软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006974]

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

基于深度学习的函数名一致性检查及推荐方法*

郑 炜[1,3,4], 唐 辉[1], 陈 翔[2], 张永杰[1]

[1](西北工业大学 软件学院, 陕西 西安 710072)

[2](南通大学 信息科学技术学院, 江苏 南通 226019)

[3](空天地海一体化大数据应用技术国家工程实验室 (西北工业大学), 陕西 西安 710072)

[4](大数据存储与管理工业和信息化部重点实验室 (西北工业大学), 陕西 西安 710072)

通信作者: 陈翔, E-mail: xchencs@ntu.edu.cn

摘 要: 函数是大多数传统编程语言中聚合行为的最小命名单元, 函数名的可读性对于程序员理解程序功能及不同模块之间的交互有着至关重要的作用, 低质量的函数名会使开发人员感到困惑, 增加代码中的坏味道, 进而引发由 API 误用而导致的软件缺陷. 为此, 提出一种基于深度学习的函数名一致性检查及推荐方法, 该方法被命名为 DMName. 首先, 对于给定的目标函数源码, 分别构建其内部上下文、交互上下文、兄弟上下文和封闭上下文, 合并后得到上下文信息标记序列, 然后利用 FastText 词嵌入技术将标记序列转换为上下文表示向量序列, 输入到 seq2seq 模型编码器中, 引入 Copy 机制和 Coverage 机制分别解决 OOV 问题和重复解码问题, 输出目标函数名预测结果的向量序列, 借助双通道 CNN 分类器进行函数名的一致性判断, 若不一致则根据向量空间相似度匹配直接映射获得推荐的函数名. 实验结果表明, DMName 方法在函数名一致性检查任务和函数名推荐任务中的 *F*1 值分别达到 82.65% 和 73.31%, 比目前最优的 DeepName 方法分别提高 2.01% 和 2.96%. 最后, 在 GitHub 大规模开源项目 lancia 中对 DMName 方法进行验证, 挖掘得到 16 个函数名不一致问题并进行合理的名称推荐, 进一步证实 DMName 方法的有效性.

关键词: 函数名; 一致性检查; 名称推荐; 深度学习; seq2seq 模型

中图法分类号: TP311

中文引用格式: 郑炜, 唐辉, 陈翔, 张永杰. 基于深度学习的函数名一致性检查及推荐方法. 软件学报. http://www.jos.org.cn/1000-9825/6974.htm

英文引用格式: Zheng W, Tang H, Chen X, Zhang YJ. Function Name Consistency Check and Recommendation Based on Deep Learning. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6974.htm

Function Name Consistency Check and Recommendation Based on Deep Learning

ZHENG Wei[1,3,4], TANG Hui[1], CHEN Xiang[2], ZHANG Yong-Jie[1]

[1](School of Software, Northwestern Polytechnical University, Xi'an 710072, China)

[2](School of Information Science and Technology, Nantong University, Nantong 226019, China)

[3](National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology (Northwestern Polytechnical University), Xi'an 710072, China)

[4](Key Laboratory of Big Data Storage and Management, Ministry of Industry and Information Technology (Northwestern Polytechnical University), Xi'an 710172, China)

Abstract: The functions are the smallest naming unit of aggregation behavior in most traditional programming languages. The readability of function names plays a vital role in programmers' understanding of program functions and the interaction between different modules. Low-quality function names may confuse developers, increase the smell in the code, and then result in software defects caused by API

* 基金项目: 国家重点研发计划 (2020YFC0833105Z1); 国家自然科学基金 (62141208)
收稿时间: 2022-10-13; 修改时间: 2023-01-07, 2023-04-06; 采用时间: 2023-05-20; jos 在线出版时间: 2023-10-11

misuse. Therefore, a method of function name consistency checking and recommendation based on deep learning is proposed, which is named DMName. Firstly, for the given source code of the target function, the internal context, interactive context, sibling context, and closed context are constructed respectively, and the context information tag sequence is obtained after merging them. Then the tag sequence is converted into the context representation vector sequence by using the word embedding technology FastText and input into the encoder of the seq2seq model. The copy mechanism and coverage mechanism are utilized to solve the OOV problem and the repeated decoding problem, respectively. Finally, the vector sequence of the prediction result of the target function name is output, and the consistency of the function name is predicted with the help of the two-channel CNN classifier. If the function name is inconsistent, the recommended function name can be obtained by direct mapping according to the vector space similarity matching. The experimental results show that the *F*1-*measure* of DMName in function name consistency check and recommendation reaches 82.65% and 73.31% respectively, which is 2.01% and 2.96% higher than the current optimal DeepName. Finally, the DMName is verified in the large-scale open-source project, namely lancia in GitHub. A total of 16 function name inconsistency problems are found, and reasonable name recommendations are made, which further confirms the effectiveness of DMName.

Key words: function name; consistency check; name recommendation; deep learning; seq2seq model

程序标识符命名是编码过程中的关键步骤, 也是开发人员必须完成的最困难任务之一[1,2]. 其中函数名是开发人员理解程序或 API 行为的最为直观的重要信息[3−6], 简洁且有意义的函数名对于程序代码的可理解性尤为重要, 不规范的函数名可能会使开发人员感到困惑, 增加代码中的坏味道 (code smell), 使软件程序更加难以理解和维护[7−12], 进而引发由 API 误用而导致的软件缺陷[13−15], 例如常用的静态分析器 FindBugs[16]就曾发现过多达 10 种与函数名不一致相关的缺陷类型.

软件设计大师 Martin[17]曾直言: “缓存失效和命名是计算机科学中普遍存在的两个难题. 任何傻瓜都可以编写计算机可以理解的代码, 而优秀的程序员则擅长编写人类可以理解的代码”. Høst 等人[18]则更加明确地指明了函数名命名在程序标识符命名中的重要性: “函数是大多数传统编程语言中聚合行为的最小命名单元, 是抽象的基石, 函数名的可读性对于程序员理解程序功能及不同模块之间的交互有着至关重要的作用”. 据有关统计显示[19], 在开发人员的日常工作中, 其阅读代码与编写代码的时间投入比例超过 10∶1, 并且当前业务逻辑代码的编写难度与相关代码的可读程度密切相关, 模糊的函数名会使代码更加难以理解和修改.

在软件项目开发中频繁地代码维护或版本更迭会大幅度提高函数命名的难度, 使得函数名受到时效性的不利影响: 当在开发阶段引入新的程序模块时, 上下文中原有的变量或函数语义会发生变化, 从而导致函数名退化为一个不符合语义的糟糕名称, 尤其是在第三方库进行版本更迭时, 函数名的命名要求更为严格. 不规范的函数名会使引入该库的软件项目无法保证向后兼容性, 在极大提高软件维护成本的同时, 也会给企业带来不可预估的损失.

在软件开发过程中, 一般存在两种可能导致函数名不一致的情况: 第 1 种情况是在软件程序开发过程中函数的不规范命名, 使其无法清晰地描述出函数的实际功能; 第 2 种情况是在软件演变和迭代过程中, 函数体的代码修改使其主要功能发生了改变, 从而与原有函数名描述信息不一致. 根据调查研究, 开发人员仅在程序维护期间就花费了大约一半的开发时间来理解代码[20], 意义不明确或与实际功能不符的函数名, 使软件程序更加难以理解和维护. 因此解决函数名的不一致性问题可以有效规避编码风险, 推荐有意义的名称可以进一步提高代码的可读性[21], 进而提高开发人员的编程效率和企业生产率[22].

当前已有的基于深度学习的函数名一致性检查及推荐方法, 普遍存在 OOV (out-of-vocabulary) 问题和解码器重复解码问题, 严重影响模型的在实际应用中的推荐准确性. 为此, 本文提出了一种更为有效的函数名一致性检查及智能推荐方法, 该方法被命名为 DMName (deep method name). 首先, 对于给定的目标函数源码, 分别构建 4 种上下文信息的标记序列, 然后利用 FastText[23]词嵌入技术将标记序列转换为上下文表示向量序列, 输入到 seq2seq 模型编码器中, 引入 Copy 机制和 Coverage 机制分别解决 OOV 问题和重复解码问题, 输出目标函数名预测结果的向量序列, 借助双通道 CNN 分类器进行函数名的一致性判断, 或者根据向量空间相似度匹配直接映射获得推荐的函数名.

我们在目前主流的 Liu 等人[24]所收集的函数名一致性检查数据集和 Nguyen 等人[25]所收集的函数名推荐数据集中分别进行了对比实验验证, 实验结果表明, 在函数名一致性检查任务中, DMName 方法在 *F*1 值和正确率指

标方面达到了 82.65% 和 76.04%, 比目前最优的 DeepName[26]方法分别提高了 2.01% 和 1.93%; 在函数名推荐任务中, DMName 方法在 $F1$ 值指标方面达到了 73.71%, 比最优的 DeepName[26]方法提高了 1.96%. 最后, 在 GitHub 现实开源项目 lancia[27]中对 DMName 方法进行了初步验证, 挖掘得到 16 个函数名不一致问题并进行了合理的名称推荐, 进一步证实了 DMName 方法的有效性.

总体来说, 本文的主要贡献如下.

(1) 提出一种基于深度学习的函数名一致性检查及推荐方法, 该方法被命名为 DMName, 其使用目前主流的 seq2seq 框架进行设计, 实验结果表明, DMName 方法在函数名一致性检查任务和函数名推荐任务中的 $F1$ 值分别达到了 82.65% 和 73.31%, 后续将 DMName 方法应用于开源项目 lancia 中, 挖掘得到了 16 个函数名不一致问题并进行了合理的名称推荐.

(2) 利用 Copy 机制将传统词表 (不包含 unknown 词) 与源序列中的词构成的词表进行合并, 在拓展词表范围的基础上, 将抽取式摘要与生成式摘要的思想综合运用于函数名推荐任务中, 同时利用 FastText 词嵌入方法可以叠加单词字符级的 n-gram 向量的优势, 解决了模型训练过程中的 OOV 问题.

(3) 利用 Coverage 机制, 通过统计各个单词在历史向量中的出现的概率值的累积和, 将累计和分别纳入解码器的惩罚项, 以解决函数名生成任务中相同单词不停循环出现的重复解码问题, 保证函数名推荐列表中不会出现相同的名称, 进一步提高了函数名推荐的准确性.

