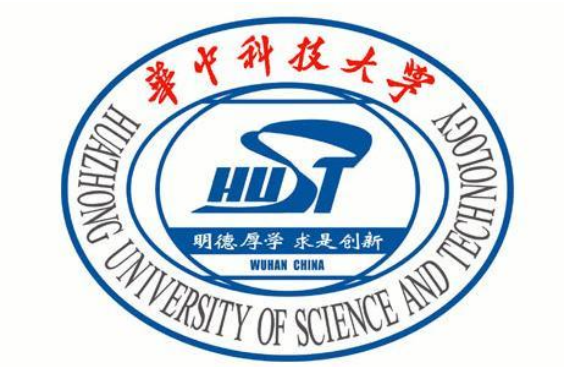
华中科技大学

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：2107 班

学　　号：U202115538

姓　　名：陈侠锟

成　　绩：

指导教师：张腾

完成日期：2023 年 5 月 14 日

目录

AdaBoost 算法实现

1. 实验要求

a) 总体要求

分别实现以对数几率回归和决策树桩为基分类器的 AdaBoost 算法，其中对数几率回归请参考课件 5.3 节，决策树桩指只有一层的决策树，决策树请参考课件第 12 章，AdaBoost 算法请参考讲义。

b) 输入输出格式说明

代码需读取 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。基分类器数目取 1，5，10，100 这四种数值。输入样例，输出样 例及评测代码详见提供的压缩包。

对预测结果所在文件命名格式说明如下：基分类器数目为 x 对应的预测文件为 basex_fold1.csv~basex_fold10.csv，1~10 指的是用作测试集的子集编号。每个预测文件分成两列，第一列为样例的序号（序号从 1 开始），第二列为该样例的预测标记。评测时子文件夹 experiments 会建立好，请不要在你的代码中强行建立此文件夹以免出错。

2. 算法设计与实现

a) 数据预处理

这次的实验中所给的数据是稀疏并且规则性较弱的，因此需要进行相关处理。本次实验我尝试了最大最小值归一化与均值归一化两种方法，并于不进行数据预处理时的预测结果进行比较，最终发现在均值归一化的情况下，预测准确率最高。数据预处理的代码如图 2.1 所示。

```python
# 数据预处理：归一化
def min_max_normalization(data):
    data_row, data_col = np.shape(data)
    for j in range(data_col):
        col_mindata = min(data[:,j])
        col_maxdata = max(data[:,j])
        for i in range(data_row):
            data[i,j] = (data[i,j] - col_mindata)/(col_maxdata - col_mindata)
    return data

# 数据预处理：均值归一化
def mean_normalization(data):
    data_row, data_col = np.shape(data)
    for j in range(data_col):
        col_mindata = min(data[:,j])
        col_maxdata = max(data[:,j])
        col_averdata = np.mean(data[:,j])
        for i in range(data_row):
            data[i,j] = (data[i,j] - col_averdata)/(col_maxdata - col_mindata)
    return data
```

图 2.1 数据预处理

b) 对数几率回归的实现

对数几率回归（Logistic Regression），也称逻辑回归，本质是一种分类算法，该模型在设计之初是用来解决 0/1 二分类问题。它的思想是找一个单调可微函数

