计算机科学
COMPUTER SCIENCE

AutoUnit:基于主动学习和预测引导的测试自动生成

张大林，张哲玮，王楠，刘吉强

引用本文

张大林，张哲玮，王楠，刘吉强. AutoUnit:基于主动学习和预测引导的测试自动生成[J]. 计算机科学, 2022, 49(11): 39-48.

ZHANG Da-lin, ZHANG Zhe-wei, WANG Nan, LIU Ji-qiang. AutoUnit:Automatic Test Generation Based on Active Learning and Prediction Guidance[J]. Computer Science, 2022, 49(11): 39-48.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220200086

AutoUnit:基于主动学习和预测引导的测试自动生成

张大林[1]　张哲玮[2]　王　楠[1]　刘吉强[1]
1 北京交通大学软件学院　北京 100044
2 北京交通大学计算机与信息技术学院　北京 100044
(dalin@bjtu.edu.cn)

摘　要　测试用例自动生成技术旨在降低测试成本,与人工生成测试用例相比,它具有更高的测试效率。现有主流的测试工具对软件中的所有文件都平等对待,但是大多数情况下含有缺陷的文件只占整个软件项目的一小部分。因此,如果测试人员能针对更易存在缺陷的文件进行测试,就能极大地节省测试资源。针对以上问题,文中设计了一种基于主动学习的预测引导的自动化测试工具 AutoUnit。首先对待测文件池中的所有文件进行缺陷预测,然后对最"可疑"的文件进行测试用例生成,之后将实际测试用例执行结果反馈给缺陷预测模型并更新该预测模型,最后根据召回率判断是否进入下一轮测试。此外,AutoUnit 还能在含缺陷文件总数未知时,通过设置不同的目标召回率来及时停止预测引导。它能依据已测文件来预测含缺陷文件总数并计算当前召回率,判断是否停止预测引导,保证测试效率。实验分析表明,当测得相同数量的缺陷文件时,AutoUnit 花费的最短时间为目前主流测试工具的 70.9%,最长时间为目前主流测试工具的 80.7%;当含缺陷文件总数未知且目标召回率设置为 95%时,与最新版本的 Evosuite 相比,AutoUnit 只需要检查 29.7%的源代码文件就能达到相同的检测水平,且其测试时间仅为 Evosuite 的 34.6%,极大地降低了测试成本。实验结果表明,该方法有效地提高了测试的效率。

关键词:测试用例生成;基于池的主动学习;缺陷预测模型;随机测试;测试效率
中图法分类号　TP391

AutoUnit:Automatic Test Generation Based on Active Learning and Prediction Guidance

ZHANG Da-lin[1],ZHANG Zhe-wei[2],WANG Nan[1] and LIU Ji-qiang[1]
1 School of Software Engineering,Beijing Jiaotong University,Beijing 100044,China
2 School of Computer and Information Technology,Beijing Jiaotong University,Beijing 100044,China

Abstract　Automated test case generation technology aims to reduce test costs. Compared with manual test generation,it has higher test efficiency. Most existing testing tools treat all files in the software equally,but in fact,files with defects account for only a small part of the whole code. Therefore,if testers can detect files that are more prone to defects,they can greatly save testing resources. To solve the above problems,this paper designs a predictive guidance test tool AutoUnit,which is based on active learning. We first predict the defect files in the whole file pool to be detected. Next,we use the detection tool to detect the most "suspicious" files. Then we feed back the actual detection results to the prediction model and update the model to enter the next round of prediction. In addition,when the total number of defective files is unknown,AutoUnit can stop in time by setting different target recall rates. It can predict the total number of defective files according to the tested files,calculate the current recall rate,judge whether to stop predict guidance and ensure testing efficiency. Experimental analysis shows that when the same number of defect files are tested,the shortest time and the longest time taken by AutoUnit is 70.9% and 80.7% of the current mainstream testing tools,respectively. When the total number of defective files is unknown and the target recall rate is set to 95%,compared with the latest version of Evosuite,AutoUnit only needs to check 29.7% of the source code files to achieve the same detection level,and its test time is only 34.6% of Evosuite,the cost of testing is greatly reduced. Experimental results show that the method effectively improves the efficiency of test.

Keywords　Test case generation,Pool-based active learning,Defect predict model,Random test,Detection efficiency

到稿日期:2022-02-16　返修日期:2022-05-19
基金项目:中央高校基本科研业务费专项资金(2021QY010)
This work was supported by the Fundamental Research Funds for the Central Universities of Ministry of Education of China(2021QY010).
通信作者:张哲玮(20120452@bjtu.edu.cn)

1 引言

软件测试的一条原则是越早发现错误越好，修复软件发布后发现故障的成本比修复开发过程中的故障要大得多[1]。但是，对于大型的工程项目，在软件发布后发现的缺陷数量经常远超过预期。其主要原因如下：1）大型项目的开发时间较长，整个研发周期中留给测试的时间并不充裕；2）随着软件规模的不断扩大，测试的成本也越来越高，测试人员往往无法检查所有的源代码文件；3）在测试过程中，测试人员只能凭经验在众多项目文件中决定要检查的源代码文件，并且检查的数量也只能由测试人员自行判断。因此，如何在有限资源条件下优化软件测试资源分配并实现软件测试收益最大化，成为了软件开发的一个重要课题。

目前已经有大量的研究工作用于解决测试资源分配问题，其思路是将更多的精力分配给预计包含更多缺陷的软件模块，即通过构建所谓的缺陷倾向性模型来决定如何分配测试资源。这些模型旨在通过利用过程或产品度量作为分类器或回归器的特征（例如代码级度量，包括圈复杂度、代码行、面向对象的度量，来自 CVS/SVN/Git 存储库的文件级指标、设计指标、需求指标等），并在具有已知缺陷数量的实例中训练分类器或回归器，使用经过训练的模型来预测故障数量未知的模块中的故障倾向[2]。传统的缺陷预测方法大多依赖于软件度量指标，通常无法区分具有不同语义的程序[3]。现有的缺陷预测方法使用机器学习来构建缺陷预测模型，在识别缺陷方面更加有效[4]。

许多研究人员通过有监督机器学习建立起了很多性能优越的软件缺陷预测模型，这些缺陷预测模型的预测结果取决于先前的缺陷信息。然而，此类信息不适用于新的项目，并且在实际应用中通常无法获取到足够的带缺陷标记的模块信息[5]。一些研究人员利用主动学习这种半监督的方法建立了软件缺陷预测模型，但现有的主动学习方法大多是从一个数据库中直接获得样例的分类结果，忽略了样例的标注问题[6]。这种简化设置下的主动学习虽然得到了广泛的研究，但并不符合实际。因此，将缺陷预测引导与实际模块测试相结合具有重要的价值。

本文提出了一种基于主动学习和预测引导的随机测试工具 AutoUnit，它能通过缺陷预测选出部分文件进行重点检测，而对剩余未被选中的文件则进行快速的检测，将测试资源更合理地分配到那些含缺陷概率更大的文件中，在保证检测精度的同时大幅提高检测效率。此外，AutoUnit 可以手动设置召回率，以决定何时停止预测引导，在实际测试中缺陷文件数未知的情况下，它可以根据已测得的结果预测剩余的缺陷文件数量，以获得当前的召回率。达到目标召回率后，AutoUnit 会停止缺陷预测引导，并对剩余的所有未检测文件进行简单快速的检测。

