软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(11):4975−4989 [doi: 10.13328/j.cnki.jos.007362] [CSTR: 32375.14.jos.007362]

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

最长公共子序列嵌入支持下的代码相似性检测*

弓媛君[1], 黄建军[1], 游 伟[1], 石文昌[1], 梁 彬[1], 边 攀[2], 张 健[3]

[1](中国人民大学 信息学院, 北京 100872)

[2](华为技术有限公司, 北京 100085)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 黄建军, E-mail: hjj@ruc.edu.cn; 梁彬, E-mail: liangb@ruc.edu.cn

摘 要: 最长公共子序列 (longest common subsequence, LCS) 是一种衡量代码相似度的可行指标. 然而, 经典 LCS 算法的时间复杂度较高, 难以应对大型数据集, 并且, 由于代码文本序列中的词 (token) 本质为一种基于离散表示的编码, 直接使用 LCS 算法无法有效识别文本不同但语义相似的代码片段中的关键语义. 针对这两方面的不足, 提出一种面向 LCS 的嵌入方法, 将代码间的 LCS 计算转换为代码低维稠密嵌入向量间的数值运算, 并可以利用近似最近邻算法进一步加速其计算. 为此, 设计了一个可嵌入的基于 LCS 的距离度量方法, 实验证明这种代码度量在提取函数关键语义的表现上优于对比嵌入工具使用的基于文本的距离或基于树的距离. 同时, 为了在嵌入过程中有重点地保留代码的关键语义, 构建了两种损失函数和相应的训练集, 识别文本上不同但语义上相似的代码元素, 使模型在检测复杂代码克隆时有更好的表现. 实验证明了该方法拥有很强的可扩展性, 且其对复杂克隆的检测能力也保持在很高水平. 将该技术应用于相似缺陷的识别, 上报了 23 个未知缺陷, 这些缺陷已被开发人员在实际项目中确认, 其中有些复杂缺陷是难以被基于文本的 LCS 算法检出的.

关键词: 最长公共子序列 (LCS); 代码相似性检测; 代码嵌入; 缺陷检测; 克隆检测

中图法分类号: TP311

中文引用格式: 弓媛君, 黄建军, 游伟, 石文昌, 梁彬, 边攀, 张健. 最长公共子序列嵌入支持下的代码相似性检测. 软件学报, 2025, 36(11): 4975–4989. http://www.jos.org.cn/1000-9825/7362.htm

英文引用格式: Gong YJ, Huang JJ, You W, Shi WC, Liang B, Bian P, Zhang J. Code Similarity Detection Supported by Longest Common Subsequence Embedding. Ruan Jian Xue Bao/Journal of Software, 2025, 36(11): 4975–4989 (in Chinese). http://www.jos.org.cn/1000-9825/7362.htm

Code Similarity Detection Supported by Longest Common Subsequence Embedding

GONG Yuan-Jun[1], HUANG Jian-Jun[1], YOU Wei[1], SHI Wen-Chang[1], LIANG Bin[1], BIAN Pan[2], ZHANG Jian[3]

[1](School of Information, Renmin University of China, Beijing 100872, China)

[2](Huawei Technologies Co. Ltd., Beijing 100085, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: The longest common subsequence (LCS) is a practical metric for assessing code similarity. However, traditional LCS-based methods face challenges in scalability and in effectively capturing critical semantics for identifying code fragments that are textually different but semantically similar, due to their reliance on discrete representation-based token encoding. To address these limitations, this study proposes an LCS-oriented embedding method that encodes code into low-dimensional dense vectors, effectively capturing semantic information. This transformation enables the computationally expensive LCS calculation to be replaced with efficient vector arithmetic, further accelerated using an approximate nearest neighbor algorithm. To support this approach, an embeddable LCS-based distance metric

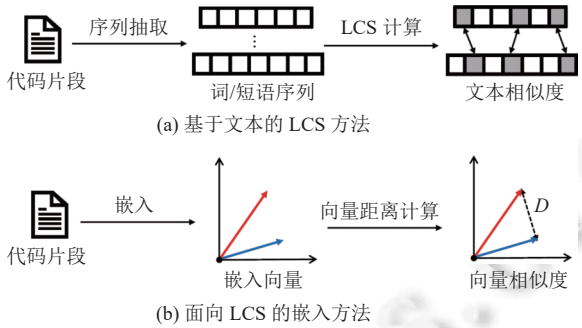
* 基金项目: 国家自然科学基金 (62272465, 62441206, 62002361, 62272464); CCF-华为胡杨林基金 (CCF-HuaweiSE202310)
收稿时间: 2023-11-06; 修改时间: 2024-08-07, 2024-10-19; 采用时间: 2024-11-22; jos 在线出版时间: 2025-05-22
CNKI 网络首发时间: 2025-05-23

is developed, as the original LCS metric is non-embeddable. Experimental results demonstrate that the proposed metric outperforms tree-based and literal similarity metrics in detecting complex code clones. In addition, two targeted loss functions and corresponding training datasets are designed to prioritize retaining critical semantics in the embedding process, allowing the model to identify textually different but semantically similar code elements. This improves performance in detecting complex code similarities. The proposed method demonstrates strong scalability and high accuracy in detecting complex clones. When applied to similar bug identification, it has reported 23 previously unknown bugs, all of which are confirmed by developers in real-world projects. Notably, several of these bugs are complex and challenging to detect using traditional LCS-based techniques.

Key words: longest common subsequence (LCS); code similarity detection; code embedding; bug detection; clone detection

代码克隆是软件工程中的一种常见现象, 相似代码广泛存在于工程内或多个工程间. 然而在很多情况下, 相似代码可能带来安全隐患, 尤其是相似的代码逻辑可能带来相似的潜在缺陷. 研究人员提出了许多识别相似代码片段的技术, 用于检测与某些已知缺陷代码相似的未知缺陷[1−7], 或用于检测可能侵犯知识产权的代码克隆[8−12]. 如何有效地衡量代码相似度已经成为一个重要的研究问题[13,14], 其中, 基于文本的相似性度量是一种自然且直接的选择[15−17], 在各个种类的文本相似性度量中, 最长公共子序列 (longest common subsequence, LCS) 是一种有效的度量, 这在最近的一项研究中得到了很好的验证[9].

在许多情况下, 两个相似的代码片段并不能完全匹配, 特别是在具有类似逻辑的缺陷代码中[1,2,18]. 在这种情况下, 两个片段之间的共同元素可能变得不连续. 两个序列的最长公共子序列是两个序列共有且长度最长的子序列, 其中, 子序列中的元素需在原序列中的前后关系相同, 但相邻元素不需要在原序列中相邻. 因此, 两个代码片段之间的最长公共子序列可以在一定程度上反映代码共有元素之间构成的相似代码逻辑, 同时忽略代码共有元素的不连续性. 现有的研究采用不同粒度的 LCS 算法来衡量代码相似度, 例如, NiCad[8]旨在通过两个代码各语句词 (token) 序列间的 LCS 长度来判定语句的相似性, NIL[9]则将 LCS 应用于 n 元词序列 (n-gram) 上. 我们在图 1(a) 中总结了现有的基于 LCS 的代码相似性度量的整体工作流程. 现有的基于 LCS 的代码相似性度量方法将源代码转换为序列, 然后计算每对序列之间基于 LCS 的相似度. 因为它们依赖于词或短语, 所以在本研究中我们称其为基于文本的 LCS 方法.



