软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(11):4046–4060 [doi: 10.13328/j.cnki.jos.006373]

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

基于图模型和孤立森林的上帝类检测方法*

刘 弋[1,2], 吴毅坚[1,2], 彭 鑫[1,2], 闫亚东[1,2]

[1](复旦大学 软件学院, 上海 200438)

[2](上海市数据科学重点实验室(复旦大学), 上海 200438)

通信作者: 吴毅坚, E-mail: wuyijian@fudan.edu.cn

摘 要: 上帝类(God class)是指同时包含多种任务职责的类, 其常见特征是包含大量的属性与方法, 并且与系统中的其他类有多种依赖关系. 上帝类是一种典型的代码坏味, 对软件的开发维护产生负面影响. 近年来, 许多研究都致力于发现和重构上帝类, 但是现有方法识别上帝类的能力不强, 检测精确率不高. 提出了一种基于图模型和孤立森林的上帝类检测方法, 主要分为两个阶段: 图结构信息分析阶段和类内度量评估阶段. 在图结构信息分析阶段, 建立类间的方法调用图和类内结构图, 采用孤立森林算法缩小上帝类的检测范围; 在类内度量评估阶段, 考虑项目的规模和架构带来的影响, 将项目中上帝类相关度量指标的平均值作为基准, 设计实验确定比例因子, 并以平均值和比例因子的乘积作为阈值, 筛选得到上帝类的检测结果. 在代码坏味标准数据集上的实验结果表明: 与现有的上帝类检测方法相比, 该方法在精确率和 $F1$ 值上分别提升了 25.82 个百分点和 33.39 个百分点,同时保持了较高的召回率.

关键词: 代码坏味; 软件维护; 上帝类; 孤立森林

中图法分类号: TP311

中文引用格式: 刘弋, 吴毅坚, 彭鑫, 闫亚东. 基于图模型和孤立森林的上帝类检测方法. 软件学报, 2022, 33(11): 4046–4060. http://www.jos.org.cn/1000-9825/6373.htm

英文引用格式: Liu Y, Wu YJ, Peng X, Yan YD. God Class Detection Approach Based on Graph Model and Isolation Forest. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 4046–4060 (in Chinese). http://www.jos.org.cn/1000-9825/6373.htm

God Class Detection Approach Based on Graph Model and Isolation Forest

LIU Yi[1,2], WU Yi-Jian[1,2], PENG Xin[1,2], YAN Ya-Dong[1,2]

[1](Software School, Fudan University, Shanghai 200438, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200438, China)

Abstract: God class refers to a class that carries heavy tasks and responsibilities. The common feature of God class is that it contains a large number of attributes and methods, and has multiple dependencies with other classes in the system. God class is a typical code smell, which has a negative impact on the development and maintenance of the software. In recent years, many studies have been devoted to discovering or refactoring the God class; however, the detection ability of existing methods is not strong, and the detection precision is not high enough. This study proposes a God class detection approach based on graph model and isolation forest algorithm, which can be divided into two stages: The stage of the graph structure information analysis and the stage of intra-class measurement evaluation. In the stage of the graph structure information analysis, inter-class method call graphs and intra-class structure graphs are established, respectively. The isolation forest algorithm is used to reduce the detection range of God class. In the stage of the intra-class measurement evaluation, the impact of the scale and architecture of the project is taken into account, and the average value of the God class related measurement indicators in the project is used as the benchmark. An experiment is designed to determine the scale factors, and the product of the average value and the scale factors are used as the threshold for the detection to obtain the God class detection result. The experimental results on the code smell benchmark data set show that the method proposed in this article improves the precision and $F1$

* 基金项目: 国家重点研发计划(2016YFB1000801)

收稿时间: 2020-11-19; 修改时间: 2021-01-29; 采用时间: 2021-05-08; jos 在线出版时间: 2021-05-20

value by 25.8 percentage points and 33.39 percentage points, respectively, compared to an existing God class detection method, with a high recall at the same time.

Key words: code smell; software maintenance; God class; isolation forest

代码坏味主要用于描述面向对象代码中不合理的设计[1], 其违背了面向对象设计的原则和规范, 影响代码的开发维护效率, 还可能导致软件质量下降. 如今, 随着软件系统的复杂度越来越高, 开发交付周期越来越短, 开发人员往往会在巨大的开发交付压力下放弃良好的编程规范和设计原则, 将代码坏味引入到系统中[2], 给软件维护和质量带来隐患. 高效准确地发现系统中的代码坏味, 对提升维护效率和代码质量变得非常重要.

Fowler 等人整理并总结了 22 种代码坏味(code smell)[1], 其中, 上帝类(god class)是一种典型的代码坏味, 指的是具有过多职责和功能的类. 上帝类违背单一职责原则, 往往难以理解和维护; 上帝类还违背开闭原则, 类内的耦合性过高, 可扩展性差, 当引入新功能时, 通常需要对类内大量的代码进行修改.

实验研究表明: 与具有上帝类的系统相比, 没有上帝类的系统会有更高的完整性、正确性与一致性[3], 并且随着软件的演化, 上帝类会变得更加难以理解和维护, 其内聚度也会下降. 因此, 在开发的过程中识别上帝类并及时重构清除是非常必要的[4]. 目前, 很多研究[5−11]都致力于检测出代码中的上帝类, 但是目前大部分方法检测上帝类的精确率都比较低[12], 这也是目前上帝类检测的一个最大的挑战. 大多数研究都只关注于类内信息, 即只从类内部提取信息或者度量指标作为特征实现对上帝类的检测, 而没有从系统的角度对类在整个系统中扮演的角色进行分析与研究. 上帝类这种代码坏味不仅表明该类的内部设计结构存在问题, 还表明该类在整个系统中承担了过多职责, 系统整体设计也存在问题. 因此, 检测上帝类不仅需要提取类内特征作为判别标准, 还需要提升到系统架构这样宏观的角度去分析该类的角色和职责. 也正因为没有从系统的角度进行研究与挖掘, 目前上帝类检测算法的精确率和 *F*1 值都比较低, 难以实际应用.

本文提出了一种基于图模型和孤立森林的上帝类检测方法, 主要分为两个阶段: 图结构信息分析阶段和类内度量评估阶段. 图结构信息分析阶段利用类间调用图和类内结构图信息缩小上帝类的检测范围, 类内度量评估阶段从筛选后的类集合中通过对关键度量值的评估确定上帝类. 在图结构信息分析阶段, 从源代码中抽取方法调用信息, 构建类间方法调用图, 并对图中每个节点计算其节点中心度[13], 用于对每个类在系统中的重要程度进行建模, 并将节点中心都作为筛选上帝类的特征; 进而, 对每个类构建类内结构图, 根据上帝类内聚度比较低、职责较多的特点, 对每个类使用社区发现算法挖掘该类的内部社区数, 作为上帝类的又一重要特征. 我们将得到的节点中心度度量指标、内部类的个数和类内结构图的社区数目作为孤立森林算法的输入, 进行上帝类的筛选, 大大缩小了上帝类的候选范围. 在类内度量评估阶段, 我们选用类的加权方法数(weighted methods per class, WMC)、类内聚的紧密程度(tight class cohesion, TCC)、访问外部数据的个数(access to foreign data, ATFD)和类内方法个数(number of methods per class, NOM)等和上帝类有关的度量作为指标. 考虑到不同的系统可能采用不同的架构和设计, 复杂程度都会有很大的差异, 因此我们以这些度量在系统中的平均值和比例因子的乘积作为阈值进行筛选, 通过实验调优, 得到了比例因子的最佳组合, 避免项目结构和设计对检测结果带来的影响, 最终得到上帝类的检测结果.

