软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(2):195−210 [doi: 10.13328/j.cnki.jos.005565]

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

方法级别的细粒度软件缺陷定位方法*

张 文[1,2], 李自强[1], 杜宇航[1], 杨 叶[3]

[1](北京化工大学 经济管理学院,北京 100029)

[2](北京工业大学 经济管理学院,北京 100124)

[3](School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030, USA)

通讯作者: 张文, E-mail: zhangwen@mail.buct.edu.cn

摘 要: 当软件缺陷报告在跟踪系统中被指派给开发人员进行缺陷修复之后,缺陷修复人员就需要根据提交的缺陷报告来进行软件缺陷定位,并做出相应的代码变更,以修复该软件缺陷.在缺陷修复的整个过程中,软件缺陷定位占用了开发人员大量的时间.提出了一种方法级别的细粒度软件缺陷定位方法MethodLocator,以提高软件修复人员的工作效率.MethodLocator首先对缺陷报告和源代码方法体利用词向量(word2vec)和TF-IDF结合的方法进行向量表示;然后,根据源代码文件中方法体之间的相似度对方法体进行扩充;最后,通过对扩充后的方法体和缺陷报告计算其余弦距离并排序,来定位为修复软件缺陷所需做出变更的方法.在4个开源软件项目ArgoUML、Ant、Maven和Kylin上的实验结果表明,MethodLocator方法优于现有的缺陷定位方法,它能够有效地将软件缺陷定位到源代码的方法级别上.

关键词: 缺陷报告;MethodLocator;细粒度缺陷定位;方法级别;词向量表示

中图法分类号: TP311

中文引用格式: 张文,李自强,杜宇航,杨叶.方法级别的细粒度软件缺陷定位方法.软件学报,2019,30(2):195−210. http://www.jos.org.cn/1000-9825/5565.htm

英文引用格式: Zhang W, Li ZQ, Du YH, Yang Y. Fine-grained software bug location approach at method level. Ruan Jian Xue Bao/Journal of Software, 2019,30(2):195−210 (in Chinese). http://www.jos.org.cn/1000-9825/5565.htm

Fine-grained Software Bug Location Approach at Method Level

ZHANG Wen[1,2], LI ZI-Qiang[1], DU Yu-Hang[1], YANG Ye[3]

[1](School of Economics and Management, Beijing University of Chemical Technology, Beijing 100029, China)

[2](College of Economics and Management, Beijing University of Technology, Beijing 100124, China)

[3](School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030, USA)

Abstract: When a software bug report is assigned to a developer for bug resolution, the developer needs to locate the bug in a source code file and make code changes correspondingly to resolve the software bug. In fact, most of time of the developer is spent on bug location in the whole process of bug resolution. This study proposes a method level fine-grained bug location approach, called MethodLocator, to improve the efficiency of software bug resolution. Firstly, it takes the vector representation of the bug report and the source code method body using the word vector (Word2Vec) and TF-IDF. Secondly, MethodLocator augments method body of each method based on similarities among all method bodies in the source code files. Thirdly, MethodLocator locates methods for change to resolve the bug based on similarities between the bug report and the augmented methods. Experimental results on four open source

* 基金项目: 国家自然科学基金(61379046, 61432001); 西安市科技计划(2016CXWL21)

Foundation item: National Natural Science Foundation of China (61379046, 61432001); Science and Technology Project of Xi'an Municipality (2016CXWL21)

收稿时间: 2017-09-06; 修改时间: 2017-10-31; 采用时间: 2018-02-09; jos在线出版时间: 2018-03-13

CNKI网络优先出版: 2018-03-14 09:18:11, http://kns.cnki.net/kcms/detail/11.2560.TP.20180314.0918.003.html

software projects as ArgoUML, Ant, Maven, and Kylin demonstrate that MethodLocator is better than state-of-the-art techniques in method level bug location.

Key words: bug report; methodLocator; fine-grained bug location; method level; word vector representation

软件缺陷是计算机程序或系统中的错误、故障或瑕疵,导致其产生不正确或意料之外的行为方式[1].软件缺陷的存在,会导致软件产品在某种程度上不能满足用户的需求.在软件项目开发过程中,一些缺陷跟踪系统常被用于管理缺陷,如 Bugzilla(Bugzilla:https://www.bugzilla.org/)、JIRA(JIRA:http://www.atlassian.com/software/jira/)、Mantis(Mantis:https://www.mantisbt.org/)等.这些缺陷跟踪系统被用于管理软件项目开发中缺陷报告的提交、确认、分配、修复、关闭等整个软件缺陷的生命周期[2].对于一个大型软件项目来说,每天都会收到用户提交的大量软件缺陷报告,而且修复这些软件缺陷耗费了缺陷修复人员大量的时间和精力.例如,在Eclipse项目版本发布日期附近,每天有将近 200 个缺陷报告被提交到 Eclipse 项目缺陷报告库.同样,每天有将近 150 个缺陷报告被提交到 Debian 项目缺陷报告库[3].根据 Jeong 等人[4]的研究,在 PostgreSQL 项目中,大部分缺陷需要 100 天~200 天被修复;甚至有 50%的缺陷报告需要将近 100 天~300 天才能被修复.根据本文对所采用的 Tomcat7 项目的观察结果:大部分缺陷在 40 天~200 天之内被修复;10%的缺陷在 10h 之内可以得到修复;另有 5%的缺陷需要将近 2 年才能被最终修复.

一旦软件缺陷报告被缺陷管理人员所确认和分派给开发人员进行缺陷修复,那么被指派的缺陷修复人员就要进行缺陷定位,也就是找出为修复该缺陷所需修改的代码片段,然后进行缺陷修复[5].对于软件维护人员来讲,要对某一个缺陷进行修复,首先必须对缺陷相关信息进行充分的了解.为此,软件维护人员需要阅读大量的软件源代码来帮助自己确定缺陷所在的位置.当缺陷报告和源代码文件的数量很多时,软件缺陷定位就是一件非常费时、费工的任务.如果一个缺陷久久不能定位到正确的位置,缺陷修复的时间就会增加,相应的软件项目的维护成本也会增加,同时用户对软件产品的满意度就会下降.

近年来,研究学者提出了一系列软件缺陷定位方法,以期辅助缺陷修复人员进行软件缺陷定位,减少其在缺陷修复时的工作量.软件缺陷定位一般可分为静态定位方法和动态定位方法:静态缺陷定位依赖于软件缺陷报告、源代码和开发过程静态信息来进行软件缺陷定位[6];动态缺陷定位依赖于插桩技术、执行监控和形式化方法等技术来进行软件运行时状态跟踪,以确定软件缺陷可能发生的位置[7].静态定位方法的优点主要是不要求一个可运行的软件系统,可以使用在软件开发和维护的任意阶段[8].本文的研究焦点在于静态缺陷定位方法,即如何利用信息检索技术来提高软件缺陷定位的精度和效率.

不同于传统的将软件缺陷定位为文件级别的方法,本文提出了一种方法级别的细粒度软件缺陷定位方法:MethodLocator,辅助软件缺陷修复人员进行缺陷定位.具体来说,为了解决方法体中的词项稀疏问题,MethodLocator 使用基于 word2vec 词向量的文档向量表示方法[9]对缺陷报告和源代码方法体内容进行向量表示;然后,利用夹角余弦计算缺陷报告和源代码方法之间的相似度,进而对查询结果进行排序.在对方法体进行自然语言预处理的过程中,考虑到单个方法相对于单个缺陷报告的文本内容较短,本文根据方法之间的相似度,利用其他方法对该单个方法的内容实施了进一步的扩充.本文的贡献包含了以下 3 个方面.

1) 在对传统的缺陷定位方法进行全面回顾的基础之上,本文提出了一种方法级别的缺陷定位方法MethodLocator.
2) 本文首次提出了结合词向量[9]和 TD-IDF 的源代码方法和缺陷报告向量表示方法,并提出利用单个方法的相似方法对其表示向量进行扩充的方法,其目的是减小单个方法向量表示的特征稀疏性.
3) 本文自行完成了对 ArgoUML、Ant、Maven 和 Kylin 这 4 个开源项目的缺陷数据及其对应的方法级别的源代码变更数据的收集,建立了方法级别的缺陷定位研究的标杆数据集.

本文第 1 节介绍近年来在软件缺陷定位方法研究方面的相关研究进展.第 2 节提出一种方法级别的细粒度缺陷定位方法 MethodLocator.第 3 节和第 4 节对本文所提出的 MethodLoactor 方法与基准方法进行实验论证和充分比较.第 5 节总结全文并提及未来的工作.

