软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(4):962−980 [doi: 10.13328/j.cnki.jos.005711]

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

代码克隆检测研究进展*

陈秋远[1], 李善平[1], 鄢 萌[1], 夏 鑫[2]

[1](浙江大学 计算机科学与技术学院,浙江 杭州 310007)

[2](Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia)

通讯作者: 鄢萌, E-mail: mengy@zju.edu.cn

摘 要: 代码克隆(code clone),是指存在于代码库中两个及以上相同或者相似的源代码片段.代码克隆相关问题是软件工程领域研究的重要课题.代码克隆是软件开发中的常见现象,它能够提高效率,产生一定的正面效益.但是研究表明,代码克隆也会对软件系统的开发、维护产生负面的影响,包括降低软件稳定性,造成代码库冗余和软件缺陷传播等.代码克隆检测技术旨在寻找检测代码克隆的自动化方法,从而用较低成本减少代码克隆的负面效应.研究者们在代码克隆检测方面获得了一系列的检测技术成果,根据这些技术利用源代码信息的程度不同,可以将它们分为基于文本、词汇、语法、语义 4 个层次.现有的检测技术针对文本相似的克隆取得了有效的检测结果,但同时也面临着更高抽象层次克隆的挑战,亟待更先进的理论、技术来解决.着重从源代码表征方式角度入手,对近年来代码克隆检测研究进展进行了梳理和总结.主要内容包括:(1) 根据源代码表征方式阐述并归类了现有的克隆检测方法;(2) 总结了模型评估中使用的实验验证方法与性能评估指标;(3) 从科学性、实用性和技术难点这 3 个方面归纳总结了代码克隆研究的关键问题,围绕数据标注、表征方法、模型构建和工程实践 4 个方面,阐述了问题的可能解决思路和研究的未来发展趋势.

关键词: 代码克隆;克隆检测;代码表征

中图法分类号: TP311

中文引用格式: 陈秋远,李善平,鄢萌,夏鑫.代码克隆检测研究进展.软件学报,2019,30(4):962−980. http://www.jos.org.cn/1000-9825/5711.htm

英文引用格式: Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Ruan Jian Xue Bao/Journal of Software, 2019,30(4):962−980 (in Chinese). http://www.jos.org.cn/1000-9825/5711.htm

Code Clone Detection: A Literature Review

CHEN Qiu-Yuan[1], LI Shan-Ping[1], YAN Meng[1], XIA Xin[2]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China)

[2](Faculty of Information Technology, Monash University, Melbourne VIC 3800, Australia)

Abstract: Code clone refers to more than two duplicate or similar code fragments existing in a software system. Code clone is a common phenomenon during software development which can facilitate development and has positive impacts on software system. However, research shows that code clone will also do harm to the development and maintenance of software system, including but not limited to the decline of stability, redundancy of source code repository, and propagation of software defects. Code clone is one of the most active research areas in software engineering. Therefore, various detection techniques are proposed to automatically detect code clone in software systems, which help improve software quality. There are a lot of achievements in this area, and these techniques can be categorized to text-based, lexis-based, syntax-based, and semantic-based categories. Current techniques have obtained effective results in text-based clone detection, but still challenges in detecting other types of code clone. More advanced and unified theoretic and technical guidelines are needed to improve code clone detection techniques. Therefore, in this paper, a literature review for code detection is presented especially from the perspective of source code representation. In summary, the contributions of this study are: (1) current code clone detection techniques are concluded and classified from the perspective of code representation; (2) the model validation and

* 收稿时间: 2018-08-21; 修改时间: 2018-10-07; 采用时间: 2018-11-15

performance measures in model evaluation are concluded; and (3) the key issues of code clone research are summarized from three aspects: scientific, practical, and technical difficulties. The possible solutions to the problems and the future development of the research are elaborated, focusing on data annotation, characterization methods, model construction, and engineering practice.

Key words: code clone, clone detection, code representation

1 引 言

代码克隆(code clone),也称为重复代码(duplicate code)或相似代码,指的是存在于代码库中两个及以上的相同或者相似的源代码片段.代码克隆产生的原因有很多,主要是开发者为了提高效率而使用的复用技术,包括复制粘贴已有的代码片段并进行修改、使用开发框架、复用设计模式等.大量的实证研究表明[1–10],代码克隆广泛存在于各个开源与闭源代码仓库之中,并且占据了相当比例,例如,有研究在 Linux 系统中检测到了 22.3%的代码克隆[5,6,11],Kamiya 等人发现在 JDK 中存在 29%的代码克隆[1],在某些软件系统中代码克隆甚至达到了 50%[3].广泛存在的代码克隆一定程度上帮助了软件系统的开发,能够产生正面的效益[7,12–15],比如可以利用克隆系统测试新增功能对原系统的影响[15],然而也有许多研究指出数量巨大的代码克隆会对软件系统造成负面的影响[4,16–19]:随着软件生命周期的进行,未得到良好克隆管理的软件系统会因为代码克隆造成代码库的不断膨胀,从而增加维护成本.软件缺陷也会因为代码克隆而在系统中被传播,降低了软件系统的可靠性.所以,如果不及时控制代码克隆的增长,对系统的管理、维护、修复等行为都会耗费额外的人力[20],导致软件维护成本的提高.

鉴于此,研究者们致力于研究并解决代码克隆衍生问题[2,4].其中,如何更快速、更准确、更便捷地发现代码克隆,是代码克隆研究的核心问题,而利用人工的检测代码克隆效率低,成本高,准确率也无法保证[20].围绕这个问题,软件工程研究者们提出代码克隆检测技术,目的在于自动化定位软件系统中的代码克隆,能够节省成本,减少出错风险[21].以此帮助开发人员和管理者及时发现代码克隆,并采取修复措施[22],有助于更好地保证软件质量[23,24].代码克隆检测在剽窃检测[25,26]、版权侵犯调查[27]、代码重构[5,28–30],以及管理代码质量[23]、寻找缺陷[11,27,31]、发现复用模式[32]等方面发挥了重要作用.除此之外,代码克隆检测也引起了工业界的注意[33,34].例如,微软将克隆检测工具应用在了自己团队的开发过程中,并取得了良好的效果[20],展示了代码克隆检测技术的实用性.

根据代码克隆相似程度的不同,Bellon 等人将代码克隆分为 4 种类型[33],即完全相同的代码(类型 1)、重命名的代码(类型 2)、几乎相同的代码(类型 3)和语义相似的代码(类型 4),从类型 1 到类型 4,代码克隆的相似程度逐渐降低,检测的难度也逐渐增加.学界产出了一批优秀的检测方法[1,34–38],但依然难以对高难度的代码克隆进行有效检测[23],因此需要我们从新的角度提升对代码克隆检测的认知.

为了认识到代码克隆检测的本质问题,我们对现有方法进行了梳理,不同的代码克隆检测方法之间的区别很大,但是它们都遵循着最基本的思路:即首先进行一定的预处理,然后进行信息抽取并对源代码进行表征,最后设计相似度算法进行对比,从而检测到代码克隆.

在这些步骤中,如何对源代码进行合适的表征是代码克隆检测的根本问题.宏观来说,对源代码的表征方式决定了对源代码信息抽取程度的上限,进而影响了所能检测的代码克隆的程度;微观来说,它决定了代码克隆检测技术的预处理方法、模型设计、部署方式、运行效率,并会影响最终结果.现有的代码克隆检测方法对源代码有不同的表征方式,我们根据对源代码信息的不同利用程度,将其分为基于文本、词汇、语法、语义 4 个层次:基于文本的表征方式仅利用源代码作为文本编码的信息,基于词汇的表征方式利用源代码符号序列化信息,基于语法的表征方式融入代码语法知识信息,基于语义的表征方式除语法之外,还利用了源代码控制流和数据流等信息.不同层次表征方式有各自的特征与优缺点,因此也需要科学系统的方法对其进行验证和评估.

目前尚无从源代码表征方式角度对该领域的研究进展进行梳理和归纳的研究工作.鉴于此,本文拟针对当前代码克隆检测研究进展,从源代码表征方式角度进行梳理,归纳和总结现有的代码克隆检测技术和评估方法,总结当前该领域存在的关键问题,并讨论解决思路与研究发展趋势.

本文第 2 节介绍代码克隆的相关概念以及克隆类型的定义,并介绍代码克隆检测的一般框架.第 3 节介绍

代码克隆中表征方式的研究进展,从 4 个层次阐述并归类现有的工作.第 4 节介绍代码克隆检测评估方法,主要从实验验证方法和评估数据集进行总结和归纳.第 5 节提出代码克隆分析研究的关键问题,包括科学问题、技术问题和工程实现 3 个方面.最后第 6 节对本文进行总结.

2 代码克隆相关概念与概况

2.1 代码克隆基本概念

代码克隆也称作重复代码或者相似代码,为了方便论述,本文将统一使用以下基本概念定义.

(1) 代码片段:代码片段(clone fragement,简称 CF)是源代码的一部分,它通常包含若干有含义的语句,代码片段可以是类、函数、有开始结束标识的代码块或者一个声明的序列.

(2) 代码克隆/克隆对:如果代码片段 1(CF1)和代码片段 2(CF2)的文本、语法或者语义相似,则其中一个就被称作另一个的代码克隆.如果这两者有一定的联系以至于两者是可以类比的,则称这两个代码片段为克隆对(CF1,CF2).

(3) 克隆类:一个克隆类是一系列的克隆对,这些克隆对之间有一定的关联,且关联是对称且对等的.

2.2 代码克隆类型定义

早期的研究并没有对代码克隆的类型进行明确的分类,直到 Bellon 等人提出了 4 种代码克隆类型的定义[33],这样的定义在后续的研究中获得了广泛认可.

(1) 类型 1(完全相同的代码):除了空格、注释之外,两个代码片段完全相同的代码对.

(2) 类型 2(重命名/参数化的代码):除了变量名、类型名、函数名之外都相同的代码对.

