软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1736−1757 [doi: 10.13328/j.cnki.jos.006559]

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

基于对象类型的 API 补全方法*

唐 泽, 李传艺, 葛季栋, 骆 斌

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

通信作者: 李传艺, E-mail: lcy@nju.edu.cn

摘 要: 近年来, 随着软件技术在各行各业、不同领域的应用不断扩展与深入, 同时伴随着软件架构、服务计算等技术的不断发展, 软件行业涌现出了功能丰富且规模庞大的第三方 API 或库, 软件开发者在实现软件功能的时候也越来越依赖这些 API. 但学习这些 API 的使用是非常困难且耗时的, 主要有两方面的原因: 1) 相关文档的缺失和错误; 2) 相关 API 用法的示例代码较少. 因此, 研究自动的 API 补全方法以帮助开发人员在开发过程中正确且快速的使用 API, 具有很大的应用价值. 然而, 现有 API 自动补全方案多数将待补全代码段看作纯文本, 忽略了 API 所属对象类型对预测 API 的影响. 为此, 探究了对象类型对补全 API 的作用, 并且在对象状态图的启发下, 设计了一种使用 API 所属对象的类型作为特征的补全方法. 具体而言, 首先从 API 调用序列中先抽取同一对象类型的子序列, 利用深度学习模型编码出每个对象的状态, 再利用对象状态生成整个方法块的状态表示进行补全. 为了验证提出的补全方法, 在 6 个流行 Java 项目上进行了验证. 实验结果证明, 提出的考虑对象类型的 API 补全方法在预测准确率上明显高于基线模型.

关键词: API 补全; 对象类型; 插件

中图法分类号: TP311

中文引用格式: 唐泽, 李传艺, 葛季栋, 骆斌. 基于对象类型的API补全方法. 软件学报, 2022, 33(5): 1736–1757. http://www.jos.org.cn/1000-9825/6559.htm

英文引用格式: Tang Z, Li CY, Ge JD, Luo B. Method of API Completion Based on Object Type. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1736–1757 (in Chinese). http://www.jos.org.cn/1000-9825/6559.htm

Method of API Completion Based on Object Type

TANG Ze, LI Chuan-Yi, GE Ji-Dong, LUO Bin

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: In recent years, with the continuous expansion and deepening of the application of software technology in various industries and fields, as well as the development of software architecture, services computing, etc., the software industry has emerged with feature-rich and large-scale third-party APIs or Libraries. Software developers are increasingly relying on these APIs when implementing software functions. However, learning the usage of these APIs is very difficult and time-consuming. There are two main reasons: 1) missing or wrong documents; 2) few sample codes for API usage. Therefore, designing automatic API completion methods to help developers use the API correctly and quickly has great application value. However, most of the existing API automatic completion methods regard the code segments to be completed as plain text, ignore the impact of the object types of APIs. Therefore, this study explores the role of the object types in completing APIs. Besides, inspired by the object state diagram, an concrete API completion method is designed and implemented that uses the types of the objects as a novel feature. Specifically, the subsequence of the same object type is first extracted from the API call sequence and a deep learning model is used to encode the state of each object. Then, the objects' states is used to generate a state

* 基金项目: 国家自然科学基金 (61802167, 61972197, 61802095); 江苏省自然科学基金 (BK20201250); 华为-南京大学下一代程序设计创新实验室合作协议子项目

本文由"领域软件工程"专题特约编辑汤恩义副教授、江贺教授、陈俊洁副教授、李必信教授以及唐滨副教授推荐.

收稿时间: 2021-08-11; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

representation of the entire method block. In order to evaluate the proposed method, comprehensive experiments are conducted on six popular java projects. The experimental results prove that the proposed API completion method achieves significantly higher predicting accuracy than the baseline approaches.

Key words: API completion; object type; plug-in

1 引 言

在软件开发过程中, 为了避免重复工作, 提高软件开发效率, 软件开发者通常会使用一些第三方的 API (application programming interface), 即应用编程接口, 来协助完成软件的功能. 这些第三方库 (Library) 的 API 通过提供对象、方法以及变量来让软件开发者调用, 借以实现一系列特定需求或功能[1]. 通过调用这些 API, 软件开发者可以实现某种功能而无需关注其内部的复杂细节; 减少代码出错的概率; 提高软件开发效率以及更加便捷地复用代码.

然而, 学会正确地使用这些 API 需要耗费软件开发者大量的时间. 一部分原因是因为这些第三方库在不断更新迭代, 另外 API 说明文档的缺失或者出错也使得学习成本增加[2]. 在这种情况下, 软件开发者不得不去技术论坛、博客或者 Github 上寻求帮助, 有时甚至需要阅读源代码来寻找这些 API 的正确使用方法. 由于第三方库的庞大数量, 即使是使用过的 API (比如 JDK, 最新版本中类已经达到 3 000 多个), 一段时间后, 软件开发者也很难清晰记住对应 API 的名称和用法. 微软的一项研究表明, 67.6% 的受访者提到在学习 API 的过程中受到了资源不足的困扰[3]. 软件开发者往往需要多次重复学习过程, 极大地影响软件开发的效率[4].

为了帮助软件开发者正确且快捷地使用这些第三方 API, 一系列 API 方法补全工具应运而生. API 方法补全是指当程序员已经拼写出一个对象名之后, 通过查询这个对象所属类别列出所有的方法供程序员选择. 图 1 展示了在 IDEA 中智能提示 API 方法的例子, 但是集成开发环境中这些智能提示只是将所有可能的 API 按照使用频率从高到低排列. 通过这种方式推荐的 API 许多是与当前方法块毫不相关的, 开发者仍需要在众多的候选 API 中挑选出正确的 API[5]. 为解决这种问题, 有研究者提出使用数据挖掘算法挖掘 API 使用模式进行推荐. 这些算法通常是从大量的源代码中抽取出 API 调用序列, 以模式识别或者传统序列模型的方法来研究 API 的使用模式, 进而实现 API 自动补全.

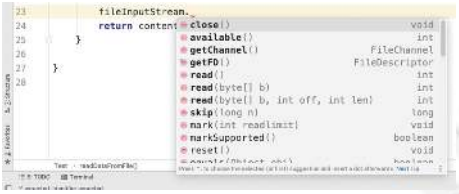

图 1 IDEA 中智能补全 API 的例子

API 自动补全的意义可以总结为以下几点.

1) 避免开发人员需要记住每个 API 的用法. 由于目前第三方 API 的数量已经非常庞大, 即使编程经验丰富的开发人员也几乎不可能记住每个 API 的具体用法. 举例来说, 当一个开发人员需要在 Java 代码中比较两个字符串的是否相似 (忽略大小写). 他应该调用 String 对象中的 equalsIgnoreCase 这个 API 来完成这个任务, 但是很有可能他不记得这个 API 的全称. 当他在一个 String 类型的对象后输入点 (.), API 补全系统会按照当前代码上下文, 分析 String 对象中每个 API 在这里使用的可能性, 排序展示给开发人员. 假设 API 补全系统已经学习到 equalsIgnoreCase 这个 API 的用法, 系统会将这个 API 排名在靠前的位置返回给开发人员. 开发人员可以轻松找到这个 API, 并利用它来实现代码功能.

2) 帮助开发人员书写错误更少的代码. API 补全系统只会返回目标对象中已有的 API 供用户选择, 这可以保证补全结果至少在语法上正确. 比如用户需要系统对 java.swing 包中的 JFrame 对象进行补全, 系统只会分析

JFrame 中已有的 API 来推荐给用户. 这可以保证不会推荐生成无法编译的代码.

3) 加快开发人员书写代码的速度. 为了帮助开发人员更容易理解 API 的用法, API 的名称通常体现了一部分这个 API 的功能描述. 这也造成有些 API 的名称可能过于复杂, 人工输入这些 API 的名称非常费时. API 补全系统可以帮助开发人员补全 API 方法的调用, 而无需要开发人员手动输入, 这可以加速代码书写速度. 有些 API 补全工具甚至可以支持多行补全, 比如自动补全 try-catch 结构块[6].

4) 代码复用. 开发人员为了避免重复开发, 通常使用复制、粘贴的方式来复用相关代码. 这种方式也被称作代码克隆 (code clone). 有研究表明, 软件系统中克隆代码的比例通常在 15%–25% 之间[7], 更有甚者可以达到 50%[8]. 但是, 这种代码克隆方式需要开发人员可以拿到相关的源代码, 并人工分析哪块代码可以被复用. API 补全系统提供了另一种代码克隆的方式, 因为 API 补全技术本质上也是从代码库中寻找与当前代码块相似的代码, 复用相似代码中的 API 使用模式, 提供相关的 API 补全建议.

为了满足开发人员实际编程需要, 提供切实可用的 API 方法补全, 本文的主要研究目标在于设计和实现一个考虑对象特征的 API 补全技术, 同时实现一个支持离线和在线的集成开发环境插件, 提供 API 方法补全服务.

本文的主要研究内容包含 3 部分: 一是从代码库中构建高质量的 API 调用序列; 二是研究利用对象信息的 API 补全技术; 三是支持离线和在线的 API 补全集成开发环境插件.

2 相关工作

2.1 基于规则的方法

Bruch 等人[5]提出了 3 种 API 方法补全的技术. 一种是基于使用频次的 API 补全方法, 推荐使用频率最高的 API. 一种是基于关联规则的 API 补全, 推荐经常一起出现的 API. 第 3 种方式是使用 K 近邻算法来进行 API 补全 (best matching neighbor, BMN), 该方法先统计代码库中出现的所有的 API 名称, 对象类型, 将这些关键词作为特征, 并利用这些特征表示方法. 方法中出现过该特征, 用 1 表示, 未出现过该特征, 用 0 表示. 这样就可以将方法表示成特征对应的 0/1 编码. 在进行 API 补全时, 通过方法编码间的汉明距离找出代码库中与当前待补全方法最相似的方法. 最后比对这两个方法编码区别, 找出待补全方法中未出现, 而相似方法中出现的 API 调用, 将它作为补全推荐. 实验表明, Bruch 提出的第 3 种方法效果最好. 但是考虑到 k 近邻算法在推荐过程中的时间开销, 该方法可能不太适合用在实际开发环境中. 同时, 由于该方法忽略了 API 调用的相对顺序, 其推荐准确率也需要考量.

Proksh 等人[9]提出了一个基于贝叶斯网络的模式识别算法 (pattern-based Bayesian networks, PBN), 使用贝叶斯网络和额外代码信息改进 BMN 算法. PBN 额外使用参数数组、类上下文和定义类型作为补充特征来编码方法, 然后通过贝叶斯网络来训练补全模型, 实验表明, PBN 与 BMN 相比, 大约可以提升 3 个百分点.

Heinemann 等人[10]使用了变量名称作为额外的特征用于 API 补全. 该方法默认变量命名使用驼峰法书写. 驼峰法是指在需要使用多个单词来表示变量名时, 将单词的首字母大写进行拼接. 他们通过字母是否大写来切分变量名, 获取多个有意义词汇, 并利用这些词汇作为特征, 使用类似于 BMN 的方法, 将方法表示成特征对应的编码向量, 使用 K 近邻算法来寻找与当前待补全方法最接近的代码, 他们使用 Jaccard 相似系数来定义方法特征向量之间的距离.

Nguyen 等人[11]提出了一种基于数据控制流图来进行 API 补全的方法, 称作 Grapacc. Grapacc 将代码转化为控制流图, 按图中的节点将图分解为单节点、双节点、三节点以及四节点子图, 构建了一个图数据库. 在需要补全 API 时, 从待补全方法中抽取出控制流子图, 去图数据库中寻找这些子图对应的父图 (父图是子图添加一个节点和 N 个边得到的图). 并基于这些父图, 统计新增节点对应 API 出现的次数, 选择出现频次最高的 API 进行补全. 这种方法虽然在捕获上下文信息时比序列模型要好, 但是需要构建一个大型的图数据库, 而且子图的数量会随节点个数呈指数级上升, 这给搜索时的空间使用和时间效率都带了影响.

