软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1551−1568 [doi: 10.13328/j.cnki.jos.006548]

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

基于预训练模型和多层次信息的代码坏味检测方法*

张 杨[1], 东春浩[1], 刘 辉[2], 葛楚妍[1]

[1](河北科技大学 信息科学与工程学院, 河北 石家庄 050018)

[2](北京理工大学 计算机学院, 北京 100081)

通信作者: 张杨, E-mail: zhangyang@hebust.edu.cn

摘 要: 目前已有的代码坏味检测方法仅依赖于代码结构信息和启发式规则, 对嵌入在不同层次代码中的语义信息关注不够, 而且现有的代码坏味检测方法准确率还有进一步提升的空间. 针对该问题, 提出一种基于预训练模型和多层次信息的代码坏味检测方法 DeepSmell, 首先采用静态分析工具提取程序中的代码坏味实例和多层次代码度量信息, 并对代码坏味实例进行标记; 然后通过抽象语法树解析并获取源代码中与代码坏味相关的层次信息, 将其中的文本信息与度量信息相结合生成数据样本; 最后使用 BERT 预训练模型将文本信息转化为词向量, 应用 GRU-LSTM 模型获取层次信息之间潜在的语义关系, 并结合 CNN 模型与注意力机制检测代码坏味. 在实验中, 选取 JUnit、Xalan 和 SPECjbb2005 等 24 个大型实际应用程序构建训练集和测试集, 并对特征依恋、长方法、数据类和上帝类等 4 种代码坏味进行检测. 实验结果表明, DeepSmell 与目前已有的检测方法相比在平均查全率和 *F*1 值上分别提高了 9.3% 和 10.44%, 同时保持了较高的查准率, DeepSmell 可以有效地实现代码坏味检测.

关键词: 代码坏味; 深度学习; 预训练模型; 抽象语法树; 多层次信息

中图法分类号: TP311

中文引用格式: 张杨, 东春浩, 刘辉, 葛楚妍. 基于预训练模型和多层次信息的代码坏味检测方法. 软件学报, 2022, 33(5): 1551–1568. http://www.jos.org.cn/1000-9825/6548.htm

英文引用格式: Zhang Y, Dong CH, Liu H, Ge CY. Code Smell Detection Approach Based on Pre-training Model and Multi-level Information. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1551–1568 (in Chinese). http://www.jos.org.cn/1000-9825/6548.htm

Code Smell Detection Approach Based on Pre-training Model and Multi-level Information

ZHANG Yang[1], DONG Chun-Hao[1], LIU Hui[2], GE Chu-Yan[1]

[1](School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050018, China)

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

Abstract: Most of the existing code smell detection approaches rely on code structure information and heuristic rules, while pay little attention to the semantic information embedded in different levels of code, and the accuracy of code smell detection approaches is not high. To solve this problem, this study proposes a novel approach DeepSmell based on a pre-trained model and multi-level metrics. Firstly, the static analysis tool is used to extract code smell instances and multi-level code metric information in the source program and mark these instances. Secondly, the level information that relate to code smells in the source code are parsed and obtained through the abstract syntax tree. The textual information composed of the level information is combined with code metric information to generate the data set. Finally, text information is converted into word vectors using the BERT pre-training model. The GRU-LSTM model is applied to obtain the potential semantic relationship among the identifiers, and the CNN model is combined with attention mechanism to code smell detection. The experiment tested four kinds of code smells including feature envy, long method, data class, and god class on 24 open

* 基金项目: 国家自然科学基金 (62172037); 河北省自然科学基金重点项目 (18960106D); 河北省高等学校科学研究计划重点项目 (ZD2019093)

本文由“领域软件工程”专题特约编辑汤恩义副教授、江贺教授、陈俊洁副教授、李必信教授以及唐滨副教授推荐.

收稿时间: 2021-08-06; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

source programs such as JUnit, Xalan, and SPECjbb2005. The results show that DeepSmell improves the average recall and *F*1 by 9.3% and 10.44% respectively compared with existing detection methods, and maintains a high level of precision at the same time.

Key words: code smell; deep learning; pre-trained model; abstract syntax tree; multi-level information

代码坏味会导致软件开发过程减缓并且增加失败的风险, 使软件难以理解、维护和演化[1], 程序员往往需要不断修改源代码以适应新的需求或修复软件系统中的设计缺陷. 为了更好地描述软件系统中的设计缺陷问题, Fowler 等人[2]对影响软件质量的设计问题进行定义, 提出了代码坏味的概念. 目前研究人员提出应用重构技术在不改变代码外部行为的前提下消除代码坏味, 代码坏味检测已成为发现代码设计缺陷并确定重构位置的常用方法[3].

目前研究人员已经设计了多种代码坏味的检测工具, 比较著名的有 JDeodorant[4]、iPlasma[5]和 Fluid [6]等, 这些检测工具大多采用基于启发式规则的方法, 在识别代码坏味时单纯依赖代码度量信息或开发人员定义的规则, 由于不同工具选取的阈值各不相同, 导致代码坏味检测结果假阳性率过高. 为了避免阈值选择问题和降低假阳性率, 很多研究人员开始采用机器学习方法来解决启发式规则方法的局限性. Fontana 等人[7]对多种机器学习方法在检测代码坏味上的性能进行评估, 证实了机器学习方法检测代码坏味的有效性. Fabiano 等人[8]则比较了启发式规则方法与机器学习方法在检测代码坏味方面的性能, 他们认为这两种检测方法都存在一定局限性, 在查准率方面有待进一步提升.

随着深度学习技术的广泛应用, 很多研究人员开始探索将深度学习技术应用于代码坏味检测. 马赛等人[9]将概念性度量与代码圈复杂度结合以对过大类进行检测. 王曙燕等人[10]采用反向传播神经网络分别将方法级和类级的代码坏味进行融合, 对同一数据集中存在不同类型代码坏味进行检测. Liu 等人[11]提出一种自动生成代码坏味的方法来构建代码坏味数据集, 并采用深度学习的方法检测代码坏味. 此外, 一些研究人员从代码坏味严重性[12]、代码坏味相关性[13]和代码坏味重构推荐[14]等方面进行了研究.

虽然现有的方法在代码坏味检测上已经取得一定的成效, 但仍存在以下几个方面的问题亟需进一步完善. (1) 现有的检测方法很少关注嵌入在多层次代码中的语义信息, 在获取文本信息之间的语义关系时通常采用长短期记忆网络 (long short-term memory, LSTM) 模型, 信息之间的交互只在 LSTM 内部进行, 获取上下文之间的潜在语义信息不充分, 不能准确地反映不同层次信息之间的语义关系; (2) 现有的工作中公开的代码坏味数据集较少, 且大多数只包含代码度量信息, 缺少语义等多层次信息; (3) 获取与代码坏味相关的多层次文本信息困难, 过程繁琐且浪费大量的时间; (4) 现有的代码坏味检测方法在查准率方面不是足够高, 还有进一步提升的空间.

针对目前研究存在的问题, 本文提出一种基于预训练模型和多层次信息的代码坏味检测方法 DeepSmell. 该方法首先采用静态分析工具从源程序中提取代码坏味实例和多层次的代码度量信息, 并对代码坏味实例进行标记. 其次, 为了实现文本信息的自动获取, 设计了自动提取多层次信息的工具, 通过解析抽象语法树 (abstract syntax tree, AST) 来获取源代码中与代码坏味相关的信息, 将多层次信息组成的文本信息与度量信息相结合生成数据样本. 最后采用 BERT 预训练模型将文本信息转化为词向量, 应用 GRU-LSTM 方法获取文本信息之间嵌入的语义关系, 并结合卷积神经网络 (convolutional neural network, CNN) 模型与注意力机制对代码坏味进行检测. 为了评估 DeepSmell 的有效性, 我们选取 JUnit、Xalan 和 SPECjbb2005 等 24 个大型实际程序构建训练集和测试集, 并对特征依恋、长方法、上帝类和数据类等代码坏味进行检测. 实验结果表明与目前已有的检测方法相比, DeepSmell 在平均查全率和 *F*1 值上分别提高了 9.3% 和 10.44%, 同时保持了较高的查准率, DeepSmell 可以有效的实现代码坏味检测.

本文主要贡献包括:

(1) 选取 24 个不同领域的大型实际应用程序作为语料库, 提取每个应用程序的代码度量信息和文本信息, 构建特征依恋、长方法、上帝类和数据类等 4 种代码坏味数据集.

(2) 提出一种基于深度学习的代码坏味检测模型 DeepSmell, 设计了一个能够自动获取源程序中不同层次信息的工具, 通过 BERT-GRU-LSTM 相结合的文本信息表示方法获取多层次信息之间的潜在语义关系, 应用 CNN 模型获取代码度量中的特征信息, 将文本信息与度量信息相结合的特征信息, 通过注意力机制与标签做映射, 提高代

码坏味检测的查准率.

(3) 将 DeepSmell 与现有的 4 种基于深度学习的代码坏味检测方法进行比较, 验证了 DeepSmell 的有效性.