(a) 基于文本的 LCS 方法

(b) 面向 LCS 的嵌入方法

图 1 基于文本的 LCS 方法与面向 LCS 的嵌入方法

基于文本的 LCS 方法在各种基线测试中都表现良好, 然而, 此方法有两个缺点值得注意. 首先, 序列上的 LCS 算法对于大型代码库来说可扩展性较差. LCS 问题的时间和空间复杂度都是 O(mn), 其中 m 和 n 分别是两个输入序列的长度, 当有大量序列需要搜索时, LCS 的成本很高. 在实践中, 大型项目往往具有大量代码, 这意味着存在大量待搜索序列, 而现实世界中的代码搜索更可能涉及数以万计的项目. 我们的实证研究表明, 测量一个给定函数与 Linux 内核中所有函数的 LCS 相似度大约需要 8 min. 如果在某些场景下有很多查询序列, 例如检测历史缺陷的克隆, 那么成本将是不可接受的. LCS 计算可以通过过滤方案加速, 例如倒排索引机制[9]. 然而, 它可能会消耗大量的计算资源, 并且无法扩展到超大规模的代码库. 另外, 直接使用词或短语序列来表示代码片段本质上是一种离散表示 (local representation), 即使是两个语义相似的词或短语也会被视为完全不同的两个元素. 如图 2 表示 Linux 内核中的一对包含相似缺陷的函数, 它们分别在第 13 行和第 29 行调用两个具有相似语义的函数 platform_driver_

register 和 pci_register_driver. 然而, 它们在基于文本的 LCS 方法中被表示为不同的词或短语, 它们的语义相似性在 LCS 计算中将被忽略. 在后文中我们将此类代码元素叫做文本不同但语义相似的元素 (literally different but semantically similar, LDSS). 同时一些语义不突出的程序元素被完整保留在序列中 (后文中称为“弱语义元素”) 与其他语义丰富的程序元素处于同等地位, 它们可能是与缺陷无关的常见函数调用 (如时间函数或日志函数) 或是与函数逻辑关联较低的字符串 (如图 2 第 9 行与第 24 行中的输出字符串), 这些弱语义元素会占据公共子序列与函数序列长度的比例, 影响相似性度量. 基于文本的 LCS 方法对 LDSS 元素和弱语义元素的处理都使其无法识别图 2 中两个函数的相似性, 因此无法检测出 ena_init 中的缺陷.

```
1  //file: drivers/usb/host/u132-hcd.c
2  static int __init u132_hcd_init(void)
3  {
4      int retval;
5      u132_instances = 0;
6      u132_exiting = 0;
7      if (usb_disabled())
8          return -ENODEV;
9      printk(KERN_INFO "driver %s\n", hcd_name);
10     workqueue = create_singlethread_workqueue("u132");
11     if (!workqueue)
12         return -ENOMEM;
13     retval = platform_driver_register(&u132_platform_driver);
14
15     return retval;
16 }
17
18 //file: drivers/net/ethernet/amazon/ena/ena_netdev.c
19 static int __init ena_init(void)
20 {
21
22     ena_wq = create_singlethread_workqueue(DRV_MODULE_NAME);
23     if (!ena_wq) {
24         pr_err("Failed to create workqueue\n");
25         return -ENOMEM;
26     }
27
28
29     return pci_register_driver(&ena_pci_driver);
30 }
31
```

图 2　一对包含相似漏洞的相似函数

本文提出了一种面向 LCS 的嵌入方法来解决上述问题. 该方法的工作流程如图 1(b) 所示. 本方法中心思想是在代码上得到以语句为匹配元素、以语句序列作为匹配目标的基于 LCS 的相似度, 然而我们不是直接对语句序列进行 LCS 计算, 而是将函数的语句序列嵌入至低维稠密向量空间中, 这样, 语句序列上的 LCS 计算可以转化为函数嵌入向量上的欧氏距离计算, 拥有更好的时空性能. 此外, 可以引入近似最近邻算法[19]来进一步加速向量相似度计算, 使得基于 LCS 的代码相似性检测可以针对大型软件项目进行扩展, 并有效捕获关键语义.

本方法中首先遇到的挑战是原始 LCS 距离不满足可嵌入距离的要求, 也就是说, 我们不能直接使用 LCS 距离来训练嵌入模型. 可嵌入距离应满足非负性、对称性和三角不等式[20]. 由于 LCS 距离不满足三角不等式, 我们设计了一个面向 LCS 的可嵌入距离 D, 通过在距离计算时引入序列的长度来限制距离范围 (详细证明参见第 2.1 节). 为此我们设计了损失函数 L_{LCS} 来训练距离 D 的嵌入空间.

另外一个挑战是, 标准的 LCS 不能捕获序列中的 LDSS 元素, 因此会在检索中错失一些具有语义相似性的函数对. 为此我们另外设计了一个损失函数 L_{LDSS} 对模型进行有针对性的训练, 使包含 LDSS 的一对代码序列的嵌入距离略小于其标准 LCS 距离, 用这种方法强调序列中 LDSS 元素的作用. 为了有效识别 LDSS 元素, 我们使用 SentencePiece 分词器[21]来分割标识符, 使用子词级别的语言模型对函数调用进行编码, 并通过函数调用嵌入的相

似度判断其 LDSS 关系.

我们进行了广泛的实验来证明本方法的可扩展性和有效性. 结果表明, 我们的方法具有非常高的运行速度, 在 Linux 上进行一次查询 (计算一个查询与 330 000 个目标之间的相似性) 平均仅需约 0.004 s, 在超大规模的代码搜索中比最先进的基于 LCS 的工具 (即 NIL) 执行相似代码检测的速度快几十倍, 并且可以扩展到更大的代码库 (见后文第 3.2 节). 在基于匹配的缺陷检测中, 我们成功地从开源项目中检测到了 23 个以前未知的缺陷, 并且都已经得到了开发者的确认, 而对比工具会错过一些涉及 LDSS 元素的缺陷 (见后文第 3.3 节). 我们的方法还在 BigCloneBench 上显示出与最先进的克隆检测工具相当的性能 (见后文第 3.4 节). 我们将本文项目和实验结果开源至 https://github.com/nk011lcs/opensource.

本文的贡献如下.

1) 提出了一种面向 LCS 的嵌入方法来支持可扩展的代码搜索, 以高效的向量相似性计算取代代价高昂的基于文本的 LCS 计算, 使得我们的工具可以在超大规模代码库上实现很高的可扩展性.

2) 设计了一种面向 LCS 的可嵌入距离, 以便训练嵌入模型将目标代码编码为所需的低维稠密向量, 提出了两种训练嵌入模型的损失函数 L_{LCS} 与 L_{LDSS}, 前者使嵌入向量精确地反映输入序列的距离, 后者使模型更好地捕捉函数中字面不同但语义相同的元素.

3) 将本方法应用于现实项目的缺陷检测中, 成功检测到 23 个新的缺陷, 并获得了开发者的确认.

本文第 1 节介绍相似代码检测的相关方法和研究现状. 第 2 节介绍本文提出的最长公共子序列嵌入支持下的代码相似性检测方法. 第 3 节通过对比实验验证了所提方法的有效性. 最后总结全文.

1 代码相似性检测相关工作

代码相似性度量在代码缺陷检测、代码克隆检测等下游任务中扮演重要角色. 为了使检测可以在大规模代码库上进行, 研究人员提出多种具有可扩展性的代码相似性检测方法.

基于相似性的缺陷检测器[5,7,17,18,22,23]查找与已公开漏洞相似度较高的潜在缺陷. VUDDY[17]生成规范化后的源代码哈希值, 然后将其与 CVE 漏洞库中的漏洞代码哈希值进行比较, 并将碰撞的代码报告为潜在漏洞. 除了文本的相似度之外, 一些工作在相似度度量中还考虑了词特征、语句特征、控制流特征和依赖特征. 例如, MVP[23]计算函数词集合的相似度, 李赞等人[7]计算函数中归一化的语句集合的相似度, VGRAPH[24]计算函数图表示的相似度, Zhang 等人[1]通过执行最佳顶点匹配来检测缺陷.

根据克隆检测中使用的代码特征, 典型的克隆检测器可以分为基于词或短语的克隆检测器[12,25]、基于序列的克隆检测器[8,9,26−28]、基于树的克隆检测器[29]和基于图的克隆检测器[30−32]. NiCad[8]是一种基于序列的技术, 以短语为匹配单位计算标准化后代码文本的 LCS. 同样地, NIL[9]在简单解析的词序列上计算 LCS 的长度, 以找到深度重叠的序列. CCAligner[25]通过滑动窗口技术将代码序列分割成多个固定长度的片段, 使用哈希技术进行匹配, 识别出可能的克隆片段并确定最终的克隆对, 对克隆片段中插入、删除和修改等操作有较高的鲁棒性. SourcererCC[12]从代码中提取出块级别的代码片段, 并将其转换为词袋模型, 通过计算不同代码块之间的词袋相似度进行克隆检测. 基于树和图的方法可以发现更多语义克隆, 例如利用程序依赖图信息的 CCGraph[32]使用 WL 图核来计算图的相似度. 基于深度学习的克隆检测器[33−35]使用编码器将源代码特征融合到嵌入向量中, 并用嵌入向量进行进一步的预测, 例如 ASTNN[34] 利用抽象语法树 (AST) 来捕捉代码的语法结构, 并在此基础上进行深度学习建模.