2.2 基于统计语言模型的方法

使用统计语言模型的 API 补全方法将代码看作一种语言, 通过语言模型对代码或者从代码提取的 API 调用

序列建模. Akbar 等人[12]使用多层级的序列挖掘技术学习 API 使用范式, 利用范式进行 API 补全. Asaduzzaman 等人[13]提出了一种上下文敏感的 API 补全方法. Amorim 等人[14]通过在代码中注入占位符将代码变得有空缺, 再将占位符替换成代码模版进行补全. Hu 等人[15]在 API 方法补全时, 额外考虑用户已经键入的单词缩写, 并使用逻辑回归模型提取输入特征, 最后通过支持向量机模型 (support vector machine, SVM) 推荐补全.

Nguyen 等人[16]提出了一种基于 API 调用序列的补全方法 HAPI (hidden Markov model of API usages), 用来改进 Grapacc 方法. 先从控制流图中抽取出 API 调用序列, 再使用隐马尔可夫模型来补全 API 空缺. 该灵感来源于对象状态图, 在对象状态图中, API 的调用会引起对象状态发生改变, 继而影响之后的 API 调用. HAPI 将对象状态作为隐藏状态, API 调用作为观测状态, 使用隐马尔可夫模型对 API 序列进行建模, 对空缺位置进行 API 推荐. 但是 HAPI 模型需要为每个对象建立一个隐马尔可夫模型, 如果需要对经常共现的多个对象一起建模, 还需要额外训练一个模型. 这意味着需要占用大量的存储空间来存储模型. 同时在预测时, 在确定了待补全方法块中涉及的对象类型后, 需要加载对应的隐马尔可夫模型到内存, 再进行补全推荐.

还有使用深度学习模型实现 API 补全任务的工作. 例如, Rayche 等人[17]使用 N-gram 模型进行 API 补全. Gvero 等人[18]研究了 Scala 语言的 API 补全方法, 通过将类型 (type) 信息转化为等价的对象 (class) 信息, 以减少 API 补全时的搜索空间, 并利用代码库中学习到的 API 权重对方法进行补全. Roos 等人[19]使用 N-gram 模型加集束搜索 (beam search) 的方式补全空缺内容. Savchenko 等人[20]使用 N-gram 模型对代码序列建模, 并在确定的 API 候选集上给出补全推荐. Yan 等人[21]提出了 APIHelper 模型补全 API 调用. 该方法在传统长短期记忆模型 (long short-term memory, LSTM) 基础上, 使用了 API 拼接编码和确定负采样技术.API 拼接编码是指分别对对象类型和 API 进行编码, 将这两个词向量拼接后作为 LSTM 的一次输入. 确定负采样技术是指在预测时, 利用空缺位置的对象类型来确定候选的 API 调用集, 将其作为采样出的样本传入 Softmax 层进行预测. 实验表明, 该方法的补全效果超过了 HAPI, 并且可以更快速给出补全建议. 与之相似的工作还有 Svyatkovskiy[22]使用 LSTM 模型对 Python 语言中的一些常用包中的常用 API 进行补全, 但与 Yan 使用 API 调用序列作为输入不同, Svyatkovskiy 先从代码中分析出的抽象语法树, 直接将树的先序遍历结果作为 LSTM 的输入. Nguyen 等人[23]利用程序分析技术从代码语料库中提取出模版, 再利用深度学习模型对候选模版进行排序推荐. Chen 等人[24]提出了 DeepAPIRec 模型, 该模型是利用 Tree-LSTM 对从源码提取出的抽象语法树进行建模, 推荐空缺位置的 API. 同时利用数据流分析技术来补全 API 参数.

另外, 由于代码中存在大量自定义的方法名和变量名, 这些词无法使用词表进行记录, 而这些词在实际 API 补全中具有重要意义[25]. 为了解决这类未收入词的补全问题. Yang 等人[26]提出了 REP 模型, 在补全时额外分析是否利用代码序列中已经出现的词进行补全. Li 等人[27]使用注意力机制和 Pointer 拷贝技术来解决这些未收入词的预测问题. Terada 等人[28]使用 LSTM 模型来编码代码块, 再将编码结果输入到 Pointer Network 中预测补全位置是代码中已出现的变量名的概率. 如果变量名对应的最大概率超过 0.5, 选择概率值最大的变量名进行补全. 否则的话, 使用 LSTM 模型的输出对词表中的词进行分析, 推荐词表中对应补全概率最大的词. Yang 等人[29]提出了一种新的代码抽象语法树的编码方式, 并且在预测时不将 UNK 放入准确率计算中, 这样可以避免这些未收入词对补全准确率的影响. 因为未收入词在测试集中都编码成 UNK, 如果预测 UNK 就算正确的话, 模型会偏好预测 UNK 来减少训练损失, 但是这种做法对实际补全并没有意义.

3 背景知识

3.1 长短期记忆模型

长短期记忆模型[30]是一种时间循环神经网络, 被广泛应用于文本处理. 它是为了解决传统时间循环网络中存在的长期依赖问题而被专门设计出来的. 传统循环网络在利用反向传播 (back propagation, BP) 时, 误差会逐级减少. 这导致计算过程中, 梯度会随着时间序列的增长而指数下降, 导致网络权重更新缓慢.LSTM 采用门控机制来解决这个问题, 门控机制是由一个 Sigmoid 函数来实现的, Sigmoid 的函数定义如下所示:

$$\sigma(x)=\frac{1}{1+\exp(-x)} \tag{1}$$

这个函数可以把输入值规划到 0–1 之间. 0 表示门关闭, 即任何信息都不传送; 1 表示门打开, 任何信息都会被传送; 其他值表示以一定的比例将信息传送入门内. LSTM 设计了 3 个门: 输入门、输出门和遗忘门. 3 个门的计算方式如下:

$$\begin{bmatrix} i_t \\ f_t \\ o_t \end{bmatrix}=\begin{bmatrix} \sigma \\ \sigma \\ \sigma \end{bmatrix}(W[h_{t-1};x_t]+b) \tag{2}$$

其中, h_{t-1} 表示上一时间的隐藏状态, x_t 表示当前时间的输入. i_t, f_t, o_t 分别表示输入门, 遗忘门和输出门. 3 个门的计算方式类似, 都是使用上一时间的隐藏状态 h_{t-1} 和当前输入 x_t 输入到全连接网络中得到. 其中, 输入门是控制是否将当前时间的输入信息加入隐藏状态中. 遗忘门是控制是否选择遗忘过去的某些信息. 输出门是控制多少信息输出到结果中. 整个门的使用可以表示为以下公式:

$$\begin{aligned} \widehat{c_t}&=\tanh(W_c[h_{t-1};x_t]+b_c) \\ c_t&=f_t\odot c_{t-1}+i_t\odot\widehat{c_t} \\ h_t&=o_t\odot\tanh(c_t) \end{aligned} \tag{3}$$

其中, c_t 是内部状态 (cell state), 它是整个模型的记忆空间, 起到类似于传送带的效果. 一次模型计算的过程如下, 给定当前输入 x_t 和上一时间模型隐藏状态 h_{t-1}. 模型会根据 x_t 和 h_{t-1} 计算当前输入的记忆状态. 接着, 根据遗忘门 f_t 和上一时间记忆状态 c_{t-1} 的点积选择保留多少之前的记忆, 根据输入门 i_t 和当前记忆状态 $\widehat{c_t}$ 的点积选择传入多少当前记忆. 新的记忆空间状态 c_t 由这两部分记忆状态相加得到. 输出门 o_t 用来表示选择输出多少记忆空间内容来当做当前模型的隐藏状态 h_t . 为了增加长短期记忆模型的表达能力, 另一种更常用的方法是使用双向长短期记忆模型 (bidirectional LSTM) 来正向和逆向地编码序列信息. 对于一个序列向量 $[x_1,x_2,\ldots,x_T]$, 前向 LSTM 从 x_1 到 x_T 读取整个序列, 逆向 LSTM 反过来, 从 x_T 到 x_1 读取整个序列. 对于每个时间 t 对象的输入向量 x_t , 将前向 LSTM 得到的隐藏状态 $\overrightarrow{h_t}$ 和逆向 LSTM 得到的隐藏状态 $\overleftarrow{h_t}$ 拼接起来的向量 $[\overrightarrow{h_t};\overleftarrow{h_t}]$ 作为当前输入的隐藏状态.

3.2 注意力机制

注意力机制是由 Chorowski 等人为机器翻译提出的[31]. 其动机来源于人类阅读文本时, 会对某些重要的词信息给予额外的关注. 它首先被应用于 Encoder-Decoder 模型, 即当需要翻译某个词时, 对源语言中的某些词进行特殊关注. 通过给定查询向量 $\overrightarrow{s}$, 对目标词 $\overrightarrow{h}$ 计算权重 α . 然后基于 α 求出整个语句序列的加权和, 用来表示整个语句对于查询向量的状态表示. 计算权重的方式主要分为以下 4 种方式:

$$\alpha_{sh}=\begin{cases} s^Th \\ s^TW_ah \\ W_a[s;h] \\ v_a^T\tanh(W_as+U_ah) \end{cases} \tag{4}$$

其中, 第 1 种方式是直接将查询向量与目标词进行点积, 点积结果越大表示对该词给予越多的关注. 第 2 种方式与第 1 种方式类似, 只是因为点积需要 $\overrightarrow{s}$ 和 $\overrightarrow{h}$ 的维度相同, 第 2 种方式通过加入矩阵 W_a 来提升第 1 种方式的通用性. 第 3 种方式是直接将 $\overrightarrow{s}$ 和 $\overrightarrow{h}$ 拼接后输入到一个全连接网络中, 输出一个一维向量. 用这个一维向量的值来表示查询向量对该词的权重. 第 4 种方式是对第 3 种方式的一种改进, 通常使用两个矩阵分别与查询向量和词向量相乘, 再输入到全连接网络, 输出一个 k 维向量, 再与一个 k 维的向量 v_a^T 点积后得到目标词对查询向量的权重. 最后依据权重对所有的求加权和.

通过使用注意力机制, 模型可以动态地关注有助于执行当前任务的输入的某些部分, 将这种相关性概念结合起来. 神经网络中注意力机制快速发展的原因主要有 3 点. 首先, 它可以被应用于多种类型的任务中, 比如机器翻译[32]、问题回答[33]、情绪分析、词性标注和对话系统. 其次, 除了可以帮助任务提升性能外, 注意力机制还可以提高模型的可解释性, 注意力机制产生的副产物: 对齐分数矩阵可以帮助理解模型学习到了什么样的特征. 第三, 注意力机制还可以一定程度上克服循环神经网络中的一些挑战, 例如随着输入长度的增加造成的性能下降, 以及

输入顺序不合理导致的计算效率低下.

4 基于对象类型的 API 补全技术

4.1 API 补全问题描述与建模

为了解决 API 补全的问题, 本节首先给出了相应的问题定义, 并基于问题描述进行建模.

问题: API 自动补全.

条件: (1) 待补全的方法块, 以及 (2) 待补全位置和待补全位置的对象类型.

目标: 对待补全位置推荐可能的 API 调用.

对于 API 补全问题, 首先确定代码方法块的待补全位置, 并利用静态分析技术抽取出方法中的 API 调用, 构建 API 调用序列. 如图 2 所示, 依据 API 调用序列和待补全位置的对象类型从候选 API 中给出补全推荐.

