基于 XGBoost 的代码克隆检测方法

151250189 翟道京[1]

[1](南京大学 匡亚明学院,江苏 南京 210046)

XGBoost Based Duplicate Code Detection Method

ZHAI Dao-Jing[1]

[1](Kuang Yaming Honors School, Nanjing 210046, China)

Abstract: Copy-paste-modify is a widely used code-reusing strategy. This often leads to numerous code fragments with similar functionalities – called clones. Cloning unnecessarily increases program size and maintenance effort. In this paper, after novel data-preprocessing method and feature extract process, we implement XGBoost algorithm on duplicate code detection project. In the finally, we compare XGBoost Algorithm with other algorithms, like GBDT, Naïve bayes, SVM.

Key words: Duplicate code detection; XGBoost, Data-preprocessing, Feature Extraction

摘 要: 抄袭-修改是常见的代码克隆技巧，代码克隆往往会带来大功能相似型代码重复出现的后果，引发程序设计不良、缺乏过程抽象。本文介绍了利用 XGBoost 实现了高性能代码克隆检测功能，同时描述了数据预处理方式与特征提取技巧。在文末，将 XGBoost 算法与 GBDT、Naïve Bayes、SVM 等算法进行了实验对比。

关键词: 代码查重, XGBoost, 数据预处理, 特征提取

代码克隆(Duplicate code,也称为代码重复)在程序设计中表示一段源代码在一个程序，或者一个团体所维护的不同程序中重复出现的现象[1]。代码克隆往往会带来大量功能相似型代码重复出现的后果。不当的代码克隆会导致程序设计不良、缺乏抽象，对程序空间复杂度与维护成本带来不必要的负面影响。

1 任务描述

1.1 问题描述

我们本次实验的任务，便是搭建分类学习器，从大量克隆代码片段中获得信息，实现对 OJ 系统中克隆源代码的判定。在本次实验中，我们基于 XGBoost 算法实现了代码的克隆检测。

1.2 数据描述

本次实验数据集和测试集数据为原始 C 语言代码，代码头文件等已经经过处理.

1.2.1 数据集

数据集是包含 83 个文件夹，共计 51016 个 C 源代码，相同文件夹内文件被判定为克隆代码，不相同文件夹内被判定为非克隆代码。

1.2.2 测试集

测试集含 10000 个 C 源代码，测试任务为判断 200,000 对代码是否为克隆代码。

1.3 问题难点

1.3.1 数据预处理

本次数据挖掘实验数据量较大，需要从 169.5MB 训练数据中挖掘关键特征，大量数据原材料提高了训练器可能达到的精度，但同时提高了计算、存储复杂度。

1.3.2 多种代码克隆情况

代码克隆问题中，有四种对源代码片段进行处理的克隆方式，造成克隆代码片段与源代码片段的差异[2]。

Type-1: 通过改变排版、添加注释造成代码的差异。

举例 1：除空格、布局和注释不同外，其余部分选择相同的代码片段。

Type-2: 通过更改变量名称和初始值造成代码差异。

举例 2: 除标识符、字面量、类型、空格、布局和注释外，语法结构相同的代码片段。

Type-3: 通过添加、删减、更改语句造成代码差异。

举例 3: 除标识符、字面量、类型、空格、布局和注释外，进一步对克隆代码片段进行改动，例如修改、增加或者删除语句，但是仅仅造成程序的冗余与难以理解，不影响程序的功能。

Type-4: 从句法上对源代码的功能进行更改，实现相同的功能。

举例 4: 两个或多个代码片段执行相同的计算，但是语法结构的实现方式不同。

上述四类克隆统称为简单克隆(Simple Clone)，将简单克隆组合成可形成结构克隆(Structural Clone).

1.3.3 样本不平衡

数据集是包含 82 个文件夹，共计 51016 个 C 源代码，每个文件夹平均代码量为 600 段，通过随机抽样，选取的两段代码属于克隆代码的概率为

$$\frac{82 * \binom{600}{2}}{\binom{51016}{2}} = 0.0113.$$

因此在训练过程中，训练集正负样本不平衡。

2 克隆代码查重方法回顾

2.1 基于Text的检测方法

基于文本的克隆检测方法是通过直接比较原代码文本，使用字符串匹配等算法来检测克隆代码，因为并没有对源程序进行词法分析，大部分仅可以较好的支持对 Type1 克隆的检测。在这种检测方式中，比较典型的 NICAD(Roy and Cordy, 2008)直接对文本序列进行了对比。

2.2 基于Token的检测方法

基于 Token 的克隆检测方法是通过对源代码进行词法分析生成源代码的 Token 序列，然后通过寻找 Token 序列中相似的子序列来检测克隆代码，因为对源代码进行了词法分析，所以可以较好的支持 Type2 克隆的检测。但由于缺乏语法和语义分析，无法较好的支持 Type3 和 Type4 克隆的检测。