本文第 1 节对所用算法的背景知识进行说明和介绍. 第 2 节介绍上帝类检测的相关工作. 第 3 节重点阐述我们方法的设计思路和实现过程. 第 4 节介绍我们设计的 4 个评估实验和实验结果. 第 5 节对本文内容进行总结, 并展望未来的工作.

1 背景知识

1.1 抽象语法树

抽象语法树(abstract syntax tree, AST)是用树的形式来表示源代码. 抽象语法树用节点来存储代码里的各类信息, 其根节点对应着一个顶层编译单元, 直接与根节点相连接的节点表示源代码中的各种高级特征, 例如导入声明节点和类节点. 对方法声明节点而言, 其方法名称、返回值类型、方法体、方法修饰符等都是它

的子节点. 抽象语法树就是通过这种形式来实现代码信息的存储. 我们从抽象语法树中提取方法调用信息, 利用调用信息构建方法调用图.

我们使用开源 Java 代码解析工具 JavaParser[14]对源码进行解析. JavaParser 提供了一种访问者模式, 利用访问者模式, 可以方便地从抽象语法树中提取需要的信息.

1.2 PageRank算法

PageRank 是由谷歌的创始人 Page 等人提出的网页排序算法[13], 该算法根据网页的入链数量和网页质量对每一个网页计算出 PageRank 值(PR 值), 并用 PR 值来表示一个网页的重要性程度. 该算法主要基于两种关键假设: 数量假设和质量假设. 数量假设是指链接到一个网页的网页越多, 该网页越重要; 质量假设是指链接到一个网页的网页质量越高, 该网页越重要. 我们使用 PageRank 算法计算出系统中每个类在方法调用图中的重要性程度.

1.3 孤立森林算法

孤立森林(isolation forest)算法由 Liu 等人提出, 一般用于结构化数据的异常检测[15,16]. 孤立森林算法将异常数据定义为容易被孤立的离群点, 即分布上较为稀疏并且与密度高的群体距离较远的数据点. 孤立森林算法采用了一种非常高效的策略, 递归地随机分割数据集, 直到所有的样本点都是孤立的. 当数据发生在某个区域的概率比较低, 该算法会认为落入到该区域的数据是异常的.

孤立森林算法采用随机分割数据集的方法, 异常点在孤立树上的路径较短, 容易在算法迭代的过程中被孤立分割, 而密度较高的正常点则需要进行多次切分才会被划分到孤立的区域. 孤立森林通过这种方法从数据集中检测出异常点.

图 1 是用孤立森林算法进行异常数据检测的一个样例, 图中白色节点是用于训练孤立森林的数据样本; 蓝色节点是测试数据集中被检测为正常数据的样本; 红色节点是测试数据集中被检测为异常数据的样本. 从图中可以看出: 被识别为异常数据的样本大多分布在数据密度较低的区域中, 并且数量较少.

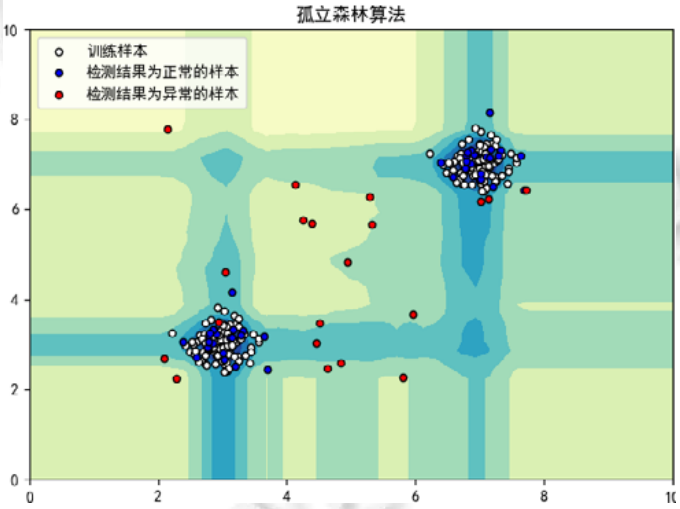


图 1 利用孤立森林算法进行异常数据检测的样例

1.4 社区发现算法

社区发现(community detection)算法主要用于发现网络中的社区结构[17], 社区即网络图中具有模块结构特性的子图结构, 社区内部节点之间相较于网络图中其他节点而言有着更加密切的联系, 而社区与社区之间的联系则比较稀疏. 整个网络是由若干个社区所组成的, 社区是一个包含顶点和边的子图.

根据使用挖掘算法的不同, 社区发现算法可以分为基于层次聚类的社区发现算法、基于随机游走的社区

发现算法和基于模块度(modularity)的社区发现算法. 在本文中, 我们采用了基于模块度的算法 Louvain[17], 相比于其他算法, 该方法在挖掘效率和挖掘效果上都有一定优势. 模块度是用来对社区划分的质量进行评估的指标, 也可用于确定社区划分的数目. 理想的社区划分结果是: 社区内部的节点相似度较高, 而不同社区节点之间的相似度较低. 模块度的计算公式如下:

$$Q=\frac{1}{2m}\sum_{i,j}\left(A_{i,j}-\frac{k_i k_j}{2m}\right)\delta(c_i,c_j) \tag{1}$$

其中, Q 表示模块度; $A_{i,j}$ 表示节点 i 和节点 j 之间的边的权重, 不带权重的网络, 所有边的权重为 1; k_i 表示与节点 i 连接的边的权重之和; c_i 表示节点 i 所属的社区; m 为网络中边的总数. $\delta(c_i,c_j)$的取值定义为: 如果节点 i 和节点 j 在同一个社区, 则取值为 1; 否则为 0.

Louvain 算法的计算过程分为 4 步骤: (a) 将每个节点作为一个独立的社区; (b) 尝试将其分配到各个邻居节点所在的社区, 利用公式(1)计算分配前后的模块度, 进而得到分配后与分配前的模块度差值, 当该差值最大(且大于 0)时, 所分配的邻居节点作为分配结果; (c) 迭代计算直到所有节点所属社区不再发生变化, 图中所有节点被分为 4 个社区; (d) 根据划分结果对现有的图结构进行压缩, 重新进行分配直到整个图的模块度不发生变化.

2 相关工作

代码坏味的检测对提升软件质量非常重要, 也吸引到了很多研究者的关注[18−20]. Tufano 等人[21]对代码坏味产生的原因和引入系统的时间进行了研究. Tufano 等人[22]研究了测试代码坏味出现的原因和测试代码坏味与设计的关系. 一些研究者则关注于代码坏味与软件演化的关系[23]. Arcoverde 等人[24]对软件系统中代码坏味的演化过程进行了研究, 并指出, 代码坏味常常会被开发人员所忽视. Chatzigeorgiou 等人[25]研究了软件演化过程中代码坏味的变化情况, 代码坏味的数量是否会随着时间而增加、是否会因为人为的干预而消失等问题. Peters 等人[26]研究了代码坏味是否会影响到开发人员理解程序的过程.

上帝类作为一种较为典型的代码坏味, 也有很多相关的研究工作. 研究者们提出的检测上帝类的方法主要可以分为 3 种: 基于度量值的方法、基于机器学习的方法和基于深度学习的方法.

2.1 基于度量值的上帝类检测方法

基于度量值的检测方法主要依赖于从代码中提取结构信息与指标, 例如类的加权方法数、内聚度、访问外部数据的个数等指标, 然后分别给不同的指标设立阈值, 将各项指标都满足阈值约束条件的类识别为上帝类[5−7,27].