1 相关研究

本文的相关研究主要包括软件缺陷静态定位方法.Poshyvanyk 等人利用潜在语义索引和概率检索模型提出了PROMESIR方法[10].该方法将源代码中的特征定位问题转化为不确定性的决策问题.PROMESIR结合了两种特征定位技术,即基于情景的事件概率排序和使用潜在语义索引的信息检索技术来进行软件缺陷定位.他们对Mozilla Web浏览器的源代码缺陷数据进行了实验,其结果表明,与独立使用一种技术相比,PROMESIR组合使用两种特征定位技术,显著提高了缺陷定位的有效性.

Lukinsu等人[5]提出了一种基于LDA(latent dirichlet allocation)的自动缺陷定位技术,并就下述5个问题开展了广泛的实验:(1) 他们在之前基于LSI的缺陷定位方法应用的数据集上进行了实验,结果显示,基于LDA的缺陷定位方法的准确性更优;(2) 在软件项目Rhino的数据集中进行实验,验证了基于LDA的缺陷定位方法的有效性;(3) 在两个软件项目Rhino和Eclipse中进行实验,验证了基于LDA的缺陷定位方法的可扩展性;(4) 他们分析了基于LDA的缺陷定位方法的准确性与软件系统规模之间的关系,发现该方法在软件缺陷定位方面的准确性与软件系统的规模并无显著相关关系;(5) 同时,他们也分析了基于LDA的缺陷定位方法的准确性与软件系统设计的稳定性之间的关系,结果显示,二者并无显著相关关系.

Zhou等人[11]提出了一种基于信息检索的缺陷定位方法BugLocator.该方法分为4个步骤,即语料库创建、缺陷报告和源代码索引、查询构建/缺陷报告检索、源代码文件排序.首先,他们在BugLocator方法中提出了修正的向量空间模型(rVSM),用以对缺陷报告和源代码文件进行文本表示;然后,他们根据给定的缺陷报告与历史缺陷报告的相似度、该缺陷报告与源代码文件的相似度以及历史缺陷报告改动的源代码文件记录这3个维度来对修复该缺陷报告可能需要修改的源代码文件进行排序.他们在Eclipse、SWT、AspectJ、Zxing这4个项目的数据集上进行了实验,实验结果表明,BugLocator定位效果优于基于LDA、LSI的定位方法.

Moreno[12]提出了一种名为Lobster的静态缺陷定位方法.该方法将缺陷报告中的堆栈信息引入到软件缺陷静态定位方法研究中来,利用堆栈分析和文本检索相结合的方法进行缺陷定位.具体来说,Lobster方法使用缺陷报告中的堆栈信息来计算缺陷报告的代码元素和软件系统的源代码之间的相似度.结合堆栈信息与源代码的相似性和文本信息与源代码之间的文本相似性来定位与缺陷报告相关的代码片段.

Saha等人[13]提出一种仅需要源代码和缺陷报告来进行缺陷定位的方法BLUiR.该方法使用TF-IDF模型,并结合结构信息检索来度量缺陷报告和源代码文件之间的相似性.首先,他们计算了缺陷报告的2个字段(即报告总结(summary)和内容描述(description))与源代码文件中的4个部分(即类名(class)、方法名(method)、变量名(variable names)和注释(comments))两两之间的相似性;然后将得到的8个相似分数组合,得到缺陷报告与源代码文件之间的最终相似度,并依据此最终相似度,针对给定的缺陷报告对源代码文件进行排名.他们在C程序软件的缺陷定位中验证了BLUiR缺陷定位方法的有效性[14].

Wang等人[15]认为,将源代码的版本历史、结构化信息、相似缺陷报告三者结合起来可以提高软件缺陷定位的性能,并提出了一种新的缺陷定位方法Amalgam.该方法集成了Rahman[16]提出的缺陷预测算法来分析源代码的版本历史,然后使用BugLocator来分析缺陷报告系统中类似的缺陷报告,最后使用BLUiR对源代码的结构化信息进行分析.他们在4个开源项目(AspectJ、Eclipse、SWT和ZXing)中进行的实验结果表明,Amalgam的缺陷定位性能优于BugLocator和BLUiR.

Le等人[17]提出了一种多模式软件缺陷定位方法,他们同时考虑了缺陷报告和程序光谱来进行软件缺陷定位.该方法通过构建Bug-Specic模型,将特定的缺陷报告映射到其可能需要修改的源代码文件.他们在4个项目AspectJ、Ant、Lucene、Rhino的157个实际缺陷中进行了实验,验证了他们提出的方法的有效性.

Wong等人[18]提出使用代码分段和堆栈跟踪分析来提高缺陷定位的性能.首先,他们将每个源代码文件分成一系列的代码片段,对于给定的一个缺陷报告,他们使用与该缺陷报告最相似的代码片段来表示该源代码文件;然后,他们分析缺陷报告中的堆栈信息与源代码文件之间的相似性;最后,通过综合两种分析结果来定位到可能发生问题的源代码文件.

Ye等人[19]定义了一个排名模型,使用Learning to Rank(LtR)方法度量缺陷报告和源文件之间关系的6个特

征来进行缺陷定位.这 6 个特征包括:(1) 缺陷报告和源代码文件之间的相似性;(2) 缺陷报告和源代码 API 文档之间的相似性;(3) 以前修复过的类似缺陷报告;(4) 缺陷修复新进度,即以月为单位的上次修复时间;(5) 缺陷修复频率,即文件被修复的频率;(6) 特征缩放.他们通过设定对应比例,综合这 6 个方面的评分,从而得出单个源文件用于当前修复该缺陷报告的可能性.Ye 等人[20]认为,以自然语言(例如英语)表达的搜索查询与通常以代码(例如编程语言)表示的检索文档之间的“词汇鸿沟”使得信息检索技术在软件工程中的搜索任务变得困难.他们提出引入词向量以解决“词汇鸿沟”的问题,之后的实验证明了使用词向量能够对之前的缺陷定位方法进行改进.缺陷定位方法的研究从最初的只考虑源代码文件与缺陷报告之间的相似性,到后来的考虑缺陷报告的结构化信息,再加上源代码文件的结构化信息、缺陷报告中的堆栈信息等,逐渐丰富了缺陷定位的信息源,使缺陷定位的准确率得到提升.这也在一定程度上减轻了软件维护人员修复缺陷的复杂度,提高了缺陷修复的效率.

上述关于软件缺陷定位的研究都在源代码文件级别,而关于方法级别上的软件缺陷研究目前较少.在软件缺陷预测的研究中,Giger 等人[21]认为,大多数缺陷预测方法是对文件级别的缺陷作出预测,这通常会使得开发人员花费大量的精力去检查文件中的所有方法,直到找到发生错误的地方.为了减少开发人员手动检查工作所需要的时间和精力,Giger 等人[21]提出了一个方法级别的缺陷预测模型,之后,原子等人[22]和 Hideak 等人[23]也对方法级别的缺陷预测进行了研究.

和软件缺陷预测类似,在软件缺陷定位工作中,大多数软件缺陷定位方法将研究粒度放在文件级别,整体存在着粒度比较粗糙的问题[24].如果将缺陷定位的粒度提高到方法级别,就可以进一步提高缺陷修复人员的工作效率,减少软件的维护成本.就目前的文献调研结果来看,仅有 Youm 等人[25]提出一种综合分析(bug localization using integrated analysis,简称 BLIA)方法来进行方法级别的软件缺陷定位.BLIA 利用缺陷报告文本、堆栈信息、源代码注释、源代码文件结构信息和源代码变更历史信息进行软件缺陷定位.值得一提的是,BLIA 1.0 基于文件级别的缺陷定位,BLIA 1.5 将文件级别的缺陷定位的粒度提高到了方法级别.在其方法级别的定位技术中,首先,他们利用 BLIA 完成对文件级别的排序;然后选取排名前 10 的文件,对此类文件中的方法体进行分析,得到源代码方法体的排序,以此来实现方法级别的软件缺陷定位.

2 细粒度软件缺陷定位

2.1 问题描述

图 1 展示的是 Maven 项目 ID 为#MNG-4367 的缺陷报告(https://issues.apache.org/jira/si/jira.issueviews:issue-html/MNG-4367/MNG-4367.html).

Fig.1 An example of a bug report for a Maven project

图 1 一个 Maven 项目的缺陷报告示例

该缺陷报告顶端是缺陷的编号和缺陷的总结;接下来是缺陷提交者对该缺陷做的详细描述(description),主要包括缺陷发生时的软件运行上下文信息;对此缺陷感兴趣的软件项目开发人员可以在详细描述下面进行评论(comment),系统会自动记录评论人和评论时间.对缺陷报告进行评论是软件开发者围绕该缺陷主题进行交流沟通的主要方法.在遇到比较难以解决的软件缺陷时,对该缺陷报告的评论可能多达几十条.

