软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(6):1418−1434 [doi: 10.13328/j.cnki.jos.005225]

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

基于关键类判定的代码提交理解辅助方法*

黄 袁[1,2], 刘志勇[1,2], 陈湘萍[2,3], 熊英飞[4,5], 罗笑南[1,2]

[1](中山大学 数据科学与计算机学院,广东 广州 510006)

[2](国家数字家庭工程技术研究中心,广东 广州 510006)

[3](中山大学 先进技术研究院,广东 广州 510006)

[4](北京大学 信息科学技术学院 软件研究所,北京 100871)

[5](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 陈湘萍, E-mail: chenxp8@mail.sysu.edu.cn

摘 要: 软件代码提交是最重要的软件版本演化数据之一,被广泛应用于软件审查和软件理解中.对于程序员,提交的理解难度随着受影响的类数量、修改的代码量的增加而增加.通过对大量数据的分析发现:识别出提交中核心的修改类(关键类)以及为了完成这个核心修改所进行的依赖性改动的类(非关键类),能够辅助代码提交的理解.受机器学习技术在分类领域有效性的启发,提出一种基于机器学习的关键类识别方法,将判定提交中的关键类建模为二分类问题(即关键和非关键类),从软件演化过程中产生的海量提交数据中抽取可判别性特征来度量类的关键性.在多个数据集上的实验结果表明:该方法判定关键类的综合准确率达到了87%;相比于开发人员直接理解提交,使用关键类信息提示来辅助理解提交,能够显著提高开发人员的效率和正确率.

关键词: 代码修改;代码修改理解;代码提交;机器学习;可判别特征

中图法分类号: TP311

中文引用格式: 黄袁,刘志勇,陈湘萍,熊英飞,罗笑南.基于关键类判定的代码提交理解辅助方法.软件学报,2017,28(6):1418−1434. http://www.jos.org.cn/1000-9825/5225.htm

英文引用格式: Huang Y, Liu ZY, Chen XP, Xiong YF, Luo XN. Auxiliary method for code commit comprehension based on core-class identification. Ruan Jian Xue Bao/Journal of Software, 2017,28(6):1418−1434 (in Chinese). http://www.jos.org.cn/1000-9825/5225.htm

Auxiliary Method for Code Commit Comprehension Based on Core-Class Identification

HUANG Yuan[1,2], LIU Zhi-Yong[1,2], CHEN Xiang-Ping[2,3], XIONG Ying-Fei[4,5], LUO Xiao-Nan[1,2]

[1](School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510006, China)

[2](National Engineering Research Center of Digital Life, Guangzhou 510006, China)

[3](Institute of Advanced Technology, Sun Yat-Sen University, Guangzhou 510006, China)

[4](Software Engineering Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[5](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

* 基金项目: NSFC-广东联合基金(U1201252); 国家重点研发计划(2016YFB1000101); 国家自然科学基金(61672545, 61672045); 广东科技计划(2015B040403005)

Foundation item: NSFC-Guangdong Joint Fund (U1201252); National Key Research and Development Program of China (2016YFB1000101); National Natural Science Foundation of China Science and Technology (61672545, 61672045); Science and Technology Planning Project of Guangdong Province (2015B040403005)

收稿时间: 2016-07-28; 修改时间: 2016-10-11; 采用时间: 2016-12-22; jos 在线出版时间: 2017-02-20

CNKI 网络优先出版: 2017-02-20 15:14:50, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1514.034.html

Abstract: Code commit is one of the most important software evolution data, and it is widely used in the software review and code comprehension. A commit involving multiple modified classes and code makes the review of code changes difficult. By analyzing a large amount of commit data, this study discovers that identifying the core modified classes in a commit can speed up commit review for developers. Inspired by the effectiveness of machine learning techniques in classification, the paper models the core class identification as a binary classification problem (i.e., core and non-core) and proposes discriminative features from a large number of commits to characterize the core modified classes. The experiments results show that the proposed approach achieves 87% accuracy, and using core class in commit review provides significant improvement than the ones without core class.

Key words: code change; code change comprehension; code commit; machine learning; discriminative feature

随着开源软件及其应用的飞速发展,在互联网上产生了规模巨大、种类丰富的开源软件资源.例如,开源软件库 Sourceforge 上托管着多于 430 000 个开源项目.最初的开源软件数据主要是软件源代码.随着软件生命周期的不断增长,软件版本管理技术被广泛应用于开源软件的代码管理中,这使得开源软件资源也相应地扩充了代码注释、软件代码修改记录、代码漏洞报告以及用户反馈等软件工程过程数据.

软件代码提交(commit)是最重要的软件版本演化数据之一,被广泛应用于软件审查和软件理解中.在软件项目开发中,为了保证代码质量和软件的一致性[1],开发人员所做的代码修改常常需要通过其他开发人员的审阅.另一方面,随着软件开发人员的更替,新加入的开发人员往往需要理解已有的软件代码变化.修改规模较大的提交,可能包含分散在多个类不同位置的多处修改,人工查看并理解其修改逻辑,是个费时费力的过程[2,3].在本文的实验中,编程人员需要花费平均 345 秒的时间去理解平均包含 4 个类的代码提交.

本文致力于帮助程序员理解程序代码中的提交.主要发现是一次提交包括多个修改,影响多个不同的类.在这些修改中,往往存在着一个或者多个被核心修改的类,它是引起提交中其他类修改的节点;而提交中的其他的类多半是为了完成这个核心修改而不得不进行依赖性改动的类.其中,被核心修改的类代表了本次提交重点修改的区域,是此次代码修改的主要对象,本文称为关键类;而提交中那些进行依赖性改动的类称为非关键类.

关键类的存在,可以通过比对一次提交中被修改的类和提交的注释(commit message)看出.我们对 120 个项目中总共 10 万条的代码提交注释进行了分析,发现有 10%的代码提交注释中提到一个或者多个类,然而并不是穷举所有修改涉及的类.这个比例在广泛被用于代码提交相关实验的项目 jHotDraw 和 jMemorize 中达到了 20%以上.在这些提交注释中,被提到的类往往是项目的关键类,而剩下的类是因为修改这些关键类所不得不修改的类.比如,图 1 列举的 5 个提交来自开源项目 jHotDraw 和 jMemorize 等.每个提交中灰色背景的类为当前提交注释信息中提到的类,也是本次提交的关键类.

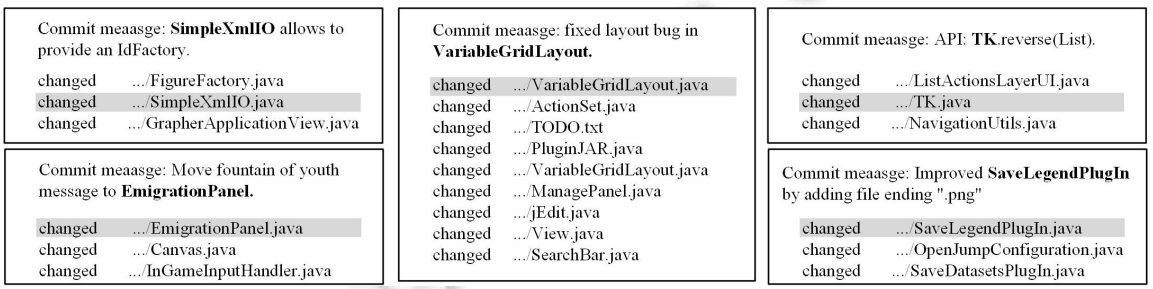


Fig.1 Examples of commits consisting of core class and non-core classes

图 1 提交中的关键类与非关键类

图 2 统计了在提交注释中给出关键类信息的提交比例与提交中受影响的类数量之间的关系.可以看到:随着提交中受影响的类数量增加,提交注释中给出关键类信息的提交比例也随之增加.也就是说,当提交中受影响的类数量越多时,提交的理解难度也越高.为了使后续人员更容易理解代码提交,程序员会更多地给出关键类信息.本文假设:理解关键类对于理解整个提交有着重要作用.因为关键类信息能够引导后续程序员了解阅读重点并安排合理的代码阅读顺序,所以有经验的程序员在书写的代码提交注释中加入了关键类信息.本文的实验也

验证了这一假设,实验结果显示:在理解代码提交时,提示关键类信息可以显著性地减少开发人员理解代码修改的时间.