AutoUnit 使用主动学习的方法将缺陷预测与随机测试相结合，对可能含有缺陷的模块投入足够的测试资源。在预测环节结束后，它会按缺陷预测结果将含缺陷概率最大的文件添加到测试队列中，使用现有的自动化测试工具为队列中的文件生成测试用例并执行这些测试用例，最后将结果信息反馈给缺陷预测模型，更新缺陷预测模型并进入下一轮测试。这种反馈机制使得缺陷预测模型每轮都能根据实际结果来更新模型，因此模型的预测结果能很快地向含缺陷文件收敛。AutoUnit 还设计了一种停止策略，当它判定测得的缺陷文件数量已经达到预期目标时便停止测试，保证了测试效率。总而言之，本文的主要贡献如下：

（1）为了提高测试效率，本文设计了一种基于预测引导的自动化随机测试工具 AutoUnit。通过构建基于主动学习的缺陷预测模型，将测试资源分配给含缺陷概率更大的文件。

（2）AutoUnit 每一轮能针对少量筛选文件进行随机测试，并根据测试结果丰富文件特征维度，实现预测模型的主动更新。

（3）当含缺陷文件总数未知时，AutoUnit 可以利用测试结果文件来预测含缺陷文件的总数，并通过计算当前的召回率来判断是否停止预测引导，从而保证测试效率。

为了检验 AutoUnit 的性能，本文从 Java 公开缺陷数据集中选取了 16 个工程，并与主流的两种 Java 自动化测试工具 Randoop 和 Evosuite 进行对比。实验结果表明，本文方法在保证测试精度的同时显著提高了测试的效率。

2 相关工作

2.1 基于引导的测试用例生成

现有的基于引导的测试用例生成技术从引导技术角度可归纳为：基于测试结果反馈引导和基于预测引导。

2.1.1 基于测试结果反馈引导

使用测试结果来引导生成测试用例是最常见的基于引导的测试用例生成技术。例如，Pacheco 等[7]提出了一种反馈导向的随机测试方法。该方法在构建输入序列时使用执行序列时的结果反馈，不再将已有的或引起异常状态的测试用例添加到序列中，以此指导生成新的且合法的序列，避免生成重复和非法的输入数据。此外，大部分测试方法都是以达到更高的覆盖率为目标来引导测试用例生成。基于搜索的测试用例生成方法使用元启发式搜索算法（通常使用遗传算法[8]），通过捕获测试目标的执行条件，使随机生成的初始用例不断地进化，取达到最高的覆盖率的结果作为测试用例[9]。基于符号执行的模糊测试技术利用符号执行方法来生成输入，以达到新的路径[10]。例如，白盒模糊测试工具 SAGE[11]，它使用符号执行来收集条件语句的路径约束，因此能覆盖到更深层的路径。Dower[12]是一个将污点跟踪、程序分析和符号执行相结合来引导检测缓冲区溢出漏洞的测试工具。动态符号执行[13]也是生成高覆盖率测试用例的有效方法，它通过使用具体值的实际执行与符号化执行并行地为被测代码生成具体的测试输入，迭代地探索被测代码中的新路径[14]，从而系统地增加被测代码的块或提高分支覆盖率。

2.1.2 基于预测引导的测试生成

评估测试用例自动生成最主要的指标是代码覆盖率。一般来说，更高的代码覆盖率意味着可能找到更多的缺陷，但达到高覆盖率需要大量的时间和计算资源，这意味着测试成本会随着软件规模的增加而呈指数式上升。Shin 等[15]在对 Mozilla Firefox 源代码文件的研究中发现，引起异常的代码

仅仅占总代码量的3%，即大部分文件是正常的。如果只对这一部分少量的异常代码生成测试用例，则测试成本会极大地降低。因此，基于预测引导的测试生成更多地是从减少测试文件数量、缩小测试范围这两方面来降低测试成本。为了达到这一目的，Li等[16]设计了一种由漏洞导向的模糊测试工具V-Fuzz。该工具使用图嵌入神经网络来训练预测模型，通过模型来判断软件中更易受攻击的部分，并使用一种进化算法来定向地生成测试用例，以到达预测结果所确定的位置。Perera等[17]使用缺陷预测来引导测试生成，他们使用项目版本的更改信息（时间戳）构建了一个缺陷预测模型，通过计算不同文件的含缺陷概率为各个文件分配相应的生成测试用例的时间。

2.2 主动学习

主动学习（Active Learning）是一种迭代的半监督学习过程，其核心思想是从大量未标注的样本中寻找质量高的样本进行人工标注，再将其加入训练集中进行学习[18]。由于只有少部分样本需要人工标注，因此能够显著降低标注成本，也能提升分类器的性能。同时，根据设计的选择策略，可以防止选择的样本与已有训练样本重复，进一步提升训练样本集的质量，因此可用于构建高性能分类器或对数据集进行标记。

主动学习的整个过程可以概括为两个部分：学习过程和选择过程[19-20]。如图1所示，主动学习在缺陷预测领域的学习过程使用分类算法将项目文件分为含缺陷文件和正常文件；选择过程负责在项目文件库中选择待标记样本集，然后对选中的文件进行测试，根据测试结果进行标记。样本文件的选择和标记策略决定着最终主动学习的效果。

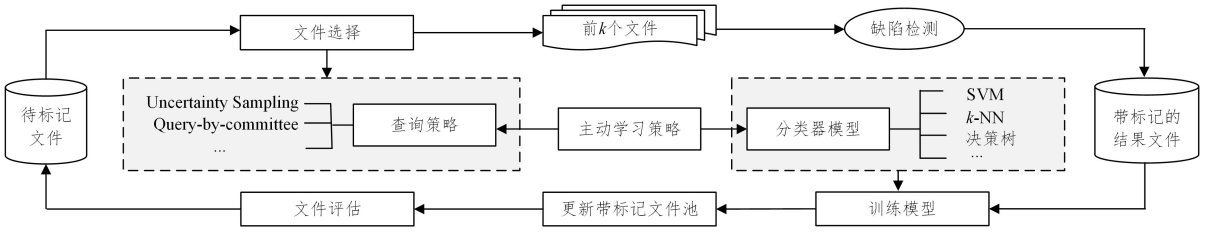

图1 主动学习驱动的缺陷检测框架

Fig. 1 Active learning-driven defect detection framework

根据样本文件的选择方法将主动学习分为两类：基于流的主动学习算法和基于池的主动学习算法。

（1）基于流的主动学习算法依次从待标记样本集中取出一个样例输入到选择模块，若满足预设的选中条件，则对其进行人工标记，反之直接舍弃。Chu等[21]将其应用于过滤商业垃圾邮件和筛选有价值的评论等领域。但是该学习过程需处理所有未标记样例，查询成本高昂。Mohamad等[22]提出了一种改进的方法，通过忽略在原特征区间内发生概念漂移的样本，在降低查询成本的同时保证了模型的精度，并将其应用于文本分类和图像识别。Liu等[23]提出了一种针对具有概念漂移的多类不平衡流数据的综合主动学习方法，用于解决图像分类问题。但是，由于基于流的样例选择策略需要预设一个样例标注条件，该条件往往需要根据不同的任务进行适当调整，因此很难将其作为一种通用方法使用。

