软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(8):2243−2257 [doi: 10.13328/j.cnki.jos.005531]

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

一种基于关联分析与 *N*-Gram 的错误参数检测方法*

李 超, 刘 辉

(北京理工大学 计算机学院,北京 100081)

通讯作者: 刘辉, E-mail: liuhui08@bit.edu.cn

摘 要: 为了检测软件系统中存在错误参数的函数调用,提出了一种基于关联分析和 *N*-Gram 语言模型的静态检测方法(ANiaD).基于海量开源代码,构建了关联分析模型以挖掘参数间存在的强关联规则.针对参数间存在强关联规则的函数调用构建*N*-Gram语言模型.基于训练过的*N*-Gram模型,计算给定函数调用语句正确的概率.低概率的函数调用被报告为异常函数调用.基于 10 个开源 Java 项目对该方法进行实验验证.实验结果表明,该方法检测的查准率约 43.40%,显著高于现有的基于相似度的检测方法(查准率 25%).

关键词: 参数;异常检测;缺陷;语言模型;关联分析

中图法分类号: TP311

中文引用格式: 李超,刘辉.一种基于关联分析与 *N*-Gram 的错误参数检测方法.软件学报,2018,29(8):2243−2257. http://www.jos.org.cn/1000-9825/5531.htm

英文引用格式: Li C, Liu H. Association analysis and *N*-Gram based detection of incorrect arguments. Ruan Jian Xue Bao/ Journal of Software, 2018,29(8):2243−2257 (in Chinese). http://www.jos.org.cn/1000-9825/5531.htm

Association Analysis and *N*-Gram Based Detection of Incorrect Arguments

LI Chao, LIU Hui

(School of Computer Science and Technology, Beijing Institute of Technolog, Beijing 100081, China)

Abstract: To detect the method calls with incorrect arguments in software systems, an association analysis and *N*-Gram based static anomaly detection approach (ANiaD) is proposed. Based on the massive open source code, an association analysis model is constructed to mine the strong association rules between arguments. An *N*-Gram model is constructed for method calls with strong association rules between arguments. Using the trained *N*-Gram model, the probability of a given method call statement is calculated. Low probability method calls are reported as potential bugs. The proposed approach is evaluated based on 10 open-source Java projects. The results show that the accuracy of the proposed approach is about 43.40%, significantly greater than that of similarity-based approach (25% accuracy).

Key words: argument; anomaly detection; bug; language model; association analysis

自从引入模块化程序设计思想以来,函数调用就成为源码中最常见的代码元素之一.而参数作为调用函数和被调用函数之间传递数据的一种重要手段[1],会对程序的正确执行产生重要影响.错误的参数往往会导致严重的软件缺陷,导致软件系统不能正常工作.错误参数检测是发现此类错误的一种重要手段,具有较高的实用价值[2−4].

现有的错误参数检测方法可分为基于语法[2,3]和基于语义的错误参数检测[4].在静态类型编程语言中,函数

* 基金项目: 国家重点研发计划(2016YFB1000801); 国家自然科学基金(61472034, 61772071, 61690205)

Foundation item: National Key Research and Development Program of China (2016YFB1000801); National Natural Science Foundation of China (61472034, 61772071, 61690205)

本文由数据驱动的软件智能化开发方法与技术专题特约编辑谢冰教授、魏峻研究员、彭鑫教授、孙海龙副教授推荐.

收稿时间: 2017-07-18; 修改时间: 2017-09-28; 采用时间: 2017-12-22; jos 在线出版时间: 2018-03-13

CNKI 网络优先出版: 2018-03-13 17:30:48, http://kns.cnki.net/kcms/detail/11.2560.TP.20180313.1730.012.html

中的每个形参都拥有一个形参类型,以确保只有满足类型匹配的元素才能作为实参[2,3].编译器在编译时将执行此类语法检测.但当拥有多个类型匹配的候选实参可供选择时,这种方法就失效了.例如,当函数 *getAnimal*(*cat*)与 *getAnimal*(*dog*)都能够满足语法要求而编译通过时,这种检测方法不能进一步判定哪个是正确的函数调用.此外,由于面向对象程序设计中的多态性以及 Java 语言相关语法规则,在许多场景下无法在编译期间确定某些函数调用中接受的实参对象的实际类型[2,3].这种情况更加削弱了基于语法检测方法的作用.

现有的基于相似度的错误参数检测方法[4]利用了程序中标识符名称的语义信息.该方法的核心是比较实参与形参的文本相似度和上下文环境中其他候选实参与形参的文本相似度.这种方法的基本前提是编程人员具有良好的标识符命名习惯.上述方法仅仅利用了函数调用上下文中的有限信息(参数类型与标识符名称),没有从项目整体出发进行考虑,更没有从海量开源代码中挖掘更多的有用信息.

目前,许多研究都证实了利用自然语言模型对程序源代码进行建模的有效性.Hindle 等人[5]证实了代码程序同样具有自然语言的重复性和可预测性的特点.这个观点在函数调用方面体现为同一函数的多次函数调用的重复性和关联性.通过阅读大量的 Java 开源代码发现:尽管实参序列的选取具有较强的灵活性,但是同一函数定义对应的函数调用集合中,实参序列间存在潜在的关联性.基于此,如果能够充分挖掘实参序列间的关联信息,提取潜在的编程规则,则有利于实现函数调用相关的异常检测.针对函数调用中的错误参数设计有效的异常检测方法,有利于提高软件的可靠性和可维护性.

鉴于此,本文提出了一种基于关联分析和 *N*-Gram 语言模型的自动化静态异常检测方法(ANiaD),它是一种数据关联性驱动的异常检测方法,用于检测函数调用中存在的参数错误.基于项目源代码,构建了关联分析模型以挖掘函数调用参数间存在的强关联规则.针对参数间存在强关联规则的函数调用构建 *N*-Gram 语言模型.基于训练过的 *N*-Gram 模型,计算给定函数调用语句是否正确的概率.低概率的函数调用被报告为潜在的异常.最后,结合人工核实,把潜在异常判定为错误参数、不常用的使用方式或重构机会.这种数据驱动的软件分析方法能够自适应编程人员的编程习惯与被检测软件自身独有的特点,通过关联分析感知数据关联性、挖掘数据间的依赖关系.

本文的主要贡献是:

- 本文提出了一种基于关联分析和 *N*-Gram 语言模型的静态异常检测方法(ANiaD).基于项目源代码,构建了关联分析模型以挖掘函数调用参数间存在的强关联规则.针对参数间存在强关联规则的函数调用构建 *N*-Gram 语言模型.基于训练过的 *N*-Gram 模型,计算给定函数调用语句是否正确的概率.低概率的函数调用被报告为存在潜在异常的函数调用.
- 基于 10 个 Java 开源项目,对 ANiaD 方法进行实验验证.实验结果表明,该方法检测的查准率约 43.40%,显著高于现有的基于相似度的检测方法(查准率 25%);该方法共检测出 18 个错误参数,远远超出现有的基于相似度的检测方法(检测出 2 个错误参数).

本文第 1 节介绍相关的研究背景.第 2 节对提出的 ANiaD 方法进行详细阐述.第 3 节进行实验验证与结果分析,进而对 ANiaD 方法进行评价.第 4 节介绍相关的研究工作.第 5 节得出结论.

