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

编程现场上下文深度感知的代码行推荐*

陶传奇[1,2,3], 包盼盼[1], 黄志球[1,2], 周 宇[1,2], 张智轶[1,2]

[1](南京航空航天大学 计算机科学与技术学院,江苏 南京 211100)

[2](高安全系统的软件开发与验证技术工信部重点实验室(南京航空航天大学),江苏 南京 211100)

[3](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 陶传奇, E-mail: taochuanqi@nuaa.edu.cn

摘 要: 在使用程序设计语言编写程序时,代码行之间通常存在一定的上下文关联关系.如果能根据已有的编程现场上下文给开发人员推荐当前代码行,不仅能够帮助开发人员更好地完成开发任务,还能提高软件开发的效率.而已有的一些方法,通常是进行代码修复或者补全,又或者只是基于关键词匹配的搜索方法,很难达到推荐完整代码行的要求.近年来, 智能化软件开发引起了学术和工业界的广泛关注.针对上述问题,一种可行的解决方案是基于已有的海量源码数据,利用深度学习析取代码行的相关上下文因子,挖掘隐含上下文信息,为精准推荐提供基础.因此,本文提出一种基于深度学习的编程现场上下文深度感知的代码行推荐方法,能够在已有的大规模代码数据集中学习上下文之间潜在的关联关系,利用编程现场已有的源码数据和任务数据得到当前可能的代码行,并推荐 Top-N 给编程人员.代码行深度感知使用 RNN Encoder-Decoder,该框架能够将编程现场已有的若干行上文代码行进行编码,得到一个包含已有代码行上下文信息的向量,然后根据该向量进行解码,得到预测的 Top-N 代码行输出.我们利用在开源平台上收集的大规模代码行数据集,对方法进行实验并测试,结果显示,本方法能够根据已有的上下文推荐相关的代码行给开发人员,Top-10 的推荐准确率有 60%左右,并且 MRR 值在 0.3 左右,表示用户满意的推荐项排在 N 个推荐结果中比较靠前的位置,方法在两个衡量指标上表现较好.

关键词: 编程现场;上下文;代码行;深度学习;RNN Encoder-Decoder

中图法分类号: TP311

中文引用格式: 陶传奇,包盼盼,黄志球,周宇,张智轶. 编程现场上下文深度感知的代码行推荐.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Tao CQ, Bao PP, Zhou Y, Zhang ZY, Huang ZQ. Code Line Recommendation Based on Deep Context-Awareness of Programming Scene. Ruan Jian Xue Bao/Journal of Software, 2018 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Code Line Recommendation Based on Deep Context-Awareness of Programming Scene

TAO Chuan-Qi[1,2,3], BAO Pan-Pan[1], HUANG Zhi-Qiu[1,2], ZHOU Yu[1,2], ZHANG Zhi-Yi[1,2]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211100, China)

[2](Ministry Key Laboratory for Safety-Critical Software Development and Verifivation (Nanjing University of Aeronautics and Astronautics), Nanjing 211100, China)

[3](National Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

* 基金项目: 国家重点研发计划项目(2018YFB1003900); 国家自然科学基金(61602267, 61402229); 计算机软件新技术国家重点实验室基金(KFKT2018B19)

Foundation item: National Key R&D Program of China(2018YFB1003900); National Natural Science Foundation of China (61602267，61402229); The Open Fund of the State Key Laboratory for Novel Software Technology (KFKT2018B19)

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: With the advance of intelligent learning techniques applied in diverse fields in software engineering, intelligent software development attracts increasing global attention. For various programming codes, there usually exist contextual relationships explicitly or implicitly among code lines. If the next code line or lines can be recommended to program developers according to the existing code lines, then it will not only help the developer to complete the development task better, but also improve the efficiency of software development. However, most existing approaches only focus on code repair or completion, which seldom considers how to meet the demand of recommending code lines based on contextual information. To solve this problem, a feasible solution is using deep learning methods to extract the relevant context factors of code lines through mining hidden context information based on the existing massive source data. Therefore, this paper proposes a novel approach based on deep learning for programming scene. In this approach, the contextual relationships among various code lines are learned from existing large-scale code data sets and then Top-N code lines are recommended to programmers. The approach utilizes the well-known RNN Encoder-Decoder framework, which can encode several lines of code to a vector with context-aware information, and then obtain new codes line based on the context vector. Finally, the approach is empirically evaluated with a large-scale code line data set collected from the open source platform. The study results show that the proposed approach can recommend the relevant code lines to developers according to the existing context, and the accuracy value is approaching to 60%. In addition, the MRR value is about 0.3, indicating that the recommended items are ranked in the top of the N recommended results.

Key words: programming scene; source code context; code line; deep learning; RNN Encoder-Decoder

由于人们对于软件的功能需求日益丰富,软件的规模越来越大,结构日益复杂.在软件开发过程中,程序开发人员很可能遇到一些软件编写困难的情况,比如某些不常见的功能如何实现.另外,巨大的需求对于程序编写的准确性和开发效率的要求日益提高.在软件开发的编程现场,有大量与当前开发任务相关的信息,如代码上下文信息、用户开发意图等.因此,在开发过程中,如果开发人员能够充分利用编程现场的已有信息,获得当前代码行的可能情况,就能进行参考、改进或直接复用,对提高程序编写的准确率和效率会有很大的帮助. 这也是智能化软件开发的重要特点.

在实际开发过程中,开发者通常会选择搜索引擎查询需要的代码[1].但是利用搜索引擎搜索通常需要确切的功能性描述[2],而对一个单一代码行而言并不具备一个完整功能.并且,由于编程语言的复杂性和多样性[3],比如数据类型、结构的多样性以及开发人员自定义变量的差异性等,导致简单的文本匹配对代码行较难有很好效果[4,5],所以查询结果通常不尽如人意.已有的一些方法通常是进行代码修复或者代码补全[6],这类工作粒度更细,并且对自动补全功能的限制性较高,主要针对确定的 API 或者已经定义的变量之类进行补全或推荐[7,8],不能实现完整的代码行推荐.

为了解决这个问题,一种可行的解决方案是基于已有的海量源码数据,利用深度学习析取代码行的相关上下文因子,挖掘隐含上下文信息,为精准推荐提供基础.受此启发,本文提出一种基于深度学习[9]的编程现场上下文深度感知的代码行推荐方法（DA4CLR）,本方法的编程现场是指软件生产中与当前编码相关的要素集合.该方法利用编程现场已有的源码数据和任务数据对当前代码行进行预测并推荐.借助深度学习模型从处理过的大规模代码行上下文数据集中学习一种隐含的上下文关联关系,作为推荐的基础.

编程现场上下文深度感知的代码行推荐方法使用 RNN Encoder-Decoder 的框架[10].该框架是一种 Sequence-to-Sequence 框架,它的编码—解码结构对解决 Sequence2Sequence 问题有独到的优势[5].编码器能够将输入序列进行编码,进而得到一个固定长度的上下文向量,该向量包含了输入序列的一些信息.而解码器能够根据编码器生成的上下文向量得到输出序列.

为了对推荐方法的有效性进行检测,我们从 GitHub[11]上关注度较高的项目和部分认可度较好的 jar 包中,收集了数百万个带代码行上下文的代码行,并选择其中的一千个作为测试数据集.经过一周左右的训练后,在准确率和 MRR 两个指标上对方法进行测试.最终,测试结果表明了编程现场上下文深度感知代码行推荐方法的可行性和有效性.

考虑到已有研究工作的困难和不足,本论文工作的主要贡献在于以下三方面:

1)针对源码提出一个统一化处理标准,消除已有开源代码数据集中代码本身的多样性和属于编程人员个人的个性化特征,实现代码行标准化、统一化形式;

2)利用深度学习模型从已有的带有代码行上下文的大规模开源数据集中学习潜在的一般性代码行上下文模式,并应用到编程现场,利用现场的上下文环境将当前可能的代码行推荐给开发人员;

3)通过编程现场任务数据捕捉开发者意图,并利用语义相似度匹配对推荐结果进行优先级调整,更好地对推荐结果进行排序,使得开发人员需要的推荐项在 n 个推荐结果中更加靠前的位置.

本文剩余部分将按以下方式展开:第 1 节描述本文用到的模型基础,即基于深度学习的自然语言处理所使用的模型和一些重要方法;第 2 节介绍本文数据来源及数据处理的一些方法;第 3 节描述我们的编程现场上下文深度感知代码行推荐方法;第 4 节对方法模型进行评估,主要根据设计的问题展示方法的实例化测试结果以及通过问卷调查的形式评估推荐结果的实用性;第 5 节主要介绍可能会对本文方法性能造成影响的一些因素;第 6 节主要介绍相关工作研究现状;文章最后对本文工作进行总结并展望未来工作.

1 模型基础

本文方法使用深度学习,主要启发性的灵感来自于深度学习在多个领域的成功应用,尤其是自然语言处理[12,13],深度学习在该领域主要应用于机器翻译和语义挖掘等方面.我们注意到,在软件开发人员眼中,编程语言和一般的自然语言的差距并不是那么明显.在编程语言(比如 Java)中也有单词(关键字、变量名)和句子(一条编程语句),并且用单词组成句子同样需要遵循特殊的语法结构.编程语言和自然语言的这些相似性意味着自然语言中成功的方法,在编程语言中可能也同样适用[14].

1.1 RNN模型

*循环神经网络(RNN, Recurrent Neural Networks)*最大的优势是用来处理序列数据[13].传统的神经网络模型从输入层到隐含层再到输出层,层与层之间是全连接的,每层之间的节点是无连接的.这种普通的神经网络对于很多问题的解决能力有限. RNN 的独特结构使得一个序列当前的输出与之前的输出相联系.如图 1 所示,具体的表现形式为网络会对前面的信息进行记忆并应用于当前输出的计算中,即隐藏层之间的节点不再是无连接的,并且隐藏层的输入不仅包括输入层的输出还包括上一时刻隐藏层的输出.理论上,RNN 能够对任何长度的序列数据进行处理.但是在实践中,为了降低复杂性往往假设当前的状态只与前面的几个状态相关.

