语句或任意大的依赖固定语句来生成修复。四、扩容算法会产生不正确的块以进行修复,从而导致在正确的语句中进行修复。最后,我们目前专注于 Java,然而, DEAR 中使用的基本表示,例如令牌、AST、依赖项,对任何程序语言都是通用的。只有第三方 FL 和带有语义检查器的后处理是语言相关的。

7 相关工作

基于深度学习的 APR 方法。 DeepRepair [42]学习代码相似性以从与错误代码相似的代码片段中选择修复成分。 DeepFix [11]学习语法规则来修复语法错误。棘轮[12],图法诺等人。 [39]和 SequenceR [6]主要使用神经网络机器翻译 (NMT) 和基于注意力的编码器-解码器和代码抽象来生成补丁。 CODIT [5]对代码结构进行编码,学习代码编辑,并采用 NMT 模型来提出修复建议。图法诺等人。 [38]使用带有代码抽象和关键字替换的序列到序列 NMT 来学习代码更改。 DLFix [18]有一个基于树的翻译模型来学习修复。 CoCoNuT [23]开发了一个上下文感知的NMT 模型。 CURE [14]提出了一种使用 GPT模型[31]的代码感知 NMT 。现有的基于 DL 的 APR 模型一次修复单个语句并且对 multi-hunk/multi-stmt 错误无效。

基于模式的 APR 方法。这些方法从先前的修复[15、17、19、27]中自动或半自动地挖掘和学习修复模式[17、19、20、27] 。 Prophet [22]从一组成功的人类补丁中学习代码正确性模型。

Droix [37]使用基于搜索的修复来学习崩溃的常见根本原因。 Genesis [21]自动从用户提交的补丁中推断出补丁生成。 HDRepair [17]用图形挖掘修复模式。 ELIXIR [33]使用来自 PAR 的模板和局部变量、字段或常量来构建固定表达式。 CapGen [40]、 SimFix [13]、 FixMiner [16]依赖于从现有补丁中提取的频繁代码更改。 Avatar [19]利用静态分析违规的修复模式。 Tbar [20]是一个基于模板的 APR 工具,具有收集的修复模式。 Angelix [25]捕获程序语义来修复方法。 ARJA [44]生成较低粒度的补丁表示,从而实现高效搜索。我们没有与Angelix 进行比较,因为我们与性能优于 Angelix 的 CapGen 进行了比较。

我们无法重现 ARJA,但是,ARJA 仅修复了 18 个错误,而 DEAR 在 Defects4J 中的相同四个项目中修复了 42 个错误。

8 结论

在这项工作中,我们做出了三个关键贡献:1)一种新颖的 FL 技术,用于将传统 SBFL 与深度学习和数据流分析相结合的多块、多语句修复； 2) 使用分而治之策略生成多块、多语句修复的组合方法； 3) 增强和编排具有注意力层和循环训练的两层 LSTM 模型。

在 Defects4J 上,DEAR在修复错误的数量方面优于基于 DL 的 APR 基线 42%–683%。在 BigFix 上,它使用 top-1 补丁修复了 31-145 个以上的错误。在 CPatMiner 上,它修复了比基线多 40-52 个 multi-hunk/multi-stmt 错误。

致谢

这项工作部分得到美国国家科学基金会(NSF) 赠款 CNS-2120386、CCF-1723215、CCF-1723432、TWC 1723198、CCF-1518897 和 CNS-1513263 的支持。

Machine Translated by Google

参考

[1] 2019.Defects4J 数据集。 https://github.com/rjust/defects4j [2] 2021.
亲爱的:一种基于深度学习的新型自动程序修复方法。 https://github.com/
AutomatedProgramRepair-2021/dear-auto-fix [3] Rui Abreu、Peter Zoeteweij
和 Arjan Jc Van Gemund。 2006.软件故障定位的相似系数评估。在第 12 届环太平洋国际可靠计算研讨会 (PRDC) 的会议记录中。 39–46。 https://doi.org/10.1109/PRDC.2006.18 [4] Nghi DQ Bui、Yijun Yu 和 Lingxiao Jiang。 2021. TreeCaps:用于源代码处理的基于树的胶囊网络。 AAAI 人工智能会议论文集 35, 1
（2021 年 5 月）,30-38。 https://ojs.aaai.org/index.php/AAAI/article/view/16074 _

