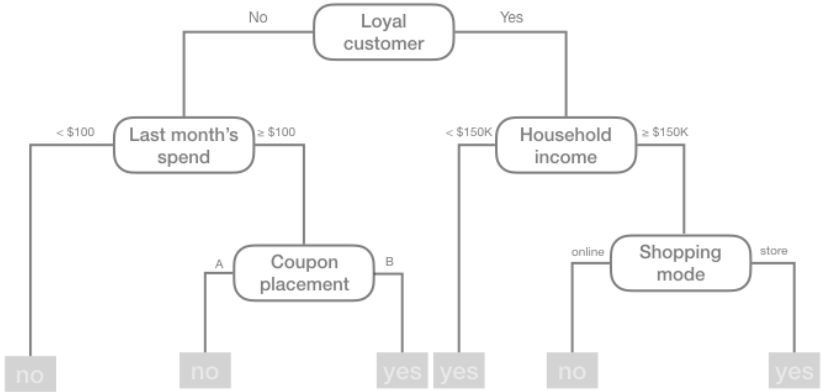
Lecture8
决策树与集成学习

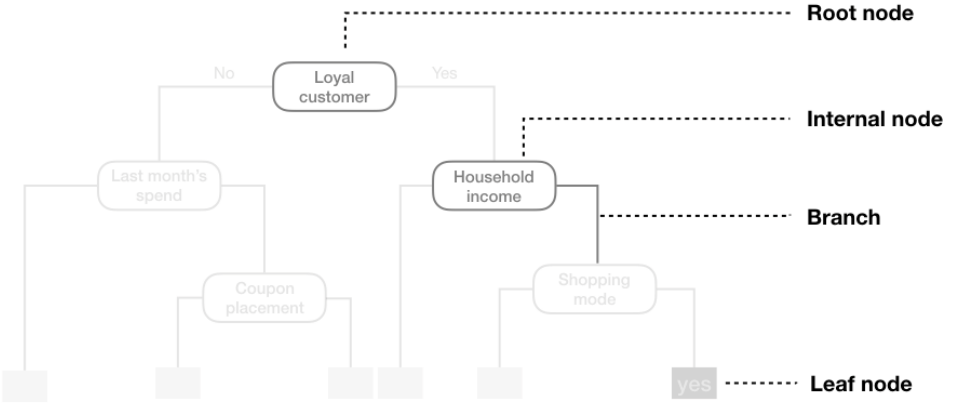
刘晨辉
邮箱：chenhuiliu@hnu.edu.cn
办公室：土木楼A422
2023.04.18

目录

1. 决策树(Decision Trees)

决策树是一种非参数(Non-parametric)算法。基于一定的规则，将数据集分割成更小的子集，使得其响应变量值尽可能一致，且认为各子集的响应变量值是固定的.
最常见的决策树构建策略为Breiman提出的CART（Classification and Regression Tree）方法，binary classification。决策树模型预测结果基于：
（1）回归（Regression）：每个子集响应变量的平均值
（2）分类（Classification）：占据主导地位的类别(majority class)。



