软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1758−1773 [doi: 10.13328/j.cnki.jos.006560]

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

预训练增强的代码克隆检测技术*

冷林珊[1], 刘 爽[2], 田承霖[2], 窦淑洁[2], 王 赞[1,2], 张梅山[1]

[1](天津大学 新媒体与传播学院, 天津 300072)

[2](天津大学 智能与计算学部, 天津 300350)

通信作者: 刘爽, E-mail: Shuang.liu@tju.edu.cn

摘 要: 代码克隆检测是软件工程领域的一项重要任务, 对于语义相似但语法差距较大的四型代码克隆的检测尤为困难. 基于深度学习的方法在四型代码克隆的检测上已经取得了较好的效果, 但是使用人工标注的代码克隆对进行监督学习的成本较高. 提出了两种简单有效的预训练策略来增强基于深度学习的代码克隆检测模型的代码表示, 以减少监督学习模型中对大规模训练数据集的需求. 首先, 使用 ngram 子词丰富对词嵌入模型进行预训练, 以增强克隆检测模型对词表之外的词的表示. 同时, 采用函数名预测作为辅助任务对克隆检测模型参数进行预训练. 通过这两个预训练策略, 可以得到一个有更准确的代码表示能力的模型, 模型被用来作为克隆检测中的代码表示模型并在克隆检测任务上进行有监督训练. 在标准数据集 BigCloneBench (BCB) 和 OJClone 上进行实验. 结果表明采用两种预训练增强的模型仅仅使用极少量的训练样例 (BCB 上 100 个克隆对和 100 个非克隆对, OJClone 上 200 个克隆对和 200 个非克隆对) 就能达到现有方法使用超过 6 百万个训练样例得到的结果.

关键词: 代码克隆; 预训练; LSTM

中图法分类号: TP311

中文引用格式: 冷林珊, 刘爽, 田承霖, 窦淑洁, 王赞, 张梅山. 预训练增强的代码克隆检测技术. 软件学报, 2022, 33(5): 1758–1773. http://www.jos.org.cn/1000-9825/6560.htm

英文引用格式: Leng LS, Liu S, Tian CL, Dou SJ, Wang Z, Zhang MS. Clone Detection with Pre-training Enhanced Code Representation. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1758–1773 (in Chinese). http://www.jos.org.cn/1000-9825/6560.htm

Clone Detection with Pre-training Enhanced Code Representation

LENG Lin-Shan[1], LIU Shuang[2], TIAN Cheng-Lin[2], DOU Shu-Jie[2], WANG Zan[1,2], ZHANG Mei-Shan[1]

[1](School of New Media and Communication, Tianjin University, Tianjin 300072, China)

[2](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

Abstract: Code clone detection is an important task in the software engineering community, it is particularly difficult to detect type-IV code clone, which have similar semantics but large syntax gap. Deep learning-based approaches have achieved promising performances on the detection of type-IV code clone, yet at the high-cost of using manually-annotated code clone pairs for supervision. This study proposes two simple and effective pretraining strategies to enhance the representation learning of code clone detection model based on deep learning, aiming to alleviate the requirement of the large-scale training dataset in supervised learning models. First, token embeddings models are pretrained with ngram subword enhancement, which helps the clone detection model to better represent out-of-vocabulary (OOV) tokens. Second, the function name prediction is adopted as an auxiliary task to pretrain clone detection model parameters from token to code fragments. With the two enhancement strategies, a model with more accurate code representation capability can be achieved, which is then used as the code representation model in clone detection and trained on the clone detection task with supervised learning. The experiments on the standard benchmark dataset BigCloneBench (BCB) and OJClone are conducedt, finding that the final model with

* 基金项目: 国家自然科学基金 (U1836214, 61802275); 天津大学自主创新基金 (2020XRG-0022)
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: 2021-08-12; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

only a very small number of training instances (i.e., 100 clones and 100 non-clones for BCB, 200 clones and 200 non-clones for OJClone) can give comparable performance than existing methods with over six million training instances.

Key words: code clone; pre-training; LSTM

1 背 景

代码克隆指的是在代码语句构成上或者语义上相似的代码片段, 其普遍存在于软件项目之中, 尤其在有众多参与者的大规模项目中存在的更多. 代码克隆产生的原因有很多, 主要是开发者在开发的过程中为了提高效率, 包括复制粘贴已有的代码片段并进行适当的增减语句或调换语句顺序, 或使用开发框架、设计模式等[1]. 代码克隆会导致 bug 传播[2], 增加代码维护成本. 现有研究表明, 20%–50% 的大型软件系统都包含代码克隆[3,4]. 因此, 准确地检测代码克隆对于软件开发和维护是至关重要的.

代码克隆检测问题一直在被广泛研究[5−11]. 其中较为流行的方法是从源代码和抽象语法树 (AST) 中提取特征, 通过计算特征的相似度判断是否为克隆对[9,12−16]. 近年来, 特征表示学习方法已经引起了学者们广泛的兴趣[10,11,15,17−19]. 这些方法通过获得词嵌入和复杂的神经网络结构来编码源代码或者 AST, 可以获得较好的检测效果.

代码克隆没有形式化定义, 较为公认的标准是将代码克隆分为 4 种类型[6,7]. Ⅰ型克隆表示两段代码在除了注释、布局和空格以外完全相同. Ⅱ型克隆指的是替换用户定义的标识符, 在注释、类型和布局上有变化, 但语法或结构相似的代码段. Ⅲ型克隆涉及代码片段的插入和删除, 经过进一步修改 (如添加或删除语句) 的复制片段, 以及对空格、标识符、布局、注释和类型的更改. 之前的工作已经较好地研究了这 3 种类型的克隆[12,14−16,19]. 本文主要研究Ⅳ型克隆, 即具有相似语义但语法结构有所差异的代码片段. Ⅳ型克隆的识别在本质上与Ⅰ–Ⅲ型克隆不同, IV 型克隆更偏向语义, 即代码片段看上去不相似, 但是都实现了相同或者相似的功能. 图 1 列出了一个Ⅳ型克隆的例子, 图 1(a) 和图 1(b) 展示了复制文件功能的两种不同实现方式, 这对代码片段在语法上差异较大, 但实现了相同功能, 故而是 IV 型代码克隆.

```
1  FileInputStream is =  new FileInputStream(source);
2  FileOutputStream os =  new FileOutputStream(dest);
3  byte[]  buf  = new byte[1024];
4  int  length;
5  while ((length = is.read(  buf)) > 0) {
6      os.write(  buf, 0, length);
7  }
8  is.close();
9  os.close();
```

(a) copyFileUsingStream (File source, File dest)

```
1  FileChannel  sourceChannel = new FileInputStream(source).getChannel();
2  FileChannel destChannel =  new FileOutputStream(dest).getChannel();
3  destChannel.transferFrom( sourceChannel, 0,  sourceChannel.size());
4  sourceChannel.close();
5  destChannel.close();
```

(b) copyFileUsingChannel (File source, File dest)

图 1 Ⅳ型克隆的一个例子

关于Ⅳ型克隆的检测问题已经有相关研究工作[9,10,20,21], 这些工作都采用了有监督学习的方法, 使用了经过标注的克隆对和非克隆对作为训练集. 例如, TBCCD[10]和 CDLH[9]方法利用 BCB 中标注的几百万个克隆对, 并将这些数据按照 8:1:1 的比例分成了训练集, 验证集和测试集以训练深度学习模型. 这些方法取得了较好的效果. 但是人工标注一个大规模的训练数据集代价较大, 通常需要对不同代码较为熟悉的有经验的程序员进行高质量的代码标注工作.

本文旨在减少克隆检测模型所需要的标注训练样本的数量. 为此, 我们提出了两种新颖的预训练方法, 即针对 token 级别的词嵌入以及函数名预测, 以提升深度学习模型的代码表征能力. 首先, 程序标识符, 例如图 1 中的“buf”“dest”等, 是由开发人员根据自己的习惯和理解定义的, 具有较高的多样性和差异性, 可以导致较为严重的 OOV (out-of-vocabulary) 问题: 即给出一个通过标准的 Word2Vec 预训练的词嵌入矩阵, 有较大比例的单词无法在矩阵中查询到. 针对该问题, 本文首次提出通过解决 OOV 问题来提升代码克隆检测准确性的研究. 我们提出采用基于子词组合嵌入的预训练方法[22]. 对于不在词表中的 token, 我们仍然可以通过子词组合获得其向量表示. 其次, 针对现有代码克隆检测方法需要大规模标注数据进行训练的问题, 我们提出利用无监督任务来辅助实现这个目

标. 本文选择函数名预测任务对神经网络的模型参数进行预训练. Ⅳ型代码克隆检测和函数名预测都是面向语义的, 编写良好的函数代码的语义可以通过函数名体现出来. 因此, 通过函数名预测任务预训练神经网络的参数, 使神经网络的参数学到函数体的语义信息表示. 而后在该参数的基础上进行代码克隆检测任务的训练, 利用函数名预测任务学到的语义表示提高代码克隆检测任务的准确率. 由于函数名预测任务的训练数据集无需人工标注, 因此该策略可以极大减少人工标注代价.

