第八章 基于树的模型

基于树的方法是一种通过将预测变量空间划分为多个简单区域进行预测的模型，每个区域对应一个决策规则，可用树状结构表示。这也被称为**决策树 (decision tree)**

简单的决策树在分类和回归任务中不如第六和第七章描述的正则化方法或样条模型；但是，我们后续介绍的集成方法将会有更高的精度，尽管牺牲了一些可解释性

8.1 决策树基础

决策树可用于回归和分类两类问题

8.1.1 回归树

用于回归问题的决策树就是**回归树 (regression tree)**

在利用回归树预测运动员工资的例子中，我们发现回归树的如下特点：

1. 良好的分支结构，清晰明了
2. 相对于线性回归，树的分支、结点似乎更容易解释

构建决策树的过程，粗略的说可以分为：

1. 划分预测器空间，对于预测器向量组 $X_1, X_2, \cdots, X_p$，划分 J 个区域
2. 对于每个落入区域 R_j 的观测值，我们作同样的预测：简单的可以认为它们都在原始观测空间 R_j 所有数据的均值上

理论上，构造的空间 R_j 可以为任意形状，但我们倾向于将其划分为高维立方体，或者可以叫做**盒子 (boxes)**。寻找能使模型获得最小 RSS 的盒子遵循公式：

$$\sum_{j=1}^{J}\sum_{i\in R_j}(y_i-\hat{y}_{R_j})^2$$

其中 $\hat{y}_{R_j}$ 是第 j 个盒子所有观测值的均值

在计算上，对于全部的特征，考虑所有的划分方式是不可能实现的。为此，我们采用**递归二分法 (Recursive Binary Splitting)**，或者称为**自顶向上的贪婪方法 (Top-Down Greedy Approace)** 来划分盒子。这种方法从树的顶端开始，选择一个特征 X_j，选择能最小化 RSS 的阈值 s，划分出一片空间，再对所有的特征递归使用此方法，可以划分出所有的空间

> 这种算法的思想是贪婪。每步只考虑了当前的最优分发，但不能保证持续的划分在全局上都是最优

对于任意的 j 和 s，我们定义**半平面对 (half-planes)**:

$$R_1(j,s)=\{X|X_j<s\}\text{ and }R_2(j,s)=\{X|X_j\leq s\}$$

然后我们寻找合适的 j 和 s，使得最小化下式：

$$\sum_{i:x_i\in R_1(j,s)}(y_i-\hat{y}_{R_1})^2+\sum_{i:x_i\in R_2(j,s)}(y_i-\hat{y}_{R_2})^2$$

其中 $\hat{y}_{R_i}$ 表示第 i 个平面训练值的平均值。当 p 不大时，寻找对应 j 和 s 的值相当快

划分完第一个特征后，我们继续进行划分，不同的是，我们不再对整个区域进行划分，而是选择划分好的一块区域，再根据新的特征，或对原来特征进一步约束来划分新区域

区域划分的终止条件可以设置为划分后剩下的叶子结点数量 (即最后的所有可能区域数)

我们拟合的树模型可能在训练集上表现较好，测试集的性能不佳，这是因为生成的复杂的树容易导致过拟合

一种解决过拟合的方案是设置分裂阈值，即到达某个阈值的 RSS 减小时，才对树进行分裂。然而，这种策略可能导致忽略前期的有用分裂 (这种分裂为后期的大量 RSS 减小创造了条件)

因此，一种更优的策略是生成一颗足够大的树，然后**剪枝 (prune)**，最后得到一颗它的**子树 (subtree)**。选择更好子树的指标是它的测试误差

> 交叉检验或验证集方法可以寻找这种子树，但可能过于繁琐

我们使用**代价复杂度剪枝 (cost complexity pruning)**，也叫**最弱连接剪枝 (weakest link pruning)** 来获取合适的子树。对于任意一颗子树，都有一个非负的惩罚项 α 对应，对 α 的每个值，子树 $T\in T_0$，满足：

$$\sum_{m=1}^{|T|}\sum_{i:x_i\in R_m}(y_i-\hat{y}_{R_m})^2+\alpha|T|$$

要使上式尽可能小。其中 $|T|$ 表示树 T 的终端结点数，R_m 是第 m 个终端结点对应的矩形 (预测空间的子集)，$\hat{y}_{R_m}$ 表示对应 R_m 中训练值的均值，调节参数 α 调控子树复杂度和拟合度的平衡

当子树的 $\alpha=0$ 时，子树 T 简单地等于 T_0，因此上式只度量训练误差。然而，随着 α 增加，多个终端结点的树付出的代价增大，对于较小的树上式值反而较小。剪枝方法将从叶子结点向上，依次删除对于 RSS 影响最小但对于模型复杂度影响最大的分支

