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

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

融合结构与语义特征的代码注释决策支持方法

黄袁[1,2], 贾楠[1,6], 周强[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)

[6](河北地质大学 管理科学与工程学院,石家庄 050031)

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

摘 要: 代码注释是辅助编程人员理解源代码的有效手段之一.高质量的注释决策不仅能覆盖软件系统中的核心代码片段,还能避免产生多余的代码注释.然而,在实际开发中,编程人员往往缺乏统一的注释规范,大部分的注释决策都取决于个人经验以及领域知识.对于新手程序员来说,注释决策显然成为了一项重要而艰巨的任务.为了减少编程人员投入过多的精力在注释决策上,文章从大量的代码注释实例中学习出一种通用的注释决策规范,并提出了一种新颖的 *CommentAdviser* 方法用以辅助编程人员在代码开发过程中做出恰当的注释决策.由于注释决策与代码本身的上下文信息密切相关,因此,从当前代码行的上下文代码中提取代码结构特征以及代码语义特征作为支持注释决策的主要依据.然后,利用机器学习算法判定当前代码行是否为可能的注释点.在 GitHub 中的 10 个大型开源软件的数据集上评估了我们提出的方法,实验结果以及用户调研表明代码注释决策支持方法 *CommentAdviser* 的可行性和有效性.

关键词: 代码注释;结构特征;语义特征;机器学习;注释决策

中图法分类号: TP311

中文引用格式: 陈翔, 顾庆, 刘望舒, 刘树龙, 倪超. 静态软件缺陷预测方法研究. 软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Chen X, Gu Q, Liu WS, Liu SL, Ni C. State-of-the-Art survey of static software defect prediction. Ruan Jian Xue Bao/Journal of Software, 2016 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Combining Structural and Semantic Features to Support Code Comment Decision

HUANG Yuan[1,2], JIA Nan[1,6],ZHOU Qiang[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)

[6](School of Management Science and Engineering, Hebei GEO University, Shijiazhuang 050031, China)

基金项目: NSFC-广东联合基金(U1611261); 国家重点研发计划(2016YFB1000101); 国家自然科学基金(61672545)

Foundation item: NSFC-Guangdong Joint Fund (U1611261); The National Key Research and Development Program of China (2016YFB1000101); The National Natural Science Foundation of China Science and Technology (61672545)

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

CNKI 在线出版时间: 0000-00-00

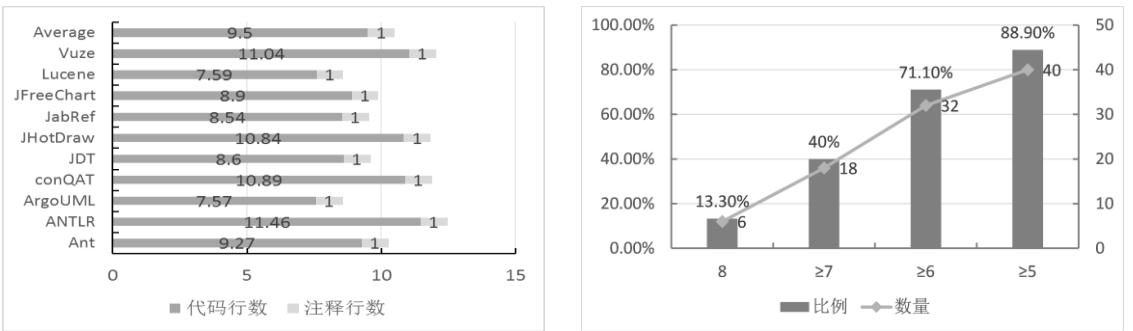
Abstract: Code comment is quite important to help developer review and comprehend source code. Strategic comment decision is desired to cover core code snippets of software system without incurring unintended trivial comments. However, in current practice, there is a lack of rigorous specifications for developers to make their comment decisions. Commenting has become an important yet tough decision which mostly depends on the personal experience of developers. To reduce the effort on making comment decisions, in this paper, we try to learn an unified commenting regulation from large of commenting instances. We propose a novel method, *CommentAdviser*, to guide developers where to comment in source code. Since making comment is closely related to the context information of source code themselves, we identify this important factor for determining where to comment and extract them as structural context feature and semantic context feature. After that, machine learning techniques are applied to identify the possible commenting locations in source code. We evaluate *CommentAdviser* on 10 data sets from GitHub. The experimental results, as well as a user study, demonstrate the feasibility and effectiveness of *CommentAdviser*.

Key words: code comment; structural feature; semantic feature; machine learning; comment decision

在规范化的软件项目开发中,为了保证源代码的易读性,编程人员会被要求在源代码中适当的位置添加适量的代码注释[1].代码注释是软件项目的重要组成部分,它用自然语言的形式阐述代码背后实现的逻辑或功能[2].因此,代码注释是编程人员理解软件代码最直观和最有效的方式.高质量的代码注释在软件维护和软件复用等领域扮演着重要的作用[3].

总的来说,代码中的注释可以分成 3 种类型[4],即:文档注释(javadoc),块注释(block comment),以及行注释(line comment).其中,文档注释用来描述一个类,一个方法,或类属性;块注释用于描述一行或多行代码,它们位于被注释代码(块)的上一行;行注释是指跟在代码行后面,并且与被注释代码处于同一行的注释,它们只为当前代码行提供描述.文档注释可以由工具自动生成[5,6],而块注释和行注释则需要编程人员手动添加.因此,在添加块注释和行注释时,编程人员需要作出恰当的注释决策.

已有的注释决策方面的研究工作大多关注于代码注释量的统计与分析[7].软件代码中存在的注释量不能太少.太少的代码注释难以覆盖系统中所有核心的代码片段,这就会增加编程人员理解代码的难度,从而增加他们审阅代码时的时间开销.其次,软件代码中的注释也不能过多,因为过多的注释意味着更多的工作量.这就要求编程人员花费更多的精力在代码注释上,从而影响开发人员的编程效率.我们统计了 10 个在软件工程研究中经常用到的开源项目[8,9,10,11],发现这些项目方法体内部的注释行数与代码行数的比例平均为 1:9.5,如图 1 (a)所示.这项统计结果表明,方法内部的注释数量和代码数量之间存在某种比例权衡.



(a) 注释行数与代码行数的比例 (b) 注释决策统计结果

Fig. 1 Statistical results

图 1 统计结果

然而,即使保证了代码数量与注释数量的比例权衡,如果将相同数量的注释添加到代码中的不同位置,它们对于编程人员理解代码所起的辅助作用显然是不同的.编程人员总是倾向于在核心或较难理解的代码片段上添加注释.为了验证编程人员对于注释位置的决策是否存在共识,我们做了一项问卷调查.我们从数据集中抽取了 20 个方法体的代码片段,然后要求 8 位参与者在每块代码片段中添加 2 处或 3 处块注释(其中 15 个代码片段要求添加 2 处块注释,5 个代码片段要求添加 3 处块注释,共 45 个注释位置决策).8 位参与人员都具有 3 年以上的 Java 编程经验.最终的统计结果如图 1 (b)所示.在所有的 45 个注释位置决策中,被 6 个或 6 个以上的参与

人员达成共识(即至少 6/8 的调研人员达成共识)的注释位置决策占 71.1%,达到了 32 个.如果将每个注释位置决策上达成共识的参与人员数量限制在 5 个或 5 个以上(即至少 5/8 的调研人员达成共识),则这个比例将达到 88.9%,即 40 个.因此,调研结果证实了编程人员添加代码注释时存在统一的注释位置决策共识.

由此可见,高质量的注释决策要求编程人员在实际开发中即保持代码中均衡的注释数量,又要保持统一的注释位置共识.然而,查阅了相关的技术文档和研究报告[12,13],并没有统一的规范指导编程人员作出这样的注释决策.大部分情况下,开发人员都需要借助自己的编程经验和领域知识做出判断.由此可见,编程人员在开发过程中做出注释决策并不是一件容易的事情,尤其是对于那些缺乏编程经验和领域知识的程序员新手.于是,本文试图从已有的注释决策实例上学习一种通用的注释决策规范用于支持代码开发过程中的注释决策.具体而言,我们提出了一种通过分析代码上下文信息辅助开发人员做出代码注释决策的方法—*CommentAdviser*.该方法从当前代码的上下文中获取了 2 种重要的信息,分别是代码结构信息以及代码语义信息.其中,代码结构信息表示了当前代码与上下文代码在位置、逻辑、以及距离等方面的关系,而代码语义信息则表示了当前代码的上下文代码的语义分布情况.使用这两种类型的信息便可以推测出当前代码与其上下文代码之间的逻辑耦合关系的强弱程度,从而推测出当前代码行是否为可能的注释点.*CommentAdviser* 基于学习的算法实现,它从 github 上已有项目的注释实例中学习出一种通用的注释决策规范,用以支持判定任意代码行是否为可能的注释点.开发人员借助于 *CommentAdviser* 推荐的注释点,最终决定是否需要在相应位置处添加注释,从而到达代码注释决策支持的目的.

