软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3071−3092 [doi: 10.13328/j.cnki.jos.007110]

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

神经程序修复领域数据泄露问题的实证研究*

李卿源, 钟文康, 李传艺, 葛季栋, 骆 斌

(计算机软件新技术国家重点实验室(南京大学), 江苏 南京 210023)

通信作者: 葛季栋, E-mail: gjd@nju.edu.cn

摘 要: 修复软件缺陷是软件工程领域一个无法回避的重要问题, 而程序自动修复技术则旨在自动、准确且高效地修复存在缺陷的程序, 以缓解软件缺陷所带来的问题. 近年来, 随着深度学习的快速发展, 程序自动修复领域兴起了一种使用深度神经网络去自动捕捉缺陷程序及其补丁之间关系的方法, 被称为神经程序修复. 从在基准测试上被正确修复的缺陷的数量上看, 神经程序修复工具的修复性能已经显著超过了非学习的程序自动修复工具. 然而, 近期有研究发现: 神经程序修复系统性能的提升可能得益于测试数据在训练数据中存在, 即数据泄露. 受此启发, 为了进一步探究神经程序修复系统数据泄露的原因及影响, 更公平地评估现有的系统: (1) 对现有神经程序修复系统进行了系统的分类和总结, 根据分类结果定义了神经程序修复系统的数据泄露, 并为每个类别的系统设计了数据泄露的检测方法; (2) 依照上一步骤中的数据泄露检测方法对现有模型展开了大规模检测, 并探究了数据泄露对模型真实性能与评估性能间差异的影响以及对模型本身的影响; (3) 分析现有神经程序修复系统数据集的收集和过滤策略, 加以改进和补充, 在现有流行的数据集上, 基于改进后的策略构建了一个纯净的大规模程序修复训练数据集, 并验证了该数据集避免数据泄露的有效性. 由实验结果发现: 调研的 10 个神经程序修复系统在基准测试集上均出现了数据泄露, 其中, 神经程序修复系统 RewardRepair 的数据泄露问题较为严重, 在基准测试集 Defects4J (v1.2.0)上的数据泄露达 24 处, 泄露比例高达 53.33%. 此外, 数据泄露对神经程序修复系统的鲁棒性也造成了影响, 调研的 5 个神经程序修复系统均因数据泄露产生了鲁棒性降低的问题. 由此可见, 数据泄露是一个十分常见的问题, 且会使神经程序修复系统得到不公平的性能评估结果以及影响系统在基准测试集上的鲁棒性. 研究人员在训练神经程序修复模型时, 应尽可能避免出现数据泄露, 且要考虑数据泄露问题对神经程序修复系统性能评估产生的影响, 尽可能更公平地评估系统.

关键词: 程序自动修复; 神经程序修复; 深度学习; 数据泄露; 程序修复数据集

中图法分类号: TP311

中文引用格式: 李卿源, 钟文康, 李传艺, 葛季栋, 骆斌. 神经程序修复领域数据泄露问题的实证研究. 软件学报, 2024, 35(7): 3071−3092. http://www.jos.org.cn/1000-9825/7110.htm

英文引用格式: Li QY, Zhong WK, Li CY, Ge JD, Luo B. Empirical Study on Data Leakage Problem in Neural Program Repair. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3071−3092 (in Chinese). http://www.jos.org.cn/1000-9825/7110.htm

Empirical Study on Data Leakage Problem in Neural Program Repair

LI Qing-Yuan, ZHONG Wen-Kang, LI Chuan-Yi, GE Ji-Dong, LUO Bin

(National Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Repairing software defects is an inevitable and significant problem in the field of software engineering, while automated

* 基金项目: 国家重点研发计划(2022YFF0711404); 江苏省第六期“333 工程”领军型人才团队项目; 江苏省自然科学基金(BK20201250)

本文由“面向复杂软件的缺陷检测与修复技术”专题特约编辑张路教授、刘辉教授、姜佳君副研究员、王博博士推荐.

收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05

CNKI 网络首发时间: 2024-03-13

program repair (APR) techniques aim to alleviate software defect problem by repairing the defective programs automatically, accurately, and efficiently. In recent years, with the rapid development of deep learning, the field of automated program repair has emerged a method that utilizes deep neural networks to automatically capture the relationship between defective programs and their patches, called neural program repair (NPR). In terms of the number of defects that can be correctly repaired on the benchmark, NPR tools have significantly outperformed non-deep learning APR tools. However, a recent study found that the performance improvement of NPR systems may be due to the presence of test data in the training data, i.e., the data leakage. Inspired by this, to further investigate the causes and effects of data leakage in NPR systems and to evaluate existing systems more fairly, this study: (1) systematically categorizes and summarizes the existing NPR systems, defines the data leakage of NPR systems based on this classification, and designs the data leakage detection method for each category of system; (2) conducts a large-scale testing of existing models according to the data leakage detection method in the previous step and investigates the effect of data leakage on model realism and evaluation performance and the impact on the model itself; (3) analyzes the collection and filtering strategies of existing NPR system datasets, improves and supplements them, then constructs a pure large-scale NPR training dataset based on the improved strategy with the existing popular dataset, and verifies the effectiveness of this dataset in preventing data leakage. From the experimental results, it is found that the ten NPR systems studied in this investigation all had data leakage on the evaluation dataset, among which the NPR system RewardRepair had the more serious data leakage problem, with 24 data leaks on the Defects4J (v1.2.0) benchmark, and the leakage ratio was as high as 53.33%. In addition, data leakage has an impact on the robustness of the NPR system, and all five NPR systems investigated had reduced robustness due to data leakage. As a result, data leakage is a very common problem and can lead to unfair performance evaluation results of NPR systems and affect therobustness of the NPR system on the benchmark. When training NPR models, researchers should avoid data leakage as much as possible and consider the impact of data leakage on the evaluation of the performance of NPR systems to evaluate the NPR systems as fairly as possible.

Key words: automated program repair (APR); neural program repair; deep learning; data leakage; program repair dataset

程序自动修复(automated program repair, APR)[1,2]技术通过自动生成补丁, 能够帮助开发人员修复缺陷程序, 以保障软件质量, 因此成为软件工程领域一个备受关注的研究方向. 近年来, 随着深度学习(deep learning, DL)的兴起, 一种基于学习(learning-based)的程序自动修复技术开始受到研究人员的重视, 被称为神经程序修复(neural programrepair, NPR)[3]. 非学习的程序自动修复技术大多依赖于程序分析, 如基于启发式搜索(例如 GenProg[4]、ASTOR[5]、ARJA[6]和 SimFix[7])、基于语义约束(例如 DynaMoth[8]、Nopol[9]、Astor[10]和 Angelix[11])以及基于修复模板(例如 AVATAR[12]、TBar[13]、PAR[14]和 SketchFix[15])的修复技术. 相比之下, 神经程序修复的修复效果已经远远超过了这些非学习的程序自动修复[16], 并且神经程序修复能够实现端到端的工作流程, 使得修复过程更加自动化.

然而有研究表明: 数据泄露会导致对神经网络的评估结果膨胀, 进而导致对其性能评估的结果不真实[17]. 我们认为: 数据泄露同样会使神经程序修复系统在基准测试集上的修复效果膨胀; 这些出现数据泄露的缺陷被修复的真正原因可能是模型在训练时, 已经学习到了该缺陷的修复信息. 此外, 还有研究表明: 数据泄露会对神经网络的泛化性造成影响, 且会造成神经网络出现过拟合的现象. 但目前, 对于神经程序修复系统中数据泄露的研究普遍存在着一些问题: 首先, 由于神经程序修复系统的复杂性, 导致还没有一个对神经程序修复系统中数据泄露的明确定义; 其次, 现有研究大多只是在基准测试上进行简单的字符串匹配检测, 并未形成系统的数据泄露检测方法; 最后, 还没有研究进一步分析数据泄露对神经程序修复系统所造成影响. 因此, 亟需一项系统的实证研究, 对神经程序修复系统中的数据泄露进行科学的定义, 并对应产生一套合理的检测手段, 以及进一步分析数据泄露对神经程序修复系统所产生的影响. 因此, 我们在现有研究的基础上, 对神经程序修复系统中的数据泄露问题进行了一次系统的实证研究, 探究数据泄露在神经程序修复系统中的严重程度、数据泄露对现有神经程序修复系统的性能评估造成的影响以及数据泄露对模型本身造成的影响. 本文的主要贡献如下.

(1) 定义了神经程序修复系统中的数据泄露, 对应不同类型的数据泄露设计了相应的检测方法, 将检测方法合并成为一个神经程序修复系统数据泄露检测工具 NPRLeakageFinder, 并使用该工具对神经程序修复系统进行数据泄露检测;

(2) 探究了数据泄露对现有的神经程序修复系统的性能评估造成的影响以及对其鲁棒性造成的影响;

(3) 针对数据泄露问题提出解决方法, 设计了一套神经程序修复系统数据集的收集、过滤和划分策略,

参照该策略构建了一个纯净的数据集 Clean4J_Benchmark 供神经程序修复系统使用, 评估了该数据集的效用, 同时验证了所提出的数据集构建策略的有效性.

本文第 1 节介绍与神经程序修复系统相关的背景知识. 第 2 节介绍神经程序修复系统中数据泄露问题的相关研究. 第 3 节介绍本文的研究计划, 包括定义研究问题、设计实验方法等. 第 4 节归纳研究结果, 并得出研究结论. 第 5 节分析可能对实证研究的有效性产生影响的因素. 第 6 节对神经程序修复系统中的数据泄露问题进行总结与展望.

1 背景知识

1.1 神经程序修复

神经程序修复是一种狭义上的补丁生成技术, 神经程序修复系统利用各种深度神经网络(deep neural network, DNN)进行缺陷代码到正确代码的变换. 该系统通常采用序列到序列(sequence to sequence, Seq2Seq)架构的深度神经网络进行缺陷修复. Seq2Seq 架构的模型通常被用于自然语言处理(natural language processing, NLP)领域的神经机器翻译任务, 而基于 Seq2Seq 架构的神经程序修复模型, 修复缺陷的思想本质上与翻译任务类似[18,19], 是利用神经网络实现从缺陷代码的语义空间向补丁代码的语义空间的转换, 在一定程度上可以理解为将缺陷代码“翻译”成正确代码. 在训练过程中, 神经程序修复系统的目标是尽可能拟合真实(ground-truth)补丁每一个词元(token)的概率分布. 如图 1 所示, 神经程序修复系统生成一个补丁一般可分为 5 个阶段.

(1) 预处理(preprocessing): 预处理阶段将给定的缺陷代码转变成神经网络可接受的形式, 预处理经常使用到自然语言处理中的分词(tokenization)和嵌入(embedding)等技术.

(2) 编码(encoding): 编码阶段将预处理阶段生成的嵌入向量进行进一步编码, 这一步是为了让神经网络理解缺陷代码的深层语义.

(3) 解码(decoding): 解码阶段将编码得到的上下文向量进行解码, 生成预测补丁当前时间步词元的概率分布.

(4) 补丁重排序(re-ranking): 重排序阶段将模型生成的补丁进行重新排序, 将更可能是正确修复的补丁前置.

(5) 补丁验证(validation): 验证阶段将重排序后的补丁在测试上进行验证, 得到似真补丁. 最后, 对于程序自动修复工具, 一般还要对似真补丁进行人工校对(manualcheck), 检查是否能够真正修复对应的缺陷.

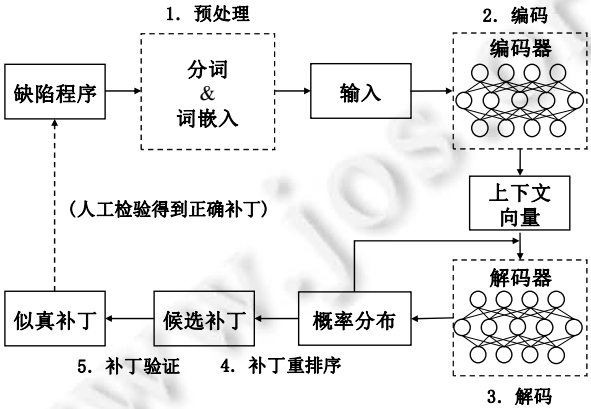


图 1 神经程序修复系统的一般流程图

近年来, 随着大语言模型(large language model, LLM)的蓬勃发展, 利用大语言模型微调(fine tuning)或零样本学习(zero-shot learning)的神经程序修复系统(例如 InferFix[20]、FitRepair[21]、AlphaRepair[22]和 SELF-

DEBUGGING[23])展示出更为强大的缺陷修复能力[24]. 这一方面得益于模型的训练数据和参数量都十分庞大, 另一方面得益于在修复缺陷程序时添加了一些提示(prompt)[25,26], 让模型知道当前要进行的是程序修复任务.

