数据挖掘导论实验——软件代码查重

摘　要: 现阶段，人们对自然语言文本处理的研究已经取得了丰硕的成果。软件代码，作为一种特殊形式的文本数据，其处理方法的相关研究还不多。本文将探讨软件代码文本处理中的一个基础部分——软件代码查重，即代码相似度计算。首先介绍一种朴素但有效的非学习策略：公共数字/字符串法。接着介绍两种机器学习分类策略：神经网络和决策树，展示三种方法的实验效果，并分析产生这种效果的原因。

关键词: 代码查重;公共数字/字符串法;神经网络;决策树

1　导言

在如今的互联网时代，代码克隆已经成为了一种很普遍的现象。为了防止代码克隆可能造成的安全隐患和抄袭问题，软件代码查重变得至关重要。软件代码是一种形式特殊的文本数据，其词法和语法与自然语言文本有很大的不同。要实现对软件代码的有效查重，首先需要对代码的组成有清晰的了解。然后针对代码的特定组成部分，提取代码的特征，利用机器学习或其他手段进行数据挖掘。

2　代码的组成分析

不同语言的代码结构可能千差万别，但几乎所有语言的代码都可以分成以下几个部分

（1）保留字（关键字）：用于指示分支，循环等结构，例如 C 语言中的 for, while, if, else；还有一些与语言特性有关，比如 struct, union, class.

（2）类型名：C 语言中的 int, double, char 等；可以视为保留字；有些语言如 Python，Matlab 中不需要类型名。

（3）操作符：用于表达式运算，例如+, -, *, /, &&, ||, !等，可以视为保留字。

（4）变量名（及函数名等）。

（5）数字/字符串：这里指的是在源代码中直接给出的数字/字符串，即在编译时即可确定的数字/字符串，例如语句 char *a = “hello, world”;中的“hello, world”，或者语句 int x = 12;中的 12；这是相当具有挖掘价值的一类数据，公共数字/字符串法便是利用这部分信息完成查重的。

（6）停用词（Stop words）：完全没有价值的文本部分，主要是标点符号，例如各种括号，逗号，分号等。

对代码文本特征的提取，无非是围绕上面几个部分进行。

需要注意的是，以上几个部分，对代码类别特征的刻画程度是不同的，有些部分几乎可以唯一地标识某一类代码，而有些则不能。例如，有些保留字和操作符是很常用的，有些则十分罕见，下面是部分保留字、类型名和操作符在测试集 10000 个代码文件中的平均出现次数：

表 1　保留字、类型名和操作符的平均出现次数

名称	平均出现次数
if	3.9608
else	1.5977
loop(for 和 while)	3.081
add(+)	9.3245
sub(-)	3.0751
mul(*)	1.9644
div(/)	0.7241
and(&&)	2.1384
or(||)	0.5846
not(!)	1.2003
bigger(>)	2.9015
smaller(<)	6.0758
equal(=)	18.7691
dot(.)	0.8978
long	0.0119
double	0.4022

int	6.0779
char	0.8608
unsigned	0.0011
void	0.0737

假设我们以词频作为特征，对于 long, unsigned, void 这样出现次数极少的类型名，可能仅凭单个特征的“有无”就能判断两则代码是否属于一类。对于 if, loop, add 这样出现次数很多的保留字，可能需要结合多个特征的统计规律，选用适当的模型，才能区分不同类别的代码。

当然，一个最根本的问题是：词频到底是不是一个很好的特征？为了实现相同的功能，代码的表达可能千差万别，比如 if-else 在某些情况下可以用 switch-case 代替，甚至用循环实现，而不同的表达方式，可能使词频的分布发生很大的变化，分类器对这种差异的容忍度，会在一定程度上影响其表现。

对于变量，需要考虑到变量名经过了混淆，不仅同类代码中可能有相同名称的变量，不同类代码中也会出现相同名称的变量，这就给追踪两代码中作用相同或相似的变量带来了难度。当然，实验数据中也存在少数没有混淆的变量，即不以“VAR_”开头的变量，这类变量可以看作特殊的字符串，用后面介绍的公共数字/字符串法进行处理。

对于变量，还可以采取与保留字类似的处理方式，统计某类型变量的词频。这里的“词频”可以分为两种：第一种是某类型变量的定义/声明次数，第二种是某类型变量的总出现次数。由于“总出现次数”更加容易受到表达方式的影响，所以我选用了“变量的定义/声明次数”作为某类型变量的词频。