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: *CommentAdviser* 推荐注释点的方法在正样本集上的精确度和召回率到达了 80.8%和 67.2%.本文的贡献有三点.首先,提出了代码决策支持的可判别特征,这使得注释决策支持成为可能;其次, *CommentAdviser* 可以为新手程序员提供了适当的注释建议;最后,提出了一套用于代码决策支持的评价机制.

文章第一节介绍基本概念;文章的主要方法,基于学习的注释决策支持方法将在第二节介绍;实验设置与实验结果将在第三、四节中讨论;用户调研章节放在第五节;第六节介绍文章的相关工作;第七节介绍总结和未来工作.

1 基本概念

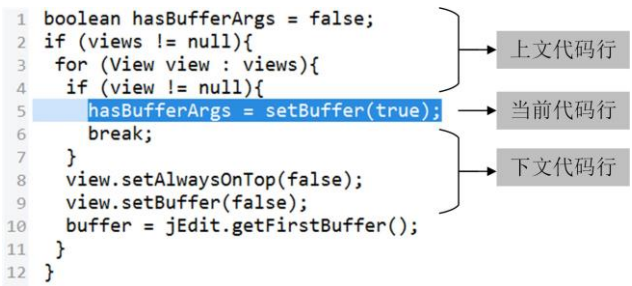


Fig. 2 Code snippet

图 2 代码片段示例

文中,为区分源代码中的控制类型与非控制类型语句,定义了控制类型代码行和非控制类型代码行.然后给出了当前代码行的定义.同时定义了上下文代码行的概念.定义如下:

定义 1 (**控制类型代码行**(***Control Statement, CS***)). 控制类型代码行负责程序的逻辑结构;编程语言中常见的控制类型代码行包括 ***CS*** ={*if, elseif, else, for, while, switch, try, catch, finally, do, synchronized*}.

定义 2 (**非控制类型代码行**(***Non-Control Statement, NCS***)). 非控制类型代码行是相对控制类型代码行定义的;常见的非控制类型代码行包括赋值语句,变量申明语句,方法调用语句,返回语句等. ***NCS***={*Assignment, FieldDeclaration, MethodInvocation, ReturnStatement, BreakStatement, ...*}.满足 ***CS*** ∩ ***NCS***=*Φ*.

定义 3 (**当前代码行**(***Current Line, CuL***)). 方法体内部的任意一行代码都可以作为当前代码行;定义为五元组(*al, cr, ia, ci, ST*),其中, *al* 为当前代码行在方法体内部的绝对位置; *cr* 为当前代码行的覆盖范围;*ia* 为当前

代码行所含标识符的数量; *ci* 为当前代码行覆盖范围内标识符的数量; *ST* 为当前代码行的类型,取值为 $ST(\boldsymbol{CuL})\in\{\boldsymbol{CS}\cup\boldsymbol{NCS}\}$.如果 $ST(\boldsymbol{CuL})\in\{\boldsymbol{NCS}\}$,则 $\boldsymbol{CuL}.cr=0$,且 $\boldsymbol{CuL}.ci=0$,即非控制类型代码行的覆盖范围为 0,它覆盖范围内标识符的数量也等于 0.

定义 4 (**上下文代码行** (***Context Line, CoL***)). 上下文代码行是相对于当前代码行定义的,分为上文代码行(***Preceding Line, pl***)和下文代码行(***Following Line, fl***).如图 2 所示.任何一条上下文代码行 ***CoL*** 被定义为六元组(*vi, mi, cd, dc, ic, cw*),其中, *vi* 表示该 ***CoL*** 是否与 ***CuL*** 调用了同一个变量; *mi* 表示该 ***CoL*** 是否与 ***CuL*** 调用了同一个方法; *cd* 表示 ***CoL*** 与 ***CuL*** 之间的距离; *dc* 表示 ***CoL*** 是否与 ***CuL*** 位于同一条控制类型代码行覆盖范围内; *ic* 表示 ***CoL*** 是否与 ***CuL*** 位于同一条控制类型代码行的覆盖范围内,并且它们之间还存在其他的控制类型代码行; *cw* 为 ***CoL*** 与 ***CuL*** 之间所有控制类型代码行的权重之和.

2 基于学习的注释决策支持方法

2.1 特征提取

本节详细介绍了从源代码中提取可判别特征的过程.由于编程人员所做的注释决策与代码行本身的上下文信息密切相关.因此,我们首先从两个维度一共提取了 11 种类型的结构上下文特征,这两个维度分别为:代码自特征(Intra-Structural Context Feature)和代码间特征(Inter-Structural Context Feature).同时,为了获取当前代码行的上下文语义信息,我们还使用词嵌入技术从代码中提取语义上下文特征.

2.1.1 结构上下文特征提取

代码具有良好的结构属性[14]. 首先,单行代码自身具有明显的结构特征,称之为代码自特征.5 种常见的代码自特征如表 1 所示:代码位置(C_{loca}),代码类型(C_{type}),覆盖范围(C_{cove}),标识符数量(C_{iden}),覆盖范围内标识符数量($C_{idencove}$);另外,代码与代码之间(文中特指 *CuL* 与 *CoL* 之间)也具有明显的结构性特征,称之为代码间特征(Inter-Structural Context Feature).代码间特征表征了当前代码行与其上下文代码行在位置,距离,耦合关系等方面的信息.6 种常见的代码间特征如表 1 所示:共同变量调用($C_{commvar}$),共同方法调用($C_{commmet}$),代码距离($C_{distance}$),直接 *CS* 覆盖($C_{direcontr}$),间接 *CS* 覆盖($C_{indircontr}$),*CS* 权重($C_{contrweigh}$).

Table 1 Structural context feature

表 1 结构上下文特征

分类	特征	描述	取值
代码自特征 (Intra-Structural Context Feature)	C_{loca}	$C_{loca}=CuL.al$	$C_{loca}\in\{1, 2, \ldots, n\}$
	C_{type}	$C_{type}=ST(CuL)$; $ST(CuL)\in\{CS\cup NCS\}$	**If** $ST(CuL)\in\{NCS\}$, $C_{type}=0$; **elseif** $ST(CuL)=$ "*if*", $C_{type}=1$; **elseif** $ST(CuL)=$"*for*", $C_{type}=2$; **elseif**…
	C_{cove}	**If** $ST(CuL)\in\{CS\}$, $C_{cove}=CuL.cr$; **elseif** $ST(CuL)\in\{NCS\}$, $C_{cove}=0$	$1\leqslant CuL.cr\leqslant 5$, $C_{cove}=1$; $6\leqslant CuL.cr\leqslant 9$, $C_{cove}=2$; $10\leqslant CuL.cr\leqslant 19$, $C_{cove}=3$;$20\leqslant CuL.cr\leqslant 29$, $C_{cove}=4$; $30\leqslant CuL.cr$, $C_{cove}=5$
	C_{iden}	$C_{iden}=CuL.ia$	$1\leqslant CuL.ia\leqslant 5$, $C_{iden}=1$; $6\leqslant CuL.ia\leqslant 9$, $C_{iden}=2$; $10\leqslant CuL.ia\leqslant 19$, $C_{iden}=3$; $20\leqslant CuL.ia$, $C_{iden}=4$
	$C_{idencove}$	**If** $ST(CuL)\in\{CS\}$, $C_{idencove}=CuL.ci$; **elseif** $ST(CuL)\in\{NCS\}$, $C_{idencove}=0$	$1\leqslant CuL.ci\leqslant 10$, $C_{idencove}=1$; $11\leqslant CuL.ci\leqslant 20$, $C_{idencove}=2$; $21\leqslant CuL.ci\leqslant 40$, $C_{idencove}=3$;$41\leqslant CuL.ci\leqslant 100$, $C_{idencove}=4$; $101\leqslant CuL.ci$, $C_{idencove}=5$
代码间特征 (Inter-Structurl Context Feature)	$C_{commvar}$	**If** $Col.vi=True$, $C_{commvar}=1$; **otherwise**, $C_{commvar}=0$	$C_{commvar}\in\{0,1\}$
	$C_{commmet}$	**If** $Col.mi=True$, $C_{commmet}=1$; **otherwise**, $C_{commmet}=0$	$C_{commmet}\in\{0,1\}$
	$C_{distance}$	$C_{distance}=Col.cd$	$1\leqslant Col.cd\leqslant 10$, $C_{distance}=Col.cd$; $11\leqslant Col.cd\leqslant 15$, $C_{distance}=11$; $16\leqslant Col.cd\leqslant 20$,

		$C_{distance}$=12;21≤*Col.cd*≤30, $C_{distance}$=13; 31≤*Col.cd*, $C_{distance}$=14
$C_{direcontr}$	**If** *Col.dc=True*, $C_{direcontr}$ = 1; **otherwise**, $C_{direcontr}$ = 0	$C_{direcontr}\in\{0,1\}$
$C_{indircontr}$	**If** *Col.ic=True*, $C_{indircontr}$ = 1; **otherwise**,$C_{indircontr}$ = 0	$C_{indircontr}\in\{0,1\}$
$C_{contrweigh}$	$C_{contrweigh}$ = *Col.cw*	0.0≤$C_{contrweigh}$

