文章编号:1671-6833(2023)06-0012-07

基于 Transformer 和卷积神经网络的代码克隆检测

贲可荣, 杨佳辉, 张 献, 赵 翀

(海军工程大学 电子工程学院,湖北 武汉 430033)

摘 要:基于深度学习的代码克隆检测方法往往作用在代码解析的词序列上或是整棵抽象语法树上,使用基于循环神经网络的时间序列模型提取特征,这会遗漏源代码的重要语法语义信息并诱发梯度消失。针对这一问题,提出一种基于 Transformer 和卷积神经网络的代码克隆检测方法(TCCCD)。首先,TCCCD 将源代码表示成抽象语法树,并将抽象语法树切割成语句子树输入给神经网络,其中,语句子树由先序遍历得到的语句结点序列构成,蕴含了代码的结构和层次化信息。其次,在神经网络设计方面,TCCCD 使用 Transformer 的 Encoder 部分提取代码的全局信息,再利用卷积神经网络捕获代码的局部信息。再次,融合 2 个不同网络提取出的特征,学习得到蕴含词法、语法和结构信息的代码向量表示。最后,采用两段代码向量的欧氏距离表征语义关联程度,训练一个分类器检测代码克隆。实验结果表明:在 OJClone 数据集上,精度、召回率、$F1$ 值分别能达到 98.9%、98.1%和 98.5%;在 BigCloneBench 数据集上,精度、召回率、$F1$ 值分别能达到 99.1%、91.5%和 94.2%。与其他方法对比,精度、召回率、$F1$ 值均有提升,所提方法能够有效检测代码克隆。

关键词:代码克隆检测; 抽象语法树(AST); Transformer; 卷积神经网络; 代码特征提取

中图分类号: TP393 **文献标志码**:A **doi**:10.13705/j.issn.1671-6833.2023.03.012

代码克隆是软件开发中的常见现象。在软件开发和演化的过程中,程序员通过复制和粘贴代码段,或者使用框架、复用设计模式和自动工具生成代码等方式加速软件开发。以上操作导致代码库中存在 2 个及以上相同或相似代码段的现象称为代码克隆[1]。开源社区中的代码克隆可以提高开发效率,但是也会导致缺陷传播,降低系统的可靠性。代码克隆检测技术目的在于自动检测软件系统中克隆的代码,可以帮助开发人员和管理者及时发现代码克隆造成的缺陷,更好地保证软件质量。

根据相似程度的不同,Bellon 等[2]将代码克隆分为 4 种类型,从 type1 到 type4,代码相似程度逐渐降低,检测难度也逐渐增加。传统的代码克隆检测方法基于人工定义的规则,对专家经验依赖程度较高。基于深度学习的方法能够自动学习出代码的特征,摆脱“特征工程”的难题,减少人工分析成本。Hindle 等[3]认为编程语言和自然语言一样,具有重复性和可预测性,能够被机器理解和分析。受 Hindle 等[3]工作的启发,许多学者将代码作为自然语言来处理。例如早期的 Nicad 方法[4]通过对比 2 个代码行中最长相同序列来检测代码克隆,该方法在文本层面进行分析,不能检测 type4 类型的克隆。基于词法的克隆检测中,代码被表示为 token 序列。CClearner[5]将程序的单词分为 8 种类型,针对每个类型比较 2 个代码片段的词频序列,得到 8 维向量送入神经网络中训练。尽管代码和自然语言有很多共性的特征,都是由一系列单词组成,但是代码具有更清晰的逻辑、丰富且复杂的结构,且标识符之间存在长距离依赖,将代码视作自然语言或者 token 序列进行处理容易丢失信息。

抽象语法树(abstract syntax tree,AST)是基于抽象语法结构将源代码转换为树结构的表示形式。研究表明,基于 AST 表征的模型明显优于基于序列表征的模型[6-8]。CDLH 模型[6]使用基于树的长短期记忆网络(tree-structured long short-term memory,Tree-LSTM)遍历 AST,为了提高效率,使用特定的哈希函数对代码向量进行哈希处理,比较散列向量之间的汉明距离来识别代码克隆。为了解决子节点数

收稿日期:2022-11-05;**修订日期**:2022-12-03

基金项目:“十三五”预研项目(2019333/6152)

作者简介:贲可荣(1963—),男,江苏海安人,海军工程大学教授,博士,博士生导师,主要从事软件工程、人工智能等方面的研究,E-mail:benkerong08@163.com。

引用本文:贲可荣,杨佳辉,张献,等. 基于 Transformer 和卷积神经网络的代码克隆检测[J]. 郑州大学学报(工学版),2023,44(6):12-18. (BEN K R, YANG J H, ZHANG X, et al. Code clone detection based on Transformer and convolutional neural network[J]. Journal of Zhengzhou University (Engineering Science),2023,44(6):12-18.)