Marinescu 等人[8]提出了利用检测策略来对软件的设计质量进行评估的方法, 将系统的设计质量抽象为度量指标的数值的高低, 帮助开发人员和维护人员检测并定位出系统中的设计问题. 数据过滤也是基于检测的策略中常用的一种手段, 阈值过滤的技术可以减少初始数据集的规模, 保留具有特殊特征的值. Munro 等人[9]提出了利用模板来进行上帝类检测的算法, 他们设计的模板包含 3 个方面的信息: 代码坏味的类别、基于文本的特征描述和用于检测的启发式算法. Moha 等人[10]提出了利用领域特定语言对上帝类进行检测的算法, 基于此算法, 推出了代码坏味检测工具 DECOR. Palomba 等人[11]提出了利用代码的历史信息来检测代码坏味的方法, 并且在开源数据集的评估上取得了良好的效果. 基于度量值的上帝类检测方法需要指定阈值, 使其可以区分坏味和非坏味的实例, 但是不同的方法通常会选择不同的阈值, 阈值的变化也会极大地影响到检测器的性能. 并且使用指标度量的方式没有考虑到项目的特点和项目的规模, 导致实际应用中的检测精确率不高.

2.2 基于机器学习的上帝类检测方法

为了解决基于度量值的方法在阈值的选取上的主观因素对检测结果的影响, 近年来, 一些机器学习的方法也被应用于上帝类的检测上来, 并为上帝类的检测提供了一种新的视角[28]. Maiga 等人[29]提出了一种基于 SVM 的上帝类检测方法 SMURF, 解决了之前的检测算法不能进行增量检测的问题, 进一步提高了检测精确

率. Maiga 等人[30]提出了基于 SVM 的上帝类检测算法 SVMDetect, 并且通过实验证明了其方法的有效性. 除了 SVM 以外, 贝叶斯信念网络的算法同样广泛应用于上帝类的检测上. Khomh 等人[31,32]提出了基于贝叶斯信念网络的上帝类的检测算法. 与其他算法相比, 贝叶斯信念网络可以处理缺失的数据, 并且允许检测人员明确指定决策过程. 代码中的语义信息对上帝类的检测也是非常重要的[33], Ma 等人[34]利用 LSI 的算法对于代码和注释中的语义信息进行建模, 将概念性度量和代码圈复杂度结合来对上帝类进行识别, 文本信息的挖掘也给上帝类的检测指出了一个新的方向.

与基于度量值的检测方法相比, 基于机器学习模型的检测方法能够解决基于度量值方法在检测和识别上帝类过程中阈值选择的问题, 同时, 机器学习的模型能够学习到各种特征之间深层的关系, 因此, 在识别的精确率和召回率上也有一定的提升. 但是现有的机器学习的方法同样没有结合项目特征, 例如类在项目中的重要程度、项目的方法调用图等.

2.3 基于深度学习的上帝类检测方法

近些年来, 随着深度学习技术的快速发展, 一些研究者们将深度学习的模型应用到上帝类的检测上来. Bu 等人[12]提出了基于长短期记忆神经网络的上帝类检测方法, 除了将针对上帝类检测的传统度量指标作为神经网络的输入外, 他们还充分利用代码中的文本信息, 使用 Word2Vector 对预处理之后的文本进行训练, 将训练得到的代码文本特征向量也作为神经网络的输入. 基于神经网络的检测方法需要大量的训练数据来防止出现过拟合的现象, 他们还提出了自动生成神经网络的训练数据的算法, 避免了人工识别代码坏味所耗费的时间和人力. Wang 等人[35]提出了基于 BP 神经网络的代码坏味检测算法, 将度量指标作为神经网络的输入, 同时对比实验研究表明: 基于神经网络的方法相较于基于度量值的方法和基于机器学习的方法, 在识别精确率和 *F*1 值上有一定的提升.

与机器学习的方法相比, 基于深度学习的方法使用了复杂的神经网络, 更能够拟合输入特征与代码坏味之间的关联关系, 从而有更高的识别精确率和召回率. 但是这些方法只针对类内的特征, 利用类内的信息作为输入得到上帝类的识别结果, 而忽视了类与类之间的关联关系, 更没有深入挖掘与分析类在系统中扮演的角色, 所以检测的精确率并不高. 我们使用方法调用图将系统中各个类之间的调用协作关系表示出来, 同时利用图论中的中心度的概念对每个类在系统中的重要程度进行建模, 提取到了上帝类的类间特征.

3 方 法

本文方法的概览如图 2 所示, 主要分为两个主要阶段: 图结构信息分析阶段和类内度量评估阶段.

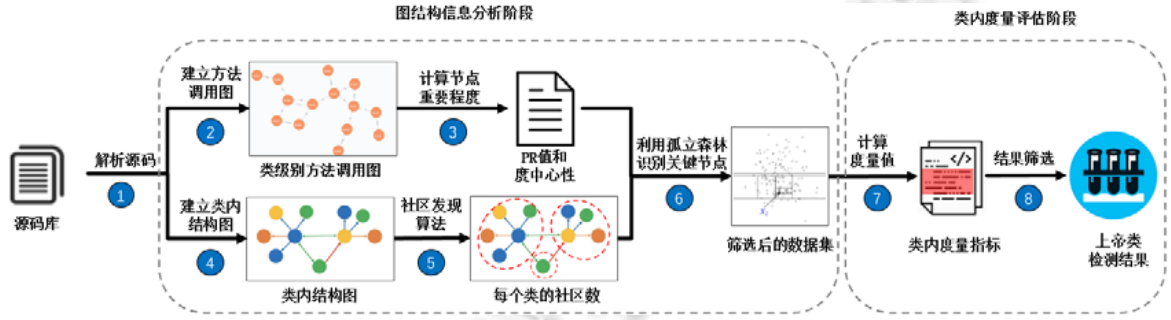


图 2 基于图模型和孤立森林的上帝类检测方法概览

- 在图结构信息分析阶段, 利用项目的类间关系, 筛选出项目中的关键类. 关键类指的是在系统中承担较重要的功能、内部结构较为复杂的类. 首先建立项目类粒度的方法调用图, 同时, 为了衡量每个节点在方法调用图中的中心程度(该类在整个项目中的所担任的职责), 根据方法调用图, 分别计算每个节点的 PR 值和度中心性. 我们提出了这种假设: 在项目中占中心地位的类不一定是上帝类, 但是在项目中占据边缘地位并且与其他类关系不紧密的类一定不是上帝类. 我们根据上帝类内聚度不高、关

注点较为分散的特点，建立项目的类内结构图，利用社区发现算法发现类内结构图中的社区数．每个类中内部类的个数也是和上帝类有关的一种代码特征，我们统计了每个类的内部类的个数．将上面得到的这些数据作为孤立森林的输入特征，以筛选出项目中关键类．

- 在类内度量评估阶段，利用类内信息，实现对上帝类的精确检测．我们计算出项目内每个类的加权方法数度(WMC)、内聚度(TCC)、对外访问数(ATFD)和类内方法个数(NOM)等代码结构特征指标，同时计算出每个项目中这些指标的平均值，对每一个项目来说，以这些指标的平均值和比例因子的乘积作为阈值进行筛选，最终得到上帝类的检测结果．

代码信息抽取的整个工作流程主要可以分为 3 个步骤：从待检测项目中提取 Java 源代码、利用静态分析工具将 Java 源代码解析为抽象语法树的表示形式、从抽象语法树中抽取方法调用信息并将信息保存到数据库．

对于每个待检测的系统，利用上一步提取的类间的方法调用信息构建系统的类粒度的方法调用图，并将其存入到图数据库 Neo4j 中．因为我们只关注与类间的关系，所以过滤了类内方法的相互调用，只保留了类间的方法调用．对于检测到的所有方法调用，只需保留其中项目内的方法调用，所以要过滤掉 JDK 的方法调用和第三方 API 的方法调用．

