计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210100077

基于演化和语义特征的上帝类检测方法

王继文　吴毅坚　彭　鑫
复旦大学软件学院　上海 200438
上海市数据科学重点实验室　上海 200438
(18212010032@fudan.edu.cn)

摘　要　随着软件开发迭代速度的加快,开发人员在编码过程中往往由于交付压力等种种原因违反软件设计的基本原则,造成代码坏味,进而影响软件质量。上帝类是最常见的代码坏味之一,指承担了太多职责的类。上帝类违反“高内聚、低耦合”的设计原则,损害软件系统的质量,会影响代码的可理解性和可维护性。因此,文中提出一种新的上帝类检测方法。首先抽取实际项目中方法在演化、语义等维度上的特征;然后对演化、语义特征进行融合,并基于融合后的结果重新聚类,将彼此关系紧密的方法重新划归到新的类簇中;通过分析实际项目中各个类的成员方法在新的聚类结果中的分布情况,计算类的内聚度,从而找出内聚度低的类作为上帝类检测结果。实验表明,所提方法优于目前主流的上帝类检测方法。与基于度量的传统检测方法相比,该方法在查全率、查准率上均提升超过20个百分点;与基于机器学习的检测方法相比,该方法尽管查全率略低,但查准率、*F*1值均有显著提升。

关键词:上帝类;代码坏味;软件演化;内聚度
中图法分类号　TP311.5

Approach of God Class Detection Based on Evolutionary and Semantic Features

WANG Ji-wen,WU Yi-jian and PENG Xin
Software School,Fudan University,Shanghai 200438,China
Shanghai Key Laboratory of Data Science,Shanghai 200438,China

Abstract　With the acceleration of software development iterations,developers often violate the basic principles of software design due to various reasons such as delivery pressure,resulting in code smells and affecting software quality. God class is one of the most common code smells,referring to classes that have taken on too many responsibilities. God class violates the design principle of “high cohesion and low coupling”,damages the quality of the software system,and affects the understandability and maintainability of the code. Therefore,a new method of god class detection is proposed. It extracts the evolutionary and semantic features of the actual project,then merges the evolution and semantic features. Based on the merged features,it re-clusters all the methods for the projects. By analyzing the distribution of the member methods of each class in the actual project in the new clustering result,it calculates the cohesion of the class,and finds the class with low cohesion as the God class detection result. Experiments show that this method is superior to the current mainstream God class detection methods. Compared with traditional measurement-based detection methods,the recall and precision rates of the proposed method are increased by more than 20%. Compared with detection methods based on machine learning,although the recall rate of the proposed method is slightly lower,but the precision rate and F1 value are significantly improved.

Keywords　God class,Code smell,Software evolution,Cohesion

1　引言

Fowler[1]最早提出了代码坏味(codesmell)这一概念,并列出22种常见的代码坏味。代码坏味指程序系统中某些违反基本设计原理并对设计质量产生负面影响的结构。近年来,研究人员致力于探究代码坏味的识别及相关领域。

研究表明,代码坏味会阻碍软件的开发效率,进而影响到软件系统的演化与升级[2];代码坏味还会损害软件系统的质

到稿日期:2021-01-10　返修日期:2021-03-21
基金项目:国家重点研发计划(2017YFB1002000);上海市科技发展基金项目(18DZ1112100,18DZ1112102)
This work was supported by the National Key R&D Program of China(2017YFB1002000) and Shanghai Science and Technology Development Funds(18DZ1112100,18DZ1112102).
通信作者:吴毅坚(wuyijian@fudan.edu.cn)

量，并增加软件重构与维护的成本[3]。因此，为了减少代码坏味对软件系统质量的影响和提升代码的质量，准确地识别与定位代码坏味至关重要。

上帝类(God class)是代码坏味中的一种，经常出现在实际项目中。上帝类是一个对象，且过多控制了系统中的其余对象，即扮演了过多的本该由其余类扮演的角色[4]。上帝类往往承担了大部分工作，而仅将一些次要、琐碎的工作留给其余的类。由此可见，上帝类违背了单一职责原则和分而治之的基本思想。如果任由其发展，将会给软件的可维护性和可理解性带来挑战[5]。在软件维护和演化过程中，随着项目不断迭代，上帝类成为那些没有被放到合适类的方法的聚集地，虽然看似功能很强，但其成员方法间的联系并不紧密，且各成员方法与其他类存在高耦合。这也提示我们，上帝类的检测可以从功能语义和演化历史的角度来考虑。

目前对于上帝类的检测方法主要分为两大类：基于度量的方法和基于机器学习(包括采用传统机器学习算法和较新的深度学习算法进行代码样本训练)的方法。基于度量的方法依赖于对源代码结构的度量，忽略了语义、演化等维度的特征；同时，此类方法需要对相关度量设定阈值，而阈值通常是根据开发者的经验得到的，这就导致不同的检测者对不同软件的检测结果差异显著，缺乏统一标准，也造成此类方法的查全率、查准率都偏低。而基于机器学习的方法，由于训练样本数量往往较少，训练所得的模型难以准确表达代码的复杂特性，从而导致查准率较低，难以实际使用。

针对上述上帝类检测方法的不足，本文提出一种采用演化和语义特征的上帝类检测方法。首先，我们分别抽取源代码中的方法间演化特征和语义特征。演化特征方面，主要通过代码的历史版本信息和代码结构分析，获取方法间的共同更改(co-change)信息，进而为每种方法建立一种演化特征向量，用来表示该方法和其他方法在演化过程中的共同更改情况。如果两种方法对应的演化特征向量在空间上的距离相近，就意味着这两种方法在演化过程中的联系较为密切。语义特征方面，将每种方法看成一段文本，经过预处理(详见3.3节)后利用 Doc2vec[6] 模型将其转换为语义特征向量，进而通过向量空间来衡量方法间的语义相似度。由于演化特征向量是一个维度较高且较为稀疏的向量，因此，为了更好地融合演化、语义的特征向量，本文采用 AutoEncoder[7] 对演化特征向量进行降维，使得演化、语义特征向量的维度统一。融合之后，每种方法对应一个包含演化和语义特征的融合向量。基于该融合向量，对所有的方法进行重新聚类，这里我们采用的是 K 均值聚类(K-Means)[8]。通过聚类可使彼此关系紧密的方法被重新划归到同一类簇(cluster)中。通过分析实际项目中各个类的成员方法的在新的聚类中的分布情况，分析类的内聚度[9]并进行排序，可以得到上帝类集合。

