软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):3015−3034 [doi: 10.13328/j.cnki.jos.006355]

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

代码自然性及其应用研究进展*

陈浙哲[1,2], 鄢 萌[1,2], 夏 鑫[3], 刘忠鑫[4], 徐 洲[1,2], 雷 晏[1,2]

[1](信息物理社会可信服务计算教育部重点实验室(重庆大学), 重庆 400044)

[2](重庆大学 大数据与软件学院, 重庆 401331)

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

[4](浙江大学 计算机科学与技术学院, 浙江 杭州 310007)

通信作者: 鄢萌, E-mail: mengy@cqu.edu.cn

摘 要: 代码自然性(code naturalness)研究是自然语言处理领域和软件工程领域共同的研究热点之一, 旨在通过构建基于自然语言处理技术的代码自然性模型, 以解决各种软件工程任务. 近年来, 随着开源软件社区中源代码和数据规模的不断扩大, 越来越多的研究人员注重钻研源代码中蕴藏的信息, 并且取得了一系列研究成果. 但与此同时, 代码自然性研究在代码语料库构建、模型构建和任务应用等环节面临许多挑战. 鉴于此, 从代码自然性技术的代码语料库构建、模型构建和任务应用等方面对近年来代码自然性研究及应用进展进行梳理和总结. 主要内容包括: (1) 介绍了代码自然性的基本概念及其研究概况; (2) 归纳目前代码自然性研究的语料库, 并对代码自然性模型建模方法进行分类与总结; (3) 总结代码自然性模型的实验验证方法和模型评价指标; (4) 总结并归类了目前代码自然性的应用现状; (5) 归纳代码自然性技术的关键问题; (6) 展望代码自然性技术的未来发展.

关键词: 代码自然性; 软件仓库挖掘; 代码语言模型

中图法分类号: TP311

中文引用格式: 陈浙哲, 鄢萌, 夏鑫, 刘忠鑫, 徐洲, 雷晏. 代码自然性及其应用研究进展. 软件学报, 2022, 33(8): 3015−3034. http://www.jos.org.cn/1000-9825/6355.htm

英文引用格式: Chen ZZ, Yan M, Xia X, Liu ZX, Xu Z, Lei Y. Research Progress of Code Naturalness and Its Application. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 3015−3034 (in Chinese). http://www.jos.org.cn/1000-9825/6355.htm

Research Progress of Code Naturalness and Its Application

CHEN Zhe-Zhe[1,2], YAN Meng[1,2], XIA Xin[3], LIU Zhong-Xin[4], XU Zhou[1,2], LEI Yan[1,2]

[1](Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing 400044, China)

[2](School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China)

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

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

Abstract: The study of code naturalness is one of the common research hotspots in the field of natural language processing and software engineering, aiming to solve various software engineering tasks by building a code naturalness model based on natural language processing techniques. In recent years, as the size of source code and data in the open source software community continues to grow, more and more researchers are focusing on the information contained in the source code, and a series of research results have been achieved. While at the same time, code naturalness research faces many challenges in code corpus construction, model building, and task application. In view of this, this paper reviews and summarizes the progress of code naturalness research and application in recent years in terms of

* 基金项目: 国家自然科学基金(62002034); 中央高校基本科研业务费(2020CDCGRJ072, 2020CDJQYA021, 2021CDJKYJH032); 国防基础科研计划(WDZC20205500308); 中国博士后基金(2020M673137); 重庆市自然科学基金(cstc2020jcyj-bshX0114)
收稿时间: 2021-01-29; 修改时间: 2021-03-25, 2021-04-14; 采用时间: 2021-04-27; jos 在线出版时间: 2021-05-20

code corpus construction, model construction, and task application. The main contents include: (1) Introducing the basic concept of code naturalness and its research overview; (2) The current corpus of code naturalness research is summarized, and the modeling methods for code naturalness are classified and summarized; (3) Summarizing the experimental validation methods and model evaluation metrics of code naturalness models; (4) Summarizing and categorizing the current application status of code naturalness; (5) Summarizing the key issues of code naturalness techniques; (6) Prospecting the future development of code naturalness techniques.

Key words: code naturalness; mining software repositories; code language model

代码自然性指代码尽管是用人工语言(如C或Java)编写的, 但也属于自然产物, 与自然语言一样具有可预测的统计特性, 这些特性可以在统计语言模型中捕获并用于解决特定的软件工程任务[1]. 代码自然性中的“自然”一词源于自然语言处理(natural language processing, NLP)领域[2−5], 该领域是计算机科学领域与人工智能领域中的一个重要方向, 其目标是自动处理自然语言中的文本, 用于诸如翻译、总结、理解和语音识别等任务. Hindle 等人[1]认为, NLP 发展背后的一个基本事实是: 自然语言可能是复杂的, 并且有大量丰富的表达, 但大部分是符合一定规则并可预测的, 软件代码具有同样的特性. 尽管理论上编程语言是复杂的、灵活的和强大的, 但事实上, 开发人员编写的程序大多是简单的和具有一定重复性的[6,7]. 研究表明: 代码的复用率极高, 比自然语言更甚[1].

代码自然性理论的提出, 使得软件工程任务可以从不同的角度入手解决[1], 研究者们提出了一系列基于代码自然性的应用技术以解决不同的软件工程问题[5,29]. 利用代码自然性不仅能够提升传统方法的性能[30]、简化困难的任务[19], 还可以增加现有工具的统计信息[8], 支持新的软件工程工具. 对于传统统计语言模型. Hindle 等人[1]在 ICSE 中使用 *n*-gram 语言模型, 大大提升了 Eclipse 自带的代码补全插件的性能. Hellendoorn 等人[31]在 FSE 中论证发现: 针对源代码优化后的 *n*-gram 模型, 在代码补全、缺陷预测问题上可以产生相当甚至超过基于循环神经网络(RNN)和长短期记忆神经网络(LSTM)的神经语言模型的效果. 之后, 研究者们提出了一系列基于统计语言模型的技术和方法, 将代码自然性应用于代码补全[1,8,9,12]、缺陷预测[8,16,18,32]、程序修复[20]、API 代码建议[33]、代码摘要[34]等软件工程任务.

基于传统统计语言模型的代码自然性建模方法深入应用的同时, 越来越多的学者将神经语言模型应用于该领域, 以更好地刻画代码自然性[30,35−60]. 神经语言模型中的深度前馈结构, 如卷积神经网络(CNN), 可以有效捕获代码编码中的丰富结构模式; 而循环结构, 如 RNN, 则能捕获比统计语言模型更长的代码上下文. 用神经网络训练语言模型是由 Bengio[61]率先提出的. 在代码自然性领域, Corley 等人[48]探索使用一种特殊的深度学习模型——document vectors (DVs)来进行特征定位. Dam 等人[35]提出了一种利用长短期记忆神经网络(一种特殊的 RNN)建立软件代码语言模型的新方法, 用于解决软件工程领域代码补全问题.

近年来, 虽然代码自然性及其应用研究取得了一定进展, 但仍存在不少问题. Rahman 等人[17]关于代码自然性的实证研究表明: 删除语料库中的语法标记(syntax tokens)包括分隔符、编程语言关键字、运算符后, 代码的重复度远没有 Hindle 等人[1]所论述的那样高. 此外, Rahman 等人的实验结果还表明: Java API 的使用重复性非常高, 比不包含语法标记的 Java 代码更具重复性和可预测性. Karampatsis 等人[38]关于代码词汇量的研究发现, 词汇量大和词汇量不足问题都将严重影响源代码神经语言模型的性能. 可见, 代码自然性研究仍需进一步探索.

然而, 目前对代码自然性及其应用研究进展缺乏系统性地梳理. 尽管基于代码自然性的不同应用之间的区别很大, 但它们都基于同一个基本思路: 捕获语料库中代码的自然性规律, 再将其应用于软件工程不同任务. 鉴于此, 本文拟针对当前代码自然性及其应用研究进展, 从数据集构建、模型构建、验证方法、评估指标和应用现状等方面进行梳理和归纳, 进而总结现有的与代码自然性相关的技术和应用, 探讨当前该领域存在的关键问题, 展望该领域未来的研究发展趋势.

本文采用以下流程完成对文献的索引与选取. 文献选取标准如下: 该文献针对代码自然性提出新理论、新技术, 或者该文献将代码自然性相关理论和技术应用于软件工程任务中. 此外, 该文献应公开发表在期刊、会议、技术报告或书籍中. 依据以上标准, 本文通过以下 3 个步骤对文献进行检索和筛选.

(1) 本文选用 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 naturalness, software naturalness, code language model, naturalness 等. 同时, 在标题、摘要、关键词和索引中进行检索;

(2) 本文对软件工程领域的主要期刊与会议进行在线搜索, 具体包括 TSE, ICSE, FSE/ESEC, ASE, ICSME, MSR, SANER 等, 搜索时间从 2012 年(即针对"代码自然性"的首次提出时间)开始至 2020 年 9 月;

(3)为避免遗漏相关研究, 本文在上述步骤的基础上, 对所获取文献集合中的文献逐一查看, 进一步筛选出与代码自然性相关的文献. 筛选方式为: 查看每篇文献的摘要, 若仍不能分辨, 则进一步查看该文献使用的模型和框架是否符合以下两个要求之一: 1. 用自然语言处理技术对代码进行建模; 2. 将代码自然性特征用于软件工程任务. 以此尽可能全地选取与代码自然性相关的文献. 通过该步骤, 本文发现部分代码自然性及其应用相关文献发表于人工智能和自然语言处理领域的期刊和会议上, 包括 IJCAI, ACL 等.

基于上述选取原则和检索步骤, 共计 44 篇文献纳入本文后续文献总结中, 其中: 22 篇文献与代码自然性直接相关, 这些文献提出了与代码自然性相关的新理论或新模型; 另外 22 篇文献将代码自然性相关理论或技术应用于实际研究中. 上述文献分布情况在图 1 中展示, 发表过相关研究较多的期刊与会议有: ICSE 论文 11 篇, FSE 论文 9 篇, ACL 论文 3 篇, IJCAI 论文 3 篇, MSR 论文 3 篇, SANER 论文 3 篇, ICML 论文 2 篇, TSE 论文 2 篇, ASE 论文 2 篇, AAAI 论文 2 篇, 其他期刊和会议论文共 4 篇.