为了验证该想法, 我们将代码克隆检测任务定义为一个二分类问题, 采用一种被广泛应用的, 包含自注意力机制的双向长短时记忆神经网络 (AttBiLSTM)[23]作为代码片段表征模型. 我们用 AttBiLSTM 神经网络对输入代码进行编码, 比较编码向量, 然后对克隆和非克隆做出最终的预测. 为提高代码表征能力, 本文采用上述两种预训练策略对 AttBiLSTM 的参数进行预训练, 然后使用少量的代码克隆以及非克隆训练样本 在代码克隆检测任务上对模型进行微调.

我们分别在 BigCloneBench (BCB) 数据集[7]和 OJClone[20]上进行了实验来评估提出方法的有效性. BCB 数据集中包含超过 800 万个Ⅳ型克隆对, 同时也有标签不平衡问题[19]. OJClone 数据集由相同问题的答案代码片段集合构成, 同一个问题的不同答案代码片段互为克隆对, 不同问题的答案代码片段互为非克隆对. 同时, 我们采用 F-measure 和 AUC 分数来衡量模型性能. 实验结果表明, 本文提出的两种预训练策略都是有效的, 两者的结合可以进一步提高模型的性能. 最后, 我们在 BCB 中选取了 100 个克隆对和 100 个非克隆对训练模型, 与现有的有监督学习方法在数百万个训练样本上训练的结果相比, 可以达到相似甚至更好的克隆检测效果 (F-measure 值大于 96%). 在 OJClone 中选择了 200 个克隆对和 200 个非克隆对进行训练, 同样也可以达到较好的效果 (F-measure 值大于 96%).

2 相关工作

代码克隆检测是软件工程中的一个重要课题, 大量研究者对该问题进行了研究[5−8,11,12,17−19]. 代码克隆检测任务可以分成两个阶段进行, 即对待检测代码片段的表示, 以及在该表示上进行代码相似度度量. 因此代码表示是代码克隆检测的关键, 代码预处理的粒度也决定了检测方法的适用范围和准确度. 基于对代码表示的不同将克隆检测方法分为 4 类: 基于度量[16,18], 基于 token[12,14,15,24], 基于树或图的表示[9,13,17,25,26].

在基于度量的方法中, 用度量值来衡量以源代码作为输入的代码克隆. 对于函数或类等语法单元, 计算语句度量值, 然后比较这些度量值, 如果两个语法单元具有相同或相似的度量值, 则可以认为它们是克隆对. Svajlenko 等[27]介绍了 CloneWorks, 在进行克隆检测之前为用户提供了完整的源代码表示形式, 之后的克隆检测通过经过修改的 Jaccard 相似度进行, 如果一对代码片段满足给定的最小的阈值, 则判断为克隆对. Sudhamani 等人[28]提出了一种检测克隆的方法. 将源代码作为输入, 同时自定义公式, 根据公式度量相似性, 对 K=2 的相似值采用 K 均值聚类进行分组, 该方法对部分类型的克隆检测效果较好. Haque 等人[29]提出将代码划分为多个函数或模块来检测来自不同输入源代码的代码克隆. Ragkhitwetsagul 等人[30]提出了一种基于图像的克隆检测方法, 将 Java 源文件作为数据集, 以 PNG 图像为中间状态, 然后利用 Jaccard 相似性进行克隆检测. 结果表明, 该方法可以较好的识别出Ⅰ型、Ⅱ型和Ⅲ型克隆. Sudhamani 等人[31]使用结构控制语句解决结构相似性检测. 为此, 相似性检测使用 C/ C++, Java 文件为数据集和自定义公式. 这种方法可以有效地检测出结构相似的克隆. 基于度量的方法一般以源代码作为输入, 或者在源代码的基础上进行一些预处理, 根据预处理之后的代码之间的距离与设定的阈值比较判断是否为代码克隆, 虽然这种方法并不复杂, 但受限于在源代码上提取的度量信息, 只能检测Ⅰ型和Ⅱ型代码克隆; 另一方面, 只提取源代码的文本特征, 忽略了代码之间的结构和语义信息, 检测准确度较低, 且检测复杂度随源代码的长度增加而增大[1].

程序也可以被表示成 token 序列或词的集合进行代码克隆检测[12,24], 基于 token 的方法是在词法的基础上进行检测, 这些方法包括词法分析和克隆检测两个步骤. 它们在词法分析器的解析之后将目标源代码转换为一系列 tokens. 扫描 token 序列以找到 token 的重复子序列, 最后, 表示重复子序列的原始代码片段将作为克隆返回. Wang 等[15]开发了基于 token 的克隆检测器 CCAligner. 它使用 C、Java 文件作为数据集, 识别Ⅰ型、Ⅱ型和Ⅲ型克隆. Yuki 等人[32]提出了一种检测多粒度代码克隆的技术, 并使用 Smith-Waterman 算法来识别相同的哈希序列. 该方法可以检测Ⅰ型、Ⅱ型和Ⅲ型克隆. Semura 等人[33]开发了另一种克隆检测工具 CCFinderSW. 不同于只能检

测特定语言的代码克隆检测工具, 它还提供了按需添加语言的扩展机制, 并在其中发现了Ⅰ型和Ⅱ型克隆. Li 等人[18]提出了一种基于深度学习的克隆检测方法 CCLearner. 这种方法将 IJaDataset 转换为 token, 并基于深度学习模型进行克隆检测. 基于 token 的检测方法将源代码拆分为 token 序列, 根据序列进行代码克隆检测. 这种方法不需要提取复杂的特征, 可以在词汇层面获取到代码片段的信息, 但是将源代码表示成 token 序列, 也丧失了代码在整体结构层面的信息, 同时, 如果对代码片段进行调换语句顺序、增加或删除操作, 将会在很大程度上改变 token 序列, 从而使得检测方法无效.

近年来, 基于树的克隆检测方法受到广泛关注, 并取得了较好的检测效果. 在基于树的克隆检测技术中, 程序首先被转换成抽象语法树 (AST), 然后在 AST 上进行编码表示、或通过子树匹配等方法衡量相似性. Gao 等人[19]遍历 AST 的中间节点, 并利用 LSTM 学习 AST 表示, 进行代码克隆检测. TBCCD[10]提出了一种基于树的卷积神经网络检测语义克隆的新方法, 充分利用了代码片段的结构信息, 通过从 AST 中获得代码片段的结构信息和从 token 中获得词汇信息, 同时充分发挥了神经网络检测代码克隆的能力, 该方法可以有效检测出Ⅳ型代码克隆. TBCAA[34]将 API 调用序列融合到 AST 中以弥补由于方法调用提取而导致的语义缺失, 用 API 增强的 AST, 使用基于树的卷积方法作为代码表示学习的基本模块, 可以有效识别语义克隆. Yang 等人[35]提出了一种基于函数的代码克隆检测技术. 首先从函数中创建 AST, 将其转换为新的树状结构, 然后利用 Smith-Waterman 算法获得函数之间的相似度评分. CDLH[9]使用 Word2Vec 模型[36]来学习 token 嵌入来捕获词法信息, 基于 LSTM 模型[37]来训练语法树 (AST), 将 token 组合成二进制向量来表示一个代码片段, 但是此方法没有充分利用 AST 中的结构信息. 只使用 token 信息的 SourcererCC[24]和只使用结构信息的 Deckard[13]都无法准确检测到Ⅳ型克隆. ASTNN[38]将每个大型 AST 拆分为子语法树, 并通过捕获语句的词汇和语法知识将子语法树编码为向量. Wang 等人[39]构建了一种称为流增强抽象语法树 (FA-AST) 的程序图表示, 用显式控制和数据流边扩充原始 AST 构成 FA-AST, 然后在 FA-AST 上应用两种不同类型的图神经网络 (GNN) 来表征并度量代码对的相似性. 曾杰等人[40]首先抽取词法单元的特征表示, 然后分析不同单词的语义相似性, 抽取 AST, 为叶子节点赋予对应单词的特征表示, 将抽象语法树转化为特征向量树, 并将其编码为定长向量, 根据向量的相似与否判断是否为代码克隆.

