

利用AdaBoost元算法提高分类性能

8.1 关于集成学习的相关笔记

8.1.1 个体与集成

集成学习 (ensemble learning) 通过**构建并结合多个学习器**来完成学习任务, 有时也被称为**多分类器系统**、**基于委员会的学习**等.

一 集成学习的基本概念和结构

- **个体学习器** (individual learner): 个体学习器通常由一个现有的学习算法从训练数据产生.
- **基学习器** (base learner): 若集成中只包含**同种类型**的**个体学习器**, 也即是**"同质"**的, 那么同质集成中的个体学习器称为**"基学习器"**. 相应的学习算法称为**"基学习算法"**.
- **组件学习器** (component learner): 若集成中包含**不同类型**的个体学习器, 也即是**"异质"**的, 则此时集成中的个体学习器称为**"组件学习器"**, 或者直接称为**个体学习器**.

图 8.1 显示出**集成学习的一般结构**: 先产生一组**"个体学习器"** (individual learner), 再用**某种策略**将它们**结合起来**.

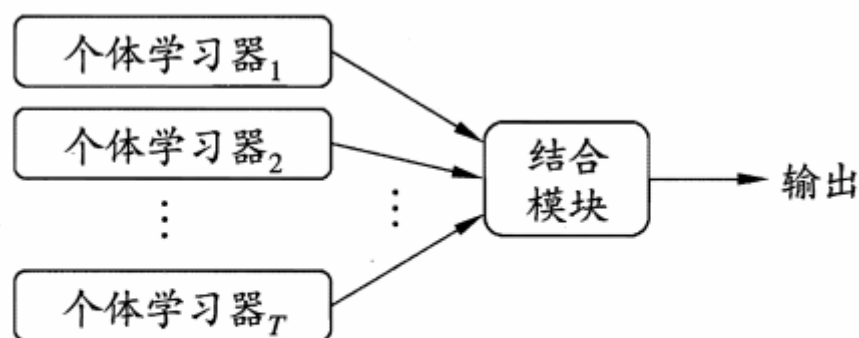


图 8.1 集成学习示意图

二 集成学习的优势及个体学习器的选择原则

1 集成学习的优势

集成学习通过将**多个学习器进行结合**, 常可获得**比单一学习器显著优越的泛化性能**. 特别是对**"弱学习器"** (弱学习器是指泛化性能略优于随机猜测的学习器), 效果更加明显. 因此集成学习的很多理论研究都是针对弱学习器进行的, 而**基学习器有时也被直接称为弱学习器**. 但实际中, 考虑到其他因素, 往往会使用比较强的学习器.

2 个体学习器的选择原则

对于集成学习的个体学习器的选择来说, 要获得好的集成效果, 个体学习器应“**好而不同**”, 具体含义是: 个体学习器要有一定的“**准确性**”, 即学习器不能太坏, 并且要有“**多样性**”, 即学习器间具有差异。

测试例1	测试例2	测试例3	测试例1	测试例2	测试例3	测试例1	测试例2	测试例3			
h_1	✓	✓	✗	h_1	✓	✓	✗	h_1	✓	✗	✗
h_2	✗	✓	✓	h_2	✓	✓	✗	h_2	✗	✓	✗
h_3	✓	✗	✓	h_3	✓	✓	✗	h_3	✗	✗	✓
集成	✓	✓	✓	集成	✓	✓	✗	集成	✗	✗	✗
(a) 集成提升性能			(b) 集成不起作用			(c) 集成起负作用					

图 8.2 集成个体应“好而不同” (h_i 表示第 i 个分类器)

三 集成学习错误率的数学表示

考虑二分类问题 $y \in \{-1, +1\}$ 和真实函数 f , 假定基分类器的错误率为 ϵ , 即对每个基分类器 h_i 有

$$P(h_i(x) \neq f(x)) = \epsilon \quad (8.1)$$

注1: $f(x)$ 为 x 的真实标记, 而 $h_i(x)$ 为基分类器 h_i 对 x 的预测标记, $f(x), h_i(x) \in \{-1, +1\}$.

假设集成通过简单投票法结合 T 个基分类器, 若有超过半数的基分类器正确, 则集成分类就正确:

$$H(x) = \text{sign}\left(\sum_{i=1}^T h_i(x)\right) \quad (8.2)$$

注2: 对于二分类问题, $y \in \{-1, +1\}$, 则预测标记 $h_i(x) \in \{-1, +1\}$, 如果有一半分类正确, 那么 $\sum_{i=1}^T h_i(x) > 0$, 则 $\text{sign}\left(\sum_{i=1}^T h_i(x)\right) = 1$, 即整体分类就正确,

其中, $\text{sign}(x)$ 函数是符号函数, 当 $x > 0$ 时, $\text{sign}(x) = 1$; 当 $x = 0$ 时, $\text{sign}(x) = 0$; 当 $x < 0$ 时, $\text{sign}(x) = -1$

$H(x)$ 为整体分类函数, 即集成分类

假设基分类器的错误率相互独立, 则由 Hoeffding 不等式可知, 集成错误率为

$$\begin{aligned} P(H(x) \neq f(x)) &= \sum_{k=0}^{\lfloor T/2 \rfloor} \binom{T}{k} (1-\epsilon)^k \epsilon^{T-k} \\ &\leq \exp\left(-\frac{1}{2}T(1-2\epsilon)^2\right) \end{aligned} \quad (8.3)$$

注3: 关于集成错误率的推导, 可根据 Hoeffding 不等式和课后习题进行推导, 此处略。

结论:

上式显示出, 随着集成中个体分类器数目 T 的增大, 集成的错误率将呈指数级下降, 最终趋向于零。

注意:

上面问题的分析是基于一个关键假设: **基学习器的误差相互独立**

现实很难满足, 实际上, 个体学习器的"**准确性**"和"**多样性**"本身互为冲突, 此消彼长. 一般情况下, 准确性很高之后, 要增加多样性就需牺牲准确性.

四 集成学习的分类

- 个体学习器间**存在强依赖关系**、**必须串行生成的序列化方法**, 代表是 **Boosting**
- 个体学习器间**不存在强依赖关系**、**可同时生成的并行化方法**, 代表是 **Bagging** 和"**随机森林**" (Random Forest)

8.1.2 Boosting

一 Boosting 的基本概念

Boosting 是一族可将**弱学习器提升为强学习器的算法**. 这族算法的**工作机制**类似于如下:

1. 先从初始训练集训练出一个基学习器
2. 再根据基学习器的表现对训练样本分布进行调整, 使得先前基学习器做错的训练样本在后续受到更多关注
3. 然后基于调整后的样本分布来训练下一个基学习器
4. 如此重复进行, 直至基学习器数目达到事先指定的值 T , 最终将这 T 个基学习器进行加权结合.

二 AdaBoost 算法的数学推导(加性模型)

Boosting 族算法最著名的代表是 AdaBoost, 其描述如图 8.3 所示, 其中, $y_i \in \{-1, +1\}$, f 是真实函数.

输入: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
基学习算法 \mathcal{L} ;
训练轮数 T .

过程:

```
1:  $\mathcal{D}_1(x) = 1/m$ .  
2: for  $t = 1, 2, \dots, T$  do  
3:    $h_t = \mathcal{L}(D, \mathcal{D}_t)$ ;  
4:    $\epsilon_t = P_{x \sim \mathcal{D}_t}(h_t(x) \neq f(x))$ ;  
5:   if  $\epsilon_t > 0.5$  then break  
6:    $\alpha_t = \frac{1}{2} \ln \left( \frac{1-\epsilon_t}{\epsilon_t} \right)$ ;  
7:    $\mathcal{D}_{t+1}(x) = \frac{\mathcal{D}_t(x)}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(x) = f(x) \\ \exp(\alpha_t), & \text{if } h_t(x) \neq f(x) \end{cases}$   
       $= \frac{\mathcal{D}_t(x) \exp(-\alpha_t f(x) h_t(x))}{Z_t}$   
8: end for  
输出:  $H(x) = \text{sign} \left( \sum_{t=1}^T \alpha_t h_t(x) \right)$ 
```

图 8.3 AdaBoost 算法

$y_i \in \{-1, +1\}$, f 是真实函数. 基于"加性模型"对 AdaBoost 进行推导.