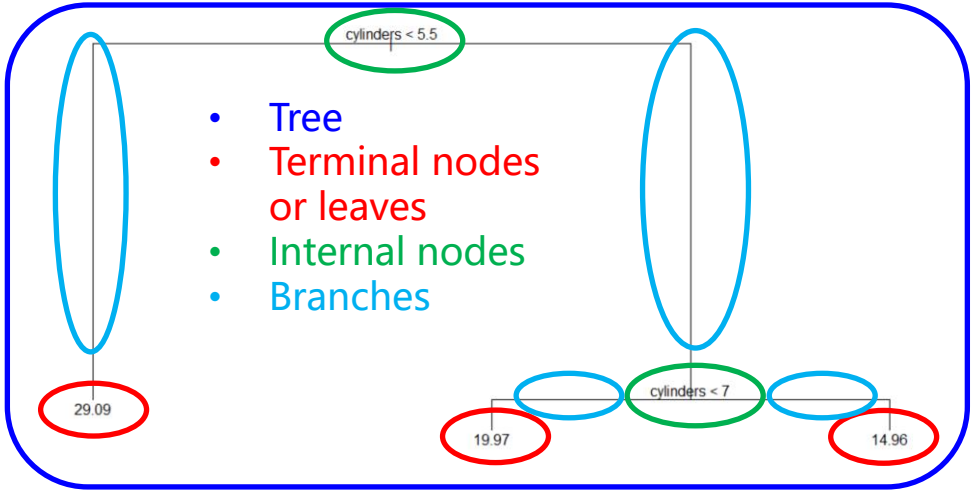


基于顾客忠诚度，家庭收入，上个月花费，优惠券类型和购物模式来预测顾客是否会使用购物券

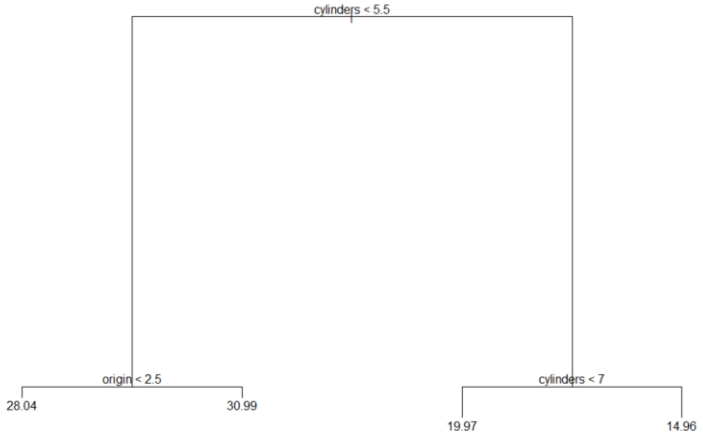
1.1 回归树(Regression Trees)

利用Auto数据集，利用车辆的cylinders（气缸数）来预测车辆的mpg（油耗指标）。

```
### Auto
tree1<- tree(mpg~cylinders,data = Auto)
plot(tree1)
text(tree1)
```




```
tree2<- tree(mpg~cylinders+origin,data = Auto)
plot(tree2)
text(tree2)
```



1.1 回归树(Regression Trees)

回归树步骤:

a. 将预测变量空间，也就是X_1， X_2，…， X_p所有可能的值，划分为J个不同且不重合的区域R_1，R_2，…，R_J。

b. 对于落在区域R_j的每个观测数据，我们对其响应值的预测值都是相同的，均为所有落在区域R_j的训练数据的响应值的平均值。

问题：如何来分割得到各区域?

方法：选择能够最小化残差平方和(RSS, residual sum of squares)的区域分割形式。

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

式中：$\hat{y}_{R_j}$是第j个区域训练数据响应值的平均值。

1.1 回归树(Regression Trees)

问题：所有变量空间可能有无限个组合来分割为J个区域，几乎不可能完成所有计算。

解决方案：我们采用一种由上往下(top-down)的贪婪(greedy)方法 - 递归二元分类(Recursive binary splitting)。

- 从树顶开始，依次分割预测变量空间，直到树底。
- 每次分割，将节点分割为两个树。
- 每次分割，选择对当前步长最优的分割方式(greedy)。

1.1 回归树(Regression Trees)

递归二元分类(Recursive binary splitting)

- **定义**:

对于一个预测变量X_j，选择分割点(Cutpoint)s将预测变量空间分割成两个区域$R_1(j,s) = \{X|X_j < s\}$和$R_2(j,s) = \{X|X_j > s\}$，能够最小化残差平方和。

- **步骤**:

（1）考虑所有的预测变量X_1，X_2，…，X_p和每个预测变量所有可能的分割点，选择能够得到最小化残差平方和的分割形式。

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

式中：$\hat{y}_{R_1}$是落在$R_1(j,s)$区域的训练数据响应值的平均值；$\hat{y}_{R_2}$是落在$R_2(j,s)$区域的训练数据响应值的平均值。