基于图的克隆检测首先将代码片段转化为程序依赖图 (PDG) 等, 然后对 PDG 的结构是否相似做出判断. Zou 等人[41]提出了一种基于 PDG 的采用图核的编码克隆检测器 CCGraph, 首先对 PDG 的结构进行规范化, 设计了两阶段的过滤方法来度量代码的特征向量, 然后采用基于改进的 Weisfeiler-Lehman 图核的近似图匹配算法检测代码克隆. Feng 等人[42]提出了一个连接两个递归自动编码器和一个比较网络的孪生网络进行代码克隆检测, 设计无权值递归自动编码器学习代码表示, 然后利用比较网络进行相似性评估, 充分利用了词汇、语义和结构信息, 可以达到很高的精度. Yuan 等人[43]提出了一种新的语义克隆检测方法, 使用控制流图 (CFG) 作为程序方法的中间表示. 将语音识别领域的动态时间规整 (DTW) 算法与两种深度神经网络模型 (双向 RNN 自编码器和图卷积网络 (GCN)) 相结合, 从局部到全局检测语义层克隆. 基于树或图的比较方法需要生成抽象语法树、或者程序依赖图等, 这种方法可以捕获到代码的结构信息和语义, 但是基于树或图的比较时间复杂度和效率复杂度相比较基于度量和基于 token 的方法要高得多.

现有基于监督学习的方法使用人工标注的数据集例如 BigCloneBench[7]和 OJClone[20]进行监督学习. 这些模型实现了很高的性能, 但需要大规模的人工标注训练数据集, 人工标注代码克隆数据集, 特别是Ⅳ型克隆的代价较高. 近期, 大规模语料预训练模型在自然语言处理领域取得了良好进展. 在程序语言的表征学习上, 大规模预训练模型也受到了关注. 如 CodeBERT[44]等基于大规模代码库的预训练模型的发布, 使得代码表征能力得到了极大提升. 在代码克隆相关领域, 也有通过预训练模型提升代码表征的方法被提出. InferCode[45]将自然语言处理中的自监督学习思想引入到代码的抽象语法树的表示中, 通过预测从 AST 上下文中自动识别的子树来训练代码表示, AST 的子树被视为训练代码的标签, 无需人工标记工作, 该方法可以用于代码克隆检测. code2vec[46]首先提取出代码片段的抽象语法树, 从其中提取不同的路径, 用路径中每个节点的词向量连接成该路径的向量, 对于每个路径按照权重加权组成最终向量, 该向量就是对应方法名字的向量. 然而这些预训练模型通常参数规模庞大, 训练及使用代价高.

基于对现有方法的总结比较, 我们需要改进基于度量和基于 token 方法获得信息不全面, 基于图和树的方法

复杂度高, 以及基于大规模预训练语料的方法标注、训练代价大的问题. 克隆检测不仅需要提取出词汇级别的信息, 更需要结构和语义级别的信息来提高检测准确度, 同时对计算效率的要求高, 这导致目前最先进的句子匹配模型 (包括输入句子对之间) 大多不适用. 通过密切相关的任务进行预训练的想法已经广泛应用于自然语言处理[47−49]. 在本工作中, 我们应用该思想进行轻量级的语义预训练, 以提升代码表征能力.

3 预训练增强的代码克隆检测方法

本节将详细介绍本文提出的方法. 首先, 概括模型整体结构, 并详细介绍模型的 3 个主要组成部分, 即嵌入层、代码表示层及分类层. 然后着重介绍两种提升代码表征的增强方式, 即子词丰富和函数名预测.

本文主要关注 IV 型代码克隆检测问题, 将其定义为一个二分类任务. 给定一对代码片段 C^A 和 C^B, 如果 (C^A, C^B) 是一个 IV 型克隆对, 则标注 1, 如果它们不是 IV 型克隆对, 则标注 0.

3.1 方法整体结构

本文提出的方法整体结构如图 2 所示, 首先, 在嵌入层对获得的数据集进行分词处理. token 是代码理解的基本单位, 类似于 NLP 领域句子理解中的单词. 因此, 在神经网络中, token 在代码表示中的作用与单词在句子编码中的作用相同. token 嵌入通过 Word2Vec 模型进行预训练, 用从开源代码库 (如 Github) 中收集的代码片段语料库替换输入语料库, 将语料中的每个函数分割成 token 序列 ($w_1^A,\dots,w_m^A$ 和 $w_1^B,\dots,w_n^B$), 然后使用 token 序列对 Word2Vec 模型进行训练, 以获得每个 token 的向量, 表示为 ($x_1^A,\dots,x_m^A$ 和 $x_1^B,\dots,x_n^B$). 在代码表示层将向量表示的 token 序列作为输入, 使用 BiLSTM 进行编码, 得到 $h_1^{A\text{BiLSTM}},\dots,h_m^{A\text{BiLSTM}}$ 和 $h_1^{B\text{BiLSTM}},\dots,h_n^{B\text{BiLSTM}}$, 通过 self-attention 层对用向量序列表示的函数进行总结, 得到 $h^{A\text{CODE}}$ 和 $h^{B\text{CODE}}$. 在分类层, 计算 $h^{A\text{CODE}}$ 和 $h^{B\text{CODE}}$ 的距离, 判定两个函数是否为克隆函数. 因为在检测一对代码片段时, 对两个代码片段的编码方式完全相同, 因此我们使用 Siamese 模型结构, 即对两个代码片段的嵌入层预训练过程以及代码表示层的 BiLSTM 网络参数都完全相同. 为了方便阅读, 以下描述以一个代码片段的处理过程为例.

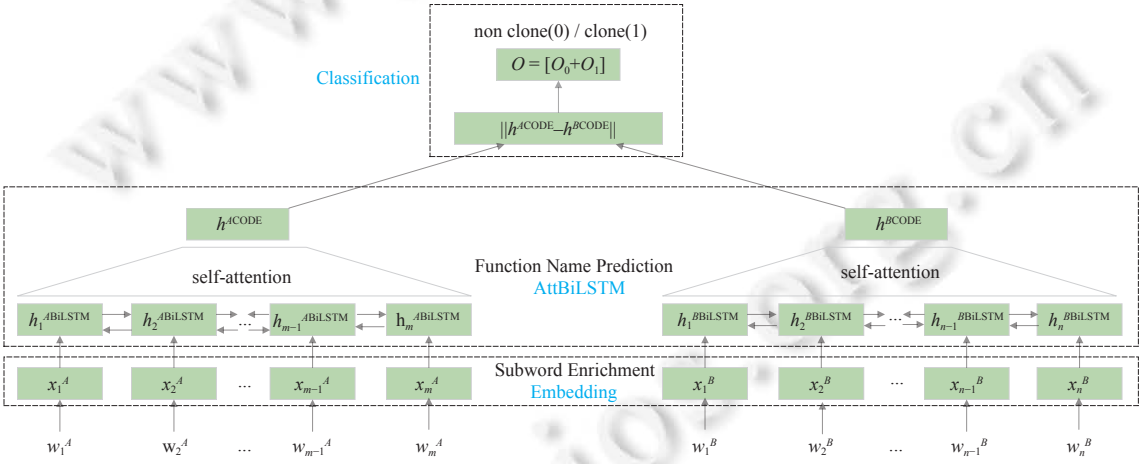

图 2 代码克隆检测模型结构

3.1.1 嵌入层

首先, 将每一个代码片段都分割成 token 序列. 对于片段 C^A, 经分割后得到 $w_1^A\dots w_m^A$, (m 为代码片段 A 的长度), 然后在矩阵 $\boldsymbol{E}_w$ 中查找每个 token 对应的向量, 将所有的 tokens w_i^A (i∈[1, m]) 转化为密集向量. 该查找矩阵已经用代码语料库进行预训练, 查找矩阵 $\boldsymbol{E}_w$ 在模型中是固定的. 查找过程可表示为:

$$x_i^A = lookup(w_i^A, \boldsymbol{E}_w) \tag{1}$$

经过嵌入层的预训练后, C^A 生成了序列 $x_1^A\dots x_m^A$. 对于代码片段 C^B, 可以用同样的嵌入方式得到对 token 序列 $w_1^B\dots w_n^B$ (n 为代码片段 B 的长度) 的向量表示 $x_1^B\dots x_n^B$.

3.1.2 代码表示层

AttBiLSTM 神经网络整合 token 嵌入形式的输入, 它由两部分组成: 双向长短时记忆部分 (BiLSTM) 和自注意部分. 前者的目的是获得一个高级上下文 token 表示序列, 后者的目的是总结序列级的输入特征, 并将每个代码片段缩减为一个单一的密集向量. 处理过程分为如下两个步骤:

$$\begin{aligned} h_1^{\text{ABiLSTM}},\ldots,h_m^{\text{ABiLSTM}} &= \text{BiLSTM}(x_1^A,\ldots,x_m^A),\\ h^{\text{ACODE}} &= \textit{Self-Attention}(h_1^{\text{ABiLSTM}},\ldots,h_m^{\text{ABiLSTM}}) \end{aligned} \tag{2}$$

经过嵌入层的预训练后, 每一个 token w_i^A 都被转化为了一个密集向量 x_i^A, 它们经过 BiLSTM 的编码后, 生成了 h_i^{ABiLSTM}, 然后经过 self-attention 层的总结后, 生成了 h^{ACODE}, h^{ACODE} 是 C^A 的最终表示.