基学习器的线性组合

$$H(x) = \sum_{t=1}^T \alpha_t h_t(x) \quad (8.4)$$

注4: $h_t(x) \in \{-1, +1\}$, 加权系数由后面的 (8.11) 给出, 且这 T 个系数 α_t 仅与自身对应的基学习器错误率 ϵ_t 有关, 相互之间无必然联系. $\alpha_t > 0$. 最后的线性加权结果 $H(x)$ 是一个实数. 取值无法确定.

来最小化指数损失函数

$$\ell_{\text{exp}}(H|\mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}} \left[e^{-f(x)H(x)} \right] \quad (8.5)$$

注5: 式 (8.5) 的理解

1. $f(x)H(x)$, 两者符号一致时为正, 不一致时为负.
2. 符号 $\mathbb{E}_{x \sim \mathcal{D}}[\cdot]$ 的含义:

\mathcal{D} 为概率分布, 可简单理解为在数据集 D 中进行一次随机抽样, 每个样本被取到的概率.

$\mathbb{E}[\cdot]$ 为期望. 综合起来 $\mathbb{E}_{x \sim \mathcal{D}}[\cdot]$ 表示在概率分布 \mathcal{D} 上的期望.

也可简单理解为对数据集 D 以概率 \mathcal{D} 进行加权后的期望.

II 代表的是指示函数 (indicator function)

它的含义是: 当输入为 $True$ 的时候, 输出为1, 输入为 $False$ 的时候, 输出为0.

例如: $\mathbb{I}(f(\mathbf{x}) = 1)$ 表示, 当 $f(\mathbf{x})$ 等于 1 时输出为 1, 否则输出为 0.

易知, 上式最后一行成立.

若 $H(\mathbf{x})$ 能令**指数损失函数最小化**, 则式 (8.5) 对 $H(\mathbf{x})$ 求偏导, 有:

$$\frac{\partial \ell_{\text{exp}}(H|\mathcal{D})}{\partial H(\mathbf{x})} = -e^{-H(\mathbf{x})} P(f(\mathbf{x}) = 1|\mathbf{x}) + e^{H(\mathbf{x})} P(f(\mathbf{x}) = -1|\mathbf{x}) \quad (8.6)$$

注6: 式 (8.6) 推导(不知道这样理解是否合理, 暂且这样推导)

因为 $f(x)$ 是真实函数, 因此可以理解为 $e^{-f(x)H(x)}$ 在真实函数 $f(x)$ 的分布下的 $\mathbb{E}_{\mathbf{x} \sim \mathcal{D}}[\cdot]$.

那么, 指数损失函数可以写为:

$$\begin{aligned} \ell_{\text{exp}}(H|\mathcal{D}) &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H(\mathbf{x})} \right] \\ &= P(f(\mathbf{x}) = 1|\mathbf{x}) * e^{-H(\mathbf{x})} + P(f(\mathbf{x}) = -1|\mathbf{x}) * e^{H(\mathbf{x})} \end{aligned}$$

其中 $P(f(\mathbf{x}) = 1|\mathbf{x})$ 和 $P(f(\mathbf{x}) = -1|\mathbf{x})$ 为常数, 对 $H(\mathbf{x})$ 求偏导, 就有

$$\frac{\partial \ell_{\text{exp}}(H|\mathcal{D})}{\partial H(\mathbf{x})} = -e^{-H(\mathbf{x})} P(f(\mathbf{x}) = 1|\mathbf{x}) + e^{H(\mathbf{x})} P(f(\mathbf{x}) = -1|\mathbf{x})$$

也即是 (8.6) 式.

令式 (8.6) 为零易得:

$$H(\mathbf{x}) = \frac{1}{2} \ln \frac{P(f(\mathbf{x}) = 1|\mathbf{x})}{P(f(\mathbf{x}) = -1|\mathbf{x})} \quad (8.7)$$

因此, 有:

$$\begin{aligned} \text{sign}(H(\mathbf{x})) &= \text{sign}\left(\frac{1}{2} \ln \frac{P(f(\mathbf{x}) = 1|\mathbf{x})}{P(f(\mathbf{x}) = -1|\mathbf{x})}\right) \\ &= \begin{cases} 1, & P(f(\mathbf{x}) = 1|\mathbf{x}) > P(f(\mathbf{x}) = -1|\mathbf{x}) \\ -1, & P(f(\mathbf{x}) = 1|\mathbf{x}) < P(f(\mathbf{x}) = -1|\mathbf{x}) \end{cases} \\ &= \arg \max_{y \in \{-1, 1\}} P(f(\mathbf{x}) = y|\mathbf{x}) \end{aligned} \quad (8.8)$$

注7: 关于最后的式 (8.8) 理解

因为, $y=1$ 或者 -1 的, 选择使得 $P(f(\mathbf{x}) = y|\mathbf{x})$ 取得最大值时的 y 值. 如果 $P(f(\mathbf{x}) = 1|\mathbf{x}) > P(f(\mathbf{x}) = -1|\mathbf{x})$, 那么取 1, 也就是 $y = 1$, 如果反过来, 则取 -1 .

这就意味着, $\text{sign}(H(\mathbf{x}))$ 达到了**贝叶斯最优错误率**. 也即是说, 若**指数损失函数最小化**, 则**分类错误率也将最小化**.

同时说明, **指数损失函数**是分类任务原本 0/1 损失函数**一致的** (consistent) **替代损失函数**

第一个基分类器 h_1 是通过直接将基学习算法用于初始数据分布而得; 此后迭代地生成 h_t 和 α_t , 当基分类器 h_t 基于分布 \mathcal{D}_t 产生后, 该基分类器的权重 α_t 应使得 $\alpha_t h_t$ 最小化如下指数损失函数:

$$\begin{aligned}\ell_{\text{exp}}(\alpha_t h_t | \mathcal{D}_t) &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_t} \left[e^{-f(\mathbf{x}) \alpha_t h_t(\mathbf{x})} \right] \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_t} \left[e^{-\alpha_t} \mathbb{I}(f(\mathbf{x}) = h_t(\mathbf{x})) + e^{\alpha_t} \mathbb{I}(f(\mathbf{x}) \neq h_t(\mathbf{x})) \right] \\ &= e^{-\alpha_t} P_{\mathbf{x} \sim \mathcal{D}_t}(f(\mathbf{x}) = h_t(\mathbf{x})) + e^{\alpha_t} P_{\mathbf{x} \sim \mathcal{D}_t}(f(\mathbf{x}) \neq h_t(\mathbf{x})) \\ &= e^{-\alpha_t} (1 - \epsilon_t) + e^{\alpha_t} \epsilon_t\end{aligned}\quad (8.9)$$

注8: 关于 (8.9) 的推导:

1. 第二个等式, 注意下相等时乘积为 1, 不等时乘积为 -1 就可
2. 类似于式 (8.6) 的推导.
3. 理解 $P_{\mathbf{x} \sim \mathcal{D}_t}(f(\mathbf{x}) \neq h_t(\mathbf{x}))$ 就是错误率 ϵ_t 即可

其中 $\epsilon_t = P_{\mathbf{x} \sim \mathcal{D}_t}(f(\mathbf{x}) \neq h_t(\mathbf{x}))$. 对上式的指数损失函数的导数, 有:

$$\frac{\partial \ell_{\text{exp}}(\alpha_t h_t | \mathcal{D}_t)}{\partial \alpha_t} = -e^{-\alpha_t} (1 - \epsilon_t) + e^{\alpha_t} \epsilon_t \quad (8.10)$$

令式 (8.10) 为零可求得:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \quad (8.11)$$

式 (8.11) 就是图 8.3 中算法第 6 行的分类器权重更新公式.

