华中科技大学

计算机科学与技术学院

《机器学习》结课报告

专　　业：计算机科学与技术

班　　级：CS2003

学　　号：U202015374

姓　　名：张隽翊

成　　绩：

指导教师：何　琨

完成日期：　2022 年 7 月 5 日

目　录

Adaboost 算法实现

1. 实验要求

本实验分别实现了以对数几率回归和决策树桩为基分类器的 AdaBoost 算法，能够对给定的测试集做出预测。main.py 代码可以读取 data.csv 及 targets.csv 两个文件，并输出在不同数目基分类器条件下的 10 折交叉验证的预测结果至 experiments/base#_fold#.csv，以供评测。

在对数几率回归中，我尝试了调整迭代次数和学习率，但效果并不显著，最后选择了相对合适的值，既能满足速度要求，也能保证一定的精度。其中使用了动态调整学习率的方法，最后采用的是多项式下降法。此外，我也尝试对分类阈值进行了调整，观察拟合效果。

在决策树桩的实现中，我主要通过调整迭代次数对模型进行修改。对于阈值的选择，我采用在一定步长范围内遍历属性列下所有可能的阈值，找到具有最低错误率的单层决策树。

2. 算法设计与实现

2.1 数据预处理

实验中在数据预处理方面，我考虑过数据清理，即通过填补缺失值、光滑噪声数据、平滑或删除离群点等。观察给定的训练集，发现没有变量缺失，但有些特征列中“0”的个数明显过多。使用如下代码进行统计：

```
1  data = pd.read_csv("data.csv", header=None)
2  tmp = (data == 0).sum(axis=0) / len(data)
3  print(tmp)
```

图 2-1 统计数据集中“0”的状况

部分统计结果如图 2-2 所示，共有 57 行（数据集有 57 列）。

```
0     0.766304
1     0.808424
2     0.585598
3     0.989402
4     0.614674
5     0.780978
6     0.822826
7     0.814674
8     0.829620
9     0.710054
10    0.844837
11    0.491848
12    0.811685
13    0.920924
14    0.925543
15    0.725000
16    0.783152
17    0.776359
18    0.294022
19    0.907609
20    0.467935
21    0.975000
22    0.850543
```

图 2-2 数据集统计结果

可以看到，第 3 列和第 21 列中“0”的比例都超过了 0.95，有理由认为其重要性不高，故将其从数据集中删除。

```
# 数据预处理
def pre_process(data):
    for col in range(data.shape[1]):
        if (data[:, col] == 0).sum() / data.shape[0] >= 0.95:
            del_list.append(col)
    data = np.delete(data, del_list, axis=1)
    return data
```

图 2-3 数据预处理代码

除此之外，我还对训练数据采用了 Z-Score 标准化：

$$x_{new} = \frac{x - \mu}{\sigma}$$

其中μ是样本数据的均值（mean），σ是样本数据的标准差（std）。经过处理后的数据变为一个均值为 0，方差为 1 的分布。

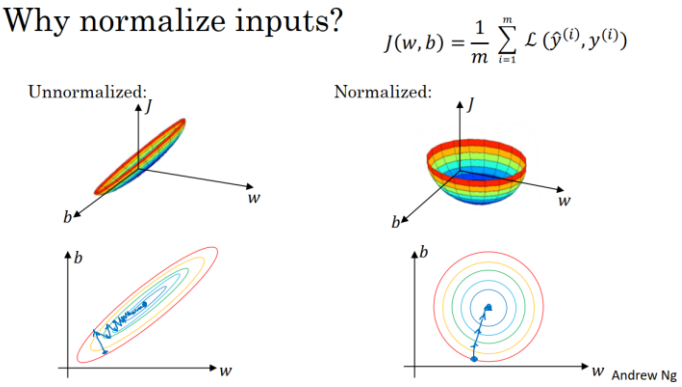

图 2-4 标准化数据对模型收敛的影响

标准化后可以更加容易得出最优参数ω和b以及计算出$J(\omega, b)$的最小值，从而达到加速收敛的效果。

2.2 对数几率回归实现

（1）对数几率回归介绍

对数几率回归（Logistic Regression，也即逻辑回归）假设数据服从伯努利分布，采用极大似然估计（MLE）的思想，运用梯度下降法求解参数，从而达到将数据进行二分类的目的。