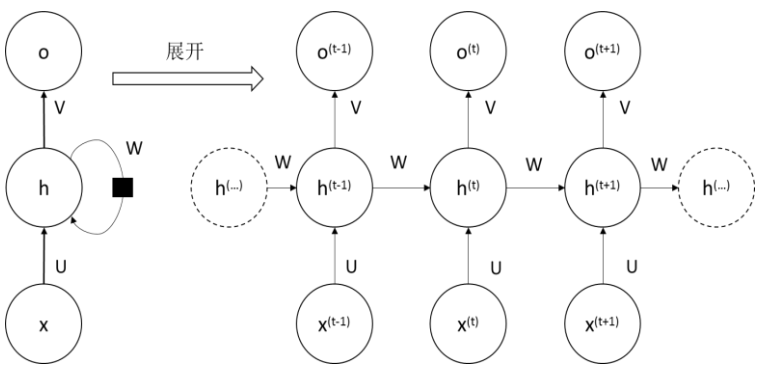

Fig.1 Recurrent neural network. The left is the RNN, drawn using the loop join. The right is the expanded structure diagram

图 1 循环神经网络结构图.左边是使用循环连接绘制的 RNN,右边是展开的结构图

实际应用中较有效的序列模型是门控 RNN[13,15],包括基于*长短期记忆(LSTM,long short-term memory)*和基于*门控循环单元(GRU,gated recurrent unit)*的网络.本文的方法采用 LSTM 结构,下面将会对 LSTM 进行详细介绍.

1.2 LSTM

循环神经网络能够成功利用已有序列信息解决实际需要,但是其实要真正使用它还需要解决一个长期依

赖问题.这是深层网络在优化阶段遇到的常见问题.因为深层的结构使得模型丧失了学习到先前信息的能力,让优化变得极其困难.

长短期记忆（LSTM,long short-term memory）网络,即 LSTM,是一种特殊的 RNN 类型,可以解决长期依赖问题,由于独特的设计结构,LSTM 适合于处理和预测时间序列中间隔和延迟非常长的重要事件,学习到长期依赖信息.LSTM 由 Hochreiter & Schmidhuber (1997) 提出,并被 Alex Graves 进行了改良和推广.在很多问题中,LSTM 都取得相当巨大的成功,并得到了广泛的使用[16].

LSTM 的特点就是在 RNN 结构以外添加了各层的阀门节点.单元彼此循环连接,类似于一般循环网络中普通的隐藏单元.阀门有 3 类:遗忘阀门（forget gate）,输入阀门（input gate）和输出阀门（output gate）.这些阀门可以打开或关闭,用于将判断模型网络的记忆状态（之前网络的状态）在该层输出的结果是否达到阈值从而加入到当前该层的计算中.所有门控单元都具有 sigmoid 非线性,而输入单元可具有任意的压缩非线性.状态单元也可以用作门控单元的额外输入.

1.3 RNN Encoder-Decoder模型

*循环神经网络编码—解码模型（RNN Encoder-Decoder Model）*是自然语言处理中比较常用的方法,通过两个 RNN 分别作为编码器和解码器,来实现对应的自然语言之间的翻译.该框架对解决 Sequence-to-Sequence 类问题有比较好的效果[16],Sequence 在这里可以理解为一个字符串序列,当给定一个字符串序列后,希望得到与之对应的另一个字符串序列.

用于映射可变长度序列到另外一个可变长度序列最简单的 RNN 架构最初是由 Cho et al.[13]提出,他将该架构称为编码—解码架构.

此架构的不足在于编码器 RNN 输出的上下文 C 的维度较小而难以适当地概括一个长序列.这种现象由 Bahdanau et al.[12]在机器翻译中观察到.因此,他们提出让 C 成为一个可变长度的序列.此外,他们还引入了将序列 C 的元素和输出序列的元素相关联的*注意力机制（attention mechanism）*.本文的方法将基于该机制.

2 数据收集和处理

2.1 源数据选取

为了进行数据收集, 我们在开源软件项目托管平台 GitHub 收集了质量比较高的 400 个开源项目[11]. 在收集过程中, 我们综合考虑 Watch, Star 和 Fork 作为选择标准,这样的综合指标比单一标准更可信. 我们在项目选取中的具体操作是:先在 GitHub 中的 Sort options 选项中按照 Fork, Star 和 Watch 数量递减排序, 分别得到单一指标最高的 800 个项目, 然后在它们的交集中取 400 个 Star 数量最高的项目作为最终选定的项目. 部分如表 1 所示.

为使数据种类更加丰富,我们还收集了 80 个最常用的 Java 的 jar 包.源码数据集的选择标准如下:首先,这些项目都是开源的,并且在软件工程相关的研究中具有较高的使用频率.因此,可以认为它们具有更高的质量,更加符合一般的编程规范,代码简洁并且上下文联系更加紧密.其次,这些 jar 包项目的版本更新较为活跃,具有悠久的演化历史.我们从中选择相对稳定的版本作为数据集.并且,大部分这些 jar 包隶属于正规组织或软件公司.部分如表 1 所示.

Table 1 Examples of open source projects and jar packages used in paper

表 1 部分所使用的开源项目和 jar 包示例

开源项目	TheAlgorithms	junit-team	square	OpenRefine	b3log
Star	8.2k	7.2k	6k	5.7	5.5k
Jar 包	junit	guava	commons-io	testng	log4j
版本	4.8	23.3	1.3	5.13	1.2

480 个项目以方法为单位进行切分, 并删除代码行少于 5 行以及不具有独立功能的方法（比如 main 方法）

之后, 方法块总数为 380000 左右, 包含的代码行数大约为 4830000 行. 这个数量级的数据, 我们认为在考虑到 Java 代码行的复杂性和多样性的情况下, 作为训练数据来说也是足够的.

因为这些项目具有较高的认可度,本文认为其比一般的项目具有更好的代码结构,代码更加简洁,上下文之间联系更加紧密,因此这些项目的源代码具有更高的质量.比如其中的 80 个 jar 包,在经历了多版本演化之后,项目代码质量相对较高.因此,这些项目在经过数据处理之后适合用来做代码行推荐的训练数据集.

2.2 源数据处理模块

训练数据处理模块对已经收集到的原始训练数据集进行数据预处理.为了对已经收集到的项目的源代码进行统一化的处理,首先需要获取源码中的变量声明、方法调用和控制结构等信息,对源码的语法结构进行分析.我们使用 Eclipse 的 JDT 编译器[17]将源码文件解析为抽象语法树（AST）.此外,由于部分 jar 包中没有源码,本文使用 jd-jui 反编译工具[18]对 jar 包中 class 文件进行反编译来获得源码.首先分析所有后缀为 Java 的文件,将变量声明和类型绑定,以及同一类型的不同变量名、变量值进行统一化处理.对于同一个类的不同对象,用对应类的类型替换所有的对象类型,用类名的小写形式替换不同的对象名.该统一化处理有利于代码行上下文关系的学习,另外在推荐选择时,开发人员可以根据实际需求对代码进行处理,更加契合实际的使用场景.

对于 Java 中的八种基本数据类型,即 byte、short、int、long、float、double、char 和 boolean,对其变量名和值都进行统一化处理.具体处理规则如表 2 所示.

Table 2 Uniform processing rules for basic data types in Java

表 2 Java 基本数据类型的统一化处理规则

数据类型	变量名（统一化）	变量值（统一化）	示例（处理前）	示例（处理后）
byte	byte_type	0	byte b = 'a';	byte byte_type = 0;
short	short_type	-1	short s = 100;	short short_type = -1;
int	int_type	1	int i = 0;	int int_type = 1;
long	long_type	10	long l = 1000;	long long_type = 10;
float	float_type	-0.1	float f = 1000;	float float_type = -0.1;
double	double_type	0.1	double d = 0.01;	double double_type = 0.01;
char	char_type	‘a’	char c = 'c';	char char_type = 'a';
boolean	boolean_type	true	boolean b = false;	boolean boolean_type = true;

对于非基本数据类型,如同一个类的不同对象,用对应类的类型替换所有的对象类型,直接用类名的小写形式替换不同的对象名. 比如, *List resultList = new ArrayList();*统一化处理之后为 *List list = new ArrayList();*.因为 List 为非基本数据类型, 所以按照规则用类名 *List* 的小写形式 *list* 替换原本的用户自定义对象名 *resultList*.

忽略那些不会影响语法结构、开发人员可知的符号,这些符号在代码行被推荐后,很容易被补全,不会影响实际的使用.比如代码行结尾的分号和某些大小括号等.

在处理完所有数据之后,构建代码段数据集 D 具有规范的统一格式.图 2 表示处理之后的方法块实例.根据程序设计的模块化特点,为了得到关系更加紧密的代码上下文,我们从方法层面构建训练数据集.同时,为了能从功能比较完整的方法块中抽取数据.为了清除无意义的行,忽略只有大括号或者只有注释的行.

2.3 训练数据处理模块

在抽取训练数据集时具体方法是:对一个功能完整的方法块,忽略第一行方法的声明,从第 n+1 行开始,以前 n 行为模型输入、第 n+1 行为模型的输出构建模型的一个实例,并依次向下直至方法块最后一行,构建训练数据集.根据经验,并充分考虑实际开发中的情况,为了得到关系最紧密的代码上下文, n 的可能取值为 1-5.理论上来说,n 取值越大越好,但是在实际开发过程中,除了功能特别复杂的方法块,普通方法块的代码行数通常不会很大.