AdaBoost 算法在获得 H_{t-1} 之后样本分布将进行调整, 使下一轮的基学习器 h_t 能纠正 H_{t-1} 的一些错误. 理想的 h_t 能纠正 H_{t-1} 全部错误, 也即最小化:

$$\begin{aligned}\ell_{\text{exp}}(H_{t-1} + h_t | \mathcal{D}) &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})(H_{t-1}(\mathbf{x}) + h_t(\mathbf{x}))} \right] \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x}) H_{t-1}(\mathbf{x})} e^{-f(\mathbf{x}) h_t(\mathbf{x})} \right]\end{aligned}\quad (8.12)$$

注9:

1. 为什么纠正后的 $H_t = H_{t-1} + h_t$, 而不是 $H_t = H_{t-1} + h_t \alpha_t$ (本文的理解过程来源于csdn博客"彬彬有礼的专栏", 博客地址:<https://blog.csdn.net/jbb0523>)

答:

【注】本式确实有问题, 虽然并不影响后面式(8.18)和式(8.19)的结果。AdaBoost 第 t 轮迭代应该求解如下优化问题从而得到 α_t 和 $h_t(\mathbf{x})$:

$$(\alpha_t, h_t(\mathbf{x})) = \arg \min_{\alpha, h} \ell_{\exp}(H_{t-1} + \alpha h \mid \mathcal{D})$$

对于该问题, 先对于固定的任意 $\alpha > 0$, 求解 $h_t(\mathbf{x})$; 得到 $h_t(\mathbf{x})$ 后再求 α_t 。

原始文献[Friedman J H, Hastie T, Tibshirani R, et al. [Additive logistic regression : A statistical view of boosting](#)[J]. Annals of Statistics, 2000, 28(2): 337-407.]第 10 页(第 346 页)对西瓜书中式(8.12)到式(8.12)的相关推导如下:

RESULT 1. *The Discrete AdaBoost algorithm (population version) builds an additive logistic regression model via Newton-like updates for minimizing $E(e^{-yF(x)})$.*

PROOF. Let $J(F) = E[e^{-yF(x)}]$. Suppose we have a current estimate $F(x)$ and seek an improved estimate $F(x) + cf(x)$. For fixed c (and x), we expand $J(F(x) + cf(x))$ to second order about $f(x) = 0$,

$$\begin{aligned} J(F + cf) &= E[e^{-y(F(x)+cf(x))}] \\ &\approx E[e^{-yF(x)}(1 - ycf(x) + c^2 y^2 f(x)^2/2)] \\ &= E[e^{-yF(x)}(1 - ycf(x) + c^2/2)], \end{aligned}$$

since $y^2 = 1$ and $f(x)^2 = 1$. Minimizing pointwise with respect to $f(x) \in \{-1, 1\}$, we write

$$(16) \quad f(x) = \arg \min_f E_w(1 - ycf(x) + c^2/2|x).$$

可以发现原文献中保留了参数 c (即 α)。当然, 对于任意 $\alpha > 0$, 并不影响推导结果。

且不管以上问题, 权且按作者思路推导一下:

本式较为简单, 只要将 $H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + h_t(\mathbf{x})$ 替换式(8.5)中的 $H(\mathbf{x})$ 即可。

2. 带入到式 (8.5) 即可得到 (8.12)。

同时, $f(\mathbf{x}), h_t(\mathbf{x}) \in \{-1, +1\}$, 所以, $f^2(\mathbf{x}) = h_t^2(\mathbf{x}) = 1$, 式 (8.12) 可使用 $e^{-f(\mathbf{x})h_t(\mathbf{x})}$ 的泰勒展开式近似为:

$$\begin{aligned} \ell_{\exp}(H_{t-1} + h_t \mid \mathcal{D}) &\simeq \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \left(1 - f(\mathbf{x})h_t(\mathbf{x}) + \frac{f^2(\mathbf{x})h_t^2(\mathbf{x})}{2} \right) \right] \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \left(1 - f(\mathbf{x})h_t(\mathbf{x}) + \frac{1}{2} \right) \right] \end{aligned} \quad (8.13)$$

注10: 关于泰勒展开式相关知识回顾。

泰勒公式是将一个在 $x = x_0$ 处具有 n 阶导数的函数 $f(x)$ 利用关于 $(x - x_0)$ 的 n 次多项式来逼近函数的方法。

若函数 $f(x)$ 在包含 x_0 的某个闭区间 $[a, b]$ 上具有 n 阶导数, 且在开区间 (a, b) 上具有 $(n + 1)$ 阶导数, 则对闭间 $[a, b]$ 上任意一点 x , 成立下式:

$$f(x) = \frac{f(x_0)}{0!} + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + R_n(x)$$

其中, $f^{(n)}(x)$ 表示 $f(x)$ 的 n 阶导数, 等号后的多项式称为函数 $f(x)$ 在 x_0 处的泰展开式, 剩余的 $R_n(x)$ 是泰勒公式的余项, 是 $(x - x_0)^n$ 的高阶无穷小。

常用函数的泰勒展开式:

$$e^x = 1 + x + \frac{1}{2!}x^2 + \cdots + \frac{1}{n!}x^n + o(x^n)$$

则易得式 (8.13) 结果

于是, 理想的基学习器: (表示方法不是很准确? 暂且按照书上的思路来推导)

$$\begin{aligned} h_t(\mathbf{x}) &= \arg \min_h \ell_{\text{exp}}(H_{t-1} + h | \mathcal{D}) \\ &= \arg \min_h \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \left(1 - f(\mathbf{x})h(\mathbf{x}) + \frac{1}{2} \right) \right] \\ &= \arg \max_h \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} f(\mathbf{x})h(\mathbf{x}) \right] \\ &= \arg \max_h \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[\frac{e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]} f(\mathbf{x})h(\mathbf{x}) \right] \end{aligned} \quad (8.14)$$

注11:

1. 关于 $H_{t-1} + h$ 的理解:(个人理解, 可能不准确)

首先我们知道 $H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + h_t(\mathbf{x})$, 也就是在 H_{t-1} 的基础上加上另一个基学习器进行调整, 得到 H_t , 调整的原则也就是指数损失函数最小化. 此时得到 $h_t(\mathbf{x})$. 所以, 上式第一个等式成立.

2. 第二个等式也就是根据式 (8.13) 而来

3. 第三个等式因为负号变为 \max , 同时常数项略去.

4. 第四个等式成立时因为 $\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]$ 是与自变量 $h(\mathbf{x})$ 无关的正的常数. 为何是正的常数? 因为 $e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} > 0$, 所以 $\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [\cdot] > 0$. 而最大化问题乘以某个正的常数

令 \mathcal{D}_t 表示一个分布, 具体为:

$$\mathcal{D}_t(\mathbf{x}) = \frac{\mathcal{D}(\mathbf{x})e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]} \quad (8.15)$$

注12: $\mathcal{D}(\mathbf{x})$ 从何而来?(个人理解, 非准确答案, 暂且理解如下)

首先 $\frac{e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]}$ 是一个关于 $e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}$ 的函数. 更具体的讲, 应该是关于 \mathbf{x} 的函数. 同时,

令 \mathcal{D}_t 表示一个分布, 所以, 加一个 $\mathcal{D}(\mathbf{x})$ 表示一个"基础分布".

根据数学期望的定义, 这等价于

$$\begin{aligned} h_t(\mathbf{x}) &= \arg \max_h \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[\frac{e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]} f(\mathbf{x})h(\mathbf{x}) \right] \\ &= \arg \max_h \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_t} [f(\mathbf{x})h(\mathbf{x})] \end{aligned} \quad (8.16)$$

同时, $f(\mathbf{x}), h_t(\mathbf{x}) \in \{-1, +1\}$, 所以有:

$$f(\boldsymbol{x})h(\boldsymbol{x}) = 1 - 2\mathbb{I}(f(\boldsymbol{x}) \neq h(\boldsymbol{x})) \quad (8.17)$$

则理想的基学习器

$$h_t(\boldsymbol{x}) = \arg \min_h \mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}_t} [\mathbb{I}(f(\boldsymbol{x}) \neq h(\boldsymbol{x}))] \quad (8.18)$$

因此, 理想的 h_t 将在分布 \mathcal{D}_t 下最小化分类误差.

