软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(5):2289−2306 [doi: 10.13328/j.cnki.jos.006961]

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

基于对抗生成网络的缺陷定位模型域数据增强方法*

张 卓[1], 雷 晏[2], 毛晓光[3], 薛建新[4], 常 曦[4]

[1](广州商学院 信息技术与工程学院, 广东 广州 510700)

[2](重庆大学 大数据与软件学院, 重庆 401331)

[3](国防科技大学 计算机学院, 湖南 长沙 410073)

[4](上海第二工业大学 计算机与信息工程学院, 上海 200127)

通信作者: 雷晏, E-mail: yanlei@cqu.edu.cn

摘 要: 缺陷定位获取并分析测试用例集的运行信息, 从而度量出各个语句为缺陷的可疑性. 测试用例集由输入域数据构建, 包含成功测试用例和失败测试用例两种类型. 由于失败测试用例在输入域分布不规律且比例很低, 失败测试用例数量往往远少于成功测试用例数量. 已有研究表明, 少量失败测试用例会导致测试用例集出现类别不平衡问题, 严重影响着缺陷定位有效性. 为了解决这个问题, 提出基于对抗生成网络的缺陷定位模型域数据增强方法. 该方法基于模型域 (即缺陷定位频谱信息) 而非传统输入域 (即程序输入), 利用对抗生成网络合成覆盖最小可疑集合的模型域失败测试用例, 从模型域上解决类别不平衡的问题. 实验结果表明, 所提方法大幅提升了 11 种典型缺陷定位方法的效能.

关键词: 缺陷定位; 测试用例; 对抗生成网络; 数据增强; 可疑值

中图法分类号: TP311

中文引用格式: 张卓, 雷晏, 毛晓光, 薛建新, 常曦. 基于对抗生成网络的缺陷定位模型域数据增强方法. 软件学报, 2024, 35(5): 2289–2306. http://www.jos.org.cn/1000-9825/6961.htm

英文引用格式: Zhang Z, Lei Y, Mao XG, Xue JX, Chang X. Model-domain Data Augmentation Using Generative Adversarial Network for Fault Localization. Ruan Jian Xue Bao/Journal of Software, 2024, 35(5): 2289–2306 (in Chinese). http://www.jos.org.cn/1000-9825/6961.htm

Model-domain Data Augmentation Using Generative Adversarial Network for Fault Localization

ZHANG Zhuo[1], LEI Yan[2], MAO Xiao-Guang[3], XUE Jian-Xin[4], CHANG Xi[4]

[1](School of Information Technology & Engineering, Guangzhou College of Commerce, Guangzhou 510700, China)

[2](School of Big Data & Software Engineering, Chongqing University, Chongqing 401331, China)

[3](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[4](School of Computer and Information Engineering, Shanghai Polytechnic University, Shanghai 200127, China)

Abstract: Fault localization collects and analyzes the runtime information of test case sets to evaluate the suspiciousness of each statement of being faulty. Test case sets are constructed by the data from the input domain and have two types, i.e., passing test cases and failing ones. Since failing test cases generally account for a very small portion of the input domain, and their distribution is usually random, the number of failing test cases is much fewer than that of passing ones. Previous work has shown that the lack of failing test cases leads to a class-imbalanced problem of test case sets, which severely hampers fault localization effectiveness. To address this problem, this study proposes a model-domain data augmentation approach using generative adversarial network for fault localization. Based on the model domain (i.e., spectrum information of fault localization) rather than the traditional input domain (i.e., program input), this approach uses the generative adversarial network to synthesize the model-domain failing test cases covering the minimum suspicious set, so as to address

* 基金项目: 国家自然科学基金 (62272072); 中央高校基本科研业务费 (2022CDJDX-005)
收稿时间: 2022-01-07; 修改时间: 2022-11-17, 2023-02-13, 2023-04-06; 采用时间: 2023-05-16; jos 在线出版时间: 2023-08-30
CNKI 网络首发时间: 2023-08-31

the class-imbalanced problem from the model domain. The experimental results show that the proposed approach significantly improves the effectiveness of 12 representative fault localization approaches.

Key words: fault localization; test case; generative adversarial network (GAN); data augmentation; suspiciousness

在软件开发和维护过程中, 缺陷定位和修复对于软件开发人员来说是最耗时耗力的活动之一[1]. 为了降低开发成本, 研究人员提出了许多缺陷定位方法[1−14], 从而帮助开发人员定位程序中的缺陷. 其中, 基于频谱的缺陷定位方法 (spectrum-based fault localization, SBFL)[2,7,15]和基于深度学习的缺陷定位方法 (deep-learning-based fault localization, DLFL)[8−13]是被广泛研究和使用的两大类缺陷定位方法.

图 1 展示了 SBFL 和 DLFL 的缺陷定位过程. 假设程序 P 包含 N 个语句, 测试用例集 T 包含 M 个测试用例, $T=\{t_1, t_2,\ldots, t_M\}$. 我们在程序 P 上运行测试用例集 T 后可以获取语句覆盖信息和测试结果, 构造模型域测试用例集 (即频谱信息). 每执行一个输入域测试用例会形成一个模型域测试用例, 它包括语句是否被执行的信息以及测试用例运行结果. 当测试用例集中的所有测试用例运行完毕后, 所有的模型域测试用例构成一个覆盖矩阵. 矩阵中的每一行代表运行一个模型域测试用例的信息; 每一列代表程序 P 中的一个语句. 每一行包含若干元素, 元素值为 1 表示对应的语句被执行, 0 表示未被执行. 矩阵最右侧的结果为 1 表示该测试用例为成功测试用例, 为 0 表示为失败测试用例. 最后, 模型域测试用例集信息输入到可疑性评估算法, 计算各个语句为缺陷的可疑值, 输出可疑值降序排列的语句列表. 其中, SBFL 和 DLFL 的可疑性评估算法分别为可疑值计算公式[4,16]或深度学习模型[8−13].

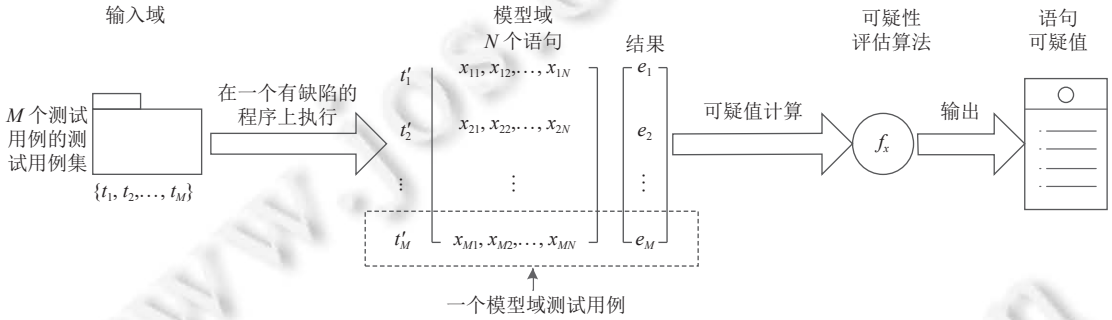

图 1 SBFL 和 DLFL 的缺陷定位流程

在缺陷定位领域中, 测试用例包括输入域的测试用例和模型域的测试用例. 前者指的是一组为某目标而编制的一组测试输入、执行条件以及预期结果; 而后者指的是一个向量, 它记录了相对应输入域测试用例在执行过之后的程序覆盖信息以及测试用例执行结果[16]. 输入域的测试用例集是缺陷定位的重要组成. 它包括两类: 成功测试用例和失败测试用例. 研究[17,18]表明具有对称数量的成功测试用例和失败测试用例是缺陷定位高效启动及运行的关键, 类别不平衡的测试用例集严重影响缺陷定位有效性. 然而, 对于一个缺陷来说, 由于该缺陷导致程序失效的输入通常只占所有输入很小比例且它们的分布规律性弱, 直接从输入域生成失败的测试用例非常困难. 因此, 在缺陷定位领域, 现有测试用例生成方法中 (例如文献 [17,19]) 主要优化或生成成功测试用例, 很少且很难直接生成失败测试用例. 后续研究[9,18,20−22]进一步表明失败测试用例往往对定位到程序缺陷是有益的. 因此, 为了提高缺陷定位有效性, 迫切需要解决测试用例类别不平衡问题.

已有的研究[8−13,22]表明, SBFL 和 DLFL 均基于一个共同的假设, 一个语句在失败 (成功) 测试用例中被覆盖, 则它的可疑值上升 (下降), 即覆盖了缺陷语句的失败测试用例对于缺陷定位是有益的[22]. 这意味着如果我们合成的模型域失败测试用例覆盖了缺陷语句, 那么它对缺陷定位是有益的. 而机器学习技术可以利用语句覆盖信息作为训练样本, 以测试用例结果作为标签, 学习一个能够反映语句覆盖信息和测试用例结果之间关联关系的模型. 之后通过构建虚拟测试用例来评估每个语句的可疑值. 在机器学习领域, 数据增强方法被广泛用来解决训练数据集类别不平衡问题[23], 合成更多的少数类样本可以提高模型的精度[24−27]. 因此, 本文提出基于对抗生成网络的缺陷定位模型域数据增强方法 GAN-Aug, 旨在机器之间创造一种竞争, 以模仿生物进化的形式在对抗中进步, 又在进步后继续对抗, 从而生成式网络就可以根据已有的全部模型域失败测试用例得到覆盖缺陷语句并且越来越逼近真

实模型域失败测试用例的新用例, 以此获得平衡的数据集来提升缺陷定位效能. 具体来说, GAN-Aug 使用对抗生成网络 (generative adversarial network, GAN) 来合成模型域失败测试用例, 覆盖现有模型域失败测试用例的共同特征 (即最小可疑集合[22]), 直至构造出类别平衡的模型域测试用例集, 即模型域失败测试用例和成功测试用例数量相同. 为了验证 GAN-Aug 的有效性, 本文应用 GAN-Aug 到 11 种典型缺陷定位技术[4,8,10,12,28,29], 缺陷定位效能得到大幅提升. 本文对方法的实现进行了在线公布 (https://github.com/zz8477/GAN_Aug).

本文第 1 节介绍背景知识. 第 2 节描述基于对抗生成网络的缺陷定位模型域数据增强方法 GAN-Aug. 第 3 节给出实验及结果. 第 4 节对比相关工作. 第 5 节进行总结.

1 相关背景

本节将介绍缺陷定位领域广泛使用的频谱信息, 以及实验中用到的 11 种缺陷定位方法和对抗生成网络.

(1) 频谱信息

图 2 显示了缺陷定位领域广泛使用的频谱信息, 常被称为程序谱模型[16]. 假设缺陷程序 P 有 N 个语句$\{S_1, S_2,\ldots, S_N\}$, 输入域数据组成的测试用例集 T 执行了程序 P. 其中 T 包含 M 个输入域测试用例, $T=\{t_1^{\text{input}}, t_2^{\text{input}},\ldots, t_M^{\text{input}}\}$. 输入域测试用例 t_i^{input} 定义如下.