将分类任务中的真实标记与线性回归模型的预测值联系起来，是一种广义线性回归。

i. 对数几率函数

对数几率回归的重要因素之一是对数几率函数，它是一种“Sigmoid”函数，即 S 形的函数，其形状如下图黑线所示：

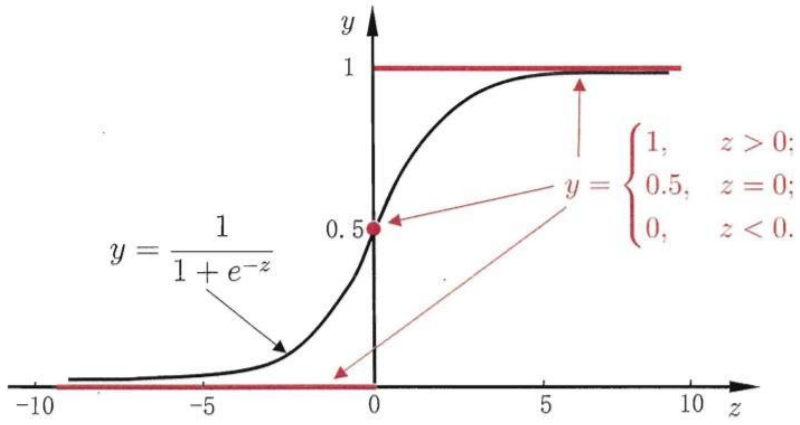


图 2.1 Sigmoid 函数

此函数具有非常好的数学性质，使用其做分类问题时，不仅可以预测出类别，还可以得到近似概率预测，同时它又是任意阶可导函数，很多数值优化算法都可以直接用于求取最优解。总的来说，模型的完全形式如下：

$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

图 2.2 对数几率回归模型

代码中的实现如下：

```
# 对数几率函数
def LR_sigmoid(self, z):
    return 1.0 / (1.0 + np.exp(-z))
```

图 2.3 sigmoid 函数代码实现

ii. 损失函数

对于任何机器学习问题，都需要先明确损失函数，通常我们会直接想到如下的损失函数形式（平均误差平方损失 MSE）：

$$L = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y)^2$$

图 2.4 平均误差平方损失 MSE

但在对数逻辑回归模型中，经查询资料得到的实际使用的损失函数如下：

$$L = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

图 2.5 对数损失函数

该损失函数的本质是 L 关于模型中线性方程部分的两个参数 w 和 b 的函数，即如下图所示：

$$L(w,b) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

其中，$\hat{y} = \frac{1}{1+e^{-z}}$，$z = w^T x + b$

图 2.6 对数损失函数是关于 w 和 b 的函数

由于损失函数连续可微，我们可以借助梯度下降法进行优化求解，以此根据预测值去更新 w 与 b 的值使分类效果更佳，经过一系列数学运算可得到优化公式如下：

$$W \leftarrow W - \alpha X^T (\hat{Y} - Y)$$

$$b \leftarrow b - \alpha (\hat{Y} - Y)$$

图 2.7 w 和 b 的优化公式

其中，α 为学习率。因此，具体代码实现如下图所示：

```
error = label - y_sigomid
self.theta += learning_rate * (np.dot(data_expan.T, weights * error))
```

图 2.8 w 和 b 的优化的代码实现

其中，label 是真实的 Y，y_sigmoid 是通过 sigmoid 函数预测出来的 Y，weights 是各个数据对应的权重，由于 AdaBoost 算法的要求是每一轮训练基分类器都要对上一轮分类错误的数据进行更进一步的考量，因此这里需要乘上 weights，否则将会使得每一个基分类器的性能几乎一样。

iii. 训练函数及预测函数

首先定义 theta 变量如下：

```
self.theta = np.zeros((data.shape[1] + 1, 1))
```

图 2.9 theta 变量的定义

前 data.shape[1]个元素代表各数据的 w，最后一个元素代表 b，为确保矩阵乘法的时候大小匹配，data 也应扩展一列变成 data_expan.

训练的关键是在一轮轮训练中更新 theta 的值，以使其分类性能逐渐提高，应注意不断调试 learning_rate 和 train_round 的值，以便使得到的 theta 最贴近预期。当得到一组 theta 后，就可以用其带入 sigmoid 函数进行预测，预测值大于 0.5 认为类别为 1，否则认为类别为 0。相关代码如下图所示。

```
def LR_train2(self, data, label, weights, learning_rate = 0.01, train_round =9000):
    learning_rate = 0.03
    row, col = data.shape
    #theta = np.zeros((col + 1, 1))
    data_expan = np.c_[data, np.ones(row)]  #c_函数：列连接
    for i in range(train_round):
        y_sigomid = self.LR_sigmoid(np.dot(data_expan, self.theta))
        error = label - y_sigomid
        self.theta += learning_rate * (np.dot(data_expan.T, weights * error))
    return self.theta

def LR_predict2(self, data, theta):
    data_expan = np.c_[data, np.ones(data.shape[0])]
    y_sigmoid = self.LR_sigmoid(np.dot(data_expan, theta))
    #print(theta.shape)
    y_sigmoid[y_sigmoid > 0.5] = 1
    y_sigmoid[y_sigmoid <= 0.5] = 0
    return y_sigmoid
```