(3) 类型 3(几乎相同的代码):有若干语句的增删,或使用了不同的标识符、文字、类型、空格、布局和注释,但是依然相似的代码对.

(4) 类型 4(语义相似的代码):相同功能的异构代码,在文本或者语法上不相似,但在语义上有相似性.

为了直观地加以说明,图 1 给出了 4 种类型代码克隆的例子.

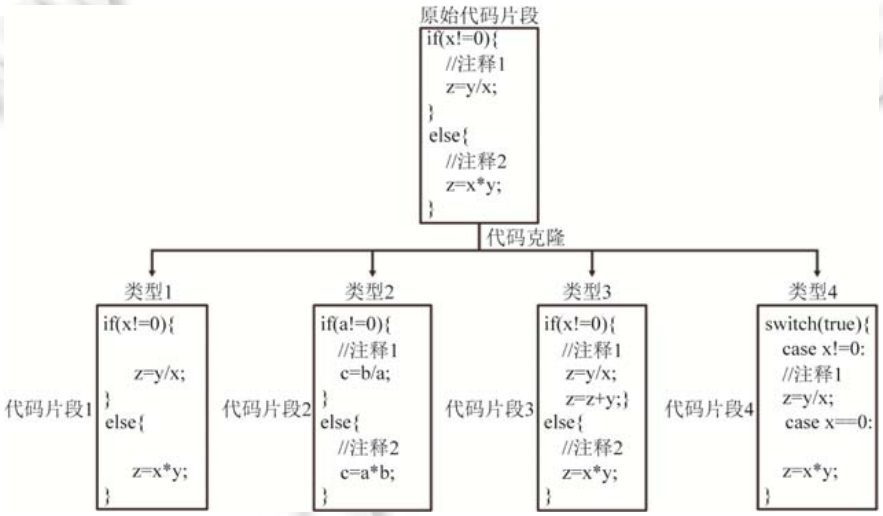

Fig.1 Original code fragment and four clone types

图 1 原始代码和 4 种类型的代码克隆

相较于原始代码片段,代码片段 1 删除了注释,代码片段 2 对原始代码变量名进行了替换,代码片段 3 增加了一行计算语句,代码片段 4 使用 Switch 语法代替了 If-Else.这种代码克隆的分类方式反映了代码片段之间相

似程度的不同[33],从类型 1 到类型 4,代码克隆的相似程度逐渐降低,检测的难度也逐渐增加.然而,这种分类并没有对代码克隆进行详尽、完备的定义,留下了一定的空间.其中,只有类型 1 和类型 2 代码克隆有严格的定义;对于类型 3 代码克隆,语句增删数量比例没有严格定义;对于类型 4 代码克隆,相似程度只有宽松的抽象层次的定义.因此,当需要具体描述时,不同论文用各自的方式进一步给出了严格定义[2,4,39].

2.3 代码克隆研究概况

代码克隆的研究至今已有超过 20 年的历史[23],代码克隆研究可以简单总结为两个问题:一是如何找到代码克隆,二是找到代码克隆之后如何分析与利用.第 1 个问题即研究代码克隆检测问题,第 2 个问题即研究代码克隆的定性分析[3,6,7,9,16–18,23]与代码克隆的管理[20,23,29,40]问题.代码克隆的分析与管理离不开代码克隆检测研究的支撑[41],因此本文聚焦于代码克隆检测的研究.

早期的代码克隆检测技术只考虑使用文本对源代码进行表征,这样只能检测到类型 1 和类型 2 的代码克隆[1].然而这样的技术难以满足实际需求.例如,许多难以被检测的类型 3 代码克隆,是经过复制粘贴然后进行一定增删形成的,而这种修改方式是开发人员经常使用的代码复用技术.为了解决这样的问题,研究者们使用了比文本更高层次的对源代码的表征方式,在词汇、语法、语义层面,提出了能够检测类型 3 甚至类型 4 代码克隆的检测技术[38,42].

代码克隆问题也引起了工业界的关注,一些克隆检测工具被应用到生产开发环节中以控制代码质量[20].研究表明,在实践中项目不同时期对代码克隆检测的应用不尽相同.比如,早期项目中克隆检测可以帮助开发者及时进行克隆重构[40],维护阶段项目则利用克隆检测工具以较低成本进行克隆缺陷管理[20],这种经验可以指导开发者与管理者更好地关注与管理代码库中的代码克隆.

2.4 代码克隆检测一般框架

检测代码克隆是解决代码克隆问题的关键一环,不同的代码克隆检测方法之间的区别很大,但是它们都遵循着基本的思路,如图 2 所示,为了展示一个克隆检测算法的基本逻辑,本文给出检测代码克隆的一般框架.

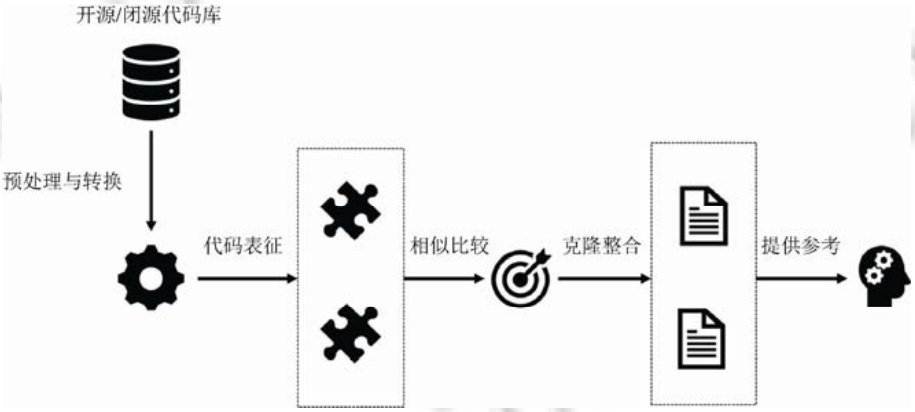

Fig.2 Research framework of code clone detection

图 2 代码克隆检测一般框架

(1) 源码预处理与转换.代码预处理的步骤将移除无意义的代码片段,将源代码转化成所需检测的单元,并决定用于比较的单元.3 个主要的预处理步骤如下:首先根据需要,将空格、不需要注释和多余的语句等删除.然后确认源代码单元,在移除不需要的信息之后,决定哪些是需要比较的单元,一般来说,有文件、类、方法、标注开始结束的代码块或者语句级别的比较.最后确认比较单元,比较单元一般是根据源代码表征方式决定的.比如说,基于符号的算法需要将源代码切分为符号,基于树的方法则需要比较子树等.

(2) 代码表征.这一步骤可以将源代码表征为文本,或者进一步利用符号进行表征,更深入的表征方式还包括将源代码转换成抽象语法树等.

(3) 代码相似度比较.在这一步骤中,每一个代码片段都会与其他代码片段进行对比来找到代码的克隆.比对的结果将以克隆对列表的方式呈现.其中,相似比较的算法很大程度上由源代码表征方式决定.

(4) 代码克隆结果过滤.并非所有技术都需要这一步骤,它的目的是过滤掉检测错误的代码克隆,过滤方法包括人工检测或者使用启发式算法等.

(5) 代码克隆结果整合.这一个步骤主要是将前几个步骤获得的代码克隆和原始的源代码关联起来并以适当的方式呈现以提供帮助,比如将克隆对聚合成为克隆类,因为作为组的克隆结果比作为克隆对更能帮助开发人员或者管理人员.

2.5 代码克隆检测评价指标

在代码克隆一般框架之下,不同代码克隆检测方法需要根据不同规模、结构、编程语言的软件系统进行不同的定制与改进,为了验证对比模型效果,以下评价指标可以作为参考以及指导方法继续改进的方向.

精确度(precision):精确度是指克隆检测算法所检测到的候选代码与克隆真实代码克隆的比例:

$$Precision = \frac{TP}{TP + FN} \tag{1}$$

召回率(recall):召回率是指所有被检测到的代码克隆数量占总体代码克隆数量的比例:

$$Recall = \frac{TP}{TP + FP} \tag{2}$$

可移植性:可移植性很大程度上与语言独立性有关,可移植性高的工具对不同编程语言有良好的支持.比如,一些基于文本或符号的方法因为可以独立于编程语言,因此有良好的可移植性.基于词法的方法需要词法解析,可移植性一般.而基于指标的方法和基于程序依赖图的方法通常需要特定语言解释器,所以其可移植性最差.

可拓展性:可拓展性主要是指算法可以在大型系统中进行应用,并且开销合理(运行时间、内存等).

克隆类型:算法检测 4 种类型的代码克隆中是评价一个代码克隆工具的重要指标,将决定代码克隆检测工具的应用场景.

克隆粒度:不同算法有不同的检测粒度,比如逐行对比、符号对比、子树对比、子图对比.有研究比较了不同粒度下的克隆检测算法[43],指出克隆比较粒度常由应用场景来决定.

结果呈现:在检测到的结果中,相较于克隆对,克隆类能够提供更多软件系统中代码克隆信息[4],比如,NICAD 以克隆类进行结果的呈现,获得了良好结果[38].

3 克隆检测中代码表征方式

在代码克隆检测中,源代码表征方式决定了对源代码信息抽取程度的上限,会决定检测方法的预处理方式、模型设计、部署方式、运行效率,并影响最终结果.比如,将源代码表征为文本,其预处理过程主要为去除噪声(如空格、注释等),其比较算法可以利用文本相似的一系列方法,能够检测到文本相似的代码克隆;而如果表征为抽象语法树,则其预处理过程需要解释器的参与,相似比较算法更多地考虑了结构相似等,能够检测到语法层面相似的代码克隆.因此,源代码表征方式是代码克隆检测的关键步骤.