在这种检测方式中，CCFinderX[4](Kamiya et al., 2002)和 SourcererCC[5](Sajnani et al., 2016)通过提取 Token 信息，实现了代码克隆检测。

2.3 基于Tree的检测方法

基于 Tree 的克隆检测方法是将源代码表示为某些树的形式（如抽象语法树、代码解析树等），然后通过树匹配算法从中寻找相似的子树来检测克隆代码，由于对源代码进行了语法分析，所以提高了克隆代码检测的准确率，可以较好的支持 Type3 克隆的检测。

目前，伴随着深度学习领域的深入研究与 GPU 高性能远算方式的应用。结合抽象语法树(AST)与深度学习网络的代码克隆查重方法得到了广泛研究。Deckard[6](Jiang et al., 2007)首次引入抽象语法树来衡量代码片

段之间的距离，基于 CDLH 深度学习网络的监督学习模型[7](Hui-Hui Wei and Ming Li,2017)也取得了不错的进展。

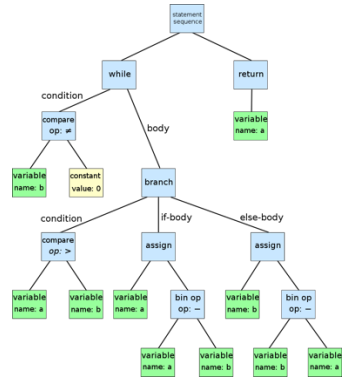

图 1: 抽象语法树

2.4 基于Graph的检测方法

基于 Graph 的克隆检测方法的主要思路是，将源代码转化为某种程序图形式(如程序依赖图)，然后通过寻找同构的子图来检测克隆代码。因为程序依赖图表现了程序的语义信息，所以该方法可以支持语义相似的 Type4 克隆代码的检测。但是由于程序依赖图生成算法和图匹配算的时间和空间复杂度较高，从而导致这类算法检测方式无法应用于大规模的克隆代码检测。

2.5 基于Metric的检测方法

基于 Metric 的克隆检测方法是先将源代码转换为某种中间表示，然后在其基础上提取度量值并抽象为一个特征向量，然后通过计算特征向量的相似度来检测克隆代码。该方法的主要优点为检测速度较快，但基于 Metric 的方法高度依赖于 Metric 的提取，对源码提取度量值的过程中会损失源码的部分语义信息，因此检测效果不够理想，使得检测范围受限。

3 数据预处理

3.1 条件限制

考虑到我们参与本次竞赛时间有限，工具受限(无 GPU)；同时经过对训练集的观察，我发现 Type1、Type2、Type3 代码克隆问题最为突出。为在短时间内实现较好的检测效果，我首先排除了基于 Graph 和基于 Metric 的检测方式。同时在初步尝试后，我放弃了通过搭建深度学习网络的方法，希望能同时结合 Tree-Token-Text 的思路，深度提取特征，综合提取信息。

3.2 特征提取

3.2.1 基于 Text 的特征

基于 Text 的克隆检测代码直接比较源代码之间的字符串序列，判断相似程度。从基于 Text 的克隆检测角度，对高频“敏感”词汇词频的检测，能有效检测 type1 克隆问题。

我关注到训练集中部分变量名称得到了重复使用，提取到以下特征

VAR 数量	Temp 数量	Value 数量
Count 数量	Num 数量	Info 数量

表 1: 基于 Text 的特征

3.2.2 基于 Token 的特征

基于 Token 的克隆检测方式对源代码进行词法分析生成源代码的 Token 序列，然后通过寻找 Token 序列中相似的子序列来检测克隆代码。分析训练集代码后，我得到了以下 Token 序列。值得注意的是，这些 Token

往往是 C++输入、输出流内部指令，作为程序运行的载体。

int 数量	double 数量	char 数量
float 数量	printf/cout 数量	scanf/cin 数量

表 2: 基于 Token 的特征

3.2.3 基于 Tree 的特征

我使用 Pycparser 分析数据集，建立起抽象语法树(AST)，观测抽象语法树结构，我发现 C++功能性语句对程序结构意义重大，搭建起整个程序框架。我将循环/判断语句作为本次任务的几种特征。

if 数量	switch 数量	case 数量
while 数量	for 数量	while 数量
for/while 嵌套层次		

表 3: 基于 Token 的特征

3.2.4 基于语义的特征

对于基于语义的特征，我关注到代码的数据结构(数值计算/数组计算/指针运算)能反映程序实现的方式，从而体现语义。我将常见数据结构标识符作为特征

[] 数量(用于标记数组)	][数量(标记二维数组)	*数量(标记指针)