因此,在保证准确率的情况下,n 的取值越小,在实际使用中实用性会提高.最合适的取值在本文后面的实例验证将讨论 n 的合适取值.在此之前我们假定 n 的取值为 3.

```
/**
 * Return the subscript of the element which is the
 * largest in given list
 * @param the given list
 * @return subscript
 */
public int getMostList(List<Integer> list){
    int index = 0;
    int max = 0;
    for(int i=0;i<list.size();i++){
        if(max < list.get(i)){
            max = list.get(i);
            index = i;
        }
    }
    return index;
}
```

```
/**
 * Return the subscript of the element which is the
 * largest in given list
 * @param the given list
 * @return subscript
 */
public int getMostList(List<Integer> list<integer>){
    int int_type = 1;
    int int_type = 1;
    for int int_type=0 int_type<list<integer>.size int_type++
        if int_type < list<integer>.get(int_type)
            int_type = list<integer>.get(int_type)
            int_type = int_type

    return int_type
}
```

Fig.2 An example of unifying source code

图 2 代码方法的格式统一化处理实例

2.4 编程现场数据处理模块

编程现场是指软件生产中与当前编码相关的要素集合,编程现场大数据是编程现场中即时产生的、以编程为中心的海量数据集,比如编程现场行为数据、源代码数据、软件任务数据和软件系统运维数据等.在本文中没有使用编程现场的所有可用数据,而是根据实际需求抽取了一部分真正需要的数据.为了更好的获取编程现场大数据,我们开发编程现场数据处理模块,包括开发现场数据采集模块、现场数据清洗模块和现场数据组织管理模块,对开发现场的数据进行实时采集、清洗、组织和管理.如图 3 所示.

开发现场数据采集模块主要处理编程现场大数据的抽取问题.在本文中,开发人员已经输入的编程现场代码行上下文通过采集模块自动获取,以便根据获取到的代码行上下文,利用训练好的模型预测最可能的当前代码行.除了代码行上下文,数据采集模块还尽可能地采集用户当前的任务数据,以获取开发人员当前可能的开发任务,主要包括类名和方法名.除此以外,编程习惯良好的开发人员会在方法编写之前,在方法声明之前的位置添加方法注释.方法注释通常是一个单词序列,描述方法功能信息.因此,采集方法注释能够帮助捕捉开发人员当前的编程意图.

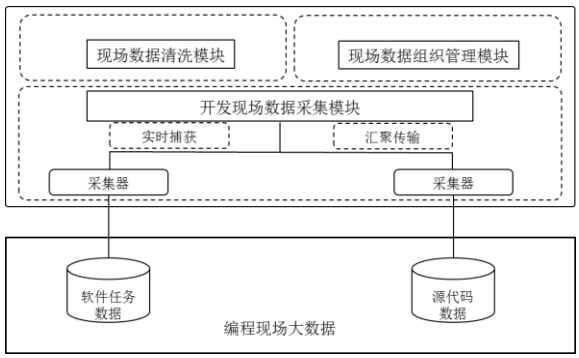

Fig.3 A structure diagram of data processing module in programming scene

图 3 编程现场的数据处理模块结构图

现场数据清洗模块主要对数据采集模块采集到原始数据进行数据清洗,因为数据采集模块采集到的编程现场的数据包含大量无用信息,无法直接使用.现场数据清洗模块接收原始的代码行上下文数据,然后按照 2.2 节提出的数据处理规则对数据进行统一化处理,处理完之后,代码行数据就具有了和训练数据集统一的规范格

式,在后续的操作中就能作为模型的输入获取当前代码行.

由于代码行上下文也包含一定的任务信息,因此,对于采集到的任务数据,现场数据清洗模块以*类名#方法名#代码行上下文#注释信息*的形式,将获取到的当前正在编写的任务信息交由现场数据组织管理模块处理,最终存储在本地,用于后面推荐过程中对推荐结果进行二次排序和优化.

3 编程现场上下文深度感知的代码行推荐方法

本节具体描述编程现场上下文深度感知的代码行推荐方法,包括学习方法模型和推荐方法.简单来说就是基于深度学习的方法,通过开发人员当前已经输入的代码行上下文生成当前代码行进行预测并推荐给开发人员.本文学习方法使用基于注意力的循环神经网络编码--解码模型（attention-based RNN Encoder-Decoder model）来实现代码行的学习的任务[7,19].下图 4 描述了方法的总体结构,包含了一个离线训练模型的阶段和一个在线代码行生成推荐的阶段.在训练阶段,我们准备了一个大规模的上下文代码行的数据集来进行模型训练.上下文代码行对用于训练一个深度学习模型,即基于注意力的循环神经网络编码—解码模型.

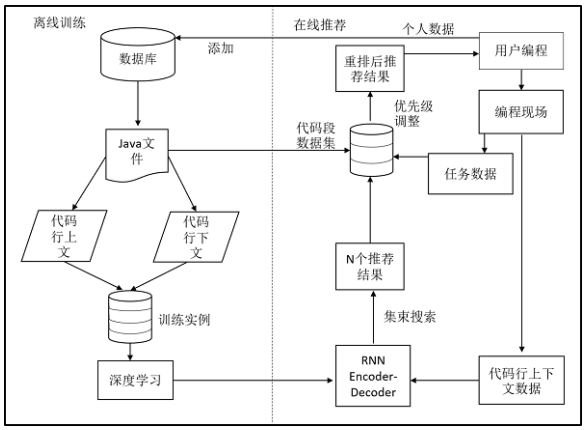

Fig.4 The Overall Workflow of DA4CLR

图 4 编程现场上下文深度感知的代码行推荐方法总体流程图

理论上,本文的方法可以应用于任何程序设计语言.为了方便描述,我们用 Java 程序设计语言为例进行描述.在下面的章节中,本文将对方法模型进行详细介绍.

3.1 方法模型

如之前在章节 2 描述,循环神经网络编码—解码模型是对基本神经语言模型的扩展,能较好解决 Sequence2Sequence 问题.因此循环神经网络编码—解码模型用来作为实现本文方法的重要手段.

图 5 展示了将循环神经网络编码器—解码器的 RNN 框架用于代码行上下文感知的示例.将已有的上下文代码行 *String string = "stringValue" for(String string : list<String>*作为输入序列、*string = string + string.trim() + "stringValue"*为输出序列,即为当前代码行.如图所示,编码器 RNN 将源语句 *String string = "stringValue" for(String string : list<String>* 中单词逐一读入,当读第一个单词 *String* 的时候,该单词被数值化表示为向量 x_1,并用该向量计算当前的隐藏状态 h_1[20].然后,读入下一个单词 *string*,数值化表示为 x_2,然后用 x_2 将隐藏状态 h_1 更新为 h_2.不断重复该过程,直到读入最后一个单词 *list<String>*,并得到最后一个状态 h_8,最后的状态就是上下文向量 c.

解码器 RNN 要根据向量 c 来计算得到目标序列 *string = string + string.trim() + "stringValue"*.目标序列的开头,被人为的添加一个开始标记<START>.在解码时,它被首先计算得到第一个单词 y_0, 然后,基于上下文向量 c 和 y_0,计算得到隐藏状态 h_1,并根据该状态预测代码行中第一个单词 *string*.然后根据前一个单词向量 y_1 和上下文向量 c 计算下一个隐藏状态 h_2,并预测代码行中第二单词“=”.该过程一直重复,直到预测得到目标序列的结

束标记<EOS>,该标记是在数据处理阶段人为的添加在目标序列的结尾的.

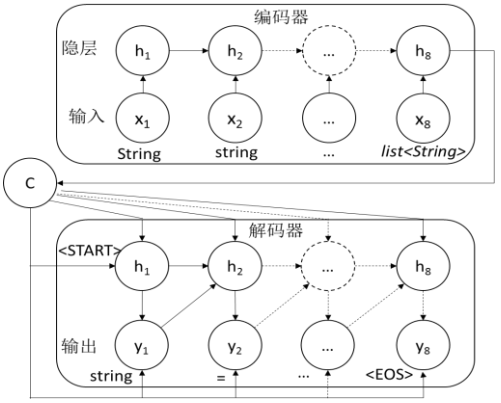

Fig.5 A sample of RNN Encoder-Decoder Model Applied to Code Lines

图 5 RNN 编码--解码模型应用于代码行的示例

已有的代码行上文中的不同部分,对产生目标序列（即当前代码行）的重要性是不同的.比如,代码行上下文 *String string = "stringValue" for String string : list<String>* 和当前代码行 *string = string + string.trim() +"stringValue"*相比较而言,单词 *String* 比单词 *for* 对得到目标序列中的单词 *string* 具有更大的影响.因此,在本文的方法中,我们采用基于注意力的 RNN 编码器--解码器模型,它能够从输入序列中选择对目标序列中当前单词最重要的部分.不同于使用相同上下文向量 c 生成目标序列,注意力模型对目标序列中每个单词 y_j 使用不同的向量 c_j.该向量的计算方式如下:

$$c_j = \sum_{t=1}^{T_x} \alpha_{jt} h_t \tag{1}$$

α_{jt} 是隐藏状态 h_t 对目标序列中单词 y_j 的权重值,该值不需要单独计算,而是可以利用另一种神经网络进行模型化,并在训练过程中学习到[12].

3.2 训练方法模型

基于注意力的循环神经网络编码--解码模型在实际应用中有很多不同的实现方法,我们使用章节 2 描述的长短期记忆网络.作为一种性能较好的 RNN 具体应用,它在很多的实际任务中取得了较好的效果[19,21].本文按照如下描述构建模型:在解码器中,使用两个 RNN 分别接收是源语句的正向和逆向语句,生成的上下文向量连接到解码器[7].解码部分使用了 attention 模型,通过上下文向量和每个时刻的上一时刻输出进行解码,进而得到最终的预测代码行.所有 RNN 都有 1000 个隐藏单元,词嵌入的维度设置为 120 维[22].

因为数据集规模很大,为了更好对模型进行训练,本文限制词汇表大小为词频最高的 10000 词.同时为了兼顾效果和效率,所有模型都使用*小批量随机梯度下降算法（MBGD）*来进行参数优化[23],该算法自动调整学习速率.批量大小（即每批次的实例数）设置为 100.

具体实现时,本文使用 TensorFlow,这是一个 Google 的开源的深度学习框架[24].该框架功能强大,并且已经实现了大部分深度学习算法和参数优化方法.整个训练过程持续约 240 小时,并重复 100 万次.