目可变的问题,将 AST 转换为全二叉树。Zeng 等[9]基于递归自动编码器(recursive autoencoder,RAE)模型提出了加权递归自编码器(weighted recursive autoencoder,WRAE),利用 WRAE 分析 AST,提取程序特征并将函数编码成向量。该方法在输入之前利用二叉树生成规则将 AST 转换成了完满二叉树。

由于程序的复杂性,AST 的结构通常又大又深,尤其是嵌套结构,这样容易产生梯度消失问题,导致 AST 丢失从远程节点到根节点携带的一些语义信息。为了简化和提高效率,直接将 AST 视为二叉树或者将 AST 转换为二叉树,这种做法破坏了源代码的原始语法结构,削弱了神经网络模型捕获真实语义的能力。为了克服上述基于 AST 的神经网络的局限性,Zhang 等[10]提出了 ASTNN,将每个大型的 AST 分割成许多语句树序列,在保留原始代码语义的基础上解决了梯度消失问题。词向量训练器将每一个语句树都编码成向量送入双向门控循环单元(bidirectional gated recurrent unit,BiGRU)中,生成代码片段的向量表示。Meng 等[11]在 ASTNN 模型的基础上进行改进,特征提取阶段用 BiLSTM 替换原本的 BiGRU,新增加一个注意力网络层提取代码特征。

在特征提取阶段,现有的大部分模型都使用基于循环神经网络(recurrent neural network,RNN)的时间序列模型。2017 年,谷歌团队[12]提出了完全基于自注意力的深度学习模型——Transformer。相较于 RNN,自注意力支持高度的并行化,并且还能轻松地实现长期上下文建模。

1　代码克隆检测方法

本文提出一种基于 Transformer 和卷积神经网络(convolutional neural network,CNN)的代码克隆检测(Transformer and CNN based code clone detection,TCCCD)方法。TCCCD 方法的总体架构如图 1 所示。基于 AST 的表征方式能够实现源代码的抽象表示,优于基于 token 的表征方式。为了解决 AST 过大带来的梯度消失问题,先将代码解析成 AST,同时保留源代码的语法结构。本文借鉴 ASTNN 切割 AST 的思想,将一段代码的 AST 切割为小型语句子树,其中,语句子树由先序遍历得到的语句节点序列构成,蕴含代码结构和层次化信息。每一组语句树通过语句编码器编码成向量,送入神经网络中。在神经网络设计方面,现有的模型大多使用基于 RNN 的时间序列模型,当 2 个相关性较大的时刻距离较远时,会产生较大的信息损失。本文使用 Transformer 的 Encoder 部分提取代码的全局信息,Transformer 能够直接对输入序列之间的更长距离的依赖关系建模。作为补充,使用 CNN 捕获代码的局部信息,学习出蕴含词法、语法和结构信息的代码向量表示。计算两段代码向量的欧氏距离表征语义关联程度,训练一个二元分类器检测代码克隆。

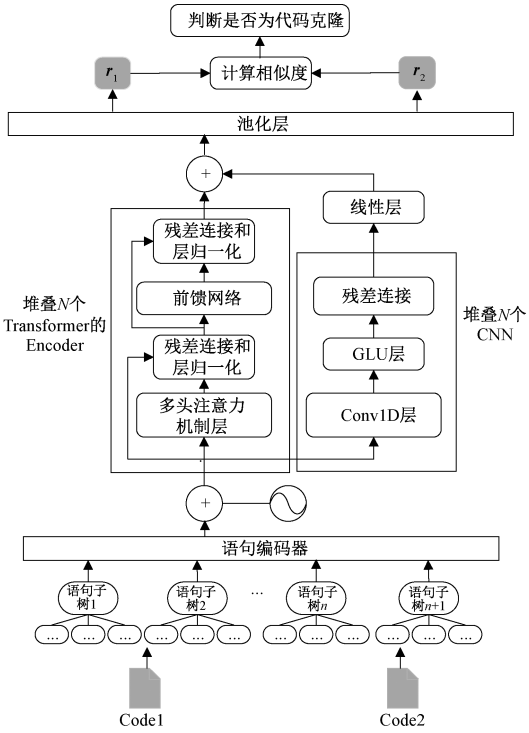

图 1　TCCCD 的总体架构

Figure 1　The architecture of TCCCD

1.1　分割 AST,编码语句树序列

首先,面向函数粒度,使用现有的语法分析工具,将代码解析为 AST。为了防止 AST 规模过大而导致梯度消失问题,不能直接将 AST 作为模型的输入,而是按照语句粒度对 AST 进行切割。