1 DMName 方法

1.1 研究动机

函数名的一致性检查及推荐研究本质上属于自然语言处理 (natural language processing, NLP) 领域中的极端摘要问题. 按照摘要的生成方式不同, 可以将其分为抽取式摘要和生成式摘要两种. 在本文所涉及的研究场景中, 抽取式摘要是指通过抽取拼接函数源代码中的关键词来生成函数名摘要, 而生成式摘要则是系统地根据函数代码蕴含的语义, 在词表中选取合适的单词对代码进行概括, 从而生成函数名. 当前基于机器学习的函数名一致性检查及推荐研究大多属于生成式摘要的范畴, 且 seq2seq+Attention 模型架构是处理该任务的主流深度学习方法, 目前此类方法仍然面临如下挑战.

(1) OOV 问题导致生成能力受限. OOV 问题的根源在于有限的词表大小无法涵盖所有可能出现的低频词, 当预测结果中出现词表中不存在的单词时, 只能以“unknown”的形式代替输出, 从而影响训练过程中模型的理解能力, 使得模型生成函数名的能力受限.

(2) 存在解码器重复解码问题. 在 seq2seq 模型中, 为了避免固定维度语义向量在数据编码时存在的信息损失, 一般会在编码器端引入了注意力机制, 以避免编码获取的单个语义向量的信息损失问题, 但注意力机制的引入也会使得解码器总是盯着同一个权重最大词, 从而导致生成结果中的单词重复, 导致解码器端的重复解码问题.

本文旨在提出一种基于深度学习的函数名一致性检查及推荐技术方案, 在解决 OOV 问题与解码器重复解码问题的基础上, 对实施各环节的关键技术选取进行技术对比实验和消融实验, 然后对所提出的方法模型进行实验评估, 最终将其应用于 GitHub 的现实开源项目中, 挖掘其中蕴含的函数名不一致性问题并进行名称推荐, 以验证其实际应用价值.

1.2 方法概述

本文提出了一种基于深度学习的函数名一致性检查及推荐方法, 最终的方法被命名为 DMName, 该方法的流程图如图 1 所示. 对于给定的目标数函数源码, 在对其进行数据处理后, 分别构建其内部上下文、交互上下文、兄弟上下文和封闭上下文, 合并后得到上下文信息标记序列, 利用 FastText 词嵌入技术对标记序列进行向量化处理, 从而获取上下文表示向量序列, 将向量序列作为 seq2seq 模型编码器的输入, 解码器则输出目标函数名预测结果的向量序列. 对于函数名的一致性检查任务, 在利用 FastText 技术完成对每个函数名词嵌入的基础上, 将真实函数名嵌入后的向量序列与模型的结果向量序列输入到双通道的 CNN 分类器中, 在经过 Softmax 归一化处理后获取

两者的相似性概率, 从而完成函数名的一致性判断; 对于函数名的推荐任务, 则将所有生成的向量序列依照其概率分布映射到词表所标记的关键词中, 并按照映射的先后顺序组成最终的函数名, 从而完成函数名的智能推荐.

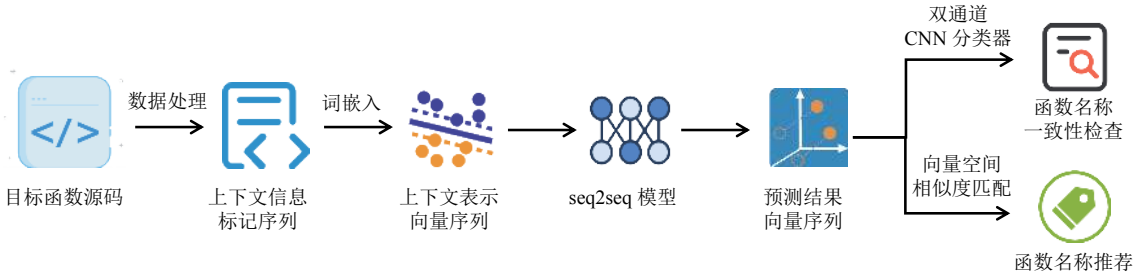

图 1 DMName 方法流程图

seq2seq 模型的编解码器设计如图 2 所示, seq2seq 模型中使用 4 个双向单层 GRU 网络作为编码器, 编码器的每个 GRU 网络的输入分别是目标函数的 4 种上下文向量序列之一, 该设计方法是考虑到不同上下文信息各自所具有的结构特征可能不同, 从而避免不同上下文向量序列在编码过程中的交叉影响. 编码器可以将所有时刻 GRU 单元隐藏状态组合在一起, 从而获得编码器的输出 $H=[h_1,h_2,\ldots,h_T]$, 最终通过编码器的自定义函数 q 计算出固定维度的上下文语义向量 c, 而为了避免固定维度语义向量 c 在数据编码时存在的信息损失, 则需要在编码器端引入注意力 (Attention) 机制, 以保证在训练时能够对不同的输入标记分配不同的注意力分布权重 a^t, 其可以看作是在输入序列上的概率分布, 并以可变的上下文语义向量 C_t 来代替固定维度的语义向量 c, 而 C_t 则可以通过计算编码器隐藏状态的加权和得到. 注意力机制的引入可以使模型推理出不同序列数据之间的复杂映射关系, 使得模型更专注于寻找输入序列中显著地与当前输出相关的有用信息, 输出每个时间步注意力向量所构成的编码序列, 提高输出预测向量的质量.

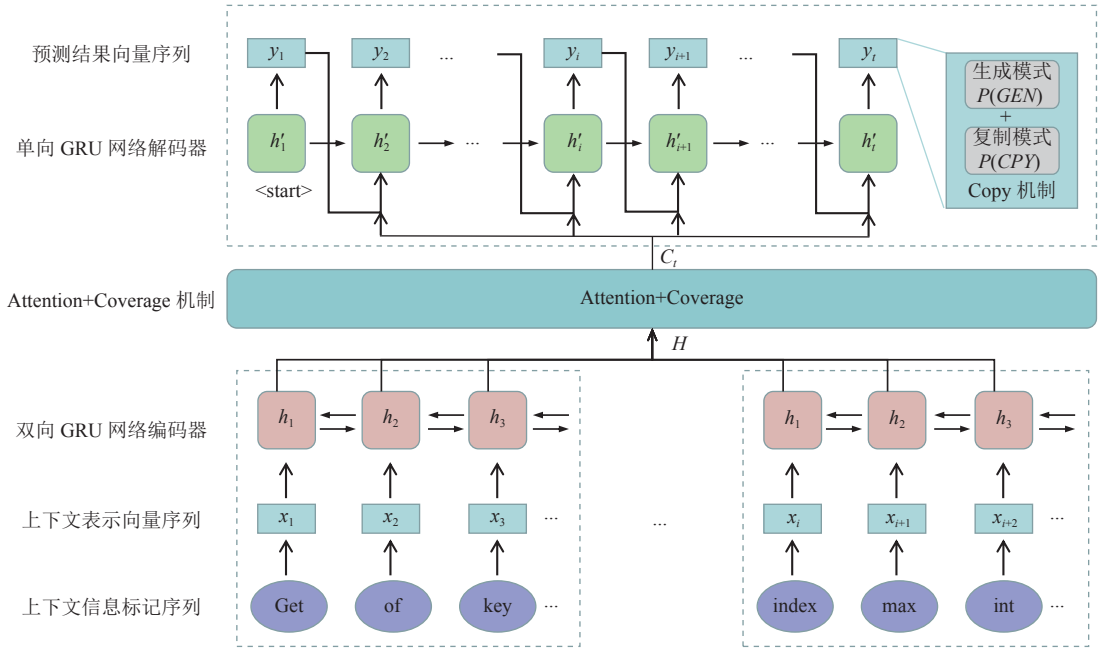

图 2 seq2seq 模型的设计框架图

然而, 注意力机制的引入使得解码器总是盯着同一个权重最大的标记词, 很容易引起解码器端的重复解码问题, 导致预测的函数名中存在重复标记, 因此本文尝试在 seq2seq 模型中引入 Coverage 机制来解决该问题. Coverage

机制的概念最早由 See 等人[28]提出, 其通过将 t 时间步之前出现的各个标记的注意力分布权重进行累加得到覆盖向量 c^t, 该向量可以告知模型输入到编码器中的已被注意的部分和未被注意的部分有哪些, 从而用先前的注意力权重决策来影响当下注意力权重的决策, 避免同一标记的重复生成, 保证生成的函数名中不会出现相同的标记, 从而解决了解码器端的重复解码问题, 最终提高函数名推荐的效率和准确性. 需要注意的是, Coverage 机制需要在模型训练的最后阶段加入, 其约占总训练时间的 1%, 如果从刚开始训练时就加入 Coverage 机制反而会影响模型的训练效果.

seq2seq 模型的解码器由一个单层单向的 GRU 网络构成, 其输出预测序列的过程本质上属于生成式摘要任务, 由于传统词表大小的限制, 会使得部分低频词无法被纳入词表, 从而导致 OOV 问题, 因此文本旨在引入 Copy 机制来解决该问题. Copy 机制的概念最早来源自 Gu 等人[29]提出的 CopyNet 模型, 其包含了生成模式 (generate-mode) 与复制模式 (copy-mode) 两个模块. 生成模式对应于生成式摘要任务, 使用和维护的是常规词表, 但是表中不包含 UNK 等无效词; 复制模式对应于抽取式摘要任务, 使用和维护的是源输入序列中的词所构成的词表, 词表的长度是源序列中的词去重后的数量.

对于 GRU 单元的每个时间步 t, 为了权衡某个预测词是从常规词表中生成的还是从输入序列词表中复制的, 需要引入一个生成概率 $p_{\text{gen}} \in [0,1]$. 在生成模式下, 对于每个时间步 t, 可变的上下文语义向量可以与解码器隐藏状态连接, 并生成各单词在常规词表上的概率分布 P_1; 在复制模式下, 单词标记在输入序列词表上的概率分布 P_2 是其注意力分布权重的累加和. 因此, 模型对单词标记 w 的最终预测概率 $P(w)$ 取决于生成模式与复制模式的预测概率之和.

$$P(w) = P(GEN) + P(CPY) = p_{\text{gen}} \times P_1 + (1 - p_{\text{gen}}) \times P_2 \tag{1}$$

简而言之, 当一个词未在常规词表中出现时, 常规词表上的概率分布 P_1 为 0, 而当一个词未在输入序列中出现时, 输入序列词表上的概率分布 P_2 则为 0. 在模型预测阶段, 其输出的函数名预测向量的概率分布维度是常规词表与输入序列词表的去重并集的大小.

