软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, [doi: 10.13328/j.cnki.jos.000000]

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

# 一种基于深度学习的上帝类检测方法*

卜依凡, 刘 辉, 李光杰

(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**: 上帝类是指某个其承担本应由多个类分别承担的多个职责的类。上帝类违背了分而治之的基本思想以及单一职责的设计原则，严重影响软件的可维护性和可理解性。但上帝类又是一种比较常见的代码坏味。因此，针对上帝类的检测与重构一直是代码重构领域的研究热点之一。为此，本文提出了一种基于深度神经网络的上帝类检测方法。该方法不仅利用了常见的软件度量，而且充分利用了代码中的文本信息，意图通过挖掘文本语义揭示每个类所承担的主要角色。此外，为了解决有监督深度学习所需的海量标签数据，本文提出了一种基于开源代码构造标签数据的方法。最后，基于开源数据集对提出的方法进行了实验验证。实验结果表明，本文所提方法优于现有的上帝类检测方法，尤其是在查全率上有大幅度的提升（提高了 35.58%）。

**关键词**: 代码坏味;软件重构;深度学习

**中图法分类号**: TP311

中文引用格式: 卜依凡,刘辉,李光杰.基于深度学习的上帝类检测.软件学报. http://www.jos.org.cn/1000-9825/0000.htm

英文引用格式: Bu Yifan, Liu Hui, Li Guangjie. Deep Learning Based God Class Detection. Ruan Jian Xue Bao/Journal of Software, 2018 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

## A God Class Detection Approach Based on Deep Learning

BU Yifan, LIU Hui，LI Guangjie

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

**Abstract**: God Classes refers to certain classes that have assumed more than one functionality, which obey the Single Responsibility Principle and consequently impact on the maintainability and intelligibility of software system. Studies, detection and refactoring included, of God Class have always attracted research attentions because of its commonness. As a result, we propose a neural network based detection approach to detecting God Class Code Smell. This detection technology not only makes use of common metrics in software, but also exploit the textual information in source code, which we intend to reveal the main roles the class plays through mining text semantics. In addition, in order to solve the massive labeled data required for supervised deep learning, we propose an approach to constructing labeled data based on open source code. Finally, the proposed approach is evaluated on an open source data set. The result of evaluation shows that the proposed approach outperforms state-of-art, especially the recall has been greatly improved by 35.58%.

**Key words**: Code Smell; software refactoring; deep learning

## 1 引言

随着产品需求的不断变更，在程序设计之初设计好的代码框架需要不断调整以实现功能的变更。长此以往，程序将逐渐偏离原有的框架，使得整个程序混乱不堪，难以进行扩展和维护。为此，人们提出了软件重构

* 基金项目: 国家重点研发计划(2016YFB1000801)、国家自然科学基金重大项目(61690205)

Foundation item: The National Key Research and Development Program of China (2016YFB1000801) and the National Natural Science Foundation of China (61690205).

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

以对此类软件进行优化，在不改变其软件外部特性的情况下提高软件的设计质量，进而提高软件的可维护性和可扩展[1]。多年来在开发过程中的实践与应用表明[1,2,3]，代码重构在提升程序的可读性和可维护性等方面都有着显著的作用。通过对系统结构的重新整理，开发人员不仅可以改进原有的系统设计，延长软件的生命周期，还能够通过改善代码逻辑来增强代码理解，有助于从中发现程序缺陷[4]。

软件重构的关键步骤之一是明确需要重构的代码片段[4]。而为了帮助开发人员确定需在程序中的何处进行重构操作，Fowler 等人提出了代码坏味（Code Smell）的概念[4]，意指软件系统中影响软件质量的设计问题。Fowler 等一共提出了 22 种代码坏味，包括克隆代码、特征依恋、长方法等。基于此定义，研究人员提出了一系列的自动或半自动的方法以从代码中检测这些代码坏味[5,6,7,8]。代码坏味的概念及其检测方法极大地推动了自动化软件重构的应用和推广，成为软件重构领域的重要研究热点和研究难点。

本文针对上帝类进行深入研究，研究其自动化的检测方法。上帝类是一种常见的代码坏味，指的是某个承担了本应由多个类分别承担的多个职责的类[9]。上帝类违背了分而治之的基本思想以及单一职责的设计原则，严重影响软件的可维护性和可理解性[4]。对于上帝类的出现，Fowler 推荐使用提取类（Extract Class）或提取子类（Extract Class）等重构操作，将一个大类拆分为小类，以提取出过大类中的一部分职责。为了提醒程序员及时处理上帝类，研究人员提出了众多检测算法以自动判定某个给定的类是否为上帝类[5,6,7,10]。现有的检测算法主要基于代码行数、圈复杂度、内聚度等常见的软件度量来判定给定的类是否为上帝类[11]。不同的检测方法往往采用不同的度量项，使用不同的阈值[12,13,14]，因此不同检测方法间的检测结果往往存在较大的差异[5]。此外，现有检测方法的查全率和查准率偏低，导致程序员这些检测方法和检测工具难以在工业界广泛使用。

为此，本文提出了一种基于深度神经网络的上帝类检测方法。该方法不仅利用了常见的软件度量，而且充分利用了代码中的文本信息，意图通过挖掘文本语义揭示每个类所承担的主要角色。此外，本方法首次将深度学习技术应用于上帝类的检测。深度学习在计算机视觉，自然语言处理等领域经过广泛的实践，得到了很大的发展。相较于传统的机器学习，深度学习可以更容易地捕捉到输入数据中的深层关联，经过多层映射和抽象，拟合出更符合输入数据分布的模型。本文利用深度神经网络在文本处理方面的特长，将文本信息加入对上帝类的验证中，同时结合与上帝类在耦合度，内聚度，类规模等属性相关的多个度量项，以深度学习善于自动选择原始数据特征的优势帮助提取出这些度量项之间的相互关联，从而综合评判待检测程序是否应为上帝类代码坏味。

有监督的深度学习通常需要大量的标记数据来作为训练样本。但手工标记上帝类样本数据需要消耗大量的人工，难以收集足够的训练样本。为此，本文提出了一种借助开源项目源码来构建标签数据集的方法。通过预定义的类合并操作，实现上帝类样本的自动生成和标注。考虑到 Github、SourceForge 等开源网站上有海量开源程序，该方法可以自动构造海量的带标签的正负训练样本，从而为基于深度学习的上帝类检测奠定了基础。

最后，对本文所提出的上帝类自动检测方法进行了实验验证。在第三方开源数据集上的实验结果表明，该方法优于现有的检测方法。在不降低查准率的前提下，较大幅度地提高了上帝类检测的查全率（35.58%=95.56% - 59.98%），最终综合提高了上帝类坏味检测的 F1 值（2.39%=8.15% - 5.76%）。

本文第 2 节介绍了相关研究的现状，并对此进行了总结与分析。第 3 节具体介绍了本文提出的上帝类检测方法。第 4 节对所提方法进行了验证与评估。第 5 节对论文进行结论和展望。

## 2 相关工作

### 2.1 上帝类

Fowler 等人提出了代码坏味（Code Smell）的概念并列出来数十种常见的代码坏味。其中大类（Large Class）是承担太多职责而变得臃肿的类。Fowler 认为，这样的类不仅会增加类中代码的理解难度，同时也容易导致其他代码坏味的出现[1]。Brown 等人引入了设计反模式（Antipattern）的概念来代指在程序设计过程中所出现

的设计缺陷[15]。Blob Class 是典型的反模式。当单个类中包含超过 60 个成员变量及方法时，Brown 认为此类违反了单一职责原则，应该对其进行重构。在此之后， Lanza 等人正式提出了上帝类（God Class）的概念，表示某些对外部数据操纵过多的类，并指出这些类通常还会出现类内成员间内聚较低或类内复杂度过高的问题[9]。大类、Blob Class 以及上帝类本质上类似，都是指某个类承担了过多的职责，从而导致该类过于复杂、缺乏内聚等问题。