3.1 计算节点PR值和度中心性

上帝类在项目中扮演着重要的角色，上帝类一定是整个系统中较为关键的类．利用这个特点，我们可以从方法调用图中筛选出关键的类节点，从而缩小我们的检索空间．在建立方法调用图之后，为了衡量图中每个节点在整个调用图中的重要程度，我们使用到了节点中心度的概念．根据方法调用图计算图中每个类节点在整个方法调用图中的中心度，对每个类在整个项目中的重要程度进行建模．

我们分别选用 PageRank 算法计算出的 PR 值和图论中的度中心性(degree centrality)两种指标来衡量图中每个节点的重要程度．PageRank 最早用于计算网页排名，该算法同样可以应用于方法调用图中，用于计算出方法调用图中每个类节点的重要程度，计算过程如下．

(a) 给方法调用图中每个节点确定初始 PR 值，初始值为 1/N, N 为方法调用图中节点的总数．

(b) 通过链接到每个节点的权重除以调用节点的方法调用总数，连续更新每个节点的 PR 值．

(c) 重复上一步，直至所有节点的 PR 值不再发生变化．

度中心性是网络中衡量节点重要性的指标．度中心性使用一个节点的度数来衡量节点在图中的中心性，最早来源于社会网络研究，是研究人员为了确定社交网络中具有影响力的人而提出的概念．度中心性基于这样一种假设：节点拥有的边越多，节点就越重要．方法调用图中的每个节点，都需要计算其度中心性的数值．

开源项目 JEdit 方法调用图中部分节点的 PR 值和度中心性计算结果如图 3 所示，图中展示了 JEdit 项目中部分类的方法调用图．

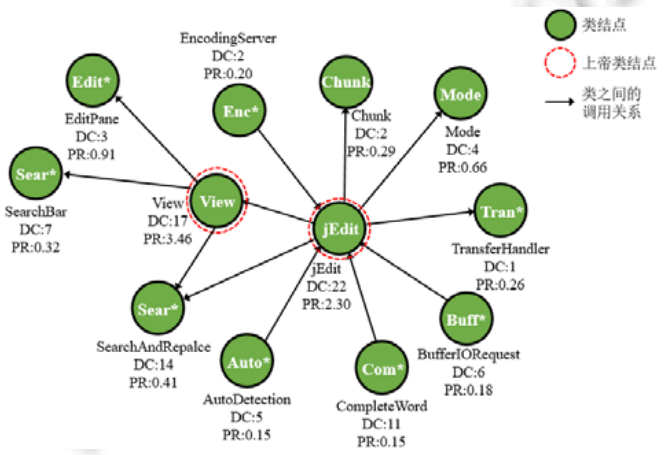


图 3 JEdit 项目方法调用图中部分节点 PR 值和度中心性计算结果

因为 JEdit 中类的数量过多，所以我们只挑选了部分类进行展示. 图中每个节点代表一个类，其中包含两个上帝类，分别是 jEdit 和 View: 类 jEdit 是整个应用程序的入口类，混杂了文件读写、界面处理和缓存操作等多种关联性不强的功能；类 View 主要用于处理整个程序视图相关的操作，但也包含了配置相关的功能. 我们计算出每个类在方法调用图中的 PR 值和度中心性值(在图中用 DC 表示). 从图中可以看出，上帝类 View 和 jEdit 的度中心性值和 PR 值明显要高于方法调用图中其他的类节点. 这说明 PR 值和度中心性可以有效地表示出上帝类的类间图结构特征.

3.2 建立类内结构图并发现社区数

上帝类的一个特征是单个类承担多种职责，内聚度不高. 这种特征在图中的表现形式为类内部的联系不够紧密，明显可以划分为几个社区. 我们使用社区发现算法对上帝类进行分析，利用社区数目来衡量一个类内部的离散程度.

我们给项目中每个类构建了类内结构图. 图中的节点类型有两种: 方法节点和类内变量节点；图中的关系类型有 3 种: 方法之间的调用(invoke)、方法使用类内变量(use)以及我们为了加强图中节点的联系而额外引入的方法同时被调用(co-invoked)关系. 组成一个功能内聚的几个方法不一定存在直接的调用关系，仅使用调用和使用两种关系不能完全建模出内聚关系，因此我们引入了同时被调用关系(co-invoked). 如果一个类中的方法同时被另外一个类调用，就使用同时被调用关系将其关联起来，利用同时被调用关系来加强内部节点之间的联系.

我们采用基于模块度的社区发现算法 Louvain 来挖掘每个类内部社区的数目, Louvain 使用模块度来描述社区内紧密的程度. 其算法计算的主要过程如下.

(a) 每个节点作为一个社区，初始的社区数目和节点的个数相同.

(b) 针对每个节点，遍历该节点的所有邻居节点，衡量把该节点加入其邻居节点所在社区所带来的模块度的收益，并选择对应最大收益的邻居节点，加入其所在社区. 重复计算，直到所有的社区都不再发生变化.

(c) 将新的社区作为节点，重复计算上一步的过程，并使用两个社区之间相邻点之间的权重作为两个社区退化成一个点后的新的权重.

图 4 给出了我们使用 Louvain 社区发现方法的一个示例，图中绿色的节点是方法节点，蓝色的节点是类内变量节点. 在此例中，一共挖掘出 3 个社区.

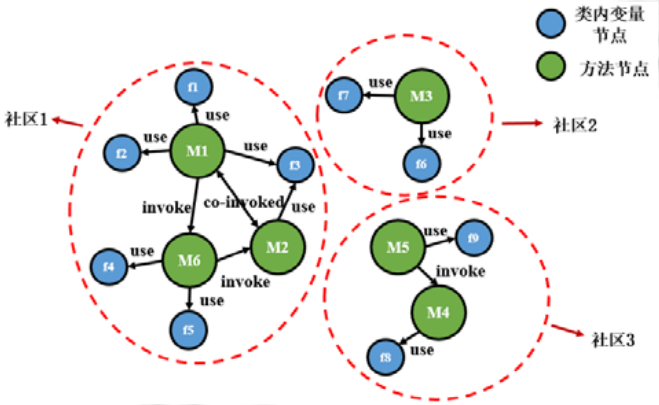


图 4 类内结构图的社区发现结果

3.3 使用孤立森林得到上帝类的候选结果

孤立森林算法主要用于检测异常数据，异常具有两个特点: 异常数据只占少量，并且异常数据的特征值和正确数据差别很大. 我们利用异常数据的这两个特点，将在整个项目方法调用图中的中心度较高并且内部

离散程度较高的节点设为异常节点, 使用孤立森林算法, 从我们的数据集中筛选出上帝类的候选结果.

除了上一步计算出的 PR 值和度中心性两个值以外, 我们还计算了每个类的内部类个数. 在 Java 里面, 定义在一个类里面或者一个方法体里面的类被称为内部类, 内部类一般分别 4 种: 成员内部类、局部内部类、匿名内部类和静态内部类. 相较于没有内部类的类而言, 有内部类的类一般有着更复杂的内部结构、更复杂的功能, 也意味着该类在整个系统中有着更重要的地位. 我们利用 PR 值、度中心性和内部类的数量等指标, 将在项目中结构复杂、较为重要的类当作孤立森林算法中待检测的异常节点, 利用孤立森林算法对数据集进行筛选, 进一步缩小了我们的检索范围, 方便后面实现对上帝类的精确检测.

孤立森林的算法主要可以分为两个阶段: 训练阶段和测试阶段. 训练阶段通过对数据集采样的方式建立孤立树; 在测试阶段, 用孤立树为每一个测试样本来计算异常分数.