二分类任务的输出标记$y \in \{0, 1\}$，而线性回归模型产生的预测值$z = \omega^T x + b$是实值，需要对其进行转换。最理想的是单位阶跃函数（unit-step function），即若预测值 z 大于 0 分为正例，小于 0 分为负例，临界零值则可任意判别。但是单位阶跃函数并不连续，我们使用对数几率函数（logistic function）作为替代：

$$y = \frac{1}{1 + e^{-z}}$$

它是一种 Sigmoid 函数，将 z 值转化为一个接近 0 或 1 的值，且输出值在 z = 0 附近变化很陡峭。将对数几率函数带入线性模型，得到：

$$y = \frac{1}{1 + e^{-\omega^T x + b}}$$

这实际上是在用线性回归模型的预测结果去逼近真实标记的对数几率，是一种分类学习方法。对率回归无需事先假设数据分布，有很好的数学性质。

（2）对数几率回归关键代码

```
def fit(self, train_data, train_label, weights):
    """
    :param train_data: 训练数据
    :param train_label: 训练标签
    :param weights: 样本权重

    :return: None
    """
    train_data = np.insert(train_data, 0, 1, 1)  # 在数据集的第 0 列增加一列 1
    self.init_args(train_data)
    label = np.where(train_label == -1, 0, 1)  # 将标签为 -1 的改为 0
    # 梯度下降算法
    for k in range(self.n_iters):
        # z = w * T + b
        z = np.dot(train_data, self.theta)
        # h = sigmoid(z)
        h = self.sigmoid(z)
        # 计算梯度 gradient
        g = np.dot((weights.T * (h.T - label.T)).T, train_data)
        if np.linalg.norm(g) <= 1e-3:  # 梯度足够小时可视为达到极值点
            break
        self.theta -= g * self.lr * (0.9 ** (k * 10.0 / self.n_iters))
    # 计算误分率
    self.clf_result = np.where(self.sigmoid(np.inner(self.theta, train_data)) >= self.threshold, 1, -1)
    self.error = np.sum((self.clf_result != train_label) * weights)
```

图 2-5 对率回归关键代码

2.3 决策树桩实现

（1）决策树桩介绍

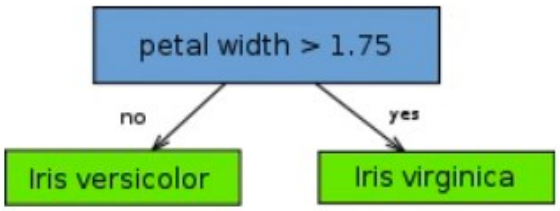

图 2-6 决策树桩示意图

决策树桩是由单层决策树组成的机器学习模型，每次只能选取一个特征，以某一阈值进行分类。从树结构上看，决策树桩由一个内部节点于叶节点直接相连，用作分类器的决策树桩的叶节点保存最终的分类结果。从实际意义上看，决策树桩根据一个属性的单个判断确定最终分类结果，比较适合用作集成学习中的弱学习器，因为它至少优于随机猜测，且计算起来比较简单。我们需要寻找具有最低错误率的决策树桩，有以下优化函数：

$$\arg\min_{1\le i\le d}\frac{1}{N}\sum_{n=1}^{N}1_{y_n\neq g_i(\mathbf{x})}$$

其中 i 表示属性列，N 为样本集大小，d 为属性列的个数。

（2）决策树桩关键代码

```
def fit(self, train_data, train_label, weights):
    """
    :param train_data: 训练数据
    :param train_label: 训练标签
    :param weights: 样本权重

    :return: None
    """
    self.init_args(train_data)
    # 选择误差最小的特征维度
    for col in range(self.N):
        features = train_data[:, col]  # 第 col 列特征
        cur_v, cur_direct, cur_err, cmp_array = self.best_threshold(features, train_label, weights)
        if cur_err < self.error:
            self.error = cur_err
            self.best_v = cur_v
            self.direct = cur_direct
            self.best_axis = col
            self.clf_result = cmp_array
        if self.error == 0.0:
            break
```

图 2-7 决策树桩关键代码

2.4 Adaboost 算法实现

（1）AdaBoost 介绍

AdaBoost 是英文“Adaptive Boosting”（自适应增强）的缩写，是一种机器学习方法，由 Yoav Freund 和 Robert Schapire 提出。AdaBoost 方法的自适应在于：前一个分类器分错的样本会被用来训练下一个分类器。AdaBoost 方法对于噪声数据和异常数据很敏感，但在一些问题中，它相对于大多数其它学习算法而言不那么容易出现过拟合现象。AdaBoost 算法中使用的分类器可能很弱（比如出现很大错误率），但只要其分类效果优于随机猜测，就能够改善最终得到的模型。而错误率高于随机分类器的弱分类器也是有用的，因为在最终得到的多个分类器的线性组合中，可以给它们赋予负系数，同样能够提升分类效果。

AdaBoost 方法是一种迭代算法，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率时停止。每一个训练样本都被赋予一个权重，代表它被某个分类器选入训练集的概率。如果某个样本点已经被准确分类，那么在构造下一个训练集时它被选中的概率会降低；相反，如果某个样本点没有被准确分类，它的权重就得到提高。通过这样的方式，AdaBoost 算法能“聚焦于”那些较难分类（即更富信息）的样本。