[5] Saikat Chakraborty、Yangruibo Ding、Miltiadis Allamanis 和 Baishakhi Ray。 2020. CODIT:使用基于树的神经模型进行代码编辑。 IEEE软件工程汇刊 (2020)。 https://doi.org/10.1109/TSE.2020.3020502 [6] Zimin Chen、Steve James Kommrusch、
Michele Tufano、Louis-Noël Pouchet、 Denys Poshyvanyk 和 Martin Monperrus。 2019. SEQUENCER:用于端到端程序修复的序列到序列学习。 IEEE 软件工程汇刊 (2019)。 https://doi.org/10.1109/TSE.2019.2940179 [7] Kyunghyun Cho、Bart van Merriënboer、Caglar Gulcehre、Dzmitry Bahdanau、Fethi Bougares、Holger
Schwenk 和 Yoshua Bengio。 2014.使用 RNN 编码器-解码器学习短语表示以进行统计机器翻译。

在 2014 年自然语言处理经验方法会议 (EMNLP)会议记录中。计算语言学协会,多哈,卡塔尔, 1724-1734 年。 https://doi.org/10.3115/v1/D14-1179 [8] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。 2019.
BERT:用于语言理解的深度双向转换器的预训练。在计算语言学协会北美分会 2019 年会议记录中:人类语言技术,第 1 卷（长文和短文）。计算语言学协会,明尼苏达州明尼阿波利斯, 4171–4186。 https://doi.org/10.18653/v1/N19-1423 [9] Claire Le Goues、Michael Dewey-Vogt、Stephanie Forrest 和 Westley Weimer。

2012. 自动程序修复的系统研究:以每个 8 美元的价格修复 105 个错误中的 55 个。在第 34 届国际软件工程会议 (ICSE 12) 的会议记录中。 3–13。 https://doi.org/10.1109/ICSE.2012.6227211 [10] Claire Le Goues、ThanhVu Nguyen、Stephanie Forrest
和 Westley Weimer。 2012.
GenProg:自动软件修复的通用方法。 IEEE 软件工程汇刊 38, 1（2012 年 1 月）,54–72。 https://doi.org/10.1109/TSE.2011.104 [11] Rahul Gupta、Soham Pal、Aditya Kanade
和 Shirish Shevade。 2017. DeepFix:通过深度学习修复常见的 C 语言错误。在第 30 届 AAAI 人工智能会议（美国加利福尼亚州旧金山)(AAAI 17) 的会议记录中。 AAAI 出版社,1345–1351。

[12] Hideaki Hata、Emad Shihab 和 Graham Neubig。 2018. 学习使用神经机器翻译生成正确的补丁。 arXiv 预印本 arXiv:1812.07170 (2018)。

[13] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. 使用现有补丁和类似代码塑造程序修复空间。
在第 27 届 ACM SIGSOFT 软件测试和分析国际研讨会（荷兰阿姆斯特丹）(ISSTA 2018)的会议记录中。美国纽约州计算机协会,298–309。 https://doi.org/10.1145/ 3213846.3213871

[14]姜南,Thibaud Lutellier,谭林. 2021. CURE:用于自动程序修复的代码感知神经机器翻译。在第 43 届国际软件工程会议 (ICSE 21)的会议记录中。 1161–1173。 https://doi.org/10.1109/ICSE43902.2021.00107 _

[15] Dongsun Kim、Jaechang Nam、Jaewoo Song 和 Sunghun Kim。 2013.从人工编写的补丁中学习自动补丁生成。在第 35 届国际软件工程会议 (ICSE 13)的会议记录中。 802–811。 https://doi.org/10.1109/ICSE.2013.6606626 [16] Anil Koyuncu、Kui Liu、Tegawendé F Bissyandé、Dongsun Kim、Jacques
Klein、Martin Monperrus 和 Yves Le Traon。 2020. Fixminer:挖掘相关修复模式以进行自动程序修复。实证软件工程 25 (2020), 1980–2024。 https://doi.org/10.1007/s10664-019-09780-z

[17] Xuan Bach D. Le、David Lo 和 Claire Le Goues。 2016. 历史驱动程序修复。在第 23 届 IEEE 软件分析、进化和再工程国际会议 (SANER 16) 的论文集中,卷。 1. 213–224。 https://doi.org/10.1109/SANER.2016.76 [18] Yi Li、Shaohua Wang 和 Tien N. Nguyen。 2020. DLFix:用于自动程序修复的基于上下文的代码
转换学习。在ACM/IEEE 第 42 届软件工程国际会议（韩国首尔)(ICSE 20)的会议记录中。美国纽约州计算机协会, 602–614。 https://doi.org/10.1145/3377811.3380345 [19] Kui Liu、Anil Koyuncu、Dongsun Kim 和 Tegawendé F. Bissyandè。 2019.

