基于编译运行，上下文与词频分析的代码克隆检测

XXXX[1]

[1](XX 大学 XXX 系,XXX 学号：XXXXXXXXX)

Detecting Code Clone with Compilation-Running, Context and Word Frequency *

XXXX[1]

[1](XXXXX, XXXX University, XXX, Student ID: XXXXXXXXX)

Abstract: Code clone refers to more than two duplicate or similar code fragments existing in a software system[1]. Code clone is frequent in software development, finding which can improve efficiency, increase stability, reduce redundancy of source code repository and prevent the propagation of software defects. In the paper, we try three different method to detect code clone: the first is detecting by compiling and running the test to compare the output of it with same input; the second is changing word in test into word vector and learning by LSTM; the third method is counting the occurrence number of word, and learning by Gradient Tree Boosting (XGBoost). I use the combination of the first and the third method to generate the answer I submit in Kaggle. The highest F1 score I get is 0.76572.

Key words: code clone, compilation, running, word frequency

摘 要: 代码克隆，是指存在于代码库汇总两个及以上相同或相似的源代码片段[1]。代码克隆在软件开发中非常常见，找到他们可以提高效率，提升软件稳定性，减小代码库冗余和减少软件缺陷传播等。本文实验了三种不同的检测代码克隆的方法：第一种是直接编译运行测试代码，比较在相同输入的情况下输出是否一样；第二种是把代码的每个词变成词向量，用 LSTM 进行学习；第三种是统计出每个代码片段的，各个词出现的次数，用梯度提升树（XGBoost）进行训练。最后比赛提交的答案，是第一种和第三种结合的方法生成的，Kaggle 上最高 F1 Score 0.76572。

关键词: 代码克隆；编译；运行；词频

1 引言

本次实验主要的任务是比较代码对是否是克隆代码对，要尽可能的使 F1 Score 高。本文尝试了三种方法：第一种方法是直接编译运行测试，看输入输出是否一样，将在第二章介绍；第二种是把代码用词向量表示，用 LSTM 学习，将在第三章介绍；第三种是统计每个测试中词出现的次数，作为特征用梯度提升数进行训练，将在第四章介绍，第五章将介绍最后采用的，结合了第一和第三种方法的，生成了最佳分数的方法。

总体来看，三种方法中第一种方法效果是最好的，这个是由于数据集的一些特殊性导致的；第二种方法效果不佳，而且训练耗时非常的长；第三种方法速度快，效果也还可以，但是比不上第一种方法。

2 基于编译运行的克隆检测方法

通过编译运行来检测两段代码是否是克隆代码并不是一个在论文中很常见的做法，之所以我在这里想到使用这种方法，主要是基于下面几点对数据集的观察：

- 数据集明显是来自于 OJ 网站，而 train 中每个文件夹中的代码应该是同一道 OJ 题目的代码，而 OJ 题目又有如下几个特征
  - 对代码的输入和输出有固定的格式要求
  - 在相同的输入下，同一道题目正确的代码应该有完全一致的输出
- 每个测试中都一定有一个 main 函数（void main()，int main()，int main(void)等）
- 测试中没有任何宏定义
- 大部分时候，测试中的变量都有其声明（但也有时候有些代码没有，应当是因为其定义为宏函数，而宏函数又被统一去掉了）

因此，可以做出如下判断：

- 测试仅仅是被去掉了宏定义=》大部分测试都有可能恢复成近似原始的代码从而运行
- 测试是来源于 OJ，且克隆代码在这里被规约为相同题目的代码=》如果两个代码，在相同的输入下，有相同的输出，那么就和很有可能是克隆代码

所以，这里我使用运行测试的方法来进行代码克隆。虽然说思想是很简单直观，要处理的细节问题也很多。首先，因为不是所有人都是在网站上提交正确的答案，例如有些人会提交一份无法运行的代码上去，有些人提交的代码运行会有段错误等问题，还有就是由于代码都被稍微处理过，去掉了所有宏定义，所以有些代码会因为缺少定义而无法运行。其次，因为如果是同一道 OJ 题目，在输入了相同的内容，接受的输入数目一定是相同的（好比有很多 OJ 题目，第一个输入的数字都是指定后面接受多少输入），如果能统计编译后的测试具体接受了多少的输入，就能获取到更多的信息。