> 这种控制方法类似于 lasso 回归

对于 α，我们可以用交叉检验的方式来选取

构建回归树的算法归结如下：

1. 使用递归二进制分裂在训练数据上生成一颗大树，直到某个区域内的观测数少于阈值
2. 应用代价复杂度剪枝，获取最优子树序列

3. 用 K 折交叉检验选择 α，对于每个 $k=1,2,\cdots,K$，用除了 k 的训练数据拟合模型，k 用于检验误差，取平均值，找到一个使平均误差最小的 α
4. 从步骤 2 中选择对应的子树

8.1.2 分类树

分类树 (classification tree) 与决策树最大的区别在于前者预测的是定性变量，而后者预测的是定量的变量。分类树的终端结点的预测值与盒子内的最多数量的种类有关，且我们关心它的比例

生成分类树比回归树更加简单。替代 RSS 的检验指标是**分类错误率 (classification error rate)**：

$$E=1-\max(\hat{p}_{mk})$$

其中 $\hat{p}_{mk}$ 表示来自第 k 类且在第 m 区域的训练观测值比例。

> 事实上，分类误差对树木的生长不够敏感，需要更优的措施

基尼指数 (Gini index) 被定义为：

$$G=\sum_{k=1}^{K}\hat{p}_{mk}(1-\hat{p}_{mk})$$

其中 $\hat{p}_{mk}$ 的定义同上。当基尼指数很小时，$\hat{y}_{mk}$ 接近 1 或 0，可用于衡量一个节点分类的纯度，即大部分是否都为同一类

> 基尼指数还有一个定义是 $G=1-\sum_{k=1}^{K}p_k^2$，等价，是上式的展开式

基尼指数的一个代替是**熵 (entropy)**:

$$D=-\sum_{k=1}^{K}\hat{p}_{mk}\log\hat{p}_{mk}$$

由于 $0\le\hat{p}_{mk}\le1$，所以 D 是一个大于 0 的值。当 $\hat{p}_{mk}$ 接近于 0 或 1 时，D 会取一个很小值，跟基尼指数很类似

构建分类树可以用上面的三种指标，对于分割质量，我们采用后两种指标；若考虑更精确的分类效果，我们采用第一个指标分类错误率

注意到一颗完全生长的决策树，一个结点分支出的两个终端结点，可能会有相同的分类结果。出现这个的原因是此划分遵循了基尼指数和熵的优化，提升了这片空间的纯度，因而 s 被选中。纯度的提升有助于高置信预测，减少过拟合并提升解释性

> 例如，两个兄弟终端结点的分类都是 yes，但右侧的纯度为 100%，若有后续预测变量分配到右侧，我们有很高的置信率去认为它也是 yes，便于解释

8.1.3 树和线性回归的比较

线性回归的预测遵循下面的式子：

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j$$

而回归树则遵循：

$$f(X) = \sum_{m=1}^{M} c_m \cdot 1_{(X \in R_m)}$$

其中 R_m 表示划分的特征空间，$1_{(X \in R_m)}$ 是指示函数，表示 $X \in R_m$ 时为 1，否则为 0，c_m 表示对应区域内的预测值

选择模型的标准：

- 特征和响应之间高度线性，使用线性回归
- 特征和响应之间高度非线性，关系复杂，决策树方法优于传统方法
  除此之外，出于可解释性和可视化的考虑，决策树的预测更受欢迎

8.1.4 树模型的优缺点

优点：

1. 容易解释。有时候，甚至比线性回归更加容易解释
2. 比经典回归和分类方法更加真实地反映人类决策
3. 树可以图形化，非专业人士可以看懂
4. 树可以很容易处理定性预测器，不需要哑变量

缺点：

1. 树的预测精度在一般情况下不如其他一些分类和回归方法
2. 树的稳健性较差，数据的微小变化容易引起树模型的较大变化

聚合许多决策树、使用 Bagging、随机森林和 Boosting 等方法可以大幅提高预测性能

8.2 树的集成方法

集成 (ensemble) 的含义是组合许多简单的模块，以获得一个独立而非常有效的模型。这些简单的积木模型有时候被称为**弱学习器 (weak learners)**，因为它们本身可能得较弱的预测

8.2.1 Bagging

自助聚集法 (Boostrap Aggregating) 是一种使用了自助法来改进决策树模型的新模型

决策树的高方差导致了其容易过拟合。例如，将数据随机分为两类，分别拟合决策树，最后的决策结果可能很不同，这是因为决策树对数据细节非常敏感。决策树是一类**高方差模型 (high variance model)**，而使用 Bagging 方法可以降低它的方差

> 与之相反，当 $n >> p$ 时，线性回归是一类**低方差模型 (low variance model)**

回顾方差的计算方法。对于 n 个独立观测值 $Z_1, \cdots, Z_n$，每个观测值的方差为 σ^2，则这些观察值的均值 $\overline{Z}$ 的方差就是 σ^2/n。因此，对于独立观察值，我们通过平均的方差可以降低方差。我们可以通过计算 $\hat{f}^1(x), \hat{f}^2(x), \cdots, \hat{f}^B_(x)$ 来计算 B 个独立模型的输出值，然后取平均作为输出值，即：