代码自特征 C_{loca} 表示了当前代码在方法体内部的绝对位置.如图 2 所示,蓝色背景高亮的代码为当前代码行,它的绝对位置为 5.特征 C_{type} 表明了当前代码行的类型.从定义 1 和 2 可知,代码行的类型可以分为控制类型和非控制类型.控制类型代码行又可细分为 *if, elseif, else, for, while* 等类型.而非控制类型代码行可细分为 *Assignment, FieldDeclaration* 等类型. 特征 C_{cove} 表示当前代码行作为控制类型代码行时,其覆盖范围内代码行的数量.如图 2 中绝对位置为 4 的 *if* 控制语句的覆盖范围为 2. C_{cove} 的取值通过启发式规则确定.我们从数据集中观察到, 62%的 *CS* 覆盖的代码行数在 1 到 9 行之间;28%的 *CS* 覆盖的代码行数在 10 到 29 行之间;只有 10%左右的 *CS* 覆盖的代码行数大于 30 行.根据这个比例,我们在不同的代码行区间内设置了 C_{cove} 不同的取值.特征 C_{iden} 表示当前代码行所包含的标识符数量.编程人员在给标识符命名时,习惯遵循驼峰命名的规范.生成 C_{iden} 特征时,以驼峰形式命名的标识符将被分解为单个的单词再进行统计,如 structuralContextFeature 分解为单词 structural, context 以及 feature.特征 $C_{idencove}$ 表示当前代码行覆盖范围内标识符数量.值得注意的是,当前代码行本身的标识符数量并不计入 $C_{idencove}$.因此,如果当前代码行的类型为 *NCS* 时,其特征 $C_{idencove}$ 等于 0.

代码间特征 $C_{commvar}$ 表示当前代码行与其上下文代码行是否调用了同一个变量.如图 2 中所示,当前代码行 5 与其上文代码行 1 调用了同一个变量 *hasBufferArgs*.$C_{commmet}$ 表示当前代码行与上下文代码行是否调用了同一个方法.图 2 中当前代码行 5 与其下文代码行 9 调用了同一个方法 *setBuffer*().为了判定两行代码是否调用了同一个变量或方法,我们首先从源代码中识别出语法类型为 *variable-invocation* 和 *method-invocation* 的代码行,然后根据这些代码行中包含的标识符名称判定两行代码是否调用了同一个变量或方法.为了准确识别每行代码的语法类型,我们使用基于抽象语法树(AST)的程序语法分析算法.特征 $C_{distance}$ 表示当前代码行与任意上下文代码行之间的距离,如图 2 中当前代码行 5 与其上文代码行 1 之间的 $C_{distance}$ 等于 5.需要注意的是,计算 $C_{distance}$ 时,空行不能被算作一行,但是大括号"{"和" }"可以被算作独立的代码行,因为它们常常表示控制类型语句的开始或结束,如图 2 中第 7, 11, 12 行.

特征 $C_{direcontr}$ 和 $C_{indircontr}$ 都是表示当前代码行和上下文代码行是否位于同一条控制类型代码行的覆盖范围内.不同的是,$C_{direcontr}$ 表示 *CuL* 与 *CoL* 位于同一条控制类型代码行的覆盖范围内,且它们之间不存在其他控制类型代码行.而 $C_{indircontr}$ 则表示 *CuL* 与 *CoL* 之间还存在其他控制类型代码行.如图 2 所示,当前代码行 5 和代码行 6 位于 *if* 控制语句(第 4 行)的覆盖范围内,它们满足 $C_{direcontr}$ 特征.当前代码行 5 和代码行 8 位于 *for* 控制语句(第 3 行)的覆盖范围内,但是它们之间还存在另外一条控制语句,即第 4 行的 *if* 语句.因此,当前代码行 5 和代码行 8 满足 $C_{indircontr}$ 特征.为了判定 $C_{direcontr}$ 和 $C_{indircontr}$ 特征,首先需要识别出代码片段中所有的控制类型代码行.我们再次使用基于抽象语法树(AST)的程序语法分析算法.该算法可以定位出每条控制类型代码行的开始位置和结束位置,从而识别出它的覆盖范围.最终便可以判定当前代码行和上下文代码行是否位于同一条控制类型代码行的覆盖范围内.为了表征当前代码行和上下文代码行之间控制类型代码行的嵌套程度,提出了特征 $C_{contrweigh}$.它通过计算当前代码行与任意上下文代码行之间存在的控制类型语句的权重之和获得.对于每种类型的控制类型代码行,随机生成一个位于 0 到 1 之间的 10 位浮点数作为其权重(比如,*if* 语句的权重为 0.1239239729,*for* 语句的权重为 0.2469030123).则图 2 中当前代码行 5 和上文代码行 1 之间的 $C_{contrweigh}$ 特征值等于 2x0.1239239729+ 0.2469030123(即,两个 *if* 语句加上一个 *for* 语句).

关于结构上下文特征的详细介绍如表 1 所示,表中给出了每种类型特征的形式化描述及其取值范围.为了获取当前代码行 *CuL* 完整的结构上下文特征,需要分析 *CuL* 与其前后 η 条上下文代码之间的代码间特征.比如,η 等于 5 时, *CuL* 与其上文 5 条代码中的每一条代码之间有 6 种代码间特征,共产生 30 维特征;同样,*CuL* 与其下文 5 条代码中的每一条代码之间也有 6 种代码间特征,再产生 30 维特征.再加上 *CuL* 本身具有的 5 维代码自特征,所以对于任意一条当前代码行 *CuL*,其完整的结构上下文特征共有 65 维. 此外,为了获得当前代码行的任意

上文(或下文)代码行与其他上文(或下文)代码行之间的结构特征,我们生成了任意上文(或下文)代码行与其他上文(或下文)代码行之间的结构上下文特征.为了使问题简化,我们只生成上文(或下文)代码行与其他上文(或下文)代码行之间的结构特征.我们将在实验章节给出使得 *CommentAdviser* 表现最优的 η 值.

2.1.2 语义上下文特征提取

程序代码在某种程度上与自然语言类似,因为代码中的标识符和自然语言中的单词一样,具有丰富的语义信息[15].我们假设一个程序中的某段代码片段被添加了注释,则位于其他程序中的具有相似语义的另外一段代码片段也很有可能被添加注释.如果这个假设成立,便可以利用代码片段之间的语义相似度,使用基于学习的方法去预测一段代码需要添加注释的可能性.但是两段具有相似语义的代码片段可能在标识符命名上选用不同的单词,这便会大大削弱常规语义相似度计算方法的效力,很难获取到两块代码片段的真实相似度.为了克服这个困难,我们引入了词嵌入(Word Embeddings)技术[16]对代码中出现的单词进行编码处理.

使用词嵌入方法编码单词时,单词不再被视为独立的符号,而是映射到一个稠密的,具有实值的高维向量空间中[16].而其中每个维度表示该单词的潜在语义或句法特征.因此,语义相似的单词在低维向量空间中具有更小的夹角.更重要的是,词嵌入方法将位于相似上下文语境中的单词看作是语义相似的[16].因此,即使是用词不同的两个标识符,只要它们位于相似的上下文代码中,词嵌入方法也认为它们是语义相似的,这样便可以很好的反应两块代码片段真实的相似度,避免使用常规语义相似度计算方法遇到的问题.

词嵌入方法是基于无监督学习的,它可以从大量无标签的文本数据中学习单词的向量表示.文中,方法体中的源代码被当做文本处理.首先,从源代码中删除标点符号及运算符号,并分割以驼峰形式命名的标识符; 然后过滤掉无意义的单词(如:*aaa*,*xxx* 等)以及虚词(如:*a*,*an*,*is* 等).同时,为了减少了整个语料库的词汇量,将过去时,未来时态,完成时态等形式出现的动词统一转化为一般现在时;最后,过滤掉在整个语料库中出现次数少于 3 次的单词.经过以上处理后,便可以获得源代码的文本形式.

为了获得源代码中每个单词的向量表示,我们使用 skip-gram 模型学习每个单词的向量表达[16].我们所需要的单词向量表达其实是 skip-gram 模型在训练过程中的中间产物,skip-gram 模型在已知当前单词的情况下,擅长在 $2k+1$ 的上下文窗口中预测周围单词(通常,$k = 2$,窗口大小为 5). skip-gram 模型的目标函数是最大化以下对数概率之和:

$$\sum_{i=1}^{n} \sum_{-k \le j \le k,\, j \ne 0} \log p(w_{i+j} \mid w_i)$$

其中,w_i 和 w_{i+j} 分别表示在 $2k+1$ 的窗口中位于中心的单词以及其上下文单词.n 表示单词序列的长度,$p(w_{i+j}|w_i)$是用 *softmax* 函数定义的条件概率:

$$\log p(w_{i+j} \mid w_i) = \frac{exp(v'^{T}_{w_{i+j}} v_{w_i})}{\sum_{w \in W} exp(v'^{T}_{w} v_{w_i})}$$

v_w 和 v'_w 分别表示单词 w 在 skip-gram 模型中的输入向量和输出向量,W 为整个语料库.可以看出, $p(w_{i+j}|w_i)$ 是计算给定单词 w_i 时,单词 w_{i+j} 出现在 w_i 的上下文中的概率.文中,我们使用负采样(negative sampling)优化该概率值的计算[16].

skip-gram 模型训练完成后,语料库中的每个单词都与一个向量相关联,形成单词-向量词典.为了获得当前代码行的上下文代码的语义信息,对于上下文代码行中的每个单词,从字典中找出与之对应的向量表示. 然后将每个单词的向量按维度叠加在一起.因为每个单词的向量具有上百维度(如:300 维),因此,最终上下文语义特征由百维向量表示.