为了在大规模代码库上实现克隆检测, 可扩展的克隆检测器[9,12,26]提出优化的匹配算法、过滤机制或各种处理大型数据库的索引方法. 其中, NIL 是大型代码库上最先进的可扩展检测器, 它利用倒排索引来减少克隆候选函数.

我们的工作也是基于序列上的 LCS 来计算的, 而不同于 NIL、NiCad 等直接使用基于动态规划等 LCS 查找算法, 我们利用嵌入模型来模拟一个 LCS 距离, 在对模型进行充分训练后, 使模型对序列的嵌入向量的欧氏距离可以近似于序列之间的基于 LCS 的距离, 这样, 极大地提高了计算效率. 在模型训练过程中, 我们使用了语句、函

数调用、操作符调用、变量等特征, 与直接或规范化后使用函数中的词 (token) 作为 LCS 距离计算单元的方法 (NIL) 相比, 可以更好地捕捉函数的语义特征, 不但可以用于发现简单克隆, 还具有较强的捕获复杂克隆的能力. 与其他基于深度学习的克隆检测工作不同, 我们并没有要求模型直接输出是否为克隆的标签, 而是使用基于 LCS 的序列距离指导嵌入向量之间的相对距离, 这对模型来说是更直接、更容易学习的. 在提高检测的可扩展性方面, 不同于 NIL 使用倒排索引对候选代码对先进行一次过滤, 我们的方法是对所有的嵌入向量执行相似性检测, 在嵌入向量上的距离计算本身只需非常轻量级的算力, 并且可以进一步利用近似最近邻算法来加速大型代码库内的匹配.

2 最长公共子序列嵌入支持下的代码相似性检测方法

我们将函数嵌入到向量空间, 以支持快速且可扩展的面向 LCS 距离的代码相似性检测. 如图 3 所示, 工作流程由 3 个阶段组成. 首先, 对代码库进行预处理, 并抽取函数的特征矩阵. 然后, 采用卷积神经网络 (CNN) 将特征矩阵嵌入到向量空间中. 最后, 对向量进行相似度计算以搜索相似函数. 图 3 中虚线框中的过程显示了嵌入模型的训练过程.

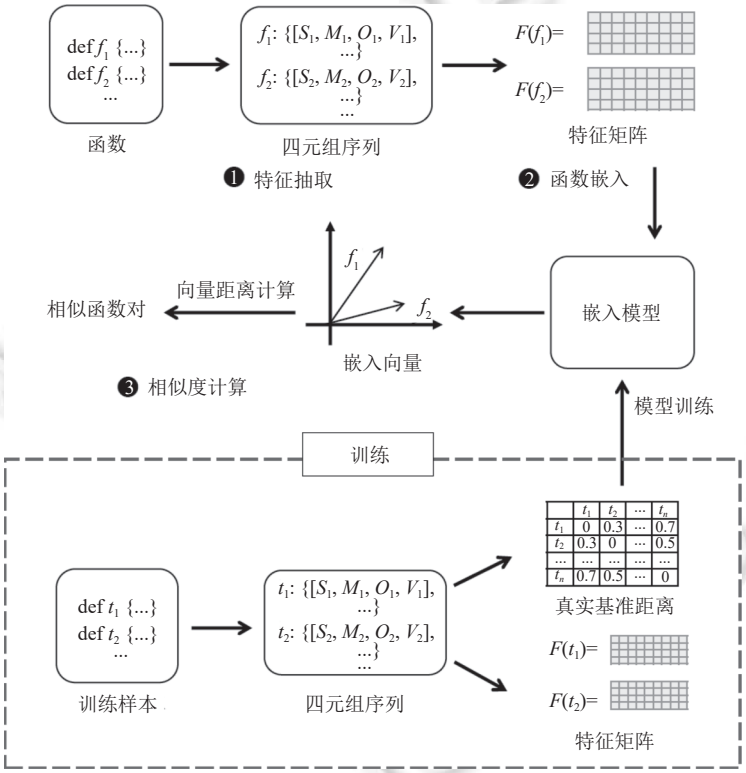


图 3 基于面向 LCS 嵌入的可扩展性代码相似性检测方法框架图

2.1 可嵌入距离 *D*

在我们的工作中, 训练嵌入模型时需要获取两个序列之间的真实基准距离. 根据文献 [20], 严格定义的距离 δ 如果能满足以下 3 个要求, 则可以嵌入到空间中.

$$\delta(x,y) \geqslant 0 \text{ (非负性)} \tag{1}$$

$$\delta(x,y) = \delta(y,x) \text{ (对称性)} \tag{2}$$

$$\delta(x,y) + \delta(y,z) \geqslant \delta(x,z) \text{ (三角不等式)} \tag{3}$$

基于文本的 LCS 方法 NIL 使用 $\frac{lcs(s_1,s_2)}{\min(len(s_1),len(s_2))}$ 作为文本相似度的度量方法, 这个距离无法满足三角不等式, 即当我们希望使用模型来快速模拟文本相似度时, 这个公式距离不能用作模型训练的基准值. 为了解决这个问题, 我们设计了一个可嵌入距离 D 来近似基于 LCS 的比较, 如公式 (4) 所示, 其中 s_1 和 s_2 表示两个序列, $lcs()$ 计算序列间的 LCS. 距离度量 D 体现了较长序列中不在 LCS 中的元素的比例. 也就是说, 它表示两个序列之间的不相似程度 (即 LCS 距离相对于较长的距离的相反程度). 因此, 距离越小, 两个被测函数之间的相似度越高.

$$D=1-\frac{len(lcs(s_1,s_2))}{\max(len(s_1),len(s_2))} \tag{4}$$

经证明, 距离 D 是一个符合距离三要素的度量, 可以被嵌入至向量空间中, 证明过程请见附录 A.

2.2 特征抽取

将代码嵌入至分布式表示时不是直接处理源代码, 而是首先从每个函数中以语句为单位提取特征, 并将这些特征编码为每个函数的特征矩阵, 然后对矩阵进行嵌入, 使得嵌入向量之间的距离可以大致模拟两个函数的语句序列之间的 D 距离.

首先解析源代码并提取每个语句的重要特征. 在本研究中, 语句级特征由 4 类重要信息组成, 用<S, O, M, V>表示. S 表示语句类型, 涉及 7 种类型, 描述语句的结构, 包括变量声明语句、if 语句、for 语句、while 语句、switch 语句、表达式语句和返回语句. 其中, 表达式语句指一般的典型语句, 例如函数调用或赋值. 对于每个语句, O 包含所涉及的运算符 (例如“+”), M 存储语句中调用的函数, V 保存语句所声明变量的数据类型. 然后, 我们将 4 个分量分别编码为 4 个特征向量, 即 f_S、f_O、f_M 和 f_V. 每个向量的生成方法如下: 由于给定的语句类型有限 (7 种), 因此将 f_S 编码为一个 7 维的独热向量; 由于可用的运算符受到语言规范的限制, 当我们在本研究中仅考虑 Java 和 C 项目时, 操作符总类型数为 32, 此时 f_O 编码为一个 32 维的词袋向量, 表示语句中的所有运算符; f_M 是语句中调用函数名嵌入的平均值; f_V 是语句中声明的变量类型的嵌入, 两个向量都是语言模型的输出向量, 在本工作中我们将 f_M 和 f_V 的长度都设置为 50 维, 这是语言模型中一个常被选用的超参数.

根据我们的经验和观察, 在流行的商用项目中, 函数名称与数据类型通常被创建为包含某些语义信息的子词的组合. 因此, 我们首先使用分词器将所有函数名称与数据类型分词, 使用语言模型来嵌入子词, 将标识符中子词的嵌入相加得到标识符 (调用的函数或数据类型) 的语义向量. 本研究中, 我们仅关注 Java 和 C 项目, 在前期调研中, 我们发现在一些项目采用驼峰命名法或蛇形命名法来组合非缩写的名词和动词, 然而在另一些项目的标识符中, 缩写可能直接相连. 例如, 在项目 LwIP 中一个名为“netconn_set_recvbufsize”的函数, 如果我们用“_”分割它, 则包含 3 个子词, 但是两个子词又是由具有语义的缩写词组成的. 因此, 在不含缩写的项目中, 我们简单地用正则表达式规则拆分标识符, 并使用在维基百科数据集上训练的预训练语言模型 GloVe[36]来实现精确的子词编码. 存在直接相连的缩写项目中, 我们通过利用训练好的一元分词器 (unigram tokenizer)[37]模式的 SentencePiece[21]分词器来对标识符进行分词, 再使用一个训练好的面向子词的 Word2Vec[38]语言模型进行子词的嵌入. 与基于子词的语言模型 (如使用基于 n-gram 分词的 FastText[39]) 相比, 这种方式可以保证子词更有语义, 因为后者可能会切分出无意义的子词.