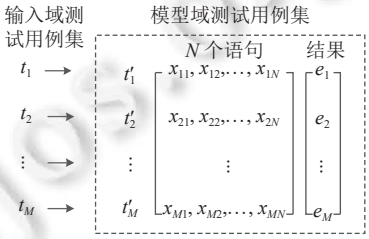

图 2 频谱信息

t_i^{input}: 测试用例输入 (即输入域的第 i 测试用例) 和测试预言 (即预期输出).

缺陷定位执行每个输入域测试用例, 并收集它们的语句覆盖信息 (语句执行或者未执行), 检查测试用例输出和测试预言是否一致, 一致则为成功的测试用例, 不一致为失败的测试用例. 根据这些信息, 缺陷定位定义并构造了模型域测试用例, 以此表示语句覆盖信息和输入域测试用例的测试结果. 模型域测试用例 t_i^{model} 定义如下.

t_i^{model}: $(N+1)$ 个元素并表示为向量 $[x_{i1}, x_{i2},\ldots, x_{iN}, e_i]$, 它记录了每个语句的覆盖信息以及输入域测试用例 t_i^{input} 的测试结果.

在 t_i^{model} 中, 元素 $x_{ij}=1$ 表示语句 S_j 被输入域的测试用例 t_i^{input} 执行; $x_{ij}=0$ 表示语句 S_j 没有被输入域的测试用例 t_i^{input} 执行. $e_i=1$ 表示输入域测试用例 t_i^{input} 为失败的测试用例; $e_i=0$ 表示输入域测试用例 t_i^{input} 为成功的测试用例. M 个 e_i ($i\in\{1, 2,\ldots, M\}$) 常表示为结果向量 (error vector) $e=[e_1, e_2,\ldots, e_M]$, $i\in\{1, 2,\ldots, M\}$.

M 个模型域测试用例组成了频谱信息 ($M\times(N+1)$ 矩阵). 它记录了运行完输入域测试用例集 T 后每个语句的覆盖信息 (语句被执行或未被执行) 和每个输入域测试用例的测试结果 (成功或失败). 因此, 频谱信息可以定义如下.

频谱信息: $M\times(N+1)$ 矩阵, 其中第 i 行是第 i 个模型域测试用例以及其测试结果, 即 $t_i^{\text{model}}=[x_{i1}, x_{i2},\ldots, x_{iN}, e_i]$. 需要说明的是, 频谱信息的每行向量中的元素可以是 0 或 1, 也可以是一个十进制数[7,14].

(2) 典型缺陷定位技术

缺陷定位技术将频谱信息作为输入, 分析频谱信息并计算出语句为缺陷的可疑值. 其中, 基于深度学习的缺陷定位技术 (DLFL) 和基于程序谱的缺陷定位技术 (SBFL) 是最为典型的缺陷定位技术. DLFL 和 SBFL 分别利用深度神经网络[8−13]和可疑值计算公式[4,28]来评估语句的可疑值. 图 3 展示了 DLFL 和 SBFL 的总体结构.

DLFL 包括一个输入层, 一个深度学习组件, 若干隐藏层和一个输出层. 在输入层, DLFL 使用了图 2 所示的频谱信息作为输入. 具体来说, $M\times N$ 矩阵中的 h 行的向量作为深度学习模型的输入, 它们对应的结果向量 (error vector) 中的 h 个结果作为标签. 它们代表了从第 i 行开始的 h 个测试用例的语句覆盖信息以及相应测试用例的测

试结果 ($i \in \{1, 1+h, 1+2h, \ldots, 1+(\lfloor M/h \rfloor+1)\times h\}$). 在深度学习组件中, 不同 DLFL 技术采用不同的神经网络. 典型的有 MLP-FL (多层感知机)[10], CNN-FL (卷积神经网络)[8], BiLSTM-FL (双向长短记忆网络)[10], FLUCCS (排序学习)[29]和 DeepFL (多种神经网络)[12]. 在输出层, 模型使用了 Sigmoid 函数[8]来将模型的输出结果映射到 0–1 之间的一个值. 定义好模型之后, DLFL 使用反向传播算法 (back propagation algorithm)[8]来更新网络模型的参数, 目标是最小化训练输出 y 和结果向量 e 之间的差值. 最后, DLFL 反复迭代训练网络并根据训练好的模型计算出各个语句的可疑值.

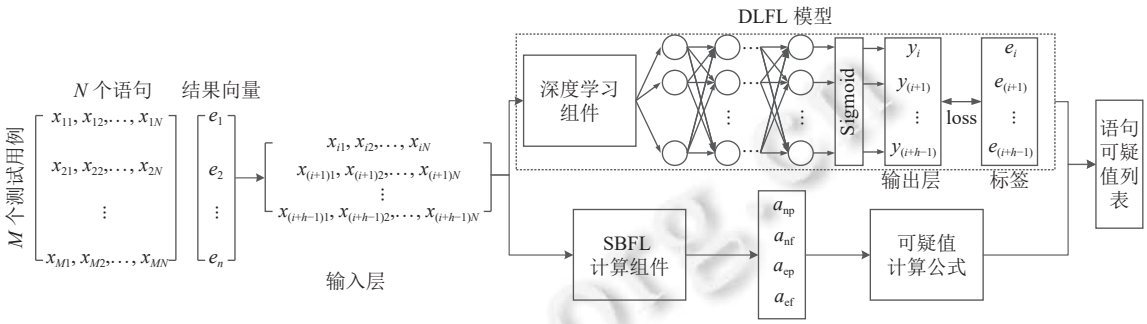

图 3 SBFL 和 DLFL 的总体结构

SBFL[16]根据频谱信息为每个语句定义 4 个变参数, 构建出不同类型的可疑值计算公式, 以此计算出各个语句的可疑值. 4 个参数定义如下:

$$\begin{cases} a_{\text{np}}(s_j) = \left|\{i \mid x_{ij} = 0 \wedge e_i = 0\}\right|, a_{\text{nf}}(s_j) = \left|\{i \mid x_{ij} = 0 \wedge e_i = 1\}\right| \\ a_{\text{ep}}(s_j) = \left|\{i \mid x_{ij} = 1 \wedge e_i = 0\}\right|, a_{\text{ef}}(s_j) = \left|\{i \mid x_{ij} = 1 \wedge e_i = 1\}\right| \end{cases} \tag{1}$$

公式 (1) 展示了根据频谱信息计算出语句 s_j 的 4 个参数 a_{np}, a_{nf}, a_{ep}, a_{ef}. 其中, a_{np} 表示未执行该语句的成功测试用例数, a_{nf} 表示未执行该语句的失败测试用例数, a_{ep} 表示执行该语句的成功测试用例数, a_{ef} 表示执行该语句的失败测试用例数. Xie 等人[28]对 SBFL 可疑值计算公式进行了分析验证, 得出 5 组最优的 SBFL 公式: ER1′, ER5, GP02, GP03 和 GP19. 后续实验[4]进一步表明 DStar 和 Ochiai 也是最优的可疑值计算公式. 表 1 列出了这 7 种类型的可疑值计算公式.

表 1 最优 SBFL 公式

Name		Formulas	Name	Formulas
ER1′	Naish1	$\begin{cases} -1, \text{ if } a_{\text{ne}} > 0 \\ a_{\text{np}}, \text{ if } a_{\text{ne}} \leqslant 0 \end{cases}$	GP02	$2\left(a_{\text{ef}} + \sqrt{a_{\text{np}}}\right) + \sqrt{a_{\text{ep}}}$
	Naish2	$a_{\text{ef}} - \dfrac{a_{\text{ep}}}{a_{\text{ep}} + a_{\text{np}} + 1}$	GP03	$\sqrt{\lvert a_{\text{ef}}^2 - \sqrt{a_{\text{ep}}}\rvert}$
	GP13	$a_{\text{ef}}\left(1 + \dfrac{a_{\text{ep}}}{2a_{\text{ep}} + a_{\text{ef}}}\right)$	GP19	$a_{\text{ef}}\sqrt{\lvert a_{\text{ep}} - a_{\text{ef}} + a_{\text{nf}} - a_{\text{np}}\rvert}$
ER5	Wong1	a_{ef}	DStar (D*)	$\dfrac{a_{\text{ef}}^*}{a_{\text{nf}} + a_{\text{ep}}}$
	Russel&Rao	$\dfrac{a_{\text{ef}}}{a_{\text{ef}} + a_{\text{nf}} + a_{\text{ep}} + a_{\text{np}}}$	Ochiai	$\dfrac{a_{\text{ef}}}{\sqrt{(a_{\text{ef}} + a_{\text{nf}})(a_{\text{ef}} + a_{\text{ep}})}}$
	Binary	$\begin{cases} 0, \text{ if } a_{\text{ne}} > 0 \\ 1, \text{ if } a_{\text{ne}} \leqslant 0 \end{cases}$		

(3) 对抗生成网络 (GAN)

在深度学习领域, 大多数人工智能算法为监督学习, 而无监督学习相对来说仍是未解决的研究领域[30]. 生成模型 (generative model, GM) 是近年来深度学习中迅速发展的研究领域[31], 为无监督学习研究开辟了新的方向. 对抗生成网络 (generative adversarial network, GAN) 是其中非常重要的一类. GAN 采取有监督的学习方法, 以生成虚假或合成的数据来实现无监督学习的应用[32]. GAN 定义两个网络, 一个为生成网络 G; 另一个为判别网络 D. D 是一个二进制分类器, 用于学习区分真实数据和生成数据. 而 G 这个生成网络首先接收一个随机的噪声, 之后通过噪声生成一个伪造的数据, 用来混淆 D[30]. 在训练过程中, 生成网络 G 的目标就是尽量生成逼近真实数据的伪造数据去欺骗判别网络 D. 而 D 的目标就是尽量把 G 生成的伪造数据和真实的数据区分开来. 因此, G 和 D 就构成了一个动态的“博弈过程”. 当训练结束时, 对于判别网络 D 来说, 希望 D 难以区分出哪个是真实的数据, 哪个是生成网络 G 生成的数据; 而对于生成网络 G 来说, 希望 G 可以生成出能够以假乱真的伪造数据.

在实际应用中, GAN 已经成功应用于许多研究领域, 诸如手写字体生成, 动画人物生成, 图像融合, 面部老化, 文本合成, 人体姿势合成, 视觉显著性预测, 目标检测, 3D 图像合成, 医学应用, 面部化妆转移, 地标检测, 图像超分辨率, 纹理合成, 语言和语音合成和音乐生成等[30]. 在软件工程领域, GAN 被成功运用于程序漏洞修复[33], 模拟对可编程网络的攻击[34], 复杂软件系统的参数配置[35], 软件老化问题的应对[36]和代码生成[37]等. GAN 被用于本文提出的 GAN-Aug 是首次将 GAN 用于软件工程领域中的缺陷定位模型域失败测试用例合成.