研究者们迄今已提出了多种方法来对这样一些在项目中承担过多职责的类进行检测与重构。Marinescu 等人提出了一种基于度量值指标的方法来确定对包括上帝类在内的 14 种代码坏味的检测策略，并将此方法实现为工具 iPlasma[6]。他们根据各代码坏味的特征与定义选择不同的度量项组合，以预设阈值的方式为各个坏味确定不同的检测方案。如公式 1 所示，iPlasma 将类内圈复杂度（Weighted Method Count，WMC），类内内聚度（Tight Capsule Cohesion，TCC）和对外访问数（Access to Foreign Data，ATFD）三个度量值综合起来以实现对上帝类的判断：

$$(\mathrm{ATFD} > \mathrm{few}\ \&\ \mathrm{WMC} \geq \mathrm{very\_high}\ \&\ \mathrm{TCC} < \mathrm{one\_third}) \rightarrow \mathrm{isGodClass} \quad (1)$$

其中，*few*，*very_high* 以及 *one_third* 均为常量。当一个类的三个度量值同时满足上述条件时，iPlasma 会将其判定为上帝类。

Moha 等人定义了一种领域特定语言（Domain-Specific Language，DSL），并利用这种语言对代码坏味检测规则进行定义，以此形成了方法 DÉCOR[7]。通过对各类坏味的概念进行文本分析，DÉCOR 将代码坏味的定义转化为以这种 DSL 语言表示的坏味检测算法，在将其给定的算法实现为真正的程序后，便可以完成对于此代码坏味的检测。DÉCOR 在利用 NMD，NAD 和 LCOM5 等度量项来定义对于上帝类的检测规则的基础上，还综合了一些文本信息来辅助其判断，如当类名中出现“Process”，“Control”，“Ctrl”等字样时，即说明此类为上帝类的可能性相对较大。

Tsantalis 等人提出了一种利用杰卡德距离来衡量两个代码实体之间的相似性的方法，并以此方法为基础实现了用于检测代码坏味的工具 JDeodorant。其最初用于检测特征依恋（Feature Envy）代码坏味并推荐移动方法（Move Method）的重构方案[8]，在之后则逐步增加了对于另外四种坏味——重复代码（Duplicated Code），switch 语句（Type Checking/Switch Statement）[16]，长方法（Long Method）[17]以及上帝类（God Class）[10]——的检测与重构推荐。JDeodorant 为类中的每个成员生成与之相关的其他成员集合（entity set），即访问此成员以及被此成员访问的其他成员的集合。他们将两个成员之间的距离定义为其相关成员集合间的杰卡德距离，计算方式如公式 2 所示：

$$\mathrm{distance}(m_1, \mathrm{m}_2) = 1 - \frac{S_{m_1} \cap S_{m_2}}{S_{m_1} \cup S_{m_2}} \quad (2)$$

其中，$m_i$ 为类中所声明的一个成员，$S_e$ 则为成员 $e$ 的相关成员集合。基于此距离度量项，JDeodorant 可以由此定义一个类内的各个成员之间的节点距离，从而为被检测类构建一个以类内成员为节点的树状图层次结构，再根据所需阈值，对树状图进行横向切割[10]。一旦树状图可以被切割为多个节点簇，则说明此类存在进行提取类（Extract Class）重构操作的必要性，即可判定此类是一个上帝类。随后钟林辉等人在 JDeodorant 的基础上对杰卡德距离进行了扩展，公式如下：

$$\mathrm{E\_distance}(\mathrm{m}_1, \mathrm{m}_2) = 1 - \frac{\left|S_{m_1} \cap S_{m_2}\right|}{\left|\sum_{i=1}^{n} \left|S_{m_i}\right|\right|} \quad (3)$$

其中，$m_i$ 为类中声明的第 $i$ 个成员，$n$ 为类中的全部成员个数。他们利用扩展的杰卡德距离实现了一种改进的层次聚类重构方式，以弥补原方法难以衡量全局范围内的成员相似性的缺陷[18]。

综上所述，上述检测方法主要依赖于不同的代码度量项（structural metrics）以及相应的阈值来判定某个类是否为上帝类。不同的检测方法往往采用不同的度量项，使用不同的阈值[5,13,14]，因此不同检测方法的检测

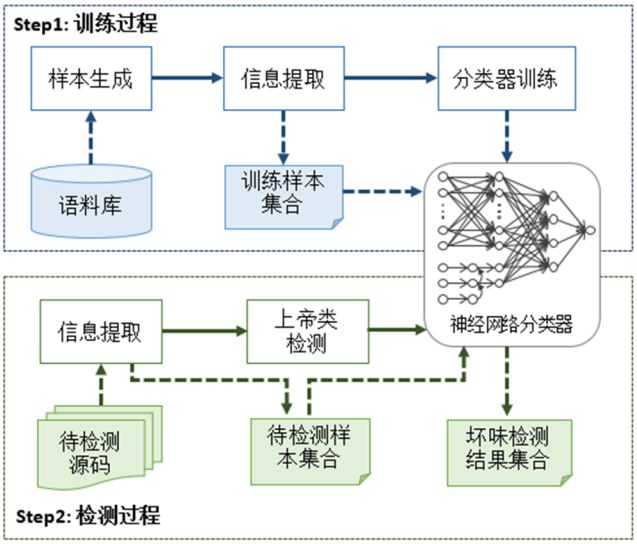


Fig.1 Overview of the Deep Learning Based God Class Detection

图 1 基于深度学习的上帝类代码坏味检测方法

结果往往存在较大的差异[5]。此外，现有检测方法的查全率和查准率偏低，导致这些检测方法和检测工具难以在工业界广泛使用。

### 2.2 基于机器学习的上帝类坏味检测

随着机器学习的演进与发展，研究者们提出了一批基于各类机器学习算法的代码坏味检测方法。Kreimer 在 2005 年时提出了一种基于决策树（Decision Tree）的对大类（Large Class）和长方法（Long Method）两种代码坏味的检测方法[19]。Khomh 等人利用贝叶斯信念网络（Bayesian Belief Network）实现了对上帝类相关反模式的检测[20]。Maiga 等人则实现了基于支持向量机（Support Vector Machine，SVM）的上帝类检测方法[21]。Palomba 等人提出了一种基于信息检索技术（Information Retrieval，IR）来利用程序中的文本信息进行坏味检测的方法[22]，类似的，马赛等人尝试了利用潜在语义分析技术来对上帝类进行检测[23]。Fontana 等人汇总了几种常见的机器学习算法（J48，JRip，ERandom Forest，Baive Bayes，SMO 以及 LibSVM）来共同检测各类代码坏味，以便比较并总结不同算法在代码坏味检测领域中的表现与差异[24]。

有别于基于传统机器学习的坏味检测方法，本文充分利用了最新的深度学习技术，能够学习更加复杂的逻辑关系。此外，本文不仅利用了软件度量信息，也使用了代码中的文本信息以挖掘给定类所承担的角色。最后，在训练集的收集方面现有方法主要依赖于手工收集训练数据，而本文提出了标签训练数据自动生成方法。

## 3 上帝类检测方法

为了自动检测上帝类，本文提出了一种基于深度学习的检测方法。3.1 节给出了本文所提方法的概览介绍，之后的多个小节详细介绍了该方法的各个关键步骤。

### 3.1 方法概述

本文提出的基于深度学习的上帝类简单方法如图 1 所示。首先，利用大量的开源软件项目工程作为代码语料库，实现了一种可以自动生成标签样本的工具来生成深度学习训练所需的大规模数据集。此工具基于如下假设，若认定开源项目的现有设计合理，那么若将其中的两个类合并为一个类，则可以认为这个大类承担