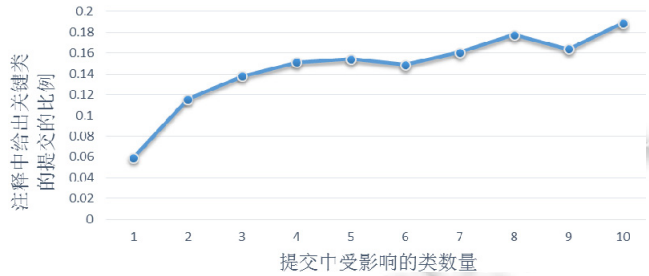


Fig.2 Percentage of commit message containing core class information

图 2 提交注释中给出关键类的提交数量变化趋势

然而,并不是所有的提交注释都给出了关键类.如前所述,包含关键类信息的提交注释只占所有提交注释的10%左右,剩下的提交注释并不包含关键类信息.同时,约 30%的提交注释并不包含任何内容.如果能对这些提交自动提示出关键类的信息,必然显著减轻程序员理解提交的工作量.基于这样的想法,本文提出一种基于机器学习的关键类识别方法——ICC(identifying core-class).由于关键类的判别涉及到软件修改中类、方法、属性以及它们之间的复杂关系,难以将其直接识别出来.已有的开源软件已经提供了大量的带有注释的软件代码提交信息,为机器学习算法提供了数据量足够大的训练集.我们的方法基于有监督的机器学习算法实现.假设关键类的判定问题可以作为一个二分类问题,那么通过训练集学习得出一个分类模型,此模型能够建立特征向量与分类(关键与非关键)之间的映射.具体来说,我们的方法从提交本中提取结构性耦合信息以及代码修改信息作为可判别特征,用来度量类的关键性.类与类之间的结构性耦合信息指明了类实体之间所满足的静态耦合关系;代码修改信息概括了类实体在本次提交中的更新程度及更新类型(新增、删除和修改).我们使用大量在注释中提到的关键类作为训练样本来训练分类模型,训练好的分类模型用于判定提交中每个类的关键性(被判定为关键类或非关键类).

我们在多个数据集上验证了文中提出的方法.实验结果表明:(1) ICC 判定关键类方法的综合准确率达到了87%;(2) 相比于开发人员直接理解提交,使用 ICC 辅助开发人员理解提交的方法,在效率和正确率上都有显著提高.本文的贡献有两点:首先,提出了判定关键类的可判别特征,这使得正确判断关键类成为可能;其次,提出了一套用于关键类判定的评价机制.

第 1 节介绍基本概念.第 2 节介绍关键类判定.第 3 节、第 4~5 节讨论实验设置与实验结果,并加以讨论.第6 节介绍相关工作.第 7 节总结和未来工作.

1 基本概念

本文首先给出了代码提交的定义.为区分提交中被核心修改的类和非核心修改的类,定义了关键类和非关键类.

定义 1(提交(commit,简称 Cm)). 定义为七元组(n,a,d,m,S,K,N),其中,n 为提交版本号,a 是提交作者,d 是提交日期,m 为注释消息,S 为本次提交受影响的类集合,K 为关键类集合,N 为非关键类集合.满足 $K\subseteq S,N\subseteq S$.

定义 2(类(class,简称 C)). 定义为六元组(k,i,o,l,m,mc),其中,k 标示了当前类是否为关键类;i 为入度,指该类在本次提交范围内引用其他类的次数;o 为出度,指该类在本次提交范围内被其他类引用的次数;l 为类中受到修改影响的代码行数量;m 为类中方法数量;mc 为类中受修改影响的方法数量.

定义 3(关键类与非关键类集合($\boldsymbol{K},\boldsymbol{N}$)). 关键类是提交中被核心修改的类,非关键类是为了完成关键类的修改而进行依赖性改动的类.一次提交中可能存在多个关键类(或非关键类),也可能不存在关键类(或非关键类).因此,$\boldsymbol{K}\subseteq\boldsymbol{S},|\boldsymbol{K}|\geqslant 0,\boldsymbol{N}\subseteq\boldsymbol{S},|\boldsymbol{N}|\geqslant 0$.满足 $\boldsymbol{K}\cup\boldsymbol{N}=\boldsymbol{S}\wedge\boldsymbol{K}\cap\boldsymbol{N}=\varnothing$.

2 关键类判定

2.1 特征提取

本节详细介绍了从提交中提取特征的过程.我们从 3 个维度一共提取了 21 种特征,这些特征为:代码耦合特征、代码修改特征以及提交类型特征.所有符号说明及特征描述见表 1 和表 2.

Table 1 List of notations

表 1 符号说明

符号	形式化表示	描述
I_{curr}	$I_{curr}=C.i$	当前类的入度
$I_{\max}$	$I_{\max}=\max\{C_j.i\},1\leqslant j\leqslant\|S\|$	当前提交中所有类中拥有的最大入度
I_{ave}	$I_{ave}=\sum_{j=1}^{\|S\|}C_j.i\Big/\|S\|$	当前提交中所有类的平均入度
I_{zero}	$I_{zero}=boolean(I_{curr}==0)$	当前类的入度是否为 0
O_{curr}	$O_{curr}=C.o$	当前类的出度
$O_{\max}$	$O_{\max}=\max\{C_j.o\},1\leqslant j\leqslant\|S\|$	当前提交中所有类中拥有的最大出度
O_{ave}	$O_{ave}=\sum_{j=1}^{\|S\|}C_j.o\Big/\|S\|$	当前提交中所有类平均的出度
O_{zero}	$O_{zero}=boolean(O_{curr}==0)$	当前类的出度是否为 0
S_{curr}	$S_{curr}=C.l$	当前类修改的语句条数
$S_{\max}$	$S_{\max}=\max\{C_j.l\},1\leqslant j\leqslant\|S\|$	当前提交中所有类中修改最多的语句条数
S_{ave}	$S_{ave}=\sum_{j=1}^{\|S\|}C_j.l\Big/\|S\|$	当前提交中所有类平均修改的语句条数
$M_{methincla}$	$M_{methincla}=C.m$	当前类中方法的总数
$M_{changedmethincla}$	$M_{changedmethincla}=C.mc$	当前类中涉及修改的方法的数量
$M_{methincomm}$	$M_{methincomm}=\sum_{j=1}^{\|S\|}C_j.m$	当前提交中方法的总数
$M_{changedmethincomm}$	$M_{changedmethincomm}=\sum_{j=1}^{\|S\|}C_j.mc$	当前提交中涉及修改的方法的总数
CC_{type}	$CC_{type}=\begin{cases}new\\change\\remove\end{cases}$	当前类的修改类型,取值为新增、修改、删除
$C_{forwardengineering}$	–	当前提交的类型是否为前向工程
$C_{reengineering}$	–	当前提交的类型是否为逆向工程
$C_{correctiveengineering}$	–	当前提交的类型是否为纠错工程
$C_{management}$	–	当前提交的类型是否为非代码修改类型

Table 2 Three types feature

表 2 3 种类型特征

分类	特征	描述
代码耦合特征 (coupling feature)	CF_1	I_{curr}
	CF_2	$I_{curr}/I_{\max}$
	CF_3	I_{curr}/I_{ave}
	CF_4	O_{curr}
	CF_5	$O_{curr}/O_{\max}$
	CF_6	O_{curr}/O_{ave}
	CF_7	if I_{zero}=true, CF_7=1, otherwise, CF_7=0
	CF_8	if O_{zero}=true, CF_8=1, otherwise, CF_8=0
代码修改特征 (modifying feature)	MF_1	$1\leqslant S_{curr}\leqslant 5$, MF_1=1; $6\leqslant S_{curr}\leqslant 9$, MF_1=2; $10\leqslant S_{curr}\leqslant 19$, MF_1=3; $20\leqslant S_{curr}\leqslant 29$, MF_1=4; $30\leqslant S_{curr}$, MF_1=5
	MF_2	$S_{curr}/S_{\max}$
	MF_3	S_{curr}/S_{ave}
	MF_4	$1\leqslant M_{changedmethincla}\leqslant 5$, MF_4=1; $6\leqslant M_{changedmethincla}\leqslant 9$, MF_4=2; $10\leqslant M_{changedmethincla}\leqslant 19$, MF_4=3; $20\leqslant M_{changedmethincla}\leqslant 29$, MF_4=4; $30\leqslant M_{changedmethincla}$, MF_4=5
	MF_5	$M_{changedmethincla}/M_{methincla}$

Table 2 Three types feature (Continued)

表 2 3 种类型特征(续)