1.2 神经程序修复系统中的数据泄露

在深度学习领域, 数据泄露是一种常见的问题, 但由于神经程序修复模型具有多样的训练方式和多样的输入输出方式, 导致对其数据泄露的判定较为模糊. 因此, 本文对神经程序修复系统中的数据泄露设计了以下分类的定义.

- 缺陷-补丁对泄露(bug-fix pair). 该类型的数据泄露是指: 神经程序修复系统在基准测试集上的某条可修复的缺陷-补丁对出现在了训练数据当中, 即该条可修复的缺陷-补丁对中的缺陷部分出现在了某条训练数据的缺陷代码中, 同时, 神经程序修复系统生成的补丁出现在了同一条训练数据的补丁代码中.
- 缺陷代码泄露(buggy code). 该类型的数据泄露是指: 神经程序修复系统在基准测试集上的某条可修复的缺陷-补丁对的缺陷部分出现在了训练数据当中, 即该条可修复的缺陷-补丁对中的缺陷部分出现在了某一条训练数据的缺陷代码中.
- 补丁代码泄露(fixed code). 该类型的数据泄露是指: 神经程序修复系统在基准测试集上的某条可修复的缺陷-补丁对的补丁部分出现在了训练数据当中, 即该条可修复的缺陷-补丁对中由神经程序修复系统生成的补丁出现在了某一条训练数据的补丁代码中.

此处的“出现”是指: 基准测试集上的缺陷-补丁对(或缺陷-补丁对中的缺陷、补丁部分)与训练数据中的缺陷-补丁对字符串完全匹配(除去空格、空行以及注释等不含代码信息的字符), 或为子串关系, 即基准测试集上的数据是某条训练数据的子串. 选择字符串完全匹配(type-1 匹配)作为判定数据泄露的原因是, 我们认为: 只有当字符串完全匹配时, 才会使得神经程序修复系统在进行测试之前就已经完全学习到某一测试样本的语义、句法、代码结构以及从缺陷到补丁的转换范式等修复信息. 这使得神经程序修复系统在测试时遇到相同样本的情况下, 仅通过对该样本修复信息的记忆来输出补丁. 我们认为: 这种学习方法没有使得神经程序修复系统理解该样本的修复范式, 而是对修复信息的简单记忆. 而其余类型的代码匹配, 我们则认为不会引入狭义的数据泄露. 表 1 是几种代码匹配类型对应的含义[27].

表 1 代码匹配类型及其定义

名称	类型	定义
Type-1 匹配	句法匹配	除去空格,空行和注释后,两段代码相同
Type-2 匹配	句法匹配	除去对标识符(函数名、类名和变量名等)的修改, 两段代码相同
Type-3 匹配	句法匹配	片段被部分重排序、增加或删除, 修改后, 两段代码相同
Type-4 匹配	语义匹配	语义相同但句法上没有联系

以上是数据泄露的分类与定义, 对于不同的神经程序修复系统, 对应的数据泄露类型可能不同; 而对于不同的数据泄露类型, 其检测的方式也不相同. 我们将在第 3.5 节对此展开详细的介绍.

1.3 神经程序修复系统的鲁棒性及其评估方法和评价指标

1.3.1 神经程序修复系统鲁棒性的定义

深度学习模型的鲁棒性通常被定义为对输入产生微小变化, 输出保持不变的能力. 现有的深度学习模型普遍存在鲁棒性较差的问题. 例如: 对于卷积神经网络(convolutional neural network, CNN), 在对输入的图片加入对抗扰动后, 卷积神经网络便不能再次识别该图片[28]; 对于神经机器翻译模型(neural machine translation, NMT)[29], 在对输入语句做出极小改变后, 可能会引起翻译结果的剧烈改变[30]. 对于神经程序修复系统, 其鲁棒性则被引申为: 对于语义一致的输入, 神经程序修复系统生成语义一致的输出的能力. 而语义一致性的定义是: 对于两段语义一致的程序, 针对域中的任何输入, 程序的行为或输出结果是一致的[31]. 神经程序修复系统鲁棒性和语义一致性的定义可被表示为以下公式.

$$\forall p, p_t \in P: p=p_t,\ NPR(p)=NPR(p_t) \tag{1}$$

$$\forall i\in I, p(i)=p_t(i)\rightarrow p=p_t \tag{2}$$

1.3.2 神经程序修复系统鲁棒性的评估方法

语义一致性变换(semantic identical transformation)是一种代码变换方法, 可以在保留代码语义或功能不变的前提下对代码进行修改. 以下是语义一致性变换的公式化定义.

$$\forall i\in I: c(i)=c_t(i), Transformation(c)=c_t\wedge Transformation(c_t)=c \tag{3}$$

对于两段程序 c 和 c_t, 若程序 c 和 c_t 是语义相同的, 则称程序 c 和 c_t 互为语义一致性变换程序. 而将程序 c 转变为 c_t 或将 c_t 转变为 c 的操作, 则被称为语义一致性变换.

例如一段 Java 语言的字符串变量声明语句“String message=“Hello, World!”;”, 如果在声明变量 message 时进行了对字符串类型的强制类型转换, 即“String message=(String) “Hello, World!”;” 则不会对语义有任何改变, 这样的操作即可被称为语义一致性变换. 根据神经程序修复系统鲁棒性的定义, 我们不难得出, 语义一致性变换可以用来评估神经程序修复系统的鲁棒性. 对于一个神经程序修复系统, 如果将一对语义一致性变换前后的程序输入到模型中, 模型两次输出的补丁语义不一致, 则该模型的鲁棒性存在问题.

1.3.3 神经程序修复系统鲁棒性的评价指标

对于神经程序修复系统鲁棒性的评估, 我们采用了 Ge 等人[31]定义的评价指标差异突变体百分比(percentage of diff mutants, *PDM*)和突变后差异百分比(percentage of diff after mutating, *PDA*).

- 差异突变体百分比

对于一个缺陷, 计算其语义一致性变换后的所有变体输入神经程序修复系统中后得到语义不一致的补丁的比例. 通过计算 *PDM*, 我们可以获得神经程序修复系统对于语义一致的输入生成一致性输出的能力, 从而反映该神经程序修复系统的鲁棒性. *PDM* 可表示为以下公式.

$$PDM=\frac{NDM}{NAM} \tag{4}$$

其中, *NDM* 为可修复的缺陷程序进行语义一致性变换后, 所有变体的补丁中出现语义不一致的数量; *NAM* 为所有可修复的缺陷程序进行语义一致性变换后, 所有变体的数量. 例如: Defects4J 中的 Closure_86 缺陷可以被神经程序修复系统 SequenceR 修复, 而 Closure_86 缺陷经过语义一致性变换后得到了 4 个变体, 其中 2 个变体的补丁与本体的补丁语义不一致(即无法被 SequenceR 再次修复), 那么 SequenceR 针对 Closure_86 缺陷的 *PDM*=50%.

- 突变后差异百分比

对于一个缺陷, 计算其语义一致性变换后的变体是否存在输入神经程序修复系统中后得到语义不一致的补丁的情况: 若出现, 该缺陷本体将被视为突变后存在差异. 通过计算 *PDA*, 我们可以获得神经程序修复系统是否会存在对于语义一致的输入生成不一致输出的情况, 从而反映该神经程序修复系统的鲁棒性. *PDA* 可表示为以下公式.

$$PDA=\frac{NDS}{NAS} \tag{5}$$

其中, *NDS* 为可修复的缺陷程序进行语义一致性变换后, 存在语义不一致的补丁的缺陷本体的数量; *NAS* 为所有可修复的缺陷程序. 对于一个缺陷, 如果其中某一变体的修复补丁与本体的补丁语义不一致, 那么这个缺陷就会被计入 *NDS*. 例如: Defects4J 中有 76 个缺陷可以被神经程序修复系统 CURE 修复, 而其中 24 个缺陷经过语义一致性变换后存在与本体的补丁语义不一致的补丁(即存在变体, 无法被 CURE 再次修复), 那么 CURE 在 Defects4J 基准测试上的 *PDA*=31.58%.

2 相关工作

数据泄露一直是困扰深度学习领域的一个问题, 当发生数据泄漏时, 模型已经学习到了部分评估数据的分布, 导致其出现过拟合(overfitting)的现象, 同时, 模型的鲁棒性也会受到影响[32].

Elangovan 等人[33]指出: 在自然语言处理领域, 训练数据和测试数据之间的重叠(overlap)会导致对模型性

能评估的不准确, 将模型的记忆能力(memorization)视作泛化能力(generalization), 导致模型在基准测试上的效果并不能作为现实场景中性能的有效参考. 他们检测了几个自然语言处理任务上的公开数据集上的数据泄露, 包括命名实体识别(named entity recognition, NER)、关系抽取和文本分类任务, 并研究了数据泄露对模型的记忆能力和泛化能力的影响. 研究表明: 进行检测的几个常用训练集都出现了不同程度的数据泄露, 其中, 关系抽取任务上的数据集 AIMED(R)的数据泄露比例高达 73%. 并且他们发现: 当测试集与训练集相似时, 模型的 *F-score* 会相对更高, 但实际上评估的却是模型的记忆能力,而非泛化能力; 但当训练模型的数据具有足够的体量以及较好的多样性时, 模型的记忆能力便不再是缺点. 但通常情况下, 训练模型的数据并没有达到足够规模. 因此, 考虑训练数据与测试数据之间的重叠是十分重要的. 否则, 当模型在真实场景下遇到训练数据中没有出现过的数据, 可能会表现不佳.

Lewis 等人[34]对开放领域问答任务上的 3 个流行的基准测试集进行了详细的研究, 并发现 60%–70%的测试数据出现在了训练集中. 他们还发现: 测试集中 30%的数据中的问题部分, 在它们对应的训练集中有一个表达相近的问题. 基于上述发现, 他们评估了现有的开放领域问答模型, 以了解这些模型的泛化性以及驱动它们性能的因素. 他们发现: 在无法直接从训练集中获取的数据上模型的性能通常很差, 存在数据泄露的数据和不存在数据泄露的数据, 模型在两者上性能的差异高达 63%. 最终, 他们通过对比实验, 发现简单的近邻模型在真实场景下的性能甚至优于 BART 问答模型, 进一步强调了模型对训练集的记忆和数据泄露, 导致了评估结果的不公平.

Jiang 等人[35]对代码语言模型(code language model)在程序自动修复场景下的应用进行了研究, 他们为了避免数据泄露造成不公平的评估结果, 构建了一个用于程序自动修复任务的基准测试集 HumanEval-Java. 该数据集是一个人工构建的数据集, 基于一个由 Python 编写的、用于代码生成任务的基准测试集 HumanEval, 他们手动将 HumanEval 中的 Python 程序及其测试用例转换为 Java 程序和 Junit 测试用例, 然后在正确的 Java 程序中注入缺陷, 以创建程序自动修复基准测试. HumanEval-Java 包含 164 个单块(single-hunk)缺陷, 涉及简单的操作符使用错误和复杂的逻辑错误, 且存在多行缺陷. 由于 HumanEval-Java 是手动转换得到的, 且其缺陷是人工注入的, 所以他们认为, HumanEval-Java 不存在数据泄露, 对于神经程序修复系统是一个公平的基准测试集. 因此, 他们在程序自动修复数据集上微调了几个基线代码语言模型, 并在 HumanEval-Java 上与现有的基于深度学习的神经程序修复系统进行了性能的比对. 研究表明, 微调后的代码语言模型比现有的基于深度学习的神经程序修复系统多修复了 46%–164%的缺陷.