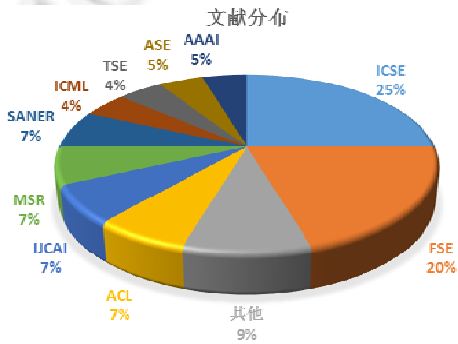

图 1 文献分布

本文第 1 节介绍代码自然性的概念及其基本概况. 第 2 节介绍代码自然性的建模方法, 从数据集构建、模型构建、验证方法和评价指标这 4 个方面进行归纳和梳理. 第 3 节介绍代码自然性的应用进展, 对代码补全、缺陷预测、摘要生成、API 建议和程序修复等代表性文献进行了总结. 第 4 节总结代码自然性领域面临的关键问题, 并分析该领域未来的发展趋势. 第 5 节是对本文的总结.

1 代码自然性概念及其概况

1.1 代码自然性基本概念

为了便于阐述, 本文将统一使用以下与代码自然性相关的基本概念定义.

(1) 标识符(token): 用户编辑时使用的名字, 用于给变量、常量、函数、语句块等命名, 以建立起名称与使用间的关系;

(2) 自然性(naturalness): 编程语言拥有和自然语言相似的特性, 能够进行统计和预测;

(3) 自然性应用(natural application): 将基于代码自然性的相关理论和技术应用于软件工程任务中;

(4) 统计语言模型(statistical language model): 从概率统计角度出发, 解决自然语言上下文相关特性的数学模型, 其核心就是判断一个句子在文本中出现的概率;

(5) 神经语言模型(neural language model, NLM): 一类用来克服维数灾难的语言模型, 它使用词的分布式表示对自然语言序列建模, 并且允许模型处理具有类似共同特征的词.

1.2 代码自然性研究概况

代码自然性研究可以简单归结为以下两个问题: 一是如何建立高效的模型来获取代码的规律; 二是得到规律后如何进行分析和应用. 第 1 个问题即研究代码自然性问题, 第 2 个问题即在代码自然性的基础上进行应用. 代码自然性应用离不开代码自然性相关理论的支撑; 代码自然性相关理论为代码自然性应用服务. 因此, 本文不仅聚焦于代码自然性的研究, 还将归类代码自然性的应用现状.

其实, 早在代码自然性这一概念提出之前, 程序员在日常编程工作中就已经发现大量的代码片段往往会再次出现. 2010 年, Gabel 等人[6]在一项大规模的代码研究中首次发现并报告了这一事实. 正如我们可以想到的, 考虑一个集成开发环境(IDE), 程序员在其中输入了部分语句:“for (*i*=0; *i*<10”. 在这种情况下, IDE 向程序员建议完成“;*i*++){”是很合理的. 因为 for 循环语句常常会大量出现在程序员的代码中.

2012 年, Hindle 等人[1]首次提出编程语言也是自然语言的一种, 它具有可重复并带有可预测的统计学规律, 并在 ICSE 中提出了“代码自然性”这一概念. Hindle 等人[1]使用自然语言中常用的统计语言模型 *n*-gram 证实了代码也可以被语言模型有效地建模, 改进了 Eclipse 的内置代码补全功能, 面向 java 语言开发了一个简单的代码补全引擎, 开启了基于代码自然性的应用先河. 2014 年, Tu 等人[8]首次提出源代码是局部重复的, 即其有有用的局部规则, 可以被局部缓存模型所捕获, 并用于软件工程任务. Ray 等人[19]在研究中发现: 带有错误的代码往往更不自然, 而得到修复后的代码会变得自然. 因此, 语言模型可以帮助锁定潜在的缺陷代码, 甚至建议使用“更自然”的代码替换缺陷代码. 2018 年, Allamanis 等人[62]针对代码自然性中的机器学习方法进行了综述, 系统性地介绍了源代码概率模型建模方法. 与本文的区别在于: 本文聚焦代码自然性及其应用研究进展; Allamanis 等人聚焦源代码概率模型, 从文献选取、文献归类与分析、未来展望等侧重点均有所不同. 例如: 本文的文献选取围绕代码自然性及其应用挑选, 而 Allamanis 等人围绕建模方法挑选; 本文介绍了代码自然性研究的语料库构建、实验验证方法、评价指标等, 并对代码自然性及其应用研究中存在的关键问题和未来发展方向进行了梳理, 而 Allamanis 等人没有提及.

2 代码自然性建模方法

2.1 数据集构建

代码自然性研究的数据集一般选自使用广泛的或者需要研究的项目所组成的语料库. Hindle 等人[1]使用了两个代码语料库: 一个是 Java 项目的集合, 一个是 Ubuntu 的应用程序集合. Ray 等人[19]选择来自不同领域的项目. Nguyen 等人[30]选择使用历史悠久、并且被广泛使用的开源 Java 项目. 不管研究者们选择了哪些项目作为实验的语料库, 其作用归根结底就是为实验提供构建语言模型的数据(见表 1).

表 1 开源语料库总结

语料库名称	编程语言	项目数量	公开时间	链接	使用此语料库的文献	语料库构建原则
GitHub Java Corpus	Java	14 785	2017.1.10	http://groups.inf.ed.ac.uk/cup/javaGithub/	[15,31,38]	首先, 该代码语料库的组成项目选自 Github 中被分享(fork)和关注(star)较多的项目, 从中再手动选择原始源的项目, 以保证所选项目的代表性和唯一性
Python Dataset	Python	949	2017.1.18	https://github.com/uclnlp/pycodesuggest/tree/master/data	[63]	
Raw C Code Corpus	C	4 601	2020.1.27	https://zenodo.org/record/3628775#.X883s4gza00	[38]	
Raw Python Code Corpus	Python	27 535	2020.1.27	https://zenodo.org/record/3628784#.X883togza00	[38]	
Python Dataset	Python	15 000	2016	https://www.sri.inf.ethz.ch/py150	[39,64–66]	该代码语料库的项目通过删除重复文件和项目分支, 仅保留进行解析的程序组成, 其构建原则遵循全面、多样、数据量大
Javascript Dataset	Javascript	15 000	2016	https://www.sri.inf.ethz.ch/js150	[39,64–66]	

2.2 模型构建

现有代码自然性技术模型构建方法主要包括两种类型: 统计语言模型和神经语言模型. 本节将对代码自然性模型构建的一般过程及以上两种建模方法进行介绍.

2.2.1 代码自然性模型构建的一般过程

语言模型构建之前需要准备数据(即语料库构建), 其建模的目的在于让计算机读懂源代码(即数据处理)、使用源代码(即模型构建)、学习如何得到更准确的概率(即平滑处理), 使模型应用于相应的软件工程任务(即模型应用), 最后给出相应任务的结果(即结果呈现). 其一般过程包括(如图 2 所示):

(1) 代码语料库构建: 选自使用广泛和需要研究的项目或者前人使用过的语料库, 其构建原则基本遵循全面、多样、数据量大;

(2) 数据预处理: 对原始语料库进行代码表征和特征提取. 对于统计语言模型而言, 将隐藏在源代码纯文本中的特征进行提取和表征, 对代码进行信息提取和结构化表示, 其目的在于方便进行词频统计, 生成用于估计统计语言模型的标记序列; 对于神经语言模型而言, 对原始语料库进行降噪、驼峰分词、去停用词(词干提取)等步骤后进行词嵌入得到词向量, 并将词向量映射到连续的空间内, 此时已解决数据稀疏问题;

(3) 代码自然性模型构建: 将处理好的语料库应用于代码自然性模型. 对于统计语言模型而言, 根据代码样本估计概率分布, 通过对语料的统计, 得到代码语句的条件概率. 并通过平滑技巧解决数据稀疏问题, 得到最终的代码自然性模型; 对于神经语言模型而言, 将得到的词向量输入选定的神经网络中(如 CNN[67], DNN[30], RNN[38])进行训练, 经过适当调参即可得出最终的代码自然性模型;

(4) 模型应用与评估: 将训练好的语言模型与实际的软件工程任务相结合, 对不同的任务发挥不同的效果. 如代码补全任务将利用语言模型进行补全, 缺陷预测任务则利用语言模型检测代码中的缺陷;

(5) 结果呈现: 将上述环节得到的最终结果以恰当的方式返回给开发人员进行选择和使用.

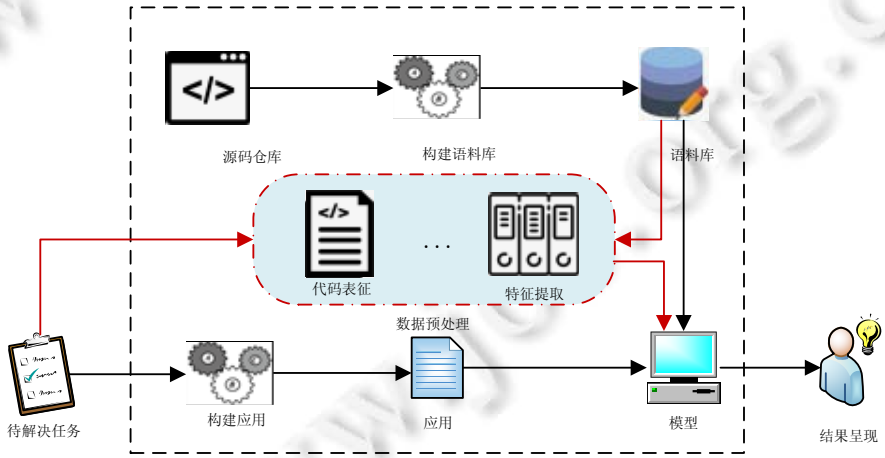

图 2 代码自然性建模的一般过程

2.2.2 基于统计语言模型的代码自然性建模

统计语言模型是自然语言处理的基础模型, 是从概率统计角度出发, 解决自然语言上下文相关特性的数学模型, 其核心是判断一个句子在文本中出现的概率. 随着越来越多项目的源代码被公开和 Gabel 等人[6]发现软件中存在的句法冗余, 越来越多的研究者将统计语言模型应用于编程语言. 统计语言模型的选择通常由实用性驱动: 使用的容易程度和对应的软件工程任务.

