软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(5):1−19 [doi: 10.13328/j.cnki.jos.005966]

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

智能代码补全方法研究进展*

杨 博[1], 张 能[1], 李善平[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: 夏鑫, E-mail: xin.xia@monash.edu

摘 要: 代码补全(code completion)是自动化软件开发的重要功能之一,是大多数现代集成开发环境和源代码编辑器的重要组件.代码补全提供即时类名、方法名和关键字等预测,辅助开发人员编写程序,直观提高软件开发效率.近年来,开源软件社区中源代码和数据规模不断扩大,人工智能技术取得卓越进展,这对自动化软件开发技术产生了极大促进作用.智能代码补全(intelligent code completion)根据源代码建立语言模型,从语料库学习已有代码特征,根据待补全位置的上下文代码特征在语料库中检索最相似的匹配项进行推荐和预测.相对于传统代码补全,智能代码补全凭借其高准确率、多补全形式、可学习迭代的特性成为软件工程领域的热门方向之一.研究者们在智能代码补全方面进行了一系列研究,根据这些方法如何表征和利用源代码信息的不同方式,可以将它们分为基于编程语言表征和基于统计语言表征两个研究方向,其中,基于编程语言表征又分为标识符序列、抽象语法树、控制/数据流图3个类别,基于统计语言表征又分为*N*-gram模型、神经网络模型两个类别.从代码表征的角度入手,对近年来代码补全方法研究进展进行梳理和总结,主要内容包括:(1) 根据代码表征方式阐述并归类了现有的智能代码补全方法;(2) 总结了代码补全的一般过程和模型评估中的模型验证方法与性能评估指标;(3) 归纳了智能代码补全的主要挑战;(4) 展望了智能代码补全的未来发展方向.

关键词: 代码补全;代码表征;软件开发工具

中图法分类号: TP311

中文引用格式: 杨博,张能,李善平,夏鑫.智能代码补全方法研究进展.软件学报,2020,31(5). http://www.jos.org.cn/1000-9825/5966.htm

英文引用格式: Yang B, Zhang N, Li SP, Xia X. Intelligent code completion: A literature review. Ruan Jian Xue Bao/Journal of Software, 2020,31(5) (in Chinese). http://www.jos.org.cn/1000-9825/5966.htm

Intelligent Code Completion: A Literature Review

YANG Bo[1], ZHANG Neng[1], LI Shan-Ping[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 Completion is one of the crucial functions of automation software development. It is an essential component of most modern integrated development environments and source code editors. Code completion provides predictions such as instant class names, method names, keywords, and assists development. People write programs to improve the efficiency of software development intuitively. In recent years, with the expanding of the source code and data scale in the open-source software community, and outstanding progress in artificial intelligence technology, the automation software development technology has been much promoted. Intelligent code completion (intelligent code completion) builds a language model for source code, learns existing code features from the corpus, and retrieves the most similar matches in the corpus for recommendation and prediction based on the context code features to be replenished. Compared to traditional code completion, intelligence code completion has become one of the hot trends in the field of software engineering with its high accuracy, multiple completion forms, and iterative learning characteristics. Researchers have conducted a series of researches on

收稿时间: 2019-08-19; 修改时间: 2019-10-28; 采用时间: 2020-01-02; jos 在线出版时间: 2020-04-07

intelligent code completion. According to how these methods represent and utilize the different forms of source code information, they can be divided into two research directions: Programming language representation and statistical language representation. The programming language is divided into three types: token sequences, abstract syntax tree, and control/data flow graph. The statistical language also has two types: *n*-gram model and the neural network model. This paper starts from the perspective of code representation and summarizes and summarizes the research progress of code completion methods in recent years. The main contents include: (1) expounding and classifying existing intelligent code completion methods according to code representation; (2) summarizing the experimental verification methods and performance evaluation indicators used in model evaluation; (3) summarizing the critical issues of intelligent code completion; (4) Looking forward to the future development of intelligent code completion.

Key words: code completion; code representation; software development tool

1 引 言

在软件工程的研究中,提高开发效率和质量是从业者和研究者们关注的核心问题.为此,许多研究通过改进软件开发方法和运用自动化工具来提高软件开发的自动化水平,如缺陷定位[1,2]、程序修复[3,4]、类型注解[5,6]和 API(application programming interface,应用程序接口)推荐[7,8]等任务中,自动化工具得到了广泛应用.其中,代码补全(code completion)作为一种直观减少软件开发人员工作量的软件自动化技术,是许多集成开发环境(IDE)的关键功能组件.代码补全技术基于开发人员的输入和已有项目代码,即时预测待补全代码中的类名、方法名和代码片段等,并为开发人员提供建议列表.通过这种方式,代码补全技术能够减轻键入负担,减少拼写错误,并且开发人员不必花费很长时间记忆不熟悉的类名和方法名,最终提高开发效率.随着代码补全技术相关研究的深入和工具[9−14]的应用,代码补全已经成为开发人员最常用操作之一[15].目前,每个主流的 IDE 都有一个基于特定语言的代码补全组件,并且开发人员常用的文本编辑器(如 Notepad++)也提供了一定的文本补全功能.

代码补全的研究最早可以追溯到 1971 年的 SPELL[16],用于检查代码中的拼写错误.早期的代码补全主要基于文本自动完成(auto-complete)功能,但是这些由文本处理移植过来的技术在实际应用中并不尽如人意,例如只能补全曾经输入过的词汇.但是对于编程开发而言,开发人员会定义不同的变量和方法名并希望代码补全工具能够补全常用的 API 方法和对应参数.同时,早期工具给出的补全建议无视了编程语言的语法规则,需要大量人rog 工修正.随着集成开发环境逐渐普及,代码补全技术作为其重要功能组件,得以成为关注的焦点.最早的代码补全插件 IntelliSense 在 1996 年首次作为集成开发环境 Visual Studio 的一个主要特性为人所知.Eclipse 利用类型检查和启发式规则对标识符(token)进行预测,推荐方法名、参数等关键字的补全.Pletcher 和 Hou 提出了 BCC 工具[17,18],对 Eclipse 给出的预测结果进行排序和筛选.

以上代码补全方法通常只利用已输入的代码和语法规则,通过人工定义启发式规则进行预测和补全,很少考虑待补全代码与上文的语义关联.随着方法和参数的版本迭代更新,这些规则就会落后甚至失效.为了解决这些问题,研究者们建立学习模型解析源代码的语义和结构信息,利用开源代码语料库对代码补全模型进行训练,动态显示补全列表.2004 年,Hill 和 Rideout 提出:代码克隆并不一定是代码中需要去除的弊病[9],可以利用代码克隆检测方法在代码库检测疑似克隆代码进行方法补全,从而提升工作效率.2008 年,Robbes 和 Lanza 指出:目前代码补全方法的限制在于过于庞大的搜索空间[19],除了代码上下文中出现的词汇表和编程语言规范外,还必须引入额外的信息来源,如代码修改记录等.2009 年,Bruch 和 Monperrus 等人[11]首次在文献中提出智能代码补全(intelligent code completion)[11−13,19−36]的概念,指出,从已有的代码库中可以挖掘到更多信息.

智能代码补全方法涵盖许多技术,如信息检索、自然语言处理等,但所有的智能代码补全模型都将待补全位置的代码上下文与从开源代码语料库中学习到的代码上下文进行关联,再推荐相似度高的补全建议.根据此流程,不同智能代码补全方法的差异通常可以归纳为两个方面:(1) 代码上下文的表征方式;(2) 上下文之间相似度的度量方式.代码表征方式指对代码进行信息提取和结构化表示的方法,在代码上下文的表征方式中,有两个主要的研究方向:其一,是专注于从源代码中提取结构特征[11,19,20],如上下文中的类型信息;其二是不选择结构特征,而是使用自然语言处理技术从源代码中提取自然语言的统计特征和重复模式[14,21].这两个研究方向也并

不是泾渭分明的,也有研究者通过选取特定的结构特征建立模型,却使用自然语言处理技术在这些模型上进行预测.例如,Bielik 等人[22]开发了一种特定领域的语言,并学习了一个统计模型预测用哪种上下文进行补全.

虽然智能代码补全方法近些年取得了一定进展,但仍然存在许多问题.Jin 和 Servant 关于代码补全方法的实证研究[26]表明:IntelliSense 在 20%的情况下可以给出长度为 1 的推荐列表并给出正确的补全建议,但是在16%的情况下推荐列表的长度很长.这表明 IntelliSense 对返回的结果不自信,并且在这种情况下往往推荐列表的第 10 位之后才能找到正确的补全结果.这影响了开发人员检查推荐列表的时间和倾向于接受补全建议的信心.Hellendoorn 等人[37]对 66 个真实开发人员的 15 000 次补全操作进行研究,发现智能代码补全方法的合成评估集准与真实世界的补全行为存在差异,导致对代码补全方法性能的高估和与实践需求之间的脱节.因此,智能代码补全方法仍需要明确实践需求,改进现有模型和实证研究.

本文首次从代码表征方式的角度对代码补全方法进行综述研究,整理和归纳当前智能代码补全技术,指出已有的研究思路及欠缺,并展望智能代码补全技术未来发展.