Huang 等人[16]则检测了现有神经程序修复系统的一些常见训练集与基准测试集 Defects4J 的数据重叠. 研究表明, 75%的训练集与 Defects4J 发生了数据重叠(https://github.com/APR-SURVEY/APR-Survey/tree/main/dataset_overlap). 这些数据重叠会导致模型在训练阶段学习到测试样本的修复知识, 以至于让研究人员产生修复结果更好的错觉. 此外, 他们认为: 在未来, 使用更细粒度的数据重叠检测在更多的基准测试集上分析重叠问题, 仍将是程序自动修复领域研究的关键课题.

以上工作大部分是对数据泄露及其影响的研究, 证实了数据泄露会对自然语言处理领域(开放领域问答、命名实体识别、关系抽取和文本分类)模型的性能评估以及模型本身造成影响; 也有少部分研究简单地检测了神经程序修复系统训练集上的重叠问题. 根据上述相关工作, 我们不难怀疑: 数据泄露确实在神经程序修复系统中存在, 且同样会对模型的性能评估以及模型本身造成影响. 因此, 在第 3 节, 我们对神经程序修复系统中的数据泄露问题展开了系统性的研究, 研究现有的神经程序修复系统中存在何种程度的数据泄露, 以及数据泄露对模型本身造成的影响.

3 研究设计

3.1 概 述

为了对神经程序修复系统中的数据泄露进行系统的研究, 我们在第 3.2 节提出了 3 个研究问题, 并通过解决这些研究问题, 系统地探究神经程序修复系统中的数据泄露问题. 图 2 是对神经程序修复系统中的数据泄

露问题进行系统性实证研究的总体框架图. 首先, 我们通过调查不同神经程序修复系统的训练方式和输入输出方式, 判断其对应的数据泄露类型; 其次, 在其训练数据和所使用的基准测试集上进行数据泄露检测, 得出结论 1; 接着, 我们根据结论 1 对基准测试集中可修复的缺陷进行分类, 分为存在数据泄露和不存在数据泄露的两组数据, 对这两组数据进行语义一致性变换, 分别计算这两组数据上的差异突变体百分比和突变后差异百分比, 分析神经程序修复系统的鲁棒性是否被数据泄露问题所影响, 以此得出结论 2; 最后, 我们对现有的神经程序修复系统的训练数据进行整理和过滤, 得到一个纯净的数据集, 供神经程序修复系统使用, 以避免发生数据泄露.

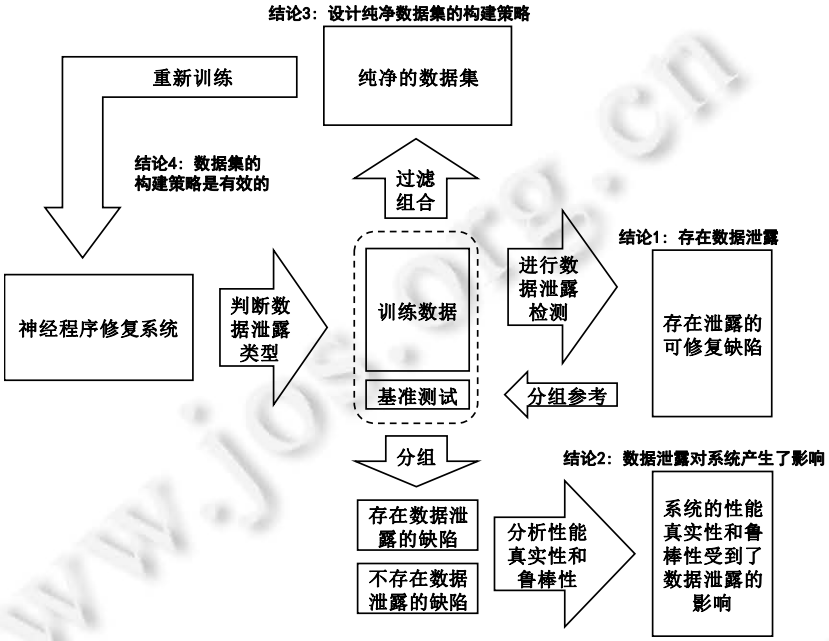


图 2 数据泄露实证研究的总体框架图

3.2 研究问题

- RQ1: 现有的神经程序修复系统中是否存在数据泄露? 该研究问题的目的是验证神经程序修复系统中存在数据泄露. 对于不同类型的数据泄露, 我们采用不同的检测方法对其进行检测, 并将检测方法总结为一个神经程序修复系统数据泄露检测工具 NPRLeakageFinder.
- RQ2: 数据泄露问题给神经程序修复系统带来了怎样的影响?
  - RQ2.1: 数据泄露问题是否会影响神经程序修复系统性能评估的真实性? 该研究问题的目的是探究数据泄露问题是否对神经程序修复系统的性能评估产生了影响. 为了确定这一问题, 我们统计了现有神经程序修复系统中有多少可修复的缺陷-补丁对是存在数据泄露的, 把这些可修复的缺陷-补丁对列入待观察的可修复缺陷(即我们不确定这些可修复缺陷是否是因为数据泄露导致其能够被修复), 并通过后续 RQ4 中的实验, 验证这些待观察的可修复缺陷中有多少是无法被神经程序修复系统真正修复的.
  - RQ2.2: 数据泄露问题是否会影响神经程序修复系统的鲁棒性? 该研究问题的目的是探究数据泄露问题是否对神经程序修复系统的鲁棒性产生了影响. 对此, 我们对现有的神经程序修复系统可修复的缺陷做语义一致性变换, 检验语义一致性变换后的缺陷是否还可以被原有的神经程序修复系统修复.
- RQ3: 如何避免或缓解数据泄露问题? 该研究问题的目的是探究现有的神经程序修复系统的数据采

集策略、数据过滤策略以及数据集划分策略存在的问题, 分析导致数据泄露问题的原因, 并以此设计一套能够避免数据泄露的数据集构建策略.

- RQ4: 本文提出的数据集构建策略在多大程度上能缓解数据泄露? 该研究问题的目的是探究在 Clean4J_Benchmark 上重新训练后, 是否能缓解或是避免数据泄露.

3.3 神经程序修复系统的选取

我们制定了一个选择神经程序修复系统的准则, 以便获取待研究的神经程序修复系统. 准则内容如下.

(1) 训练集可得性: 该神经程序修复系统的训练集是可以获得的. 我们要对神经程序修复系统的训练集进行数据泄露检测, 所以训练集不可得的神经程序修复系统无法进行研究.

(2) 基准测试可得性: 使用了公开的基准测试集对性能进行了评估. 如果神经程序修复系统没有使用基准测试集进行性能评估, 就无须对其进行数据泄露检测.

(3) 修复结果可得性: 神经程序修复系统在基准测试集上具体修复的缺陷是可以获得的. 由于复现深度神经网络模型可能存在结果上的差异, 因此我们只检测公开(或部分公开)了具体修复结果的神经程序修复系统, 例如 SequenceR 公开了其在 Defects4J (v1.2.0)上具体修复的 14 个缺陷, 因此 SequenceR 满足条件.

(4) 补丁正确性自动评估(automated patch correctness assessment, APCA)系统不做考虑. 自动补丁正确性评估系统不会生成补丁, 因此不涉及缺陷修复相关的数据泄露, 所以我们不作考虑.

(5) 可用性: 神经程序修复系统的源代码是可以获得且可以使用的. 因为我们要对语义一致性变换后的缺陷进行修复, 所以没有开源的神经程序修复系统无法进行研究.

(6) 可修改性: 神经程序修复系统的源代码是可以修改的. 我们要保证语义一致性变换后的缺陷可以再次利用该神经程序修复系统, 所以神经程序修复系统的源代码是可以被修改为能够接受语义一致性变换后的缺陷作为输入的形式.

其中, 满足准则(1)–准则(4)的神经程序修复系统可以进行数据泄露检测, 而满足准则(5)和准则(6)的神经程序修复系统则可以进一步分析鲁棒性问题. 我们将研究场景聚焦在了可修复的语言类型为 Java 或包括 Java 的神经程序修复系统(Java 程序修复是目前程序修复界最流行的修复场景, 且目前主流的程序自动修复基准测试绝大部分是针对 Java 语言的, 如最知名的 Defects4J 基准测试集).

最终, 我们选择了 10 个神经程序修复系统, 分别为 Recoder[36]、SequenceR[37]、RewardRepair[38]、TRANSFER[39]、CoCoNuT[40]、CURE[41]、DLFix[42]、DEAR[43]、AlphaRepair[22]和 CIRCLE[44]. 其中: AlphaRepair、CIRCLE、DLFix 和 DEAR 无法满足准则(5), 而 TRANSFER 无法满足准则(6). 因此, 只有 Recoder、SequenceR、RewardRepair、CoCoNuT 和 CURE 能够进行鲁棒性分析. 以下是对上述神经程序修复系统的简要介绍.

Recoder[36]是一个基于语法引导的、带有占位符生成的编辑解码器结构的神经程序修复系统. Recoder 可以生成缺陷方法抽象语法树(abstract syntax tree, AST)上的编辑操作, 进而生成修复补丁. 因为 Recoder 生成的是编辑而不是直接生成修改后的代码, 所以 Recoder 可以有效地表示微小的编辑操作. Recoder 使用抽象语法树遍历序列(AST traversal sequence)、标签嵌入(tag embedding)和基于抽象语法树的图(AST-based graph)作为编码器的输入, 解码器部分使用提供者/决策者(provider/decider)结构, 在当前抽象语法树上进行编辑操作. Recoder 的编码器包括 3 个部分, 分为 Code Reader、AST Reader 和 Tree Path Reader; 解码器由 Providers(又包括 Rule Predictor、Tree Copier 和 Mutation Locator 这 3 种结构)和 Decider 两部分组成.

SequenceR[37]是一个带有复制机制(copy mechanism)的基于序列到序列架构的神经程序修复系统, 本质上, SequenceR 是一个基于统计学的神经机器翻译模型, 它被训练用作将缺陷代码翻译为修复代码. SequenceR 仅适合单行缺陷的修复, 采用带缺陷定位的抽象缺陷类(只保留内部变量的声明、非缺陷方法的声明和缺陷方法的类)作为输入, 输出修复行.

RewardRepair[38]提出了一种改进基于神经机器翻译的程序修复工具的方法, 其在语义训练(semantic training)时的损失函数基于生成的候选补丁的编译和执行信息, 通过构造与编译执行信息相关的损失函数, 以

激励神经网络生成能够通过编译和正确执行的补丁. RewardRepair 首先对神经网络进行句法训练(syntactic training), 这和一般的神经程序修复系统的训练过程基本相同; 然后, 使用可编译的缺陷-补丁对和可执行的测试用例继续对神经网络进行语义训练. 语义训练基于一个判别模型, 这个判别模型使用了4种鉴别器, 分别为区别鉴别器(difference discriminator)、可编译性鉴别器(compilability discriminator)、似真性鉴别器(plausibility discriminator)以及回归鉴别器(regression discriminator), 分别用来鉴别生成的修复代码是否同缺陷代码有区别、生成的修复代码是否能够通过编译、生成的修复代码是否能够通过测试、生成的修复代码是否正确. 该判别模型会根据不同的结果给出不同的激励, 用于构成语义训练时 RewardRepair 的损失函数.

TRANSFER[39]是一个利用深度学习技术的缺陷定位和神经程序修复系统. TRANSFER 利用深度神经网络在大规模的开源代码数据集上学习深层语义特征和代码领域的迁移知识, 以改进基于模板的程序修复技术. TRANSFER 的程序修复模块是一个基于双向长短期记忆神经网络(bidirectional long short-term memory network, Bi-LSTM)的多分类器模型, 该分类器利用代码的深层语义特征对程序修复所用的修复模板进行排序, 其本质上是一个选择修复模板的选择器. 基于模板的程序修复可分为 4 个步骤: 故障定位、修复模板选择、供体代码搜索和候选补丁验证. 而 TRANSFER 的程序修复模块对第 2 个步骤进行优化, 在候选的 11 种修复模板(TRANSFER 根据缺陷的普遍程度从 TBar 中选择了 11 种修复模板作为候选)中进行选择.