**Table 1** Metrics

**表 1** 度量项介绍

| 度量类型 | 度量项 | 度量项定义 |
|---|---|---|
| 耦合度相关 | ATFD | 被检测类所访问的外部类属性的个数 |
| | DCC | 被检测类所访问的外部类的个数 |
| | DIT | 被检测类距离其所在的继承关系树根节点的最大层数 |
| 内聚度相关 | TCC | 类中通过访问相同属性而发生连接的方法对的个数 |
| | LCOM | 类中访问过相同属性的方法对数与从未访问过同样属性的方法对数的差 |
| | CAM | 基于被检测类内部方法参数类型相似度的内聚度 |
| 代码规模相关 | WMC | 类的圈复杂度 |
| | LOC | 类内代码行数 |
| | NOPA | 类内的 public 属性个数 |
| | NOAM | 类内的 get/set 方法个数 |
| | NOA | 类内属性个数 |
| | NOM | 类内方法个数 |

了两个类的职责。因此本工具在不改变类的外部行为的前提下，通过尽可能多地对源码中的类进行两两合并以形成大类正样本集合，随后从其余未参加合并的类中随机抽取出相应数目的类作为负样本集合。为了尽可能保留代码中与上帝类代码坏味相关的特征，从正负样本集中分别提取出符合预设输入格式的文本信息与软件度量，作为神经网络分类器的输入，分类器的预期输出为样本的标签（即是否为上帝类）。经过多次迭代训练后，可以得到最终被训练好的神经网络分类器。对于给定的待测程序，首先提取每个待测试类的文本信息和软件度量。将此信息依次输入训练好的神经网络分类器，得到最终的检测结果。后续章节将依次介绍每个关键步骤的具体细节。

### 3.2 神经网络的输入

由于直接以全部代码文本作为深度神经网络分类器的输入时所要求的模型学习难度过大，我们需要对代码文本进行一部分的预处理操作，从代码文本中提取出与上帝类代码坏味相关的特征集合，并且摒弃一部分就本学习任务而言无价值的无关特征，以便降低模型构建的难度。为了在充分利用代码中的各类特征与信息时避免因特征过多而导致的维度爆炸，我们经过反复考量与对比，选取了数个与上帝类代码结构相关的代码度量项作为神经网络分类器的软件度量特征，同时从代码中提取了部分相关标识符来作为神经网络分类器的文本信息特征。然而尽管已剔除了一部分的无关特征与冗余特征，这些原始特征彼此间的相互关联以及输入特征与输出标签间的潜在映射关系依然需要进一步地分析。因此我们利用了神经网络分类器来对原始特征进行映射与学习，以便最终输出与样本标签所对应的分类结果。

软件度量信息是代码坏味检测研究中常用的判断依据。我们综合了 12 个可以在不同方面体现上帝类特征的代码度量项，基本涵盖上帝类代码坏味结构特征的各个方面，以期能够在耦合度，内聚度，复杂度以及代码规模等方面更全面地表示代码的结构特征。表 1 为所选度量项的详细信息。

此外，我们收集了被检测类中所声明的各成员标识符来作为输入的一部分。Arnaoudova 等人的研究表明，有意义的标识符可以有效揭示代码组件在程序中的角色、行为与功能[25]。一个类中的各个成员通过完成各自的功能，共同构成其所在类的对外行为与角色，故可以认为在理想情况下，存在于一个类中的多个成员标识符之间应该存在着语义上的相互关联。因此，我们将这种隐含在标识符内的语义关联作为衡量被检测类内聚度的一个重要依据，并结合上述的软件度量指标，组成了本文所提方法的输入：

$$\text{input} = < \text{identifiers}, \text{metrics} > \tag{4}$$

$$\text{identifiers} = \{\text{name}(m_1), \text{name}(m_2), \dots, \text{name}(m_n)\} \tag{5}$$

$$\text{metrics} = \{metric_1, metric_2, \ldots, metric_{12}\} \quad (6)$$

其中，$m_i$ 为被检测类中所声明的第 $i$ 个方法或属性，$name(m_i)$为 $m_i$ 的标识符，$metrics$ 则为前述 12 个度量项的集合。

### 3.3 标识符的表示方式

为了能够挖掘标识符之间的深层语义关联，我们利用 Mikolov 等人所提出的著名词向量化模型 Word2Vector 将标识符中的词语映射到高纬向量空间[26,27]，以词向量在高维空间中的分布来揭示词与词之间的相似性关系。作为自然语言处理领域的重要工具，Word2Vector 构建了一个以给定的文本作为输入输出的神经网络。在进行训练之后，可以利用此模型的隐含层将词语转化为稠密向量，实现以向量相似性来表示语义相似性的目的。我们利用大量项目源码作为代码语料库，对 Word2Vector 模型进行训练，构建了一个针对程序语言的向量空间。随后根据如下步骤对将作为神经网络分类器输入的各标识符分别进行预处理：

(1) 根据驼峰命名法规则（Camel Case）和下划线命名法规则对标识符进行分词，将单个标识符拆分为多个逻辑单字；

(2) 利用已训练好的 Word2Vector 模型将各逻辑单字分别映射为高维空间中固定长度（200 维）的词嵌入向量（word embedding）；

(3) 将单个标识符中包含的各分词向量相加后取均值构成一个新的词向量，以作为此标识符在向量空间中的表示[28]。

针对训练过程所用的 12 个项目源码进行的统计分析表明，训练集中 95.8%的类中不会声明超过 50 个成员。因此出于神经网络的设计需要，我们将神经网络输入中的单个样本的标识符个数固定在 50 个，即只针对被检测类中所声明的前五十个成员进行预处理，类中成员数少于 50 个则以全零向量来做补零扩展。

### 3.4 基于深度神经网络的分类器

本文所提的基于深度神经网络的分类器结构如图 2 所示。如上文所述，此分类器的输入分为文本输入与度量输入两部分。文本输入由类中的成员标识符组成。类中的成员标识符在经过预处理（详见 3.3 节）后，已经由文本信息转为数值信息，将以词向量（输入形式为 50×200 矩阵）的形式经过输入数据屏蔽层（Masking，mask_value=0）进入长短时记忆网络（Long Short-Term Memory, LSTM）中，其中 LSTM 层激活函数为 sigmoid 函数，输出维度为 2，并对该层权重做均匀分布（uniform）初始化。长短时记忆网络作为循环神经网络（Recurrent Neural Network, RNN）的变体，通过规避梯度消失问题弥补了循环神经网络长期记忆失效的缺陷[29]，因而在

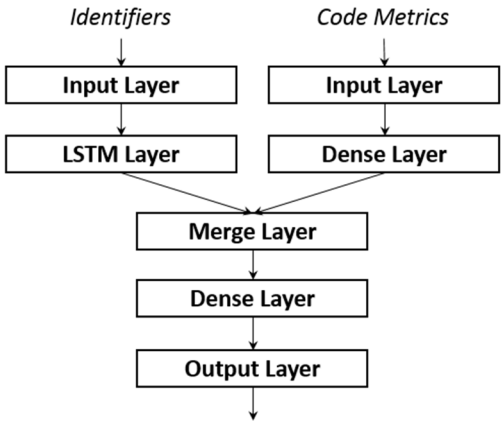


Fig.2 Neural Network based Classifier

图 2 神经网络分类器网络结构

自然语言处理领域中获得了充分应用。我们希望利用长短时记忆网络善于从长序列中提取关键语义特征的优势来对所输入的多个标识符词向量进行处理，以帮助分类器分析被检测类在语义上的内聚性。度量输入体现了被检测类的结构特征。通过将从类中提取出的 12 个度量值输入全连接层（Dense），分类器可以在有监督的情况下对输入进行迭代训练，逐步调整出与训练集标签最匹配的参数组合，其中本全连接层激活函数为 tanh 函数，输出维度为 12，并对本层权重做均匀分布（uniform）初始化。之后两部分数据会经由融合层（Merge）以向量拼接（concatenate，axis=-1）的形式合并，再由一层全连接层映射到最终的 Sigmoid 输出层，此间的全连接层激活函数为 tanh 函数，输出维度为 4，权重初始化为全零矩阵（zero）；输出层激活函数为 sigmoid 函数，输出维度为 1。最终选取的模型损失函数（loss function）为 binary_crossentropy 函数，优化器（optimizer）为 adam 自适应方法，迭代次数（epochs）为 10 次，批尺寸（batch_size）为 5。