3.3 推荐过程

到目前为止,本文已经讨论了编程现场上下文深度感知代码行推荐方法的训练阶段.本文的方法自动获取开发人员已经输入的编程现场代码行上下文,然后,训练好的模型会根据获取到的代码行上下文来预测最可能的当前代码行.程序开发语言的编写过程中,代码行之间具有一定的无序性,即交换某些行的顺序并不会影响程序功能的实现.换言之,即根据已有的代码行,得到的当前代码行可能并不唯一.因此,为了程序开发人员能更好

地获得想要的代码行,我们应该尽可能获得所有可能正确的当前代码行,并按可能性优先级进行排序,然后推荐给开发人员让其进行选择.

具体推荐过程的伪代码如下:

算法 1:编程现场上下文深度感知代码行推荐过程算法

```
Input:编程现场数据
BEGIN
1:    Action <-- getUserAction();
2:    //获取用户动作
3:    DO WHILE(Action == True)
4:    //如果用户选择推荐
5:        setParameters();//参数设置
6:        CodeLines <-- getCodelines();//获取编程现场代码行数据
7:        CodeStructureInformation <-- AST(CodeLines);
8:        //利用抽象语法树抽取结构信息
9:        UnifiedCodeLines <-- unifyCodelines(CodeStructureInformation);
10:       //数据处理模块对初始代码行进行统一化处理
11:       TaskData <-- GetTaskData();
12:       //采集编程现场任务数据
13:       RecommendedLines <-- deepModel(UnifiedCodeLines);
14:       //以统一化处理之后的代码行上下文作为模型输入得到推荐代码行结果
15:       OrderedRecommendedLines <-- beamSearch(RecommendedLines);
16:       //集束搜索实现初次推荐结果排序
17:       FOR j=0 TO n DO
18:           <SippetData,SIMILARITY> <-- LSA(TaskData, θ, SippetDataSets);
19:           //根据 LSA 相似度和预设的阈值获取代码段数据集中和当前编写方法类似
的代码段及相似度
20:       COUNTER <-- 0;
21:       DO WHILE(COUNTER < n)
22:       //对 n 个推荐结果进行二次排序
23:           IF (OrderedRecommendedLines IN SippetData) THEN
24:               SecondOrderedLines                                    <--
Reordering(OrderedRecommendedLines,SIMILARITYS);
25:               //调整推荐代码行优先级
26:           END IF
27:           COUNTER++;
28:   END WHILE
END
Output:N 个排好序的代码行推荐结果
```

为了实现上述的目标,本文使用*集束搜索(Beam Search)*[25].集束搜索是一种启发式搜索算法,通常用在图的解空间比较大的情况下,为了减少搜索所占用的空间和时间,在每一步深度扩展的时候,剪掉一些质量比较差的结点,保留下一些质量较高的结点.这样减少了空间消耗,并提高了时间效率.在束搜索中,仅保留预定义数目的最优部分解.

图 6 展示了一个集束搜索示例结合具体的代码行推荐任务,如果要推荐 2 个代码行给用户进行选择,则在每个时间步,选择损失值最小的 2 个单词序列,然后按照这个规则依次向前计算,直到遇到句子结尾符则停止.根据已有的代码行上下文, <START>为开始标记.然后,根据损失值的大小计算得到损失值最小的两个单词,分别为损失值为 2 的 string 和损失值为 6 的"stringValue",其它的单词被忽略.然后一直重复这样的过程,直到选择句子结束符作为下一个单词时,结束整个过程.这里是为了方便描述,选择推荐 2 个代码行,实际上可以推荐任意 N 个结果.

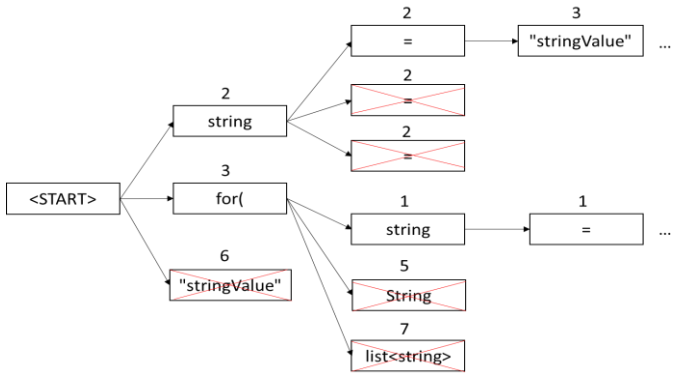

Fig.6 A sample for Beam Search

图 6 集束搜索示例

通过集束搜索得到经过初次排序的 n 个推荐结果之后,还需要根据 2.3 节所采集到的当前任务数据对推荐结果的优先级进行基于检索的方法的二次排序.具体操作是: 采集到的当前任务数据的形式是*类名#方法名#代码行上下文#注释信息*, 利用已有的类名#方法名#代码行上下文#注释信息和代码段数据集 D 中的代码段信息进行相似度比较. 即将代码段数据集 D 中每个代码段信息抽取整合为类名#方法名#代码行上下文#注释信息形式, 然后和当前任务数据进行比较, 计算相似度. 如果存在相似度高于阈值的代码段,并且该代码段中含有推荐结果中优先级为 k 的推荐的代码行,则将该推荐结果重新排序, 将该推荐项在整个包含 k 个推荐项的推荐结果中的优先级进行上调,相似度越高,优先级越高.相似度衡量使用 LSA (latent semantic analysis)潜在语义分析,LSA 的核心思想是将词和文档映射到潜在语义空间,再比较其相似性,从而去除了原始向量空间中的一些“噪音”.由于相似度匹配考虑到了语义相似度,而不仅仅是进行关键词匹配,故提高了信息检索的精确度.这里的阈值设置为 θ,具体的值在后面的实验中通过实验得到.

为了易于理解, 我们对整个过程举例描述, 为了方便表示, 这里指定推荐代码行数目为 2，如下所示为：

在实际开发过程中, 用户在类 Tools 中编写 getFileContent 方法块时, 对方法的功能描述为: Gets the file contents for the specified file path, 并且已经键入:

String string = "stringValue"

for(String string : list<String>)

基于注意力的循环神经网络编码--解码模型以此为输入得到的推荐结果为：

1、 continue

2、string = string + string.trim() + "stringValue"

二次排序时,实时捕获用户当前的软件任务数据，获取开发人员当前可能的开发意图, 具体即为用户当前开发对应的*类名#方法名#代码行上下文#注释信息*,分别对应上述具体信息. 将其和已有的代码段数据集中每个方法段的*类名#方法名#代码行上下文#注释信息*进行相似度比较时, 存在相似度高于阈值的代码段 readFile, 并且该代码段中含有优先级为 2 的推荐项 continue, 则将该推荐结果重新排序, 进行优先级上调.如下:

1、string = string + string.trim() + "stringValue"

2、 continue

重排后推荐结果作为最终结果被推荐给用户.

4 方法模型评估

这部分主要对本文的代码行推荐方法进行评估,除了根据设计的问题对方法进行实例化验证,我们还通过问卷调查的形式来评估推荐结果的实用性.

4.1 实验设计

我们主要通过测量所推荐代码行的准确率、MRR 来评估本文的编程现场上下文深度感知的代码行推荐方法是否有效.为了更好地进行实例验证,我们特别设计了四个问题从不同方面对方法进行评估.为了使实验更具有针对性并简化实验过程, 第一和三这两个问题的实验中没有对推荐结果进行二次排序,而在第二和四这两个实验中进行二次优先级调整, 着重关注二次排序对推荐结果的影响.

4.1.1 研究问题

我们主要研究如下的一些问题,并在后续的章节通过一些实验来对其进行回答.

- *问题 1:方法的性能是否会被代码行数据集预处理的程度所影响*

在之前数据处理部分,我们详细描述了如何对收集到的项目的源代码进行预处理.这样做的原因是,希望通过统一化的源码数据,代码行上下文的深度学习能学得更加一般的代码行上下文隐含模式,进而获得更好的推荐结果.但是,对数据集的处理程度越高,统一化程度越高,也意味着开发人员即使得到推荐结果时,为了使代码行变得真正可用,他所需要修改的地方也很多.因此,我们设计问题 1 来探究数据预处理程度对方法准确率的影响,并希望能通过这个问题的研究得到最好的数据集

- *问题 2: 本文提出的方法和类似的代码推荐方法以及类似相关工作相比准确率如何*

已有的一些源代码相关的工作主要的关注点是代码补全、代码纠错和源码查询等.从粒度上来看,大部分工作关注的是更加细粒度的 API 或 API 序列的获取和推荐.也有一部分研究聚焦于方法粒度级别的工作.对代码行的处理和推荐的工作比较少见.因此,我们比较代码搜索的相关方法来验证其是否可以替代本文的方法.

此外, 为了进一步验证本文方法的有效性, 我们选择两个相似的代码行补全的代表性方法与本方法进行对比.

- *问题 3:本方法推荐的代码行准确率是否受到一些参数设置的影响*

在我们的方法中,要检测的参数主要是上下文代码行数 n 和推荐结果数 k.

正如之前所说,我们为了获取匹配的上下文,从第 n+1 行开始,以前 n 行为模型输入、第 n+1 行为模型的输出构建模型的一个实例.为了关系最紧密的代码上下文,n 的取值不能太大,也不能太小.n 值太小的话可能无法得到足够的上下文信息.而 n 值太大的话,距离 n-2 行的两个代码行之间可能联系已经非常弱了,反而会成为无意义的代码行. 基于这些考虑和已有的经验来看, n 的可能取值为 1、2、3、4 或者 5.而更加准确的取值则需要通过实验来得到.

对于 k,我们将设置为 1、5 和 10 分别进行测试来观察推荐结果数对推荐结果的影响.

- *问题 4:本方法中对推荐结果进行二次排序是否能使相关度更高的结果具有更高优先级以及比较合适的 θ 值设置是多少*

在我们的方法中,在通过集束搜索得到最合适的 n 个排好序推荐结果之后,还要利用之前提到的 LSA 相似度对推荐结果进行二次排序,然后才将最终的结果推荐给用户.这一过程是否必要,需要通过实验验证.对于优先级调整的 LSA 相似度阈值设置,设置太低或者太高,虽然对二次排序结果影响很大,但是对排序所花费时间影响同样很大.因此,根据经验,将 LSA 相似度的可能的取值设为 0.4、0.5、0.6、0.7 和 0.8,并根据实验找出最合适的 θ 值.