分类	特征	描述
代码修改特征 (modifyingfeature)	MF_6	$M_{changedmethincla}/M_{changedmethincomm}$
	MF_7	$M_{changedmethincla}/M_{methincomm}$
	MF_8	if CC_{type}=*new*, MF_8=0; else if CC_{type}=*change*, MF_8=1; else if CC_{type}=*remove*, MF_8=2
提交类型 (commit type feature)	TF_1	if $C_{forwardengineering}$=true, TF_1=1, otherwise, TF_1=0
	TF_2	if $C_{reengineering}$=true, TF_2=1, otherwise, TF_2=0
	TF_3	if $C_{correctiveengineering}$=true, TF_3=1, otherwise, TF_3=0
	TF_4	if $C_{management}$=true, TF_4=1, otherwise, TF_4=0
	CI_1	$1\leqslant\|S\|\leqslant5$, CI_1=1; $6\leqslant\|S\|\leqslant9$, CI_1=2; $10\leqslant\|S\|\leqslant19$, CI_1=3; $20\leqslant\|S\|\leqslant29$, CI_1=4; $30\leqslant\|S\|$, CI_1=5

2.1.1 代码耦合特征提取

根据代码修改传播机制[4],当两个类之间存在结构性耦合关系时,代码修改倾向于从修改发生的起始节点传播到另一个与其有耦合关系的节点.另一方面,提交中非关键类的修改往往是由关键类的修改引起的.即,关键类中的修改向非关键类中传播,且这种修改传播为有向传播.这种传播在代码中反映为关键类与非关键类之间很可能存在结构性耦合关系,且这种耦合关系的方向是确定的.因此,我们可以从提交中类与类之间存在的耦合结构关系出发,提取特征来度量类的关键性.我们称这种特征为代码耦合特征.

具体地,在代码层次,使用入度与出度来表示一个类的结构性耦合特征:入度指当前类调用提交中其他类的次数;出度指当前类被提交中其他类调用的次数.出度和入度表明了耦合关系的方向性.表 2 中,CF_1 和 CF_4 分别表示一个类的入度和出度.CF_1 越大,表示当前类调用提交中的其他类的次数越多;CF_4 越大,表示当前类被提交中的其他类调用的次数越多.$I_{\max}$ 描述了当前提交范围内所有类中拥有的最大入度,$O_{\max}$ 描述了当前提交范围内所有类中拥有的最大出度.因此,CF_2 和 CF_5 是 CF_1 和 CF_4 的归一化形式;CF_3 和 CF_6 描述了当前类的入度和出度与提交中的入度和出度的平均水平的一个比值.CF_7 表示当前类只有出度,没有入度;CF_8 正好相反,表示当前类只有入度,没有出度.通过设置特征 CF_1~CF_8,试图从类的结构性耦合信息方面来区分关键类与非关键类.

2.1.2 代码修改特征提取

不同于 CF_1~CF_8,特征 MF_1~MF_8 是从代码修改的信息来判别关键类与非关键类之间的差异性.由于关键类在一个提交中往往是被核心修改的对象,因此,关键类与非关键类在代码修改方面存在差异性.这种差异可能体现在二者涉及修改的代码量上.代码量的差异可以更细粒度地分为涉及修改的方法数或代码行数.如果仅仅用涉及修改的代码量去度量类的关键性,可能会降低结果的准确性.因此,我们在特征中还引入了代码相对修改量的概念.即:在当前提交范围内,当前类的代码修改量相对于提交中平均代码修改量处于怎样一个水平.为了量化这些差异,我们设置了特征 MF_1~MF_7.

MF_1 表示当前类涉及修改的代码行,MF_1 的值由涉及修改的代码行数所属区间决定,比如,如果当前类修改了 15 行代码,则 MF_1=3;MF_2 是 MF_1 的归一化形式;MF_3 描述了当前类涉及修改的代码行与提交中的平均水平的一个比值;MF_4 表示当前类中涉及修改的方法数量;MF_5 表示当前类中涉及修改的方法数量与该类中方法总数的一个比值;MF_6 表示当前类中涉及修改的方法数量与当前提交中涉及修改的方法之间的比值;MF_7 表示当前类中涉及修改的方法数量与当前提交的方法总数之间的比值.特征 MF_8 表示当前类的修改类型,修改类型包括新增、修改、删除这 3 种类型.

2.1.3 提交类型特征提取

根据代码修改要达到的目标不同,提交中被核心修改的类也会有所不同.例如:以增加新功能为目标的提交,其核心修改的类往往是新增类型的类.因此,如果能够首先判定一个提交的类型,就可以把当前提交降维到一个更小的特征空间内,这样有利于判别模型继续使用代码耦合特征,以及代码修改特征进一步区分某一提交类型下的关键类或非关键类.

本文参考了已有工作[5]中对于提交的分类,将提交分为 4 种类型,即:前向工程、逆向工程、纠错工程及非代码修改类型.此方法通过识别提交注释中的关键字来判定提交的类型.比如,提交注释信息中含有 new,

add,create,implement,initial 等关键字,则判定为前向工程.我们在文中使用相同的方法用于判定提交类型,并设置 TF_1~TF_4 四维相应的特征.提交类型和对应的关键字词根见表 3.此外,提交中受影响的类的数量 S 也被认为是一种特征,即特征 CI_1.

Table 3 Commit types and keywords list

表 3 提交分类及关键字列表

分类	关键字
前向工程 (forward engineering)	implement, add, request, new, test, start, include, initial, introduc, create, increas
逆向工程 (reengineering)	optimiz, adjust, update, delet, remov,chang, refactor, replac, modif, (is, are) now, enhance, improve, design change, renam, eliminate, duplicat, restrutur, simplify, obsolete, rearrang, miss, enhance, improv
纠错工程 (corrective engineering)	bug, issue, error, correct, proper, deprecate, broke
非代码修改类型 (management)	clean, license, merge, release, structure, copyright, documentation, manual, javadoc, comment, migrat, repository, codereview, polish, upgrade, style, formatting, TODO

2.2 机器学习算法及样本优化处理

本文提出的关键类识别方法 ICC 是基于有监督的机器学习算法.有监督的机器学习算法被广泛地应用于各种分类问题中.其原理为:首先,用一个带标签的训练集去建立特征向量与分类之间的映射关系;然后,训练得到的模型用于预测未带标签的特征向量对应的类型.文中采用随机森林算法[6]判定提交中的关键类.随机森林算法的基本思想是:利用多个弱分类器创建一个更强的分类器,其中,每个弱分类器都是利用决策树学习算法在训练集的一个子集上进行训练.

在机器学习算法的应用中,样本的质量是影响准确率的重要因素之一.由于我们的样本来自开源软件提供的提交信息,项目差异大且质量参差不齐,需要对质量低的样本和“坏样本”进行处理.

针对质量低的样本,我们通过观察大量的样本,设置了相应的过滤规则.

1) 过滤掉注释信息中没提到关键类的提交;
2) 过滤掉注释信息中提到的类数量等于其所涉及的类总量的提交;
3) 过滤掉只涉及一个类修改的提交;
4) 过滤掉超过 20 个类修改的提交.此类提交大部分是由多个原子类型的提交合并而成,用于版本更新.此类提交中往往存在多个关键类,其数据特征与其他原子提交中类与类之间的关系存在较大差异,因此将其删除;
5) 过滤掉注释信息过短的提交(<3 个单词).虽然这样的提交注释信息中也可能包含类名,但是由于信息量少,注释本身缺乏语义表达力,很难判断注释中提到的类就是关键类;
6) 过滤掉注释信息过长的提交(>200 个单词).虽然在这样的提交注释中也包含类名,但是注释中罗列了提交中大部分琐碎的修改,不能突出关键类信息,从而很难保证提交注释中出现的类名就是当前提交的关键类.

“坏样本”是指在样本集中同时存在两个特征向量完全相同的样本,且它们其中一个样本被标注为正样本‘1’;而另外一个样本被标注为负样本‘0’.“坏样本”产生的原因可能是因为特征细分的粒度不够,也可能是样本错误造成.“坏样本”对于随机森林算法建立有效的分类模型起到了很大的干扰作用,在第 5 节,我们将进一步讨论“坏样本”的产生过程.因此,必须剔除掉样本集中的“坏样本”.见算法 1.

算法 1. Bad sample optimization algorithm.