1 关联分析与 *N*-Gram 语言模型

1.1 关联分析模型

在数据挖掘领域,关联分析技术经常被用于发现大数据集中令人感兴趣的关联关系,即关联规则.Apriori 是关联性数据挖掘领域的核心算法,它基于一种将大数据集中频繁项集逐步聚合的策略[6].可以利用生成的频繁项集产生强关联规则.

关联规则的强度通常使用支持度和可信度这两个指标进行衡量,分别表示为 s 和 c.可信度 $c(X\rightarrow Y)$表示 X 出现时,Y 同时出现的概率.支持度 $s(X\rightarrow Y)$表示 X 与 Y 同时出现的项集在总数据集中占有的比例:

$$s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N} \tag{1}$$

$$c(X \to Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \tag{2}$$

其中,X,Y 表示项集,N 表示总数据集中的数据量,$\sigma(X \cup Y)$表示 X 和 Y 共同出现的次数.

1.2 *N*-Gram语言模型

统计语言模型是用来计算一个句子的概率的概率模型.假设 W 是由 T 个词 $w_1,w_2,w_3,\ldots,w_T$ 按照一定的顺序构成的一个句子,则 $P(W)$为句子的概率:

$$P(W) = P(w_1^T) = P(w_1, w_2, w_3, ..., w_T) \tag{3}$$

利用贝叶斯公式可把公式(3)转化为

$$P(W) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1^3)...P(w_T \mid w_1^T) \tag{4}$$

Suen 等人[7]提出了 *N*-Gram 统计语言模型以处理自然语言理解和文本处理问题,该模型已经在许多的自然语言处理应用中取得了成功,包括基于统计的机器翻译[8]和文本分类[9].*N*-Gram 的核心思想是,假设在标记流中(例如语句中的有序单词序列)只有有限个连续标记会对之后的标记产生影响.图 1 展示了一个滑动窗口长度为 3 的 *N*-Gram 语言模型,此窗口在语句中按照单词顺序向后滑动,每个窗口包含 3 个单词.

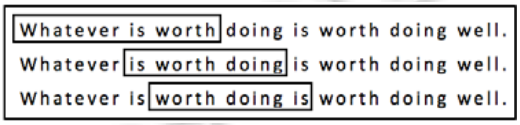

Fig.1 Sliding windows (window size: 3)

图 1 滑动窗口(窗口大小为 3)

简单来说,*N*-Gram 语言模型的思想是,假设一个词出现的概率只与该词前面固定数量的词相关.若一个词出现的概率只与它前面的 N–1 个词相关:

$$P(W_K \mid W_1^{K-1}) \approx P(W_K \mid W_{K-N+1}^{K-1}) = P(W_K \mid W_{K-N+1}, W_{K-N}, W_{K-N-1}, ..., W_{K-1}) \tag{5}$$

N-Gram 语言模型能够构建出覆盖所有标记流的概率分布,*N*-Gram 概率的计算过程利用了一个固定长度的滑动窗口遍历整个标记流,期间记录每个窗口中元素共同出现的条件概率.假设一个标记流为 $s=w_1w_2w_3\ldots w_m$,它的概率被估计为

$$P(s) = \prod_{i=1}^{m} P(w \mid h_{i-1}) \tag{6}$$

其中,$h_i=w_{i-n}\ldots w_i$ 表示在 *N*-Gram 语言模型中下一个标记 w_i 出现的概率仅仅依赖于之前的 n–1 个标记.

2 错误参数检测方法

2.1 方法概述

本文提出的基于关联分析与 *N*-Gram 的异常检测方法(ANiaD)能够通过关联分析感知数据关联性、挖掘数据间的依赖关系,获取项目中包含异常参数的函数调用情况.排除这些异常参数,有助于提升软件的可靠性和可维护性.

N-Gram 方法基于海量开源代码构建了关联分析模型以挖掘函数调用参数间存在的强关联规则,针对参数间存在强关联规则的函数调用构建 *N*-Gram 语言模型.基于训练过的 *N*-Gram 模型,计算给定函数调用语句是否正确的概率.低概率的函数调用被报告为潜在的缺陷.

本方法分为 4 个步骤,如图 2 所示.

- 第 1 步,代码解析与样本分类.通过构建抽象语法树对程序源代码进行解析,获取代码中包含的函数调用及其相关信息.每一个样本由函数调用的函数标识符名称、所在类的标识符名称、所在包的名称、函数形参类型序列和函数实参标识符名称序列组成.最后把同一函数定义对应的所有函数调用样本

数据划分为一组.

- 第 2 步是构建关联分析模型.关联分析模型能够评价函数调用中参数间的关联程度,为接下来是否执行异常检测提供判断依据,本方法只对参数间存在强关联规则的函数调用执行异常检测.
- 第 3 步是构建 *N*-Gram 语言模型,以训练得到一个概率分布,可计算出函数调用中存在错误参数的概率.
- 第 4 步是执行异常检测,通过设置一个合理的阈值对函数调用进行分类,低概率的函数调用被报告为潜在异常,它可能是真实的错误参数、不常用的代码片段或潜在的重构机会.接下来,程序员需要根据报告的信息,追溯到代码版本库中进行人工核实.

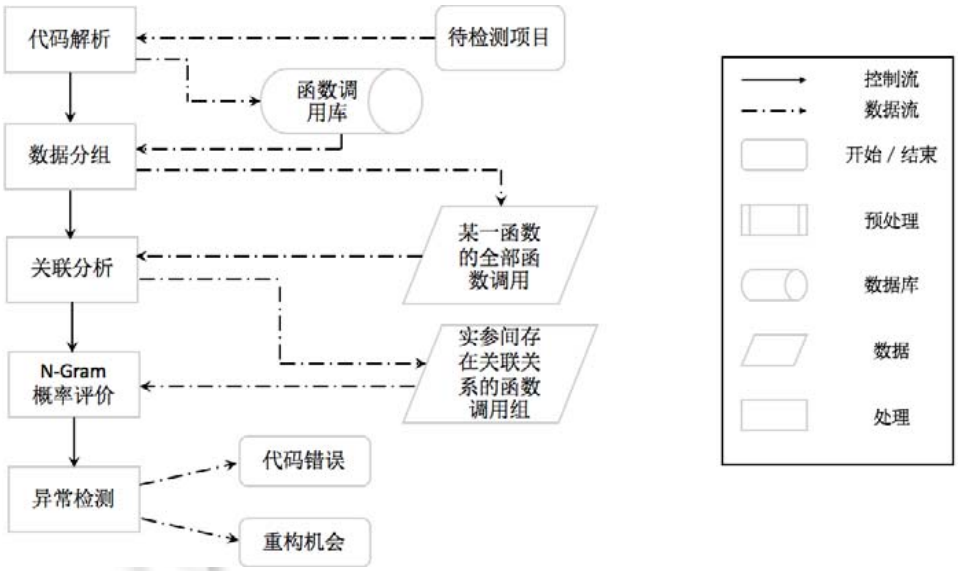

Fig.2 Flowchart of the approach

图 2 方法流程图

2.2 代码解析与样本分类

为构建关联分析模型和 *N*-Gram 语言模型,我们需要收集能够体现程序员如何选择函数实参的样本数据.直接且通用的做法是构建抽象语法树(ASTs),它是程序源代码中抽象语法结构的树形表示.