假定对于一个软件项目有 n 个缺陷报告 $BR=(br_1,br_2,\ldots,br_n)$,$br_i$ 表示其中的一个缺陷报告.为了将 br_i 所描述的缺陷进行修复,其所修改的文件为集合为 $f(br_i)=\{f_1^{br_i},f_2^{br_i},\ldots,f_{|f(br_i)|}^{br_i}\}$ ($f_j^{br_i}\in F$,F 表示该软件项目的所有源代码文件的集合,$|f(br_i)|$表示集合 $f(br_i)$中元素的数量).事实上,当一名缺陷修复人员在修改文件时,他仅仅修改了文件中的 1 个或多个方法,$m(f_j^{br_i})=\{m_1,\ldots,m_p\}(m(f_j^{br_i})\subset mall(f_j^{br_i}),mall(f_j^{br_i})$ 表示文件 $f_j^{br_i}$ 中所有方法的集合).在传统的方法中,软件缺陷定位描述为如何利用缺陷报告 br_i 从 F 中准确定位所需修改的文件集合 $f(br_i)$.然而在本文中,这里所关注的问题是如何利用缺陷报告 br_i 从 F 中准确定位所需修改的文件集合 $f(br_i)$及其对应的方法 $m(f_j^{br_i})$.

表 1 展示了 Maven 项目修复 MNG-4367 所涉及的具体文件及方法示例.缺陷报告 MNG-4367 如图 1 所示.Maven 项目包含的源代码文件总数为 898 个.为了修复缺陷报告 MNG-4367 描述的缺陷,需要对 2 个源代码文件即 DefaultMirrorSelector.java 和 DefaultMirrorSelector.java 中的 6 个方法做出修改,其中,DefaultMirrorSelector.java 文件总共包含了 5 种方法,实际需要修改其中的 3 种方法;MirrorProcessorTest.java 文件总共包含了 14 个方法,实际需要修改其中的 3 种方法.具体需要修改的方法见表 1.由此,本文将方法级别的软件缺陷定位问题描述为:利用根据缺陷报告 MNG-4367 的描述内容,从 Maven 项目 898 个源代码文件中找到要修改的 2 个文件(DefaultMirrorSelector.java,MirrorProcessorTest.java),并进一步找出在这 2 个文件中需要修改的 6 种方法.

Table 1 Source code files and methods for fixing MNG-4367 as well as ranking of the changed methods by MethodLocator

表 1 修复 MNG-4367 所涉及具体文件和方法以及 MethodLocator 对所需修改方法的排序

修改文件	包含方法数	修改方法	本文方法定位结果
DefaultMirrorSelector.java	5	Mirror getMirror(ArtifactRepository repository,List〈Mirror〉 mirrors)	3
		Boolean matchesLayout(ArtifactRepository repository,Mirror mirror)	6
		Boolean matchesLayout(String repoLayout, String mirrorLayout)	9
MirrorProcessorTest.java	14	Mirror newMirror(String id,String mirrorOf,String layouts,String url)	4
		void testLayoutPattern()	1
		void testMirrorLayoutConsideredForMatching()	10

2.2 MethodLocator细粒度软件缺陷定位方法

本文提出了 MethodLoactor 方法,以缺陷报告 br_i(包括缺陷报告中的总结内容 summary、描述内容 discription 和评论内容 comment)作为查询,以 $M=\{m_1,\ldots,m_k\}$(M 表示所有源文件中方法的集合,m_j 表示其中的一个方法)作为查询对象.使用基于词向量 word2vec 的表示方法[9]对缺陷报告和源代码方法体内容进行向量表示,并利用夹角余弦计算缺陷报告和源代码方法之间的相似度,进而对查询结果进行排序.

表 1 中给出了 MethodLocator 方法对于 Maven 项目缺陷编号为 MNG-4367 的缺陷的定位效果,可以看到,6 种被修改的方法分别出现在结果排序的第 3 位、第 6 位、第 9 位、第 4 位、第 1 位、第 10 位.

图 2 展示了本文提出的错误定位方法 MethodLocator 的总体架构.如图 2 中步骤③~步骤⑤所示,当新的缺陷报告被提交时,MethodLocator 将其视为查询并计算源代码库中的各方法与该缺陷报告的余弦相似度;从源代码方法的查询中返回定位到的相关方法的排名;最后,MethodLocator 按相似度降序排列所返回的方法,以定位可能导致该缺陷的方法.

考虑到方法级别的方法体(包括方法名称和方法体内容)的内容相对于缺陷报告来说较少,长度也较短,在查询匹配时往往带来了大量的稀疏性.在对近几年关于短文本扩充和查询扩充相关主题研究成果[26−29]进行调研的基础上,本文提出了一种方法体短文本扩充方法.它根据方法体之间的相似性,利用其他方法体对当前方法

体短文本进行扩充.MethodLocator 大致可以分为方法体扩充和相似度计算两个阶段,其具体细节见第 2.3 节和第 2.4 节.

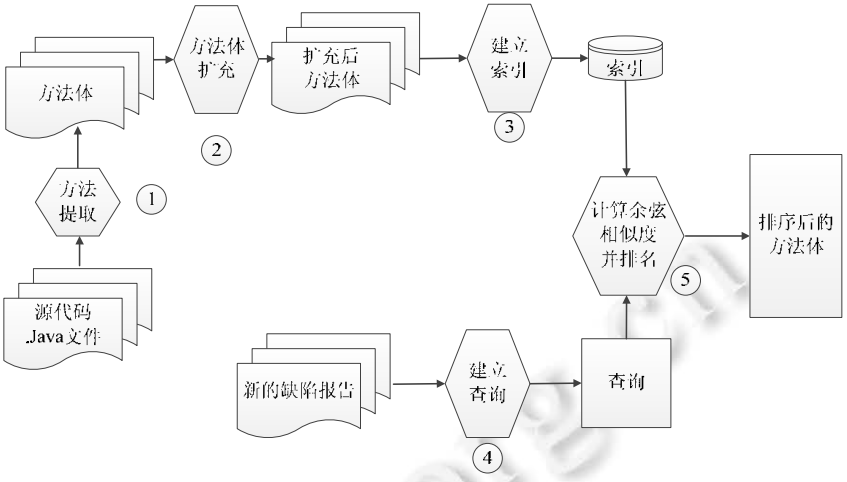


Fig.2 Overall structure of MethodLocator

图 2 MethodLocator 的总体结构图

2.3 方法体向量表示及方法体扩充

图 3 详细解释了图 2 中的方法体提取①和方法体扩充②部分.如图 3 中第 I 部分所示,需要对方法内容进行预处理.首先,从源代码文件中提取方法体,这里,通过抽象语法树(abstract syntax tree,简称 AST)来实现对源代码文件的解析,对于从源代码.Java 文件中提取出的一个方法体,记为 $m_i(1\leqslant i\leqslant n, n$ 为源代码中方法体的总数);然后,对每个方法体进行文本预处理,包括依据 Java 编程驼峰命名规则从方法名中分离出英文单词(Saha 等人[13]的研究表明,将方法名进行分离,对于基于信息检索的缺陷定位方法来说,可以提高缺陷定位的准确率)、去掉停用词、去掉 Java 保留关键词、去掉各种符号,得到预处理后的方法 m_i'; 最后,对 m_i' 进行向量表示,并将对方法 m_i 预处理后的向量表示为 m_i^*.