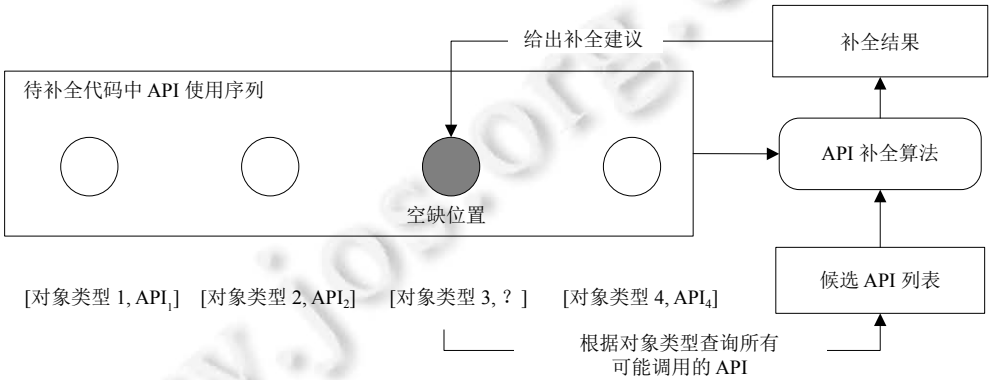

图 2 API 补全问题建模

4.2 设计动机

本文研究的方法是基于 API 调用序列的 API 补全模型. 现有的使用 API 调用序列进行补全的模型[16,21]都是将 API 序列当作纯文本序列来处理, 而忽略了 API 调用序列的部分有序性[34]. 部分有序性是 API 序列中任意两个 API 调用之间不一定存在先后顺序. 举例来说, 图 3 是在 Java 程序中调用 HttpClient 包实现 http 请求的例子.

```
public void httpGet(String url) throws Exception{
    CloseableHttpClient httpClient = HttpClients.createDefault();
    RequestConfig requestConfig = RequestConfig.custom()
                                                .setConnectTimeout(5000)
                                                .build();
    HttpGet httpGet = new HttpGet(url);
     httpGet.setConfig(requestConfig);
    httpCLient.execute(httpGet);
     …
}
```

图 3 使用 HttpClient 实现 http 请求的 Java 代码

从图 3 代码可以看出, RequestConfig.custom 和 HttpGet.init 两个 API 之间并没有顺序关系, 虽然形式上看 RequestCofig.custom 在 HttpGet.init 方法之前被调用, 但由于这两 API 之间并不存在数据依赖关系, 交换这两个 API 的调用顺序并不会影响代码的功能实现和正确性. 利用数据控制流图可以更加直观的看出这种部分有序性的特点. 图 4 展示了上段代码的数据控制流图. 由于数据控制流图是一个有向无环图, 因此只有当两个 API 调用之间存在通路时, 这两个 API 才存在数据依赖, 也才会产生先后顺序关系. 举例来说, RequestConfig.custom 和 HttpGet.setConfig 这两个方法之间存在通路, 那这两个 API 之间有先后顺序关系. 而 RequestConfig.custom 和 HttpGet.init 之间并没有通路, 因此他们之间就没有先后顺序关系.

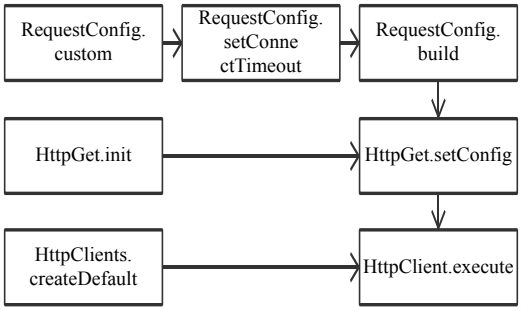

图 4 http 请求的数据控制流图

那么, 为什么 API 调用序列会有这种部分有序性的特点呢? 其主要原因是由于面向对象语言具有封装性, 即对象行为 (方法调用) 原则上只能改变对象内部属性 (私有变量). 也就是说, API 的调用 (API 也是一种特殊的公开的方法) 只能改变其所属对象的属性, 从而引起其所属对象的状态发生改变, 并不会影响到其他对象状态. 比如, RequestConfig.custom 和 HttpGet.init 之间并没有顺序关系, 因为它们影响的只是各自所属对象的状态, 互相之间并没有干扰. 但是, 如果简单地认为不同对象的 API 调用之间毫无关系, 那也过于武断了. 比如 RequestConfig.build 和 HttpGet.setConfig 这两个不同对象的 API 之间就存在明显的顺序关系, 这又是因为什么呢?

先设想这样一个例子: 多媒体播放有且仅有两个 API: 播放视频 playVideo 以及播放音乐 playMusic. 通常情况下, 播放视频 playVideo 这个 API 调用的频数多一些, 不妨假设调用 playVideo 和 playMusic 的概率分别为 0.8 和 0.2. 但如果代码上下文已经有另一个对象调用了指定音乐源这个 API, 那么此时显然调用 playMusic 这个 API 更加合理. 这是因为上下文相关的 API 调用概率实质上是一个条件概率, 在这个例子中就是已知音乐源情况下的调用概率, 即 $P\left(x \in \{\text{playVideo}, \text{playMusic}\} | \text{音乐源}\right)$. 虽然申明音乐源属于另一个对象的 API, 但是它的确影响了多媒体播放中 API 调用的选择. 因此, 本文将 API 调用的规律总结如下: 虽然 API 调用只会影响其所属对象状态发生改变, 但是对象会选择什么动作, 即选择哪一个 API 进行调用, 不仅仅受到自身状态的影响, 还会受到其他对象状态的影响.

基于这种思路, 本文设计了一种新的处理 API 调用序列的方式, 如图 5 所示, 不同于传统的按顺序分析 API 调用序列, 再进行补全的方式, 先将属于同一个对象的 API 调用聚集在一起, 提取出对象状态. 再根据所有的对象状态共同决定空缺位置应该调用哪个 API.

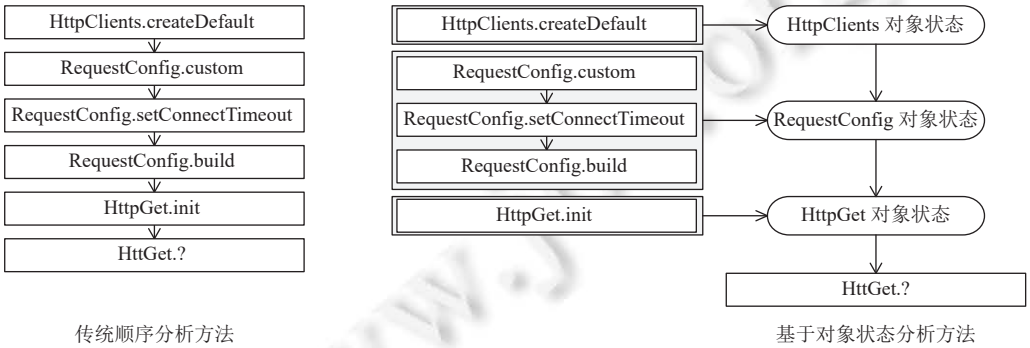

图 5 分析 API 调用序列的不同方式

4.3 模型设计

基于对象类型的 API 补全模型整体框架图如图 6 所示. 首先从待补全代码中提取出 API 调用序列, 待补全位置会用 Hole 进行替换. 按照第 4.2 节的设计动机, API 调用序列会按照对象类型进行分组. 属于同一个对象类型

的 API 调用会被分到同一个组中. 组与组之间顺序由组内最后一个 API 调用位置决定. 将分组后的 API 调用输入补全模型. 模型包括 3 个部分: 编码层, 层次网络以及预测层, 如图 7 所示. 接下来将详细介绍这 3 部分.

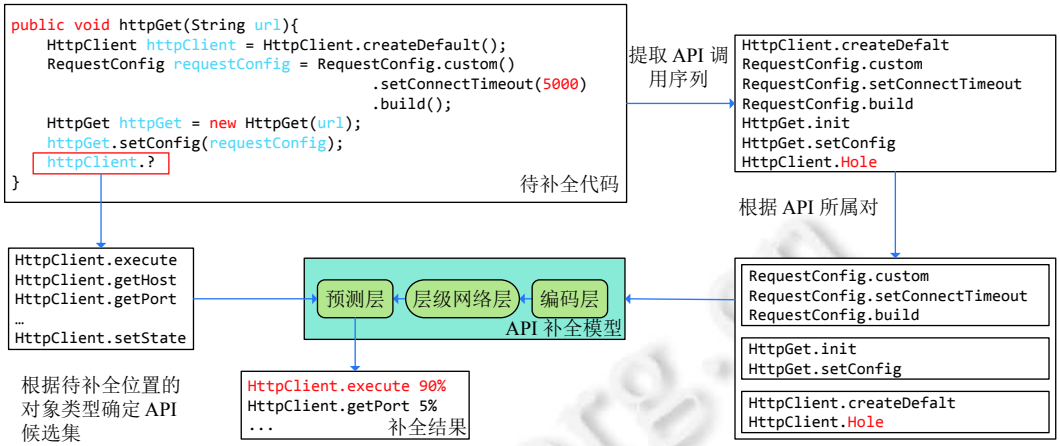

图 6 API 补全总体框架

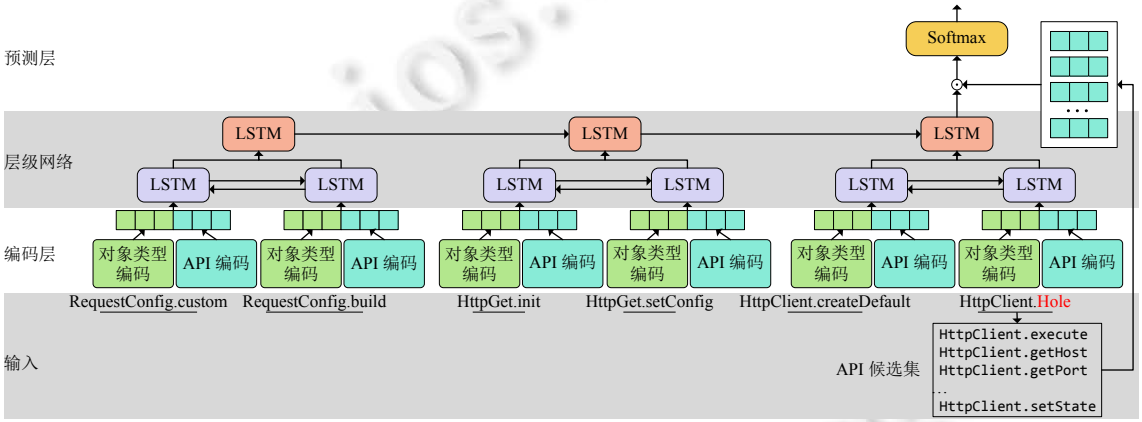

图 7 API 补全模型结构图

1) 编码层: 在获取到 API 调用序列后, 需要先对每个 API 调用进行编码, 才可以将它们输入到神经网络模型中. 有两种直观的思路来对 API 调用序列进行编码: 一种是直接对所有的 API 进行编码, 即把“对象.API”这样一个结构看成是一个整体, 对这个整体进行索引, 构建词汇表. 但这样做的缺点是会忽略掉对象类型信息, 即同一个对象的 API 调用无法直接根据向量编码看出它们的关系. 另一个直观的思路是将“对象.API”这样一个结构拆分开, 即对对象类型和 API 调用分别编码, 并把它们各自当作一个时间节点, 以“对象类型->API 调用-> 对象类型->API 调用…”这样一种形式的序列输入到神经网络中. 这样做的好处是可以让模型显性地知道当前 API 的对象类型, 但是这样做会导致原本的序列长度会扩张成原有长度的两倍. 而序列模型, 包括长短期记忆模型, 其实都不适合用来处理长度过长的序列, 因此这样做的效果并不好.

在本模型中, 本文使用了两个词嵌入矩阵 W_c^e 和 W_a^e 用来分别对应对象类型和 API 的嵌入层. $c_t.a_t$ 表示时间 t 的 API 调用, 通过公式 (5):

$$e_t^c = W_c^{e[c_t]};\ e_t^a = W_a^{e[a_t]} \tag{5}$$

分别对对象类型和 API 进行编码, 然后将这两个词向量表示拼接在一起, 得到时间 t 的输入, 即:

$$x_t = [e_t^c; e_t^a] \tag{6}$$

这样就可以在不改变输入序列长度的情况下, 显式地告知模型同一个对象内 API 调用的联系. 因为若两个 API 调用属于同一个对象, 那这两个 API 调用对应的编码前一半是一样的. 同时, 因为空缺位置的 API 调用是未知的, 本文使用“hole”来表示空缺位置的 API 调用. “hole”一词在 API 编码时被当作一个特殊的 API, 它在 API 词汇表中的序号是 1 (0 是 UNK, 表示不在词汇表的 API 调用). 举例来说, $c_t.hole$ 表示该序列的空缺位置在时间 t, 同时空缺位置的对象类型是 c_t. 这样就可以保持输入时的编码格式的统一.