考虑到 \mathcal{D}_t 和 \mathcal{D}_{t+1} 的关系, 根据 (8.15) 有:

$$\begin{aligned} \mathcal{D}_{t+1}(\boldsymbol{x}) &= \frac{\mathcal{D}(\boldsymbol{x})e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} [e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} \\ &= \frac{\mathcal{D}(\boldsymbol{x})e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})} e^{-f(\boldsymbol{x})\alpha_t h_t(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} [e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} \\ &= \mathcal{D}_t(\boldsymbol{x}) \cdot e^{-f(\boldsymbol{x})\alpha_t h_t(\boldsymbol{x})} \frac{\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} [e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}]}{\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} [e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} \end{aligned} \quad (8.19)$$

注13: (8.19) 的推导

1. 第一个等式根据式 (8.15) 直接得到
2. 第二个等式由 $H_t(\boldsymbol{x}) = H_{t-1}(\boldsymbol{x}) + \alpha_t h_t(\boldsymbol{x})$ 而来, (这里可以注意到与前面注9的矛盾之处, 暂且记录在此)
3. 为了得到 $\mathcal{D}_t(\boldsymbol{x})$, 把式子变换而来.

这也就是图 8.3 中算法第 7 行的样本分布更新公式.

于是, 由式 (8.11) 和 (8.19) 可见, 基于加性模型迭代式优化指数损失函数的角度推导出了图 8.3 的 AdaBoost 算法.

三 对数据分布进行学习的方法

Boosting 算法要求基学习器能对特定的数据分布进行学习

1. 重赋权法(re-weighting)

在训练过程的每一轮中, 根据样本分布为每个训练样本重新赋予一个权重.

2. 重采样法(re-sampling)

对无法接受带权样本的基学习算法, 则可通过"重采样法" (re-sampling)来处理, 即在每一轮学习中, 根据样本分布对训练集重新进行采样, 再用重采样而得的样本集对基学习器进行训练.

一般而言, 这两种做法没有显著的优劣差别.

需注意的是, Boosting 算法在训练的**每一轮都要检查**当前生成的**基学习器是否满足基本条件**, 一旦条件**不满足**, 则当前基学习器即被**抛弃**, 且学习过程**停止**.

在此种情形下, 初始设置的学习轮数 T 也许遥远未达到, 可能导致最终集成中只**包含很少的基学习器而性能不佳**.

若采用"重采样法", 则可获得"重启动"机会以避免训练过程过早停止, 即在抛弃**不满足条件**的当前基学习器之后, 可根据当前分布**重新对训练样本进行采样**, 再基于新的采样结果**重新训练出基学习器**, 从而使得学习过程可以**持续到预设的 T 轮完成**.

8.1.3 Bagging 与随机森林

欲得到泛化性能强的集成, 集成中的个体学习器应尽可能相互独立; 虽然"独立"在现实任务中无法做到, 但可以设法使基学习器尽可能具有较大的差异.

给定一个训练数据集, 一种可能的做法是对训练样本进行采样, 产生出若干个不同的子集, 再从每个数据子集中训练出一个基学习器. 这样, 由于训练数据不同, 我们获得的基学习器可望具有比较大的差异. 然而, 为获得好的集成, 我们同时还希望个体学习器不能太差.

但是, 如果采样出的每个子集完全不同, 则每个基学习器只用到了了一小部分训练数据, 甚至不足以进行有效学习, 这显然无法确保产生出比较好的基学习器. 此时, 可以考虑使用相互交叠的采样子集.

— Bagging

Bagging 是并行式集成学习方法最著名的代表. 它是直接基于自助采样法(具体请看第二章关于自助采样法的相关知识).

我们可以采样出 T 个含 m 个训练样本的采样集, 然后基于每个采样训练集训练出一个基学习器, 再将这些基学习器进行结合. 这就是 Bagging 的基本流程.

在对预测输出进行结合时, Bagging 通常对分类任务使用简单投票法, 对回归任务使用简单平均法.

Bagging 的算法描述如图 8.5 所示.

输入: 训练集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$;
基学习算法 \mathcal{L} ;
训练轮数 T .

过程:

- 1: for $t = 1, 2, \dots, T$ do
- 2: $h_t = \mathcal{L}(D, \mathcal{D}_{bs})$
- 3: end for

输出: $H(\mathbf{x}) = \arg \max_{y \in \mathcal{Y}} \sum_{t=1}^T \mathbb{I}(h_t(\mathbf{x}) = y)$

图 8.5 Bagging 算法

训练一个 Bagging 集成与直接使用基学习算法训练一个学习器的复杂度同阶. 这说明 Bagging 是一个很高效的集成学习算法.

另外, 与标准 AdaBoost 只适用于二分类任务不同, Bagging 能不经修改地用于多分类、回归等任务.

同时, 自助采样有 36.8% 的样本可用作验证集来对泛化性能进行"包外估计".

令 D_t 表示 h_t 实际使用的训练样本集, 令 $H^{oop}(\mathbf{x})$ 表示对样本 \mathbf{x} 的包外预测, 即仅考虑那些未使用 \mathbf{x} 训练的基学习器在 \mathbf{x} 上的预测, 有

$$H^{oob}(\mathbf{x}) = \arg \max_{y \in \mathcal{Y}} \sum_{t=1}^T \mathbb{I}(h_t(\mathbf{x}) = y) \cdot \mathbb{I}(\mathbf{x} \notin D_t) \quad (8.20)$$

注14: 也就是除去 D_t 训练样本外的其他样本, $h_t(\mathbf{x})$ 尽可能多的和对应的 y 相等情况下, 利用简单投票法得到最后的 y

那么 Bagging 泛化误差的包外估计为:

$$\epsilon^{oob} = \frac{1}{|D|} \sum_{(\mathbf{x}, y) \in D} \mathbb{I}(H^{oob}(\mathbf{x}) \neq y) \quad (8.21)$$

包外样本还有许多其他用途, 如:

1. 当基学习器是决策树时, 可使用包外样本来辅助剪枝, 或用于估计决策树中各结点的后验概率以辅助对零训练样本结点的处理.
2. 当基学习器是神经网络时, 可使用包外样本来辅助早期停止以减小过拟合风险.

二 随机森林

1 随机森林的基本概念和含义

随机森林 (Random Forest, 简称 RF) 是 **Bagging** 的一个扩展变体. RF 在以决策树为基学习器构建 Bagging 集成的基础上, 进一步在决策树的训练过程中引入了**随机属性选择**.

具体来说, 传统决策树在选择划分属性时是在**当前结点的属性集合** (假定有 d 个属性) 中**选择一个最优属性**; 而在 RF 中, 对基决策树的每个结点, 先从**该结点的属性集合中随机选择一个包含 k 个属性的子集**, 然后再从这个**子集中选择一个最优属性**用于划分.

这里的参数 k 控制了随机性的引入程度; 若令 $k = d$, 则基决策树的构建与传统决策树相同; 若令 $k = 1$, 则是随机选择一个属性用于划分; 一般情况下, 推荐值 $k = \log_2 d$

2 随机森林的扰动

随机森林简单、容易实现、计算开销小, 但它在很多现实任务中展现出**强大的性能**, 被誉为"**代表集成学习技术水平的方法**"

Bagging 中基学习器的"**多样性**"仅通过**样本扰动** (通过对初始训练集采样) 而来, **随机森林中基学习器的多样性**不仅来自**样本扰动**, 还来自**属性扰动**, 这就使得最终集成的**泛化性能**可通过个体学习器之间**差异度的增加**而进一步提升.