AST 经过先序遍历,得到语句节点的集合 S,将 *Method Declaration* 视为一种特殊的语句节点,故 $S'=S\cup\{Method\ \ Declaration\}$,集合 S'中的每一个节点都对应代码中的一条语句。对于嵌套语句 C,$C=\{FuncDef, While, DoWhile, Try, For, If, Switch\}$,定义一系列独立节点 $P=\{block, body\}$。其中,*block* 用于拆分嵌套语句的头和主体部分;*body* 用于方法声明。对于所有的语句节点 s,$s\in S'$,将 s 的后代定义为 $D(s)$。节点 $d\in D(s)$,如果 s 和 d 之间存在一条路径通过节点 p,$p\in P$,表明节点 d 被节点 s 中的一条语句包含,则称节点 d 为节点 s 的子语句节点。那么,以 s 为根的语句树是由 s 及其所有后代节点 $D(s)$组成。按照语句树的定义,构造函数递归地创建语句树,并将其依次添加到语句树序列中。语句

树序列的顺序蕴含了代码结构的层次化信息。

针对分割得到的语句树序列,将每棵语句树编码成向量。首先,经过 Word2Vec 训练符号嵌入向量,得到初始输入;其次,遍历语句树,并递归地将当前节点的符号作为新的输入进行计算,同时计算其叶子节点的隐藏状态。对于非叶子节点来说,以节点 n 为例:

$$\boldsymbol{v}_n = \boldsymbol{W}_e^{\mathrm{T}}\boldsymbol{x}_n。\tag{1}$$

式中:$\boldsymbol{x}_n$ 表示节点 n 的 one hot 向量;$\boldsymbol{W}_e$ 表示预训练好的词嵌入矩阵。

$$\boldsymbol{h} = \sigma(\boldsymbol{W}_n^{\mathrm{T}}\boldsymbol{v}_n + \sum_{i\in[1,C]}\boldsymbol{h}_i + \boldsymbol{b}_n)。\tag{2}$$

式中:$\boldsymbol{W}_n \in \mathbf{R}^{d\times k}$ 表示权重矩阵;C 表示子节点的个数;$\boldsymbol{h}_i$ 为叶子节点的隐藏状态;$\boldsymbol{b}_n$ 表示偏置;$\sigma(\cdot)$ 为激活函数;$\boldsymbol{h}$ 表示更新后的隐藏状态。

由于每棵语句树子节点的个数不一致,计算叶子节点的隐藏状态时,为了利用矩阵运算进行批处理,使用 ST-tree 的动态批处理样本的算法[10]。

如式(3)所示,经过最大池化,可以得到每一个语句树最后的向量表示 $\boldsymbol{e}_t$,其中,N 表示语句树的节点个数。

$$\boldsymbol{e}_t = [\max \boldsymbol{h}_{i1}, \max \boldsymbol{h}_{i2}, \cdots, \max \boldsymbol{h}_{ik}], i = 1,2,\cdots,N。\tag{3}$$

1.2 Transformer 的 Encoder 捕获全局信息

在 RNN 中,由于信息是沿着时刻逐层传递的,因此,当 2 个相关性较大的时刻距离较远时,会产生较大的信息损失。虽然引入了门控机制,如 LSTM、GRU 等,可以部分解决这种长距离依赖问题,但是它们的记忆窗口也是有限的。CNN 善于捕获局部特征,如果想要捕获远距离特征,需要增加网络深度。Transformer 完全基于自注意力,能够直接建模输入序列之间更长距离的依赖关系。综合考虑 Transformer 的性能与参数量,本文使用 Transformer 的 Encoder 提取全局的特征。

Transformer 的 Encoder 部分捕获全局信息主要包含以下 5 个步骤。

步骤 1 位置编码。输入为经过词嵌入后的张量,故只需对输入进行位置编码,位置编码的方法为

$$PE_{(pos,2i)} = \sin(pos/10\ 000^{2i/d});\tag{4}$$

$$PE_{(pos,2i+1)} = \cos(pos/10\ 000^{2i/d})。\tag{5}$$

$\boldsymbol{X}$ 表示经过词嵌入后得到的张量,将词嵌入与位置编码相加,可以得到经过位置编码后的输出 $\boldsymbol{Y}$:

$$\boldsymbol{Y} = \boldsymbol{X} + PE(\boldsymbol{X});\tag{6}$$

步骤 2 将向量 $\boldsymbol{Y}$ 作为输入,送入多层注意力机制层。注意力函数是基于 3 个矩阵($\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$)同时计算的。

$$\boldsymbol{Y}_{\text{Attention}} = \text{softmax}(\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}/\sqrt{d_K})\boldsymbol{V}。\tag{7}$$