2.3 嵌入模型与训练

我们进一步将函数的特征矩阵嵌入到一个稠密向量中, 以进行可扩展性的代码相似度检测. 为此, 我们提出了一个嵌入模型, 它接受函数的特征矩阵作为输入并输出 256 维向量.

图 4 展示了嵌入一个函数的工作流程. 从输入特征矩阵开始, 它由 3 个步骤组成. 首先, 对应于 f_O、f_M 和 f_V 的每个子矩阵通过 3 个激活的线性层进行压缩, 并将 3 个输出连接起来形成 $m\times30$ 维的特征矩阵 (m 是特征序列的长度限制. 其取值在后文第 3.1 节中决定). 将该矩阵乘以 f_S 得到 $m\times210$ 维的张量, 然后, 将张量输入到 4 个两层一维卷积层 (1D-CNN layer), 卷积核相应设置为 3、4、5 和 6. 每个卷积层后面都有一个最大池化层 (max-pooling layer). 最后, 每个网络中最后一个卷积层/池化层的输出被连接并展平, 然后线性化以获得最终的嵌入向量.

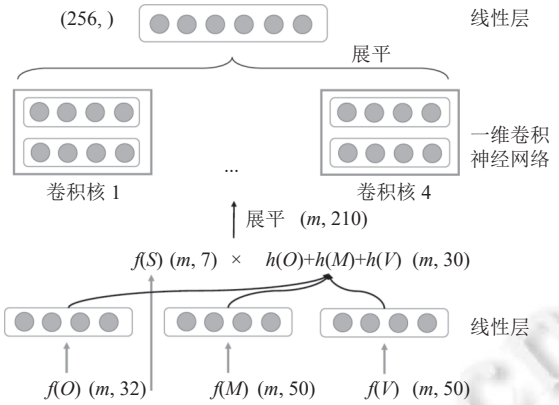


图 4 嵌入模型的网络结构

我们使用孪生网络 (siamese network) 结构来训练模型, 孪生网络的嵌入模型对一组样本进行共享参数的嵌入, 并输出嵌入向量之间的欧氏距离. 向量之间的欧氏距离可以近似模拟函数语句序列之间的 LCS 距离, 并同时考虑了两个函数语句序列中 LDSS 元素的相似度.

2.3.1 损失函数 L_{LCS} 与训练集 TS_{LCS}

训练集 TS_{LCS} 由三元组样本<S, P, N> 组成. S 是锚点, P 比 N 更接近 S, 即 $D(S,P) < D(S,N)$.

因为现实世界代码库中相似函数对的数量远远少于不相似函数对的数量, 并且没有已标注的训练集, 我们通过数据增强 (data augmentation) 来构建 TS_{LCS}. 具体来说, 通过修改或删除一个函数中的一些语句以生成新的语句, 对于任何函数 S, 我们都会生成相应的样本 P 和 N. 旧样本 S 和新样本 (P 或 N) 之间的距离由更改的语句数量控制. 构造三元组后, 可以计算任意两个样本之间的真实基准距离 D.

损失函数 L_{LCS} 如公式 (5) 所示. L_{LCS} 是均方误差损失 L_m 和三元组损失 L_t 的组合, 使模型生成具有模仿函数 D 距离能力的嵌入.

$$L_{\mathrm{LCS}}(S,P,N) = \alpha(L_m(S,P) + L_m(N,P) + L_m(S,N)) + \beta L_t(S,P,N) \tag{5}$$

$$L_m(S,P) = (\|v(S) - v(P)\|_2 - D(S,P))^2 \tag{6}$$

$$L_t(S,P,N) = \max\{0, \|v(S) - v(N)\|_2 - \|v(S) - v(P)\|_2 - (D(S,N) - D(S,P))\} \tag{7}$$

其中, α 和 β 是控制均方误差损失和三元损失的超参数, v 表示序列在空间中的嵌入向量值. L_m 反映了向量间的近似距离与真实基准距离之间的差别, L_t 体现了模型在维持两对存在远近关系的序列的嵌入向量间的相对距离的能力, 即, 让真实距离更近的一对序列 S、P 的嵌入向量距离也小于另一对序列 S、N.

2.3.2 损失函数 L_{LDSS} 与训练集 TS_{LDSS}

使用 L_{LCS} 训练模型可以使嵌入距离模拟 LCS 距离, 但不能捕获那些 LDSS 序列. 我们的目标是解决后一种情况, 以检测复杂的代码相似性. 为此, 我们提出损失函数 L_{LDSS} 并在训练集 TS_{LDSS} 上训练模型.

TS_{LDSS} 仅包含配对样本<S_1, S_2> 且每对配对样本中均包含 LDSS 语句. 生成训练集 TS_{LDSS} 时, 我们扫描前文中生成的数据增强后的代码库, 如果两个函数中包含至少一对相似的语句, 则生成配对样本. 我们使用语义嵌入向量的相似度来判断一对函数中各个语句是否相似. 值得注意的是, 真实基准距离 $D(S_1,S_2)$ 是通过将 LDSS 语句视为不同的语句来计算的. 考虑到 LDSS 语句的相似性, <S_1, S_2> 的嵌入距离应略小于基准距离 $D(S_1,S_2)$. 因此, 我们将 L_{LDSS} 表示为公式 (8), 其中, ε 是边距参数.

$$L_{\mathrm{LDSS}}(S_1,S_2) = \max(0, \|v(S_1) - v(S_2)\|_2 - D(S_1,S_2) - \varepsilon) \tag{8}$$

使用 L_{LDSS}, 我们强制模型为具有 LDSS 语句的代码片段生成比仅基于 L_{LCS} 的模型更接近一些的嵌入. 在进行 LDSS 元素识别时, 我们计算相应标识符的余弦距离 (记为 $sim(\cdot,\cdot)$), 当 $0.95 < sim(f_M^{S_1}, f_M^{S_2}) < 1$ 时, 我们认为两个语句含有 LDSS 元素.

2.3.3 迭代训练

我们使用 TS_{LCS}/L_{LCS} 和 TS_{LDSS}/L_{LDSS} 迭代训练模型. 该模型首先使用 TS_{LCS}/L_{LCS} 进行训练. 在训练中使用 TS_{LCS} 中的每个训练样本后 (第 1 个 epoch), 我们使用 TS_{LDSS}/L_{LDSS} (第 2 个 epoch) 微调模型, 然后切换回 TS_{LCS}/L_{LCS} (第 3 个 epoch). 总共每个训练集迭代应用 100 个 epoch 来优化网络. 在 200 个 epoch 后, 在 TS_{LCS}/L_{LCS} 上进行一轮额外的训练, 以确保模型面向 LCS 的嵌入能力. TS_{LCS} 与 TS_{LDSS} 的比例由参数 r 控制, 经过预实验, 我们将参数 r 选定为 1:2.

2.4 相似度计算

模型训练完成后, 我们可以将所有函数嵌入到稠密的向量空间中, 然后计算查询向量和目标向量之间的欧氏距离. 这里, 查询向量对应于克隆检测中种子函数的嵌入向量或基于匹配的缺陷检测中缺陷函数的嵌入向量. 在克隆检测场景中, 如果欧氏距离小于阈值 T_D, 我们将相应的目标函数视为识别出的克隆函数. 在缺陷检测场景中, 我们按照距离排名, 选择排名靠前的函数进行审核.