1.1 文献选取方式

本文采用以下流程完成对相关文献的获取.本综述的目的是了解智能代码补全技术的研究现状.面向这一目标,本综述在公开的期刊及会议论文、出版书籍中,检索在智能代码补全方法研究中提出新模型、新算法,或为代码补全方法提供实证研究支持的文献.本文分 3 步并根据上述原则在文献库中进行检索和选取.

(1) 本综述选用 ACM 电子文献数据库、IEEE Xplore 电子文献数据库、Springer Link 电子文献数据库、中国知网搜索引擎及 Google 学术搜索引擎等进行原始搜索.论文检索的关键字包括 code completion, intelligent completion,api completion 等.同时,在标题、摘要、关键词和索引中进行检索;

(2) 本综述依据中国计算机学会(CCF)推荐国际学术会议和期刊列表中软件工程和人工智能领域分布中进行文献检索,有 TOSEM,TSE,EMSE,JSS,ICSE,FSE/ESEC,ASE,MSR 等,搜索时间从 2004 年开始;

(3) 为避免遗漏相关研究,在之前两步搜索的基础上,根据每篇文献的参考文献列表进行寻找与代码补全问题相关的研究文献,并添加到相关文献中.

基于上述选取原则和检索步骤,本文选取 56 篇文献作为综述总结的相关文献.在这些文献中:提出了智能代码补全方法的新理论、新算法的直接相关文献有 28 篇;为智能代码补全方法提供理论支持,如代码表征方式和源代码语言模型的部分相关文献有 8 篇;其他为智能代码补全的背景动机、评估方法提供实证研究支持的部分相关文献有 9 篇.上述文献分布情况在图 1 中具体展示,发表过相关研究较多的期刊与会议有:ASE 9 篇,ICSE8 篇,FSE8 篇,ICSM/ICSME5 篇.

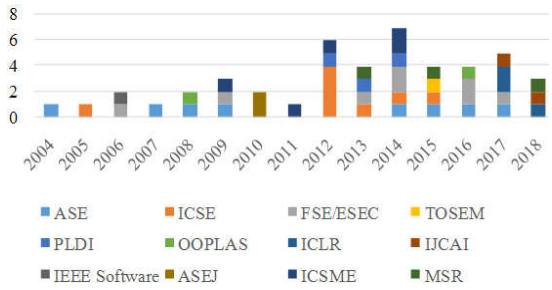

Fig.1 Illustration of literatures by year

图 1 文献分布

从总体趋势来看:从 2004 年开始,每年都有智能代码补全方法相关研究出现,并且每年数量总体呈上升趋势;从期刊和会议主题来看,智能代码补全方法的研究主要集中在软件工程领域,人工智能领域也有 4 篇论文.

1.2 本文整体结构

本文第 2 节介绍代码补全方法的相关概念与研究概况.第 3 节介绍代码补全方法中的代码表征方式,将已

有工作划分为两类研究思路.第 4 节介绍代码补全方法的模型验证方法和评估指标.第 5 节指出智能代码补全方法的主要挑战.第 6 节讨论智能代码补全方法未来研究方向.第 7 节对本文进行总结.

2 智能代码补全相关概念与研究概况

2.1 智能代码补全相关概念

为了便于论文阐述,将智能代码补全相关概念陈述如下.

(1) 标识符(token):标识符是源代码编译过程中的最小单位,包括关键词、函数名、变量名、运算符等;

(2) 代码片段(code snippet):代码片段是一小段的源代码,它包含一些功能性的语句,例如类声明、函数声明,也可以是一个有起止标识符的代码块;

(3) 输入前缀:开发人员已经键入的若干字符,用于限定当前位置的完整标识符,缩小可能的补全结果.开发人员可以直接从空白获取完整代码片段,如直接补全以默认参数填充的完整模板;

(4) 代码补全:基于开发人员的输入前缀和代码片段,预测待补全代码片段中的类名、方法名和代码片段等,并为开发人员提供建议列表.

2.2 代码补全类型定义

根据待补全对象,代码补全主要可以分为标识符补全、代码片段补全和关键词/缩略词补全.

(1) 标识符补全

根据输入前缀,对不完整的标识符进行补全,补全对象包括方法名、变量名、参数名等,如图 2 所示.

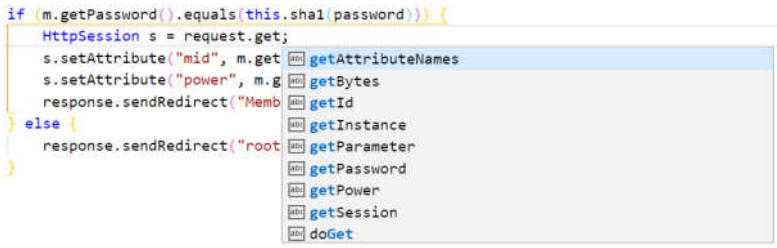

Fig.2 Example of Visual Studio's token completion

图 2 Visual Studio 补全插件的标识符补全示例

标识符补全中有一种使用频率高的补全方式是 API 方法调用补全,即在类名后输入“.”调用方法或变量,并补全以默认参数填充的完整模板,如图 3 所示.

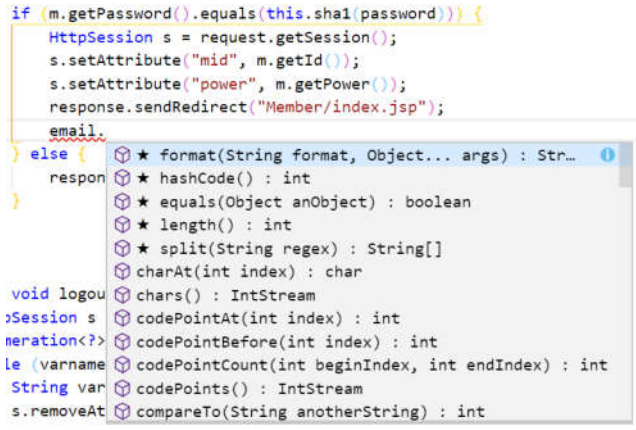

Fig.3 Example of Visual Studio's API method call completion

图 3 Visual Studio 补全插件的 API 方法调用补全示例

(2) 代码片段补全

在一些代码补全工具,如 SLANG[14]中,可以输入带有空缺的代码片段,工具会自动生成符合编程语言规则的语句进行补全,如图 4 所示.

```
void exampleMediaRecorder() throws IOException{
    Camera camera=Camera. open();
    camera.setDisplayOrientation(90);
    ?   {camera}    //Hole1
    SurfaceHolder holder=getHolder();
    holder.addCallback(this);
    holder.setType(SurfaceHolder. SURFACE_TYPE_PUSH_BUEEERS);
    MediaRecorder rec=new MediaRecorder();
    ?   {rec}   //Hole2
    rec.setAudioSource(MediaRecorder. AudioSource. MIC);
    rec.setVideoSource(MediaRecorder. VideoSource. DEFAULT);
    rec.setOutputFormat(MediaRecorder. OutputFormat. MPEG_4);
    ?   {rec}   //Hole3
    rec.setOutputFile("file. mp4");
    rec.setPreviewDisplay(holder. getSurface());
    rec.setOrientationHint(90);
    rec.prepare();
    ?   {rec}   //Hole4
}
```

(a) 带有空缺的代码片段

```
void exampleMediaRecorder() throws IOException{
    Camera camera=Camera. open();
    camera.setDisplayOrientation(90);
    camera.unlock();
    SurfaceHolder holder=getHolder();
    holder.addCallback(this);
    holder.setType(SurfaceHolder. SURFACE_TYPE_PUSH_BUEEERS);
    MediaRecorder rec=new MediaRecorder();
    rec.setCamera(camera);
    rec.setAudioSource(MediaRecorder. AudioSource. MIC);
    rec.setVideoSource(MediaRecorder. VideoSource. DEFAULT);
    rec.setOutputFormat(MediaRecorder. OutputFormat. MPEG_4);
    rec.setAudioEncoder(1);
    rec.setVideoEncoder(3);
    rec.setOutputFile("file. mp4");
    rec.setPreviewDisplay(holder. getSurface());
    rec.setOrientationHint(90);
    rec.prepare();
    rec.start();
}
```

(b) 补全后的代码片段

Fig.4 Code snippet with vacancies, and completed code snippet

图 4 带有空缺的代码片段和补全后的代码片段

(3) 关键词/缩略词补全

输入简短的、未预定义的短语或缩略词,补全为完整的函数和参数.这种补全方式是由 Little 和 Miller 在 2007 年提出的关键词编程[38]所采用的,如图 5 所示.

```
public List<String> getLines(BufferedReader in) {
    List<String> lines = new Vector<String>();
    while(in.ready()){
        add line
    }
    return lines;
}
```

(a) 带有关键词的代码片段

```
public List<String> getLines(BufferedReader in) {
    List<String> lines = new Vector<String>();
    while(in.ready()){
        lines.add(in.readLine());
    }
    return lines;
}
```

(b) 补全后的代码片段

Fig.5 Code snippet with keywords, and completed code snippet

图 5 带有关键词的代码片段和补全后的代码片段

Han 和 Miller 在 2009 年提出了缩略词补全[39,40].与关键词补全不同的是,缩略词更加简洁,并且一次可以翻译多个关键词,提高了补全效率,如图 6 所示.缩略词补全要求输入的关键词或缩略词未经预先定义,但是结果的模糊化和出错体验较差,在实际中的应用不多,导致这种补全方式不是主流的研究方向.