然而,尚未有研究工作以代码表征视角研究克隆检测问题,相关工作从使用的方法[4]或用途[23]归纳代码克隆检测.比如,Rattan 等人从软件工程领域的顶级会议和期刊中,收集了 213 篇代码克隆检测相关文章,以代码克隆检测相关技术为切入点,综述了代码克隆领域的发展情况,并介绍了代码克隆管理等相关领域[10].然而他们的工作总结了早期的代码检测技术,近期诸如深度学习等进展并没有被讨论到,且没有对源代码表征方式深入讨论.与他们的工作不同,本文深入调研了代码克隆检测中源代码表征方式,在前人工作的基础上[4,8,10,23],我们从代码表征方式角度重新看待问题,根据对源代码信息的不同利用程度,将表征方式分为文本、词汇、语法、语义 4 个层次,并将现有的工作进行了归纳总结,从而能够帮助理解代码克隆检测问题的本质,为未来的研究提供新的视野.本节将对这 4 个层次的表征方式进行详细的阐释.

3.1 基于文本表征方式的研究

基于文本表征方式的检测技术将源代码当作文本编码,利用一系列文本相似度算法来检测代码克隆.文本相似的源代码属于相似程度较高的类型 1 到类型 2 代码克隆,表 1 展示了基于文本表征方式的克隆检测技术及其技术特征[4].这种根据源代码的字符串来进行检测的方法理论上只能检测类型 1 的代码克隆(直接进行比较)和一些基本的类型 2 代码克隆(进行一些简单的预处理,如隐去变量名等).源代码有其特殊的含义,仅当作文本处理会丢失大量的信息,所以基于文本的表征方式利用源代码信息的程度最低,被划分在了第 1 个层次.

Table 1 Characteristics of text-based clone detection approaches

表 1 基于文本的代码克隆检测的技术特征

检测方法	预处理方法	源代码表征方式	相似对比算法	可移植性	克隆类型	结果呈现
Dup[27]	去掉空格和注释	参数化字符串匹配	基于后缀树的模式匹配	需要词法分析器	类型 1 类型 2	克隆对
Duploc[44]	去掉注释和空格	文本中的代码行	动态模式匹配	需要词法分析器	类型 1 类型 2	克隆对
NICAD[38]	去掉空格和注释, 抽取语句	方法函数文本 (划分过的序列)	最长相同序列(LCS)	需要解释器	类型 1 类型 2 类型 3	克隆对
SDD[45]	无改变	用文本建立倒排索引	*N* 近邻	不需要词法 分析器/解释器	类型 1 类型 2 类型 3	可视化代码 克隆对

知名的基于文本的检测方法有 Roy 和 James 提出的 NICAD 方法[38]以及 Lee 等人提出的 SDD 方法[45].NICAD 使用了两种技术检测类型 1 到类型 3 的代码克隆,这两种技术分别是基于文本的和基于树的,两者可以独立使用来互相补充不足.NICAD 分为 3 个步骤:首先使用了一个解析器(Turing eXtender language,简称 TXL)来将代码段分割成行,然后利用一些转换规则来进行转换,接着将剩下的潜在的克隆对进行重命名,最后利用动态的模式匹配来找到最长的相同的子序列.NICAD 之所以能够检测到类型 3 的克隆,是因为使用了唯一字符串比例(percentage of unique strings,简称 PUS)来控制检测松弛度,如果其值为 0%,则两段代码为类型 1 完全相同,如果其在 0%到某个阈值之间,则检测出的结果为类型 2 或者类型 3.NICAD 能在代码段和函数粒度进行检测代码克隆,它有着较高的精确度和召回率[46].NICAD 的解析器让其充分利用了抽象语法树的技术的好处,但其利用文本行的比较而非子树的比较,所以还是一个基于文本的检测方法,拥有较低的空间复杂度和计算复杂度,能够胜任大型系统的扫描.

SDD 是一个 Eclipse 插件,它能够在大型软件系统中高效地检测代码克隆,对源代码的利用是基于文本的表征方式.SDD 使用了倒排索引和 *n* 近邻算法,在算法运作时,它建立的倒排索引可以有效减少时间开销,从而能够在大型软件系统中高效运作,帮助开发者在使用 IDE 时检测代码克隆.

Dup 方法使用了后缀树算法,检测源代码文本或者哈希后的相似子序列[27].具体的方法是匹配超过阈值长度的最大代码段,其检测步骤分为精确匹配和参数化匹配:精确匹配将完全相同的源代码进行匹配检测,参数化匹配将除了变量名不同的代码预处理后进行匹配,比如,将所有变量替换成同一字符串.这种文本方法忽略了注释和空格,基于文本行进行检测,能够检测到类型 1 和类型 2 代码克隆.

Duploc 方法使用字符串操作对代码行进行了简单的预处理,然后使用基本的字符串方法来检测最长相同子序列[44].这样,基于纯文本检测方式让 Duploc 无需解析器或者词法分析器,拥有较好的可移植性,独立于不同编程语言,然而也限制了检测的代码克隆类型.Duploc 一般能够检测类型 1 和类型 2 代码克隆.

利用文本进行源代码表征的检测技术几乎不会检测到文本差异大的代码克隆,因此精确度很高,很少会出现假阳率(错误地将非代码克隆识别成克隆).因为只从文本方面考虑,这种表征方式独立于编程语言,部署成本较低,计算开销较小,有很强的可拓展性和易用性,因此空间复杂度和时间复杂度比较低.但是也有它的缺点和局限性,主要表现在:源代码虽然以文本方式编码,但是还保留有语法等信息,在检测中如果只使用文本方式进行源代码表征,会损失大量信息.

3.2 基于词汇表征方式的研究

基于词汇的检测技术也叫作基于符号(token)的检测技术,这种技术利用了解析器将源代码分成符号序列,然后这些符号序列会被组织成符号的语句,最后将这些符号组成的语句进行比较.基于词汇的方法利用了更符合编译原理的符号序列,对源代码信息有了更进一步的利用,所以被划分到了比单纯的基于文本更深入的层次.

表 2 展示了基于词汇的克隆检测技术及其关键步骤的特征.知名的基于词汇的克隆检测技术有 Kamiya 等人提出的 CCFinder[1]和针对大规模情景的基于分布式的 D-CCFinder[47],以及 Li 等人提出的 CP-Miner[37]等.

Table 2 Characteristics of lexis-based clone detection approaches

表 2 基于词汇的代码克隆检测的技术特征

检测方法	预处理方法	源代码表征方式	相似对比算法	可移植性	克隆类型	结果呈现
CCFinder[1]	去掉空格、注释并执行参数转换	正则化序列以及参数化符号	基于后缀树的符号匹配	需要词法分析器和转换规则	类型 1 类型 2	可视化的克隆对和克隆类
CP-Miner[37]	将相似语句标识符映射成数字	基本代码块	频繁子项挖掘技术	需要解析器	类型 1 类型 2	克隆对
Boreas[48]	过滤无用的字符并抽取符号	根据其他特征的字符串匹配	余弦相似度函数	需要解析器	类型 1 类型 2 类型 3	克隆类
FRISC[49]	去掉空格、注释,将源代码映射成序列,并替代参数	符号的哈希序列	后缀数组	需要词法分析器	类型 1 类型 2 类型 3	克隆对和克隆类
CDSW[50]	去掉空格、注释,将源代码映射成序列,并替代参数	每个语句的哈希值	Smith-Waterman Alignment 算法	需要词法分析器	类型 1 类型 2 类型 3	克隆对
XIAO[34]	去掉空格、注释,对源代码进行参数替换	每个语句的哈希值	哈希比较算法	需要解析器	类型 1 类型 2 类型 3	克隆对
CCAlingner[51]	去掉空格注释	每个窗口的哈希值	哈希比较算法	不需要词法分析器/解释器	类型 1 类型 2 类型 3	克隆对
CCLearner[52]	词嵌入(word embedding)	特征向量	利用学习到的特征比较	需要解析器	类型 1 类型 2 类型 3 类型 4	克隆对

CCFinder 的检测步骤分为 4 个部分:首先使用词法分析器解析成为符号序列并去除所有空格和注释,然后符号序列经过一定的转化规则进行变形,并进行参数替换将标识符变成特殊的符号,接着匹配检测阶段使用后缀树匹配算法检测出代码克隆对和克隆类,最后再将代码克隆的行数映射到源代码文件中.CCFinder 也使用了一些指标度量来检测潜在的代码克隆:如源代码长度、克隆类的群体大小、代码克隆的覆盖率.在算法复杂度问题上,它优化了源代码程序的长度以减少计算复杂度.CCFinder 虽然能够获得较高的召回率,但是准确率却不尽如人意[6].一次也只能接受一种编程语言来检测.

CP-Miner 致力于检测大型系统中的代码克隆以及与代码克隆有关的软件缺陷,它使用频繁项挖掘[53]技术来达到这一目的.CPMiner 首先使用解析器来将源代码变成一个序列的集合,产生一个数据库来表征每一段源代码,然后使用改进的 CloSpan 算法[53]来帮助达到频繁项挖掘的间隔限制,最后的结果将会包含正确和错误的复制粘贴代码段,最后会检查相邻的代码段来进行过滤,得到最终的代码克隆检测结果.相比于 CCFinder,CP-Miner 能够做到更高的准确率,同时召回率也不会下降太多,但其问题在于很难确定频繁项挖掘中子项的长度:如果太长,则会严重影响效率,如果太短,则会丢失非常多的信息,最终影响效果.

近期基于符号的方法还有 Wang 等人提出的 CCAligner 方法[51],在将源代码进行预处理之后,他们使用了一个滑动的窗口,将窗口中的源代码进行了哈希,最后进行多重对比,从而在符合阈值的范围内检测到代码克隆.CCAlingner 对于有较大间隔的类型 3 取得了非常不错的效果,其中,大间隔代码克隆是指代码片段 1(CF1)行数与代码片段(CF2)行数比值小于或等于 0.7 的代码克隆对[51].

Li 等人的 CCLearner 工作采用了深度学习方法来学习符号层面的代码信息.他们使用 BigCloneBench[54]作为训练样本,抽取了其中方法级别的符号序列,用一个全连接神经网络数据,对数集中标注的克隆对和非克隆对进行训练,学习这些特征并用于检测代码克隆.需要注意的是,为了提高信息利用程度,他们也抽取了部分抽象语法树的信息进行训练,但并不是模型的重点.