通过访问 ASTs 中函数调用子树,可以获取函数调用的基本信息,包括函数标识符名称、实参标识符名称.为区分具有相同函数标识符名称的不同函数(例如重载函数),还需要访问对应的函数定义子树,可以从中获取函数所在包名、所在类名和形参列表等信息,公式(7)表示解析并存储到数据库中数据的格式.在数据分组时,把函数标识符名称、类名、包名和形参类型相同的函数调用划分为一组,所以重载函数被视为不同的函数.

$$semple=\langle funName,pacName,claName,\arg Name_1,\ldots,\arg Name_n,parType_1,\ldots,parType_n\rangle \quad (7)$$

在 Java 语言中,函数实参表达式的类型有很多.在处理不同类型时采取了不同的策略.

- 本地变量,实参名称为变量名称;
- 字段访问,例如 round.size,实参名称为字段名称,忽视字段的底层信息;
- 函数调用,例如 *sum*(*a*,*b*),实参名称为函数标识符名称,忽视实参列表表达式和函数的底层信息;
- 数组访问,例如 apple[2],实参名称为数组表达式的名称,忽视索引表达式.

值得注意的是,常量(例如数字、字符串)和复杂表达式(例如 5+*sum*)不能通过解析获取名称.对于这种情况,我们把常量或表达式转换为字符串以作为名称.本文方法利用了 EclipseJava 开发工具(JDT)解析程序源文件以收集函数调用样本数据.

2.3 构建关联分析模型

构建基于 Apriori 算法的关联分析模型,如图 3 所示.Apriori 算法包含 4 个子模块:首先,初始化频繁项集导入模型;其次,顺序执行生成候选集、精简候选集、计算候选集支持度.计算得到的支持度信息会反馈到候选集生成器.此过程迭代执行,直至确定最终的候选集.模型中每个项集包含一个函数调用中的实参标识符序列.在候选集生成的过程中,为了满足高支持度的要求,每一代候选集都用于构建下一代候选集,每一个新的候选集都构建于之前的候选集[10].

在精简候选集阶段,需要确保每一个新生成的候选集不仅仅由前一代的两个候选集组合而成,而且它的所有子集(通过移除候选集中一个元素来构建)都出现在之前生成的候选集中.这是一个逐步的构建候选集的过程.在计算支持度阶段,需要计算每一个潜在候选集的支持度,即项集中包含的所有项在数据集中同时出现的次数.如果候选项集中的所有项都出现在某一函数调用的实参列表中,支持度计数就增加,伪代码如图 3 所示.

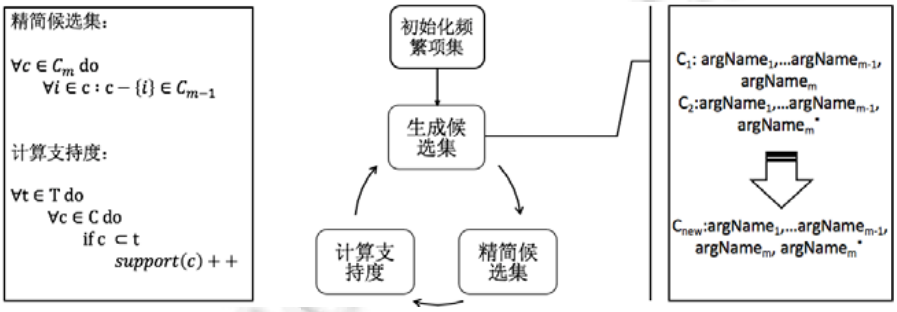

Fig.3 Diagram of Apriori algorithm and partial pseudo code

图 3 Apriori 算法示意图及部分伪代码

构建关联分析模型的目的是把参数间不存在强关联规则的函数调用过滤掉,不进行后续的异常检测.具体做法是获取函数参数个数大于等于 2 的函数调用,对其中的每个函数定义对应的函数调用组独立的构建关联分析模型.正如本文多次强调“本方法只对参数间存在强关联规则的函数调用执行异常检测”,意指此方法仅适用于多参数的情况(参数个数大于等于 2),不适用于参数个数为 1 的情况.在实验中,在构建关联分析模型之前已经过滤掉参数个数为 1 的函数调用(参数个数为 1 时,挖掘参数间的强关联规则没有意义).如果最终得到的频繁项集的长度小于函数调用中参数列表的长度,这组函数调用会被过滤掉,不执行后续的异常检测.如果最终得到的频繁项集的长度等于函数调用中参数列表的长度,会建立 *N*-Gram 模型对每一个函数调用进行评价.

2.4 构建*N*-Gram语言模型

训练 *N*-Gram 语言模型得到概率分布,基于训练后的模型计算函数调用中包含错误参数的概率.模型输入是函数调用中的标识符序列,包括函数标识符名称和实参标识符名称列表,如式(8)所示.

$$data_i=\langle funName,\arg Name_1,\ldots,\arg Name_n\rangle \tag{8}$$

为了训练 *N*-Gram 语言模型,需要把同一函数定义(具有相同函数名、包名、类名和形参类型)对应的函数调用划分为一组,以这组数据作为 *N*-Gram 语言模型的训练数据.在构建 *N*-Gram 语言模型时,窗口大小是一个重要的参数,它依赖于应用本身[11].综合考虑源代码中函数调用的特征后,方法把滑动窗口的大小设置为 2,详细分析见第 3.4 节.利用 *N*-Gram 语言模型对参数间存在强关联规则的函数调用计算存在错误参数的概率,低概率的函数调用被报告为潜在异常,它可能是真正的错误参数、不常用的代码片段或潜在的重构机会.

使用公式(4)这种方法计算概率时会存在两个问题:参数空间过大和数据稀疏严重.参数空间过大会导致计算概率时的时间复杂度和空间复杂度较高,严重影响方法性能.数据严重稀疏是指当且仅当被计算概率的序列已经在训练样本中存在时,才能计算出此序列的有效概率;否则计算出的概率为 0,没有意义.由于 *N*-Gram 模型假设当前标识符出现的概率只与它前面出现的一个或几个标识符相关,当使用 *N*-Gram 语言模型计算概率时,

模型的参数空间的维数缩减为 N,进而有效缩减了计算概率的时间复杂度和空间复杂度.由于参数空间的维数越大数据稀疏越严重,N-Geam 模型通过降低参数空间的维数能够有效减轻数据稀疏的影响.综上所述,本文采用了 N-Gram 语言模型来计算概率.

2.5 异常参数检测

上述模型计算出给定函数调用语句是否正确的概率,接着,通过设置合理的阈值对函数调用进行分类,低于阈值的情况视为潜在异常.针对潜在异常,需结合人工核实在源代码中进行确认,如果异常所在代码行存在可修复的错误,核实为错误参数.

在人工核实潜在异常时,采用了在项目版本库中进行确认的方式.把异常所在代码与新版本项目代码进行对照,如果函数调用在项目新版本中进行了修改,此潜在异常被核实为真实错误.为了确保上述异常为真实代码错误,通过人工的方式检查代码修改处的相关版本提交信息.被视为代码错误的相关代码修改的提交信息一般是非常明确的,例如“code cleanup: wrong argument was used”,“fixed bugs:correctly argument”.基于此,在实验阶段选用的所有开源项目均拥有可公开获取到的版本控制系统(GIT,SVN 或 CVS).