同样, 我们可以用相同的计算流以 x_1^B …x_n^B 作为输入为 C^B 计算 h^{BCODE}. 在 Siamese BiLSTM 中, 神经网络 BiLSTM(·) 和 Self-Attention(·) 中有许多模型参数, C^A 和 C^B 共享 BiLSTM 和 self-attention 的模型参数.

3.1.3 分类层

当 h^{ACODE} 和 h^{BCODE} 被训练好后, 我们就可以通过以下过程计算克隆和非克隆的得分:

$$h^{\text{DIFF}} = \|h^{\text{ACODE}} - h^{\text{BCODE}}\|,\ O = [O_0, O_1] = W_{\text{CLONE}} h^{\text{DIFF}} \tag{3}$$

其中, h^{ACODE} 与 h^{BCODE} 的向量维度相同, h^{DIFF} 是由它们对应位置相减得到的向量, 也是分类的最终特征, 反映了两个代码片段之间的语义差别, O_0 和 O_1 分别表示这两段代码为克隆与非克隆的概率, W_{CLONE} 是分类参数. 本工作采用有监督模型的交叉熵损失训练或微调模型参数:

$$loss = -\log p_y = \log \frac{e^{O_y}}{e^{O_0} + e^{O_1}} \tag{4}$$

其中, y 代表输入为克隆或非克隆的真实标签. 训练的目标是最小化所有训练样例的累计损失, 通过不断地缩小累计损失来更新 O_0 和 O_1 的值, 最后不断更新 W_{CLONE}, 使模型达到最好的分类效果.

3.2 子词丰富

与自然语言中的单词相比, 编程语言中的 token 可以以更灵活和多样化的方式命名. 除程序语言关键字外, 标识符可以被命名为任何符合词法规则的字符串. 例如, 几乎所有合法的英语单词 (如“read”和“length”), 子词 (如“buf”和“dest”), 以及单词组合 (如“sourceChannel”) 都是合法的程序语言 tokens. 因此, 编程语言 tokens 的数量可能是无限大的. 另一方面, 编程语言预训练语料库的规模比自然语言小得多, 因为我们只能在少数的开源代码库中获得语料. 上述两个问题可能导致 token 表示中严重的 OOV 问题. 表示学习中的 OOV 问题已经在自然语言处理领域得到了广泛的研究. 典型的方法包括纯字符级模型, 如 Elmo[50], ngram 子词模型[22]和基于字节对编码的子词模型[51,52]. 我们针对代码表示中的 OOV 问题进行了检测, 发现在 BCB 数据集中 OOV 比率高达 62.68%, 在 OJClone 数据集中 OOV 比率达到了 16.82%.

为了解决上面提到的两个问题, 我们引入了 ngram 子词来丰富词语表示, 采用基于 ngram 的子词组合的 token 嵌入来解决 token 表示中的 OOV 问题. 对于在词表中预先训练过的单词, 直接利用预先训练的嵌入, 在矩阵 E_w 中查找向量作为 token 表示. 而对于 OOV 单词, 通过其 ngram 来组成单词表示. 该预处理方法是由 Bojanowski 等人[22]首次提出的对标准 Word2Vec 模型的简单扩展. 本文基于子词丰富的思想解决 OOV 问题, 以提升代码克隆检测模型的准确率.

为了方便, 本文不区分单词和 token, 因为 token 可以被视为编程语言中的单词. Word2Vec 的基本思想是利用语言建模来学习词 (tokens) 嵌入. Skip-Gram 模型通过预测给定单词的上下文单词来学习单词表示. 给定一个单词序列 $w_1,\ldots, w_i,\ldots, w_n$, 和一个源单词 w_i, 上下文窗口大小 c, 模型预测它周围的单词 $w_{i-c},\ldots, w_{i-1},\ldots, w_{i+1},\ldots, w_{i+c}$. 这个过程是一个典型的分类问题, 标签数量与词表大小相同. 尽管分类标签数量很大, 我们可以通过负采样来控制. 模型的目标交叉熵损失函数定义为:

$$loss = -\sum_{C} \log p(w_j|w_i) \tag{5}$$

在上述公式中, $C = 2c$, w_j 是 w_i 周围单词中的一个. Skip-Gram 模型使用一个简单的网络来计算公式 (5). 网络

有两个查找矩阵 E_w 和 E_f, 它们都是随机初始化的模型参数, 然后在训练过程中根据目标函数进行微调. 在大规模原始语料上进行训练后, 最终的嵌入矩阵 E_w 被输入到等式 (1) 中的克隆检测模型中.

给出一对源词和上下文单词 (w_i, w_j), 可以从 E_w 中得到源词嵌入 v_j, 从 E_f 中得到上下文词嵌入 u_i. 接下来, (w_i, w_j) 的相关分数由 u_i 和 v_j 的数积得到 (T 代表转置).

$$s(w_i, w_j) = u_i^{\mathrm{T}} v_j \tag{6}$$

分类概率的计算公式为:

$$p(w_i|w_j) = \frac{\mathrm{e}^{s(w_i, w_j)}}{\sum_{*} \mathrm{e}^{s(w_i, *)}} \tag{7}$$

其中, *代表任何其他的被随机负采样的词, 分母可以看作是概率计算的归一化因子.

对于子词组合, 唯一的区别是源词 w_i 的表示. 除了 w_i 的全词嵌入外, 我们还利用了一种由 w_i 的 ngram 合成表示. 例如, 对于单词“source”, 它的 4 个字符组合包括“sour”“ourc”和“urce”. 我们学习这些单词的嵌入, 然后由它们来组成 OOV 单词嵌入.

在预训练过程中, 源词 w_i 以如下方式被计算:

$$v_i = v_{w_i} + \frac{1}{Q} \sum_{i=1}^{Q} v_{ngram(w_i)} \tag{8}$$

ngram(.) 包括所有可能的限于固定长度 (本文中为 3 到 6) 范围的 ngram 子词, Q 是在 w_i 中 ngram 的总数. 在整合之后, 全词和 ngram 子词的嵌入可以共同学习.

在探索过程中, 对于完整的单词, 我们通过查阅 E_w 直接获得它们的嵌入. 而对于 OOV 单词, 我们使用公式 (8) 中被包含的 ngram 子词获得它们的嵌入. 例如, 对于 OOV 单词“sourceChannel”, 它的嵌入可以通过平均单词中包含的所有 ngram 嵌入来计算.

3.3 函数名预测

在进行Ⅳ型代码克隆检测时, 最直观的方法是使用经过标注的代码克隆对来训练神经网络, 在给出大量的代码片段对以及标签 (1 和 0 分别代表克隆和非克隆) 后, 通过训练深度学习模型来区分克隆与非克隆代码片段. 然而训练深度神经网络需要大量经过高质量标注的数据, 标注工作不仅费时而且需要很多专业人员. 为了解决标注的高成本问题, 我们设计了合理的预训练任务, 使用函数名预测辅助模型学习程序语义信息. 函数名预测任务与面向语义的 IV 型克隆检测任务高度相关. IV 型代码克隆检测判断两个代码片段是否语义相似. 对于没有任何拼写错误, 变量以及函数名符合命名规范的代码, 其函数名通常是对函数体语义的描述总结, 可以反映函数编写者的编程意图, 也可以反映函数体的语义信息, 因此本工作中我们选取函数名预测任务作为辅助预训练任务使深度学习模型的特征表示层学习到代码片段的语义, 进而通过少量有标注的代码克隆、非克隆数据微调模型参数, 达到较好的克隆检测效果.

函数名预测形式上类似于 Skip-Gram 模型. 给定一对代码片段 C^A 和它的函数名 N^B, 我们首先计算它们的向量表示. 如图 2 所示, 对于函数体表示 C^A, 利用与代码克隆检测模型相同的网络结构表示代码, 可以得到 h^{ACODE}. 对于函数名表示, 我们首先用几个简单的规则将函数名分割成一系列有意义的单词, 例如“copyFileUsingStream”被分割成“copy File Using Stream”, 然后在单词序列上使用一个简单的平均池化网络来获得其表示. 假设 $N^B = w_1, \ldots, w_l$, 它的向量表示以如下方式计算:

$$h_i^{\mathrm{NAME}} = \frac{\sum_{k=1}^{l} lookup(w_k, \boldsymbol{E}_{\mathrm{natural}})}{l} \tag{9}$$

查找矩阵 $\boldsymbol{E}_{\mathrm{natural}}$ 是自然语言词嵌入的集合. 这里我们直接使用一种已公开发布的 Glove 词嵌入[53]. 在本文中, 词嵌入是固定的, 所以这部分没有任何模型参数需要训练.