CoCoNuT[40]是一个使用结合上下文感知的卷积神经网络作为神经机器翻译架构, 同时使用集成学习(ensemble learning)的神经程序修复系统. 为了更好地表征缺陷代码上下文中的语义信息, CoCoNuT 设计了一个结合上下文感知的神经机器翻译架构, 分别处理缺陷代码及其周围的上下文. 这种结构使得 CoCoNuT 能够更好地区分缺陷行和其上下文. CoCoNuT 使用卷积神经网络替代循环神经网络(recurrent neural network, RNN), 因为卷积神经网络可以被堆叠, 以提取层次特征用, 以在不同粒度(如语句和函数)上更好地表征源代码的结构和语义信息. 此外, CoCoNuT 可以修复多种程序语言的缺陷(Java、C、Python 和 JavaScript).

CURE[41]是一个基于预训练程序语言模型的神经程序修复系统. 首先, CURE 在进行程序自动修复任务之前, 先在大量的程序源代码上进行预训练; 其次, CURE 设计了一种新的代码感知搜索策略, 通过专注于可编译的补丁和长度接近缺陷代码的补丁, 以寻找更为正确的修复补丁; 最后, CURE 使用子词分词技术获得更小的搜索空间, 以包含更为正确的修复补丁.

DLFix[42]是一个具有双层结构的神经程序修复系统. 它将程序自动修复任务视为代码变换任务, 从训练数据中的缺陷-补丁对中进行学习. DLFix 的第 1 层称为上下文学习层(context learning layer), 是一个基于树的循环神经网络, 用于学习缺陷上下文的语义信息, 第 1 层的输出被用作第 2 层的附加加权输入; 第 2 层称为变换学习层(transformation learning layer), 与第 1 层的结构相同, 用于学习缺陷代码到补丁代码的变换信息.

DEAR[43]是一个能够修复 1 个或多个代码块中的 1 个或多个连续语句缺陷的神经程序修复系统. DEAR 由一个两层的基于树的长短期记忆神经网络构成, 使用循环训练并采用分治策略来学习代码变换, 以帮助 DEAR 在缺陷代码及其上下文中学习到正确的缺陷修复变换.

AlphaRepair[22]是一个直接利用大型预训练程序语言模型的神经程序修复系统. AlphaRepair 没有在程序自动修复任务上进行微调或重新训练, 而是采用零样本学习的方式进行程序自动修复任务. AlphaRepair 采用 CodeBERT[45]作为基线模型, 通过对缺陷行进行掩码处理, 得到格式为缺陷行、上下文和掩码行的输入. CodeBERT 本身是一种掩码语言模型(masked language model, MLM), 即: 已知掩码位置的上下文, 预测掩码位置处的正确词元. 这与 AlphaRepair 设计的输入形式相似, 因此其零样本学习的效果较好.

CIRCLE[44]是一个跨语言的神经程序修复系统, 采用 T5[46]作为基线模型, 通过对数据进行处理进而生成带有提示的缺陷代码, 用来作为模型的输入. 此外, CIRCLE 使用了持续学习(continual learning), 使得模型可以修复不同类型程序语言的缺陷.

3.4 基准测试集的选取

根据第 3.3 节中选择的 10 个神经程序修复系统, 我们选择了相应被用作性能评估的基准测试集, 分别为 Bugs.jar[47]、Defects4J[48]、IntroClassJava[49]和 QuixBugs[50]. 以下是对选取的基准测试集的简要介绍.

Bugs.jar[47]是一个基于 Java 语言的缺陷基准测试集. 该数据集中的缺陷来自真实世界中的 8 个大型且流行的开源 Java 项目, 包含 Apache 开源项目中的 1 158 个程序缺陷. Bugs.jar 不仅包含了测试用例, 还包含缺陷的问题编号(issue id)以及缺陷报告(bug report), 可以用来获取相应的缺陷在项目中的记录.

Defects4J[48]是一个对 Java 程序的受控测试研究的基准测试集, 由一个数据集和可扩展的框架组成, 其初始版本(v0.1.0)包含了来自 5 个真实世界开源程序中的 357 个缺陷, 每一个缺陷程序中都包含至少一个缺陷, 以及相应的测试套件可以暴露该缺陷. Defects4J 是可扩展的, 且非常便于访问其缺陷和修复程序以及相应的测试套件. 此外, Defects4J 还为软件测试研究中的常见任务提供了一个高级接口, 方便研究人员使用. 随着该基准测试集的不断扩展, 如今 v2.0.0 版本已经扩展到 17 个项目共 835 个缺陷.

IntroClassJava[49]是一个基于 Java 语言的基准测试集, 并带有配套的 JUnit 单元测试用例, 可用于 Java 缺陷程序的故障定位或缺陷修复任务. 由 Durieux 和 Monperrus 通过人工编写转换规则的自动化脚本, 将 IntroClass(基于 C 语言)转换得到的. 通过执行 InetoroClass 提供的输入, 来检查对应转换得到的 IntroClassJava 程序是否具有与原始 C 程序相同的行为. 此外, 他们还将原始的输入输出测试用例转换为标准的 JUnit 单元测试用例. 由于删除了其中的部分重复程序以及未能正确转换的程序, 最终数据集中包含 297 个缺陷程序.

QuixBugs[50]是一个多语言程序修复基准测试集, 包含了 Python 和 Java 两种语言的缺陷程序. QuixBugs 由 40 个缺陷程序组成, 其中, 程序代码由几行到几十行不等. 每个程序包含的缺陷都是单行的, 其中, Java 语言的缺陷程序是人工编写的, 与 Python 语言的缺陷程序相互对应. QuixBugs 基准测试集基于 Quixey Challenge[51], 而 Quixey Challenge 是一个让程序员在 1 min 的时间内修复一个简短的程序缺陷的挑战.

3.5 实验方法

这里首先需要注意: 我们所选择的神经程序修复系统, 并没有在所选择的全部基准测试上进行了实验; 另外, 进行了实验的某些神经程序修复系统, 其具体的修复结果(即可以修复的缺陷的名称、ID 等可以确定具体修复了哪些缺陷)并未公开. 我们对未进行实验或未公开修复结果的神经程序修复系统不再进行额外实验或复现实验, 因为额外的实验或复现并不能保证可与其宣称在该基准测试上修复缺陷的个数相匹配, 从而会影响后续对性能进行重新排序和鲁棒性分析的工作, 所以我们并不考虑对这部分进行额外实验或复现实验.

对于 RQ1: 首先, 我们对现有的神经程序修复系统进行系统的调查研究, 以确定每个神经程序修复系统的训练方式以及输入输出方式等特征; 然后, 我们根据这些特征确定其对应的数据泄露类型; 最后, 我们根据其数据泄露类型进行不同的数据泄露检测. 我们将现有神经程序修复系统的训练方式进行以下分类.

- 直接训练(training from scratch)
- 预训练-微调(pretrain then finetune)
- 预训练-零样本学习(pretrain then zero-shot)

将输入输出方式进行以下分类.

- 缺陷方法-修复行(buggy method-fixed line)
- 缺陷行+上下文-修复行(buggy line+context-fixed line)
- 缺陷类-修复行(buggy class-fixed line)
- 缺陷行-修复行(buggy line-fixed line)
- 缺陷方法-修复方法(buggy method-fixed method)
- 缺陷类-修复类(buggy class-fixed class)
- 输出其他(-other)

以下是针对不同训练方式和输入输出方式的神经程序修复系统, 对应的数据泄露类型.

- 缺陷-补丁对泄露: 从训练方式判断, 直接训练和预训练-微调的神经程序修复系统, 其数据泄露类型均为缺陷-补丁对泄露. 因为无论是进行直接训练还是进行微调(预训练数据对下游任务的影响相对较小), 其训练数据都是缺陷-补丁对的格式, 因此其数据泄露类型为缺陷-补丁对泄露.
- 补丁代码泄露: 针对训练方式为预训练-零样本学习的神经程序修复系统, 其数据泄露类型为补丁代

码泄露. 因为仅进行预训练的神经程序修复系统, 并不是神经程序修复任务上的专用模型, 其训练数据也并不是缺陷-补丁对的格式. 本质上, 此类神经程序修复系统是一个生成模型, 所以我们只需要保证模型在预训练时没有学习到补丁代码的信息, 即没有发生补丁代码泄露.

- 缺陷代码泄露: 从输入输出方式判断, 绝大部分的输入输出方式都不影响我们采用训练方式对数据泄露类型的判断结果, 只有一种输入输出方式会产生影响, 即输出其他(-other). 输出为其他的神经程序修复系统的数据泄露类型为缺陷代码泄露, 因为对于一个神经程序修复模型, 如果模型的输出不是代码, 而是其他(如 TRANSFER 的神经程序修复部分的输出是一个概率分布, 用来从 TBar 中选择修复模板), 那么对于此类神经程序修复系统, 我们只需要保证模型在训练时没有学习到缺陷代码的信息, 即没有发生缺陷代码泄露.

当确定了一个神经程序修复系统的数据泄露类型时, 我们便能够采用特定的数据泄露检测方式进行检测. 以下是针对不同数据泄露类型的检测方法.

- 检测缺陷-补丁对: 该数据泄露检测方式针对缺陷-补丁对泄露. 我们检测基准测试集 B 中缺陷-补丁对是否出现在训练集 T 中. 具体做法为: 我们检测基准测试集 B 的某一缺陷-补丁对 j 的缺陷代码是否与训练集 T 的某一缺陷-补丁对 i 的缺陷代码字符串完全匹配, 或 j 的缺陷代码是 i 的缺陷代码的子字符串. 若满足上述条件, 我们继续检查 j 的补丁代码是否与 i 的补丁代码字符串完全匹配, 或为子串关系. 若同样满足该条件, 我们认为基准测试集 B 中第 j 条数据出现了缺陷-补丁对泄露, 由训练集 T 的第 i 条数据引起的数据泄露.
- 检测缺陷代码: 该数据泄露检测方式针对缺陷代码泄露. 我们检测基准测试集 B 的某一缺陷-补丁对 j 的缺陷代码是否与训练集 T 的某一缺陷-补丁对 i 的缺陷代码字符串完全匹配, 或 j 的缺陷代码与 i 的缺陷代码为子串关系. 若满足, 我们认为基准测试集 B 中第 j 条数据出现了缺陷代码泄露, 由训练集 T 的第 i 条数据引起的缺陷代码泄露.
- 检测补丁代码: 该数据泄露检测方式针对补丁代码泄露. 我们检测基准测试集 B 的某一缺陷-补丁对 j 的补丁代码是否与训练集 T 的某一缺陷-补丁对 i 的补丁代码字符串完全匹配, 或 j 的补丁代码与 i 的补丁代码为子串关系. 若满足, 我们认为基准测试集 B 中第 j 条数据出现了补丁代码泄露, 由训练集 T 的第 i 条数据引起的补丁代码泄露.

我们将以上 3 种检测方法集成在 NPRLeakageFinder 中. 使用 NPRLeakageFinder 时, 只需选择检测方法, 即可对神经程序修复系统进行数据泄露检测. 表 2 统计了现有的神经程序修复系统(修复语言为 Java 或包括 Java)[3]的训练方式、输入输出方式、训练集、测试集、基准测试集的使用情况以及数据泄露检测方式等信息.

对于 RQ2, 我们计算了 10 个神经程序修复系统在不同基准测试集上的性能真实性, 并进行了排名.而后我们设计实验, 验证数据泄露对神经程序修复系统鲁棒性存在影响的假设. 首先, 我们对现有的神经程序修复系统中可修复的缺陷代码, 按照 RQ1 的结论划分为存在数据泄露的可修复缺陷和不存在数据泄露的可修复缺陷; 然后, 我们将 5 个能够进行鲁棒性分析的神经程序修复系统所有可修复的缺陷方法解析出来; 接着, 我们分别对以上两组缺陷方法进行语义一致性变换, 将变换后的代码再次输入到原先的神经程序修复系统中进行修复, 通过计算鲁棒性评估指标, 对比两组缺陷代码的修复效果是否存在显著的差异. 如果存在数据泄露的可修复缺陷, 经语义一致性变换后被再次修复的比例低于不存在数据泄露的可修复缺陷, 就证实了数据泄露会影响系统的鲁棒性. 该实验适用于在基准测试集存在测试用例的情况下进行, 当我们计算 *PDM* 和 *PDA* 时, 是将错误的补丁和似真补丁视为语义不一致的补丁, 因此需要该基准测试集有相应的测试用例, 能够对修复后的程序进行验证. 一般情况下, 语义一致性变换会改变代码行数以及缺陷行位置, 但神经程序修复系统普遍不具有缺陷定位功能. 因此, 如果采用了会导致代码行数以及缺陷行位置发生改变的语义一致性变换, 需要额外对缺陷行进行重新定位. 不过, 变量重命名(variable renaming, VR)和条件重排序(reorder condition, RC)这两种语义一致性变换可以保证变换前后代码行数和缺陷行位置均不发生改变. 我们采用这两种方法对基准测试集中的缺陷代码进行语义一致性变换, 供鲁棒性分析实验使用.