AVATAR:使用静态分析违规修复模式修复语义错误。
在第 26 届 IEEE 软件分析、演化和再工程国际会议论文集 (SANER 19) 中。 1-12。 https://doi.org/10.1109/SANER。 2019.8667970

[20] Kui Liu、Anil Koyuncu、Dongsun Kim 和 Tegawendé F. Bissyandé。 2019. TBar:重新审视基于模板的自动化程序修复。在第 28 届 ACM SIGSOFT 软件测试与分析国际研讨会（中国北京)(ISSTA 19)的会议记录中。美国纽约州计算机协会, 31-42。 https://doi.org/10.1145/3293882.3330577 [21] Fan Long、Peter Amidon 和 Martin Rinard。 2017.用于补丁生成的代码转换的自动推理。在第 11 届软件工程基
础联席会议（德国帕德博恩)(ESEC/FSE 17) 的记录中。

美国纽约州计算机协会,727–739。 https://doi.org/10.1145/3106237.3106253 [22]范龙和马丁里纳德。 2016. 通过学习正确代码
自动生成补丁。在第 43 届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会（美国佛罗里达州圣彼得堡)的会议记录中

（POPL 16）。美国纽约州计算机协会,298–312。 https://doi.org/10.1145/2837614.2837617
[23] Thibaud Lutellier、Hung Viet Pham、Lawrence Pang、Yitong Li、Moshi Wei 和Lin Tan。 2020. CoCoNuT:使用 Ensemble 结合上下文感知神经翻译模型进行程序修复。在第 29 届 ACM SIGSOFT软件测试与分析国际研讨会论文集（虚拟活动,美国）

（ISSTA 20）。美国纽约州计算机协会,101–114。 https://doi.org/10.1145/3395363.3397369

[24]马蒂亚斯·马丁内斯和马丁·蒙佩鲁斯。 2016. ASTOR:Java 程序修复库（演示）。在第 25 届软件测试与分析国际研讨会（德国萨尔布吕肯)(ISSTA 16) 的会议记录中。美国纽约州计算机协会,441–444。 https://doi.org/10.1145/2931037.2948705 _

[25] Sergey Mechtaev、Jooyong Yi 和 Abhik Roychoudhury。 2016. Angelix:通过符号分析的可扩展多行程序补丁合成。在第 38 届国际软件工程会议（德克萨斯州奥斯汀）(ICSE 16) 的会议记录中。
美国纽约州纽约市计算机协会,691–701。 https://doi.org/10.1145/2884781.2884807 [26] Hoan Anh Nguyen、Tien N. Nguyen、
Danny Dig、Son Nguyen、Hieu Tran 和Michael Hilton。 2019. 基于图的野外细粒度语义代码更改模式挖掘。在第 41 届国际软件工程会议 (ICSE 19)的会议记录中。 IEEE 出版社,819–830。 https://doi.org/10.1109/ICSE.2019.00089 _

[27] Hoang Duong Thien Nguyen、Dawei Qi、Abhik Roychoudhury 和 Satish Chan dra。 2013. SemFix:通过语义分析进行程序修复。在第 35 届国际软件工程会议 (ICSE 13) 的会议记录中。 772–781。 https://doi.org/10.1109/ICSE.2013.6606623

[28] Tung Thanh Nguyen、Hoan Anh Nguyen、Nam H. Pham、Jafar Al-Kofahi 和Tien N. Nguyen。 2010. 面向对象程序中的重复错误修复。在第 32 届 ACM/IEEE 软件工程国际会议论文集中 -第 1 卷（南非开普敦)(ICSE 10)。计算机机械协会,美国纽约州纽约市,315–324。 https://doi.org/10.1145/1806799.1806847 [29] Jeffrey Pennington、Richard Socher 和 Christopher D. Manning。 2014. GloVe:用于词表示的全局向量。在自然语言处理
(EMNLP) 的经验方法中。 1532–1543。 http://www.aclweb.org/anthology/D14-1162 _

[30] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. 自动程序修复随机搜索的强度。在第 36 届国际软件工程会议论文集（印度海得拉巴）
（ICSE 14）。美国纽约州计算机协会,254–265。 https://doi.org/10.1145/2568225.2568254

[31] 亚历克·拉德福德、卡尔西克·纳拉西姆汉、蒂姆·萨利曼斯和伊利亚·萨茨克维尔。 2018. 我通过生成预训练证明语言理解。 (2018)。
[32] Baishakhi Ray 和 Miryung Kim。 2012. 分叉项目中跨系统移植的案例研究。在 ACM SIGSOFT 第 20 届软件工程基础国际研讨会（北卡罗来纳州卡里)(FSE 12) 的会议记录中。