接下来, 我们可以通过它们的表示计算 C^A 和 N^B 的相关分数:

$$s(C^A, N^B) = (h^{\mathrm{ACODE}})^{\mathrm{T}} h_i^{\mathrm{NAME}} \tag{10}$$

在给定 C^A 条件下, N^B 的概率以如下方式计算:

$$p(N^B|C^A)=\frac{e^{s(C^A,N^B)}}{\sum_* e^{s(C^A,N^*)}} \tag{11}$$

相关分数和概率公式分别与 Skip-Gram 模型的公式 (6) 和公式 (7) 高度相似. 因为分类标签的数量巨大 (与合法的函数名总数相同), 在这里, 同样应用负采样方法进行有效的概率计算. 例如, 我们可以抽样 5 个函数名来近似分母, 这个方法大大降低了计算成本.

最后, 我们使用下面的损失函数 (还有交叉熵损失) 作为训练目标:

$$loss=-p(N^B|C^A) \tag{12}$$

克隆检测模型中的参数与函数名预测模型中的参数完全相同. 模型预训练的效果来源于大规模的函数名预测任务, 因为构建这样的语料库不需要人工标注. 我们最终目标是预训练 AttBiLSTM 参数, 将预训练得到的参数应用在克隆检测模型中. 我们简单地使用预训练的网络参数来初始化神经克隆检测模型的 AttBiLSTM 部分, 然后根据克隆检测目标对这些参数进行调整. 该过程类似于 CodeBERT, 但区别于 CodeBERT, 本文采用相似任务进行预训练, 在训练数据集以及模型参数上, 较 CodeBERT 都更为轻量级.

4 实 验

4.1 实验设置

本工作所有实验采用的服务器的配置为 2 Intel Xeon Platinum 8260 CPU @2.30 GHz, 8 NVIDIA GeForce RTX2080 Ti GPU (每块显卡内存为 11 GB), 512 GB RAM, PyTorch 版本为 1.8.0.

4.1.1 克隆数据集

本实验采用 BigCloneBench 数据集[7]来评估本文提出方法的有效性. 该数据集是一个被广泛使用的 Java 代码克隆检测的基准[9,18], 它包含 8 654 345 个标注好的真实代码克隆对, 其中 8 219 320 个为Ⅳ型克隆对 (占比 95.00%), 279 032 个为非克隆对. 训练集为随机采样的 100 个 IV 型克隆对和 100 个非克隆对, 其他的 IV 型克隆对和非克隆对用于测试. 为了减少单一数据集对实验结果造成的影响, 我们还使用 OJClone 数据集[20]对我们的方法进行评估. 该数据集是由 OnlineJudge[20]上的编程问题的正确答案组成, 由于每个问题有多种正确答案, 同一问题两种不同的答案可以看作一对克隆对, 不同问题的答案可以作为非克隆对.

4.1.2 预训练语料构造

对于 BigCloneBench, 为了预训练 token 嵌入和代码表示, 我们从 GitHub 收集了 329 个高质量的 Java 项目 (按照获得 Star 的数量排序). 总共有 296 300 个文件和 2 097 213 个方法. 经过处理, 语料中包含超过 1.9 亿个 tokens, 其中有 2 489 036 个不同的 tokens, 构成了词表. 在 BCB 数据集中, 仅有 928 908 个 token 可以在该预训练得到的词表中搜到, 这导致了 BigCloneBench 上的 OOV 比率为 62.68%.

对于 OJClone 数据集, 我们用同样的思路选取 Github 上 Star 数量较高的 C 语言项目, 提取项目中所有 C 语言函数的函数名以及函数体, 使用这些语料信息进行预训练. 但排名较高的 C 语言项目多数为底层功能开发, 如操作系统的部分实现, 其代码实现逻辑及命名方式等与 OJClone 数据集中的代码片段 (变量名多为单个字符) 有较大区别, 这导致无法得到 OJClone 数据集中 token 的准确向量表示. 为了验证函数名预测预训练任务的有效性, 我们尝试使用 OJClone 数据集中的部分数据进行预训练, 我们在 15 个问题的每个问题中随机抽取 400 个答案, 将其作为预训练任务中的函数体. 因为 OJClone 数据集中并没有每个问题的描述, 且答案代码片段中的函数名全部为 main, 无法提供准确的语义信息. 为了得到对应这些函数体的函数名, 两位志愿者通过阅读代码, 根据代码语义为每个问题标注函数命名. 对每一个问题, 随机抽取 100 个答案, 两人阅读这些答案代码片段以后, 分别根据代码片段解决的问题给出函数名, 并通过讨论确定每个代码片段对应的函数名. 这 15 个问题的函数名包括 BubbleSort, getDays 等, 该标注方法最大程度地保证了函数名的准确性, 最终得到了 6 000 个正样本以及 6000×5 个负样本. 在 OJClone 数据集中我们所使用的预训练语料共有 59 311 个不同的 tokens, 而在整个数据集中有 71 306 个不同的 tokens, OOV 比率为 16.82%.

4.1.3 参数设置

参数设置包括 token 嵌入预训练, 函数名预测训练和代码克隆检测等几个训练任务的超参数. 对于 token 嵌入预训练, 我们直接使用作者发布的 fastText 工具[22], 其中向量维度设置为 100, 其他的超参数为该工具默认值. 对于代码表示的 BiLSTM 网络结构, 所有隐藏层的维度大小设置为 300. 在函数名预测和代码克隆检测中, 输入嵌入层采用 dropout[43], LSTM 隐藏层采用 Adam 算法[46]进行参数优化, 初始学习率为 5×10^{-4}, 梯度剪切阈值为 5, mini-batch 大小为 32. 我们将训练周期和负采样的大小分别设置为 20 和 5, 对于代码克隆检测模型训练 epoch 设置为 100.

对于本文采用给的对比方法, 我们采用原文中报告的参数设置. 对于 TBCCD, 我们采用其网站上开源的数据集划分方式, 即将 BCB 数据集按照 4:1:1 的比例划分为训练集、测试集和验证集. 卷积核大小为 600, 滑动窗口大小为 2, 全连接层的维度为 50, 使用随机梯度下降进行优化. 训练一共进行 10 轮, batch size 设置为 1. 对于 ASTNN, 将 BCB 数据集按照 4:1:1 的比例划分为训练集、测试集和验证集, 嵌入大小为 128, ST-tree 和双向 GRU 的维度为 100, 使用 AdaMax 方法进行优化. 训练过程一共进行了 2 轮, batch size 为 64, 克隆检测的阈值设置为 0.5, 学习率为 0.002.

4.1.4 评估指标

在我们的测试中, 克隆和非克隆的比例是极不平衡的, 我们对于克隆和非克隆独立报告了精度 (P), 召回率 (R) 和相应的 F-measure 值, 并且在两种类型上使用平均 F-measure 值 (F_{avg}) 作为主要度量依据. 与总体精度相比 F_{avg} 更合适. 以克隆对的计算为例, P, R, F-measure 分别以如下方式计算:

$$P=\frac{TP}{TP+FP} \tag{13}$$

$$R=\frac{TP}{TP+FN} \tag{14}$$

$$F\text{-}measure=\frac{2PR}{(P+R)} \tag{15}$$

其中, TP, FP, FN 分别为检测出的克隆对中为真实克隆对的数量, 检测出的克隆对中并不是克隆对的数量以及真实克隆对中未被检测出来的数量. 除此之外, 我们也报告了 AUC 值, 因为克隆与非克隆的边界可以根据实际需要手动重置. 为尽量消除使用随机种子初始化模型参数对克隆检测模型的影响, 我们在每个配置上运行了 5 次实验, 并计算其平均值.

4.2 实验结果

为了找到可以平衡准确率以及时间成本的训练集大小, 我们比较了不同训练数据集大小的实验效果. 为了探究预训练数据集大小对实验结果的影响, 我们在 C 语言预训练数据集上进行了实验, 比较了不同数据集大小对实验结果的影响. 为了比较两种预训练方法对代码克隆任务的提升效果, 实验设计在基准模型的基础上分别增加了子词丰富、函数名预测任务, 以及同时增加两个预训练任务. 为了与现有的主流方法进行比较, 我们选择了两个当前克隆检测效果最好的模型, 即 TBCCD 和 ASTNN 进行比较. 同时我们也对模型的效率进行了测试.

4.2.1 训练集大小对实验结果的影响

首先, 我们在克隆检测模型中检查训练规模的影响, 因为我们的主要目的是使用增强的代码表示来减少人工标注的需求. 我们在实验中使用的克隆对和非克隆对的比例均为 1:1, 在比较过程中使用的克隆对的数量分别是 20, 50, 100, 200, 500, 1 000. 表 1 展示了随着训练数据的增加, F 值的具体数据和增加的趋势. 可以看到, 训练规模越大, F 值表现越好. 当训练量达到 100 时, 增加训练样本数所得到的效果提升非常不明显. 在后续实验中, 我们选择了数据集中克隆对大小为 100 的设置.