构建孤立森林的算法如下所示.

(1) 初始化森林, 设置孤立树的个数为 T, 子采样的大小为 M.

(2) 设置树的高度限制为 $l=ceiling(\log_2 M)$.

(3) 对于每一棵树, 将输入数据进行采样得到新的数据, 放入树的根节点.

(4) 随机指定一个维度, 根据指定数据的极值, 在当前数据中随机生成切割点 p.

(5) 根据切割点生产的超平面, 将当前节点数据空间划分为两个不同的子空间, 则该维度空间中小于 p 的数据作为当前节点的左孩子, 大于 p 的数据作为当前节点的右孩子.

(6) 在每个孩子节点中递归进行步骤(4)和步骤(5), 重复构造新的孩子节点, 直到所有的样本都已经被孤立, 或者孤立树达到指定高度.

3.4 以项目平均值和比例因子的乘积作为阈值得到检测结果

基于度量值的上帝类检测方法使用固定的阈值来对上帝类进行筛选[5−7,27], 但是由于不同的项目规模不同, 并且可能采用了不同的框架或者架构, 所以不同项目中类的度量值也会有波动. 在这种情况下, 如果对所有的项目都是使用同样的评判标准或是检测阈值来进行上帝类的检测, 就会降低检测结果的精确率. 所以, 我们以项目中每个度量值的平均值和比例因子的乘积作为该项目的检测阈值, 如果一个类的各个度量值都不满足阈值的约束条件, 就会被视为是上帝类. 在指标的选择上, 我们选择了下面几个指标[36].

(1) 类的加权方法数(WMC): 一个类中所有方法的统计复杂度的和.

(2) 类内聚紧密程度(TCC): 通过访问相同的属性而发生联系的方法的个数.

(3) 访问外部数据的个数(ATFD): 对于一个给定的类, 它所访问的外部类的个数. 它可以直接访问这些外部类的属性, 也可以通过访问器方法访问这些属性.

(4) 类内方法个数(NOM): 即一个类中方法的个数.

上帝类的识别公式如下, 其中, $ATFD\_AVG$, WMC_AVG, NOM_AVG 和 TCC_AVG 分别是项目的 ATFD, WMC, NOM 和 TCC 的平均值. 我们以项目的平均值和比例因子的乘积作为阈值进行上帝类的检测, 并且设计了实验, 探究什么样的比例因子的取值会取得最佳的检测效果. 根据实验得到 ATFD, WMC, NOM 和 TCC 这 4 个比例因子的取值分别为 2, 5, 5 和 6:

$$(ATFD>2\times ATFD\_AVG \ \&\ WMC>5\times WMC\_AVG \ \&\ NOM>5\times NOM\_AVG \ \&\ TCC<6\times TCC\_AVG)\rightarrow isGodClass \quad (2)$$

4 实验验证

为了评估方法的有效性, 我们围绕以下 4 个问题开展了实验验证, 并与相关方法进行了对比.

- RQ1: 本文提出的方法是否优于现有方法?
- RQ2: 利用孤立森林算法是否能够缩小上帝类的检索范围?
- RQ3: 相关超参数对实验结果带来何种影响?
- RQ4: 本文提出的方法在时间性能上表现如何?

4.1 实验数据集和对比方法

为了与现有的方法进行对比，我们采用 Palomba 等人[37]发布的一个开源代码坏味数据集作为基准数据集，同时选取了和深度学习检测方法[12]测试集一样的 13 个项目来对本文提出的方法进行评估. 这个数据集中所有的代码坏味都已经经过人工验证. 相关项目信息见表 1.

表 1 测试数据集的信息

项目名称	项目版本	类的数量	上帝类的数目
Ant	v1.8.2	1 166	6
Derby	v10.9.1.0	2 797	26
FreeMind	v0.9.0	438	2
Hadoop	v0.9	238	1
HBase	v0.94	1 054	8
Ivy	v2.0.0	330	2
JEdit	v4.5.0	513	6
JHotDraw	v7.5.1	613	2
Pig	v0.8.0	1 021	3
Qpid	v0.18	2 217	6
Structs	v2.2.1	1 837	2
Wicket	v1.4.20	2 002	4
Xerces	v2.3.0	718	6

现有的上帝类检测方法主要分为 3 类: 基于度量值的方法、基于机器学习的方法、基于深度学习的方法. 由于 Wang 等人[35]的工作已经说明基于深度学习的方法远优于基于机器学习的方法，因此我们没有选择基于机器学习的方法进行对比，而是仅选择了基于度量值的代码坏味检测工具 JDeodorant[27]和采用深度学习的检测方法[12]进行对比. 所选的基于深度学习的方法采用 12 个开源 Java 项目作为训练数据集[12]. 据我们所知，它是目前效果最好的上帝类检测算法. 在对比实验中，选用了精确率、召回率和 $F1$ 值等评估指标对检测结果进行评价. 计算方法分别见公式(3)–(5). TP 是本身为上帝类且被识别为上帝类的类的数量; FN 是本身是上帝类，但识别结果不为上帝类的类的数量; FP 是本身不是上帝类但是被识别为上帝类的类的数量.

- 精确率(precision)的计算公式为

$$precision = \frac{TP}{TP + FP} \tag{3}$$

- 召回率(recall)的计算公式为

$$recall = \frac{TP}{TP + FN} \tag{4}$$

- $F1$ 的计算公式为

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{5}$$

4.2 检测结果对比(RQ1)

对比结果见表 2, JDeodorant 检测的精确率和召回率都比较低，说明 JDeodorant 不仅存在漏检现象,并且还会出现大量的误报.

从表中可以看出: 与 JDeodorant 这种传统的基于度量值的方法相比，我们的方法在精确率、召回率和 $F1$ 值上都取得了巨大的提升，分别提升了 27.02 个百分点、23.00 个百分点和 35.80 个百分点，符合实验预期. 这说明图信息对上帝类的识别是非常有效的，可以筛除掉大量的非上帝类节点，提高我们方法的精确率，并且以每个项目的平均指标和比例因子的乘积作为阈值来进行上帝类检测的策略，也都实现了预期的效果.

与深度学习的方法相比，我们的方法在精确率和 $F1$ 值上分别提升了 25.82 个百分点和 33.39 个百分点. 虽然我们的方法在召回率上低于深度学习的方法，但是在综合考虑了精确率和召回率的 $F1$ 指标上实现了 33.39 个百分点的提升，这说明本文方法在检测效果上总体优于深度学习的方法. 基于深度学习方法的召回率较高是因为其考虑了更多的特征信息，综合了 12 种不同类型的代码度量值作为神经网络的输入特征[12]，并且神经网络的结构较为复杂，参数量大，特征处理能力更强. 神经网络虽然能够识别出上帝类的一些特征，但是

对于一些特征和上帝类比较接近的类, 神经网络却不能有效对它们进行区分, 存在大量的误报现象. 深度学习方法检测的精确率只有 4.29%, 较低的精确率使其很难应用到工业界的实践中, 仍需要专业的维护人员进行进一步的人工识别与检测. 这说明之前的检测方法只依靠类内信息来识别与检测上帝类是远远不够的, 不仅仅要从内部来提取上帝类相关的特征, 更需要从整个系统的角度, 依靠系统的方法调用图来深入分析每个类在系统中所扮演的角色与承担的功能职责, 这样才可以更精确地定位出上帝类.