图 2.10 训练函数及预测函数

c) 决策树桩的实现

决策树是一种非参数监督学习算法，用于分类和回归任务，它具有分层的树的结构，由根节点、分支、内部节点和叶节点组成，其主要结构如下图所示。

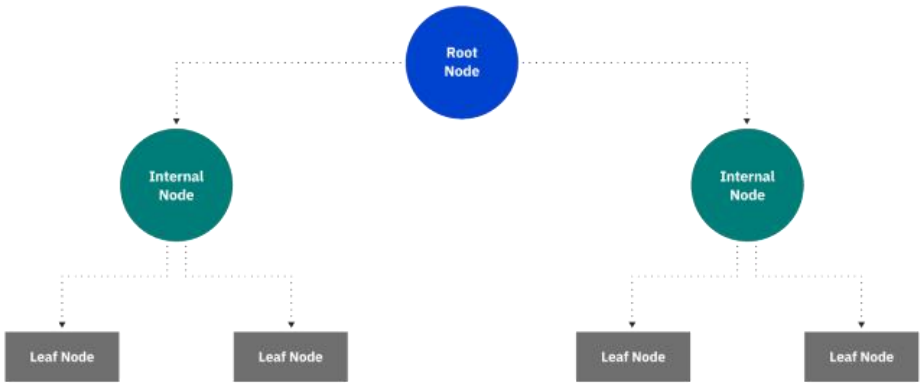


图 2.11 决策树的主要结构

决策树学习采用分而治之的策略，通过贪婪搜索来识别树中的最佳分割点。然后以自上而下的递归方式重复此拆分过程，直到所有或大多数记录都已分类到特定的类别标签下。本次作业仅要求实现决策树桩，即只有一层的决策树，因此只需要找到原始数据中分类准确率最高的那个特征的取值即可。

算法实现如下图所示。

```
# 通过遍历所有的特征，每个特征取不同的阈值进行测试，找到分类效果最佳的特征和阈值
def DT_build_stump(self, data, label, weights, learning_rate):
    data_mat = np.mat(data)
    label_mat = np.mat(label)
    label_mat[label_mat == 0] = -1
    m, n = data_mat.shape
    y_predict = np.mat(np.zeros(shape=(m, 1)))
    min_error = float('inf')
    for i in range(n):  # 遍历所有的特征
        unique_feature = np.unique(data[:, i])
        for j in range(len(unique_feature)):    # 遍历第i个特征下的所有不同值，尝试将其（+0.001）作为阈值
            for method in ['lt', 'gt']:  # 遍历所有判断方法，大于阈值赋1或者小于阈值赋1
                threshold = unique_feature[j] + 0.001
                # 计算在当前分支阈值条件下，决策树的分类结果
                y_predict_temp = self.DT_stump_classify(data_mat, i, threshold, method)
                # error矩阵用于保存决策树的预测结果
                error = np.mat(np.ones(shape=(m, 1)))
                # 将error矩阵中被当前决策树分类正确的样本对应位置的值置为0
                error[y_predict_temp == label_mat] = 0
                # 计算分类错误率（按位相乘并求和），错误率=所有分类错误样本的权重求和
                weights_error = weights.T * error
                # 如果误差率降低，保存最佳分类方法的相关信息
                if weights_error < min_error:
                    #print('i的值是：',i)
                    min_error = weights_error
                    y_predict = y_predict_temp.copy()
                    best_feature = i
                    best_threshold = threshold
                    best_method = method
    return best_feature, best_threshold, best_method, min_error, y_predict
```