### 3.5 训练集的生成方式

深度神经网络的结构决定了其对于训练样本数量的要求。为了防止出现过拟合现象，我们需要大量的数据样本来帮助提高神经网络的泛化能力。对此我们提出了一种通过合并类操作来自动构建标签样本集的工具来辅助收集上节所述分类器在训练过程中所需的样本集。在项目源码可编译的前提下，我们可以利用本样本生成工具源源不断地生成分类器所需的正负样本，以及从所生成的样本集中提取神经网络分类器所需要的全部特征输入。利用本工具来帮助收集带标签的训练样本集，可以省去人工识别上帝类代码坏味时所需要耗费的大量时间和人力。由于此工具实现了自动的样本生成和特征提取预处理操作，因此只要能够获取到优质的可编译开源项目资源，整个训练样本的生成及特征提取过程均不需要人工干预。在这样的前提下，以互联网中基数庞大的开源项目为基础，原则上我们可以利用此工具获取到足够充足的上帝类标签样本集。

假设某个程序的现有设计是合理的，那么将其中的两个类合并成一个大类则是不合理的。这个合并起来的大类承担本该由多个类一起承担的多个角色，因此是一个上帝类。基于此，我们利用开源的高质量代码，自动构造上帝类的正样本。为了保证合并后大类的外部行为与合并之前保持一致，我们会在将两类合并的基础上尽可能不对两类源码进行修改。

如算法 1 所示，在获取到可编译的开源项目源码后，为了在最大化节省样本生成时间的基础上秉持多多

**算法 1 类型合并前后的判断逻辑**

```
Input: project
Output: newUnits

singleSet ← ∅
newUnits ← ∅
//对 project 中的各编译单元进行第一次筛选，判断编译单元是否可以进行合并类操作
for each unit in project do
    if isSimpleUnit(unit) then
        singleSet.add(unit)
    end if
end for
for each unit1 in singleSet do
    for each unit2 in singleSet do
    //第二次筛选，初步判断 unit1 与 unit2 是否可以合并
        if canBeCombined(unit1, unit2) then
            newUnit ← combine(unit1, unit2)
            //第三次筛选，判断 unit1 与 unit2 合并后是否出现错误
            if newUnit=null then
                continue
            else
                newUnits.add(newUnit)
            end if
        end if
    end for
end for
return newUnits
```

益善的原则尽可能多地生成训练集正样本，我们会对项目内的各编译单元进行三次筛选以确定最终可用于生成训练样本集的编译单元集合。三次筛选的具体实现步骤如下：

(1) 第一次筛选是对单个编译单元中的顶层类进行判断，主要目的是筛选出能进行合并操作的常规 Class 类。

(2) 随后我们将对剩余的编译单元集合进行两两配对并开始第二次筛选，这次筛选为初步判断配对后的两个编译单元是否可以在不导致语法错误的前提下进行合并，如判断两个类中是否存在成员名冲突，或两个类是否继承于不同的父类。

(3) 一对编译单元如果可以通过上述判断，便对此二者实现合并类操作，以得到最终需要的大类。此时进行第三次筛选，用以确定此次合并类操作没有导致任何的程序错误与外部行为变化。

两个类 C1 和 C2 的合并类算法如下：

(1) 生成一个空的新类（NewClass），并 C1 和 C2 中的模块导入语句（import statement）以及全部的成员声明语句都迁移至 NewClass 类内；

(2) 在整个工程中搜索 C1 和 C2 以属性类型、方法参数类型、方法返回类型等一系列形式出现在工程中其他类中的情况，将这些类中的 C1 和 C2 类型替换为 NewClass 类型；

(3) 删除工程中的 C1 和 C2 类。此时整个工程中全部与 C1 和 C2 相关的代码均已被 NewClass 替换，NewClass 彻底取代了 C1 和 C2 类在此系统中的角色与职责。

我们将通过三次筛选的 NewClass 定义为人为注入的代码坏味，由此便可以计算 NewClass 的 12 个度量值。根据 3.2 中所设计的输入格式收集到所需数据后，我们即可导出以下格式的训练集正样本：

$$\text{positiveSample} = < \text{input}, \text{output} > \tag{7}$$

$$\text{input} = < \text{identifiers}, \text{metrics} > \tag{8}$$

$$\text{identifiers} = \{\text{name}(m_1), \text{name}(m_2), \dots, \text{name}(m_n)\} \tag{9}$$

$$\text{metrics} = \{metric_1, metric_2, \dots, metric_{12}\} \tag{10}$$

$$\text{output} = 1 \tag{11}$$

为了获取负样本集合，我们将经过第一次筛选的编译单元集合定义为全集 *collection*，将可成功合并为大类的类的集合定义为集合 *posCollection*，由此可定义 *posCollection* 在 *collection* 上的补集为 *negCollection*，即

$$\text{negCollection} = \complement_{collection} posCollection \tag{12}$$

假设某个程序的现有设计是合理的，那么 *negCollection* 中任意一个类都是上帝类的负样本。为保证训练效果，我们从每个开源程序中收集的正负样本的比例控制在 1:1。为了保证训练集负样本的典型性以及此样本生成工具设计思路的合理性，我们从 *negCollection* 中随机抽取一个与正样本集合样本容量相同的集合，并入最终的带标签的训练样本。

## 4 实验验证

为了对所提方法进行验证，我们收集了 12 个高质量开源项目源码作为样本生成所需的代码语料库，并以此为基础具体实现了所提方法，相关的代码与数据已上传至 https://github.com/bby8808/GodClassDetection。同时，我们以 Palomba 等人提出的代码坏味数据集作为测试样本[30]，对本文所提出的方法进行验证与分析。

### 4.1 研究问题

在实验验证阶段，我们希望通过分析以下三个问题来对所提出的方法进行评估：

(1) 研究问题 1：该方法是否能准确有效地检测出上帝类？其查全率和查准率是否优于现有方法？

(2) 研究问题 2：所提神经网络分类器中的两个特征输入（代码文本特征与代码结构特征）对最终结果分别有什么影响？即如果只有其中的一个特征输入，分类器的性能会如何变化？

(3) 研究问题 3：利用其他网络模型（如卷积神经网络 CNN 和全连接网络 Dense）替代神经网络分类器中所使用的长短时记忆网络（LSTM），能否进一步提高分类器的性能，如查全率、查准率等？

(4) 研究问题 4：该方法训练一个神经网络分类器的时间需要多久？利用已训练好的分类器进行预测又需要多长时间？

研究问题 1 关注的是所提方法与当前方法的检测结果在查准率（*precision*）与查全率（*recall*）等指标上的区别。为了回答这个问题，我们选择了 JDeodorant[10]作为评估阶段的对比实验对象。之所以选择 JDeodorant 作为参照，是因为 JDeodorant 作为知名的代码重构推荐工具，其代码坏味的检测能力已经获得了业内的广泛认可[5,23,30]。比较新的一些上帝类检测方法，如 TACO[22]和 DÉCOR[7]，都没有公开的实现体（源代码或者可执行程序），因此难以与他们在同一数据集上进行实际比较。

研究问题 2 关注神经网络分类器输入特征选取的有效性。我们在保持模型其他部分不变的情况下，分别剪除原模型中的度量项代码结构特征输入与标识符代码文本特征输入，将原模型拆分为两个独立的单一输入神经网络并分别加以调优训练。以各分类器在同一测试集上的最优平均 F1 值作为衡量指标来分析所提的两个特征分别在整个方法中所起的作用。

研究问题 3 主要关注在本方法所构造的神经网络分类器中文本特征的处理方式。我们通过将所提网络模型中的长短时记忆网络替换为全连接神经网络和卷积神经网络，并同样以各分类器在同一数据集上的最优平均 F1 值作为指标来帮助考察和分析在已有的方法架构下三种神经网络对于最终结果的影响。