如果潜在异常不是错误参数,而是影响代码可读性与稳定性的重构机会,包括重命名重构、封装字段重构等.不恰当的标识符名称会影响代码的可读性与代码质量.封装字段禁止对字段直接访问,能够实现对字段更好的保护.相关研究表明,有意义的标识符名称能够更利于代码的理解[4].有理由相信,遵循本方法提供的重命名重构建议能够极大地提高被检测项目的代码可读性.同样,封装字段重构能够实现对数据更好的保护,进而提高代码的正确性与稳定性.

除了错误参数和重构机会之外,一些潜在异常是不常用或特殊的使用方式,这些情况造成了误报.为了减少误报,我们降低了阈值以减少报告潜在异常的数量,使得报告的潜在异常为真实异常的可能性更大.

2.6 算法示例

本文提出的异常检测方法的算法示例,如图 4 所示.图中 *Port*.1 展示了从某一项目中解析出的函数 *getConnection*(*String driver*,*String user*,*String password*)的函数调用样本,假设 N_1,…,N_8 表示从此项目中解析出的对该函数的所有调用情况(call site),其中,函数调用 N_3 的参数有误:编程人员把密码“*ps*”参数误写为端口号“*port*”;尽管这不会导致编译时错误,但在运行时会出现数据库连接失败.函数调用 N_5 中参数顺序出现错误,密码“*ps*”和用户名“*user*”两个参数的顺序出现了异常.这同样会导致运行时数据库连接失败.下面以这个例子阐述本文提出的异常检测方法如何检测出参数异常.

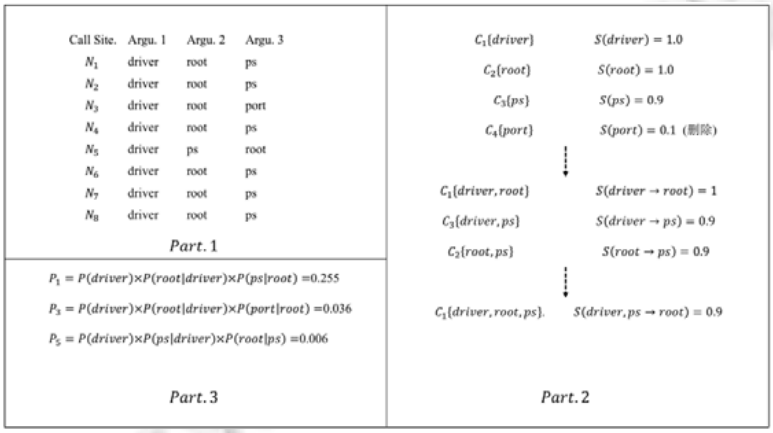

Fig.4 Examples of anomaly detection

图 4 异常检测示例

如图 4 所示,*Port*.2 为构建关联分析模型的示意图.

- 首先,使用函数调用实参序列初始化频繁项集,可得到 C={(*driver*),(*root*),(*ps*)}.由于项 *port* 的支持度小于阈值(S(*port*)=0.1),所以 *port* 不属于频繁项集.接着执行生成候选集与精简候选集,可得到:

$$C=\{(driver,root),(driver,root),(root,ps)\}.$$

- 其次计算候选集支持度,例如 $S(driver\rightarrow root)=1$,支持度大于某一阈值的候选项组成频繁项集.此过程迭代执行直至生成最终的频繁项集,即为 C={(*driver*),(*root*),(*ps*)}.

最终得到的频繁项的长度(为 3)等于函数调用中参数列表的长度,所以这组函数调用被视为函数参数间存在强关联规则的函数调用而进入下一阶段的异常检测.

Port.3 为构建 *N*-Gram 语言模型的示意图,以计算出函数调用中存在参数错误的概率.低概率的函数调用被报告为存在潜在异常.从图中可以看出:N_3 与 N_5 为低概率的函数调用,其余函数调用的概率与 N_1 的概率 P_1 相同.至此检测过程结束,N_3 与 N_5 被报告为存在潜在参数异常的函数调用.

2.7 方法实现

本文针对 Java 语言实现了上述异常检测方法,语言相关部分构建于 Eclipse Java Development Tools(JDTs),这个平台提供了解析程序抽象语法树的全面解决方案.本文异常检测方法与 Java 语言特性不相关,因此可以在其他编程语言中直接扩展本方法的实现.

本文方法的实现采用了 Eclipse 插件的形式.当项目开发结束或在开发进程中的某个阶段,可启动全局检测对整个项目进行检测.此时,插件会检测项目中涉及的所有函数调用,报告所有的存在潜在异常的函数调用,同时包括每一个潜在异常的详细信息与修改建议.

Eclipse 插件的可视化界面如图 5 所示,在 CodeSemlls 视图中,依次列出了在异常检测时检测到的存在异常的函数调用记录,包括函数名、实参序列及推荐实参序列.当双击某一条记录时,对话框会呈现此异常函数调用的详细信息及修改建议,包括异常解释信息、函数调用的位置信息及修改建议.

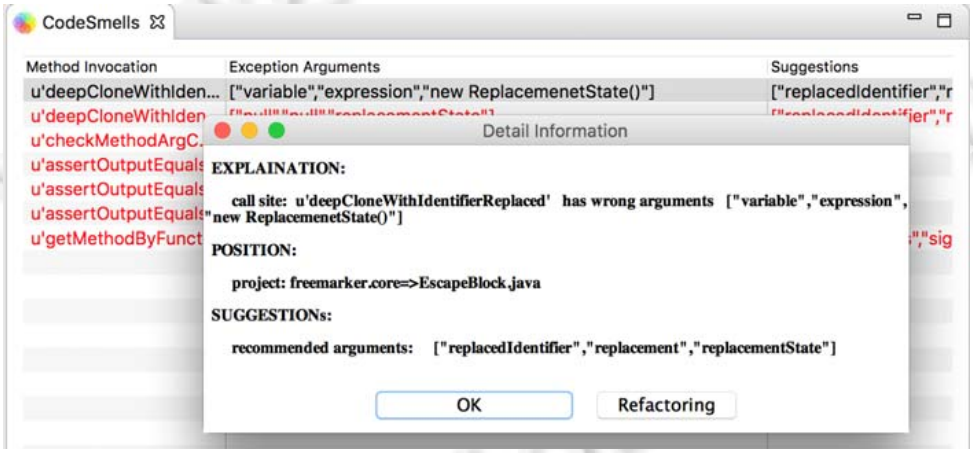

Fig.5 Screenshot of Eclipse plug-in

图 5 Eclipse 插件截图

3 实验验证

本节将利用开源项目以验证 ANiaD 方法在错误参数检测问题中的有效性.

3.1 实验设计和研究问题

为了评价 ANiaD 方法,我们进行了实证研究以回答下面的问题.

- RQ1:*N*-Gram 中的参数 *n* 应该如何设置?