3 随机森林的收敛性

随机森林的收敛性与 Bagging 相似.

如图 8.7 所示, 随机森林的**起始性能**往往相对**较差**, 特别是在集成中**只包含一个基学习器**时 (因为通过引入属性扰动, 随机森林中个体学习器的性能往往有所降低). 然而, 随着**个体学习器数目的增加**, 随机森林通常会收敛到**更低的泛化误差**.

同时, **随机森林的训练效率常优于 Bagging**. 因为在个体决策树的构建过程中, Bagging 使用的是"**确定型**"决策树, 在**选择划分属性**时要对结点的**所有属性进行考察**, 而随机森林使用的"**随机型**"决策树则只需考察一个属性子集.

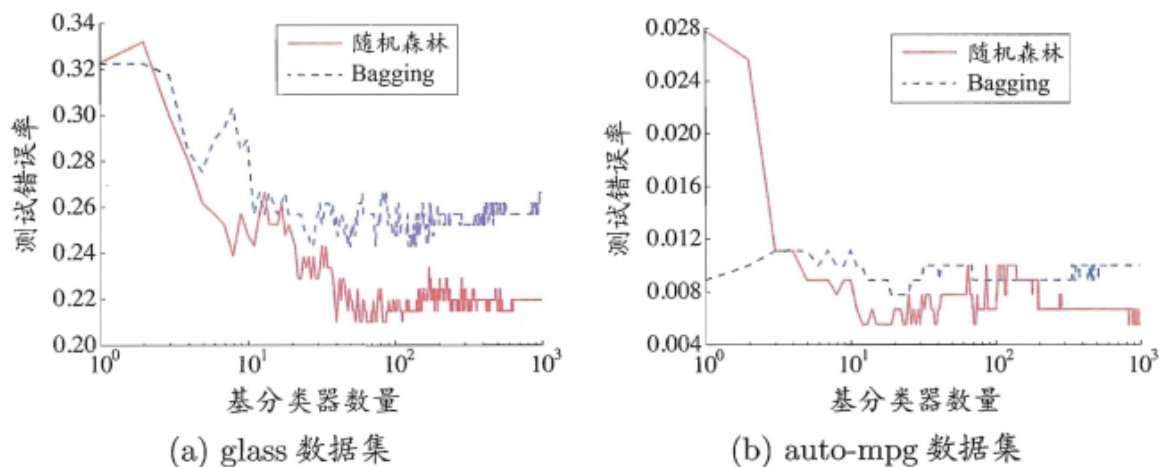


图 8.7 在两个 UCI 数据上, 集成规模对随机森林与 Bagging 的影响

8.1.4 结合策略

学习器结合可能会从三个方面带来好处:

- 首先, 从统计的方面来看, 由于学习任务的假设空间往往很大, 可能有多个假设在训练集上达到同等性能, 此时若使用单学习器可能因误选而导致泛化性能不佳, 结合多个学习器则会减小这一风险。
- 第二, 从计算的方面来看, 学习算法往往会陷入局部极小, 有的局部极小点所对应的泛化性能可能很糟糕, 而通过多次运行之后进行结合, 可降低陷入糟糕局部极小点的风险。
- 第三, 从表示的方面来看, 某些学习任务的真实假设可能不在当前学习算法所考虑的假设空间中, 此时若使用单学习器则肯定无效, 而通过结合多个学习器, 由于相应的假设空间有所扩大, 有可能学得更好的近似。

直观的示意图, 如图 8.8:

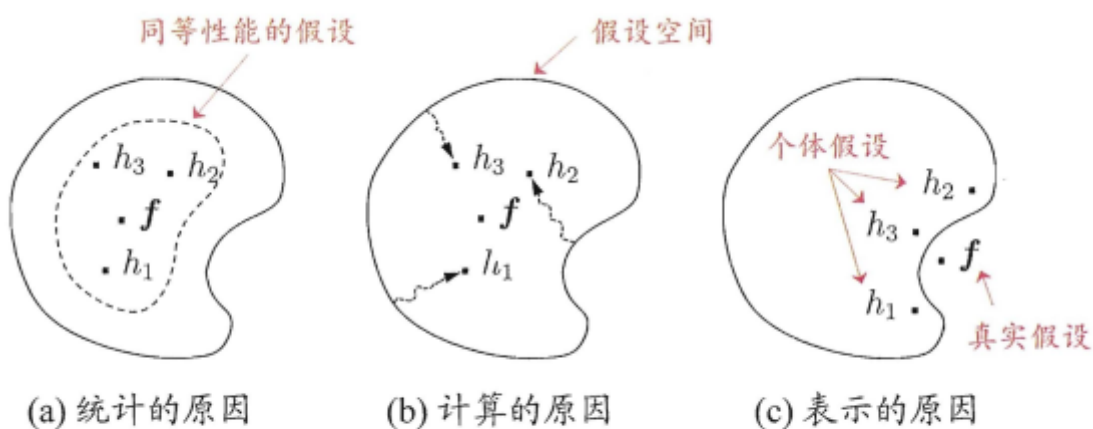


图 8.8 学习器结合可能从三个方面带来好处 [Dietterich, 2000]

假定集成包含 T 个基学习器 $\{h_1, h_2, \dots, h_T\}$, 其中 h_i 在示例 \mathbf{x} 上的输出为 $h_i(\mathbf{x})$.

一 平均法

对数值型输出 $h_i(\mathbf{x}) \in \mathbb{R}$, 最常见的结合策略是使用平均法 (averaging).

- 简单平均法 (simple averaging)

$$H(\mathbf{x}) = \frac{1}{T} \sum_{i=1}^T h_i(\mathbf{x}) \quad (8.22)$$

- 加权平均法 (weighted averaging)

$$H(\mathbf{x}) = \sum_{i=1}^T w_i h_i(\mathbf{x}) \quad (8.23)$$

其中, w_i 是个体学习器 h_i 的权重, 通常要求 $w_i \geq 0, \sum_{i=1}^T w_i = 1$.

加权平均法和简单平均法之间的关系:

简单平均法是加权平均法令 $w_i = 1/T$ 的特例.

事实上, 加权平均法可认为是集成学习研究的基本出发点.

加权平均法的权重一般是从训练数据中学习而得, 现实任务中的训练样本通常不充分或存在噪声, 这将使得学出的权重不完全可靠. 当集成规模较大时, 学习的权重比较多, 容易导致过拟合.

加权平均法未必一起优于简单平均法,

选择的原则:

一般而言, 在个体学习器性能相差较大时宜使用加权平均法, 而在个体学习器性能相近时宜使用简单平均法.

二 投票法

对分类任务来说, 学习器 h_i 将从类别标记集合 $\{c_1, c_2, \dots, c_N\}$ 中预测出一个标记, 最常见的结合策略是使用投票法 (voting).

将 h_i 在样本 \mathbf{x} 上的预测输出表示为一个 N 维向量 $(h_i^1(\mathbf{x}); h_i^2(\mathbf{x}); \dots; h_i^N(\mathbf{x}))$, 其中 $h_i^j(\mathbf{x})$ 是 h_i 在类别标记 c_j 上的输出.

- 绝对多数投票法 (majority voting)

$$H(\mathbf{x}) = \begin{cases} c_j, & \text{if } \sum_{i=1}^T h_i^j(\mathbf{x}) > 0.5 \sum_{k=1}^N \sum_{i=1}^T h_i^k(\mathbf{x}) \\ \text{reject}, & \text{otherwise.} \end{cases} \quad (8.24)$$

即若某标记得票过半数, 则预测为该标记; 否则拒绝预测.

- 相对多数投票法 (plurality voting)

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T h_i^j(\mathbf{x})} \quad (8.25)$$

即预测为的票数最多的标记, 若同时有多个标记获得最高票, 则从中随机选取一个.

- 加权投票法 (weighted voting)

$$H(\mathbf{x}) = c_{\arg \max} \sum_{i=1}^T w_i h_i^j(\mathbf{x}) \quad (8.26)$$

与加权平均法类似, w_i 是 h_i 的权重, 通常要求 $w_i \geq 0$, $\sum_{i=1}^T w_i = 1$.