图 4 展示了 GAN 结构框架. z 是噪声数据, 当随机噪声输入到生成网络 G 后, 会输出一个生成数据. GAN 试图用这个生成的数据冒充真实的数据去欺骗判别网络 D, 而判别网络 D 负责判别生成网络输出的生成数据是真实数据还是假的生成数据. 在 GAN 训练判别网络 D 的时候, 上一轮迭代生成网络 G 输出的生成数据和真实数据拼接起来作为输入给判别网络 D, 这时判别网络 D 会把标签 1 给真实数据, 而把标签 0 给生成数据. 而拼接的数据通过判别网络时会有一个输出 score, 判别网络 D 会计算输出 score 和标签 1 或 0 的差值 loss, 之后就可以根据这个差值 loss 进行反向传播训练网络参数. 当训练生成网络 G 的时候, 判别网络 D 的参数为不可训练的, 这时生成网络 G 和判别网络 D 可以看作为一个整体. 当噪声数据输入到生成网络 G 的时候, G 会输出一个生成数据, 而这时判别网络 D 会对生成数据进行打分, 得到一个分值 score, 最后生成网络 G 会计算 score 和 1 之间的差值作为 loss 进行反向传播训练生成网络 G 的参数. 因此, 生成网络 G 和判别网络 D 就构成了一个“动态博弈”的过程.

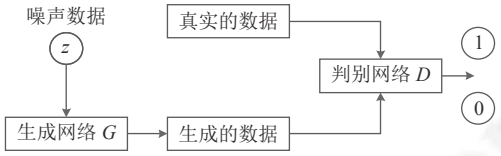

图 4 对抗生成网络架构

2 GAN-Aug

(1) 方法介绍

GAN-Aug 的基本思想是在保持输入域测试用例不变的基础上, 利用对抗生成网络 GAN 合成模型域失败测试用例, 获取类别平衡的模型域测试用例集, 以此提升缺陷定位效能. 合成的模型域失败测试用例, 与第 1 节定义的模型域测试用例 t_i^{model} 具有相同结构, 即标签为失败的模型域测试用例.

数据生成: GAN 的优点是超越了传统神经网络分类和特征提取的功能, 能够按照已有模型域失败测试用例的特点生成新的用例. 判别网络 D 和生成网络 G 在对抗中进步, 在进步后又继续对抗, 由生成网络 G 得到的模型域失败测试用例也就越来越完美, 逼近真实的模型域失败测试用例. 生成网络 G 从给定噪声中产生合成数据, 想方设法从随机噪声中, 模拟真实模型域失败测试用例的潜在分布, 试图产生更逼近真实的数据. 判别网络 D 分辨生成器的输出和真实数据. 试图更完美地分辨真实数据和生成数据. 生成网络 G 和判别网络 D 均采用深度神经网络实现, 建立的数据生成模型使判别网络 D 能够更精确地模拟出真实模型域失败测试用例的数据分布. 网络训练等

效于目标函数的极大极小问题.

因此, 本文提出 GAN-Aug, 以所有模型域失败测试用例为样本, 通过判别网络 D 和生成网络 G 生成一个新向量. 这个向量和模型域失败测试用例拥有较高的相似度, 且同时又保留了差异性. 每个语句对应的元素不再是 1 或者 0, 取而代之的是一个 0 到 1 之间的值. 在训练时 GAN-Aug 对所有模型域失败测试用例进行学习, 新生成的模型域测试用例会包含所有模型域失败测试用例的共同特征, 即最小可疑集合. 最后, GAN-Aug 将生成的新模型域失败测试用例加入原有的模型域测试用例集, 以此组成了新的模型域测试用例集, 解决类别不平衡问题.

$$\min_G \max_D V(D,G) = E_{x\sim P_{\text{data}}(x)}[\log D(x)] + E_{z\sim P_z(z)}[\log(1-D(G(z)))] \tag{2}$$

图 5 显示了 GAN-Aug 架构. 这里继续使用第 1 节假设的程序 P 和测试用例集 T. 语句覆盖情况与测试结果组成 $M\times(N+1)$ 的矩阵, 左边 $M\times N$ 矩阵表示语句执行情况, 最右侧是结果向量 errors. GAN-Aug 包括一个生成网络 G 和一个判别网络 D, G 和 D 均为一个包含若干隐藏层的多层感知机 (multilayer perceptron, MLP), MLP 的隐藏层个数根据输入数据的规模设置为 3–5 层, 层间的激活函数为 ReLU[10], 最后一个线性层连接一个 Sigmoid 函数[10], 使输出为一个 0–1 之间的值. 首先是训练 D (判别器 Discriminator). 选取 N 维向量 $[y_1, y_2,\ldots, y_N]$, 经过噪声 noise 处理后输入到 G (生成器 Generator) 中得到生成数据 $[z_1, z_2,\ldots, z_N]$. 之后选取模型域原始失败测试用例 (即 $[x_{i1}, x_{i2},..., x_{iN}]$, $i\in\{1, 2,..., M\}$ 且 e_i=1) 作为真实数据, 和生成数据 $[z_1, z_2,\ldots, z_N]$ 拼接起来一起输入到 D 中. D 把标签 1 给真实数据 $[x_{i1}, x_{i2},\ldots, x_{iN}]$, 而把标签 0 给生成数据 $[z_1, z_2,\ldots, z_N]$. D 的输出 score 和标签的差值作为 loss 反向传播进行训练. 第二是训练 G, 固定 D 的参数不变, 将 D 和 G 看成一个整体, 将经过噪声处理的 $[y_1, y_2,\ldots, y_N]$ 作为 G 的输入输出生成数据 $[z_1, z_2,\ldots, z_N]$, 用参数固定的判别器 D 进行打分, 输出 score 和标签 1 的差值作为 loss 反向传播从而训练 G. 公式 (2) 为生成器 G 和判别器 D 互相博弈的训练过程. 在公式 (2) 中, $P_{\text{data}(x)}$ 为真实数据 $[x_{i1}, x_{i2},\ldots, x_{iN}]$ 的分布, $P_z(z)$ 为生成数据 $[z_1, z_2,\ldots, z_N]$ 的分布, 一开始 G 较弱, D 很容易能判别出真实数据与生成数据. 随着训练 G 逐渐增强, D 慢慢地区分不出真实数据和判别数据, 最后输出 score 只能是 0.5, 即判别为真实数据和生成数据的概率相等. 整个过程为一个最大最小的博弈过程. D 最大化判别出生成数据 $[z_1, z_2,\ldots, z_N]$, 而 G 也在不断努力提升自己在生成数据时模仿真实数据的能力, 即最小化 D 判别出生成数据, 最后达到一个纳什平衡.

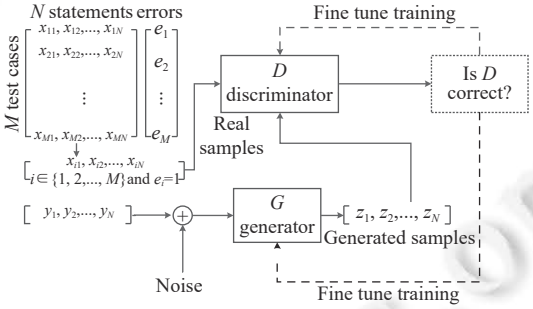

图 5 GAN-Aug 架构

数据量: 在确定如何生成模型域失败测试用例之后, GAN-Aug 下一步应该确定生成多少数量才能达到最优效果. 已有研究[18]发现一个类别平衡的测试用例集更有利于缺陷定位, 即类别平衡的测试用例集的缺陷定位效能要优于在不平衡的测试用例集[38]. 因此, GAN-Aug 不断生成模型域失败测试用例, 直至模型域失败测试用例数量与模型域成功测试用例数量相等.

算法 1 对 GAN-Aug 进行了介绍. 算法的输入为 T 在模型域的测试用例集为 *TOrig* (图 2 所示的频谱信息) 和训练批次 *epochNum*, 输出为新的模型域测试用例集 *TNew*. 第 1–4 行对 *TNew*, *TOrigF*, *Pnum* 和 *Fnum* 进行初始化. *TOrigF* 为初始模型域失败测试用例, *Pnum* 为原始测试用例集中成功测试用例数量, *Fnum* 为原始测试用例集中失败测试用例数量. 第 5–15 行循环生成新的模型域失败测试用例, 直到新的模型域测试用例集拥有相同数量的成功测试用例和失败测试用例. 其中, 第 5–9 行为判别网络 D 和生成网络 G 的训练过程, 一共训练 *epochNum* 个批次. 第 6 行为生成一个随机噪声数据, 维度等同于一个模型域测试用例, 即拥有等同于程序中可执行语句数的元素的

向量 (如图 2 所示). 第 7 行利用原始模型域失败测试用例和噪声数据训练判别器 D. 第 8 行在保持 D 的参数不变的情况下利用噪声数据 z 作为输入训练生成网络 G. 经过 *epochNum* 个批次的训练, 可以得到训练好的生成网络 G. 第 10 行计算需要生成的模型域失败测试用例数量 *FNewnum*. 第 11–15 行重复生成模型域失败测试用例, 数量为 *FNewnum*. 其中第 12 行生成噪声数据 z, 第 13 行以 z 作为输入利用训练好的生成网络生成模型域测试用例 t_{new}. 第 14 行将这个模型域失败测试用例赋予失败测试用例的标签 1, 第 15 行将这个最终的结果 t_{new} 添加到新的模型域测试用例集 *TNew* 中. DLFL 和 SBFL 以这个新的模型域测试用例集 *TNew* 作为输入, 计算各个语句的可疑值, 并降序排列.

算法 1. GAN-Aug 算法.

Input: 原始模型域测试用例集 *TOrig*, 训练批次 *epochNum*;
Output: 新的模型域测试用例集 *TNew*.

1. *TNew* = *TOrig*;
2. *TOrigF* = *getFailingTests(TOrig)*;
3. *Pnum* = *getNumberofPassingTests*(*TOrig*);
4. *Fnum* = *getNumberofFailingTests*(*TOrig*);
5. **for** $i = 1$; $i \leqslant$ *epochNum*; i++ **do**
6. z = *fakeData*();
7. D = *TrainModelD*(*TOrigF*, z);
8. G = *TrainModelG*(z, D);
9. **end for**
10. *FNewnum* = *Pnum* – *Fnum*
11. **for** $i = 1$; $i \leqslant$*FNewnum*; i++ **do**
12. z = *fakeData*();
13. $t_{new} = G(z)$;
14. Fix a failing label to t_{new};
15. *Add*(*TNew*, t_{new});
16. **end for**

(2) 示例

如图 6 所示, 本节为 GAN-Aug 应用的示例. 在图 6 中, 假设有一个包含 16 个语句的程序 P, 其中 s_3 为缺陷语句. s_1–s_{16} 下每个单元格表示语句是否被左侧第 1 列的测试用例执行 (1 表示被执行, 0 表示未被执行), suspicousness 行的单元格表示每个语句在左侧第 1 列缺陷定位方法下计算的可疑值, 最右侧单元格表示表示测试用例的执行是成功还是失败, 0 表示成功, 1 表示失败. 其中, t_1 和 t_6 为失败测试用例. 本例选择 GP02 和 MLP-FL 分别代表 SBFL 和 DLFL 两种缺陷定位方法. GAN-Aug 迭代生成模型域失败测试用例, 直到拥有相同数量的模型域成功测试用例和失败测试用例. t_7 和 t_8 为生成的两个模型域失败测试用例. 根据算法 1 的第 19 行将这两个生成的向量的标签置为 1, 从而测试用例集拥有 4 个成功模型域测试用例和 4 个失败模型域测试用例.