Fig.6 Example of abbreviation completion

图 6 缩略词补全示例

2.3 智能代码补全研究概况

代码补全问题可以归结为预测标识符(token)的问题.传统的代码补全方法基于已输入的代码和语法规则进行补全,需要制定许多启发式规则进行判别.随着方法和参数的版本迭代更新,这些规则就会落后甚至失效.推荐列表按字母排序,增加了开发人员的额外键入和选择时间.

鉴于这些问题,研究者将判别和推荐的任务交给计算机,让计算机在已有代码中学习规律,与待补全位置匹配,推荐相似度高的补全建议,即智能代码补全.在智能代码补全中,研究者们对代码表征方式进行更深入的研究[11−13,19−36],并引入额外信息,如开源代码托管仓库 Github[41]中许多项目的源代码数据和包括变更历史和提交说明等在内的项目信息.智能代码补全方法的关键点在于代码表征方式,这影响待补全代码和代码语料库的处理方式,和上下文之间相似度的比较方式.一部分研究者们专注于模型改进,从源代码中提取额外的结构特征和语义信息.例如,Gvero 等人在他们的工具 InSynth[36]中对代码类型开发一种简洁的判断表示方法,通过类型合并减少可能补全结果的搜索空间,并基于代码语料库进行排序.

不同于从源代码进一步挖掘额外信息,另一部分研究者们尝试通过其他角度理解代码,该方向一个标志性研究成果是 2012 年 Hindle 等人假设编程语言是自然语言的一种,它具有可重复并带有可预测的统计学规律.他们通过自然语言中常用的概率模型 *N*-gram 对 Java 语言进行建模[21],在代码补全任务上进行了实验验证,得到了比 Eclipse 自带的代码补全插件更好的性能.这一工作展示了利用统计语言模型表征代码的有效性,并带动了一系列使用统计语言模型和自然语言处理技术在软件工程领域中的应用[12,14,30,42−45].

用神经网络训练统计语言模型是由 Bengio 提出[46]的.随着可用于训练的计算资源的增加,研究者们更广泛地采用神经网络预测句子的概率[47].神经网络模型不仅仅可以推荐下一个位置的单词和固定数量的前序单词之间的规则性,并且能提取单词之间距离较远的关系.具体介绍请见第 3.2 节.

同样,深度学习作为神经网络的热门研究方向,也在代码补全领域得到了应用.胡星等人[84]已经对采用深度学习技术的程序生成和代码补全进行了文献综述.本文不仅包括应用深度学习技术的代码补全研究,而且包括其他方法的研究,并且依据代码补全的一般过程,从代码表征角度出发,对代码补全方法进行了梳理分类.

近年来,代码补全方法的实用性也受到一些研究者的质疑,实证研究[26,37]均表明,代码补全方法在一些情况下并不如直观设想地有用.智能代码补全方法仍需要进一步的研究探索和实践应用.

2.4 代码补全一般过程

智能代码补全方法在补全方式、语言模型和评估方法存在很大差异,但是基本框架和主要流程相似.智能补全方法的目的在于让计算机读懂源代码(即提取源代码特征)、学习如何寻找补全内容(即模型学习和相似比较),将补全内容与待补全位置进行匹配(即结果过滤整合),最后给出相应的补全建议(即结果呈现).图 7 展示的是智能代码补全方法的一般过程,包括代码表征、模型学习、相似比较、结果过滤整合和补全结果呈现等.

- 首先让计算机读懂代码,需要将隐含在源代码纯文本中的特征进行提取和表征,即代码表征环节,这是

对源代码的第一步处理,也是最重要的步骤之一.代码表征方法是对代码进行信息提取和结构化表示的方法,对源代码特征提取和表征的不同层次、不同方式,直接决定了后续的模型构建和相似比较.代码表征环节将源代码转化为可比较的结构化信息,如提取源代码的抽象语法树(abstract syntax tree,简称 AST)或程序依赖图,或者为源代码建立统计语言模型,都是对源代码的不同表征方式;

- 然后在模型学习和相似比较环节,计算机通过学习代码语料库中的代码特征,对比补全位置的上下文代码和其他代码片段,预测并推荐相似度高的代码片段.不同的代码表征方式往往会采用不同的相似度和比较算法,如标识符序列会采用向量空间距离、抽象语法树会采用树匹配技术等;
- 在补全结果过滤整合环节,智能代码补全方法会根据类型定义和语法规则,对补全结果进行过滤筛选整合去重,过滤掉不符合语法规则的补全结果;
- 最后在代码补全结果呈现环节,会将上述环节得到的最终推荐结果以适宜的方式返回给开发人员进行选用,如弹出候选窗口供开发人员通过键盘选取.

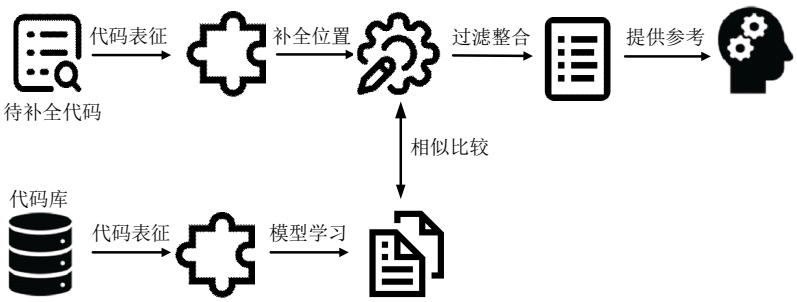

Fig.7 General process of intelligent code completion

图 7 智能代码补全方法一般过程

2.5 模型验证方法

模型验证方法是指通过科学的方法对数据集进一步划分,一部分作为训练集用于模型学习,另一部分作为测试集用于性能评估.在智能代码补全方法研究中,经常采用的模型验证方法是 K 折交叉验证(K-fold cross validation).许多智能代码补全方法的工作取 K 值为 10,采用了 10 折交叉验证的方法[21,27,30,42,43]进行模型验证.K 折交叉验证是一种将数据集划分为较小子集的方法,随机地将数据集打乱并分割为 K 个子样本,称为 K 折;一个子集作为测试集,剩余 K–1 个子集作为训练集.重复 K 次,将每个子样本都作为测试集进行验证计算,最后输出 K 个结果的平均值,作为一次 K 折交叉验证的结果.在实际应用中,研究者们进行 10 次 10 折交叉验证,最后将得到的 10 个结果的均值作为模型最后的评估结果.需要说明的是,并不是所有文献都给出模型的验证方法,本文仅将较常见的方法总结如上.

2.6 代码补全评估指标

为了衡量补全方法的性能,并且由于补全方法会推荐最有可能的 K 个结果,所以常用于评估代码补全模型和算法的性能指标有:平均倒数排名(mean reciprocal rank,简称 MRR)、Top K 个推荐结果的准确率(*Accuracy@K*)、精确率(*Precision@K*)、召回率(*Recall@K*)和 *F*1-*measure@K*.

平均倒数排名 *MRR* 被广泛用于信息检索和推荐系统的性能评估,是针对返回有序的推荐列表算法的标准衡量指标,核心思想是:以第 1 个正确答案的位置作为衡量标准,计算方式为对于一次补全推荐,如果正确答案首次出现在列表中的排名为 n,那么 *MRR* 值即为 1/n,*MRR* 得分越接近 1,则表示正确的补全越靠近推荐列表的顶端.直观来说,*MRR* 的倒数反映了平均正确结果在推荐列表中的位置.对于测试数据中的所有待补全的标识符 T,代码补全方法的 *MRR* 取所有测试补全 *MRR* 得分的平均数:

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

准确率(*Accuracy@K*)是指智能代码补全方法能在 Top *K* 个候选代码中得到正确的补全结果占所有补全操作次数的比例:

$$Accuracy@K=\frac{recommendations_{\text{True}}}{recommendations_{made}} \tag{2}$$

精确率(*Precision@K*)是指智能代码补全方法推荐的 Top *K* 个候选代码中真实代码所占的比例:

$$Precision@K=\frac{recommendations_{made\cap relevant}}{recommendations_{made}} \tag{3}$$

召回率(*Recall@K*)是指智能代码补全方法推荐的 Top *K* 个候选代码中的真实代码占所有应当被推荐的真实代码的比例:

$$Recall@K=\frac{recommendations_{made\cap relevant}}{recommendations_{relevant}} \tag{4}$$

F-measure 由精确率和召回率计算得到,计算方法如下:

$$F=\frac{(1+\beta^2)\cdot precision\cdot recall}{\beta^2\cdot precision+recall} \tag{5}$$

F-measure 允许研究者通过调整β的值来调整精准率和召回率的权重,在本评估中,精准率和召回率拥有相等的权重,即β=1,也被称为 *F*1-*measure*.