除此之外,Boreas 方法使用了余弦相似度和一种按比例的相似度的方法来检测代码克隆;Murakami 等人提出的 FRISC[49]将重复指令转换成了一种特殊的格式,然后使用后缀数组的算法来检测克隆,但其检测出来的代码克隆有更高的错误率;CDSW 使用了 Smith-Waterman 算法[55]来检测代码克隆,其准确度受到算法中的参数的影响,且有较大的波动.XIAO 使用了与 CP-Miner 类似的预处理方法[34],但是根据实际情况加入了更多的指标度量来帮助检测代码克隆,它将语句序列化并使用哈希后的值来进行比较,检测软件系统中的代码克隆.

相对于基于文本的代码表征方式来说,基于词汇的代码表征方式能够匹配到许多代码特有的信息.但从本质上说,基于词汇的代码表征方式与文本一样都是序列化的表征方法,只是对源代码的利用程度有所提升,它有如下几个缺点.

(1) 基于词汇的代码表征方式没有考虑到代码行的顺序,如果代码克隆的顺序被改变了,代码克隆将不会被检测到.

(2) 基于词汇的代码表征方式对于增加删改了符号的代码语句比较敏感,容易漏检特定细微差别的代码克隆.

(3) 基于词汇的代码表征方式依然没有充分利用源代码的信息,比如它会忽略掉源代码中的结构信息.

3.3 基于语法表征方式的研究

基于语法的代码表征方式更多地考虑到了源代码的语法规则,从而能够对其中蕴含的信息加以利用.基于语法的代码表征方式主要有基于语法树的方法和基于指标的方法.

抽象语法树(AST)是源代码特有的一种表现形式,它是编译源代码的一个中间结果,以树的形式包含了源代码中的语法信息.在代码克隆检测中,基于树的方法会将源代码解析成抽象语法树作进一步的处理并用于检测.为了更加直观,图 3 展示了一段辗转相除法代码转换而成的抽象语法树,要检测它的代码克隆,可以利用树匹配等算法进行子树的比较,其中形状完全相同的树可以认为是类型 1 的代码克隆,而如果忽略节点的标识符或者进行一定程度的剪枝(例如,忽略某层节点下面的子节点),对剩余的结构进行比较,经过修改的代码克隆也能够被检测到.

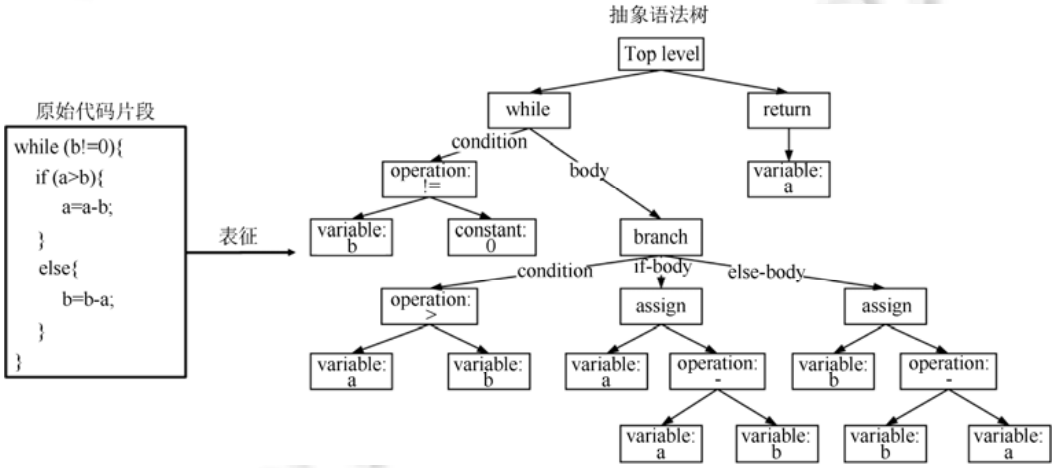

Fig.3 Represent source code with abstract syntax tree

图 3 使用抽象语法树表征源代码

根据语法特性,一段源代码拥有很多指标度量(比如代码行数、循环数量、变量数量等),这些指标度量一定程度上能够表征这段源代码.因此,在代码克隆检测中,基于指标的方法抽取目标代码段的指标度量表征这段源代码,然后进行比较,从而检测出代码克隆.

表 3 展示了基于语法的克隆检测技术及其技术特征.知名的基于树的检测方法有 Baxter 等人提出的 CloneDR[35]和 Jiang 等人提出的 Deckard[42]等.基于指标来进行代码克隆检测的方法包括 Mayrand 等人[56]的工作和 Kontogiannis 等人[32]的工作,抽取了预先定义的代码特征,然后进行比较并获得代码克隆.

Table 3 Characteristics of syntax-based clone detection approaches

表 3 基于语法的代码克隆检测的技术特征

检测方法/作者	预处理方法	源代码表征方式	相似对比算法	可移植性	克隆类型	结果呈现
CloneDR[35]	解析成抽象语法树	抽象语法树	树匹配技术	需要解析器	类型 1 类型 2	克隆对
Wahler[57]	解析成抽象语法树	抽象语法树	频繁项集	需要解析器	类型 1 类型 2	AST 的相似节点
Deckard[42]	解析成解析树然后转化为向量	向量	局部敏感哈希匹配算法(LSH)	需要解析器	类型 1 类型 2 类型 3	文本格式
CDLH[39]	转化成抽象语法树,并进行哈希	抽象语法树	利用学习到的特征比较	解析器(parser)	类型 1 类型 2 类型 3 类型 4	克隆对
White 等人[58]	去掉空格注释等	抽象语法树	利用学习到的特征比较	需要解析器	类型 1 类型 2 类型 3 类型 4	克隆对
Hotta 等人[43]	使用 JDT 解析提取代码块	哈希化代码块	基于哈希值将代码块分组	需要解析器	类型 1 类型 2	克隆对和克隆类
Mayrand 等人[56]	解析成抽象语法树	指标度量	21 个函数指标度量	需要 Dartix 工具	类型 1 类型 2 类型 3	克隆对和克隆类
Kontogiannis 等人[59]	将指标转化为特征	特征向量	利用编辑距离比较指标和动态程序	需要解析器和其他工具	类型 1 类型 2 类型 3	克隆对
Kodhai 等人[60]	去除空格和注释,映射和预处理	指标度量	字符匹配和文本比较	需要解析器和其他工具	类型 1 类型 2	克隆对和克隆类
Abdul-El-Hafiz 等人[61]	预处理和提取指标	指标度量	数据挖掘聚类和几何聚类算法	独立于编程语言	类型 1 类型 2	克隆类
Raheja 等人[62]	计算 Java 的指标度量	指标度量	3 个步骤的比较算法	需要编译器	类型 1 类型 2 类型 3	呈现在 EXCEL 表格中

CloneDR 使用了基于语法树的方法,能检测到完全和几乎相同的代码克隆,并且能够利用抽象语法树来进行代码重构.在 CloneDR 方法中,在源代码被解析成抽象语法树之后,它利用 3 种主要的算法来进行代码克隆检测:第 1 种算法能够检测整棵树中的子树的克隆,使用哈希方法将子树分割然后比较这些子树;第 2 种算法检测子树中变长序列的克隆;第 3 种算法结合了其他检测方法找到更多的代码克隆.CloneDR 虽然使用了抽象语法树,能够检测到基于文本或者词汇无法检测到的代码克隆,但却不能检测到语义相似的代码克隆.

Deckard 使用了基于树的方法和欧式距离相似度,它的主要步骤如下:首先使用一个语法解析器将源代码解析成解析树,然后解析树被用来生成一个向量集用于承载解析树中的结构信息,接着,这些向量被局部敏感哈希算法进行了聚类,从而可以寻找一个向量的近邻,最终一些后续操作将结果转化为检测到的代码克隆.Decard 能够检测到改变了顺序的代码克隆和不相邻的代码克隆.抛开解析器的不同仅比较树的相似度,Deckard 能够做到一定程度的语言独立.但是 Decard 的比较速度比较慢,其解析过程也会消耗非常多的时间.

Mayrand 等人的工作计算了源代码函数的名称、层次、表达式和控制流抽取出来的指标度量,如果两个指

标是相似的,这两个函数则被认为是代码克隆.他们的工作着眼于定位相似函数而非相似代码段,但在真实情况下,相似的代码片段要比单纯的相似函数出现的频率高很多.

Kontogiannis 等人的工作用了两个步骤来检测代码克隆相似度:第 1 种方法抽取了源代码中的指标度量,然后使用简单的数值比较.第 2 种方法使用了动态程序来计算,并使用了最小的编辑距离来测量两个代码段的相似度.这种方法只能给出初步的结果,用户还需要自行作进一步的人工检查来确定代码克隆的正确性.

Wei 等人的 CDLH[39]使用了深度学习方法来学习代码克隆的特征.CDLH 将代码克隆检测问题转化为学习源代码的哈希特征的有监督学习问题,主要分为 3 步:首先将源代码用抽象语法树表征之后用一定的编码规则进行编码,然后将编码后的数据输入一个修改过的卷积神经网络当中进行训练,最后用训练到的特征进行代码克隆检测.CDLH 可以检测到类型 1 到类型 4 的代码克隆,而且做到了非常高的精确度和召回率.

除此之外,基于树的方法还有 Wahler 等人的工作[57],将抽象语法树用 XML 形式进行了转换然后进行比较与检测,但是只能检测到类型 1 和类型 2 的代码克隆.基于度量的方法还有 Kodhai[60]等人的工作,但是只能检测类型 1 和类型 2 的代码克隆;Abdul-El-Hafiz 等人的工作[61]使用了数据挖掘中几何聚类的算法,但是没有具体的精确度和召回率,也不能检测类型 4 的代码克隆;Raheja 等人的工作[62]利用了字节码进行检测,进而映射到源代码中,它可以检测到一部分语义相似的源代码,然而需要编译器进行编译,部署成本比较高,且只对一种语言有效.White 等人的工作[58]人工采样了 398 个文件级别和 480 个方法级别的克隆对作为训练样本,将源代码转换为其定义的树的结构,采用卷积神经网络进行训练,最后用学习到的特征检测代码克隆.