我们同样在 OJClone 数据集上进行了训练数据大小对实验结果影响的比较实验, 结果如表 2 所示: 逐渐将训练数据增加到 1 000 个时, F_{clone} 值逐渐增加到 97.58%, 但是从表中可以观察到, 到训练数据增加到 200 条时, F_{clone} 值已经达到了 96.47%, 并且当逐渐增加数据时, 效果提升地较为缓慢, 因此以下关于 OJClone 的实验我们均采用 200 对作为训练集的大小 (200 对克隆代码对和 200 对非克隆代码对).

表 1 BCB 训练集上训练大小的影响 (%)

Training size (对)	F_{clone}	$F_{non\text{-}clone}$	F_{avg}	AUC
20	78.41	13.08	45.75	83.20
50	87.12	30.44	58.18	93.15
100	96.85	50.88	73.87	97.88
200	97.72	61.03	79.38	97.91
500	98.88	74.80	86.84	97.92
1000	98.91	75.42	87.17	97.94

表 2 OJClone 训练集上训练大小的影响 (%)

Training size (对)	F_{clone}	$F_{non\text{-}clone}$	F_{avg}	AUC
20	81.39	77.64	79.52	76.34
50	86.81	85.15	85.98	87.40
100	91.67	90.98	91.33	93.42
200	96.47	96.37	96.42	98.86
1000	97.58	97.67	97.72	99.48

4.2.2 预训练数据集大小对实验结果的影响

为了探究预训练语料的构造规模和训练情况对 BCB 数据集结果的影响, 我们随机选取了 20 万、50 万、100 万、200 万对克隆代码片段对模型进行预训练, 实验结果如表 3 所示.

表 3 BCB 预训练集上预训练大小的影响 (%)

Pre-training size (万对)	F_{clone}	$F_{non\text{-}clone}$	F_{avg}	AUC
20	95.84	45.02	70.37	97.09
50	96.12	47.34	72.03	97.25
100	96.40	48.86	72.85	97.38
200	96.85	50.88	73.87	97.88

表 3 表明随着预训练数据大小的增加, 最终结果也呈上升趋势, 为了平衡训练时间与训练效果, 最终选择数据集大小为 200 万对的设置.

我们同样在 OJClone 上进行了预训练数据集大小对实验结果影响的实验. 我们分别在 15 个问题中选择 100, 200, 300 个答案, 按照正负样本 1:5 的比例构造预训练数据集对模型进行函数名预测的预训练, 实验结果如表 4 所示.

从表 4 可以看出, 随着预训练数据集的增大, 克隆检测任务的 F 值和 AUC 值都会呈现上升趋势, 即对代码克隆的检测效果会不断增强. 考虑到需要预留出足够的答案样本用来构造训练集和测试集, 我们选择 400 作为 C 语言实验中的预训练数据集大小.

表 4 C 语言实验中预训练数据集大小对实验结果的影响 (%)

Pre-training size (对)	Clone			Non-clone			F_{avg}	AUC
	P	R	F	P	R	F		
100	90.75	92.40	91.57	92.32	90.58	91.45	91.51	93.62
200	92.55	94.63	93.59	94.51	92.38	93.44	93.52	95.79
300	93.60	96.41	94.50	96.30	93.41	94.86	94.68	97.12
400	95.26	97.70	96.47	97.64	95.14	96.37	96.42	98.86

4.2.3 不同增强效果比较

对于 BigCloneBench 数据集, 我们从训练池中选取 100 个克隆语料和 100 个非克隆语料作为主要设置, 目的是在最小化训练语料规模的同时达到良好的性能. 表 5 展示了结果. 我们报告了克隆和非克隆的 P, R 和 F 得分, 并报告了 F 得分的平均值和 AUC 值. Baseline 一行表示使用标准的基于全词嵌入的模型, +子词丰富一行表示使用 ngram 子词丰富嵌入的方法, +函数名预测表示在函数名预测任务上使用 AttBiLSTM 预训练参数的方法. 最后的模型整合了子词丰富和函数名预测预训练两种提升方法.

如表 5 所示, 可以发现两种提升方法都是高度有效的. 对于克隆对的 F-score, 两种预训练方法可以分别获得 9.72 和 10.58 的改进效果. 对于 non-clone 的 F-score, 提升结果分别为 23.67 和 29.11. +函数名预测的预训练结果好于+子词丰富的结果. 当同时采用两种方法时, 可以获得更好的效果. F_{avg} 和 AUC 在 4 个模型上的效果变化也与 F_{clone} 和 $F_{non\text{-}clone}$ 相同.

对于 OJClone 数据集, 本实验的训练集的构造方式为选择每个问题的 100 个与预训练数据集不同的答案, 在

这些答案中随机选择 200 对克隆代码对和 200 对非克隆代码对组成训练集, 之所以选择 200 作为训练集大小, 是因为经过对上文中关于训练集大小的探究实验后, 我们发现训练集大小为 200 可以最大程度地兼顾代码克隆检测效果和实验过程的时间成本. 测试集的大小为克隆对与非克隆对各 15 万对, 构造方式为在上述 100 个答案组成的集合中去除了训练集的 200 对克隆与 200 对非克隆后进行随机挑选. 测试结果如表 6.

表 5 在 BCB 上 100 个克隆对和 100 个非克隆对的结果 (%)

Model	Clone			Non-clone			F_{avg}	AUC
	P	R	F	P	R	F		
Baseline	99.74	75.75	86.10	11.35	94.36	20.26	53.18	96.51
+子词丰富	99.84	92.11	95.82	28.51	95.69	43.93	69.86	97.12
+函数名预测	99.83	93.73	96.68	33.32	95.27	49.37	73.03	97.21
Final(+both)	**99.87**	**94.01**	**96.85**	**34.58**	**96.25**	**50.88**	**73.87**	**97.88**

表 6 在 OJClone 上 200 个克隆对和 200 个非克隆对的结果 (%)

Model	Clone			Non-clone			F_{avg}	AUC
	P	R	F	P	R	F		
Baseline	82.11	84.68	83.38	84.19	81.54	82.84	83.11	87.80
+子词丰富	86.29	90.27	88.24	89.80	85.66	87.68	87.96	90.97
+函数名预测	93.05	96.53	94.76	96.40	92.79	94.56	94.66	98.25
Final(+both)	**95.26**	**97.70**	**96.47**	**97.64**	**95.14**	**96.37**	**96.42**	**98.86**

我们注意到在 BCB 数据集上所有的 4 个模型上 $F_{\text{non-clone}}$ 的值都要显著低于 F_{clone} 的值, 而在 OJClone 数据集上 $F_{\text{non-clone}}$ 的值与 F_{clone} 的值却相差不多, 主要原因是在 BCB 数据集上的克隆与非克隆之间存在严重的数据不平衡 (接近 30:1). 非克隆的精确值偏低, 是由于部分克隆分类错误造成的. 虽然百分比值很小, 但它大大增加了预测的非克隆数量. 这也是我们不使用精确度作为度量标准的原因.

在表 5 和表 6 中, 可以看到在 BCB 数据集和 OJClone 数据集上我们最终模型 (Final) 的效果要远好于 Baseline, 这得益于我们提出的两种预训练策略. 通过对两个数据集上的实验效果以及在实验过程的案例进行分析, 我们初步总结出了两种预训练策略分别对应的适用范围如下所示.

(1) 子词丰富

(a) 对于变量名由较为完整的英文单词组合构成的情况尤为有效. 例如在测试过程中出现了变量名“cardSlot”, 但在训练过程中并未见过该变量名, 因而其是 OOV 词汇. 因为字典中有“card”和“slot”两个词的向量, 子词丰富方法可以通过组合这两个单词的向量来表示“cardSlot”.

(b) 对于例如由单个字母构成的没有任何语意信息的变量名, 例如图 3 中的 (来自 OJClone 数据集的一个代码片段) 变量名 n, f 等, 子词丰富的效果较差. 实际上 OJClone 数据集中有较多这种单个字母作为变量名的用法, 这也是 OJClone 上子词丰富策略对最终分类结果提升效果没有在 BCB 上高的原因之一.

(2) 函数名预测

对于两个功能相同, 但是在语法上相似度较低的代码片段尤为有效, 例如图 4 中的两个代码段, 均为拷贝文件功能, 但是在代码实现上差距较大. 函数名预测任务能很好的学习到代码片段的语义, 从而提高代码克隆检测的准确率.