2.1 测试预处理

我这里在编译代码之前会对代码进行预处理：

- 自己编写了一个头文件“AddHead.h”，其中：
  - 系统头文件（bits/stdc++.h，用以包含所有）
  - 用宏重新定义了的输入函数
    - 通过宏定义，把所有的 printf 都替换为自己所写的 SCANF，其通过统计输入字符串中有多少个百分号“%”来记录这次函数调用请求了多少个外部输入
    - 通过宏函数，把 cin 变成自己所写的，类型为 CIN_OPERATOR 的 CIN_OBJ 对象，其实现了几乎所有 cin 的输入函数，如 get，getline，>>等，还有逻辑比较函数*()和 operator!()，以适应 cin 被作为 while，if 中的判断的情况
    - 代码获取输入的次数通过标准错误输出获得

- 把所有的，不把结果赋值给任何一个变量的 cin.get()替换成不统计输入次数的输入函数 CIN_OBJ.get_not_increase_input_num()
  - 这是因为很多代码都会用 cin.get()来跳过输入中的换行符，而如果不处理，会造成克隆代码间输入次数差别过大的问题
- 把代码中所有 void main()和 void main(void)替换为 int main()
- 把所有的整数类型（short，long，long long）变成 int 类型，并把 printf 中的%hd，%ld，%ld 变成%d
  - 这是由于有好多代码变量类型和 printf 中的类型不匹配，导致编译失败

在这样处理后，大部分代码都能正常运行，下面的表格记录了 10000 个测试编译和运行的情况，其中有“全部”前缀的表明，是在三个

2.2 输入

由于大部分 OJ 题目都是 要求输入数字，即使是要求输入字符串，输入的数字也可以被当做字符串，因此我这里构造的三组收入都是数字，且数值都很小，分别为一百个“1”，一百个“3”和一百个“5”，以防止因为数字过大导致输入过多，数组溢出。

2.3 比较输出

代码在编译，且输入一组输入后，用类 ExeResult 来储存结果，其有三个成员：output 存储代码的输出，input_times 存储代码输入的次数，result 存储代码运行状态（即 mian 函数返回的数值，一般来说 0 表示运行正常）。特殊的，对于编译失败的测试，result 置为-1。

由于我这里一共有三组输入，因此每一个测试均有三个 ExeResut 对象，这三个对象用 python 的 list 存储。

比较两个测试 test1 和 test2 的输出的过程如下（详情见函数 compare_test6）：

1. 令 list1 和 list2 分别为 test1 和 test2 的结果的数组，succ 置为 0
2. For r1, r2 in (list1, list2): # 对每一个输入的结果
   a) 如果 r1 或 r2 的 result 不为 0 # 表示编译或运行失败
      i. Continue
   b) 如果 r1 和 r2 的输出不一致，或在输出为数字的情况下输出不一致：
      i. Return False
   c) succ += 1
3. If succ >= 1:
   a) Return True
4. Else:
   a) Return False

2.4 编译运行结果与最后效果

下面的表格展示了在经过代码预处理后，10000 个 test 中的测试编译运行的情况。

全部编译成功	部分编译成功	全部运行成功	部分运行成功
9430	558	9116	22

可以看到大部分测试都是能完全成功运行的，完全不能运行的占全部测试的不到十分之一

因为编译运行需要的时间比较长，线下我采用两个方法来大致测试一下效果：

- 从 train 中随机挑出 100 对非克隆代码对
- 从 train 中随机挑出 100 对克隆代码对

由于这种方法没有用到 train 中的数据，所以直接在 train 上看效果相当于在测试集上看效果。

第一种方法，100 对测试全部成功预测为非克隆对，其中 77 对两个测试都成功的编译运行。之所以正确率如此之高，这是因为：

1. 如果两个代码都能成功运行，由于不同 OJ 题目的输入输出差别一般来说都很大，因为很难完全相同
2. 如果有测试不能编译运行，那么就会判定为非克隆对

第二种方法，100 对测试只有 56 个是预测为是克隆对，而一共有 80 对测试是编译运行完全成功的。由于如果一个测试对要预测为克隆对，其需要是完全编译运行成功的，所以可见查全率仅为约 0.7，这里主要的原因有：