（2）基于池的主动学习算法每次从动态更新的待标记样本集中按预设的选择规则选取一个样例交给基准分类器进行识别，当基准分类器对其识别出现错误时进行人工标注。相比基于流的主动学习的方法，基于池的方法每次都可以选出当前样例池中对分类贡献度最高的样例，这既降低了查询样例成本，也降低了标注代价，使得基于池的样例选择策略得到广泛使用。Sinha等[24]应用其进行图像分类和语义分割。Wu[25]提出了一种基于池的回归主动学习（Active Learning for Regression，ALR）方法，并将其应用于声学情感计算。Beluch等[26]使用基于高维数据和卷积神经网络分类器的主动学习方法进行图像分类。

2.3 缺陷预测

缺陷预测模型可用于将测试工作定向到容易出现缺陷的代码文件，然后可以在软件项目交付前检查代码中的潜在缺陷，且在交付前解决这些缺陷的成本远远小于交付后再解决这些缺陷。与软件测试和人工审查相比，软件缺陷预测方法在检查软件缺陷方面更具成本效益[27]。因此，有效的缺陷预测技术在指导测试工作、降低测试成本以及进一步提高软件质量方面有重要意义。

缺陷预测模型一般由4个主要元素组成：自变量、因变量、预测模型以及评价指标。下文是对这4个元素的详细说明。

（1）自变量。各种自变量对预测性能的影响是许多研究工作的主题。以往研究中使用的自变量主要分为软件指标（如静态代码数据）和历史信息（如以前的更改和缺陷数据）以及与开发人员相关的指标。静态代码特征是缺陷倾向预测初期比较有用的一般指标，但是根据代码静态特征分类的结果不能满足实际需求。而历史信息，如代码更改信息，以往的错误报告和崩溃信息是最好的预测器[28]，例如将代码的更改次数以及相关开发人员数量添加到自变量中能显著提高预测性能，但是这种信息并不稳定，无法作为预测的主要依据。许多研究工作转向使用其他的自变量，例如Mizuno等[29]使用源代码本身的文本特征作为自变量，其效果十分出色。

（2）因变量。文件的预测结果通常以评分的形式给出，其反映了该文件含缺陷的概率。对于缺陷预测引导的测试生成，只需要对含缺陷概率超过阈值的文件生成测试用例而无须关心其他文件，从而提高测试效率。

（3）预测模型。容易出现缺陷的软件文件的识别通常通过二元预测模型来实现，该模型将文件分为有缺陷或无缺陷两种类型，因此缺陷预测经常使用可训练的分类器。这种分类器通过训练数据（自变量）来构建模型，在以往的研究中已经使用了很多分类技术，支持向量机（Support Vector Ma-

chines,SVM)是最常用的一种。作为一种高效的分类器模型,SVM 经常用于解决机器学习和数据挖掘领域中的分类和回归问题。

(4)评价指标。由于缺陷预测是一个二分类问题,可以利用混淆矩阵来评估模型性能,即将样本根据其真实类别与模型预测类别的组合划分为真正例(True positive,TP)、假正例(False Positive,FP)、真反例(True Negative,TN)和假反例(False Negative,FN) 4 种情况。通过这种划分方式,我们可以得到缺陷预测结果的混淆矩阵,进一步将其组合为召回率、准确率等评价指标。

3 架构描述

3.1 缺陷预测问题

一般来说,在软件项目中,缺陷文件数量只占项目文件总数的一小部分。但是目前绝大多数的测试方法都是针对整个项目进行测试,对每个源代码文件都生成测试用例造成了很多浪费。尤其是当项目规模较大、复杂度较高时,测试人员更希望把大部分精力集中在关键文件或更易存在缺陷的文件(如核心算法代码所在的文件、数据库的读写操作所在的文件等)上。这类文件的数量往往不会太多,但是也衍生出了一个问题:如何认定一个文件是否关键(潜在含有缺陷)?在实际测试中,测试人员通常会选择大量的相关文件来确保不会"漏掉"某些关键文件,这也是难以降低测试成本的主要原因。对此,AutoUnit 构建了一个基于主动学习的缺陷预测模型,从整个项目中选出含缺陷概率最大的少量文件进行重点测试,尽可能地将更多的测试资源分配给那些含缺陷概率更高的文件。如图 2 所示,预测模型所选择的需要重点检查的文件数量只占文件总数的一部分,在随机测试时也只需要这一部分文件进行针对性测试,其余文件可在预测引导结束后在较短的时间内进行快速测试并输出结果。

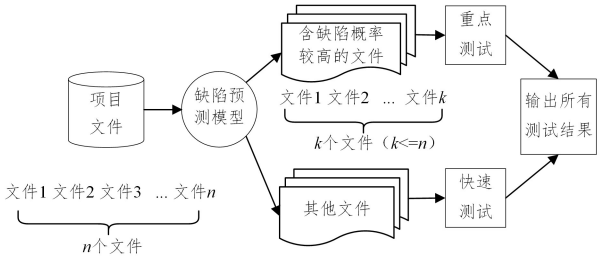

图 2 基于缺陷预测引导的测试过程

Fig. 2 Guided testing based on defect prediction

缺陷预测的核心问题是如何在检查最少的代码的情况下找到大多数缺陷。因此,可以将缺陷预测问题概括为总召回(Total Recall)问题。总召回问题的目标是优化成本,在循环运行的过程中由人工标记数据,实现了极高的召回率(理想情况下非常接近 100%)[30]。

本文中测试过程中的各类文件的变量表示如下:

(1)E 表示所有的项目文件集合;

(2)R 表示所有含缺陷的文件集合,每个文件至少含有一个由缺陷引起的 bug;

(3)L 表示已测试文件集合;

(4)L_R 表示被测文件中含缺陷文件集合,即 $L \cap R$;

(5)L_I 表示被测文件中无缺陷文件集合,即 $L - L_R$。

则总召回问题可以被描述为:在给定集合 E 及其一个子集 R 的情况下,不断地从 E 中筛选元素添加到另一个集合 L 中(初始为 $\varnothing$),求出 L 中的元素个数最少且 $L \cap R$ 中的元素个数最多的情况。

此外,考虑到漏掉缺陷可能带来的严重后果,本文认为将缺陷预测问题视为总体召回问题而不是分类问题更合适。因此,本文使用了一种基于主动学习的框架来解决缺陷预测问题,该框架已经被证明在应用于其他总召回问题时表现良好。

3.2 缺陷预测引导的测试

在测试任务中,AutoUnit 有两个主要目标:1)通过缺陷预测来指导测试,AutoUnit 从源文件的代码中提取特征,根据有缺陷的文件和无缺陷的文件的特征不同这一特点,选择出存在缺陷的概率更大的文件生成测试用例。根据已有的预测模型构建技术[31],本文应用了一种主动学习的方法来构建缺陷预测模型,其核心是一个不断更新的支持向量机(SVM);2)通过随机测试结果来实时调整 AutoUnit 的预测模型,对于测试完毕的文件,其测试结果的揭错部分(该部分揭示了文件中是否含有缺陷)成为了原文件的标签,这一部分文件数据使预测模型能实时地根据实际结果进行更新。此外,测试结果的其他部分(如代码覆盖率)会被添加到原文件的特征中,具体过程见 4.6 节中的详细描述。AutoUnit 还能根据现有的所有测试结果判断剩余的待测文件中还存在的含缺陷的文件数量,使 AutoUnit 能在含缺陷文件总数未知的情况下及时结束测试。该停止策略的核心是,用 Logistics 回归对剩余的文件进行暂时标注,对含缺陷文件总数进行预估,计算临时召回率,判断是否停止测试。