提取保留字和变量的词频后，每个代码文件便形成了一个特征向量，可以用于后续的机器学习训练。

3 公共数字/字符串法

3.1 公共数字/字符串对代码类别的刻画

如果说保留字和变量与代码类别的联系还不太紧密，那么数字/字符串则能在很大程度上反映代码的类别信息。我们注意到，实现同一功能的代码必定有相同的输入和输出（假设它们的代码逻辑是正确的），而这些输入和输出虽然很多是在运行时决定的，但不乏一些预先存储在源代码中的。另外，还有一些与输入输出无关，但程序运行时需要用到的数字/字符串，也可作为代码分类的判据。最重要的一点，这种公共数字/字符串法几乎不受语法结构的影响，我们关心的是数字/字符串本身，而不是它出现在哪个语法结构中。

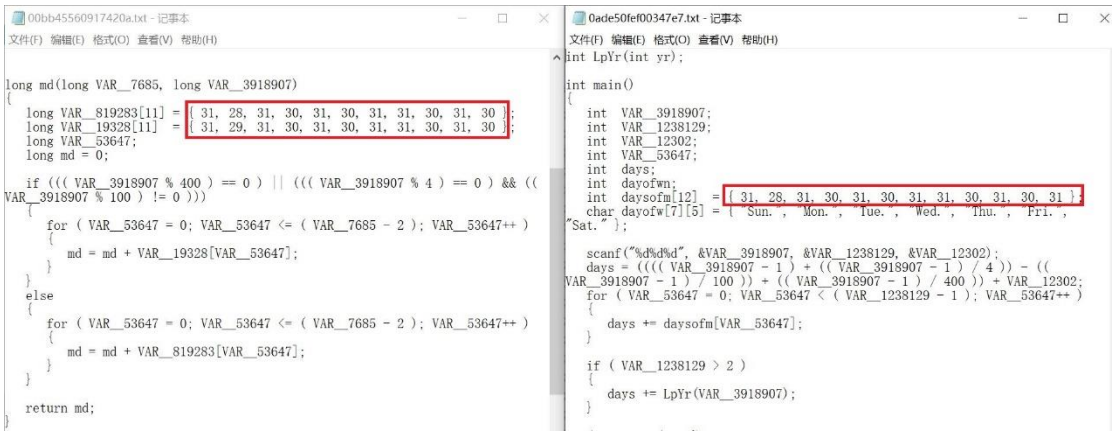

图 1　公共数字

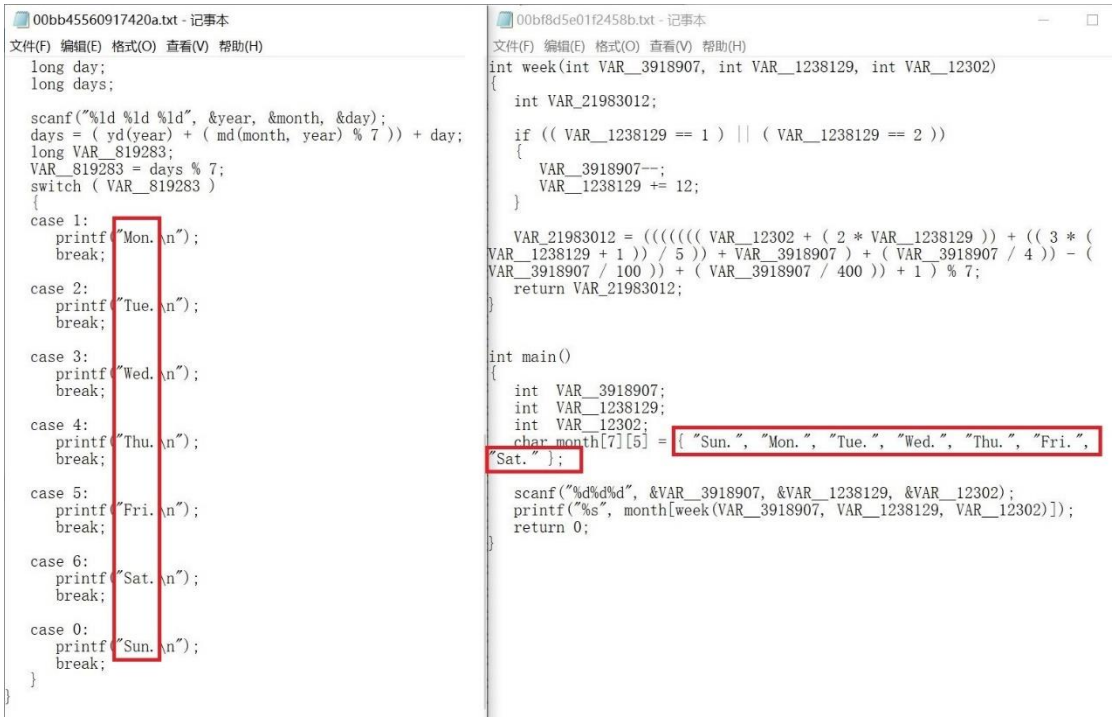

图 2　公共字符串

从上图可以看到，公共字符串可以出现在不同的语法结构中，例如 printf 函数的参数，字符串数组。

相比于公共字符串，公共数字对于类别的刻画能力要弱一些，有些很平凡的数字可能出现在任何类别的代码中，例如 0 和 1，在提取代码中的数字时，要将这些数字排除。

3.2　数字/字符串的提取

具体的提取方式比较简单，对于数字，只要在代码中搜索所有 0~9 打头的位置，然后找出位置开始的最长数字字符串（包括小数点）即可，不计正负号，只取绝对值。

对于字符串，找双引号或单引号打头的位置，提取引号中的内容即可。

每处理一则代码，输出一个文件，存放该代码提取出的数字/字符串。

3.3　代码同类与否的判定

现有代码 A, B, 分别提取 A, B 中的数字/字符串，形成 set_A, set_B, 设 set_A 中元素个数为 $size\_A, set\_B$ 中元素个数为 $size\_B$. 对 set_A 和 set_B 取交集，形成 set_C, 元素个数为 $size\_C$. 定义：

$$min\_hit = min\{size\_C / size\_A, size\_C / size\_B\}$$

$$max\_hit = max\{size\_C / size\_A, \text{size\_C} / size\_B\}$$

设定一个阈值 t, 当 $min\_hit > t$ 时，判定 A 与 B 为一类，否则不为一类。

或者，设定一个阈值 t', 将所有代码对按照关键字顺序：$min\_hit, max\_hit, size\_C$ 降序排序，将前 t' 个代码对判为正例。