本文通过在 Landfill 数据集[10]上进行实验，验证了本文方法在检测上帝类方面的可行性与优势。与基于度量的检测方法相比，本文方法的查准率和查全率都提升了 20 个百分点以上；与效果较好的基于机器学习的方法相比，本文方法尽管在查全率上略有不足，但是查准率和 $F1$ 值都有提升显著。另外，我们通过对比实验验证了引入语义特征、演化特征的价值。

2 相关工作

2.1 上帝类及其检测技术

Lanza 等[4]首次提出了上帝类的概念。上帝类违反了单一职责原则，通常会表现出类内成员间内聚度较低、复杂度过高等特征[4]。作为一种典型的代码坏味，一方面，上帝类会影响代码的可读性和可理解性；另一方面，上帝类的存在会增大其他代码坏味出现的概率。因此，检测上帝类逐渐成为软件设计领域一个重要的方向。

近年来，上帝类检测方法主要可以分为基于度量的方法和基于机器学习的方法。

在基于度量的方法方面，Tsantalis 等[11]利用两个代码实体(例如方法)间共享的属性个数定义代码实体间的距离，进而综合考虑类内和类间代码实体距离的远近来推测是否存在上帝类。Reddy 等[12]基于度量依赖程度来检测上帝类。具体地，定义类之间的耦合度为类之间的依赖关系，对系统中的每一个类，计算其与剩下的类之间的耦合度，并取平均值，以此来判断该类是否为上帝类。基于度量的方法大多存在以下问题：

(1)大多数方法仅仅考虑代码间的结构信息[13]，如方法的调用和属性的使用，而隐藏在代码中的语义特征、演化特征往往被忽略。

(2)该类方法的查全率、查准率都偏低，难以实际运用。

基于机器学习的方法方面，Khomh 等[14]利用贝叶斯信念网络(Bayesian Belief Network)实现了对上帝类的检测；Fontana 等[15]基于其余检测工具，通过训练得到上帝类的检测规则。除此之外，Bu 等[16]通过挖掘文本语义，提出了一种基于深度神经网络的上帝类检测方法。此类方法往往依赖于训练集的选择，由于存在坏味的代码占系统总代码的比重比较小，因此训练集往往过小，从而导致训练模型的效果难以提升，结果偏差较大，随机性较高，查准率也偏低。

2.2 软件演化研究

软件演化特征体现了软件长期变化的情况。其不仅包含当前版本的信息，还包含历史变化信息，因此，对全面描述软件特性有着重要的作用。Zhang 等[17]通过探究含有坏味的文件在各历史版本中的变化情况，证明了代码坏味会对文件的修改产生影响。这说明通过演化特征分析检测代码坏味具有可行性。

针对刻画软件演化的过程，学术界提出了多种方法。Wu 等[18-19]使用光谱来研究软件的演化。Gall 等[20]提出了基于时间和属性值的表示方法，来检测软件的历史信息。Lanza[21]采用演化矩阵来表现软件系统中的各个实体的演化情况。Gîrba 等[22]基于历史快照生成元模型，用以表示软件演化过程。Robbes 等[23]通过对软件中信息的变化进行建模，从而监控软件的运行。Kouroshfar[24]采用共同更改矩阵研究了软件变更历史在衡量软件质量和预测缺陷中起到的重要作用，也表明了代码实体的共同更改矩阵在表示代码演化特征

方面的效果。鉴于此，本文采用方法级别的共同更改矩阵来表示类代码的演化特征。

方法级别的共同更改矩阵的每一行都代表一种方法，该行中的各个值代表该方法与其他方法的共同更改次数，即矩阵中的元素 $\boldsymbol{A}[i][j]$ 表示第 i 种方法和第 j 种方法间的共同更改次数。

为了得到方法间的共同更改矩阵，必须准确定位每次版本修改的位置，即修改代码所在的方法。AST（抽象语法树）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构。在 AST 的基础上，通过匹配不同版本的差异，可以得到代码修改的细节，准确定位修改代码所属的代码单元（例如方法）。本文采用 AST 进行分析与匹配，从而获得类和方法中代码添加、删除、修改的情况，进而构建方法级共同更改矩阵。

2.3 代码的语义特征提取

为了提取代码的语义特征，需要挖掘代码标识符之间的深层语义关联。在自然语言处理领域中，有许多模型可用于提取文本的语义特征。

LDA（Latent Dirichlet Allocation）算法[25]建立了一个包含词、主题和文档的 3 层贝叶斯概率模型结构。但是 LDA 没有考虑词间的顺序。TF-IDF[26]通过一系列代表性词汇的出现频率对文档进行编码，这些词汇都是从大型语料库中选取的。虽然 TF-IDF 给每个词汇都定义了不同的维度，但是忽略了词汇间的关系。除此之外，TF-IDF 为了保证效果，需要尽可能大的词汇表，这会带来维度灾难问题，进而给计算带来巨大的负担。