三 学习法

当训练数据很多时, 一种更为强大的结合策略是使用"学习法", 即通过另一个学习器来进行结合. Stacking 是学习法的典型代表.

在这个新数据集中, 初级学习器的输出被当作样例输入特征, 而初始样本的标记仍被当作样例标记. Stacking 的算法描述如图 8.9 所示. 同时, 初级集成是异质的.

输入: 训练集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$;
 初级学习算法 $\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_T$;
 次级学习算法 \mathcal{L} .

过程:

```

1: for  $t = 1, 2, \dots, T$  do
2:    $h_t = \mathcal{L}_t(D)$ ;
3: end for
4:  $D' = \emptyset$ ;
5: for  $i = 1, 2, \dots, m$  do
6:   for  $t = 1, 2, \dots, T$  do
7:      $z_{it} = h_t(\mathbf{x}_i)$ ;
8:   end for
9:    $D' = D' \cup ((z_{i1}, z_{i2}, \dots, z_{iT}), y_i)$ ;
10: end for
11:  $h' = \mathcal{L}(D')$ ;
输出:  $H(\mathbf{x}) = h'(h_1(\mathbf{x}), h_2(\mathbf{x}), \dots, h_T(\mathbf{x}))$ 

```

图 8.9 Stacking 算法

8.2 机器学习实战中的集成学习

8.2.1 集成学习的基本概念

集成方法 (ensemble method) 或者元算法 (meta-algorithm):

可以将不同的分类器组合起来, 而这种组合结果则被称为集成方法 (ensemble method) 或者元算法 (meta-algorithm)

集成方法的形式:

- 1 不同算法的集成
- 2 同一算法在不同设置下的集成
- 3 数据集不同部分分配给不同分类器之后的集成

其中, 下面的bagging和boosting集成方法都是基于同一种分类器多个不同实例的方法.

一 bagging: 基于数据随机重抽样的分类器构建方法

bagging方法的概念:

自举汇聚法 (bootstrap aggregating), 也称为bagging方法, 是在**从原始数据集选择 S 次后得到 S 个新数据集的一种技术. 新数据集和原数据集的大小相等**. 每个数据集都是通过在原始数据集中随机选择一个样本来进行替换而得到的. 这里的替换就意味着可以多次地选择同一样本. 这一性质就允许新数据集中可以有重复的值, 而原始数据集的某些值在新集合中则不再出现.

注: 这里的意思是从原始集合中随机选择一个样本, 然后随机选择一个样本来代替这个样本. 在其他书中, bagging中的数据集中通常被认为是放回取样得到的, 比如要得到一个大小为n的新数据集, 该数据集中的每个样本都是在原始数据集中随机抽样 (即抽样之后又放回) 得到的 (西瓜书中就是如此)

bagging方法的具体过程:

在S个数据集建好之后, 将某个学习算法分别作用于每个数据集就得到了S个分类器. 当我们要对新数据进行分类时, 就可以应用这S个分类器进行分类. 与此同时, **选择分类器投票结果中最多的类别作为最后的分类结果**.

其中, 随机森林就是bagging方法的优秀代表.

二 boosting

boosting是一种与bagging很类似的技术.

相同点:

不论是在boosting还是bagging当中, 所使用的**多个分类器的类型都是一致的**.

不同点:

- 从**分类器相互关系**上来说:
 - boosting中, 不同的分类器是通过串行训练而获得的, 每个新分类器都根据已训练出的分类器的性能来进行训练; boosting是通过集中关注被已有分类器错分的
 - 而bagging中, 各个分类器相互独立, 没有多大的联系.
- 从**分类结果的处理**上来说:
 - boosting分类的结果是基于所有**分类器的加权求和结果**的. 在boosting中的**分类器权重并不相等**, 每个权重代表的是其对应分类器在上一轮迭代中的成功度
 - 而在bagging中, 各个分类器权重是相等的

而在boosting方法中, 最流行的版本是**AdaBoost**, 后面的实战例子都是基于AdaBoost的.

8.2.2 AdaBoost的训练过程: 基于错误提升分类器的性能

AdaBoost是adaptive boosting (自适应boosting) 的缩写, 其运行过程如下:

- 训练数据中的每个样本, 并赋予其一个权重, 这些权重构成了向量D. 一开始, 这些权重都初始化成相等值.
- 首先在训练数据上训练出一个弱分类器并计算该分类器的错误率, 然后在同一数据集上再次训练弱分类器.
- 在分类器的第二次训练当中, 将会重新调整每个样本的权重, 其中**第一次分对的样本的权重将会降低, 而第一次分错的样本的权重将会提高**.
- 为了从所有弱分类器中得到最终的分类结果, AdaBoost为每个分类器都分配了一个权重值alpha, 这些alpha值是基于每个弱分类器的错误率进行计算的.

其中, 错误率 ε 的定义如下:

$$\varepsilon = \frac{\text{未正确分类的样本数目}}{\text{所有样本数目}}$$

而 alpha 的计算公式如下:

$$\alpha = \frac{1}{2} \ln \left(\frac{1 - \varepsilon}{\varepsilon} \right)$$

AdaBoost算法的流程如下所示:

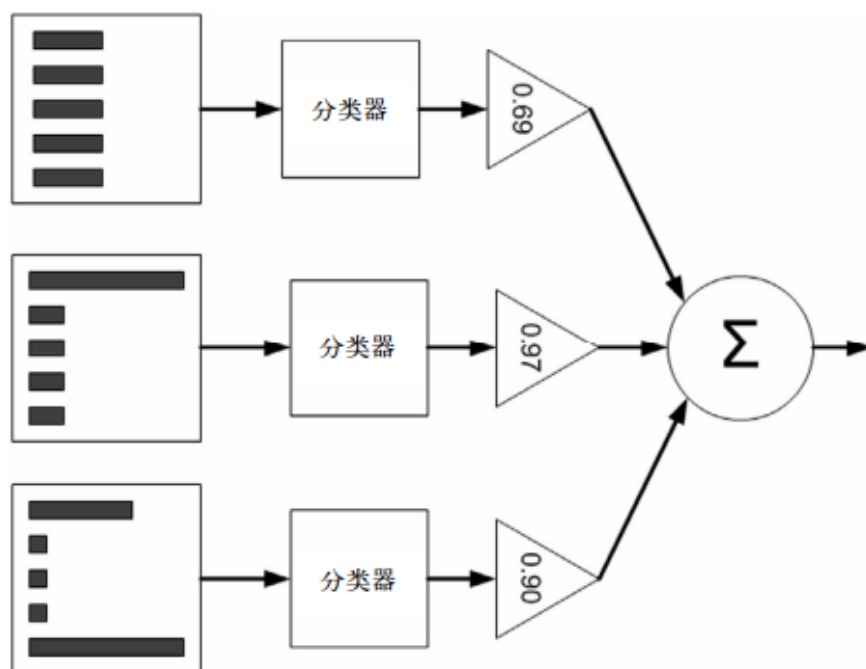


图7-1 AdaBoost算法的示意图。左边是数据集, 其中直方图的不同宽度表示每个样例上的不同权重。在经过一个分类器之后, 加权的预测结果会通过三角形中的alpha值进行加权。每个三角形中输出的加权结果在圆形中求和, 从而得到最终的输出结果

计算初alpha值之后, 可以对权重向量 D 进行更新, 以使得那些正确分类的样本的权重降低而错分样本的权重提高. D 的计算方法如下:

如果某个样本被正确分类, 那么该样本的权重更改为:

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{-\alpha}}{\text{Sum}(D)}$$

而如果某个样本被错分类, 那么该样本的权重更改为:

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{\alpha}}{\text{Sum}(D)}$$

在计算出D之后, AdaBoost又开始进入下一轮迭代. AdaBoost算法会不断地重复训练和调整权重的过程, 直到训练错误率为0或者弱分类器的数目达到用户的指定值为止.