本文在现有的主动学习的技术基础上,结合文本挖掘的方法,构建了 AutoUnit 的缺陷预测部分。项目文件会被标记为两种类型,即阴性(Negative)和阳性(Positive),分别表示该文件的预测结果为无缺陷和有缺陷,若文件被标记为阳性,则下一步 AutoUnit 会对该文件进行随机测试。AutoUnit 使用 SVM 模型学习之前测试的结果,对其他待测文件进行标记。这种基于主动学习的缺陷预测模型避免了在大量无缺陷文件上浪费检查工作,并利用实时的测试结果来更新模型,使得预测结果向含缺陷文件加速收敛,其大致流程如图 3 所示,各个环节的实现方式如下。

(1)特征提取。从源代码文件中提取文本特征作为每个文件的特征向量。将待测文件集合 H 初始化为 E,将已测文件集合 L 初始化为 $\varnothing$,将已测得的缺陷文件集合 L_R 也初始化为 $\varnothing$。

(2)初始采样。在模型训练尚未开始之前选择文件进行测试,每次选择 N_1 个文件生成测试用例,根据测试结果对文件进行标记。当至少出现一个阳性文件时停止采样,并在更新后的已测文件集合 L 和 L_R 上构建缺陷预测模型。

(3)训练模型。将最新的测试结果添加到对应文件的特征中,在更新后的 L 上训练一个分类器作为预测模型。

(4)缺陷预测。在集合 H 上应用已构建好的缺陷预测模型,选择 N_2 个预测结果中含缺陷概率最高的文件,将其添加

到队列 Q 中,之后在 H 中删除这些文件,防止重复选取。

(5)测试生成。应用自动化测试工具对队列 Q 中的文件生成测试用例并执行,根据结果文件的揭错部分将文件标记为阳性或阴性。测试完毕后,将文件从 Q 中取出并放入 L 中,若文件为阳性,则再将其复制到 L_R 中。统计结果文件的其他信息,并将其添加到该文件的特征向量中参与训练。

(6)召回率估计。预测文件总集 E 中的阳性文件数量 R_E,若有 $|L_R|/|R_E| \geqslant T_{\text{rec}}$($T_{\text{rec}}$ 为设定的目标召回率),则结束缺陷预测引导,进入快速测试环节,否则跳转到步骤(3)进入下一轮测试生成。

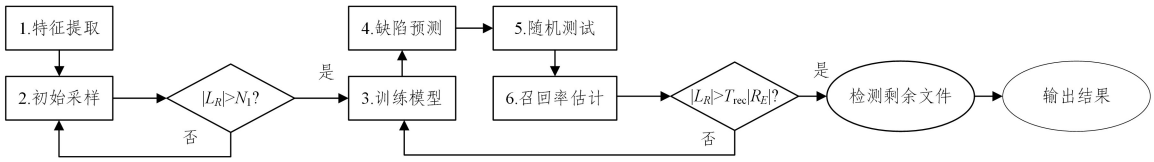

图 3 AutoUnit 测试流程

Fig. 3 AutoUnit test process

4 AutoUnit

4.1 特征提取

从项目文件中提取出来的各种特征都可用于 AutoUnit 的缺陷预测部分。在实践中,AutoUnit 会根据实际被测工程实践效果,来决定选取作为缺陷预测输入的特征种类。对于代码文件,主要有以下 3 类特征。

(1)软件静态特征

软件的静态特征包括代码行数、函数的个数、分支个数等各项度量信息。该类信息不需要编译文件,可直接从源代码文件中提取。本文选取的静态特征如下:

1)*CountLineCode* 表示每个文件的代码行数;

2)*CountDeclClass* 表示声明的类的个数;

3)*CountDeclMethod* 表示声明的方法的数量;

4)*CountDeclExecutableUnit* 表示可执行单元总数;

5)*CountDeclInstanceVariable* 表示实例变量总数;

6)*SumCyclomatic* 表示圈复杂度总和;

7)*MaxCyclomatic* 表示最大圈复杂度;

8)*SumEssential* 表示基本复杂度总和;

9)*MaxNesting* 表示控制结构的最大嵌套级别;

10)*RatioCommentToCode* 表示代码注释率。

(2)历史版本缺陷信息

项目在早期调试或检测过程中会出现一批含缺陷的文件,即使在修复之后这些文件含有缺陷的可能性也会比其他文件更大。本文收集了原数据集不同历史版本的缺陷信息作为该类特征。

(3)文本特征

1)词频矩阵。词频矩阵是最基本的文本特征,它通过统计各个文档内单词的出现频率,来构成单词-文件式的矩阵。但是,当文档数量过大、文档内容复杂时,会出现维度爆炸的情况。为了避免这一情况,本文在构建词频矩阵时,选取出现频率最高的前 M 个单词作为基准,计算每个单词在各个文件中出现的次数,最后将其归一化,从而得到最终的特征矩阵。

2)词频-逆文档频率(Term Frequency-inverse Document Frequency,TF-IDF)是由单词在各个文档中出现的频率决定的,单个或少部分文档中常见的单词往往比冠词和介词等常见单词具有更高的 TF-IDF 值[32]。对于第 t_i 个单词,其 TF-IDF 得分为:

$$Tfidf(t_i)=\sum_{d\in D} Tfidf(t_i,d) \tag{1}$$

其中,D 代表项目的所有文件,对于文件 d 中的单词 t_i 来说:

$$Tfidf(t_i,d)=w_d^{t_i}\times\left(\log\frac{|D|}{\sum\limits_{d_i\in D}\operatorname{sgn}(w_{d_i}^{t_i})+1}\right) \tag{2}$$

其中,$w_d^{t_i}$ 是单词 t_i 在文件 d 中出现的次数。

计算得到 TF-IDF 得分后,选出得分最高的 M 个单词(与词频矩阵相同),对于某一个项目文件,这 M 个单词的 TF-IDF 得分即为该文件的特征,并使用 L_2-norm 标准化每个特征向量。这里,本文根据文件的数量来决定 M 的大小(根据相关文献,本文将 M 的默认值设置为 2000)。

4.2 初始采样

AutoUnit 在首次选择文件时有两种不同的策略。

(1)随机采样

在待测文件池中随机抽取 N_1 个文件进行测试,直到发现第一个含缺陷的文件。该方法的随机性较大,在缺陷数量较少时可能在排除大量文件后才能找到首例。

(2)给定初始样例

首先人工标记一个含有缺陷的文件,在随机抽取的基础上将该文件加入首次测试的队列中(即随机选取的文件个数变为 N_1-1),加快了学习过程,也解决了随机采样的不稳定性。