研究问题 4 则关注所提方法的时间复杂度情况。由于基于机器学习的分类器训练通常都需要在训练过程中花费过多的时间成本，因此对于时间代价的考察也应成为我们对本文所提方法的一个评估因素。对研究问题 4 的考察可以根据所提方法在训练过程与预测过程中分别所需花费的时间成本，探讨基于深度学习的本方法在时间性能上的具体表现。

### 4.2 实验数据

#### 4.2.1 训练数据

我们选择了 12 个开源项目以自动构造神经网络的训练数据。表 2 给出了训练集所用工程各自的名称，版本，类的数量（NOC），方法数量（NOM）以及代码行数（LOC）。为了保证训练集样本的可靠性，我们选取了如下的 12 个开源项目来帮助神经网络分类器的训练。作为知名的开源项目，这些项目拥有更高的代码质量，因此可以帮助我们获取到更准确的标签样本，从而提高神经网络的训练效率。同时，我们在选取训练样本代码语料库的时候，会刻意参考项目的开发者与项目所涉及的领域。这一筛选条件的目的主要是为了消除特定开发者或特定项目领域可能会引入的特定代码风格倾向，以便保证训练集标签样本的综合性。

#### 4.2.2 测试数据

在测试数据的准备方面，我们利用了 Palomba 等人提出的一个开源代码坏味数据集来作为评估实验的实

**Table 2** Projects for Train Set

**表 2** 训练集所涉项目列表

| 项目名称 | 项目领域 | 项目版本 | NOC | NOM | LOC |
|---|---|---|---|---|---|
| android-backup-extractor | abe 文件解析 | 20140630 | 1,693 | 13,837 | 304.458 |
| ArtOfIllusion | 3D 动画 | v3.0 | 486 | 6,916 | 152,207 |
| Areca | 文件备份 | v7.4.7 | 473 | 5,055 | 88,126 |
| c3p0 | 数据库连接 | v0.9.5 | 122 | 2,117 | 29,158 |
| DavMail | Exchange 服务器代理 | v4.5.1 | 137 | 1,478 | 36,445 |
| FCKeditor | 文本编辑 | v2.6 | 52 | 224 | 5,518 |
| FreePlane | 思维导图 | v1.3.12 | 787 | 6,938 | 124,937 |
| Grinder | 性能测试 | v3.6 | 490 | 4,291 | 101,293 |
| JDeodorant | 代码结构分析 | v5.0.8 | 391 | 4265 | 84,726 |
| JExcelAPI | Excel 操作接口 | v2.6.12 | 424 | 3,118 | 90,555 |
| JUnit | 单元测试 | v4.10 | 123 | 866 | 11,734 |
| weka | 机器学习 | v3.9.0 | 1,348 | 20,182 | 444,493 |

**Table 3** Projects for Test Set

**表 3** 测试集所涉项目列表

| 项目名称 | 项目领域 | 项目版本 | NOC | LOC | NOPS | NONS |
|---|---|---|---|---|---|---|
| Ant | 源码编译 | v1.8.2 | 1,166 | 253,347 | 6 | 1,080 |
| Derby | 数据库 | v10.9.1.0 | 2,797 | 1,217,300 | 26 | 2,383 |
| FreeMind | 思维导图 | v0.9.0 | 438 | 86,172 | 2 | 392 |
| Hadoop | 分布式架构 | v0.9 | 238 | 49,985 | 1 | 195 |
| HBase | 分布式数据库 | v0.94 | 1,054 | 365,454 | 8 | 940 |
| Ivy | 项目跟踪管理 | v2.0.0-beta2 | 330 | 71,680 | 2 | 680 |
| JEdit | 文本编辑 | v4.5.0 | 513 | 185,571 | 6 | 503 |
| JHotDraw | 应用程序框架 | v7.5.1 | 613 | 133,811 | 2 | 547 |
| Pig | 分布式数据库 | v0.8.0 | 1,021 | 398,301 | 3 | 959 |
| Qpid | 消息中间件 | v0.18 | 2,217 | 389,177 | 6 | 1,740 |
| Struts | Web 程序框架 | v2.2.1 | 1,837 | 261,666 | 2 | 1618 |
| Wicket | Web 程序框架 | v1.4.20 | 2,002 | 28,873 | 4 | 1,804 |
| Xerces | XML 解析 | v2.3.0 | 718 | 197,050 | 6 | 455 |

验对象，其中的代码坏味记录均经由人工验证[30]。之前的代码坏味相关研究通常会综合多个代码坏味检测工具的检测结果作为研究数据的来源[24,31]。尽管通过多个工具的互补可能可以获取到不错的检测精度，但人工进行的代码坏味检测依然可以被认为是最为可信的评估参考标准。此数据集中包含了在 30 个软件项目的 395 个历史版本中 13 种代码坏味的检测情况。我们选择包含上帝类而且可以通过编译的 13 个项目作为测试项目。表 3 展示了验证所用的 13 个开源项目的详细信息，包括其中的正样本（在原有数据集上被标记为上帝类的类）个数（NOPS）与负样本（没有被标记为上帝类的类）个数（NONS）。

### 4.3 实验过程

我们以表 2 中所列出的各开源项目作为代码语料库，利用如前所述的标签样本生成工具（详见 3.5 小节）逐条生成固定格式的正负样本数据（详见 3.2 小节），以构建出神经网络分类器的训练集（其中包括正样本 845 条，负样本 775 条）。随后经过在训练集上的多轮训练，我们可以得到一个以类中成员标识符集合与软件度量值集合为输入的分类器。此分类器能够输出被检测类中存在上帝类代码坏味的概率。

以表 3 中所列的各项目作为测试项目，我们解析其中的源码并按照神经网络所定义的输入格式从项目中提取数据作为测试样本，由此生成测试集。将测试集输入已训练好的神经网络分类器后，得到的输出集合即为神经网络分类器在此测试集上的预测结果。作为参照，我们在 JDeodorant 中同样对此测试集进行上帝类代码坏味检测，同时导出其检测结果以便与本文所提方法进行对比。

所提模型代码基于 keras 实现。在模型优化阶段，我们以交叉熵作为损失函数，并选择自适应学习率的 Adam 作为优化算法。同时，为了直观对比不同方法的上帝类代码坏味检测能力，我们利用 Palomba 所提出的代码坏味数据集作为测试样本的正确标签，并以如下方法来分别计算两种检测结果与正确标签之间的查准率（*precision*）、查全率（*recall*）以及 F1 值：

$$precision = \frac{true\ positives}{true\ positives\ + false\ positives} \tag{13}$$

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{14}$$

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{15}$$

### 4.4 实验结果与分析

为了回答研究问题 1，我们总结了本方法与 JDeodorant 在相同测试集上的上帝类检测结果，如下表 4 所示。其中第 1 列显示测试集中的项目名称，2-4 列显示我们所提方法测试结果的查准率与查全率，5-7 列显示

**Table 4** Evaluation Result on God Class Detection

**表 4** 上帝类检测结果

| 项目名称 | 本文所提方法 | | | JDeodorant | | |
|---|---|---|---|---|---|---|
| | 查准率 | 查全率 | F1 值 | 查准率 | 查全率 | F1 值 |
| Ant | 5.13% | 100% | 9.76% | 1.7% | 50% | 3.29% |
| Derby | 5.05% | 92.3% | 9.58% | 4.01% | 42.3% | 7.33% |
| FreeMind | 4.88% | 100% | 9.31% | 4.08% | 100% | 7.84% |
| Hadoop | 3.84% | 100% | 7.40% | 0.00% | 0.00% | 0.00% |
| HBase | 6.25% | 100% | 11.76% | 2.52% | 62.5% | 4.84% |
| Ivy | 3.28% | 100% | 6.35% | 4.00% | 100% | 7.69% |
| JEdit | 6.98% | 100% | 13.05% | 4.40% | 66.7% | 8.26% |
| JHotDraw | 1.96% | 100% | 3.84% | 1.69% | 50% | 3.27% |
| Pig | 2.16% | 100% | 4.23% | 0.00% | 0.00% | 0.00% |
| Qpid | 3.49% | 100% | 6.74% | 3.23% | 83.3% | 6.22% |
| Struts | 0.98% | 50% | 1.92% | 1.64% | 100% | 3.23% |
| Wicket | 4.82% | 100% | 9.20% | 4.17% | 75% | 7.90% |
| Xerces | 6.98% | 100% | 13.05% | 8.82% | 50% | 14.99% |
| **平均值** | **4.29%** | **95.56%** | **8.17%** | **3.09%** | **59.98%** | **5.76%** |