基于语法的代码表征方式能够考虑到源代码的结构特性,从而对信息的利用更加充分,相比于文本和词汇是更加深入的一个层次.它对源代码的顺序变换更加不敏感,对于细微修改的代码克隆也可以做到检测.但是信息利用程度的增加也带来了一些弊端,基于语法的方法有如下缺点.

(1) 基于树的方法不能识别出标识符和文本值的不同.

(2) 基于树的方法由于需要遍历树,计算开销比较大.

(3) 基于指标的方法在获取特定指标时需要解释器或者程序依赖图来获得.

(4) 基于指标的方法难以保证较高的精确度,即使是两个代码片段的指标是相同的,这两段代码也有可能是不相似的.

3.4 基于语义表征方式的研究

基于语义的代码表征方式,不仅考虑了源代码的语法,还要利用其语义信息,其中,语义信息是指能够反映一段代码功能的信息,比如代码的控制流和数据流.在代码克隆检测问题下,基于语义表征源代码的技术可以分为基于图的技术和混合技术两类.

基于图的检测技术使用由程序生成的数据流图和控制流图来组成程序依赖图(PDG),数据流代表了源代码中数据的走向,控制流代表了源代码中逻辑的走向,它们组成的程序依赖图不仅是基于语法的,也表征了一段源代码的语义信息.为了直观说明,图 4 展示了使用包含控制流和数据流的程序依赖图表征源代码的例子,其中,每一个节点代表一个语句或者一个断言,每一条边则代表数据流(实线)或者控制流(虚线).将源代码用图的方式表征,只比较图的相似度,能够将问题转化为检测相似图的问题,最终检测到代码克隆.

基于混合技术的克隆检测不再是单一的技术,而是利用多种技术的混合来达到语义检测的目的.比如说,可以利用基于文本的方法结合基于指标度量的方法先后表征一段代码,从两种不同的维度刻画它的特征,从而能够得到其更精确的语义特征[63].

表 4 展现了基于语义的检测方法及其技术特征.知名的基于图的检测技术有 Komondoor 和 Horwitz 提出的利用同构程序依赖图的切片来检测克隆的方法[64]和 Krinke 等人提出的 Duplix[36].知名的混合技术则有 Hummel 等人提出的 ConQAT[65]方法,它利用了符号化代码和哈希等方法,结合了第 1 层次和第 2 层次的源代码语义表征来进行代码克隆的检测.

Komondoor 等人提出的方法使用了程序切片来寻找同构的程序依赖图的子图,切片的克隆检测算法分为 3 个步骤:首先将程序依赖图中的节点划分到同等的类当中,在每个类中,每两个节点都是相对应的,然后它会去

掉节点中包含的其他子节点,这样就不会有父节点与子节点被重复检测的问题,最后将克隆对进行聚合给出检测的结果.

Krink 提出的 Duplix 方法使用的是类似于抽象语法树和传统的程序依赖图,因此,它使用的程序依赖图的顶点和边代表了语句的组件,也包含了控制流的边.Duplix 的检测方法与 Komondoor 等人的工作类似,它们在检测最大相似程序依赖图上取得了较高的精确度和召回率.

Hummel 等人提出的 ConQAT 使用了一种混合方法来检测代码克隆.它分为 3 个步骤:首先将源代码进行预处理,然后将其变为符号化序列,所有的符号都被整理到一个声明中,然后在后续的工作中寻找相同的子字符串,最后使用了克隆的索引在所有的文件中寻找类似的代码克隆.在 ConQAT 中,序列化的方法使用了 MD5 的哈希算法[48,66],相同哈希值的两个序列被认为是相同的代码克隆对.

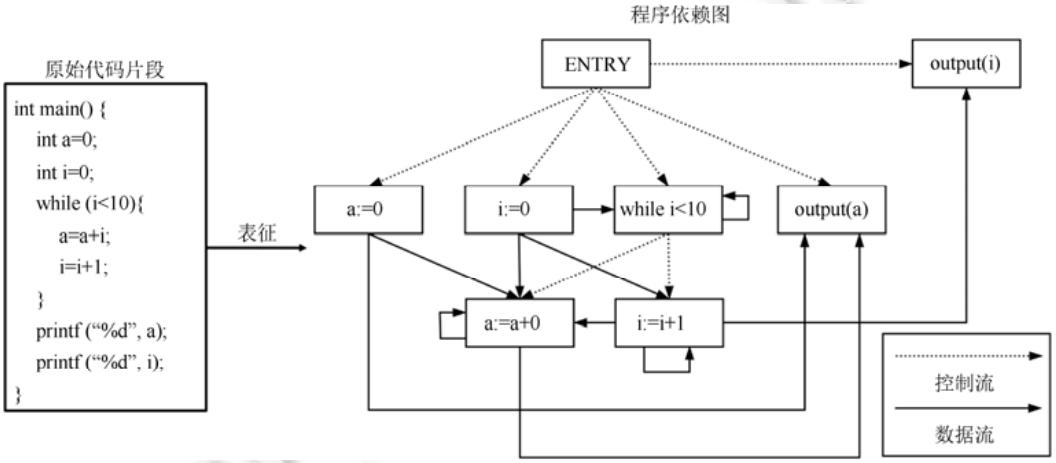

Fig.4 Represent source code with program dependency graph

图 4 使用程序依赖图表征源代码

Table 4 Characteristics of semantic-based clone detection approaches

表 4 基于语义的代码克隆检测的技术特征

检测方法/作者	预处理方法	源代码表征方式	相似对比算法	可移植性	克隆类型	结果呈现
Duplix[36]	转换成程序依赖图	程序依赖图	*K*-length patch	需要转换工具	类型 1 类型 4	克隆类
GPLAG[67]	使用 CodeSurfer 生成程序依赖图	程序依赖图	同构程序依赖图 子图匹配方法	需要转换工具	类型 1 类型 2 类型 3	克隆对
Higo 等人[68]	转换成程序依赖图	程序依赖图	代码克隆检测模块	需要转换工具	类型 3	克隆对文件
Komondoor 等人[64]	使用 CodeSurfer 生成程序依赖图	程序依赖图	使用后序切片的同构程 序依赖图子图匹配方法	需要转换工具	类型 3 类型 4	克隆对和克隆类
ConQAT 等人[65]	将源代码分割成为 符号,移除注释替换 变量名,然后 正则化分组	符号	基于后缀树和基于 索引的检测算法	独立于编程语言	类型 1 类型 2	文本格式克隆
Agrawal 等人[63]	转化成符号	符号	行到行的文本比较	词法解析器(lexer)	类型 1 类型 2 类型 3	文本格式克隆
Funaro 等人[69]	解析成为抽象 语法树并序列化 抽象语法树	抽象语法树	文本比较	解释器(parser)	类型 1 类型 2 类型 3	克隆字符串
Saini 等人[70]	抽取 24 个指标度量, 进行筛选和分组	指标度量与 特征向量	指标度量、哈希比较和 深度学习判定, 3 个步骤的比较算法	需要转换工具	类型 1 类型 2 类型 3	文本格式克隆

基于图的方法还有 GPLAG[67],它也使用了同构的程序依赖图的子图的对比算法,能够在小数据集上有着不错的效率,但是随着数据集的变大,计算复杂度将会呈指数增长;Higo 等人的工作[68,71]使用了简化的方法,能够检测非相邻的代码克隆对,比一般的基于程序依赖图的方法更快.

近期使用了混合方法的工作有 Saini 等人提出的方法 Oreo[70].Oreo 是一个代码克隆检测的框架,在它提出的流程中,使用了基于指标度量的方法在预处理过程中进行了初步筛选与分组,然后进一步从函数中抽取语义,并使用了基于哈希的方法进一步筛选,能够对类型 1 和类型 2 的代码克隆进行较好的覆盖,在流程的最后加入了深度学习的方法,提升了对类型 3 代码克隆的检测能力.其中,输入数据以对(pair)的形式输入是代码克隆检测的一个技术特点,Oreo 在深度学习结构中使用的 Siamese 结构[72]很好地解决了代码克隆输入的对称性问题,即(CF1,CF2)与(CF2,CF1)在作为输入时是等效的,值得借鉴.

混合方法还有 Funaro 等人的工作[69],综合了基于文本的方法和符号的方法,能够做到对类型 3 克隆的检测;Agrawal[63]等人的工作综合了基于文本和指标度量的方法,但是只能检测 C 的代码克隆.

综上所述,基于语义的方法能够在深入的层次利用源代码的信息,从而检测代码克隆,这些方法充分利用了源代码不同于一般的自然语言文本的特性,包括了结构、顺序以及特殊的语法等信息,也能够获得一定程度的语义信息.但其还具有如下缺点.

(1) 基于图的技术需要一个程序依赖图的生成器.而对于不同的语言会产生技术隔阂.

(2) 基于图的图匹配检测技术,计算的开销非常大.

(3) 基于混合技术表征源代码语义步骤繁杂,难以部署.

4 克隆检测评估方法

不同层次表征方式有各自的特征与优缺点,因此也需要科学、系统的方法对其进行验证和评估.为了评估代码克隆对软件系统的影响,在早期研究中,主要报告检测到的代码克隆在软件系统中的比例[2,23,73–75].这样的实证研究能够为后续研究提供证据支撑,启发思维.在提出克隆检测方法的过程中,为了验证和对比模型效果,研究者需要对检测模型进行实验验证.其中,评价指标中重要的精确度和召回率需要已知克隆对这样的先验知识,然而,面临未知软件系统却难以预先获得代码克隆准确数量.评估代码克隆检测的关键是收集客观、有效的先验数据集,因此,研究者们使用数据标注方法和数据生成方法获取代码克隆先验知识以评估模型.