现有工作中, 基于序列[1,8,14]、基于树[68]、基于图[33]的统计语言模型甚至隐马尔可夫模型[69]均有研究者在使用, 以应用于各种软件工程任务. 考虑其实用性, 最常用的统计语言模型是基于序列的模型——*n*-gram 语

言模型. 其假设是第 n 个词出现的概率只与前面 n–1 个词有关. n-gram 模型利用一句话中各个词出现概率的乘积来估计该句子出现的概率. 随着该领域的快速发展, 越来越多的研究人员对原始 n-gram 语言模型进行改进, 在基础的 n-gram 模型上增加额外的缓存组件, 以提高其性能或更有针对性地应用于软件工程.

Hindle 等人[1]首次提出代码跟自然语言一样, 也是重复可预测的, 因此它们具有有用的可预测统计属性, 这些属性可以在统计语言模型中捕获并用于软件工程任务. 为了证实以上观点, Hindle 等人使用最常用的统计语言模型 n-gram 对编程语言语料库进行建模, 捕获规律, 从而得出代码甚至比自然语言更加重复的结论.

Tu 等人[8]在 Hindle 等人的研究基础上提出了缓存语言模型, 其中包括标准的 n-gram 语言模型和添加的“缓存”组件, 两者用于捕获不同的规律. 此模型的提出是为了解决 n-gram 语言模型只考虑语料库全局、忽略文件或项目内部, 从而可能无法捕获最正确的规律, 导致准确率降低的问题. 其工作背后的主要思想是: 将一个全局(静态)语言模型与一个从邻近的局部上下文估计的局部(动态)模型相结合, 以提高语言模型的整体性能.

Tonella 等人[14]提出了一种新的基于 Interpolated n-gram 语言模型的测试用例推导方法. Tonella 指出: 当 n 增加时, 虽然 n-gram 语言模型可以提供更长的上下文, 但它在很大程度上变得不完整. 为了解决这个问题, Tonella 等人在 n-gram 语言模型原有的推荐概率基础上增加了事件条件概率, 改变长上下文的权重以达到更好的效果. 并证实了 Interpolated n-gram 语言模型在可行性和覆盖率方面都优于普通 n-gram 语言模型和传统方法的测试用例派生方法.

Allamanis 等人[15]基于 3.5 亿行 Java 代码构建了 Giga-token 概率语言模型, 是第一个基于源代码构建的千兆标识符模型. 此模型的主要改进之处在于, 其工作量(使用的代码行数)是 Hindle 等人构建的 n-gram 语言模型工作量的 100 倍之多. Allamanis 等人发现: 在给定足够数据的情况下, n-gram 语言模型可以尽可能多地学习代码的语法结构, 因此该模型的代码捕获能力比小规模模型好得多.

Hellendoorn 等人[31]提出: 通过一些精心的工程设计, 传统的建模方法也可以击败深度学习模型. 并指出: 若人为地限制词汇量, 也许可以提高内在表现, 但并不会提高建议任务的实际表现. 鉴于此, Hellendoorn 等人提出一种使用 Jelinek-Mercer 平滑方法的可动态更新的、嵌套范围的、基于无限词汇数的语言模型, 当使用非参数语言建模时, 该模型可以获得同类最佳的性能.

基于统计语言模型的代码自然性建模方法见表 2.

表 2 基于统计语言模型的代码自然性建模方法

模型名称	改进方法	描述	动机	代表文献
N-gram	–	N-gram 语言模型可以捕获代码的规律	利用自然软件的大语料构建统计语言模型来帮助一系列不同的软件工程任务	[1]
Cache N-gram	在原始 n-gram 基础上增加一个“缓存”组件	N-gram 语言模型捕获全局(语料库)规律, 缓存组件捕获局部(文件或项目)规律, 两者结合达到更好的效果	为了解决 n-gram 语言模型只能捕获全局规律而无法照顾到局部规律的问题	[8]
Interpolated N-grams	改进了概率的计算方法, 为权值选择了一个指数增长的函数	当 n 增大时, 需要对 n-gram 统计量进行插值, 以弥补 n 值较大时缺失的 n 元组	当 n-gram 语言模型的 n 增加时, 赋予较长上下文更高的权重	[14]
Giga-token Model	将语料库扩大了 N 倍	基于更大语料库构建的语言模型, 其统计性能提高, 能够尽可能地了解代码的语法结构	在足够大的语料库上构建语言模型不仅可以减轻 OOV 问题还可以让模型尽可能地学习到代码之间的关系	[15]
Open-vocabulary Neural Language Model	改进了 Tu 等人的缓存模型, 使用开放词汇表, 利用源代码的特殊属性进行模型构建	针对源代码的特殊性, 从开放词汇、范围嵌套、动态更新这 3 个方面构建传统的语言模型	为源代码仔细调整 n-gram 模型可以产生甚至超过基于 RNN 和 LSTM 的深度学习模型的性能	[31]

2.2.3 基于神经语言模型的代码自然性建模

神经语言模型是一类用来克服维数灾难的语言模型, 它使用词的分布式表示对自然/编程语言序列建模. 不同于 *n*-gram 语言模型, 神经语言模型能够更好地捕捉单词间的相似性和差异性. 此外, 神经语言模型可以处理更长的历史记录, 对于给定大小的训练集, 神经语言模型比 *n*-gram 语言模型具有更高的预测精度. 但是这种性能的提高也是有代价的: 其训练速度明显慢于传统语言模型.

自神经网络语言模型(neural network language model, NNLM)和代码自然性被提出, 越来越多的研究人员发现, 可以用“自然”的方法完成软件工程任务. 此后, 神经语言模型得到进一步的发展, 用于解决一系列的软件工程问题.

Nguyen 等人[30]提出一种用于集成语法和类型上下文的多原型 DNN 语言模型 Dnn4C, 该模型主要用于解决统计语言模型处理程序和应用接口元素(类和方法调用)名称中的歧义问题. Dnn4C 可以在语法和类型层面进行建模, 以补充词汇代码元素. Nguyen 等人设计了一种结合上下文的方法, 用于源代码的语法和类型注释, 以便 Dnn4C 模型学会在不同的语法和类型上下文中区分词法标记. 实验表明, 该模型在处理名称歧义方面具有比当时其他最先进的模型更好的效果.

Dam 等人[35]提出: 现有的语言模型, 例如 *n*-gram 语言模型, 虽然在用于捕获重复的代码时是有用且直观的, 但该模型在长距离依赖建模方面性能较差. 递归结构如 RNN 虽然具有比 *n*-grams 语言模型捕获更长上下文的潜力, 却又容易受到梯度消失或爆炸的影响, 因此很难训练长序列. 鉴于此, Dam 等人提出了建立在基于长短期记忆神经网络的深度学习架构之上的 LSTM 语言模型以解决这个问题, 这种架构能够学习软件代码中经常出现的长期依赖关系.

Karampatsis 等人[38]提出: 统计语言模型目前面临的最严峻的问题是 OOV(out-of-vocabulary)问题, 随着新的标识符名称的激增, 代码引入新词汇的速度远远高于自然语言. 因此他们提出, 寻找方法将神经语言模型扩展到更大的软件语料库是一个非常重要的目标. 鉴于此, Karampatsis 等人提出建立开放词汇表 NLM 以降低 OOV 问题带来的影响. 他们建立了第一个用于源代码的 BPE NLM, 此模型利用 BPE 降低词汇表的大小, 并成功地预测 OOV 词汇.

Bhoopchand 等人[63]提出: 目前的 IDE 能够对于静态类型语言提供良好的建议, 但由于它们对类型注释的依赖, 使其无法像静态类型语言那样为动态编程语言提供支持. 鉴于此, Bhoopchand 等人构建了一个包含 41 亿行代码的 Python 语料库, 提出一种具有稀疏指针网络的神经语言模型, 捕获更长更远的依赖关系, 为动态编程语言提供有效的建议. 此外, Bhoopchand 等人将该神经语言模型的性能与 LSTM, *N*-gram 语言模型进行比较, 得到了更好的应用效果.

Rabinovich 等人[70]提出: 语义解析和代码生成等任务具有一定的挑战性, 因为它们是结构化的(输出必须是格式良好的), 但是不同步的(输出结构与输入结构不同). 针对这一问题, Rabinovich 等人引入了抽象语法网络(ASN), 它提供一种模块化的编码器-解码器体系结构, 特别适用于存在递归分解的情况, 并且提供与输出的内在结构极其相似的简单解码过程. 该模型适应于各种具有结构化输出空间的任务.

Dowdell 等人[71]发现: 尽管 LSTM 和 GRU 被认为是基于 RNN 的模型顶峰, 但这两者都无法解决长期依赖问题; 而 Transformer 模型则可以克服 RNN 模型的这一局限. 此外, Transformer-XL 模型允许输入更大量的信息, 具有更强的表征学习能力和准确性, 在捕获代码自然性方面展现出了比基于 RNN 的语言模型更优越的性能, 并且计算成本更低. 因此, Dowdell 等人提出, 该模型可应用于与源代码相关的语言建模任务. 例如: Kim 等人[75]已将 Transformer 模型应用于代码补全, Hammad 等人[46]将其应用于代码克隆, Buratti 等人[72]将其应用于抽象语法树(AST)特征提取.