研究者们在评估时也会指定推荐列表长度 *K*,一般取 1,3 和 10.这 3 个指标都是为了评估智能代码补全系统对于一个指定查询的表现.为了得到系统在多个代码补全任务上的整体性能,研究者们会计算所有补全得到的平均值.

补全时间:指代码补全方法给出推荐列表的时间,一般计算测试数据上的平均补全时间.Jin 和 Servant 等人的实证研究指出[26]:补全时间无法表明开发人员在实际使用中的时间开销,应以开发人员接受补全作为结束,以持续时间作为代码补全方法实用性的衡量指标之一.

对于上述的单一指标,有研究者为补全任务提出一个结合前缀长度、排名和补全时间的综合评估指标[19].对于每一个查询的输入前缀,*i* 代表前缀长度,等级 G_i 的计算方法为

$$G_i=\frac{\sum_{j=1}^{10}\frac{results(i,j)}{j}}{attempts(i)} \tag{6}$$

其中,*results*(*i*,*j*)表示对于前缀长度 *i* 在排名 *j* 处的正确匹配次数,*attempts*(*i*)表示对前缀长度 *i* 进行评估的时间.

3 代码补全的代码表征方式

对代码的表征方式,决定了代码的表征方式和特定代码特征的抽取,以及待补全代码和代码语料库的处理方式,和上下文之间相似度的比较方式,从而影响补全性能.本文根据智能代码补全方法的一般过程,从代码表征的视角对现有的智能代码补全方法进行分类.根据对代码信息的利用和特征提取的不同形式,智能代码补全方法可以分为基于编程语言表征和基于统计语言表征两个方向.在本文中,编程语言指高级语言,基于编程语言表征指对高级语言的文本进行建模的研究方向;统计语言指可以利用统计方法处理的自然语言[85],基于统计语言表征指使用统计语言模型处理代码文本的研究方向.本文将基于统计语言表征的补全方法与基于编程语言表征的补全方法分开阐述,并将基于编程语言表征的补全方法分为标识符序列、抽象语法树和控制/数据流图这 3 个类别进行阐述;基于统计语言表征的补全方法分为基于 *N*-gram 模型和基于神经网络模型这 2 个类别进行阐述.

本文深入调研了代码补全方法的发展历程,发现对代码表征的程度从浅显到深入,提取到的结构和语义信息从单一到多元.本文根据不同的代码表征方式,将总结不同研究思路及各自研究成果,并对部分基础模型提供

了理论阐述,指出每个方向的优缺点,为未来的研究提供参考.本文将各个研究方向的优缺点总结在表 1 中.

Table 1 Summary of different intelligent code completion approaches

表 1 不同智能代码补全方法研究方向的优缺点

研究方向	优点	缺点
标识符序列	模型简洁,补全类型多	性能一般
抽象语法树	不同编程语言通用性好	需要额外的语法树抽取
控制/数据流	结构/上下文信息丰富	模型复杂
N-gram 模型	灵活性强,可扩展性高	只能提取一定范围内信息
神经网络	长距离信息提取能力	训练时间长,存储开销大

3.1 基于编程语言表征的研究

基于编程语言表征的研究通过对代码的静态或动态分析,利用词法分析、语法分析、控制/数据流分析等技术,提取代码特征.由于对编程语言信息的提取程度不同,本文将基于编程语言表征的智能代码补全方法又分出标识符序列、抽象语法树语法和词汇控制/数据流图这 3 个层次进行详细阐述.

3.1.1 基于标识符序列表征方式的研究

基于标识符序列表征的代码补全技术将源代码表征为标识符序列或者根据序列特征提取特征矩阵,通过特征向量距离或特征矩阵的最近邻匹配进行相似度比较.

表 2 是基于标识符序列表征的智能代码补全方法的代表文献和技术特征.可以发现,关键词/缩略词补全的代码补全类型都属于标识符序列表征层次.

Table 2 Summary of token-based intelligent code completion approaches

表 2 基于词汇表征的智能代码补全方法的代表文献总结

文献	代码表征	相似比较算法	评估指标	补全方式
Little 等人[38]	自定义的三元组	自定义的解释向量	准确率和补全时间	关键词补全
Han 等人[39,40]	标识符序列	维特比算法	准确率	缩略词补全
Perelman 等人[48]	自定义的部分表达式	特征向量	排名和补全时间	标识符补全
Bruch 等人[11]	特征矩阵	最近邻匹配 BMN	准确率、召回率和 *F*1-*measure*	标识符补全
Proksch 等人[20]	特征矩阵	基于模式的贝叶斯网络	*F*1-*measure*、补全时间	标识符补全

Han 和 Miller 等人提出了一种与关键词补全[38]类似的补全形式——多缩略词补全[39,40],通过隐马尔科夫链,从语料库中学习标识符之间的转移规律,避免了单个关键词翻译带来的额外键盘开销问题.该方法的补全精确度与关键词序列的长度关系较大.

Perelman 等人[48]定义了一种部分表达式的语言,并将部分表达式作为代码补全系统的输入,返回合成后的代码片段.该文献开发了一种算法来补全部分表达式,并通过定义类型距离等特征计算评分.

Bruch 和 Monperrus 等人[11]按照手工设计的特征类型提取目标上下文的方法调用,增加结构信息,编码为特征二进制矩阵,通过分析已有代码库建立词汇表,然后利用 KNN 算法的最近邻匹配(best matching neighbor,简称 BMN)算法在已有代码库中找到与待补全代码最近似的补全片段,给出方法调用的候选.文中还指出,许多机器学习算法可以引入代码补全的研究中.该方法提出了从已有代码库学习并引入机器学习算法,在当时是一种先进的思路.不过由于词汇表的存在,对于不在词汇表中的未知词处理能力较差.

Proksch 等人[20]针对 Bruch 等人[11]利用上下文信息的缺失和评估维度的不足,扩展了结构特征信息类型集,并对 BMN 算法进行改进,提出一种基于模式的贝叶斯网络(pattern-based Bayesian network,简称 PBN)算法来检测置信度最高的方法调用,并提出一种帮助模型学习的聚类算法以减小模型规模.该方法以内存开销换取了补全速度的提升,并且集成了更多上下文信息,提高了原模型的性能.

基于标识符序列表征的智能代码补全方法将源代码表征为标识符的序列,利用了源代码的序列化特征,模型简洁且补全类型广泛,但是对于信息利用仍然欠缺,无法利用更深层次的信息.

3.1.2 基于抽象语法树表征方式的研究

基于抽象语法树表征的智能代码补全方法能够对源代码中的语法规则和结构信息进行提取和利用.基于抽象语法树的方法使用语法分析器提取语法特征,利用抽象语法树进行表示,将待补全代码视作语法树的缺失节点,将树形结构作为代码特征进行匹配.抽象语法树是一种将语法结构抽象为树形的代码表示方法.

表 3 是基于抽象语法树表征的智能代码补全方法的代表文献和技术特征.

Table 3 Summary of AST-based intelligent code completion approaches

表 3 基于抽象语法树表征的智能代码补全方法的代表文献总结

文献	代码表征	相似比较算法	评估指标	补全方式
Holmes 等人[49]	抽象语法树	启发式规则匹配	准确率	标识符补全
Robbes 等人[19]	含修改历史的抽象语法树	频繁项集	综合评估指标	标识符补全
Zhong 等人[28]	残缺的抽象语法树	特征向量	准确率	标识符补全
Bajaj 等人[50]	DOM 结构抽象语法树	动态模式匹配	准确率	标识符补全
Bielik 等人[22]	抽象语法树	自定义规则匹配	错误率	标识符补全
Raychev 等人[51]	抽象语法树	决策树生成	准确率	标识符补全

Holmes 等人[49]根据待补全代码生成结构上下文描述,与预先从语料库中提取的代码结构匹配,将最佳匹配的结构返回给开发人员参考.结构匹配方法结合了 6 种启发式方法,每种启发式方法针对不同类型的结构信息,产生不同的结果,将结果综合后返回给开发人员,并开发了 Strathcona 工具.

Robbes 和 Lanza 等人[19]提出了一个评估代码补全准确率的基准框架和一个“积极”的代码补全策略:给出的补全推荐结果中应只保留少数语义最相关的补全方案.作者记录了 7 个项目的完整修改历史,并用抽象语法树结构存储,通过重现抽象语法树的改变进行评估.尽管该研究指出了代码补全方法的一些问题,但所提出的方法是资源密集型的,且对不同项目、不同语言仍无法起到通用的评估效果.另外,数据需要用 IDE 插件经过长时间的收集.

Zhong 等人[28]在 2017 年提出了一种分析部分程序的通用框架 GRAPA,可以通过对部分程序进行补全得到完成的程序,从而输入不同的静态分析工具(比如缺陷检测等).通过 WALA 工具[52]实现了对 Java 部分程序构建抽象语法树,通过在 Java 文档定位待补全的方法名称,实现部分代码补全;并且该方法不需要完整程序,在部分程序中仍能进行未知方法调用的补全.

Bajaj 等人[50]在 2014 年针对动态交互的具有文档对象模型(DOM)结构的 JavaScript 代码进行研究,提出了 Dompletion 工具[53],可以推断现有的 DOM 结构,动态分析 JavaScript 代码,通过结合 DOM 结构的抽象语法树和动态分析技术,实现了 JavaScript 代码补全.