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)$$

在实际中，这是不实际的，因为我们通常不拥有多个数据集。但是我们可以在单个数据集中使用 bootstrap 方法来生成多个小数据集，然后分别训练模型，再输出结果：

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

Bagging 可以改进许多回归方法的预测，但对决策树特别有用。通过继承数百甚至上千颗未经过修剪的决策树，然后取平均输出，Bagging 已被证明在准确性上有显著改进

> 树数量 B 的大小不是关键参数，不会导致模型的过拟合。实践中我们采用显著降低误差的 B 作为最终大小

对于分类问题，我们的 Bagging 方法采用**多数投票 (majority vote)** 来获得最终的结果。即得到所有子决策树的输出，然后取最多的那一类预测

事实证明，一个简单的估计 Bagging 模型测试误差的方法是**袋外误差估计 (out-of-bag Error Estimation)**。由于 boostrap 方法是放回抽样，每次约有 $1 - \frac{1}{e}$ (约为 $1/3$) 的数据没有被抽取到。我们利用袋外数据来估计单颗子树的测试误差，再把所有的误差平均。当 B 足够大时，OOB 估计实验误差等价于 LOOCV。OOB 估计对大数据集较为友好

Bagging 方法损失了可解释性，以获得模型准确率 (我们不能再通过单颗子树绘图了)。但是，我们可以考虑一些指标来评估每个特征的重要性，也即**变量重要性 (variable importance)**

我们可以这样收集变量重要性：

- 对于回归树，在每颗子树中，对于一个特征的每次分裂，记录 RSS 的减少总和，最后对所有 B 颗树进行平均，以显著降低 RSS 的特征为重要
- 对于分类树，在每颗子树中，对于一个特征的每次分裂，记录基尼指数的下降总和，最后对所有 B 颗树进行平均，以显著降低基尼指数的特征为重要

> 在教材中，变量重要性被映射到一个 $[0, 100]$ 的区间上

8.2.2 随机森林

随机森林 (random forest) 实现了对 Bagging 方法的改进。Bagging 方法每次构建树时，都是从同一个数据集中进行抽样再建立的，这将会导致树与树之间更加强的相关关系，从而弱化模型的泛化能力

与 Bagging 类似，随机森林也是在一个数据集中抽取多个 Bootstrap 样本，但有一个关键的改进：每次分裂时，从 p 个特征中随机选取 m 个，然后再从 m 个中选择最优的进行分裂。实践中，$m \approx \sqrt{p}$

> 例如，若 p 中存在一个高强的预测器，其他为中强，则大部分树都会选择此高强预测器，导致树之间高度相关。在这种情况下，平均所有树的输出不能显著降低模型的方差

随机森林迫使每个分裂只考虑预测变量的一个子集，这可以看作是对树的去相关处理，减小树输出的平均值变动。随机森林的关键即是选择 m 的大小，当 $m = p$ 时，等同于 Bagging；当大量相关的预测变量存在时，我们使用很小的 m 值会有很大帮助

8.2.3 Boosting

Boosting是一种集成学习方法，通过组合多个弱学习器来构建一个强学习器 (就像 Bagging)，可用于多种模型中提升效率。下面的讨论将仅限于树模型

Boosting 同样生成一系列的子树，不同的是，每棵树都是利用先前的树的信息生长的。Boosting 不涉及 bootstrap 抽样，而是每棵树在原始数据集的修改版本上进行拟合

与直接对 Y 来拟合一颗决策树不同，Boosting 方法对模型的残差进行拟合。然后，将一颗新的决策树加入拟合函数中，更新残差，这样算法中的每一颗树都很小 (少量终端结点，由算法中的 d 决定)，通过残差对小树拟合，在表现不好的区域慢慢提升 $\hat{f}$，允许更多不同的形状来削减残差

> 注意 Boosting 与 Bagging 不同在每颗树都依赖于前面的树，不是独立

Boosting 算法应用于回归树可以如下描述：

1. 设置初始模型 $\hat{f}(x) = 0$，初始化残差 $r_i = y_i$ (容易从 $\hat{f}(x) = 0$ 推出)
2. 对于每一颗子树，从 $1, 2, \cdots, B$，重复
   2.1 拟合一棵树 $\hat{f}^b$，并划分 d 个分支 (最终得到 $d + 1$ 个终端结点) 到训练数据 (X, r)
   2.2 通过添加一个缩小版本的新子树，更新总体模型 $\hat{f}$：