2) 层级网络层: 层级网络 (hierarchical network) 主要应用于自然语言处理中文本摘要领域. 由于文本信息天然具有层级性: 词构成句, 句构成文章. 因此通常的做法是先由词提取出句子级别的特征, 再基于句子级别的特征提取出整篇文章的重点信息. 本文的思想与之类似, 如果将词映射为 API 调用, 句子特征映射为对象状态. 对象状态可以从属于该对象的 API 调用中提取, 而整个方法的特征可以从全部对象状态中提取.

首先在对象层, 根据空缺位置将 API 调用序列划分为 3 部分: 空缺位置之前的 API 调用, 空缺位以及空缺位置之后的 API 调用. 对于空缺位置之前的 API 调用, 先按照对象类型抽取出同属于一个对象的 API 调用子序列, 并将每个调用子序列使用 LSTM 来编码对象状态, 编码对象状态过程如公式 (7) 所示:

$$\left.\begin{aligned} h_t &= LSTM(x_t) \\ \alpha_{ct} &= \frac{\exp(e_t^c W_c h_t)}{\sum_j \exp\left(e_j^c W_c h_t\right)} \\ h_c &= \sum_j \alpha_{cj} h_j \end{aligned}\right\} \tag{7}$$

其中, x_t 是 t 时刻的 API 编码表示, e_t^c 是 t 时刻 API 的对象类型的词嵌入表示. α_{ct} 是 t 时刻 API 调用在其对象状态中所占权重. 对象 c 的状态由其 API 子序列中的隐藏状态乘以对应权重累加得到. 对于空缺位置及空缺位置之后的 API 序列, 使用相同的方法得到对应的对象状态. 接着, 将对象状态按照其子序列中最后一个 API 调用在整体序列中的位置进行排序, 得到最终的对象状态序列.

接下来需要获取整个 API 调用序列对应的方法状态. 由于空缺位置不确定, 因此需要设计一个空缺位置感知的网络模型来编码对象状态. 同时根据程序局部性原理, 模型应当更加关注空缺位置周围的对象状态. 本文设计了一个双向的 LSTM 来解决这一问题. 对于空缺位置和空缺位置之前的对象状态序列, 将它输入到一个正向的 LSTM 模型中, 得到空缺位置之前的方法状态 $\overrightarrow{h_f}$. 相似的, 将空缺位置和空缺位置之后的对象状态序列输入到逆向的 LSTM 模型中, 得到空缺位置之后的方法状态 $\overleftarrow{h_f}$. 然后将这两个状态拼接起来, 得到最终的方法状态表示 [$\overrightarrow{h_f}$; $\overleftarrow{h_f}$]. 特别地, 对于空缺位置在 API 序列末尾的, $\overleftarrow{h_f}$ 为全零向量; 空缺位置在 API 序列开头的, $\overrightarrow{h_f}$ 为全零向量. 接下来, 将阐述利用方法状态进行补全推荐的过程.

3) 预测层:上一小节中介绍了如何利用对象信息对 API 序列进行编码的过程, 本节将介绍如何利用序列编码进行空缺位置补全. 常见的预测补全位置 API 的方法是将方法状态输入到一个与维度等同于全体 API 的全连接网络中, 再将结果输入到 Softmax 层, 概率最大的位置对应的 API 即最终推荐的补全 API. 但是, 这样做有两个缺点: 首先, 全体 API 的数量非常大, 在本文的数据集中, 全体 API 的数量是 35 380 个, 这意味着需要使用一个十分庞大的全连接网络来进行预测, 继而引发内存占用和时间效率的问题. 另一方面, 由于空缺位置对象类型已知, 不属于该类型的 API 不应该被推荐, 否则会导致程序语法错误. 因此本文设计了一个基于补全位置对象类型的预测机制.

首先, 根据空缺位置对象类型确定 API 候选集. 因为空缺位置对象类型已知, 不属于该对象的 API 不可能被调用, 因此可以根据对象类型中的全部 API 确定调用候选集. 但是这会引起另一个问题, 因为每个对象类型中的 API 数量不固定, 这意味着不同对象类型的补全对应的概率输出维度也不相同. 本文采用了类似于 Word2Vec[35]中的思想, 使用两个向量的余弦距离来表示两个向量之间的相似度. 具体做法是, 首先将方法状态输入到一个全连接网络中, 将方法状态向量转化成和 API 编码维度相同. 然后将 API 候选集中的 API 通过 API 嵌入层查询其对应的向量表示. 再计算方法状态向量与候选 API 向量的内积, 并将结果通过 Softmax 函数转化为概率表示, 从而确定每

个候选 API 对应的推荐可能性. 具体过程如公式 (8) 所示:

$$o_f = \tanh(W_o h_f + b_o)$$
$$p_i = \frac{\exp\left(o_f{}^T e_i{}^a\right)}{\sum_j \exp\left(o_f{}^T e_j{}^a\right)} \tag{8}$$

其中, W_o 是一个全连接网络, 将方法状态 h_f 转化为与 API 词向量维度相同的向量 $o_f{}^T$. $o_f{}^T e_i{}^a$ 表示两个向量的点积. p_i 表示 API 候选集中第 i 个 API 作为补全 API 的概率. 这种方法仅仅需要训练一个输出维度等同于词向量维度的全连接网络, 同时在计算 Softmax 函数时, 只需要计算 API 候选集中每个 API 作为补全推荐的概率. 这在减少内存使用空间的同时, 大大提升了模型预测速度, 让模型作为插件的一部分, 在用户本机运行成为可能. 接着利用模型计算得到的每个 API 对应的补全概率, 按照大小排序, 推荐概率值较大的前几位 API 给用户, 用于补全空缺位置的 API 调用.

5 实　验

本文设计和执行了全面的对比实验, 并通过回答如下研究问题以验证本文提出的基于对象类型的 API 补全技术有效性.

问题 1: 基于对象类型的 API 补全技术与当前先进的补全方法相比准确率以及性能如何?

问题 2: 基于对象类型的 API 补全技术在不同补全位置下性能如何?

问题 3: 基于对象类型的 API 补全技术在不同对象个数下性能如何?

问题 4: 基于对象类型的 API 补全技术各组件是否均对补全产生帮助?

5.1 实验设计

数据集: 本实验采用的训练集是使用爬虫程序从 Github 上按照 star 数爬取的前 15 000 个 Java 项目. 在分析数据时, 我们发现其中有一部分项目属于教程, 即项目中是对某些技术的介绍, 其中并不包含 Java 代码. 在过滤掉这一部分项目后, 最终我们得到了 14 785 个 Java 项目. 通过对项目中源文件抽取出 API 调用序列, 再过滤掉序列长度小于 2 的数据, 最终我们获得了 1 130 203 条 API 调用序列. 测试集使用 Galaxy、Log4j、JGit、Itext、FroyoEmail 和 Grid-Sphere. 这 6 个项目在之前的工作中也被用来当作测试集来使用[36−38]. 在对模型进行性能评估时, 分别对这 6 个项目进行测试, 以保证训练集和测试集之间的独立性. 具体的代码库描述如表 1 所示. 测试集不同项目 API 序列长度统计如图 8.

表 1　数据集描述

训练集	项目个数	代码行数	API序列个数	API序列平均长度	类个数	API个数
值	14 785	352 312 696	1 130 203	4	5 084	35 380
测试集	Galaxy	Log4j	JGit	Itext	FroyoEmail	Grid-Sphere
API序列个数	209	829	2 256	1 546	566	683

评估标准: 本文使用了两种验证维度来验证模型效果: Top-K ACC 和 MRR (mean reciprocal rank)[37]. Top-K 和 MRR 是评价推荐算法的两种常用指标. Top-K ACC 的计算方式如公式 (8) 所示:

$$Top\text{-}K\ ACC = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\delta(rank_i \leqslant k) \tag{9}$$

其中, $rank_i$ 表示真实标签在推荐结果中的次序. δ 是一个指示函数, 如公式 (9) 所示:

$$\delta(x) = \begin{cases} 1 \text{ if } x \text{ is true} \\ 0 \text{ else} \end{cases} \tag{10}$$

它表示如果真实标签在推荐结果中的次序小于 k, 就标记为一个命中. 统计测试集中的命中次数, 再除以测试集的大小 $|Q|$, 得到真实标签在前 k 个推荐结果中就可以找到的概率. Top-K ACC 的值越高, 表示该推荐算法推荐

效果越好. MRR 是一种更加综合的评价指标, 它的计算方式如公式 (10) 所示:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{11}$$

它表示如果真实标签在推荐列表中的第 1 个匹配, 分数为 1, 第 2 个匹配的分数为 0.5, 第 n 个匹配的分数为 $1/n$. 计算整个测试集中的匹配得分之和, 再除以测试集大小, 得到真实概率在整个测试集中的平均得分情况. 它同样也是越高越好.

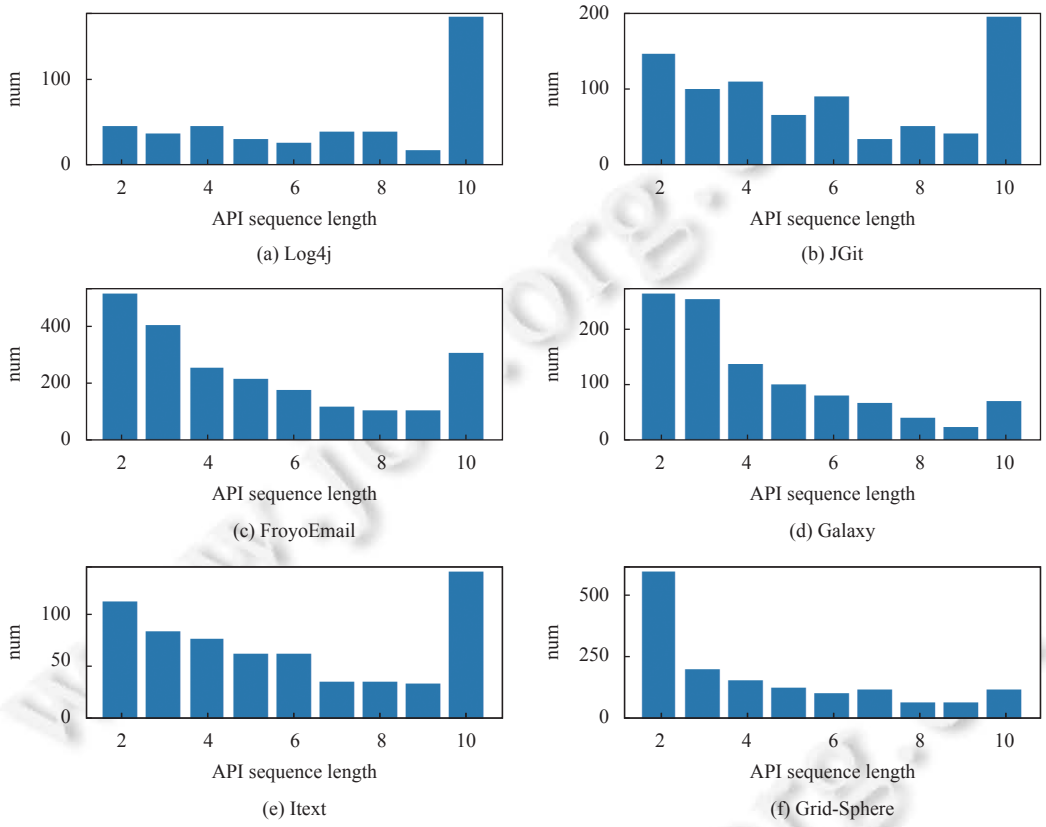

图 8 测试集不同项目 API 序列长度统计图

基线模型:本文使用了 4 个基准模型来与本文提出的模型进行对比实验, 包括: N-gram [17], HAPI [16], LSTM [39] 和 APIHelper [21].