Bielik 等人提出了 PHOG[22]生成模型和一种高阶文法(higer order grammar),通过对抽象语法树的动态有监督学习,补全新的叶子结点.由于模型基于抽象语法树,可以对任何解析为语法树的编程语言进行训练和学习,解决了不同软件工程任务和编程语言中模型重用的关键问题.PHOG 模型效率高、计算快,并且可以动态学习和表示,对 JavaScript 代码补全任务的评估表明该模型的错误率较低.

Raychev 等人提出了 TGEN[51]模型,基于决策树学习算法,学习特定语言代码的抽象语法树,可以在动态的上下文中调整预测结果.TGEN 模型可以集成不同的决策树学习算法,该文献以 ID3 决策树算法为例,实现了 DEEP3 系统,并对 JavaScript 和 python 代码补全进行了实证评估,分别取得了 82%和 69%的准确率.

抽象语法树是最常见的基于语法的代码表征方式之一,研究也相对较多.基于抽象语法树表征的代码补全方法可以利用语法结构特征,比标识符序列表征深入.基于抽象语法树表征的模型移植性较好,可以在其他语言的抽象语法树上进行补全.代码特征信息的增多,也造成抽象语法树的遍历和操作往往需要较大计算开销,导致很难动态更新待补全代码的语法结构,并且在获得特定指标时需要额外工具来获得.

3.1.3 基于控制/数据流图表征方式的研究

抽象语法树虽然可以抽象出编程语言的语法结构,但是对于项目代码来说,特殊意义的变量命名反映代码功能的信息被丢失,基于控制/数据流图表征的智能代码补全方法在抽象语法树的基础上,结合了代码的部分控

制/数据流信息,在代码补全任务中得以应用.表 4 是基于控制/数据流图表征的智能代码补全方法的代表文献和技术特征.

Table 4 Summary of control/data-flow-graph-based intelligence code completion approaches

表 4 基于控制/数据流图表征的智能代码补全方法的代表文献总结

文献	代码表征	相似比较算法	评估指标	补全方式
Nguyen 等人[24]	抽象语法树加上 API 调用序列	特征向量	准确率	代码片段补全
Li 等人[13]	抽象语法树加上数据流	特征向量	准确率	标识符补全

Nguyen 等人提出了 GraPacc[24]方法,为抽象语法树添加额外信息,生成一种基于图的表示方式,包括结构信息和 API 调用关系,通过外部代码库学习 API 使用模式以创建 API 使用数据库,并对比图之间的相似性,给出最相似的补全推荐.该方法一次可以推荐多条语句的补全结果,但是由于基于外部代码库,对于自定义方法的补全较差.

Li 和 King[13]通过对程序依赖图(PDG)进行修改和串联,利用全局递归神经网络和局部的注意力机制,实现了从词汇表预测生成,又实现了局部上下文的复制,更符合补全逻辑,并且能够补全词汇表中不包含的未知词.该文献使用神经网络模型捕获长距离信息,但代码表示是基于控制/数据流图的改进结构,仍归为语义表征的方法.该方法主要针对不在词汇表的未知词进行特别优化,对于项目中的自定义方法调用表现较好.

基于控制/数据流图的表征方式充分利用编程语言和源代码中的语义信息,但是实现难度也更大,尚没有能够完全提取源代码中的语义信息的表征方式,已有的研究往往是选取一定程度的语义信息进行建模.由于语义信息复杂,计算开销大,基于控制/数据流图的智能代码补全方法无法全部利用,性能上并不一定优于基于语法的智能代码补全方法.

3.2 基于统计语言表征的研究

不同于基于编程语言表征的研究尝试去明确地提取代码中的结构和语义信息,基于统计语言表征的研究利用自然语言中的统计语言模型对代码进行表征.2010 年,通过对 SourceForge 上 6 000 多个项目中包含的 4 亿多行代码的分析,Gabel 等人[54]发现软件中存在的句法冗余(syntactic redundancy),例如打印语句“System.out.println(…)”在许多源文件中频繁出现.2012 年,Hindle 等人[21]受到统计语言模型在语音识别[55]、拼写纠错[56]、手写识别[57]和机器翻译[58,59]等方面的成果应用的启发,提出编程语言虽然在理论上十分复杂,但是由人编写的“自然”程序,大多也是简单而重复的,因此可以用统计语言模型去捕获其中可预测的统计特征,从而用于软件工程任务.该文献通过对语料库建立 *N*-gram 语言模型[60],通过标准交叉熵的计算和一系列横向纵向的比较,证明该 *N*-gram 模型确实提取到了深层次的统计规律,并开发了一个代码补全的工具,显著提升了 Eclipse 自带的补全插件的性能.

本节将对基于统计语言表征的智能代码补全研究进行总结.下面先简要介绍相关概念和理论.

3.2.1 *N*-gram 模型的相关概念

(1) 统计语言模型(statistical language model)

一个语言模型为句子 s 构建一个概率分布 $P(s)$,$P(s)$代表 s 作为一个句子在训练语料中出现的概率[85].如果训练语料是源代码,则概率$P(s)$表示句子 s 是否是源代码语句的概率.对于一个由 n 个词组成的句子 $s=\omega_1\omega_2\dots\omega_n$,则 $P(s)=P(\omega_1\omega_2\dots\omega_n)$,由条件概率公式:

$$P(s)=P(\omega_1,\omega_2,\dots,\omega_n)=P(\omega_1)P(\omega_2|\omega_1)P(\omega_3|\omega_1\omega_2)\dots P(\omega_n|\omega_1\dots\omega_{n-1}) \tag{3.1}$$

依据统计语言模型上述定义,问题归结为计算公式(3.1)等号右边各项的值.公式(3.1)看似简单,但存在两个问题:首先是自由参数过多,假设上面句子中的所有单词都来自大小为 V 的词典,计算其中所有的条件概率 $P(\omega|*)$,则模型的自由参数数量为 V^n,如果句子长度的增加,自由参数量级将会呈指数增加,实际情况下不可能计算出所有的参数;其次是数据稀疏性,也叫数据匮乏,每一个ω都有 V 种取值,构造出了许多在实际中不会出现的词对,但如果按照最大似然估计的方法,最后得到的概率可能为 0.研究者们使用 *N*-gram 模型解决自由参数过多

的问题,并通过平滑方法解决数据稀疏性问题.

(2) *N*-gram 模型

研究者们假设语料库中每个词出现的概率只与前面的 *N*–1 个词有关,与其他词无关,该假设称为马尔可夫假设(Markov property),则有:

$$P(\omega_i|\omega_1,\omega_2,\ldots,\omega_{i-1})=P(\omega_i|\omega_{i-N+1},\omega_{i-N+2},\ldots,\omega_{i-1}) \tag{3.2}$$

根据定义,*N* 取值为 3 时,语料库中每个词的出现仅与之前的 2 个词有关,称 3-gram 模型,自由参数量级为 V^3.理论上来说,*N* 的值越大,表明考虑了前面更多的词,结果更加精确.但是随着相关词数量的增加,模型减少自由参数的效果越弱,所以 3-gram 语言模型是最常见的使用形式.

(3) 平滑方法

研究者们又通过对语料库中的序列概率重新分配来解决数据稀疏,这种做法叫做数据平滑(smoothing).最简单的平滑方法是加一平滑(即拉普拉斯平滑),默认每一个词组都出现 1 次,这样就避免了某一词组没有出现,导致频次和涉及到的词对条件概率为 0 的情况,可以有效地缓解数据稀疏性.平滑方法还有古德-图灵平滑 Witten-Bell 平滑等算法.

(4) 语言模型的评估指标

研究者们使用测试数据上的性能指标如精度、召回率等评价算法的优劣.但评价语言模型的好坏,需要用到信息论中熵的概念.交叉熵是衡量模型的估计结果与实际情况的差异的一种指标,它的值越小,说明模型与真实概率分布之间的偏差越小,也就是模型越好.对于一个语言模型 *M* 来说,它对句子 $s=\omega_1\omega_2\ldots\omega_n$ 出现的概率估计为 $P_M(s)$,交叉熵的计算公式如公式(3.3)所示,困惑度(perplexity)是交叉熵的指数变换:

$$H_M(s)=-\frac{1}{n}\log P_M(\omega_1\omega_2\ldots\omega_n) \tag{3.3}$$

又由公式(3.1)和公式(3.2)可得:

$$H_M(s)=-\frac{1}{n}\sum_1^n\log P_M(\omega_i\mid\omega_1\ldots\omega_{i-1}) \tag{3.4}$$

3.2.2 基于 *N*-gram 模型表征方式的研究

基于 *N*-gram 模型表征方式的智能代码补全方法采用 *N*-gram 语言模型作为基础,主要研究内容是改进 *N*-gram 语言模型(如添加缓存组件、优化平滑方法等)以利用代码的局部重复性和添加额外代码上下文信息(如结合语义注释和代码更改记录)以利用代码的结构信息.表 5 总结了基于 *N*-gram 模型表征的智能代码补全方法的代表文献和技术特征.