4.2.4 与其他方法的比较

我们也将我们的模型与现有效果最优的方法进行了比较. 表 7 展示了比较结果. 我们列出了基本方法的结果和我们最终模型 (即加子词丰富及预训练提升) 的结果. TBCCD[10]是基于抽象语法树的检测方法, 但是还利用了卷积树. ASTNN[38]将 AST 拆分为子树进行检测. 这两个模型使用来自 BigCloneBench 的 800 万个实例作为训练语料库. 我们使用原文的设置对 TBCCD 和 ASTNN 进行实验, 具体参数设置见第 4.1.3 节. 对于本文提出的方法, 仍采用 100 对克隆和 100 对非克隆进行训练.

```
int ifsushu(int n){
    if(n == 1)
        return 1;
    else
    {
        for(int f = n - 1;f >= 2; f--)
        {
            if(n % f == 0) break;
            if (f == 2)
                return 1;
        }
    }
    return 0;
}
```

图 3 子词丰富的一个不适用例子

```
public void  CopyFileUsingStream(File src, File dst){
InputStream in = new FileInputStream(src);
OutputStream out = new FileOutputStream(dst);
byte[] buf = new byte[1024];
int len;
while((len = in.read(buf)) > 0)
      out.write(buf, 0, len);
in.close();
out.close();
}
```

(a) copyFileUsingStream (File src, File dst)

```
public void Copy(File is, File sw){
InputStream is = getStream();
StringWriter sw = new StringWriter();
IOUtils.copy(is, sw, "UTF-8");
IOUtils.closeQuietly(is);
return sw.toString();
}
```

(b) Copy (File is, File sw)

图 4 函数名预测的有效例子

在 OJClone 数据集上的 TBCCD 和 ASTNN 的两种方法的结果为原文中给出的结果, 从表 7 可以看出, 在 BCB 数据集上, 我们的效果好于上述两种方法, 而在 OJClone 数据集上, 我们的方法 *F* 值要优于 ASTNN, 稍差于 TBCCD. 但是我们的方法仅仅才用了 400 对标注克隆数据进行训练, 而 TBCCD 则采用了超过 200 万对标注数据, 总体上, 我们的方法对于标注的数据集稀缺的代码克隆检测任务是很有必要的.

表 7 与相关研究工作的比较

Dataset model	BCB (%)			OJClone (%)		
	P	*R*	*F*	*P*	*R*	*F*
Baseline	99.74	75.75	86.10	82.11	84.68	83.38
Ours	**99.87**	**94.01**	**96.85**	95.26	97.70	96.47
TBCCD[10]	95.43	94.03	94.73	**99**	**99**	**99**
ASTNN[38]	96.00	92.30	94.10	98.9	92.7	95.5

4.2.5 效 率

表 8 展示了本文提出方法与对比方法 TBCCD, ASTNN 的时间开销对比. 本文提出方法的训练时间为 27 min, 测试 100 对所需时间为 0.024 s, 在这个过程中只需要 100 对克隆对和 100 对非克隆对的训练. TBCCD 训练所需时间为 2 640 min, 测试 100 对所需时间为 1.36 s. ASTNN 训练时间为 9 360 min, 测试 100 对所需时间为 12.62 s. 时间较长的原因为是在训练阶段及测试阶段, 都需要先将源代码表示成抽象语法树, 然而这个过程很耗时. 从结果中我们可以看到, 我们的方法测试时间仅为 TBCCD 的 1.8%, ASTNN 的 0.19%. 本文函数名预测预训练任务训练时间为 8 640 min. 该预训练任务无需人工干预, 且仅需进行一次. 因而对比人工标注大量代码克隆对的代价, 该预训练的代价可以接受.

表 8 不同方法的效率对比结果

方法	训练 (min)	测试 (s/100对)	预训练 (min)
Ours	27	0.024	8 640
TBCCD	2 640	1.36	—
ASTNN	9 360	12.62	—

4.2.6 抄袭检测应用

为了进一步验证本文方法的实际可行性, 我们将其应用于代码作业的抄袭检测. 实验样本为天津大学智算学部编译原理课程作业, 该作业要求学生实现 C 语言编译器中词法分析以及语法分析的功能. 由于作业编码语言多

样, 包括 C/C++, Python 等, 我们选取 10 组用 C/C++语言编码的作业作为检测目标.

首先将每一份作业代码按照函数进行拆分, 10 份作业共得到 123 个函数, 6 298 对代码对 (每份作业的每一个函数与其他作业的一个函数形成一个代码对). 我们使用在 OJClone 数据集上训练好的模型对这些克隆片段进行预测, 并对预测结果为克隆的概率在 80% 以上的函数片段所在的作业进行全面的人工检查.

经过克隆检测模型的检测, 我们的方法预测出 850 对克隆, 召回率为 96.30%, *F*1 值为 74.82%. 我们直接采用 OJClone 数据集上训练好的模型进行克隆检测, 由于: (1) OJClone 数据集中的变量、函数等的命名与待检测代码有差距, OJClone 数据集中的代码经常用单个字母, 如 i, j, k 等命名, 而作业数据集的命名较为规范, 因而子词丰富策略对作业数据集作用不明显; (2) 我们未用作业代码片段微调检测模型; 因而检测结果准确率没有在 OJClone 数据集上测试的结果高. 两份作业之间最多包含 4 对克隆代码片段, 最少的为不包含克隆代码片段. 人工检查结果发现十组作业中并未有抄袭, 其出现克隆代码对的原因为在词法分析和语法分析的过程中, 部分功能或算法是固定的, 例如词法分析中判断该 token 是否为关键字和数字等, 语法分析中构建 first 集和 follow 集的算法等. 通过代码克隆检测模型的辅助, 我们仅需要人工检查 8 对作业, 减少了 82% 的工作量. 该应用案例证实了我们方法的实用性和有效性.

4.3 有效性威胁

4.3.1 内部威胁

本实验代码共由 3 部分组成: 数据的处理过程, 代码的表示过程以及最终的分类过程. 3 部分的代码的主体逻辑都由作者自己实现. 代码实现中存在潜在的 bug, 可能会影响方法的准确性. 为了降低该内部威胁, 代码中重要的实现部分逻辑由主要作者仔细检查, 并进行了充分的测试. 另外, 由于 OJClone 数据集中并没有给出可用的函数名, 为了进行函数名预测任务, 我们人工对函数进行命名, 为了避免命名错误或不准确等问题, 我们对每个问题至少阅读了 100 个答案并且经过所有作者的讨论之后, 最终确定了该问题所对应的函数名.

4.3.2 外部威胁

本实验调用了一些第三方库, 例如 PyTorch 等, 这些第三方库潜在的 bug 可能会对本工作的实验结果构成威胁. 为了降低该部分的威胁, 本工作选取被业界广泛使用的, 经过充分测试的第三方库函数.

4.3.3 构造威胁

数据集构造上, 在阅读了大量的论文之后, 我们决定使用大多数实验所采用的数据集 BigCloneBench (BCB), 以保证与其他相关工作尽可能相同的实验设置. 为了尽量避免 BCB 数据集构建时所采用启发式搜索带来的潜在数据标注错误问题对该实验结果的影响, 我们还使用了 OJClone 数据集对我们的方法进行准确性评估. 在 BCB 数据集上的函数名预测部分, 所使用的数据集为从 Github 上获取的代码, 为了尽可能的提高函数名预测的准确度, 我们将 Github 中的项目按照其所获得的 Star 数量进行排序, 并获取前 329 个项目的代码, 从而确保我们所获得的数据集是高质量的.

5 总结与展望

在本文中, 我们研究了低资源设置下的代码克隆检测. 我们提出了两种预训练策略来增强代码表示, (1) 使用 token 嵌入的子词丰富, (2) 对从 token 组合到代码片段的函数名预测. 使用增强代码表示, 我们可以用最小的训练语料库训练出一个强大的代码克隆检测模型. 在 BigCloneBench 和 OJClone 数据集上的实验结果表明, 我们提出的两种策略对Ⅳ型代码克隆的检测是有效的, 同时性能上有所提升. 我们的模型只用极少个训练实例 (BCB 上 100 个克隆对和 100 个非克隆对, OJClone 上 200 个克隆对和 200 个非克隆对) 可以达到, 甚至胜过以前使用数百万个训练实例的监督模型的效果, 使用更少的数据进行训练可以减少人力物力的消耗. 本文采用的是轻量级、针对性的预训练方法, 与大规模预训练模型 CodeBERT 的对比显示, 我们的方法取得了更优的效果.

References:

[1] Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 962–980

(in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[2] Vislavski T, Rakić G, Cardozo N, Budimac Z. LICCA: A tool for cross-language clone detection. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, 2018. 512–516. [doi: 10.1109/SANER.2018.8330250]

[3] 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]

[4] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proc. of the IEEE Int'l Conf. on Software Maintenance-1999 (ICSM '99). Software Maintenance for Business Change (Cat. No. 99CB36360). Oxford: IEEE, 1999. 109–118. [doi: 10.1109/ICSM.1999.792593]

[5] Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Proc. of the Int'l Conf. on Software Maintenance (Cat. No. 98CB36272). Bethesda: IEEE, 1998. 368–377. [doi: 10.1109/ICSM.1998.738528]

[6] Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. IEEE Trans. on Software Engineering, 2007, 33(9): 577–591. [doi: 10.1109/TSE.2007.70725]

[7] 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 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. Victoria: IEEE, 2014. 476–480. [doi: 10.1109/ICSME.2014.77]

[8] Milea NA, Jiang LX, Khoo SC. Vector abstraction and concretization for scalable detection of refactorings. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 86–97. [doi: 10.1145/2635868.2635926]

[9] Wei HH, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. Melbourne: AAAI Press, 2017. 3034–3040.