表 2 不同方法的上帝类检测结果对比

项目名称	我们的方法			JDeodorant			深度学习方法[12]		
	精确率(%)	召回率(%)	*F*1 (%)	精确率(%)	召回率(%)	*F*1 (%)	精确率(%)	召回率(%)	*F*1 (%)
Ant	31.25	83.33	45.45	1.70	50.00	3.29	5.13	100.00	9.76
Derby	50.00	70.37	58.46	4.01	42.30	7.33	5.05	92.30	9.58
FreeMind	28.57	100.00	44.44	4.08	100.00	7.84	4.88	100.00	9.31
Hadoop	16.67	100.00	28.58	0.00	0.00	0.00	3.84	100.00	7.40
HBase	42.86	75.00	54.54	2.52	62.50	4.84	6.25	100.00	11.76
Ivy	28.57	100.00	44.44	4.00	100.00	7.69	3.28	100.00	6.35
JEdit	75.00	100.00	85.71	4.40	66.70	8.26	6.98	100.00	13.05
JHotDraw	0.00	0.00	0.00	1.69	50.00	3.27	1.96	100.00	3.84
Pig	30.00	100.00	46.15	0.00	0.00	0.00	2.16	100.00	4.23
Qpid	14.29	66.67	23.54	3.23	83.30	6.22	3.49	100.00	6.74
Structs	16.67	100.00	28.58	1.64	100.00	3.23	0.98	50.00	1.92
Wicket	12.12	100.00	21.62	4.17	75.00	7.90	4.82	100.00	9.20
Xerces	45.45	83.33	58.82	8.82	50.00	14.99	6.98	100.00	13.05
平均值	30.11	82.98	41.56	3.09	59.98	5.76	4.29	95.56	8.17

为了深入分析检测结果, 我们分别统计分析了检测结果中的误报结果和漏报结果. 漏报结果是 16 条, 误报结果是 132 条. 漏报结果的 PR 值和度中心性值普遍较低, 漏报所有类的平均 PR 值和平均度中心性值分别为 0.49 和 7.81, 低于所有项目中上帝类的平均 PR 值 0.69 和平均度中心性值 9.30. 例如 JHotDraw 中被漏报的上帝类 org.jhotdraw.draw.DefaultDrawingView, 该类的 PR 值为 0.15, 度中心性值为 2, 远低于上帝类的平均 PR 值和平均度中心性值. 因为漏报类的图特征并不突出, 所以被本文提出的方法所漏报. 误报类的平均 PR 值为 1.15, 平均度中心性值为 8.95, 误报类的平均 PR 值要高于上帝类, 平均度中心性值与上帝类均值较为接近. 因为误报类的图特征值接近或高于上帝类, 所以被本文提出的方法所误报. 这也说明了如果我们想进一步提升检测的精确率和召回率, 仅靠 PR 值和度中心性值这两种图特征是不够的, 还需要结合其他图结构特征.

4.3 孤立森林算法有效性(RQ2)

为了探究我们提出的方法是否能够有效检索出项目中的上帝类, 进一步缩小我们的检索范围, 我们设计了实验对于这个过程进行评估. 我们分别统计了原数据集中类的总数和使用孤立森林之后类的总数, 统计结果见表 3.

表 3 孤立森林筛选前后数据对比

项目名称	原数据集		使用孤立森林筛选后			
	类的数量	上帝类数量	筛选后类的数量	筛选后上帝类的数量	筛选后类的数量占原数据集的比例(%)	筛选后上帝类的数量占原数据集的比例(%)
Ant	1 166	6	144	6	12.35	100.00
Derby	2 797	26	433	22	15.48	84.62
FreeMind	438	2	66	2	15.07	100.00
Hadoop	238	1	65	1	27.31	100.00
HBase	1 054	8	346	8	32.83	100.00
Ivy	330	2	58	2	17.58	100.00
JEdit	513	6	134	6	26.12	100.00
JHotDraw	613	2	59	0	9.62	0.00
Pig	1 021	3	275	3	26.93	100.00
Qpid	2 217	6	310	5	13.98	83.33
Structs	1 837	2	153	2	8.33	100.00
Wicket	2 002	4	174	4	8.69	100.00
Xerces	718	6	70	6	9.75	100.00
总数/总比例	14 944	74	2 287	67	15.30	90.54

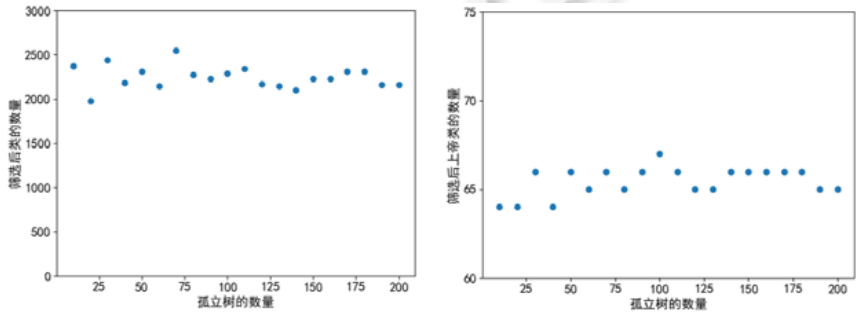
使用孤立森林算法前后数据集的对比结果如表 3 所示，可以看出，整个数据集的大小变为原来的 15.30%，但是上帝类的占比为原来的 90.54%. 可以看出，当我们在很大程度上压缩了整个数据集的大小时，上帝类的占比变化不大. 孤立森林算法缩小了上帝类的检索范围，有助于后面实现对上帝类的精确检测.

4.4 超参数影响(RQ3)

4.4.1 孤立树数量的影响

在我们的孤立森林算法中，对算法执行过程产生影响的主要是两个超参数：采样的样本大小 M 和构建孤立树的数量 T. 在我们的方法中，我们将采样的样本大小设置为 128，将构建孤立树的数量设置为 100. 为了研究这两个参数的变化会给实验结果造成怎么样的影响，以及什么样的参数选择是最优的，我们设计了下面的对比实验：我们先固定了采样的样本大小是 128，然后调整孤立树的数量，观察筛选效果.

数据集中，上帝类的个数随孤立树的数量变化情况如图 5 所示. 可以看到，随着孤立树数量的变化，筛选后数据集中上帝类的个数基本稳定在 65 个左右. 这说明在使用孤立森林对代码坏味数据集进行图结构信息分析的过程中，无论是构建 10 棵还是构建 200 棵孤立树，都能很好地对数据集进行训练和测试，达到较好的筛选效果.



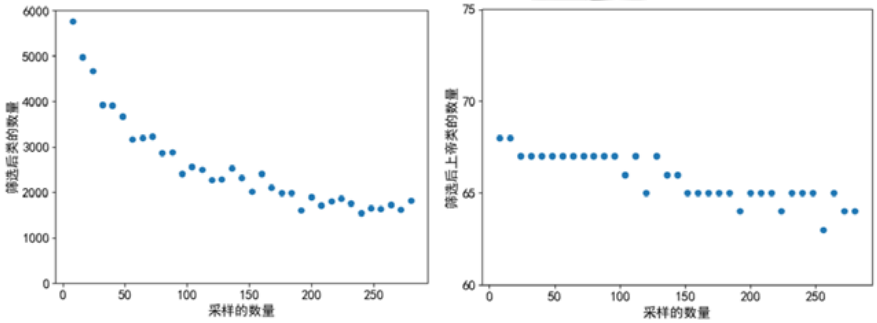
(a) 筛选后类的数量变化　　(b) 筛选后上帝类的数量变化

图 5　数据集中上帝类的个数和类的总数随孤立树数量变化情况

孤立树的数量从 10 变化到 200，对上帝类的筛选效果影响不大. 筛选后，数据集中上帝类的数量和类的数量上下波动，未出现明显的变化. 这说明构建孤立树数量在 10–200 之间都可以实现较好的数据过滤效果.