因此, 借助 Copy 机制的生成模式和复制模式, 使没有出现在传统词表中的单词标记能够在输入序列中直接复制, 再结合 FastText 词嵌入技术可以叠加单词字符级的 n-gram 向量的优势, 从而有效缓解了标记生成时所存在的 OOV 问题. 最后, 可以由解码器输出每个时间步的预测向量, 最终组成函数名预测结果的标记向量序列.

1.3 上下文表示学习

1.3.1 上下文信息定义

本文所采用的代码表征形式是基于标记 (token) 的上下文信息代码表征形式, 需要在收集函数相关源代码数据集的基础上, 构建内部上下文、交互上下文、兄弟上下文和封闭上下文等 4 种上下文信息, 然后按照驼峰式命名法 (camel-case) 将其中各类实体名称按照单词出现的先后顺序转换为 token 序列, 从而实现基于标记的代码特征表示, 以供下游任务使用, 4 种上下文信息定义如下.

(1) 内部上下文 (internal context): 包括当前函数所传入的参数的类型及名称, 函数的返回类型, 函数体内部除部分介词、独立数字和标点符号等无效特征词以外的代码数据.

(2) 交互上下文 (interaction context): 包括调用当前函数的调用方函数 (caller) 的名称及其内部上下文, 当前函数体内被调用的其他函数 (callee) 的名称及其内部上下文.

(3) 兄弟上下文 (sibling context): 若当前函数是类内的成员函数, 那么兄弟上下文是指与当前函数同属于同一个类中的兄弟函数的名称及其内部上下文.

(4) 封闭上下文 (enclosing context): 是指当前函数所属封闭类的名称以及类中的程序实体、方法调用、字段访问、变量和常量的名称.

1.3.2 上下文信息构建

对于内部上下文、兄弟上下文和封闭上下文来说, 可以直接从相关源代码中提取各类程序实体、返回类型、参数值及参数类型的名称, 然后与按照驼峰式命名法将所有名称分解为子标记, 从而构成 3 种不同的上下文标记

序列. 以图 3 中的 Java 代码为例, 所构建的 MaxHeap 为封闭的大顶堆类, 若以第 14 行中的 getElementKey 方法为函数名一致性检查及推荐的目标, 那么其内部上下文包括第 13–16 行的代码内容, 构建步骤如下.

```
1  import DataStructures.Heaps.Heap;
2  ...
3  public sealed class MaxHeap implements Heap {
4  ...
5      /**
6      * Get the element at a given index. The key for the list is equal to index value - 1
7      */
8      public HeapElement getElement (int elementIndex) {
9          if ((elementIndex <= 0) || (elementIndex > maxHeap.size()))
10             throw new IndexOutOfBoundsException("Index out of heap range");
11         return maxHeap.get(elementIndex - 1);
12     }
13     // Get the key of the element at a given index
14     private double getElementKey(int elementIndex) {
15         return maxHeap.get(elementIndex - 1).getKey();
16     }
17     // Swaps two elements in the heap
18     private void swap(int index1, int index2) {
19         HeapElement temporaryElement = maxHeap.get(index1 - 1);
20         maxHeap.set(index1 - 1, maxHeap.get(index2 - 1));
21         maxHeap.set(index2 - 1, temporaryElement);
22     }
23     ...
24     @Override
25     public void insertElement(HeapElement element) {
26         ...
27     }
28     @Override
29     public void deleteElement(int elementIndex) {
30         ...
31     }
32 ...
```

图 3　上下文信息构建的代码示例

(1) 对第 13 行的函数头部注释进行分解, 得到 [Get, key, of, element, at, given, index] 的子标记序列. 需要注意的是, 序列中按照正则表达式过滤掉了“the”和“a”等无效特征词, 但保留了“of”“at”等可能会在名称中使用的介词.

(2) 对第 14 行中的函数修饰词“private”、返回值类型“double”、参数名称“elementIndex”、参数类型“int”进行子标记分解, 构成 [private, double, element, Index, int] 的子标记序列. 需要注意的是, 第 14 行代码中的标点符号都会被过滤掉, 已命名的函数名 getElementKey 会被作为提取出来作为样本标签, 不会放在内部上下文标记序列中.

(3) 对第 15, 16 行中的函数体代码进行分解, 在过滤掉标点符号和数字的基础上, 得到 [return, max, Heap, get, element, Index, -, get, Key] 的子标记序列.

(4) 对前面得到的所有子序列进行合并, 在保证原子序列单词顺序的基础上, 获得 [Get, key, of, element, at, given, index, private, double, element, Index, int, max, Heap, get, element, Index, -, get, Key] 的内部上下文标记序列, 由于在后续的向量表示任务中, 需要对序列中的词频进行统计, 因此要避免对其进行去重操作.

由于示例中的目标函数是类内的成员函数, 因此还需要提取该函数的兄弟上下文, 即与当前函数同属于同一个类中的兄弟函数的名称及兄弟函数的内部上下文. 在图 3 的代码示例中, 第 8、18、25、29 行的 getElement、swap、insertElement、deleteElement 成员函数分别为类内目标函数的兄弟函数, 因此可以按照之前内部上下文的构建步骤对每个兄弟函数进行子序列的构建, 区别在于每个兄弟函数的函数名也要加入子序列之中. 最后, 只需将每个兄弟函数所构建的子序列进行拼接, 即可获得兄弟上下文的标记序列.

Java 语言中的封闭类是指被 sealed 修饰的类或接口, 同时可以使用 permits 关键字来指定可以继承或实现该类的子类类型, 此外, 封闭类中被 sealed、final、non-sealed 修饰的 3 种子类也属于本文要提取的封闭上下文的范围. 由于代码示例中目标函数所在类被 sealed 关键字修饰, 因此需要对该类的相关内容提取封闭上下文信息, 提取范围包括封闭类的名称以及类中的程序实体、方法调用、字段访问、变量和常量的名称, 提取方法依然按照驼峰式命名法将其中各类实体名称按照单词出现的先后顺序转换为子标记序列.

与前面 3 种上下文信息相比, 交互上下文的标记序列一般无法直接从目标函数邻近的源代码中直接获取, 因此需要借助 Soot 工具[30]构建函数调用图 (call graph), 从而识别并提取出目标函数的调用方函数 (caller) 和被调用函数 (callee). 在函数名的一致性检查中, 由于目标函数的函数名已经给出, 所以其交互上下文同时包括调用方函数和被调用函数各自的名称及内部上下文; 在函数名的推荐任务中, 由于目标函数的函数名未给出, 所以无法获取其调用方函数, 因此该任务的交互上下文只包括被调用函数的名称及内部上下文. 其中, 调用方函数和被调用函数的内部上下文标记序列的构建方法与目标函数内部上下文的构建方法一致, 因此不再赘述.

1.3.3 上下文向量表示

上下文向量表示旨在将目标函数的 4 种上下文信息所对应的标记序列转换为词向量序列, 该方法实质上是将每个标记序列视为一条英文句子, 每个标记视为句子中的一个单词, 向量表示的过程本质上是对句子中的各个单词实现词嵌入的过程. 为了实现上下文向量表示, DMName 方法使用 FastText 技术作为上下文标记序列的词嵌入方法, FastText 可以通过叠加单词中字符级的 *n*-gram 向量来构建词典, 并统计其中的词频和数据的上下文关系, 以此来完成每个单词的向量化过程.

FastText 中假定每个单词标记由 n 个字符组成, 并采用字符级别的 *n*-gram 表示方法, 其输入是多个单词标记及单词标记的字符级别的 *n*-gram 向量, 利用隐含层对多个词向量进行叠加平均, 最终借助 Hierarchical Softmax 进行分类输出, FastText 可以用于实际的文本分类任务, 词向量其实是该过程的中间产物. 由于低频词仍然可以被分解为 *n*-gram 表示, 因此 FastText 对低频词生成的词向量效果会更好. 此外, 对于训练语料库之外的单词标记, FastText 仍然可以通过叠加单词字符级的 *n*-gram 向量的方式, 构建 OOV 单词标记的词向量, 因此可以在一定程度上缓解模型预测时的 OOV 问题.

在本文任务中, 对于一个上下文标记序列 $X=[x_1,\ldots,x_N]$, 其中 x_i 表示序列中的第 i 个标记, 那么其经过 FastText 向量化之后得到的向量序列 $V=[v_1,\ldots,v_N]$ 中的第 i 个向量 v_i 即为 x_i 的词向量, 即标记序列中每个标记的顺序与向量序列中的向量顺序是一致的, 由于不同上下文标记序列 X 中包含的标记数目 N 不同, 使得词嵌入后得到的向量序列 V 的长度 N 是不固定的, 因此需要提前设置好向量序列 V 的固定长度 L. 假如向量序列的长度 $N>L$, 那么需要将向量序列从 L 长度处截断; 若向量序列的长度 $N<L$, 那么需要对向量序列的尾部填充零向量直到其长度达到 L 为止.

1.4 函数名一致性检查及推荐

在对词表 (常规词表与输入序列词表的去重并集) 中的所有标记计算获得各自的预测概率后, 模型会选择当前时间步中预测概率最高的标记的表示向量作为此时解码器的输出, 最终得到当前函数在所有时间步长所输出的函数名预测向量集合 V_{cur}, 其中每个向量都保持输出时的原始顺序. 需要注意的是, 解码器最后获得的是预测向量集合, 而不是预测标记集合, 是因为编码器的原始输入是 FastText 词嵌入后的标记向量序列, 所以解码器的每个时间步输出也是预测概率最高的标记的表示向量. 在获得函数名预测向量集合 V_{cur} 后, 就可以根据任务场景对 V_{cur} 进行不同的处理操作, 即函数名的一致性检查任务或者函数名的智能推荐任务.

在函数名的一致性检查任务中, 需要提前对已有的函数名进行处理, 通过对其进行标记拆分和 FastText 词嵌入, 从而获得已有函数名的标记向量集合 V_{exist}. 那么对已有函数名的一致性检查则转换为对预测向量集合 V_{cur} 和标记向量集合 V_{exist} 的对比, 其本质上一个二分类问题, 因此可以引入一个双通道的 CNN 网络模型作为分类器, 并使用随机采样技术在数据集中抽取训练标记词对分类器进行训练, 从而可以将 V_{cur} 和 V_{exist} 视为两个矩阵 M_{cur} 和 M_{exist} 输入到 CNN 模型中, 每个矩阵占用一个通道, 然后将 CNN 模型的输出进行 Softmax 归一化处理, 最终获得一个 0

到 1 之间的分类结果, 其中 1 代表已有函数名与预测的函数名一致, 0 代表已有函数名与预测的函数名不一致.