本文第 1 节介绍相关研究的现状, 并对此进行总结与分析. 第 2 节介绍本文提出的代码坏味检测方法. 第 3 节对所提出方法进行验证与评估. 第 4 节进行总结和未来工作的展望.

1 相关工作

目前, 关于代码坏味检测方法的研究主要分为 3 类: 基于规则的方法、基于机器学习的方法和基于深度学习的方法.

1.1 基于规则的检测方法

基于规则的方法单纯依赖代码中的度量信息或者依靠有经验的专家定义的启发式规则识别特定的代码坏味. 王莹等人[15,16]对现有的代码坏味检测方法进行总结, 提出一种加权聚类的方法级网络软件自动重构方法. Moha 等人[17]提出一种 DECOR 方法, 定义了检测代码坏味的规范, 采用一组特定的规则描述受代码坏味影响类的内部特性. Sales 等人[18]提出一种基于依赖的方法 JMove, 通过比较依赖的相似性来推荐移动方法的重构机会. Tsantalis 等人[3]开发了 JDeodorant 坏味检测工具, 采用杰卡德距离计算两个实体之间的相似性来检测特征依恋坏味, 并推荐方法可移动的重构机会. 经过改进, 增加了对其他 4 种代码坏味 (重复代码、switch 语句、长方法和上帝类) 推荐重构的机会. Fard 等人[19]提出了一种基于动态语言的代码异味检测工具 JSNose, 利用面向对象语言的一些指标来检测 JavaScript 语言中的代码坏味. Fernandes 等人[20]对代码坏味检测工具进行了系统的总结, 通过基于定量和定性的数据讨论了相关的可用性问题, 为开发人员设计代码坏味检测工具提出了指导方针[21,22]. 然而, 开发人员的主观性影响了代码坏味检测工具的设计, 不同工具之间的选择阈值也缺乏一致性.

1.2 基于机器学习的检测方法

为了解决基于启发式规则方法的局限性, 各种机器学习算法用来检测代码坏味, 例如: Naive Bayes[23]、SVM[24]和 Decision Tree[25]等. Amorim 等人[26]将决策树算法与人工 oracle、现有的检测方法和其他机器学习算法在一个包含 4 个开源项目的数据集上进行了比较, 证实了决策树算法识别代码坏味具有更好的性能. Nucci 等人[27]指出以前的工作中只包含一种代码坏味数据集不能代表真实的场景, 通过对类级和方法级数据进行结合来生成符合真实场景的数据集, 揭示了当前技术的局限性. Guggulothu 等人[28]分析了 Nucci 工作中性能下降原因, 采用多标签分类来检测代码是否受到多种坏味的影响. Khomh 等人[23]提出了 BDTEX, 一种基于目标问题度量的方法. 从反模式的定义构建贝叶斯信念网络, 通过 3 种反模式来验证 BDTEX 在检测上帝类时的性能. Maiga 等人[29]提出了一种新的基于机器学习技术的反模式检测方法 SVMDetect. 基于涉及 3 个主题系统的数据集上进行验证, 在检测一组类级别代码坏味时, SVMDetect 的性能优于 DETEX. 王继娜等人[30]提出一种基于排序损失的集成分类器链多标签代码坏味检测方法, 将随机森林作为基础分类器并采取多次迭代 ECC 的方式, 优化代码坏味检测顺序问题. 现有的工作[7]研究表明, 机器学习算法在检测代码坏味上仍存在一定的局限性.

1.3 基于深度学习的检测方法

近几年一些研究人员开始将深度学习方法用于检测代码坏味, 它可以从提取的特征中学习来识别代码坏味. Liu 等人[31,32]通过自动生成类级别和方法级别的代码坏味生成数据集, 采用文本信息和度量信息作为分类器输入, 对代码坏味进行检测. Ananta 等人[33]采用 CNN 模型对大脑类和大脑方法进行了检测. 郭学良等人[34]提出一种文本信息的表示方法来获取潜在语义信息, 并与 CNN 模型相结合来检测特征依恋坏味. Singh 等人[35]提出基于功能相关性和语义相关性检测长方法和长链表. 房春荣等人[36]通过提取源代码的语法和语义信息作为模型输入, 应用图神经网络对克隆代码进行检测. 刘弋等人[37]利用图结构建立类间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, 以平均值和比例因子的乘积作为阈值筛选上帝类. 目前, 采用深度学习技术检测代码坏味处于探索阶段, 上述工作虽然取得较好的效果, 但基于深度学习的代码坏味检测尚需做进一步研究.

2 代码坏味检测方法

为了提高检测代码坏味的查准率, 本文提出了一种基于预训练模型和多层次信息的代码坏味检测方法. 在第 2.1 节给出本文所提出方法的框架图, 第 2.2–2.7 节对该方法的各个关键部分进行介绍.

2.1 方法概述

基于深度学习的代码坏味检测方法 DeepSmell 框架图如图 1 所示. 首先, 选取 24 个实际应用程序作为语料库, 采用静态分析工具提取源程序中的代码坏味实例和多层次代码度量信息, 并对代码坏味实例进行标记; 然后通过抽象语法树解析并获取源代码中与代码坏味相关的层次信息, 将其中的文本信息与度量信息相结合生成数据样本, 以此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 (即是否为代码坏味). 经过多次在训练集上训练, 最终获取训练好的分类器, 通过测试集测试分类器的性能, 并给出代码坏味的检测结果.

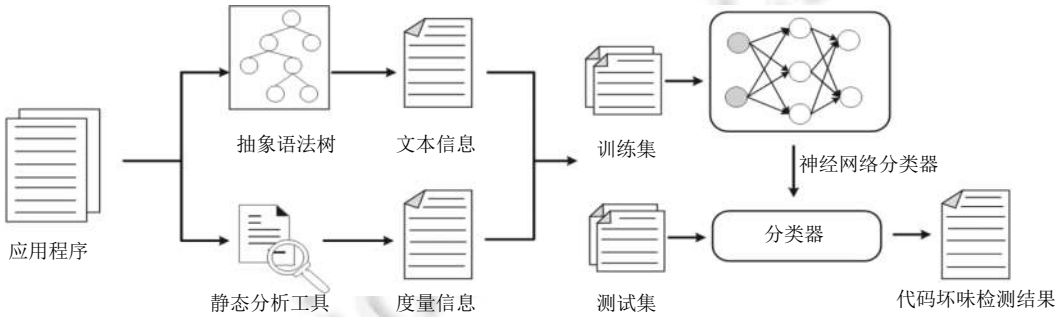


图 1 基于深度学习的代码坏味检测方法框架图

2.2 输入的生成

本节介绍如何提取源程序中的文本信息与度量信息, 从而生成深度学习模型的输入.

2.2.1 方法级代码坏味输入

对于检测特征依恋和长方法两种方法级别的代码坏味, 选取项目、包、类、方法等 4 种不同层次的代码度量项作为模型的度量信息输入. 代码度量项的详细信息如表 1 所示, 总计选取了 32 种不同级别的度量项, 其中, 项目级度量 2 种、包级度量 2 种、类级度量 13 种、方法级度量 15 种, 每种不同级别的度量信息都包含多种特征信息, 不同级别的度量信息之间存在包含关系.

源程序中很多代码的命名方式都基于它的功能, 因此嵌入在不同层次中的语义信息可能与代码坏味相关联. 此外不同层次信息之间在语义关系上也存在某种联系, 因此我们通过获取源程序中项目、包、类和方法等多层次信息作为模型的文本信息输入. 文本信息主要包含项目名、包名、类名、方法名等组成的长单词序列. 本文方法级别的输入包含文本信息和度量信息两部分, 如公式 (1) 所示.

$$\left.\begin{aligned} &input=\langle text\_input, metric\rangle\\ &text\_input=\langle name(project), name(package), name(class), name(method)\rangle\\ &metric=\langle metric_1, metric_2, \ldots, metric_n\rangle \end{aligned}\right\} \tag{1}$$

其中, $input$ 表示分类器输入, $text\_input$ 表示文本信息输入, $metric$ 表示度量信息输入, $name(project)$、$name(package)$、$name(class)$、$name(method)$ 分别为项目、包、类、方法的名称, $metric_i$ 表示第 i 个度量项, n 为度量项个数.

2.2.2 类级代码坏味输入

对于上帝类和数据类两种代码坏味, 选取 13 种类级代码度量项作为分类器的度量信息输入.

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, 因此, 一个类中的多个成员之间应该存在着语义上的相互关联[31], 这里选取了类名及类中所有方法名组成文本信息作为模型的文本信息输入. 类级代码坏味的分类器输入如公式 (2) 所示:

$$\left.\begin{aligned} &input = \langle text\_input, metric \rangle \\ &text\_input = \langle name(class), name(m_1), name(m_2), \ldots, name(m_n) \rangle \\ &metric = \langle metric_1, metric_2, \ldots, metric_n \rangle \end{aligned}\right\} \tag{2}$$

其中, m_i 为被检测类中声明的第 i 个方法名, 其余符号表示同公式 (1).