步骤 3 通过残差连接和层归一化,使矩阵运算维数一致。将网络中的隐藏层归一化为标准正态分布,加快模型的训练速度和收敛速度。

$$\boldsymbol{Y}' = \boldsymbol{Y} + \boldsymbol{Y}_{\text{Attention}};\tag{8}$$

$$\boldsymbol{Y}'' = LayerNorm(\boldsymbol{Y}')。\tag{9}$$

步骤 4 通过前馈层和 2 个线性映射层,使用激活函数来生成向量 $\boldsymbol{Y}_{\text{hidden}}$。

$$\boldsymbol{Y}_{\text{hidden}} = \sigma(Linear(Linear(\boldsymbol{Y}'')))。\tag{10}$$

步骤 5 进行残差连接和层归一化,得到最终向量 $\boldsymbol{C}_1$。

$$\boldsymbol{Y}'_{\text{hidden}} = \boldsymbol{Y}_{\text{hidden}} + \boldsymbol{Y}'';\tag{11}$$

$$\boldsymbol{C}_1 = LayerNorm(\boldsymbol{Y}'_{\text{hidden}})。\tag{12}$$

1.3 CNN 捕获全局信息

根据文献[13-15]的研究,CNN 和 self-attention 在特征信息的提取上存在差异。self-attention 焦点在全局上下文中决定在哪里投射更多的注意力,而 CNN 更多地关注所输入的局部特征信息。不同的模型和结构通常会学习到不同的特征[16],Transformer 与 CNN 可以互补[17]。

借鉴 ConvS2S[18],选取其中的卷积块结构,利用级联一维卷积获得代码行局部特征信息。如式(13)所示,每个卷积层包括一个一维卷积、一个门控线性单元(gated linear unit, GLU)和一个残差连接。

$$\boldsymbol{F} = \boldsymbol{Y} + GLU(Conv(\boldsymbol{Y}))。\tag{13}$$

式中:$\boldsymbol{Y}$ 表示 CNN 的输入,为经过词嵌入层和位置编码得到的向量,与 Transformer 的 Encoder 共享。Conv 表示一维卷积层,其卷积核大小常设置为 3。

最后,为了使卷积层输出的特征向量 $\boldsymbol{C}_2$ 与上文中的向量 $\boldsymbol{C}_1$ 具有相同维数,增加一个线性层网络:

$$\boldsymbol{C}_2 = Linear(\boldsymbol{F})。\tag{14}$$

1.4 融合 Transformer 的 Encoder 和 CNN 学习到的特征

将 Transformer 的 Encoder 学习到的全局上下文特征与 CNN 学习到的局部代码特征表示相融合,相比以往的方法大多只用到单个模型学习代码特征,本文方法能充分挖掘代码信息。

$$\boldsymbol{Z} = Concat(\boldsymbol{C}_1, \boldsymbol{C}_2)。\tag{15}$$

如式(15)所示,通过矩阵连接将 2 个特征向量进行合并,得到代码表示向量 $\boldsymbol{Z}$,相当于将 CNN 部分嵌入 Transformer 的 Encoder 中,通过设置 layer 的大小,可以实现 Transformer 的 Encoder 部分与 CNN 的堆叠。

考虑到不同语句的重要性直观上是不相等的,例如,ForStatement 语句中的 *API* 调用可能包含更多的函数信息,因此,使用最大池化来捕获最重要的语义。该模型最终生成一个向量 $\boldsymbol{r} \in \mathbf{R}^{2m}$,它被视为代码片段的向量表示。

1.5　计算两段代码向量的相似度

经过以上步骤,两段代码分别转换成固定长度的向量表示 $\boldsymbol{r}_1$ 和 $\boldsymbol{r}_2$,根据 $\boldsymbol{r}=|\boldsymbol{r}_1-\boldsymbol{r}_2|$ 计算距离,表示两段代码的语义关联程度。经过 sigmoid 函数得到输出 $\hat{\boldsymbol{y}}$, $\hat{y}_i \in [0,1]$,将输出 $\hat{\boldsymbol{y}}$ 作为两段代码的相似性衡量指标。

$$\begin{cases}\hat{\boldsymbol{y}} = \mathrm{sigmoid}(\hat{\boldsymbol{x}});\\ \hat{\boldsymbol{x}} = \boldsymbol{W}_o\boldsymbol{r} + \boldsymbol{b}_o。\end{cases} \tag{16}$$

式中:$\boldsymbol{W}_o \in \mathbf{R}^{2m\times M}$ 为权值矩阵;$\boldsymbol{b}_o$ 为偏置。

二元交叉熵被用作损失函数。

$$Loss = \sum [\boldsymbol{y}\ln \hat{\boldsymbol{y}} + (1-\boldsymbol{y})\ln(1-\hat{\boldsymbol{y}})]。 \tag{17}$$