在函数名的智能推荐任务中, 由于函数名是未知的, 因此与函数名一致性检查任务的处理方式有所不同, 对于预测向量集合 V_{cur} 中的每一个预测向量 v_w, 通过在向量化词表 (常规词表与输入序列词表的去重并集的标记向量表示集合) 的标记向量中寻找与 v_w 距离最近的向量 v_{w^*}, 并使用 v_{w^*} 对应的预测标记 w^* 作为构成函数名的子标记之一. 在利用该方法获得所有的预测标记后, 按照原预测向量集合 V_{cur} 中的顺序对所有的预测标记进行拼接, 将其作为最终生成的函数名, 最终完成函数名的智能推荐.

2 实验验证

2.1 数据集选取与处理

对于函数名一致性检查任务来说, 为了便于与其他基准方法进行比较, 本文使用了目前主流的函数名一致性检查数据集, 该数据集最早由 Liu 等人[24]所收集处理, 表 1 展示了数据集的统计信息, 其数据来源于 Apache、Spring、Hibernate 和 Google 等 4 个社区的 430 个高评级 Java 开源项目的最新版本, 并从中提取了 2 119 573 个符合一致性的函数及函数名, 挖掘了 1 402 个不一致的函数名数据样本. 函数名一致性检查实验时, 测试集由 1 402 个不一致的函数名数据样本和 1 402 个一致的函数名数据样本组成, 其中一致的函数名数据样本是从 2 119 573 个一致性函数名数据中随机抽取的, 而剩余的 2 118 171 个一致性函数名样本数据则作为模型的训练集. 需要说明的是, 虽然函数名的一致性检查任务的最终目标是获得是否一致性的分类结论, 但本文所训练的 seq2seq 模型本质上处理的是摘要生成任务, 只是在模型测试阶段才使用双通道 CNN 网络进行名称的一致性判断, 因此训练过程中所使用的训练集并不需要负样本, 即不一致的函数名数据样本.

对于函数名的推荐任务来说, 本文采用的数据集最早由 Nguyen 等人[25]所收集处理, 其参考了 Liu 等人[24]的数据采集方法, 也是该任务领域目前最通用的数据集. 表 2 展示了数据集的统计信息, 数据主要来源于 GitHub 开源社区的 10 222 个排名靠前的高质量 Java 开源项目, 从中提取了 14 458 828 个函数及函数名, 按照项目数量比例 9:1 的策略进行数据集划分, 将 9 200 个项目中的 13 992 028 个函数归为训练集, 剩余的 1 022 个项目中的 466 800 个函数归为测试集. 这种按照项目数量来划分数据集的方法使得训练集和测试集中的项目完全不同, 其检测结果可以更真实地反映模型在现实项目中的表现.

在上述已知的数据集中, 主要内容包括项目名、包名、类名、函数名、函数参数及类型、函数返回类型、函数体内部代码标记、封闭类名称及类内实体标记, 因此该数据集基本已涵盖 4 种代码表征形式中的两种, 即内部上下文和封闭上下文. 因此需要如何根据已有的两种上下文信息来获取兄弟上下文及交互上下文, 并对其数据处理过程进行简单介绍.

表 1 函数名一致性检查数据集的统计信息

一致性	训练集	测试集	总计
名称一致	2 118 171	1 402	2 119 573
名称不一致	0	1 402	1 402
总计	2 118 171	2 804	2 120 975

表 2 函数名推荐数据集的统计信息

统计量	训练集	测试集	总计
函数数量	13 992 028	466 800	14 458 828
文件数量	1 756 282	51 631	1 807 913
项目数量	9 200	1 022	10 222

对于兄弟上下文来说, 本文根据目标函数所处项目中的特定包下的特定类, 对数据集中同一个类下的函数数据样本进行筛选, 从而获得目标函数所在类内的成员函数. 然后将除目标函数以外的所有成员函数的内部上下文信息进行提取, 最后拼接得到目标函数的兄弟上下文.

而交互上下文的挖掘过程则相对复杂, 由于其一般无法直接从目标函数邻近的源代码中直接获取, 因此需要首先根据目标函数所在的项目名称, 爬取并保存项目源码, 然后借助 Soot 工具对目标函数所在源码构建函数调用图, 输出得到函数调用图的 DOT 文件, 然后对 DOT 文件进行解析, 寻找到目标函数节点所在的直接前驱节点 (调用方函数) 和直接后继节点 (被调用函数), 并在源码中对调用方函数和被调用函数进行定位, 在获取相关函数源码之后, 还需要进一步对源码进行数据清洗. 数据清洗的目的在于消除数据中与本文任务无关的特征和琐碎的

信息, 一般采用正则表达式来对其进行处理, 需要识别并删除的元素包括: (1) 标点符号; (2) 部分介词, 例如“the”“a”等; (3) 独立数字; (4) 作者信息, 例如“@author”; (5) 时间信息, 例如“@date”; (6) 网址信息, 例如“http://”. 最后使用NLTK 软件包分割其中实体名称的单词标记, 最终即可获得交互上下文的标记序列.

对两类数据集中包含的标记数量进行分析后发现, 在函数名一致性检查数据集中, 函数名中蕴含标记数量的平均值和中位数分别为 4.7 和 2, 上下文信息标记序列的平均值和中位数分别为 410.8 和 281; 而在函数名推荐数据集中, 函数名中蕴含标记数量的平均值和中位数分别为 4.1 和 2, 上下文信息标记序列的平均值和中位数分别为 433.4 和 301, 对标记数量的统计分析结果也为实验的参数设置提供了依据.

需要说明的是, 本文在实验中采用十折交叉验证的方法, 将训练集随机划分为 10 个互不相交的子集, 轮流选取每个子集作为验证子集, 其余 9 个子集作为训练子集, 使用训练子集训练模型, 并在验证子集上评估模型的性能, 最终选取多次实验的指标平均值来评估模型性能. 交叉验证的方法可以更好地利用有限的训练数据, 降低评估误差并提高模型性能评估的准确性, 由于每次选取的验证子集不同, 因此也能更全面地反映模型对不同样本的处理能力. 最后, 在测试集上对所选模型进行测试, 以评估其泛化能力和应用效果.

2.2 度量指标

2.2.1 函数名一致性检查

机器学习任务中的混淆矩阵 (confusion matrix) 可以用来衡量分类结果的混淆程度, 混淆矩阵中共有 4 种组合解释, 在函数名一致性检查任务中的含义分别如下.

(1) *TP* (true positive, 真阳): 模型预测为名称不一致, 实际答案为名称不一致.

(2) *FP* (False Positive, 假阳): 模型预测为名称不一致, 实际答案是名称一致.

(3) *TN* (True Negative, 真阴): 模型预测为名称一致, 实际答案为名称一致.

(4) *FN* (False Negative, 假阴): 模型预测为名称一致, 实际答案为名称不一致.

据此, 以推算出函数名一致性检查任务中的 4 个度量指标, 分别是精确率 (*Precision*)、召回率 (*Recall*)、*F*1 值 (*F*1-*measure*) 和正确率 (*Accuracy*). 精确率表示被模型预测为名称不一致, 而实际上的确不一致的样本数量占被模型预测为名称不一致的所有样本数量的比例.

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

召回率表示被模型预测为名称不一致, 而实际上的确不一致的样本数量占实际上不一致的所有样本数量的比例.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

*F*1 值表示精确率和召回率的调和平均值, 属于模型综合性指标, 可以更好地评价模型分类的效果.

$$F1\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4}$$

正确率表示被模型正确预测为名称一致和正确预测为名称不一致的样本数量之和占所有样本的比例.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

2.2.2 函数名推荐

函数名推荐任务的度量指标主要有 4 个, 分别是精确率、召回率、*F*1 值和完全匹配率 (exact-matched, ExMatch). 这些度量指标的计算方法并不依赖于混淆矩阵, 而是依赖于子标记集合的概念, 即定义 *subtoken*(*m*) 为构成函数名 *m* 的子标记集合, 例如 *subtoken*(“getFinallySet”)={“get”, “Finally”, “Set”}, 该子标记集合的提取方法与数据处理方法一致, 都是利用 NLTK 软件包进行实体名称的子标记提取的.

精确率表示模型推荐函数名 *r* 与真实函数名 *e* 两者的子标记集合交集占模型推荐的函数名 *r* 子标记集合的比例.

$$Precision(e, r) = \frac{subtoken(r) \cap subtoken(e)}{subtoken(r)} \tag{6}$$

召回率表示模型推荐函数名 *r* 与真实函数名 *e* 两者的子标记集合交集占真实函数名 *r* 子标记集合的比例.

$$Recall(e,r)=\frac{subtoken(r)\cap subtoken(e)}{subtoken(e)} \tag{7}$$

$F1$ 值表示精确率和召回率的调和平均值.

$$F1\text{-}measure(e,r)=\frac{2\times Precision(e,r)\times Recall(e,r)}{Precision(e,r)+Recall(e,r)} \tag{8}$$

在实际度量模型的函数名推荐能力时, 上述 3 个指标使用的是数据集所有样本的指标平均值, 且会对获取的子标记集合的单词标记全部转为小写字母. 此外, 对于实验中的所有样本, 本文还会计算函数名的完全匹配率 ExMatch, 即模型推荐函数名 r 与真实函数名 e 两者的子标记集合和标记顺序完全相同的样本数量占所有数据样本数量的比例, 而且在计算完全匹配率时, 子标记集合中的单词标记依然维持原本的大小写字母.

2.3 基准方法

为了评估本文所提出的 DMName 方法的有效性, 需要选择相关的函数名一致性检查及推荐方法作为基准方法, 在考虑已有研究方法的数据集来源、表现效果以及先进性的基础上, 本文在函数名的一致性检查任务中选择了 Liu 等人的方法[24]、MNire[25]和 DeepName[26]这 3 种基准方法, 而在函数名的推荐任务中选择了 code2vec[31]、MNire[25]和 DeepName[26]这 3 种基准方法. 其中由于 MNire 方法与 DeepName 方法能够同时运用于两种任务中, 因此本文所使用的基准方法共有 4 种, 而且这些基准方法所用的数据集来源与本文中的一致, 简单介绍如下.