表 1 度量项

名称	度量项	说明	名称	度量项	说明
项目级	LOC	项目代码行数	包级	NOC	类的数量
	NOM	方法数量		NOM	方法数量
类级	NOAM	类内的 get/set 方法个数	方法级	NOP	参数个数
	NOM	类内方法个数		CC	改变的类
	ATFD	被检测类所访问的外部类属性个数		ATFD	被检测方法所访问的外部类属性个数
	FANOUT	–		FDP	外部访问数据数量
	NOA	属性数		CM	改变的方法
	DIT	被检测类距离其所在继承关系树根节点的最大层数		MAXNESTING	最大嵌套级别
	LOC	类代码行数		LOC	方法代码行数
	TCC	类中通过访问相同属性而发生关联的方法对个数		CYCLO	圈复杂性
	NOPA	公共属性的数量		NOLV	局部变量个数
	CBO	对象类之间的耦合		NOAV	访问的变量数
	WMC	类的圈复杂度		LAA	属性访问的局部性
	WOC	加权方法数量		FANOUT	–
	AWM	平均方法权重		CINT	耦合强度
				CDISP	–
				isStatic	是否为静态方法

2.3 数据预处理

首先根据获取的文本信息进行数据预处理, 依据驼峰命名法和下划线规则对文本信息进行分词, 每个长单词序列由 4 种不同层次信息组成. 其次, 对每个长单词序列进行数据清洗, 去除层次信息中的特殊符号和无用字符, 将处理后的单词序列进行 token 化处理生成 token 序列, 并计算每个单词序列的长度. 最后通过 BERT 预训练模型词表将 token 序列转化成词向量.

通过对文本信息中的每个长单词序列进行统计发现 96% 以上的长单词序列长度均小于 60 维. 为了获取多层次信息之间的语义关系, 本文设置词向量映射维度为 60 维. 对于没有达到 60 维的序列, 采用 padding 处理全部以 0 进行填充达到 60 维, 对于超过 60 维的序列只截取前 60 个词向量作为多层次信息之间潜在语义关系的表示.

2.4 文本信息表示

LSTM 一直被用来捕获文本信息中多层次信息之间的潜在语义关系, 通过直接将当前时刻的信息和隐藏状态信息作为 LSTM 的输入, 信息之间的交互只在 LSTM 内部进行, 信息之间的交互不充分, 获取的语义信息不能充分反映多层次信息之间的潜在语义关系. 为了能够更加充分挖掘文本信息中的潜在语义关系, 本文提出一种文本语义信息的表示方法. 该方法包含 BERT[38]词嵌入层、门控循环单元 (gate recurrent unit, GRU)[39]层和 LSTM 层, 如图 2 所示.

现有的工作大多采用 Glove[40]和 Word2Vec[41]等词嵌入模型, 虽然在一定程度上考虑了上下文的信息, 但无法获取同义词之间的相关性, 也很难计算词之间的相似度. BERT 预训练模型能够充分融合词之间的上下文信息, 可以更加充分的表示词的多义性. 在图 2 中应用 BERT 模型将文本信息中的长单词序列映射为固定维度词向量, 以向量之间的相似性反映多层次信息之间的语义相关性. 将 BERT 模型映射出的词向量作为 GRU 层的输入, GRU 单元内部包含重置门和更新门, 通过重置门定义隐藏信息与输入信息结合的量, 将重要的信息传输到更新门, 更新

门定义当前时间步保留隐藏信息的量, 得到 GRU 单元的输出结果. 通过引入额外的 GRU 层来增加信息之间的交互, 更好的获取上下文之间的语义信息, 将 GRU 的输出视为新的输入和新的隐藏状态作为 LSTM 的输入. 计算过程如公式 (3) 所示:

$$\left.\begin{aligned} r &= \sigma\left(\omega\left[g^{t-1}, V(W_t)\right]+b\right) \\ z &= \sigma\left(\omega\left[g^{t-1}, V(W_t)\right]+b\right) \\ g' &= \tanh\left(\omega\left[g^{t-1}\odot r, V(W_t)\right]+b\right) \\ g^t &= (1-z)\odot g^{t-1} + z\odot g' \end{aligned}\right\} \tag{3}$$

其中, r、z、σ、$V(W_t)$、tanh、$\odot$ 、ω 分别表示重置门、更新门、Sigmoid 激活函数、层次信息映射的词向量、tanh 激活函数、乘法运算符、权重矩阵; g^{t-1} 、g' 和 g^t 分别表示前一时刻传输的状态、重置之后的状态和当前时刻的状态.

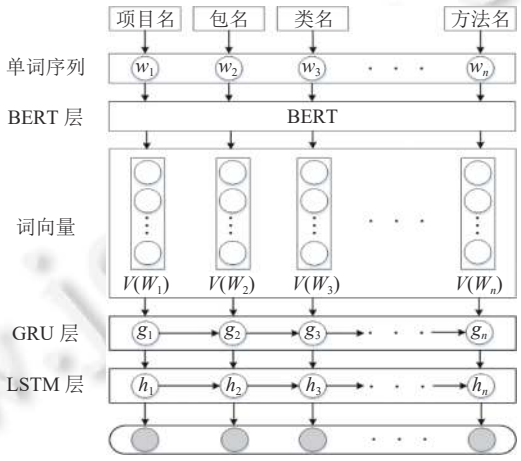


图 2 文本信息表示方法

LSTM 层通过输入门、遗忘门、输出门来控制内部的状态流, 通过前一单元的状态和隐藏状态来更新当前时间步单元的状态和隐藏状态. 输入门控制每个单元更新的信息量, 遗忘门控制前一个单元遗忘的信息量, 输出门控制内部存储状态应该输出的信息量. 输入和隐藏的特征维度均设置为 256, 计算过程如公式 (4) 所示. 通过采用 GRU-LSTM 相结合的方法增加文本信息中层次信息之间的交互, 更好的表示这些层次信息之间潜在的语义关系.

$$\left.\begin{aligned} c_t' &= \tanh\left(\omega\left[h_{t-1}, Y(g)\right]+b\right) \\ h_t &= o_t\tanh\left(f_t\odot c_{t-1}' + i_t\odot c_t'\right) \end{aligned}\right\} \tag{4}$$

其中, i_t, f_t, o_t 分别表示输入门、遗忘门和输出门, 计算方法与 GRU 中重新门和更新门相同. c_t' 表示当前单元状态, c_{t-1}' 表示 c_t' 前一时刻单元状态, b 表示可训练参数. $Y(g)$ 表示 GRU 层的输出作为 LSTM 的输入, h_t 表示存储单元的输出信息.

2.5 度量特征表示

对于从源程序中获取的代码度量信息作为分类器的度量信息输入, 采用 CNN 模型来获取代码度量信息中的深层次特征. CNN 模型由输入层、卷积层、最大池化层和全连接层组成, 卷积核大小为 1, CNN 滤波器为 128, 卷积方式为 same, 步长为 1. 通过将卷积层和最大池化层相结合的方法, 逐层提取特征信息. ReLU 函数增加各层之间的非线性关系, 最大池化层降低特征维度来提高运算效率, Dropout 层来减少隐藏节点的相互作用, 防止过拟合的影响, 通过全连接层不断调整最优的参数组合, 度量特征表示如图 3 所示.

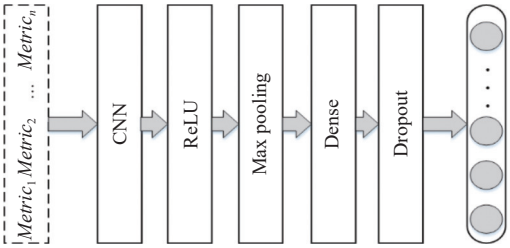


图 3 度量特征表示

2.6 基于深度学习的检测

该检测方法的输入分为两部分: 文本信息输入和度量信息输入. 检测方法结构分为 3 部分: 文本信息方法表示、代码度量信息特征表示、信息融合的代码坏味检测, 深度学习模型结构如图 4 所示.

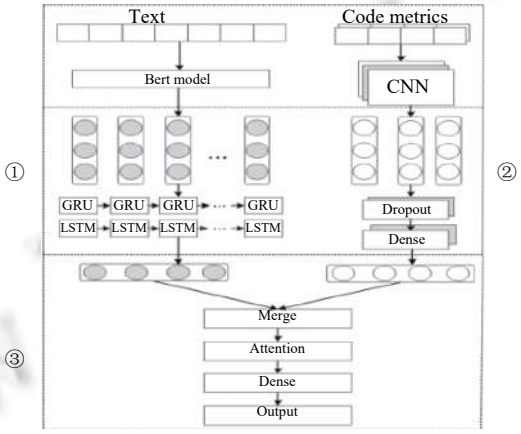


图 4 深度学习模型结构

在①部分中, 将文本信息作为输入信息, 通过 BERT 词嵌入模型将文本信息映射为词向量, 使用 GRU-LSTM 结合的方法充分获取文本中的语义信息 (见第 2.4 节).

在②部分中, 我们将获取的代码度量信息作为 CNN 模型的输入, 通过 CNN 模型提取代码度量信息中的相应数据特征 (见第 2.5 节).