表 2 神经程序修复系统的相关信息

神经程序修复系统	训练方式	输入输出方式	检测方式	可修复语言种类	训练集	测试集	使用的基准测试集
Recoder	training from scratch	method-method	bug-fix	Java	Recoder	Recoder	Defects4J, QuixBugs, IntroClassJava
DLFix	training from scratch	method-line	bug-fix	Java	BigFix	BigFix	Defects4J, Bug.jar
SequenceR	training from scratch	class-line	bug-fix	Java	Bugs2Fix, CodRep	CodRep4	Defects4J
RewardRepair	training from scratch	method-line	bug-fix	Java	CoCoNuT, MegaDiff, CodRep	–	Defects4J, Bug.jar, QuixBugs
TRANSFER	training from scratch	method-other	bug	Java	TRANSFER	TRANSFER	Defects4J
DEAR	training from scratch	method-method	bug-fix	Java	BigFix, CPatMiner	BigFix, CPatMiner	Defects4J
CoCoNuT	training from scratch	method-line	bug-fix	Java, C, Python, JavaScript	CoCoNuT	CoCoNuT	Defects4J, QuixBugs
CURE	pretrain then finetune	method-line	bug-fix	Java, C, Python, JavaScript	CoCoNuT	CoCoNuT	Defects4J, QuixBugs
AlphaRepair	pretrain then zero-shot	method-line	fix	Java, Python	CodeSearchNet	CodeSearchNet	Defects4J, QuixBugs
CIRCLE	pretrain then finetune	method-line	bug-fix	Java, C, Python, JavaScript	CoCoNuT	CoCoNuT	Defects4J
CodeBERT	pretrain then finetune	line-line	bug-fix	Java	ManySStuBs4J	ManySStuBs4J	无
Cornor, *et al*.	Training from scratch	line-line	bug-fix	Java	Cornor	Cornor	无
T5	pretrain then finetune	method-method	bug-fix	Java	Bugs2Fix	Bugs2Fix	无
CompDefect	training from scratch	method-method	bug-fix	Java	Function-SStuBs4J	Function-SStuBs4J	无
SeqTrans	pretrain then finetune	method-method	bug-fix	Java	Bugs2Fix, Ponta	Bugs2Fix, Ponta	无
CoditT5	pretrain then finetune	method-method	bug-fix	Java	Bugs2Fix	Bugs2Fix	无

变量重命名会在方法中选择一个变量或参数进行重命名. 因为方法的参数或变量是局部性的, 因此该方法不会改变程序的语义. 图 3 是一个变量重命名的示例.

```
public int compareTo (ApplicationId other) {
    int compareAppIds = this.getAppID()
        .compareTo(other.getAppId());
    if (compareAppIds == 0) {
            return this.getAttemptId() – other.
        getAttemptId();
    }else {
        return compAppIds;
    }
}
```

```
public int compareTo (ApplicationId another) {
    int compAppIds = this.getAppID()
        .compareTo(another.getAppId());
    if (compAppIds == 0) {
        return this.getAttemptId() – another.
        getAttemptId();
    }else {
        return compAppIds;
    }
}
```

图 3 变量重命名示例图

条件重排序会对判断语句的条件参数进行位置调换, 并对关系运算符进行相应的调整. 改变判断语句条件参数的位置不影响判断语句的执行结果, 因此该方法不会改变整个程序的语义. 图 4 是一个条件重排序的示例.

我们使用 JavaTransformer (https://github.com/mdrafiqulrabin/JavaTransformer)实现上述两种方法.

JavaTransformer 是一个针对 Java 方法级别源代码文件的转换工具, 可以实现多种语义一致性变换方法. 图 5 展示了 JavaTransformer 在 Defects4J 上进行语义一致性变换后, 不同项目(Defects4J project)上所得到的变换体个数.

```
public void meReceive(context c, message e) {
    ChannelBuffer buf = (ChannelBuffer) e
                        .getMessage();
    XDR rsp = new XDR(buf.array());
    int state = rsp.readInt();
    if(state == Nfs3Status.NFS3_WRONG) {
        LOG.error("Create failed !");
        return;
}
```

```
public void meReceive(context c, message e) {
    ChannelBuffer buf = (ChannelBuffer) e
                        .getMessage();
    XDR rsp = new XDR(buf.array());
    int state = rsp.readInt();
    if(Nfs3Status.NFS3_WRONG == state) {
        LOG.error("Create failed !");
        return;
}
```

图 4 条件重排序示例图

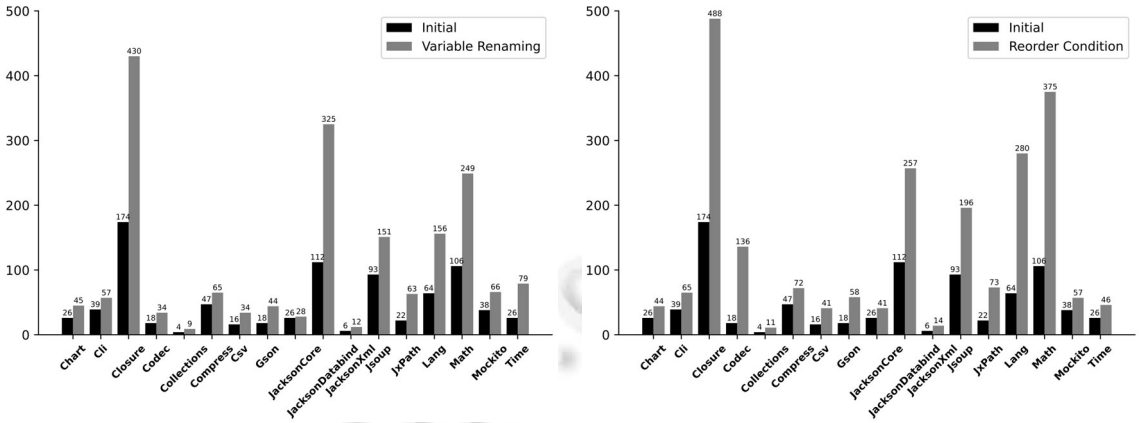


图 5 Defects4J 语义一致性变换的变体个数统计图

对于 RQ3, 我们分析了现有的神经程序修复系统数据集的构建方法, 根据其收集和过滤策略加以改进和补充, 提出了一套改进后的数据采集、过滤以及划分策略, 并在现有流行的数据集上依照此策略构建了一个可以用以缓解神经程序修复系统数据泄露问题的训练集. 如果神经程序修复系统在该数据集上进行重新训练, 可以在很大程度上避免数据泄露. 随后, 我们验证了该数据集的有效性.

4 研究结论

4.1 RQ1: 现有的神经程序修复系统是否存在数据泄露?

表 3 是 10 个神经程序修复系统在其进行性能评估的基准测试集上存在数据泄露的可修复缺陷. 为进一步探究这些存在数据泄露的可修复缺陷的类型及复杂程度, 我们选取了在 Defects4J 上存在泄露的缺陷, 并将这些缺陷进行如下分类: 符号错误、条件语句错误、循环语句错误、控制流语句错误、接口错误、数据类型声明错误、其他. 根据上述缺陷分类, 我们以修复该缺陷的复杂程度(即缺陷与补丁之间不相似的程度)以及可以被神经程序修复系统修复的比例作为参考依据, 对上述缺陷的复杂度进行了简单的划分. 其中, 复杂度 1–3 分别表示复杂度低、复杂度较高、复杂度高. 复杂度低的缺陷仅修改 1–2 个字符串便能被修复; 而复杂度高的缺陷可能存在与修复补丁完全不同的情况, 且超过 50%不能被神经程序修复系统所真正修复(存在不可被修复的变体).

(1) 符号错误

(1.1) 关系运算符错误: 复杂度 1

(1.2) 逻辑运算符错误: 复杂度 2

(1.3) 数学运算符错误: 复杂度 3

(2) 条件语句错误

(2.1) 条件语句错误: 复杂度 3

(2.2) 条件语句冗余: 复杂度 3

(2.3) 条件语句缺失: 复杂度 3

(3) 循环语句错误

(3.1) 循环语句错误: 复杂度 3

(3.2) 循环语句冗余: 复杂度 3

(3.3) 循环语句缺失: 复杂度 3

(4) 控制流语句错误

(4.1) 控制流语句错误: 复杂度 2

(4.2) 控制流语句冗余: 复杂度 1

(4.3) 控制流语句缺失: 复杂度 1

(5) 接口错误

(5.1) 接口参数错误: 复杂度 3

(5.2) 接口调用错误: 复杂度 3

(6) 数据类型声明错误: 复杂度 1

(7) 其他

表 3 神经程序修复系统在基准测试集上的数据泄露

神经程序修复系统	Defects4J (v1.2.0)
AlphaRepair	Chart1, Closure 86, Closure 109,125-126, Lang 24, Lang 51, Math 5, Math 57, Time 9
CIRCLE	Chart 1, Closure 86, Closure 92, Closure 93, Lang 29, Lang 59, Math 22, Math 75, Math 77, Math 79, Math 98
CoCoNuT	Chart 1, Closure 86, Closure 92, Lang 29, Lang 57, Lang 59, Math 22, Math 65, Math 77
CURE	Chart 1, Closure 86, Closure 92, Lang 29, Lang 57, Lang 59, Math 22, Math 65, Math 75, Math 79, Math 98
DEAR	Chart 19, Closure 40, Closure 86, Lang 27, Lang 43, Lang 47, Math 4, Math 22, Math 50, Math 57, Math 90, Time 13
DLFix	Closure 40, Closure 86, Closure 115, Closure 126, Lang 7, Lang 22, Lang 51, Math 50, Math 57
Recoder	Chart 26, Closure 2, Closure 7, Closure 21, Closure 33, Closure 46, Closure 109, Closure 115, Closure 118, Lang 55, Math 50, Math 65
RewardRepair	Chart 1, Chart 12, Closure 31, Closure 62, Closure 70, Closure 73, Closure 86, Closure 92, Closure 101, Math 11, Math 30, 33-34, Math 41, Math 57, Math 59, Math 70, Math 75, Math 80, Math 85, Math 94, Math 104,Lang 29, Lang 59
SequenceR	Closure 86, Math 57
TRANSFER	Closure 11, 21-22, Closure 31, Closure 40, Closure 46, Closure 115, Closure 126, Lang 10, Lang 61, Math 50, Math 65, Math 77
神经程序修复系统	Defects4J (v2.0.0)
AlphaRepair	Cli 8, Cli 17, Cli 28, Codec 2, JacksonCore 5, JacksonDatabind 1, Jsoup 57
CURE	Codec 4, JxPath 10
Recoder	Cli 5, Cli 28, Cli 32, Compress 27, Compress 31, Csv 5, Csv 9, Jsoup 24, Jsoup 68
RewardRepair	Cli 17, Cli 28, Codec 1-3, Codec 7, Codec 17, Collections 26, JacksonDatabind 102, Jsoup 24
TRANSFER	Compress 24, Compress 27
神经程序修复系统	QuixBugs
AlphaRepair	BREADTH_FIRST_SEARCH
CoCoNuT	BREADTH_FIRST_SEARCH
RewardRepair	FIND_FIRST_IN_SORTED
CURE	无
Recoder	无法获得
神经程序修复系统	Bugs.jar
RewardRepair	wicket_1b7afefc, commons-math_6d6649ef, wicket_2b1ce91d, jackrabbit-oak_459bd065, flink_dc78a747, wicket_e93fdd5a, maven_bef7fac6, wicket_7da4ad17, commons-math_83f18d52, commons-math_6dd3724b, camel_3f70d612, logging-log4j2_411dad65, wicket_3d8c9d75, commons-math_faf99727, accumulo_0cf2ff72, jackrabbit-oak_36e70bd7, commons-math_cedf0d27, flink_a56aad74, commons-math_3f645310, commons-math_0596e314, camel_dd0f74c0, jackrabbit-oak_5138a1e2, commons-math_f4a4464b, commons-math_ebadb558, commons-math_b6bf8f41commons-math_b6bf8f41
DLFix	无法获得

注: 表中的“无”表示未检测到数据泄露; “无法获得”表示该神经程序修复系统在基准测试集上具体的修复结果未公开, 因此无法获得具体存在泄露的缺陷名称或 ID. 另外, Recoder 在 IntroClassJava 上具体的修复结果未公开, 故不再列入表中

经统计后发现: 这些存在数据泄露的可修复缺陷, 约有 31%的缺陷类型为条件语句错误; 接口错误出现频率也较高, 约为 20%; 另外有 34%的缺陷类型比较复杂, 无法较好地归为一类, 故被分类为其他. 除去分类为其他的缺陷, 这些出现数据泄露的可修复缺陷的平均复杂度为 2.7. 由此能够看出: 这些存在数据泄露的可修复缺陷复杂度普遍较高, 修复难度较大.