4.3 模型训练

缺陷预测的目标是对文件进行分类并标记,AutoUnit 使用 SVM 来构建预测模型。SVM 的主要思想是将输入数据映射到高维空间,并训练一个超平面作为最优决策平面,它可以对两类数据进行分类,同时最大化点到该超平面之间的距离[33]。每个项目文件由其文本特征加测试结果数据组成的特征向量表示,该向量所对应的点到超平面的距离是缺陷预测的重要指标。选择 SVM 作为分类器,是因为其已被证明是一种高效的分类器模型,并已经被应用于各种缺陷预测方法[34-35]。此外,AutoUnit 使用了积极采样(Aggressive Sampling)技术来平衡训练数据,它会丢弃最接近 SVM 决策平面的多数 L_I 对应的训练数据,只保留达到相同数量的少数 L_R 对应的训练数据。仅当 L_R 中的文件数量达到一定程度时,才进行积极采样,它能使 AutoUnit 在更快地构建更好的模型与更多地应用模型以节省检查工作之间保持平衡。算法 1 描述了模型的具体训练过程,包括计算文件特征和训练模型两个环节。

(1)计算文件特征

对于在上一轮测试完毕的文件,将其测试结果中的各项指标添加到它的文本特征中,得到其最终用于训练模型的特征向量。算法1中第3-6行描述了文件的特征向量的计算过程,假设文件的TF-IDF评分向量为 $\boldsymbol{x}=[x_1,x_2,x_3,\cdots]$,从测试结果中抽取的特征向量为 $\boldsymbol{y}=[y_1,y_2,y_3,\cdots]$,则文件最终的特征向量为 $\boldsymbol{z}=[\boldsymbol{x},\boldsymbol{y}]$。

(2)训练模型

模型的训练方法是在已测得的所有文件 L 所对应的特征矩阵 $\boldsymbol{F}$ 上训练一个软边界SVM分类器,将文件分为negative(无缺陷)和positive(有缺陷)两种类型。算法1中第8-19行描述了模型的训练过程,当 $t_p > t$ 时应用积极采样技术平衡训练数据,在计算阳性文件特征向量到决策平面的距离时,本文加入了代码覆盖率 Cov 作为可信度权重。若代码覆盖率越高,则该文件的标签的可信度越高。之后在新的已测文件集合 L' 对应的特征矩阵 $\boldsymbol{F}'$ 上训练决策平面。

算法1 缺陷预测模型训练算法

输入:已测文件集合L,测得阳性文件集合 L_R,测得阴性文件集合 L_I,初始特征矩阵 **FI**,测试结果矩阵 **UR**,上一轮的测试队列 Q

输出:缺陷预测模型 DPM

```
1. t_p = L_R的文件数量 /* 第1步:将文件的测试结果与初始特征结合,转化为用于训练缺陷预测模型的特征 */
2. for each x∈Q do:
3.    F(x) ← UR(x).extend(FI(x)) /* F为文件x最终的特征矩阵 */
4. end for
5. DPM ← SVM(F,kernel=linear) /* 第2步:更新已测试文件的最终特征矩阵F后,训练缺陷预测模型DPM */
6. if t_p > t then /* 应用积极采样技术平衡训练数据 */
7.    for i in L_I /* 计算L_I中各文件的特征向量到SVM平面的距离DL */
8.       DL'(i) = DL(i)×Cov(i) /* Cov是文件的测试结果中覆盖率的相关函数 */
9.    end for
10.   L_I' ← argsort(DL'(i))[0:t_p]
11.   L' ← L.extend(L_I')
12.   F' ← F.select(L') /* 从F中提取出L'对应的特征矩阵F' */
13.    DPM ← SVM(F',kernel=linear)
14. end if
```

4.4 缺陷预测

应用训练好的缺陷预测模型,在待测试文件集合 H 对应的特征矩阵上进行预测,选出 N_2 个含缺陷概率最大的文件添加到队列 Q 中等待测试,并将这些文件从 H 中删除。通过测试的结果来更新缺陷预测模型的参数,这种实时更新策略让缺陷预测模型能更快地达到较为准确的状态,在多轮训练后,缺陷预测模型选出的文件中含有缺陷文件的概率逐渐增大。

4.5 测试生成

本文在该环节扩展了现有的随机测试工具Evosuite,来检测缺陷预测模型选择出来的文件。在该环节中,AutoUnit为队列 Q 中的文件生成测试用例并执行。AutoUnit将队列 Q 中的文件按含缺陷概率大小进行排序,以测试用例生成时间作为测试资源,为含缺陷概率更大的文件分配更多的测试用例生成时间,整个队列中的文件测试时间在 $(t-t',t+t')$ 区间内浮动。最短不少于未被选中文件的测试时间。

文件测试结果的揭错部分会作为该文件的标签,参与缺陷预测模型的训练。而测试结果的其他部分如覆盖率等信息,则作为文件的特征向量的补充。对于某个测试文件 f,其内部包含函数 $(f_1,f_2,f_3,\cdots,f_n)$,这些函数的测试信息(如生成的测试用例个数、通过的测试用例个数、方法覆盖率等)构成了4.3节中从文件测试结果中抽取的特征向量 $\boldsymbol{y}_F=[y_1,y_2,y_3,\cdots]$。本文从测试结果中提取的信息如下:

(1)Line表示行覆盖率;

(2)Branch表示分支覆盖率;

(3)Excepted表示异常覆盖率;

(4)WeakMutation表示弱变异覆盖率;

(5)Output表示输出覆盖率;

(6)Method表示方法覆盖率;

(7)MethodExcepted表示方法异常覆盖率;

(8)Cbranch表示分支结构覆盖率;

(9)CountTest表示生成的测试方法数量;

(10)MutationScore表示测试套件的变异评分;

(11)Fitness表示最佳适应度评分;

(12)Defects表示缺陷密度。

对 $\boldsymbol{y}_F$ 进行归一化,之后与初始特征向量 $\boldsymbol{x}_F$ 一同组成文件 f 的特征向量 $\boldsymbol{z}_F$。将 F 的测试结果的揭错部分(即 f 中检测到的缺陷个数)记为 $\boldsymbol{l}_F$。则将带标记的训练数据 $(\boldsymbol{z}_F,\boldsymbol{l}_F)$ 作为测试的最终结果返回给缺陷预测模型,参与下一轮的训练过程。

4.6 召回率预测

根据已经测得的结果(包括本轮结果)对剩余待测文件进行估计,并根据召回率判断是否停止测试。AutoUnit构建了一个Logistic回归模型对未测文件 H 进行临时标记,若标记后所得到的召回率高于预设值,则结束缺陷预测引导;否则返回算法2中的步骤3进行下一轮测试生成。算法2描述了预测当前阳性文件总数量 R_E 的计算过程。

算法2 缺陷文件总数预估算法

输入:总文件集合E,待测文件集合H,测得阳性文件集合 L_R,测得阴性文件集合 L_I,缺陷预测模型 DPM

输出:预测的阳性文件总数量 R_E

```
1. t_p = length(L_R), t_p' ← 0
2. for each x∈E do /* 第1步:确定初始训练数据DL与其标签TL */
3. DL(x) ← SVM.decision(x)×Cov(x) /* 以代码覆盖率为置信度,计算x到SVM平面的距离 */
4.    if x∈L_R then
5.       TL(x) ← positive /* 将x代表的文件标记为阳性 */
6.    else
7.       TL(x) ← negative/* 将x代表的文件标记为阴性 */
8.    end if
9. end for
10. while t_p ≠ t_p' do /* 第2步:预测阳性文件总数量R_E */
11.       LReg ← LogisticRegression(DL,TL)
12.       TL ← TemporaryLabel(LReg,H,TLR)/* 应用Logistic回归模型对H上的文件进行暂时标注 */
```