- RQ2:ANiaD 方法是否优于现有的错误参数检测方法?
- RQ3:ANiaD 方法对大中小型项目的检测效果有无差异?
- RQ4:使用 ANiaD 方法,对 Java 开源项目实施异常检测会耗费多长时间?

RQ1 能够为 *N*-Gram 模型窗口大小的选择提供依据.相关研究表明,随着窗口大小的增加,*N*-Gram 语言模型的计算复杂度呈指数增长[7].本应用中,*N*-Gram 模型语料是函数调用中的标识符序列.根据开源项目中函数调用的特点选取合适的滑动窗口大小具有重要意义.RQ3 主要分析此异常检测方法对大中小型项目的检测结果,进而分析该方法是否对某一类型的项目有更好的结果.RQ4 主要专注于方法响应时间,如果能在较短时间内实现异常检测,方法会具有更高的应用价值,我们打算以 Eclipse 插件形式制作基于 ANiaD 方法的测试辅助工具.异常检测过程消耗过多时间会削减开发人员的工作效率.

RQ2 主要关注 ANiaD 方法的性能表现(例如查准率).ANiaD 方法与最近公布的方法作对比,能够判断 ANiaD 是否提升了现有的水平.据我们所知,Liu 等人[4]提出的基于相似度的异常检测方法是检测错误参数相关异常的最新方法,而且基于相似度的异常检测方法与 ANiaD 具有相同的目标与作用范围.因此,我们选择基于相似度的异常检测方法进行对比实验.

3.2 实验准备

在实验准备阶段,我们从 GitHub(http://github.com)上选取流行度较高的 10 个 Java 开源项目作为实验对象,如表 1 所示.这些应用软件共包含 1 317 480 行源代码,其中涵盖小型、中型和大型应用软件.项目代码行数在 31 007 行~565 166 行之间,项目规模跨度较大.基于这些项目,能够全面考察本文方法在不同规模项目中的性能表现.

Table 1 Open-Source applications in empirical studies

表 1 实证研究中使用的开源项目

名称	所有代码行	可执行代码行
Mondrian	283 834	196 750
itext7	83 420	56 063
UniversalMediaServer	69 728	47 942
Vuze	895 668	565 166
Hibernate	144 013	82 364
Plantuml	155 755	88 287
Purdue-fastr	60 796	48 883
Freemarker	57 202	31 007
Hsqldb	274 302	148 055
Omegat	100 410	52 963
合计	2 125 128	1 317 480

在实验项目选择过程中主要考虑到以下几个方面.选取的所有项目都是开源项目,既保证了源代码真实性,又便于外界研究人员重现实验.选取的所有项目都有多个版本,在旧版本中检测出异常能够在新版本中得到核实.选取的所有项目都是流行度较高的项目,保证了较高的代码质量.

针对每个待检测开源项目,按照以下步骤进行实验.

- 开源项目代码解析,解析数据存储到数据库;
- 构建关联分析模型和 *N*-Gram 模型,计算函数调用是否正确的概率;
- 函数调用分类,低概率函数调用报告为潜在异常;
- 人工核实潜在异常.在代码版本库中进行确认,如果异常所在代码段已在项目新版本中被修改,核实为错误参数.

3.3 评价指标

针对 ANiaD 检测出的潜在异常,结合人工核实把潜在异常划分为 3 种类型:错误参数、重构机会和误报.重构机会是源代码中质量较差的实现,可以通过重构这些代码片段使得代码更加规范化.不属于错误参数和重构机会的潜在异常记为误判,即负样本.错误参数和重构机会的总量记为正样本.

为衡量 ANiaD 方法的性能,我们定义两个度量指标:错误参数检测的查准率(P_{bug})和异常参数检测的查准率($P_{anomaly}$).错误查准率用来衡量代码错误的概率,如公式(9)所示.

$$P_{bug}=\frac{Bugs}{Reported} \tag{9}$$

其中,*Bugs* 代表代码错误量,*Refs* 代表重构机会量,*Reported* 代表潜在缺陷量.异常查准率用来衡量检测出的潜在异常有意义的概率,如公式(10)所示.

$$P_{anomaly}=\frac{Bugs+Refs}{Reported} \tag{10}$$

由于我们无法获知一个项目中包含的全部代码错误的数量,所以没办法计算查全率.为此,我们主要通过查找到的错误参数或者异常参数的数量来比较不同方法的查全率.对于给定的软件系统,其错误参数数量(N)是一定的,因此,只需要比较不同方法在相同软件系统上查找到的错误参数的数量(n_1,n_2),即可比较他们的查全率($R_1=n_1/N$,$R_2=n_2/N$).

实验选择对开源项目的旧版本进行异常检测,在开源项目新版本中进行人工核实.针对潜在异常进行人工核实时,如果潜在异常所在源代码在项目新版本中已得到修正,则记为错误参数.对非错误参数的报告,判断是否为重构机会.在判断潜在异常是否为重构机会时,参照了代码坏味准则[12].

3.4 实验结果与分析

下面将阐述上述研究问题的实验结果,具体如下:

(1) RQ1:设置 *N*-Gram 中的参数 *N*.

正如第 2.4 节中所介绍,*N*-Gram 语言模型的语料是函数调用中的标识符序列.根据 Java 开源项目中函数调用的特点选择合适的滑动窗口大小具有重要意义.在 Java 开源项目中函数调用的实参有多少个?为了回答这个问题,统计了 10 个开源项目中出现的所有函数调用的实参个数 η(排除无参数函数调用的情况),统计结果如图 6 和表 2 所示.

由于在构建关联分析模型之前已经过滤掉参数个数为 1 的函数调用,在所有需要进行参数检测的数据中,71.46%的数据的实参个数等于 2,28.54%的数据的参数个数大于等于 3,11.68%的数据的参数个数大于等于 4.函数调用标识符序列作为 *N*-Gram 语言模型的语料,超过 70.00%的标记流长度小于等于 2,这就导致当窗口长度大于 2 时随着窗口长度的增加,不会对 *N*-Gram 模型的评价结果造成显著影响.综合考虑 *N*-Gram 语言模型的计算复杂度,在构建 *N*-Gram 语言模型时,选择了 2-Gram.

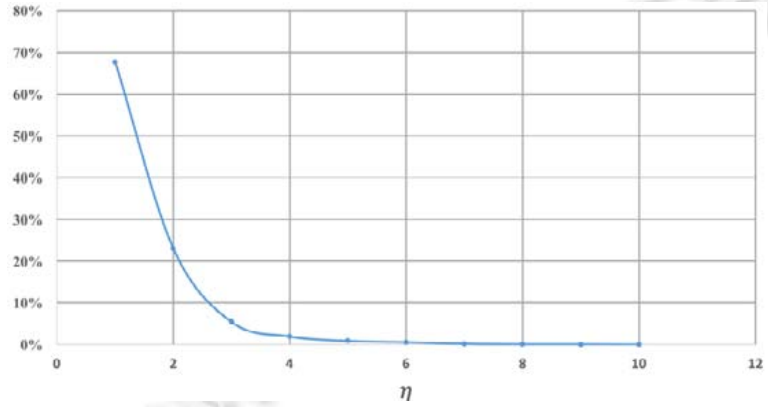

Fig.6 Probability distribution of the number of arguments

图 6 函数调用实参个数概率分布

Table 2 Statistical results of the number of arguments

表 2 函数调用实参个数统计结果

参数个数	函数数量	所占百分比(%)
1	160 275	67.73
2	54 580	23.06
3	12 875	5.44
4	4 638	1.96
5	2 166	0.92
6	1 325	0.56
7	458	0.19
8	203	0.09
9	91	0.04
10	29	0.01

(2) RQ2:ANiaD 与基于相似度的异常检测方法对比

在实证研究阶段,针对每个开源项目,首先分别使用 ANiaD 方法和基于相似度的方法进行异常检测,其次对这两种检测方法报告的潜在异常进行人工核实,最后统计出错误参数的数量和重构机会的数量.如表 3 所示,表中第 2 列~第 4 列分别记录了 ANiaD 方法检测出的潜在异常数量、错误参数数量和重构机会数量.表中第 5 列~第 7 列分别记录了基于相似度方法检测出的潜在异常数量、错误参数数量和重构机会数量.