Table 5 Summary of *N*-gram-model-based intelligence code completion approaches

表 5 基于 *N*-gram 模型表征的智能代码补全方法的代表文献总结

文献	统计模型	评估指标	补全方式
Hindle 等人[21]	*N*-gram 模型	准确率	标识符补全
Raychev 等人[14]	*N*-gram 模型、RNN	准确率	代码片段补全
Nguyen 等人[30]	结合语义信息的 *N*-gram 模型	准确率	标识符补全
Tu 等人[27]	“缓存”*N*-gram 模型	*MRR* 和准确率	标识符补全
Franks 等人[61]	“缓存”*N*-gram 模型	*MRR*	标识符补全
Nguyen 等人[62]	结合结构信息的 *N*-gram 模型	准确率	API 方法调用补全
Roos 等人[12]	基于定向搜索的 *N*-gram 模型	精确率	API 方法调用补全
Nguyen 等人[43]	结合细粒度代码更改的 *N*-gram 模型	准确率	API 方法调用补全

Allamanis 等人[63]在 Hindle 等人的研究[21]基础上扩大了语料库规模,在超过 10 亿行代码上训练了一个 *N*-gram 模型.该实证研究表明:随着数据量增加,*N*-gram 模型性能逐渐提升,不存在性能瓶颈.

Nguyen 等人针对 *N*-gram 模型只能提取 *N* 元范围内局部的规律的问题,提出了 SLAMC[30],在代码标识符上增加数据类别(如 string)和角色(如 variable)的语义注释,对这样的语义单位(sememe)而非原始 *N*-gram 中使用的词汇单位(lexeme)进行建模,并且结合主题模型和 *N*-gram 模型来捕获全局信息.Nguyen 等人的研究指明了

N-gram模型的不足之处,即没有考虑编程语言的语义特性和局部性特性,后续关于*N*-gram的研究也基本上在这两个方面进行完善与改进.

Tu等人[27]同样发现代码存在的局部性规则在*N*-gram模型中没有被利用的问题,类似Hindle证明代码中的自然性(naturalness),同样通过提出假设和实验验证的方法,证明了源代码具有局部重复性(localness),可以通过局部缓存捕获局部重复规律并应用在软件工程任务中,提出了在*N*-gram模型中增添“缓存”组件,相对于Nguyen等人的语义模型更加简单和通用,对训练语料库的规模要求也更小.尽管该方法为如何处理局部重复性提出了解决方法,但是对于编程语言特有的结构信息没有加以利用.

Raychev等人提出了SLANG[14]工具,在文献[21]的方向上进行更深入的研究,利用Github[41]进行训练模型,对比了*N*-gram、循环神经网络在预测API调用序列中的应用,并展示了如何将统计语言模型如何与经典编程语言概念(如别名分析)结合得到了最优的效果,指出了未来统计方法在代码补全中发展的方向.该方法

Franks等人开发了名为CACHECA[61]的Eclipse插件,将Eclipse默认补全结果与基于缓存的*N*-gram模型[27]补全结果结合起来,发现组合方法的性能相对于默认插件提升了三分之一,证实了局部重复规律对于代码补全方法性能有显著影响的.

Nguyen等人针对*N*-gram模型在API调用补全中无法提取结构信息,并且API调用序列存在独特的使用模式,比如顺序无关、上下文距离大等问题,提出了一个针对API调用序列图的语言模型GraLan[62],从语料库中学习和计算特定子图的出现概率,用以预测下一个出现的API元素.

Bettenburg等人[42]提出对于统计语言模型,从开源代码仓库中获取的数据集存在极大可变性,对模型质量有不利影响,建议将数据集进行聚类和整合以解决数据的可变性.通过对软件工程领域数据集的案例研究表明:使用局部数据集构建的模型在模型拟合和预测性能方面要优于传统模型,但是聚类算法及参数选择会对模型质量造成重大影响.

Roos同样使用*N*-gram模型表征代码[12],结合了Witten-Bell Backoff等4种平滑方法和一个基于定向搜索的剪枝算法,能够快速有效地完成API补全任务,实现毫秒级的补全速度和Top3中89%的准确度.但该方法同样受数据集规模和选取限制,且在不同语言之间进行切换时需要重新训练,迭代能力差无法适新更新的库资源.

Nguyen等人针对API调用补全存在不同开源项目存在特定的API使用模式,统计语言模型从全部项目中学习存在噪音的问题,提出APIREC[43],利用细粒度代码更改的重复性[64],构建细粒度代码更改语料库并学习API出现概率,并结合了一个基于关联关系的推理模型捕获局部经常发生修改的部分.通过实证评估表明:该工具用较少的训练集也可以获得良好的性能;并表明,个性化的代码补全模型也可以通过开源仓库进行训练,通过开发人员自己的代码修改历史进一步完善.

基于统计语言模型表征的智能代码补全方法通过对*N*-gram模型的不断改进以优化方法性能,但是仍然无法从根本上解决*N*-gram模型自身的局限性,即无法提取长距离的关系,研究者们亟需新的模型突破*N*-gram模型的瓶颈.

3.2.3 神经网络模型的相关概念

循环神经网络(recurrent neural network,简称RNN)是Bengio在2001年提出[46]利用三层神经网络构建*N*-gram模型,通过词的分布式表达来解决数据稀疏性对统计语言模型的影响,优势在于自带平滑,通过随机梯度下降法优化后就得到了各项参数和完整模型,不用像传统模型一样对平滑算法进行选择.Bengio使用的语言模型仍采用前*N*–1个单词作为输入,对距离较远单词之间的关系无法提取.鉴于此,Mikolov将RNN结构应用于统计语言模型[47],如图8所示.

循环神经网络会记录上一个隐藏层的输出,联合当前层输入计算隐藏层,给出当前层输出.隐藏层不断在计算中被使用和更新,使得每一层的输出都是由之前的序列和当前的输入共同决定的.这样的循环结构使循环神经网络不再仅从定长的前序序列中提取信息,而是真正充分利用所有的上文信息进行预测.但RNN在实际应用中存在梯度消失(vanishing gradient)问题[65],需要进一步进行优化,现有的研究方向有优化训练算法[66]和利用LSTM(long short-term memory)和门控神经元(gating neurons)改进网络[67].

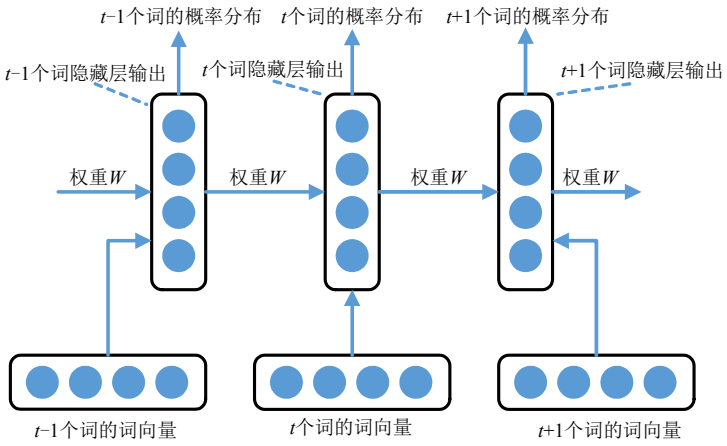

Fig.8 Stucture of recurrent neural network

图 8 循环神经网络结构

3.2.4 基于神经网络模型表征方式的研究

基于神经网络模型表征方式的智能代码补全方法主要采用改进的 RNN 模型,包括指针网络、LSTM 和门控网络等.表 6 是基于神经网络模型表征的智能代码补全方法的代表文献和技术特征.

Table 6 Summary of neural-network-based intelligence code completion approaches

表 6 基于神经网络模型表征的智能代码补全方法的代表文献总结

文献	统计模型	评估指标	补全方式
Bhoopchand[34]	指针网络	准确率	标识符补全
Hellendoorn 等人[68]	优化的“缓存”*N*-gram 模型、LSTM+RNN	*MRR* 和准确率	标识符补全
Allamanis 等人[69]	门控图神经网络	*F*1-*measure* 和准确率	变量名补全

Bhoopchand 等人[34]提出一种新的神经网络模型(sparse pointer network),用以捕获长距离的依赖关系.该文献针对 python 语言补全构建了大规模语料库,发现传统的神经网络模型在补全长距离的标识符和自定义的标识符上的表现一般,而通过维护一个指向全局词汇表的指针结构,结合自定义标识符的指针概率分布,在自定义标识符预测的准确率上获得了 25%的显著提升.不过,该方法需要维护一个全局词汇表,没有考虑词汇表外的标识符补全情况.

Hellendoorn 等人[68]对前人在 *N*-gram 模型上的优化[61,62]加以整合,基于依赖模型捕获长距离上下文信息,利用缓存组件记录局部修改,并且对比了不同平滑方法,与使用 LSTM 的 RNN 网络技术的深度学习模型对比后发现,精心优化后的 *N*-gram 模型无论是性能还是速度都要优于深度学习模型,Jelinek-Mercer 平滑方法要优于其他方法.

Allamanis 等人[69]提出用图来表示源代码的结构和语义信息,指出图神经网络在变量补全和变量误用上的性能要优于卷积神经网络,并且可以实现多个变量的补全.但是该方法仅在变量名补全上进行应用,尚没有推广到标识符补全.