13. $t_p' = t_p$

14. $t_p \leftarrow$ length(TLR)

15. end while

得到当前阳性文件总数量 R_E 后，可计算当前的召回率并与目标召回率（T_{rec}）进行比较，若达到目标召回率，则停止缺陷预测引导，进入较小时间代价的快速测试环节。快速测试环节是将剩余的所有未测文件整合，把每个文件的测试时间设置为最小值依次进行快速测试，最后输出所有的测试结果。我们设置快速测试模块的目的是在测试效率与文件覆盖率之间取得平衡。

AutoUnit 的具体测试步骤如图 4 所示，它通过缺陷预测来指导测试工具应该具体为哪些文件生成测试用例，并使用测试用例的执行结果实时更新缺陷预测模型，形成一种正反馈机制，极大地提高了测试效率。此外，AutoUnit 能通过召回率预测自行判断是否停止测试，当缺陷文件总数未知时，可通过给定召回率使测试及时停止，最大程度地兼顾效率与准确度。

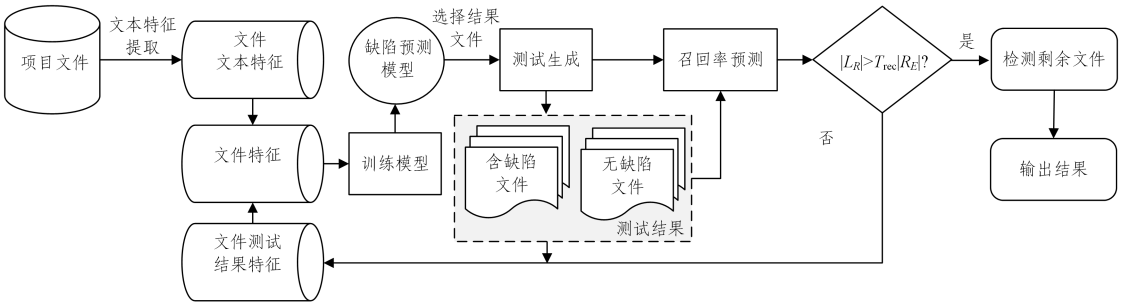

图 4 AutoUnit 测试项目文件的具体过程

Fig. 4 Detailed process of testing project files using AutoUnit

5 实验与分析

5.1 数据集

Defects4j[36] 是用于对 Java 程序进行受控测试研究的含有已知缺陷的数据集，它是从现实世界的 Java 软件系统中提取的可复现的缺陷集合，以及支持这些错误的基础环境（包括含缺陷文件的依赖文件及编译所需的其他文件）。在 2.0.0 版本中它包含 835 个由缺陷造成的真实可复现的 bug（包括编译器、解析器、测试基础设施和各种库）以及用于进行软件测试和调试研究实验的环境支持组件。Defects4j 中的每个 bug 都与源代码文件密切相关且相互之间的影响较小，因此本文选择它作为实验数据集。

本文从 Defects4j 中选择了 16 个项目，以及每个项目所对应的可复现的缺陷文件，每一个缺陷文件对应一个或多个 bug。最终，在选择的 3060 个项目文件中，有 410 个文件含有缺陷。表 1 列出了测试时所使用的软件项目的基本信息。

表 1 从 Defects4j 数据集中整理的部分项目

Table 1 Projects selected from Defects4j dataset

项目名称	项目文件个数	含缺陷文件个数
Chart	553	24
Cli	23	12
Closure	543	86
Collections	319	4
Compress	201	28
Csv	11	6
Gson	67	12
JacksonCore	112	19
JacksonDatabind	419	94
JacksonXml	37	4
Jsoup	68	34
JxPath	171	20
Lang	76	16
Math	136	17
Mockito	169	15
Time	156	19

5.2 实验设计

本文实验分 3 部分来验证 AutoUnit 的实际效果。

（1）对 AutoUnit 中基于主动学习的缺陷预测模型的有效性进行验证。在该部分中，本文对缺陷预测模型进行了测试，确认模型的预测结果能有效地选出含缺陷文件。为了排除其他干扰，本文将测试结果用原数据集中的测试结果信息作为替代。同时，以随机选择作为对比，观察达到不同的召回率时所花费的时间成本。

（2）对缺陷预测引导的测试能力进行验证。检验 AutoUnit 在以实际测试结果为反馈的情况下的测试能力。Defects4j 数据集中集成了两个测试工具，即 Randoop[37] 和 Evosuite[38]，因此本文选择这两个工具作为对比。以每个文件的测试时间来控制测试成本，用这两种工具对整个项目文件进行检测，取二者测得的缺陷文件数量的最大值，观察 AutoUnit 检测到相同数量的缺陷文件时所花费的时间以及检查的文件数量。此时，AutoUnit 使用测试工具代替查询数据库为被选中的文件进行标记。

（3）验证 AutoUnit 的自动停止策略是否有效。本文设计了一种召回率预测方法，通过设置期望达到的召回率，让 AutoUnit 自行判断何时停止。本文将目标召回率设置为 70%，80%，90%，95%，100%，观察 AutoUnit 的实际召回率以及检查的文件数量。

在评价指标方面，本文使用了召回率（recall）、测试成本（cost）以及测试时间（time）这 3 种指标。其中，召回率和测试成本的计算方式如下：

$$recall = \frac{\text{测得的含缺陷文件数量}|L_{pos}|}{\text{含缺陷文件总数}|R|} \tag{3}$$

$$cost = \frac{\text{测得的文件数量}|L|}{\text{文件总数}|E|} \tag{4}$$

5.3 基于主动学习的缺陷预测模型的有效性验证

为了验证 AutoUnit 中的缺陷预测模型能够有效地筛选出含有缺陷的文件进行测试，本文设计了针对缺陷预测模型有效性的实验。在该部分中，本文希望仅对缺陷预测模型进行测试，因此本文将测试环节用原数据集中的真实信息进行

替换。此外，由于测试结果数据的缺失，本文使用文本特征作为每个文件的特征向量，即令 $\boldsymbol{F}=\boldsymbol{FI}$。在召回率预测部分中，本文使用 R 代替 R_E，避免因测试结果数据缺失而造成干扰。

AutoUnit 缺陷预测部分的停止条件为 $|L_R|/|R_E| \geqslant T_{rec}$，因此本文测试了当 T_{rec}（目标召回率）设置为 70%，80%，90%，95%，99%，100%时，不同的缺陷预测模型所需要的成本。我们对 4.1 节中提到的各类特征进行比较，包括：