尽管基于向量计算的 LCS 搜索已经比基于文本的 LCS 搜索快得多, 但还是可以使用各种近似最近邻算法进一步加速向量距离计算. 在本研究中, 我们利用工业级近似最近邻搜索工具 Faiss[19]来支持大规模代码库中的快速相似度计算. Faiss 是 Facebook 团队开发的一个对大量向量数据进行相似向量搜索的库, 通过索引和聚类快速地实现相似性搜索. 我们应用 Faiss 中的 IndexIVFPQ 方法来管理我们生成的嵌入数据库. IVFPQ 即基于乘积量化 (product quantizer, PQ) 的倒排文件系统 (inverted file system, IVF). IVF 首先对嵌入数据集里的向量做聚类, 在之后的每次查询中只对与查询向量较近的几个簇中的向量做查询, 大大减少了需要距离计算的次数. PQ 将嵌入数据库的长向量分割成多个短向量, 为所有对应维数的短向量集合进行聚类并分配一个类别值, 提前计算各个聚类中心之间的距离, 要得到两个长向量间的距离, 只需要分别累加各个短向量所在聚类中心之间的距离, 缩短了距离计算的时间. 因此, 在使用 IndexIVFPQ 模式对大规模向量库内的最近邻搜索进行加速时, 算法会先使用 IVF 筛选出最近的几个簇, 再在其中计算并返回乘积量化距离最小的前几条记录.

3 实验分析

我们将本方法应用于真实项目和基准数据集上, 以评估本方法的可扩展性和有效性. 所有实验均在有 64 GB 内存和 GeForce RTX 2080 Ti GPU 的服务器上完成.

3.1 实验环境及参数

在实验评估阶段中, 本方法在 C 语言项目上的训练集基于开源项目 OpenSSL 进行数据增强生成, 在 Java 语言项目上的训练集基于 BigCloneBench[40,41]进行数据增强生成. 克隆检测对比实验均在 BigCloneBench 的测试集上进行, 如果被测工具需要训练, 则在与本方法相同的训练集上对其训练.

使用 IJaDataSet 来评估被测工具的可扩展性, IJaDataSet 是一个拥有 2.5 亿行代码 (MLOC) 的大规模 Java 代码库, 在 NIL[9]、SourcererCC[12]等工作中被用来评估克隆检测工具在大型项目上的表现. 由于大型数据集 IJaDataSet 没有人工验证的基准标注 (ground truth), 我们分别使用不同实验对象进行工具的可扩展性检测和准确率、召回率的分析, 在 NIL、CCAligner、SourcererCC 等工作中同样如此.

选择 NiCad[8]、NIL、SourcererCC、 CCAligner[25]、ASTNN[34]和 VUDDY[17]作为本工作的对比工作, 其中 NiCad 和 NIL 是两种基于 LCS 的克隆检测器, VUDDY 通过比较函数的哈希值来识别与种子相同的缺陷.

下面讨论超参数设置.

由于基于 CNN 的网络严格规定了输入特征序列的长度, 因此我们限制了函数的大小, 按照文献 [8,9,25] 的做法将函数的最短语句个数设置为 6. 我们分别确定了 Java 代码数据集和 C 代码数据集的函数最大长度, 短于 m 的函数的特征序列用 0 填充, 长于 m 的函数将被截断. 对于 Java 代码, 我们在 BigCloneBench 中统计了长度大于 6 行的函数, 发现其中超过 95% 的函数不超过 80 行, 因此, Java 代码中 m 设为 80. 对于 C 代码, 我们统计了 Linux 内核和 OpenSSL 中的函数, 并将 m 设置为 60.

T_D 越大, 通常表示召回率越高, 但精确率越低. 为了与其他工具进行公平比较, 我们调整 T_D 以接近对比工具的精确率, 并比较召回率 (见后文第 3.4 节).

对于本文中参与实验的对比工作, 我们均使用其开源代码中的默认参数设置.

为了尽可能多地学习计算机术语的缩写分词, 我们在 libxml2、QEMU、mRuby、WireShark、OpenSSL、PostgreSQL、Linux Kernel、ImageMagick、httpd 和 RIOT 这 10 个项目上训练 C 语言的 SentencePiece 分词器和 Word2Vec 语言模型.

3.2 可扩展性研究

与基于 LCS 的方法相比, 高可扩展性是本方法的一个基本特征.

我们首先证明, 即使不采用 Faiss 来加速向量搜索, 与朴素的 LCS 算法相比, 本工作所提出的面向 LCS 的嵌入方法也具有很高的相似性计算速度. 我们使用 Linux 内核作为数据集 (其中包含超过 330 000 个函数), 计算每个函数与所有其他函数之间的欧氏距离. 使用面向 LCS 的嵌入进行计算仅需 23 min, 即每个函数与所有其他函数的平均计算时间约为 0.004 s, 相比之下, 朴素 LCS 算法对一个函数进行一遍查询运算平均需要 8 min. 这种可扩展性的提升是本文的一个核心的贡献.

将我们的方法与两种基于 LCS 的相似函数检测方法 (即 NiCad 和 NIL) 进行比较. 我们使用 IJaDataSet 来评估被测工具的可扩展性, 对其中的每个函数, 在整个数据集中搜索相似函数. 实验分为两个阶段. 首先, 所有工具都在具有 100 MLOC 的较小数据集上运行, 其次, 在第 1 轮中成功的工具在完整数据集 (250 MLOC) 上进行测试. 每个工具运行的最大内存为 55 GB. 我们记录工具每个步骤的时间成本, 并将结果列在表 1. 表 1 中, NIL-100 MLOC 表示使用默认索引块数进行实验, NIL-100 MLOC*则表示使用了非默认索引块数 (10). 另外, 使用本文方法在 100 MLOC 数据集与 250 MLOC 数据集上测试时, 分别统计了对 Faiss 不同数量初筛结果进行相似度计算的时间开销, 由上自下分别为前 10 条记录、前 100 条记录与前 1 000 条记录.

表 1　在 IJaDataSet 上各工具的分步骤运行时间

实验设置	解析	中间结果处理	计算相似度
NiCad-100 MLOC	14 h 48 min 17 s	1 h 5 min 22 s	错误
NIL-100 MLOC	29 min 22 s		> 6 h
NIL-100 MLOC*	3 min 48 s		2 h 45 min 47 s
NIL-250 MLOC	内存溢出		—
本文方法-100 MLOC	14 min 10 s	1 h 55 min	3 min 55 s 5 min 16 s 9 min 36 s
本文方法-250 MLOC	3 h 5 min 45 s	5 h 47 min 54 s	14 min 25 s 28 min 32 s 51 min 31 s

在我们的方法中, 我们选择使用工业级的近似最近邻算法 Faiss 初步筛选查询的前 10 条、前 100 条、前 1 000 条最近邻记录, 再进行进一步的距离计算. 在最近邻搜索返回值个数的选择上, 我们对查询结果进行了人工分析, 在筛选前 1 000 条记录的场景下保留的大部分函数均已超出克隆阈值 T_D, 可以认为 1 000 已经远远超过了一个函数可能存在的克隆数量, 因此我们认为实验中的召回率不受 Faiss 的影响.

从结果来看, NiCad 的可扩展性最差, 由于超出了支持的最大字符数 (6 亿) 的限制, 它无法在 100 MLOC 数据集中执行查询. NIL 效果较好, 我们使用其开源工具在 100 MLOC 数据集上使用默认索引块数量 (50) 进行检索时在 6 个多小时内没有任何输出, 当将索引数改为 10, 构建索引库大约需要 3 min 48 s, 相似度计算则需要 2 h 45 min 以上, 然而在完整数据集 (250 MLOC) 上构建索引时会耗尽分配的内存 (55 GB). 相比而言, 我们的方法在完整数据集上也可以正确运行, 在查询所有函数的前 10 条记录的情况下, 仅花费 3 min 55 s (100 MLOC) 和 14 min 25 s (250 MLOC), 即使查询数据集中排名前 1 000 条的记录也仅需要 51 min 31 s.

上述讨论表明, 我们的方法实现了较高的空间和时间效率, 证明了其在超大规模代码搜索上的可扩展性.

3.3 基于匹配的缺陷检测

本节应用我们的方法检测 C 语言项目中包含已知缺陷逻辑的缺陷函数. 按照文献 [24] 的做法, 我们从开源项目的提交日志中收集有缺陷的查询函数. 首先, 用感兴趣的关键字筛选出提交日志, 然后保留仅有一行修补的函数, 将修补前的函数用作查询种子. 查询、目标项目以及缺陷函数如表 2 所示.