4.1 标注数据收集方法

一些研究者们在提出方法时会自己搜集若干代码库[30,33,76–78],利用上下文,修改历史和人工检查等标注代码克隆,并用于评估自己的检测模型,然而不同的系统有不同的规模和结构等,这样的缺陷是代码库规模不够大[54],难以形成统一的评价基础.为了解决这样的问题,Svajlenko 等人提出了 BigCloneBench[54].BigCloneBench 是一个 Java 代码集,它包含了从类型 1 到类型 4 的大量的人工标注的克隆对,包含 10 个功能,耗费了 8 个专家 216 个小时标注 600 万对代码克隆和 26 万对负样本.很多后续的工作基于这个数据集评估它们自己的方法[39,51,52],BigCloneBench 推进了代码克隆检测的研究.但是这样的数据集有两个缺陷:第一,它只有 10 个功能,这不符合真实的软件系统的情况;第二,这个数据集的构建方法是基于启发式的搜索加上人工标注,而启发式搜索在某种程度上限制了代码克隆的模式.

4.2 评估数据生成方法

在代码克隆检测中,召回率需要已知软件系统中的所有代码克隆,然而这难以在每一个新的软件系统中实现.Roy 等人借鉴软件测试中的变异测试[79],提出了一种变异插入的测试代码克隆的方法[46],其思想是在一段源码中插入一段人工代码,从而人为制造出不同类型的代码克隆对,以此作为测试集进而评估代码克隆检测工具的效果.这样的数据生成方法能够有效获得召回率,比如,Wang 等人在评估 CCAligner 针对大间隔代码克隆的召回率时使用了这种方法[51].虽然这样操作部分解决了代码克隆评价指标的问题,但是人工构造的代码克隆对不是真实数据集,难以从科学和工程上证明有效性,因此只能作为一种辅助测试方式.

5 代码克隆检测关键问题与解决思路

虽然代码克隆检测的研究已经有超过 20 年的历史,获得了非常多的成果,但是现有的研究与工业界的期望还有一定的距离,这是作为研究者需要正视的问题.本节首先讨论代码克隆检测研究中的关键问题,然后讨论针对这些问题的解决思路.

5.1 关键问题

虽然代码克隆检测以往获得了大量研究者的关注,取得了较大进展,但仍然存在一些亟待解决的关键问题.本节将从科学问题、实用性、技术难点和工程实践 3 个方面阐述目前代码克隆研究中存在的问题.

5.1.1 代码克隆检测研究的科学问题

代码克隆检测研究的科学问题分为 3 个方面.

(1) 代码克隆产生的归因分析与检测技术的有机结合.即分析所检测到的代码克隆产生的具体原因,以及研究如何进一步提高代码克隆检测的准确性相关研究,从历史角度研究了代码演进过程,从克隆族谱的角度研究了代码克隆的形成过程[6],以及形成后的更替与传播[78],然而现有归因分析仅仅对代码克隆的影响进行了定性研究[16].进一步地,研究者需要针对代码克隆形成原因,设计相关克隆检测技术,提高克隆代码检测技术对克隆代码的预防能力和时效性.

(2) 代码克隆检测中数据标注的准确性.准确的有标注的代码克隆数据对于代码克隆检测的研究至关重要,是检测模型训练、评估及技术对比的基准.虽然,类型 1 和类型 2 代码克隆因为定义具体而易于准确标注,然而,类型 3 和类型 4 代码克隆却难以准确标注,特别是没有具体定义的类型 4,一般是指功能相似的异构代码(即结构不相同的代码)[4,33],如何对异构代码进行标注是代码克隆数据标注中的一大难点.现有的 BigCloneBench 对类型 4 代码克隆的标注采用人工审查代码的方法[54],而如果从功能和需求出发,结合代码功能或需求描述文档,将有助于提高代码克隆标注的准确性.例如,在针对文件级克隆进行标注时,可参考需求文档中的功能描述;在针对方法级或片段级克隆进行标注时,可参考代码注释中的功能描述.

(3) 代码克隆检测的可拓展性.如今在代码大数据[80]的环境下,软件系统的规模日益变大,这也给代码克隆检测带来了新的挑战.近期的研究表明,一些早期的代码克隆检测技术[38,42]已经难以胜任千万行级别甚至上亿行级别的代码检测任务[41,51],因此,未来的代码克隆检测技术研究不仅要考虑精确度和召回率等检测指标,还需提升对大规模代码检测的能力.此外,代码增加与迭代速度也日益变快,比如开源仓库 Github 中每天都会有成千上万的贡献者贡献代码,在这样的情况下,如果每次都进行全量的代码克隆检测将花费较高的代价[68,81,82].因此,在未来的研究中,代码克隆检测的可拓展性,即能否将检测范围拓展到大规模代码仓库中,能否在高频率增量代码的场景下达到开销与性能的平衡,是研究者需要考虑的问题.

5.1.2 克隆研究的实用性

现有代码克隆研究主要在于以克隆对或者克隆类的形式给出结果,不同的研究给出了不同的粒度属性.克隆对或者克隆类仅仅只有源代码的指向,没有更进一步的分析与展示;在研究工作中选择的粒度也仅仅根据现有数据的方便程度而非工程中的实用程度.针对软件开发或者维护过程中如何有效利用代码克隆检测的结果尚未有统一的观点,因此,需要展开深入的实证研究,结合开发过程,以调研开发者实际需要哪种类型的代码克隆结果,如何能够更加有效地利用代码克隆检测结果等问题.

5.1.3 技术难点

如何更准确地标注代码克隆对是代码克隆检测中的技术难点之一.一方面,虽然有变异插入的标注方法可以大量生成代码克隆对,但是这种方法难以保证在真实的代码库中奏效;另一方面,虽然有研究者花费人力标注了大量的数据集,但是这样的数据集有其本身的局限性,在这个基础上设计的模型,特别是机器学习模型,一旦用到一个新的软件系统中,难以表现出良好的泛化性能.因此,研究者们在针对新进项目获取标注数据时面临着实际应用上的挑战.

源代码的表征方法也是一个技术难点,从文本到词汇、语法、语义,对于源代码信息的利用程度会逐渐增

高,但是带来的副作用是对于语言的独立性逐渐下降,对于如解释器等的额外工具的需求也会逐渐增加,大大提高了算法开发与实际应用的门槛.所以考量多个维度,在适当的层次抽取源代码,考虑它们之间的易用性、关联性也是一大技术难点.

现有代码克隆检测技术在模型构建上存在着不同的策略:在源代码的表征方法方面,基于文本、词汇、语法、语义等方式有多种选择;预处理可以进行不同的抉择,而只做简单的消除或者复杂变形;模型中相似度的计算则受到前面阶段选择的局限;最后结果的呈现也有克隆对克隆类,甚至考虑重新设计可视化界面等多种选择.但是正因为选择太多,所以需要权衡各种策略的利弊,这也给模型的设计增添了困难.因此,针对场景,选择合适的模型是代码克隆检测设计的一大难点.最后,工业界对于类型 4 有着较大的需求,但是现有的工具难以满足这一需求,因此也是需要研究突破的方向.

5.2 解决思路

针对上一节所总结的关键问题,本节围绕数据标注、表征方法、模型构建和工程实践 4 个方面,阐述问题可能得以解决的思路和研究的未来发展趋势.

5.2.1 综合考虑多源软件制品,提出更准确的数据标注方法

现代大型软件项目都会使用多个系统对软件开发和维护中产生的数据进行存储和管理,例如代码版本控制系统(如 Git)、代码审查系统(如 Gerrit)等.这些不同系统中存储着不同的软件制品,代码克隆的产生是一个复杂过程,并与多种软件制品相关,包括源代码、需求报告、代码审查、代码静态扫描等,现在还缺乏这方面的研究.综合考虑多源异构的软件制品,有助于更立体地获得真实软件系统中标注的代码克隆数据.同时也应该注意,从多个源头的软件制品获取代码克隆也会引入噪音,比如从需求映射到代码中的噪音问题[83]、代码静态扫描中的误报问题[84]、代码注释与代码的不匹配问题等[85],需要谨慎甄别,以保证标注数据的准确性不受干扰.

5.2.2 综合考虑实际需求,选择或提出更加合适的代码表征方法

在自然语言处理领域,表征学习是一个被关注的热点,同时在软件工程领域,如何将源代码进行适当的表征也是大量工作的基础,从而引起了研究者们的关注[86]:将源代码进行适当的表征,就可以增强对代码克隆的检测能力.与此同时,也要结合已有的表征方法,对优缺点进行权衡,比如基于语义表征代码中基于图的表征方式,早期利用图表征源代码进行代码克隆检测的研究使用了传统的图匹配算法[36,64,68],算法的可移植性和可拓展性都比较差,但是如果考虑图嵌入[87]技术,用嵌入向量的相似比较完成代码克隆检测,既利用了以图为基础的代码中的语义信息,又能够兼顾算法的可用性.虽然基于图的表征方式中图的定义与构建都需要许多研究工作来探索[67,86],但依然是一个可以尝试的方向.因此,使用针对源代码的方法,提取多维特征来增强源代码的表征能力,将是代码克隆研究的一个发展方向.