因此，本文选取了基于 Word2vec[27-28]的 Doc2vec[6]来提取语义特征，即将方法的代码文本转换为一个固定长度的特征向量（200 维）。Word2vec 由 Mikolov 等提出[27-28]，它是在神经网络语言模型（Neural Network Language Model，NNLM）的基础上建立的。Word2vec 用 Huffman 树作为最后一层输出层，通过局部上下文来学习有意义的词向量。词在转化为向量时，意思相近的词会被映射到空间中相近的位置。Word2vec 根据这一特性，使用夹角余弦[29]来反映词语间的关联程度。

Mikolov 等[6]在 Word2vec 的基础上提出了 Doc2vec，旨在从变长文本中学习得到固定长度的特征表示。Doc2vec 模型基于 Word2vec，仅在输入中添加了段落向量。其中，语义相似度也是由词向量间的夹角余弦值来表示。由于 Doc2vec 能够学习得到整段文本的特征表示向量，因此，我们将代码看成文本，在去除特殊词汇后，利用 Doc2vec 获得带有语义信息的代码特征向量。

3 基于演化和语义特征的上帝类检测方法

为了自动检测上帝类，本文提出一种基于演化和语义特征的检测方法。

3.1 方法概览

图 1 给出了基于演化和语义特征的上帝类检测方法的主要步骤。首先，从被测项目的代码库中检出最新版本代码；其次，提取各个类的成员方法的代码语义特征，同时从被测项目的代码库的历史版本中获取代码变更信息并提取代码演化特征；随后，融合两种特征，得到类中的各成员方法在演化和语义两方面的融合特征，并基于此融合特征进行聚类；最后，根据类中方法在新的聚类中的分布情况度量类的内聚性并排序，基于排序结果得到上帝类。

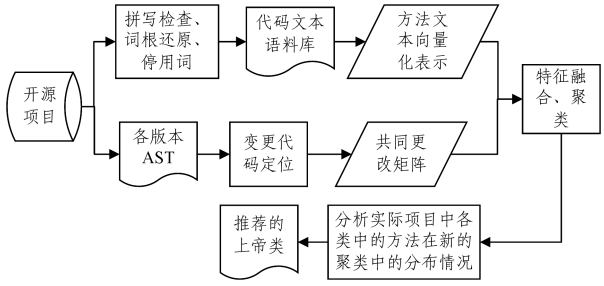

图 1 检测步骤概览

Fig. 1 Overview of the proposed approach

为了得到演化特征，我们利用版本管理工具解析出各个项目所有的版本，通过提取与匹配 AST，定位每次版本修改的代码片段，记录演化历史，并将其存入关系型数据库中。基于提取的版本演化历史，得到共同更改矩阵。在共同更改矩阵中，每种方法有一个与之对应的演化特征向量，如果两种方法对应的演化特征向量在空间上的距离相近，那么就意味着这两种方法在演化过程中的联系较为密切。

语义特征方面，针对最终版本中的每种方法，抽取方法体和方法签名，通过拼写检查、词根还原、停用词等步骤，将每种方法的代码文本抽取成对应的向量，并将其输入到 Doc2vec 模型中，得到文本对应的语义特征向量。

由于演化特征向量是一个维度高、较为稀疏的向量，为了更好地融合演化特征和语义特征，本文通过 AutoEncoder 对演化特征向量进行降维，使得演化特征向量和语义特征向量具有相同的维度。然后基于得到的融合向量，对实际项目中所有方法重新进行聚类，通过分析聚类结果，计算得到实际项目中各类的内聚度和离散程度并排序，推荐出上帝类集合。

3.2 基于 AST 匹配的演化特征

软件在生命周期中一直在发生演化，对软件演化的分析可以揭示软件发展的一些基本规律。现有方法通常使用改变的代码行数作为软件演化的指标，但是这项指标不能反映软件架构的变化，也无助于理解软件演化的具体情况。因此，本文选取共同更改矩来表征软件的演化特征，共同更改矩阵的具体获取过程如图 2 所示。

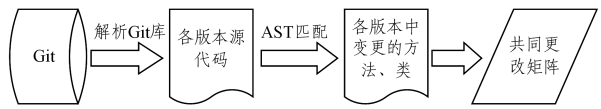

图 2 构建共同更改矩阵的过程

Fig. 2 Process of building co-change matrices

如图 3 所示，为了采集这些演化关系特征，本文首先从软件项目的 Git 版本控制系统中采集该项目的代码提交信息（得到该目标软件项目的 Git 库）；其次，通过开源工具 Jgit 解析 Git 库，将每次提交的信息存入关系型数据库中，为后续工作奠定基础。

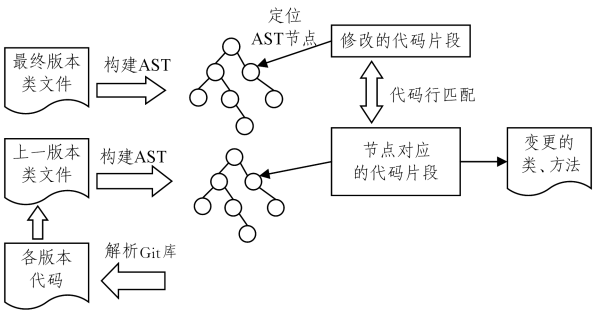

图 3 通过 AST 匹配修改的代码片段

Fig. 3 Match modified code snippets through AST

项目的每次版本变更涉及相邻的两个版本，根据时间顺序，分别记为较前的版本与较后的版本。我们需要得到此次变更涉及的方法的集合。具体算法如算法 1 所示。

算法 1 基于 AST 匹配得到修改方法集合算法

输入：此次版本变更涉及的代码片段集合 $X=\{X_1, X_2, \cdots\}$

输出：此次版本变更中共同出现的方法集合 $M=\{M_1, M_2, \cdots\}$

对于 X 中的每个代码片段 X_i，进行如下步骤：

{

步骤 1 通过字符串匹配的方式在较后的版本中定位该代码片段，找到其所属类 C，并根据行号范围确定其涉及哪些方法；

步骤 2 通过 JavaParser 工具对 C 的前后两个版本的代码构建 AST，分别记为 T_1 和 T_2；

步骤 3 在 T_2 中，通过方法签名匹配，将 X_i 定位到具体的 AST 方法节点集合 $N=\{n_1, n_2, \cdots\}$；

步骤 4 使用开源工具 GumTree 对 T_1 和 T_2 的节点进行匹配，将 N 中各节点对应到 T_1 中，得到方法节点集合 $N'=\{n_1', n_2', \cdots\}$，并将其加到 M 中；