8.2.3 AdaBoost的一般流程和优缺点

AdaBoost的一般流程:

- (1) 收集数据: 可以使用任意方法.
- (2) 准备数据: 依赖于所使用的弱分类器类型, 本章使用的是单层决策树, 这种分类器可以处理任何数据类型. 当然也可以使用任意分类器作为弱分类器, 第2章到第6章中的任一分类器都可以充当弱分类器. 作为弱分类器, 简单分类器的效果更好.
- (3) 分析数据: 可以使用任意方法.
- (4) 训练算法: AdaBoost的大部分时间都用在训练上, 分类器将多次在同一数据集上训练弱分类器.
- (5) 测试算法: 计算分类的错误率.
- (6) 使用算法: 同SVM一样, AdaBoost预测两个类别中的一个. 如果想把它应用到多个类别的场合, 那么就要像多类SVM中的做法一样对AdaBoost进行修改.

AdaBoost的优缺点:

- 优点: 泛化错误率低, 易编码, 可以应用在大部分分类器上, 无参数调整.
- 缺点: 对离群点敏感.
- 适用数据类型: 数值型和标称型数据.

8.2.3 AdaBoost算法的构建

单层决策树 (decision stump, 也称决策树桩) 是一种简单的决策树. 前面我们已经介绍了决策树的工作原理, 接下来将构建一个单层决策树, 而它仅基于单个特征来做决策. 由于这棵树只有一次分裂过程, 因此它实际上就是一个树桩.

一 基于单层决策树构建弱分类器

将最小错误率`minError`设为`+\infty`

对数据集中的每一个特征 (第一层循环):

 对每个步长 (第二层循环):

 对每个不等号 (第三层循环):

 建立一棵单层决策树并利用加权数据集对它进行测试

 如果错误率低于`minError`, 则将当前单层决策树设为最佳单层决策树

返回最佳单层决策树

总共包含三个函数, 一个是简单的数据集构建(`loadSimpleData`), 第二个是用于测试是否有某个值小于或者大于我们正在测试的阈值(`stumpClassify`), 第三个函数是在一个加权数据集中循环, 并找到具有最低错误率的单层决策树(`buildStump`).

具体的代码如下:

```

import numpy as np

"""
函数说明:创建单层决策树的数据集
parameters:
    无
Returns:
    dataMat - 数据矩阵
    classLabels - 数据标签
"""

def loadSimpleData():
    dataMat = np.matrix([[1,2.1],
                          [2,1.1],
                          [1.3, 1],
                          [1,1],
                          [2,1]])
    classLabels = [1,1,-1,-1,1]
    return dataMat, classLabels

"""
函数说明:单层决策树分类函数
Parameters:
    dataMatrix - 数据矩阵
    dimen - 第dimen列, 也就是第几个特征
    threshVal - 阈值
    treshIneq - 阈值不等式(两个,lt和gt)(分别表示less than和great than)
Returns:
    retArray - 分类的结果(np.array类型)
"""

def stumpClassify(dataMatrix, dimen, threshVal, threshIneq):
    #初始化分类结果,都为1
    retArray = np.ones((np.shape(dataMatrix)[0],1))
    #thresh_ineq == 'lt'表示阈值不等式取lt(less than)
    if threshIneq == 'lt':
        # data_mat[:, dimen] 表示数据集中第dimen列的所有值
        #如果小于等于阈值,则赋值为-1
        retArray[dataMatrix[:,dimen] < threshVal] = -1
    #表示阈值不等式取gt(great than)
    else:
        #如果大于阈值,则赋值为1
        retArray[dataMatrix[:,dimen] > threshVal] = 1
    return retArray

"""
函数说明:找到数据集上的最佳单层决策树
    单层决策树是指只考虑其中的一个特征, 在该特征的基础上进行分类, 寻找分类错误率最低的阈值即可
Parameters:
    dataArr - 数据矩阵
    classLabels - 数据标签
    D - 样本权重
Returns:
    bestStump - 最佳单层决策树信息
    minError - 最小误差
    bestClassEst - 最佳的分类结果
"""

def buildStump(dataArr, classLabels, D):
    #进行一些基本的初始化操作
    dataMatrix = np.matrix(dataArr)

```

```

lableMat = np.matrix(classLabels).T
m,n = np.shape(dataMatrix)

numSteps = 10
bestStump = {}
bestClassEst = np.mat(np.zeros((m,1)))
#最小误差初始化为正无穷大
minError = float('inf')

#第一层循环，遍历所有特征
for i in range(n):
    #每次找到该特征中的最小值和最大值
    rangeMin = dataMatrix[:,i].min()
    rangeMax = dataMatrix[:,i].max()
    #计算步长
    stepSize = (rangeMax - rangeMin) / numSteps
    #第二层循环
    for j in range(-1, int(numSteps)+1):
        """
        lt(less than)是指在该阈值下，如果<阈值，则分类为-1
        gt(greater than)是指在该阈值下，如果>阈值，则分类为-1
        """
        #第三层循环时再大于和小于之间切换不等式
        #遍历小于和大于的情况
        for threshIneq in ['lt', 'gt']:
            #计算阈值
            threshVal = (rangeMin + float(j)*stepSize)
            #计算分类结果
            predictedVals = stumpClassify(dataMatrix, i, threshVal,
threshIneq)

            #初始化误差矩阵
            errArr = np.mat(np.ones((m,1)))
            #分类正确的，赋值为0
            errArr[predictedVals == lableMat] = 0
            #基于权重向量D而不是其他错误指标来评价分类器的，不同的分类器计算方法不一样
            #计算误差--这里没有采用常规方法来评价这个分类器的分类准确率，而是乘上了权重
            weightedError = D.T * errArr

            #找到误差最小的分类方式
            if weightedError < minError:
                minError = weightedError
                bestClassEst = predictedVals.copy()
                bestStump['dim'] = i
                bestStump['thresh'] = threshVal
                bestStump['ineq'] = threshIneq
        return bestStump, minError, bestClassEst

if __name__ == '__main__':
    dataArr, classLabels = loadSimpleData()
    D = np.mat(np.ones((5,1)) / 5)
    bestStump, minError, bestClassEst = buildStump(dataArr, classLabels, D)
    print('bestStump: \n', bestStump)
    print('minError: \n', minError)
    print('bestClassEst: \n', bestClassEst)

```

结果如下:

```
Boost_Hsu01 x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/Boost_Hsu01.py
bestStump:
  {'dim': 0, 'thresh': 1.4, 'ineq': 'lt'}
minError:
  [[0.2]]
bestClassEst:
  [[-1.]
   [ 1.]
   [-1.]
   [-1.]
   [ 1.]]

Process finished with exit code 0
```

二 完整 AdaBoost 算法的实现

```
"""
函数说明：
    利用AdaBoost提升分类器性能
Parameters:
    dataArr - 数据矩阵
    classLabels - 数据标签
    numIt - 最大迭代次数
Returns:
    weakClassArr - 训练好的分类器
    aggClassEst - 类别估计累计值
"""
def adaBoostTrainDS(dataArr, classLabels, numIt = 40):
    #一些初始化操作
    weakClassArr = []
    m = np.shape(dataArr)[0]
    D = np.mat(np.ones((m,1)) / m)
    aggClassEst = np.mat(np.zeros((m,1)))

    for i in range(numIt):
        #得到决策树的模型
        bestStump, error, classEst = buildStump(dataArr, classLabels,D)
        print('D:', D.T)
        # 计算弱学习算法的权重alpha, 使error不等于0, 因为分母不能为0
        # alpha 目的主要是计算每一个分类器实例的权重(加和就是分类结果)
        # 计算每个分类器的 alpha 权重值
        alpha = float(0.5 * np.log((1 - error) / max(error, 1e-16)))
        #存储弱学习算法的权重
        bestStump['alpha'] = alpha
        #存储单层决策树
        weakClassArr.append(bestStump)
        print('classEst: ', classEst.T)
        #计算e的指数项
        expon = np.multiply(-1 * alpha * np.mat(classLabels).T, classEst)
```

```

#更新样本权重
D = np.multiply(D, np.exp(expon))
D = D / D.sum()

#计算AdaBoost误差，当误差为0时，退出循环
aggClassEst += alpha * classEst
print('aggClassEst:', aggClassEst.T)
aggErrors = np.multiply(np.sign(aggClassEst) != np.mat(classLabels).T,
np.ones((m,1)))
errorRate = aggErrors.sum() / m
print('total error:', errorRate)
if errorRate == 0:
    break
return weakClassArr, aggClassEst

if __name__ == '__main__':
    dataArr, classLabels = loadSimpleData()
    weakClassArr, aggClassEst = adaBoostTrainDS(dataArr, classLabels)
    print(weakClassArr)
    print(aggClassEst)