1: **Input**: N: The set of negative samples; P: The set of positive samples;
2: T: Total sample set, $T=N\cup P$; C: Total commit set;
3: **Output**: *new_SampleSet*; *new_CommitSet*.
4: *BadSampleOptimization*(N,P,T,C):

```
5:    foreach ps∈P do
6:      foreach ns∈N do
7:          cosineSimilar←cosine(ps.featureVector,ns.featureVector);
8:          if cosineSimilar==1 do
9:          badSample_Set.add(ps); badSample_Set.add(ns);
10:       end if;
11:     end foreach
12:   end foreach
13:   if badSample_Set!=null do
14:     foreach Commit do
15:       if Commit.S⊂badSample_Set do
16:         C.remove(Commit);
17:       end if
18:     end foreach
19:     foreach bs∈badSample_Set do
20:       T.remove(bs);
21:     end foreach
22:   end if;
23:   new_SampleSet←T; new_CommitSet←C;
24: return new_SampleSet && new_CommitSet;
```

算法 1 将正样本 *P*、负样本 *N*、总样本集 *T* 及总的提交集合 *C* 作为输入;将新生成的样本集 *new_SampleSet* 和新的提交集合 *new_CommitSet* 作为输出.对于每一个正样本 *ps* 和负样本 *ns*(第 5 行、第 6 行),算法通过计算向量空间夹角余弦值 cosine 来度量 *ps* 与 *ns* 之间的相似度 cosine*Similar*(第 7 行).如果相似度为 1,则将当前 *ps* 与 *ns* 加入坏样本集合 *badSample_Set* 中(第 8 行~第 10 行);如果 *badSample_Set* 不为空(第 13 行),则对于每个提交,检查其涉及修改的类集合 *S* 是否为 *badSample_Set* 的子集合:如果是,则将当前提交从提交集合中删除(第 14 行~第 18 行),然后再从总样本集 *T* 中删除属于 *badSample_Set* 中的样本(第 19 行~第 21 行).最后,生成并返回新的样本集 *new_SampleSet* 和提交集合 *new_CommitSet* (第 23 行、第 24 行).

3 实验验证

本节以大量在实际开发中产生的提交数据为对象,首先评估 ICC 判定关键类方法的准确率,然后从多个角度研究 ICC 判定关键类方法对于开发人员理解代码提交的有用性.

3.1 数据收集及参数设置

我们从 Sourceforge 的 SVN 库中收集了 120 个开源项目的提交.通过过滤规则,过滤掉不满足条件的提交,得到有效提交的数量为 5 198,我们将 5 198 个提交用于生成样本数据.具体地,提交注释中提及到的类作为当前提交的关键类,生成正样本;其他没在注释中提及到的类作为非关键类,生成负样本.最后,所有关键类与非关键类一共产生 23 773 条样本数据.按照大概 2:1 的比例,将提交数据分为训练集与测试集.训练集含有 3 480 个提交、15 135 条样本;测试集含有 1 718 个提交、8 638 条样本.数据集统计信息见表 4.

Table 4 Data set

表 4 数据集

开源项目数量	编程语言	有效提交数量	训练集提交数量	测试集提交数量	训练样本	测试样本
120	Java	5 198	3 480	1 718	15 135	8 638

在实验过程中,我们对随机森林进行了以下参数调优,包括:*BagSizePercent*=30,对于随机森林的每个分类器,随机抽取原训练样本集的 30%进行训练;*NumFeatures*=6,对于随机森林的每个分类器,随机抽取特征集的 6 个特征进行训练;*NumIterations*=300,随机森林中包含 300 个分类器.

3.2 研究问题及评估方法

本文的主要目标是:从提交中自动判定关键类,并将关键类推荐给开发人员,辅助开发人员理解代码提交.因此,我们主要关注:ICC 判定关键类的准确率能达到多少?关键类的判定是否对开发人员理解代码提交有积极作用?因此,我们设置了如下两个研究问题.

- 问题 1:ICC 判定关键类的精确度和召回率分别是多少?

ICC 从提交中提取可判别特征,然后采用有监督的机器学习算法判定类的关键性.因此,我们在评估 ICC 的效果时,主要考察两个度量指标:精确度(precision)和召回率(recall).精确度用来衡量结果集的精确性,表示被 ICC 判定为关键类的结果集与数据集中真正的关键类集合的吻合程度;而召回率用来度量结果集的安全性,表示被 ICC 判定为关键类的结果集能够覆盖数据集中真正的关键类集合的程度.精确度和召回率的定义同样适用于非关键类的判定.同时,为了考察 ICC 判定关键类与非关键类的综合效果,还给出了综合准确率(accuracy)的定义.这 3 个度量指标定义如下:

$$precision=\frac{|ActualCoreClasses\cap EstimatedCoreClasses|}{|EstimatedCoreClasses|}\times 100\% \tag{1}$$

$$recall=\frac{|ActualCoreClasses\cap EstimatedCoreClasses|}{|ActualCoreClasses|}\times 100\% \tag{2}$$

$$accuracy=\frac{|CorrectJudgedClasses|}{|TotalClasses|}\times 100\% \tag{3}$$

其中,*ActualCoreClasses* 表示数据集中真正的关键类集合,*EstimatedCoreClasses* 表示被 ICC 判定为关键类的集合,*CorrectJudgedClasses* 表示被 ICC 判定正确的类的数量,*TotalClasses* 表示数据集中总的类数量.

另一方面,对于任意一个提交提交,当且仅当 ICC 将提交中的关键类判定为关键类、非关键类判定为非关键类时,才认为 ICC 正确地判定了该提交;只要提交中有一个类被 ICC 判定为错误的类别,则认为整个提交被判定错误.为此,给出了 ICC 在判定提交时的绝对准确率的定义:

$$absoluteaccuracy=\frac{|CorrectJudgedCommit|}{|TotalCommit|}\times 100\% \tag{4}$$

其中,*CorrectJudgedCommit* 表示数据集中被 ICC 判定正确的提交数量,*TotalCommit* 表示数据集中总的提交数量.

- 问题 2:关键类的判定是否对开发人员理解代码提交有积极作用?

相比于开发人员直接理解提交,ICC 提示关键类的方法是否能显著提高开发人员的效率?为了验证这个结论,我们从数据集中选取了 30 条代码提交向用户发出问卷调查.根据不同类型的提交(按照提交涉及修改的类数量不同进行分类)在数据集中存在的比例不同,选取的提交涉及修改的类数量及提交数量分别为 2(12 条),3(8 条),4(5 条),5(3 条),7(1 条)和 9(1 条)个.在问卷调查中,我们将这 30 条提交设置为 30 道问卷调查题目.每道题目被设置为两种类型:包含关键类提示与没有关键类提示两种.每个参与者被分配到的题目中,一半为包含提示,一半不包含提示.对于每道题目,我们给出当前提交涉及修改的类及修改类的源代码,并且把涉及修改的代码段高亮处理.答题者参考关键类提示信息(如果有),理解提交中的代码修改后,手写出注释消息,并记录下理解提交的开始时间及结束时间以及理解当前提交所花费的总时间.

调查问卷以两种方式发起:一种是本地调查,一种是网络调查.在本地调查中,我们向 8 位参与者发出了问卷调查.在这 8 位参与者中,有 1 位高校教师,2 位博士研究生以及 5 位硕士研究生.所有参与者都来自中山大学,并从事与计算机专业相关的工作或研究.在网络调查中,我们以任务的方式将调查问卷发布到猪八戒网(http://www.zbj.com)上,同时要求答题人提供相应的编程经验数据.收到的问卷中,参与网络调查的对象都是从

事与计算机专业相关的工作,并且平均拥有 4 年以上 Java 编程经验.在实验中,本地调查一共获得 122 份完整的调查问卷;在网络调查中,一共得到 96 份完整的调查问卷.调查问卷的格式如下所示.

问卷调查		
项目名:jEdit	版本号:24039	提交作者:dal
参与人姓名:		
开始时间:	___:___	
关键类信息:	*LookAndFeelPlugin.java*	
源代码:	*LookAndFeelOptionPane.java* *LookAndFeelPlugin.java* *SystemLookAndFeelInstaller.java*	
手写注释:		
结束时间:	___:___	
所用时间:	_____(分钟)	

值得注意的是,调查问卷中隐藏了提交的原始注释消息.这样做的目的是为了让调查对象真正通过查看涉及修改的源代码理解提交,而不是直接查看提交注释后理解提交.为了判定调查对象是否真正理解了一个提交,当他们查看完一个提交时,要求他们按照自己的理解手写当前提交的注释.调查结束后,我们将提交的原始注释与调查对象手写的注释进行对比,如果二者表达了相同或相似的意思,则证明调查对象真正理解了当前提交,他所做的问卷调查为有效调查问卷.

为了让调查对象真正通过查看涉及修改的源代码理解提交并写出注释,用于问卷调查的提交都是经过严格筛选的.即:被选中作为问卷的提交都包含一个关键类和若干个非关键类,且非关键类的修改是为关键类的修改服务的.这就需要被测人员顺着“关键类”到“非关键类”的路径去阅读整个提交代码后,才能真正理解该提交,然后才能写出有效的提交注释.另外,只有当被测人员写出了与原注释高度匹配的提交注释时才认为该注释是有效的.这里的“高度匹配”一方面是指注释在语义上的匹配,即,表达相同或相似的意思;另一方面是指注释在描述力度上的匹配,比如,有被测人员写了这样的注释:“added method *m* in class *C*”(该提交的原有的注释为“added encryption functionality in class *C*”).虽然被测人员写的这条注释描述了该提交中涉及变化的软件实体,但是它的描述力度不够.相比于原有注释,它没有描述出该提交中代码修改所要实现的目标或目的,即加密功能.因此,我们认为此条注释是无效的.