5.2.3 研究更先进的建模技术,取得建模技术上的突破

近年来,深度学习成为机器学习的热点研究领域,被大量应用到如图像处理、语音识别等研究中.相比传统的机器学习技术,研究者发现深度学习在这些领域的应用可以取得更好的性能[39,52].将目前机器学习领域大量研究与应用的技术,如递归神经网络应用到代码克隆检测技术中,将有可能进一步提升代码克隆研究的性能.在现有的工作中,深度学习已经展现了一定的能力,例如 CCLearner 以基于符号的方式学习代码克隆[52],CDLH 利用了源代码的抽象语法树结构[39],Oreo 提出的深度学习框架解决了代码克隆输入的对称问题,并能在初步检测的基础上进一步提高检测性能[70],这些方法都在对类型 3 和类型 4 的代码克隆检测中取得了一定成效,但是由于标注数据集本身限制的问题,例如被广泛使用的 BigCloneBench,只标注了 10 个功能的代码克隆[54],不足以满足现实需求.要更好地研究深度学习模型,一方面,需要大量准确的标注数据,虽然研究者提出的变异测试方法[46]可以生成大量的训练数据,但是真实、准确的标注代码克隆也是不可或缺的.另一方面,代码表征的复杂性需要被充分探索,相关研究使用到了词汇[52]和语法[39]的方式用于研究,本文归纳的基于文本、词汇、语法、语义的表征方式都可以作为输入应用于深度学习的检测中,如何有效地将这些表征技术应用到代码克隆检测中需要未来工作进一步分析与研究.

5.2.4 知识图谱的技术也是解决类型 4 克隆检测的手段之一

对于类型 4 的代码克隆来说,功能相同但是结构不同是其主要的特点,这个特点背后的逻辑是它们都拥有相同的业务性或者系统性的知识,如果能够建立起一个软件系统的知识图谱,就能够帮助跨过符合类型 4 定义的代码克隆之间天然的语义鸿沟,自上而下地检测编码方式不同但是实现了相同功能或业务的不同粒度的源代码.比如说,有研究将 Java 的 API 知识引入了代码检索领域[88],其主要思想是利用额外的 API 文档作为知识库提高检索的准确率,类似地,借助构建的或已有的外部知识,可以增强对源代码的表征能力,提高对类型 4 的代码克隆检测能力.解决实践中的工程问题,推动代码克隆的广泛应用.

在目前的技术中,代码克隆检测技术能够对类型 1 到类型 3 的代码克隆的检测有不错的效果,但却并不能完全应用到工业界之中.首先,这些工具有自己的局限性,不能完全满足第 5.2 节中提到的特性.其次,代码克隆的相关问题在工业界的应用场景中需要更多的实践才能凸显其价值,比如 Dang 等人[20]评估了 XIAO[34]这一工具在微软不断迭代取得的成功:XIAO 的主要成就在于集成到了 Visual Studio 并且在微软开发团队的日常运作中被广泛使用.从中我们可以得到两个方面的启示:(1) 技术方面,在开发技术原型时要结合初步的调查,考虑到在实际中它们的使用情况(即使这样的设想有时候不如预期),第二则是要考虑交互问题,第三是要考虑模型的效率和可靠性.(2) 社会方面,首先,对于目标的用户要有内部知情人来帮助交互,从而能够协调两边的信息;第二要能够真诚地与生产团队交互;第三是要有良好的反馈机制,帮助产品的迭代与优化;第四是要有高效的协作处理过程;最后则是要能够积极、主动地去寻找、拓展产品的领域.除此之外,一个科研原型的落地有两种模式:pull 模式和 push 模式,pull 模式就是根据已有的问题来进行科研并提出解决方案;而 push 模式则是根据科研问题和初步调查,结合科研直觉开发出原型工具,并尝试将原型进行“推销”,在与早期使用者的交互中获得更多反馈并加以改进.

6 总 结

代码克隆对软件系统的开发、维护产生一定程度的影响,其中负面影响包括降低软件稳定性、造成代码库的冗余和软件缺陷的传播.学术界对代码克隆的研究有超过 20 年的历史,但是尚有许多问题,特别是代码克隆的检测问题需要研究.本文基于当前研究进展,重新梳理归纳了代码克隆检测研究,也总结讨论了当前代码克隆研究面临的关键问题及未来发展趋势.主要工作总结如下:(1) 从源代码表征方式这种不同于以往研究该问题的角度阐述并归类了现有的克隆检测方法;(2) 总结了模型评估中使用的实验验证方法与性能评估指标;(3) 从科学性、实用性和技术难点 3 个方面归纳总结了代码克隆研究的关键问题,围绕数据标注、表征方法、模型构建和工程实践 4 个方面,阐述了问题的可能解决思路和研究的未来发展趋势.

References:

[1] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, 2002,28(7):654–670.

[2] Roy CK, Cordy JR. A survey on software clone detection research. Queen’s School of Computing TR, 2007,541(115):64–68.

[3] Rieger M, Ducasse S, Lanza M. Insights into system-wide code duplication. In: Proc. of the 11th Working Conf. on Reverse Engineering. IEEE, 2004. 100–109.

[4] Sheneamer A, Kalita J. A survey of software clone detection techniques. Int’l Journal of Computer Applications, 2016,137(10): 1–21.

[5] Chen W-K, Li B, Gupta R. Code compaction of matching single-entry multiple-exit regions. In: Proc. of the Int’l Static Analysis Symp. Springer-Verlag, 2003. 401–417.

[6] Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. ACM SIGSOFT Software Engineering Notes, 2005,30:187–196.

[7] Aversano L, Cerulo L, Di Penta M. How clones are maintained: An empirical study. In: Proc. of the European Conf. on Software Maintenance and Reengineering. IEEE, 2007. 81–90.

[8] Koschke R. Survey of research on software clones. In: Proc. of the Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007.

[9] Mondal M, Rahman MS, Saha RK, Roy CK, Krinke J, Schneider KA. An empirical study of the impacts of clones in software maintenance. In: Proc. of the 19th IEEE Int'l Conf. on Program Comprehension. IEEE, 2011. 242–245.

[10] Rattan D, Bhatia R, Singh M. Software clone detection: A systematic review. Information and Software Technology, 2013,55(7): 1165–1199.

[11] Patenaude J-F, Merlo E, Dagenais M, Laguë B. Extending software quality assessment techniques to Java systems. In: Proc. of the 7th Int'l Workshop on Program Comprehension. IEEE, 1999. 49–56.

[12] Hotta K, Sano Y, Higo Y, Kusumoto S. Is duplicate code more frequently modified than non-duplicate code in software evolution: An empirical study on open source software. In: Proc. of the Joint ERCIM Workshop on Software Evolution (EVOL) and Int'l Workshop on Principles of Software Evolution (IWPSE). ACM, 2010. 73–82.

[13] Lozano A, Wermelinger M, Nuseibeh B. Evaluating the harmfulness of cloning: A change based experiment. In: Proc. of the 4th Int'l Workshop on Mining Software Repositories. IEEE Computer Society, 2007. 18.

[14] Krinke J. A study of consistent and inconsistent changes to code clones. In: Proc. of the 14th Working Conf. on Reverse Engineering (WCRE 2007). IEEE, 2007. 170–178.

[15] Kapser CJ, Godfrey MW. "Cloning considered harmful" considered harmful: Patterns of cloning in software. Empirical Software Engineering, 2008,13(6):645.

[16] Mondal M, Rahman MS, Roy CK, Schneider KA. Is cloned code really stable. Empirical Software Engineering, 2018,23(2): 693–770.

[17] Juergens E, Deissenboeck F, Hummel B, Wagner S. Do code clones matter. In: Proc. of the 31st IEEE Int'l Conf. on Software Engineering. IEEE, 2009. 485–495.

[18] Mondal M, Roy CK, Schneider KA. Dispersion of changes in cloned and non-cloned code. In: Proc. of the 6th Int'l Workshop on Software Clones. IEEE Press, 2012. 29–35.

[19] Lozano A, Wermelinger M. Tracking clones' imprint. In: Proc. of the 4th Int'l Workshop on Software Clones. ACM, 2010. 65–72.

[20] Dang Y, Zhang D, Ge S, Huang R, Chu C, Xie T. Transferring code-clone detection and analysis to practice. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering in Software Engineering in Practice Track (ICSE-SEIP). 2017. 53–62.

[21] Lessmann S, Baesens B, Mues C, Pietsch S. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. on Software Engineering, 2008,34(4):485–496.

[22] Goto A, Yoshida N, Ioka M, Choi E, Inoue K. How to extract differences from similar programs: A cohesion metric approach. In: Proc. of the 7th Int'l Workshop on Software Clones. IEEE Press, 2013. 23–29.

[23] Roy CK, Zibran MF, Koschke R. The vision of software clone management: Past, present, and future. In: Proc. of the Software Evolution Week—IEEE Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). 2014. 18–33.

[24] Fontana FA, Zanoni M, Ranchetti A, Ranchetti D. Software clone detection and refactoring. Int'l Scholarly Research Notices, 2013.

[25] Yuan Y, Guo Y. CMCD: Count matrix based code clone detection. In: Proc. of the 18th Asia Pacific Software Engineering Conf. (APSEC). IEEE, 2011. 250–257.

[26] Prechelt L, Malpohl G, Philippsen M. Finding plagiarisms among a set of programs with JPLAG. Journal of Universal Computer Science, 2002,8(11):1016.

[27] Baker BS. On finding duplication and near-duplication in large software systems. In: Proc. of the 2nd Working Conf. on Reverse Engineering. IEEE, 1995. 86–95.

[28] Tsantalis N, Mazinanian D, Krishnan GP. Assessing the refactorability of software clones. IEEE Trans. on Software Engineering, 2015,41(11):1055–1090.

[29] Tsantalis N, Mazinanian D, Rostami R. Clone refactoring with Lambda expressions. In: Proc. of the 39th Int'l Conf. on Software Engineering. IEEE Press, 2017. 60–70.

[30] Meng N, Hua L, Kim M, McKinley KS. Does automated refactoring obviate systematic editing. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol. 1. IEEE Press, 2015. 392–402.

[31] Walenstein A, Lakhotia A. The software similarity problem in malware analysis. In: Proc. of the Schloss Dagstuhl- Leibniz-Zentrum für Informatik. 2007.

[32] Kontogiannis K, Galler M, DeMori R. Detecting code similarity using patterns. In: Proc. of the Working Notes of the 3rd Workshop on AI and Software Engineering. 1995. 6.