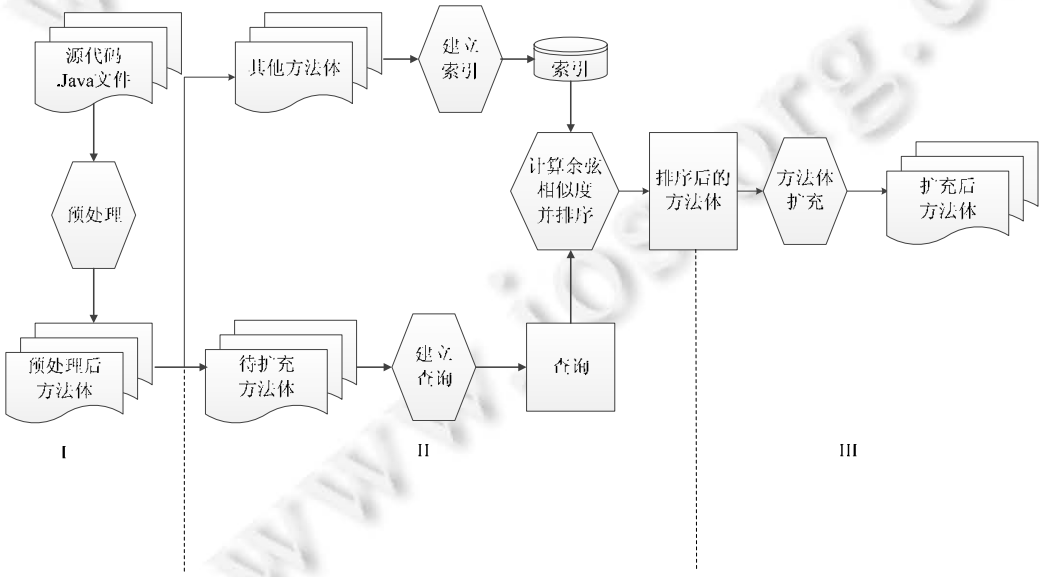


Fig.3 Method body pretreatment

图 3 方法体预处理

2.3.1 方法体向量表示

TF-IDF[30]是一种统计方法,用以评估一个字词对一个文件集或一个语料库中的其中一份文件的重要程度.字词的重要性随着它在文件中出现的次数呈正比增加,但同时会随着它在语料库中出现的频率呈反比下降.虽然TF-IDF能够体现出各个词在文档中的重要程度,但是使用TF-IDF对文档进行向量表示没有考虑文档中词之间的顺序问题,句子中词之间没有联系,会丢失很重要的信息.结合本文实际问题,源代码中方法体内容较少,包含词的数量比较少,以Maven项目中方法体boolean equals(Object obj)为例,经过预处理后,该方法体只包含7个词.使用TF-IDF表示后,得到类似于这样形式的向量表示($a_1,a_2,\ldots,a_7$,0,...,0,0,0,0,0,0),其中包含了5 494个0项(词典中词根总数为5 501),所以只使用TF-IDF对方法体进行向量表示,还会使得向量表示具有很大的稀疏性.

将词映射到一个新的空间中,并以多维的连续实数向量进行表示,叫做“Word Represention”或“Word Embedding”,其最大的贡献就是使相关或者相似的词在距离上更接近了.本文使用Word2Vec(word2vec:https://deeplearning4j.org/word2vec)模型中的Skip-gram模型进行训练,得到词所对应的向量表示.Skip-gram模型[31]根据当前词语来预测上下文的概率,如图4所示.采用基于Word2Vec的方法体向量表示方法一方面可以降低向量表示的纬度,减少向量表示的稀疏性;另一方面挖掘了词之间的关联属性,从而提高了向量语义上的准确度.Ye等人也在文献[20]中使用词向量来解决自然语言表述的缺陷报告和代码表示的源代码文件之间的“词汇鸿沟”问题,提高了软件缺陷定位的准确率.

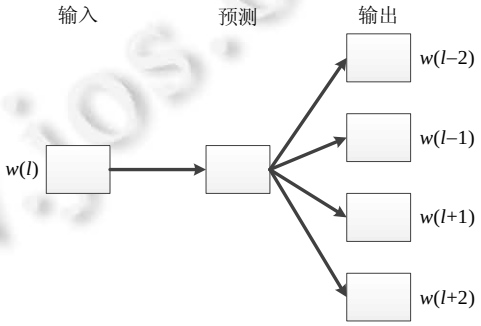


Fig.4 Skip-gram model

图4 Skip-gram模型

唐明等人在文献[9]中提出了一种基于word2vec的文档向量表示方法.该方法结合了词向量和TF-IDF方法,并利用实验验证了该方法的有效性.本文中研究的方法体具有如下特点:包含内容少、文本长度较短.采用基于词向量的表示方法可以挖掘出词之间的关联属性,从而提高了向量语义上的准确度.使用TF-IDF可以考虑单个词对整个方法体的影响力.为了能够挖掘出方法体中更多的信息,方便之后的方法体扩充,本文采用了文献[9]中的文档向量表示方法,即结合词向量和TD-IDF对方法体进行向量表示.

对 m_i' 进行词向量表示生成 m_i^* 的具体过程如下.

- 首先,将所有的缺陷报告 br_i 和方法 m_i' 通过Skip-gram模型[28]训练,得到 m_i' 中每个词项对应的 N 维词向量 w,即 $w=(v_1,v_2,\ldots,v_N)$,其中,v_N 表示在第 N 个维度的值.本文在使用Skip-gram模型[31]进行训练时,结合一般经验[32],将维度 N 的值设置为300.由于方法体中包含的单词个数比较少,所以将Skip-gram模型的最低频率设置为1,将窗口数设为默认值5.
- 然后,MethodLocator选用的基于词向量的表示方法[9]结合了词向量和经典的TF-IDF方法[30],其一方面利用Skip-gram模型[30]计算每个词项的词向量;另一方面,也同时计算每个词项的 *tfidf* 值,也就是分析每个词项的词汇频率 *tf* 和逆文本频率 *idf*,然后计算其 *tfidf* 值.$\{t_1,t_2,\ldots,t_m\}$ 表示从方法体 m_i' 中提取出的词项,m 表示词干总数,则对于单个词项 t_i,其 *tfidf* 计算方式如公式(1)所示.

$$tfidf(t_i)=tf(t_i)\times idf(t_i) \tag{1}$$

公式(1)中,$tf(t_i)$,$idf(t_i)$的计算方式如公式(2)所示.

$$tf_d(t_i)=\log(f_{t_i m_j})+1,\ idf(t_i)=\log\left(\frac{|M|}{m_{t_i}}\right) \tag{2}$$

其中, $f_{t_i m_j}$ 是指方法 m_j 中的词干 t_i 的频率, m_{t_i} 是指包含词干 t_i 的方法数量,$|M|$表示所有源文件中方法数量的总和.对于每种方法 $m_j(1\leqslant j\leqslant|M|)$,其经过词向量表示集合 *tfidf* 处理后的表示形式如公式(3)所示.

$$m_j^*=\sum_{t_i\in m_j'} w_{t_i} tfidf(t_i) \tag{3}$$

这里, w_{t_i} 表示词项 t_i 的词向量.

2.3.2 方法体扩充

图 3 第 II 部分展示了方法之间的相似度计算过程.首先,以第 $k(1\leqslant k\leqslant|M|)$种方法 m_k^* 作为查询,将其他方法 m_i^* $(i\neq k)$视为查询对象;然后,通过计算 m_k^* 和 m_i^* 余弦相似度来对 m_i^* $(i\neq k)$进行排序,由此得到一个大小为$|M|-1$ 的序列;最后,通过对所有$|M|$种方法中的每一种方法的相似方法进行排序,从而得到$|M|$个大小为$|M|-1$ 的序列.

图 3 第 III 部分展示了方法体的扩充过程.这里以第 $k(1\leqslant k\leqslant|M|)$种方法 m_k^* 为例,假设其与其他$|M|-1$ 种方法之间的夹角余弦相似度为 $s_{k,1},s_{k,2},\ldots,s_{k,|M|-1}$.其中,方法 m_k^* 与方法 m_i^* 的夹角余弦相似度如公式(4)所示.

$$s_{k,i}=\cos(m_k,m_i)=\frac{m_k^*\bullet m_i^*}{|m_k^*|\times|m_i^*|} \tag{4}$$

公式(5)表示方法 m_k^* 与其他$|M|-1$ 个方法之间(不包括方法 m_k^*)的夹角余弦相似度的平均值θ_k.

$$\theta_k=\left(\sum_{i=1,i\neq k}^{|M|} s_{k,i}\right)\Big/(|M|-1) \tag{5}$$

对于方法 m_i^*,如果它与方法 m_k^* 的相似度 $S_{k,i}>\theta_k$,则将方法 i 的向量表示 m_i^* 扩充到第 k 个方法 m_k^* 中.为了保持方法 m_k^* 的原始向量在扩充后的方法向量中占主导地位,本文在进行方法扩充的时候添加了一个启发式扩充速率α,用于控制其他方法 m_i^* $(i\neq k)$对 m_k^* 的扩充影响.扩充后的第 k 个方法的向量表示 am_k 如公式(6)所示.