Buratti 等人[72]发现, 目前的代码分析方法很大程度上依赖于从 AST 中派生出来的特性. 因此, Buratti 等人将基于 Transformer 的 Bert 语言模型(C-BERT)应用于原始源代码上, 首次研究了该语言模型是否能够自动发现 AST 特征. 此外, Buratti 等人为了缓解 BERT 应用于源代码上引起的 OOV 问题, 探索了 3 种分词器, 并

取得了良好的应用效果.

Mou 等人[40,73]指出, 源代码与自然语言[76]的不同之处在于: 程序包含丰富、明确且复杂的结构信息. 因此, 对软件工程任务而言, 获取代码的结构信息显得尤为重要. 虽然前人已经尝试过使用 DNN, RNN, CNN 等神经网络捕获程序的树形结构信息, 但它们的效果都不尽人意. 于是, Mou 等人提出一种基于抽象语法树的新型树型卷积神经网络(TBCNN), 结合连续二叉树和动态池化方法处理不同大小和形状的 AST, 并在检测代码片段和程序分类任务中, 从效果上超越了其他对比模型.

Zhang 等人[74]指出: 传统的基于神经语言模型的方法往往将程序视为自然语言文本, 从而忽略源代码的重要语义信息. 虽然基于 AST 的神经模型可以更好地表示源代码, 但 AST 的大小过大又容易出现长期依赖问题. 因此, Zhang 等人提出一种新的基于 AST 的神经语言模型(ASTNN)表示源代码, 该模型将大型 AST 拆分为一系列小语句树, 结合双向 RNN 模型, 利用语句的自然性, 生成代码片段的向量表示. ASTNN 有效缓解了基于树的神经模型出现的梯度消失问题. 此外, Chakraborty 等人[77]同样使用了 AST+RNN(LSTM)的组合模型处理代码更改问题.

基于神经语言模型的代码自然性建模方法见表 3

表 3 基于神经语言模型的代码自然性建模方法

模型名称	神经网络	是否解决 OOV 问题	描述	代表文献
Dnn4C	DNN	未提及	该模型通过句法和类型上下文来补充词汇代码元素的局部上下文	[30]
LSTM Language Model	RNN+ LSTM	未提及	基于长短期记忆神经网络的深度学习模型以捕获相关代码分散到较远地方的上下文	[35]
Open-vocabulary Neural Language Model	RNN	是, 该文献对词汇表设计选择进行了彻底的研究, 研究了对语料库进行的种种处理会带来的影响, 从而更透彻深入地克服 OOV 问题	该 NLM 模型利用 BPE 降低词汇表的大小并成功地预测 OOV 词汇	[38]
Neural Language Model	NN+ Attention	未提及	带有注意机制的神经语言模型能够捕获更长的上下文依赖关系, 为动态编程语言提供有效的代码建议	[63]
Neural Language Model	NN+ LSTM	未提及	利用抽象语法网络,提供与输出的内在结构极其相似的简单解码过程, 以更好地解决语义解析和代码生成等任务	[70]
Transformer-XLModel	Transformer-XL	未提及	该模型的 self-attention 机制, 进一步提升了捕获长期依赖和解决长序列问题的能力, 以更好的捕获代码自然性	[71]
C-BERT	Bert+ Transformer	是, 该文献对源代码的子词进行分词, 并探索了 3 种分词器, 有效降低了 OOV 问题和稀有词对研究的影响	基于 Transformer 的 C-BERT 模型在源代码的角度提取结构化信息, 以更好地完成 AST 标注任务	[72]
TBCNN	CNN+ AST	未提及	基于抽象语法树的新型 TBCNN 能更好地捕获程序的结构信息	[40,73]
ASTNN	RNN+ AST	未提及	该模型结合 AST 和 RNN, 有效缓解了长期依赖问题, 以更好地获取源代码的结构信息	[74]

2.3 实验验证与评价指标

实验验证方法是指在模型评估中, 如何将数据集划分为训练集和测试集. 本节详细介绍当前代码自然性研究中常用的实验验证方法:交叉验证法.

- 交叉验证(cross validation)[1].

交叉验证是机器学习领域常用的模型验证方法, 目前, 大部分关于代码自然性研究的工作也采用了交叉验证的方法来验证代码自然性建模的效果. 代码自然性模型的交叉验证包括跨项目和同项目两种方式: 跨项目验证对训练集与测试集的划分以项目为单位, 同项目验证对训练集与测试集的划分以项目内的代码实体为单位.

评价指标是指针对代码自然性建模结果, 研究者提出不同的指标来量化模型预测效果. 本节详细阐述在代码自然性研究中常用的性能指标, 包括交叉熵、困惑度.

- 交叉熵(cross entropy).

交叉熵是评估语言模型的常用指标, 用于衡量模型的估计结果与实际情况的差异. 其在自然语言中的意义是模型对文本识别的难度, 其主要刻画的是实际输出(概率)与期望输出(概率)的距离. 交叉熵的值越小, 两个概率分布越接近. 在代码自然性研究中, 交叉熵也被用于衡量测试代码对于从语料库中估计的分布模型来说是否合理. Hindle 等人[1]通过交叉熵这一评估指标对 Java 语言和自然语言(English)进行比较, 最终得出包括 Java 语言在内的编程语言和自然语言一样, 都是非常重复且具有高度规律性的; Ray 等人[19]和 Lanchantin 等人[47]使用交叉熵体现代码语句的自然程度, 他们认为, 没有错误的(即更加自然的)代码语言比有缺陷的(即不自然的)代码语句的熵更低.

对于一个语言模型 M 来说, 其对句子 $s=\omega_1\omega_2\ldots\omega_n$ 出现的概率估计为 $P_M(S)$, 交叉熵的计算公式如下所示:

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

- 困惑度(perplexity).

困惑度用于度量一个概率分布或概率模型预测样本的好坏程度, 也可以用于比较两个概率模型在预测样本的能力上的优劣, 低困惑度的概率模型往往能更好地预测样本. 困惑度与语句序列的概率相关, 其基本思想是: 语句序列越好(概率越大), 语言模型越好, 困惑度越小. Luong 等人[78]发现, 困惑度与翻译质量之间存在很强的关联性; Chen 等人[79]使用困惑度验证所训练模型 SEQUENCER 的性能; Hindle 等人[1]通过困惑度这一评价指标度量代码语言的自然性.

困惑度是交叉熵的指数变换, 其公式如下所示:

$$H_M(s)=-\frac{1}{n}\sum_{1}^{n}\log P_M(\omega_i\mid\omega_1...\omega_{i-1}).$$

3 代码自然性应用

2012 年, Hindle 等人[1]假设编程语言是自然语言的一种, 它具有与自然语言类似的统计特性, 介绍了将代码自然性应用于软件工程任务的开创性工作. 如今, 统计语言建模技术已经成功地应用到大型源代码语料库中, 产生了各种新的软件开发工具, 例如用于代码推荐、缺陷预测、程序修复、API 代码推荐、语言迁移、代码摘要、程序特性分析等软件工程任务的工具. 许多工作都利用了软件的“自然性”来辅助软件工程任务.

3.1 代码补全(code completion)

基于代码自然性的代码补全通过语言模型捕获代码中可预测的统计特征[80], 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, 为开发人员提供建议列表.

Hindle 等人[1]首次提出编程语言是语言的一种, 并且不再从源代码方面进一步挖掘额外信息, 而是着手从 NLP 方向发掘代码补全的可能性. Hindle 等人发现, 建立在编程语言语料库上的经过平滑处理的 n-gram 语言模型的熵远远低于建立在自然语言上的, 因此只需要经过几次尝试该模型便可以猜测出下一个标识符. 鉴于此, Hindle 等人通过自然语言最常用的 n-gram 语言模型对 Java 语言进行建模, 得到了比 Eclipse 自带插件更好的代码补全效果.

Tu 等人[8]发现: 编程语言有可用的局部规律, 但是 n-gram 语言模型却未能利用这种属性. 因此, 他们在

n-garm 语言模型的基础上增添了一个用于捕获局部规律的缓存组件, 从而提出了能够动静态结合的 cache *n*-gram 语言模型, 并用实验证明: 在代码补全方面, cache *n*-garm 语言模型比 *n*-gram 模型表现得更好, 基于 cache *n*-gram 语言模型的建议引擎甚至可以实现跨项目代码补全. Franks 等人[12]在此基础上提出了 CACHECA 插件, 该插件由 Eclipse 内置的建议引擎和 cache *n*-gram 模型建议引擎结合而成, 使 Eclipse 的建议准确度提高了超过一倍.

Yang 等人[9]发现: 目前最先进的技术是利用词法信息捕获标识符级别的代码规律, 这样的语言模型更适合预测简短的标识符序列, 而不利于较长语句的预测. 因此, Yang 等人提出了 PCC(即 IR *n*-gram)来优化基于标识符级别的语言建模, 此模型在 *n*-gram 语言模型的基础上增加了源代码的中间表示(IR), 它利用语义和变量的相对顺序将标识符分组. 通过这种方式, PCC 能够处理长标识符序列; 此外, PCC 采用遗传算法和最长公共子序列算法相结合的模糊匹配技术, 使预测更加准确, 进而提升了代码补全的效果.

Li 等人[39]发现: 标准的神经语言模型, 即使有注意机制, 也不能正确地预测限制代码补全性能的 OOV 词汇. 因此, Li 等人提出一种混合指针网络以降低 OOV 词汇对代码补全的影响, 它可以通过全局词汇表生成一个单词或从局部上下文复制一个单词来预测下一个单词. 该网络由两个组件组成: 一个带 attention 的神经网络和一个指针网络. 混合指针网络是两个组件的加权组合, 进行预测时, 每次选择其中一个组件用于生成一个单词. 通过这种全局结合局部的方式, 可以有效地降低 OOV 问题带来的影响, 并在很大程度上提升代码补全的效果.

Raychev 等人[13]提出了一种新的代码补全方法, 该方法将程序分析与统计语言模型相结合, 并在名为 SLANG 的工具中实现. 其主要思想是, 将代码补全问题归约为预测句子概率的自然语言处理问题. Raychev 等人设计了一个可以伸缩的静态分析方法, 其主要操作流程是: 先从一个大型代码库中提取一系列方法调用序列, 再将上述调用序列索引到统计语言模型中, 最后用语言模型找出排名最高的句子.