Table 3 Comparison of the experimental results of bug detection guided by ANiaD and similarity-based approach

表 3 ANiaD 与基于相似度的缺陷检测方法的实验结果对比

项目	ANiaD 方法			基于相似度的方法		
	Reported	IsBugs	IsRefs	Reported	IsBugs	IsRefs
Mondrian	25	3	4	7	0	2
itext7	10	3	1	2	0	0
UniversalMediaServer	4	1	0	0	0	0
Vuze	9	1	3	11	0	3
Hibernate	10	1	8	0	0	0
Plantuml	9	1	2	2	1	0
Purdue-fastr	19	4	2	0	0	0
Freemarker	3	1	2	0	0	0
Hsqldb	16	2	6	4	1	0
Omegat	1	1	0	2	0	0
合计	106	18	28	28	2	5
P_{bug}	–	16.98%	–	–	7.14%	–
$P_{anomaly}$	–	43.40%	–	–	25.00%	–

实验结果表明,相对于基于相似度的错误参数检测方法,本文方法的 P_{bug} 和 $P_{anomaly}$ 都有明显的改善.P_{bug} 由 7.14%提升到了 16.98%,$P_{anomaly}$ 由 25.00%提升到了 43.40%.更深入分析,相对于基于相似度的错误参数检测方法,ANiaD 方法的 P_{bug} 提升了 137.82%=(16.98%−7.14%)/7.14%,$P_{anomaly}$ 提升了 73.60%=(43.40%−25.00%)/25.00%.

由各个项目的统计数据可进一步发现:相对于基于相似度的错误参数检测方法,对每个项目 ANiaD 方法都能够检测到更多的错误参数与重构机会.正如第 4.1 节中所介绍,这些项目涵盖了小型、中型和大型应用软件,从而说明 ANiaD 方法能够在各种规模的软件项目中得到很好的效果.

下面示例说明 ANiaD 检测错误参数的过程,见表 4.在 omegat 应用项目中,函数 *addTranslation*()分别在 7 个 Java 文件中被调用,在表 4 中列出了函数调用所在语句.首先,通过关联分析模型迭代生成候选集,最终得到 $s(c)=0.8571>threshodValue$,$c=\{getKey,getValue,tr,\text{false},\text{null},this\}$,且集合 c 中项的个数等于函数调用实参的个数,则此函数调用实参间存在强关联规则;其次,通过训练 N-gram 模型生成评价概率为 $P(s)=0.1428<threshodValue$,$s=(id,source,translation,fuzzy,\text{null},this)$;最后,异常检测阶段把此低概率函数调用视为潜在异常,结合人工核实确定此函数调用为错误参数.

Table 4 Examples of Incorrect arguments detected by ANiaD

表 4 ANiaD 检测出的错误参数示例

SrtFilter.java	*PoFilter.java*

if (!*StringUtil.isEmpty*(*tr*)) {
entryAlignCallback.addTranslation(*en.getKey*(),
en.getValue(),*tr*,false,null,*this*);
}
	if (*entryAlignCallback* !=null) {
	entryAlignCallback.addTranslation(*id,*
	source,translation,fuzzy,null,this);
	}

RcFilter.java	*MozillaDTDFilter.java*

if (!*StringUtil.isEmpty*(*tr*)) {
entryAlignCallback.addTranslation(*en.getKey*(),
en.getValue(),*tr*,false,null,*this*);
}
	if (!*StringUtil.isEmpty*(*tr*)){
	entryAlignCallback.addTranslation(*en.getKey*(),
	en.getValue(),*tr*,false,null,*this*);
	}

INIFilter.java	*ResourceBundleFilter.java*

if (!*StringUtil.isEmpty*(*tr*)) {
entryAlignCallback.addTranslation(*en.getKey*(),
en.getValue(),*tr*,false,null,*this*);
}
	if (!*StringUtil.isEmpty*(*tr*)) {
	entryAlignCallback.addTranslation(*en.getKey*(),
	en.getValue(),*tr*,false,null,*this*);
	}

MagentoFilter.java	
if (!*StringUtil.isEmpty*(*tr*)) { *entryAlignCallback.addTranslation*(*en.getKey*(),*en.getValue*(),*tr*,false,null,*this*);}	

深入分析,表 4 所示案例是通过 ANiaD 检测出的错误参数,而基于相似度的方法不能够检测出此错误参数.究其原因,ANiaD 是一种数据关联性驱动的大数据异常检测方法.方法通过挖掘整个项目中函数调用间的关联信息,获取项目中极少见的函数调用模式.这种方法的优点在于能够自适应编程人员的编程习惯与软件自身独有的特点.而基于相似度的检测方法的核心是比较实参与形参的文本相似度与上下文环境中其他候选实参与形参的文本相似度.这种方法利用的有限知识是函数形参标识符名称,应用前提是编程人员具有良好的标识符命名习惯,具有一定程度的局限性.

此外,通过分析误报案例,发现造成误报的函数调用处于极为不常见的代码片段中.究其原因分为两种:(1) 代码片段与项目整体代码风格不一致;(2) 代码片段处于较为特殊的使用场景.值得注意的是,这些正是值得编程人员格外关注的代码段.确保这些代码段的正确性,能够促进系统的正确性与稳定性.

当程序员向函数调用传入错误参数时,可能会导致程序模块功能的丧失或影响程序执行路径等后果,从而影响程序的正确性.例如,在 Hsqldb 项目中出现的函数调用 *updateColmnWhere*(“*c_binary*”,“*id*”)在新版本中被修改为 *updateColmnWhere*(“*c_binary*”,“*id*”),其中,“*c_varbinary*”是指更新的表名,“*id*”是指更新的列名.尽管这个参数错误不会造成编译或运行时错误,但是它造成了错误的数据库更新,底层数据出现错误会对项目造成致命影响.在 itext 项目中,*Assert.assertEquals*(*new PdfName*(“*name*”),*a.getPdfObject*(3)) 在新版本中被替换为 *Assert.assertEquals*(*new PdfName*(“*name*”),*a.get*(3)),*Assert* 类提供了很多在编写测试用时常用的断言方法,这些方法用于记录失败的断言.此方法用来检查两个变量或者两个等式是否平衡.当函数调用传入错误对象时,会对函数的判别造成影响,会直接造成错误的断言记录.