图 6 中共有 4 个可疑值降序排列的语句列表. 在原始测试用例集下的 GP02 可疑值列表为{s_7, s_8, s_9, s_{12}, s_{10}, s_{11}, s_{14}, s_{15}, s_{16}, s_{13}, s_1, s_2, s_3, s_6, s_4, s_5}, MLP-FL 的可疑值列表为{s_5, s_4, s_{16}, s_{11}, s_{15}, s_6, s_{13}, s_{12}, s_2, s_8, s_9, s_3, s_{14}, s_{10}, s_1, s_7}. 在新的测试用例集下, GP02 为{s_{12}, s_7, s_8, s_9, s_{10}, s_3, s_1, s_{14}, s_{13}, s_2, s_{11}, s_{15}, s_{16}, s_6, s_5, s_4}, MLP-FL 为{s_5, s_1, s_{14}, s_9, s_3, s_2, s_{15}, s_{11}, s_8, s_{16}, s_4, s_{13}, s_{12}, s_{10}, s_7, s_6}. 在原始测试用例集下, 缺陷语句 s_3 在 GP02 和 MLP-FL 下排名分别为第 13 和第 12. 在新的测试用例集下, 缺陷语句 s_3 在 GP02 和 MLP-FL 下排名分别为第 6 和第 5, 均提升了 7 个位次. 这表明 GAN-Aug 提升了两种缺陷定位方法的效能.

Program P (maximal value of a,b,c)															Bug information			
s_1: *Read*(a,b,c) s_2: d_1=0,d_2=0,d_3=0; s_3: if(b<0){ s_4: d_1=b; s_5: d_2=c; s_6: d_3=a;} s_7: else {d_1=b+1;		s_8: d_2=c+1; s_9: if(a<0){ s_{10}: a=a+c;} s_{11}: else a=a+b; s_{12}: d_3=a+1;} s_{13}: if(c>0){ s_{14}: *output*(d_1);}			s_{15}: else(output(d_2); s_{16}: output(d_3);}		t_7 and t_8 are generated virtual test cases for GAN-Aug. Then there are 4 passing test cases and 4 failing test cases.							s_3 is faulty. Correct form: if(b<6){			
T	a,b,c	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	result
t_1	−1,5,3	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	1
t_2	−2,−7,5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0
t_3	5,−6,−8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0
t_4	−5,8,−8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0
t_5	4,7,11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0
t_6	4,2,−1	1	1	1	0	0	0	1	1	1	0	1	1	1	0	1	1	1
t_7		0.97	0.94	0.97	0	0	0	0.98	0.97	0.95	0.91	0	0.99	0.96	0.9	0	0	1
t_8		0.98	0.97	0.98	0	0	0	1	0.99	0.97	0.93	0	1	0.98	0.92	0	0	1
GP02	suspicious-ness	6.0	6.0	6.0	4.24	4.24	4.24	8.24	8.24	8.24	6.46	6.46	8.24	6.0	6.24	6.24	6.24	
	rank	11	12	13	15	16	14	1	2	3	5	6	4	10	7	8	9	
GP02	suspicious-ness	9.9	9.82	9.9	4.24	4.24	4.24	12.2	12.16	12.08	10.14	6.46	12.22	9.88	9.88	6.24	6.24	GAN-Aug
	rank	7	10	6	16	15	14	2	3	4	5	11	1	9	8	12	13	
MLP	suspicious-ness	0.487	0.493	0.492	0.5	0.51	0.495	0.487	0.492	0.492	0.49	0.498	0.494	0.494	0.491	0.495	0.499	
	rank	15	9	12	2	1	6	16	10	11	14	4	8	7	13	5	3	
MLP	suspicious-ness	0.802	0.791	0.798	0.769	0.809	0.739	0.752	0.774	0.799	0.755	0.787	0.758	0.761	0.801	0.787	0.771	GAN-Aug
	rank	2	6	5	11	1	16	15	9	4	14	8	13	12	3	7	10	

图 6 GAN-Aug 示例

3 实 验

(1) 实验设计

为了验证 GAN-Aug 的有效性, 本文选取表 2 所示的 14 组大型程序进行实验. 选取它们的原因为: (1) 它们是缺陷定位领域中广泛使用的程序[8−12,14,22]; (2) 它们都是中大型程序, 代码行数从 5.4k 行到 491k 行; (3) 它们均是公开数据集并很容易获取, 易于进行复现和进一步研究. 在这些程序中, 前 6 个程序 (chart, math, lang, closure, mockito 和 time) 取自 Defects4J[39], Python, gzip 和 libtiff 从 ManyBugs[40]获取, space, nanoxml_v1, nanoxml_v2, nanoxml_v3 和 nanoxml_v5 从 SIR[41]获取. 每个程序版本都有一个 diff 命令来比较错误版本和正确版本, 可以使用它来获取缺陷位置. 本文对以上缺陷位置进行了总结和在线公布[42]. 实验选取当前 11 种先进的典型缺陷定位方法[4,8,10,27,43−45], 即 MLP-FL, CNN-FL, BiLSTM-FL, ER5, GP02, GP03, DStar, ER1′, GP19, Ochiai, CombineFL. 对于基于深度学习的缺陷定位方法 (MLP-FL, CNN-FL, BiLSTM-FL), 本文基于先前研究[8,10]对它们进行了实现. 具体来说, 对于 MLP-FL, 根据可执行语句的数量采用一个输入层, 一个输出层和若干隐藏层. 对于 CNN-FL, 采用一个输入层, 一个具有 32 内核的卷积层, 一个具有 64 内核的卷积层, 两个池化层, 一个修正线性单元和一个输出层. 对于 BiLSTM-FL, 采用一个输入层, 两个循环层和一个输出层. 在输入层中, 根据可执行语句数量将其分为若干组. 在循环层中, 将两个 LSTM (一个向前, 一个向后) 组合成一个双向 LSTM, 通过堆叠两个双向 LSTM 从而形成一个深层结构, 以获得更高级别的抽象能力. 对于基于程序谱的缺陷定位方法 (ER5, GP02, GP03, DStar, ER1′, GP19, Ochiai), 本文基于广泛使用的 SBFL 源代码 GZoltar (https://gzoltar.com/) 对其进行了实现. 对于 CombineFL, 本文采用了 SBFL 和切片相结合的方法, 针对 Java 程序使用的工具是 JSlice[46]和 JavaSlicer[47], 针对 C 程序使用的工具是 Wet[48]. 实验在两种场景下比较 11 种缺陷定位方法的有效性: 使用 GAN-Aug 和未使用 GAN-Aug. 此外, 实验对比了 GAN-Aug 与两种典型的数据优化方法 (降采样[49]和重采样[9,18]). 其中, 降采样去除了多数类数据, 重采样则是复制了少数类数据.

表 2 实验对象

Program	Description	Versions	KLOC	Test	Type
chart	JFreeChart	26	96	2 205	Real
math	Apache commons math	106	85	3 602	Real
lang	Apache commons-lang	65	22	2 245	Real
closure	Closure compiler	133	90	7 927	Real
mockito	Framework for unit tests	38	6	1 075	Real
time	Joda-time	27	53	4 130	Real
Python	General-purpose language	8	407	355	Real
gzip	Data compression	5	491	12	Real
libtiff	Image processing	12	77	78	Real
space	ADL interpreter	38	6.1	13 585	Real
nanoxml_v1	XML parser	7	5.4	206	Seeded
nanoxml_v2	XML parser	7	5.7	206	Seeded
nanoxml_v3	XML parser	10	8.4	206	Seeded
nanoxml_v5	XML parser	7	8.8	206	Seeded

实验设置为: 一是训练超参数设置. 初始学习率 learning rate 值为 0.000 1–0.01, 根据输入数据的规模动态调整; dropout rate 的搜索范围为{0.98, 0.96, 0.95, 0.94, 0.93}; batch size 为{16, 32, 64, 128}; 我们使用的优化器为 Adam; 训练批次 epoch 我们根据输入数据的规模设置为 1 000–5 000. 二是实验的物理环境. CPU 为 I7-9700, 64 GB 物理内存, GPU 为 12 GB 的 NVIDIA TITAN X Pascal, 操作系统为 Ubuntu 16.04.3, 我们在 Matlab R2016b 上进行了数据统计和绘图显示.

(2) 评价指标

实验选取 4 种广泛使用的评价指标, 即 Top-*N*[50], mean average rank (*MAR*)[12], mean first rank (*MFR*)[12]和 relative improvement (*RImp*)[51,52]. 其中, Top-*N* 指的是在所有可疑值降序排列的语句列表中, 缺陷语句在前 *N* 个位置内的百分比. *MAR* 指的是使用缺陷定位方法定位到缺陷语句的平均排名 (rank). *MFR* 指的是同一个缺陷出现在多个位置时, 使用缺陷定位方法定位到的第 1 个缺陷语句所在位置的排名 (rank). *RImp* 指的是使用 GAN-Aug 定位到一个程序的所有缺陷时需要检查的语句总数和未使用 GAN-Aug 定位到一个程序的所有缺陷时需要检查的语句总数的比值.

(3) 结果分析

Top-*N*, *MAR* 和 *MFR*: 实验在 11 种缺陷定位方法上使用 Top-*N* (*N*=1, 5, 10), *MAR* 和 *MFR* 对 GAN-Aug 进行了评估. 表 3 显示了实验结果. 当 11 种缺陷定位方法使用了 GAN-Aug 后, 它们 Top-*N* 值提升, *MFR* 和 *MAR* 值降低. 这表明 GAN-Aug 提升了 11 种缺陷定位方法的效能.

RImp: 为了进一步验证 GAN-Aug 的有效性, 实验采用 *RImp* 来评估 GAN-Aug. 图 7 和图 8 展示了两种场景下 GAN-Aug 的 *RImp* 对比值: 图 7 为 GAN-Aug 在 11 种缺陷定位方法上的 *RImp* 对比值, 图 8 为 GAN-Aug 在 14 组对象程序上的 *RImp* 对比值. 如图 8 所示, *RImp* 的值均小于 100%, 这表明 GAN-Aug 提升了所有缺陷定位方法的定位效能. 以 MLP-FL 为例, *RImp* 的值为 64.35%, 这表明使用 GAN-Aug 后, MLP-FL 定位到所有缺陷需要检查的语句总数为不使用 GAN-Aug 时 MLP-FL 定位到所有缺陷需要检查的语句总数的 64.35%. 换句话说, GAN-Aug 为 MLP-FL 节省了 35.65% (100% – 64.35% = 35.65%) 的可检查语句数. GAN-Aug 的 *RImp* 值从 GP03 的 49.97% 到 CombineFL 的 95.24%, 表明 GAN-Aug 最大节省 50.03% (100% – 49.97% = 50.03%) 的可检查语句数, 最小节省 4.76% (100% – 95.24% = 4.76%) 的可检查语句数.