代码补全常用的评估指标有: 交叉熵、Top-*K* 准确率、平均倒数排名 MRR (mean reciprocal rank)等.

- Top-*K* 准确率: 用于计算预测结果中概率最大的前 *K* 个结果包含正确答案的占比;
- MRR: 以第 1 个正确答案的位置作为衡量标准, 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, 则表示正确答案越靠近推荐列表的顶端. 在代码补全中, 对于数据中的所有待补全的标识符 *T*, MRR 取所有补全 MRR 得分的平均数:

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

3.2 缺陷预测/检测(defect prediction/identification)

基于代码自然性的缺陷预测/检测旨在根据软件历史开发数据以及已发现的缺陷, 借助机器学习等方法预测软件项目中的缺陷数目和类型等. 一方面, 可以帮助研究人员通过交叉熵验证代码的自然程度(即代码语句是否存在 bug); 另一方面, 还可以帮助研究人员通过序列概率的高低来进行缺陷预测/检测.

Ray 等人[19]提出“不自然”的代码更有可能是错误的, 并验证了此观点. Ray 等人利用 *n*-gram 语言模型在 10 个不同的 Java 项目上进行实验, 发现“非自然”的代码即熵更高的代码更有可能出现在错误修复提交中; 相应的, 有缺陷的代码在修复后会变得“自然”. 因此, Ray 等人论证了熵值是缺陷预测的有效辅助手段, 并且表明熵值可以改进缺陷预测, 为静态缺陷查找器提供改进的优先级排序, 提高故障定位算法的性能, 甚至建议使用“更自然”的代码来替换有缺陷的代码. Ray 等人提出将熵值的排序应用到静态缺陷查找器中, 将产生更高的效益.

Wang 等人[11]提出: 一个规则即使存在, 如果它没有频繁出现, 模型也无法学到该规则, 所以会错失许多 bug. 鉴于此, Wang 等人提出一种利用 *n*-gram 语言模型而不是规则来检测缺陷的新方法: Bugram. Bugarm 利用 *n*-gram 语言模型对程序标识符进行顺序建模, 根据它们在学习模型中的概率评估来自程序的标识符序列, 低概率序列将被标记为潜在错误. Wang 等人用实验证明, Bugram 是对现有的基于规则的错误检测方法的补充.

Lanchantin 等人[47]在 Ray 等人的贡献之上提出了一种更先进的语言建模技术——LSTM, 基于熵对源代

码建模并对错误行进行分类. 此模型能更好地建模具有长期依赖关系的语言, 并且能比以前的技术(如 *n*-gram 语言模型)捕获更优质的源代码规律. 此外, Lanchantin 等人将此模型的实验结果与 Ray 等人提出的基于熵对缺陷行进行分类的结果进行比较, 结果表明, 长短期记忆神经网络优于 *n*-gram 语言模型.

Campbell 等人[21]发现, *n*-gram 语言模型能够增强编译器定位丢失的标记、额外的标记和被替换的标记的能力, 由此提供了一种利用软件源代码自然性检测语法错误的方法和工具, 通过构造一个简单的经过源代码标记的 *n*-gram 语言模型来定位语法错误, 并且提出了由一个编译包装器、一个词法分析器和一个语言模型组成的工具: UnnaturalCode. 该模型不仅会随着错误的修复不断更新语料库, 还可以随着项目上下文的变化而变化. 研究表明: 当 UnnaturalCode 的结果与编译器自己的结果结合在一起时, 正确的位置, 即语法错误的位置, 会比单独编译器的结果更早被报告出来.

Yan 等人[81,82]受到 Ray[19], Campbell[21], Santos[20]等人研究的启发, 基于代码自然性提出了即时缺陷检测与定位的两阶段框架, 并开发了一个名为 JITO 的 IDE 插件. 在即时缺陷定位阶段, Yan 等人提出了一种基于软件自然性的即时缺陷定位方法, 该方法利用 *n*-gram 语言模型在历史无缺陷代码的基础上构建一个代码语言模型, 该模型用于对有问题的变更代码行计算熵值, 随后对每一行的熵值进行排序, 以确定最有可能的缺陷位置.

缺陷预测/检测常用的评估指标有: 交叉熵、MRR、Top-*K* 准确率、召回率、精确率、*F*1-*score* 等.

代码自然性应用的代表文献总结见表 4.

表 4 代码自然性应用的代表文献总结

应用	代码自然性建模方法	评估指标					代表文献	应用效果概述
		交叉熵	MRR	Top-*K* 准确率	召回率	其他		
代码补全	*n*-gram	√					[1]	基于代码自然性的代码补全在 Java, C, Python, Javascript 等编程语言的语料库上取得了良好的应用效果, 特别是为动态类型语言提供了有效的代码补全方法, 开发了一系列代码建议插件
	Cache *n*-gram		√				[8]	
	Cache *n*-gram		√	√			[12]	
	n-gram			√			[9]	
	RNN+ LSTM					准确度*	[39]	
	n-gram/ RNN					手动检查	[13]	
缺陷预测/检测	*n*-gram	√					[19]	基于代码自然性的缺陷预测/检测主要应用于 Java 语料库, 其检测效果优于传统静态缺陷查找器, 发现了熵值可以用于量化代码自然程度, 开发了一系列缺陷预测/检测的框架
	n-gram				√	手动检查精确率 *F*1-score	[11]	
	RNN+ LSTM	√				AUC	[47]	
	n-gram		√				[21]	
	n-gram		√	√		MAP	[81,82]	
代码摘要	*n*-gram					R-precision	[34]	基于代码自然性的代码摘要主要应用于 Java 语料库, 形成了深度学习方法和代码自然性相结合的机制, 开发了一系列用于生成代码摘要的模型, 取得了优于当前最先进方法的效果
	RNN/ *n*-gram				√	精确率 *F*1-score BLEU-4 METEOR	[83]	
程序修复	LSTM/ *n*-gram		√				[20]	基于代码自然性的程序修复主要应用于 Java 语料库, 发现了缓解 OOV 问题的复制机制, 开辟了很有前途的研究方向
	RNN+ LSTM					困惑度 准确度*	[79]	

表 4 代码自然性应用的代表文献总结(续)

应用	代码自然性建模方法	评估指标					代表文献	应用效果概述
		交叉熵	MRR	Top-*K* 准确率	召回率	其他		
API 推荐	GraLan/ ASTLan			√			[33]	基于代码自然性的 API 推荐, 发现了基于图的语言模型可以挖掘正确的语法模板, 开发了一系列 API 建议引擎, 得到了比最先进方法更准确的建议
语言迁移	RNN					BLEU-4 SCR* SeCR*	[30]	基于代码自然性的语言迁移在 Java 源代码迁移到 C#时, 提高了语义和语法的准确性
属性预测	Probabilistic Graphical Models				√	准确率 精确率	[84]	基于代码自然性的属性预测使用概率图形模型, 其效果优于传统推理算法, 形成了结构化预测方法, 开发了一系列预测引擎
程序理解	*n*-gram				√	准确率 精确率	[85]	基于代码自然性的程序理解在 Java 语料库上取得了良好的应用效果, 形成了统一的代码风格, 开发了一系列学习代码风格的框架

注: *表示该度量指标为作者自定义指标

3.3 代码摘要(code summarization)

基于代码自然性的代码摘要旨在更便捷地理解代码的目的以及提取重要的信息, 利用机器学习等方法, 根据上下文自动生成代码摘要或进行代码总结.

Guerrouj 等人[34]提出通过访问文档和在线文章以理解源代码, 或者阅读大量的文本才能理解某些重要代码的目的, 而这些代码恰恰与重要的问题领域知识有关. 为了更便捷地总结代码元素的用途和目的, Guerrouj 等人提出一种根据上下文自动生成标识符摘要的方法, 该方法利用非正式文档(包括嵌入其中的源代码文本信息)提供准确的摘要, 通过 *n*-gram 语言模型进行检验. 此外, 与其他仅在有限上下文讨论代码元素的文档不同, Guerrouj 等人总结的摘要包含了广泛的上下文, 可以帮助研究者们快速理解与任务相关的代码元素.

Hu 等人[83]提出: 以前的摘要生成方法是从相似的代码片段中检索以生成摘要, 然而这些方法严重依赖于是否可以检索到相似的代码片段, 并且不能捕获源代码中的 API 知识, 而这些知识带有关于源代码功能的重要信息. 因此, Hu 等人提出一种生成代码摘要的新方法 TL-CodeSum, 它可以将在不同但相关的任务中学习到的 API 知识用于代码总结. Hu 等人提出: 代码段的功能与其 API 序列相关, 特定的 API 序列可以实现新功能, API 知识可以辅助代码总结. 此外, Hu 等人用实验证明, TL-CodeSum 模型在生成摘要方面明显优于当时最先进的方法.

代码摘要常用的评估指标有 BLEU-4, METEOR 等.

- BLEU-4[86].

用于评估代码自然性模型生成的序列和实际序列的匹配程度, 它的取值范围在 0 到 1 之间. BLEU-4 分数越接近 1, 则代表两个序列越匹配. BLEU-4 公式如下:

$$BLEU-4 = BP \times \exp\left(\sum_{n=1}^{4} W_n \log P_n\right),$$

其中, W_n 是 *n*-gram 的权重, P_n 是 *n*-gram 的精确度, BP 是惩罚过短生成序列的惩罚因子.

- METEOR.

通过将生成结果与参考结果对齐并计算句子级相似度分数来评估生成结果. METEOR 的计算公式如下:

$$Score=(1-Pen)F_{mean},$$

其中, *Pen* 是惩罚生成结果中的词序与参考结果中的词序不同的惩罚因子, F_{mean} 是 *Precision* 和 *Recall* 加权调和平均.

3.4 程序修复(program repair)

基于代码自然性的程序修复使用在正确源代码上训练的语言模型查找不正确的标识符以发现语法错误, 通过语言模型合成修复程序, 确定在估计的错误位置上哪些标识符更有可能出现, 以建议对标识符进行可能的更改, 修复已识别的错误.