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

   2.3 同时更新残差

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. 输出总的 Boosted 模型：

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

> 2.1 中提到的训练数据 (X, r) 表示一个聚合矩阵，其中 X 表示特征矩阵，r 在此处表示残差向量。每个模型都试图通过这两份数据更新

在决策树中应用 Boosting 方法，有三个调谐参数：

1. 子树数量 B，不同于 Bagging 方法，过大的 B 会导致 Boosting 过拟合。通过交叉检验选择合适的 B
2. 收缩参数 λ，也即是学习率。经典的值取 0.01 或者 0.001，正确地选择取决于问题。非常小的 λ 需要一个非常大的 B 以获取较优的模型
3. 每颗子树的分裂数 d，控制模型复杂性。$d = 1$ 通常表现的较好，这时候树变为**树桩 (stump)**，只有一组分裂，此时 Boosting 是加性模型，每次迭代只有一个特征；更广泛地，d 被称为**交互深度**，多次的分裂可以学习变量之间的交互关系

> 单颗树的加性模型有助于解释模型的作用

8.2.4 贝叶斯加性回归树

现在我们讨论**贝叶斯加性回归树 (Bayesian additive regression trees, BART)**，这也是一种决策树的集成模型，它综合了前面两类集成学习的特点，新颖之处在于生成树的方式

定义 K 表示回归树数量，B 表示模型的迭代次数，$\hat{f}_k^{(b)}(x)$ 表示在第 b 论迭代中，第 k 颗树对输入 x 的预测

在 BART 的第一轮中，每棵树都只有一个根节点 (即不做任何划分)，预测值为所有目标值 y_i 的均值，除以树的数量 k：

$$\hat{f}^1(x) = \sum_{k=1}^{K} \hat{f}_{k=1}^1(x) = \frac{1}{n} \sum_{i=1}^{n} y_i$$

在 BART 的后续迭代中，它将逐步更新每棵树 (类似梯度提升树 GBDT)。对于第 b 轮需要更新的第 k 棵树，计算**部分残差 (partial residual)**：

$$r_i = y_i - \sum_{k' < k} \hat{f}_{k'}^b(x_i) - \sum_{k' > k} \hat{f}_{k'}^{b-1}(x_i)$$

> 上式的负数项可以分解为此轮已被更新的树模型预测值和未被更新的树模型的预测值；表示 T_k 的残差

然后根据所有观测值的残差，更新树。BART 不是将一颗新树来拟合这部分残差，而是从一组可能的**扰动 (perturbation)** 来更新，这包括：

1. 改变树的结构。包含分裂扩展和剪枝分支
2. 改变这棵树每个终端结点的预测值

通常，舍弃迭代轮次前面的模型，因为这些模型的拟合效果可能不太好，处于所谓的**预热期 (burn-in)**。最终模型的预测均值将不包含预热期的值，记为：

$$\hat{f}(x)=\frac{1}{B-L}\sum_{b=L+1}^{B}\hat{f}^b(x)$$

其中 L 表示预热期模型数量

BART 算法可以概述为：

1. 拟合初始模型 $\hat{f}_1^1(x)=\hat{f}_2^1(x)\cdots=\hat{f}_K^1(x)=\frac{1}{nL}\sum_{i=1}^n y_i$
2. 计算第一次预测值 $\hat{f}^1(x)=\sum_{k=1}^K\hat{f}_k^1(x)=\frac{1}{n}\sum_{i=1}^n y_i$
3. 对于后续的迭代次数 d：
   3.1 对于 $k=1,2,\ldots,K$：
   3.1.1 对于 $i=1,2,\ldots,n$，计算部分残差 $r_i=y_i-\sum_{k'<k}\hat{f}_{k'}^b(x_i)-\sum_{k'>k}\hat{f}_{k'}^{b-1}(x_i)$
   3.1.2 用随机扰动拟合新树 $\hat{f}_k^b(x)$ 对 r_i，选择提升最好的那颗树
   3.2 计算本次预测值 $\hat{f}^b(x)=\sum_{k=1}^K\hat{f}_k^b(x)$
4. 计算预热期 L 之后的预测均值：

$$\hat{f}(x)=\frac{1}{B-L}\sum_{b=L+1}^{B}\hat{f}^b(x)$$

> 关键步骤是 3.1.2，即采用随机扰动修改树，不是完全重新拟合一颗新的树，避免过拟合

实验表明，BART 的训练和测试误差在 K 增大到一定程度后趋于平缓，而 Boosting 的训练误差随着 B 增大一直减小，测试误差则下降到一定程度趋于平缓。这表明前者没有产生过拟合，后者产生了。BART 防止过拟合主要归功于随机扰动的树修改方法

随机扰动对树的修改使用了拟合树集合的贝叶斯方法，每次的修改实际上是从后验分布中绘制一颗新的树。上述算法可以看做拟合 BART 的**马尔科夫链蒙特卡洛 (Markov chain Mont Carlo)** 算法

在实际应用中，我们通常选择较大的 B 和 K 值，适中的 L 值。BART 拥有很好的箱外表现，也即在最小的参数调优代价下的更优表现

8.2.5 集成树模型的总结

决策树是很适用于集成方法的，原因在于其灵活性和处理多个特征的能力。下面对上面提到的集成树模型进行总结：

- Bagging 的树木在随机观测样本上独立生长，树木之间往往非常相似，容易陷入局部最优，不能彻底探索模型空间
- 随机森林中的树木同样独立生长在观测的随机样本上，然而每次的分裂都是使用特征的随机子集，去除了部分相关性
- Boosting 方法只使用原始数据而不再次抽样，树木依次生长，采用缓慢的学习方法，每颗树都适应早期树木留下来的信号，并在使用前收缩