$$am_k=m_k^*+\alpha\sum\nolimits_i s_{k,i}m_i^*, s_{k,i}>\theta_k \tag{6}$$

图 5 显示了扩充的伪代码.

```
Algorithm. Method Expansion.   //目的:方法体扩充
//输入:待扩充方法体、其余方法体、对应相似度、平均相似度、方法体扩充系数
Input:  m_k^*;   //待扩充方法体
        m_i^* (1≤i≤|M|且 i≠k 其余方法体;
        S:S_{k,i}(1≤i≤|M|且 i≠k);   //方法体之间相似度
        θ_k;   //平均相似度
        α.   //方法扩充速率
//输出:扩充后方法体
Output:am_k.   //扩充后的方法体
/*过程:判断待扩充方法体与其余方法体之间的相似度是否大于平均相似度,
  如果大于,则将该方法体扩充到待扩充方法体中.*/
Begin:am_k= m_k^*   //方法体初始化
    For S_{k,i} in S
    If S_{k,i}>θ_k   //判断方法体 m_i^* 与方法 m_k^* 的相似度是否大于平均相似度 am_k=am_k+ αs_{k,i}m_i^*
            //将满足条件的方法体扩充到待扩充方法体中
    Return am_k
End
```

Fig.5 Method body expansion algorithm

图 5 方法体扩充算法

2.4 缺陷报告与方法体相似度计算及排序

在第 2.3 节中,Skip-gram 模型[31]的输入为所有缺陷报告 br_i 和所有方法 m_i'.同理,对缺陷报告的内容也利用词向量进行表示.具体而言,对于第 k 个缺陷报告 br_k,其包含的词项为 $br_k=\{w_{k,1},...,w_{k,|br_k|}\}$.首先,利用词向量将缺陷报告 br_k 中的每个词项 $w_{k,i}$ 表示为 $w_{k,i}=(v_1^{(k,i)},v_2^{(k,i)},...,v_N^{(k,i)})$;然后,将 br_k 中所有的词项量进行聚集,并利用最大池化 MaxPooling 方法[33]在每个特征维度上选取最大值作为 br_k 的表示向量 br_k^* 在该维度上的最大值,即 br_k^* 中的第 i 个维度的值为 $\max\{v_i^{(k,1)},...,v_i^{(k,|br_k|)}\}$.需要说明的是,结合前人的经验及人工观察,本文选取了缺陷报告中的总结、描述和评论这 3 个部分的内容.经过上述处理后,如图 2 中步骤③~步骤⑤所示,MethodLocator 以所有经过处理的方法 am_i 作为查询对象,以缺陷报告 br_k^* 作为查询.通过计算二者的余弦相似度,选取相似度较大的方法视作为修复缺陷 br_k^* 而可能修改的方法.

3 实 验

3.1 标杆数据集建立

为了验证 MethodLocator 方法在实际软件缺陷定位中的有效性,本文选取了 4 个开源软件项目:ArgoUML、Ant、Maven、Kylin,并收集它们的缺陷报告和源代码变更信息,以开展本文的实验.实验数据集的收集步骤具体如下所述.

(1) 获取源代码文件.

这一步的目的是从开源项目代码库中获取实验所需的源代码文件,对于 ArgoUML 项目和 Ant 项目,本文利用 SVN 工具来获取源代码数据;对于 Maven 项目和 Kylin 项目,本文利用 Git 工具来获取源代码数据.

(2) 建立由缺陷修复引起的文件变更数据集.

首先,利用 SVN 或者 Git 工具将步骤(1)中所收集的源代码文件中所有的.java 文件的 log 日志收集下来,并就每一个.java 文件,在其 log 日志中将 bug_number(缺陷编号,与缺陷报告编号相同)利用 SZZ 算法[34]抽取出来.然后,从 log 日志中获取该 bug_number 对应的.java 文件的当时版本号.随后,从 log 日志中找出该 bug_number 当时版本的前面所有版本(按修改时间由近到远排序).接着,利用 diff 命令比较前面版本与当时版本,选择一个最近的有改动的前面版本作为基础版本.比较当时版本与基础版本的 diff(不同的地方),作为修复该 bug_number 的 bug 所导致的代码变化.然后,利用 AST 抽象语法树对每个基础版本源代码文件和当时版本源代码文件进行解析,提取出所有的 method,并查找 diff 结果代码行所属于的 method.最后,将 bug_number、修改的 Java 文件、修改的代码行以及修改的 method 集中起来,建立一个数据集.

(3) 缺陷报告获取.

本文分别从这 4 个软件项目的缺陷跟踪系统中获取到对应的缺陷报告.结合前人的经验及人工观察[2],本文选取了缺陷报告的总结(summary)、描述(description)和评论(comment)这 3 个字段的内容作为缺陷报告的自然语言描述内容.为了保证实验的可重复性和可验证性,本实验从全部缺陷报告中选出缺陷变更记录可追踪性良好的部分缺陷报告作为实验数据,即通过缺陷编号(bug_number),能够准确地对应缺陷报告以及步骤(2)中的源代码修改记录.表 2 展示了本文收集到的 4 个开源项目的各类数据的信息.

Table 2 Information on experimental data

表 2 实验数据信息

项目名称	文件数量	Method 数量	缺陷报告数量	缺陷报告时间
Ant	1 233	11 805	230	2000/01−2014/01
Maven	898	6 459	491	2004/08−2016/10
Kylin	996	7 744	323	2015/02−2016/08
ArgoUML	1 870	12 176	751	2001/01−2014/10

这里需要说明的是,在步骤(2)中,为了探明对某一个缺陷号 bug_number 所做出的在源代码中的具体修改,

本文将源代码文件的当时版本和之前的所有源代码版本进行了 diff 比较操作.这样做的原因是,在某些源代码的提交过程中,仅仅是对源代码的注释或者在版本控制系统的批注信息做了添加和变更操作,而未对源代码本身做出实质的变更.因此,当在源代码文件中得到一个缺陷号之后,我们需要追踪离当时版本最近的一次的软件源代码变更,并将该变更视作为修复该缺陷所做出的变更.

3.2 Word2vec训练

本文利用 deeplearning4j(deeplearning4j:https://deeplearning4j.org/word2vec)中的 Word2Vec 完成各项目中的词的训练及词向量的获得.训练基本情况见表 3.

Table 3 Basic information of Word2Vec training

表 3 Word2Vec 训练的基本信息

项目名称	训练文本行数	计算机相关信息	词数	模型	时间消耗(ms)
Ant	12 035	Backend used: [CPU]; OS: [Windows 7]; Cores: [4]; Memory: [0.8GB]	7 236	SkipGram	8 846
Maven	6 950	Backend used: [CPU]; OS: [Windows 7]; Cores: [4]; Memory: [0.8GB]	5 501	SkipGram	7 145
Kylin	8 067	Backend used: [CPU]; OS: [Windows 7]; Cores: [4]; Memory: [0.8GB]	3 936	SkipGram	7 774
ArgoUML	12 927	Backend used: [CPU]; OS: [Windows 7]; Cores: [4]; Memory: [0.8GB]	11 822	SkipGram	16 541

3.3 基准方法

为了论证 MethodLocator 方法在缺陷定位方面的实际性能,本文选取了两种基准方法进行对比实验,包括 BugLocator[11]和 BLIA 1.5[25].Zhou 等人提出的 Buglocator[10]是最近提出的静态缺陷定位方法中比较典型和广为接受的文件级别的定位方法.由于本文的关注点在于方法级别的软件缺陷定位,因此,我们将 BugLocator 方法的定位粒度由原始的文件级别调整到方法级别,而其基本的计算过程得以完全重复,即由以前的使用源代码文件建立索引变更为使用方法体建立索引;由考察相似缺陷报告修改的源代码文件情况变更为考察相似缺陷报告修改的方法体情况.在方法级别的基于信息检索的静态缺陷定位方面,目前仅有 Youm[25]提出的 BLIA 1.5 方法可用于实际比较.因此,本文同时也选择了 Youm 的 BLIA 1.5 作为基准方法,用于论证 MethodLocator 的有效性.BLIA 1.5 定位方法针对特定的缺陷报告,首先对可能作出变更的源代码文件进行排序,然后选取排序靠前的文件对它们中的方法做进一步的排序,从而实现方法级别的定位.也就是说,BLIA 1.5 定位方法过滤掉了排名较后的文件中的方法.BLIA 1.5 方法中有 4 个参数可以对定位效果进行控制.经过反复实验,本文确定了对应的各实验数据的参数设置,具体设置如下:Ant(α=0.3,β=0.2,γ=0.4,k=120),Maven(α=0.2,β=0.2,γ=0.3,k=120),Kylin(α=0.0,β=0.2,γ=0.5,k=120),ArgoUML(α=0.3,β=0.0,γ=0.5,k=120).