4.1.2 衡量指标

一个比较理想的推荐方法应该推荐不止一个相关性较高的推荐项,并且用户越满意的推荐项应该排在推荐结果中更加靠前的位置.在方法测试过程中,我们使用 MRR (Mean Reciprocal Rank)和准确率（Precision）作为衡量指标来评估方法以及用来比较的方法.

MRR 是目前比较通用的对搜索算法进行评价的机制,即第一个结果匹配分数为 1,第二个匹配分数为 0.5,第 n 个匹配分数为 1/n,如果没有匹配的句子分数为 0.最终的分数为所有得分之和.具体的计算公式如下:

$$MRR = \frac{1}{|N|}\sum_{i=1}^{|N|}\frac{1}{rank_i} \tag{2}$$

其中|N|是查询个数,$rank_i$是第 i 个查询,第一个相关的结果所在的排列位置.

准确率（Precision）则定义为所有正确的推荐结果数占推荐结果总数的比例,计算公式如下:

$$Precision = \frac{|Relevance|}{|Retrieved|} \tag{3}$$

这里的分子|Relevance|是 k 个推荐结果中相关推荐项的个数,而分母|Retrieved|则是推荐结果总数,即为 k,在不同参数时分别取 1、5 和 10.因为我们的方法不是简单的文本匹配推荐,所以这里的相关项|Relevance|并不是简单的与测试集的理论结果完全匹配的项.为此,我们使用 BLEU 相似度得分来衡量一个推荐项是否有效.该相似度计算公式如下:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{4}$$

其中,N 可取 1、2、3、4,而 w_n一般对所有 n 取常数值,即 1/n.p_n表示长度为 n 的子序列的比率.BP 是惩罚因子,计算公式如下:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c <= r \end{cases} \tag{5}$$

其中 r 是参考序列的长度,c 是候选序列的长度.

当推荐项和参考推荐结果的 BLEU 相似度得分大于阈值 62.79（%）时,我们评定该推荐项为有效推荐项.该阈值是在试验 3 中通过对 100 个测试集进行人工观测评定准确性后,取有效推荐项计算平均 BLEU 相似度得分所得.在四个实验中,我们对实验 3 的结果进行人工观测评估,其它 3 个实验采用 BLEU 相似度得分来衡量推荐结果中推荐项是否有效.

4.2 实验结果

本节将展示实验的结果,并对 4.1 中提出的四个问题逐一回答.

4.2.1 对问题1的回答

为了回答问题 1,我们探讨源代码数据集的处理程度对推荐结果的影响.在该实验中,我们对原始数据集进行不同程度的处理.按照处理程度的不同,分为 4 个等级,从 0—3 分别为原始数据集、对 API 进行统一化、对基本数据类型进行处理和删除某些特定字符（如无意义的括号和变量值）,高等级包括所有低等级的处理.

具体来说就是:level0 表示对数据集不进行任何的统一化处理, 保留从源码中获取的原始数据集; level1 表示对源码中所有的 API（非基本数据类型）按照前文所述规则进行统一化处理, 比如 *List resultList = new ArrayList()*统一化处理为 *List list = new ArrayList()*; level2 表示不仅包括 level1 等级的处理, 即 API 等非基本数据类型的处理, 还有对基本数据类型进行统一化处理; level3 则不仅包括 level1 和 level2 的所有统一化处理, 还删除一些特定的字符, 比如代码行中的括号和变量名.

为了控制变量,在进行该实验时,我们将 k 值设置为 10,即对每个测试用例推荐 10 个供选择的代码行.用于测试的实例数共 1000 个.表 3 中给出 5 个实验中真实使用的实例.由表可知:对源码进行统一化处理的程度不同对推荐结果的准确率和 MRR 两个指标都有影响.

Table 3 Instance of data processing

表 3 数据集处理实例

ID	输入	输出
1	boolean boolean_type = true for(int int_type=0 int_type<list.size() int_type++)	if(list.get(int_type).equals("stringValue"))
2	List<String> list<string> = new ArrayList<String>() String string = "stringValue"	while(string = bufferedreader.readLine() != null)
3	bufferedwriter.append list<string>.get(int_type) bufferedwriter.newLine()	bufferedwriter.flush()
4	String string = list<string>.get(int_type) if(string.equals("stringValue"))	continue
5	String string = "stringValue" for(String string : list<String>)	string = string + string.trim() + "stringValue"

表 4 总结了两个衡量指标在 4 个不同等级处理程度的结果.从表中我们可以看到,在没有对数据集进行任何处理时,准确率仅仅只有 17%,这是因为没有做任何处理的源代码中包含大量的个性化字符,包括用户自定义的变量名、变量值和对象名等.这些字符都是噪声,会对方法的推荐结果产生影响.而对 API 进行处理和对数据类型进行统一化之后,准确率分别是 17%提高到 29%、29%提高到 55%.这说明推荐结果的准确率受处理的 API 和数据类型影响较大.而对于 MRR,我们从表中可以看到对 0—3 不同处理等级,对应的值分别是 0.3104、0.4464、0.4986 和 0.5148.在对 API 和对数据类型进行统一化之后,分别增加了 0.136 和 0.0522.对比于除某些特定字符后增加的 0.0162,说明也受到比较大的影响.这也说明,在进行了数据类型级别的数据处理之后,方法的性能已经达到一个较好的状态.继续对数据集进行处理,虽然也还是能够对方法效果进行优化,但是性能的提高已经不是很明显.

Table 4 Experimental results of two metrics at different levels

表 4 不同处理等级在两个衡量指标上的实验结果

指标	Level 0	Level 1	Level 2	Level 3
Precision	17%	29%	55%	58%
MRR	0.0528	0.1295	0.2743	0.2986

从更加直观的角度,图 7 是两个不同指标在 4 个等级上的变化趋势.从图中我们可以发现:在准确率这个指标上,相对于第 3 级的处理,第 1 级和第 2 级的处理对方法推荐结果准确率的提高更明显.而在 MRR 上,相对于第 1 级和第 3 级的处理,第 2 级的处理对推荐结果的影响更加明显.这说明我们的方法在能够推荐正确结果时,最匹配的结果在所有推荐中排在比较靠前的位置.从通用的推荐算法评价标准来看,这是一个很好的趋势.

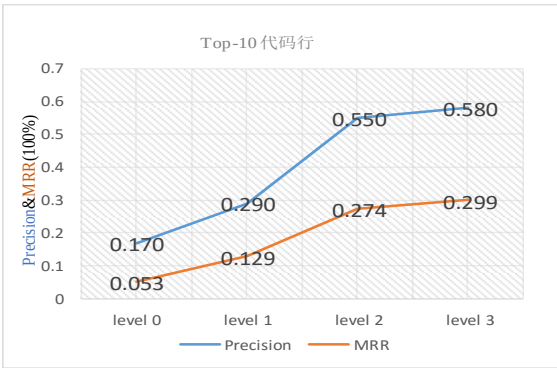

Fig.7 Trends on two metrics at different processing levels

图 7 不同处理等级在两个衡量指表上的变化趋势

对问题 1 的回答:编程现场上下文深度感知的代码行推荐方法的准确率和 MRR 会被数据集的处理程度影

响,并且方法性能随着数据集的处理程度的变高而提升.而且,不同程度的处理对方法性能影响也是不同的.第 1 等级,即对源代码的 API 进行统一化处理,是一个比较重要的处理步骤,在方法推荐结果的准确率和 MRR 两个指标上都有明显提升.第 3 等级的统一化处理之后,源代码中几乎只保留了 Java 关键字、类名和 API,此时方法性能最好.但是,考虑到对数据集进行处理的程度越高,也意味着开发人员即使得到推荐结果时,他所需要修改的地方也很多.因此,我们放弃对源码进行第 3 级的处理,保留具有 Java 结构特征的一些字符(主要是一些括号).我们认为这样会使得本文的方法对开发人员而言具有最大的使用价值.

4.2.2 对问题2的回答

问题 2 在我们的第二个实验中被测试,该实验的内容是将本文提出的代码行上下文深度感知的方法和具有类似的功能的方法进行比较.具体使用的两个代码搜索的比较方法是:BM25 和 TF-IDF[26,27].这是两个比较具有代表性的计算文本相似度的算法,在这里,我们用这两个算法来实现代码搜索.

虽然本文的代码行推荐方法并不是有查询推荐,而是自动获取编程现场的代码行上下文,然后通过计算进行推荐.但是,并不影响本文方法和输入自由形式查询的方法进行比较.因为在我们对方法进行测试时,使用的 1000 个可用的测试用例,是以代码行上下文对的形式出现的.具体来说,测试实例中的输入可以看成是有查询搜索中的查询.这样,我们就可以将本文无查询的方法和有查询搜索的方法进行比较.

为了有效进行实验比较,我们使用准确率和 MRR 这两个衡量指标对本文方法以和两个对比的基于文本的代码搜索方法进行评估.

图 8 是三种方法 DA4CLR、BM25 和 TF-IDF 的实验结果统计表示图,包括最大值、最小值、平均值和中位数值.从图中可以看出,本文的推荐方法具有接近 60%的较高准确率,而两种基于文本的代码搜索方法 BM25 和 TF-IDF 准确率相近.考虑到源代码的复杂应用场景,简单的文本匹配能取得这样的推荐准确率已经算是比较理想的情况.分析的原因是,测试集中有一部分是具有 API 的代码行,这类实例的特征具有独特性,可以与其它实例进行明显的区分.具有这种特性的实例,即使使用文本搜索算法也能得到理想的推荐结果.但是,对于一些具有更加复杂结构的代码行,文本匹配的方法很难取得令人满意的效果.