Santos 等人[20]提出了一种方法定位语法错误发生的位置, 并建议对标识符进行更改, 以修复所识别的错误. 该方法通过使用在正确源代码上训练的语言模型(*n*-gram 和 LSTM 语言模型)寻找不正常的标记, 从而发现语法错误; 通过语言模型进行综合考虑, 确定在估计的错误位置更有可能出现什么标记, 以此进行修复. 此外, Santos 等人设计的方法在从 GitHub 收集的大量手写 Java 源代码的语料库上进行训练, 但根据大量真实的学生错误进行评估, 因此更具有实用性. 实验表明, Santos 等人设计的 LSTM 模型能够修复语料库中几乎一半的单标记语法错误.

Chen 等人[79]首次提出用语言无关的通用方式进行程序修复, 完全依赖机器学习来捕获语法和语法规则, 生成格式良好的可编译程序. 鉴于此, Chen 等人提出了一种基于 seq2seq 学习的端到端(end-to-end)程序修复方法: SEQUENCER. 该方法的基础模型是一个类似于自然语言处理架构的循环神经网络, 该方法的独特之处在于将编码器-解码器架构与复制机制相结合, 编码器-解码器架构能够捕获修复所需的长期依赖关系, 复制机制可以克服源代码中的 OOV 问题. 经过验证, 将两者结合后的 SEQUENCER 可以修复训练集外的任务.

程序修复常用的评估指标有 MRR、困惑度等.

3.5 API推荐(API recommended)

基于代码自然性的 API 推荐旨在根据研究人员开发时的需要, 预测 API 推荐序列, 为开发人员提供建议列表.

Nguyen 等人[33]提出: 由于 *n*-gram 语言模型中的序列性质和源代码中的语法语义的结构性质之间的不匹配, *n*-gram 语言模型在捕获 API 使用模式时面临挑战. 因此, Nguyen 等人提出了一种基于图的统计语言模型 GraLan, 它可以从一个图的语料库(从源代码的训练数据产生)中学习, 并计算给定观察到的(子)图的出现概率. 基于 GraLan 语言模型, Nguyen 等人开发了一个 API 建议引擎, 实证表明, 该引擎在 API 建议方面比现有的方法更准确. API 推荐常用的评估指标有 Top-*K* 准确率、交叉熵等.

3.6 语言迁移(language migration)

基于代码自然性的语言迁移旨在提高将一种编程语言的源代码翻译成另一种编程语言的代码时的准确率, 因为经过训练和调参的语言模型可以捕获特征的区别性信息.

Nguyen 等人[30]提出: 虽然统计语言模型(LM)已经被广泛应用于一些软件工程任务中, 但它们在语言迁移方面的准确性还是有待提高. 因此, Nguyen 等人提出了一个基于 DNN 的语言模型: Dnn4C. 该模型可以获取有关特征的区别信息, 从而在更高的抽象级别上学习区分不同语法和语义上下文中的词汇标记. 该模型可以应用于代码迁移, 当使用机器翻译模型将源代码从 Java 迁移到 c#时, Dnn4C 有助于提高 *n*-gram 语言模型的准确性. 虽然该模型在语言迁移准确率方面有较大的提高, 但仍局限于具有非常相似特征的语言之间的转换(从 Java 到 C#), 对于其他语言的迁移还需不停地探索.

语言迁移常用的评估指标有 BLEU-4、研究者自定义的指标等. Nguyen 等人[30]使用自定义的指标 SCR (syntactic correctness)即通过编译的迁移方法数量与迁移方法总数的比率和 SeCR (semantic correctness)即语义正确的迁移方法数量与迁移方法总数的比率衡量迁移过后的代码语言的语法正确性和语义正确性.

3.7 属性预测(properties prediction)

基于代码自然性的属性预测旨在根据代码片段预测程序标识符的名称和变量的类型注释, 帮助程序员理解代码的用途.

Raychev 等人[84]提出一种从大量代码库(big code)中预测程序属性的新方法, 并引入一种新的统计方法, 其原理是, 通过从已有的代码库中学习一个概率模型来预测给定程序的属性. 这个方法的核心思想是: 将属性推理问题表述为机器学习中的结构化预测问题, 从而利用概率图形模型以实现程序属性的联合预测. Raychev 等人的方法能够帮助程序员更好的理解代码的用途, 以及恢复可能的标识符名称从而使代码再次可读. 在该方法的基础上, Raychev 等人构建了一个预测引擎 JSNICE, 用于预测 JavaScript 的名称和类型注释, 取得了良好的应用效果. 属性预测常用的评估指标有召回率、精确率、准确率等.

3.8 程序理解(program comprehension)

基于代码自然性的程序理解旨在利用代码语言模型提供标识符以及代码风格建议, 帮助不同的程序员统一编程风格.

Allamanis 等人[85]提出: 每个程序员都有其各自的风格, 当共同协作一个项目时, 程序员们需要遵守编码约定, 然而并不是每个程序员都能保证遵守. 鉴于此, Allamanis 等人提出了一个学习代码风格的框架: NATURALIZE, 该框架能够建议自然标识符名称和格式约定以提高编程风格的一致性. NATURALIZE 建立的工作基础是统计自然语言处理成功应用于源代码的论证, 在此基础上, Allamanis 等人还提出编码约定也是自然的, 并证明 NATURALIZE 在提出自然的建议方面是有效的. 程序理解常用的评估指标有召回率、精确率、准确率等.

4 代码自然性关键问题和未来展望

4.1 科学问题

4.1.1 代码自然性形成机理及其与自然语言的异同之处

代码自然性的形成基于一个基本事实: 代码与自然语言类似, 具有高度重复性和可预测性. Gabel 和 Su[6]首次发现并报道了这一事实, 他们发现, 大量的代码片段往往会重复出现. 这就为 Hindle 等人[1]提出代码和自然语言一样都是非常重复并且可以用统计语言模型捕获奠定了理论基础. 然而, 近期的研究表明[38], 代码和自然语言之间还存在着不小的差异: 一是代码语料库会随着软件开发人员的标识符定义而不断更新变得更加复杂, 二是代码语料库还包括分隔符、编程语言关键字、API、运算符等特有词. 因此, 需要针对上述相似和差异性进行深入的研究, 探索代码词汇表的规律和特点, 以构建更符合代码特性的语言模型.

4.1.2 代码自然性与不同软件工程任务的关联关系

如今, 在代码大数据[62]的环境下, 软件工程任务日益复杂, 这也给代码自然性应用带来了新的挑战. 随着代码自然性研究的深入, 越来越多的软件工程任务可以基于代码自然性这一特性提出解决方案. 然而, 不同软件工程任务具有不同的特点和应用场景, 例如针对代码补全和缺陷预测, 代码自然性构建的语料库选择、建模方法有所不同, 而针对何种特点的软件工程任务适合用怎样的代码自然性建模理论和方法尚没有文献明确提出. 其原因在于: 现有的代码自然性建模理论和方法多是根据代码的重复性和可预测性提出的, 针对具体软件工程任务设计的代码自然性建模理论和方法研究比较匮乏. 因此, 在未来的研究中, 需要进一步研究代码自然性与不同软件工程任务的关联关系, 提出针对具体任务的代码自然性建模理论和方法.

4.1.3 代码自然性的挑战和局限性

Hindle 等人[1]表明: 源代码和自然语言一样都具有高度重复性, 并且可以被为自然语言处理领域开发的语言模型有效地建模以应用于各种软件工程任务. 但是代码自然性在发展过程中仍然存在着与代码特性相关的挑战和局限性. Tu 等人[8]率先发现: 相比自然语言, 源代码有一个特殊的属性——局部性, 并因此提出了能够捕获源代码局部自然性的 cache *n*-gram 语言模型. Mou 等人[40]提出: 程序包含比自然语言更为丰富、明确且

复杂的结构信息, 代码自然性模型不能仅考虑代码文本信息, 还应该利用结构信息对代码进行建模. 同时, 考虑到程序的正确性对语法和语义的要求非常严格, 也有研究者指出, 代码自然性存在一定的局限性. Ray 等人[19]将代码自然性应用于缺陷预测任务时发现: 当代码片段中引入新的标识符或新的语句, 尽管这部分代码是正确的, 也依然会被赋予更低的自然性. 相比自然语言, 源代码不允许近义词的替换, Allamanis 等人[62]提出: 某些编程语言允许的未定义行为可能会导致语义歧义, 并且由于非标准编译器的存在, 语法问题也可能出现. 因此, 在未来的研究中, 需要更全面的考虑代码特性对代码自然性建模的影响, 以更好地应用于各种软件工程任务.

4.2 技术难点

关于代码自然性的研究已经取得了一定成果, 并且在不同的软件工程任务中得到了应用, 但是仍存在还未完全解决的问题.

4.2.1 OOV 问题处理

如何降低 OOV 问题对代码语言模型的影响, 是代码自然性建模问题中的主要挑战之一. 与自然语言不同的是, 代码可以随意增加标识符, 并且标识符名称特别具有个人风格, 所以 OOV 问题不可能被完全克服. 而词汇量大和词汇量不足问题都将严重影响源代码的语言模型, 降低模型的性能, 使模型无法伸缩. Karampatsis 等人[38]提出一个用于源代码的大规模开放词汇表 NLM 来降低 OOV 问题带来的影响, 并且得到了验证. 虽然改进后的语言模型在很大程度上能使新的工具应用于基于代码自然性的任务, 但此研究仍有一定局限性. 因此, 研究者们在处理 OOV 问题上面临着实际应用的挑战.