2.2 特征选择与模型训练

从代码中提取结构上下文特征和语义上下文特征之后,将这两种特征向量首尾相连便生成当前代码行的特征向量.值得注意的是,当方法体中的第一行代码作为当前代码时,它没有上文代码行.类似地,方法体中的最

后一行代码作为当前代码时,它没有下文代码行.此时,为了保持向量维度不变,上文代码行(或下文代码行)的特征向量都用相同维度的零向量填充.

由于有两种类型的上下文特征,并且每种类型的特征都可以被分为更细粒度的特征,这将最终导致高维特征向量出现.然而,特征向量中的某些维度有可能是无关紧要的,甚至它们对预测模型的建立起负面作用. 因此,有必要去除不相关的特征维度,从而降低整个特征向量的维数.文中,我们采用信息增益去除掉不相关的特征维度.信息增益最初用于决定决策树中属性的排序,它被广泛的用于评估特征向量中每个维度的重要性[17].对于处理后的特征向量,我们可以使用一系列的机器学习算法(例如随机森林,决策树,朴素贝叶斯,支持向量机等)来训练并获取注释决策支持模型.在实验分析部分,我们将进一步评估每种机器学习算法的准确率.

2.3 数据收集

我们在 10 个开源项目上进行了数据收集,如表 2 所示.它们是 Ant, ANTLR, ArgoUML, ConQAT, JDT, JHotDraw, JabRef, JFreeChart, Lucene,Vuze.所有这些项目都是用 Java 语言编码,并且具有 10 万行以上的代码规模.我们基于以下因素选择这 10 个项目作为数据集:首先,这些项目都是开源的,它们在软件工程相关的研究中具有较高的使用频率.因此,它们拥有广大的用户基础和贡献者;其次,这 10 个项目的版本更新较为活跃(大于 2000+的代码提交)和悠久的演化历史(大于 10 年的演化历史).我们从中选择相对稳定的版本作为数据集.最后,大部分这些项目隶属于正规组织(如:Apache)或软件公司(如:IBM).因此,这 10 个项目的具有较高的代码质量.

Table 2 Data set

表 2 数据集

项目	项目来源	发布年份	演化历史(年)	方法总数	选定版本
Ant	Apache	2000	17	10667	1.10.1
ANTLR	ANTLR team	1992	25	2723	4.5.2
ArgoUML	ArgoUML team	2002	15	13494	0.34
ConQAT	ConQAT team	2007	10	9808	2013.12
JDT	IBM	2003	14	59271	4.0
JHotDraw	Open source	2000	17	2569	7.0.6
JabRef	JabRef team	2003	14	7306	3.6
JFreeChart	JFreeChart team	2000	17	9913	1.0.19
Lucene	Apache	2004	13	23872	6.3.0
Vuze	Vuze team	2003	14	26747	5.7.4.0

我们相信这 10 个项目的源代码中具有较规范的注释实例.因为这些项目为了满足开源的需求,必然需要提供高质量的代码注释供项目使用者或审核者进行代码分析,代码理解及代码审查.尤其是在经历了 10 多年的版本演化之后,项目中的注释实例达到了一个较高的水准.此外,这 10 个项目都由正规组织或软件公司中有经验的编程人员开发和实施的,组织和公司内部必然会严格控制代码质量(包括注释质量). 因此,这 10 个项目中的注释实例非常适合用于注释决策的研究.

我们从这 10 个目标项目的源代码中收集包括正、负样本的数据集.其中,有注释的代码行对应正样本,而没有注释的代码行对应负样本. 具体来说,对于方法中的每条代码行,如果在它上一行有块注释(需要说明的是,由于行注释随意性较强,*CommentAdviser* 不考虑行注释的决策),则该条代码行被标记为正样本,它对应的标签为‘1’. 否则,被标记为负样本,其对应的标签为‘0’.但是,并不是所有的正样本都纳入数据集,因为某些正样本可能是噪声数据.例如,我们观察到有些注释是以“// TODO”的形式存在.很显然,这样的注释是由工具(例如:Eclipse)自动生成的标记,又或者是编程人员人为添加的标记,用于提醒自己还未实现的功能.因此,“// TODO”被看作注释时没有实际意义.第二种噪声数据是被注释掉的代码行,例如 “// if(obj.getName()== null)”.这种情况在源代码中很常见,它们可能是测试代码,或者是开发人员改变原来的实现方案而废弃的代码.要过滤掉这样的注释,我们定义了一系列特定的正则表达式来检测它们.另外,还有一些特定的注释也需要被过滤.例如,不包含英文单词的注释(如: “//:+ =”),工具自动生成的注释(如://Auto-generated catch block)等.

3 实验验证

3.1 研究问题

本节将从多个角度评估 *CommentAdviser* 的表现. 文章的主要目标是使用 *CommentAdviser* 判定当前代码行是否需要添加注释,并将结果推荐给编程人员,辅助编程人员做出注释决策.因此,我们主要关注以下几方面的问题:

问题 1: *CommentAdviser* 判定注释点的精确度和召回率分别是多少?

问题 2: 不同的 η 值对 *CommentAdviser* 的准确率是否有影响?

问题 3: 两种类型的特征对 *CommentAdviser* 的准确率有何影响?

问题 4: 不同机器学习算法对 *CommentAdviser* 的准确率有何影响?

问题 5: 跨项目评估是否对 *CommentAdviser* 的准确率有影响?

3.2 评估方法

实验中,我们调用 Weka 编程接口[18]进行机器学习模型训练与结果评估.Weka 工具包实现了大部分常见的机器学习算法,包括: 朴素贝叶斯,LogitBoost,随机树,决策树,支持向量机,随机森林等等.针对训练集中正负实例比例失衡的问题(正负比例大约为 1:9.5),使用 Weka 工具包自带的 SMOTE 算法[19],人为的生成部分正样本,使得训练集中的正负样本比例均衡,而测试集中的正负样本保持原始比例不变.

在评估 *CommentAdviser* 的效果时,我们在正样本集(Positive Instances)和负样本集(Negative Instances)上考察了 3 个指标:精确度(*precision*, *PRE*),召回率(*recall*, *REC*),以及 F 值(*F-measure*, *FM*).它们的定义如下:

$$PRE = \frac{TP}{TP + FP} \qquad REC = \frac{TP}{TP + FN} \qquad FM = 2 \times \frac{PRE \times REC}{PRE + REC}$$

其中,*TP* 表示被正确分类的正样本;*TN* 表示被正确分类的负样本;*FP* 表示被错误分类的正样本;*FN* 表示被错误分类的负样本.精确度用来衡量结果集的精确性,表示被 *CommentAdviser* 判定为需要添加注释的代码行集合与数据集中真正添加了注释的代码行集合的吻合程度;而召回率用来度量结果集的安全性,表示被 *CommentAdviser* 判定为需要添加注释的代码行集合能够覆盖真正添加了注释的代码行集合的程度. *FM* 是调和平均数,它是一个用来评估精确度和召回率的综合指标.精确度,召回率,以及 *FM* 的定义同样适用于负样本集的评估.

4 实验结果

4.1 问题1: *CommentAdviser*判定注释点的精确度和召回率分别是多少?

Table 3 The number of positive and negative instances in the data sets

表 3 数据集中正负样本数量

	Ant	**ANTLR**	**ArgoUML**	**ConQAT**	**JDT**
正例数	2680	644	3571	1243	3061
负例数	27756	7450	27123	13543	26369
	JHotDraw	**JabRef**	**JFreeChart**	**Lucene**	**Vuze**
正例数	529	2735	3126	3490	1718
负例数	5794	23391	27874	26509	18963

表格 3 展示了 10 个数据集中正、负样本的数量。可以看出，所有数据集中的正、负样本数量均不平衡。数据集 JHotDraw 的样本总量最少，而数据集 ArgoUML 的样本总量最多。表格 4 展示了 *CommentAdviser* 在 10 个数据集上的判定结果,包括在正样本和负样本上的精确度,召回率,*FM* 值.整个评估过程中,使用随机森林算法以及 10 折交叉验证法.根据 10 折交叉验证的定义,我们将每个数据集等分为 10 份,其中 1 份作为测试集,剩下 9 份作为训练集,依次循环 10 次,最后取 10 次判定结果的平均值.为了使训练集中的正负样本数量平衡,使用 SMOTE 算法人为的生成部分正样本,而测试集中的正负样本比例则保持不变.