[10] Yu H, Lam W, Chen L, Li G, Xie T, Wang QX. Neural detection of semantic code clones via tree-based convolution. In: Proc. of the IEEE/ACM 27th Int'l Conf. on Program Comprehension (ICPC). Montreal: IEEE, 2019. 70–80. [doi: 10.1109/ICPC.2019.00021]

[11] Zhang YY, Li M. Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector. Proc. of the AAAI Conf. on Artificial Intelligence, 2019, 33(1): 5813–5820. [doi: 10.1609/aaai.v33i01.33015813]

[12] 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]

[13] 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 (ICSE'07). Minneapolis: IEEE, 2007. 96–105. [doi: 10.1109/ICSE.2007.30]

[14] 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: IEEE, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

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

[16] Saini V, Farmahinifarahani F, Lu YD, Baldi P, Lopes CV. Oreo: Detection of clones in the twilight zone. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena: ACM, 2018. 354–365. [doi: 10.1145/3236024.3236026]

[17] 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 (ASE). Singapore: IEEE, 2016. 87–98.

[18] Li LQ, Feng H, Zhuang WJ, Meng N, Ryder B. CCLearner: A deep learning-based clone detection approach. In: Proc. of the 2017 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Shanghai: IEEE, 2017. 249–260. [doi: 10.1109/ICSME.2017.46]

[19] Gao Y, Wang Z, Liu S, Yang L, Sang W, Cai YF. TECCD: A tree embedding approach for code clone detection. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Cleveland: IEEE, 2019. 145–156. [doi: 10.1109/ICSME.2019.00025]

[20] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence. Phoenix: AAAI, 2016. 1287–1293.

[21] Wei HH, Li M. Positive and unlabeled learning for detecting software functional clones with adversarial training. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. Stockholm: AAAI, 2018. 2840–2846.

[22] 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]

[23] Lin ZH, Feng MW, dos Santos CN, Yu M, Xiang B, Zhou BW, Bengio Y. A structured self-attentive sentence embedding. arXiv: 1703.03130, 2017.

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

[25] 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]

[26] Büch L, Andrzejak A. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Hangzhou: IEEE, 2019. 95–104. [doi: 10.1109/SANER.2019.8668039]

[27] Svajlenko J, Roy CK. Fast and flexible large-scale clone detection with CloneWorks. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering Companion (ICSE-C). Buenos Aires: IEEE, 2017. 27–30. [doi: 10.1109/ICSE-C.2017.3]

[28] Sudhamani M, Rangarajan L. Code clone detection based on order and content of control statements. In: Proc. of the 2nd Int'l Conf. on Contemporary Computing and Informatics (IC3I). Greater Noida: IEEE, 2016. 59–64. [doi: 10.1109/IC3I.2016.7917935]

[29] Haque SMF, Srikanth V, Reddy ES. Generic code cloning method for detection of clone code in software development. In: Proc. of the 2016 Int'l Conf. on Data Mining and Advanced Computing (SAPIENCE). Ernakulam: IEEE, 2016. 335–339. [doi: 10.1109/SAPIENCE.2016.7684149]

[30] Ragkhitwetsagul C, Krinke J, Marnette B. A picture is worth a thousand words: Code clone detection based on image similarity. In: Proc. of the 12th IEEE Int'l Workshop on Software Clones (IWSC). Campobasso: IEEE, 2018. 44–50. [doi: 10.1109/IWSC.2018.8327318]

[31] Sudhamani M, Rangarajan L. Structural similarity detection using structure of control statements. Procedia Computer Science, 2015, 46: 892–899. [doi: 10.1016/j.procs.2015.02.159]

[32] Yuki Y, Higo Y, Kusumoto S. A technique to detect multi-grained code clones. In: Proc. of the 11th IEEE Int'l Workshop on Software Clones (IWSC). Klagenfurt: IEEE, 2017. 1–7. [doi: 10.1109/IWSC.2017.7880510]

[33] Semura Y, Yoshida N, Choi E, Inoue K. CCFinderSW: Clone detection tool with flexible multilingual tokenization. In: Proc. of the 24th Asia-Pacific Software Engineering Conf. (APSEC). Nanjing: IEEE, 2017. 654–659. [doi: 10.1109/APSEC.2017.80]

[34] Chen L, Ye W, Zhang SK. Capturing source code semantics via tree-based convolution over API-enhanced AST. In: Proc. of the 16th ACM Int'l Conf. on Computing Frontiers. Alghero: ACM, 2019. 174–182. [doi: 10.1145/3310273.3321560]

[35] Yang YM, Ren ZL, Chen X, Jiang H. Structural function based code clone detection using a new hybrid technique. In: Proc. of the IEEE 42nd Annual Computer Software and Applications Conf. (COMPSAC). Tokyo: IEEE, 2018. 286–291. [doi: 10.1109/COMPSAC.2018.00045]

[36] Word2Vec embeddings. https://radimrehurek.com/gensim/models/word2vec.html

[37] Tai KS, Socher R, Manning CD. Improved semantic representations from tree-structured long short-term memory networks. arXiv: 1503.00075, 2015.

[38] 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 IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]

[39] Wang WH, Li G, Ma B, Xia X, Jin Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 261–271. [doi: 10.1109/SANER48275.2020.9054857]

[40] Zeng J, Ben KR, Zhang X, Li XW, Zhou Q. Code clone detection based on program vector tree. Journal of Frontiers of Computer Science & Technology, 2020, 14(10): 1656–1669 [doi: 10.3778/j.issn.1673-9418.1910019]

[41] 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 (ASE). Melbourne: IEEE, 2020. 931–942.

[42] Feng CH, Wang T, Yu Y, Zhang Y, Zhang YZ, Wang HM. Sia-RAE: A siamese network based on recursive AutoEncoder for effective clone detection. In: Proc. of the 27th Asia-Pacific Software Engineering Conf. (APSEC). Singapore: IEEE, 2020. 238–246. [doi: 10.1109/APSEC51365.2020.00032]

[43] Yuan Y, Kong WQ, Hou G, Hu Y, Watanabe M, Fukuda A. From local to global semantic clone detection. In: Proc. of the 6th Int'l Conf. on Dependable Systems and Their Applications (DSA). Harbin: IEEE, 2020. 13–24. [doi: 10.1109/DSA.2019.00012]

[44] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. arXiv: 2002.08155, 2020.

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

[46] Alon U, Zilberstein M, Levy O, Yahav E. Code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 3: 40. [doi: 10.1145/3290353]

[47] Conneau A, Kiela D, Schwenk H, Barrault L, Bordes A. Supervised learning of universal sentence representations from natural language inference data. arXiv: 1705.02364, 2018.

[48] Cer D, Yang YF, Kong SY, Hua N, Limtiaco N, St. John R, Constant N, Guajardo-Cespedes M, Yuan S, Tar C, Sung YH, Strope B, Kurzweil R. Universal sentence encoder. arXiv: 1803.11175, 2018.

[49] Ahmad WU, Bai XY, Peng NY, Chang KW. Learning robust, transferable sentence representations for text classification. arXiv: 1810.00681, 2018.

[50] Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L. Deep contextualized word representations. arXiv: 1802.05365, 2018.

[51] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018.

[52] Devlin J, Chang MW, Lee K, Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv: 1810.04805, 2019.

[53] 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 (EMNLP). Doha: Association for Computational Linguistics, 2014. 1532–1543. [doi: 10.3115/v1/D14-1162]

附中文参考文献:

[1] 陈秋远, 李善平, 鄢萌, 夏鑫. 代码克隆检测研究进展. 软件学报, 2019, 30(4): 962–980. http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[40] 曾杰, 贲可荣, 张献, 李晓伟, 周全. 基于程序向量树的代码克隆检测. 计算机科学与探索, 2020, 14(10): 1656–1669. [doi: 10.3778/j.issn.1673-9418.1910019]

冷林珊(1998－), 女, 硕士生, 主要研究领域为自然语言处理, 软件克隆检测.

窦淑洁(1998－), 男, 硕士生, 主要研究领域为自然语言处理, 软件测试技术.

刘爽(1987－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件测试, CPS 测试与异常检测等.

王赞(1979－), 男, 博士, 教授, 博士生导师, CCF 会员, 主要研究领域为软件工程、软件测试等.

田承霖(1999－), 男, 硕士生, 主要研究领域为代码克隆检测, 代码表示.

张梅山(1981－), 男, 博士, 长聘副教授, 博士生导师, CCF 专业会员, 主要研究领域为自然语言处理, 人工智能.