对所有参数进行优化,存储训练后的模型。对于新的代码片段,应将其预处理为语句子树序列,然后输入重新加载的模型进行预测。输出 $\hat{y}_i$ 是[0,1]的单个值,因此,预测值 *prediction* 为

$$prediction = \begin{cases} True, \hat{y}_i > \delta;\\ False, \hat{y}_i \leqslant \delta。\end{cases} \tag{18}$$

式中:δ 为阈值。本文实验将 δ 设为 0.5[10-11]。如果相似度阈值高于 0.5,得到预测值为 *True*,将代码对标识为克隆。

2　实验与结果分析

2.1　实验数据与评测指标

代码克隆检测使用的数据集包括 OJClone 和 BigCloneBench,这 2 个数据集已被许多研究人员用于代码相似性检测和克隆检测[6,10-11]。OJClone 数据集的标签分为 2 类;BigCloneBench 数据集的标签分为 5 类,分别表示类型 1 至类型 4 的代码克隆,其中类型 3 的代码克隆分为强 3 型和中度 3 型。

数据集基础信息如表 1 所示。对于 OJClone 数据集,7 500 个代码段组合可以得到超过 2 800 万个代码对,随机选取 50 000 个代码对构成实验数据集。对于 BigCloneBench,59 688 个代码段组合可以得到超过 600 万个真代码克隆对和超过 26 万个假代码克隆对。随机抽取了 20 000 对假代码克隆对,抽取中度 3 型和类型 4 代码克隆对共 20 000 对,类型 1、类型 2 和强 3 型数量较少,均小于 20 000 对,以上全部加入实验数据集中。2 个实验数据集中的数据均以 3∶1∶1 的比例随机划分为训练集、验证集和测试集。

代码克隆检测常使用混淆矩阵来评估模型的性能。所有指标的计算方式为

$$\begin{cases} Precision = \dfrac{TP}{TP+FP};\\ Recall = \dfrac{TP}{TP+FN};\\ F1 = 2\times\dfrac{Precision\times Recall}{Precision+Recall}。\end{cases} \tag{19}$$

式中:*Precision* 表示精度;*Recall* 表示召回率;*TP* 表示被分类系统识别为克隆的克隆对数量;*FP* 表示被识别为克隆的非克隆对的数量;*TN* 表示被识别为非克隆的非克隆对的数量;*FN* 表示被识别为非克隆的克隆对的数量。

2.2　实验环境与参数设置

本实验的硬件环境为 Inter(R)Core i7-8700K CPU、NVIDIA GeForce GTX 1080 Ti(discrete graphics)、RAM 32.00 GB。软件环境为 Anaconda 3.6.0、Pycharm 3.4.0、Pytorch 1.6.0。使用 pycparser 解析 C 代码、Javalang 解析 Java 代码得到 AST;使用 Word2Vec 得到词嵌入表示;AdaMax 为优化器。

经过反复多次实验,调整模型参数,直至得到性能最优的模型。详细的模型参数如下:*Encode dimension* 为 128;*Embedding dimension* 为 128;*Hidden dimension* 为 256;*Epoch* 为 15;*Batch size* 为 32;*Number of head* 为 8;*Kernel size* 为 3;*Number of encoder block layer* 为 6;*max length* 为 512;δ 为 0.5。

2.3　实验结果

2.3.1　模型结构分析

为了分析模型结构对代码克隆检测的影响,验证 TCCCD 中融合 2 个网络后提取特征的能力更强,设计了 3 种神经网络结构进行代码克隆检测比较,即 TCCCD、CNN 及 Transformer 的 Encoder。

表 1　代码克隆检测数据集

Table 1　Code clone detection dataset

数据集	代码段数量	真代码克隆对占比/%	最长序列长度	平均序列长度	AST 最大深度	AST 平均深度	AST 最大节点数	AST 平均节点数
OJClone	7 500	6.6	2 271	244	60	13.2	1 624	192
BigCloneBench	59 688	95.7	16 253	227	192	9.9	15 217	206

在OJClone数据集上进行实验,图2展示了经过每一轮训练之后,3个神经网络结构在OJClone验证集上的损失曲线。图3展示了3个神经网络结构在OJClone数据集上检测代码克隆的精度、召回率和$F1$值。可以看出,使用Transformer的Encoder和CNN相结合的结构,即TCCCD,相对于其他2个结构收敛速度更快。经过1轮训练之后,TCCCD在验证集上的损失值就降低到了0.126 8。在15轮的训练过程中,TCCCD每一轮验证损失值均低于另外2个网络结构。经过15轮训练之后,TCCCD结构的精度为98.9%,召回率为98.1%,$F1$值为98.5%。3个评价指标值均高于其他2个网络结构。