（2）AdaBoost 算法伪码

Input: ℓ, α, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H_0 = 0$
$\forall i : w_i = \frac{1}{n}$
for *t=0:T-1* **do**
  $h = \text{argmin}_h \sum_{i:h(\mathbf{x}_i)\neq y_i} w_i$ $[h = \mathbb{A}((w_1, \mathbf{x}_1, y_1), ..., (w_n, \mathbf{x}_n, y_n))]$
  $\epsilon = \sum_{i:h(\mathbf{x}_i)\neq y_i} w_i$
  if $\epsilon < \frac{1}{2}$ **then**
    $\alpha = \frac{1}{2}\ln(\frac{1-\epsilon}{\epsilon})$
    $H_{t+1} = H_t + \alpha h$
    $\forall i: \ w_i \leftarrow \frac{w_i e^{-\alpha h(\mathbf{x}_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}$
  else
    return (H_t)
  end
end
return (H_T)

图 2-8 AdaBoost 算法伪代码描述

（3）AdaBoost 关键函数

```
# 初始化参数
    def init_args(self, datasets, labels):
        self.X = datasets  # 训练集
        self.Y = labels  # 训练标签
        self.M, self.N = datasets.shape  # 初始化 M, N
        self.weights = np.ones(self.M) / self.M  # 初始化权重为 1 / M

    # 计算 alpha
    def _alpha(self, error):
        return 0.5 * np.log((1 - error) / error)

    # 权值更新并规范化
    def _w(self, a, clf):
        self.weights *= np.exp(-1 * a * self.Y * clf)
        sum_of_weights = sum(self.weights)
        self.weights /= sum_of_weights  # 归一化权重
```

图 2-9 AdaBoost 关键函数

3. 实验环境与平台

PC：Lenovo Legion R7000 2020

Python 版本：Anaconda3 Python 3.9.12

工具：PyCharm 2022.1.3 Professional + VS Code 1.68.1

CPU：AMD Ryzen 5 4600H

GPU：AMD Radeon Graphics + NVIDIA GeForce GTX 1650

4. 结果与分析

（1）决策树桩迭代次数的设置

调整图 4-1 所示代码中的 n_steps 参数，寻找合适的迭代次数。

```python
class DecisionTreeStump:
    def __init__(self, n_steps=50):
        self.error = 0  # 最小加权错误率
        self.best_v = 0  # 选定特征的最优阈值
        self.best_axis = None  # 最佳特征索引
        self.direct = None  # 阈值符号，positive 为正向，negative 为反向
        self.clf_result = None
        self.n_steps = n_steps  # 迭代次数，默认为 50
        self.M, self.N = None, Non
```

图 4-1 决策树桩迭代次数参数示意

设置迭代次数从 10 到 100，每次增加 10，以不同基分类器数目下预测的准确率为纵坐标作图，如图 4-2 所示。

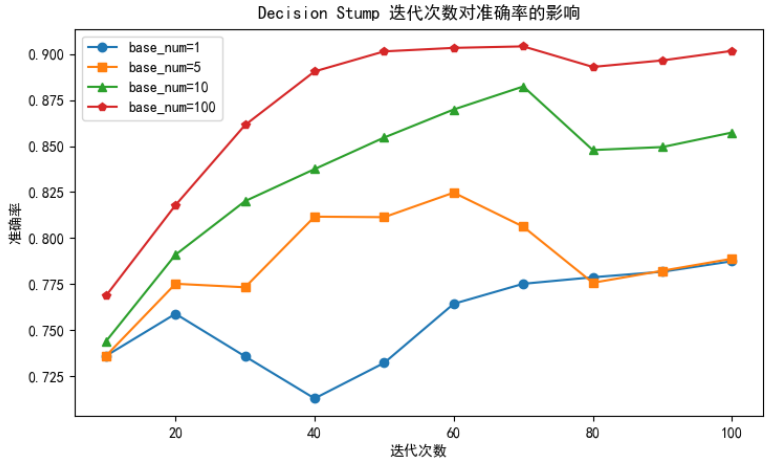

图 4-2 决策树桩迭代次数对准确率的影响

结合图像趋势，综合训练时间和准确率两个因素，我选择了 n_steps=60 作为最终的参数。使用该参数时在本地 evaluate.py 评测的结果如表 4-1 所示。

表 4-1 决策树桩本地评测结果

基分类器数目 base_num	准确率 accuracy
1	0.7644021739130434
5	0.8247282608695652
10	0.8698369565217391
100	0.9032608695652173