在③部分中, 将文本特征表示与代码度量特征表示通过合并层 (merge) 进行拼接. 通过注意力机制[42]将特征信息与预期标签做映射, 计算特征信息对检测代码坏味的重要程度, 增加对预测结果重要信息的关注, 抑制其他无用信息, 并对信息重新进行加权求和计算, 为序列中每个特征都重新分配一个权重系数, 再通过全连接层映射到最后的分类.

2.7 数据集生成

收集 24 个开源的大型实际应用程序作为语料库, 这些应用程序及其配置如表 2 所示. 可以看出, 选取的应用程序中总计达到 70 万以上代码行数, 其中 Xalan、HSQLDB、Cassandra 等 13 个程序中代码行数超过 1 万以上; 选取的应用程序共包含了 6880 个类, 其中 Displaytag、Cobertura、Freedomotic 等 15 个程序中包含 100 个以上的类; 包含了 60 000 多个方法, 其中 JBoss、JGroups、RxJava 等 12 个程序中包含 1 000 个以上的方法. 这些应用程序来源于分布式调度框架、Java 虚拟机实现、安全物联网框架等多个领域, 我们通过获取这些应用程序的文本信息和代码度量信息生成代码坏味数据样本, 通过对数据样本进行标记生成 4 种代码坏味的数据集.

为了能够自动获取这些应用程序中的文本信息, 我们设计了一个可以在 AST 上提取多层次信息的工具. 该工具在 Eclipse JDT 框架下开发, 首先将源程序 AST 解析, 将获取的对象存放在 ICompilationUnit 对象中, 通过继承

ASTVisitor 实现一个子类作为具体的访问者遍历所有 AST 节点, 访问的参数类型为 PackageDeclaration、TypeDeclaration、MethodDeclaration, 获取对应类型的多层次信息.

表 2 应用程序

	应用程序	用途	NOC	NOM	LOC
方法级	Displaytag	自定义标记的开源套件	262	945	9 019
	Cobertura	Java代码覆盖率报告工具	165	1 156	14 723
	Freedomotic	安全物联网框架	501	3 867	33 857
	JavaCC	解析器生成器	180	1 487	20 861
	JSmooth	可执行包装器	101	886	10 411
	Freecs	开源测试程序	139	1 404	20 720
	FitJava	开源测试程序	61	456	2 916
	JUnit	单元测试	93	796	5 108
	Xmojo	JMX规范实现	29	266	3 392
类级	JBoss	应用服务器	612	5 269	75 513
	JGroups	群组通讯工具	273	2 235	15 587
	Job	分布式任务调度框架	46	227	1 638
	RxJava	响应式扩展的Java虚拟机实现	736	4 181	41 273
	Xalan	XSLT处理器	968	10 413	171 427
	Jadventure	Java文本游戏	36	145	1 215
	JUnit	单元测试	462	3 960	20 645
	HSQLDB	数据库	548	11 043	190 614
	SPECjbb2005	Java 应用服务器测试	76	747	12 713
	Blueblock	Java文本游戏	13	88	1 178
	Cassandra	分区行存储	1 066	10 640	83 001
	JavaStud	Java示例系列项目	218	459	4 229
	Mmseg4j	Java中文分析器	16	97	716
	Mybaits3	测试代码	224	1 003	5 183
	Redomor	Java文本游戏	55	463	3 359
	总计	—	**6 880**	**62 233**	**749 298**

为了获取源程序中的代码坏味实例, 对于每一种代码坏味, 都选取一组检测工具和规则, 这些检测工具和规则在工作 [34,43,44] 中被广泛应用. 此外, 我们选择了尽可能多的检测规则, 也更加倾向于使用工具实现的检测规则. 表 3 展示了 4 种代码坏味的描述及用于识别代码坏味实例的检测工具和规则. 通过选定的检测规则, 为每个类和方法生成一个标签, 只有同时满足多组特定规则的代码坏味实例才标记为 1 (表示有代码坏味), 其他情况均标记为 0 (表示无代码坏味). 最后, 通过人工检查的方式来确认代码坏味数据标记的正确性, 生成 4 种代码坏味数据集.

表 3 代码坏味描述

代码坏味	描述	检测工具和规则
特征依恋	方法使用大量其他类的成员	iPlasma, Trifu rule[44]
长方法	方法代码过长包含大量函数	iPlasma (大脑方法), PMD, Marinescu rule[43]
数据类	类中仅含公共成员变量或操作的函数	iPlasma, Trifu rule[44]
上帝类	类中包含大量方法和属性	iPlasma (上帝类, 大脑类) , PMD

3 实验验证

所有实验均在 Lenovo 工作站上进行, 该工作站配备 CPU AMD Ryzen 处理器, GPU Nvidia GTX1050Ti, 主频

为 2.1 GHz, 内存为 16 GB. 软件上, 操作系统使用 Windows 10, 使用 Python 3.7 和 PyTorch 1.6 作为深度学习的运行支撑环境. 在实验参数配置中, 迭代次数为 20 次, 选择 Adam 作为优化器, 输出维度为 1, 学习率设置为 0.000 1, 批尺寸 (batch_size) 为 128, 正则化 (dropout) 为 0.5, BERT 模型输入特征维度设置为 768, 隐藏层的特征维度为 256. 对于检测方法级别将的代码坏味, 实验采用 K (K=9) 次验证方法的性能. 对于类级别生成的数据集, 随机划分为 7:3. 通过将 DeepSmell 与现有的基于深度学习的方法在 4 种代码坏味数据集上进行对比, 来验证本文方法的有效性, 表 4 展示了不同方法的具体参数设置, 我们在实验中设置相同的参数来减少这一因素对代码坏味检测结果的影响.

表 4 不同方法的参数设置

方法	网络层	网络层输入	网络层输出
文献[10]	Linear_1	768	128
	Linear_2	12(方法)、32(类)	128
	Linear_3	256	128
	Linear_4	128	2
文献[11]	Conv1d_1	768	256
	Linear_1	256	128
	Conv1d_2	12、32	128
	Linear_2	256	2
LSTM-CNN	LSTM	768	256
	Linear_1	256	128
	Conv1d	12、32	128
	Linear_2	256	2
文献[34]	LSTM	768	256
	Attention	256	256
	Conv1d	12、32	128
	Linear_1	256	2
Wide&Deep[45]	Linear_1	768	32
	Linear_2	32	64
	Linear_3	64	2
DeepFM[46]	Linear_1	768	256
	Linear_2	256	128
	Linear_3	256	2
DeepSmell	GRU	768	256
	LSTM	256	256
	CNN	12、32	128
	Attention	256	256
	Linear_1	256	2

3.1 研究问题

在实验中, 我们提出了 7 个研究问题 (research question, RQ), 通过回答这些问题对 DeepSmell 方法进行评估.

(1) RQ1: DeepSmell 是否能准确有效地检测出特征依恋坏味?其查全率和查准率是否优于现有方法?

(2) RQ2: DeepSmell 是否能准确有效地检测出长方法?其查全率和查准率是否优于现有方法?

(3) RQ3: DeepSmell 在检测类级别坏味上是否适用?其查全率和查准率是否优于现有方法?

(4) RQ4: DeepSmell 在检测代码坏味上, 其查全率和查准率是否优于改进后的 LSTM-CNN 方法?

(5) RQ5: 不同特征信息作为 DeepSmell 输入对代码坏味检测结果有什么影响? 如果只考虑一种特征输入, DeepSmell 的性能会如何?

(6) RQ6: DeepSmell 与其他的深度学习算法在检测代码坏味上性能比较如何?

(7) RQ7: DeepSmell 在检测代码坏味时各部分时间性能表现如何?

RQ1 关注的是 DeepSmell 与现有的方法检测特征依恋坏味上的比较. 为了回答这个问题, 我们选择文

献 [10,11,34] 中的方法作为评估阶段的对比实验对象. 之所以选择这些方法作为参照, 是因为这些都是最新提出的基于深度学习检测代码坏味方法, 是目前在检测代码坏味方面具有代表性的工作.

RQ2 关注的是 DeepSmell 与现有的方法在检测长方法坏味上性能的比较. 我们选择文献 [10,11,34] 中的方法进行评估阶段的对比实验.

RQ3 关注的是 DeepSmell 在检测类级别代码坏味时的适用性. 为了回答这个问题, 我们提取了数据类和上帝类两种代码坏味的数据集作为输入, 在两种数据集上依据查准率、查全率和 *F*1 值等 3 个指标来分析 DeepSmell 和文献 [10,11,34] 中的方法对于最终结果的影响.

RQ4 关注的是对现有的方法进行改进是否优于 DeepSmell. 在获取度量信息时, 将全连接层替换为 CNN 层, 应用 LSTM 层获取多层次信息之间的语义关系, 测试方法是否具有更好的性能, 并与 DeepSmell 在特征依恋、长方法、数据类和上帝类等 4 种代码坏味的数据集上进行测试, 并根据查准率、查全率和 *F*1 值等指标给出实验分析结果.