美国纽约州纽约市计算机协会,第 53 条,11 页。 https://doi.org/10.1145/2393596.2393659

[33] Ripon K Saha、Yingjun Lyu、Hiroaki Yoshida 和 Mukul R Prasad。 2017. Elixir:有效的面向对象程序修复。在第 32 届 IEEE/ACM国际自动化软件工程会议 (ASE 17) 的会议记录中。 648–659。 https://doi.org/10.1109/ASE.2017.8115675

[34] Seemanta Saha、Ripon K. Saha 和 Mukul R. Prasad。 2019. 利用进化进行多块程序修复。在第 41 届国际软件工程会议 (ICSE 19) 的会议记录中。 IEEE 出版社,13–24。 https://doi.org/10.1109/ICSE.2019.00020 _

[35] Abigail See、Peter J Liu 和 Christopher D Manning。 2017. 进入正题:使用指针生成器网络进行总结。 arXiv 预印本 arXiv:1704.04368 (2017)。

[36] Kai Sheng Tai、Richard Socher 和 Christopher D. Manning。 2015.树结构长短期记忆网络作品的改进语义表示。在计算语言学协会第 53 届年会和第 7 届自然语言处理国际联合会议论文集（第 1 卷:长篇论文)中。中国北京计算语言学协会,1556-1566 年。 https://doi.org/10.3115/v1/P15-1150

Machine Translated by Google

[37] Shin Hwei Tan、Zhen Dong、Xiang Gao 和 Abhik Roychoudhury。 2018. 修复Android 应用程序中的崩溃。在第 40 届国际软件工程会议（瑞典哥德堡)(ICSE 18) 的会议记录中。计算机机械协会,美国纽约州纽约市,187–198。 https://doi.org/10.1145/3180155。 3180243

[38] Michele Tufano、Jevgenija Pantiuchina、Cody Watson、Gabriele Bavota 和 Denys Poshyvanyk。 2019. 关于通过神经机器翻译学习有意义的代码更改。在第 41 届 IEEE/ACM 国际软件工程会议 (ICSE 19)的会议记录中。 25–36。 https://doi.org/10.1109/ ICSE.2019.00021 [39] Michele Tufano、Cody Watson、Gabriele Bavota、Massimiliano Di Penta、Martin White 和 Denys Poshyvanyk。 2018. 对通过神经机器翻译在野外学习错误修复补丁的实证调查。在第 33 届 ACM/IEEE 自动化软件工程国际会议（法国蒙彼利埃） (ASE 18)的会议记录中。美国纽约州计算机协会,832–837。 https://doi.org/ 10.1145/3238147.3240732 [40] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018.

用于更好地自动修复程序的上下文感知补丁生成。在第 40 届国际软件工程会议（瑞典哥德堡)(ICSE 18) 的会议记录中。美国纽约州计算机协会， 1-11。 https://doi.org/ 10.1145/3180155.3180233

[41] Martin White、Michele Tufano、Matias Martinez、Martin Monperrus 和 Denys Poshyvanyk。 2019. 通过深度学习代码相似性对程序修复成分进行排序和转换。在第 26 届IEEE软件分析、演化和再工程国际会议论文集 (SANER 19) 中。 479–490。 https:// doi.org/10.1109/SANER.2019.8668043

[42] Martin White、Michele Tufano、Christopher Vendome 和 Denys Poshyvanyk。 2016. 用于代码克隆检测的深度学习代码片段。在第 31 届 IEEE/ACM 国际自动化软件工程 (ASE)会议记录中。美国计算机学会,87–98。

[43]齐鑫和史蒂文·赖斯。 2017. 利用语法相关代码进行自动程序修复。在第 32 届 IEEE/ACM 国际自动化软件工程会议（美国伊利诺伊州厄巴纳-香槟)(ASE 17) 的会议记录中。 IEEE出版社,660–670。

[44]元元和沃尔夫冈·班扎夫。 2020. ARJA:通过多目标遗传编程自动修复 Java程序。 IEEE 软件工程汇刊 (TSE) 46, 10 (2020), 1040–1067。 https://doi.org/10.1109/TSE.2018。 2874648

[45] J. Zhu、T. Park、P. Isola 和 AA Efros。 2017.使用循环一致的对抗网络进行不成对的图像到图像的转换。 2017年IEEE国际计算机视觉会议（ICCV）。 2242–2251。 https://doi.org/ 10.1109/ICCV。 2017.244