最后,对比提示关键类与不提示关键类的有效调查问卷,评估 ICC 给出关键类的方法是否能减少开发人员理解提交的时间;同时,通过对比两种调查方式中产生的有效调查问卷的数量,考察 ICC 给出关键类的方法是否能提高开发人员理解提交的准确率.

4 实验结果

4.1 ICC判定关键类的精确度和召回率

ICC 判定关键类的结果见表 5.实验中,测试集里的提交数量一共为 1 718.其中,1 109 个提交被 ICC 判定正确,609 个提交被 ICC 判定错误.1 718 个提交中,一共包含 8 638 个涉及修改的类.因此,样本总量为 8 638,正样本数量为 2 074,负样本数量为 6 564.在整个实验中,正样本用‘1’标注,负样本用‘0’标注.被 ICC 正确分类为正样本的数量为 1 424,被正确分类为负样本的数量为 5 915.被 ICC 错误分类为正样本的数量为 649,被错误分类为负样本的数量为 650.

Table 5 Results

表 5 判定结果

提交数量		样本数量		正确分类的样本数量		错误分类的样本数量	
正确判定的提交数量	错误判定的提交数量	正样本(1)	负样本(0)	正确分类为 1	正确分类为 0	错误分类为 1	错误分类为 0
1 109	609	2 074	6 564	1 424	59 15	649	650
合计:1 718		合计:8 638		合计:7 339		合计:1 299	

表 6 列出了 ICC 在判定关键类与非关键类时的精确度和召回率,其中,ICC 在判定关键类时(即正样本),其精确度和召回率分别为 68.69%和 68.66%;ICC 在判定非关键类时(即负样本),其精确度和召回率分别达到了 90.09%和 91.11%.由表 6 可见,ICC 在判定关键类和非关键类时的表现存在较大的差异.这主要是因为正负样本的数量差异造成的.因为正、负样本的数量比约为 1:3,原本为负样本的样本被 ICC 误断为正样本后(或者原本为正样本的样本被 ICC 误断为负样本后),在正、负样本总数不变的情况下,ICC 判定关键类和非关键类的精确度和召回率的下降幅度的比值大约为 3:1.

Table 6 Precision and recall

表 6 精确度与召回率

样本类型	Precision (%)	Recall (%)
正样本	68.69	68.66
负样本	90.09	91.11

表 7 和表 8 分别列出了 ICC 在综合准确率和绝对准确率方面的表现.表 7 显示,ICC 判定关键类和非关键类的综合准确率可达 84.96%,即,ICC 可以正确分类出大部分关键类与非关键类;表 8 显示,ICC 判定提交中的关键类和非关键类的绝对准确率为 64.55%.

Table 7 Accuracy

表 7 综合准确率

TotalClasses	CorrectJudgedClasses	Accuracy (%)
8 638	7 339	84.96

Table 8 Absolute accuracy

表 8 绝对准确率

TotalCommit	CorrectJudgedCommit	Absoluteaccuracy (%)
1 718	1 109	64.55

在未进行“坏样本”处理之前,*ICC* 的绝对准确率为 64.55%.通过分析实验中的样本集,我们发现,数据集中存在“坏样本”,它们在建立有效分类模型中会起干扰作用.因此,我们使用算法 1 剔除掉样本集中的“坏样本”,并重新进行分类模型的训练.我们一共剔除了 523 条“坏样本”,然后按照 2:1 的比例重新分配训练样本和测试样本.最后得到的综合准确率和绝对准确率见表 9 和表 10.其中,综合准确率提升了 2.11%,为 87.07%;绝对准确率提升了 5.82%,达到了 70.37%.由于不同机器学习算法的分类结果并没有特别显著的差异,我们在第 5 节进行讨论.

Table 9 Accuracy

表 9 综合准确率

TotalClasses	CorrectJudgedClasses	Accuracy
8 115	7 066	87.07% (+2.11%)

Table 10 Absolute accuracy

表 10 绝对准确率

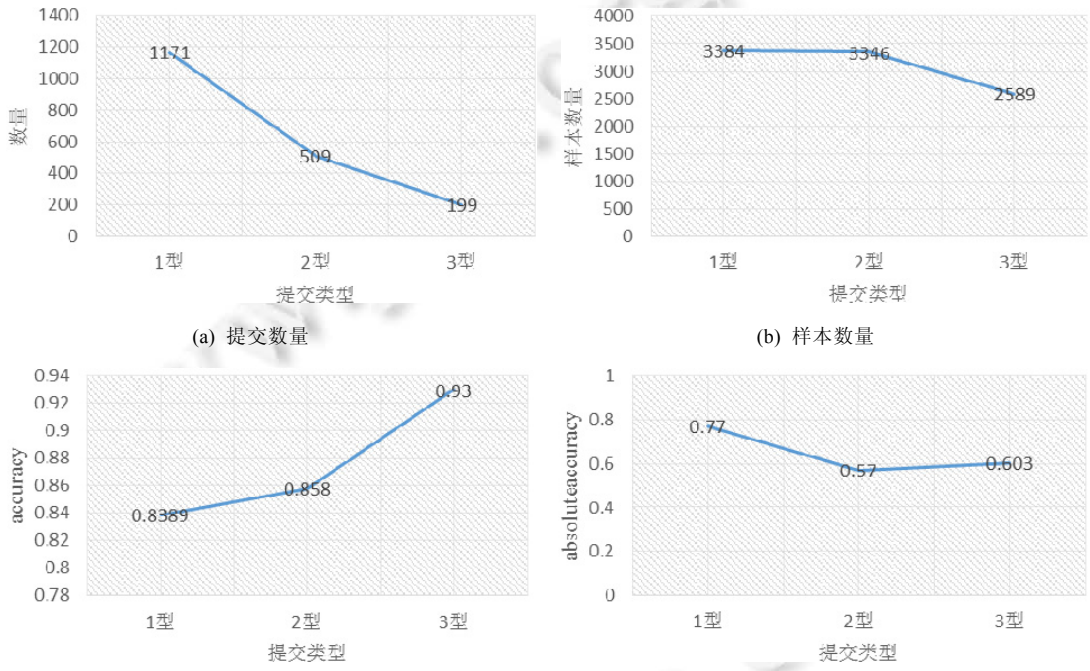
TotalCommit	CorrectJudgedCommit	Absoluteaccuracy
1 671	1 176	70.37% (+5.82%)

为了考察提交中受影响的类的数量(即提交定义中的$|S|$)对于 ICC 判定关键类的正确性的影响,我们根据$|S|$值的不同将提交分为 3 种类型,如下所示:

- 1 型:$2\leqslant|S|\leqslant4$,表示受影响的类的数量大于等于 2 小于等于 4 的提交;
- 2 型:$5\leqslant|S|\leqslant10$,表示受影响的类的数量大于等于 5 小于等于 10 的提交;
- 3 型:$11\leqslant|S|\leqslant20$,表示受影响的类的数量大于等于 11 小于等于 20 的提交.

图 3(a)展示了数据集中不同类型的提交的数量:1 型提交数量为 1 171,2 型提交数量为 509,3 型提交数量为

199.从统计数据可以看出:随着提交中受影响的类的数量增加,相应类型的提交数量逐渐减少.图 3(b)统计了 3 种提交类型下的样本数量,即,提交中包含的相关类的数量.1 型样本数量为 3 384,2 型样本数量为 3 346,3 型样本数量为 2 589.图 3(c)显示了不同类型提交的综合准确率.当提交为 1 型~3 型时,ICC 的综合准确率分别为 83.89%,85.8%和 93%.图 3(d)显示了 ICC 在各种提交类型下的绝对准确率.当提交为 1 型~3 型时,ICC 的绝对准确率分别为 77%,57%和 60.3%.也就是说,随着提交中涉及的类数量的增加,其综合准确率提高,而绝对准确率却递减.因为当提交中涉及的类数量越多时,提交中的非关键类的数量也越多.而 ICC 判定非关键类的精确度和召回率达到 90%以上,从而拉高了 ICC 的综合准确率.另一方面,由于在一个提交中必须正确判定所有类的关键性,其难度随着提交中涉及的类数量的增加而增加.因此,我们的结论就是:提交中受影响的类数量会直接影响 ICC 判定方法的准确率.