- BART 同样使用原始数据，树木依次生长。为了避免局部极小值，每颗树的扰动对模型进行了更加彻底的探索

8.3 实验: 基于树的模型

基础包：

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
from statsmodels.datasets import get_rdataset
import sklearn.model_selection as skm
from ISLP import load_data, confusion_table
from ISLP.models import ModelSpec as MS
```

本章新用到的包：

```
from sklearn.tree import ( DecisionTreeClassifier as DTC ,
DecisionTreeRegressor as DTR, plot_tree , export_text)
from sklearn.metrics import (accuracy_score , log_loss)
from sklearn.ensemble import \
    (RandomForestRegressor as RF, GradientBoostingRegressor as GBR)
from ISLP.bart import BART
```

8.3.1 拟合分类树

下面使用分类树拟合 `Carseats` 数据集。首先，我们用 `np.wahere()` 方法来将连续的 `sale` 特征转换为离散特征：

```
#加载数据, 构建分类变量
Carseats = load_data('Carseats')
High = np.where(Carseats.Sales > 8,
                "Yes",
                "No")
```

使用 `DecisionTreeClassifier()` 函数构造分类树模型，它被重命名为 `DTC` 。 `DTC` 接受一个无截距的特征矩阵，而不是一个 `df` ，因此需要先进行修改：

```
#准备特征矩阵
model = MS(Carseats.columns.drop('Sales'), intercept=False)
D = model.fit_transform(Carseats) #获取特征df
feature_names = list(D.columns)
X = np.asarray(D) #将特征df转换为一个矩阵
```

拟合分类树，并用 `accuracy_score()` 方法来查看分类的准确率：

```
#拟合分类树
clf = DTC(criterion='entropy',
          max_depth=3,
          random_state=0)
clf.fit(X, High) #ClassifcationRegressionTree接受X而不是D

#显示分类得分
print(accuracy_score(High, clf.predict(X))) #得分0.7275
```

事实上，树模型在处理定性特征时，只需要按特征分裂，而不是需要先转换为数字再分裂，这不同于线性回归模型。

> `statsmodels`是将定性特征转换为独热编码或者哑变量再拟合的，与其通用性需求有关

对于分类树，我们用`log_loss()`来获取它模型的偏差，它基于：

$$-2\sum_m\sum_k n_{mk}\log\hat{p}_{mk}$$

其中n_{mk}表示第m个终端结点中属于k类的观测值数量。这可以用于反应模型对数据的拟合效果：

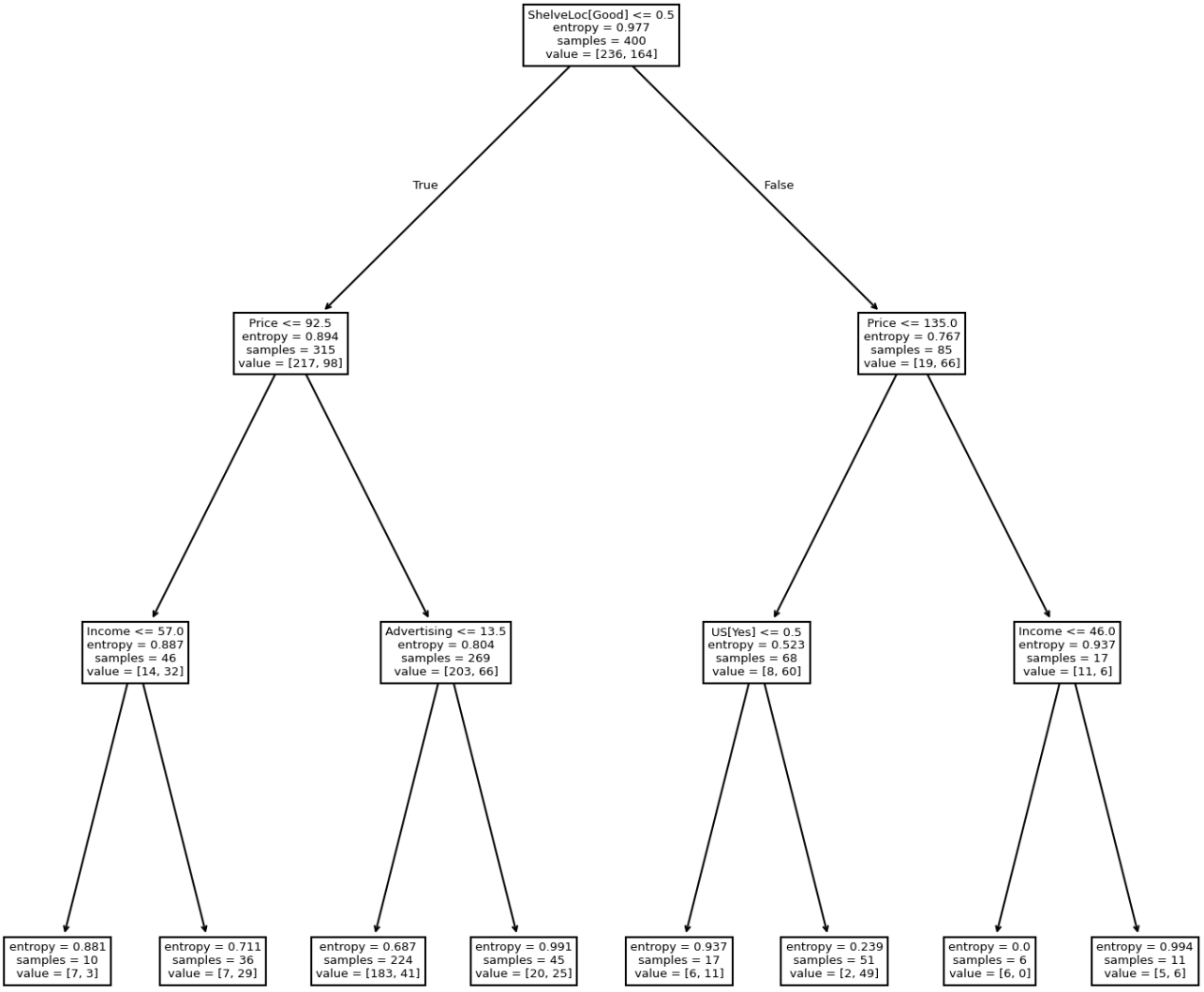
```
#显示偏差
resid_dev = np.sum(log_loss(High, clf.predict_proba(X)))
print(resid_dev)
```

这个值十分接近于我们定义的熵。越小的偏差表明越好的训练数据拟合效果

用`plot_tree()`函数绘制分类树的图像：