RQ5 关注的是不同特征信息对代码坏味检测结果的影响, 通过比较文本信息输入、度量信息输入、文本信息和度量信息相结合输入方式下的分类器在特征依恋坏味数据集上的检测结果.

RQ6 关注的是 DeepSmell 与其他的深度学习算法相比, 在检测代码坏味上的性能如何, 我们选择 Wide & Deep [45]和 DeepFM[46]两种深度学习算法进行评估阶段的对比实验.

RQ7 关注的是 DeepSmell 在检测代码坏味上的时间性能, 针对测试数据集上 9 个开源程序, 记录了 DeepSmell 在整个检测特征依恋代码坏味上的耗时情况.

3.2 评估标准

实验通过在测试集上的查准率、查全率、*F*1 值等指标来评估方法的性能. 查准率 (*Precision*) 表示预测结果中, 预测为正样本的样本中, 正确预测为正样本的概率, 计算公式如下:

$$Precision=\frac{TP}{TP+FP} \tag{5}$$

查全率 (*Recall*) 表示在原始样本的正样本中, 最后被正确预测为正样本的概率, 计算公式如下:

$$Recall=\frac{TP}{TP+FN} \tag{6}$$

其中, *TP* 和 *TN* 分别代表样本正类和样本负类, *FP* 表示将错误样本分成正确样本数量, 而 *FN* 表示正确样本分成错误样本数量. 在分类任务中查准率和查全率都达到很高, 但在实际情况中查准率和查全率往往相互影响. 因此, 需要选择两者之间的平衡点, *F*1 值相当于查准率和查全率的加权平均值, 取值范围在 0 到 1 之间, *F*1 值越高说明查准率和查全率同时达到最高取得平衡, 计算公式如下:

$$F1=2\times\frac{Precision\times Recall}{Precision+Recall} \tag{7}$$

3.3 实验结果与分析

本节给出相关实验结果, 并对结果进行了分析.

3.3.1 特征依恋检测结果

为了回答 RQ1, 将 DeepSmell 与现有的基于深度学习的检测方法在 9 个开源程序上进行比较. 实验结果如表 5 和表 6 所示, 其中, 第 1 列展示测试的项目名称, 第 2–7 列显示其他工作中方法测试结果的查准率、查全率和 *F*1 值, 第 8–10 列展示 DeepSmell 测试结果的查准率、查全率和 *F*1 值. 表中最后一行列出了 3 种方法各项指标的平均值.

实验结果表明, DeepSmell 在测试程序上的整体性能最优, 平均查准率、查全率和 *F*1 值分别为 83.65%、84.59%、83.56%, 与相比其他代码坏味检测方法相比, 平均查准率提升了 10.69%(=83.65%−72.96%), 平均查全率提升了 6.2%(=84.59%−78.39%), 平均 *F*1 值提升了 10.11%(=83.56%−73.45%). 此外, 在 Freedomotic 测试程序上 DeepSmell 获得最高的测试性能, 查准率、查全率、*F*1 值分别为 92.35%、93.54%、92.58%; 而在 Xmojo 测试程

序上获得最低的测试性能, 查准率为 73.59%, 查全率 74.80%, $F1$ 值为 72.52%, 性能较差的可能原因是 Xmojo 程序中样本数量较少, 可能导致测试结果不稳定.

表 5 与文献 [10,11] 比较特征依恋检测结果 (%)

项目名称	文献[10]			文献[11]			DeepSmell		
	查准率	查全率	$F1$	查准率	查全率	$F1$	查准率	查全率	$F1$
Displaytag	74.19	81.41	74.08	63.88	36.96	42.43	84.34	85.71	84.43
Coberture	87.30	85.07	78.56	84.88	86.20	85.33	87.52	88.66	87.26
JavaCC	77.73	84.31	77.86	85.65	86.53	86.00	87.46	88.59	87.33
FitJava	54.82	64.40	58.85	58.94	70.59	62.40	78.89	79.26	79.05
JSmooth	85.07	81.72	73.93	69.31	78.78	72.67	82.45	84.20	82.65
JUnit	66.64	74.34	70.03	83.44	81.29	82.12	86.86	87.53	87.05
Xmojo	47.49	59.84	52.31	77.07	77.56	75.62	73.59	74.80	72.52
Freccs	78.29	83.36	77.25	87.48	88.40	87.04	79.41	78.99	79.20
Freedomotic	87.35	90.35	88.70	89.72	91.54	90.45	92.35	93.54	92.58
平均值	**73.21**	**78.31**	**72.40**	**77.82**	**77.54**	**76.01**	**83.65**	**84.59**	**83.56**

表 5 展示了 DeepSmell 与文献 [10] 和 [11] 方法进行比较的实验结果, 可以看出 DeepSmell 在 9 个开源测试程序中上的整体性能上明显优于现有的方法. 具体表现为, 与基于 BP 神经网络[9]的方法相比, 在平均查准率上提高了 10.44%(=83.65%−73.21%), 平均查全率提高了 6.28%(=84.59%−78.31%), 平均 $F1$ 值提升了 11.16%(=83.56%−72.40%). 尽管 BP 神经网络在识别多种混合坏味时获得较好的性能, 但在反映文本信息中多层次信息之间的语义相关性方面效果较差.

将 DeepSmell 与 CNN 模型[11]在构建的特征依恋数据集上进行比较, 之所以没有在 Liu 等人[11]的数据集上进行比较, 主要原因是本文在提取文本信息和度量信息方面各不相同. 尽管在 Xmojo 和 Freccs 测试程序上本文方法性能低于 CNN 模型的性能, 但 DeepSmell 在其他测试程序上均获得较好性能. 此外, CNN 模型在 JSmooth 和 Displaytag 两个测试程序上获得了最低性能, 而 DeepSmell 在多个测试程序中的性能更加稳定, 在整体的性能上仍具有较明显优势. 在平均查准率上 DeepSmell 相比采用 CNN 模型检测方法提升了 5.83%(=83.65%−77.82%), 在查全率和 $F1$ 值上分别提升了 7.05%(=84.59%−77.54%) 和 7.55%(=83.56%−76.01%). 现有的工作[32]表明, 在获取文本信息中多层次信息之间潜在语义关系上 LSTM 明显优于 CNN 和全连接等方法, 本文通过对文献 [33] 的方法进行改进, 对于获取文本信息部分将 CNN 模型替换为 LSTM 模型.

在表 6 中将 DeepSmell 与文献 [34] 方法和 LSTM-CNN 模型进行了比较, 本文方法整体获得更好的性能. 其中, 与 LSTM-CNN 方法相比, 在平均查准率上分别提升了 16.2%(=83.65%−67.45%), 在平均查全率和平均 $F1$ 值上分别提升了 3.87%(=84.59%−80.72%) 和 10.37%(=83.56%−73.19%). 虽然 LSTM 模型可以获取文本信息中多层次信息之间的语义关系, 但信息之间的交互都是在 LSTM 内部之间进行, 获取的潜在语义信息不充分. 在 DeepSmell 中通过引入 GRU 层增加了文本信息之间的交互, 提高了模型的建模能力, 获取多层次信息之间更加深层的语义相关性信息. 相比于文献 [34] 方法, 在平均查准率、查全率和 $F1$ 值上分别提升了 11.37%(=83.65%−72.28%)、7.59%(=84.59%−77.00%) 和 11.35%(=83.56%−72.21%), 证实了 DeepSmell 在检测特征依恋坏味的有效性. 本文通过引入 GRU 层来增加多层次信息之间的交互性, 采用注意力机制计算文本信息和度量信息对预测结果的相关性, 进行重新分配权重, 提高了代码坏味检测的整体性能.

3.3.2 长方法检测结果

为了回答 RQ2, 我们将 DeepSmell 与现有的基于深度学习检测方法在长方法数据集上进行比较, 表 7 和表 8 给出长方法代码坏味的检测结果. 从实验结果可以看出 DeepSmell 在检测长方法上明显优于现有的方法, 平均的查准率、查全率和 $F1$ 值分别为 79.61%, 81.19% 和 79.56%. 相比于现有的方法, 在查准率、查全率和 $F1$ 值分别提高了 3.79~7.99%、4.08%~7.09% 和 6.54%~9.47%, 数据表明采用 DeepSmell 在检测长方法坏味方面仍具有更好的性能.