（2）对率回归分界阈值的设置

调整图 4-3 所示代码中的 self.threshold 参数，寻找合适的分界阈值。

```python
class LogisticRegression:
    def __init__(self, n_iters=200, learning_rate=1.0):
        self.n_iters = n_iters  # 迭代次数
        self.lr = learning_rate  # 学习率
        self.threshold = 0.5  # 分界阈值
        self.error = 0.0  # 错误率
        self.clf_result = None
        self.M, self.N = None, None
        self.theta = None  # 权重
```

图 4-3 对率回归分界阈值参数示意

设置分界阈值从 0.1 到 0.9，每次增加 0.1，以不同基分类器数目下预测的准确率为纵坐标作图，如图 xxx 所示。

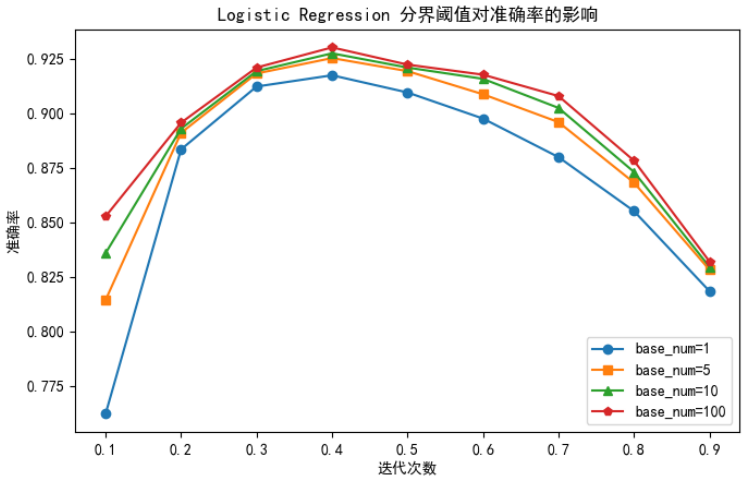

图 4-4 对率回归分界阈值对准确率的影响

根据图像的变化趋势，我选择了 self.threshold=0.4 作为最终的参数。使用该参数时在本地 evaluate.py 评测的结果如表 4-2 所示。

表 4-2 对率回归本地评测结果

基分类器数目 base_num	准确率 accuracy
1	0.9173913043478261
5	0.9252717391304348
10	0.9274456521739131
100	0.930163043478261

5. 个人体会

实现 AdaBoost 算法，我一开始时直接调用库函数，查阅资料分析其实现原理和方法，再自己尝试逐个模块复现，对不合适的地方进行改写和重构，最终替换掉了所有模型库。实验过程中我遇到过以下问题：

（1）训练速度很慢，准确率较低

刚开始的模型勉强能够运行，但速度非常慢。对于决策树桩，经常训练一次（基分类器数目取 10 个）要十多分钟，准确率则相对正常，说明算法逻辑基本正确，实现上可以改进。经过多次尝试，最终发现将代码中的 for 循环和列表操作替换为 numpy 内置函数可以极大提高程序速度。这是因为 numpy 提供了一个快速、高效、功能全面的数组：ndarray，可以使用它替换 Python 内置的列表。而对于对率回归，训练速度很快，但准确率一直不高。通过设置断点单步调试和跟踪变量等方法，最终定位到问题在于 fit 中有一句修改标签的代码使用的是引用而不是复制，导致标签被错误修改。提高准确率我主要通过将数据进行归一化处理和选择合适的列标签实现。

（2）梯度下降学习率的设置

一开始我将学习率设置的很大，导致参数几乎一样的模型得出的结果差别很大。在调试过程中，我逐步减小学习率，当不足以影响到最终的模型时再往回增大，直到选取到一个合适的范围。在此过程中，我也学习了梯度下降学习率的设置，如固定学习率、分参数设定、动态调整、自适应等，最终将动态调整应用到模型中。

（3）弱分类器错误率大于随机猜测时

分类误差为所有分类错误的样本权重之和，但是当其分类误差大于 0.5 的时候算法就停止迭代了，这一点主要体现在对率回归做基分类器时。解决这个问题主要有以下几种方法：

①选择较强的弱分类器；

②遇到错误率大于 0.5 时反转分类器的结果；

③遇到错误率大于 0.5 时重新设定权值；

④反复重采样和生成基分类器直到错误率小于 0.5。

实验中我采用的是第 2 中方法，即反转分类器的结果。

总得来说，这次大作业让我对 AdaBoost 算法有了更加深入的认识。在训练决策树桩为基分类器的模型中，我感受到了 AdaBoost 对于分类器强度的提升效果。而在对率回归为基分类器的模型中，AdaBoost 算法的提升没有那么显著，可能是单个基分类器的强度已经足够，也可能是由于对率回归已经是线性模型，使用线性加权得到的仍然是一个线性模型，没有改变模型的本质。还有不足的是，我在数据预处理方面下的功夫不够，最终实现的模型可能存在过拟合现象，这一点在以后的学习中我会注意。