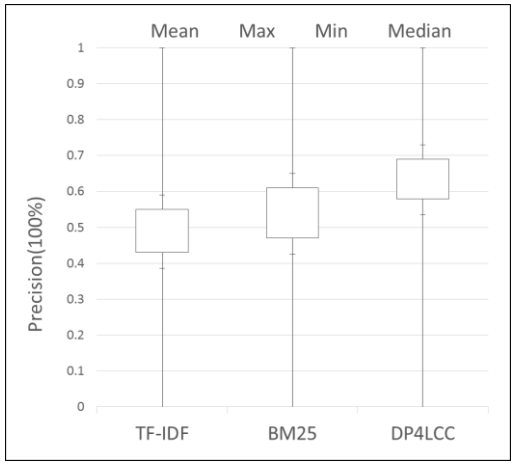

Fig.8 Statistical representation for Precision of three different methods

图 8 三种不同方法的准确率统计量图

为了进一步验证本文方法的有效性, 我们选择两个相似的代码行补全的代表性方法与本方法进行对比, 分别是利用统计语义语言模型[28]和 RNN 模型[29]来进行代码行补全的方法, 衡量指标采用平均 BLEU 得分.

Table 5 BLEU scores for three relevant methods with different Top-K

表 5 三种相关方法在不同 Top-K 时 BLEU 得分

方法	Top1	Top5	Top10
Lexical n-gram	0.2053	0.2625	0.3164

RNN	0.2539	0.3313	0.3717
DA4CLR	0.3865	0.4286	0.4541

表 5 是三种方法 Lexical n-gram, RNN 和 DA4CLR 在推荐结果数分别为 1,5 和 10 时对应 BLEU 平均得分. 从表中我们可以看出,本文的代码行推荐方法在 Top1 时得分是相当高的,分别高于 Lexical n-gram 模型 0.18 和 RNN 模型 0.13.但是随着推荐项数量增多, 这种情况开始改变, 当推荐项数为 5 和 10 时, 其平均得分和 Lexical n-gram 以及 RNN 的得分差距已经开始逐渐减小. 这说明我们的方法中推荐正确的推荐项出现在推荐结果列表中比较靠前的位置, 这从一定程度上说明本文利用信息检索模型对推荐结果列表进行二次排序是比较成功的.总的来说, 本文的方法在平均 BLEU 得分指标上对比于 Lexical n-gram 和 RNN 仍然有较大的优势.

Table 6 Ranking information of recommended results for five different methods

表 6 5 种不同的方法的最佳推荐结果位置信息

方法	FRank	LRank	MRR
DA4CLR	1	10	0.3066
BM25	1	6	0.2566
TF-IDF	1	7	0.2454
RNN	1	10	0.2741
Lexical n-gram	1	8	0.2803

表 6 是 5 种方法 DA4CLR, BM25, TF-IDF ,Lexical n-gram, RNN 的最佳推荐结果在所有结果中的位置信息的比较.从表中我们可以看出, BM25 和 TF-IDF 在这项指标上总体差距不是很明显.Frank 和 LRank 分别是最好的推荐结果在所有推荐结果中最靠前和最靠后的位置信息.从表中我们可以看出,最靠前的情况,5 种方法都有在排在第一的位置得到最佳结果的情况,而对于靠后的最佳推荐结果, BM25 和 TF-IDF 没有像我们的方法一样有在第十的排序上得到最佳结果的情况.这说明,两个文本搜索方法如果能得到正确的推荐,那么该结果就会排在一个比较靠前的位置.结合前面的准确率分析,可以看出,文本搜索方法只是对某些代码行具有比较好的效果. 而 RNN 和 Lexical n-gram 方法在 Frank 和 LRank 中相互差距都不是很大.但在推荐结果项数为 10 的时候, 对应的 MRR 得分远小于我们的 DA4CLR 方法, 对比类似的 RNN 方法,在一定程度上说明本文融合的基于检索方法来进行二次排序效果较好. MRR 值越大, 说明推荐项的整体优先级排序情况是比较符合用户需求的. 因此, 本文方法能够将用户满意的推荐项排在推荐结果中靠前的位置, 从方法的综合效果考虑,我们的方法具有更大的优势.

对问题 2 的回答:从上述比较结果可以看出,本文的代码行推荐方法在准确率上对比文本搜索方法以及类似相关方法有一定的提升.并且,考虑到准确率和更加一般的适用性,上述实验结果表明,和具有类似功能的方法相比,本文方法能够在不降低推荐准确率的情况下,仍然保证相关性较高的代码行在推荐结果中的位置也相对靠前.

4.2.3 对问题3的回答

我们设计的第三个实验,是为了测试本文方法在不同参数条件下的推荐情况.检测的参数主要是上下文代码行数 n 和推荐结果数 k.如我们前面所说,上下文代码行数 n 代表着对已有信息的获取程度,它的不同,可能直接对方法性能产生影响.而合适推荐结果数 k 则意味着,可以在我们方法的推荐效果和推荐效率之间取得最佳平衡.所以,对这两个参数进行研究,以找到最佳取值,对我们的方法来说是很有必要的.

根据 n 的可能取值,我们将 n 分为设置 1、2、3、4 和 5 进行实验.对 n 进行实验时,将 k 值设为 10;对于 k, 我们设置为 1、5、8 和 10 分别进行测试来观察推荐结果数对推荐结果的影响,对 k 进行测试时,将 n 值设置为 2.所有的数据都经过前面所定义的第 2 等级的统一化处理,并且去掉一些不会影响结构的字符.用于测试的实例数共 100 个,全部采用人工观测的方式进行推荐结果有效性判断.

图 9 显示在上下文代码行数 n 为 3 左右时,准确率达到一个比较高的程度,然后随着行数的增大,准确率并

没有很大的提升.造成这种情况的原因可能是,对于源代码而言,上下文代码行达到四行的时候,代码行之间的联系已经比较紧密,对应 n 为 3.当 n 大于 3 时,对推荐结果准确率的提高并不会有所帮助.

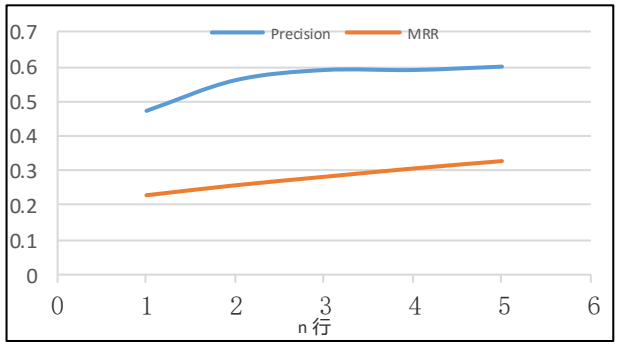

Fig.9 The trend for two metrics with different contextual code lines

图 9 不同上下文代码行数下两个指标趋势图

表 7 则表明,随着推荐结果数增加,准确率也一直变大.这符合我们理论上的预测,因为推荐结果越多,结果中涵盖正确推荐的可能性就越大.但是,我们也应该注意到,当推荐结果数分别为 8 和 10 时,准确率的提高已经很小.再结合 MRR 的对应值,特别是考虑到对开发者的实用性,我们可以总结出,对于推荐结果数 k,取值为 8 已经是一个比较好的选择.当然,出于更好的性能需求考虑,在实验中都暂时将 k 值定为 10.

Table 7 The Precision and MRR of different N values

表 7 不同的 TOP-N 对应的准确率和 MRR

Top-N	1	5	8	10
Precision	21%	37%	55%	58%
MRR	0.2100	0.2698	0.2963	0.2995

对问题 3 的回答:对于上下文代码行数 n 和推荐结果数 k,不同的参数取值确实会对我们的方法的推荐结果产生影响.根据已有的实验结果,我们得出的结论是,充分考虑到开发过程中一个完整方法块本身就不会很长,以及避免序列过长对推荐速度产生影响,在保证推荐准确率不会有太大影响情况下,将上下文代码行数 n 取为 3、将推荐结果数 k 取为 8 应该是对我们方法而言比较合适的参数取值.但是,出于更好的性能需求考虑,在其它的实验中都暂时将 k 值定为 10.

4.2.4 对问题4的回答

我们设计第四个实验,是为了检验对推荐结果进行二次排序是否会使得本方法的推荐结果的优先级排序和推荐所花费时间更符合用户需求.一个比较理想的推荐方法,用户越满意的推荐项应该排在推荐结果中更加靠前的位置.因此,虽然利用 LSA 对推荐结果进行二次排序并不能显著提高推荐结果的准确率,但能够帮助用户更好的获取想要的推荐结果,提高用户体验.所以,对推荐结果二次排序的必要性进行检验,以及找到最佳的阈值 θ 取值是必要的.

根据 θ 的合适取值,我们将 θ 分别设为 0.4、0.5、0.6、0.7 和 0.8 进行实验.由于对推荐结果进行二次排序并不会对准确率产生影响,所以该实验只在 MRR 和推荐所花费时间上对二次排序前后推荐结果进行对比.二次排序所花费时间只取一个近似整数值,单位为秒（s）.用于测试的实例数共 1000 个.并且由于实验 3 中已经有进行二次排序之前的 MRR 值,所以在此次实验中不需要再进行重复实验.为了控制变量,我们将 n 设为 4,k 设为 10.

表 8 则表明,在推荐结果二次推荐之后,随着 θ 值的变化,MRR 值也随之变化.这符合我们理论上的预测,因为阈值不同,被调整优先级的推荐结果项数也不同.但是,虽然推荐结果调整使得 MRR 越高,表示用户满意的推

荐项排在推荐结果中更加靠前的位置,但是阈值越低也表示需要对比的代码段个数越多,计算结果所花费的时间也越长.最理想的情况是,在 MRR 值有所改进时,二次排序所花费的时间也不会太长.在 θ 值为 0.4 和 0.5 时,虽然 MRR 很高,但是其花费的推荐时间太长,在实际使用中实用性较差.综合考虑 MRR 和二次排序花费时间,我们考虑到在 MRR 指标相差不大的情况下,二次排序所花费的时间越短,实用性越好,所以 θ 设为 0.7 是相对比较好的选择.

Table 8 MRR and time for different θ values after re-ranking

表 8 二次排序后不同 θ 值对应的 MRR 和时间

θ 值	0.4	0.5	0.6	0.7	0.8
排序后 MRR	0.3318	0.3221	0.3186	0.3124	0.3066
时间（s）	7	4	2	1	1

对问题 4 的回答:比较实验 4 的和实验 3 可以看出,对推荐结果进行二次排序确实能改善所推荐的结果,使用户越满意的推荐项应该排在推荐结果中更加靠前的位置.不同的θ取值对推荐结果质量和二次排序花费时间影响都很大.综合考虑之后,出于更好的性能需求考虑,在尽可能提高 MRR 值和减少推荐花费的时间前提下,将 θ 值设为 0.7 应该是一个比较好的选择.