N-gram 模型使用了两个模型的联合概率来对 API 进行补全. 全局模型 (global model) 是使用外部代码库训练的 N-gram 语法模型, 本地模型 (local model) 是基于该项目下所有源代码文件训练得到的语法模型. 在需要对一处 API 空缺进行补全时, 利用空缺位置周围 N–1 个 API 调用组成的调用序列分别输入本地模型和全局模型中计算概率. 根据本地模型和全局模型的联合概率分布进行 API 补全推荐. 在实验中, 本文将 N 设置为 5. 全局模型由训练集构建, 本地模型是在对具体项目进行测试时, 使用对应的项目进行构建.

HAPI 模型是使用一个隐马尔可夫模型来进行 API 补全推荐. 其主要思想是将对象状态当作隐藏状态, 将 API 调用当作观测序列. 使用一个隐马尔可夫模型来刻画 API 调用过程. 因为它本身提出是为了分析 Android 源代码中的 API 调用. 本文重新实现了这个模型, 并为训练集中的 5 084 个对象类型每个训练了一个隐马尔可夫模型. 同时为训练集中在方法块中共现超过 200 次的对象类型额外训练了它们对应的多对象隐马尔可夫模型. 在验证模型性能时, 本文没有分开验证单对象隐马尔可夫模型和多对象隐马尔可夫模型的性能. 而是先寻找是否有包

含 API 序列中所有对象类型的模型, 如果有, 直接调用. 如果没有, 寻找包含序列中对象类型集合的最大子集的模型. 在预测时, 优先使用包含对象类型最多的模型. 如果存在多个包含一样多对象类型的模型, 我们使用这些模型的预测概率计算其联合概率, 使用联合概率推荐补全.

LSTM 是 Dam 在 2016 年提出的代码补全模型. 因为原文是将源代码直接作为文本序列输入到 LSTM 中, 为了对比实验的数据源统一性, 本文重新实现了该模型并将输入改为了 API 调用序列.

APIHelper 是 Yan 在 2018 年提出的 API 补全模型. 他们在标准 LSTM 模型基础上, 使用了拼接编码以及确定的负采样算法. 拼接编码与本文类似, 是将对象类型和 API 调用分开编码后拼接成最终的向量表示. 确定的负采样算法是让模型在训练时不需要对全体 API 计算推荐概率, 而是对采样出的负样本和真实标签计算概率, 进而提升模型的收敛速度. 确定的意思是指不同于标准的随机采样算法, 负样本选择的都是属于空缺位置对象类型的 API, 这样采样出的负样本都是属于该对象类型的 API 调用. 预测时只需要使用类似的采样算法, 选取出空缺位置对象类型的所有 API 进行预测.

实验环境: 本文中的实验使用了一台显卡型号为 NVIDIA GeForce 2080Ti 的 centos 服务器. 其中 API 序列最大长度限定为 10. 词向量和隐藏状态维度均设为 128. 训练使用了交叉熵作为损失函数, 学习率设为 1E–3, batch size 设置为 32.

5.2 实验结果与分析

5.2.1 针对问题 1 的结果分析

图 9 对这 6 个测试项目上的 Top-K 准确率做了进一步的分析. 从图中可以更加明显的看到, 本文提出的模型在除 Grid-Sphere 项目外的 Top-K 值都处在较高位置, 并且在 6 个项目中的 Top-1 准确率均是最高. 在 JGit、Log4j 和 Itext 这 3 个测试项目中, 本文提出的模型从 Top-1 到 Top-10 准确率均超过基线模型. 而在 Grid-Sphere 项目中, 由于其数据分布的独特性, 即长度为 2 的 API 调用序列个数占比超过 50% (图 8), 因此在该项目中, HAPI 模型表现的效果最好. 但是, 本文提出的模型效果依旧超过了 APIHelper, 仅次于 HAPI 模型, 排在第 2 位. 总体来看, LSTM 和 Nested-Cache N-gram 模型在测试集上的表现不是很好, 处于较低的水平. HAPI 模型在短调用序列补全中效果非常出色, 但是并不擅长较长的调用序列补全. APIHelper 在长序列补全上可以获得比 HAPI 模型更高的准确率, 但在短序列补全中表现的并不如 HAPI 模型. 本文提出的模型在所有测试项目中都比 APIHelper 模型表现的更好, 在较短的 API 序列补全上也取得了与 HAPI 相近的效果, 并且在 Top-1 准确率上要优于所有基线模型.

表 2 展示了在这 6 个测试项目中不同模型的 MRR 比较. 从表中可以看出, 本文提出的模型在除 Grid-Sphere 外其余 5 个项目上的 MRR 值均是最高. 而在 Grid-Sphere 项目中 HAPI 模型的 MRR 是 0.655, 比本文提出的模型高出了 0.005. 但是 HAPI 模型在不同项目上的 MRR 值并不稳定, 在 Grid-Sphere、Froyo-Email、Galaxy 这 3 个项目中, HAPI 的 MRR 值均较高. 而在 JGit、Log4j 和 Itext 这 3 个项目中, HAPI 的 MRR 值比较低, 甚至会比 LSTM 模型的准确率还要低. 因此, 本文对这 6 个项目中 API 序列长度进行了统计. 其中, Grid-Sphere、Froyo-Email 和 Galaxy 这 3 个项目中的 API 调用序列长度大多分布在 2–4 之间, 且长度为 2 的调用序列个数均最多. 而剩余的 3 个项目 Log4j、JGit 以及 Itext 这 3 个项目的 API 调用序列均偏长, 最高值都在长度等于 10 的地方出现. 因此可以看出, HAPI 模型对长度在 2–4 之间的 API 调用序列有着较好的补全准确率, 但是如果 API 调用序列长度偏长, HAPI 模型的推荐效果将会受到很大影响. APIHelper 模型由于使用了 LSTM 长短期记忆模型, 不会过多受到长度问题的影响, 因此在 JGit、Log4j 和 Itext 这 3 个平均 API 调用序列长度较长的项目中, 均优于 HAPI 模型. 但是在 API 序列较短的项目中, 模型效果就不能得到保证, 有时会低于 HAPI 模型.

另外, 本文分析了模型在预测时的时间效率和空间占用率. 考虑到很多用户私人笔记本中显卡配置并不高, 因此在进行预测对比实验时, 所有使用深度学习的模型均使用内存加载进行预测. 全部模型的预测时间和内存占用如表 3 所示. 从预测时间来看, HAPI 模型的预测时间最长. 因为在预测时, HAPI 模型需要根据待补全位置的对象类型生产候选的 API 集合. 然后将集合中的 API 替换到空缺位置, 生成所有可能的补全结果. 最后将这些补全后的序列输入到隐马尔可夫模型中计算序列的联合概率, 按照概率大小推荐补全的 API. 因为需要多次调用模型计

算序列的联合概率, 它的预测时间会明显长于其他深度学习模型的预测时间. LSTM 和 Nested-Cache N-gram 模型的预测时间相似, 因为它们在预测时, 都需要对一个和整体 API 大小同维度的向量进行 Softmax 运算. 因为当输入向量维度较高时, Softmax 函数的计算会变得非常耗费时间. 因此它们的预测时间也明显偏高. APIHelper 模型由于使用了确定的采样算法, 在预测时只需要对采样出的 API 计算概率, 因此预测时间会明显降低. 本文提出的模型, 也是使用候选集机制来避免高维度的 Softmax 运算, 同时在对象层, 由于对象状态的计算不需要依赖其他对象的 API 调用, 可以在抽取出相同对象类型的 API 调用序列后并行地输入模型进行计算, 因此可以在最短时间给出模型预测结果.

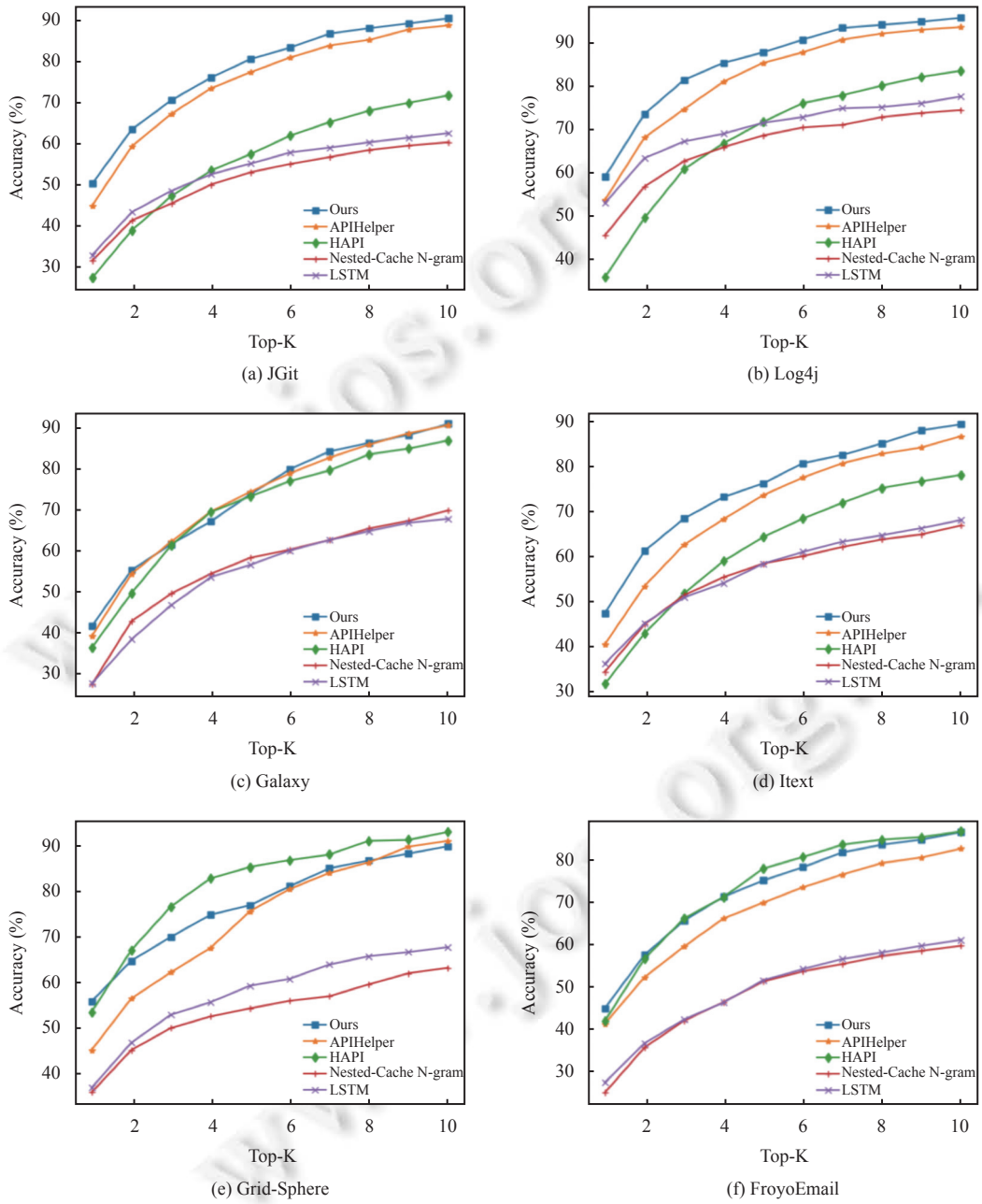

图 9 不同项目中 Top-K Accuracy 变化曲线

表 2 不同模型在 6 个测试项目上的 MRR 对比

模型	JGit	Log4j	Itext	Grid-Sphere	Galaxy	Froyo-Email	All
N-gram	0.414	0.556	0.454	0.455	0.415	0.37	0.422
LSTM	0.432	0.614	0.465	0.474	0.405	0.387	0.438
HAPI	0.403	0.503	0.45	**0.665**	0.514	0.564	0.544
APIHelper	0.594	0.667	0.551	0.583	0.549	0.546	0.57
Ours	**0.635**	**0.714**	**0.609**	0.66	**0.561**	**0.586**	**0.616**

表 3 不同模型的性能对比

模型	内存占用 (MB)	预测时间 (ms)
N-gram	182	21
LSTM	48.5	73
HAPI	328	20
APIHelper	356	13.5
Ours	**248**	**13.2**

5.2.2 针对问题 2 的结果分析

本节主要分析补全位置对模型预测准确率的影响. 通过将测试集按照 API 调用序列空缺位置对数据进行划分. 空缺位置出现在序列前 1/5 位置的, 标记为第 1 类; 出现在 1/5~2/5 位置的标记为第 2 类. 以此类推, 将整个测试集划分为 5 部分. 图 10 展示了模型对不同补全位置的推荐准确率.