1. 由于测试来自于 OJ 提交代码，不是所有的代码都是正确的，因此对于错误的代码，在相同输入下输出会和正确代码不一致
2. 输入不符合 OJ 题目规定的输入范围
3. 对于无法运行的测试，这个方法难以进行有效判断

总的来说，这个方法可以做到很高的查准率，但查全率有所欠缺。我最早交的是只有这个方法的代码，线上 F1 Score 是 0.70 左右。

3 基于上下文的克隆检测方法

这里“基于上下文”主要是指的考虑词与词之间的顺序关系。这里我采用的方法是先把每个词按 one-hot 进行编码，然后把变换后的测试输入到 LSTM 网络中，把每个代码变成一个向量，比较两个代码是否克隆就是看两个代码的向量之间距离多少。

之所以会想到使用这样的做法，是因为对数据集进行观察后有如下几个特点：

- 同为克隆代码的两个测试，变量名可能有不小的差别
  - 如，有些人用拼音命名，有些人用英文命名
- 不是克隆代码的两个测试，可能变量名非常相似
  - 如，i，j，k 这种在循环中很常见的变量，以及 n，m 这种 OJ 经常出现的变量
  - 很多 OJ 题目可能没有自己独有的变量名
- 克隆代码在结构上会有不少相似之处
  - 如，输入，输出所在的循环深度，处理部分的循环深度

我也尝试了用 word2vec 来把测试中的每个词变成词向量，然后把变换后的测试输入到 LSTM 神经网络中，不过两种方法效果差不多，结果都不是很理想。

3.1 网络结构

网络结构主要参考的是检测句子相似度的网络结构[2]，如 Figure 1 所示。

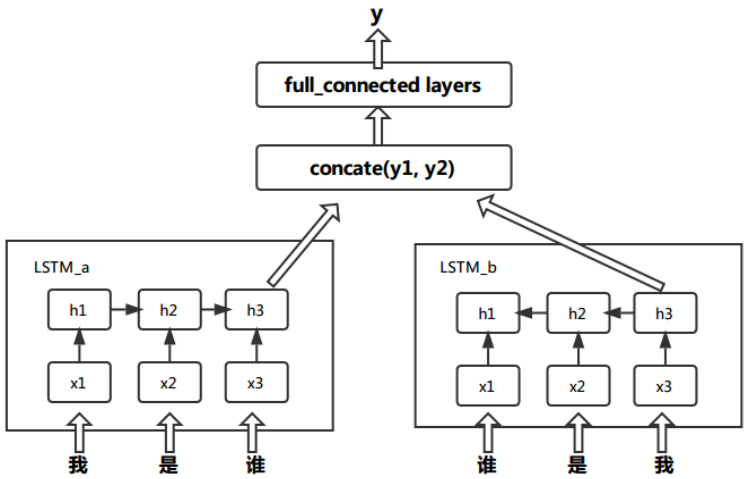

Figure 1 网络结构图[2]

可以看到，其主要分为三个部分：

- 嵌入层：处理输入（one-hot 编码）
- LSTM 层：循环神经网络，处理输入
- 全连接层：比较测试的对应的向量，输出是否是克隆代码

可以看到，整个网络的思路是比较简单直接的：希望能让学习器学到代码的上下文信息，从而判断是否是克隆代码。

还有一个比较重要的点就是代码分词，即代码那些成分，如变量，符号，运算符，可以算作是一个词。我尝试了三种：第一种是以所有变量名为词；第二种是以所有变量名以及所有符号，如分号，引号，加减号等，看做是词；第三种是把所有的变量名统一换成字符串“@VAR”，所有的字符串统一换成“@STR”，所有的数字统一换成“@NUM”等。第二种和第三种做法的出发点是，希望学习器能更少的关变量名等字面值，而更多关注代码结构上的一些信息。三种方法效果均差不多，最后是用了一种折中的方法：