JDeodorant 的测试结果的查准率与查全率。表中最后一行列出了各方法表现的平均值。

根据表 4，我们可以看出：

- 在对于上帝类代码坏味的检测能力上，所提方法在此测试集中的表现总体上优于 JDeodorant，平均 F1 值提高了 2.39%=（8.15% - 5.76%）。
- 其中本方法相对于 JDeodorant 的优势在查全率上尤为明显，平均提高 35.58%=（95.56% - 59.98%）。
- 尽管所提方法的平均查准率（4.29%）是高于已有方法的（3.09%），但总体上两种方法的查准率依然都偏低。

针对研究问题 2，我们设计了一组对比实验来考察所提方法中的代码文本特征与代码结构特征分别对于最终上帝类检测结果的影响。通过对比单一的代码文本特征输入、单一的代码结构特征输入和二者综合输入这三种输入方式下的各神经网络分类器在同一数据集上的上帝类代码坏味检测能力，我们可以直观看出所提方法中的两个输入特征在神经网络中的作用。三种输入方式在测试集上的具体表现如表 5 所示。

由表 5 可以看出：

- 当代码文本特征与代码结构特征均为神经网络的输入时，分类器在测试集上的综合表现优于任何一种单一输入的神经网络分类器，具体表现为双输入分类器的平均 F1 值相较于单文本输入分类器和单数值输入分类器分别提高了 7.01%=（8.17% – 1.16%）和 4.45%=（8.17% - 3.72%）。
- 与代码文本特征相比，代码结构特征对于分类器的预测成功率起到了更大的作用，尤其在查准率上的影响十分明显，平均查准率高出了 1.31%=（1.90% - 0.59%）。

**Table 5** Results on God Class Detection among Features

**表 5** 不同特征输入下的上帝类检测结果

| 项目名称 | 代码文本特征+代码结构特征 | | | 仅代码文本特征 | | | 仅代码结构特征 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 查准率 | 查全率 | F1 值 | 查准率 | 查全率 | F1 值 | 查准率 | 查全率 | F1 值 |
| Ant | 5.13% | 100% | 9.76% | 0.55% | 100% | 1.10% | 1.85% | 100% | 3.64% |
| Derby | 5.05% | 92.3% | 9.58% | 1.08% | 100% | 2.14% | 3.06% | 100% | 5.93% |
| FreeMind | 4.88% | 100% | 9.31% | 0.51% | 100% | 1.01% | 1.74% | 100% | 3.42% |
| Hadoop | 3.84% | 100% | 7.40% | 0.51% | 100% | 1.02% | 1.56% | 100% | 3.08% |
| HBase | 6.25% | 100% | 11.76% | 0.84% | 100% | 1.67% | 2.55% | 100% | 4.97% |
| Ivy | 3.28% | 100% | 6.35% | 0.29% | 100% | 0.58% | 1.08% | 100% | 2.14% |
| JEdit | 6.98% | 100% | 13.05% | 1.18% | 100% | 2.33% | 3.02% | 100% | 5.85% |
| JHotDraw | 1.96% | 100% | 3.84% | 0.36% | 100% | 0.73% | 1.03% | 100% | 2.04% |
| Pig | 2.16% | 100% | 4.23% | 0.31% | 100% | 0.62% | 0.89% | 100% | 1.76% |
| Qpid | 3.49% | 100% | 6.74% | 0.34% | 100% | 0.68% | 1.49% | 100% | 2.93% |
| Struts | 0.98% | 50% | 1.92% | 0.12% | 100% | 0.25% | 0.80% | 100% | 1.58% |
| Wicket | 4.82% | 100% | 9.20% | 0.22% | 100% | 4.42% | 2.04% | 100% | 4.00% |
| Xerces | 6.98% | 100% | 13.05% | 1.30% | 100% | 2.57% | 3.66% | 100% | 7.06% |
| **平均值** | **4.29%** | **95.56%** | **8.17%** | **0.59%** | **100%** | **1.16%** | **1.90%** | **100%** | **3.72%** |

**Table6** Results on God Class Detection among Approaches to Text Feature Exacting

**表 6** 不同文本特征处理方式下的上帝类检测结果

| 项目名称 | LSTM | | | Dense | | | CNN | | |
|---|---|---|---|---|---|---|---|---|---|
| | 查准率 | 查全率 | F1 值 | 查准率 | 查全率 | F1 值 | 查准率 | 查全率 | F1 值 |
| Ant | 5.13% | 100% | 9.76% | 5.08% | 50% | 9.23% | 2.56% | 100% | 5% |
| Derby | 5.05% | 92.3% | 9.58% | 8.43% | 57.69% | 14.71% | 4.24% | 100% | 8.14% |
| FreeMind | 4.88% | 100% | 9.31% | 0.00% | 0.00% | 0.00% | 2.11% | 100% | 4.12% |
| Hadoop | 3.84% | 100% | 7.40% | 0.00% | 0.00% | 0.00% | 1.92% | 100% | 3.77% |
| HBase | 6.25% | 100% | 11.76% | 5.26% | 25% | 8.70% | 3.04% | 87.5% | 5.88% |
| Ivy | 3.28% | 100% | 6.35% | 5.26% | 50% | 9.52% | 1.40% | 100% | 2.72% |
| JEdit | 6.98% | 100% | 13.05% | 13.64% | 50% | 21.43% | 3.61% | 100% | 6.98% |
| JHotDraw | 1.96% | 100% | 3.84% | 1.33% | 50% | 2.60% | 0.58% | 50% | 1.42% |
| Pig | 2.16% | 100% | 4.23% | 2.38% | 33.33% | 4.44% | 1.21% | 100% | 2.4% |
| Qpid | 3.49% | 100% | 6.74% | 2.5% | 33.33% | 4.65% | 1.39% | 66.67% | 2.72% |
| Struts | 0.98% | 50% | 1.92% | 1.52% | 50% | 2.94% | 0.51% | 50% | 1.01% |
| Wicket | 4.82% | 100% | 9.20% | 5.56% | 50% | 10.00% | 2.45% | 100% | 4.79% |
| Xerces | 6.98% | 100% | 13.05% | 4.92% | 50% | 8.96% | 4.14% | 100% | 7.95% |
| **平均值** | **4.29%** | **95.56%** | **8.17%** | **4.30%** | **38.41%** | **7.47%** | **2.43%** | **88.78%** | **4.36%** |

此外，为了回答研究问题 3，我们分别将全连接神经网络（Dense）、卷积神经网络（CNN）以及长短时记忆网络（LSTM）这三种网络模型运用于分类器中的文本特征处理环节。各分类器经过调优后在同一测试集上的具体表现如表 6 所示。需注意的是，表中三种神经网络分类器除文本特征提取环节所用模型不同外，网络其余部分均保持一致。其中，2-4 列数据相关的分类器采用长短时记忆网络处理代码文本特征信息，5-7 列所对应的全连接神经网络在文本特征提取部分的隐含层激活函数为 sigmoid 函数，8-10 列相关分类器则采用三层一维卷积神经网络进行文本特征提取。

由表 6 可以看出：

- 在本文所提方法的架构下，长短时记忆网络的代码文本特征提取能力整体上高于全连接神经网络与卷积神经网络，LSTM 分类器的 F1 值相较于另两者分别高出了 0.7%=（8.17% - 7.47%）和 3.81%=（8.17% - 4.36%）；
- Dense 分类器的表现在测试集中的不同项目间差异很大，既有不少项目的查准率和 F1 值均高于 LSTM 分类器，同时也出现了检测不出项目中的上帝类代码坏味的情况（FreeMind 和 Hadoop）。
- 针对上述情况，为保稳妥，我们认为选择长短时记忆网络（LSTM）作为整个神经网络中的代码文本特征提取层时的最终效果较为良好。