表 6 与文献 [34] 和 LSTM-CNN 模型比较特征依恋检测结果 (%)

项目名称	LSTM-CNN			文献[34]			DeepSmell		
	查准率	查全率	*F*1	查准率	查全率	*F*1	查准率	查全率	*F*1
Displaytag	66.64	81.63	73.38	72.77	80.27	74.48	84.34	85.71	84.43
Coberture	71.72	84.69	77.67	78.59	78.26	78.42	87.52	88.66	87.26
JavaCC	77.26	81.05	77.33	71.35	84.47	77.36	87.46	88.59	87.33
FitJava	54.29	73.68	63.52	62.16	71.52	63.80	78.89	79.26	79.05
JSmooth	66.04	81.26	72.86	71.90	79.68	73.83	82.45	84.20	82.65
JUnit	65.70	81.06	72.57	68.86	76.50	71.87	86.86	87.53	87.05
Xmojo	46.93	68.50	55.70	71.03	69.29	58.15	73.59	74.80	72.52
Freccs	69.47	83.19	75.71	69.49	83.36	75.80	79.41	78.99	79.20
Freedomotic	89.01	91.46	89.97	84.34	69.64	76.21	92.35	93.54	92.58
平均值	**67.45**	**80.72**	**73.19**	**72.28**	**77.00**	**72.21**	**83.65**	**84.59**	**83.56**

表 7 与文献 [10] 和文献 [11] 比较长方法检测结果 (%)

项目名称	文献[10]			文献[11]			DeepSmell		
	查准率	查全率	*F*1	查准率	查全率	*F*1	查准率	查全率	*F*1
Displaytag	69.44	75.51	65.79	67.06	64.40	65.55	78.40	80.05	77.60
Coberture	79.57	82.42	79.51	80.90	81.50	81.17	82.20	84.07	81.37
JavaCC	78.73	81.82	78.60	76.13	76.27	76.20	77.92	80.74	78.61
FitJava	64.57	70.54	58.89	68.36	71.73	67.04	78.60	79.46	78.59
JSmooth	74.42	78.31	73.69	69.75	64.86	66.80	83.12	84.16	83.18
JUnit	75.97	79.49	75.33	76.93	79.95	76.68	85.39	86.18	85.05
Xmojo	61.40	64.58	60.12	78.93	68.63	59.48	61.12	63.10	61.53
Freccs	69.25	64.05	66.14	73.59	78.10	73.17	77.25	79.58	77.69
Freedomotic	92.24	93.11	92.41	90.71	91.72	91.08	92.46	93.33	92.43
平均值	**73.95**	**76.65**	**72.28**	**75.82**	**75.24**	**73.02**	**79.61**	**81.19**	**79.56**

表 8 与文献 [34] 和 LSTM-CNN 比较长方法检测结果 (%)

项目名称	LSTM-CNN			文献[34]			DeepSmell		
	查准率	查全率	*F*1	查准率	查全率	*F*1	查准率	查全率	*F*1
Displaytag	57.02	75.51	64.97	72.75	71.20	71.88	78.40	80.05	77.60
Coberture	77.34	81.32	77.38	78.62	81.32	73.45	82.20	84.07	81.37
JavaCC	72.29	78.27	74.02	84.65	81.05	72.87	77.92	80.74	78.61
FitJava	75.61	75.30	70.26	64.57	70.54	58.89	78.60	79.46	78.59
JSmooth	73.65	77.87	73.38	80.27	79.49	71.64	83.12	84.16	83.18
JUnit	76.89	79.95	76.49	81.70	81.34	75.96	85.39	86.18	85.05
Xmojo	71.57	68.27	60.50	49.74	61.25	51.03	61.12	63.10	61.53
Freccs	54.87	42.32	47.19	65.88	76.47	68.77	77.25	79.58	77.69
Freedomotic	85.38	88.06	86.58	92.10	91.36	87.44	92.46	93.33	92.43
平均值	**71.62**	**74.10**	**70.09**	**74.48**	**77.11**	**70.21**	**79.61**	**81.19**	**79.56**

从表 7 中可以看出相比于文献 [10] 的方法, DeepSmell 在查准率、查全率和 *F*1 值分别提高了 5.66%(=79.61%−73.95%)、4.54%(=81.19%−76.65%) 和 7.28%(=79.56%−72.28%). 其中, 在 FitJava 项目中 *F*1 值提高最大达到 19.7%, Freedomotic 项目中 *F*1 值提高最小为 0.02%. 相比于文献 [11] 中的方法, DeepSmell 在查准率、查全率和 *F*1 值分别提高了 3.79%(=79.61%−75.82%)、5.95%(=81.19%−75.24%) 和 6.54%(=79.56%−73.02%). 在 Displaytag 测试程序上 DeepSmell 的查准率提升最高达到 11.34%(=78.40%−69.44%). 此外, 在 Xmojo 项目中, DeepSmell 的查准率下降了 0.28%(=61.40%−61.12%), 但在 *F*1 值上提高了 1.41%(=61.53%−60.12%).

在表 8 中可以看出, DeepSmell 在检测长方法的整体性能上优于其他两种方法. 相比于 LSTM-CNN 模型, DeepSmell 在平均查准率、查全率和 $F1$ 值分别提高了 7.99%(=79.61%−71.62%)、7.09%(=81.19%−74.10%) 和 9.47%(=79.56%−70.09%). 在 Freccs 测试程序中获得最大提升, 查准率、查全率和 $F1$ 值分别提高了 22.38%(=77.25%−54.87%)、37.26%(=79.58%−42.32%) 和 30.5%(=77.69%−47.19%). 相比于文献 [34] 的方法, DeepSmell 在平均查准率、查全率和 $F1$ 值分别提高了 5.13%(=79.61%−74.48%)、4.08%(=81.19%−77.11%) 和 9.35%(=79.56%−70.21%).

3.3.3 数据类和上帝类检测结果

为了回答 RQ3, 本文将 DeepSmell 与现有的方法在检测两种类级别代码坏味进行验证. 表 9 展示了数据类和上帝类的的实验测试结果, 可以看出 DeepSmell 在 15 个开源测试程序中上的整体性能上优于现有的方法. 在检测数据类坏味上, 查准率、查全率和 $F1$ 值分别为 80.10%、86.90% 和 81.13%. 相比于现有的方法, 在查准率、查全率和 $F1$ 值上平均提升了 4.30%、3.12% 和 1.86%. 在检测上帝类坏味中, 查准率、查全率和 $F1$ 值分别为 95.73%、95.77% 和 95.49%. 相比于现有的方法, 在查准率、查全率和 $F1$ 值上平均提升了 2.66%、2.48% 和 2.38%.

表 9 类级代码坏味测试结果 (%)

项目名称	数据类			上帝类		
	查准率	查全率	$F1$	查准率	查全率	$F1$
文献[10]	77.90	86.55	80.95	91.89	92.17	91.93
文献[11]	74.34	74.83	74.58	95.15	95.28	95.19
LSTM-CNN	75.71	**87.01**	80.97	92.99	93.20	93.01
文献[32]	75.24	86.74	80.58	92.26	92.51	92.30
DeepSmell	**80.10**	86.90	**81.13**	**95.73**	**95.77**	**95.49**

对于 RQ4, 通过将 LSTM-CNN 模型与 DeepSmell 在 4 种代码坏味数据集上进行验证. 在检测特征依恋、长方法和上帝类 3 种代码坏味上, DeepSmell 的整体获得最优的性能. 在检测数据类上, LSTM-CNN 在查全率上优于本文方法, 提升了 0.11%(=87.01%−86.9%). 而在查准率和 $F1$ 值上 DeepSmell 的性能较优, 分别提升了 4.39%(=80.1%−75.71%) 和 0.16%(=81.13%−80.97%). 通过上述 4 个问题的研究, 证实了 DeepSmell 在检测代码坏味上的有效性.

3.3.4 不同特征下的特征依恋检测结果

为了回答 RQ5, 我们比较了不同特征信息输入对于特征依恋检测结果的影响. 通过对比文本信息输入、度量信息输入、文本信息和度量信息相结合输入方式下的分类器在特征依恋坏味数据集上的检测结果. 3 种输入方式在测试集上的检测结果如表 10 所示. 当文本信息与度量信息均为 DeepSmell 的输入时, DeepSmell 在测试集上的性能优于任何一种单一特征信息输入的分类器. 具体表现为文本信息与度量信息相结合作为输入分类器的平均 $F1$ 值相对于文本信息输入分类器和度量信息输入分类器分别提高了 36.47%(=42.29%−5.82%) 和 14.68%(=42.29%−27.61%). 与文本信息相比, 度量信息对于分类器的预测结果起到了很大的作用, 尤其在查准率上的影响十分明显, 平均查准率高出了 13.94%(=16.70%−2.76%).

3.3.5 DeepSmell 与其他深度学习算法对比结果

为了回答 RQ6, 我们将 DeepSmell 与 Wide & Deep [45]和 DeepFM[46]两种深度学习算法进行对比, 表 11 展示了这 3 种深度学习算法在特征依恋坏味上的测试结果.

在表 11 中可以看出, 与 Wide & Deep [45]算法相比, DeepSmell 在平均查准率、查全率和 $F1$ 值分别提高了 0.87%(=83.65%−82.78%)、0.49%(=84.59%−84.10%) 和 2.47%(=83.56%−81.09%). 在 Cobertura、Xmojo、Displaytag 和 FitJava 等 6 个测试程序上 DeepSmell 的表现较好, 尤其在 Xmojo 测试程序上查准率和 $F1$ 值优势更为明显, 分别提高了 7.98%(=73.59%−65.61%) 和 9.37%(=72.52%−63.15%). 对于 Freecs、Freedomotic、JavaCC 等测试程序, DeepSmell 的表现略有下降, 可能的原因是这几个实际程序中正负样本比例差距较大, 导致

DeepSmell 将一些无代码坏味样本预测为有代码坏味样本. 但从整体上来看, DeepSmell 在代码坏味检测上的表现优于 Wide & Deep 算法.