- 去除所有的符号，如中括号[]，小括号()等
- 数字统一换成“@NUM”，字符串统一换成“@STR”，字符统一换成“@CHAR”
- 剩下的部分不变

3.2 最后效果

由于 LSTM 网络训练速度特别慢，因此我只试了两种参数配置下的网络，且只保留了一个网络的模型，不过两个模型实际效果差别不大。

下面是网络具体的参数配置：

- min_count，即忽略总频率低于此值的所有单词：5
- max_len，即代码序列的最大长度，不足则补 0：300
- embedding_output_size，即嵌入层输出大小：50
- lstm_output_len，即 LSTM 层输出大小：50
- lstm_dropout，即 LSTM 层输入的遗忘系数：0.1

- lstm_recurrent_dropout，即 LSTM 层线性变换的循环状态的遗忘系数：0.1
- dropout_rate，即两个 Dropout 层的系数
- dense_unit：即全连接层输出大小：50

测试和前面编译运行的类似：

- 从 train 中随机挑出 1000 对非克隆代码对
- 从 train 中随机挑出 1000 对克隆代码对

前一个 1000 对测试 854 个判别非克隆代码对，后一个 1000 对测试对 119 个判别为克隆代码对。

可以看到，即使是直接拿 train 来作为测试集，两个的效果均不理想，且模型是已经收敛了的。

我和其他一几个人就这个方法有比较过效果，效果均不理想，但原因暂不清楚，怀疑还是因为 LSTM 学习结构信息不成功，有可能用 AST 树的话，可以让其学习到更多的信息。

4 基于词频的克隆检测方法

基于词频，即直接统计每个测试的每个单词出现的次数，用向量表示，如第一维对应单词“main”的频率，第二维对应“printf”的频率，然后直接以这个向量作为特征，用学习器进行学习，这里我是用 XGBoost 进行学习。

4.1 实现

这个方法实现上来比较简单，首先按照空格和各种符号进行分词，然后用 tensorflow 的 Tokenizer 统计每个词在所有 train 的测试中出现的次数，挑选出出现次数最高的 300 个单词，每个单词对应向量的一个维度，存储在字典中。然后，再通过这个字典来转化所有的测试为向量，作为决策树的输入。

决策树通过调用 XGBoost 库实现。这里我训练了 83 个决策树，每个决策树对对应于 train 中的一个测试集，用来判断输入是否对应于这个测试集，这样我就可以通过这个 83 个学习器，把每个测试表示成一个 83 维的向量，最后通过比较测试的向量是否在某一维度上都是数值很高，其他维度上数值很低，来预测是否是克隆代码对。

4.2 最后效果

测试类似于前面的，不过由于这个方法运行速度很快，所以可以进行更大规模的测试。

首先我们看如果直接在 train 上进行测试的效果：

- 从 train 中随机挑出 10000 对非克隆代码对
- 从 train 中随机挑出 10000 对克隆代码对

由很明显可以感到有过拟合的问题：10000 对非克隆代码对全部预测为非克隆，而 10000 对克隆代码对中有 8728 个被预测为克隆代码对。

而如果我们吧 trian 的钱 2/3 部分用作训练，后 1/3 部分用作测试，并且测试数量一致：

- 从 train 后 1/3 中随机挑出 10000 对非克隆代码对
- 从 train 后 1/3 中随机挑出 10000 对克隆代码对

其中后 1/3 是指 train 的 83 个文件夹，每个文件夹下后 1/3 个测试，2/3 同理。在这种测试方法下，10000 个非克隆对预测有 9998 个非克隆对，10000 个克隆对预测有 7117 个克隆对，可以明显感觉到效果要比第二种方法好很多。。