}

对于最终得到的集合 M，我们称 M 内的方法具有演化依赖关系，并以此构建共同更改矩阵。如果两种方法共同更改次数更多，那么在共同更改矩阵中所对应的值就更大，即这两种方法更倾向于同时修改。

共同更改矩阵表示方法间彼此的演化耦合程度。其中，每一行都代表其对应的方法的演化特征向量（即与其他哪些方法存在何种强度的演化耦合）。如果两种方法对应的演化特征向量在空间上的距离相近，那么就意味着这两种方法在演化过程中更相似。但我们并不直接计算每种方法的演化特征向量的距离，而是将其与语义特征向量进行融合（见 3.4 节）后再综合考虑方法间的相似程度。

3.3 基于 Doc2vec 的语义特征

由于上帝类承担了过多的职责，因此其所含方法的功能差别较大，在语义方面会表现出类内方法间文本相似度内聚度较低。因此，我们可以通过提取方法的语义特征，来分析类的内聚度。

本文使用 Doc2vec 模型将代码文本转换为一个固定长度的词嵌入向量（200 维）。为了提高代码文本中语义信息提取的准确度，我们进行如下步骤：

（1）分词：基于下划线命名法、驼峰命名法等命名规则，对代码文本中的标识符进行拆分；

（2）词根还原：去掉单词的词缀，提取单词的主干部分。

至此，我们已提取每种方法的语义特征信息，后续将其与方法的演化特征信息进行融合处理。

3.4 演化特征与语义特征的融合与聚类

3.4.1 特征融合

由于共同更改矩阵是一个 $n\times n$ 的矩阵（n 为实际项目中方法的数量），而实际项目所包含的方法数往往非常多，且大多数方法不存在或仅存在少量的共同变更情况，因此通过共同变更矩阵得到的演化特征向量是一个维度较大（n 维，n 为实际项目中方法的数量）、较为稀疏的向量。由于语义特征向量维度仅 200 维，如果不加以处理直接融合，则语义特征的作用势必会受到影响。基于上述原因，为了更好地融合演化特征和语义特征，充分发挥各特征在方法中的作用，我们需要先对演化特征向量进行降维，使其维度和语义特征相同，进而对演化特征与语义特征进行有效融合。

本文通过自编码器（AutoEncoder）[7] 来对演化特征向量进行降维。AutoEncoder 是一种无监督的多层人工神经网络，可以对数据进行编码，从而达到特征提取、特征表示学习的目的，已被广泛应用于数据降维。AutoEncoder 将输入信息进行压缩，提取出数据中最具代表性的信息。其目的是在保证重要特征不丢失的情况下，降低输入信息的维度，减小神经网络的处理负担。因此，AutoEncoder 适用于将演化特征向量降维。

AutoEncoder 主要包含两大模块：编码器（encoder）和解码器（decoder）。其中，编码器的主要作用是降维，将高维的原始数据降到具有一定维数的低维嵌套结构上。然后通过解码器将低维嵌套上的点还原成高维数据。由于编码后的特征向量 $\boldsymbol{h}$ 的维度远小于输入层，因此，如果将编码后的中间表示的低维嵌套结构抽取出来，就可以得到样本的低维抽象表达，从而达到降维的效果。

如图 4 所示，本文利用 AutoEncoder 将维度为 n 的高维演化特征向量 $\boldsymbol{X}$ 作为输入传进网络。AutoEncoder 对输入的演化特征向量按照一定规则及训练算法进行编码，将其原始特征用隐藏层（hidden layer）中维度为 m 的低维向量 $\boldsymbol{h}$ 重新表示。其中，m 为语义特征向量的维度。这样，将中间表示 $\boldsymbol{h}$ 提取出后，演化和语义特征向量的维度得到统一。

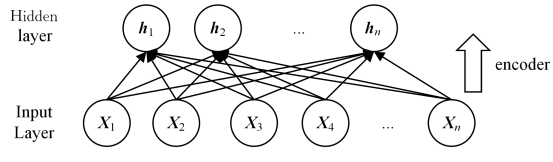

图 4 AutoEncoder 的 Encoder 结构

Fig. 4 Structure of Encoder in AutoEncoder

为了能够更加有效地发挥演化特征和语义特征的作用，我们还对这两个特征向量进行归一化处理。

基于上述的步骤，我们将演化特征向量和语义特征向量相加，得到包含演化、语义特征的融合特征向量。

3.4.2 K-Means 聚类与内聚度分析

演化特征与语义特征融合后，可以得到包含 n 个元素的融合特征向量集合 $\boldsymbol{A}$，表示为 $\boldsymbol{A}=\{\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3, \cdots, \boldsymbol{A}_n\}$。其中，$n$ 为被分析项目中的方法数。$\boldsymbol{A}_i(1\leqslant i\leqslant n)$ 表示第 i 种方法的

融合特征向量,且 $\boldsymbol{A}$ 中每种元素都具有 m 个维度(m 为 200)的属性。

为了更好地描述实际项目中各类的内聚度,我们需要把实际项目中各类中的方法重新打散,根据各自对应的融合特征向量重新聚类。即在演化特征和语义特征的直观影响下,对集合 $\boldsymbol{A}$ 中的元素进行重新聚类,将集合中的每个元素划分到与之空间距离最相近的类簇中。本文中采用 K-Means 聚类[8]。K-Means 聚类算法是一种常见的聚类算法,其主要思想是将集合中相似的元素自动归到一起,从而得到 k 个类簇。