(1) Metrics 表示软件静态特征；

(2) Version 表示历史版本缺陷信息；

(3) Count 表示词频特征；

(4) Tfidf 表示 TFIDF 评分特征。

此外，我们以随机顺序检查源代码文件作为基线（Baseline），在该方案下 AutoUnit 不参与随机抽查的任何部分。

图 5 中，随着设置的目标召回率不断提高，所需要的缺陷预测引导检查的文件数量不断增多，所花费的时间也逐渐增加。其中，使用 TFIDF 评分作为特征时的缺陷预测模型的成本在相同召回率下是最少的，这也意味着其选择标记待检查的文件数量最少。在测试所需时间方面，图中的时间为训练模型的时间与测试时间的总和，可以看到，即使加上训练模型的时间，其中使用 TFIDF 评分作为特征时缺陷预测模型所花费的时间也是最短的。以上实验结果说明了 AutoUnit 使用 TFIDF 评分作为缺陷预测特征引导测试生成的合理性和有效性。

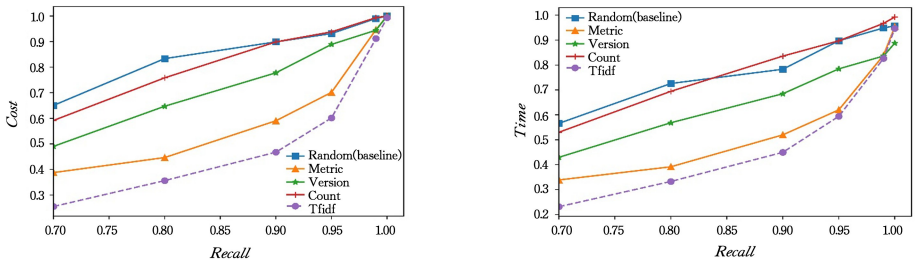

图 5 不同目标召回率下使用各类特征的缺陷预测模型的时间成本对比

Fig. 5 Comparison of test cost and time cost of defect prediction models using various characteristics with different target recall rates

5.4 相同缺陷文件个数下的缺陷预测引导的测试能力验证

本文将 AutoUnit 与现有最常用的两种自动化测试工具 Randoop 和 Evosuite 进行比较。以 Defects4j 中这两种工具的默认检测时间为基准，根据相关研究工作[39]，将每个文件的平均检测时间设置为 60s，90s，120s，模拟在实际测试中不同测试资源下的测试情况，得到 Randoop 和 Evosuite 检测完所有文件后测得的最大缺陷文件数量（即 $\max\{R_{\text{Randoop}}, R_{\text{Evosuite}}\}$），比较 AutoUnit 测得相同数量的缺陷文件时所耗费的时间。

表 2 列出了在单个文件的测试时间相同的情况下，测出相同数量的缺陷文件时，各个测试工具所花费的时间（单位为 min）以及相应情况下的召回率，这里 AutoUnit 所用时间为缺陷预测时间与测试时间之和。从表 2 可以看到，在测得的缺陷文件数量相同的情况下，即使加上缺陷预测时间，AutoUnit 所花费的时间也比 Randoop 和 Evosuite 更短，总时间成本降低了 23%。从召回率方面来看，AutoUnit 的召回率与 Evosuite 近似，这表明 AutoUnit 在测试精度方面并没有因为检测文件数量的减少而下降，这也证明了 AutoUnit 使用缺陷预测来引导测试任务的有效性。该实验的结果说明了在测试效率方面 AutoUnit 优于现有的主流自动化测试工具。

表 2 3 种工具在不同的单个文件测试时间下所能达到的相同（或最大）的召回率以及所花费的时间

Table 2 Same(or maximum) recall rate and time spent by the three tools under different single file test times

Test time of single file/s	Cost-time /min			Recall/%		
	Randoop	Evosuite	AutoUnit	Randoop	Evosuite	AutoUnit
60	467.29	463.52	331.09	25.7	34.6	34.6
90	616.43	622.17	491.36	26.1	35.2	35.2
120	767.25	771.89	611.57	28.8	38.4	38.4

5.5 AutoUnit 自动测试停止策略验证

AutoUnit 能根据已测得的文件（无论该文件是否含有缺陷）预测剩余待测文件中缺陷文件的数量，并自动判断是否满足停止测试的条件。为了减小测试工具误报的影响，本文使用 5.3 节中单个文件测试时间为 120 s 时 Evosuite 的测试结果作为对比。在该实验中，我们使用目标召回率作为停止条件，检验当目标召回率设置为 70%，80%，90%，95%，99%，100%时 AutoUnit 能否正确停止。

从表 3 可以看到，在使用召回率预测来决定何时停止预测引导的情况下，当目标召回率设置为 95%时，AutoUnit 达到的实际召回率比直接使用 Evosuite 测试略高，且检查的文件仅为其文件数量的 30.2%。

表 3 在不同的目标召回率下 AutoUnit 的实际召回率和测试成本

Table 3 Actual recall rate and cost of AutoUnit with different target recall

Target Recall/%	Actual Recall(L_R/R)		Cost(L/E)	
	Evosuite	AutoUnit	Evosuite	AutoUnit
70	26.9%(110)	27.8%(113)	0.676(2068)	0.211(645)
80	30.7%(125)	31.1%(127)	0.758(2319)	0.242(740)
90	34.6%(141)	35.3%(144)	0.917(2806)	0.272(832)
95	36.5%(149)	36.8%(150)	0.981(3001)	0.297(908)
99	38.1%(156)	38.1%(156)	0.993(3038)	0.568(1738)
100	38.4%(157)	38.4%(157)	1.000(3060)	0.786(2405)

表 3 中的数据说明，让 AutoUnit 自行判断何时停止时，其测试结果的召回率与期望值相差较小，能够初步满足测试需要。在实际测试中，含缺陷文件总数未知的情况下，此时

AutoUnit 的自动停止策略能够在检测到绝大部分缺陷文件时及时停止,保证测试效率。

结束语 本文针对目前主流的自动化测试工具注重代码覆盖率而忽略文件出现缺陷的概率不同的缺点,提出了一种基于主动学习和预测引导的自动化测试工具 AutoUnit。它与普通测试工具的区别主要在以下两个方面:1)能根据缺陷预测模型从待测文件池中选出更有可能存在缺陷的文件进行测试,减少了测试文件的数量;2)使用了一种主动学习的策略,每轮得到的测试结果都能用于训练下一轮缺陷预测模型,这种实时的反馈让 AutoUnit 在选择文件时能够更准确地选择含有缺陷的文件,提高了测试效率,并且 AutoUnit 能根据已有的测试结果预测剩余的缺陷文件数量,自动判断何时停止测试。

本文在 Defects4j 数据集上的实验表明,在直接使用数据集给定的信息进行模拟时,只需要检测 54.2%的文件就能测出至少 95%的含缺陷文件。在与主流测试工具 Randoop 和 Evosuite 进行比较时,在测得的含缺陷文件数量相同的情况下,AutoUnit 检测的文件数量更少、测试时间更短。此外,AutoUnit 的自动停止机制也能有效平衡召回率和测试成本,使用自动停止策略的结果与主流测试工具的测试结果相差很小。综上所述,AutoUnit 能有效地降低测试的成本,提高软件的缺陷测试效率。

但是,AutoUnit 仍存在一些局限,其中一个问题是预测的精度仍然较低,即使是在理想状态下,AutoUnit 仍需要检查一半左右的文件才能检测出 95%的缺陷文件,且当目标召回率设置为 100%时可以看到 AutoUnit 的测试成本大幅增加,与普通测试的成本近似。因此,提高缺陷预测模型的预测能力能有效地提高 AutoUnit 的测试效率。此外,现在 AutoUnit 是在文件级粒度进行预测,若能实现对特定函数或者特定代码行的预测,则可以进一步降低测试成本。

参 考 文 献

[1] SHIN Y, WILLIAMS L. Can traditional fault prediction models be used for vulnerability prediction?[J]. Empirical Software Engineering, 2013, 18(1): 25-59.