结论 1: 现有的神经程序修复系统普遍存在数据泄露问题, 且存在数据泄露的可修复缺陷的复杂程度较高. 在调查研究的 10 个神经程序修复系统中, 仅有 CURE 未在 QuixBugs 基准测试集上发现数据泄露, 最严重的为 RewardRepair, 在 Bugs.jar 上有高达 27 处数据泄露. 上述 10 个神经程序修复系统在 Defects4J 上存在数据泄露的可修复缺陷的复杂程度为 2.7, 修复难度普遍较大. 对于不同的基准测试集, Defects4J (v1.2.0)上 RewardRepair 的数据泄露问题最为严重, 有 24 处数据泄露; Defects4J (v2.0.0)上同样是 RewardRepair 的数据泄露问题最为严重, 达到了 10 处; 在 QuixBugs 则出现了较为轻微的数据泄露, AlphaRepair、CoCoNuT 和 RewardRepair 均出现了 1 处; 而在 Bugs.jar 上, RewardRepair 的数据泄露达到了 27 处. 我们发现: 出现严重数据泄露问题的神经程序修复系统的数据集一般是没有过滤策略的, 因为常用的基准测试集在 GitHub 被其他项目使用到的概率很大, 这就导致在收集神经程序修复系统数据集时, 如果没有经过过滤, 就很有可能引入包含常用的基准测试集的项目; 此外, 由于绝大部分的代码具有相似性, 没有进行字符串匹配过滤的神经程序修复系统数据集中也很有可能包含常用的基准测试集内的缺陷-补丁对, 造成数据泄露.

4.2 RQ2: 数据泄露问题给神经程序修复系统带来了怎样的影响?

4.2.1 RQ2.1: 数据泄露问题是否会影响神经程序修复系统性能评估的真实性?

见表 4, 我们对本次实验所研究的 10 个神经程序修复系统的性能评估结果进行了排名, 并将出现数据泄露的可修复缺陷列为待观察缺陷, 判断这些缺陷在全部可修复缺陷中的占比, 以分析神经程序修复系统性能评估的真实性.

表 4 神经程序修复系统性能真实性和待观察缺陷及相关信息

基准测试集	神经程序修复系统	原始性能排名	待观察缺陷个数	忽略待观察缺陷排名	性能真实性	待观察缺陷排名(占比)
Defects4J (v1.2.0)	AlphaRepair	**1**	10	**1**	86.91%	**1 (13.51%)**
	CIRCLE	3	11	3	82.81%	3 (17.19%)
	CoCoNuT	8	9	7	79.55%	6 (20.45%)
	CURE	4	11	4	80.70%	4 (19.30%)
	DEAR	6	12	6	74.47%	9 (25.53%)
	DLFix	9	9	8	77.50%	7 (22.50%)
	Recoder	5	12	5	77.36%	8 (22.64%)
	RewardRepair	7	24	9	46.67%	10 (53.33%)
	SequenceR	10	**2**	10	85.71%	2 (14.29%)
	TRANSFER	2	13	2	80.60%	5 (19.40%)
Defects4J (v2.0.0)	AlphaRepair	2	7	2	80.56%	2 (19.44%)
	CURE	3	**2**	3	89.47%	**1 (10.53%)**
	Recoder	4	9	4	52.63%	5 (47.37%)
	RewardRepair	**1**	10	**1**	77.78%	3 (22.22%)
	TRANSFER	5	2	5	66.67%	4 (33.33%)
QuixBugs	AlphaRepair	**1**	1	**1**	96.43%	2 (3.57%)
	CoCoNuT	5	1	4	92.31%	4 (7.69%)
	CURE	2	**0**	2	100.00%	**1 (0.00%)**
	Recoder	4	无法获得	5	无法获得	无法获得
	RewardRepair	3	1	3	95.00%	3 (5.00%)
Bugs.jar	DLFix	**1**	无法获得	2	无法获得	无法获得
	RewardRepair	2	**27**	**1**	74.23%	1 (25.77%)

这里, 我们将性能评估的真实性定义为神经程序修复系统的原始性能减去待观察缺陷个数的差值, 与真实性能的比值, 如公式(6)所示.

$$PV = \frac{Perf_{\mathrm{O}} - Perf_{\mathrm{L}}}{Perf_{\mathrm{T}}} \tag{6}$$

其中, $Perf_O$ 表示原始性能, $Perf_L$ 表示待观察缺陷的个数, 而 $Perf_T$ 表示真正的性能. 在无法确认数据泄露是否影响了模型真正的性能的前提下, 我们认为 $Perf_T=Perf_O$.

结论 2.1: 数据泄露问题给神经程序修复系统性能评估的真实性造成了影响. 从表中可以看出, 所有的神经程序修复系统性能评估的真实性都受到了影响. 其中, RewardRepair 受影响最为严重, 其性能真实性在 Defects4Jv1.2.0 上下降了 24, 待观察缺陷占比 53.33%. 这说明, 神经程序修复系统的性能评估受到了数据泄露问题的影响. 因此, 研究人员在训练一个神经程序修复系统时, 应该对其训练数据进行过滤, 并且考虑在其修复结果上再次过滤, 以保证性能评估的公平性与准确性.

4.2.2 RQ2.2: 数据泄露问题是否会影响神经程序修复系统的鲁棒性?

我们在基准测试集 Defects4J 进行了神经程序修复系统的鲁棒性分析(Defects4J 是唯一被上述 5 个神经程序修复系统均作为基准测试的数据集). 如表 5 所示, 是神经程序修复系统中存在与不存在数据泄露的可修复的缺陷经过变换后不能被再次修复的比例. 其中, SequenceR 的 2 个存在数据泄露的可修复缺陷, 经过语义一致性变换后, 均不能再次被修复, 突变后差异百分比高达 100%; 而 RewardRepair 的鲁棒性也受到了较为严重的影响, 其差异突变体百分比和突变后差异百分比分别达到了 46.00%和 46.43%; Recoder 的鲁棒性受到的影响相对较小, 得益于 Recoder 在收集训练数据时对 Defects4J 进行了一些过滤策略(参考后文表 6)[36]. 数据泄露问题对神经程序修复系统造成鲁棒性影响, 是因为神经程序修复系统在进行训练时遇到了泄露的数据, 并利用记忆能力记住该数据. 当神经程序修复系统再次遇见相同的缺陷时, 直接输出被记住的修复代码(或称为过拟合现象).

表 5 神经程序修复系统的两组可修复缺陷在 Defects4J 上的 *PDM* 和 *PDA*

鲁棒性评估指标	神经程序修复系统	存在数据泄露的可修复缺陷		不存在数据泄露的可修复缺陷		全部缺陷
		变量重命名(VR)	条件重排序(RC)	变量重命名(VR)	条件重排序(RC)	VR&RC
差异突变体百分比(*PDM*)	CoCoNuT	44.74% (17/38)	43.75% (14/32)	10.47% (18/172)	9.67% (16/165)	15.97%
	CURE	14.29% (7/49)	9.30% (4/43)	9.72% (21/216)	7.25% (15/207)	9.13%
	Recoder	13.64% (6/44)	10.00% (4/40)	8.06% (15/186)	7.30% (13/178)	8.48%
	RewardRepair	54.71% (93/170)	30.14% (85/282)	46.00% (92/200)	24.65% (53/215)	**37.25%**
	SequenceR	94.18% (16/17)	29.41% (5/17)	38.64% (34/88)	27.17% (25/92)	**37.38%**
突变后差异百分比(*PDA*)	CoCoNuT	55.56% (5/9)	55.56% (5/9)	17.14% (6/35)	14.29% (5/35)	23.86%
	CURE	38.46% (5/13)	30.77% (4/13)	19.05% (12/63)	11.11% (7/63)	18.42%
	Recoder	33.33% (4/12)	25.00% (3/12)	17.07% (7/41)	9.76% (4/41)	16.98%
	RewardRepair	67.65% (23/34)	32.35% (11/34)	46.43% (26/56)	28.47% (16/56)	**42.22%**
	SequenceR	100.00% (2/2)	100.00% (2/2)	41.67% (5/12)	33.33% (4/12)	**46.43%**

结论 2.2: 数据泄露对神经程序修复系统的鲁棒性造成了影响. 从表 5 中可以明显看出: 存在数据泄露的缺陷部分, 其 *PDM* 和 *PDA* 的值均大于不存在数据泄露的缺陷部分. 另外, RewardRepair 和 SequenceR 的鲁棒性受到的影响最为明显. 我们分析后认为: RewardRepair 是因为其本身数据泄露问题较为严重, 导致鲁棒性受影响较为明显; 虽然 SequenceR 数据泄露的占比很低, 但因其模型简单, 且鲁棒性本身相对较差, 导致其抵抗数据泄露影响的能力较差. 因此, 我们得出以下结论: 在出现数据泄露的可修复缺陷部分, 神经程序修复系统的鲁棒性比在未出现数据泄露的可修复缺陷部分的表现差. 显而易见的, 这使得神经程序修复系统在整个基准测试集上的鲁棒性变差, 即数据泄露对神经程序修复系统的鲁棒性造成了负面影响.

4.3 RQ3: 如何避免或缓解数据泄露问题?

表 6 是 10 个神经程序修复系统数据集的相关信息. 可以看出: 绝大部分的神经程序修复系统的数据集都来源于 GitHub, 然而在大部分数据集的构建过程中, 都没有考虑过滤掉诸如 Defects4J 的常用基准测试, 和那些使用了常用基准测试集的项目, 更加没有一个数据集的过滤策略是直接对常用基准测试进行字符串匹配过滤, 因此导致了数据泄露的发生. 为此, 我们总结了一些有效的数据采集、过滤和划分策略, 并提出了一套新的数据采集和过滤策略, 以尽可能地避免数据泄露问题, 策略如下(结论 3).

(1) 以带有“bug”“error”“issue”和“fix”“repair”“solve”或“patch”等关键词的提交作为数据集来源。

(2) 丢弃那些是常用基准测试集克隆的项目, 或使用了常用基准测试集的项目。

(3) 收集完成的数据集要在 Defects4J、Bugs.jar 和 QuixBugs 等常用基准测试集上进行字符串匹配过滤。

(4) 训练集、验证集和测试集互相没有重叠, 且划分比例为(training/validation/testing)8/1/1.