同理, 在图 8 中, *RImp* 的值均小于 100%, 这表明 GAN-Aug 在所有实验对象程序上提升了缺陷定位效能. 以 chart 为例, *RImp* 的值为 65.46%, 这表明使用 GAN-Aug 后, 11 种缺陷定位方法定位 chart 的所有缺陷需要检查的语句总数为不使用 GAN-Aug 时 11 种缺陷定位方法定位到 chart 所有缺陷需要检查的语句总数的 65.46%. 换句话说, GAN-Aug 为 chart 节省了 34.54% (100% – 65.46% = 34.54%) 的可检查语句数. GAN-Aug 的 *RImp* 值从 math

的 58.92% 到 libtiff 的 97.35%, 表明 GAN-Aug 最大节省 41.08% (100% – 58.92% = 41.08%) 的可检查语句数, 最小节省 2.65% (100% – 97.35% = 2.65%) 的可检查语句数. 因此, 从 *RImp* 的结果可以看出, GAN-Aug 提升了缺陷定位效能.

表 3 GAN-Aug 在 11 种缺陷定位方法上的 Top-*N*, *MAR* 和 *MFR* 对比值

Comparison	Top-1 (%)	Top-5 (%)	Top-10 (%)	*MFR*	*MAR*
MLP-FL	0	2.1	3.1	213	352
MLP-FL (GAN-Aug)	2.6	5.3	12.2	189	325
CNN-FL	1.6	3.1	8.2	179	263
CNN-FL (GAN-Aug)	2.6	8.7	17.0	117	251
BiLSTM-FL	0	1.6	3.1	235	412
BiLSTM-FL (GAN-Aug)	1.6	4.1	9.3	210	353
ER5	0	6.2	12.4	247	421
ER5 (GAN-Aug)	1.6	8.7	15.5	209	376
GP02	1.6	15.5	20.6	263	545
GP02 (GAN-Aug)	2.6	18.6	24.2	219	473
GP03	3.1	11.3	12.4	217	363
GP03 (GAN-Aug)	3.1	16.4	18.3	191	297
DStar	3.1	20.1	26.3	241	357
DStar (GAN-Aug)	3.6	24.1	34.5	224	323
ER1′	3.1	18.6	26.3	242	371
ER1′(GAN-Aug)	3.6	24.1	29.4	218	319
GP19	3.1	9.3	15.5	253	391
GP19 (GAN-Aug)	3.1	10.8	21.1	239	372
Ochiai	1.6	20.1	24.2	215	363
Ochiai (GAN-Aug)	3.1	24.1	27.3	192	281
CombineFL	4.4	28.1	45.6	67	94
CombineFL (GAN-Aug)	4.4	29.4	47.9	45	72

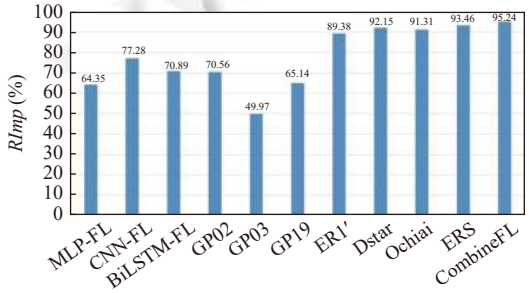

图 7 GAN-Aug 在 11 种缺陷定位方法上的 *RImp* 对比值

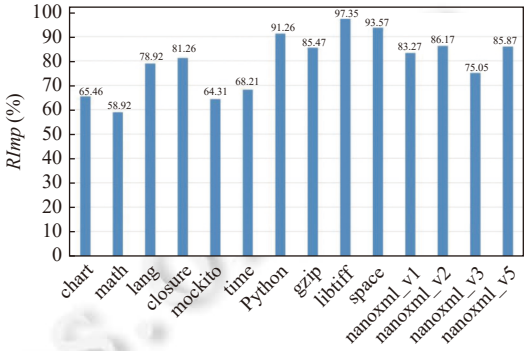

图 8 GAN-Aug 在 14 组对象程序上的 *RImp* 对比值

统计比较: 本文采用 Wilcoxon-signed-rank test 符号秩检验[53]来验证 GAN-Aug 在统计学上的有效性. Wilcoxon-signed-rank test 符号秩检验用于测试一对数据之间的差异, 假设有 $F(x)$ 和 $G(y)$, 给定一个参数 φ, 可以使用 2-tailed 和 1-tailed p-value 来得到一个结果. 对于 2-tailed p-value, 如果 $p \geqslant \varphi$, 则接受假设 H_0: $F(x)$ 和 $G(y)$ 没有差异; 否则接受假设 H_1: $F(x)$ 和 $G(y)$ 有差异. 对于 1-tailed p-value 有两种情况: 分别是 1-tailed (right) 和 1-tailed (left). 对于 1-tailed (right), 如果 $p \geqslant \varphi$, 则接受假设 H_0: $F(x)$ 和 $G(y)$ 相比结果为 Worse 或 Similar; 否则接受假设 H_1: $F(x)$ 和 $G(y)$ 相比结果为 Better. 对于 1-tailed (left) , 如果 $p \geqslant \varphi$, 则接受假设 H_0: $F(x)$ 和 $G(y)$ 相比结果为 Better 或 Similar; 否则接受假设 H_1: $F(x)$ 和 $G(y)$ 相比结果为 Worse[54].

对于每一种缺陷定位方法或实验对象程序, 实验设置使用 GAN-Aug 的 rank 值为 $F(x)$, 未使用 GAN-Aug 的

rank 值为 $G(y)$. 实验同时使用了 2-tailed 和 1-tailed, φ 的值设为 0.05. 例如, 给定一种缺陷定位方法为 FL1, 实验将使用 GAN-Aug 的 FL1 在所有错误版本的 rank 值作为 $F(x)$, 并将未使用 GAN-Aug 的 FL1 在所有错误版本的 rank 值作为 $G(y)$. 在 2-tailed 检测中, 当 H_0 在 φ 为 0.05 的水平被接受, 这表明使用 GAN-Aug 的 FL1 和未使用 GAN-Aug 的 FL1 具有相似的定位效能, 结果为 Similar. 在 1-tailed (right) 检测中, 当 H_1 在 φ 为 0.05 的水平被接受, 这表明使用 GAN-Aug 的 FL1 不如未使用 GAN-Aug 的 FL1 的定位效能好, 结果为 Worse. 在 1-tailed (left) 检测中, 当 H_1 在 φ 为 0.05 的水平被接受, 这表明使用 GAN-Aug 的 FL1 比未使用 GAN-Aug 的 FL1 的定位效能好, 结果为 Better. 本实验使用的置信水平是 0.95.

表 4 展示了两种场景下的 GAN-Aug 的 Wilcoxon-signed-rank test 对比. 一是 GAN-Aug 在 11 种缺陷定位方法上的对比; 二是 GAN-Aug 在 14 组对象程序上的对比. 以 MLP-FL (MFT-Aug) vs. MLP-FL 和 gzip (MFT-Aug) vs. gzip 为例, 在 MLP-FL (MFT-Aug) vs. MLP-FL 上, GAN-Aug 获得了 14 个 Better (14/14=100%), 0 个 Similar (0/14=0), 0 个 Worse (0/14=0); 在 gzip (MFT-Aug) vs. gzip 上, GAN-Aug 获得了 7 个 Better (7/11=63.6%), 4 个 Similar (4/11=36.4%), 0 个 Worse (0/12=0). GAN-Aug 共获得 198 个 Better. 结果表明本文方法 GAN-Aug 能大幅提升缺陷定位效能. 为了进一步评估显著性水平, 本文利用非参数 Vargha-Delaney A 检验[55]. 对于 Vargha-Delaney A 检验, Vargha 等人[56]认为 A 检验的效应值 (effect size) 大于 0.64 (或小于 0.36) 表示“中等”效应值, 大于 0.71 (或小于 0.29) 表示“较高”效应值. 从表 4 可以看出, A 检验的效应值为 0.77, 为较高效应值..

表 4 GAN-Aug 有效性的 Wilcoxon-signed-rank test 对比

在缺陷定位方法上的对比	Better	Similar	Worse	在缺陷定位方法上的对比	Better	Similar	Worse
MLP-FL (GAN-Aug) vs. MLP-FL	14 (100%)	0 (0)	0 (0)	CNN-FL (GAN-Aug) vs. CNN-FL	14 (100%)	0 (0)	0 (0)
BiLSTM-FL (GAN-Aug) vs. BiLSTM-FL	14 (100%)	0 (0)	0 (0)	GP19 (GAN-Aug) vs. GP19	8 (57.1%)	6 (42.9%)	0 (0)
Ochiai (GAN-Aug) vs. Ochiai	7 (50%)	7 (50%)	0 (0)	ER5 (GAN-Aug) vs. ER5	8 (57.1%)	6 (42.9%)	0 (0)
GP02 (GAN-Aug) vs. GP02	7 (50%)	7 (50%)	0 (0)	GP03 (GAN-Aug) vs. GP03	9 (64.3%)	5 (35.7%)	0 (0)
DStar (GAN-Aug) vs. DStar	7 (50%)	7 (50%)	0 (0)	ER1′(GAN-Aug) vs. ER1′	6 (42.9%)	8 (57.1%)	0 (0)
CombineFL (GAN-Aug) vs. CombineFL	5 (35.7%)	9 (64.3%)	0 (0)				
在对象程序上的对比	Better	Similar	Worse	在对象程序上的对比	Better	Similar	Worse
gzip (GAN-Aug) vs. gzip	7 (63.6%)	4 (36.4%)	0 (0)	libtiff (GAN-Aug) vs. libtiff	5 (45.5%)	6 (54.5%)	0 (0)
lang (GAN-Aug) vs. lang	4 (36.4%)	7 (63.6%)	0 (0)	closure (GAN-Aug) vs. closure	5 (45.5%)	6 (54.5%)	0 (0)
Python (GAN-Aug) vs. Python	4 (36.4%)	7 (63.6%)	0 (0)	space (GAN-Aug) vs. space	6 (54.5%)	5 (45.5%)	0 (0)
chart (GAN-Aug) vs. chart	6 (54.5%)	5 (45.5%)	0 (0)	math (GAN-Aug) vs. math	10 (90.9%)	1 (9.1%)	0 (0)
mockito (GAN-Aug) vs. mockito	9 (81.8%)	2 (18.3%)	0 (0)	time (GAN-Aug) vs. time	8 (72.7%)	3 (27.3%)	0 (0)
nanoxml_v1 (GAN-Aug) vs. nanoxml_v1	9 (81.8%)	2 (18.3%)	0 (0)	nanoxml_v2 (GAN-Aug) vs. nanoxml_v2	8 (72.7%)	3 (27.3%)	0 (0)
nanoxml_v3 (GAN-Aug) vs. nanoxml_v3	9 (81.8%)	2 (18.3%)	0 (0)	nanoxml_v5 (GAN-Aug) vs. nanoxml_v5	9 (81.8%)	2 (18.3%)	0 (0)
A-Test	0.77						

与典型数据优化方法对比: 为了进一步验证 GAN-Aug 的有效性, 实验将 GAN-Aug 与两种数据优化方法 (降采样[49]用 undSam 表示和重采样[9,18]用 reSam 表示) 进行了比较. 表 5 和表 6 分别给出了 GAN-Aug 与降采样, GAN-Aug 与重采样有效性的 Wilcoxon-signed-rank test 对比. 在表 5 和降采样的对比中, GAN-Aug 共获得 206 个 Better. 在表 6 和重采样的对比中, GAN-Aug 共获得 156 个 Better. 表 5 和表 6 的 A 检验效应值分别 0.76 和 0.65, 表示 GAN-Aug 与降采样, GAN-Aug 与重采样相比为较高效应值和中等效应值.