3.4 评价指标

为了论证本文提出的缺陷定位方法的有效性及意义,本文选择前 N 排名(top N rank)、平均准确率(mean average precision,简称 MAP)和平均倒数排名(mean reciprocal rank,简称 MRR)这 3 个指标进行实验结果比较.对于某个项目的第 i 个缺陷报告 br_i,为修复该缺陷,实际修改的方法体的集合为 $m(f_j^{br_i})$.在对本文提出的方法进行评价时,需利用实际修改的方法体集合 $m(f_j^{br_i})$ 中所有方法体在本文推荐结果列表中出现的位置进行相应的计算,以此来评价本文提出的方法的优劣.

1) 前 N 排名(top N rank)

它表示缺陷报告对应做出变更的方法体出现在返回结果的前 N(N=1,5,10)位中的数量的比率.使用该度量方法,对于给定的缺陷报告,如果前 N 个查询结果包含至少 1 个修复缺陷的方法体,就认为缺陷被准确定位.Top N Rank 度量值越大,说明缺陷定位方法的定位性能就越好.

2) 平均准确率值(mean average precision,简称 MAP)

它表示对所有缺陷报告进行源代码定位后的准确率的平均值.MAP 值反映了缺陷定位方法在全部缺陷上

准确定位所有需要修改的源代码的单值指标.缺陷定位方法检索出来的需要修改的源代码方法体越靠前(rank越高),MAP 就越大;反之,如果缺陷定位方法没有检索出需要修改的源代码方法体,则准确率默认为 0.其中,单个缺陷的平均精度(表示为 AvgP)如公式(7)所示.

$$AvgP=\frac{1}{|R|}\sum_{k=1}^{|R|}\frac{k}{rank_k} \tag{7}$$

其中,R 表示一次缺陷定位中所能正确定位的源代码方法体排序的集合,$|R|$表示正确定位的源代码方法体个数,$rank_k$ 表示第 k 个正确的源代码方法体的排名.针对所有的缺陷报告的 MAP 如公式(8)所示.

$$MAP=\sum_{j=1}^{|Q|}\frac{AvgP_j}{|Q|} \tag{8}$$

其中,Q 为缺陷报告的集合,$|Q|$表示 Q 中缺陷报告的数目,$AvgP_j$ 表示第 j 个缺陷报告的平均精度值.

3) 平均倒数排名(mean reciprocal rank,简称 MRR)

表示相关源代码方法体的位置倒数的平均值,MRR 越高,说明算法的准确率越高.MRR 计算公式如下.

$$MRR=\frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{9}$$

其中,Q 为缺陷报告的集合,$|Q|$表示 Q 中缺陷报告的数目,$rank_i$ 表示定位出的与第 i 个缺陷报告相关的方法体最靠前排名的位置.

4 实验结果及分析

为了全面评价 MethodLocator 在方法级别的软件缺陷定位上的性能,本文设置了以下 3 个研究问题,对 MethodLocator 的参数设置以及基准方法进行了全面的分析.

- 研究问题 1:在 MethodLocator 中,方法体扩充速率α的变化对软件缺陷定位的性能有何影响?
- 研究问题 2:当把文件级别的缺陷定位方法(BugLocator)运用于方法级别缺陷定位时,其性能与本文所提出的 MethodLocator 相比,孰优孰劣?
- 研究问题 3:本文提出的 MethodLocator 与现有的方法级别的缺陷定位方法 BLIA 1.5 相比,孰优孰劣?

4.1 对研究问题1的分析

MethodLocator 中,方法体扩充系数α变化对定位效果的影响如图 6~图 8 所示,其中,图 6 显示了α变化对 MAP 的影响,图 7 显示了α变化对 MRR 的影响,图 8 显示了α变化对 ToP-N 的影响.本文实验设置α从 0 到 1 变化,每次增加 0.05.当α=0 时,显示的就是对方法体不进行扩充的定位效果.

从图 6~图 8 展示的效果可以看出:

- 首先,在选定的 4 个项目中,当α=0.05 时取得最优效果;当α>0.05 时,定位效果逐渐变差,MAP、MRR 和 ToP-N 值逐渐降低.当α的值增大到一定程度之后,Top-N 指标、MAP 指标和 MRR 指标的性能都会降低到α=0 对应性能水平的下方.也就是说,当α的取值较大时,扩充的效果明显不如不扩充的效果.对于这个结果的解释是,当α较小时,其他方法 m_i^* 对当前方法 m_k^* 的扩充表示向量 am_k 形成了有益补充;当α较大时,其他方法 m_i^* 给当前方法 m_k^* 的扩充表示向量 am_k 带来了大量噪声.这也说明,在对方法体进行扩充时,在增强原方法体信息表示的同时也会引入一些噪声干扰.α在一定的取值范围内,增强效果大于干扰效果,这样就使定位效果得到提升;当α取值离开该范围时,干扰效果大于增强效果,这样就会使定位效果变差.在 MethodLocator 中,方法体扩充时如何减少干扰信息是一个难点.方法体扩充不可避免地会引入进干扰信息,α值的作用就是尽量控制干扰信息的影响.α值与定位效果的好坏息息相关.对于不同的方法体,其α值可能也会不同.本文在进行实验时,将α值作为定值,以 0.05 为步长进行实验.关于如何更为精确和自动化地设定α的取值,本文会在未来的工作中进一步研究.
- 其次,就 4 个项目来说,在所有的度量指标 Top-N、MAP 和 MRR 上,MethodLocator 在 Ant 项目上的表

现最好,而在其余 3 个项目 ArgoUML、Maven 和 Kylin 上的表现不相上下.经过对实验数据的分析,我们发现,修复一个缺陷,在 Ant 项目中平均要修改 4 个方法体;在 Kylin 项目中平均要修改 8 个方法体;Maven 项目中平均要修改 5.4 个方法体;Argouml 项目中平均要修改 6.2 个方法体.就修复一个缺陷所要修改方法体的数量方面,明显 Ant 项目需要修改的数量最少,这在一定程度上降低了缺陷定位的难度,使得 MethodLocator 在 Ant 项目上具有较好的表现.