(1) Liu 等人的方法[24]: 一种基于相似度计算的函数名一致性检查方法, 通过利用段落向量来提取函数名的向量空间表示, 并使用卷积神经网络来提取当前函数名所对应函数体的向量空间表示, 从而分别获取已知函数的函数名特征向量集合 NS 以及其函数体代码的特征向量集合 BS. 对于待检测的函数 m, 将其名称和函数体分别转换为向量表示后, 分别在集合 NS 和集合 BS 中检索进行向量集合相似度检索, 如果两者的检索结果存在某种程度的交集, 那说明函数名符合一致性, 否则, 函数名就不符合一致性.

(2) code2vec 方法: 一种基于 AST 路径与注意力网络结合的函数名推荐方法. 通过将函数体表示为分布式向量, 然后借助注意力机制为函数中的每个 AST 路径计算加权注意力值, 并通过加权求和将它们聚合为单个向量来表示函数体, 其中共享相似 AST 结构的函数向量在连续分布空间中彼此接近, 从而可以有效捕获函数之间以及函数名之间的语义相似性, 通过在海量函数数据中使用丰富的 AST 结构进行训练, code2vec 可以检索到与给定函数在向量空间中十分接近的函数体, 并以该向量空间所指示的函数名作为目标函数的推荐名称.

(3) MNire 方法: 一种基于 seq2seq 模型的函数名一致性检查及推荐方法. 该工具通过构建函数的 3 种上下文信息组成语料库, 在进行词嵌入的基础上获得函数上下文标记的向量集合, 并借助注意力机制赋予每个上下文信息中不同标记的权重, 最终借助编解码器的结构实现推荐函数名的生成, 而在函数名一致性检查任务中, MNire 则通过直接比较模型推荐函数名与实际函数名的标记重叠数量, 来判断某个重叠阈值下实际的函数名是否符合一致性. MNire 方法与本文提出的 DMName 方法相比, 其所挖掘的函数上下文信息不够全面, 而且并没有解决 OOV 问题和解码器循环解码问题.

(4) DeepName 方法: 一种基于 seq2seq 模型的函数名一致性检查及推荐方法. 其模型设计与 MNire 方法类似, 但其拓展了内部上下文、交互上下文、兄弟上下文和封闭上下文等 4 种不同的函数代码表征形式, 并在 seq2seq 模型的编码器中加入了 Copy 机制, 又提出了一种 Non-copy 机制, 通过直接拷贝原序列中的关键词来替换不在训练数据中的单词名称. DeepName 方法是目前最先进的函数名一致性检查及推荐方法, 其中 Copy 机制的使用为本文解决 OOV 问题的研究方向提供了有效参考, 但 DeepName 方法依然没有解决解码器端的重复解码问题.

2.4 实验环境

为了训练和评估本文提出的 DMName 方法, 并进行相关实验验证, 本文采用了深度学习框架 PyTorch 的 1.11.0 版本来实现模型, 编程语言选用的是 Python 的 3.9.5 版本, FastText 词嵌入与训练模型可以通过 4.2.0 版本的 Gensim 库进行训练和加载, 为了提高模型训练的效率, 本文在装有 NVIDIA GeForce RTX 2080Ti 显卡的本地服务器上进行了该实验, 服务器搭载 Intel(R) Xeon(R) Sliver 4110 处理器, CPU 主频为 2.10 GHz, 内存和磁盘容量分别为 64 GB 和 4 TB, 操作系统为 Windows 10 专业版 (64 位).

2.5 参数设置

在词向量模型的训练阶段, 本实验设置的词向量维度为 256, 初始学习率为 0.025, 迭代次数为 5 次, 选择 Hierarchical Softmax 作为优化方法. 为了保证多次实验时的可重现性, 需要设置固定的随机种子数, 并将模型限制为单个工作线程 (workers=1), 从而消除操作系统线程调度所引起的排序抖动, 此外, 还需要设置 PYTHONHASHSEED 环境变量来控制 Python3 解释器启动时的哈希随机化.

在 seq2seq 模型的训练和测试阶段, 部分重要的参数设置如表 3 所示. 其中初始学习率为 1.0, 学习率衰减指数为 0.75, 且每经过 1 000 次迭代之后就将学习率置为之前的 0.75 倍, 最大迭代总次数为 750 000 次; 输入特征维度为 256, 与标记向量的特征维度相同; 为了避免模型过拟合, 设置的神经网络单元的随机失活率为 0.2; 编码器网络由 4 个双向单层循环神经网络构成, 解码器网络由 1 个单向单层循环神经网络构成, 默认网络单元为 GRU 单元; 训练和测试时的最小批次数量均为 128; 选用 Adagrad 作为优化器, 其初始累加器值为 0.1; 循环神经网络单元随机均匀初始化的幅度为 0.02; 最终通过设置 is_copy 和 is_coverage 两个参数来决定是否引入 Copy 机制和 Coverage 机制, 方便进行两种机制的消融实验. 需要注意的是, Coverage 机制要在模型训练的最后阶段再加入, 其约占总训练时间的 1%, 如果从刚开始训练时就加入 Coverage 机制反而会影响模型的训练效果.

表 3　模型训练和测试阶段的参数设置

参数名称	设置数值	参数说明
learning_rate	1.0	初始学习率
lr_decay	0.75	学习率衰减指数
lr_decay_steps	1 000	学习率衰减迭代轮数
max_iterations	750 000	最大迭代次数
input_size	256	输入特征维度
dropout	0.2	随机失活率
num_encoder_layers	4	编码器网络个数
num_decoder_layers	1	解码器网络个数
train_batch_size	128	训练时的最小批次
test_batch_size	128	测试时的最小批次
adagrad_init_acc	0.1	Adagrad的初始累加器值
rand_unif_init_mag	0.02	RNN单元随机均匀初始化的幅度
is_copy	True	是否使用Copy机制
is_coverage	True	是否使用Coverage机制
lr_coverage	0.15	加入Coverage机制后的学习率

2.6 实验结果分析

2.6.1 一致性检查的对比实验

函数名一致性检查基准方法的对比实验结果如表 4 所示, 相较于 Liu 等人的方法、MNire 和 DeepName 这 3 种基准方法, DMName 方法在精确率方面分别提高了 22.54%、14.42% 和 1.84%, 最终达到了 74.14%; 在召回率方面分别提高了 12.79%、5.36% 和 2.22%, 最终达到了 93.37%; 在 $F1$ 值方面分别提高了 19.74%、11.49% 和 2.01%, 最终达到了 82.65%; 在正确率方面分别提高了 21.34%、12.15% 和 1.93%, 最终达到了 76.04%.

对实验结果分析后发现, 相较于本文的 DMName 方法, 在 Liu 等人的方法和 MNire 方法预测错误的函数样本中, 存在十分相似的函数体代码 (内部上下文), 而其对应的真实函数名却不同, 严重影响了以相似度计算为主的两种方法的判断, 导致其各项指标都相对较低, 而 DMName 方法引入了多个不同的上下文信息, 可以对这些相似函数体的函数名进行区分. 此外, 经过对 DeepName 方法预测错误的函数样本进行检查, 发现某些错误样本的实际函数名的某个单词是上下文信息中单词的其他形式, 例如在实际函数名“getParameters”中, DeepName 预测的函数名为“getParameter”, 因为在上下文信息中只有其单数形式“Parameter”出现, 而正确的函数名中使用的却是复数形式,

对于此类情况, 本文的 DMName 方法使用 FastText 的字符级 *n*-gram 向量作为词嵌入, 有效地解决了此类不同单词标记形式表征的 OOV 问题.

表 4　函数名一致性检查基准方法的对比实验结果 (%)

指标	Liu等人的方法[24]	MNire[25]	DeepName[26]	DMName
精确率	51.60	59.72	72.30	74.14
召回率	80.58	88.01	91.15	93.3
*F*1值	62.91	71.16	80.64	82.65
正确率	54.70	63.89	74.11	76.04

2.6.2　名称推荐的对比实验

函数名推荐基准方法的对比实验结果如表 5 所示, 相较于 code2vec、MNire 和 DeepName 这 3 种基准方法, DMName 方法在精确率方面分别提高了 14.32%、8.15% 和 2.34%, 最终达到了 74.54%; 在召回率方面分别提高了 20.43%、10.87% 和 1.59%, 最终达到了 72.90%; 在 *F*1 值方面分别提高了 17.63%、9.57% 和 1.96%, 最终达到了 73.71%; 在完全匹配率方面分别提高了 22.53%、6.49% 和 0.72%, 最终达到了 43.63%.

表 5　函数名推荐基准方法的对比实验结果 (%)

指标	code2vec[31]	MNire[25]	DeepName[26]	DMName
精确率	60.22	66.39	72.20	74.54
召回率	52.47	62.03	71.31	72.90
*F*1值	56.08	64.14	71.75	73.71
完全匹配率	21.10	37.14	42.91	43.63

对实验结果分析后发现, 相较于本文的 DMName 方法, code2vec 方法要求两个函数代码具有相似的 AST 结构才能进行名称推荐, 这种严苛的相似性规则导致其推荐效果较差, *F*1 值偏低, 而且 AST 结构路径无法推测代码实体的实际编程顺序, 使其预测的函数名的标记顺序混乱, 能够完全匹配的函数名样本数量少, 导致其完全匹配率指标表现极差; MNire 方法所用表征形式涵盖的函数上下文信息不够全面, 没有办法挖掘其交互上下文和兄弟上下文的特征, 从而导致其各项指标上的表现平平; DeepName 方法的各项指标的表现与 DMName 方法十分接近, 但由于其无法从字符级层面预测 OOV 的单词标记, 导致其指标的提升存在瓶颈, 而且对于具有长度较长的函数名, 其预测的函数名中依然存在重复的单词标记. 总体来说, DMName 方法在各项指标上都有相对卓越的表现, 而且与最先进的 DeepName 方法相比, 在一定程度上突破了其所存在的性能瓶颈.

本文还对不同方法在模型训练和函数名预测时的平均时间开销进行了统计, 结果如表 6 所示. 需要说明的是, 由于 code2vec 方法需要以 AST 结构进行训练, 因此与本文中所采用的表征形式不符, 实验中仅使用其已训练好的开源模型进行预测, 没有具体的训练时间. 从训练时间开销来看, 由于 MNire 方法模型结构相对简单, 因此其训练时间最短, 仅为 43.70 h, 而本文所提出的 DMName 方法由于加入 Coverage 机制的缘故, 训练时间开销较大, 为 53.14 h. 从平均预测时间开销来看, code2vec 方法的平均预测时间耗时最短, 仅为 36.11 ms, 而 DMName 方法以 68.09 ms 的平均预测时间次之, 且与 DeepName 方法的平均预测时间开销的差距很小. 虽然 DMName 方法的训练时间较长, 但当模型训练完成后, 调用模型进行函数名预测的耗时大大缩短, 也进一步说明了本文所提 DMName 方法在实际生产过程中的可用性.