从表格 4 中可以观察到,负样本的平均精确度,召回率以及 *FM* 值都超过了 0.96,这个结果表明数据集中大部分的负样本是可以被正确判定的.另一方面,在正样本集上,*CommentAdviser* 获得的平均精确度,召回率以及 *FM* 值只有 0.735,0.66,以及 0.665.其与负样本上的准确率有较大的差异.66%的召回率说明大约有三分一的正样本没被正确判定.而较低的精确度(73.5%)则有可能是因为测试集中正负样本比例相差悬殊造成的.由于这是一个二分类问题,而负样本的数量又是正样本的 10 倍,根据精确度的定义,被错误分类的负样本将会加倍降低正样本的精确度.

Table 4 Evaluation results

表 4 判定结果

数据集	Positive Instances			Negative Instances		
	PRE	***REC***	***FM***	***PRE***	***REC***	***FM***
Ant	0.364	0.764	0.493	0.971	0.856	0.910
ANTLR	0.879	0.662	0.756	0.965	0.990	0.978
ArgoUML	0.725	0.615	0.666	0.958	0.974	0.966
ConQAT	0.608	0.750	0.672	0.967	0.937	0.952
JDT	0.797	0.714	0.625	0.946	0.985	0.965
JHotDraw	0.810	0.756	0.782	0.982	0.987	0.984
JabRef	0.724	0.456	0.559	0.931	0.977	0.954
JFreeChart	0.803	0.806	0.805	0.980	0.979	0.979
Lucene	0.850	0.598	0.702	0.952	0.987	0.969
Vuze	0.794	0.464	0.586	0.963	0.992	0.977
平均值	0.735	0.660	0.665	0.961	0.966	0.963

总的来说,在正样本集上较低的精确度和召回率说明 *CommentAdviser* 并没有充分的从源代码中学习到通用的注释决策.我们有理由相信训练集中存在一小部分实例,它们代表了错误的注释决策.比如,有的实例本应该添加注释的,但其实并没有添加(此时实例的标签为 0),而另外一些实例本不应该添加注释的,却被添加了注释(此时实例的标签为 1).这些实例都会影响机器学习算法建立有效的判定模型,称之为噪音数据,因此有必要将噪音数据从数据集中过滤掉.

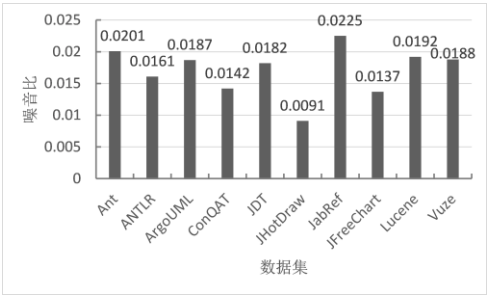


Fig. 3 Noise degree in data sets

图 3 数据集中的噪音比

为了过滤数据集中的噪音数据,使用了一种简单有效的噪音过滤方法 CLNI [20].对于任意一个被检测的实例,CLNI 判定其周围最近邻的 *k* 个实例的标签,如果在这 *k* 个实例中有 *φ*%的实例都与被检测的实例标签相反,则被检测的实例标记为噪音数据.通常,*k* 取值为 5,*φ* 取值为 60[20].图 3 中,噪音比表示了每个数据集中的噪音实例所占的比例.可以看出,在这 10 个数据集上,噪音比从 0.0091 上升到 0.0225.这个结果说明绝大部分实例与其周围的实例具有相同的标签.

如表 5 所示,经过去噪处理后,*CommentAdviser* 在 10 个数据集的负样本上获得的平均精确度,召回率以及 *FM* 值都达到或超过了 0.97.而在正样本上的平均召回率从 0.66 上升到了 0.672,平均精确度从 0.735 上升到了 0.808.同时,正样本上的平均 *FM* 值也达到了 0.733.与去噪处理前相比,*CommentAdviser* 在正样本和负样本上的所有指标均有所提升.这个结果足以说明适当的去除数据集中的噪音实例可以让 *CommentAdviser* 更有效的从源码中学习到通用的注释策略.

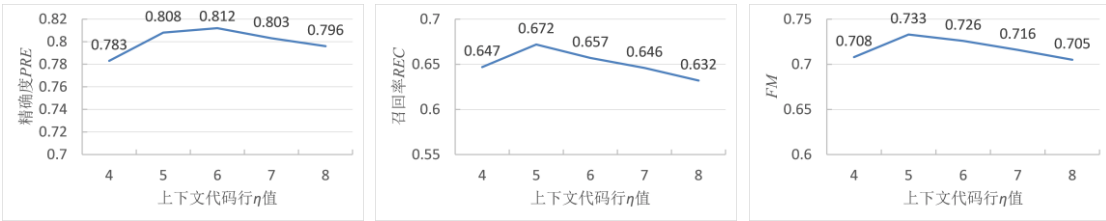
Table 5 Noise-handling evaluation results

表 5 去除噪音数据后的判定结果

数据集	Positive Instances			Negative Instances		
	PRE	***REC***	***FM***	***PRE***	***REC***	***FM***
Ant	0.787	0.613	0.689	0.968	0.986	0.977
ANTLR	0.843	0.672	0.784	0.972	0.989	0.980
ArgoUML	0.762	0.642	0.696	0.966	0.980	0.973
ConQAT	0.674	0.741	0.706	0.966	0.953	0.960
JDT	0.829	0.617	0.708	0.959	0.986	0.973
JHotDraw	0.893	0.735	0.806	0.985	0.995	0.990
JabRef	0.755	0.563	0.645	0.961	0.983	0.972
JFreeChart	0.811	0.864	0.837	0.987	0.980	0.983
Lucene	0.860	0.727	0.788	0.971	0.987	0.979
Vuze	0.866	0.550	0.673	0.971	0.994	0.982
平均值	0.808	0.672	0.733	0.970	0.983	0.977

4.2 问题2：不同的η值对*CommentAdviser*的准确率是否有影响？

文章假设代码注释决策与代码本身的上下文代码行密切相关.因此,需要确定一个最优的上下文代码行 η 值,使得 *CommentAdviser* 能够达到最佳的准确率.我们先后比较了 η 值为 4,5,6,7,8 时,*CommentAdviser* 获得的精确度,召回率,以及 *FM* 值.详细结果如图 4 所示.



(a) *PRE* (b) *REC* (c) *FM*

Fig. 4 Effectiveness of η values on *CommentAdviser*

图 4 η 值对 *CommentAdviser* 准确率的影响

由于我们关心的是 *CommentAdviser* 在代码中做出注释决策的准确率,也就是对应数据集中正样本的准确率.因此,图 4 所示的结果为 *CommentAdviser* 在 10 个数据集的正样本集上的平均精确度,平均召回率,以及平均 *FM* 值.整个判定过程中,使用随机森林算法作为分类器,并在每个数据集上使用 10 折交叉验证法.我们观察到,当 η 值等于 6 时,*CommentAdviser* 获得最佳的精确度;当 η 值等于 5 时,*CommentAdviser* 获得最佳的召回率和 *FM* 值.仔细比较可以发现,当 η 值等于 5 时,*CommentAdviser* 获得的精确度只比 η 值等于 6 时少 0.004,而此时的召回率却比 η 值等于 6 时高出 0.015,*FM* 值高出 0.007.因此,在文章的整个实验中, η 值被设置为 5,即把当前代码行的前后 5 行代码作为其上下文代码行.

4.3 问题3：两种类型的特征对*CommentAdviser*的准确率有何影响？

文中,我们假设代码注释决策与上下文代码的结构(structural)和语义(semantic)存在密切的关系,因此,提出了两种类型的特征用来判定代码中可能的注释点,即上下文结构特征和上下文语义特征.为了验证文中提出的假设,我们让 *CommentAdviser* 单独使用上下文结构特征和上下文语义特征进行注释决策判定,然后再融合两种特征进行注释决策判定.该验证同样使用随机森林算和 10 折交叉验证法.表格 6 展示了 *CommentAdviser* 使用不同特征在 10 个数据集的正样本集上的评估结果.