通过以上对比分析, 我们发现 GAN-Aug 在不使用增强策略的缺陷定位、降采样及重采样 3 种情形下都取得了较好的效果. 究其原因, 一是通过不断生成模型域测试用例, GAN-Aug 构建了类别平衡的测试用例集; 二是失败测试用例占比较少, 裁剪成功的测试用例达到类别平衡会导致信息丢失严重, 而 GAN-Aug 避免了信息丢失; 三是 GAN-Aug 以所有失败测试用例为学习对象, 在学习最小可以集合的基础上最大化模拟真实模型域失败测试用例

的潜在分布, 避免了较多的冗余信息.

(4) 讨论分析

通过以上分析, 我们发现 GAN-Aug 有效提升了现有缺陷定位方法的有效性. 这说明我们试图生成最大化模拟真实模型域失败测试用例潜在分布的新用例是有效的. 为了验证该分析, 我们设计了覆盖所有测试用例共同特征 (最小可疑集合[22]) 并在剩余覆盖信息中随机生成数据的向量 (用 Aug-Random 表示), 将 GAN-Aug 与 Aug-Random 进行对比. 表 7 展示了 GAN-Aug 和 Aug-Random 有效性的 Wilcoxon-signed-rank test 对比. 结果表明 GAN-Aug 共获得 182 个 Better, 验证了本文方法有效的原因.

表 5 GAN-Aug 和降采样有效性的 Wilcoxon-signed-rank test 对比

在缺陷定位方法上的对比	Better	Similar	Worse	在缺陷定位方法上的对比	Better	Similar	Worse
MLP-FL (GAN-Aug) vs. MLP-FL (undSam)	14 (100%)	0 (0)	0 (0)	CNN-FL (GAN-Aug) vs. CNN-FL (undSam)	14 (100%)	0 (0)	0 (0)
BiLSTM-FL (GAN-Aug) vs. BiLSTM-FL (undSam)	14 (100%)	0 (0)	0 (0)	GP19 (GAN-Aug) vs. GP19 (undSam)	9 (64.3%)	5 (35.7%)	0 (0)
Ochiai (GAN-Aug) vs. Ochiai (undSam)	6 (42.9%)	6 (42.9%)	2 (14.2%)	ER5 (GAN-Aug) vs. ER5 (undSam)	8 (57.1%)	6 (42.9%)	0 (0)
GP02 (GAN-Aug) vs. GP02 (undSam)	9 (64.3%)	4 (28.6%)	1 (7.1%)	GP03 (GAN-Aug) vs. GP03 (undSam)	10 (71.4%)	4 (28.6%)	0 (0)
DStar (GAN-Aug) vs. DStar (undSam)	6 (42.9%)	7 (50%)	1 (7.1%)	ER1′(GAN-Aug) vs. ER1′ (undSam)	6 (42.9%)	7 (50%)	1 (7.1%)
CombineFL (GAN-Aug) vs. CombineFL (undSam)	7 (50%)	7 (50%)	0 (0)				
在对象程序上的对比	Better	Similar	Worse	在对象程序上的对比	Better	Similar	Worse
gzip (GAN-Aug) vs. gzip	7 (63.6%)	4 (36.4%)	0 (0)	libtiff (GAN-Aug) vs. libtiff	5 (45.6%)	6 (54.5%)	0 (0)
lang (GAN-Aug) vs. lang (undSam)	7 (63.6%)	4 (36.4%)	0 (0)	closure (GAN-Aug) vs. closure (undSam)	6 (54.5%)	4 (36.4%)	1 (9.1%)
Python (GAN-Aug) vs. Python (undSam)	9 (81.8%)	2 (18.2%)	0 (0)	space (GAN-Aug) vs. space (undSam)	7 (63.6%)	4 (36.4%)	0 (0)
chart (GAN-Aug) vs. chart (undSam)	9 (81.8%)	2 (18.2%)	0 (0)	math (GAN-Aug) vs. math (undSam)	7 (63.6%)	2 (18.2%)	2 (18.2%)
mockito (GAN-Aug) vs. mockito (undSam)	8 (72.7%)	3 (27.3%)	0 (0)	time (GAN-Aug) vs. time (undSam)	7 (63.6%)	3 (27.3%)	1 (9.1%)
nanoxml_v1 (GAN-Aug) vs. nanoxml_v1 (undSam)	7 (63.6%)	3 (27.3%)	1 (9.1%)	nanoxml_v2 (GAN-Aug) vs. nanoxml_v2 (undSam)	8 (72.7%)	3 (27.3%)	0 (0)
nanoxml_v3 (GAN-Aug) vs. nanoxml_v3 (undSam)	8 (72.7%)	3 (27.3%)	0 (0)	nanoxml_v5 (GAN-Aug) vs. nanoxml_v5 (undSam)	8 (72.7%)	3 (27.3%)	0 (0)
A-Test	0.76						

(5) 有效性威胁

实验有一个潜在假设: 当程序发生失效, 缺陷语句应该被执行. 这个假设通常成立, 实验结果也证实了假设的合理性. 然而, 在某些情况下, 则该假设可能会不成立. 例如在多缺陷的情况下, 程序发生失效时可能只会覆盖部分缺陷语句, 从而最小可疑集合出现缺陷语句丢失情况. 本文主要针对单一缺陷进行研究, 它们也是研究多缺陷定位的基础. 我们也可以采用聚类方法[57]分离出多缺陷, 转换成单缺陷场景, 以此缓解特殊情况带来的影响.

实验对象程序也是有效性威胁之一. 实验选择广泛应用于缺陷定位领域的代表性程序. 这些程序中包括一些只包含一个或两个失败测试用例的缺陷. 在此种情况下, 我们会重复将这个 (这些) 模型域失败测试用例作为输入不断迭代, 收敛后生成网络 G 的输出则会是这些测试用例的复制, 最后获得的类别平衡的测试用例集也是有利于缺陷定位的. 其次是现实调试中存在许多未知因素, 例如, 在偶然正确性测试用例大量存在的测试用例集中, 生成模型域失败测试用例并不一定比降采样有效, 因此针对这种情况需要进一步对其进行研究. 这说明它们不能覆盖

与适用于现实中的所有情况. 因此, 未来工作使用更多的真实大型程序将会是本文研究的重点. 三是数据集存在有效性威胁. 当失败测试用例数量极少时 (1–2 个), 我们反复将这些极少数相同的失败测试用例作为输入, 模型的输出几乎为这些测试用例的复制或就是对他们的复制, 而这种新生成的用例对于缺陷定位也是有益的[9]. 针对极少失败测试用例实验程序的平均训练开销, chart 为 42 min, math 为 29 min, lang 为 48 min, closure 为 52 min, mockito 为 33 min, time 为 58 min.

本文采用 Top-*N*, *MAR*, *MFR* 和 *RImp* 作为评估指标, 并使用 Wilcoxon-signed-rank test 来验证 GAN-Aug 相对于对比方法的改进是否显著. 这些评估指标是缺陷定位领域广泛使用的指标, 这方面的有效性威胁可以忽略. 此外, 实验程序也可能包含许多未知的错误, 在本文实现的数据增强方法可能会存在潜在的错误, 这些因素都可能会影响到结果的有效性.

表 6 GAN-Aug 和重采样有效性的 Wilcoxon-signed-rank test 对比

在缺陷定位方法上的对比	Better	Similar	Worse	在缺陷定位方法上的对比	Better	Similar	Worse
MLP-FL (GAN-Aug) vs. MLP-FL (reSam)	7 (50%)	5 (35.7%)	2 (14.3%)	CNN-FL (GAN-Aug) vs. CNN-FL (reSam)	8 (57.1%)	6 (42.9%)	0 (0)
BiLSTM-FL (GAN-Aug) vs. BiLSTM-FL (reSam)	12 (85.7%)	2 (14.3%)	0 (0)	GP19 (GAN-Aug) vs. GP19	8 (57.1%)	6 (42.9%)	0 (0)
Ochiai (GAN-Aug) vs. Ochiai	5 (35.7%)	7 (50%)	2 (14.3%)	ER5 (GAN-Aug) vs. ER5 (reSam)	4 (28.6%)	9 (64.3%)	1 (7.1%)
GP02 (GAN-Aug) vs. GP02 (reSam)	9 (64.3%)	5 (35.7%)	0 (0)	GP03 (GAN-Aug) vs. GP03 (reSam)	8 (57.1%)	4 (28.6%)	2 (14.3%)
DStar (GAN-Aug) vs. DStar(reSam)	6 (42.9%)	8 (57.1%)	0 (0)	ER1′(GAN-Aug) vs. ER1′ (reSam)	6 (42.9%)	7 (50%)	1 (7.1%)
CombineFL (GAN-Aug) vs. CombineFL	5 (35.7%)	6 (42.9%)	3 (21.4%)				
在对象程序上的对比	Better	Similar	Worse	在对象程序上的对比	Better	Similar	Worse
gzip (GAN-Aug) vs. gzip (reSam)	5 (45.6%)	5 (45.6%)	1 (9.1%)	libtiff (GAN-Aug) vs. libtiff (reSam)	4 (36.4%)	7 (63.6%)	0 (0)
lang (GAN-Aug) vs. lang (reSam)	4 (36.4%)	5 (45.6%)	2 (18.2%)	closure (GAN-Aug) vs. closure (reSam)	5 (45.6%)	6 (54.5%)	0 (0)
Python (GAN-Aug) vs. Python	4 (36.4%)	7 (63.6%)	0 (0)	space (GAN-Aug) vs. space	5 (45.6%)	6 (54.5%)	0 (0)
chart (GAN-Aug) vs. chart (reSam)	5 (45.6%)	5 (45.6%)	1 (9.1%)	math (GAN-Aug) vs. math (reSam)	7 (63.6%)	3 (27.3%)	1 (9.1%)
mockito (GAN-Aug) vs. mockito (reSam)	8 (72.7%)	3 (27.3%)	0 (0)	time (GAN-Aug) vs. time (reSam)	6 (54.5%)	4 (36.4%)	1 (9.1%)
nanoxml_v1 (GAN-Aug) vs. nanoxml_v1 (reSam)	5 (45.6%)	4 (36.4%)	2 (18.2%)	nanoxml_v2 (GAN-Aug) vs. nanoxml_v2 (reSam)	7 (63.6%)	2 (18.2%)	2 (18.2%)
nanoxml_v3 (GAN-Aug) vs. nanoxml_v3 (reSam)	6 (54.5%)	4 (36.4%)	1 (9.1%)	nanoxml_v5 (GAN-Aug) vs. nanoxml_v5 (reSam)	7 (63.6%)	4 (36.4%)	0 (0)
A-Test				0.65			

4 相关工作

本节主要简介缺陷定位中的类别不平衡研究和程序缺陷定位技术的研究. 更多缺陷定位工作可参考 Wong 等人发表的综述[1].