研究者们在代码元素的重复性和局部性进行了许多研究,大多将统计语言模型应用在 API 方法调用补全上,有工作[70]表明,API 使用相较于一般代码更具有重复性和可预测性,解释了统计语言模型在 API 推荐的成功.基于统计语言的表征方式欠缺之处就是对代码特征表征不够显式,如果与代码特征进一步结合,或许可以获得进一步的性能提升.Rahman 等人最新的实证研究[71]指出,图形具有比 *N*-gram 模型更高级别的重复模式,建议对统计代码图模型进行进一步研究以准确捕获更复杂编码模型.

4 智能代码补全方法的主要挑战

关于智能代码补全方法的研究在 10 余年中已经取得了一定的成果,并且在实际开发中得到应用和实践,但仍然面临以下挑战.

(1) 缺乏统一的模型性能评估指标:在上文中对现有的智能代码补全方法的评估指标进行了整理,最常用的是机器学习领域使用的准确率、召回率和信息检索领域使用的平均倒数排名,但是大部分文献都只采用其中一个指标进行性能评估,采用不同评估指标的模型难以互相换算性能指标,也难以进行性能比较;

(2) 缺乏能够有效衡量补全性能的真实基准语料库和数据集:性能评估指标很大程度上取决于语料库和测试集,目前智能代码补全方法采用的语料库大多是研究者在开源代码仓库中用爬虫抓取的个人数据集,也有少部分研究者公开的数据集[34,51].有实证研究[68]指出:随机移除标识符进行补全的评估方式与真实场景的代码补全操作差异较大,合成和模拟的评估基准无法准确反映智能代码补全方法的性能.该研究还对真实的补全操作进行了分析,发现项目内部的标识符的补全是更常见的补全场景,但是已有的代码补全方法在此场景上的表现并不如文献中宣称的那么好.智能代码补全方法需要一个基于真实开发场景的基准语料库、数据集和统一的性能评估指标才能准确地衡量模型的先进性与实用性;

(3) 智能代码补全方法在代码表征和模型构建上有不同方向且各有所短:实证研究[37]对比了抽象语法树表征的最近邻匹配(BMN)算法[11,20]、RNN 模型[29,68]和 *N*-gram 模型[68],发现神经网络模型在核心方法调用优于 *N*-gram 模型,但在第三方库调用上稍显逊色;BMN 算法虽然整体性能较差,但是得益于结构信息和类型信息,在第三方库调用中优于精心设计的 RNN 模型.该实证研究说明,不同的代码表征和模型构建方法在不同的使用场景下存在性能差异.所以尚不能定论哪种技术是更好的智能代码补全模型,代码补全的表征方法和模型构建仍需要研究者进一步探索.

5 智能代码补全方法的未来方向

智能代码补全方法的研究工作面临上节提到的一些挑战,为了应对这些挑战,本节从场景需求和模型构建角度讨论了智能代码补全方法未来可能的研究方向.

(1) 需要进一步挖掘真实补全场景:研究[37]表明:不仅在真实场景下的基准数据和人工合成的基准数据之间存在较大差异,同一种待补全标识符也有着不同的补全场景,影响了代码补全的效率和准确度.例如,方法调用是最普遍的待补全标识符之一,但是代码补全方法在同一个项目的自定义方法调用补全逊色于第三方调用补全.这表明当前的代码补全方法无法学习项目内部信息,无法对软件项目进行定制化开发.在未来的研究中,研究者们需要更多的真实场景下的基准数据,同时应当更关注补全缓慢和补全失败的情况;

(2) 模型的复杂不能代表性能的优越,模型构建需要针对代码补全任务的特殊性进一步研究:代码补全方法正在从单标识符补全转变为代码块的多标识符补全,随着卷积神经网络[72−74]在图像处理[75,76]、自然语言处理[77,78]、语音识别[79,80]等任务中表现出了卓越性能,将机器学习领域的其他先进技术,如神经网络、深度学习(deep learning)[81,82]应用到智能代码补全方法中已经成为当前研究工作的一个重要方向.Liu 等人的研究[83]表明,神经网络模型并不是在所有任务中都能取得比传统模型更好的性能.有实证研究[68]表明,精心设计的 *N*-gram 代码补全方法并不逊色于当时最先进的神经网络模型方法.也有实证研究[71]表明,统计代码图模型相对于 *N*-gram 模型具有更好的代表性和更高级别的重复模式.这些实证研究表明:尽管神经网络模型具有优秀的长期记忆能力,但是研究者们不应盲目追求模型的复杂和高端,未来的智能代码补全方法需要针对代码补全任务的特殊性进行大量创新,为代码补全方法定制更深入的表征方式,以便适应更复杂的实践需求.

6 总 结

智能代码补全作为开发人员在集成开发环境中常用辅助工具,对软件开发效率有着极为重要的影响.学术界对代码补全方法的研究有超过 10 年的历史,尚没有综述类文章总结代码补全方法研究进展和成果.本文从代码表征方式的角度梳理归纳了智能代码补全研究的 2 种研究思路,总结讨论了当前智能代码补全方法面临的关键问题及未来发展趋势.主要工作总结如下:(1) 本文从代码表征方式的角度出发,对现有的智能代码补全方法进行了归纳总结,将智能代码补全方法的研究思路分为基于编程语言表征和基于统计语言表征两大类;(2) 本文介绍了在智能代码补全方法中常用的实验验证方法,并总结了代码补全评估指标和计算公式;(3) 本文指出了目前智能代码补全方法研究的仍面临的主要挑战,并围绕挑战展望了智能代码补全方法的未来发展和研究方向.

References:

[1] Lo D, Xia X. Fusion fault localizers. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2014. 127−138.

[2] Xia X, Lo D, Pan SJ, Nagappan N, Wang X. Hydra: Massively compositional model for cross-project defect prediction. IEEE Trans. on Software Engineering, 2016,42(10):977−998.

[3] Le Goues C, Nguyen T, Forrest S, Weimer W. Genprog: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2011,38(1):54−72.

[4] Xiong Y, Liu X, Zeng M, Zhang L, Huang G. Identifying patch correctness in test-based program repair. In: Proc. of the 40th Int'l Conf. on Software Engineering. ACM, 2018. 789−799.

[5] Bellamy B, Avgustinov P, De Moor O, Sereni D. Efficient local type inference. ACM SIGPLAN Notices, 2008,43:475−492.

[6] Pierce BC, Turner DN. Local type inference. ACM Trans. on Programming Languages and Systems (TOPLAS), 2000,22(1):1−44.

[7] Huang Q, Xia X, Xing Z, Lo D, Wang X. API method recommendation without worrying about the task-API knowledge gap. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2018. 293−304.

[8] Nguyen P, Di Rocco J, Ruscio D, Ochoa L, Degueule T, Di Penta M. Focus: A recommender system for mining api function calls and usage patterns. In: Proc. of the 41st ACM/IEEE Int'l Conf. on Software Engineering (ICSE). 2019.

[9] Hill R, Rideout J. Automatic method completion. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering. IEEE Computer Society, 2004. 228−235.

[10] Asaduzzaman M, Roy CK, Schneider KA, Hou D. Cscc: Simple, efficient, context sensitive code completion. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 71−80.

[11] Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2009. 213−222.

[12] Roos P. Fast and precise statistical code completion. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol.2. IEEE, 2015. 757−759.

[13] Li J, Wang Y, Lyu MR, King I. Code completion with neural attention and pointer networks. arXiv preprint arXiv:1711.09573, 2017.

[14] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. ACM SIGPLAN Notices, 2014,49:419−428.

[15] Murphy GC, Kersten M, Findlater L. How are Java software developers using the elipse ide? IEEE Software, 2006,23(4):76−83.

[16] Gorin RE. SPELL: A spelling checking and correction program. Online Documentation for the DEC-10 Computer, 1971. 147−160.

[17] Pletcher DM, Hou D. BCC: Enhancing code completion for better API usability. In: Proc. of the 2009 IEEE Int'l Conf. on Software Maintenance. IEEE, 2009. 393−394.

[18] Hou D, Pletcher DM. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In: Proc. of the 2011 27th IEEE Int'l Conf. on Software Maintenance (ICSM). IEEE, 2011. 233−242.

[19] Robbes R, Lanza M. How program history can improve code completion. In: Proc. of the 2008 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE Computer Society, 2008. 317−326.

[20] Proksch S, Lerch J, Mezini M. Intelligent code completion with Bayesian networks. ACM Trans. on Software Engineering and Methodology (TOSEM), 2015,25(1):3.

[21] Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. In: Proc. of the 2012 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 837−847.

[22] Bielik P, Raychev V, Vechev M. PHOG: Probabilistic model for code. In: Proc. of the Int'l Conf. on Machine Learning. 2016. 2933−2942.

[23] Lee YY, Harwell S, Khurshid S, Marinov D. Temporal code completion and navigation. In: Proc. of the 2013 Int'l Conf. on Software Engineering. IEEE, 2013. 1181−1184.

[24] Nguyen AT, Nguyen HA, Nguyen TT, Nguyen TN. GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool. In: Proc. of the 2012 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 1407−1410.