此外, 我们还对测试集中不同函数长度下的精确率和召回率的变化情况进行了分析, 其结果如图 4 所示. DMName 方法在预测函数名含有 1–25 个字符个数范围内的表现效果良好, 在所字符数量大于 25 的函数名中, DMName 方法的各项指标有相对明显的下降, 准确率和召回率分别降至 44.92% 和 42.03%, 与预期结果一样, 真实的函数名越长, 模型预测的效果越差.

然而, 在对实验结果中预测错误的样本集进行分析的过程中, 我们发现 DMName 方法除了在预测的真实函数

名过长的样本上表现不佳外, 部分预测错误样本中还存在注释表述有误、注释标记词数量占样本所有标记词的数量比例过大的情况. 此类样本数量大约占预测错误样本总数的 13.1%, 也进一步反映了错误的注释信息会对 DMName 方法的性能造成一定的不良影响, 且影响程度取决于该数据样本中错误注释信息标记词占所有标记词的比例.

表 6　不同方法的模型训练和预测时间对比

方法	训练时间 (h)	预测时间 (ms)
code2vec[31]	—	36.11
MNire[25]	43.70	82.29
DeepName[26]	51.25	70.33
DMName	53.14	68.09

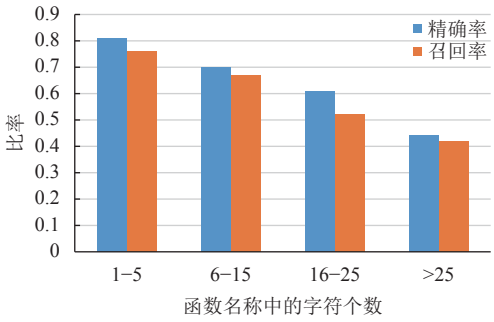

图 4　测试集中不同函数长度下的预测结果

此外, 我们还发现部分预测错误样本存在上下文信息标记词数量过少 (标记词数量少于 30) 和标记词数量过多 (标记词数量大于 300) 的情况, 这两种情况的样本数量比例分别占预测错误样本总数的 17.4% 和 20.8%, 在上下文信息标记词过少的样本中, DMName 方法很难根据有限的特征词进行有效的函数名推荐, 而在上下文信息标记词过多的样本中, 由于词嵌入后得到的向量序列长度受模型输入向量序列最大长度的限制, 因此在截断后会丢失某些重要的有效特征, 从而对 DMName 方法的推荐准确性产生影响. 因此, DMName 方法所训练模型的实际效果很大程度上依赖于训练和验证数据集的质量, 在保障数据集质量和时效性以及设置合适参数的情况下, DMName 方法才能在函数名一致性检查及推荐任务中表现除更优越的性能.

2.6.3　不同机制的消融实验

本文的重要研究内容就是解决目前基于机器学习的方法所存在的两个问题, 即 OOV 问题和解码器重复解码问题, 因此引入了 Copy 机制和 Coverage 机制来解决该问题, 通过对两种机制进行模型的消融实验, 来确定两者对模型效果的影响程度, 并判断两种机制的引用是否有效缓解了上述问题.

在函数名一致性检查任务中, 两种不同机制的消融实验结果如表 7 所示. 以 seq2seq 框架+Attention 机制为基准, 精确率在引入 Copy 机制后提高了 4.94%, 在此基础上引入 Coverage 机制则又提高了 2.59%; 召回率在引入 Copy 机制后提高了 2.28%, 在此基础上引入 Coverage 机制则又提高了 1.75%; $F1$ 值在引入 Copy 机制后提高了 4.03%, 在此基础上引入 Coverage 机制则又提高了 2.30%; 正确率在引入 Copy 机制后提高了 4.37%, 在此基础上引入 Coverage 机制则又提高了 2.68%.

在函数名推荐任务中, 两种不同机制的消融实验结果如表 8 所示. 以 seq2seq 框架+Attention 机制为基准, 精确率在引入 Copy 机制后提高了 2.93%, 在此基础上引入 Coverage 机制则又提高了 1.35%; 召回率在引入 Copy 机制后提高了 2.38%, 在此基础上引入 Coverage 机制则又提高了 1.06%; $F1$ 值在引入 Copy 机制后提高了 2.65%, 在此基础上引入 Coverage 机制则又提高了 1.20%; 完全匹配率在引入 Copy 机制后提高了 2.53%, 在此基础上引入 Coverage 机制则又提高了 2.41%.

对实验结果分析后发现, Copy 机制的引入对两类任务实际预测结果的影响程度较大, 相比之下, Coverage 机制的引入对结果的影响程度相对较小, 而通过对引入 Copy 机制前后的测试集预测结果差异样本进行分析, 我们发现 Copy 机制的引入对于那些未在传统词表中出现, 而且输入序列中出现的标记的函数名预测结果更加准确, 有效缓解了未引入 Copy 机制时所存在的 OOV 问题; 通过对引入 Coverage 机制前后的测试集预测结果差异样本进行分析, 我们发现 Coverage 机制的引入对于那些标记数量较多的函数名, 不会对相同的标记进行重复预测输出, 有效缓解了未引入 Coverage 机制时所存在的重复解码问题.

图 5 是某个目标函数源码的部分上下文信息, 该源码来源于服务云计算平台 Apache CloudStack 开源项目, 正确的目标函数名为“getDefaultPublicNetworkFQN”. 图 6 是对图 5 中目标函数源码的概率前 10 的函数名预测排名.

当未加入两种机制时, 其中最接近正确名称的“getDefaultPublicNetworkUNK”位于第 4 条, 由于用于预测的词表为包含 UNK 词的常规词表, 因此在多个预测名称结果中包含了 OOV 的 UNK 词, 第 7 条中的“Default”标记词也被二次重复预测. 当加入 Copy 机制后, 用于预测的词表则转换为常规词表 (清除 UNK 词) 与输入序列词表的去重并集, 预测结果中的 UNK 词被其他特征词代替, 且不同标记词的预测概率也发生了变化, 此时模型已经能够预测出正确的“getDefaultPublicNetworkFQN”函数名, 但二次重复预测的“Default”标记词现象仍然存在. 当继续加入 Coverage 机制后, “Default”标记词在第 1 次输出后其注意力权重被惩罚, 后续预测时由其他标记词代替, 也进一步证实了 Coverage 机制的作用.

表 7　函数名一致性检查不同机制的消融实验结果 (%)

机制	精确率	召回率	*F*1值	正确率
seq2seq+Attention	66.61	89.34	76.32	68.99
seq2seq+Attention+Copy	71.55	91.62	80.35	73.36
seq2seq+Attention+Copy +Coverage (DMName)	74.14	93.37	82.65	76.04

表 8　函数名推荐不同机制的消融实验结果 (%)

机制	精确率	召回率	*F*1值	完全匹配率
seq2seq+Attention	70.26	69.46	69.86	38.69
seq2seq+Attention+Copy	73.19	71.84	72.51	41.22
seq2seq+Attention+Copy +Coverage (DMName)	74.54	72.90	73.71	43.63

```
1  package org.apache.cloudstack.network.contrail.management;
2  ...
3  public class ContrailManagerImpl extends ManagerBase implements ContrailManager {
4  ...
5      // Method Name: getDefaultPublicNetworkFQN
6      public String XXXXXXXXXXXXXXXX() {
7          String name = VNC_ROOT_DOMAIN + ":" + VNC_DEFAULT_PROJECT + ":" + "__default_Public__";
8          return name;
9      }
10     ...
11     public net.juniper.contrail.api.types.Project getDefaultVncProject() throws IOException {
12         net.juniper.contrail.api.types.Project project = null;
13         project = (net.juniper.contrail.api.types.Project)_api.findByFQN(net.juniper.contrail.api.types.Project.class,
VNC_ROOT_DOMAIN + ":" + VNC_DEFAULT_PROJECT);
14         return project;
15     }
16     ...
17     public String getFQN(Network net) {
18         // domain, project, name
19         String fqname = getDomainName(net.getDomainId());
20         fqname += ":" + getProjectName(net.getAccountId()) + ":";
21         return fqname + getCanonicalName(net);
22     }
23     ...
24     public boolean isSystemDefaultNetwork(VirtualNetwork vnet) {
25         ...
26     }
27 ...
```

图 5　函数名推荐示例

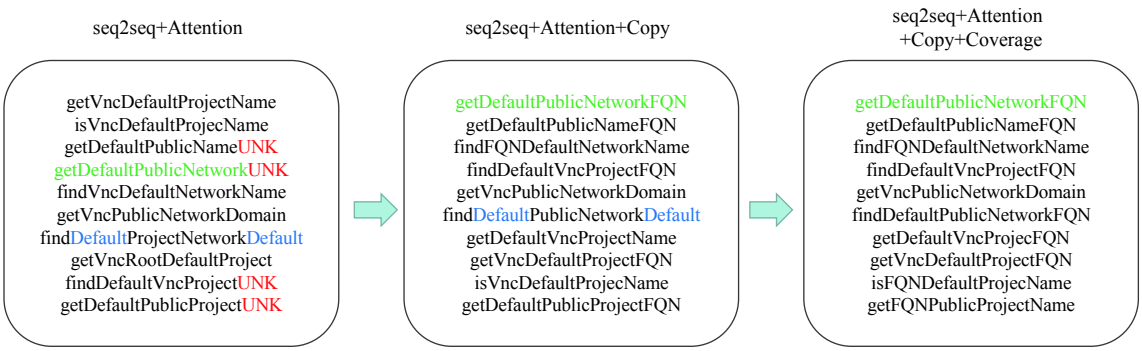

图 6　前 10 的函数名推荐结果

3 讨 论

3.1 DMName 方法在实际项目中的应用

为了探索 DMName 方法在实际开源项目中的表现效果, 我们在 GitHub 开源社区中选择了中等规模的 lancia[27] 开源项目进行验证, lancia 在 GitHub 中的 Star 数量超过 100, 支持任何 URL 或 HTML 内容转换为 PDF 文件或图像的渲染功能. 具体来说, 在获取其 v1.3.1 版本源码的基础上, 从中共分离出 164 个类包含 235 个函数, 按照本文的数据处理方法, 首先消除重写和重载函数, 保障目标函数样本不属于重写和重载函数的类别, 进而确定可以进行函数名一致性检查和预测的目标函数数量及范围, 然后对源码中非空函数的函数体与函数名分离, 构建函数体源码的 4 种上下文信息, 需要说明的是, 在构建每个目标函数所对应的 4 种上下文信息的过程中, 并不会删除与目标函数相关的重写和重载函数, 因此不会存在丢失大量上下文信息的情况. 在完成源码数据清洗后, 利用训练好的 DMName 方法模型对其进行函数名一致性检查, 若检测结果为不一致, 则进一步输出模型预测的名称, 最终经过人工检查筛选后, 共获得了 16 个函数名不一致的问题 (https://github.com/HughTang/resultOfLancia/blob/main/lancia.txt).