表 2 本文方法查找到的已确认缺陷

#	种子项目	目标项目	种子函数	缺陷函数
1	LinuxKernel	LinuxKernel	smu_v12_0_fini_smc_tables	yellow_carp_fini_smc_tables
2	LinuxKernel	LinuxKernel	amdgpu_connector_lcd_native_mode	radeon_fp_native_mode
3	LinuxKernel	LinuxKernel	mips_cdmm_phys_base	mips_cpc_default_phys_base
4	LinuxKernel	LinuxKernel	play_deferred	nfcmrvl_play_deferred
5	LinuxKernel	LinuxKernel	amdgpu_mode_dumb_create	radeon_mode_dumb_create
6	LinuxKernel	LinuxKernel	l2cap_ecred_connect	l2cap_le_connect
7	LinuxKernel	LinuxKernel	amdgpu_sa_bo_next_hole	radeon_sa_bo_next_hole
8	LinuxKernel	LinuxKernel	u132_hcd_init	ena_init
9	OpenSSL	OpenSSL	BN_hex2bn	BN_dec2bn
10	OpenSSL	OpenSSL	verify_integrity	ossl_pkcs5_pbkdf2_hmac_ex
11	sdb	sdb	foreach_list_cb	sdbkv_new2
12	LinuxKernel	FreeBSD	Ib_device_register_sysfs	ib_device_register_sysfs
13	LinuxKernel	FreeBSD	ib_mad_post_receive_mads	ib_mad_post_receive_mads
14	LinuxKernel	FreeBSD	mlx4_opreq_action	mlx4_opreq_action
15	LinuxKernel	FreeBSD	ib_cm_insert_listen	ib_cm_insert_listen
16	LinuxKernel	FreeBSD	rem_slave_fs_rule	rem_slave_fs_rule
17	LinuxKernel	FreeBSD	mlx4_QP_FLOW_STEERING_DETACH_wrapper	mlx4_QP_FLOW_STEERING_DETACH_wrapper
18	LinuxKernel	FreeBSD	ucma_create_id	ucma_create_id
19	LinuxKernel	FreeBSD	clean_mr	clean_mr
20	LinuxKernel	FreeBSD	uverbs_user_mmap_disassociate	uverbs_user_mmap_disassociate
21	LwIP	MbedOS	tcp_keepalive	tcp_keepalive
22	LwIP	MbedOS	tcp_rst	tcp_rst
23	LwIP	MbedOS	tcp_zero_window_probe	tcp_zero_window_probe

审计人员在审核缺陷报告时通常会选择相似度排名最高的候选者, 我们有意将阈值设置为一个较大的值 (0.5) 以捕获更多候选, 对于每个查询, 在候选函数超过 10 个的时候只检查相似度排前 10 的函数. 其他工具不对结果进行排名, 因此我们检查了所有结果.

我们首先在项目内搜索与查询函数类似的函数, 即在与种子相同的项目中查询相似的函数以进行缺陷检测. 在表 2 中, 通过这种方式发现了 11 个缺陷, 其中 7 个与种子相比涉及了大量编辑 (例如插入, 删除, 调用新函数), 即#4–#8、#10 和#11. 表 2 中其余 12 行显示了通过使用种子跨工程查找类似函数而发现的缺陷. 其中两个有缺陷的函数 (#13 和#19) 涉及除变量重命名之外的复杂编辑. 在审计效率方面, 除了查询函数#11 有 8 个检索结果外, 其余的函数均只有 1 个检索结果. 研究人员对总共 30 个候选函数进行了人工审计后, 认为 23 个函数可能存在缺陷, 向其开发者提交补丁后这些缺陷均得到确认和修补. 在模型泛化性方面, 4 个项目 (即 sdb、FreeBSD、LwIP 和 MbedOS) 没有用于训练分词器或嵌入网络, 但 23 个缺陷中有 13 个 (#11–#23) 与这 4 个项目有关, 这进一步证明了模型在面对新项目时可以获得良好的结果而不需要重新训练模型.

将本文方法与另外两种基于 LCS 的代码相似性检测工具 NIL 进行比较, 通过将候选函数与种子进行匹配来检测缺陷. 本实验中, 我们只关注 9 个 III 型克隆缺陷 (#4、#5、#6、#7、#8、#10、#11、#13、#19), 它们的函数包含除简单变量重命名之外的编辑. NiCad 可以发现#4、#7、#9 这 3 个缺陷, NIL 漏掉了 2 个项目内缺陷和 1 个项目间缺陷. 我们手动检查代码, 发现与#4、#13 和#19 相关的函数仅涉及简单的代码更改, 例如语句插入/删除或

标记插入. #5、#6 和#7 包含 LDSS 函数调用, 但整个函数非常相似, 这使得它们很容易被 NIL 检测到.

3.4 一般性克隆检测

我们在 BigCloneBench 上进行代码克隆检测, 结果如表 3 所示. BigCloneBench 考验工具对各种类型克隆的捕捉能力, 其中复杂类型克隆的实例更多, 在测试集中 T1/2 (I、II 型克隆)、VST3 (极强 III 型克隆)、ST3 (强 III 型克隆)、MT3 (中等 III 型克隆)、WT3/4 (弱 III 型克隆及 IV 型克隆) 分别有 19 352、2 235、3 424、13 282、683 747 对.

表 3 各工具在 BigCloneBench 上代码克隆检测结果 (%)

检测工具	精确率	召回率						
		T1/2	VST3	ST3	MT3	WT3/4	$R^{\text{T1-MT3}}$	R^{ALL}
NIL	99.09	99.93	99.41	84.78	20.53	0.55	71.01	4.29
NiCad	99.83	99.92	71.77	19.25	1.4	0.0	56.45	3.00
VUDDY	100	98.81	6.98	0.23	0.0	0.0	50.36	2.67
SourcererCC	99.89	99.83	96.55	59.17	1.06	0.02	61.74	3.30
本文方法T_D=0.47	98.99	99.96	90.38	47.20	26.49	0.93	69.20	4.55
ASTNN	53.28	99.96	99.91	93.20	76.11	6.67	91.08	11.15
CCAligner	46.88	99.96	99.91	98.74	74.02	14.79	90.85	18.83
本文方法T_D=0.67	52.69	99.96	98.93	86.07	80.86	15.24	92.03	19.31

BigCloneBench 中包含了 45 种功能类型的函数及其之间的克隆关系. 我们对 BigCloneBench 的训练集、测试集的分割方法为: 将功能 ID 小于等于 25 的函数和克隆关系作为训练集, 大于 25 的函数和克隆关系作为测试集. 与对比工作 ASTNN 等直接将所有克隆按比例划分不同, 这种分割方法可以确保训练集和测试集之间没有交叉, 更能够考验模型的语义捕捉能力和泛化能力.

表 3 中的实验指标意义如下. 当工具报告数量为 F, 且其对 T1/2、VST3、ST3、MT3、WT3/4 各类型克隆命中的数量分别为 $H(\text{T1/2})$、$H(\text{VST3})$、$H(\text{ST3})$、$H(\text{MT3})$、$H(\text{WT3/4})$ 时, 精确率的计算为 $\frac{H(\text{T1/2})+H(\text{VST3})+H(\text{ST3})+H(\text{MT3})+H(\text{WT3/4})}{F}$, 特定类型 (如 T1/2) 的召回率计算方法为 $\frac{H(\text{T1/2})}{19352}$, 其中分母是该类型克隆的总数量. 由于 WT3/4 类型克隆数量极多但捕捉难度较大, 某些工具在 WT3/4 上表现较差, 为了更好地进行比较, 我们在计算召回率时使用了不考虑 WT3/4 和考虑 WT3/4 的两个指标 $R^{\text{T1-MT3}}$ 和 R^{ALL}. $R^{\text{T1-MT3}}$ 计算前 4 种克隆的召回率, 即 $\frac{H(\text{T1/2})+H(\text{VST3})+H(\text{ST3})+H(\text{MT3})}{19352+2235+3424+13282}$, R^{ALL} 计算所有克隆的召回率, 即 $\frac{H(\text{T1/2})+H(\text{VST3})+H(\text{ST3})+H(\text{MT3})+H(\text{WT3/4})}{19352+2235+3424+13282+683747}$.

BigCloneBench 中的复杂类型克隆 (MT3、WT3/4) 比较难以捕捉, 对于对比工作亦是如此. 一些对比工作保证了较高的精确率, 但在召回率上表现不好, 此时将阈值调低, 反之亦然. 在对比工作中, NIL、NiCad、VUDDY、SourcererCC 倾向于确保精确率, 因此将阈值 T_D 调整至 0.47 使精确率与这 3 种工具近似, 并与其比较召回率; 同样地, 复杂克隆检测工具 ASTNN 和 CCAligner 对各种类型的克隆在召回率上表现良好, 但精确率较低, 因此我们将阈值 T_D 设置为一个较大的值 (0.67) 使精确率与这 3 种工具近似, 并比较召回率.