4.3 实用性验证

前面的实例化验证评估侧重于检测方法在准确性和 MRR 等通用推荐衡量指标上的表现.而推荐结果对于开发人员的实用性是另外一个重要的问题,我们通过对 100 名初步学习 Java 软件开发的学生进行调查来研究这个问题.这项调查要求每个参与者对我们的编程现场上下文深度感知的代码行推荐方法的推荐结果有效性进行评分.

调查以问卷的方式进行,要求参与者在尝试使用我们方法的推荐结果协助编程开发之后,从以下四个待选项中选择最符合个人体验的选项:

1、推荐结果比较有用,能够有效帮助开发人员更好地进行程序开发;

2、推荐结果部分可用,但所推荐代码行需要较多的人工修改才完全可用;

3、在大多数的情况下,推荐的代码行无用或不可用;

4、我无法正确地回答这个问题;

我们分析了参与调查问卷的同学反馈回来的意见,发现 100 参与者中有 62 个(62%)选择第一个选项.剩下的 38 个参与者中,选择第二个选项的有 14 个,选择第三个选项的有 24 个(24%).这个调查结果一方面说明我们的方法在实际开发过程中确实能够对开发人员有所帮助,并且在提高数据集的处理程度以提高准确性,和降低数据集的处理程度以保证更好的可用性之间获得较好的平衡,实用性较好.但是,24%的负面评价也说明我们方法的推荐性能仍有很大的提升空间.第三个选项*在大多数的情况下,推荐的代码行无用或不可用*说明在大部分情况下,所推荐的 10 个代码行结果中都没有符合开发人员需求的推荐项,这表明我们的编程现场上下文深度感知的代码行推荐方法的推荐结果准确性并不能令开发人员十分满意,方法性能仍有较大的提升空间.

我们会在接下来的工作中进一步完善本文的方法,一方面尝试获取更多的编程现场信息,以提高方法推荐结果准确性.同时,我们也会尝试对推荐模型方面进行改进,看能否进一步提高我们的编程现场上下文深度感知的代码行推荐方法的性能.

5 有效性威胁

在这部分,我们讨论可能对本文方法的有效性产生威胁的情况.

上下文环境:第一个会对我们的编程现场上下文深度感知的代码行推荐方法的性能产生影响的就是上下文环境,也是最大影响因素.在代码编写过程中,可以将一个具体功能的实现大致分为两个阶段:准备阶段和实现阶段.准备阶段的工作主要是变量声明和初始化等.而实现阶段则通过语句编写实现具体功能.对我们的方法

而言,需要通过对上下文的学习来得到一种上下文关系,进而实现推荐.但是准备阶段的变量声明之间是没有顺序关系的,也就是说如果已有的上下文是准备阶段的代码行,那么很可能会由于缺乏上下文顺序关系,直接威胁到本文提出的方法的性能.而且,在真实的代码编写过程中,准备和实现两个阶段是没有具体界限的,即开发人员可能在实现过程中,发现需要变量时才进行声明和初始化.

在之前的实验中,我们假设只有一些具有重要结构（比如 if）的代码行才对开发人员具有实际使用价值,所以数据集大部分都是具有重要结构的.但是这个问题我们还是会在未来工作中进行进一步研究.

语言环境:我们实验中所有的代码行都是 Java 代码行.虽然从理论上来说,本文的方法适用于所有编程语言,有的编程语言由于具有更加简洁的结构（比如 Python）可能具有更好的效果.但是,由于本文只在 Java 中进行实验,没有进行其它语言的实验,所以编程语言可能也是一个对我们方法有效性产生威胁的因素.将来,我们会扩展本文的方法到其它的语言环境进行测试.

6 相关工作

代码搜索:代码搜索一直以来都是比较受关注的研究问题,按照输入类型来划分,可以将代码搜索分为多个方向:自由文本[30]、代码上下文[31]、结构化信息[32]和其他输入类型[1,33].在上述各个类别的检索技术中,只能检索技术又不断的被引入进来,并与其它的一些相关的信息相结合,如代码之间的调用关系、用户对代码段的反馈信息,这些智能检索技术的使用进一步提高了代码检索结果的质量.早期的代码搜索尝试使用文本相似度或语义相似度对用户的输入查询和代码段进行直接匹配.为了提高性能,一些研究者针对匹配算法提出了新的改进方案,具体来说,2011 年,Hill 等人[34]提出基于词组概念的搜索技术,不同于以往的信息检索方法,该技术还考虑了词在查询中不同的语义角色.Nguyen 等人[35]提出通过线性组合的方法把改进的向量空间模型与词向量进行结合以对代码进行向量表示,该方法比使用单一的向量表示效果更好.为了解决基于直接匹配方法中因为词项失配而影响推荐结果的问题,研究人员进行了很多的相关探索,早期的方法多采用人机交互的方式完成用户查询的获取[36],现在则尝试使用自动化的方法对用户原始查询的质量进行评价,而后借助共现关系[37]、近义词词典[38]以及 API 文档[2,8]等获取扩展词以对原始查询进行扩展.Haiduc 等人[39]提出预测查询的质量并推荐更加合适的查询方法,该方法首先利用一个训练好的分类器来预测一个查询的质量.针对质量较低的查询,利用多种策略对查询进行重写.Hill 等人[37]提出一种基于查询扩展的代码搜索方法,该方法利用了词的共现关系.Lu 等人[38]提出一种基于 WordNet 查询扩展的代码推荐方法,效果更好.Rahman 等人[40]提出一个代码搜索工具 RACK,首先吧自然语言的查询通过众智扩展的方式转换成对应的 API 序列,而后以这些序列作为输入,从 GitHub 中搜索相关的项目段并推荐给用户.

代码生成:计算机自动生成代码是近年来软件工程的研究热点之一.代码自动生成在很大程度上减少了程序员的工作量,提高了开发效率.随着开源社区的发展,我们可以通过分析大量的代码从而进行代码生成.Scott Reed 等人[41]提出神经程序解释器方法,该方法由一种基于循环神经网络的组合神经网络构建,由一个任务无关的循环神经网络内核,一个用于保存键值对的程序存储器和一个与应用领域相关的编码器组成,能够完成加法、排序和图形旋转等简单程序的执行.在神经程序解释器方法的基础上,Chengtao Li 等人[42]提出一种 NPL 方法,该方法也是基于大粒度的高层计算机程序通常由小粒度低层子程序组合而成这一基本原理进行探索.该方法在小规模训练数据集上具有较高的准确率.代码补全任务是较成熟的程序生成应用,传统方法通过对代码静态分析来进行 API 调用、常见代码块的补全.近年来,许多研究者提出利用概率模型对大量的代码片段进行学习从而实现代码补全.常用的模型有 N-gram 模型和神经网络等.Martin Vechev 等人[43]利用 N-gram 和循环神经网络对代码的 API 调用进行补全.Allamanis M 等人[44]提出利用 Log-Bilinear 的模型进行代码补全.他们利用该模型学习代码片段,并且为该代码段推荐更加合理的方法名和类名,这对于代码风格的规范是有意义的.随着深度学习在自然语言处理等领域的成功应用,最近的许多研究都关注于深度学习方法在源代码中的使用,尝试将深度学习引入代码生成和补全中.在代码生成中使用深度学习模型比较常见.比如,Mou 等人[45]提出使用 RNN 编码—解码模型来从自然语言输入中获取用户企图,进而生成用户想要的代码.他们的实验显示了在大规模数

据集上进行代码生成的可能性.深度学习也可以用于做代码补全任务[43,44].比如,White 等人[29]提出将 RNN 语言模型用到源码文件中,并展现了在预测软件分词方面的有效性.

代码推荐:有关代码推荐的研究很多,这是因为代码重用能够提高开发人员工作效率,具有很大的实用价值.一般主要关注代码段[41,45]或者 API 的推荐[7,46].Dang 等人[46]利用获得的编程上下文来查询知识库,以检索推荐的一个或多个代码片段.Jiang 等人[48]利用信息检索和有监督学习推荐代码片段,引入机器学习之后,有检索的代码段推荐效果比简单的文本匹配更好.为了能够将代码段呈现到推荐列表更加靠前的位置,需要对初始的推荐列表进行重排.Wang 等人[48]从推荐结果入手,采取半自动相关反馈方法对推荐的代码段列表进行评价,算法根据人工的评价对推荐结果进行重新排序,把相关的结果尽量排到推荐结果更加靠前的位置.Ishihara 等人[49]提出在推荐代码段时不仅要考虑到查询与代码段之间的文本相似度,还要考虑代码段的受欢迎程度.某个代码段的重用次数越多,表示其受欢迎程度越高.Ponzanelli 等人[50]提出一种代码推荐系统,该系统通过实时查看开发者在浏览器和开发环境中的行为,判断开发者的搜索意图和偏好,进而对推荐结果进行重排,将更好的推荐结果呈现到开发者面前.API 在软件复用以及软件开发平台中占有重要的技术地位,已经成为开发人员完成开发任务时的核心关注点.API 推荐的主要目的是找到满足开发任务需要的 API 或 API 使用代码.为此,研究人员将信息检索和程序分析技术相结合,提出了一系列 API 搜索方法和技术.Stylos 等人[51]开发了一个用于搜索 API 类和方法以及 API 使用样例的网页搜索工具 Mica.Bajracharya 等人[52]设计了一个在大量代码库中搜索 API 使用样例的工具 SAS,可以对 API 搜索结构过滤后以代码片段视图的方式展示搜索结果,并使用标签云对常用的 API 信息进行可视化以提高搜索效率.从发发展趋势看,API 的研究已经从帮助开发人员找到合适的 API 发展到帮助他们正确的使用 API.Gu 等人[7]针对用户输入的自然语言查询,推荐实现查询功能所需的 API 序列.Xu 等人[47]不仅推荐 API,而且找出这些 API 方法最可能被使用的位置,具有更好的实用性.