我们将这些问题通过邮件的形式反馈给开发人员, 最终被开发人员所认可, 开发人员表示 DMName 方法所推荐的函数命名建议十分中肯, 修改后有利于提升程序的可读性, 但 lancia 项目的部分 API 设计方法依赖于 Node.js 的第三方 Puppeteer 库设计, 因此函数名在更改时可能会对某些程序模块的第三方库引用逻辑造成影响, 其整体维护成本较高, 因此需要在后续的版本迭代或重构中才能实施, 这也从侧面反映出函数名修改所带来的维护成本会受到软件实际规模、开发阶段和第三方库的影响. 但开发人员对函数名推荐结果的认可进一步证实了 DMName 方法推荐的函数名的有效性, 这表明 DMName 方法在实际开源项目中对不一致函数名的检测以及函数名推荐方面是有用的.

3.2 有效性威胁因素分析

本文所提出的 DMName 方法虽然在实验验证和实际项目中取得了良好的效果, 但其有效性仍然受到以下 3 个方面有效性威胁因素的影响.

(1) DMName 方法所使用的训练和测试数据集主要来源于 Liu 等人[24]所收集的函数名一致性检查数据集和 Nguyen 等人[25]所收集的函数名推荐数据集, 因此训练所得模型的实际效果很大程度上依赖于该数据集的质量, 且未来可能会受到数据集时效性的影响. 数据集中只涉及 Java 语言, 会受到编程语言单一的限制, 因此很难应用和推广到其他编程语言项目中, 未来可以尝试扩充多种编程语言的数据集, 从而进一步拓展 DMName 方法的适用范围.

(2) DMName 方法无法对整个项目源码进行分析, 因此很难做到对相同名称的重载/重写函数进行准确的函数名推荐. 此外, 在实际的模型调参过程中, 我们选用网格搜索技术实现了超参数调参, 直接获取了性能最好的一组参数配置, 因此在实验分析中并没有深入探究参数变化对模型效果的影响, 可能会存在更优的参数选择, 从而进一步提升模型的性能.

(3) 由于实际的实验研究时间有限, 我们仅将 DMName 方法应用于单个开源项目中, 未来需要将该方法应用于更多的开源项目中, 搜集开发人员的评估和反馈信息, 进一步确认和改进我们所提出的 DMName 方法的有效性.

4 相关工作

研究人员曾提出一系列方法来检查函数命名的不一致问题, 并试图为其推荐更为合适的函数名, 从而提高函数名的命名质量, 相关工作主要包括基于信息检索的方法和基于机器学习的方法.

4.1 基于信息检索的方法

研究者认为相似的函数体、函数返回类型及调用参数应该具有相似的函数名, 因此提出了基于信息检索的方法, 其本质上是一种基于代码相似性理论的方式.

Jiang 等人[32]提出了一种基于代码搜索的函数名一致性检查及推荐方法, 通过在已构建的代码数据库中检索具有相似返回类型和参数的函数名, 并以此为依据推导出对应的函数名.

Høst 等人[18]提出了一种基于返回类型、控制流和参数信息来自动推断函数命名规则的技术, 其提出了规则冲突 (rule violation) 的概念, 即函数名与具体功能实现之间存在冲突. 为了解决该规则冲突, 他们根据候选短语语料库中语义特征的等级, 以及不规范短语与候选短语之间的语义距离来对候选列表进行排序, 从而过滤相关函数形成中的短语是否存在违反规则的情况, 以此来推断当前函数名是否存在不一致问题.

Deissenboeck 等人[3]认为函数命名规则的制定对于加强程序的一致性非常重要, 因此基于函数的命名规则与推荐名称的映射关系创建了一个函数名推荐模型, 可以用于创建清晰准确的函数名. 此外, 他们还在模型中构建了对应的名称词典, 从而确保程序的各类标识符名称能够在软件生命周期中保持一致.

高原等人[33]提出了一种基于代码库和特征匹配的函数名推荐方法, 其在构建开源软件函数库的基础上, 从函数库中检索与被推荐函数相似的函数体, 对检索函数名结果进行二次解析, 通过解析获得的标注词条, 并将其与被推荐函数体中的特征代码建立映射关系, 从而自动生成合适的函数名推荐.

Liu 等人[24]提出了一种基于相似度计算的函数名一致性检查方法, 他们将函数名和函数体分别嵌入到带有段落向量[34]和卷积神经网络[35]的各自的向量空间中, 然后检索名称向量空间和实体向量空间中与目标函数相近的函数名集, 若其交集为空集, 则说明该函数名不符合一致性.

基于信息检索的方法往往具有简单易用且计算成本低的优点, 但其缺点也很明显, 其构建函数检索数据库的人力成本很高, 检索效率也会随数据库规模的不断增大而降低, 由于函数检索数据库不可能涵盖所有情境下的逻辑代码, 因此无法生成不在数据库中的函数名, 且制定的检索规则也会受到时效性的不利影响.

4.2 基于机器学习的方法

基于机器学习的方法的通用思路是将函数的相关结构化信息映射到向量空间中, 利用传统机器学习模型或注意力神经网络对其进行子序列抽取, 最终将子序列中的关键词按照一定顺序组成函数名.

Allamanis 等人[36]通过引入对数双线性神经语言模型 (log-bilinear neural language model) 来解决函数命名问题, 该模型包括在源代码中捕获远距离上下文的特征函数以及可以预测新词 (即未出现在训练集中的名称) 的子序列模型. 该模型将每个标记嵌入到一个高维连续空间中, 并推荐与函数功能最相符的函数名. Allamanis 等人[37]后来将函数命名视为源代码的极端摘要 (extreme summary) 问题, 其中函数名被视为函数体的摘要. 为了生成代码摘要, 他们引入了一个注意力神经网络, 该网络可以对函数体内的输入标记进行卷积, 使得模型可以学习到源代码高级功能特征, 以此来检查函数名的一致性并对函数名进行相关推荐.

Alon 等人[38]提出了一种通用的基于路径的函数表示方法 Path-Rep, 其主要思想是利用 AST 中的叶子和与相应叶子相关联的标识符之间的路径袋来表示函数名, 这使得机器学习模型能够有效利用源代码的结构化特性, 而不是将其视为一个简单的标记序列.

Alon 等人[31,39]先后提出了 code2vec 和 code2seq 方法, 两种方法本质上都是通过将抽象语法树 (abstract syntax tree, AST) 的路径袋与注意力网络结合, 进一步将函数体表示为分布式向量[40]. 注意力机制为函数中的每个 AST 路径计算加权注意力值, 并通过加权求和将它们聚合为单个向量来表示函数体, 其中共享相似 AST 结构的函数向量在连续分布空间中彼此接近, 从而可以有效捕获函数之间以及函数名之间的语义相似性.

Nguyen 等人[25]对大型开源数据集中函数命名的性质进行了实证研究, 发现大部分函数名中包含的关键词都可以在给定函数的主体、接口 (函数的参数类型和返回类型) 和封闭类这 3 种上下文信息中找到, 于是他们提出了一种基于 seq2seq 模型的函数名一致性检查及推荐工具 MNire, 该工具通过构建函数的 3 种上下文信息组成语料库, 并借助注意力机制赋予每个上下文信息中关键词的权重, 以此来生成合适的函数名.

Li 等人[26]在参考 MNire 的基础上提出了 DeepName 工具. 其拓展了内部上下文、交互上下文、兄弟上下文和封闭上下文等 4 种不同的函数代码表征形式, 并在 seq2seq 模型的编码器中加入了 CopyNet 中提到的 Copy 机制, 又提出了一种新的 Non-copy 机制, 通过直接拷贝原序列中的关键词来替换不在训练数据中的单词名称, 且在

实际项目中对 DeepName 进行了有效性验证.

Liu 等人[41]提出了一种基于全局 Transformer 的函数名推荐方法 GTNM, 为了更加全面的挖掘函数名预测的有效特征信息, 他们使用了局部上下文、特定项目上下文和文档上下文等 3 种不同的目标函数表征形式, 其中拓展的文档上下文信息主要是指目标函数所对应 API 文档或注释中的功能描述信息和在项目中的实际用途, 并在 seq2seq 架构中引入了注意力机制来赋予各个上下文信息的权重比例. 他们在 Nguyen 等人[25]所收集处理的数据集中进行了训练和评估, 结果证明其提出的 GTNM 方法在性能方面优于 DeepName 基准方法, 但并没有解决过程中存在的 OOV 问题和解码器端的重复解码问题.

DeepName 方法中 Copy 机制的使用为本文解决 OOV 问题的研究方向提供了有效参考, 但 DeepName 方法和 GTNM 方法都并未解决因注意力机制的引入而带来的解码器重复解码问题. 此外, 本文的 DMName 方法同时利用 FastText 词嵌入方法可以叠加单词字符级的 *n*-gram 向量的优势, 进一步缓解了模型训练过程中存在的 OOV 问题, 在实验验证中取得了更高的 *F*1 值和准确率.

基于机器学习的方法的优点在于其推荐准确率高, 缺点是很多研究并未充分考虑函数的上下文信息, 而且一些在词表之外的单词会影响模型的理解能力, 使模型的生成能力受限, 从而引发 OOV 问题. 此外, 注意力机制的引入容易导致同一单词的重复解码, 造成单个关键词在名称推荐中不停循环出现的问题.