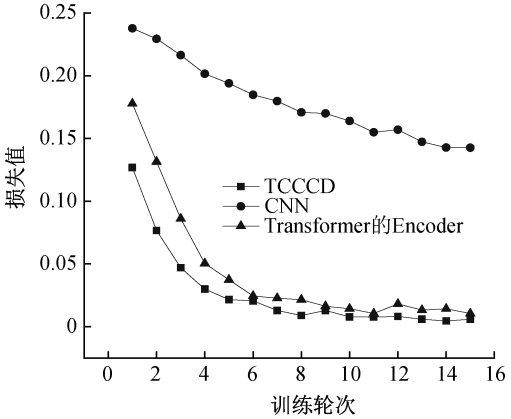

图2 不同神经网络结构在OJClone验证集上的损失曲线

Figure 2 Loss curves for different neural network on the validation set of OJClone

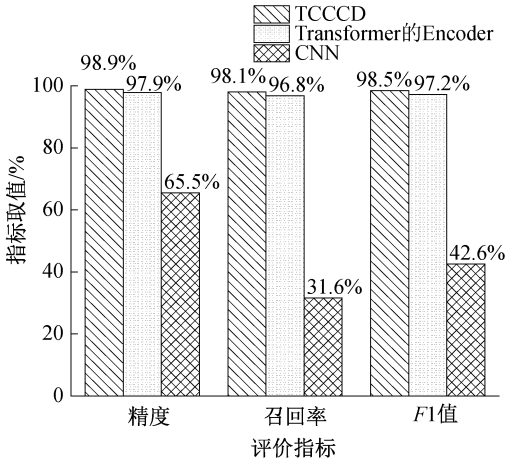

图3 不同神经网络结构在OJClone检测代码克隆结果

Figure 3 Performance comparison of different neural network for detecting code clone on OJClone dataset

表2展示了不同神经网络结构在数据集BigCloneBench上对4种不同类型的代码克隆检测结果,其中,BCB-T1、BCB-T2、BCB-T4分别代表数据集BigCloneBench中的类型1、类型2、类型4,BCB-ST3和BCB-MT3分别代表数据集BigCloneBench中的强3型和中度3型。实验表明,在4种类型的代码克隆中,TCCCD方法精度、召回率及$F1$值均为最高。TCCCD相对于使用Transformer的Encoder部分或者CNN提取代码特征有优势,可以提高检测精度,同时降低误报率和漏报率。且TCCCD在识别BCB-T1和BCB-T2中2个代码片段的相似性方面都非常有效,因为除了不同的标识符名称、注释等,这2个代码片段几乎相同。而在其他代码克隆类型中,TCCCD检测效果稍有下降,尤其在BCB-T4中,检测精度可以达到99.3%,召回率却只有91.2%,$F1$值也只有93.8%。说明检测的假阳性率较高,有一些语义上相似的代码对并不属于代码克隆对。

表2 不同神经网络结构在BigCloneBench数据集上对4种不同类型的代码克隆检测结果

Table 2 Different neural networks detect of four types code clone on BigCloneBench

神经网络结构	类型	精度/%	召回率/%	$F1$/%
CNN	BCB-T1	96.5	95.7	96.1
	BCB-T2	96.3	95.6	95.8
	BCB-ST3	95.9	92.5	93.9
	BCB-MT3	91.1	87.2	89.3
	BCB-T4	90.3	82.1	86.3
Transformer的Encoder部分	BCB-T1	99.6	99.8	99.7
	BCB-T2	99.3	99.6	99.4
	BCB-ST3	99.0	94.1	96.5
	BCB-MT3	98.1	91.2	94.1
	BCB-T4	97.6	86.3	91.2
TCCCD	BCB-T1	100.0	100.0	100.0
	BCB-T2	100.0	100.0	100.0
	BCB-ST3	99.6	94.9	96.8
	BCB-MT3	99.4	92.1	96.1
	BCB-T4	99.3	91.2	93.8

实验表明,Transformer的Encoder部分和CNN学习到了不同的代码特征,2个网络相结合提取特征的能力更强。

2.3.2 各方法在2个数据集上的检测结果对比

表3展示了各模型在OJClone数据集上检测代码克隆的结果。可以看到,RAE和CDLH方法的精度和$F1$值都比较小,均低于60%。这是因为RAE基于token序列进行编码,无法有效学习到代码的结构特征。OJClone是由学生提交的编程题构造而来,一般很少引用第三方库,通常会定义i、j、k等没有意义的变量名。因此,CDLH虽然是基于AST,依旧会丢失很多词汇信息,只能捕获一些语法信息。ASTNN、At-BiLSTM、TCCCD检测代码克隆的效果较好,精度、召回率和$F1$值均高于90%。3种方法都基于语句子树进行编码,说明将大型AST切割成语句子树的方法有助于模型学习代码的结构信息、语