表 10 不同特征下的特征依恋检测结果 (%)

项目名称	文本信息+度量信息			度量信息			文本信息		
	查准率	查全率	*F*1	查准率	查全率	*F*1	查准率	查全率	*F*1
Cobertura	75.00	40.74	52.80	15.31	100	26.56	1.69	100	3.33
Displaytag	22.22	87.53	28.57	0.00	0.00	0.00	1.10	100	2.18
FitJava	50.00	100	66.67	26.32	100	41.67	2.03	100	3.92
Freecs	18.79	100	28.21	16.64	100	28.53	5.19	100	9.88
Freedomotic	30.77	71.50	36.36	7.28	100	13.56	1.50	60.00	7.60
JavaCC	25.00	94.41	34.15	15.53	100	26.89	5.50	.92.86	10.38
JSmooth	17.39	94.79	29.63	18.74	100	31.56	1.87	100	3.62
JUnit	42.86	100	60.00	18.94	100	31.85	1.20	100	2.36
Xmojo	33.55	65	44.26	31.50	100	47.90	4.75	100	9.07
平均值	**35.06**	**83.77**	**42.29**	**16.70**	**88.89**	**27.61**	**2.76**	**86.44**	**5.82**

表 11 不同深度学习算法检测特征依恋的结果 (%)

项目名称	文献[45]			文献[46]			DeepSmell		
	查准率	查全率	*F*1	查准率	查全率	*F*1	查准率	查全率	*F*1
Cobertura	85.13	86.77	83.66	95.16	62.10	75.16	87.52	88.66	87.26
Displaytag	82.49	82.49	80.51	90.80	75.96	82.72	84.34	85.71	84.43
FitJava	76.06	77.09	72.12	90.90	23.91	37.79	78.89	79.26	79.05
Freecs	80.73	84.03	79.32	87.73	86.44	87.18	79.41	78.99	79.20
Freedomotic	97.50	97.11	97.30	96.49	53.92	69.18	92.35	93.54	92.58
JavaCC	93.34	92.68	93.00	90.00	21.95	35.21	87.46	88.59	87.33
JSmooth	82.12	83.75	79.99	71.42	47.62	53.15	82.45	84.20	82.65
JUnit	82.08	83.69	80.06	90.10	81.99	85.86	86.86	87.53	87.05
Xmojo	65.61	69.29	63.15	88.33	32.92	47.96	73.59	74.80	72.52
平均值	**82.78**	**84.10**	**81.09**	**88.99**	**54.09**	**63.80**	**83.65**	**84.59**	**83.56**

相比于 DeepFM[46]算法, DeepSmell 在平均查全率和 *F*1 值上分别提高了 30.5%(=84.59%−54.09%) 和 19.76%(=83.56%−63.80%). 在 JavaCC 测试程序上 DeepSmell 的查全率和 *F*1 值提升最为显著, 分别提高了 66.64%(=88.59%−21.95%) 和 52.12%(=87.33%−35.21%). 同时, 我们也发现 DeepSmell 在平均查准率上相比于 DeepFM 算法降低了 5.34%(=88.99%−83.65%), 但 DeepSmell 在所有测试程序的整体表现上更加稳定, 对不同测试程序的适用性更好, 而 DeepFM 在不同程序上的测试结果差距较大, 可能更适用于某些特定的测试程序. 在检测代码坏味的综合表现上, DeepSmell 优于 Wide & Deep 和 DeepFM 两种深度学习算法.

3.3.6 DeepSmell 时间性能评估

对于回答 RQ7, 我们评估了 DeepSmell 在检测特征依恋代码坏味上的时间性能, 针对测试数据集, 记录了 DeepSmell 在整个检测特征依恋代码坏味上的耗时情况, 如表 12 所示.

从表 12 可以看出, 使用 DeepSmell 在 9 个项目上完成特征依恋代码坏味检测的总时长为 780 min, 平均每个项目耗时 87 min. 其中, 耗时最久的步骤是 DeepSmell 分类器的训练花费 674 min, 占用整个检测时长的 85% 以上, 这主要是因为 BERT 模型需要对文本信息中的多层次信息进行预训练, 以此获取不同层次信息之间的语义关系, 这个过程使用了多个 Transformer 层对文本信息进行数据处理, 操作过程较为复杂, 耗时时间较长. 在文本信息生成总时长为 4 min (平均每个项目约 27 s, Xmojo 耗时最短 3 s, Freedomotic 耗时最长 60 s), 通过静态分析工具获取代码度量信息平均每个程序花费 42 s, 数据集和分类器的构建分别花费了 19 min 和 62 min, 在 9 个开源程序上的测试过程中耗时 13 min (平均每个项目 1 分 27 秒).

表 12 DeepSmell 完成各个步骤所花费的时间 (min)

步骤名称	所花费时间
文本信息生成	4
度量信息生成	7
建立数据集	19
数据预处理	1
建立分类器	62
DeepSmell训练	674
DeepSmell测试	13
总计	**780**

3.4 有效性威胁

本节讨论了实验中可能影响测试结果的有效性威胁. 第 1 个有效性的威胁是本文在评估中仅选取 24 个应用程序进行数据集的构建, 然而这些程序并不能代表所有程序. 为了减少对数据集构建的影响, 我们尽量选择的来自不同领域的应用程序, 尽可能保证数据的多样性. 第 2 个有效性的威胁是在构建数据集时采用现有的工具和检测规则, 由于不同的工具和规则选取的阈值和标准存在差异, 这可能会导致数据集的构建存在差异. 为了减少这方面的有效性威胁, 我们对于每种代码坏味都选择两种以上的规则进行标记, 只有满足所有规则才标记为 1 (有代码坏味).

4 总 结

本文提出一种基于预训练模型和多层次信息的代码坏味检测方法 DeepSmell. 该方法首先利用静态分析工具获取多个实际应用程序中代码坏味实例和多层次的代码度量信息, 并对代码坏味实例进行标记. 通过抽象语法树解析并获取源程序中与代码坏味相关层次信息, 将层次信息组成的文本信息和度量信息相结合生成数据样本. 然后使用 BERT 预训练模型将文本信息转化为词向量, 通过 GRU-LSTM 方法获取文本信息中层次信息之间潜在的语义关系, 并与 CNN 模型和注意力机制相结合实现对代码坏味进行检测. 实验在 24 个开源应用程序上对 4 种代码坏味进行检测, 并与现有的代码坏味检测方法进行比较. 实验结果表明与目前已有的检测方法相比, DeepSmell 在平均查全率和 *F*1 值上分别提高了 9.3% 和 10.44%, 同时保持了较高的查准率, DeepSmell 可以有效的实现代码坏味检测. 在后续的工作中, 我们将对其他代码坏味进行检测来继续验证本文方法的适用性, 同时还将继续对本文方法进行改进以提高代码坏味检测的查准率.

References:

[1] Mayvan BB, Rasoolzadegan A, Jafari AJ. Bad smell detection using quality metrics and refactoring opportunities. Journal of Software: Evolution and Process, 2020, 32(8): e2255. [doi: 10.1002/smr.2255]

[2] Fowler M. Refactoring: Improving the Design of Existing Code. AddisonWesley, 1999. 16–50.

[3] Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 482. [doi: 10.1145/3180155.3182532]

[4] Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A. JDeodorant: Identification and application of extract class refactorings. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: ACM, 2011. 1037−1039. [doi: 10.1145/1985793.1985989]

[5] Marinescu C, Marinescu R, Mihancea PF, Ratiu D, Wettel R. iPlasma: An integrated platform for quality assessment of object-oriented design. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance. Budapest: ICSM, 2005. 77–80.

[6] Nongpong K. Integrating "code smells" detection with refactoring tool support [Ph.D. Thesis]. Milwaukee: University of Wisconsin-Milwaukee, 2012. 106–112.