5 结 论

函数名是开发人员理解程序或 API 行为的最为直观的重要信息, 简洁且有意义的函数名代表了聚合行为抽象的基石, 对于程序代码的可理解性尤为重要. 本文提出了一种基于 seq2seq 模型的函数名一致性检查及智能推荐技术方案, 最终的方案被命名为 DMName, 其使用 GRU 网络作为编解码器的神经网络单元设计, 同时在利用 FastText 词嵌入技术对每个函数的 4 种不同上下文信息完成向量表示的基础上, 借助注意力机制对不同的输入词分配不同的注意力权重, 在利用 FastText 词嵌入方法可以叠加单词字符级的 *n*-gram 向量的优势以及 Copy 机制的基础上, 解决了模型存在的 OOV 问题, 同时引入 Coverage 机制解决了解码器端的重复解码问题, 在解码器端获得每个函数名所对应的预测结果向量序列, 然后分别借助双通道 CNN 分类器和向量序列概率分布映射, 最终完成对函数名的一致性检查及函数名推荐任务.

DMName 方法在各项指标上都取得了良好的实验结果, 并得到了现实开发人员的认可, 本文未来的研究方向依然可以从以下方面进行拓展: (1) 由于数据集中函数源码的编程语言的限制, DMName 方法目前只能对 Java 语言编写的代码进行函数名的一致性检查和名称推荐, 未来可以通过扩充多种编程语言的数据集, 使得 DMName 方法能够运用到不同编程语言的代码中, 从而拓展 DMName 方法的适用范围; (2) 由于 DMName 方法不是直接对整个项目的源码进行分析, 因此很难对多个相同名称的重载/重写函数进行有效的名称推荐, 未来可以对 DMName 方法相关的功能模块进行拓展, 研究出特定于重载/重写函数的名称一致性检查和推荐方法.

References:

[1] Johnson P. Don’t go into programming if you don’t have a good thesaurus. 2013. https://www.itworld.com/article/2833265/cloud-computing/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html

[2] Johnson P. The 9 hardest things programmers have to do. 2013. https://www.cio.com/article/220287/the-9-hardest-things-programmers-have-to-do.html

[3] Deissenboeck F, Pizka M. Concise and consistent naming. Software Quality Journal, 2006, 14(3): 261–282. [doi: 10.1007/s11219-006-9219-1]

[4] Gethers M, Savage T, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A. CodeTopics: Which topic am I coding now? In: Proc. of the 33rd Int’l Conf. on Software Engineering. Honolulu: IEEE, 2011. 1034–1036. [doi: 10.1145/1985793.1985988]

[5] Bavota G, Oliveto R, Gethers M, Poshyvanyk D, de Lucia A. Methodbook: Recommending move method refactorings via relational topic models. IEEE Trans. on Software Engineering, 2014, 40(7): 671–694. [doi: 10.1109/TSE.2013.60]

[6] Deissenboeck F, Pizka M. Concise and consistent naming: Ten years later. In: Proc. of the 23rd IEEE Int ’l Conf. on Program

Comprehension. Florence: IEEE, 2015. 3. [doi: 10.1109/ICPC.2015.9]

[7] Liblit B, Begel A, Sweetser E. Cognitive perspectives on the role of naming in computer programs. In: Proc. of the 18th Annual Workshop of the Psychology of Programming Interest Group. Brighton: Psychology of Programming Interest Group, 2006. 53–67.

[8] Lawrie D, Morrell C, Feild H, Binkley D. What's in a name? A study of identifiers. In: Proc. of the14th IEEE Int'l Conf. on Program Comprehension. Athens: IEEE, 2006. 3–12. [doi: 10.1109/ICPC.2006.51]

[9] Arnaoudova V, Eshkevari LM, Di Penta M, Oliveto R, Antoniol G, Guéhéneuc YG. REPENT: Analyzing the nature of identifier renamings. IEEE Trans. on Software Engineering, 2014, 40(5): 502–532. [doi: 10.1109/TSE.2014.2312942]

[10] Arnaoudova V, Di Penta M, Antoniol G. Linguistic antipatterns: What they are and how developers perceive them. Empirical Software Engineering, 2016, 21(1): 104–158. [doi: 10.1007/s10664-014-9350-8]

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

[12] Hofmeister J, Siegmund J, Holt DV. Shorter identifier names take longer to comprehend. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Klagenfurt: IEEE, 2017. 217–227. [doi: 10.1109/SANER.2017.7884623]

[13] Butler S, Wermelinger M, Yu YJ, Sharp H. Relating identifier naming flaws and code quality: An empirical study. In: Proc. of the 16th Working Conf. on Reverse Engineering. Lille: IEEE, 2009. 31–35. [doi: 10.1109/WCRE.2009.50]

[14] Abebe SL, Haiduc S, Tonella P, Marcus A. The effect of lexicon bad smells on concept location in source code. In: Proc. of the 11th IEEE Int'l Conf. on Source Code Analysis and Manipulation. Williamsburg: IEEE, 2011. 125–134. [doi: 10.1109/SCAM.2011.18]

[15] Abebe SL, Arnaoudova V, Tonella P, Antoniol G, Guéhéneuc YG. Can lexicon bad smells improve fault prediction? In: Proc. of the 19th Working Conf. on Reverse Engineering. Kingston: IEEE, 2012. 235–244. [doi: 10.1109/WCRE.2012.33]

[16] Hovemeyer D, Pugh W. Finding bugs is easy. ACM SIGPLAN Notices, 2004, 39(12): 92–106. [doi: 10.1145/1052883.1052895]

[17] Martin RC. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River: Pearson Education, 2008.

[18] Høst EW, Østvold BM. Debugging method names. In: Proc. of the 23rd European Conf. on Object-oriented Programming. Genoa: Springer, 2009. 294–317. [doi: 10.1007/978-3-642-03013-0_14]

[19] Von Mayrhauser A, Vans AM. Program understanding behavior during debugging of large scale software. In: Proc. of the 7th Workshop on Empirical Studies of Programmers. Alexandria: ACM, 1997. 157−179. [doi: 10.1145/266399.266414]

[20] Corbi TA. Program understanding: Challenge for the 1990s. IBM Systems Journal, 1989, 28(2): 294–306. [doi: 10.1147/sj.282.0294]

[21] Takang AA, Grubb PA, Macredie RD. The effects of comments and identifier names on program comprehensibility: An experiential study. Journal of Program Languages, 1996, 4(3): 143–167.

[22] Hendrix D, Cross JH, Maghsoodloo S. The effectiveness of control structure diagrams in source code comprehension activities. IEEE Trans. on Software Engineering, 2002, 28(5): 463–477. [doi: 10.1109/TSE.2002.1000450]

[23] Joulin A, Grave E, Bojanowski P, Mikolov T. Bag of tricks for efficient text classification. In: Proc. of the 15th Conf. of the European Chapter of the Association for Computational Linguistics. Valencia: ACL, 2017. 427–431.

[24] Liu K, Kim D, Bissyandé TF, Kim T, Kim K, Koyuncu A, Kim S, Le Traon Y. Learning to spot and refactor inconsistent method names. In: Proc. of the 41st Int'l Conf. on Software Engineering. Montreal: ACM, 2019. 1–12. [doi: 10.1109/ICSE.2019.00019]

[25] Nguyen S, Phan H, Le T, Nguyen TN. Suggesting natural method names to check name consistencies. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1372–1384. [doi: 10.1145/3377811.3380926]

[26] Li Y, Wang SH, Nguyen TN. A context-based automated approach for method name consistency checking and suggestion. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 574–586. [doi: 10.1109/ICSE43902.2021.00060]

[27] Aoju. lancia. 2022. https://github.com/aoju/lancia

[28] See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. In: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics. Vancouver: ACL, 2017. 1073–1083. [doi: 10.18653/v1/P17-1099]

[29] Gu JT, Lu ZD, Li H, Li VOK. Incorporating copying mechanism in sequence-to-sequence learning. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. Berlin: ACL, 2016. 1631–1640. [doi: 10.18653/v1/P16-1154]

[30] Soot-Oss. Soot. 2023. https://github.com/soot-oss/soot

[31] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 40. [doi: 10.1145/3290353]

[32] Jiang L, Liu H, Jiang H. Machine learning based recommendation of method names: How far are we. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 602–614. [doi: 10.1109/ASE.2019.00062]

[33] Gao Y, Liu H, Fan XZ, Niu ZD. Method name recommendation based on source code depository and feature matching. Ruan Jian Xue Bao/Journal of Software, 2015, 26(12): 3062−3074 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4817.htm [doi:

10.13328/j.cnki.jos.004817]

[34] Le Q, Mikolov T. Distributed representations of sentences and documents. In: Proc. of the 31st Int'l Conf. on Machine Learning. Beijing: JMLR.org, 2014. 1188–1196.

[35] Matsugu M, Mori K, Mitari Y, Kaneda Y. Subject independent facial expression recognition with robust face detection using a convolutional neural network. Neural Networks, 2003, 16(5–6): 555–559. [doi: 10.1016/S0893-6080(03)00115-1]

[36] Allamanis M, Barr ET, Bird C, Sutton C. Suggesting accurate method and class names. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 38–49. [doi: 10.1145/2786805.2786849]

[37] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proc. of the 33rd Int'l Conf. on Machine Learning. New York: PMLR, 2016. 2091–2100.

[38] Alon U, Zilberstein M, Levy O, Yahav E. A general path-based representation for predicting program properties. ACM SIGPLAN Notices, 2018, 53(4): 404–419. [doi: 10.1145/3296979.3192412]

[39] Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. In: Proc. of the 7th Int'l Conf. on Learning Representations. New Orleans: ICLR, 2019.

[40] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: Proc. of the 3rd Int'l Conf. on Learning Representations. San Diego: ICLR, 2015.

[41] Liu F, Li G, Fu ZY, Lu S, Hao YY, Jin Z. Learning to recommend method names with global context. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1294–1306. [doi: 10.1145/3510003.3510154]

附中文参考文献:

[33] 高原, 刘辉, 樊孝忠, 牛振东. 基于代码库和特征匹配的函数名称推荐方法. 软件学报, 2015, 26(12): 3062−3074. http://www.jos.org.cn/1000-9825/4817.htm [doi: 10.13328/j.cnki.jos.004817]

郑炜(1975－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件测试, 软件安全, 软件仓库挖掘.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件缺陷预测, 软件缺陷定位, 回归测试, 组合测试.

唐辉(1997－), 男, 硕士, 主要研究领域为自然语言处理, 软件测试.

张永杰(2000－), 女, 硕士生, 主要研究领域为知识图谱, 软件漏洞分析.