[33] Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. IEEE Trans. on Software Engineering, 2007,33(9).

[34] Dang Y, Zhang D, Ge S, Chu C, Qiu Y, Xie T. XIAO: Tuning code clones at hands of engineers in practice. In: Proc. of the 28th Annual Computer Security Applications Conf. ACM, 2012. 369–378.

[35] Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Proc. of the Int'l Conf. on Software Maintenance. IEEE, 1998. 368–377.

[36] Krinke J. Identifying similar code with program dependence graphs. In: Proc. of the 8th Working Conf. on Reverse Engineering. IEEE, 2001. 301–309.

[37] Li Z, Lu S, Myagmar S, Zhou Y. CP-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. on Software Engineering, 2006,32(3):176–192.

[38] Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension. IEEE, 2008. 172–181.

[39] Wei H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. AAAI Press, 2017. 3034–3040.

[40] Wang W, Godfrey MW. Recommending clones for refactoring using design, context, and history. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution (ICSME). 2014. 331–340.

[41] Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: Scaling code clone detection to big-code. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). 2016. 1157–1168.

[42] Jiang L, Misherghi G, Su Z, Glondu S. Deckard: Scalable and accurate tree-based detection of code clones. In: Proc. of the 29th Int'l Conf. on Software Engineering. IEEE Computer Society, 2007. 96–105.

[43] Hotta K, Yang J, Higo Y, Kusumoto S. How accurate is coarse-grained clone detection: Comparision with fine-grained detectors. In: Proc. of the Electronic Communication of the European Association of Software Science and Technology. 2014. 63.

[44] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proc. of the IEEE Int'l Conf. on Software Maintenance. IEEE, 1999. 109–118.

[45] Lee S, Jeong I. SDD: High performance code clone detection system for large scale source code. In: Proc. of the Companion to the 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. ACM, 2005. 140–141.

[46] Roy CK, Cordy JR. A mutation/injection-based automatic framework for evaluating code clone detection tools. In: Proc. of the IEEE Int'l Conf. on Software Testing, Verification, and Validation Workshops. 2009. 157–166.

[47] Livieri S, Higo Y, Matushita M, Inoue K. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In: Proc. of the 29th Int'l Conf. on Software Engineering. 2007. 106–115.

[48] Yuan Y, Guo Y. Boreas: An accurate and scalable token-based approach to code clone detection. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, 2012. 286–289.

[49] Murakami H, Hotta K, Higo Y, Igaki H, Kusumoto S. Folding repeated instructions for improving token-based code clone detection. In: Proc. of the 12th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). IEEE, 2012. 64–73.

[50] Murakami H, Hotta K, Higo Y, Igaki H, Kusumoto S. Gapped code clone detection with lightweight source code analysis. In: Proc. of the 21st Int'l Conf. on Program Comprehension (ICPC). IEEE, 2013. 93–102.

[51] Wang P. CCAligner: A token based large-gap clone detector. In: Proc. of the 40th Int'l Conf. on Software Engineering. ACM, 2018. 1066–1077.

[52] Li L, Feng H, Zhuang W, Meng N, Ryder B. CCLearner: A deep learning-based clone detection approach. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2017. 249–260.

[53] Yan X, Han J, Afshar R. CloSpan: Mining: Closed sequential patterns in large datasets. In: Proc. of the 2003 SIAM Int'l Conf. on Data Mining. SIAM, IEEE, 2003. 166–177.

[54] Svajlenko J, Roy CK. Evaluating clone detection tools with BigCloneBench. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2015. 131–140.

[55] AMoZ J. Identification of common molecular subsequences. Journal of Molecular Biology, 1981,147(1):195−197.

[56] Mayrand J, Leblanc C, Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. In: Proc. of the 1996 Int'l Conf. on Software Maintenance. 1996. 244.

[57] Wahler V, Seipel D, Wolff J, Fischer G. Clone detection in source code by frequent itemset techniques. In: Proc. of the 4th IEEE Int'l Workshop on Source Code Analysis and Manipulation. IEEE, 2004. 128−135.

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

[59] Kontogiannis KA, DeMori R, Merlo E, Galler M, Bernstein M. Pattern matching for clone and concept detection. Automated Software Engineering, 1996,3(1-2):77–108.

[60] Kodhai E, Kanmani S, Kamatchi A, Radhika R, Saranya BV. Detection of type-1 and type-2 code clones using textual analysis and metrics. In: Proc. of the Int'l Conf. on Recent Trends in Information, Telecommunication and Computing. IEEE, 2010. 241–243.

[61] Abd-El-Hafiz SK. A metrics-based data mining approach for software clone detection. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. IEEE, 2012. 35–41.

[62] Raheja K, Tekchandani R. An emerging approach towards code clone detection: Metric based approach on byte code. Int'l Journal of Advanced Research in Computer Science and Software Engineering, 2013,3(5).

[63] Agrawal A, Yadav SK. A hybrid-token and textual based approach to find similar code segments. In: Proc. of the 4th Int'l Conf. on Computing, Communications and Networking Technologies (ICCCNT). IEEE, 2013. 1–4.

[64] Komondoor R, Horwitz S. Using slicing to identify duplication in source code. In: Proc. of the Int'l Static Analysis Symp. Springer-Verlag, 2001. 40–56.

[65] Hummel B, Juergens E, Heinemann L, Conradt M. Index-based code clone detection: incremental, distributed, scalable. In: Proc. of the 2010 IEEE Int'l Conf. on Software Maintenance (ICSM). IEEE, 2010. 1–9.

[66] Rivest R. The MD5 message-digest algorithm. 1992. https://tools.ietf.org/html/rfc1321

[67] Liu C, Chen C, Han J, Yu PS. GPLAG: Detection of software plagiarism by program dependence graph analysis. In: Proc. of the 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. ACM, 2006. 872–881.

[68] Higo Y, Yasushi U, Nishino M, Kusumoto S. Incremental code clone detection: A PDG-based approach. In: Proc. of the 18th Working Conf. on Reverse Engineering (WCRE). IEEE, 2011. 3–12.

[69] Funaro M, Braga D, Campi A, Ghezzi C. A hybrid approach (syntactic and textual) to clone detection. In: Proc. of the 4th Int'l Workshop on Software Clones. ACM, 2010. 79–80.

[70] Saini V, Farmahinifarahani F, Lu Y, Baldi P, Lopes C. OREO: Detection of clones in the twilight zone. In: Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2018. 354−365.

[71] Higo Y, Sawa K, Kusumoto S. Problematic code clones identification using multiple detection results. In: Proc. of the 16th Asia-Pacific Software Engineering Conf. IEEE, 2009. 365–372.

[72] Baldi P, Chauvin Y. Neural networks for fingerprint recognition. Neural Computation, 1993,5(3):402–418.

[73] Zibran MF, Saha RK, Asaduzzaman M, Roy CK. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In: Proc. of the 16th IEEE Int'l Conf. on Engineering of Complex Computer Systems (ICECCS). IEEE, 2011. 295–304.

[74] Gabel M, Jiang L, Su Z. Scalable detection of semantic clones. In: Proc. of the 30th Int'l Conf. on Software Engineering. ACM, 2008. 321–330.

[75] Saha RK, Asaduzzaman M, Zibran MF, Roy CK, Schneider KA. Evaluating code clone genealogies at release level: An empirical study. In: Proc. of the 10th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. IEEE, 2010. 87–96.

[76] Rysselberghe FV, Demeyer S. Evaluating clone detection techniques from a refactoring perspective. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering. IEEE Computer Society, 2004. 336–339.

[77] Shafieian S, Zou Y. Comparison of clone detection techniques. Technical Report, Queen's University, 2012. https://www.researchgate.net/publication/267840728_Comparison_of_Clone_Detection_Techniques

[78] Rahman MS, Roy CK. A change-type based empirical study on the stability of cloned code. In: Proc. of the 14th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). IEEE, 2014. 31–40.

[79] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments. In: Proc. of the 27th Int'l Conf. on Software Engineering. IEEE, 2005. 402–411.

[80] Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 2018,51(4):81.

[81] Göde N, Koschke R. Incremental clone detection. In: Proc. of the 13th European Conf. on Software Maintenance and Reengineering. IEEE, 2009. 219–228.

[82] Nguyen TT, Nguyen HA, Al-Kofahi JM, Pham NH, Nguyen TN. Scalable and incremental clone detection for evolving software. In: Proc. of the 25th IEEE Int'l Conf. on Software Maintenance. IEEE, 2009. 491–494.

[83] Gotel OC, Finkelstein CW. An analysis of the requirements traceability problem. In: Proc. of the 1st IEEE Int'l Conf. on Requirements Engineering. IEEE, 1994. 94–101.

[84] Heckman S, Williams L. A systematic literature review of actionable alert identification techniques for automated static code analysis. Information and Software Technology, 2011,53(4):363–387.

[85] Tan L, Yuan D, Krishna G, Zhou Y. /* icomment: Bugs or bad comments /*. ACM SIGOPS Operating Systems Review, 2007,41: 145–158.

[86] Allamanis M, Khademi M, Brockschmidt M. Learning to represent programs with graphs. arXiv Preprint arXiv: 1711.00740, 2017.

[87] Yan S, Xu D, Zhang B, Zhang H-J, Yang Q, Lin S. Graph embedding and extensions: A general framework for dimensionality reduction. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2007,29(1):40–51.

[88] Lin Z, Zou Y, Zhao J, Xie B. Improving software text retrieval using conceptual knowledge in source code. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2017. 123–134.

陈秋远(1995－),男,四川成都人,博士生,主要研究领域为智能软件工程,软件仓库挖掘.

鄢萌(1989－),男,博士,助理研究员,CCF 专业会员,主要研究领域为智能软件工程,软件仓库挖掘,软件维护与演化.

李善平(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式计算,软件工程,操作系统内核.

夏鑫(1986－),男,博士,讲师,博士生导师,CCF 专业会员,主要研究领域为软件仓库挖掘,经验软件工程.