本文利用 K-Means 算法的目标就是将 $\boldsymbol{A}$ 中 n 种方法的融合特征向量依据相似性聚集到指定的 k 个类簇中,每个元素属于且仅属于一个到类簇中心距离最小的类簇。

我们首先随机初始化 k 个聚类中心$\{C_1,C_2,C_3,\cdots,C_k\}$,为了保证聚类后的方法类簇分布尽可能符合实际项目情况,设定 k 为实际项目中类的个数。

然后通过计算每一种方法对应的融合特征向量到每一个聚类中心的欧氏距离:

$$dis(\boldsymbol{A}_i,\boldsymbol{C}_j)=\sqrt{(\sum_{t=1}^{m}\boldsymbol{A}_{it}-\boldsymbol{C}_{jt})^2} \tag{1}$$

其中,$\boldsymbol{A}_i$ 表示融合特征向量的集合 $\boldsymbol{A}$ 中第 i 个元素(即方法对应的向量),$1\leqslant i\leqslant n$;$\boldsymbol{A}_{it}$ 表示第 i 个元素的第 t 个属性($1\leqslant t\leqslant m$);$\boldsymbol{C}_{jt}$ 表示第 j 个聚类中心的第 t 个属性。

依次比较每一个元素到每一个聚类中心的距离,将元素分配到距离最近的聚类中心的类簇中,得到 k 个类簇$\{S_1,S_2,S_3,\cdots,S_k\}$。

之后,对每个类簇 S_l 重新计算其类簇中心 C_l,计算式如式(2)所示:

$$C_l=\frac{\sum\limits_{X_i\in S_l}X_i}{|S_l|} \tag{2}$$

其中,C_l 表示第 l 个类簇的中心,$1\leqslant l\leqslant k$;$|S_l|$ 表示第 l 个类簇中元素的个数;X_i 表示第 l 个类簇中的第 i 个元素,$1\leqslant i\leqslant |S_l|$。

重复上述过程直至最终收敛,从而可以将样本中的每个元素归到其所对应的类簇中。由于每个元素代表其对应的方法的特征向量,因此可得到对实际项目中所有方法重新聚类后的情况,每种方法都属于一个新的类簇。

3.5 确定候选的上帝类

由于上帝类通常表现出类内成员间内聚度较低、复杂度过高等特征[4],因此本文选取内聚度这一指标作为检测上帝类的评价指标。对于实际项目中的每个类,如果某个类中各种方法在经过重新聚类后分散在不同的类簇中,那么该类的内聚度较低。

本文通过离散程度(degree of dispersion)来衡量类内成员方法间的内聚度,进而识别上帝类。具体地,在经过特征融合与聚类之后,实际项目中的每种方法都被重新划归到了新的类簇中。对于实际项目中的每个类来说,通过分析其成员方法在新的聚类结果中的分布情况,可以反映出其类内方法间的内聚度。

对于实际项目中的每个类 C,x_i 为第 i 个成员方法聚类后对应的新的类簇编号。我们将 x_i 在类内按递增顺序重新进行排序,并将其依次映射到从 1 开始的自然数上(相同的类簇编号映射到相同的自然数上),得到了 x_i'。x_i' 屏蔽了类簇编号不同带来的影响,这样类簇编号的大小偏离即可准确反映出类内各方法在融合特征向量方面的离散程度(即类中的方法在新的类簇中越分散,则类簇编号差异就越大)。内聚度计算公式如式(3)所示:

$$\sigma=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i'-\mu)^2} \tag{3}$$

其中,N 为方法的数量;μ 为 x_i' 的平均值,即:

$$\mu=\frac{1}{N}\sum_{i=1}^{N}x_i' \tag{4}$$

这样,一个类的 σ 值越高,代表其内部方法成员间的离散程度越大、内聚度越低,则该类是上帝类的概率也就越大。

我们对实际项目中所有的类计算出其对应的 σ 值并进行排序。有研究表明,现实软件项目中出现上帝类的概率约为 1.7%[30]。为实现查全率与查准率的平衡,我们将上帝类判定比例设定为 2.7%(比经验概率高 1 个百分点),即根据 σ 值降序排列的前 2.7%的类判定为上帝类。

4 实验

本文基于开放的代码坏味数据集[10],对所提上帝类检测方法进行验证和分析。同时,我们收集了 6 个高质量开源且可编译通过的 Java 项目作为测试项目。

4.1 研究问题

在实验阶段,我们希望通过分析以下 3 个问题来对所提出的方法进行验证与分析.

研究问题 1:该方法能否准确、高效地检测出上帝类?其查全率、查准率如何?与主流检测方法的对比情况怎样?

研究问题 2:在本文方法中,演化特征对上帝类的检测结果有怎样的贡献?即,考虑演化特征与不考虑演化特征相比,检测结果的查全率、查准率等指标有什么变化?

研究问题 3:在本文方法中,语义特征对上帝类的检测结果有怎样的贡献?即,考虑语义特征与不考虑语义特征相比,检测结果的查全率、查准率等指标有什么变化?