句级的词法和语法信息。TCCCD 在各项度量指标上均高于其他任何一个模型。在检测精度上,TCCCD 与 ASTNN 均具有最高值 98.9%;在召回率上,TCCCD 比 ASTNN 高 5.4 百分点,比 At-BiLSTM 高 2.8 百分点,达到了 98.1%;在 *F*1 值上,TCCCD 比 ASTNN 高 3 百分点、比 At-BiLSTM 高出 2.5 百分点,达到了 98.5%。Transformer 的 Encoder 部分和 CNN 相辅相成,可以学习出融合代码全局信息和局部信息的稠密向量表示。TCCCD 能提取到更多的词法、语法和结构信息来衡量代码的相似性。

表 3　各方法在 OJClone 数据集上的实验结果

Table 3　Results of various methods on OJClone %

方法	精度	召回率	*F*1
RAE	48.9	65.6	56.0
CDLH	47.0	73.0	57.0
ASTNN	98.9	92.7	95.5
At-BiLSTM	96.8	95.3	96.0
TCCCD	98.9	98.1	98.5

表 4 展示了各方法在 BigCloneBench 上的检测结果。

表 4　各方法在 BigCloneBench 数据集上的实验结果

Table 4　Results of various methods on BigCloneBench %

方法	精度	召回率	*F*1
RAE	73.5	60.1	56.0
CDLH	92.0	74.0	82.0
ASTNN	97.9	88.4	92.9
At-BiLSTM	95.9	91.1	93.4
TCCCD	99.1	91.5	94.2

由于 BigCloneBench 包含 4 种代码克隆类型,所以最终的结果是根据 4 种类型的代码对数量加权计算得来。相较于 OJClone 数据集,RAE 和 CDLH 方法检测效果均有提升,RAE 的检测精度由 48.9%提升到了 73.5%,CDLH 的检测精度由 47.0%提升到了 92.0%。这是由于 BigCloneBench 是从真实项目中挖掘到的,实现同一个问题时往往会用到相同的第三方库的类以及 Java 基础类。而 ASTNN、At-BiLSTM、TCCCD 检测效果稍有下降,原因是类型 4 的代码对占代码克隆对总数的一半以上。而类型 4 为语义相似的代码克隆对,有一些语义上相似的代码对并不属于代码克隆,致使假阳性率升高,使得召回率和 *F*1 值均有所降低。虽然上述 3 个方法在 BigCloneBench 上检测效果略有下降,但还是优于 RAE 和 CDLH 方法。

在检测精度上,TCCCD 为 99.1%,比 ASTNN 高出 1.2 百分点,比 At-BiLSTM 高出 3.2 百分点;在召回率值上,TCCCD 比 ASTNN 高出 3.1 百分点、比 At-BiLSTM 高出 0.4 百分点;在 *F*1 值上,TCCCD 比 ASTNN 高出 1.3 百分点,比 At-BiLSTM 高出 0.8 百分点,达到了 94.2%。相比之下,TCCCD 可以实现更高检测精度,同时降低误报率。

3　结论

本文构建了基于 Transformer 和卷积神经网络的代码克隆检测模型——TCCCD。为了解决 AST 过大带来的梯度消失问题,同时保留源代码的语法结构,将 AST 切割为多个语句子树。在神经网络设计方面,本文将注意力模型与 CNN 相结合来提取代码特征。TCCCD 使用 Transformer 的 Encoder 来提取代码的全局信息,由于 Transformer 与卷积神经网络形成互补,再利用 CNN 模型,捕获代码的局部信息,进而学习出蕴含词法、语法和结构信息的代码向量表示。本文方法在 OJClone 和 BigCloneBench 数据集上进行评测,并与代表性代码克隆检测方法进行比较,精度、召回率、*F*1 值均有提升。实验结果表明,该方法能够有效检测代码克隆。

参考文献:

[1] 陈秋远, 李善平, 鄢萌, 等. 代码克隆检测研究进展[J]. 软件学报, 2019, 30(4): 962-980.
CHEN Q Y, LI S P, YAN M, et al. Code clone detection: a literature review[J]. Journal of Software, 2019, 30(4): 962-980.

[2] BELLON S, KOSCHKE R, ANTONIOL G, et al. Comparison and evaluation of clone detection tools[J]. IEEE Transactions on Software Engineering, 2007, 33(9): 577-591.