从图 10 中可以看出, 本文提出的模型对不同空缺位置的 API 补全效果没有明显的起伏性变化, 都维持在较高的水平. 而其他模型都会在一定程度上受到空缺位置的影响. 比如在对 Log4j 项目测试 MRR 值时, APIHelper 模型对于 0~1/5 位置推荐补全的 MRR 值仅有 0.591, 而在 3/5~4/5 位置的推荐补全率可以达到 0.772. HAPI 模型对 0~1/5 的 MRR 值为 0.587, 而在 4/5~1 位置的推荐补全率仅有 0.429. Nested-Cache N-Gram 和 LSTM 模型的预测效果对补全位置更加敏感. 比如 Nested-Cache N-Gram 对 3/5~4/5 和 4/5~1 的预测补全率分别在 0.66 和 0.496, LSTM 在对应位置的预测准确率分别在 0.69 和 0.52. 这种对空缺位置敏感的推荐模型会影响开发人员的实际使用体验, 而本文提出的模型对不同空缺位置的预测准确率均保持在较高的位置, 并且波动不超过 0.1, 这说明本文提出的模型可以在实际使用中可以更好地支持 API 补全工作, 不会在某些情况下, 推荐效果出现很大幅度的下滑. 其背后的原因主要有两个. 一是在本文提出的模型在方法层使用了前向和逆向的 LSTM 模型分别对空缺位置前后的对象状态序列进行编码. 这就使得模型可以从结构上获取空缺位置信息, 而不需要通过学习空缺位置编码的方式来获取待补全位置. 另一方面是因为本文提出的模型在对象层就将 API 调用序列按照空缺位置划分为两部分, 对两部分分别进行处理, 这样可以更加有效地区分空缺位置前后的 API 调用序列, 使得模型不会过多受到空缺位置的影响.

总体来看, 本文提出的模型预测性能不会过多受到补全位置的影响, 一直保持着较高的水平. 而基线模型的预测效果会随着补全位置的不同而产生波动. 因为模型最终的应用场景是在开发环境中提供持续的 API 补全服务, 也就是说空缺位置不断发生变化. 从该试验可以看出, 本文提出的模型可以在不同位置下一直提供较高质量的补全性能, 相较于基线模型可以为开发人员提供更好的补全体验.

5.2.3 针对问题 3 的结果分析

本小节讨论了 API 序列中对象个数对模型性能的影响. 图 11 给出了基于对象类型的补全模型和其余 4 个基线模型在测试集上的数据对比. 从图 11 中可以看出, 对于除 Nested-Cache N-gram 模型以外的模型, 对象类型个数与模型预测性能之间并没有明显的线形关系. 这主要是因为, 虽然 API 序列中对象类型的个数增多, 但是会对空缺位置 API 调用产生影响的对象个数并不会线性增长. 因此模型的预测准确率高点大部分出现在对象类型个数在 2~4 之间. 这说明在大部分情况下, 会影响补全位置 API 调用的对象类型个数在 2~4 个之间, 少于 2 个时由于提供的补全信息不足让模型无法准确给出补全判断, 多于 4 个时由于无关对象的 API 调用增多, 同样会干扰模型预测. 但是 Nested-Cache N-Gram 模型受到对象类型个数的影响比较大. 比如在对 JGit 项目测试时, Nested-Cache N-Gram 模型在对象个数为 1 时的准确率可以达到 0.548, 超过了其余模型, 但是在对象类型个数在 5 个以上时, 预测

准确率下滑到了 0.311. 这是因为 Nested-Cache N-Gram 使用 N-gram 模型作为基础模型, 只会考虑空缺位置周围的 N–1 个 API 调用, 其余的 API 调用会被忽略. 随着 API 调用序列中对象类型个数的增多, 其中起作用的 API 调用有着更高的可能性不在补全位置的周围. 这使得 Nested-Cache N-Gram 模型在处理对象类型较多的 API 调用序列时, 非常容易将某些重要的 API 调用给抛弃掉. APIHelper 和本文提出的模型由于使用了 LSTM 作为基础模型, 可以在一定程度上忽略无关 API 调用的影响, 因此这两个模型受到对象类型个数的影响并不大. 而 HAPI 模型由于在训练过程中, 只会训练单个对象和共现次数较多的多个对象对应的隐马尔可夫模型, 因此它其实是利用统计的方法, 事先从训练集提取出关联比较紧密的多个对象. 在进行预测时, 会忽略掉与待补全位置对象类型共现次数不多的 API 调用, 从而一定程度上可以减少无关的 API 调用对模型补全的影响.