表 6 神经程序修复系统数据集的相关信息

神经程序修复系统	数据集	数据来源	数据收集与过滤策略	划分比例
AlphaRepair	CodeSearchNet	GitHub	(1) 来自开源且没有 fork 的仓库; (2) 至少被 1 个其他的项目使用; (3) 删除项目中短于 3 行的函数; (4) 删除项目中函数名称具有子串“test”的函数	30/1/1
CIRCLE	CIRCLE (part of CoCoNuT)	–	–	–
CoCoNuT	CoCoNuT	GitHub GitLab	(1) 丢弃 2010 年之后的提交; (2) 只保留提交信息中含有“fix”, “bug”或“patch”等关键词的提交; (3) 使用 6 种提交信息的反模式过滤提交	150/0/1
CURE	CURE (CoCoNuT)	–	–	–
DEAR	CPatMiner	GitHub	(1) 项目至少有 100 个 star; (2) 项目至少有 10 个 committers; (3) 项目至少有 1 000 处有 Java 源代码改动的提交; (4) 对于每个项目,收集最近的 1 000 个有 Java 代码改动的提交	–
	BigFix	GitHub	(1) 方法检查过滤器: 用于检查缺陷是否在方法内部; (2) 注释检查过滤器: 用于检查缺陷是否在代码语句中, 而不是在注释中; (3) one-hunk 缺陷过滤器: 用于检查有缺陷的位置是否只有 1 个代码块, 修复后的代码是否也只有 1 个代码块; (4) 缺陷信息中包含 bug id, 且至少包含 1 个错误相关的关键词, 如: {error,bug,fix,issue, mistake, incorrect,fault,defect,flaw,type}	10/0/1
DLFix	BigFix	GitHub	–	–
Recoder	Recoder	GitHub	(1) 提交信息中至少包含以下 2 组中的 1 组: {fix, solve}, {bug,issue}; (2) 丢弃是一个克隆到 Defects4J 的项目或使用 Defects4J 的程序修复项目; (3) 根据 AST 的比较, 丢弃补丁修改方法与 Defects4J v1.2 或 v2.0 中的任何补丁相同的方法	4/0/1
RewardRepair	Megadif	GitHub	(1) 至少包含 1 个 Java 源文件的变化; (2) 在*.java 文件中有改变的代码少于 40 行	–
	CodRep	–	(1) 只保留源代码文件(Java files); (2) 舍弃仅有注释的修改(如//TODO with//Fixed 等); (3) 插入或删除的行不是空行, 也不是仅有空格的修改; (4) 整个文件中只有 1 处替换的代码行	–
	CoCoNuT	–	–	–
SequenceR	Bugs2Fix	GitHub	(1) 保留(“fix”或“solve”)和(“bug”“issue”“problem”或“error”)关键词的提交; (2) 抛弃非 Java 文件的提交; (3) 抛弃在修复缺陷的提交中创建的文件	8/1/1
	CodRep	–	–	–
TRANSFER	TRANSFER	GitHub	(1) 删除 4 个项目, 包括 Joda-Time、Closure Compiler、Apache commons-lang 和 Apache commons-math; (2) 提交信息包含“error” “bug”“fix” “issue”或“mistake”等关键词	10/0/1

我们从这些数据集中挑选满足上述策略(1)和策略(2)的作为原始数据集, 被选中的数据集有以下 3 个.

(1) CoCoNuT 数据集[40], 是从 GitHub 和 GitLab 上收集得到的. 该数据集分为两部分: 第 1 部分包含 2006 年之前提交的所有实例, 第 2 部分则包含 2010 年之前提交的实例. 而 2010 年后提交的实例则全部被丢弃. 这样做的目的是尽可能地避免 CoCoNuT 数据集与常用的基准测试集发生数据泄露.

(2) Bugs2Fix[19]数据集, 是从 GitHub 上收集得到的, 来自 2010 年 3 月–2017 年 10 月的缺陷修复提交. 通过使用 GitHub Compare API (https://developer.github.com/v3/repos/commits/#compare-two-commits) 对比提交前后的源代码获得数据.

(3) Recoder 数据集[36], 是从 GitHub 上收集得到的, 来自 2011 年 3 月–2018 年 3 月之间的缺陷修复提交,

经过过滤后, 被划分为训练集和验证集.

在此基础上, 我们对这 3 个数据集进行过滤和划分, 使之满足策略(3)和策略(4). 最终, 我们得到了一个纯净的数据集 Clean4J_Benchmark. 我们采用了一种严格的过滤策略: 如果数据集中某条数据 i 的缺陷代码与基准测试集中某条数据 j 的缺陷代码满足字符串匹配或子串匹配, 我们就把数据 i 从 Clean4J_Benchmark 删除; 同样的, 如果 i 的补丁代码与 j 的补丁代码满足字符串匹配或子串匹配, 我们也将数据 i 从 Clean4J_Benchmark 删除. 此外, 该数据集分为 3 部分: 第 1 部分 dataset_1 的数据格式为⟨buggy line+buggy context,fixed line⟩; 第 2 部分 dataset_2 的数据格式为⟨buggy method,fixed method⟩; 第 3 部分 dataset_3 的数据格式为⟨buggy class, fixed class⟩. 该数据集能够用于训练输入输出方式不同、训练方式不同的神经程序修复系统. 对于直接训练的神经程序修复系统, 如果其输入输出均为方法级别的代码, 我们可以采用 Clean4J_Benchmark 的 dataset_2 部分进行训练; 如果其输入为被分割成错误行+上下文的代码段, 而输出则为行级别的代码段, 我们可以采用 Clean4J_Benchmark 的 dataset_1 部分进行训练; 对于预训练-微调的神经程序修复系统或直接训练的神经程序修复系统, 我们可以采用 Clean4J_Benchmark 的 dataset_3 部分进行训练或微调. dataset_3 是类级别的且带有缺陷定位的缺陷-补丁对, 可供训练方式为预训练-微调的神经程序修复系统进行微调, 或抽象后供直接训练的神经程序修复系统进行训练. 表 7 是 Clean4J_Benchmark 数据集的相关信息.

表 7 数据集 Clean4J_Benchmark 的相关信息

数据集名称	数据集格式	划分比例	数据集大小
(Clean4J_Benchmark) dataset_1	⟨buggy line+buggy context,fixed line⟩	80%/10%/10%	5 834 720
(Clean4J_Benchmark) dataset_2	⟨buggy method,fixed method⟩	80%/10%/10%	579 967
(Clean4J_Benchmark) dataset_3	⟨buggy class,fixed class⟩ with bug location	80%/10%/10%	157 008

4.4 RQ4: 本文提出的数据集构建策略在多大程度上能缓解数据泄露?

我们选择 CoCoNuT、RewardRepair、SequenceR 和 Recoder, 在 Clean4J_Benchmark 上进行重新训练. 通过重新训练后, 与原始模型对比性能的真实性和鲁棒性, 以此验证 Clean4J_Benchmark 对于缓解数据泄露影响的效果, 同时验证本文提出的数据收集、过滤、划分策略的有效性. 在重新训练神经程序修复系统时, 我们尽可能地采用与原始论文中完全相同的各项变量、参数和方法, 包括训练神经网络使用的各项超参数、优化方法, 以及其他非训练相关变量. 但由于时间和算力成本, 我们统一将 4 个神经程序修复系统的候选补丁数设置为 200.

表 8 是重新训练后的神经程序修复系统的各项信息, 其中,“重合的缺陷”意为重新训练后, 神经程序修复系统不能再次修复的缺陷与其待观察缺陷相互重合的部分; 而“重合比例”则是指这部分重合的缺陷, 在全部待观察缺陷中所占的比例. 通过重新训练后模型不能再次修复的缺陷与待观察缺陷的对比, 我们发现, 有超过半数的待观察缺陷是无法被重新训练后的神经程序修复系统再次修复的. 我们认为: 这部分不能再次被修复的待观察缺陷是神经程序修复系统因数据泄露造成的对修复补丁的记忆, 而非对修复模式的理解, 不能被认为是模型本身的能力. 所以, 我们将这部分缺陷从模型的原始性能中除去, 作为模型真正的性能. 因此, 重新训练后, 模型的 $Perf_{\mathrm{T}}$ 下降, 而 $Perf_{\mathrm{O}}$ 和 $Perf_{\mathrm{L}}$ 保持不变, 因此, 神经程序修复系统的性能真实性将会提高.

表 8 重新训练后神经程序修复系统的性能真实性及相关信息

神经程序修复系统	训练集	基准测试集	性能真实性	重合的缺陷	重合比例
CoCoNuT	dataset_1	Defects4J (v1.2.0)	89.74%	Closure 86, Lang 57, Lang 59, Math 65, Math 77	55.56% (5/9)
RewardRepair	dataset_1+ dataset_2	Defects4J (v1.2.0)	65.63%	Math 34, Math 41, Math 70, Math 80, Math 94 等	54.17% (13/24)
SequenceR	dataset_3	Defects4J (v1.2.0)	92.31%	Math 57	50.00% (1/2)
Reocder	dataset_2	Defects4J (v1.2.0)	89.13%	Chart26, Closure 2, Closure 21, Closure 109, Closure 118, Lang 55, Math 50	58.33% (7/12)

注: RewardRepair 重合的缺陷具体为 Closure 31、Closure 70、Closure 92、Closure 101、Lang 59、Math 11、Math 33、Math 104、Math 34、Math 41、Math 70、Math 80、Math 94

表 9 则是重新训练后的神经程序修复系统的鲁棒性与原始鲁棒性的对比结果. 通过重新训练后模型的 *PDM*、*PDA* 与原始模型的平均 *PDM*、*PDA* 的对比, 我们发现, 重新训练后的模型, *PDM* 和 *PDA* 值绝大部分低于原始模型的平均值或与原始模型的平均值持平. 这表明重新训练后, 神经程序修复系统在基准测试上的鲁棒性表现要好于之前的原始模型; 同时也证明了, 数据泄露会对神经程序修复系统的鲁棒性产生负面影响.

表 9 重新训练后神经程序修复模型鲁棒性与原始模型鲁棒性的对比

神经程序修复系统	原始模型的平均 *PDM* (%)	重新训练后的 *PDM* (%)	原始模型的平均 *PDA* (%)	重新训练后的 *PDA* (%)
CoCoNuT	15.97	**16.52**	**23.86**	17.39
RewardRepair	**37.25**	31.14	**42.22**	35.48
SequenceR	**37.38**	10.65	**46.43**	32.14
Reocder	**8.48**	8.34	**16.98**	12.84

结论 4: 通过重新训练后的系统与原始系统的比较, 我们发现: 重新训练后, 神经程序修复系统的性能真实性有所提高; 与此同时, 鲁棒性也有所增强. 因而印证了本文提出的数据集构建策略对于缓解数据泄露的有效性.

5 有效性影响因素

本节讨论神经程序修复领域数据泄露问题实证研究的有效性影响因素. 有效性影响因素是指可能对实验有效性产生影响的因素.

首先, 我们没有考虑使用 Type-2、Type-3 或 Type-4 类型的匹配作为定义数据泄露的匹配标准. 我们根据已有研究, 选择使用 Type-1 匹配对神经程序修复领域的数据泄露进行定义, 也验证了该定义下的数据泄露会对神经程序修复系统产生显著的影响. 但如果训练数据与基准测试集产生了弱相关匹配(如 Type-2、Type-3 或 Type-4 类型的匹配)是否也会对神经程序修复系统产生轻微的影响, 我们暂时没有进一步的研究.

其次, 我们忽略了子串关系可能造成假阳性的情况. 我们在数据泄露的定义中包括了子串匹配, 即基准测试集上的缺陷-补丁对是训练集数据中缺陷-补丁对的子集这一情况. 而由于训练数据体量的庞大, 在基准测试上出现数据泄露的数据通常会在训练数据中反复出现多次, 经统计, 子串匹配只占其中的 8.4%(仅在基准测试集 Defects4J 上进行了统计), 因此, 我们认为造成假阳性的可能性十分微小.

另外, 我们尽可能遵循了神经程序修复模型原论文中的各项超参数. 对于在 Clean4J_Benchmark 上进行重新训练的神经程序修复系统, 在大部分情况下, 我们采用了同原论文中完全一致的超参数. 但由于时间关系, 在推理设置中, 我们统一将候选补丁数(beamsize)设置为 200, 这可能导致最终的修复结果与原模型的轻微差异. 但有研究表明: 在候选补丁数达到 150 左右时, 模型已经达到了最优性能的 90%[52]. 如 SequenceR 在重新训练后, 仍能在 Defects4J 上修复 14 个补丁, 除了 Math 57 是由数据泄露导致的不可再次被修复, 其余有 Chart_9、Chart_11、Lang_59 不可再次被修复, 但却多修复了 Closure_38、Closure_125、Math_2、Math_80. 因此我们认为: 将候选补丁数设置为 200, 对实验有效性造成影响的可能十分微小.