从与 NIL、NiCad、VUDDY、SourcererCC 的比较结果来看, 本文工作在精确率相当的情况下在 $R^{\text{T1-MT3}}$ 上超过了除 NIL 外的所有方法, 在 R^{ALL} 上超过所有方法. 特别地, 我们的工具虽然在精确克隆检测中略差于 NIL, 但在整体的克隆检测中强于 NIL, 这说明了我们的工具在复杂克隆的检测中表现更好, 更加体现了本方法对代码特征的抽象作用强, 非常适用于广泛搜索语义克隆的情境. 从与 ASTNN、CCAligner 的比较结果来看, 当 T_D=0.67 时, 本方法的精确率与 ASTNN 相当, 而召回率大幅超出 ASTNN; 同时, 本方法的召回率与 CCAligner 相当, 而在精确率上大幅超出 CCAligner.

上述比较表明, 本文工作可以胜任高精度需求的克隆检测也可以胜任高召回需求的克隆检测, 对简单到复杂

克隆的检测都具有较高的能力, 说明了基于面向 LCS 的嵌入的克隆检测的有效性.

3.5 消融实验

本文工作使用 L_{LCS} 和 L_{LDSS} 两种损失函数对模型交替训练, 其中 L_{LCS} 使模型具有使用嵌入距离严格模拟距离 D 的能力, L_{LDSS} 使模型在衡量两个有相似语句的序列时生成稍近的距离. 为了确定 L_{LDSS} 的必要性, 我们同时在不考虑 TS_{LDSS}/L_{LDSS} 的情况下训练了模型. 表 4 中记录了两个模型在 T_D=0.47 时的检测结果. 在不使用 L_{LDSS} 的情况下, 精确率只有小幅上升, I、II 型克隆的召回率保持不变, 但 III、IV 型克隆的召回率有明显下降, 表明 LDSS 的克隆没有被正确识别. 可以看到, L_{LDSS} 主要影响 III、IV 型克隆的检测, 表明 L_{LDSS} 可以提高本文方法的性能. 即使没有 L_{LDSS}, 方法也可以基本实现比对比工作更高的召回率, 这也验证了 L_{LCS} 的有效性.

表 4 使用不同损失函数在 BigCloneBench 上代码克隆检测结果 (%)

损失函数	精确率	召回率						
		T1/2	VST3	ST3	MT3	WT3/4	$R^{T1\text{-}MT3}$	R^{ALL}
L_{LCS}	99.74	99.94	81.83	34.67	6.75	0.21	60.72	3.42
L_{LCS} + L_{LDSS}	98.99	99.96	90.38	47.20	26.49	0.93	69.20	4.55

3.6 有效性威胁分析

我们使用 BigCloneBench 对所有工具进行一般性克隆检测, 它具有确定的真值, 因此对所有工具并无偏差, 但是使用其他基准可能会得到不同的结果. 我们使用搜索结果的精确率和不同种类克隆的召回率作为测试指标, 这是大多数克隆检测工作所选择的测试指标.

由于大型数据集 IJaDataSet 没有基准真值, 我们按 NIL、CCAligner、SourcererCC 等工作的做法, 使用不同的数据集进行可扩展性检测和准确率召回率的检测. 在 IJaDataset 上进行可扩展性检测实验时, 尽管我们在预实验中进行了人工分析, 认为近似最近邻工具返回的前 1 000 条值已远远超出 IJaDataset 中一个函数可能含有的克隆数, 但不排除使用其他数据集时这个值需要设置得更大, 从而延长检索时间, 但随着通过初步筛选的潜在克隆对的增多, 其他可扩展性工具的运行时间亦会随之增加.

在本研究中, 我们在 libxml2、QEMU、mruby、WireShark、OpenSSL、PostgreSQL、Linux Kernel、ImageMagick、httpd 和 RIOT 这 10 个项目上训练 C 语言的 SentencePiece 分词器和 Word2Vec 语言模型, 并在实验中证明模型的分词能力可以支持在没有用于训练分词器或嵌入网络的项目上的检测, 但通过增加训练样本来源, 模型可以获得更好的分词效果.

4 总 结

本文提出了一种具有可扩展性与有效性的方法来进行代码相似性度量, 该方法将函数嵌入到向量空间中, 向量间的距离可以体现函数间的一种基于 LCS 的距离. 为此, 我们提取语句级特征并将其编码为特征矩阵, 然后将其嵌入到函数级向量中, 此外, 为了使嵌入模型有效地识别包含 LDSS 元素的序列, 我们提出了针对 LDSS 的损失函数来调整模型. 通过广泛评估并将其与对比工具进行了比较, 验证了我们方法的可扩展性和有效性. 在多个项目中发现了 23 个缺陷并已被开发者确认, 其中一些缺陷在对比工具的检测中被遗漏. 当把我们的方法应用于克隆检测时, 在准确率相同的情况下, 召回率超越了多种对比工具, 并在可扩展性上超过了当前已知具有最高可扩展性的克隆检测工具.

References:

[1] Zhang XH, Gong YJ, Liang B, Huang JJ, You W, Shi WC, Zhang J. Hunting bugs with accelerated optimal graph vertex matching. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 64–76. [doi: 10.1145/3533767.3534393]

[2] Huang JJ, Han SM, You W, Shi WC, Liang B, Wu JZ, Wu YJ. Hunting vulnerable smart contracts via graph embedding based bytecode matching. IEEE Trans. on Information Forensics and Security, 2021, 16: 2144–2156. [doi: 10.1109/TIFS.2021.3050051]

[3] Yamaguchi F, Lindner F, Rieck K. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In: Proc. of the 5th USENIX Workshop on Offensive Technologies. San Francisco: USENIX Association, 2011. 118–127.

[4] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proc. of the 28th Annual Computer Security Applications Conf. Orlando: ACM, 2012. 359–368. [doi: 10.1145/2420950.2421003]

[5] Feng Q, Zhou RD, Xu CC, Cheng Y, Testa B, Yin H. Scalable graph-based bug search for firmware images. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 480–491. [doi: 10.1145/2976749.2978370]

[6] Xu XJ, Liu C, Feng Q, Yin H, Song L, Song D. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 363–376. [doi: 10.1145/3133956.3134018]

[7] Li Z, Bian P, Shi WC, Liang B. Approach of leveraging patches to discover unknown vulnerabilities. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1199–1212 (in Chinese with English abstract).http://www.jos.org.cn/1000-9825/5505.htm [doi: 10.13328/j.cnki.jos.005505]

[8] Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension. Amsterdam: Springer, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

[9] Nakagawa T, Higo Y, Kusumoto S. NIL: Large-scale detection of large-variance clones. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 830–841. [doi: 10.1145/3468264.3468564]

[10] Duan RA, Bijlani A, Xu M, Kim T, Lee W. Identifying open-source license violation and 1-day security risk at large scale. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2169–2185. [doi: 10.1145/3133956.3134048]

[11] Woo S, Park S, Kim S, Lee H, Oh H. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In: Proc. of the 43rd Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 860–872. [doi: 10.1109/ICSE43902.2021.00083]

[12] Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: Scaling code clone detection to big-code. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: IEEE, 2016. 1157–1168. [doi: 10.1145/2884781.2884877]

[13] Roy CK, Cordy JR. A survey on software clone detection research. Technical Report, No. 2007-541, Queen's University at Kingston, 2007.

[14] Sheneamer A, Kalita J. A survey of software clone detection techniques. Int'l Journal of Computer Applications, 2016, 137(10): 1–21. [doi: 10.5120/ijca2016908896]

[15] Baker BS. On finding duplication and near-duplication in large software systems. In: Proc. of the 2nd Working Conf. on Reverse Engineering. Toronto: IEEE, 1995. 86–95. [doi: 10.1109/WCRE.1995.514697]

[16] Lee S, Jeong I. SDD: High performance code clone detection system for large scale source code. In: Companion to the 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. San Diego: ACM, 2005. 140–141. [doi: 10.1145/1094855.1094903]

[17] Kim S, Woo S, Lee H, Oh H. VUDDY: A scalable approach for vulnerable code clone discovery. In: Proc. of the 2017 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2017. 595–614. [doi: 10.1109/SP.2017.62]

[18] Li Z, Lu S, Myagmar S, Zhou Y. CP-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. on Software Engineering, 2006, 32(3): 176–192. [doi: 10.1109/TSE.2006.28]

[19] Johnson J, Douze M, Jégou H. Billion-scale similarity search with GPUs. IEEE Trans. on Big Data, 2021, 7(3): 535–547. [doi: 10.1109/TBDATA.2019.2921572]

[20] Burago D, Burago Y, Ivanov S. A course in metric geometry. Graduate Studies in Mathematics, AMS, 2001.