你可能要问了，为什么不取 max_hit, 而偏取 min_hit, 一开始，我的确取的是 max_hit, 但在保留同样多正例的情况下，取 max_hit 的 F1-score 要明显低于取 min_hit 的情况，这可能是因为在某些代码对中，$size\_A$ 和 $size\_B$ 极度不平衡，导致 set_A, set_B 中较小的集合命中率虚高，从而产生了误导性的结果。为了说明这一点，我从训练集中采样得到了一个小型的测试集，该测试集有 415 个正例和 7750 个负例：

表 2　*max_hit* 与 *min_hit* 的表现差异

指标名称	指标值	指标名称	指标值
TP	165	TP	241
FP	250	FP	174
TN	7500	TN	7576
FN	250	FN	174
precision	0.3976	precision	0.5807

recall	0.3976	recall	0.5807
F1-score	0.3976	F1-score	0.5807
accuracy	0.9388	accuracy	0.9574
备注	按照关键字顺序：*max_hit*, *min_hit*, *size_C* 降序排序，取前 415 个为正例	备注	按照关键字顺序：*min_hit*, *max_hit*, *size_C* 降序排序，取前 415 个为正例

还有人会问：为什么只考虑“有没有”的问题，而不考虑“有多少”的问题？为什么不把每个数字/字符串出现的次数考虑进来呢？我也尝试过将 *set_A*, *set_B*, *set_C* 中的字符串在代码中的总出现次数，代替 *size_A*, *size_B*, *size_C* 进行判别，但是效果不佳。主要是因为虽然数字/字符串本身不受表达方式影响，但数字/字符串出现的次数却受表达方式影响，换句话说，“有没有”比“有多少”要重要得多。

3.4 缺陷

不幸的是，这种基于集合求交的判定法并不是那么的灵活，最大的缺陷就是无法应对字符串的拆分。例如“Mon.”可以拆成“Mon”和“.”两部分，为了应对这种细微的变化，可以不再严格求交，而是将 *set_A*, *set_B* 中编辑距离小于 1 的元素数量作为 *size_C*，但这样会将许多毫无语义关联的字符串关联到一起，从而降低准确度，所以不宜采用。

另一方面，即使采用了这样那样的替代方案，还是无法囊括所有的拆分方式，在最极端的情况下，甚至有逐字符拆分的情况，比如“Mon”拆分成“M”，“o”，“n”，这种模式很难在不降低准确率的情况下识别出来。

另一个问题是：有些代码中根本没有非平凡的数字/字符串，例如，在 10000 则代码的测试集中，就有 162 则没有提取出任何数字/字符串，一旦这些代码与其他代码比较，由于 *min_hit* 默认置为 0，一定会被判为不同类。同时，注意到有些代码虽然提取出了数字/字符串，但提出的数字/字符串数量极少，如果参与比较的两则代码都是这种情况，则极可能出现命中率虚高的情况。