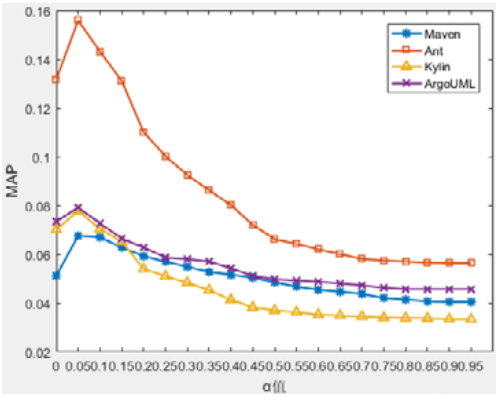


Fig.6 Effect of expansion coefficient α on MAP value in MethodLocator

图 6 MethodLocator 中,扩充系数α对 MAP 值的影响

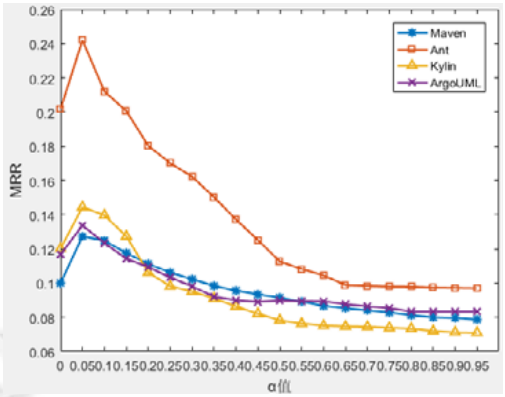


Fig.7 Effect of expansion coefficient α on MRR value in MethodLocator

图 7 MethodLocator 中,扩充系数α对 MRR 值的影响

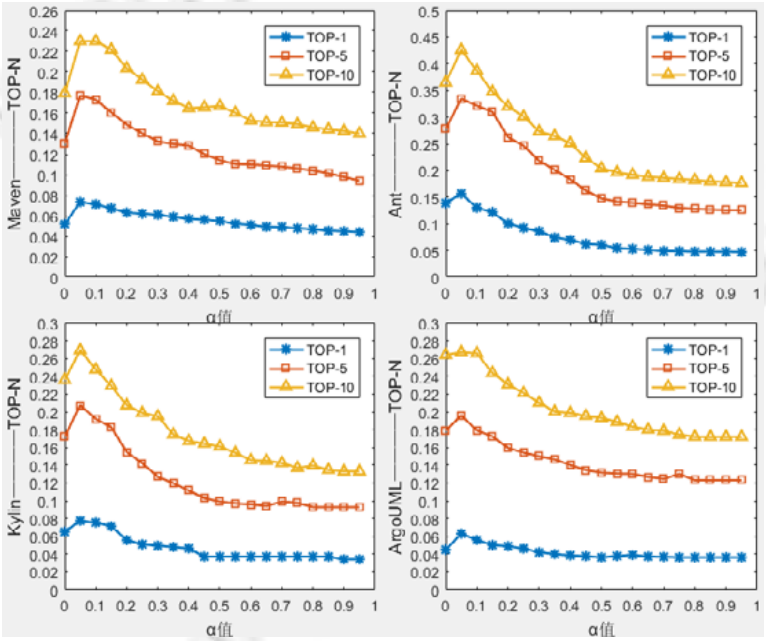


Fig.8 Effect of expansion coefficient α on ToP-N value in MethodLocator

图 8 MethodLocator 中,扩充系数α对 ToP-N 值的影响

4.2 对研究问题2的分析

本文将传统的文件级别的缺陷定位方法 BugLocator 应用到方法级别进行实验,以考察文件级别定位方法

被运用到方法级别定位时的表现.当α=0.05 时,与本文提出的方法 MethodLocator 相比,二者的不同之处主要在于,BugLocator 对方法体进行向量表示时直接采用 tfidf 的向量表示方法,且没有对方法体进行扩充;MethodLocator 在对方法体进行向量表示时,采用将 word2vec 和 tfidf 相结合的方法,并且对方法体进行扩充.表 4 显示了 MethodLocator 和 BugLocator 在本文选择的 4 个项目上的实验结果.

Table 4 MethodLocator and BugLocator comparison of experimental results

表 4 MethodLocator 和 BugLocator 实验结果对比

项目名称	方法名称	ToP-1	ToP-5	ToP-10	MAP	MRR
Ant	BugLocator	0.139	0.278	0.365	0.134	0.210
	MethodLocator, α=0.05	**0.157**	**0.335**	**0.426**	**0.156**	**0.242**
	MethodLocator, α=0	**0.139**	**0.278**	**0.366**	0.132	0.202
Maven	BugLocator	0.053	0.138	0.185	0.054	0.101
	MethodLocator, α=0.05	**0.073**	**0.177**	**0.230**	**0.068**	**0.127**
	MethodLocator, α=0	0.052	0.130	0.180	0.051	0.100
Kylin	BugLocator	0.068	0.176	0.241	0.071	0.128
	MethodLocator, α=0.05	**0.077**	**0.207**	**0.269**	**0.078**	**0.144**
	MethodLocator, α=0	0.065	0.172	0.236	0.070	0.120
ArgoUML	BugLocator	0.046	0.183	**0.273**	0.073	0.118
	MethodLocator, α=0.05	**0.063**	**0.196**	0.267	**0.079**	**0.134**
	MethodLocator, α=0	0.045	0.179	0.264	**0.073**	0.117

从表中可以看到,MethodLocator 具有比 BugLocator 更好的效果.具体来说,在 MAP 指标上,MethodLocator 较之 BugLocator 在 4 个项目上分别提高了 16.4%、25.9%、9.9%、8.2%;在 MRR 指标上,MethodLocator 较之 BugLocator 在 4 个项目上分别提高了 15.2%、25.7%、12.5%、13.6%;在 TOP-*N* 指标上,以 ToP-1 为例,MethodLocator 较之 BugLocator 在 4 个项目上分别提高了 12.9%、37.7%、13.2%、37.0%.方法体相对于源代码文件来说,文本内容及可包含信息明显较少,对于传统文件级方法运用到方法级别定位时,直接对方法体进行向量表示并进行定位,就会因为方法体信息表示不足而取得较差的结果.

当α=0 时,表 4 中展示的即为不对方法体进行扩充时的缺陷定位效果.可以发现,在 20 个指标中,有 3 个是与 BugLocator 相同的,只有在 1 个指标上比 BugLocator 表现好.所以,只使用新的向量表示方法而不对方法体进行扩充,MethodLocator 缺陷定位效果并没有优于基准方法,进一步显示出方法体扩充的重要性.关于采用新的向量表示方法与方法体扩充对准确率的影响更为详细的测算问题,则需要设置更为精确的对比实验来探讨.

4.3 对研究问题3的分析

BLIA 1.5 方法是一种综合了多方面信息源进行方法级缺陷定位的方法.从文献[25]中各信息源的影响分析中可以看到,如果一个缺陷报告包含有缺陷的堆栈信息,那么在缺陷定位中,堆栈信息的分析将起到主要作用,其次才是源代码与缺陷报告之间的相似性;如果一个缺陷报告中不包含堆栈信息,在缺陷定位时,源代码与缺陷报告之间的相似性则起主要作用.与本文所提出的 MethodLocator 相比,BLIA 1.5 考虑的信息源较多.但是由于多源信息之间往往会存在相互冲突和噪声情况,因此,BLIA 1.5 在方法级别缺陷定位上的性能需要进一步验证.

从表 5 可以看出,MethodLocator 方法相对于 BLIA 1.5 方法在方法级别的软件缺陷定位方面达到了较好的效果.具体来说,在 MAP 指标上,MethodLocator 较之 BLIA 1.5 在 4 个项目上分别提高了 16.4%、23.6%、6.9%、3.9%;在 MRR 指标上,MethodLocator 较之 BLIA 1.5 在 4 个项目上分别提高了 14.7%、24.5%、10.8%、11.7%;在 TOP-*N* 指标上,以 ToP-1 为例,MethodLocator 较之 BLIA 1.5 在 4 个项目上分别提高了 12.1%、32.7%、8.5%、28.6%.尽管 BLIA 1.5 方法单独考虑了缺陷报告中的堆栈信息,将其与总结、描述和评论文本中的自然语言文本分开处理,但是首先,在本文所考察的项目中,所有缺陷报告中包含堆栈信息的缺陷报告所占比例并不大,普遍为 5%~10%之间,这些堆栈信息显然并不能在缺陷定位的结果中起到决定性的作用;其次,即使缺陷报告中含有堆栈信息,然而,由于堆栈信息所包含信息往往从最表面调用的函数代码追踪到最深层出现问题的源代码,整个堆栈轨迹的路径较长,从而导致对可能发生缺陷的源代码的误判.例如,在本文考察的 ArgoUML 项目缺陷报告 4 173 中,它包含的堆栈信息由表及里的追踪到了 11 个类的 14 个函数.也就是说,11 个类和 14 个函数以及它们

依赖的类和函数都可能是造成该缺陷的原因.进一步来说,通过本文调研发现,堆栈信息一般在文件级别的粗粒度缺陷定位中有较好的效果[14,16].但是对于方法级别的细粒度定位,由于堆栈信息在引起缺陷的方法的指向性信息上弱化,因此并未见利用其改善缺陷预测或者定位的研究成果.由于以上诸多原因,在本文的实验中,堆栈信息并不能显著提高方法级别的缺陷定位的效果.

Table 5 MethodLocator and BLIA 1.5 comparison of experimental results

表 5 MethodLocator 和 BLIA 1.5 实验结果对比

项目名称	方法名称	ToP-1	ToP-5	ToP-10	MAP	MRR
Ant, α=0.05	BLIA 1.5	0.140	0.308	0.405	0.134	0.211
	MethodLocator	**0.157**	**0.335**	**0.426**	**0.156**	**0.242**
Maven, α=0.05	BLIA 1.5	0.055	0.136	0.183	0.055	0.102
	MethodLocator	**0.073**	**0.177**	**0.230**	**0.068**	**0.127**
Kylin, α=0.05	BLIA 1.5	0.071	0.184	0.260	0.072	0.130
	MethodLocator	**0.077**	**0.207**	**0.269**	**0.078**	**0.144**
ArgoUML, α=0.05	BLIA 1.5	0.049	0.185	**0.274**	0.076	0.120
	MethodLocator	**0.063**	**0.196**	0.267	**0.079**	**0.134**

5 结 论

对于任何软件项目,软件缺陷的出现都是不可避免的.为了提高软件质量和用户满意度,必须快速而及时、准确而有效地修复软件在实际使用中的缺陷.面对一个缺陷报告,软件开发人员需要找到缺陷发生的位置并修复该缺陷,这是一项费时、费力的任务.为了帮助软件开发人员能快速找到缺陷发生的位置,本文提出了一种基于信息检索的方法级别的缺陷定位方法 MethodLocaor.不同于传统的文件级别的定位方法,该方法旨在缩小缺陷定位的粒度而尽量减少缺陷修复人员的修复工作量.通过词向量方法和 TF-IDF 方法,MethodLocaor 实现了源代码方法的初步向量表示.通过方法体扩充方法,MethodLocaor 解决了源代码方法在文本表示上的稀疏性.在 4 个实际项目中的实验表明,MethodLocaor 的定位效果受到方法体扩充系数α的影响,在本文实验的数据集上,α在 0.05 处定位效果最优;与传统文件级定位方法 BugLocator 应用到方法级别的效果相比,MethodLocaor 具有更好的定位性能;与方法级定位方法 BLIA 1.5 方法相比,MethodLocaor 有更好的表现.