图 2.12 决策树桩的实现

首先要对原数据集的所有特征进行遍历，针对每一个特征，遍历原始数据中所有不同的取值，并以此值加上 0.001 作为分类阈值（用变量 threshold 表示），调用 DT_stump_classify 函数针对此阈值进行分类，用变量 error 统计此次分类错误的数据，用 weights_error 计算分类错误率，注意这里用了 weights 与 error 相

乘，这样是为了搭配 AdaBoost 算法使用，使每一个基分类器的性能有所提升。遍历完所有特征的所有可能取值之后，可以找到使错误率最小的特征及阈值，即得到了一组弱基分类器。

需要注意的是，由于不能判断阈值前后的数据究竟属于哪种类别，因此需要讨论两种情况，一种是当特征取值大于阈值时，归类为类别 1，另一种是特征取值小于阈值时，归类为类别 1。这两种可能在代码中用‘lt’‘gt’代表。

在预测时，只需要使用已经得到的一组特征与阈值，调用一次 DT_stump_classify 函数即可。

d) AdaBoost 算法的实现

AdaBoost 是 Adaptive boosting 的缩写，基本原理是将多个弱分类器进行合理的结合，使其成为一个强分类器。

AdaBoost 采用迭代的思想，每次迭代只训练一个弱分类器，训练好的弱分类器将参与下一次迭代的使用，也就是说，在第 N 次迭代中，一共就有 N 个弱分类器，其中 N-1 个是以前训练好的，其各种参数都不再改变，本次训练第 N 个分类器。其中弱分类器的关系是第 N 个弱分类器更可能分对前 N-1 个弱分类器没分对的数据，最终分类输出要看这 N 个分类器的综合效果。

AdaBoost 算法框架如下。

Algorithm 1: AdaBoost 算法

输入：训练数据集 $\mathcal{S}=\{(\boldsymbol{x}_i,y_i)\}_{i\in[m]}$，基学习算法 $\mathcal{L}$，迭代轮数 T

$\mathcal{D}_1(i)\leftarrow 1/m$；// 初始化权重分布

for $t\leftarrow 1$ to T do

  $h_t\leftarrow\mathcal{L}(\mathcal{S},\mathcal{D}_t)$；// 在 $\mathcal{S}$ 上以权重 $\mathcal{D}_t$ 训练 h_t

  $\epsilon_t\leftarrow\mathbb{P}_{(\boldsymbol{x}\sim\mathcal{D}_t,y)}\mathbb{I}(h_t(\boldsymbol{x})\neq y)$；// 计算加权错误率

  if $\epsilon_t>0.5$ then break; // 若基分类器比随机猜测还差则中止算法

  $\alpha_t\leftarrow\frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$；// 计算 h_t 的权重系数

  for $i\leftarrow 1$ to m do

    if $h_t(\boldsymbol{x}_i)=y_i$ then

      $\mathcal{D}_t(i)\leftarrow\mathcal{D}_t(i)\exp(-\alpha_t)$

    else

      $\mathcal{D}_t(i)\leftarrow\mathcal{D}_t(i)\exp(\alpha_t)$

    end

  end

  $s\leftarrow\sum_{i\in[m]}\mathcal{D}_t(i)$;

  for $i\leftarrow 1$ to m do $\mathcal{D}_t(i)\leftarrow\mathcal{D}_t(i)/s$; // 归一化权重

end

输出：$\operatorname{sign}(\sum_{t\in[T]}\alpha_t h_t(\boldsymbol{x}))$

图 2.13 AdaBoost 算法框架

i. 训练过程实现

AdaBoost 的主要步骤如下：

①选取对应的方法，训练出一个弱分类器（对数逻辑回归要求得到一组 theta，决策树桩要求找出分类特征及其阈值）

②计算分类错误率，其中决策树桩的错误率就是上文提到的 weights_error，对数回归的错误率计算函数如下图所示。

```
# 计算弱分类器的误差率
def cal_e (self, y_predict, label):
    return sum([self.weights[i] for i in range(len(label)) if y_predict[i] != label[i]])
```