针对研究问题 4，我们对所提方法的时间性能进行了考察。通过在普通配置的个人电脑（16GB RAM，Intel Core CPU i7-6700）上运行所提方法的全部流程，我们记录了所提出的分类器的整个训练与预测过程的耗时情况。表 7 列出了训练集所涉各工程提取训练样本数据时所耗的时间（以分钟为单位）。

**Table 7** Time of Train Set Data Generation

**表 7** 训练样本提取所用时间

| 项目名称 | 训练集生成所需时间（分钟） |
|---|---|
| android-backup-extractor | 191 |
| ArtOfIllusion | 4 |
| Areca | 42 |
| c3p0 | 1 |
| DavMail | 1 |
| FCKeditor | 1 |
| FreePlane | 56 |
| Grinder | 52 |
| JDeodorant | 76 |
| JExcelAPI | 6 |
| JUnit | 3 |
| weka | 127 |
| **总计** | **559** |

可以看出，收集样本数据的过程耗时颇久。为从 12 个开源项目中提取分类器的训练集，我们累计花费了 559 分钟，平均每个项目需要 46.6 分钟。其中，耗时最多的两个项目 android-backup-extractor（LOC＝304,458）与 weka（LOC＝444,493）花费了超过总时长一半（56.9%＝318÷559）的时间。与耗费了近 10 个小时的训练集生成过程相比，神经网络在 12 个开源项目上的训练过程仅需 46 秒。在 13 个开源项目上的测试过程中耗时 36 分钟 46 秒（平均每个项目 2 分钟 50 秒），其中 36 分钟 24 秒均用于从测试项目中提取输入信息，占了总测试时间的 99.1%。但是训练集的自动生成是可以提前准备的，并不需要在每次使用该方法进行上帝类检测的时候再次生成训练集。

### 4.5 有效性威胁

对于评估有效性的第一个威胁在于用来验证实验结果的数据集中只包含了 25 个开源项目。这些项目的某些特性可能会使结论产生偏差，从而导致所得结论不适用于其他项目。为了减少这一威胁，我们选择的训练及测试项目均出自不同的研究领域及开发人员，以期减少某些项目间的特定关联对于验证结果造成影响。

其次，对于评估有效性的第二个威胁在于评估所用的测试集中被人工标注的上帝类代码坏味个数普遍较少。在这样的数据集上进行上帝类检测时，很容易出现检测效果两极分化的情况。为了尽可能保证所得出的结论适用于实际应用，我们选择了符合实际开发过程中上帝类出现情况与分布概率的标签数据集作为结果验证的基准。在现实开发场景中，项目中出现上帝类代码坏味的概率平均保持在 1.7%左右。

对于评估有效性的第三个威胁在于所用的开源代码坏味数据集中的上帝类坏味是由开发人员手动标注的。人工标注虽然更符合实际应用场景，但难以避免数据集贡献者在标记时由于个人主观原因而出现判断误差。为了减少这一威胁，我们选择了各类代码坏味数据集中研究人员最常用的开源数据集，希望通过使用经过反复验证的数据集可以更有效的避免主观性标注所带来的影响。

## 5 讨论

### 5.1 样本噪音

由于所提方法中的样本生成方法（详见 3.5 小节）是建立在所涉及的各个类均符合单一职责原则的假设上的，即假设用于构建训练集正负样本的全部类均不存在上帝类代码坏味，且均可独立承担其作为一个类的职责。然而实际上，这样的假设也许并不成立。我们用以构造训练集的项目源码中是可能出现上帝类代码坏味的。因此这样的样本生成方式可能会导致所生成的训练集中出现噪音——即训练集中有些样本的标签是错误的。为了减少这种噪音对于分类器训练结果的影响，我们尽可能地选取了高质量的项目源码用以构造训练集，以尽量避免设计不够完善的类出现在样本集合中。同时，为了增加神经网络的鲁棒性，在神经网络的设计过程中，我们选择了在自然语言处理领域获得很大成功的长短时记忆网络，并辅以一些防过拟合手段来帮助减少样本噪音对于训练结果的干扰。

### 5.2 文本预处理

所提方法在对神经网络的文本输入进行预处理时（详见 3.3 小节），为了将从源码中提取出的标识符以词向量的形式输入至神经网络，我们会以大写字母和下划线为分隔符来对标识符执行分词操作。这种分词方式对于代码中的大部分标识符（如 AbstractMethodFragment，text_length）是有效的，但对于由多个大写字母连接组成的词语（如 XMLReader）则并不能完全按照词语语义进行划分。针对这一情况，我们对大量的开源工程源码文本进行类似的分词操作，并将分词后的结果作为 Word2Vector 训练的语料库。在这样的语料库下训练出的 Word2Vector 模型隐含层将既可以为语料库中已出现的词语在高维空间中映射词向量，同样可以为语料库中连续出现的字母序列映射空间距离相近的高维向量。此外，标识符中的缩写词（如，numberOfChildren 缩写为 numOfChildren）也可能会导致标识符语义上的不准确。为此，我们在对 Word2Vector 的训练语料库的选取过程中遵循了大规模和高质量两个原则，从而尽量保证完整收集常见缩写词并且避免非常规缩写词的出现。

### 5.3 过拟合问题

在神经网络的应用过程中，过拟合是最常见的影响网络模型泛化能力的原因之一[32]。对此，研究人员提出了降低模型复杂性[32]，添加正则惩罚项（Regularization）[33]，增加随机噪声（Noise）[34]，随机删除隐层神经元（Dropout）[35]以及选择适合的迭代训练次数（Early Stopping）[36]等多种方法以提高神经网络的泛化性能。由于本文所提方法中的神经网络模型参数数量较大，我们也在模型训练过程中加入了一些手段来降低过参数化对网络模型的影响。在尽量减少不必要的神经元和神经网络隐层的同时，我们尝试了上述的多种防过拟合手段，最终确定了目前的整体模型。首先，经过实验对比，我们发现相较于添加高斯噪声和正则化参数，随机删除隐藏层内的一部分神经元对于所提分类器最终测试效果的提升更为明显。在实际训练过程中，我们将Dropout 的丢弃率设为 60%。其次，我们以训练样本集中 10%的随机样本集作为验证集，在每轮迭代后对权重更新后的网络进行验证，发现经过 10 次迭代训练后，神经网络的训练结果进入相对稳定阶段，且验证结果数据没有与训练结果数据出现大的偏差，由此确定了神经网络训练的迭代次数为 10 次。通过上述消除过拟合的方法和手段，我们最终显著提高了神经网络分类器的泛化能力，所提方法在测试集上的平均 F1 值由之前的0.18%提高至了如上文所述的 8.17%。

## 6 总结

在本文中，我们提出了一种基于深度学习的上帝类代码坏味检测方法。通过分析大量源码的文本信息与软件度量信息，深度神经网络在反复迭代中学习从输入中提取出与上帝类坏味相关的特征，最终生成分类器模型。就目前所知，尚未有研究者如我们所做，将深度学习应用于上帝类代码检测。为了满足深度神经网络训练过程对于大规模样本集的需求，我们还实现了一种可以自动利用开源项目源码来构建上帝类样本的方式来生成大量训练数据的工具。

为保证实验评估结果的精准可靠，我们利用开源代码坏味数据集来对所提方法进行验证。我们从 12 个开源项目中提取训练样本以训练神经网络模型，并在 13 个人工标记的开源项目上对所生成的网络模型进行测试。实验结果表明，相对于现有方法，本文所提上帝类检测方法在测试集上的综合表现更佳，具体体现为 F1 值平均提升 2.39%，查全率和查准率分别提高 35.58%和 1.20%。