[3] HINDLE A, BARR E T, SU Z D, et al. On the naturalness of software[C]//2012 34th International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2012: 837-847.

[4] CORDY J R, ROY C K. The NiCad clone detector[C]//2011 IEEE 19th International Conference on Program Comprehension. Piscataway: IEEE, 2011: 219-220.

[5] LI L Q, FENG H, ZHUANG W J, et al. CCLearner: a deep learning-based clone detection approach[C]//2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). Piscataway: IEEE, 2017: 249-260.

[6] WEI H H, LI M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code[C]//Proceedings of the 26th International Joint Conference on Artificial Intelligence. New York: ACM, 2017: 3034-3040.

[7] ALON U, LEVY O, YAHAV E. CODE2SEQ: generating

sequences from structured representations of code[EB/OL]. (2018-08-04)[2022-09-11]. https://arxiv.org/abs/1808.01400.

[8] ALON U, ZILBERSTEIN M, LEVY O, et al. CODE2VEC: learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3: 40.

[9] ZENG J, BEN K R, LI X W, et al. Fast code clone detection based on weighted recursive autoencoders[J]. IEEE Access, 2019, 7: 125062-125078.

[10] ZHANG J, WANG X, ZHANG H Y, et al. A novel neural source code representation based on abstract syntax tree[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2019: 783-794.

[11] MENG Y, LIU L. A deep learning approach for a source code detection model using self-attention[J]. Complexity, 2020, 2020: 1-15.

[12] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[C]//Advances in Neural Information Processing Systems 30. Long Beach: NIPS, 2017:5998-6008.

[13] CORDONNIER J B, LOUKAS A, JAGGI M. On the relationship between self-attention and convolutional layers[EB/OL]. (2019-11-08)[2022-09-11]. https://arxiv.org/abs/1911.03584.

[14] GONG J J, QIU X P, CHEN X C, et al. Convolutional interaction network for natural language inference[C]//Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Stroudsburg: Association for Computational Linguistics, 2018: 1576-1585.

[15] YANG G, ZHOU Y L, CHEN X, et al. Fine-grained pseudo-code generation method via code feature extraction and transformer[C]//2021 28th Asia-Pacific Software Engineering Conference (APSEC). Piscataway: IEEE, 2022: 213-222.

[16] 张安琳, 张启坤, 黄道颖, 等. 基于CNN与BiGRU融合神经网络的入侵检测模型[J]. 郑州大学学报(工学版), 2022, 43(3): 37-43.
ZHANG A L, ZHANG Q K, HUANG D Y, et al. Intrusion detection model based on CNN and BiGRU fused neural network[J]. Journal of Zhengzhou University (Engineering Science), 2022, 43(3): 37-43.

[17] YUAN Y H, HUANG L, GUO J Y, et al. OCNet: object context for semantic segmentation[J]. International Journal of Computer Vision, 2021, 129(8): 2375-2398.

[18] GEHRING J, AULI M, GRANGIER D, et al. Convolutional sequence to sequence learning[C]//Proceedings of the 34th International Conference on Machine Learning. New York: ACM, 2017: 1243-1252.

Code Clone Detection Based on Transformer and Convolutional Neural Network

BEN Kerong, YANG Jiahui, ZHANG Xian, ZHAO Chong

(College of Electronic Engineering, Naval University of Engineering, Wuhan 430033, China)

Abstract: Code clone detection, based on deep learning, is often applied to use the model to extract features in the sequence of tokens or the entire AST. That may lead to the missing of important semantic information and induce gradient disappearance. Aiming at these problems, a method of code clone detection based on Transformer and CNN was proposed. First of all, source code was parsed into AST. Then the AST was cut into statement subtrees, which were input into the neural network. Statement subtrees were composed of a sequence of statement nodes obtained by pre-traversal, which contained the structure and hierarchical information. In terms of neural network design, Encoder of Transformer was used to extract global information of the code. CNN was used to capture the local information. Fusion of features were extracted from two different networks. Finally, a vector containing lexical, syntax, and structural information could be learned. The Euclidean distance was used to represent the degree of semantic association. A classifier is trained to detect code clone. Experimental results showed that on OJClone dataset, the *Precision*, *Recall*, and *F*1 values could reach 98.9%, 98.1%, and 98.5%, respectively. On BigCloneBench dataset, the *Precision*, *Recall*, and *F*1 values could reach 99.1%, 91.5%, and 94.2%, respectively. Compared with the relevant methods, the *Precision*, *Recall*, and *F*1 values were all improved. This method could effectively detect code clone.

Keywords: code clone detection; abstract syntax tree(AST); Transformer; convolutional neural network; code feature extraction