```
#绘制分类树的图像
ax = plt.subplots(figsize =(12 ,12))[1]
plot_tree(clf , feature_names=feature_names , ax=ax) #利用到之前获得的特征名列表
plt.show()
```

根据图象，最重要的特征是`ShelveLoc`，被作为根节点的分类



使用`export_text()`函数我们还能看到用文本形式显示的树的分类，指定`show_weight`参数为`True`时，还能看到为分类数量的具体值：

```
#文本形式的分类图像
print(export_text(clf,
                  feature_names=feature_names,
                  show_weights=True))
```

准确的评估分类树性能涉及到训练集和测试集，下面计算分类树的测试误差：

```
#计算测试误差
validation = skm.ShuffleSplit(n_splits=1, test_size=200, random_state=0) #划分策略指定，仅划分一次测试集
results = skm.cross_validate(clf, D, High , cv=validation) #获取交叉检验结果
print(results['test_score']) #显示测试分数，仅有一个值
```

> `skm.ShuffleSplit`是一个划分策略对象，用于`cross_validate`中的`cv`参数

我们尝试对全树进行剪枝，并用交叉检验获取最好的剪枝树。首先划分训练集和测试集：

```
#划分训练集和测试集
(X_train,
 X_test,
 High_train,
 High_test) = skm.train_test_split(X,
                                   High,
                                   test_size=0.5,
                                   random_state=0)
```

然后，拟合全树，并设计交叉检验方案。注意这里的 `DTC` 没有限制最大深度：

```
#重新在训练集上拟合完整的树，然后剪枝
clf = DTC(criterion='entropy',
          random_state=0)
clf.fit(X_train, High_train)
ccp_path = clf.cost_complexity_pruning_path(X_train, High_train)
kfold = skm.KFold(10, random_state=1, shuffle=True)
```

使用网格搜索，获取最后的结果：

```
#利用网格搜索获取最后的结果
grid = skm.GridSearchCV(clf,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X_train, High_train)
print(grid.best_score_)
```

这是一颗非常复杂的树，我们来看看它的叶子数量：

```
#查看图像
ax = plt.subplots(figsize =(12, 12))[1]
best_ = grid.best_estimator_ #获取grid的最优估计，后续查看属性
plot_tree (best_ , feature_names=feature_names , ax=ax)
plt.show()

#查看叶子数
print(best_.tree_.n_leaves)
```

实际上，交叉检验只为我们减少了5个终端结点数量，同时模型准确率相比于全模型下降了

查看测试集中的混淆矩阵：

```
#显示混淆矩阵
print(accuracy_score(High_test, best_.predict(X_test)))
```

```
confusion = confusion_table(best_.predict(X_test), High_test)
print(confusion)
```

交叉检验能为我们修剪树，但是效果非常有限，这与单颗树的大方差有关

8.3.2 拟合回归树

拟合回归树的操作与分类树类似。我们先划分数据集：

```
#准备数据
Boston = load_data("Boston")
model = MS(Boston.columns.drop('medv'), intercept=False)
D = model.fit_transform(Boston)
feature_names = list(D.columns)
X = np.asarray(D)