(a) 提交数量 (b) 样本数量

(c) 综合准确率 (d) 绝对准确率

Fig.3 $|S|$ affects ICC's performance

图 3 提交中受影响的类的数量对 ICC 判定效果的影响

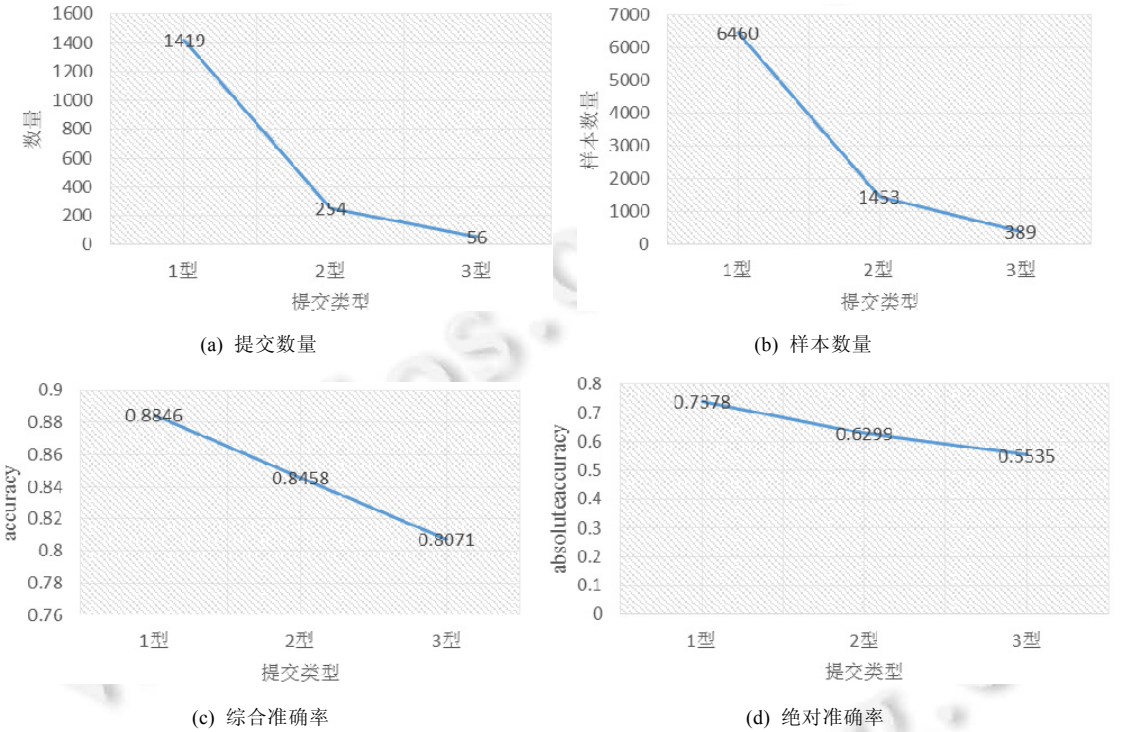
实验中所用的提交,其关键类可能只有 1 个,也可能有多个(即,关键类集合定义中的$|K|\geq 1$).为了考察 ICC 对包含不同数量关键类的提交的判定效果,我们也将提交划分为 3 种类型,如下所示:

- 1 型:$|K|=1$,表示提交中只有 1 个关键类;
- 2 型:$|K|=2$,表示提交中有 2 个关键类;
- 3 型:$|K|=3$,表示提交中有 3 个关键类.

图 4(a)分别统计了 3 种类型下的提交数量,其中,1 型的提交数量为 1 419 个,2 型的提交数量为 254 个,3 型的提交数量为 56 个.可以看出,大部分提交只包含一个关键类.图 4(b)统计了 3 种提交类型下的样本数量,其中,1 型的样本数量为 6 460 个,2 型的提交数量为 1 453 个,3 型的提交数量为 389 个.从图 4(b)可以看出,1 型提交的样本数量占了很大比例.

图 4(c)显示:ICC 在判定包含不同关键类数量的提交时,其综合准确率有显著差异.ICC 判定 1 型提交时的效果最好,达到了 88.46%;判定 2 型的提交效果次之,为 84.58%;判定 3 型的提交效果最差,只有 80.71%.同样地,ICC

在判定 3 种类型的提交时,其绝对准确率也有显著差异,分别为 73.78%,62.99%和 55.35%.由于一个提交中的关键类数量越多,其特征相对减弱,因此,当提交中关键类的数量越多时,ICC 的误判率越高.幸运的是,实际应用中的大部分提交只包含一个关键类.我们得出的结论为:当提交中包含的关键类数量增多时,ICC 判定关键类的准确率和绝对准确率均有所下降.



(a) 提交数量 (b) 样本数量

(c) 综合准确率 (d) 绝对准确率

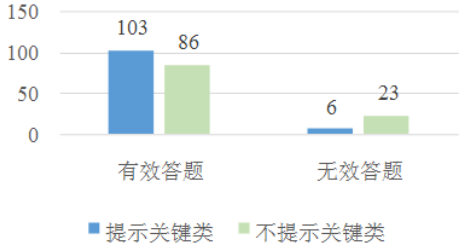
Fig.4 |*K*| affects ICC's performance

图 4 提交中关键类的数量对 ICC 判定效果的影响

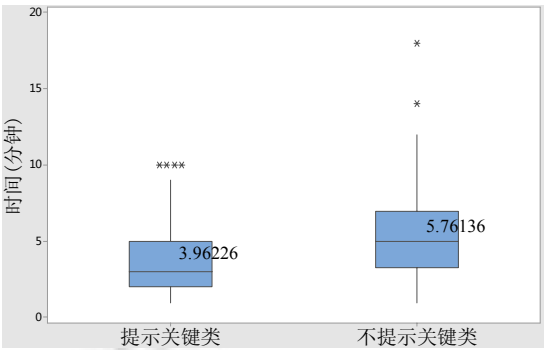
4.2 关键类的判定是否对开发人员理解代码提交有积极作用

对于收集到的问卷调查,只有当参与者手写注释与提交自带的注释表达相同或相近的意思时,才认为参与者做的答题为有效答题,否则为无效答题.根据统计,在整个问卷调查中,带关键类提示的有效答题总数为 103 份,不带关键类提示的有效答题总数为 86 份,带关键类提示的无效答题总数为 6 份,不带关键类提示的无效答题总数 23 份,如图 5(a)所示.

从统计结果中可以看出:带关键类提示信息时产生的有效调查问卷比不带关键类提示信息时产生的有效调查问卷多 17 份;而带关键类提示信息时产生的无效调查问卷比不带关键类提示信息时产生的无效调查问卷少 17 份.这就说明 ICC 给出关键类的方法确实能够提高开发人员理解提交的准确度.同时,我们还回访了部分参与调查的人员.他们在做问卷调查中感受到:“如果提示了关键类,就比较容易把握代码修改的整体逻辑”;“在一个提交中,很多时候是由于关键类的修改,导致了其他非关键类的修改”;“如果不提示关键类,自己也能通过查看涉及修改的代码段推测出关键类,只是会花费更多的时间”.因此,无论从数据统计还是参与者事后回访中都能看到,ICC 给出的关键类的方法确实在很大程度上方便了开发人员准确地理解代码提交.



(a) 有效答题与无效答题数量统计



(b) 理解提交时提示关键类与不提示关键类所用时间对比

Fig.5 Questionnaire result

图 5 问卷调查结果统计

此外,我们还统计了带关键类提示信息和不带关键类提示信息时产生有效答题所用的时间,如图 5(b)所示:带关键类提示的有效答题所用平均时间为 3.96 分;不带关键类提示的有效答题所用平均时间为 5.76 分.也就是说,带关键类提示可以让参与人员在正确理解一个提交前提下平均节省 1.8 分.其中,提示关键类时所用时间最长为 10 分,最短为 1 分;不提示关键类时所用时间最长为 18 分,最短为 1 分.为了验证 ICC 提示关键类与不提示关键类的方法在理解代码提交上所花的时间有显著性差异,我们使用 Wilcoxon 假设检验方法[7]检验如下的空假设:

假设 H_0:ICC 提示关键类的方式在理解代码提交上所花时间与不提示关键类的提交理解方式所花时间没有显著差异.

检验结果见表 11,检验统计量 w=8546.5 的 p 值在对结调整时小于 0.000 1.由于 p 值小于所选水平 0.05,因此有充分的证据否定原假设 H_0,认为 ICC 提示关键类的方式在理解代码提交上所花的时间少于不提示关键类的方式.这些结论都可以肯定本文引言中提出的假设,即:理解关键类对于理解整个提交有着重要作用.