最后, 由于一些神经程序修复系统源代码的复现难度较大, 我们在 Clean4J_Benchmark 上的复现可能不够完备. 例如: CURE 在训练阶段的第 2 个 epoch 时, 会出现损失函数值变为 Nan 的情况, 在更换损失函数和调整学习率等尝试后依然无法解决. 因此, 我们没有使用 CURE 在 Clean4J_Benchmark 进行验证.

6 总结与展望

神经程序修复的性能目前已经优于传统的程序自动修复技术, 但现有的神经程序修复系统中仍然存在着低召回率、修复效率低下(CURE 的束搜索大小为 1 000, 导致对补丁进行验证时需要消耗大量时间)以及工业场景中的实用性仍待考量等诸多问题[53]. 本文对神经程序修复系统中的数据泄露问题进行了系统的研究, 发现其对神经程序修复系统的性能评估以及鲁棒性均造成不同程度的影响. 首先, 我们定义了神经程序修复系统中的数据泄露并进行了数据泄露检测, 发现现有的神经程序修复系统中均存在着不同程度的数据泄露; 随

后, 我们通过语义一致性变换实验, 验证了数据泄露问题影响了神经程序修复系统的鲁棒性; 最后, 我们提出了一个数据收集、过滤和划分策略, 并在此基础上构建了一个纯净的数据集, 并在 Clean4J_Benchmark 数据集上重新训练了神经程序修复模型, 验证了我们制定的数据收集、过滤和划分策略的有效性.

数据泄露问题对于神经程序修复系统的重要性不言而喻, 因此有必要对神经程序修复系统中的数据泄露问题继续进行研究. 后续研究可以从以下 3 个方面开展: 首先, 本文并未对基于大语言模型(参数量超过 10 亿)的神经程序修复系统进行研究, 由于大语言模型的训练数据十分庞大, 对于其数据泄露的定义以及检测都具有挑战性, 研究人员可以在这个场景下对神经程序修复系统的数据泄露问题继续研究, 并且设计更加完备的过滤策略以及收集更多数据, 构建一个更为大型的神经程序修复系统数据集, 供基于大语言模型的神经程序修复系统进行训练或微调; 其次, 研究人员可以对神经程序修复系统常用的基准测试集进行系统的实证研究, 并在此基础上设计一个与现有的神经程序修复系统训练数据没有重叠的基准测试, 保证该基准测试的纯净, 这样可以避免数据泄露、保证在该基准测试上进行评估的公平性; 最后, 研究人员可以继续深入研究数据泄露会对神经程序修复系统造成哪些负面影响, 并设法消除这些影响.

References:

[1] Monperrus M. Automatic software repair: A bibliography. ACM Computing Surveys, 2018, 51(1): 1−24.

[2] Goues CL, Pradel M, Roychoudhury A. Automated program repair. Communications of the ACM, 2019, 62(12): 56−65. [doi: 10.1145/3318162]

[3] Zhang Q, Fang C, Ma Y, *et al*. A survey of learning-based automated program repair. arXiv:2301.03270, 2023.

[4] Le Goues N, Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2012, 38(1): 54−72. [doi: 10.1109/TSE.2011.104]

[5] Martinez M, Monperrus M. ASTOR: A program repair library for Java. In: Proc. of the 25th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2016. 441−444. [doi: 10.1145/2931037.2948705]

[6] Yuan Y, Banzhaf W. ARJA: Automated repair of Java programs via multi-objective genetic programming. IEEE Trans. on Software Engineering, 2018, 46(10): 1040−1067. [doi: 10.1109/TSE.2018.2874648]

[7] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2018. 298−309.

[8] Durieux T, Monperrus M. DynaMoth: Dynamic code synthesis for automatic program repair. In: Proc. of the 11th Int'l Workshop on Automation of Software Test. 2016. 85−91. [doi: 10.1145/2896921.2896931]

[9] Xuan J, Martinez M, Demarco F, *et al*. Nopol: Automatic repair of conditional statement bugs in Java programs. IEEE Trans. on Software Engineering, 2016, 43(1): 34−55. [doi: 10.1109/TSE.2016.2560811]

[10] Martinez M, Monperrus M. Ultra-large repair search space with automatically mined templates: The cardumen mode of Astor. In: Proc. of the Int'l Symp. on Search Based Software Engineering (SSBSE). Springer, 2018. 65−86.

[11] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE). IEEE, 2016. 691−701. [doi: 10.1145/2884781.2884807]

[12] Liu K, Koyuncu A, Kim D, *et al*. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 1−12.

[13] Liu K, Koyuncu A, Kim D, *et al*. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2019. 31−42. [doi: 10.1145/3293882.3330577]

[14] Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). IEEE, 2013. 802−811. [doi: 10.1109/ICSE.2013.6606626]

[15] Hua JR, Zhang MS, Wang KY, Khurshid S. Towards practical program repair with on-demand candidate generation. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). ACM, 2018. 12−23. [doi: 10.1145/3180155.3180245]

[16] Huang K, Xu Z, Yang S, *et al*. A survey on automated program repair techniques. arXiv:2303.18184, 2023.

[17] Samala RK, Chan HP, Hadjiiski L, *et al*. Hazards of data leakage in machine learning: A study on classification of breast cancer using deep neural networks. In: Proc. of the Medical Imaging 2020: Computer-aided Diagnosis, Vol.11314. SPIE, 2020. 279−284. [doi: 10.1117/12.2549313]

[18] Tufano M, Pantiuchina J, Watson C, *et al*. On learning meaningful code changes via neural machine translation. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 25−36. [doi: 10.1109/ICSE.2019.00021]

[19] Tufano M, Watson C, Bavota G, *et al*. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans. on Software Engineering and Methodology, 2019, 28(4): 1−29. [doi: 10.1145/3340544]

[20] Jin M, Shahriar S, Tufano M, *et al*. InferFix: End-to-end program repair with llms. arXiv:2303.07263, 2023.

[21] Xia CS, Ding Y, Zhang L. Revisiting the plastic surgery hypothesis via large language models. arXiv:2303.10494, 2023.

[22] Xia CS, Zhang L. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In: Proc. of the 2022 30th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2022. 959−971. [doi: 10.1145/3540250.3549101]

[23] Chen X, Lin M, Schärli N, *et al*. Teaching large language models to self-debug. arXiv:2304.05128, 2023.

[24] Xia CS, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). IEEE, 2023.

[25] Cao J, Li M, Wen M, *et al*. A study on prompt design, advantages and limitations of ChatGPT for deep learning program repair. arXiv:2304.08191, 2023.

[26] Xia CS, Zhang L. Conversational automated program repair. arXiv:2301.13246, 2023.

[27] Bellon S, Koschke R, Antoniol G, *et al*. Comparison and evaluation of clone detection tools. IEEE Trans. on Software Engineering, 2007, 33(9): 577−591. [doi: 10.1109/TSE.2007.70725]

[28] Ji SL, Du TY, Deng SG, *et al*. Robustness certification research on deep learning models: A survey. Chinese Journal of Computers, 2022, 45(1): 190−206 (in Chinese with English abstract).

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

[30] Cheng Y, Tu ZP, Meng FD, *et al*. Towards robust neural machine translation. In: Proc. of the 56th Annual Meeting of the Association for Computational Linguistics (ACL), Vol. 1. ACL, 2018. 1756−1766. [doi: 10.18653/v1/P18-1163]

[31] Ge H, Zhong W, Li C, *et al*. RobustNPR: Evaluating the robustness of neural program repair models. Journal of Software: Evolution and Process, 2024, 36(4): e2586. [doi: 10.1002/smr.2586]

[32] Goyal C. Data leakage and its effect on the performance of an ML model. 2021. https://www.analyticsvidhya.com/blog/2021/07/data-leakage-and-its-effect-on-the-performance-of-an-ml-model/

[33] Elangovan A, He J, Verspoor K. Memorization vs. generalization: Quantifying data leakage in NLP performance evaluation. arXiv:2102.01818, 2021.

[34] Lewis P, Stenetorp P, Riedel S. Question and answer test-train overlap in open-domain question answering datasets. arXiv:2008.02637, 2020.

[35] Jiang N, Liu K, Lutellier T, *et al*. Impact of code language models on automated program repair. arXiv:2302.05020, 2023.

[36] Zhu Q, Sun Z, Xiao Y, *et al*. A syntax-guided edit decoder for neural program repair. In: Proc. of the 2021 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2021. 341−353.

[37] Chen ZM, Kommrusch SJ, Tufano M, Pouchet LN, Poshyvanyk D, Monperrus M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Trans. on Software Engineering, 2019, 47(9): 1943−1959. [doi: 10.1109/TSE.2019.2940179]

[38] Ye H, Martinez M, Monperrus M. Neural program repair with execution-based backpropagation. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). IEEE, 2022. 1506−1518. [doi: 10.1145/3510003.3510222]

[39] Meng X, Wang X, Zhang H, *et al*. Improving fault localization and program repair with deep semantic features and transferred knowledge. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). IEEE, 2022. 1169−1180.

[40] Lutellier T, Pham HV, Pang L, *et al*. Coconut: Combining context-aware neural translation models using ensemble for program repair. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2020. 101−114.

[41] Jiang N, Lutellier T, Tan L. Cure: Code-aware neural machine translation for automatic program repair. In: Proc. of the 43rd Int'l Conf. on Software Engineering (ICSE). IEEE, 2021. 1161−1173. [doi: 10.1109/ICSE43902.2021.00107]

[42] Li Y, Wang S, Nguyen TN. DLFix: Context-based code transformation learning for automated program repair. In: Proc. of the 42nd Int'l Conf. on Software Engineering (ICSE). IEEE, 2020. 602−614. [doi: 10.1145/3377811.3380345]

[43] Li Y, Wang S, Nguyen TN. DEAR: A novel deep learning-based approach for automated program repair. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). IEEE, 2022. 511−523. [doi: 10.1145/3510003.3510177]

[44] Yuan W, Zhang Q, He T, *et al*. CIRCLE: Continual repair across programming languages. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2022. 678−690. [doi: 10.1145/3533767.3534219]

[45] Feng Z, Guo D, Tang D, *et al*. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155, 2020.

[46] Raffel C, Shazeer N, Roberts A, *et al*. Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research, 2020, 21(1): 5485−5551. [doi: 10.5555/3455716.3455856]

[47] Saha RK, Lyu Y, Lam W, *et al*. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2018. 10−13. [doi: 10.1145/3196398.3196473]

[48] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 23rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2014. 437−440.

[49] Durieux T, Monperrus M. IntroClassJava: A benchmark of 297 small and buggy Java programs. 2016. https://hal.science/hal-01272126/

[50] Lin D, Koppel J, Chen A, *et al*. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge. In: Proc. Companion of the 2017 ACM SIGPLAN Int'l Conf. on Systems, Programming, Languages, and Applications: Software for Humanity. ACM, 2017. 55−56.

[51] Lawler R. How do you hire great engineers? Give them a challenge. 2012. https://www.eejournal.com/fresh_bytes/how-do-you-hire-great-engineers-give-them-a-challenge/

[52] Zhong W, Ge H, Ai H, *et al*. StandUp4NPR: Standardizing setup for empirically comparing neural program repair systems. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2022. 1−13.

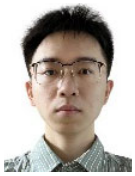
[53] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2665−2690 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

附中文参考文献:

[28] 纪守领, 杜天宇, 邓水光, 等. 深度学习模型鲁棒性研究综述. 计算机学报, 2022, 45(1): 190−206.

[53] 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665−2690. http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

李卿源(2000－), 男, 硕士生, CCF 学生会员, 主要研究领域为软件工程, 自然语言处理.



钟文康(1997－), 男, 博士生, 主要研究领域为软件工程, 自然语言处理, 程序自动修复.

李传艺(1991－), 男, 博士, 准聘助理教授, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 业务过程管理, 自然语言处理.

葛季栋(1978－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为自然语言处理, 智能软件工程, 分布式计算, 边缘计算, 服务计算, 业务过程管理.

骆斌(1967－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为分布式计算, 边缘计算, 自然语言处理, 智能软件工程.