4.2.2 语料库选择

在目前的代码自然性研究中, Java 项目集合是大多数研究者们的首选, 其次是 C/C++语言项目和 Python 项目, 因为他们认为 Java、 Python、 C/C++语言是目前使用较为频繁的编程语言. 尽管大多数研究者都会从不同的领域选择项目组成语料库, 但不管是从编程语言方面考虑还是从项目领域方面考虑, 都不能认为这是具有代表性的. 所以往往会出现这样一种现象: 以某一语料库为数据集训练并表现良好的语言模型无法在其他数据集中展现原有的水准[30,38]. 此外, 根据软件工程任务的不同, 需要选择的语料库也不同. 例如, 语言迁移任务则至少需要两种不同语言的语料库. 因此, 如何构建合理的语料库、如何选择更具代表性的编程语言, 于研究者们而言, 仍是该研究中的一大主要挑战.

4.2.3 模型构建

现有代码自然性相关的技术在模型构建上存在着不同的策略, 有研究者使用复杂的神经语言模型, 也有研究者主张使用传统的统计语言模型. 在神经语言模型中, 又有 RNN 语言模型、LSTM 语言模型等. 不同的建模技术在模型应用上存在差异, 并且效果也不同. 虽然神经语言模型在捕获远距离标识符序列等方面比统计语言模型表现的更好, 但仍有研究者提出, 经过调参和改进的统计语言模型的效果不一定比神经语言模型差. 此外, 不同的软件工程任务适用的语言模型也不尽相同, 统计语言模型更适合要求训练时间少、精度较低的任务; 而神经语言模型相对更适合处理精度高、要求捕获更长上下文的任务. 因此, 如何根据具体应用选择合适的建模技术、如何设计任务相关的先进建模技术, 仍然是代码自然性相关研究中的一大主要挑战.

4.3 未来展望

代码自然性相关的研究工作存在上一节所提到的主要挑战, 为了应对这些挑战, 本节将从 OOV 问题处理、数据集构建、模型构建和工程实践这 4 个方面展望代码自然性研究的未来趋势.

4.3.1 研究词汇表设计选择的影响, 减少 OOV 问题

研究表明: 代码自然性研究中, 任何在大规模软件语料库上训练的模型都必须处理极其庞大和稀疏的代码词汇. 此外, 如果在训练集中没有观察到某一个标识符, 语言模型将无法预测它们, 因此会导致 OOV 问题. 数据稀疏问题可以通过平滑处理进行缓解. 对于 OOV 问题来说, 改进分词算法[87]、进行符号归一化[88]、改进 RNN 模型(例如采用 Copy 机制[78])、扩大语料库[89]等都是可行的方法. 但最根本的还是要对词汇表设计选择

进行分析, 研究这些选择对词汇表的影响, 以及它们如何影响 OOV 比率. 因此, 深入研究词汇设计选择对语料库的影响[38], 减少 OOV 问题带来的影响, 将是代码自然性研究的一大趋势.

4.3.2 研究跨语言代码自然性建模方法, 构建大规模自适应的语料库

现有代码自然性研究中往往会使用多个项目[1]或者多领域[19]的项目作为语料库, 在编程语言上也尽量选择更多的类型[38]. 但是大部分文献在有效性威胁中还是会表明选取的编程语言或者语料库可能不具有代表性. 此外, 不同语言间的语法也不甚相同, 需要解析的模型就不能同时适用于多种编程语言. 鉴于此, 跨语言代码自然性的研究应该提上日程, 针对待分析的项目类型、项目特点、编程语言, 应该自适应地选择语料库, 例如: 选择相同及相似编程语言的、相似项目类型的语料库; 针对不同的软件工程任务, 应该综合考虑编程语言和数据集, 尽可能多地、全面地选取各领域的项目. 构建跨语言代码自然性模型和大规模自适应的语料库, 将是代码自然性研究的一个发展方向.

4.3.3 研究代码自然性与具体任务的关联关系, 提出针对特定任务的代码自然性建模新技术

复杂的模型不能代表优越的性能, 模型构建需要针对自然性应用的特殊性进一步研究. 随着神经方法在计算机视觉、语音识别等方面展现出卓越的性能, 研究者们将机器学习的其他先进技术, 如神经网络, 应用到代码自然性的相关研究中, 已经成为一个重要方向. 但 Liu 等人[90]的研究表明, 神经方法并不能在所有任务中都能取得比传统方法更好的效果. Hellendoorn 等人[31]的研究表明, 精心设计的 *N*-gram 模型并不逊色于当时最先进的神经网络模型. 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, 未来的代码自然性研究需要对软件工程任务的特殊性进行大量创新. 因此, 如何将代码自然性模型与特定的任务相关联从而进行创新, 将是代码自然性研究的一大技术难点.

4.3.4 解决工程实践中的问题, 推动代码自然性的广泛应用

代码自然性技术虽然能够在代码补全、缺陷预测/检测、程序修复、API 推荐、语言迁移、代码摘要、程序特性分析等软件工程任务中发挥不错的效果, 但在工程实践中, 在工业项目语料库构建、代码自然性模型应用效率、准确率等方面仍然面临许多挑战. 克服现有工程实践应用挑战思路主要有: (1) 研究代码语料库的特点和规律, 分析词汇表的设计选择, 构建面向工程实践应用的开源项目与闭源项目相结合的代码语料库; (2) 明确在工程实践中代码自然性与软件工程任务的关联关系, 设计针对软件工程任务的代码自然性技术, 以提高代码自然性模型的应用效率; (3) 针对不同的任务需求, 选择合适的自然性模型构建方法和语料库尺寸, 以提高代码自然性模型的准确率.

5 总 结

近年来, 代码自然性相关技术成为软件工程领域和 NLP 领域的研究热点, 对软件开发有着极为重要的影响, 代码自然性相关理论对软件工程任务的解决起到了积极作用. 学术界对代码自然性的研究有将近 10 年的时间, 但尚有许多问题需要解决, 也少有综述类文章总结过代码自然性及其应用研究的进展. 鉴于此, 本文围绕代码自然性的数据集构建、模型构建、实验验证、评价指标等方面, 梳理并总结当前研究进展. 基于当前进展分析, 本文总结了当前代码自然性研究面临的关键技术和未来发展趋势, 主要工作总结如下: (1) 介绍了代码自然性研究的数据集构建选择和两大类模型构建方式, 总结了常用的实验验证方法以及评估指标; (2) 对将代码自然性研究应用于代码建议、缺陷预测、程序修复、API 代码建议、语言迁移、代码摘要等任务的代表性文献进行了总结; (3) 从科学问题、技术难点的角度总结了当前代码自然性面临的关键问题, 并从 OOV 问题处理、数据集构建、模型构建和工程实践这 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] Hindle A, Barr ET, Su Z, *et al*. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 837–847.

[2] Hirschberg J, Manning CD. Advances in natural language processing. Science, 2015, 349(6245): 261–266.

[3] Cambria E, White B. Jumping NLP curves: A review of natural language processing research. IEEE Computational Intelligence Magazine, 2014, 9(2): 48−57.

[4] Khurana D, Koli A, Khatter K, *et al*. Natural language processing: State of the art, current trends and challenges. arXiv: 1708.05148, 2017.

[5] Sharma A, Tian Y, Lo D. NIRMAL: Automatic identification of software relevant tweets leveraging language model. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2015. 449−458.

[6] 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 (FSE 2010). ACM, 2010. 147−156.

[7] Casalnuovo C, Lee K, Wang H, *et al*. Do people prefer "natural" code? 2019.

[8] 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.

[9] Yang Y, Jiang Y, Gu M, *et al*. A language model for statements of software code. In: Proc. of the 32nd IEEE/ ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 682−687.

[10] Allamanis M, Tarlow D, Gordon AD, *et al*. Bimodal modelling of source code and natural language. In: Proc. of the 32nd Int'l Conf. on Machine Learning, Vol.37. 2015. 2123−2132.

[11] Wang S, Chollak D, Movshovitz-Attias D, *et al*. Bugram: Bug detection with *n*-gram language models. In: Proc. of the 31st IEEE/ ACM Int'l Conf. on Automated Software Engineering (ASE 2016). ACM, 2016. 708−719.

[12] Franks C, Tu Z, Devanbu P, *et al*. CACHECA: A cache language model based code suggestion tool. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 705−708.

[13] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). ACM, 2013. 419−428.

[14] Tonella P, Tiella R, Nguyen CD. Interpolated *n*-grams for model based testing. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE 2014). Hyderabad: ACM, 2014. 562−572.

[15] 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 (MSR). IEEE, 2013. 207−216.

[16] Movshovitz-Attias D, Cohen WW. Natural language models for predicting programming comments. In: Proc. of the 51st Annual Meeting of the Association for Computational Linguistics (Vol.2: Short Papers). 2013. 35−40.

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

[18] Devanbu P. New initiative: The naturalness of software. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Software Engineering. IEEE, 2015. 543−546.

[19] Ray B, Hellendoorn V, Godhane S, *et al*. On the "naturalness" of buggy code. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE 2016). ACM, 2016. 428−439.

[20] Santos EA, Campbell JC, Patel D, *et al*. Syntax and sensibility: Using language models to detect and correct syntax errors. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. 311−322.

[21] Campbell JC, Hindle A, Amaral JN. Syntax errors just aren't natural: improving error reporting with language models. In: Proc. of the 11th Working Conf. on Mining Software Repositories (MSR 2014). ACM, 2014. 252−261.

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

[23] Yin P, Neubig G. A syntactic neural model for general-purpose code generation. 2017.

[24] Oda Y, Fudaba H, Neubig G, *et al*. Learning to generate pseudo-code from source code using statistical machine translation (t). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2015. 574−584.

[25] Nguyen AT, Nguyen TT, Nguyen TN. Lexical statistical machine translation for language migration. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, 2013. 651.

[26] Lin B, Nagy C, Bavota G, *et al*. On the impact of refactoring operations on code naturalness. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 594−598.

[27] Jimenez M, Maxime C, Le Traon Y, *et al*. On the impact of tokenizer and parameters on *n*-gram based code analysis. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2018. 437−448.

[28] Karaivanov S, Raychev V, Vechev M. Phrase-based statistical translation of programming languages. In: Proc. of the 2014 ACM Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014). ACM, 2014. 173−184.