Table 11 Results of hypothesis testing

表 11 假设检验结果

点估计值	95%置信区间	w	p
−2.0	(−2.0,−1.0)	8 546.5	<0.0001

5 讨 论

在实验过程中,我们还对比了多种机器学习算法在判定关键类时的表现.参与比较的机器学习算法都是基于监督的学习算法,它们是 AdaBoost、LogitBoost、C4.5 决策树、支持向量机、朴素贝叶斯、概率神经网络、随机森林.为了公平比较,所有算法在实验之前都将它们各自的参数调到最优的水平.比如,概率神经网络算法含有可调节参数 spread.spread 是径向基函数的平滑参数.实验中,通过设置 spread 不同的取值,考察概率神经网络判定关键类的准确率的变化情况.最终,spread 的值设置为 0.4 时,概率神经网络的判定结果最优.表 12 例举了各种机器学习算法的综合准确率及绝对准确率.从结果中可以观察到:除了概率神经网络和随机森林,其他各种机器学习算法判定关键类时的综合准确率和绝对准确率没有显著性差异.其中,随机森林算法的表现要优于概率神经网络.

实验中,为了保证样本集的质量,我们在样本选择时设置了多重过滤规则.虽然我们从 sourceforge 的 SVN 库中收集了 120 个开源项目的提交,但是很多项目的提交质量较差,无法满足实验要求.主要体现在:

1) 部分提交是关于非代码修改的提交,比如配置文件修改、文件路径修改等;

2) 部分项目的提交缺少涉及修改的类信息;

3) 部分项目的提交缺少注释消息.

这 3 种情况下的提交,在我们的实验中就无法使用.用于实验的提交,要求是关于代码修改并且给出了涉及修改的类信息,以及在注释消息中给出了关键类信息.在这个要求下,满足条件的提交数量只有 8 628 个;再加上提交中涉及修改的类数量需要大于等于 2 且小于等于 20,使得满足条件的提交数量下降到了 6 047 个;同时,还需要满足提交的注释消息的单词数在 3 个~200 个之间,使得提交数量下降到了 5 198 个;最后,过滤掉"坏样本",提交数量最终剩下 5 013 个.因此,实验中的样本规模较小也是影响 ICC 准确率的一个因素;同时,因为只收集到有限满足条件的提交数据,最终的生成样本集中的正样本数量相对较少,这也是造成了 ICC 判定正样本的精确度和召回率相对较低的原因之一.

Table 12 Comparison for multiple machine learning algorithms

表 12 机器学习算法准确率比较

机器学习算法	综合准确率	绝对准确率
AdaBoost	0.840 6	0.648 4
LogitBoost	0.846 3	0.659 5
C4.5 决策树	0.847 8	0.662 5
支持向量机	0.845 0	0.655 2
朴素贝叶斯	0.848 8	0.658 2
概率神经网络	0.861 0	0.685 8
随机森林	0.870 7	0.703 7

"坏样本"的存在,降低了 ICC 判定关键类的准确率."坏样本"的产生过程为:在数据集中存在两个类,它们的特征向量中的每一维都相同.其中一个类在它所在的提交注释消息中被提到了,所以把它标注为正样本'1';而另外一个类在它所在提交的注释消息中没被提到,所以被标注为负样本'0'.这样,虽然两个样本的特征向量完全相同,但是它们分别被标注成了正负样本,这样便产生了"坏样本".对于数据集中"坏样本"的处理,选择同时删除正负样本是受数据清理方法 Tomek links[8,9]的启发.Tomek links 方法的基本原理为:对于两个属于不同分类的样本 s_1 和 s_2,用 $d(s_1,s_2)$定义它们之间的距离.如果找不到第 3 个样本 s_3,使 $d(s_1,s_3)<d(s_1,s_2)$或者 $d(s_2,s_3)<d(s_1,s_2)$,那么 s_1 和 s_2 很可能是噪音数据,应该从样本集中删除.在我们的数据集中,被删除的"坏样本"s_1 和 s_2 属于不同分类,又由于它们的特征向量相同,所以 $d(s_1,s_2)=0$,显然找不到样本 s_3,使 $d(s_1,s_3)<d(s_1,s_2)$或者 $d(s_2,s_3)<d(s_1,s_2)$.因此它们属于噪音样本,需同时删除.

此外,ICC 判定关键类的方法适用于所有面向对象的编程语言.因为 ICC 方法提取的可判别特征包括代码耦合特征、代码修改特征以及提交类型特征,所有这些特征都不与特定编程语言(比如 Java)相关.具体来说,代码耦合特征是用类的出度和入度来度量,而所有面向对象的编程语言都可以无差别地提取类的出度和入度;代码修改特征是用涉及修改的方法数或代码行数来度量,此特征也可以无差别地从任意一种面向对象的编程语言中提取;提交类型特征以提交注释信息中含有的关键字作为判定依据,与具体的编程语言无关.

6 相关工作

代码提交的主要作用是帮助开发人员后期理解代码修改,有效地进行代码复查、代码共享等软件活动.本文使用机器学习的算法从代码提交中定位出关键类,然后将关键类作为开发人员理解代码提交时的重要提示信息.本节主要从两个方面介绍与本文相关的一些研究工作:一是辅助开发人员理解代码修改相关的工作,二是机器学习在以代码分析为基础的软件工程中的应用.

6.1 代码修改理解

开发人员对代码修改的理解是综合的过程,学者从不同侧面对此问题进行研究,比如定位代码修改[10]、自动生成代码修改注释消息[11,12]、将复合类型的代码修改分解为原子类型[2,3]、代码修改回看[13]、代码修改可视化[14]等.所有这些研究都可以帮助开发人员更容易地理解代码修改.

代码修改理解的第 1 步是定位修改发生在代码中的哪些区域,因此,有学者提出了 ChangeDistiller[8]工具.

ChangeDistiller 工具通过比较两个版本中的代码在抽象语法树方面的差异,识别出当前版本中发生了修改的代码区域.此外,为了使开发人员在不用阅读代码的情况下就能理解代码修改,很多研究人员提出了自动生成修改注释的方法.DeltaDoc[11]工具通过分析修改前和修改后的代码行为生成可供开发人员阅读的修改注释.Vásquez 等人[12]提出一种自动注释代码提交的方法,该方法首先分析提交中发生修改的函数类型,然后根据函数的类型判定提交本身的类型,最后根据提交的类型生成相应的代码注释.Barnett 等人[2]提出一种从代码修改集合中聚类出相关联代码修改的方法,该方法从代码修改集合中检测变量申明与变量使用,然后将涉及同一个变量的申明或使用的代码修改区域聚类在一起.不过,他们的方法在小规模的代码修改集合中表现得非常高效,当随着代码修改集规模的增加,很多修改都倾向于相关联,这样就会造成单个聚类的规模特别大.为了探索怎样的提交能让开发人员更容易地理解代码修改,Herzig 等人[3]设置实验验证了原子类型提交的重要性.在文中,他们还提出一种基于启发式的算法来将复合类型的提交分解为原子类型.此外,Maruyama 等人[13]利用程序切片技术抽取出程序员关心的代码修改,这样就可以避免程序员花过多时间在那些无关紧要的代码修改上.Gómez 等人[12]提供一个可视化工具,用于支持面向对象程序 Smalltalk 的代码修改可视化问题.以上这些研究,都是为了帮人开发人员更容易的理解代码修改.

6.2 机器学习在代码分析中的应用

近年来,机器学习算法已经应用于各种以代码分析为基础的软件工程研究中,包括复合提交分解[15]、程序功能判定[16]、项目自动分类[17]、相关联代码修改判定[18]以及可追踪性链恢复[19]、代码漏洞定位[20]等等.