[2] MAGGIO M, HOFFMANN H, SANTAMBROGIO M D, et al. Controlling software applications via resource allocation within the heartbeats framework[C]// 49th IEEE Conference on Decision and Control (CDC). IEEE, 2010: 3736-3741.

[3] WANG S, LIU T, NAM J, et al. Deep semantic feature learning for software defect prediction[J]. IEEE Transactions on Software Engineering, 2018, 46(12): 1267-1293.

[4] IQBAL A, AFTAB S, ALI U, et al. Performance analysis of machine learning techniques on software defect prediction using NASA datasets[J]. International Journal of Advanced Computer Science and Applications, 2019, 10(5): 300-308.

[5] THOTA M K, SHAJIN F H, RAJESH P. Survey on software defect prediction techniques[J]. International Journal of Applied Science and Engineering, 2020, 17(4): 331-344.

[6] YU G, CHEN X, DOMENICONI C, et al. Cmal: Cost-effective multi-label active learning by querying subexamples[J]. IEEE Transactions on Knowledge and Data Engineering, 2020, 34(5): 2091-2105.

[7] PACHECO C, LAHIRI S K, ERNST M D, et al. Feedback-directed random test generation[C]// 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 75-84.

[8] MY H L T, THANH B N, THANH T K. Survey on mutation-based test data generation[J]. International Journal of Electrical and Computer Engineering, 2015, 5(5): 1164-1173.

[9] MCMINN P. Search-based software test data generation: a survey[J]. Software Testing, Verification and Reliability, 2004, 14(2): 105-156.

[10] BALDONI R, COPPA E, D'ELIA D C, et al. A survey of symbolic execution techniques [J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39.

[11] GODEFROID P, LEVIN M Y, MOLNAR D A. Automated whitebox fuzz testing[C]// NDSS. 2008, 8: 151-166.

[12] HALLER I, SLOWINSKA A, NEUGSCHWANDTNER M, et al. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations[C]// 22nd {USENIX} Security Symposium ({USENIX} Security 13). 2013: 49-64.

[13] GODEFROID P, KLARLUND N, SEN K. DART: Directed automated random testing[C]// Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. 2005: 213-223.

[14] PANDITA R, XIE T, TILLMANN N, et al. Guided test generation for coverage criteria[C]// 2010 IEEE International Conference on Software Maintenance. IEEE, 2010: 1-10.

[15] SHIN Y, MENEELY A, WILLIAMS L, et al. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities[J]. IEEE Transactions on Software Engineering, 2010, 37(6): 772-787.

[16] LI Y, JI S, LV C, et al. V-fuzz: Vulnerability-oriented evolutionary fuzzing[J]. arXiv: 1901.01142, 2019.

[17] PERERA A, ALETI A, BÖHME M, et al. Defect prediction guided search-based software testing[C]// 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020: 448-460.

[18] SETTLES B. Active learning literature survey[D]. Madison: University of Wisconsin-Madison, 2019.

[19] VIJAYANARASIMHAN S, GRAUMAN K. Large-scale live active learning: Training object detectors with crawled data and crowds[J]. International Journal of Computer Vision, 2014, 108(1): 97-114.

[20] CASSEL S, HOWAR F, JONSSON B, et al. Active learning for extended finite state machines[J]. Formal Aspects of Computing, 2016, 28(2): 233-263.

[21] CHU W, ZINKEVICH M, LI L, et al. Unbiased online active learning in data streams[C]// Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2011: 195-203.

[22] MOHAMAD S, SAYED-MOUCHAWEH M, BOUCHACHIA A. Active learning for classifying data streams with unknown number of classes[J]. Neural Networks, 2018, 98: 1-15.

[23] LIU W, ZHANG H, DING Z, et al. A comprehensive active learning method for multiclass imbalanced data streams with concept drift[J/OL]. Knowledge-Based Systems, 2021, 215. https://www.sciencedirect.com/science/article/pii/S0950705121000411.

[24] SINHA S, EBRAHIMI S, DARRELL T. Variational adversarial active learning[C]// Proceedings of the IEEE/CVF International Conference on Computer Vision. 2019: 5972-5981.

[25] WU D. Pool-based sequential active learning for regression[J]. IEEE Transactions on Neural Networks and Learning Systems, 2018, 30(5): 1348-1359.

[26] BELUCH W H, GENEWEIN T, NüRNBERGER A, et al. The power of ensembles for active learning in image classification [C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 9368-9377.

[27] WAHONO R S. A systematic literature review of software defect prediction[J]. Journal of Software Engineering, 2015, 1(1): 1-16.

[28] D'AMBROS M, LANZA M, ROBBES R. On the relationship between change coupling and software defects[C]// 2009 16th Working Conference on Reverse Engineering. IEEE, 2009: 135-144.

[29] MIZUNO O, IKAMI S, NAKAICHI S, et al. Spam filter based approach for finding fault-prone software modules[C]// Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007). IEEE, 2007.

[30] GROSSMAN M R, CORMACK G V, ROEGIEST A. TREC 2016 Total Recall Track Overview[C]// TREC. 2016.

[31] YU Z, THEISEN C, WILLIAMS L, et al. Improving vulnerability inspection efficiency using active learning[J]. IEEE Transactions on Software Engineering, 2019, 47(11), 2401-2420.

[32] RAMOS J. Using tf-idf to determine word relevance in document queries[C]// Proceedings of the First Instructional Conference on Machine Learning. 2003: 29-48.

[33] LI H, CHUNG F, WANG S. A SVM based classification method for homogeneous data[J]. Applied Soft Computing, 2015, 36: 228-235.

[34] WEI H, HU C, CHEN S, et al. Establishing a software defect prediction model via effective dimension reduction[J]. Information Sciences, 2019, 477: 399-409.

[35] WANG K, LIU L, YUAN C, et al. Software defect prediction model based on LASSO-SVM[J]. Neural Computing and Applications, 2021, 33(14): 8249-8259.

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

[37] PACHECO C, ERNST M D. Randoop: feedback-directed random testing for Java[C]// Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. 2007: 815-816.

[38] FRASER G, ARCURI A. Evosuite: automatic test suite generation for object-oriented software[C]// Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. 2011: 416-419.

[39] VIRGÍNIO T, MARTINS L A, SOARES L R, et al. An empirical study of automatically-generated tests from the perspective of test smells[C]// Proceedings of the 34th Brazilian Symposium on Software Engineering. 2020: 92-96.

ZHANG Da-lin, born in 1983, Ph.D, associate professor, Ph.D supervisor, is a member of China Computer Federation. His main research interests include software engineering theory and technology.

ZHANG Zhe-wei, born in 1999, posgraduate. His main research interests include machine learning and software automated testing.

(责任编辑:喻藜)