(3) RQ3:ANiaD 方法对大中小型项目的检测效果差异

正如第 3.2 节所述,本实验选取流行度较高的 10 个 Java 开源项目作为实验对象.这些项目涵盖小型、中型和大型应用软件.为分析 ANiaD 方法在不同规模软件项目中的适用性,需分类统计大、中、小规模项目的检测效果.

- 首先,根据项目规模对 10 个开源项目进行分类,把可执行代码行数小于 5 万行的项目作为小型项目,把可执行代码行数大于 5 万行、小于 10 万行的项目作为中型项目,把可执行代码行数大于 10 万行的项目作为大型项目.
- 其次,使用缺陷覆盖率($C_{anomaly}$)来衡量不同规模项目的检测效果.缺陷覆盖率的定义如公式(11)所示.

$$C_{anomaly}=\frac{Bugs+Refs}{LOC}\times 10000 \tag{11}$$

其中,*Bugs* 代表代码错误量,*Refs* 代表重构机会量,*LOC* 代表软件可执行代码行数.$C_{anomaly}$ 代表每万行代码中检

测到的代码缺陷量.

统计结果如表 5 所示,小型项目的平均缺陷覆盖率为 0.801 2,中型项目的平均缺陷覆盖率为 0.583 7,大型项目的平均缺陷覆盖率为 0.322 3.由此可知,ANiaD 在小型项目中检测效果最好,中型项目次之.究其原因,小型项目适用于小团队开发,更容易保证良好的代码规范与代码的一致性.ANiaD 方法的核心是挖掘同一函数的多次函数调用的重复性和关联性,代码的一致性能够为此提供保障,进而获得更好的检测效果.

Table 5 Detection results by ANiaD in applications of different sizes

表 5 ANiaD 在不同规模项目中的检测结果

名称	项目类型	$C_{anomaly}$	平均
Freemarker	小	0.967 5	0.801 2
UniversalMediaServer	小	0.208 6	
Purdue-fastr	小	1.227 4	
Omegat	中	0.188 8	0.583 7
itext7	中	0.713 5	
Hibernate	中	1.092 7	
Plantuml	中	0.339 8	
Hsqldb	大	0.540 3	0.322 3
Mondrian	大	0.355 8	
Vuze	大	0.070 8	

(4) RQ4:ANiaD 异常检测方法的响应时间

正如第 3.1 节所述,异常检测响应时间会影响异常检测方法的应用价值.在 ANiaD 异常检测方法中,时间主要消耗在两个阶段:代码解析时间和检测算法执行时间.其中,代码解析时需要获取大量的源代码数据并存储到数据库中,需要耗费较多的时间.

在表 6 所示的运行环境下,对实证研究中开源项目的异常检测时间进行统计.统计结果见表 7,异常检测的平均响应时间为 12.175s,异常检测响应时间在 4.135s~41.386s 之间,所有项目的异常检测过程都可以在 45s 内完成.此外,代码解析时间平均为 9.258s,检测算法执行时间平均为 2.917s.

Table 6 Runtime environment

表 6 运行环境

名称	描述
机器	MacBookPro
处理器	2.2GHzIntelCorei7
内存	16GB1600MHzDDR3
图形卡	IntelIrisPro1536MB
操作系统	MacOSSierra

Table 7 Response time of bug detection

表 7 异常检测的响应时间

项目	时间消耗		
	代码解析	检测执行	总时间
Mondrian	14.959	4.369	19.328
itext7	3.526	2.049	5.575
UniversalMediaServer	5.010	2.165	7.175
Vuze	32.918	8.468	41.386
Hibernate	9.222	1.911	11.133
Plantuml	6.899	1.496	8.395
Purdue-fastr	3.481	2.158	5.639
Freemarker	2.553	1.582	4.135
Hsqldb	8.106	3.390	11.496
Omegat	5.905	1.578	7.483
平均	9.258	2.917	12.175

结果表明,ANiaD 异常检测方法的检测响应时间在程序员可接受的范围内,不会对程序员的生成效率造成

较大影响.此方法具有很高的应用价值,后续可以基于 ANiaD 方法制作测试辅助工具为测试工作提供帮助.

4 相关工作

(1) 统计语言模型

目前,许多研究都证实了利用自然语言模型对程序源代码进行建模的有效性.Hindle 等人[5]提出,程序源代码同样具有自然语言的重复性和可预测性的特点,并且通过实验证实了利用统计语言模型能够对程序源代码进行有效的建模,并且深入证实了 *N*-Gram 语言模型能够用于捕获软件源代码间的规律.Ray 等人[13]分析了 10 个不同的 Java 项目在提交时记录的 7 139 个程序错误,重点关注含有错误的程序语句的语言统计特征.统计发现,含有错误的程序语句往往更加的不常见.

统计语言模型已经广泛的应用于软件工程相关领域,例如代码补全[14−17]、代码风格一致性检测[18,19]等.Han 等人[20]提出了一种用于推断下一个标识符的基于隐马尔可夫模型的算法.Oda 等人[21]利用 *N*-Gram 语言模型从项目源码中自动生成代码.Hsiao 等人[22]从互联网上收集了 280 万个 JavaScript 程序,在这个语料库上建立了 *N*-Gram 程序语言模型,通过实验证实了 *N*-Gram 语言模型在克隆检测应用中有出色的表现.Nessa 等人[23]利用 *N*-Gram 分析方法对软件中可执行代码段的可疑水平进行估计,根据可疑水平对可执行代码段进行排序.通过实验证实,这种方法能够有效地缩减排除程序错误的时间消耗.以上的研究表明,利用统计语言模型对程序源代码进行建模是挖掘程序源代码信息的一个有效途径,在缺陷检测领域有很高的应用价值和宽阔的应用前景.

通过阅读大量的 Java 开源代码发现,同一函数定义对应的多个函数调用的实参序列间存在潜在的关联性.本文工作充分利用同一函数的多次函数调用的重复性和关联性,挖掘实参序列间的关联信息,提取潜在的编程规则,从而实现函数调用相关的异常检测.

(2) 缺陷预测与规则挖掘

目前,许多编程规则挖掘与异常检测技术已经出现[24−27].这些技术能够从源代码、版本历史或源代码注释中获取潜在的编程规则.Jiang 等人[25]实现从历史提交记录中挖掘规则:首先,从版本控制系统的提交记录中解析源代码变化;其次,通过分组和聚合相关的代码变化获取代码提交的轨迹模式.实验证实:借助这些模式和不同开发时期代码提交轨迹模式的变化,能够发现软件进化规则和潜在缺陷,有助于软件版本管理和质量保证.Dai 等人[26]提出了一种静态分析方法:首先,从源代码中挖掘面向对象组件相关的方法;然后,利用方法的相关性引申出事件的相关性.Qian 等人[28]提出了一种挖掘软件中逻辑克隆的方法,以揭露高水平的软件业务规则与编程规则.

与现有的规则挖掘方法不同的是,提出的 ANiaD 方法着重于挖掘函数调用中实参选择的潜在规则,目前尚未有统计语言模型用于函数调用中错误参数检测的研究出现.在现有的错误参数检测技术中,Liu 等人[4]提出了一个静态检测方法.方法的核心思想是计算出实参和形参间的文本相似度,如果实参候选集中其他的选项与形参的文本相似度明显较高,则视为一个潜在缺陷.这种方法仅仅利用标识符名称信息,具有一定程度的局限性.Gao 等人[29]提出了一种从错误描述中生成 OpenAPI 使用规则的方法,但这种方法只有在满足 Open API 平台文档中具有详细的错误描述的条件下,才能生成与参数相关的 API 使用规则.