#划分数据集
(X_train,
 X_test,
 y_train,
 y_test) = skm.train_test_split(X,
                                Boston['medv'],
                                test_size=0.3,
                                random_state=0)
```

然后用`DTR()`函数建立回归树模型：

```
#建立回归树模型
reg = DTR(max_depth=3)
reg.fit(X_train, y_train)
ax = plt.subplots(figsize=(12,12))[1]
plot_tree (reg , feature_names=feature_names, ax=ax)
```

现在尝试交叉检验对性能的影响：

```
#用交叉检验查看剪枝对性能的影响
ccp_path = reg.cost_complexity_pruning_path(X_train, y_train)
kfold = skm.KFold(5, shuffle =True , random_state=10)
grid = skm.GridSearchCV(reg, {'ccp_alpha': ccp_path.ccp_alphas}, refit =True
, cv=kfold , scoring='neg_mean_squared_error')
G = grid.fit(X_train, y_train)
best_ = grid.best_estimator_
print(np.mean((y_test - best_.predict(X_test))**2)) #计算测试集均方误差

#绘制最优树
ax = plt.subplots(figsize =(12 ,12))[1]
plot_tree(G.best_estimator_,
```

```
            feature_names=feature_names,
            ax=ax)
plt.show()
```

8.3.3 Bagging和随机森林

Bagging实际上是随机森林的选择特征子集参数$m = p$的一个特例，因此，我们使用`RandomForestRegressor()`函数，它来自`sklearn.ensembel`包，来拟合这两个模型

首先拟合Bagging模型：

```
#建立bagging模型
bag_boston = RF(max_features=X_train.shape[1], random_state=0)
bag_boston.fit(X_train, y_train)

#绘图，显示预测效果
ax = subplots(figsize=(8, 8))[1]
y_hat_bag = bag_boston.predict(X_test)
ax.scatter(y_hat_bag, y_test)
print(np.mean(y_test - y_hat_bag)**2) #显示MSE
```

其中参数`max_features`用于指定选择特征子集的大小

通过改变`n_estimators`参数，我们可更改森林中树木的数量(默认100)：

```
#更改树木数量，重新拟合模型
bag_boston = RF(max_features=X_train.shape[1],
                n_estimators=500,
                random_state=0).fit(X_train, y_train)
y_hat_bag = bag_boston.predict(X_test)
print(np.mean(y_test - y_hat_bag)**2)
```

不断提升B的大小不会导致过拟合，但是不足的B将会导致欠拟合

拟合随机森林只需要重新指定`max_features`参数大小就行了：

```
#生长随机森林
RF_boston = RF(max_features=6,
               random_state=0).fit(X_train, y_train)
y_hat_RF = RF_boston.predict(X_test)
print(np.mean(y_test - y_hat_RF)**2)
```

在RF中我们可以查看变量的重要性：