[21] Kudo T, Richardson J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing: System Demonstrations. Brussels: Association for Computational Linguistics, 2018. 66–71. [doi: 10.18653/v1/D18-2012]

[22] Jang J, Agrawal A, Brumley D. ReDeBug: Finding unpatched code clones in entire OS distributions. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 48–62. [doi: 10.1109/SP.2012.13]

[23] Xiao Y, Chen BH, Yu CD, Xu ZZ, Yuan ZM, Li F, Liu BH, Liu Y, Huo W, Zou W, Shi WC. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: Proc. of the 29th USENIX Security Symp. USENIX Association, 2020. 1165–1182.

[24] Bowman B, Huang HH. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In: Proc. of the 2020 IEEE European Symp. on Security and Privacy. Genoa: IEEE, 2020. 53–69. [doi: 10.1109/EuroSP48549.2020.00012]

[25] Wang PC, Svajlenko J, Wu YZ, Xu Y, Roy CK. CCAligner: A token based large-gap clone detector. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 1066–1077. [doi: 10.1145/3180155.3180179]

[26] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, 2002, 28(7): 654–670. [doi: 10.1109/TSE.2002.1019480]

[27] Wu M, Wang PC, Yin KQ, Cheng HY, Xu Y, Roy CK. LVMapper: A large-variance clone detector using sequencing alignment approach. IEEE Access, 2020, 8: 27986–27997. [doi: 10.1109/ACCESS.2020.2971545]

[28] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proc. of the 1999 Int'l Conf. on Software Maintenance. Oxford: IEEE, 1999. 109–118. [doi: 10.1109/ICSM.1999.792593]

[29] Jiang LX, Misherghi G, Su ZD, Glondu S. DECKARD: Scalable and accurate tree-based detection of code clones. In: Proc. of the 29th Int'l Conf. on Software Engineering. Minneapolis: IEEE, 2007. 96–105. [doi: 10.1109/ICSE.2007.30]

[30] Liu C, Chen C, Han JW, Yu PS. GPLAG: Detection of software plagiarism by program dependence graph analysis. In: Proc. of the 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Philadelphia: ACM, 2006. 872–881. [doi: 10.1145/1150402.1150522]

[31] Wang M, Wang PC, Xu Y. CCSharp: An efficient three-phase code clone detector using modified PDGs. In: Proc. of the 24th Asia-Pacific Software Engineering Conf. Nanjing: IEEE, 2017. 100–109. [doi: 10.1109/APSEC.2017.16]

[32] Zou Y, Ban BH, Xue YX, Xu Y. CCGraph: A PDG-based code clone detector with approximate graph matching. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 931–942.

[33] 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.

[34] Zhang J, Wang X, Zhang HY, Sun HL, Wang KX, Liu XD. A novel neural source code representation based on abstract syntax tree. In: Proc. of the 41st Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]

[35] Bui NDQ, Yu YJ, Jiang LX. InferCode: Self-supervised learning of code representations by predicting subtrees. In: Proc. of the 43rd Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1186–1197. [doi: 10.1109/ICSE43902.2021.00109]

[36] Pennington J, Socher R, Manning C. GloVe: Global vectors for word representation. In: Proc. of the 2014 Conf. on Empirical Methods in Natural Language Processing. Doha: ACL, 2014. 1532–1543. [doi: 10.3115/v1/D14-1162]

[37] Kudo T. Subword regularization: Improving neural network translation models with multiple subword candidates. In: Proc. of the 56th Annual Meeting of the Association for Computational Linguistics, Vol. 1 (Long Papers). Melbourne: ACL, 2018. 66–75. [doi: 10.18653/v1/P18-1007]

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

[39] Bojanowski P, Grave E, Joulin A, Mikolov T. Enriching word vectors with subword information. Trans. of the Association for Computational Linguistics, 2017, 5: 135–146. [doi: 10.1162/tacl_a_00051]

[40] Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM. Towards a big data curated benchmark of inter-project code clones. In: Proc. of the 30th IEEE Int'l Conf. on Software Maintenance and Evolution. Victoria: IEEE, 2014. 476–480. [doi: 10.1109/ICSME.2014.77]

[41] Svajlenko J, Roy CK. BigCloneEval: A clone detection tool evaluation framework with BigCloneBench. In: Proc. of the 2016 IEEE Int'l Conf. on Software Maintenance and Evolution. Raleigh: IEEE, 2016. 596–600. [doi: 10.1109/ICSME.2016.62]

附中文参考文献:

[7] 李赞, 边攀, 石文昌, 梁彬. 一种利用补丁的未知漏洞发现方法. 软件学报, 2018, 29(5): 1199–1212. http://www.jos.org.cn/1000-9825/5505.htm [doi: 10.13328/j.cnki.jos.005505]

附录 A. 对距离 *D* 可嵌入性的证明

下面证明 D 满足三角不等式以表明所提出的 D 是一个可嵌入的度量. 其他两个性质很容易证明, 这里省略. 我们将证明限制在一种条件下: 对于序列 a、b、c, 当 $len(a) < len(b) < len(c)$ 时, 证明 $D(a,b)+D(a,c) \geqslant D(b,c)$ 成立. 其他条件下的证明可类似, 本文也略去. 将三角不等式中的 D 替换为公式 (4) 并经过变换, 我们应该证明公式 (A1).

$$len(b) \geqslant len(lcs(a,b)) + \frac{len(b)}{len(c)} \times (len(lcs(a,c)) - len(lcs(b,c))) \tag{A1}$$

如果 $len(lcs(a,c)) \leqslant len(lcs(b,c))$, 公式 (A1) 成立, 否则, 如果能证明公式 (A2) 成立, 公式 (A1) 也成立.

$$len(b) \geqslant len(lcs(a,b)) + len(lcs(a,c)) - len(lcs(b,c)) \tag{A2}$$

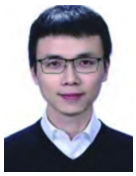
设 $t = lcs(a,b,c)$, $x = lcs(a,b) - t$, $y = lcs(a,c) - t$, $z = lcs(b,c) - t$. 此处的"–"号表示相对补集, 即 $x = \{e_i | e_i \in lcs(a,b) : e_i \notin (a,b,c)\}$. 由于 t 中的元素在 $lcs(a, b)$ 中全部存在且有序排列, 因此 $len(x) + len(t) = len(lcs(a,b))$. 因此公式 (A2) 的右半部分可以被转换为公式 (A3):

$$\begin{aligned} &len(lcs(a,b)) + len(lcs(a,c)) - len(lcs(b,c)) \\ &= len(x) + len(t) + len(y) + len(t) - len(z) - len(t) \\ &= len(x) + len(y) + len(t) - len(z) \end{aligned} \tag{A3}$$

由于序列 a 中的元素 e 只存在以下 4 种可能: 1) $e \in lcs(a,b,c)$; 2) $e \in lcs(a,b)$ 但 $e \notin lcs(a,b,c)$; 3) $e \in lcs(a,c)$ 但 $e \notin lcs(a,b,c)$; 4) $e \notin lcs(a,b)$ 且 $e \notin lcs(a,c)$. 因此, $len(x) + len(y) + len(t) \leqslant len(a)$. 由于 $len(a) < len(b)$ 且 $len(z) \geqslant 0$, 因此公式 (A1) 成立, 三角不等式可被证明.

弓媛君(1996–), 女, 博士生, 主要研究领域为软件缺陷检测.

梁彬(1973–), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为信息安全, 软件分析.



黄建军(1986–), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件安全分析, 移动应用安全.

边攀(1987–), 男, 博士, CCF 专业会员, 主要研究领域为软件缺陷检测, 缺陷自动修复.

游伟(1988–), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为安全漏洞挖掘, 恶意程序分析, 移动安全, Web 安全.

张健(1969–), 男, 博士, 研究员, 博士生导师, CCF 会士, 主要研究领域为自动推理与约束求解, 软件测试及分析.

石文昌(1964–), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为网络空间系统安全, 网络空间安全治理, 网络空间安全.