4 基于机器学习的方法

4.1 特征的组成

“代码的组成分析”一节提到，保留字和变量的词频组成了大部分特征，但再此基础上，我还添加了两个特征：数字总长度和字符串总长度，用以刻画数字/字符串的类别特征。这两个特征值来自于公共数字/字符串法中对数字/字符串的提取，每则代码中所有数字会被放入一个集合中，字符串也会被放入一个集合中，只要对数字位数/字符串长度分别求和即可。注意到，相同类别的代码，其数字/字符串集合应该是大致相同的，于是其数字总长度和字符串总长度也应该大致相同，所以数字总长度和字符串总长度在一定程度上能够刻画代码的类别特征。不用“总个数”而用“总长度”是因为总个数刻画类别的能力要更加弱一些，两则不同类别的代码拥有相同个数的数字/字符串应当是很常见的现象。

经过上面的处理，每个代码文件提取出了 22 个特征向量，它们分别为：

3 个保留字的词频：if, else, loop(for 和 while);

11 个操作符的词频：add(+), sub(-), mul(*), div(/), and(&&), or(||), not(!), bigger(>), smaller(<), equal(=), dot(.);

6 个类型名的词频：long, double, int, char, unsigned, void;

2 个额外的特征：数字总长度和字符串总长度。

4.2 代码同类与否的判定

代码查重不同于传统的机器学习分类，我们的最终目的不是判断两则代码分别属于哪一类，并不需要输出两则代码各自的具体类别，只需要计算出两者的相似度。某位同学提出的 Siamese 网络的确能够做到这一点，该网络是一种孪生的神经网络，一次要提供两组输入[1]，对于代码查重这个特定的问题来说，如何决定每次提供的两组输入，如何对训练集进行组织、配对，都是需要考虑的问题。

传统的神经网络以及其他机器学习方式也是可以解决代码查重的问题的，只不过它们预先限制了代码可选的类别范围，对于不属于训练集中任何一类的代码，它们是很难正确处理的。

假设测试集的确不会超出训练集中的代码类别范围，那么代码查重就变成了经典的多分类问题，判断两则代码是否属于同一类，只要判断两者所属的具体类别是否相同即可。然而，在本实验中，训练集中的代码类别相较于传统的多分类问题实在是太多了，因此，分类的准确率可能不会很高，如果采用“具体类别相同”这一过于严格的标准，很可能效果不理想。一个补救措施是：将两则代码属于每一类的概率求均方差，均方差在某一阈值以下的判为正例。

4.3 神经网络

为了简单起见，我采用了最原始的神经网络结构[2]，该网络只有一层隐含层，为一 22×83 的权值矩阵 **W**，列数为 83 是因为训练集有 83 类。输入为 40670 则代码的特征向量组成的矩阵 **X**，大小为 40670×22，行数为 40670 而非 83 * 500 = 41500 是因为每一类抽取了 10 则代码作为测试集。另外，还有一个行向量 **b** 作为偏置系数，将 **b** 加到 **XW** 的每一行上，再用 softmax 函数归一化，得到最终的输出矩阵 **Y**, 即

$$\mathbf{Y} = \text{softmax}(\mathbf{XW} + \mathbf{b})$$

输出矩阵的大小为 40670×83，每一行代表该代码属于 83 个类别的概率。我们的目标输出是矩阵 **Y**’，每行是反映该代码正确类别的 one-hot 码，即只有列号等于正确的类别编号时，该位置上的值为 1，其他位置上的值均为 0. 对于这样的多分类问题，我们采用交叉熵（cross-entropy）作为损失函数，表达式如下：

$$L(\mathbf{y}', \mathbf{y}) = -\sum_i y'_i \log y_i$$

其中，**y** 代表输出矩阵 **Y** 中的一行，**y**’代表目标矩阵 **Y**’中的一行，计算出每一行的交叉熵后再取均值，就得到整体的交叉熵，我们只要用梯度下降法不断减少交叉熵即可完成训练。下面是训练了 40000 次的模型，在小型测试集（415 正例，7750 负例）上的预测表现：

表 3 神经网络在小型测试集上的预测表现

指标名称	指标值	指标名称	指标值
TP	175	TP	173
FP	318	FP	242
TN	7432	TN	7508
FN	240	FN	242
precision	0.3550	precision	0.4169
recall	0.4217	recall	0.4169
F1-score	0.3855	F1-score	0.4169
accuracy	0.9317	accuracy	0.9407
备注	只有具体类别相同才判为同类	备注	将均方差最小的 415 个代码对判为正例