线上单独这种方法具体得分没有进行过实验，但是有拿这个方法和第一个方法结合进行提交，分数略微有

提升，而第二个方法在和第一个方法结合提交的时候效果又明显下降，可以推测第三种方法应该是比第二种方法要好很多的，但具体还是得要进一步实验论证。

5 最佳方法

5.1 三种方法效果对比

三种方法在克隆对，非克隆对上的预测效果如下：

	克隆对准确率	非克隆对准确率
第一种方法：编译运行	0.56	1.00
第二种方法：上下文，LSTM	0.12	0.85
第三种方法：词频，决策树	0.71	1.00

虽然从表格来看，第三种比第一种方法要好，但是在实际线提交的时候是反过来的（虽然说第三种方法没有单独交过，但是可以明显感觉到没有第一种方法效果好），这个很有可能是因为 test 中有不少的代码是不属于 train 中 83 个类别的，而且有 test 中不能编译运行的测试数量感觉也要比 train 中的少，所以导致实际线上的效果第一种要比第三种感觉好。

5.2 线上最佳方法

我得分最高的方法主要是第一和第三个方法的结合，外加了一个对代码的静态的判断：如果两个测试中的字符串的集合完全一致（即测试中用双引号括起来的部分如果在一份代码中有出现，那么另一份代码也有这一部分），那么也判定两段代码互为克隆代码。

整个流程如下：

- 用第一种方法，生成所有测试的，运行后的结果，储存到文件中
- 用第三种方法，训练处学习器，储存到文件中，并把 test 中的测试处理成词频向量（即学习器可接受的输入），存储到文件中
- 加载前面文件中存储的信息，对每一对测试对
  - 当下面条件之一为真时判断测试对为克隆代码
    - 第一种方法返回 True
    - 第三种方法返回 True
    - 提取两个测试的字符串的集合，如果集合相等则返回 True

通过这个方法，我取得了线上 0.76572 的成绩。

6 比较和讨论

由于第一，二种方法所需的时间比较长，所以这里我没有采用 k 折交叉检验的方法来比较，从前面几章最后线下的测试来看，可以明显感觉第一种方法效果特别的好，而后面两种则效果不明显。

但是，第一种方法也有明显的限制，就是对数据集有很高的要求。这次是凑巧，每个测试都有 main 函数，且大部分都能运行，但如果测试变成一个个非 main 函数了，或者只有代码片段，这个时候第一种方法就不起作用了。

虽然第三种方法没有给出完整的，但就两个方法和第一种方法结合后，在线上提交的结果来看，第三种方法要比第二种方法好很多：线上第一种方法和第二种方法结合，成绩会从 0.74 掉到 0.69，而第一种方法和

第三种方法结合后，能维持在 0.74 左右，其中用了第三种方法和没用第三种方法的相比，预测结果中预测为克隆对的测试对数量前者要比后者多出将近 800 的样子，表明第三种方法确实是对答案有造成改变，且效果不差。之所以第三钟方法会比第二种方法好，我推测是：

- 代码上下文之间关系没那么的紧，或者说关系的距离非常的远，如输入语句和处理语句之间可能隔了十多行
- 词频包含了很多结构上的信息，如 i，j，k 的出现次数可以大致表明代码中循环的情况

第二种方法如果要改进的话，可能可以尝试使用 ast 语法树，这样网络就能更加直接的接触到结构信息，从而提高准确度。

致谢 在此,我向数据挖掘课的老师和助教表示感谢.

References:

[1] Chen QY, Li SP, Yan M, Xia X. Code Clone Detection: A Literature Review. Journal of Software, 2019, 30(4): 962-980(in Chinese).http://www.jos.org.cn/1000-9825/5711.htmWegner P, Zdonik SB. Inheritance as an incremental modification mechanism or what like is and isn't like. In: Gjessing S, Nygaard K, eds. Proc. of the ECOOP'88. LNCS 322, Heidelberg: Springer-Verlag, 1988. 55-77.

[2] https://www.jianshu.com/p/a649b568e8fa