Table 6 Different features effect

表 6 不同特征下的判定结果

数据集	Structural			Semantic			Structural+Semantic		
	PRE	***REC***	***FM***	***PRE***	***REC***	***FM***	***PRE***	***REC***	***FM***
Ant	0.725	0.475	0.574	0.322	0.806	0.460	0.787	0.613	0.689
ANTLR	0.846	0.344	0.489	0.365	0.844	0.509	0.843	0.672	0.784

ArgoUML	0.757	0.472	0.582	0.348	0.834	0.491	0.762	0.642	0.696
ConQAT	0.698	0.518	0.595	0.385	0.888	0.537	0.674	0.741	0.706
JDT	0.809	0.452	0.580	0.440	0.828	0.574	0.829	0.617	0.708
JHotDraw	0.800	0.588	0.678	0.429	0.706	0.533	0.893	0.735	0.806
JabRef	0.646	0.528	0.581	0.310	0.817	0.450	0.755	0.563	0.645
JFreeChart	0.812	0.729	0.768	0.605	0.905	0.725	0.811	0.864	0.837
Lucene	0.843	0.542	0.660	0.479	0.846	0.612	0.860	0.727	0.788
Vuze	0.774	0.372	0.503	0.326	0.775	0.459	0.866	0.550	0.673
平均值	0.771	0.502	0.601	0.400	0.825	0.535	0.808	0.672	0.733

如表格 6 所示,当单独使用结构上下文特征时,*CommentAdviser* 在 10 个数据集上的平均精确度,召回率以及 *FM* 值分别为 0.771,0.502 和 0.601;而单独使用语义上下文特征时,*CommentAdviser* 在 10 个数据集上的平均精确度,召回率以及 *FM* 值分别为 0.4,0.825 和 0.535.可以看出,结构上下特征让 *CommentAdviser* 获得较高的精确度,而语义上下文特征能让 *CommentAdviser* 的召回率明显提高.当结合结构上下文特征和语义上下文特征时,*CommentAdviser* 能获得较均衡的精确度和召回率,此时的平均精确度,召回率以及 *FM* 值分别为 0.808,0.672,以及 0.733.这个结果表明,无论是在结构上下文特征中加入语义上下文特征,还是在语义上下文特征中加入结构上下文特征,都能对 *CommentAdviser* 准确率的提高起积极作用,从而验证了文章中提出的假设.

4.4 问题4: 不同机器学习算法对*CommentAdviser*的准确率有何影响?

在实验过程中,我们还对比了多种机器学习算法在支持注释决策时的表现.参与比较的机器学习算法全是基于监督的学习算法.它们是:朴素贝叶斯,LogitBoost,随机树,C4.5决策树,支持向量机,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, 设置随机森林算法的*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;设置*NumFeatures*=20,对于随机森林的每个分类器,随机抽取特征集的20个特征进行训练;设置*NumIterations*=300,随机森林中包含300个分类器.

Table 7 Comparison for multiple machine learning algorithms

表 7 机器学习算法 *FM* 比较

机器学习算法	正样本 *FM*	负样本 *FM*
朴素贝叶斯	0.240	0.775
LogitBoost	0.290	0.854
随机树	0.469	0.929
C4.5 决策树	0.461	0.940
支持向量机	0.710	0.958
随机森林	0.733	0.977

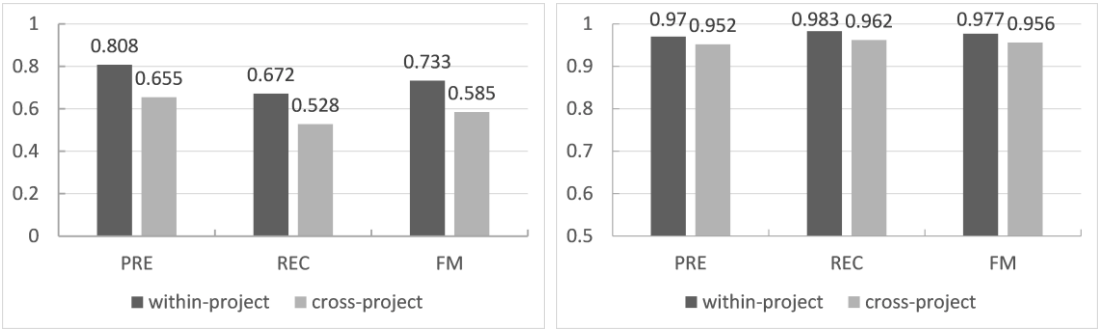
表格 7 例举了各种机器学习算法在 10 个数据集上的平均正样本 F-measure 和负样本 F-measure.从结果中可以观察到,除了支持向量机和随机森林,其他各种机器学习算法在支持注释决策上都表现欠佳.其中,随机森林和支持向量机在正样本上的 F-measure 分别达到了 0.733 和 0.71,而其余 4 个算法的 F-measure 从 0.24 到 0.469 不等.总的来说,随机森林算法的表现最优.因此,在本文全部实验中,随机森林算法被用作默认分类器.

4.5 问题5: 跨项目评估是否对*CommentAdviser*的准确率有影响?

以上 4 个问题都是在项目内(within-projet)进行的评估,即训练集和测试集来自于同一个项目的数据集.在项目内评估时,*CommentAdviser* 能够获得较高的精确度和召回率.这就使得 *CommentAdviser* 能够很好的应用于如下场景,例如:在一个项目中需要开发新的组件时, *CommentAdviser* 可以从该项目已有的注释实例中学习注释决策规范,然后应用于新开发的组件中.然而在实际开发中,很多项目都是从零开始,此时并没有可以参考的原项目,此时需要 *CommentAdviser* 通过跨项目(cross-project)学习,然后在新项目中作出注释决策.

在跨项目评估中,我们依然使用相同的 10 个数据集.不同的是,我们每次从其中一个数据集中随机抽取 2000 条实例作为训练集,而从剩下的 9 个项目中的每个项目随机抽取 2000 条实例作为训练集.一共进行 10 次这样的评估.与项目内的注释决策相比,跨项目注释决策的学习对于 *CommentAdviser* 来说更具挑战性.因为不同的项目在注释策略上可能存在较大的差异,这就使得 *CommentAdviser* 很难学习到一种能够适合所有数据集的注释策略.为了尽量提升 *CommentAdviser* 的准确率,我们选择那些在项目间被频繁共同使用的特征用作判别模

型的训练.比如,代码自特征 C_{loca} 和 C_{type},以及代码间特征 $C_{commvar}$ 和 $C_{commmet}$ 等等.



(a) 正样本 (b)负样本

Fig. 5 Cross-projects evaluation results

图 5 跨项目评估结果

图 5 展示了跨项目评估与项目内评估的结果对比.跨项目评估时,*CommentAdviser* 在正样本上的精确度从 0.808 降到了 0.655,而召回率从 0.672 降到了 0.528.这说明 *CommentAdviser* 在进行跨项目学习时,它的效力出现了较严重的退化,同时也印证了不同项目的注释策略存在较大的差异.而对于负样本的评估,*CommentAdviser* 在精确度,召回率,以及 FM 值上也呈现出不同程度的下降.我们将在用户调研章节进一步分析造成 *CommentAdviser* 低精确度和召回率的原因.

5 用户调研

CommentAdviser 在正样本数据集上表现出较低的召回率,这说明存在很多漏判的情况(即,原本添加了注释的代码行被判定为不用添加注释,对应 *FP*),而较低的精确度则说明存在很多误判的情况(即,没添加注释的代码行被判定为应该添加注释,对应 *FN*).我们生成了所有数据集上被 *CommentAdviser* 漏判或误判的实例.通过分析这些被漏判或误判的实例,发现了大量重复出现的案例.如图 6(a)所示,图中的标记“//*********”表示被 *CommentAdviser* 误判的注释; 标记“//#########” 表示被 *CommentAdviser* 漏判的注释.这种案例具有的一个共同特点,即被 *CommentAdviser* 误判代码注释附近总是存在漏判的代码注释,它们之间的距离一般不超过 3 行,如图 6 (a)所示,我们称之为“误判漏判注释对”.通过观察大量的“误判漏判注释对”,发现它们的作用范围存在交叉或覆盖的情况.如图 6 (a)所示,误判的注释①的作用范围覆盖了整个 *if* 语句,而漏判的注释②的作用范围覆盖了整个 *if* 语句的内部代码.因此,我们提出了一个假设,即,“误判漏判注释对”对于编程人员理解代码具有等价作用.也就是说,误判和漏判的注释虽然在代码中处于不同的位置,但是它们辅助编程人员理解代码时能够发挥同等的作用.

```
int n = adaptor.getChildCount(tree);
① // *********
if ( n==0 ) {
    ② // #########
    return;
}
```

(a)误判漏判注释对

```
题目 1，代码片段如下：
    int n = adaptor.getChildCount(tree);
    ① // *********
    if ( n==0 ) {
        ② // #########
        return;
    }
1）为了使上述代码片段变得更容易理解，请选择您认为最需要添加注释的地方：
   A）在位置①处    B）在位置②处    C）在位置①和②处都添加
2）您认为理解该段代码的难度为：
   A）极易    B）较易    C）适中    D）较难    E）极难
```