```
#提取变量重要性
feature_imp = pd.DataFrame( #生成df格式的数据，便于查看
```

```
    {'importance': RF_boston.feature_importances_},
    index=feature_names #重命名行名
)
print(feature_imp.sort_values(by='importance', ascending=False)) #按照重要性
降序显示
```

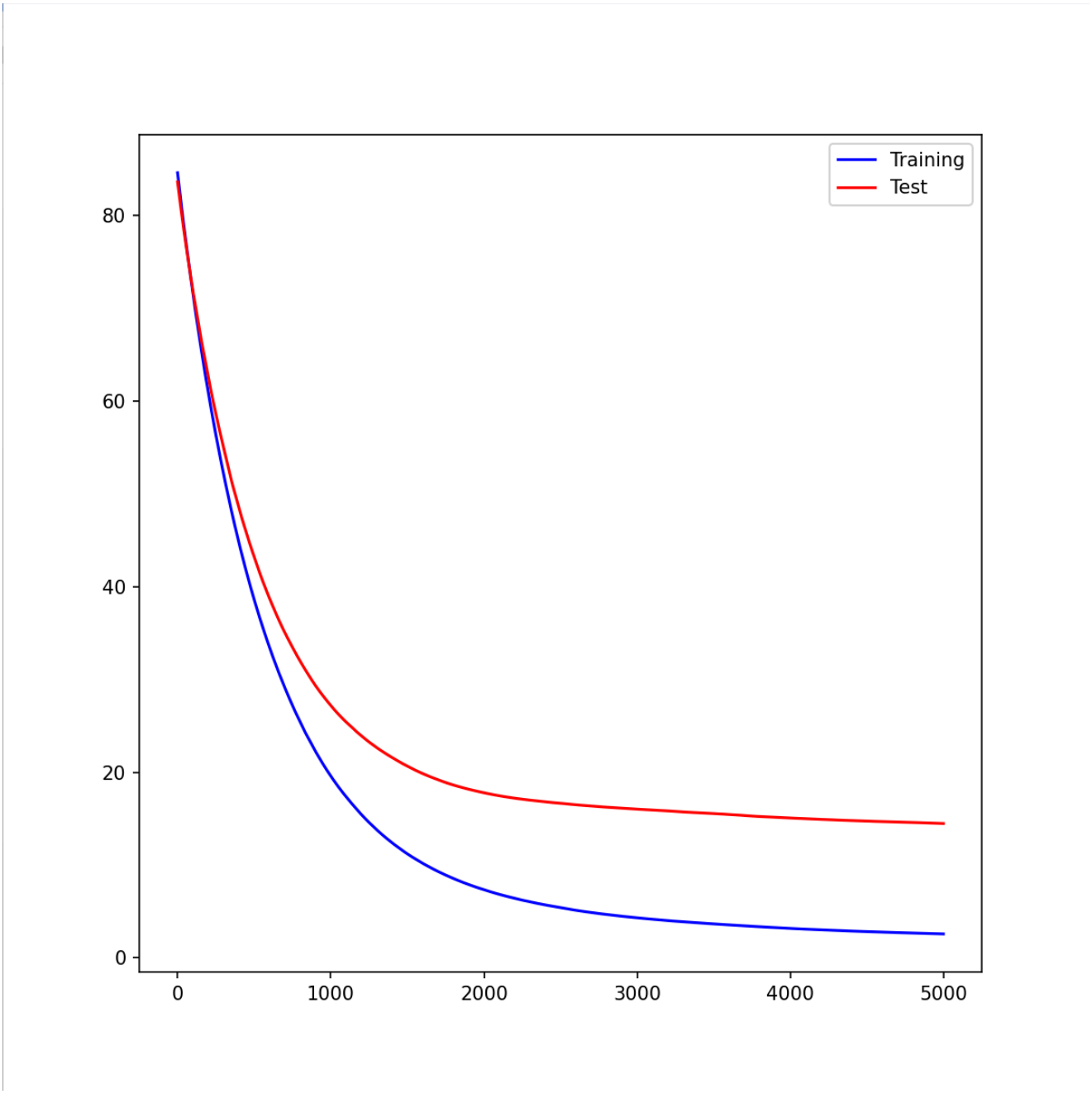
8.3.4 Boosting

下面使用 `GradientBoostingRegressor()` 函数，它来自 `sklearn.ensemble` 包来用Boosting方法拟合Boston数据集；对于分类问题则使用 `GradientBoostingClassifier()` 。其中参数 `n_estimators` 指定树木数量， `max_depth` 指定每颗树的深度，学习率参数 `learning_rate` 就是参数λ：

```
#拟合Boosting模型
boost_boston = GBR(n_estimators=5000,
                   learning_rate=0.001,
                   max_depth=3,
                   random_state=0)
boost_boston.fit(X_train, y_train)
```

通过 `train_score_` 属性可以看到训练误差是如何通过训练路径减小的，而 `stage_prediction()` 方法则可以得到沿着路径的预测值：

```
#查看误差减小路径
test_error = np.zeros_like(boost_boston.train_score_)
for idx, y_ in enumerate(boost_boston.staged_predict(X_test)):
    test_error[idx] = np.mean((y_test - y_)**2)
plot_idx = np.arange(boost_boston.train_score_.shape[0])
ax = plt.subplots(figsize=(8, 8))[1]
ax.plot(plot_idx,
        boost_boston.train_score_,
        'b',
        label='Training')
ax.plot(plot_idx,
        test_error,
        'r',
        label='Test')
ax.legend()
plt.show()
```



我们来看看测试误差：

```
#查看测试误差
y_hat_boost = boost_boston.predict(X_test)
print(np.mean((y_test - y_hat_boost)**2))
```

更改学习率，再次查看测试误差。结果表明，在这个数据集中，这两种学习率对模型的差距不大：

```
#修改学习率，再次查看测试误差
boost_boston = GBR(n_estimators=5000,
                   learning_rate=0.2,
                   max_depth=3,
                   random_state=0)
boost_boston.fit(X_train, y_train)
```

```
y_hat_boost = boost_boston.predict(X_test)
print(np.mean((y_test - y_hat_boost)**2))
```

8.3.5 贝叶斯加性回归树

`ISLP.bart`是专门用于定量变量的BART模型包。此外，其他的模型如分类结果的逻辑和概率拟合没有被讨论。下面拟合一个BART模型，并显示测试误差：

```
#拟合BART
bart_boston = BART(random_state=0, burnin=5, ndraw=15)
bart_boston.fit(X_train, y_train)

#显示测试误差
yhat_test = bart_boston.predict(X_test.astype(np.float32))
print(np.mean((yhat_test - y_test)**2))
```

为了解释BART模型，我们查看每个特征在树中被提到的平均概率：

```
#检查每个变量在树的集合中出现的次数
var_inclusion = pd.Series(bart_boston.variable_inclusion_.mean(0),
                          index=D.columns)
print(var_inclusion)
```

#ML
#CS