[29] Hellendoorn VJ, Devanbu PT, Bacchelli A. Will they like this? Evaluating code contributions with language models. In: Proc. of the 12th IEEE/ACM Working Conf. on Mining Software Repositories. IEEE, 2015. 157−167.

[30] Nguyen AT, Nguyen TD, Phan HD, *et al*. A deep neural network language model with contexts for source code. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. 323−334.

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

[32] Zhang X, Ben KR, Zeng J. A code naturalness based defect prediction method at slice level. Ruan Jian Xue Bao/Journal of Software, 2021,32(7): 2219−2241 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6261.htm [doi: 10.13328/j.cnki.jos.006261]

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

[34] Guerrouj L, Bourque D, Rigby PC. Leveraging informal documentation to summarize classes and methods in context. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Software Engineering. IEEE, 2015. 639−642.

[35] Dam HK, Tran T, Pham T. A deep language model for software code. In: Proc. of the Int'l Symp. on Foundations Software Engineering. 2016. 1−4.

[36] Loyola P, Marrese-Taylor E, Matsuo Y. A neural architecture for generating natural language descriptions from source code. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics. 2017. 287−292.

[37] Bhatia S, Singh R. Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv: 1603.06129, 2016.

[38] Karampatsis RM, Babii H, Robbes R, *et al*. Big code != big vocabulary: Open-vocabulary models for source code. In: Proc. of the ACM/IEEE 42nd Int'l Conf. on Software Engineering. ACM, 2020. 1073−1085.

[39] Li J, Wang Y, Lyu MR, *et al*. Code completion with neural attention and pointer networks. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. 2018. 4159−4165.

[40] Mou L, Li G, Zhang L, *et al*. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2015. 1287−1293.

[41] Gu X, Zhang H, Zhang D, *et al*. Deep api learning. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2016. 631−642.

[42] Li G, Liu H, Jin J, *et al*. Deep learning based identification of suspicious return statements. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). 2020. 480−491.

[43] Arisoy E, Sainath TN, Kingsbury B, *et al*. Deep neural network language models. In: Proc. of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the *N*-gram Model? On the Future of Language Modeling for HLT. 2012. 20−28.

[44] Gu X, Zhang H, Zhang D, *et al*. DeepAM: Migrate apis with multi-modal sequence to sequence learning. In: Proc. of the IJCAI Int'l Joint Conf. on Artificial Intelligence. 2017. 3675−3681.

[45] Pradel M, Sen K. DeepBugs: A learning approach to name-based bug detection. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 1−25.

[46] Hammad M, Babur Ö, Basit HA, *et al*. DeepClone: Modeling clones to generate code predictions. In: Proc. of the Int'l Conf. on Software and Systems Reuse. 2020. 135−151.

[47] Lanchantin J, Gao J. Exploring the naturalness of buggy code with recurrent neural networks. arXiv: 1803.08793, 2018.

[48] Corley CS, Damevski K, Kraft NA. Exploring the use of deep learning for feature location. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2015. 556−560.

[49] White M, Vendome C, Linares-Vasquez M, *et al*. Toward deep learning software repositories. In: Proc. of the 12th IEEE/ACM Working Conf. on Mining Software Repositories. IEEE, 2015. 334−345.

[50] Rahman MdM, Watanobe Y, Nakamura K. A bidirectional LSTM language model for code evaluation and repair. Symmetry, 2021, 13(2): 247.

[51] Hu X, Li G, Xia X, *et al*. Deep code comment generation. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). 2018.

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

[53] Karampatsis RM, Sutton C. Maybe deep neural networks are the best choice for modeling source code. arXiv: 1903.05734, 2019.

[54] Liu F, Li G, Zhao Y, *et al*. Multi-task learning based pre-trained language model for code completion. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). 2020.

[55] Ben-Nun T, Jakobovits AS, Hoefler T. Neural code comprehension: A learnable representation of code semantics. arXiv: 1806.07336, 2018.

[56] Malik RS, Patra J, Pradel M. NL2Type: Inferring javascript function types from natural language information. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 304−315.

[57] Allamanis M, Barr ET, Bird C, *et al*. Suggesting accurate method and class names. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, 2015. 38−49.

[58] Iyer S, Konstas I, Cheung A, *et al*. Summarizing source code using a neural attention model. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol.1: Long Papers). Association for Computational Linguistics, 2016. 2073−2083.

[59] Cummins C, Petoumenos P, Wang Z, *et al*. Synthesizing benchmarks for predictive modeling. In: Proc. of the 2017 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). IEEE, 2017. 86−99.

[60] Gaunt AL, Brockschmidt M, Singh R, *et al*. TerpreT: A probabilistic programming language for program induction. arXiv: 1612.00817, 2016.

[61] Bengio Y, Ducharme R, Vincent P, *et al*. A neural probabilistic language model. Journal of Machine Learning Research, 2003, 3(6): 1137−1155.

[62] Allamanis M, Barr ET, Devanbu P, *et al*. A survey of machine learning for big code and naturalness. ACM Computing Surveys, 2018, 51(4): 1−37.

[63] Bhoopchand A, Rocktäschel T, Barr E, *et al*. Learning python code suggestion with a sparse pointer network. arXiv: 1611.08307, 2016.

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

[65] Liu C, Wang X, Shin R, *et al*. NEURAL code completion. 2017. https://openreview.net/pdf?id=rJbPBt9lg

[66] Raychev V, Bielik P, Vechev M. Probabilistic model for code with decision trees. ACM SIGPLAN Notices, 2016, 51(10): 731−747.

[67] Li J, He P, Zhu J, *et al*. Software defect prediction via convolutional neural network. In: Proc. of the 2017 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2017. 318−328.

[68] Allamanis M, Sutton C. Mining idioms from source code. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE 2014). 2014. 472−483.

[69] Nguyen TT, Pham HV, Vu PM, *et al*. Learning API usages from bytecode: A statistical approach. In: Proc. of the 38th Int'l Conf. on Software Engineering. 2016. 416−427.

[70] Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics. 2017. 1139−1149.

[71] Dowdell T, Zhang H. Language modelling for source code with transformer-xl. arXiv: 2007.15813v1, 2020.

[72] Buratti L, Pujar S, Bornea M, *et al*. Exploring software naturalness through neural language models. arXiv: 2006.12641, 2020.

[73] Mou L, Li G, Zhang L, *et al*. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv: 1409.5718, 2014.

[74] Zhang J, Wang X, Zhang H, *et al*. A novel neural source code representation based on abstract syntax tree. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 783−794.

[75] Kim S, Zhao JM, Tian YC, *et al*. Code prediction by feeding trees to transformers. In: Proc. of the 43rd Int'l Conf. on Software Engineering (ICSE). 2021.

[76] Mou L, Men R, Li G, *et al*. Natural language inference by tree-based convolution and heuristic matching. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. 2016. 130−136.

[77] Chakraborty S, Ding Y, Allamanis M, *et al*. CODIT: Code editing with tree-based neural models. IEEE Trans. on Software Engineering, 2022, 48(4): 1385−1399.

[78] Luong MT, Sutskever I, Le QV, *et al*. Addressing the rare word problem in neural machine translation. arxiv: 1410.8206, 2014.

[79] Chen Z, Kommrusch S, Tufano M, *et al*. SequenceR: Sequence-to-sequence learning for end-to-end program repair. IEEE Trans. on Software Engineering, 2021, 47(9): 1943−1959.

[80] Yang B, Zhang N, Li SP, *et al*. Survey of intelligent code completion. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1435−1453 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5966.htm [doi: 10.13328/j.cnki.jos.005966]

[81] Qiu F, Yan M, Xia X, *et al*. JITO: A tool for just-in-time defect identification and localization. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1586−1590.

[82] Yan M, Xia X, Fan Y, *et al*. Just-in-time defect identification and localization: A two-phase framework. IEEE Trans. on Software Engineering, 2022, 48(1): 82−101.

[83] Hu X, Li G, Xia X, *et al*. Summarizing source code with transferred api knowledge. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence (IJCAI 2018). 2018. 2269−2275.

[84] Raychev V, Vechev M, Krause A. Predicting program properties from "big code." In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2015). ACM, 2015. 111−124.

[85] Allamanis M, Barr ET, Bird C, *et al*. Learning natural coding conventions. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE 2014). 2014. 281−293.

[86] Papineni K, Roukos S, Ward T, *et al*. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting on Association for Computational Linguistics (ACL 2002). 2001. [doi: 10.3115/1073083.1073135]

[87] Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. arXiv: 1508.07909, 2015.

[88] Wang YG, Zheng Z, Li Y. word2vec-ACV: Word vector generation model of OOV context meaning. Application Research of Computers, 2019, 36(6): 1623−1628 (in Chinese with English abstract).

[89] Jean S, Cho K, Memisevic R, *et al*. On using very large target vocabulary for neural machine translation. In: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th Int'l Joint Conf. on Natural Language Processing. 2015. 1−10.

[90] Liu Z, Xia X, Hassan AE, *et al*. 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 (ASE 2018). ACM, 2018. 373−384.

附中文参考文献:

[32] 张献, 贲可荣, 曾杰. 基于代码自然性的切片粒度缺陷预测方法. 软件学报, 2021, 32(7): 2219−2241. http://www.jos.org.cn/1000-9825/6261.htm [doi: 10.13328/j.cnki.jos.006261]

[80] 杨博, 张能, 李善平, 等. 智能代码补全研究综述. 软件学报, 2020, 31(5): 1435−1453. http://www.jos.org.cn/1000-9825/5966.htm [doi: 10.13328/j.cnki.jos.005966]

[88] 王永贵, 郑泽, 李玥. word2vec-ACV: OOV 语境含义的词向量生成模型. 计算机应用研究, 2019, 36(6): 1623−1628.

陈浙哲(1997－), 女, 学士, 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, 研究员, 博士生导师, 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.

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.

刘忠鑫(1994－), 男, CCF 专业会员, 主要研究领域为智能化软件工程, 软件文档自动生成.

徐洲(1990－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为软件仓库挖掘, 软件缺陷预测.

雷晏(1985－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件错误定位, 软件自动修复.