在缺陷定位领域, 研究者们对数据集类别不平衡问题进行了研究. Wong 等人[58]表明数据集的类别不平衡问题会影响缺陷定位的效能. Japkowizc 等人[59]通过实验证明了类别不平衡问题对分类会产生负效应. Hao 等人[60]通过对测试用例集进行裁剪来提升缺陷定位的效能. 然而, Yu 等人[61]表明对测试用例集进行裁剪, 反而会减少缺陷定位的有用信息, 从而降低其效能. Gong 等人[17]通过实验证明相同数量成功测试用例和失败测试用例有利于缺

陷定位. 这些研究主要是成功测试用例, 很少为缺陷定位生成失败测试用例. 本文以模型域而非传统输入域作为突破口, 提出了模型域失败测试用例增强方法 GAN-Aug, 以此获取平衡数据集来改进缺陷定位效能.

基于程序覆盖信息的缺陷定位技术通过执行输入域测试用例从而构建模型域测试用例, 之后根据模型域测试用例设计算法评估程序实体包含缺陷的可能性并降序排列. 基于频谱的缺陷定位技术 (SBFL)[16]是最典型的基于程序覆盖信息的缺陷定位技术, 它通过构建可疑值计算公式来评估程序实体包含缺陷的可能性. Chen 等人[62]提出了 Jaccard 方法, Jones 等人[63]提出了 Tarantula. Abreu 等人[64,65]针对单点缺陷提出了 Ochiai 技术. Wong 等人[66,67]通过分析关联关系提出了 Wong1-3, Wong3'. 之后 Wong 等人[68]提出了最优的 SBFL 方法之一的 DStar. Lee 等人[7]提出了根据执行次数进行可疑值计算的缺陷定位方法. Dean 等人[69]提出了融合线性方法的 SBFL 技术. Abreu 等人[70,71]针对多缺陷提出基于模型诊断和 SBFL 的定位方法. 近年来, 随着深度学习技术的飞速发展, 基于学习的缺陷定位技术开始崭露头角并取得了不错的效果. Zhang 等人[8,10]将模型域测试用例作为输入, 提出了基于多层感知机, 卷积网络和循环网络的缺陷定位技术. Sohn 等人[29]提出了基于排序学习方法的 FLUCCS. Li 等人[12]利用深度学习和多种已有缺陷定位方法提出了面向方法级的缺陷定位技术 DeepFL. Jiang 等人[44]和 Zou 等人[45]通过融合多种缺陷定位技术来进行缺陷定位并取得了较好的效果. 本文的方法在上述基于程序覆盖信息的方法的基础上进行了模型域测试用例的生成, 提高了缺陷定位的效能.

表 7 GAN-Aug 和 Aug-Random 有效性的 Wilcoxon-signed-rank test 对比

在缺陷定位方法上的对比	Better	Similar	Worse	在缺陷定位方法上的对比	Better	Similar	Worse
MLP-FL (GAN-Aug) vs. MLP-FL (Aug-Random)	8 (57.1%)	6 (42.9%)	0 (0)	CNN-FL (GAN-Aug) vs. CNN-FL (Aug-Random)	9 (64.3%)	5 (35.7%)	0 (0)
BiLSTM-FL (GAN-Aug) vs. BiLSTM-FL (Aug-Random)	10 (71.4%)	4 (28.6%)	0 (0)	GP19 (GAN-Aug) vs. GP19 (Aug-Random)	8 (57.1%)	6 (42.9%)	0 (0)
Ochiai (GAN-Aug) vs. Ochiai (Aug-Random)	7 (50%)	7 (50%)	0 (0)	ER5 (GAN-Aug) vs. ER5 (Aug-Random)	8 (57.1%)	6 (42.9%)	0 (0)
GP02 (GAN-Aug) vs. GP02 (Aug-Random)	9 (64.3%)	5 (35.7%)	0 (0)	GP03 (GAN-Aug) vs. GP03 (Aug-Random)	7 (50%)	7 (50%)	0 (0)
DStar (GAN-Aug) vs. DStar (Aug-Random)	8 (57.1%)	6 (42.9%)	0 (0)	ER1′(GAN-Aug) vs. ER1′ (Aug-Random)	9 (64.3%)	5 (35.7%)	0 (0)
CombineFL(GAN-Aug) vs. CombineFL(Aug-Random)	8(57.1%)	6 (42.9%)	0 (0)				
在对象程序上的对比	Better	Similar	Worse	在对象程序上的对比	Better	Similar	Worse
gzip (GAN-Aug) vs. gzip (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)	libtiff (GAN-Aug) vs. libtiff (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)
lang (GAN-Aug) vs. lang (Aug-Random)	7 (63.6%)	4 (36.4%)	0 (0)	closure (GAN-Aug) vs. closure (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)
Python (GAN-Aug) vs. Python (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)	space (GAN-Aug) vs. space (Aug-Random)	7 (63.6%)	4 (36.4%)	0 (0)
chart (GAN-Aug) vs. chart (Aug-Random)	5 (45.4%)	6 (54.5%)	0 (0)	math (GAN-Aug) vs. math (Aug-Random)	7 (63.6%)	4 (36.4%)	0 (0)
mockito (GAN-Aug) vs. mockito (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)	time (GAN-Aug) vs. time (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)
nanoxml_v1 (GAN-Aug) vs. nanoxml_v1 (Aug-Random)	6 (54.5%)	5 (45.4%)	0 (0)	nanoxml_v2 (GAN-Aug) vs. nanoxml_v2 (Aug-Random)	8 (72.7%)	3 (27.3%)	0 (0)
nanoxml_v3 (GAN-Aug) vs. nanoxml_v3 (Aug-Random)	8 (72.7%)	3 (27.3%)	0 (0)	nanoxml_v5 (GAN-Aug) vs. nanoxml_v5 (Aug-Random)	7 (63.6%)	4 (36.4%)	0 (0)
A-Test				0.63			

5 结 论

本文提出基于对抗生成网络的缺陷定位模型域数据增强方法 GAN-Aug. 该方法利用对抗生成网络合成模型

域失败测试用例, 从而获取类别平衡的模型域测试用例集, 以此解决缺陷定位面临的类别不平衡问题. 实验结果表明, 在 12 种缺陷定位方法和 14 组实验对象程序上, GAN-Aug 大幅提升了缺陷定位效能. 未来工作包括方法优化和在多缺陷定位的应用, 进一步提升其准确性. 同时, 可以发现对抗网络有效的原因尚不完全明确, 需要结合机器学习可解释性等研究工作继续对其进行探索, 这也是未来的工作之一.

References:

[1] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016, 42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]

[2] Xie XY, Kuo FC, Chen TY, Yoo S, Harman M. Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In: Proc. of the 5th Int'l Symp. on Search Based Software Engineering. St. Petersburg: Springer, 2013. 224–238. [doi: 10.1007/978-3-642-39742-4_17]

[3] Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE). Honolulu: IEEE, 2011. 746–755. [doi: 10.1145/1985793.1985898]

[4] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 609–620. [doi: 10.1109/ICSE.2017.62]

[5] Cleve H, Zeller A. Locating causes of program failures. In: Proc. of the 27th Int'l Conf. on Software Engineering. St. Louis: IEEE, 2005. 342–351. [doi: 10.1109/ICSE.2005.1553577]

[6] Yoo S. Evolving human competitive spectra-based fault localisation techniques. In: Proc. of the 4th Int'l Symp. on Search Based Software Engineering. Riva del Garda: Springer, 2012. 244–258. [doi: 10.1007/978-3-642-33119-0_18]

[7] Lee HJ, Naish L, Ramamohanarao K. Effective software bug localization using spectral frequency weighting function. In: Proc. of the 34th IEEE Annual Computer Software and Applications Conf. Seoul: IEEE, 2010. 218–227. [doi: 10.1109/COMPSAC.2010.26]

[8] Zhang Z, Lei Y, Mao XG, Li PP. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Hangzhou: IEEE, 2019. 445–455. [doi: 10.1109/SANER.2019.8668002]

[9] Zhang Z, Lei Y, Mao XG, Yan M, Xu L, Wen JH. Improving deep-learning-based fault localization with resampling. Journal of Software: Evolution and Process, 2021, 33(3): e2312. [doi: 10.1002/smr.2312]

[10] Zhang Z, Lei Y, Mao XG, Yan M, Xu L, Zhang XH. A study of effectiveness of deep learning in locating real faults. Information and Software Technology, 2021, 131: 106486. [doi: 10.1016/j.infsof.2020.106486]

[11] Zhang Z, Lei Y, Tan QP, Mao XG, Zeng P, Chang X. Deep learning-based fault localization with contextual information. IEICE Trans. on Information and Systems, 2017, E100(12): 3027–3031. [doi: 10.1587/transinf.2017EDL8143]

[12] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 169–180. [doi: 10.1145/3293882.3330574]

[13] Li Y, Wang SH, Nguyen T. Fault localization with code coverage representation learning. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 661–673. [doi: 10.1109/ICSE43902.2021.00067]

[14] Lei Y, Mao XG, Zhang M, Ren JG, Jiang YH. Toward understanding information models of fault localization: Elaborate is not always better. In: Proc. of the 41st IEEE Annual Computer Software and Applications Conf. (COMPSAC). Turin: IEEE, 2017. 57–66. [doi: 10.1109/COMPSAC.2017.246]

[15] Zhang Z, Tan QP, Mao XG, Lei Y, Chang X, Xue JX. Effective fault localization approach based on enhanced contexts . Ruan Jian Xue Bao/Journal of Software, 2019, 30(2): 266–281 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5677.htm [doi: 10.13328/j.cnki.jos.005677]

[16] Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans. on Software Engineering and Methodology, 2011, 20(3): 11. [doi: 10.1145/2000791.2000795]

[17] Gong C, Zheng Z, Li W, Hao P. Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. Workshops. Izmir: IEEE, 2012. 470–475. [doi: 10.1109/COMPSACW.2012.89]

[18] Zhang L, Yan LF, Zhang ZY, Zhang J, Chan WK, Zheng Z. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. Journal of Systems and Software, 2017, 129: 35–57. [doi: 10.1016/j.jss.2017.04.017]

[19] Baudry B, Fleurey F, Le Traon Y. Improving test suites for efficient fault localization. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 82–91. [doi: 10.1145/1134285.1134299]

[20] He HB, Garcia EA. Learning from imbalanced data. IEEE Trans. on Knowledge and Data Engineering, 2009, 21(9): 1263–1284. [doi: 10.1109/TKDE.2008.239]

[21] Krawczyk B. Learning from imbalanced data: Open challenges and future directions. Progress in Artificial Intelligence, 2016, 5(4): 221–232. [doi: 10.1007/s13748-016-0094-0]