4.4.2 采样数量的影响

我们固定孤立树的数量为 100，然后调整采样数目,观察筛选效果. 根据实验可以看出：随着采样数量的增加，孤立森林的筛选条件越来越严格，精确率会得到一定的提高；但是召回率会下降. 为了在保证较高召回率的同时尽可能地提高精确率，我们选择 128 的采样数是比较合适的. 随着采样数量的变化，数据集中上帝类的数量变化和类总数的数量变化如图 6 所示. 可以看出，随着采样数量的增加，两者都呈下降趋势.



(a) 筛选后类的数量变化　　(b) 筛选后上帝类的数量变化

图 6　数据集中上帝类的数量和类的数量随采样数目变化情况

4.4.3 比例因子的影响

为了探究以项目指标的平均值和比例因子的乘积作为阈值筛选上帝类时, 比例因子对实验结果的影响, 我们选择了 4 个指标用于最后精确筛选出上帝类, 分别是类的加权方法数(WMC)、类内聚的紧密程度(TCC)、访问外部数据的个数(ATFD)、类内方法个数(NOM)这 4 个指标[8,36]. 我们分别定义了 4 个比例因子作为检测中的超参数, 并将指标平均值和比例因子的乘积作为检测的阈值. 检测的优化目标是: 在保证较高召回率的基础上[12,35], 尽可能地提高检测的精确率. 为了验证这些比例因子的选择对上帝类检测结果的影响, 设计了对比实验进行验证. 先控制其他 3 个因子的值不变, 修改其中一个因子, 寻找该因子最佳参数值. 重复这个过程,直到寻找到 4 种比例因子的最佳组合值. 实验结果如表 4 所示.

表 4 比例因子对实验结果的影响

ATFD 比例因子	WMC 比例因子	NOM 比例因子	TCC 比例因子	平均 精确率(%)	平均 召回率(%)	平均 *F*1 值(%)
1	4	4	3	21.08	76.00	31.47
2	4	4	3	22.92	76.00	33.46
3	4	4	3	24.79	70.90	35.08
4	4	4	3	27.81	65.21	36.96
5	4	4	3	29.29	60.79	37.29
2	1	4	3	18.81	76.00	28.90
2	3	4	3	19.73	76.00	29.93
2	4	4	3	22.92	76.00	33.46
2	5	4	3	26.58	76.00	37.18
2	6	4	3	31.89	73.43	41.22
2	5	1	3	21.87	77.85	32.45
2	5	2	3	21.98	76.57	32.50
2	5	3	3	24.14	76.28	34.82
2	5	4	3	26.58	76.00	37.18
2	5	5	3	30.51	76.00	41.23
2	5	6	3	36.84	69.37	45.02
2	5	5	1	30.98	63.92	38.47
2	5	5	2	30.93	74.72	41.43
2	5	5	4	30.60	77.56	41.46
2	5	5	5	30.15	81.41	41.26
2	**5**	**5**	**6**	**30.11**	**82.98**	**41.56**

根据实验结果可知, 当 ATFD, WMC, NOM 和 TCC 这 4 个比例因子分别为 2, 5, 5 和 6 时, 检测结果最符合我们的预期.

4.5 时间性能评估(RQ4)

为了评估本文所提出的方法的时间性能, 我们针对测试数据集上 13 个开源项目, 记录了本文提出的方法在各个步骤上完成计算所需要花费的时间(以 s 为单位), 见表 5. 实验所采用计算机的配置为 CPU Intel Core i5-8250U 1.60 GHz, RAM16.0 GB. 操作系统为 Windows 10 专业版.

表 5 在测试数据集上完成各个步骤所花费的时间

步骤名称	所花费的时间(s)
解析源码	391.6
建立方法调用图	20.0
计算节点中心度值	3.9
建立类内结构图	21.1
社区发现算法	13.7
利用孤立森林识别关键节点	6.9
计算度量值	86.0
结果筛选	19.9
总时长	563.1

由表 5 可以看出: 使用本文提出的方法完成测试数据集中 13 个项目的检测总时长是 563.1 s, 平均每个项目耗时 43.3 s. 其中, 耗时最久的步骤是解析源码, 占用总检测时长的 69.54%. 因为 JavaParser 需要对每段代

码进行解析、生成对应的抽象语法树、访问抽象语法树并取得需要的信息, 这个过程需要对大量的代码数据进行处理, 操作步骤较为复杂, 耗时较久. 从表中可以看出, 孤立森林算法识别关键节点仅用时 6.9 s. 这表明, 孤立森林算法是非常高效的.

4.6 有效性威胁

在对比评估实验中, 我们选用了开源代码坏味数据集中的 13 个开源项目作为测试项目, 数据集中的所有代码坏味都经过人工标注. 虽然与自动识别代码坏味的方法相比, 人工识别代码坏味有着更高的可信度, 但是人工标注的主观性差异也会对标注结果造成影响, 从而影响到标注结果的正确性. 此外, 仅用一个基准数据集难以验证我们的方法在其他数据集上也能取得预期的检测效果, 在后续的工作中, 我们将会用更多的开源数据集来对本文提出方法的正确性和有效性进行验证.

在探究比例因子的变化对实验结果带来何种影响的研究实验中, 我们选用了有限的比例因子组合进行实验并得到检测结果, 没有验证因子在全部取值范围内的实验结果. 选取范围的有限性, 使我们不能保证选取到了最优的比例因子组合. 为了使选取的比例因子组合尽可能地接近最优解, 我们设计了较多的组合值进行研究实验.

5 总结和展望

本文提出了基于图模型和孤立森林的上帝类检测算法, 主要可以分为两个阶段: 图结构信息分析阶段和类内度量评估阶段. 在图结构信息分析阶段, 我们的主要目的是利用图结构信息筛除项目中大量的非上帝类节点, 缩小上帝类的检测范围. 首先从项目源码中提取信息构建类粒度的方法调用图, 用类节点在方法调用图中的中心程度来衡量每个类在整个项目中的重要程度, 同时对每个类构建其类内结构图, 使用社区发现算法计算每个类内结构图中社区的数量, 用类内结构图中社区的数目对类的内聚程度进行建模. 为了尽可能缩小上帝类的检索范围, 我们使用了孤立森林的方法, 将图模型挖掘出的信息作为孤立森林算法的输入, 筛选出符合上帝类特征的类, 大大缩小检测范围. 在类内度量评估阶段, 为了进一步确定上帝类, 我们计算了和上帝类有关的度量值, 并计算出了这些度量值的项目平均值, 以度量值的项目平均值和比例因子的乘积作为阈值进行上帝类的检测. 对比实验结果表明: 我们目前提出的上帝类检测算法总体优于现有的基于深度学习的检测算法, 在精确率和 $F1$ 值上分别提高了 25.82 个百分点和 33.39 个百分点.

本文提出的上帝类的检测方法虽然相比于现有方法提升较大, 但是在精确率上仍有提升空间. 除了基于图的结构信息和度量值等信息以外, 代码文本中也蕴含了上帝类的特征信息. 在后面的工作中, 我们将提取代码中的文本信息, 包括用户自定义的标识符和注释中的文本, 结合自然语言处理领域的相关模型, 进一步提高对上帝类检测的精确率和召回率. 另一方面, 图模型不仅可以建模出上帝类的特征, 还可以扩展应用到其他类型的代码坏味, 例如特征依恋和霰弹式修改等. 在未来的工作中, 我们将研究用何种类型的图特征对其他类型的代码坏味进行建模, 使本文提出的基于图模型和孤立森林的检测方法可以扩展应用到更多类型的代码坏味上.