(b) 问卷题目

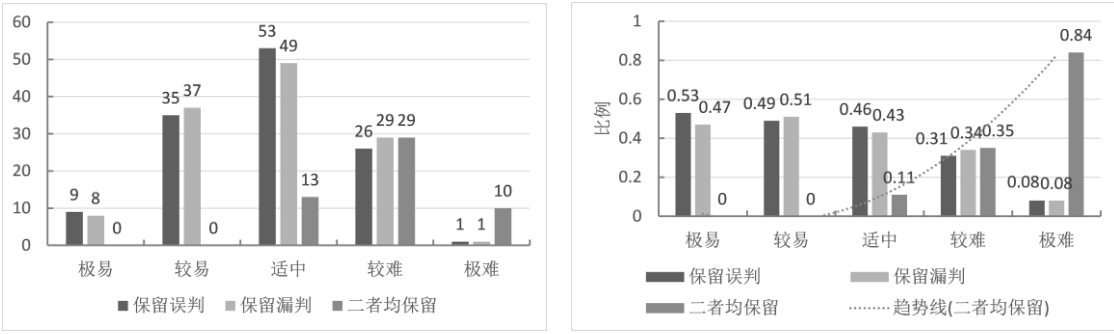
Fig. 6 *FP*&*FN* pair and questionnaire

图 6 误判漏判注释对及问卷题目

为了验证这个假设,我们以问卷的方式向20名参与者发起了用户调研.在这20位参与者中,有1位高校教师,4位博士研究生以及15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, 平均拥有2.5年以上Java 编程经验.我们从所有误判和漏判案例中随机抽取了15个“误判漏判注释对”,对应问卷调

查中的15道题目,每道题目共设置两个问题.参与者首先回答他/她认为代码片段中应该添加注释的位置,他/她们事先并不知道标记“//*********”和“//#########”的区别.然后再回答理解该段代码的难度,难度等级分为极易,较易,适中,较难,极难.题目如图6(b)所示.

我们按照参与者反馈的问题难易程度对问卷结果进行了统计,如图7所示.面对极易的题目时,有9次参与者选择保留误判注释,8次选择保留漏判注释,如图7(a)所示;面对较易的题目时,有35次参与者选择了保留误判,37次选择了保留漏判.由此可知,对于极易和较易的代码片段来说,参与者选择保留误判或漏判注释的概率几乎一样(即0.49:0.51,如图7(b)所示);当面对难度适中的题目时,有53次参与者选择了保留误判,49次选择保留漏判,13次选择二者均保留.这个结果同样表明参与者在面对难度适中的代码片段时,对于选择保留误判还是选择漏判没有显著性的差异;当面对难度较难的题目时,有26次参与者选择了保留误判,29次选择保留漏判,29次选择二者均保留.可以看出,此时参与者更多的是希望二者均保留,而对于选择保留误判还是选择漏判还是没有显著性的差异.当面对极难的题目时,参与者在84%以上的情形下都会选择二者均保留,如图7(b)所示.通过对比所有难度下保留误判,保留漏判,以及二者均保留的次数,可以发现它们的比例分别为37.4%,36.8%,以及25.8%.这个比例再次证明参与者在选择保留误判和选择漏判上没有显著性的差异.



(a)数量对比 (b)比例对比

Fig. 7 Questionnaire result

图7 问卷调查结果统计

因此,可以接受我们提出的假设,即“误判漏判注释对”中的误判注释和漏判注释对于编程人员理解代码具有等价作用.事实上,想要预测添加的注释在代码中的精确位置是件非常困难的事情.因为当前代码行的上下文代码有成千上万种可能,在其上下文代码中增加或减少一行代码,便可能影响当前代码的注释决策.这个结论也可以从*CommentAdviser*在正样本数据集上出现比较严重的漏判和误判中得出.但是从用户调研结果中可以看出,很多误判的注释其实是与漏判注释存在作用范围交叉或覆盖的情况.也就是说*CommentAdviser*作出的注释决策虽然没能恰好落在漏判的注释上,但是却落在它的附近(不超过3行代码),而这个距离能够帮助编程人员借助推荐的注释点理解代码.因此,如果将“误判漏判注释对”也算作有效的注释决策,*CommentAdviser*的真实准确率会相应提高.

6 相关工作

注释决策的主要作用是告知编程人员应该在代码中何处添加注释,从而有效的提高编程人员的开发效率.本文首先分析代码行的上下文结构信息和语义信息,然后使用机器学习的算法定位出可能添加注释的代码行,最后将注释决策推荐给开发人员.本节主要从 3 个方面介绍与本文相关的研究工作.首先介绍基于抽象语法树的程序结构分析;然后介绍词嵌入技术在软件工程相关研究中的应用;最后介绍与代码注释相关的研究和实践.

6.1 基于抽象语法树的程序结构分析

CommentAdviser 首先分析代码自身和代码之间的结构信息,从而获得代码的结构特征. 文中提出的大部分结构特征是通过分析代码的抽象语法树获得的.近年来,许多研究人员通过解析代码的抽象语法树而获得代码的语法结构,然后将代码的语法结构用于各种各样与软件工程相关的研究中.

Mou 等人[14]设计了一个基于抽象语法树的卷积神经网络,他们提出的网络在抽象语法树上做卷积运算,此操作更能够凸显出抽象语法树上的代码结构特征.最终,他们的方法被用于程序功能自动识别中.此外,抽象语法树被广泛的应用于基于树结构的代码克隆检测中.Dyer 等人[21]从 9 百万个 Java 文件中挖掘出超过十亿个抽象语法树节点,他们试图从这些海量的抽象语法树节点中分析出 Java 版本发布的新特性在不同时间段内的使用情况,最终他们分析出了最受欢迎的 Java 新特性及其随着时间变化的使用情况. White 等人[22]提出了一种基于语法与语义的代码克隆检测方法.与传统的基于树结构的代码克隆检测方法不同,他们在提取代码结构信息时将抽象语法树转换为满二叉树.与此同时,也有一些学者将抽象语法树应用于代码缺陷预测中,Wang 等人[23]利用深度学习算法自动获得代码漏洞与代码语义信息之间的映射关系.在他们提出的方法中,代码的语义信息从程序对应的抽象语法树上获得.为了修复代码中反复出现的崩溃错误,Gao 等人[24]从编程问答网站上挖掘出可用于修复崩溃错误的 fixed 代码片段.他们通过比较 buggy 代码片段与 fixed 代码片段的抽象语法树,从而获得从 buggy 代码片段转化到 fixed 代码片段的自动修复脚本.除此之外,抽象语法树也成功地应用于编程模式识别[25],代码修改模式识别[26],软件演化[27]等领域.

6.2 词嵌入技术在软件工程相关研究中的应用

为了解决 one-hot 技术中可能出现的维度灾难问题,研究人员引入了词嵌入技术(word embedding)[16]. word2vec 简化了词嵌入技术原有的模型,使得训练速度大为提升,使其成为最常见也是最实用的一种词嵌入技术.因此,word2vec 也被众多学者应用于软件工程相关的研究中.

Xu 等人[28]认为开发者问答社区(stack overflow)上围绕每个问题都能构成一个知识单元.而在解答问题的过程中,开发者往往会引用其他相关联的知识单元来解决当前问题.为了度量知识单元之间的语义相关性,他们使用 word2vec 将知识单元中的单词向量化,然后利用卷积神经网络预测两个知识单元之间的相关性.Guo 等人[29]将 word2vec 应用于可追踪性链恢复领域.由于传统的可追踪性链恢复方法都是基于信息检索的方式,很难将深层次的语义知识和领域知识融入到追踪链的建立过程中.因此,他们利用 word2vec 技术建模软件需求文档与实现文档中的深层次语义和领域知识,然后使用循环神经网络学习需求文档与实现文档在语义层面的信息,最终生成需求文档与实现间的可追踪性链.软件文档与代码之间存在的语义鸿沟是基于搜索的软件工程中面临的巨大挑战,Ye 等人[30]首先利用 word2vec 技术将软件文档与代码映射到同一个向量空间中,然后再计算软件文档与代码之间的语义相似度.他们的方法在基于搜索的代码漏洞定位中表现出良好的效果.除此之外,为了协助中文用户能顺利使用开发者问答社区(stack overflow),有学者利用 word2vec 和卷积神经网络将中文问题翻译为对应的英文问题[31]. 程序员在修复相似的代码漏洞时往往具有更高的效率.因此,有学者将 word2vec 技术应用于相似代码漏洞推荐上[32].他们使用 word2vec 将与漏洞描述相关的所有信息进行词向量化,然后计算任意代码漏洞之间的相似性.

6.3 代码注释相关的研究和实践

高质量的代码注释可以很好的用于程序理解和代码维护等实践活动中.目前与代码注释相关的实践和研究主要关注 2 方面的问题:代码注释质量评估与代码注释自动生成.

很多学者提出了各种各样的方法用于评估代码中的注释质量.据我们所知,最早关于注释质量评估的方法可以追溯到 1992 年[7],Oman 等人最先通过计算源代码中注释所占的比例来评估代码注释质量,显然他们的方法是不够精确和严谨的.随后,Steidl 等人[2]提出了一种基于机器学习的方法用于评估注释质量,他们的方法基于一系列简单的评判标准,比如注释的有用性,完整性等,然后使用决策树来判别注释质量.Khamis 等人[3]开发了一个注释质量评价工具－JavadocMiner.他们的工具基于一套启发式算法实现,比如:代码与注释的关联性等. 除此之外, Storey 等人[33]都在源代码中的注释上做了实证研究,他们的研究结果揭示了代码注释在软件开发过程中是如何支持各种软件活动的.