在未来的工作中,我们将综合考虑缺陷报告和源代码方法的丰富信息来进一步提高 MethodLacator 在方法级别上的缺陷定位效果.具体来说,在缺陷报告向量表示方面,我们将考虑缺陷报告提交人员的级别、缺陷报告提交时间、超文本链接信息、堆栈信息和代码信息等;在源代码方法方面,我们将考虑方法的圈复杂度、方法之间的依赖关系、开发人员版本信息等.目的是研究有关缺陷报告和源代码方法的全方位信息,并利用其来进行软件方法级别的缺陷定位,以期得到效果更好的缺陷定位方法.

References:

[1] Zhao Y, Leung H, Yang Y, Zhou Y, Xu B. Towards an understanding of change types in bug fixing code. Information & Software Technology, 2017,86:37–53.

[2] Wu W, Zhang W, Yang Y, Wang Q. DREX: Developer recommendation with *K*-nearest-neighbor search and expertise ranking. In: Proc. of the Asia Pacific Software Engineering Conf. Ho CHI Minh, DBLP, 2011. 389–396.

[3] Zhang W, Wang S, Wang Q. BAHA: A novel approach to automatic bug report assignment with topic modeling and heterogeneous network analysis. Chinese Journal of Electronics, 2016,25(6):1011–1018.

[4] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In: Proc. of the Joint Meeting of the European Software Engineering Conf. and the ACM Sigsoft Symp. on the Foundations of Software Engineering. ACM Press, 2009. 111–120.

[5] Lukins SK, Kraft NA, Etzkorn LH. Bug localization using latent dirichlet allocation. Information & Software Technology, 2010, 52(9):972–990.

[6] Youm KC, Ahn J, Kim J, Lee E. Bug localization based on code change histories and bug reports. In: Proc. of the Asia-Pacific Software Engineering Conf. 2015. 190–197.

[7] Naish L, Hua JL, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans. on Software Engineering & Methodology, 2011,20(3):1−32.

[8] Wang X, Zhang W, Wang Q. Two-phase bug localization method based on defect repair history. Computer Systems & Applications, 2014,23(11):99−104 (in Chinese with English abstract).

[9] Tang M, Zhu L, Zou XC. Document vector representation based on Word2Vec. Computer Science, 2016,43(6):214−217 (in Chinese with English abstract).

[10] Poshyvanyk D, Gueheneuc YG, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. on Software Engineering, 2007,33(6):420−432.

[11] Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proc. of the ICSE 2012. 2012. 14−24.

[12] Moreno L, Treadway JJ, Marcus A, Shen W. On the use of stack traces to improve text retrieval-based bug localization. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 151−160.

[13] Saha RK, Lease M, Khurshid S, Perry DE. Improving bug localization using structured information retrieval. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM Press, 2015. 345−355.

[14] Saha RK, Lawall J, Khurshid S, Perry DE. On the effectiveness of information retrieval based bug localization for C programs. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 161−170.

[15] Wang S, Lo D. Version history, similar report, and structure: Putting them together for improved bug localization. In: Proc. of the Int'l Conf. on Program Comprehension. ACM Press, 2014. 53−63.

[16] Rahman F, Posnett D, Hindle A, Barr E, Devanbu P. BugCache for inspections: Hit or miss? In: Proc. of the ACM Sigsoft Symp. on the Foundations of Software Engineering (SIGSOFTSoft/FSE 2011). DBLP, 2011. 322−331.

[17] Le TDB, Oentaryo RJ, Lo D. Information retrieval and spectrum based bug localization: Better together. In: Proc. of the Joint Meeting on Foundations of Software Engineering. ACM Press, 2015. 579−590.

[18] Wong CP, Xiong Y, Zhang H, Hao D, Zhang L, Mei H. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE Computer Society, 2014. 181−190.

[19] Ye X, Bunescu R, Liu C. Learning to rank relevant files for bug reports using domain knowledge. In: Proc. of the ACM Sigsoft Int'l Symp. on Foundations of Software Engineering. ACM Press, 2014. 689−699.

[20] Ye X, Shen H, Ma X, Bunescu R, Liu C. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proc. of the IEEE/ACM Int'l Conf. on Software Engineering. IEEE, 2016. 404−415.

[21] Giger E, D'Ambros M, Pinzger M, Gall HC. Method-level bug prediction. In: Proc. of the ESEM 2012. 2012. 171−180.

[22] Yuan Z, Yu LL, Liu C. Bug prediction method for fine-grained source code changes. Ruan Jian Xue Bao/Journal of Software, 2014, 25(11):2499−2517 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4559.htm [doi: 10.13328/j.cnki.jos.004559]

[23] Hata H, Mizuno O, Kikuno T. Bug prediction based on fine-grained module histories. In: Proc. of the Int'l Conf. on Software Engineering. IEEE, 2012. 200−210.

[24] Wen M, Wu R, Cheung SC. Locus: Locating bugs from software changes. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2016. 262−273.

[25] Youm KC, Ahn J, Lee E. Improved bug localization based on code change histories and bug reports. Information & Software Technology, 2016,82:177−192.

[26] Phan XH, Nguyen LM, Horiguchi S. Learning to classify short and sparse text & Web with hidden topics from large-scale data collections. In: Proc. of the WWW 2008. 2008. 91−100.

[27] Quan X, Liu G, Lu Z, Ni X, Liu W. Short text similarity based on probabilistic topics. Knowledge and Information Systems, 2010, 25(3):473−491.

[28] Chen M, Jin X, Shen D. Short text classification improved by learning multi-granularity topics. In: Proc. of the Int'l Joint Conf. on Artificial Intelligence (IJCAI 2011). Barcelona, 2011. 1776−1781.

[29] Ma HF, Zeng XT, Li XH, Zhu ZQ. Short text feature extension method of improved frequent term set. Computer Engineering, 2016,42(10):213−218 (in Chinese with English abstract).

[30] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of Int'l Conf. on 27th Conf. on Neural Information Processing Systems (NIPS 2013). Lake Tahoe, 3111−3119.

[31] Joachims T. A probabilistic analysis of the rocchio algorithm with TFIDF for text categorization. In: Proc. of the 14th Int'l Conf. on Machine Learning. Morgan Kaufmann Publishers Inc., 1997. 143−151.

[32] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. In: Proc. of Workshop at 2013 Int'l Conf. on Learning Representation. 2013.

[33] Boureau YL, Bach F, Lecun Y, Ponce J. Learning mid-level features for recognition. In: Proc. of the Computer Vision and Pattern Recognition. IEEE, 2010. 2559−2566.

[34] Zimmermann T, Zeller A. When do changes induce fixes? In: Proc. of the Int'l Workshop on Mining Software Repositories. ACM Press, 2005. 1−5.

附中文参考文献:

[8] 王旭,张文,王青.基于缺陷修复历史的两阶段缺陷定位方法.计算机系统应用,2014,23(11):99−104.

[9] 唐明,朱磊,邹显春.基于 Word2Vec 的一种文档向量表示.计算机科学,2016,43(6):214−217.

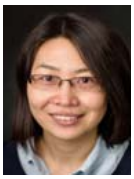
[22] 原子,于莉莉,刘超.面向细粒度源代码变更的缺陷预测方法.软件学报,2014,25(11):2499−2517. http://www.jos.org.cn/1000-9825/4559.htm [doi: 10.13328/j.cnki.jos.004559]

[29] 马慧芳,曾宪桃,李晓红,朱志强.改进的频繁词集短文本特征扩展方法.计算机工程,2016,42(10):213−218.

张文(1981－),男,湖北洪湖人,博士,教授,博士生导师,CCF 专业会员,主要研究领域为软件工程,数据挖掘.

杜宇航(1994－),男,硕士生,主要研究领域为软件工程,数据挖掘.

李自强(1994－),男,硕士生,主要研究领域为软件工程,数据挖掘.



杨叶(1977－),女,博士,教授,博士生导师,主要研究领域为软件工程.