```

结果如下:

```

Boost_Hsu01 x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/Boost_Hsu01.py
D: [[0.2 0.2 0.2 0.2 0.2]]
classEst: [[-1.  1. -1. -1.  1.]]
aggclassEst: [[-0.69314718  0.69314718 -0.69314718 -0.69314718  0.69314718]]
total error: 0.2
D: [[0.5  0.125 0.125 0.125 0.125]]
classEst: [[1.  1.  1.  1.  1.]]
aggclassEst: [[-0.14384104  1.24245332 -0.14384104 -0.14384104  1.24245332]]
total error: 0.2
D: [[0.33333333 0.08333333 0.25      0.25      0.08333333]]
classEst: [[ 1. -1. -1. -1. -1.]]
aggclassEst: [[ 0.66087792  0.43773437 -0.94855999 -0.94855999  0.43773437]]
total error: 0.0
[{'dim': 0, 'thresh': 1.4, 'ineq': 'lt', 'alpha': 0.6931471805599453}, {'dim': 0, 'thresh': 0.9, 'ineq': 'lt', 'alpha': 0.66087792}, {'dim': 0, 'thresh': 0.9, 'ineq': 'lt', 'alpha': 0.43773437}, {'dim': 0, 'thresh': 0.9, 'ineq': 'lt', 'alpha': -0.94855999}, {'dim': 0, 'thresh': 0.9, 'ineq': 'lt', 'alpha': -0.94855999}]
[[ 0.66087792]
 [ 0.43773437]
 [-0.94855999]
 [-0.94855999]
 [ 0.43773437]]

Process finished with exit code 0

```

三 测试算法：基于 AdaBoost 的分类

一旦拥有了多个弱分类器以及其对应的alpha值, 进行测试就变得相当容易了. 测试函数如下:

```

"""
函数说明：
    AdaBoost分类函数
Parameters:
    datToClass - 待分类样列
    classifierArr - 训练好的分类器
Returns:

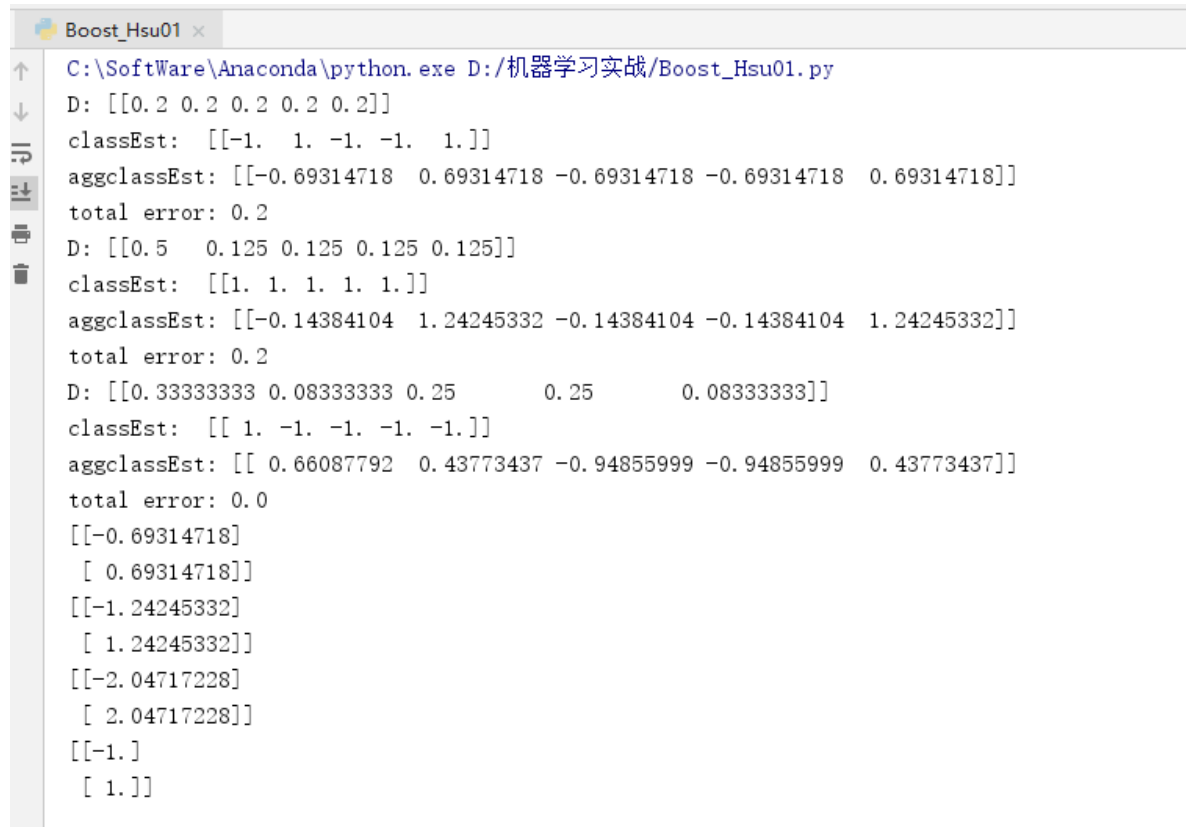
```

分类结果

```
"""
def adaClassify(dataToClass, classifierArr):
    dataMatrix = np.mat(dataToClass)
    m = np.shape(dataMatrix)[0]
    aggClassEst = np.mat(np.zeros((m,1)))
    #遍历所有的分类器，进行分类
    for i in range(len(classifierArr)):
        classEst = stumpClassify(dataMatrix, classifierArr[i]['dim'],
                                classifierArr[i]['thresh'],
                                classifierArr[i]['ineq'])
        aggClassEst += classifierArr[i]['alpha'] * classEst
    print(aggClassEst)
    return np.sign(aggClassEst)

if __name__ == '__main__':
    dataArr, classLabels = loadSimpleData()
    weakClassArr, aggClassEst = adaBoostTrainDS(dataArr, classLabels)
    print(adaClassify([[0,0],[5,5]], weakClassArr))
```

结果如下:



```
Boost_Hsu01 x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/Boost_Hsu01.py
D: [[0.2 0.2 0.2 0.2 0.2]]
classEst: [[-1.  1. -1. -1.  1.]]
aggclassEst: [[-0.69314718  0.69314718 -0.69314718 -0.69314718  0.69314718]]
total error: 0.2
D: [[0.5  0.125 0.125 0.125 0.125]]
classEst: [[1.  1.  1.  1.  1.]]
aggclassEst: [[-0.14384104  1.24245332 -0.14384104 -0.14384104  1.24245332]]
total error: 0.2
D: [[0.33333333 0.08333333 0.25  0.25  0.08333333]]
classEst: [[ 1. -1. -1. -1. -1.]]
aggclassEst: [[ 0.66087792  0.43773437 -0.94855999 -0.94855999  0.43773437]]
total error: 0.0
[[-0.69314718]
 [ 0.69314718]]
[[-1.24245332]
 [ 1.24245332]]
[[-2.04717228]
 [ 2.04717228]]
[[-1.]
 [ 1.]]
```