虽然利用均方差判别的 F1-score 要略高一些，但是 FN 却比“具体类别相同”判别结果还要高，这有点不符合预期，因为在类别如此多的情况下，两则代码恰好同类的概率应当很小，“具体类别相同”这一严格标准，应该会将很多本是同类的判为不同类，FN 应当更高一些，造成这种现象的原因可能与均方差阈值的选取有关。

同时，我们也看到这种简单的神经网络在小型测试集上并不能取得很好的效果，而事实证明，它在正式的测试集上也没有取得很好的效果。

4.4 决策树

在非神经网络的学习方法中，决策树应当是与代码查重这一问题契合度较高的。因为代码分类往往需要综合多种属性进行判断，比如 if 的词频在某范围内，同时 loop 的词频在另一范围内，而 add 的词频又在某一范围内的代码被分为一类，这样的分类判据正好对应决策树中的一条路径。拥有这样自然的联系，决策树分类的准确率应当较高。下面是 scikit-learn 提供的 CART 决策树[3]在小型测试集（415 正例，7750 负例）上的预测表现：

表 4 决策树在小型测试集上的预测表现

指标名称	指标值
TP	293
FP	21
TN	7729
FN	122
precision	0.9331

recall	0.7060
F1-score	0.8038
accuracy	0.9825
备注	只有具体类别相同才判为同类

可以看到，决策树在小型测试集上取得了相当不错的效果，F1-score 超过了 0.8. 但是，由于决策树无法提供代码属于各类的概率，不能采用均方差的方式控制正例数量，只能采用“具体类别相同”的严格标准，导致在正式测试集上出现了正例过少的情况（如何确定正式测试集中的正例数量会在后面介绍），正例数只有 4716，远小于预期值 10006. 既然正例数已经这么少了，这些正例的质量如何呢，真正例占多大比例呢？为了增加正例数量，我一方面保留了决策树中的所有正例，另一方面选取了公共数字/字符串法预测结果中命中率较高的一些代码对，凑齐了大约 10000 个正例，不幸的是，F1-score 只有 0.60 左右，比单独使用公共数字/字符串法的 F1-score（0.65 以上）还要低，说明在决策树在正式测试集上预测效果不佳。

之所以会出现这种现象，可能是因为我假定了测试集中的所有代码都不会超出训练集的 83 个类别。由于小型测试集是直接从训练集中选出的，所以符合这一假设，但正式数据集却不一定符合。

5 正式测试集中的正例数量

已知 F1-score 的公式为

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

其中

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

假设全部预测为 1，则$TP + FP = 200000$，$FN = 0$，于是有

$$precision = \frac{TP}{200000}$$

$$recall = 1$$

又知道全部预测为 1 的情况下$F_1 = 0.09529$，所以

$$0.09529 = \frac{2 \cdot \frac{TP}{200000} \cdot 1}{\frac{TP}{200000} + 1}$$

解上述方程，得

$$TP \approx 10005.7 \approx 10006$$

所以，正式测试集中大约有 10006 个正例，任何性能较好的分类器在正式测试集上预测的正例数量应该接近这个数字，但是反过来不一定成立，对于一个分类器来说，使正例数量恰好为 10006 也并不一定能获得最好的预测效果，因为我们的分类器还没有达到那么优秀。

另一方面，我们可以仿照正式测试集的正负例比例生成自己的小型测试集，这也就是我的小型测试集有 415 个正例和 7750 个负例的原因，正负例比例接近于正式测试集中 1:20 的比例。

6 结论

德国建筑学家 Mies Van der Rohe 提出“Less is more”，有的时候，一些很简单、朴素的方法就能够取得不错的效果。在代码查重这一问题上，公共数字/字符串法这种利用极少代码信息的非学习方法也能够媲美一些相对复杂的机器学习方法。这同时也印证了另一著名定律“No free lunch”，只有最适合特定问题的算法才是最好的算法，成功的关键不在于你的方法有多复杂，而在于你的方法是否抓住了问题的本质。

当然，这并不意味着机器学习方法在代码查重问题上不能取得较好效果，只是因为我采用的机器学习算法过于简陋，没有经过优化，所以才表现不佳。

References:

[1] S. Chopra, R. Hadsell, Y. LeCun. Learning a Similarity Metric Discriminatively, with Application to Face Verification. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005,1:539-546.

[3] scikit-learn User Guide, 1.10. Decision Trees, https://scikit-learn.org/stable/modules/tree.html

附中文参考文献:

[2] TensorFlow 中文社区 TensorFlow 官方文档中文版 MNIST 机器学习入门 http://www.tensorfly.cn/tfdoc/tutorials/mnist_beginners.html