研究问题 1 主要关注本文提出的方法是否在查准率等指标上优于主流方法。为了回答这个问题,我们从基于度量和基于机器学习的检测方法中各选取一种典型方法进行对比实验。在基于度量的检测方法中,我们选择了 JDeodorant[31]作为对比实验工具。该工具作为业内知名的代码重构工具,其对上帝类等代码坏味的检测能力得到了公认。基于机器学习的检测方法中,我们选择了一种较新的采用深度学习的方法[28],该方法考虑了语义信息,具有对比价值。

研究问题 2 和研究问题 3 主要关注的是所引入演化特征和语义特征的有效性,即分别剪除演化特征或语义特征,仅采用另一个特征来检测上帝类。我们通过将最优平均 $F1$ 值作为衡量指标来分析所提出的两个特征在整个方法中的作用。

4.2 数据集准备

Landfill[10]是一个在线代码坏味数据集平台,为对比代码坏味检测技术提供了基准。它从 30 多个开源的大型项目中标记了多种不同的代码坏味。为了保证结果的精确,本文在

Landfill 数据集的基础上进一步加以验证，过滤了重复的、不存在的实例。

本文选取了 6 个经典且包含上帝类的高质量开源 Java 项目：Struts，Ivy，Log4j，Lucene，Derby，Ant。这 6 个项目所在领域以及在 Landfill 中最近版本的类数、方法数、代码行数(LOC)和演化时间范围如表 1 所列。所选项目都具有一定规模，且演化时间范围为 1～15 年不等。

表 1 数据集所涉项目

Table 1 Projects involved in the dataset

项目名	项目领域	类数	方法数	LOC	演化时间范围
Struts	Web 程序框架	2106	14150	303250	2006-02－2015-10
Ivy	项目跟踪管理	327	3017	42518	2005-06－2006-10
Log4j	日志系统	519	4671	73693	2000-11－2007-02
Lucene	搜索引擎	1428	14607	288337	2001-09－2010-03
Derby	数据库	2422	28756	939599	2004-08－2008-06
Ant	源码编译	1218	12110	266897	2000-01－2015-06

4.3 实验结果的评价标准

为了更好地评估实验结果，实验采用查准率(precision)、查全率(recall)和 $F1$ 值这 3 个评价指标。其中，查准率考查检测的精确度，查全率考查检测的完整性。具体定义如下：

$$查准率=\frac{正确识别出的上帝类总数}{识别出的上帝类总数}$$

$$查全率=\frac{正确识别出的上帝类总数}{Landfill\ 中标记的上帝类总数}$$

$$F1=2\times\frac{precision\times recall}{precision+recall}$$

4.4 实验结果与分析

对于研究问题 1，首先，将本文方法和主流的基于度量的上帝类检测方法 JDeodorant 进行了对比，结果如表 2 所列。可以看出：

(1)在对上帝类的检测能力上，本文方法总体表现优于 JDeodorant，查全率提升了 26.54 个百分点(50.50%～77.04%)。

(2)本文方法在查准率上的提升也尤为明显，提升了 22.74 个百分点(3.88%～26.79%)，这表明本文方法可以在推荐更少上帝类的同时保证查准率，这无疑可以更便捷地定位上帝类。

表 2 本文方法与 JDeodorant 的效果对比

Table 2 Result comparison of the proposed method and JDeodorant

(单位：%)

项目名称	本文提出的方法			JDeodorant		
	查准率	查全率	$F1$ 值	查准率	查全率	$F1$ 值
Struts	**20.41**	**83.33**	**32.79**	4.23	50.00	7.80
Ivy	**45.56**	71.43	**55.63**	7.06	**85.71**	13.05
Log4j	**25.00**	**80.00**	**38.10**	3.33	25.00	5.88
Lucene	**16.67**	**66.67**	**26.61**	2.96	50.00	5.59
Derby	**12.07**	**87.50**	**21.21**	4.01	42.30	7.33
Ant	**40.00**	**73.33**	**51.76**	1.70	50.00	3.29
平均值	**26.62**	**77.04**	**37.68**	3.88	50.50	7.16

其次，我们选取基于深度学习的上帝类检测方法[16]作为比较对象。该方法也是在 Landfill 上选取一系列开源项目作为实验数据集，且比传统的基于机器学习的方法显示出更好的效果。由于本文方法目前主要适用于基于 Git 托管的项目，且 Landfill 的版本变动导致有些开源项目被删除，我们最终选取了 Struts，Ivy，Derby，Ant 这 4 个现存的项目作为对比实验数据集。

本文方法与基于深度学习的方法的平均查准率、查全率、$F1$ 值如表 3 所列。可以看到，尽管对比方法有着更好的查全率(85.58%)，但是由于其在查准率上的极大优势，本文方法在综合指标 $F1$ 值上提升了 33.45 个百分点(6.90%～40.35%)。这意味着采用本文方法可以大大减少误报，从而避免开发人员因得到的有效结果密度太低而放弃人工验证。

表 3 本文方法与深度学习方法的效果对比

Table 3 Result comparison of the proposed method and deep learning based approach

(单位：%)

项目名称	本文提出的方法			深度学习方法		
	查准率	查全率	$F1$ 值	查准率	查全率	$F1$ 值
Struts	**20.41**	**83.33**	**32.79**	0.98	50.00	1.92
Ivy	**45.56**	71.43	**55.63**	3.28	**100.00**	6.35
Derby	**12.07**	87.05	**21.21**	5.05	**92.30**	9.58
Ant	**40.00**	73.33	**51.76**	5.13	**100.00**	9.76
平均值	**29.51**	78.90	**40.35**	3.61	**85.58**	6.90

针对研究问题 2 和研究问题 3，我们设计了两组对比实验来分别考查演化特征、语义特征在本文方法中的作用。在这两组实验中，我们分别仅采用语义特征和演化特征，测试的数据集和其余步骤都不变，实验结果如表 4 所列。可以看出：

(1)仅采用语义特征或演化特征后，相比表 2，$F1$ 值均有所下降。这意味着对于定位上帝类来说，引入演化特征、语义特征确实能够起到积极作用。同时，这也代表多特征融合后的效果更优。虽然去掉某一维度后指标偶尔会更高，但是总体效果不如多特征融合后的效果。

(2)从 $F1$ 值来看，语义特征对检测结果的贡献较大(33.16%＞28.91%)。

表 4 仅使用语义特征或演化特征的对比实验

Table 4 Comparison test based on semantic features or evolutionary features only

(单位：%)