图 2.14 错误率计算函数

③计算 alpha 值（alpha 值决定了最后线性组合弱分类器结果时，某个结果占的比重），代码如下图所示。

```
# 计算弱分类器的α值
def cal_alpha(self, e):
    return 0.5 * np.log((1 - e) / max(e, 1e-16))
```

图 2.15 计算 alpha 值的函数

④更新数据权重，当前弱分类器预测错误的那些数据，权重要增大，使下一个分类器在训练时更加关注这些数据，同时还要注意权重归一化，要保证所有权重之和为 1。

```
# 更新权重
def update_weights(self, Z, alpha, y_real, y_predict):
    for i in range(len(y_real)):
        #self.weights[i] = (self.weights[i] * np.exp(-1 * alpha * y_real[i] * y_predict[i])) / Z
        self.weights[i] = self.weights[i] * np.exp(-1 * alpha * y_real[i] * y_predict[i])
    self.weights = self.weights / self.weights.sum() # 权重归一化
```

图 2.16 更新数据权重的函数

⑤重复①~④过程，直至训练完所有的基分类器，得到所有的 alpha 值。

ii. 预测过程实现

取出每一组弱分类器与之前得到的 alpha 值，先用弱分类器预测结果，再将该结果乘 alpha 值加到最终的结果当中。需要注意的是，虽然数据集中的类别是 0 和 1，但是在线性组合的过程中却不能按 0 和 1 去与 alpha 相乘，这样会导致预测结果为 0 的那部分失去效果，最后得到的预测结果几乎全都是类别 1。因此，应先将弱分类器预测结果为 0 的那部分，类别定为-1，然后再进入到线性组合预算当中，最后的和若大于 0，则预测为类别 1，若小于 0，则预测为类别 0。

3. 实验环境与平台

实验环境：python3.9
使用工具：anaconda+spyder
电脑系统：Windows11
电脑配置：CPU：11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz
基准速度：2.69GHz
内存：16GB