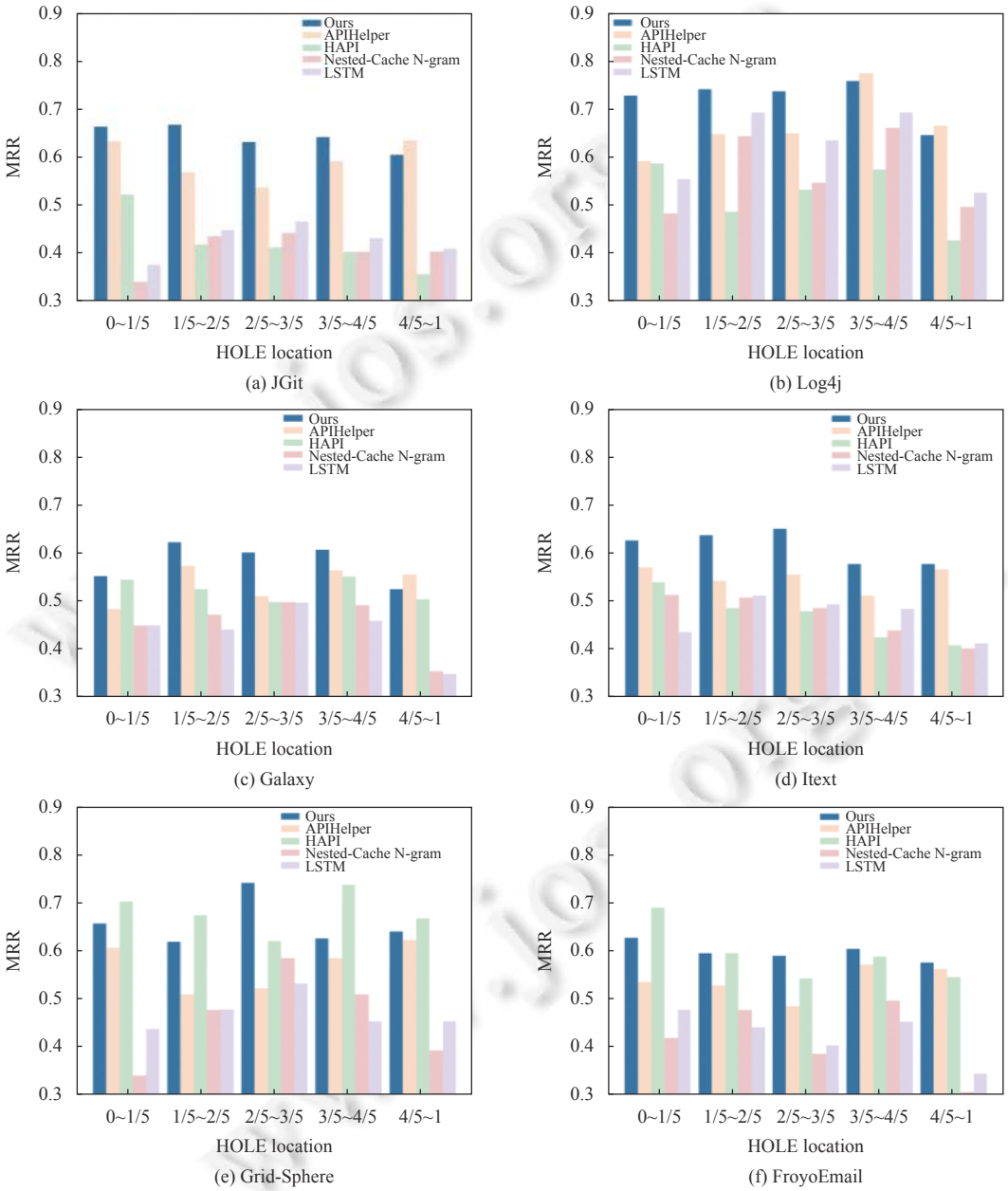

图 10 待补全位置对不同模型的性能影响

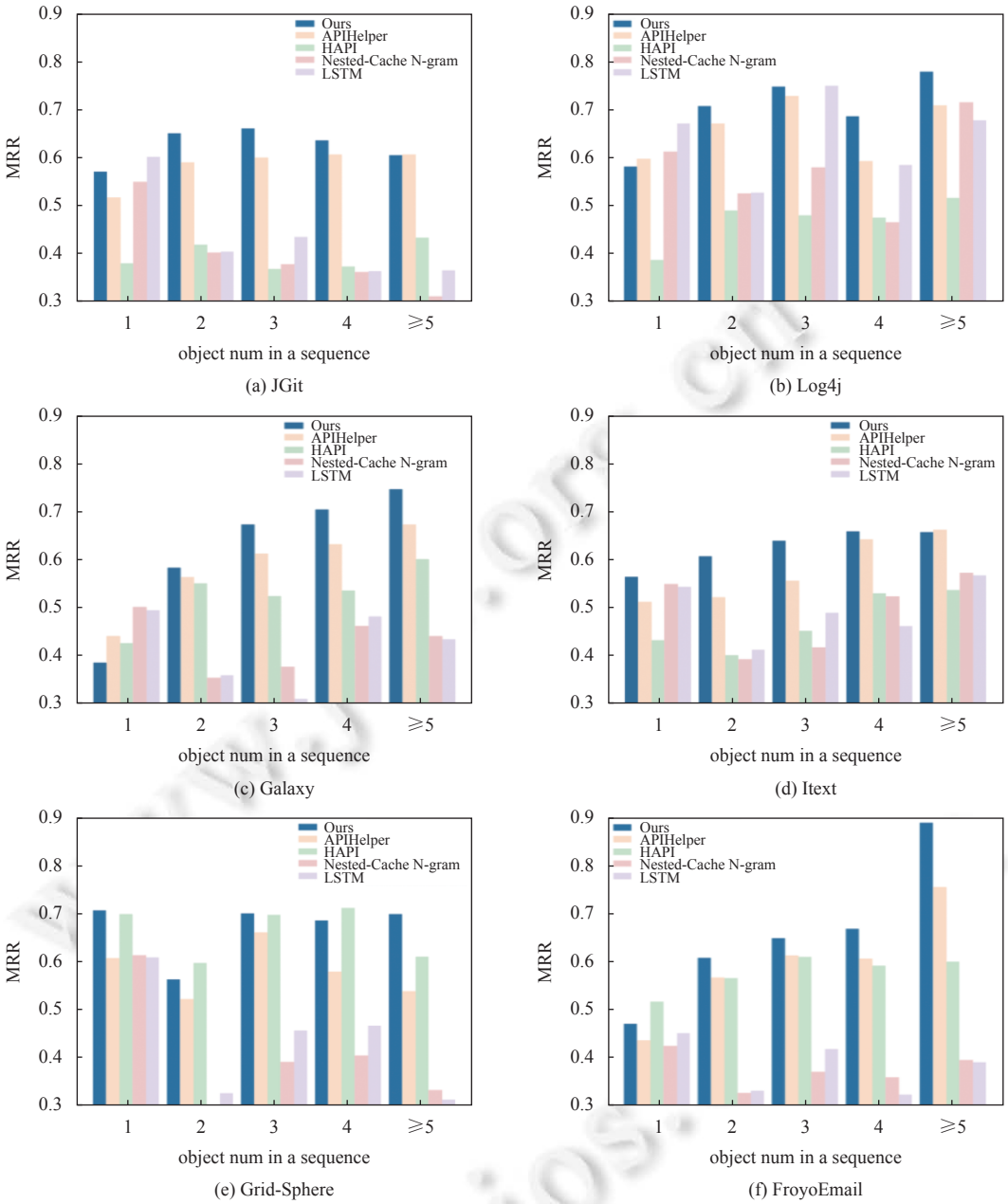

图 11 API 序列中对象个数对模型性能的影响

总体来看, 由于不是所有出现的对象都会影响空缺位置 API 调用, 因此 API 序列中对象个数并不会对除 Nested-Cache N-Gram 以外的模型造成影响. 因为其他模型都使用了各自的策略来减少无关对象的 API 调用对补全效果的影响. 其中, HAPI 使用了统计的方法来过滤掉无关对象的调用, 本文提出的模型、APIHelper 和 LSTM 由于都是用了长短期记忆模型来“遗忘”掉无关调用. 由于 Nested-Cache N-Gram 仅仅会考虑空缺位置周边的 N–1 个词, 因此有着很大概率将重要的 API 调用给过滤掉. 因此 Nested-Cache N-Gram 模型的补全性能随着对象类型个数的增多呈现下降趋势.

5.2.4 针对问题 4 的结果分析

本小节介绍本文提出的基于对象类型的 API 补全模型各个组件的作用. 表 4 展示了实验的结果.

表 4 模型拆解实验

拆解模型	JGit	Log4j	Itext	Grid-Sphere	Galaxy	Froyo-Email
-拼接编码	0.55	0.62	0.53	0.52	0.48	0.54
-对象层	0.59	0.66	0.55	0.58	0.54	0.56
-方法层	0.57	0.65	0.57	0.59	0.52	0.57
-候选集策略	0.43	0.61	0.46	0.47	0.40	0.42
All	**0.64**	**0.71**	**0.61**	**0.66**	**0.56**	**0.59**

表 4 中“-”表示在本文提出的模型基础上, 不使用这一技术.“-拼接编码”表示在嵌入层不使用对象类型和 API 分别编码, 再拼接成向量表示的策略, 直接使用 API 进行编码. “-对象层”表示直接将 API 调用序列输入到方法层, 使用两个 LSTM 模型分别对空缺位置之前的 API 调用序列和空缺位置之后的 API 调用序列进行编码. “-方法层”表示将对象层生成的对象状态, 以求均值的方式直接表示为方法状态, 输入到预测层中. “-候选集策略”表示不使用候选集策略确定 API 候选集, 直接使用一个全连接网络来对补全位置进行预测. 从这六个测试集的实验中可以看出, 无论减少了哪一部分组件, 模型的性能都会有所下降, 这说明本文提出的 API 补全模型各个组件都是有效的. 其中, 不使用候选集策略下降的最为明显. 因为如果不使用候选集策略, 模型需要在预测层对所有的 API 进行预测, 这将使得模型在训练过程中很难得到拟合. 不使用拼接编码策略模型性能下降的也比较多, 这是因为使用拼接编码后, 模型可以通过 API 编码的前一半判断 API 是否属于同一个对象, 这可以让模型更快速的学习到该对象相关的特征. 不使用对象层也会造成性能下降, 因为 API 调用的部分有序性, 导致在不使用对象层后, API 输入的顺序可能是混乱的, 这会增加模型学习的难度. 不使用方法层同样会造成性能下降. 这是因为当不使用方法层时, 模型只能通过学习 hole 的词向量来获取空缺位置信息. 而在使用方法层后, 由于方法层是将空缺位置两侧的对象状态信息向空缺位置汇总, 这可让模型更加直接地了解预测位置, 同时也会让模型更加关注于紧邻空缺位置两侧的 API 调用.

因此可以得出结论, 本文提出的模型各个组件, 包括 API 拼接编码层、对象状态编码层、方法状态编码层以及使用候选集的预测层均可以提升模型的补全效果, 缺失了任何一部分都会造成模型补全性能的下降.

5.3 局限与不足

本文提出的方法是使用对象类型信息来帮助 API 补全, 但对于一些没有对象概念的编程语言, 比如 C 语言, 或是操作系统内核相关 API, 并不能直接使用本文提出的方法. 一种可行的解决方案是将所有 API 当作属于同一个对象类型的 API, 这样等价于直接使用一个双向 LSTM 模型来进行补全. 另一种方案是将 API 所在文件的名称作为对象类型, 比如 C 语言中的头文件信息. 但这种方案会存在一定的缺陷, 因为同一个文件下的 API 可能并没有关系, 也不能构成对象的概念. 比如 C 语言中 math.h 下的 API 只是实现数学功能的 API 的集合, 将这个文件作为对象是不合理的. 当然, 如果 API 所在文件具有对象的部分特征, 比如文件流 fstream 或者输入输出流 iostream, 将 API 所在文件看作对象类型也具有一定的合理性.

除此之外, 由于本方法需要获取变量的对象类型信息, 并从待补全代码中提取 API 调用序列, 这使得本方法需要对代码进行静态分析, 这会导致一定的时间和空间开销, 速度上可能不如一些将代码作为纯文本进行分析补全的方法, 但本文的方法可以保证推荐补全的 API 不会有语法错误, 这是将代码作为纯文本分析的方法做不到的.

5.4 定性分析

本小节分析了两个真实的补全场景, 如表 5 所示. 其中示例 1 和示例 2 分别对应补全时有干扰和无干扰场景. 干扰是指当代码上下文中是否存在无关 API 调用. 其中展示了 3 种不同的自动补全方法, 包括 IDEA 自带的补全插件、APIHelper 补全插件以及基于对象类型的补全插件. 由于是对 BufferedReader 对象进行补全, 有干扰情况设

置为代码中存在许多与 BufferedReader 无关的 API 调用 (示例 2 中的 while 循环里增加了 String 和 List 对象的 API 调用). 由于 IDEA 自带的补全插件仅仅是按照 API 的使用频次进行推荐, 并不会对代码上下文进行分析, 它的补全效果是最差的. APIHelper 插件在没有无关 API 调用时, 正确推荐了补全的 API. 但是当存在无关 API 调用时, 补全效果下降明显. 而本文提出的基于对象类型的 API 补全插件在存在干扰 API 调用的情况下, 分析出 ready 和 readLine 方法几乎不可能被调用, 因此推荐补全 close 的概率不降反升, 以 98.36% 的补全可能性推荐补全 close 方法. 这说明基于对象类型的 API 补全方法在补全时有着很强的抗干扰能力.

表 5 API 补全实例

示例1		

待补全代码	public void read_file(String filename) throws Exception{
	FileInputStream fileInputStream = new FileInputStream(filename);
	BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(fileInputStream));
	String line = null;
	while((line = bufferedReader.readLine()) != null){
	}
	bufferedReader.?
	}
	}	

正确补全	bufferedReader.close()	
IDEA (Top-3)	bufferedReader.readLine() bufferedReader.read() bufferedReader.read(char[] cbuf)	
APIHelper (Top-3)	bufferedReader.close() bufferedReader.ready() bufferedReader.readLine()	61.99% 0.33% 6.08%
Ours (Top-3)	bufferedReader.close() bufferedReader.ready() bufferedReader.readLine()	79.73% 10.36% 8.36%
示例2		

待补全代码	public void read_file(String filename) throws Exception{
	FileInputStream fileInputStream = new FileInputStream(filename);
	BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(fileInputStream));
	String line = null;
	List<String> lables = new ArrayList<>();
	List<String> inputs = new ArrayList<>();
	while((line = bufferedReader.readLine()) != null){
	String[] items = line.split(";");
	inputs.add(items[0]);
	labels.add(items[1]);
	}
	bufferedReader.?
	}
	}	

正确补全	bufferedReader.close()	
IDEA (Top-3)	bufferedReader.readLine() bufferedReader.read() bufferedReader.read(char[] cbuf)	
APIHelper (Top-3)	bufferedReader.readLine() bufferedReader.ready() bufferedReader.readLine()	28.56% 26.21% 24.48%
Ours (Top-3)	bufferedReader.close() bufferedReader.ready() bufferedReader.readLine()	98.36% 1.06% 0.48%

5.5 API 补全插件

本文实现了一个基于 IDEA 的自动补全插件, 支持离线和在线补全. 整体框架采用了使用 B/S 结构, 并使用分层结构对补全系统进行了相应设计. 依次划分为客户端应用层、服务器控制和计算层以及服务器数据存储层. 如图 12 所示.

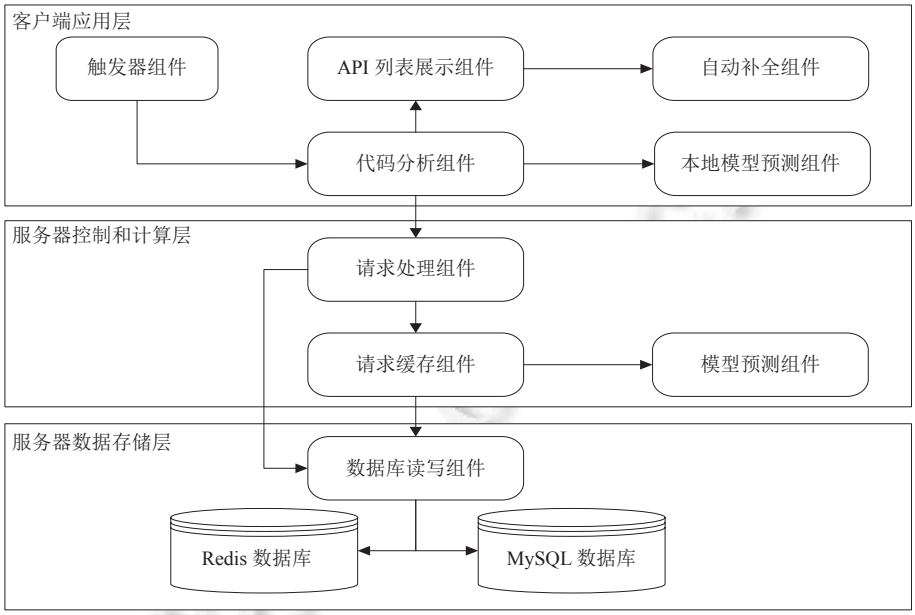

图 12 API 补全系统框架图

客户端应用层是基于 IDEA 插件开发. 主要实现了 API 补全系统与用户交互的功能, 包括触发器组件, API 列表展示组件以及自动补全组件. 同时, 系统将代码分析组件放在了客户端本地进行分析, 这主要是出于对用户隐私和服务器性能的考虑. 客户端应用层还提供了一个本地模型预测组件, 用于让用户本地加载模型, 离线实现补全功能.

服务器控制和计算层主要是包括两部分功能: 一是为在线 API 补全提供服务支持, 二是记录用户使用补全服务时传入的真实待补全 API 序列以及最终选择的补全 API. 为了支持多人同时使用在线补全服务, 本文在该层设计了一个请求缓存组件, 来将用户请求缓存入优先队列中, 然后按批次取出一起放入模型进行预测. 另外, 在用户完成了一次补全服务后, 会调用请求处理组件存储本次服务对应的空缺 API 序列和最终补全结果, 用于之后进一步的模型分析和优化.