项目名称	仅语义特征			仅演化特征		
	查准率	查全率	$F1$ 值	查准率	查全率	$F1$ 值
Struts	**18.37**	**75.00**	**29.51**	12.24	50.00	19.67
Ivy	36.36	57.14	44.44	**54.55**	**85.71**	**66.67**
Log4j	15.00	60.00	24.00	15.00	60.00	24.00
Lucene	**14.58**	**58.33**	**23.33**	8.33	33.33	13.33
Derby	**12.07**	**87.50**	**21.21**	6.90	50.00	12.13
Ant	**43.64**	**80.00**	**56.47**	29.09	53.33	37.65
平均值	**23.34**	**69.66**	**33.16**	21.02	55.40	28.91

4.5 有效性威胁

首先，本文实验结果的有效性威胁在于用来验证实验结果的开源项目的可靠性。为此，我们选择的 6 个开源项目都是经过数百次乃至几千次版本变更的高质量开源 Java 项目，且上帝类都经过了人工判定，数据的准确性有保障。

其次，Landfill 数据集中人工标注上帝类总量并不多。由于已有研究发现上帝类的出现概率约在 1.7%[30]，因此我们选取的项目也是遵从这个比例，以求尽量符合真实开发情况，尽可能降低数据偏差的可能。

最后，我们仅对开源项目实验数据集中的代码进行了检

测，并未在企业级软件项目中进行验证，因此这并不能代表所有软件项目均有本文实验的效果。但我们选择的开源项目涵盖了多个领域，已经尽可能减少了这一有效性威胁。

5 讨论

5.1 特征融合的降维处理

为了更好地融合演化特征与语义特征，本文利用 AutoEncoder 对高维的演化特征向量进行降维（详见 3.4 节）。这是在降维前后重要信息不丢失的基础上进行的。事实上，在数据降维的过程中必然会导致数据信息的损失。为了衡量降维前后的数据信息损失情况，我们对降维前后的数据各自进行了聚类，聚类效果相似，基本可以忽略数据信息损失带来的影响。

但是，在以后更复杂的应用中，数据信息的损失可能会带来难以预测的后果，因此需要更好的特征融合手段来进行融合。

5.2 耗时

获取演化特征需要获取每次版本变更涉及的方法集合。而我们所选取的项目都有几千次甚至上万次的版本变更，因此，耗时较为严重是难以避免的问题。经统计，平均每次版本变更处理需要耗时 1 s 左右，具体各项目的演化信息预处理耗时如表 5 所列。

表 5 各项目演化信息预处理耗时

Table 5 Time-consuming of each project in evolutionary information preprocessing

项目名称	截止当前快照的版本数	信息预处理耗时/s
Struts	6088	4365
Ivy	576	2416
Log4j	2668	2074
Lucene	4657	4506
Derby	8269	12978
Ant	16967	12495

4.4 节的结果表明，即使去除了演化特征，本文方法在各种指标上，尤其是查准率上，依然有着很大的优势。这表明，如果需要缩减耗时，可以从语义信息出发，辅以多种度量指标。通过这种方式检测上帝类，尽管在查全率、查准率上可能不如多特征融合后的效果明显，但是可以在保证一定查全率、查准率的同时，大大缩减耗时。

5.3 缺少基于结构依赖的度量指标是否会有影响

与传统的基于度量的上帝类检测方法相比，本文方法并没有借助大量的基于结构依赖的度量指标。

传统上帝类检测方法中的度量指标的阈值往往由开发者的经验确定，因此，选择不同的阈值往往会带来不同的结果，这就对阈值的选择有很高的要求。并且，此类方法的查准率、查全率普遍表现不佳，将本文方法和该类方法中的一种进行对比，结果表明本文方法在查全率、查准率上都具有一定优势。当然，本文方法也可以叠加基于结构依赖的度量指标，前提是对这些指标设置选取合适的阈值。

结束语 本文针对上帝类这一常见的代码坏味，提出了一种基于演化和语义特征的检测方法。为了得到演化特征，需要分析在演化过程中，哪些方法有着相似的变更轨迹。本文从实际项目的版本变更历史中得到各版本的变更信息，再通过匹配，定位出每次版本变更中有哪些方法共同出现，进而得到方法间的共同更改矩阵。在语义特征方面，通过一系列的文本预处理流程与 Doc2vec 网络模型，抽取出各方法所对应的向量化表示结果。为了更好地叠加演化和语义特征，防止演化特征维度过大造成误差，本文引入 AutoEncoder 对高维的演化特征进行降维，基于降维后的结果，对演化和语义特征进行融合。最后，基于融合后的特征向量，将原项目中的各方法打散并重新聚类，对聚类后的结果加以分析，得到各类的内部方法成员在新的聚类中的分布情况，计算得到类的内聚度，最终得到上帝类集合。

为了保证实验结果的可靠性，我们利用开源项目对本文提出的方法进行验证。我们选取 6 个可靠的、高质量的大型开源项目作为数据集进行实验。为了验证所提方法的优势，本文选取两种不同类型的主流的上帝类检测方法进行对比。实验结果表明，与基于度量的检测工具 JDeodorant 相比，本文方法在查全率、查准率、$F1$ 值方面都有显著的优势。与基于机器学习的方法相比，本文方法尽管在查全率上有所下降，但查准率和 $F1$ 值都有较大提升，表明其有可能将开发人员从大量的误报中解放出来，专注于所发现的上帝类问题。

为了更好地评估演化和语义特征对本文方法的贡献与作用，本文设置了对比实验。结果表明，无论是演化特征还是语义特征，都对结果的改进有着积极的贡献。

至此，本文探究了演化与语义特征在上帝类检测中的可行性与作用。未来，我们将基于此思路，辅以更多的评价指标，从而实现对其他低内聚代码坏味如长方法（long method）等进行检测。

参 考 文 献

[1] FOWLER M. Refactoring: improving the design of existing code [M]. Addison-Wesley Longman Publishing Co. Inc., 1999.