5 总结与展望

本文提出了一种基于关联分析和 *N*-Gram 语言模型的自动化静态异常参数检测方法(ANiaD),它能够挖掘出参数间存在强关联规则.基于 *N*-Gram 模型对参数间存在强关联规则的函数调用的正确性进行评价,低概率的函数调用被报告为潜在异常.在实证研究阶段,我们利用 10 个 Java 开源项目对 ANiaD 方法和基于相似度的方法进行对比.实验结果表明,ANiaD 的查准率为 43.40%,共检测出 18 个错误参数和 28 个需要重构的代码片段;而基于相似度的方法检测相关异常的查准率为 25%,只检测出 2 个错误参数和 5 个需要重构的代码片段.

本文方法是通过挖掘检测项目本身来获取编程规则,在未来可考虑在海量开源项目中学习来获取编程规则.此外,为缩短代码解析环节的时间消耗,可采用增量式的解析方案.在首次执行缺陷检测时,保存项目的解析

数据;再次执行缺陷检测时,只对已修改的文件进行代码解析,并对已修改文件对应的解析数据进行更新.

References:

[1] Rajala N, Campara D, Mansurov N. Method and apparatus for identifying indirect messaging relationships between software entities. Int'l Cl, 2001.

[2] Buckley A, Rose J, Darcy J. Method and system for compiling a dynamically-typed method invocation in a statically-typed programming language. Int'l Cl, 2013.

[3] Bruce KB. Safe type checking in a statically-typed object-oriented programming language. In: Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1993. 285−298.

[4] Liu H, Liu Q, Staicu CA, Pradel M, Luo Y. Nomen est Omen: Exploring and exploiting similarities between argument and parameter names. In: Proc. of the 38th Int'l Conf. on Software Engineering. New York: ACM Press, 2016. 1063−1073.

[5] Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 2012. 837−847.

[6] Baker ZK, Prasanna VK. Efficient hardware data mining with the apriori algorithm on FP-GAs. In: Proc. of the 13th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Piscataway: IEEE Press, 2005. 3−12.

[7] Suen CY. *N*-Gram statistics for natural language understanding and text processing. IEEE Trans. on Pattern Analysis & Machine Intelligence, 1979,1(2):164−172.

[8] Doddington G. Automatic evaluation of machine translation quality using *N*-Gram co-occurrence statistics. In: Proc. of the 2nd Int'l Conf. on Human Language Technology Research. San Francisco: Margan Kaufmann Publishers, 2002. 138−145.

[9] Cavnar W, Trenkle J. *N*-Gram-Based text categorization. In: Proc. of the 3rd Annual Symp. on Document Analysis and Information Retrieval. Las Vegas: Information Science Research Institute, 1994. 161−175.

[10] Ye Y, Chiang CC. A parallel apriori algorithm for frequent itemsets mining. In: Proc. of the 4th Int'l Conf. on Software Engineering Research, Management and Applications. Washington: IEEE Computer Society, 2006. 87−94.

[11] Sidorov G, Velasquez F, Stamatatos E, Chanona-Hernandez L. Syntactic *N*-Grams as machine learning features for natural language processing. Expert Systems with Applications, 2014,41(3):853−860.

[12] Fowler M. Refactoring: Improving the design of existing code. In: Proc. of the 1st Agile Universe Conf. on Extreme Programming and Agile Methods. London: Springer-Verlag, 2002. 256.

[13] Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P. On the "Naturalness" of buggy code. In: Proc. of the 38th Int'l Conf. on Software Engineering. New York: ACM Press, 2016. 428−439.

[14] Han S, Wallace DR, Miller RC. Code completion from abbreviated input. In: Proc. of the 24th IEEE/ACM Int'l Conf. on Automated Software Engineering. Washington: IEEE Computer Society, 2009. 332−343.

[15] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2014. 419−428.

[16] Hill R, Rideout J. Automatic method completion. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering. Washington: IEEE Computer Society, 2004. 228−235.

[17] Nguyen AT, Hilton M, Codoban M, Nguyen HA, Mast L, Rademacher E, Nguyen TN, Dig D. API code recommendation using statistical learning from fine-grained changes. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2016. 511−522.

[18] Allamanis M, Barr ET, Bird C, Sutton C. Learning natural coding conventions. In: Proc. of the 22st ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2014. 281−293.

[19] Hellendoorn VJ, Devanbu PT, Bacchelli A. Will they like this? Evaluating code contributions with language models. In: Proc. of the 12th Working Conf. on Mining Software Repositories. Piscataway: IEEE Press, 2015. 157−167.

[20] Han S, Wallace DR, Miller RC. Code completion from abbreviated input. In: Proc of the 24th IEEE/ACM Int'l Conf. on Automated Software Engineering. Washington: IEEE Computer Society, 2009. 332−343.

[21] Oda Y, Fudaba H, Neubig G, *et al*. Learning to generate pseudo-code from source code using statistical machine translation (T). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Washington: IEEE Computer Society, 2015. 574−584.

[22] Hsiao CH, Cafarella M, Narayanasamy S. Using Web corpus statistics for program analysis. In: Proc. of the 2014 ACM Int'l Conf. on Object Oriented Programming Systems Languages & Applications. New York: ACM Press, 2014. 49−65.

[23] Nessa S, Abedin M, Wong WE, Khan L, Qi Y. Software fault localization using *N*-Gram analysis. In: Proc. of the 3rd Int'l Conf. on Wireless Algorithms, Systems, and Applications. Berlin, Heidelberg: Springer-Verlag, 2008. 548−559.

[24] Tan L, Yuan D, Krishna G, Zhou Y. /* iComment: Bugs or bad comments? */. In: Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles. New York: ACM Press, 2007. 145−158.

[25] Jiang Q, Peng X, Wang H, Xing Z, Zhao W. Understanding systematic and collaborative code changes by mining evolutionary trajectory patterns. Journal of Software: Evolution and Process, 2017,29(3):e1840.

[26] Dai Z, Mao X, Chen L, Lei Y, Zhang Y. Finding related events for specification mining. In: Proc. of the 2013 IEEE Int'l Symp. on Software Reliability Engineering. Washington: IEEE Computer Society, 2014. 1−2.

[27] Zhu Z, Zou Y, Xie B, Jin Y, Lin Z, Zhang L. Mining API usage examples from test code. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. Washington: IEEE Computer Society, 2014. 301−310.

[28] Qian W, Peng X, Xing Z, Jarzabek S, Zhao W. Mining logical clones in software: Revealing high-level business and programming rules. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Washington: IEEE Computer Society, 2013. 40−49.

[29] Gao C, Wei J. Generating open API usage rule from error descriptions. In: Proc. of the 2013 IEEE 7th Int'l Symp. on Service-Oriented System Engineering. Washington: IEEE Computer Society, 2013. 245−253.

李超(1992－),男,河南商丘人,硕士,主要研究领域为软件工程.

刘辉(1978－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程.