自动地生成代码注释是研究人员一直追求的目标.Wong 等人[34]提出了一种从 StackOverflow 上挖掘大规模问答数据来自动生成代码注释的方法.他们的方法计算输入代码片段和 StackOverflow 上的代码片段之间的相似性,并将 StackOverflow 上的代码片段的描述用于生成输入代码片段的注释.之后,Wong 等人[35]又提出了另

一种通过挖掘现有软件代码库自动生成代码注释的方法.他们使用代码克隆检测技术从软件代码库中发现语法相似的代码片段,并将代码注释应用于其他语法类似的代码片段上.此外,Sridhara 等人[5]提出了一种新颖的方法用来生成 Java 方法的注释.给定一个方法的签名和方法体,他们的方法可以自动生成注释用于描述该方法的动作. Moreno 等人[6]提出了一种生成 Java 类注释的方法.他们方法生成的注释主要描述了类本身涵盖的内容以及该类所起的作用.

目前与代码注释相关的研究大部分都是关于注释质量评估和注释自动生成的,而没有直接与注释决策支持相关的.有学者提出了代码日志决策支持方法[36],他们的方法与我们的方法有类似之处.但是不同之处在于,他们的方法仅仅是告知编程人员一段代码是否需要添加日志,而我们的方法是告知编程人员需要在代码中何处添加注释.

7 结论

代码注释在程序理解和软件维护中扮演着极其重要的角色,恰当的代码注释决策成为了软件编程人员追求的目标.文章提出了一种新颖的 *CommentAdviser* 方法用以辅助软件编程人员在代码开发过程中做出恰当的注释决策.*CommentAdviser* 从当前代码行的上下文代码中提取代码结构特征以及代码语义特征作为支持注释决策的主要依据.然后,利用机器学习算法判定当前代码需要添加注释的可能性.在 GitHub 中的 10 个数据集上评估了提出的方法,实验结果表明了 *CommentAdviser* 的可行性和有效性. 在未来的研究工作中,我们将从当前代码行的上下文代码语法分布入手,加入更多的可判别特征,进一步提高 *CommentAdviser* 的精确度和召回率.同时,我们打算将 *CommentAdviser* 方法嵌入到 IDE 中.开发人员编写代码的同时,*CommentAdviser* 向其提供注释决策建议.

References:

[1] Tenny T. Program readability: Procedures versus comments. IEEE Trans. on Software Engineering, 1988, 14(9):1271–1279.

[2] Steidl D, Hummel B, Juergens E. Quality analysis of source code comments. In: Proc. of the 21th Int'l Conf. on Program Comprehension, 2013, 83–92.

[3] Khamis N, Witte R, Rilling J. Automatic quality assessment of source code comments: The javadocminer. In: Proc. of the Int'l Conf. on Natural Language Processing and Information Systems, 2010, 68–79.

[4] Fluri B, Wursch M, Gall HC. Do code and comments co-evolve? on the relation between source code and comment changes. In: Proc. of the 14th Int'l Conf. on Working Conference on Reverse Engineering, 2007, 70–79.

[5] Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for java methods. In: Proc. of the Int'l Conf. on Automated Software Engineering, IEEE/ACM, 2010, 43–52.

[6] Moreno L, Aponte J, Sridhara G. A. Marcus, L. Pollock, and K. Vijay- Shanker. Automatic generation of natural language summaries for java classes. In: Proc. of the 21th Int'l Conf. on Program Comprehension, 2013, 23–32.

[7] Oman P, Hagemeister J. Metrics for assessing a software system's maintainability. In: Proc. of the Int'l Conf. on Software Maintenance, 1992, 337–344.

[8] Dit B, Holtzhauer A, Poshyvanyk D, Kagdi HH. A dataset from change history to support evaluation of software maintenance tasks. In: Proc. of the Int'l Conf. on Mining Software Repositories, 2013, 131–134.

[9] Huang Y, Chen XP, Liu ZY, Luo XN, Zheng ZB. Using discriminative feature in software entities for relevance identification of code changes. Journal of Software: Evolution and Process, 2017.

[10] Huang Y, Zheng. QY, Chen XP, Xiong YF, Liu ZY, Luo XN. Mining version control system for automatically generating commit comment. In: Proc. of the 11th Int'l Conf. on Empirical Software Engineering and Measurement, 2017.

[11] Zhang J, Chen JJ, Hao D, Xiong YF, Xie B, Zhang L, Mei H. Search-based inference of polynomial metamorphic relations. In: Proc. of the 29th Int'l Conf. on Automated Software Engineering, IEEE/ACM, 2014, 701–712.

[12] Gosling J, Joy B, Steele G, Bracha G, Buckley A. The java language specification (java se 8 edition), java se 8 ed, 2014.

[13] Vermeulen A, Ambler SW, Bumgardner EMG, Misfeldt T, Shur J, Thompson P. The elements of java style, 2000.

[14] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th Int'l Conf. on Artificial Intelligence, 2016, 1287–1293.

[15] Beniamini G, Gingichashvili S, Orbach AK, Feitelson DG. Meaningful identifier names: The case of singleletter variables. In: Proc. of the 25th Int'l Conf. on *Program Comprehension*, 2017, 45–54.

[16] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 26th Int'l Conf. on Neural Information Processing Systems, 2013, 3111–3119.

[17] Aggarwal CC, Zhai C. A survey of text classification algorithms. in Mining text data. Springer, 2012, 163–222.

[18] Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The weka data mining software: An update. *SIGKDD* explorations newsletter, 2009, 11(1), 10-18.

[19] Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. Smote: Synthetic minority over-sampling technique. Journal of artificial intelligence research, 2002, 16(1), 321-357.

[20] Kim SH, Zhang HY, Wu RX, Gong L. Dealing with noise in defect prediction. In: Proc. of the 33th Int'l Conf. on Software Engineering, 2011, 481–490.

[21] Dyer R, Rajan H, Nguyen HA, Nguyen TN. Mining billions of ast nodes to study actual and potential usage of java language features. In: Proc. of the 36th Int'l Conf. on Software Engineering, 2014, 779–790.

[22] White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proc. of the 31th Int'l Conf. on Automated Software Engineering, 2016, 87–98.

[23] Wang S, Liu TY, Tan L. Automatically learning semantic features for defect prediction. In: Proc. of the 38th Int'l Conf. on Software Engineering, 2016, 297–308.

[24] Gao Q, Zhang HS, Wang J, Xiong YF, Zhang L, Mei H. Fixing recurring crash bugs via analyzing q&a sites (t). In: Proc. of the 30th Int'l Conf. on Automated Software Engineering, 2015, 307–318.

[25] Nguyen AT, Nguyen TN. Graph-based statistical language model for code. In: Proc. of the 37th Int'l Conf. on Software Engineering, 2015, 858–868.

[26] Negara S, Codoban M, Dig D, Johnson RE. Mining fine-grained code changes to detect unknown change patterns. In: Proc. of the 36th Int'l Conf. on Software Engineering, 2014, 803–813.

[27] Jiang QT, Peng X, Wang H, Xing ZC, Zhao WY. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes. In: Proc. of the 22th Int'l Conf. on Software Analysis, Evolution, and Reengineering, 2015, 361–370.

[28] Xu BW, Ye DH, Xing ZC, Xia X, Chen GB, Li SP. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: Proc. of the 31th Int'l Conf. on Automated Software Engineering, 2016, 51-62.

[29] Guo J, Cheng JH, Cleland-Huang J. Semantically enhanced software traceability using deep learning techniques. In: Proc. of the 39th Int'l Conf. on Software Engineering, 2017, 3-14.

[30] 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 38th Int'l Conf. on Software Engineering, 2016, 404-415.

[31] Chen GB, Chen CY, Xing ZC, Xu BW. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In: Proc. of the 31th Int'l Conf. on Automated Software Engineering, 2016, 744-755.

[32] Yang XL, Lo D, Xia X, Bao LF and Sun JL. Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. In: Proc. of the 27th Int'l Conf. on Software Reliability Engineering, 2016, 127-137.

[33] Storey MA, Ryall J, Bull RI, Myers D, Singer J. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In: Proc. of the 30th Int'l Conf. on *Software Engineering*, 2008, 251–260.

[34] Wong E, Yang JQ, Tan L. Autocomment: Mining question and answer sites for automatic comment generation. In: Proc. of the 28th Int'l Conf. on Automated Software Engineering, 2013, 562–567.

[35] Wong E, Liu TY, Tan L. Clocom: Mining existing source code for automatic comment generation. In: Proc. of the 22th Int'l Conf. on Software Analysis, Evolution, and Reengineering, 2015, 380–389.

[36] Zhu JM, He PJ, Fu Q, Zhang HY, Lyu MR, Zhang DM. 2015. Learning to log: helping developers make informed logging decisions. In: Proc. of the 37th Int'l Conf. on Software Engineering, 2015, 415-425.