[2] CHATZIGEORGIOU A, MANAKOS A. Investigating the evolution of Code Smells in object-oriented systems[J]. Innovations in Systems and Software Engineering, 2014, 10(1): 3-18.

[3] HAMZA H, COUNSELL S, HALL T, et al. Code Smell eradication and associated refactoring[M]. World Scientific and Engineering Academy and Society (WSEAS), 2008.

[4] LANZA M, MARINESCU R, DUCASSE S. Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems [M]. Berlin: Springer-Verlag, 2006.

[5] FOWLER M. Trans. Refactoring: Improving the Design of Existing Code (2nd ed) [M]. Beijing: Posts and Telecom Press, 2015.

[6] LE Q, MIKOLOV T. Distributed representations of sentences and documents [C] // International Conference on Machine Learning. PMLR, 2014: 1188-1196.

[7] HINTON G E, SALAKHUTDINOV R R. Reducing the dimensionality of data with neural networks[J]. Science, 2006, 313: 504-507.

[8] JAINA K. Data clustering: 50 years beyond K-means[J]. Pat-

tern Recognition Letters, 2010, 31(8): 651-666.
[9] ETZKORN L H, GHOLSTON S E, FORTUNE J L, et al. A comparison of cohesion metrics for object-oriented systems[J]. Information & Software Technology, 2004, 46(10): 677-687.
[10] PALOMBA F, NUCCI D D, TUFANO M, et al. Landfill: An Open Dataset of Code Smells with Public Evaluation[C]// Mining Software Repositories. IEEE, 2015: 482-485.
[11] TSANTALIS N, CHATZIGEORGIOU A. Identification of Extract Method Refactoring Opportunities[C]// European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2009: 119-128.
[12] REDDY K R, RAO A A. Dependency oriented complexity metrics to detect rippling related design defects[J]. ACM Sigsoft Software Engineering Notes, 2009, 34(4): 1-7.
[13] PALOMBA F, BAVOTA G, PENTA M D, et al. Detecting bad smells in source code using change history information[C]// International Conference on Automated Software Engineering. ACM, 2013: 268-278.
[14] KHOMH F, VAUCHER S, GUEHENEUC Y, et al. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns [J]. J. Syst. Softw., 2011, 84(4): 559-572.
[15] FONTANA F A, ZANONI M, MARINO A, et al. Code smell detection: Towards a machine learning-based approach[C]// 2013 IEEE International Conference on Software Maintenance. IEEE, 2013: 396-399.
[16] BU Y F, LIU H, LI G J. A God class detection method based on deep learning[J]. Journal of Software, 2019, 30(5): 161-176.
[17] ZHANG X F, ZHU C. Empirical study of code smell impact on software evolution[J]. Journal of Software, 2019, 30(5): 1422-1437.
[18] WU J, HOLT R, HASSAN A. Exploring software evolution using spectrographs[C]// Proceeding of the 11th Working Conference on Reverse Engineering. IEEE Press, 2004: 80-89.
[19] WU J, SPITZER C W, HASSAN A E, et al. Evolution spectrographs: Visualizing punctuated change in software evolution [C]// Proceeding of the 7th International Workshop on Principles of Software Evolution. ACM Press, 2004: 57-66.
[20] GALL H, JAZAYERI M, RIVA C. Visualizing software release histories: The use of color and third dimension[C]// Proceeding of the International Conference on Software Maintenance. IEEE Press, 1999: 99-108.
[21] LANZA M. The evolution matrix: Recovering software evolution using software visualization techniques[C]// Proceeding of the 1st Workshop on Principles of Software Evolution. New York: ACM Press, 2001: 37-42.
[22] GÎRBA T, DUCASSE S. Modeling history to analyze software evolution[J]. Journal of Software Maintenance and Evolution: Research and Practice, 2006, 18(3): 207-236.
[23] ROBBES R, LANZA M. A change-based approach to software evolution[J]. Electronic Notes in Theoretical Computer Science, 2007, 166: 93-109.
[24] KOUROSHFAR E. Studying the effect of co-change dispersion on software quality[C]// International Conference on Software Engineering, 2013: 1450-1452.
[25] GUO X, XIANG Y, CHEN Q, et al. LDA-based online topic detection using tensor factorization [J]. J. Inf. Sci., 2013, 39(4): 459-469.
[26] DEY A, JENAMANI M, THAKKAR J J. Lexical TF-IDF: An n-gram feature space for cross-domain classification of sentiment reviews[C]// International Conference on Pattern Recognition and Machine Intelligence. Cham: Springer, 2017: 380-386.
[27] MIKOLOV T, CHEN K, CORRADO G, et al. Efficient estimation of word representations in vector space[J]. arXiv: 1301.3781, 2013.
[28] MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed representations of words and phrases and their compositionality [C]// Advances in Neural Information Processing Systems. 2013: 3111-3119.
[29] LAHITANI A R, PERMANASARI A E, SETIAWAN N A. Cosine similarity to determine similarity measure: Study case in online essay assessment[C]// 2016 4th International Conference on Cyber and IT Service Management. IEEE, 2016: 1-6.
[30] PALOMBA F, BAVOTA G, PENTA M, et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation[J]. Empir Software Eng., 2018, 23: 1188-1221.
[31] FOKAEFS M, TSANTALIS N, STROULIA E, et al. JDeodorant: identification and application of extract class refactorings [C]// 2011 33rd International Conference on Software Engineering (ICSE). IEEE, 2011: 1037-1039.

WANG Ji-wen, born in 1997, postgraduate. His main research interests include software design analysis and software evolution analysis.

WU Yi-jian, born in 1979, Ph. D, associate professor, is a member of China Computer Federation. His main research interests include big code analysis, software evolution analysis and code clone detection and management.