Dias 等人[15]利用软件开发过程中产生的细粒度的代码修改信息,将复合类型的代码修改分解为更为简单的原子类型.他们的方法基于机器学习的算法,考虑的主要特征有涉及修改的代码段的空间特征、结构特征以及修改时间特征等.Mou 等人[16]利用卷积神经网络识别代码功能,该方法首先将代码映射成 AST 树,然后在 AST 树上做卷积,进而凸显代码在结构方面的特征.使用卷积神经网络识别代码功能,可达到 94%的准确率.Vásquez 等人[17]提出一种利用机器学习算法对开源项目库中的项目进行自动分类的方法,该方法的创新点是,直接从项目源码所依赖的第三方应用程序编程接口(API)中提取特征来完成对项目的分类.注意到机器学习算法在分类领域的有效性,我们文献[18]提出了使用概率神经网络来识别相关联的代码修改.我们的方法从软件实体中提取可判别特征来度量代码修改的相关性,该方法的准确率可达 90%.Le 等人[19]从程序自动生成的注释消息和开发人员手写的注释消息中提取丰富的文本特征,利用随机森林算法来恢复问题报告与代码提交之间的可追踪性链.Lam 等人[20]利用深度神经网络并配合使用向量空间模型进行代码漏洞定位.通过训练,深度神经网络可以增强漏洞描述文件和源码文件之间的匹配成功率.为了方便将 Java 编程语言环境下开发的代码迁移到 C#编程语言环境下,Nguyen 等人[21]利用统计学习的算法自动学习出 Java 与 C#的 API 之间的映射关系.Nguyen 等人[22]使用贝叶斯模型开发出了 API 推荐引擎.在实际开发中,该引擎可以推荐 API 对应的编程模板给开发人员.Zanoni 等人[23]从软件代码的微观结构中获取特征,利用机器学习的算法检测代码中存在的设计模式.Liu 等人[24]从多个软件项目中获取先验知识来建立软件质量评估模型,他们一共评估了 17 种机器学习算法在软件质量评估中的表现.

7 结 论

代码提交在软件维护活动中扮演着极其重要的角色,高效地理解代码提交,成为了软件开发人员追求的目标.本文根据类实体在代码提交中所扮演的角色,分为了关键类和非关键类.关键类对于软件开发人员理解代码提交有着重要的作用.为了自动定位提交中的关键类,提出了 ICC 方法.该方法从提交中提取可判别特征,并使用带监督的机器学习算法来判定类的关键性.通过问卷调查,验证了 ICC 在帮助软件维护人员理解代码提交的有效性.在未来的研究工作中,我们将继续从版本控制器中收集更多提交数据生成训练集,进一步提高 ICC 判别关键类的准确率.我们重点收集那些在注释消息中指明了关键类的提交,进一步扩大训练集中正样本的数量,使最终的正负样本数量达到相对平衡的比例.同时,考虑到文章实验只针对 Java 语言的数据集做了关键类的判定,后续工作会将 ICC 应用于更多面向对象编程语言的数据集上.另外,我们考虑将 ICC 做成插件集成到 eclipse 中.

当开发者准备将代码提交发布到版本控制器中时,ICC 插件自动提示开发者本次提交的关键类.开发者确认关键类信息无误后,当前提交的关键类信息被标注,并随提交一同发布到版本控制器中.如果 ICC 提示的是错误的关键类,开发者也可以自己重新选择正确的关键类进行标注.

References:

[1] Sun XB, Li BX, Tao CQ. Using LoCMD to support software change analysis. Ruan Jian Xue Bao/Journal of Software, 2012,23(6): 1368−1381 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4072.htm [doi: 10.3724/SP.J.1001.2012.04072]

[2] Barnett M, Bird C, Brunet J, Lahiri SK. Helping developers help themselves: Automatic decomposition of code review changesets. In: Proc. of the 37th Int'l Conf. on Software Engineering. IEEE, 2015. 134−144.

[3] Herzig K, Zeller A. The impact of tangled code changes. In: Proc. of the 10th Conf. on Mining Software Repositories, IEEE, 2013. 121−130.

[4] Hassan AE, Holt RC. Predicting change propagation in software systems. In: Proc. of the 20th Conf. on Software Maintenance. IEEE, 2004. 284−293. [doi: 10.1109/ICSM.2004.1357812]

[5] Hattori L, Lanza M. On the nature of commits. In: Proc. of the 23rd Int'l Conf. on Automated Software Engineering. IEEE/ACM, 2008. 63−71. [doi: 10.1109/ASEW.2008.4686322]

[6] Breiman L. Random forests. Machine Learning, 2001,45(1):5−32. [doi: 10.1109/ASEW.2008.4686322]

[7] Groggel DJ. Practical nonparametric statistics. Technometrics, 2000,42(3):317−318.

[8] Tomek I. Two modifications of CNN. IEEE Trans. on Systems, Man and Cybernetics, 1976,6:769−772.

[9] Batista GE, Prati RC, Monard MC. A study of the behavior of several methods for balancing machine learning training data. ACM Sigkdd Explorations Newsletter, 2004,6(1):20−29.

[10] Fluri B, Wuersch M, PInzger M, Gall H. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. on Software Engineering, 2007,33(11):725−743. [doi: 10.1109/TSE.2007.70731]

[11] Buse RP, Weimer WR. Automatically documenting program changes. In: Proc. of the 23rd Int'l Conf. on Automated Software Engineering. IEEE/ACM, 2010. 33−42. [doi: 10.1145/1858996.1859005]

[12] Linares-Vásquez M, Cortés-Coy LF, Aponte J, Poshyvanyk D. Changescribe: A tool for automatically generating commit messages. In: Proc. of the 37th Int'l Conf. on Software Engineering. IEEE/ACM, 2015. 709−712. [doi: 10.1109/ICSE.2015.229]

[13] Maruyama K, Kitsu E, Omori T, Hayashi S. Slicing and replaying code change history. In: Proc. of the 27th Int'l Conf. on Automated Software Engineering. IEEE/ACM, 2012. 246−249. [doi: 10.1145/2351676.2351713]

[14] Verónica UG, Stéphane D, Theo D. Visually characterizing source code changes. Science of Computer Programming, 2015,98(3): 376−393. [doi: 10.1016/j.scico.2013.08.002]

[15] Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S. Untangling fine-grained code changes. In: Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution and Reengineering. IEEE/ACM, 2015. 341−350.

[16] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 9th AAAI Conf. on Artificial Intelligence. 2016. 1287−1293.

[17] Linares-Vásquez M, Mcmillan C, Poshyvanyk D, Grechanik M. On using machine learning to automatically classify software applications into domain categories. Empirical Software Engineering, 2014,19(3):582−618. [doi:10.1007/s10664-012-9230-z]

[18] Huang Y, Chen XP, Zou QW, Luo XN. A probabilistic neural network-based approach for related software changes detection. In: Proc. of the 21st Asia-Pacific Software Engineering. 2014. 279−286. [doi:10.1109/APSEC.2014.50]

[19] Le TB, Vásquez ML, Lo D, Poshyvanyk D. RCLinker: automated linking of issue reports and commits leveraging rich contextual information. In: Proc. of IEEE the 23rd Int'l Conf. on Program Comprehension. 2015. 36−47. [doi: 10.1109/ICPC.2015.13]

[20] Lam AN, Nguyen AT, Nguyen HA, Nguyen TN. Combining deep learning with information retrieval to localize buggy files for bug reports. In: Proc. of the 22nd Int'l Conf. on Automated Software Engineering. IEEE/ACM, 2015. 476−481. [doi: 10.1109/ASE. 2015.73]

[21] Nguyen AT, Nguyen HA, Nguyen TT, NguyenTN. Statistical learning approach for mining API usage mappings for code migration. In: Proc. of the 29th Int'l Conf. on Automated Software Engineering. IEEE/ACM, 2014. 457−468. [doi: 10.1145/2642937.2643010]

[22] Nguyen AT, Nguyen TN. Graph-Based statistical language model for code. In: Proc. of the 37th Int'l Conf. on Software Engineering. IEEE/ACM, 2015. 858−868.

[23] Zanoni M, Fontana FA, Stella F. On applying machine learning techniques for design pattern detection. Journal of Systems and Software, 2015,103(12):102−117. [doi: http://dx.doi.org/10.1016/j.jss.2015.01.037]

[24] Liu Y, Khoshgoftaar T, Seliya N. Evolutionary optimization of software quality modeling with multiple repositories. IEEE Trans. on Software Engineering, 2010,36(6):852−864. [doi: 10.1109/TSE.2010.51]

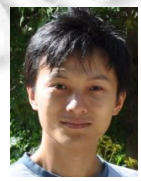
附中文参考文献:

[1] 孙小兵,李必信,陶传奇.基于 LoCMD 的软件修改分析技术.软件学报,2012,23(6):1368−1381. http://www.jos.org.cn/1000-9825/4072.htm [doi: 10.3724/SP.J.1001.2012.04072]

黄袁(1987－),男,四川内江人,博士,主要研究领域为软件工程,程序理解,软件工程过程数据挖掘.

刘志勇(1990－),男,硕士,主要研究领域为软件工程,软件工程过程数据挖掘.

陈湘萍(1981－),女,博士,助理研究员,CCF 专业会员,主要研究领域为数据驱动的软件工程,程序理解,Web 工程.



熊英飞(1982－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为软件工程,程序设计语言.

罗笑南(1963－),男,博士,教授,博士生导师,主要研究领域为图形图像处理,三维仿真 CAD 技术,数字家庭技术.