[22] Lei Y, Sun CN, Mao XG, Su ZD. How test suites impact fault localisation starting from the size. IET Software, 2018, 12(3): 190–205. [doi: 10.1049/iet-sen.2017.0026]

[23] Shorten C, Khoshgoftaar TM. A survey on image data augmentation for deep learning. Journal of Big Data, 2019, 6(1): 60. [doi: 10.1186/s40537-019-0197-0]

[24] Xian YQ, Lorenz T, Schiele B, Akata Z. Feature generating networks for zero-shot learning. In: Proc. of the 2018 IEEE/CVF Conf. on Computer Vision and Pattern Recognition. Salt Lake City: IEEE, 2018. 5542–5551. [doi: 10.1109/CVPR.2018.00581]

[25] Xian YQ, Sharma S, Schiele B, Akata Z. F-VAEGAN-D2: A feature generating framework for any-shot learning. In: Proc. of the 2019 IEEE/CVF Conf. on Computer Vision and Pattern Recognition. Long Beach: IEEE, 2019. 10267–10276. [doi: 10.1109/CVPR.2019.01052]

[26] Zhou FT, Huang S, Xing Y. Deep semantic dictionary learning for multi-label image classification. In: Proc. of the 35th AAAI Conf. on Artificial Intelligence. AAAI, 2021. 3572–3580. [doi: 10.1609/aaai.v35i4.16472]]

[27] Tantithamthavorn C, Hassan AE, Matsumoto K. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. on Software Engineering, 2020, 46(11): 1200–1219. [doi: 10.1109/TSE.2018.2876537]

[28] Xie XY, Chen TY, Kuo FC, Xu BW. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. on Software Engineering and Methodology, 2013, 22(4): 31. [doi: 10.1145/2522920.2522924]

[29] Sohn J, Yoo S. FLUCCS: Using code and change metrics to improve fault localization. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 273–283. [doi: 10.1145/3092703.3092717]

[30] Jabbar A, Li X, Omar B. A survey on generative adversarial networks: Variants, applications, and training. ACM Computing Surveys, 2021, 54(8): 157. [doi: 10.1145/3463475]

[31] Aissa FB, Mejdoub M, Zaied M. A survey on generative adversarial networks and their variants methods. In: Proc. of the 12th Int'l Conf. on Machine Vision. Amsterdam: SPIE, 2020. 1006–1012. [doi: 10.1117/12.2559848]

[32] De Rosa GH, Papa JP. A survey on text generation using generative adversarial networks. Pattern Recognition, 2021, 119: 108098. [doi: 10.1016/j.patcog.2021.108098]

[33] Harer JA, Ozdemir O, Lazovich T, Reale CP, Russell RL, Kim LY, Chin P. Learning to repair software vulnerabilities with generative adversarial networks. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 7944–7954.

[34] AlEroud A, Karabatis G. SDN-GAN: Generative adversarial deep NNS for synthesizing cyber attacks on software defined networks. In: Proc. of the 2020 OTM Confederated Int'l Conf. on the Move to Meaningful Internet Systems. Rhodes: Springer, 2020. 211–220. [doi: 10.1007/978-3-030-40907-4_23]

[35] Bao L, Liu X, Wang FZ, Fang BY. ACTGAN: Automatic configuration tuning for software systems with generative adversarial networks. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 465–476. [doi: 10.1109/ASE.2019.00051]

[36] Chouhan SS, Rathore SS. Generative adversarial networks-based imbalance learning in software aging-related bug prediction. IEEE Trans. on Reliability, 2021, 70(2): 626–642. [doi: 10.1109/TR.2021.3052510]

[37] Sun HW, Nie YP, Li X, Huang MH, Tian JW, Kong W. An automatic code generation method based on sequence generative adversarial network. In: Proc. of the 7th IEEE Int'l Conf. on Data Science in Cyberspace (DSC). Guilin: IEEE, 2022. 383–390. [doi: 10.1109/DSC55868.2022.00059]

[38] Krawczyk B, McInnes BT. Local ensemble learning from imbalanced and noisy data for word sense disambiguation. Pattern Recognition, 2018, 78: 103–119. [doi: 10.1016/j.patcog.2017.10.028]

[39] Defects4J. 2023. https://github.com/rjust/defects4j

[40] ManyBugs. 2019. https://github.com/squaresLab/ManyBugs

[41] SIR. 2019. http://sir.unl.edu/portal/index.php

[42] Bug_Location. 2021. https://github.com/oy-sarah/bug_location

[43] Abreu R, González A, Zoeteweij P, Van Gemund AJC. Automatic software fault localization using generic program invariants. In: Proc. of the 2008 ACM Symp. on Applied Computing. Fortaleza: ACM, 2008. 712–717. [doi: 10.1145/1363686.1363855]

[44] Jiang JJ, Wang R, Xiong YF, Chen XP, Zhang L. Combining spectrum-based fault localization and statistical debugging: An empirical

study. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 502–514. [doi: 10.1109/ASE.2019.00054]

[45] Zou DM, Liang JQ, Xiong YF, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Trans. on Software Engineering, 2019, 47(2): 332–347. [doi: 10.1109/TSE.2019.2892102]

[46] Wang T, Roychoudhury A. JSlice. 2008. http://jslice.sourceforge.net/

[47] Hammacher. JavaSlicer. 2016. https://github.com/hammacher/javaslicer

[48] WET. 2010. http://wet.cs.ucr.edu/

[49] Wang HF, Du B, He J, Liu Y, Chen X. IETCR: An information entropy based test case reduction strategy for mutation-based fault localization. IEEE Access, 2020, 8: 124297–124310. [doi: 10.1109/ACCESS.2020.3004145]

[50] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 199–209. [doi: 10.1145/2001420.2001445]

[51] Lei Y, Mao XG, Dai ZY, Wang CS. Effective statistical fault localization using program slices. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. Izmir: IEEE, 2012. 1–10. [doi: 10.1109/COMPSAC.2012.9]

[52] Debroy V, Wong WE, Xu XF, Choi B. A grouping-based strategy to improve the effectiveness of fault localization techniques. In: Proc. of the 10th Int'l Conf. on Quality Software. Zhangjiajie: IEEE, 2010. 13–22. [doi: 10.1109/QSIC.2010.80]

[53] Richardson A. Nonparametric statistics for non-statisticians: A step-by-step approach by Gregory W. Corder, Dale I. Foreman. Int'l Statistical Review, 2010, 78(3): 451–452. [doi: 10.1111/j.1751-5823.2010.00122_6.x]

[54] Zhang Z, Lei Y, Mao XG, Chang X, Xue JX, Xiong QY. Fault localization approach using term frequency and inverse document frequency. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3448–3460 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6021.htm [doi: 10.13328/j.cnki.jos.006021]

[55] Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE). Honolulu: IEEE, 2011. 1–10. [doi: 10.1145/1985793.1985795]

[56] Vargha A, Delaney HD. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 2000, 25(2): 101–132. [doi: 10.3102/10769986025002101]

[57] Jones JA, Bowring JF, Harrold MJ. Debugging in parallel. In: Proc. of the 2007 Int'l Symp. on Software Testing and Analysis. London: ACM, 2007. 16–26. [doi: 10.1145/1273463.1273468]

[58] Wong E, Wei TT, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization. In: Proc. of the 1st Int'l Conf. on Software Testing, Verification, and Validation. Lillehammer: IEEE, 2008. 42–51. [doi: 10.1109/ICST.2008.65]

[59] Japkowicz N, Stephen S. The class imbalance problem: A systematic study. Intelligent Data Analysis, 2002, 6(5): 429–449. [doi: 10.3233/IDA-2002-6504]

[60] Hao D, Pan Y, Zhang L, Zhao W, Mei H, Sun JS. A similarity-aware approach to testing based fault localization. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. Long Beach: ACM, 2005. 291–294. [doi: 10.1145/1101908.1101953]

[61] Yu YB, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. In: Proc. of the 30th ACM/IEEE Int'l Conf. on Software Engineering. Leipzig: IEEE, 2008. 201–210. [doi: 10.1145/1368088.1368116]

[62] Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic Internet services. In: Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks. Washington: IEEE, 2002. 595–604. [doi: 10.1109/DSN.2002.1029005]

[63] Jones JA. Fault localization using visualization of test information. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 54–56. [doi: 10.1109/ICSE.2004.1317420]

[64] Abreu R, Zoeteweij P, van Gemund AJC. An evaluation of similarity coefficients for software fault localization. In: Proc. of the 12th Pacific Rim Int'l Symp. on Dependable Computing (PRDC 2006). Riverside: IEEE, 2006. 39–46. [doi: 10.1109/PRDC.2006.18]

[65] Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJC. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009, 82(11): 1780–1792. [doi: 10.1016/j.jss.2009.06.035]

[66] Wong WE, Qi Y, Zhao L, Cai KY. Effective fault localization using code coverage. In: Proc. of the 31st Annual Int'l Computer Software and Applications Conf. (COMPSAC 2007). Beijing: IEEE, 2007. 449–456. [doi: 10.1109/COMPSAC.2007.109]

[67] Wong WE, Debroy V, Choi B. A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software, 2010, 83(2): 188–208. [doi: 10.1016/j.jss.2009.09.037]

[68] Wong WE, Debroy V, Li YH, Gao RZ. Software fault localization using DStar (D*). In: Proc. of the 6th IEEE Int'l Conf. on Software Security and Reliability. Gaithersburg: IEEE, 2012. 21–30. [doi: 10.1109/SERE.2012.12]

[69] Dean BC, Pressly WB, Malloy BA, Whitley AA. A linear programming approach for automated localization of multiple faults. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Auckland: IEEE, 2009. 640–644. [doi: 10.1109/ASE.2009.54]

[70] Abreu R, Zoeteweij P, van Gemund AJC. Localizing software faults simultaneously. In: Proc. of the 9th Int'l Conf. on Quality Software. Jeju: IEEE, 2009. 367–376. [doi: 10.1109/QSIC.2009.55]

[71] Abreu R, Zoeteweij P, van Gemund AJC. Simultaneous debugging of software faults. Journal of Systems and Software, 2011, 84(4): 573–586. [doi: 10.1016/j.jss.2010.11.915]

附中文参考文献:

[15] 张卓, 谭庆平, 毛晓光, 雷晏, 常曦, 薛建新. 增强上下文的错误定位技术. 软件学报, 2019, 30(2): 266–281. http://www.jos.org.cn/1000-9825/5677.htm [doi: 10.13328/j.cnki.jos.005677]

[54] 张卓, 雷晏, 毛晓光, 常曦, 薛建新, 熊庆宇. 基于词频-逆文件频率的错误定位方法. 软件学报, 2020, 31(11): 3448–3460. http://www.jos.org.cn/1000-9825/6021.htm [doi: 10.13328/j.cnki.jos.006021]

张卓(1984－), 男, 博士, 主要研究领域为软件错误定位, 软件自动修复, 智能软件工程.

薛建新(1980－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为并发理论, 程序分析.

雷晏(1985－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件错误定位, 软件自动修复.

常曦(1979－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为程序测试, 程序分析.

毛晓光(1970－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为可信软件, 软件维护与演化.