References:

[1] Fowler M, Beck K. Refactoring: Improving the Design of Existing Code. 2nd ed., Addison-Wesley Professional, 2018.

[2] Nucci D, Palomba F, Tamburri D, Serebrenik A, Lucia A. Detecting code smells using machine learning techniques: Are we there yet? In: Proc. of the 25th Int'l Conf. on Software Analysis, Evolution and Reengineering. 2018. 612−621.

[3] Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M. A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software, 2004, 72(2): 129−143.

[4] Xu WB, Hua QB, Fei N. Object-oriented software refactoring. Computer Engineering, 2005, 31(5): 82−84 (in Chinese with English abstract).

[5] Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. IEEE Trans. on Software Engineering, 2009, 35(3): 347−367.

[6] Vaucher S, Khomh F, Moha N, *et al*. Tracking design smells: Lessons from a study of god classes. In: Proc. of the 16th Working Conf. on Reverse Engineering. 2009. 145−154.

[7] Azadi U, Fontana FA, Taibi D. Architectural smells detected by tools: A catalogue proposal. In: Proc. of the 2nd Int'l Conf. on Technical Debt. 2019. 88−97.

[8] Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of the 20th Int'l Conf. on Software Maintenance. 2004. 350−359.

[9] Munro M. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In: Proc. of the 11th IEEE Int'l Software Metrics Symp. 2005. Article No.15.

[10] Moha N, Gueheneuc Y, Duchien L, Le Meur AL. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2010, 36(1): 20−36.

[11] Palomba F, Bavota G, Penta MD, *et al*. Mining version histories for detecting code smells. IEEE Trans. on Software Engineering, 2015, 41(5): 462−489.

[12] Bu YF, Liu H, Li GJ. God class detection approach based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1359−1374 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5724.htm [doi: 10.13328/j.cnki.jos.005724]

[13] Page L, Brin S, Motwani R, Winograd T. The Page Rank Citation Ranking: Bringing Order to the Web. Stanford Digital Libraries Working Paper, 1998.

[14] Smith N, Bruggen DV, Tomassetti F. JavaParser: Visited analyse, transform and generate your Java code base. 2017. https://enterprise.leanpub.com/javaparservisited

[15] Liu FT, Ting KM, Zhou ZH. Isolation forest. In: Proc. of the 8th Int'l Conf. on Data Mining. 2008. 413−422.

[16] Liu FT, Ting KM, Zhou ZH. Isolation-based anomaly detection. ACM Trans. on Knowledge Discovery from Data, 2012, 6(1): 1−39.

[17] Blondel V, Guillaume J, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment. 2008, 2008(10): Article No.10008.

[18] Abbes M, Khomh F, Gueheneuc YG, *et al*. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proc. of the European Conf. on Software Maintenance & Reengineering. 2011. 181−190.

[19] Yamashita A, Moonen L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proc. of the 35th Int'l Conf. on Software Engineering. 2013. 682−691.

[20] Zhang M, Hall T, Baddoo N. Code bad smells: A review of current knowledge. Journal of Software Maintenance & Evolution Research & Practice, 2011, 23(3): 179−202.

[21] Tufano M, Palomba F, Bavota G, *et al*. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. on Software Engineering, 2017, 43(11): 1063−1088.

[22] Tufano M, Palomba F, Bavota G, *et al*. An empirical investigation into the nature of test smells. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. 2016. 4−15.

[23] Zhang XF, Zhu C. Empirical study of code smell impact on software evolution. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1422−1437 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5735.htm [doi: 10.13328/j.cnki.jos.005735]

[24] Arcoverde R, Garcia A, Figueiredo E. Understanding the longevity of code smells: Preliminary results of an explanatory survey. In: Proc. of the 4th Workshop on Refactoring Tools. 2011. 33−36.

[25] Chatzigeorgiou A, Manakos A. Investigating the evolution of bad smells in object-oriented code. In: Proc. of the 7th Int'l Conf. on the Quality of Information and Communications Technology. 2010. 106−115.

[26] Peters R, Zaidman A. Evaluating the lifespan of code smells using software repository mining. In: Proc. of the 16th European Conf. on Software Maintenance and Reengineering. 2012. 411−416.

[27] Fokaefs M, Tsantalis N, Chatzigeorgiou A. JDeodorant: Identification and removal of feature envy bad smells. In: Proc. of the 23rd IEEE Int'l Conf. on Software Maintenance. 2007. 519−520.

[28] Fontana FA, Mäntylä MV, Zanoni M, *et al*. Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 2016, 21(3): 1143−1191.

[29] Maiga A, Ali N, Bhattacharya N, *et al*. SMURF: A SVM-based incremental anti-pattern detection approach. In: Proc. of the 19th Working Conf. on Reverse Engineering. 2012. 466−475.

[30] Maiga A, Ali N, Bhattacharya N, Sababe A, Guhneuc Y, Antoniol E, Ameur E. Support vector machines for anti-pattern detection. In: Proc. of the Int'l Conf. on Automated Software Engineering. 2012. 278−281.

[31] Khomh F, Vaucher S, Gueheneuc Y, Sahraoui H. A Bayesian approach for the detection of code and design smells. In: Proc. of the 9th Int'l Conf. on Quality Software. 2009. 305−314.

[32] Khomh F, Vaucher S, Gueheneuc Y, *et al*. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. Journal of Systems & Software, 2011, 84(4): 559−572.

[33] Palomba F, Panichella A, Lucia AD, *et al*. A textual-based technique for Smell Detection. In: Proc. of the 24th IEEE Int'l Conf. on Program Comprehension. 2016. 1−10.

[34] Ma S, Dong D. Detection of large class based on latent semantic analysis. Computer Science, 2017, 44(S1): 495−498 (in Chinese with English abstract).

[35] Wang SY, Zhang YQ, Sun JZ. Detection of bad smell in code based on BP neural network. Computer Engineering, 2020, 46(10): 216−222, 230 (in Chinese with English abstract).

[36] Zhang HR, Wu YJ, Zhao WY. Software metrics set for code design quality monitoring. Computer Applications and Software, 2020, 37(3): 13−21, 66 (in Chinese with English abstract).

[37] Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. Empirical Software Engineering, 2018, 23(3): 1188−1221.

附中文参考文献:

[4] 许文波, 华奇兵, 费娜. 面向对象的软件重构. 计算机工程, 2005, 31(5): 82−84.

[12] 卜依凡, 刘辉, 李光杰. 一种基于深度学习的上帝类检测方法. 软件学报, 2019, 30(5): 1359−1374. http://www.jos.org.cn/1000-9825/5724.htm [doi: 10.13328/j.cnki.jos.005724]

[23] 章晓芳, 朱灿. 代码坏味对软件演化影响的实证研究. 软件学报, 2019, 30(5): 1422−1437. http://www.jos.org.cn/1000-9825/5735.htm [doi: 10.13328/j.cnki.jos.005735]

[34] 马赛, 董东. 基于潜在语义分析的 Large Class 检测. 计算机科学, 2017, 44(S1): 495−498.

[35] 王曙燕, 张一权, 孙家泽. 基于 BP 神经网络的代码坏味检测. 计算机工程, 2020, 46(10): 216−222, 230.

[36] 张海锐, 吴毅坚, 赵文耘. 面向代码设计质量监控的软件度量指标集研究. 计算机应用与软件, 2020, 37(3): 13−21, 66.

刘弋(1997－), 男, 硕士生, 主要研究领域为智能化软件开发.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 移动计算, 云计算.

吴毅坚(1979－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件体系结构, 软件复用和产品线, 工业环境中的软件演化.

闫亚东(1998－), 男, 硕士生, 主要研究领域为智能化软件开发.