4. 结果与分析

a) 对数逻辑回归

运行过程中的部分输出如下图所示。

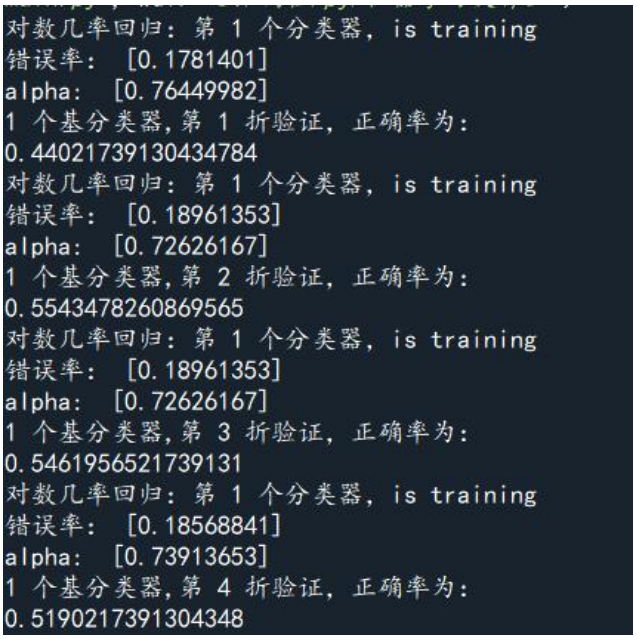


图 4.1 对数逻辑回归运行过程中的输出

最终预测结果的正确率如下图所示。（学习率 0.03，单个分类器训练轮次 9000，已是我能调试出的最佳状态）

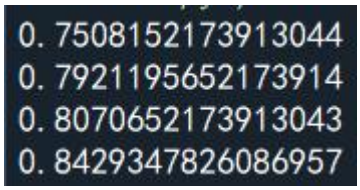


图 4.2 对数逻辑回归预测结果的正确率

b) 决策树桩

运行过程中的部分输出如下图所示。

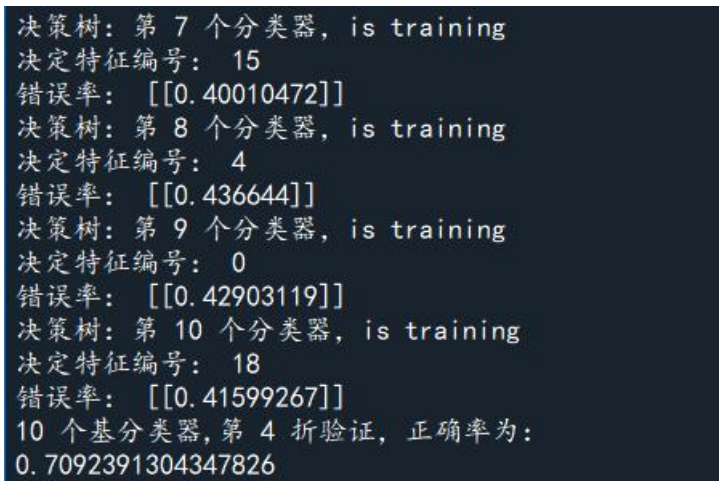


图 4.3 决策树桩运行过程中的部分输出

最终预测结果的正确率如下图所示。

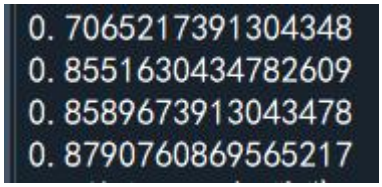


图 4.4 决策树桩预测结果的正确率

c) 分析

从两种方法的预测正确率可以看出，AdaBoost 算法确实可以优化弱分类器准确率不高的问题，通过若干弱分类器的组合得到一个强分类器，且一般情况下，随着弱分类器数目的增多，正确率会逐步提高（当然比如对数回归，在特定学习率和轮次下有可能出现反常）。相比于那些直接实现强分类器的算法，AdaBoost 算法较容易实现，这也启发我们，弱分类器并不一定就毫无用武之地，如果换个视角去看它们，可能就会得到意想不到的结果。

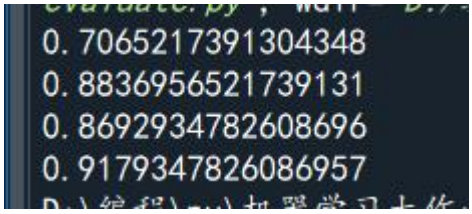
5. 个人体会

本次大作业对我来说是一项颇具挑战性的任务，原因一是我对 python 语言不够熟悉，这点无疑是致命的，虽然不至于每一句话都看不懂，但关于列表、矩阵、元组等数据类型的运算我需要一个个去查，增加了较大的工作量；原因二是我对机器学习的内容也不够熟悉，尽管上完了机器学习的课程，但对知识还是半懂不懂的状态，而通常机器学习相关的学习资料数学公式数量居多，描述语言也不够具体，AdaBoost 算法对我来说又是全新的一项内容……以上种种，在一开始似乎成了一个我不可能完成的任务。

但好在，有老师同学们的帮助，让我不再是一头雾水的状态。然而，理论和实操完全是两回事。在初步完成了代码之后，我首先面临的是大量数据类型不匹配的问题（比如矩阵乘法中两个矩阵的大小不匹配），于是我只能各种定义方式都尝试一遍，最后勉强拼凑出来了能运行的代码。