服务器数据存储层主要实现缓存请求数据和存储真实补全样例的功能. 存储层使用了 Mysql 数据库来存储真实补全样例, 使用 Redis 数据库来缓存用户请求. 图 13 展示了 API 补全插件的配置图. 用户可以选择使用离线模式和在线模式调用 API 补全服务. 使用在线模式时, 还需要配置服务器地址.

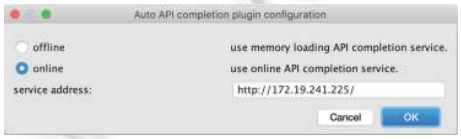

图 13 API 补全插件配置图

在正确的配置了 API 补全服务后, 用户可以使用插件来调用 API 补全服务. 图 14 展示了插件补全效果图. 图中代码实现的是一个读文件功能. 推荐补全的 API 及其补全可能性在推荐列表中按可能性大小从高到低依次展示. 原 IDEA 提供的 API 补全列表紧跟其后, 这主要是为了解决用户需要补全的对象类型不在 JDK 包中的情况.

当待补全位置的对象类型不属于 JDK 包, API 补全插件将无法提供相应的补全服务, 这时插件会使用 IDEA 自带的补全功能将可供选择的 API 展示给用户.

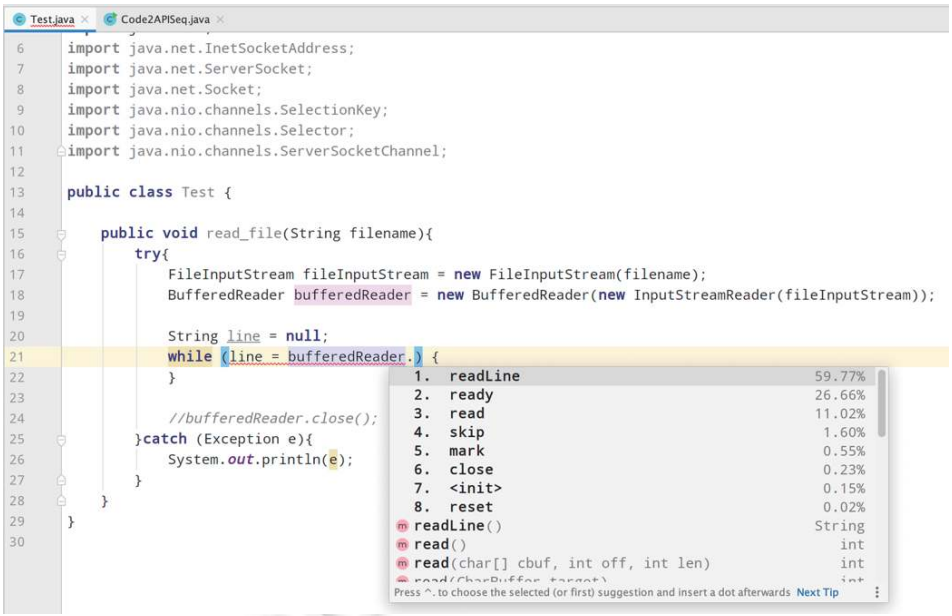

图 14 插件补全效果展示图

6 总结与展望

本文针对现有 API 补全方案中忽略 API 所属对象类型的问题, 探究了对象类型对 API 补全效果的影响, 并创新性地设计了一种使用待补全代码段中对象类型作为额外特征的 API 补全方法, 该方法的核心是一个包含 4 个层级的深度学习模型: 输入层、编码层、层级网络层和预测层. 具体而言, 在输入层先从待补全代码段的 API 调用序列中抽取出属于同一对象类型的 API 调用子序列; 在编码层先分别为对象类型和 API 编码, 然后拼接; 层级网络层的核心是两层 LSTM 网络, 第 1 层是一系列 LSTM 模型, 每一种对象类型对应一个 LSTM, 其输入是编码层拼接好的向量表示, 第 1 层各 LSTM 的隐藏状态作为对象类型的向量表示输入第 2 层 LSTM, 其最后的输出作为待补全代码块的向量表示; 最后预测层结合代码块向量表示和候选 API 的向量表示计算每一个候选 API 的作为正确结果的概率. 实验结果表明, 本文提出的方法在补全准确率以及效率上均优于基线模型.

本文提出的基于对象类型的 API 补全方法在 Java 编程语言上取得了一定的效果, 未来我们将继续探究其在其他类型的编程语言上的 API 推荐效果, 如 Python、C++等. 此外, 如何在 API 推荐中使用代码段中更多信息作为额外特征也是本文未来重要的研究方向.

References:

[1] Nguyen TD, Nguyen AT, Phan HD, Nguyen TN. Exploring API embedding for API usages and applications. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 438–449. [doi: 10.1109/ICSE.2017.47]

[2] Zhou Y, Gu RH, Chen TL, Huang ZQ, Panichella S, Gall H. Analyzing APIs documentation and code to detect directive defects. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 27–37. [doi: 10.1109/ICSE.2017.11]

[3] Robillard MP. What makes APIs hard to learn? Answers from developers. IEEE Software, 2009, 26(6): 27–34. [doi: 10.1109/MS.2009.193]

[4] Piccioni M, Furia CA, Meyer B. An empirical study of API usability. In: Proc. of the ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Baltimore: IEEE, 2013. 5–14. [doi: 10.1109/ESEM.2013.14]

[5] 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. Amsterdam: ACM, 2009. 213–222. [doi: 10.1145/1595696.1595728]

[6] Hill R, Rideout J. Automatic method completion. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering. Linz: IEEE, 2004. 228–235. [doi: 10.1109/ASE.2004.1342740]

[7] Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 2009, 74(7): 470–495. [doi: 10.1016/j.scico.2009.02.007]

[8] Rieger M, Ducasse S, Lanza M. Insights into system-wide code duplication. In: Proc. of the 11th Working Conf. on Reverse Engineering. Delft: IEEE, 2004. 100–109. [doi: 10.1109/WCRE.2004.25]

[9] Proksch S, Lerch J, Mezini M. Intelligent code completion with Bayesian networks. ACM Trans. on Software Engineering and Methodology, 2015, 25(1): 3. [doi: 10.1145/2744200]

[10] Heinemann L, Hummel B. Recommending API methods based on identifier contexts. In: Proc. of the 3rd Int'l Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation. Waikiki: ACM, 2011. 1–4. [doi: 10.1145/1985429.1985430]

[11] Nguyen AT, Nguyen HA, Nguyen TT, Nguyen TN. GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 1407–1410. [doi: 10.1109/ICSE.2012.6227236]

[12] Akbar RJ, Omori T, Maruyama K. Mining API usage patterns by applying method categorization to improve code completion. IEICE Trans. on Information and Systems, 2014, E97-D(5): 1069–1083. [doi: 10.1587/transinf.E97.D.1069]

[13] Asaduzzaman M, Roy CK, Schneider KA, Hou DQ. CSCC: Simple, efficient, context sensitive code completion. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. Victoria: IEEE, 2014. 71–80. [doi: 10.1109/ICSME.2014.29]

[14] de Souza Amorim LE, 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. Amsterdam: ACM, 2016. 163–175. [doi: 10.1145/2997364.2997374]

[15] Hu S, Xiao C, Ishikawa Y. Scope-aware code completion with discriminative modeling. Journal of Information Processing, 2019, 27: 469–478. [doi: 10.2197/ipsjjip.27.469]

[16] Nguyen TT, Pham HV, Vu PM, Nguyen TT. Recommending API usages for mobile apps with hidden Markov model. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 795–800. [doi: 10.1109/ASE.2015.109]

[17] 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. Paderborn: IEEE, 2017. 763–773. [doi: 10.1145/3106237.3106290]

[18] Gvero T, Kuncak V, Kuraj I, Piskac R. InSynth: A system for code completion using types and weights. In: Proc. of the Software Engineering & Management, Dresden, 2015. 39–40.

[19] Roos P. Fast and precise statistical code completion. In: Proc. of the IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 757–759. [doi: 10.1109/ICSE.2015.240]

[20] Savchenko V, Volkov A. Statistical approach to increase source code completion accuracy. In: Proc. of the Ershov Informatics Conf. Moscow: Springer, 2018. 352–363. [doi: 10.1007/978-3-319-74313-4_25]

[21] Yan JP, Qi Y, Rao QF, He H. Learning API suggestion via single LSTM network with deterministic negative sampling. In: Proc. of the 30th Int'l Conf. on Software Engineering and Knowledge Engineering. San Francisco, 2018. [doi: 10.18293/SEKE2018-193]

[22] Svyatkovskiy A, Zhao Y, Fu SY, Sundaresan N. Pythia: AI-assisted code completion system. In: Proc. of the 25th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining. Anchorage: ACM, 2019. 2727–2735. [doi: 10.1145/3292500.3330699]

[23] Nguyen S, Nguyen T, Li Y, Wang SH. Combining program analysis and statistical language model for code statement completion. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 710–721. [doi: 10.1109/ASE.2019.00072]

[24] Chen C, Peng X, Sun J, Xing ZC, Wang X, Zhao YF, Zhang HR, Zhao WY. Generative API usage code recommendation with parameter concretization. Science China Information Sciences, 2019, 62(9): 192103. [doi: 10.1007/s11432-018-9821-9]

[25] Hellendoorn VJ, Proksch S, Gall HC, Bacchelli A. When code completion fails: A case study on real-world completions. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 960–970. [doi: 10.1109/ICSE.2019.00101]

[26] Yang YX, Chen X, Sun JG. Improve language modeling for code completion through learning general token repetition of source code with optimized memory. Int'l Journal of Software Engineering and Knowledge Engineering, 2019, 29(11–12): 1801–1818. [doi: 10.1142/S0218194019400229]

[27] Li J, Wang Y, Lyu MR, King I. Code completion with neural attention and pointer networks. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. Stockholm: AAAI Press, 2018. 4159–4165.

[28] Terada K, Watanobe Y. Code completion for programming education based on recurrent neural network. In: Proc. of the IEEE 11th Int'l

Workshop on Computational Intelligence and Applications (IWCIA). Hiroshima: IEEE, 2019. 109–114. [doi: 10.1109/IWCIA47330.2019.8955090]

[29] Yang YX. Improving the robustness to data inconsistency between training and testing for code completion by hierarchical language model. arXiv: 2003.08080, 2020.

[30] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735–1780. [doi: 10.1162/neco.1997.9.8.1735]

[31] Chorowski JK, Bahdanau D, Serdyuk D, Cho K, Bengio Y. Attention-based models for speech recognition. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 577–585.

[32] Vinyals O, Kaiser Ł, Koo T, Petrov S, Sutskever I, Hinton G. Grammar as a foreign language. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 2773–2781.

[33] Sukhbaatar S, Szlam A, Weston J, Fergus R. End-to-end memory networks. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 2440–2448.

[34] Zhong H, Mei H. An empirical study on API usages. IEEE Trans. on Software Engineering, 2019, 45(4): 319–334. [doi: 10.1109/TSE.2017.2782280]

[35] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv: 1301.3781, 2013.

[36] Nguyen AT, Nguyen TN. Graph-based statistical language model for code. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 858–868. [doi: 10.1109/ICSE.2015.336]

[37] 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 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 511–522. [doi: 10.1145/2950290.2950333]

[38] Hindle A, Barr ET, Su ZD, Gabel M, Devanbu P. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 837–847. [doi: 10.1109/ICSE.2012.6227135]

[39] Dam HK, Tran T, Pham T. A deep language model for software code. arXiv: 1608.02715, 2016.

唐泽(1994－), 男, 博士生, 主要研究领域为代码理解, API 补全.

葛季栋(1978－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件工程, 分布式计算与边缘计算, 业务过程管理, 自然语言处理.

李传艺(1991－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为软件工程, 业务过程管理, 自然语言处理.

骆斌(1967－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件工程, 人工智能.