[25] Omori T, Kuwabara H, Maruyama K. A study on repetitiveness of code completion operations. In: Proc. of the 2012 28th IEEE Int'l Conf. on Software Maintenance (ICSM). IEEE, 2012. 584−587.

[26] Jin X, Servant F. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In: Proc. of the 2018 IEEE/ACM 15th Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2018. 70−73.

[27] Tu Z, Su Z, Devanbu P. On the localness of software. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2014. 269−280.

[28] Zhong H, Wang X. Boosting complete-code tool for partial program. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2017. 671−681.

[29] White M, Vendome C, Linares-Vásquez M, Poshyvanyk D. Toward deep learning software repositories. In: Proc. of the 12th Working Conf. on Mining Software Repositories. IEEE, 2015. 334−345.

[30] Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code. In: Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013. 532−542.

[31] Amorim LE de S, Erdweg S, Wachsmuth G, Visser E. Principled syntactic code completion using placeholders. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Software Language Engineering. ACM, 2016. 163−175.

[32] Hou D, Pletcher DM. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In: Proc. of the 2nd Int'l Workshop on Recommendation Systems for Software Engineering. ACM, 2010. 26−30.

[33] Jacobellis J, Meng N, Kim M. Cookbook: In situ code completion using edit recipes learned from examples. In: Companion Proc. of the 36th Int'l Conf. on Software Engineering. ACM, 2014. 584−587.

[34] Bhoopchand A, Rocktäschel T, Barr E, Riedel S. Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307, 2016.

[35] Nguyen TT, Pham HV, Vu PM, Nguyen TT. Recommending API usages for mobile apps with hidden Markov model. In: Proc. of the 2015 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2015. 795−800.

[36] Gvero T, Kuncak V, Kuraj I, Piskac R. Complete completion using types and weights. ACM SIGPLAN Notices, 2013,48:27−38.

[37] Hellendoorn VJ, Proksch S, Gall HC, Bacchelli A. When code completion fails: A case study on real-world completions. In: Proc. of the 41st Int'l Conf. on Software Engineering. Piscataway: IEEE, 2019. 960−970.

[38] Little G, Miller RC. Keyword programming in Java. Automated Software Engineering, 2009,16(1):37.

[39] Han S, Wallace DR, Miller RC. Code completion from abbreviated input. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2009. 332−343.

[40] Han S, Wallace DR, Miller RC. Code completion of multiple keywords from abbreviated input. Automated Software Engineering, 2011,18(3-4):363−398.

[41] Https://github.com/. 2019. https://github.com/

[42] Bettenburg N, Nagappan M, Hassan AE. Towards improving statistical modeling of software engineering data: Think locally, act globally! Empirical Software Engineering, 2015,20(2):294−335.

[43] Nguyen AT, Hilton M, Codoban M, Nguyen HA, Mast L, Rademacher E, Nguyen TN, Dig D. API code recommendation using statistical learning from fine-grained changes. In: Proc. of the 2016 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2016. 511−522.

[44] Arora C, Sabetzadeh M, Briand L, Zimmer F. Automated checking of conformance to requirements templates using natural language processing. IEEE Trans. on Software Engineering, 2015,41(10):944−968.

[45] Falessi D, Cantone G, Canfora G. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. IEEE Trans. on Software Engineering, 2011,39(1):18−44.

[46] Bengio Y, Ducharme R, Vincent P, Jauvin C. A neural probabilistic language model. Journal of Machine Learning Research, 2003, 3(Feb):1137−1155.

[47] Mikolov T, Karafiát M, Burget L, Černockỳ J, Khudanpur S. Recurrent neural network based language model. In: Proc. of the 11th Annual Conf. of the Int'l Speech Communication Association. 2010.

[48] Perelman D, Gulwani S, Ball T, Grossman D. Type-Directed completion of partial expressions. ACM SIGPLAN Notices, 2012,47: 275−286.

[49] Holmes R, Murphy GC. Using structural context to recommend source code examples. In: Proc. of the 27th Int'l Conf. on Software Engineering (ICSE 2005). IEEE, 2005. 117−125.

[50] Bajaj K, Pattabiraman K, Mesbah A. Dompletion: Dom-aware Javascript code completion. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2014. 43−54.

[51] Raychev V, Bielik P, Vechev M. Probabilistic model for code with decision trees. ACM SIGPLAN Notices, 2016,51:731−747.

[52] Http://wala.sourceforge.net/. 2019. http://wala.sourceforge.net/wiki/index.php/Main_Page

[53] Https://github.com/saltlab/dompletion. 2019. https://github.com/saltlab/dompletion

[54] Gabel M, Su Z. A study of the uniqueness of source code. In: Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2010. 147−156.

[55] Rabiner LR. A tutorial on hidden Markov models and selected applications in speech recognition. Proc. of the IEEE, 1989,77(2): 257−286.

[56] Brill E, Moore RC. An improved error model for noisy channel spelling correction. In: Proc. of the 38th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, 2000. 286−293.

[57] Hinton GE, Revow M, Dayan P. Recognizing handwritten digits using mixtures of linear models. In: Proc. of the Advances in Neural Information Processing Systems. 1995. 1015−1022.

[58] Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, 2002. 311−318.

[59] Brown PF, Pietra VJD, Pietra SAD, Mercer RL. The mathematics of statistical machine translation: Parameter estimation. Computational Linguistics, 1993,19(2):263−311.

[60] Bahl LR, Jelinek F, Mercer RL. A maximum likelihood approach to continuous speech recognition. IEEE Trans. on Pattern Analysis & Machine Intelligence, 1983,??(2):179−190.

[61] Franks C, Tu Z, Devanbu P, Hellendoorn V. Cacheca: A cache language model based code suggestion tool. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol.2. IEEE, 2015. 705−708.

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

[63] Allamanis M, Sutton C. Mining source code repositories at massive scale using language modeling. In: Proc. of the 10th Working Conf. on Mining Software Repositories. IEEE, 2013. 207−216.

[64] 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. ACM, 2014. 803−813.

[65] Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Trans. on Neural Networks, 1994,5(2):157−166.

[66] Martens J, Sutskever I. Learning recurrent neural networks with hessian-free optimization. In: Proc. of the 28th Int'l Conf. on Machine Learning (ICML 2011). Citeseer, 2011. 1033−1040.

[67] Sundermeyer M, Schlüter R, Ney H. LSTM neural networks for language modeling. In: Proc. of the 13th Annual Conf. of the Int'l Speech Communication Association. 2012.

[68] Hellendoorn VJ, Devanbu P. Are deep neural networks the best choice for modeling source code? In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 763−773.

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

[70] Gu X, Zhang H, Zhang D, Kim S. Deep API learning. In: Proc. of the 2016 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2016. 631−642.

[71] Rahman M, Palani D, Rigby PC. Natural software revisited. In: Proc. of the 41st Int'l Conf. on Software Engineering. IEEE, 2019. 37−48.

[72] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In: Proc. of the Advances in Neural Information Processing Systems. 2012. 1097−1105.

[73] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition. 2015. 1−9.

[74] He K, Gkioxari G, Dollár P, Girshick R. Mask r-cnn. In: Proc. of the IEEE Int'l Conf. on Computer Vision. 2017. 2961−2969.

[75] Ren S, He K, Girshick R, Sun J. Faster r-cnn: Towards real-time object detection with region proposal networks. In: Proc. of the Advances in Neural Information Processing Systems. 2015. 91−99.

[76] Girshick R. Fast r-cnn. In: Proc. of the IEEE Int'l Conf. on Computer Vision. 2015. 1440−1448.

[77] Collobert R, Weston J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In: Proc. of the 25th Int'l Conf. on Machine Learning. ACM, 2008. 160−167.

[78] Kim Y. Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882, 2014.

[79] Abdel-Hamid O, Mohamed A, Jiang H, Penn G. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In: Proc. of the 2012 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2012. 4277−4280.

[80] Xiong W, Wu L, Alleva F, Droppo J, Huang X, Stolcke A. The Microsoft 2017 conversational speech recognition system. In: Proc. of the 2018 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2018. 5934−5938.

[81] LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015,521(7553):436.

[82] Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. In: Proc. of the 26th Conf. on Program Comprehension. ACM, 2018. 200−210.

[83] Liu Z, Xia X, Hassan AE, Lo D, Xing Z, Wang X. Neural-machine-translation-based commit message generation: how far are we? In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2018. 373−384.

[84] Hu X, Li G, Liu F, Jin Z. 基于深度学习的程序生成与补全技术研究进展. Ruan Jian Xue Bao/Journal of Software, 2019,30(5): 1206−1223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/30/1206.htm

[85] Zong CQ. 统计自然语言处理. Beijing: 清华大学出版社, 2013 (in Chinese).

附中文参考文献:

[84] 胡星,李戈,刘芳,金芝.基于深度学习的程序生成与补全技术研究进展.软件学报,2019,30(05):1206−1223. http://www.jos.org.cn/1000-9825/30/1206.htm

[85] 宗成庆.统计自然语言处理.北京:清华大学出版社,2013.

杨博(1997－),男,江苏沭阳人,学士,主要研究领域为智能软件工程,软件仓库挖掘.

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

张能(1990－),男,博士,助理研究员,主要研究领域为软件工程,服务计算.

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