可以看出，尽管所提方法在测试集上的查准率较现有方法已有提高，但总体来说依旧偏低，导致所提方法可能并不适合直接应用于实际开发场景的完全自动化检测。然而得益于本文所提方法的查全率有了较大幅度的提高，上帝类的检出成功率得到了更高的保障，被检测程序中的潜在上帝类坏味中的大多数都可由此方法检测出来。因此本方法可用于实际开发中的上帝类代码坏味辅助检测，通过提供上帝类坏味候选清单，来帮助开发人员缩小人工检测的范围，从而更快地锁定上帝类代码坏味的重构时机。同时，在未来的研究中，我们将针对上帝类代码坏味检测的可用性做进一步改进，以实现完全自动化的上帝类坏味检测。此外，在本文所提的代码坏味检测的模型基础上，我们也将对基于深度学习的提取类重构操作进行进一步的研究。

上帝类代码坏味属于类级别的代码坏味，因而在本文中我们的训练样本自动生成工具的合并粒度也在类级别上。同时由于上帝类是因违反了单一职责原则而引发的代码坏味，因此我们为模拟生成上帝类代码坏味，实现了为单个类中注入多个职责的自动化工具，以此来实现上帝类代码坏味标签样本的自动生成。此外，在开发人员人工判定上帝类代码坏味的过程中，相对于检测数据类等坏味会更偏向于对代码规模和类内内聚度的考察，而相对于长方法等坏味则更偏向于对代码耦合度的考量，因此我们综合了各种与上帝类代码坏味相关的结构特征，以上文中的 12 个度量项作为神经网络分类器的代码结构特征输入。

然而，由于不同的代码坏味的特点不同，在坏味检测过程中所需的文本特征和结构特征也不同。针对除上帝类以外的其他代码坏味，我们可以基于所提方法的框架模式，通过替换各类坏味所特有的特征表现以及坏味合成方式来实现不同的基于深度学习的坏味检测方法。例如，与上帝类代码坏味在类级别上过于臃肿不同，长方法属于方法级别的代码坏味。因此，若要在本文所提方法的基础上实现长方法代码坏味的深度学习检测方法，我们可以通过批量调用方法内联（inline method）重构操作来实现长方法坏味样本集的自动构建，

并以与长方法相关的几项度量值作为二分类神经网络的输入特征，来实现针对长方法代码坏味的深度学习检测方法。结合以上分析，我们可在未来以本文所提方法模式为基础来实现对数据类，长方法等其他代码坏味在不同程序语言中的深度学习检测方法。

**References**:


[1] Opdyke WF. Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign, 1992.

[2] Mens T, Tourwe T. A survey of software refactoring. IEEE Trans. on Software Engineering, 2004, 30(2): 126-139.

[3] Liu H, Li GJ. Research on Software Refactoring. Beijing Institute of Technology Press, 2016(in Chinese)

[4] Fowler M, Wrote; Hou J, Xiong J, Trans. Refactoring: improving the design of existing code. Boston: Addison-Wesley Longman Publishing Co. Inc. 1999(in Chinese).

[5] Fontana FA, Braione P, Zanoni M, et al. Automatic detection of bad smells in code: An experimental assessment. The Journal of Object Technology, 2012, 11(2): 1-38.

[6] Marinescu C, Marinescu R, Mihancea PF, et al. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In: Proc. of the 21st IEEE Int' | Conf. on ICSM, 2005: 77-80.

[7] Moha N, Gueheneuc Y, Duchien L, et 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.

[8] Fokaefs M, Tsantalis N, Chatzigeorgiou A, et al. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In: Proc. of the 23rd IEEE Int' | Conf. on Software Maintenance, 2007: 519-520.

[9] Lanza M, Marinescu R, Ducasse S. Object-Oriented Metrics in Practice. Secaucus: Springer-Verlag New York, Inc. 2005.

[10] Fokaefs M, Tsantalis N, Stroulia E, et al. JDeodorant: identification and application of extract class refactorings. In: Proc. of the 33rd Int' | Conf. on Software Engineering, 2011: 1037-1039.

[11] Jiang DX, Ma PJ, Su XH, Wang TT. Related Work Analysis of Code Bad Smell Detection and Refactoring. Intelligent Computer and Applications, 2014, 4(3):23-27 (in Chinese).

[12] Zhang M, Hall T, Baddoo N. Code Bad Smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice, 2011, 23(3): 179-202.

[13] Dallal JA. Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information & Software Technology, 2015, 58(58): 231-249.

[14] Nucci DD, Palomba F, Tamburri DA, et al. Detecting code smells using machine learning techniques: Are we there yet. In: Proc. of the 25th Int' | Conf. on Software Analysis Evolution and Reengineering, 2018: 612-621.

[15] Brown WH, Malveau RC, McCormick Iii HW, et al. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley & Sons, Inc. 1998.

[16] Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In: Proc. of the 12th European Conf. on Software Maintaince and Reengineering, 2008: 329-331.

[17] Tsantalis N, Chatzigeorgiou A. Identification of Extract Method Refactoring Opportunities.In: Proc. of the 13th European Conf. on Software Maintaince and Reengineering, 2009: 119-128.

[18] Zhong LH, Zhang NW, Hou CY, Zong HY. Impoved software refactoring method based on hier-archical clustering algorithm. Computer Engineering and Applications, 2015, 51(20):50-54 (in Chinese).

[19] Kreimer J. Adaptive Detection of Design Flaws. Electronic Notes in Theoretical Computer Science, 2005, 141(4): 117-136.

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

[21] Maiga A, Ali N, Bhattacharya N, et al. SMURF: A SVM-based Incremental Anti-pattern Detection Approach.In: Proc. of the 19th Working Conference on Reverse Engineering, 2012: 466-475.

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

[23] Ma S, Dong D. Detection of Large Class Based on Latent Semantic Analysis. Computer Science, 2017, 44(s1):495-498 (in Chinese).

[24] Fontana F A, Zanoni M, Marino A, et al. Code Smell Detection: Towards a Machine Learning-Based Approach. In: Proc. of the 2013 IEEE Int' | Conf. on Software Maintenance, 2013: 396-399.

[25] Arnaoudova V, Eshkevari LM, Penta MD, et al. REPENT: Analyzing the Nature of Identifier Renamings. IEEE Trans. on Software Engineering, 2014, 40(5): 502-532.

[26] Mikolov T, Sutskever I, Chen K, et al. Distributed Representations of Words and Phrases and their Compositionality. Neural Information Processing Systems, 2013: 3111-3119.

[27] Mikolov T, Chen K, Corrado G, et al. Efficient Estimation of Word Representations in Vector Space. Computation and Language, 2013.

[28] Allamanis M, Barr E T, Bird C, et al. Suggesting accurate method and class names. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015: 38-49.

[29] Hochreiter S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 1998, 6(2): 107-116.

[30] Palomba F, Bavota G, Penta MD, et al. On the diffuseness and the impact on maintainability of code smells : A large scale empirical investigation. Empirical Software Engineering, 2017: 1-34.

[31] Khomh F, Penta M D, Gueheneuc Y, et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering, 2012, 17(3): 243-275.

[32] Reed R. Pruning algorithms-a survey. IEEE transactions on Neural Networks, 1993, 4(5): 740-747.

[33] Girosi F, Jones M, Poggio T. Regularization theory and neural networks architectures. Neural computation, 1995, 7(2): 219-269.

[34] An G. The effects of adding noise during backpropagation training on a generalization performance. Neural computation, 1996, 8(3): 643-674.

[35] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 2014, 15(1): 1929-1958.

[36] Sjöberg J, Ljung L. Overtraining, regularization, and searching for minimum in neural networks. Adaptive Systems in Control and Signal Processing 1992. 1993: 73-78.

**附中文参考文献**:

[3] 刘辉, 李光杰. 软件重构技术研究. 北京理工大学出版社, 2016.

[11] 姜德迅, 马培军, 苏小红,等. 代码坏味检测及重构的现状分析. 智能计算机与应用, 2014, 4(3):23-27.

[18] 钟林辉, 张能伟, 侯长源,等. 一种改进的基于层次聚类的软件重构技术研究. 计算机工程与应用, 2015, 51(20):50-54.

[23] 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.