7 结论

本文将深度学习方法应用到编程现场的代码行的推荐使用中.具体来说,首先收集编程现场的信息,包括编程现场任务数据和代码行上下文数据.然后对数据格式进行处理,并将得到具有统一形式的数据存储.模型训练过程就是利用循环神经网络编码—解码模型对编程现场的代码行上下文使用模式进行深度学习,并根据已有的编程现场上下文对当前代码行进行推荐.学习的使用的数据就是已经统一化格式的源码数据.能够对代码行上下文使用模式进行深度学习的先决条件是,获取大规模代码行上下文数据集,并进行统一化的处理.

在推荐过程中,开发现场数据采集模块自动采集用户已经输入的代码行数据,经由现场数据清洗模块清洗之后,和训练中使用的数据具有统一的形式.然后就可以作为已经训练好的循环神经网络编码—解码模型的输入,并利用集束搜索算法,得到初次排序后的 N 个代码行推荐结果.为了使开发人员越满意的推荐项排在推荐结果中更加靠前的位置,我们对集束搜索得到的 N 个推荐结果进行二次排序,使用的数据是开发现场数据采集模块收集到的编程现场任务数据.利用编程现场任务数据,将推荐结果中已经在训练数据中出现过的推荐项调整到推荐结果中更加靠前的位置.相似度衡量使用 LSA,相似度越高,优先级越高.我们的实例验证表明,本文的方法确实是有效的.并且,方法性能优于其它的类似方法,比如代码搜索.

未来,我们会进一步完善本文的方法,并研究深度学习方法在编程语言的编程现场中的其它使用,包括其它的编程语言,以及从方法块到 API 的不同粒度的应用问题.

References:

[1] Kim J, Lee S, Hwang S W, et al. Towards an Intelligent Code Search Engine[C]// Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, Usa, July. DBLP, 2010.

[2] Holmes R, Murphy G C. Using structural context to recommend source code examples[C]// International Conference on Software Engineering. 2005:117-125.

[3] Hainry E, Péchoux R. A Type-Based Complexity Analysis of Object Oriented Programs[J]. 2018.

[4] Brandt J, Dontcheva M, Weskamp M, et al. Example-centric programming:integrating web search into the development environment[C]// Sigchi Conference on Human Factors in Computing Systems. ACM, 2010:513-522.

[5] Holmes R, Cottrell R, Walker R J, et al. The end-to-end use of source code examples: An exploratory study[C]// IEEE International Conference on Software Maintenance. IEEE, 2009:555-558.

[6] Robbes R, Lanza M. Improving code completion with program history[J]. Automated Software Engineering, 2010, 17(2):181-212.

[7] Gu X, Zhang H, Zhang D, et al. Deep API learning[J]. 2016.

[8] Lv F, Zhang H, Lou J G, et al. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E)[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE, 2016:260-270.

[9] Lecun Y, Bengio Y, Hinton G. Deep learning[J]. Nature, 2015, 521(7553):436.

[10] Tran V K, Le M N, Tojo S, et al. Neural-based Natural Language Generation in Dialogue using RNN Encoder-Decoder with Semantic Aggregation[C]// Sigdial. 2017.

[11] Github. https://github.com.

[12] Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate[J]. Computer Science, 2014.

[13] Cho K, Van Merrienboer B, Gulcehre C, et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation[J]. Computer Science, 2014.

[14] Allamanis M, Tarlow D, Gordon A D, et al. Bimodal modelling of source code and natural language[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2015:2123-2132.

[15] Sundermeyer M, Schlüter R, Ney H. LSTM Neural Networks for Language Modeling[C]// Interspeech. 2012:601-608.

[16] Sutskever I, Vinyals O, Le Q V. Sequence to Sequence Learning with Neural Networks[J]. 2014, 4:3104-3112.

[17] Eclipse JDT. http://www.eclipse.org/jdt/.

[18] Java Decompiler http://jd.benow.ca/.

[19] Suzuki J, Nagata M. RNN-based Encoder-decoder Approach with Word Frequency Estimation[J]. 2016.

[20] Mikolov T, Sutskever I, Chen K, et al. Distributed representations of words and phrases and their compositionality[J]. Advances in neural information processing systems, 2013, 26:3111-3119.

[21] Firat O, Cho K, Bengio Y. Multi-Way, Multilingual Neural Machine Translation with a Shared Attention Mechanism[C]// Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2016:866-875.

[22] Swietojanski P, Renals S. Learning hidden unit contributions for unsupervised speaker adaptation of neural network acoustic models[C]// Spoken Language Technology Workshop. IEEE, 2015:171-176.

[23] Desnos A, Gueguen G. Android: From Reversing to Decompilation[J]. Proc of Black Hat Abu Dhabi, 2011.

[24] Wongsuphasawat K, Smilkov D, Wexler J, et al. Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow[J]. IEEE Transactions on Visualization & Computer Graphics, 2017, PP(99):1-1.

[25] Bennell J A, Cabo M, Martínez-Sykora A. A beam search approach to solve the convex irregular bin packing problem with guillotine guts[J]. European Journal of Operational Research, 2018.

[26] Nan L I, Tao H C. A Novel News Summary Algorithm Combining BM25 and Text Features[J]. Journal of Chengdu University of Information Technology, 2018.

[27] Xu W, Sun X, Xia X, et al. Scalable Relevant Project Recommendation on GitHub[C]// Asia-Pacific Symposium on Internetware. ACM, 2017:1-10.

[28] Nguyen T T, Nguyen A T, Nguyen H A, et al. A statistical semantic language model for source code[C]// Joint Meeting on Foundations of Software Engineering. 2013:532-542.

[29] White M, Vendome C, Linares-Vasquez M, et al. Toward Deep Learning Software Repositories[C]// Mining Software Repositories. IEEE, 2015:334-345.

[30] Keivanloo I, Rilling J, Zou Y. Spotting working code examples[C]// Proceedings of the 36th International Conference on Software Engineering. ACM, 2014:664-675.

[31] Galenson J, Reames P, Bodik R, et al. CodeHint: dynamic and interactive synthesis of code snippets[M]. ACM, 2014.

[32] Reiss S P. Semantics-based code search demonstration proposal[C]// IEEE International Conference on Software Maintenance. IEEE, 2009:385-386.

[33] Bajracharya S, Ossher J, Masiero P C, et al. A test-driven approach to code search and its application to the reuse of auxiliary functionality[J]. Information & Software Technology, 2011, 53(4):294-306.

[34] Hill E, Pollock L, Vijayshanker K. Improving source code search with natural language phrasal representations of method signatures[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE, 2011:524-527.

[35] Nguyen T V, Nguyen A T, Phan H D, et al. Combining Word2Vec with Revised Vector Space Model for Better Code Retrieval[C]// International Conference on Software Engineering Companion. IEEE Press, 2017:183-185.

[36] Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of NL-queries for software maintenance and reuse[C]// IEEE, International Conference on Software Engineering. IEEE, 2009:232-242.

[37] Roldan-Vega M, Mallet G, Hill E, et al. CONQUER: A Tool for NL-Based Query Refinement and Contextualizing Code Search Results[C]// IEEE International Conference on Software Maintenance. IEEE Computer Society, 2013:512-515.

[38] Lu M, Sun X, Wang S, et al. Query expansion via WordNet for effective code search[C]// IEEE, International Conference on Software Analysis, Evolution and Reengineering. IEEE, 2015:545-549.

[39] Haiduc S, Rosa G D, Bavota G, et al. Query quality prediction and reformulation for source code search: The Refoqus tool[J]. 2013:1307-1310.

[40] Rahman M M, Roy C K, Lo D. RACK: Automatic API Recommendation Using Crowdsourced Knowledge[C]// IEEE, International Conference on Software Analysis, Evolution, and Reengineering. IEEE, 2016:349-359.

[41] Reed S, Freitas N. Neural Programmer-Interpreters[C]// ICML. 2016.

[42] Li C, Tarlow D, Gaunt A L, et al. Neural program lattices[J]. 2016.

[43] Raychev V, Vechev M, Yahav E. Code completion with statistical language models[C]// ACM, 2014:419-428.

[44] Allamanis M, Barr E T, Bird C, et al. Suggesting accurate method and class names[C]// Joint Meeting on Foundations of Software Engineering. ACM, 2015:38-49.

[45] Mou L, Men R, Li G, et al. On End-to-End Program Generation from User Intention by Deep Neural Networks[J]. Computer Science, 2015.

[46] Dang Y, Zhong C, Wu Q, et al. Code recommendation[J]. 2016.

[47] Xu C, Sun X, Li B, et al. MULAPI: Improving API Method Recommendation with API Usage Location[J]. Journal of Systems & Software, 2018.

[48] Wang S, Lo D, Jiang L. Active code search: incorporating user feedback to improve code search relevance[C]// ACM/IEEE International Conference on Automated Software Engineering. ACM, 2014:677-682.

[49] Ishihara T, Hotta K, Higo Y, et al. Reusing reused code[J]. 2013.

[50] Ponzanelli L, Scalabrino S, Bavota G, et al. Supporting Software Developers with a Holistic Recommender System[C]// Ieee/acm, International Conference on Software Engineering. IEEE, 2017:94-105.

[51] Stylos J, Myers B A. Mica: A Web-Search Tool for Finding API Components and Examples[C]// Visual Languages and Human-Centric Computing. IEEE Computer Society, 2006:195-202.

[52] Bajracharya S, Ossher J, Lopes C. Searching API usage examples in code repositories with sourcerer API search[C]// ICSE Workshop on Search-Driven Development: Users, Infrastructure, TOOLS and Evaluation. ACM, 2010:5-8.

附中文参考文献:

[1] 黎宣,王千祥,金芝.基于增强描述的代码搜索方法.软件学报,2017,28(6):1405−1417. http://www.jos.org.cn/ 1000- 9825/5226.htm.

[2] 李正,吴敬征,李明树.API 使用的关键问题研究.软件学报,2018,29(6):1716−1738. http://www.jos.org.cn/1000- 9825/5541.htm