表 3: 基于语义的特征

如此，从文本、语法、语义三个层次，我选择了以上 21 个特征值描述样本信息。

3.3 样本提取

3.3.1 样本

本次学习任务样本为代码对，样本标签为 0/1 标签(0 表示克隆代码，1 表示非克隆代码)，该学习问题为二分类问题。本次实验将样本代码对各特征间绝对值距离作为该维度特征，选择欧式距离作为样本间距离计量方式。

3.3.2 样本不平衡问题

面对之前提到的样本不平衡问题，我通过引入一定程度的克隆代码数据，调节正负样本案例的比例。

由于我们未知在测试集中正负样本比例(即测试集分布信息)，平衡正负样本的程度具有人为选择性。我的处理方法是，在某次获得预测成绩后，从预测成绩中获得信息，调节训练样本正负样本比例。

4 模型选择与评估

集成学习通过构建并结合多个学习器完成学习任务，场获得比单一学习器显著优越的繁华特征。我们本次任务特征提取仍有不完善之处，对 Type3 和 Type4 类克隆代码的判断能力有限，若单独选取学习器，其泛化特性略优于随机猜测，被称为“弱学习器”(weak learner)。考虑到集成学习对弱学习器的优势尤为明显，我们本次数据挖掘最终选择基于决策树的 XGBoost(Exterme Gradient Boost)算法，并将 XGBoost(Exterme Gradient Boost)算法与 GBDT(Gradient Boost Decision Tree)算法，Naïve Bayes 算法和 SVM 算法进行了对比。

4.1 XGBoost算法[9][10][11]

首先回忆 Gradient Boosting 算法，该算法如下图所示[8]

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

图 2: Gradient Boosting 算法[8]

XGBoost 算法是在 GBDT 算法的一次提升，在 GBDT 基础上对 Boosting 算法进行了以下改进[9]：

1. 目标函数通过二阶泰勒展开式做近似
2. 定义了树的复杂度，并应用在目标函数中。
3. 分裂节点处通过结构打分和分割损失动态成长
4. 分裂节点的候选集合通过一种分布式 Quantile Sketch 得到
5. 通过特征的列采样防止过拟合。

XGBoost 建立回归树时使用的近似算法与贪婪算法如下图所示：

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ ***to*** m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.
  Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ ***to*** m **do**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max score only among proposed splits.

图 3：XGBoost 近似算法

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted(I, by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

图 3: XGBoost 回归树与贪婪算法

4.2 实验结果与对比

我们将基于决策树的 XGBoost(Exterme Gradient Boost)算法与 GBDT(Gradient Boost Decision Tree)算法，Naïve Bayes 算法和 SVM 算法进行了对比，在相同的训练集中进行训练，四种算法在测试集中的表现如下表所示

算法类型	XGBoost	GBDT	Naïve Bayes	SVM
F1-Score	62.18%	58.68%	50.64%	49.50%

表 4: 四种算法测试结果对比

其中 XGBoost 展现出优良的性能，均优于其他三种学习器，我们选择 XGBoost 作为最终的结果。

5 总结与讨论

通过本次数据挖掘实验，我认识到真实世界数据预处理的困难程度，选择合适的特征会带来较好的训练效果；同时，集成学习对传统学习器在数据挖掘任务中往往有着很大的优势性。

References:

[1] Roy and Cordy, Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[2] Sajnani *et al.,* Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Soft- ware Engineering*, pages 1157–1168, Austin, TX, USA, 2016.

[3] Socher *et al.*, Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Man- ning. Semi-supervised recursive autoencoders for predict- ing sentiment distributions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 151–161, Edinburgh, UK, 2011.

[4] Svajlenko *et al.*, Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Moham- mad Mamun Mia. Towards a big data curated bench- mark of inter-project code clones. In *Proceedings of the 30th IEEE International Conference on Software Mainte- nance and Evolution*, pages 476–480, Victoria, BC, Cana- da, 2014.

[5] Tai *et al.*, Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic represen- tations from tree-structured long short-term memory net- works. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and*

the 7th In- ternational Joint Conference on Natural Language Pro- cessing of the Asian Federation of Natural Language Pro- cessing, pages 1556–1566, Beijing, China, 2015.

[6] Jiang *et al.*, Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Ste´phane Glondu. DECKARD: scal- able and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Soft- ware Engineering*, pages 96–105, Minneapolis, MN, USA, 2007.

[7] H.-H. Wei and M. Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17), Melbourne, Australia, 2017, 3034-3040.

[8] Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie, The Elements of Statistical Learning, Springer, 2011

[9] GitHub: XGBoost https://github.com/dmlc/xgboost

[10] http://djjowfy.com/2017/08/01/XGBoost%E7%9A%84%E5%8E%9F%E7%90%86/

[11] T. Chen and C. Guestrin, XGBoost: A Scalable Tree Boosting System, KDD, 2016