[7] Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 2016, 21(3): 1143–1191. [doi: 10.1007/s10664-015-9378-4]

[8] Pecorelli F, Palomba F, Di Nucci D, De Lucia A. Comparing heuristic and machine learning approaches for metric-based code smell detection. In: Proc. of the 27th Int'l Conf. on Program Comprehension (ICPC). Montreal: IEEE, 2019. 93–104. [doi: 10.1109/ICPC.2019.00023]

[9] 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). [doi: 10.11896/j.issn.1002-137X.2017.6A.110]

[10] 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). [doi: 10.19678/j.issn.1000-3428.0055862]

[11] Liu H, Xu ZF, Zou YZ. Deep learning based feature envy detection. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 385–396. [doi: 10.1145/3238147.3238166]

[12] Tiwari O, Joshi RK. Functionality based code smell detection and severity classification. In: Proc. of the 13th Innovations in Software Engineering Conf. on Formerly Known as India Software Engineering Conf. Jabalpur: ACM, 2020. 14. [doi: 10.1145/3385032.3385048]

[13] Zhang SD, Wu HT, Gao JH. Correlation analysis of code smells with severity by principal axis factor method. Journal of Chinese Computer Systems, 2021, 42(4): 853–860 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2021.04.031]

[14] Wei HH. A recommendation strategy for two-order odor reconstruction based on hierarchy and code change [MS. Thesis]. Shanghai: Shanghai Normal University, 2020 (in Chinese with English abstract). [doi: 10.27312/d.cnki.gshsu.2020.001882]

[15] Wang Y, Yu H, Zhu ZL, Zhang W, Zhao YL. Automatic software refactoring via weighted clustering in method-level networks. IEEE Trans. on Software Engineering, 2018, 44(3): 202–236. [doi: 10.1109/TSE.2017.2679752]

[16] Meng FY, Wang Y, Yu H, Zhu ZL. Refactoring of complex software systems research: Present, problem and prospect. Computer Science, 2020, 47(12): 1–10 (in Chinese with English abstract). [doi: 10.11896/jsjkx.200800067]

[17] Moha N, Gueheneuc YG, Duchien L, Le Meur AF. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. on Software Engineering, 2010, 36(1): 20–36. [doi: 10.1109/TSE.2009.50]

[18] Terra R, Valente MT, Miranda S, Sales V. JMove: A novel heuristic and tool to detect move method refactoring opportunities. Journal of Systems and Software, 2018, 138: 19–36. [doi: 10.1016/j.jss.2017.11.073]

[19] Fard AM, Mesbah A. JSNOSE: Detecting JavaScript code smells. In: Proc. of the 13th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). 2013. 116–125. [doi: 10.1109/SCAM.2013.6648192]

[20] Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E. A review-based comparative study of bad smell detection tools. In: Proc. of the 20th Int'l Conf. on Evaluation and Assessment in Software Engineering. Limerick: ACM, 2016. 18. [doi: 10.1145/2915970.2915984]

[21] Zhang Y, Shao S, Zhang DW. Automated refactoring approach for fine-grained lock based on pushdown automaton. Ruan Jian Xue Bao/Journal of Software, 2021, 32(12): 3710–3727 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6132.htm [doi: 10.13328/j.cnki.jos.006132]

[22] Zhang Y, Li LX, Zhang DW. A survey of concurrency-oriented refactoring. Concurrent Engineering, 2020, 28(4): 319–330. [doi: 10.1177/1063293X20958932]

[23] Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. Journal of Systems and Software, 2011, 84(4): 559–572. [doi: 10.1016/j.jss.2010.11.921]

[24] Amal A, Hamoud A. Code smell detection using feature selection and stacking ensemble: An empirical investigation. Information Software Technology. 2021, 138: 106648. [doi: 10.1016/j.infsof.2021.106648]

[25] Kreimer J. Adaptive detection of design flaws. Electronic Notes in Theoretical Computer Science, 2005, 141(4): 117–136. [doi: 10.1016/j.entcs.2005.02.059]

[26] Amorim L, Costa E, Antunes N, Fonseca B, Ribeiro M. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: Proc. of the 26th Int'l Symp. on Software Reliability Engineering. Gaithersbury: IEEE, 2015. 261–269. [doi: 10.1109/ISSRE.2015.7381819]

[27] Nucci DD, Palomba F, Tamburri DA, Serebrenik A, De 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. Campobasso: IEEE, 2018. 612–621. [doi: 10.1109/SANER.2018.8330266]

[28] Guggulothu T. Code smell detection using multilabel classification approach. arXiv: 1902.03222, 2019.

[29] Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Antoniol G, Aïmeur E. Support vector machines for anti-pattern detection. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 278–281. [doi: 10.1145/2351676.2351723]

[30] Wang JN, Chen JH, Gao JH. ECC multi-label code smell detection method based on ranking loss. Journal of Computer Research and Development, 2021, 58(1): 178–188 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2021.20190836]

[31] Liu H, Jin JH, Xu ZF, Zou YZ, Bu YF, Zhang L. Deep learning based code smell detection. IEEE Trans. on Software Engineering, 2021, 47(9): 1811–1837. [doi: 10.1109/TSE.2019.2936376]

[32] 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]

[33] Das AK, Yadav S, Dhal S. Detecting code smells using deep learning. In: Proc. of the 2019 IEEE Region 10 Conf. (TENCON). Kochi: IEEE, 2019. 2081–2086. [doi: 10.1109/TENCON.2019.8929628]

[34] Guo XL, Shi CY, Jiang H. Deep semantic-based feature envy identification. In: Proc. of the 11th Asia-Pacific Symp. on Internetware. Fukuoka: ACM, 2019. 19. [doi: 10.1145/3361242.3361257]

[35] Singh R, Bindal A, Kumar A. Long method and long parameter list code smells detection using functional and semantic characteristics. Int'l Journal of Recent Technology and Engineering, 2020, 8(6): 2223–2232. [doi: 10.35940/ijrte.E5888.038620]

[36] Fang CR, Liu ZX, Shi YY, Huang J, Shi QK. Functional code clone detection with syntax and semantics fusion learning. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2020. 516–527. [doi: 10.1145/3395363.3397362]

[37] 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 (in Chinese with English abstract), 2021. http://www.jos.org.cn/1000-9825/6373.htm [doi: 10.13328/j.cnki.jos.006373]

[38] Devlin J, Chang MW, Lee K, Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: Association for Computational Linguistics, 2018. 4171–4186. [doi: 10.18653/v1/N19-1423]

[39] Chung J, Gulcehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. In: Proc. of the NIPS 2014 Workshop on Deep Learning. 2014.

[40] Pennington J, Socher R, Manning C. GloVe: Global vectors for word representation. In: Proc. of the 2014 Conf. on Empirical Methods in Natural Language Processing. Doha: ACL, 2014. 1532–1543. [doi: 10.3115/v1/D14-1162]

[41] Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 27th Annual Conf. on Neural Information Processing Systems 2013. Lake Tahoe, 2013. 3111–3119.

[42] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc. , 2017. 6000–6010.

[43] Marinescu R. Measurement and quality in object-oriented design. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance. Budapest: IEEE, 2005. 701–704. [doi: 10.1109/ICSM.2005.63]

[44] Trifu A, Marinescu R. Diagnosing design problems in object oriented systems. In: Proc. of the 12th Working Conf. on Reverse Engineering. Pittsburgh: IEEE, 2005. 1–10. [doi: 10.1109/WCRE.2005.15]

[45] Cheng HT, Koc L, Harmsen J, Shaked T, Chandra T, Aradhye H, Anderson G, Corrado G, Chai W, Ispir M, Anil R, Haque Z, Hong LC, Jain V, Liu XB, Shah H. Wide & deep learning for recommender systems. In: Proc. of the 1st Workshop on Deep Learning for Recommender Systems. Boston: ACM, 2016. 7–10. [doi: 10.1145/2988450.2988454]

[46] Guo HF, Tang RM, Ye YM, Li ZG, He XQ. DeepFM: A factorization-machine based neural network for CTR prediction. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. Melbourne, 2017. 1725–1731. [doi: 10.24963/ijcai.2017/239]]

附中文参考文献:

[9] 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. [doi: 10.11896/j.issn.1002-137X.2017.6A.110]

[10] 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. [doi: 10.19678/j.issn.1000-3428.0055862]

[13] 张生栋, 吴海涛, 高建华. 利用主轴因子法的严重性代码异味相关性分析. 小型微型计算机系统, 2021, 42(4): 853–860. [doi: 10.3969/j.issn.1000-1220.2021.04.031]

[14] 位欢欢. 基于层次和代码变更的两阶异味重构推荐策略 [硕士学位论文]. 上海: 上海师范大学, 2020. [doi: 10.27312/d.cnki.gshsu.2020.001882]

[16] 孟繁祎, 王莹, 于海, 朱志良. 复杂软件系统的重构技术: 现状、问题与展望. 计算机科学, 2020, 47(12): 1–10. [doi: 10.11896/jsjkx.200800067]

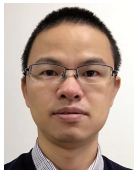
[21] 张杨, 邵帅, 张冬雯. 基于下推自动机的细粒度锁自动重构方法. 软件学报, 2021, 32(12): 3710–3727. http://www.jos.org.cn/1000-9825/6132.htm [doi: 10.13328/j.cnki.jos.006132]

[30] 王继娜, 陈军华, 高建华. 基于排序损失的ECC多标签代码异味检测方法. 计算机研究与发展, 2021, 58(1): 178–188. [doi: 10.7544/issn1000-1239.2021.20190836]

[32] 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]

[37] 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, 2021. http://www.jos.org.cn/1000-9825/6373.htm [doi: 10.13328/j.cnki.jos.006373]

张杨(1980−), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件重构, 并发软件.



刘辉(1978−), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件分析, 软件重构, 软件测试.

东春浩(1996−), 男, 硕士生, CCF 学生会员, 主要研究领域为并发软件分析, 智能化软件.

葛楚妍(1997−), 女, 硕士生, 主要研究领域为智能化软件.