除此之外，我印象比较深刻的还有两个问题，一是在写对数逻辑回归的训练函数时，更新 w 和 b 的值时没有用到 weights，导致每一个基分类器的性能都相同，从而也导致了不管基分类器数目有多少，预测正确率总是相同；二是在对数逻辑回归中定义 theta 这个变量时，错误地定义在了 train 函数里，由于我的对数回归和 AdaBoost 算法的实现是在两个类中，每训练一个弱分类器都要调用一次对数回归类中的 train 函数，也就导致了每一个弱分类器在开始训练时 theta 都经历了一次初始化，此问题最后表现出来的结果是，随着基分类器数目的增多，正确率反而越来越低，这一问题是我在本次大作业中解决了最久的问题。

有些遗憾的是，我在暂未解决对数逻辑回归的问题时，先把决策树完成了，当时预测的最高正确率可以到达 90%以上，而当我把对数逻辑回归的问题解决后，不确定修改了哪一部分的代码，导致决策树的正确率始终在 87%左右，到最后也没能恢复之前的正确率，这里放一张当时的截图作为证明吧。



总的来说，本次大作业对我来说是一个挑战，同时也是一个提升自我的机会，让我对机器学习的知识有了从认识到理解再到实践的转化，也是让我今后可以不断回顾的一次宝贵的项目经历。

此外，我还想提一个小小的建议，大作业可以提前几节课布置，给同学们留下更多的时间去思考和尝试，希望这门课程将来可以越来越好。

6. 实验问题

a) 你对 AdaBoost 算法有何新的认识？

答：首先，Boost 又被称为增强学习或提升法，是一种重要的集成学习方法，它能够将预测精度仅仅比随机猜测略高的弱学习器增强为预测精度很高的强学习器，这是在直接构造强学习器较为困难的情况下，为学习算法提供了一种有效的新思路和新方法。

其次，Ada 是 Adaptive 的缩写，意为自适应，即被前一个基本分类器误分类的样本权值会增大，而正确分类的样本权值会减小，并再次用来训练下一个基分类器。

通过本次作业可以看出，AdaBoost 算法很好地利用了弱分类器，通过线性组合地方式将其综合为一个强分类器，是以数量换取质量的思想。前文已经提到过，这种思路启发我们，弱分类器并不一定就完全没有用处，或许它可以用来针对一些特定情况，又或许它可以像在 AdaBoost 算法中这样被巧妙的使用。如果我们能开辟思路，不局限于一种算法或一种工具的本身特点，那么说不定还会有更多巧妙的算法被开发出来。

b) 关于基分类器类型、超参数设置对最终模型性能的影响，你有何发现？

答：首先，基分类器类型对 AdaBoost 算法的性能有着很大的影响，从本次作业中可以看出，对数逻辑回归的正确率要明显低于决策树桩的正确率。推测这与模型的复杂度有关，决策树桩简单的思想可能反而成为了它的优点，如果本次作业不要求用决策树桩，而设计更多层决策树的话，效果可能完全是另一种情况。

其次，基分类器数量也对 AdaBoost 算法有一定的影响，本次实验中可以看出，一般情况下，预测正确率会随着基分类器的数目增多而增多，但在代码中如果出现一些逻辑错误，就可能出现意想不到的结果。比如每一个基分类器正确率都很低（即 AdaBoost 算法提供的数据权重没有起到优化分类器的作用）且各不相同，那么线性组合出来的结果，正确率仍然很低，因此不管基分类器数目多少，正确率都差不多且都仅仅略高于随机猜测；比如每一个弱分类器都存在过拟合现象，那么最后的结果在基分类器数目很少的时候正确率可能会很高，但随着基分类器数目的增加，正确率反而会下降。

超参数设置同样是非常关键的一点，学习率过大会导致梯度爆炸或损失函数不收敛，即过拟合，学习率过低会导致收敛速度过慢，从而正确率偏低。若训练轮次过多，即使学习率较低，也可能出现过拟合现象，并且还会导致训练时长增加，训练轮次过少，又可能导致学习数据过少，导致模型效果差。因此若想要达到高正确率，不断调试超参数的取值是不可避免的过程。

因此，不论是基分类器类型、基分类器数量，还是超参数设置，都需要根据具体的数据和具体任务去确定。