寻找j和s的过程可以很快完成，特别是当预测变量数目p有限的情况。

（2）针对两个区域，选择能够进一步最小化RSS的预测变量和分割点。分割完毕后，决策树有三个区域。

（3）重复上述过程，直到满足某个分割停止的标准，比如RSS减少的门槛值，或者是每个区域的数据个数。

（4）对于检验数据，我们用落在对应区间的训练数据响应值的平均值作为预测值。

1.1 回归树(Regression Trees)

对于产生的决策树，当过于复杂时，可能会产生过拟合的情况，即高方差，低偏差；当过于简单时，可能会产生预测不够精准的情况，即低方差，高偏差。

策略：首先建立一个很大的决策树T_0，然后逐步修剪(pruning)直到得到一个子树(subtree)。

- 可以采用交叉验证(CV)或者验证集法，选择能够最小化test error rate的子树，但是这种放过于繁琐，因为可能有无数子树选择。
- 成本复杂度修剪(Cost complexity pruning, weakest link pruning)：只考虑一系列由非负调节参数α索引的子树。
  对于每一个调节参数α，存在一个子树$T \subseteq T_0$，使得最小化

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

式中：$|T|$是树T的终端节点的数目，R_m是对应于第m个终端节点的分割区，$\hat{y}_{R_m}$是分割区R_m内训练数据响应值的平均值，α用来控制平衡子树的复杂度与训练数据预测精确度。

α可以通过采用交叉验证或者验证集法得到。

1.1 回归树(Regression Trees)

回归树修剪
(Tree Pruning)

Algorithm 8.1 *Building a Regression Tree*

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.

   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results for each value of α, and pick α to minimize the average error.

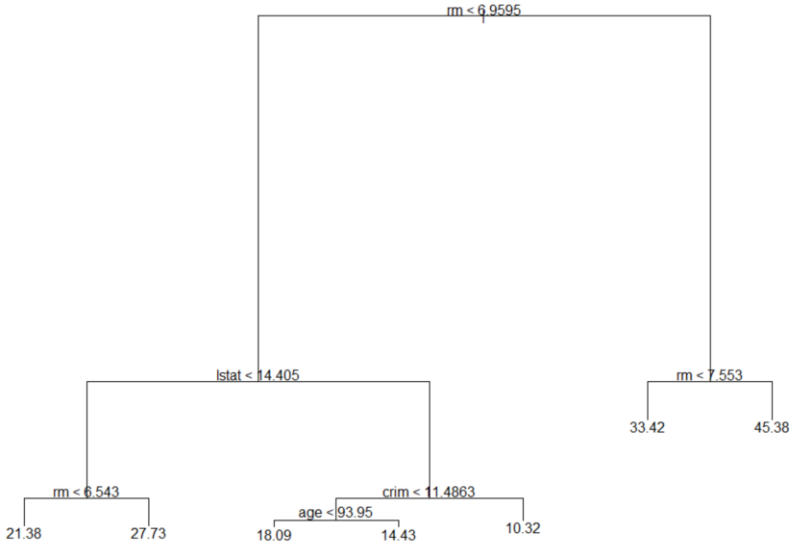
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

1.1 回归树(Regression Trees)

回归树分析案例：Boston数据集

```
## 建立决策树模型
library(tree)
set.seed(1)
train<- sample(1:nrow(Boston),nrow(Boston)/2)
tree_boston<- tree(medv~.,Boston,subset = train)
summary(tree_boston)
## 画出决策树
plot(tree_boston)
text(tree_boston,pretty = 0)
```

```
> summary(tree_boston)

Regression tree:
tree(formula = medv ~ ., data = Boston, subset = train)
Variables actually used in tree construction:
[1] "rm"    "lstat" "crim"  "age"
Number of terminal nodes:  7
Residual mean deviance:  10.38 = 2555 / 246
Distribution of residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

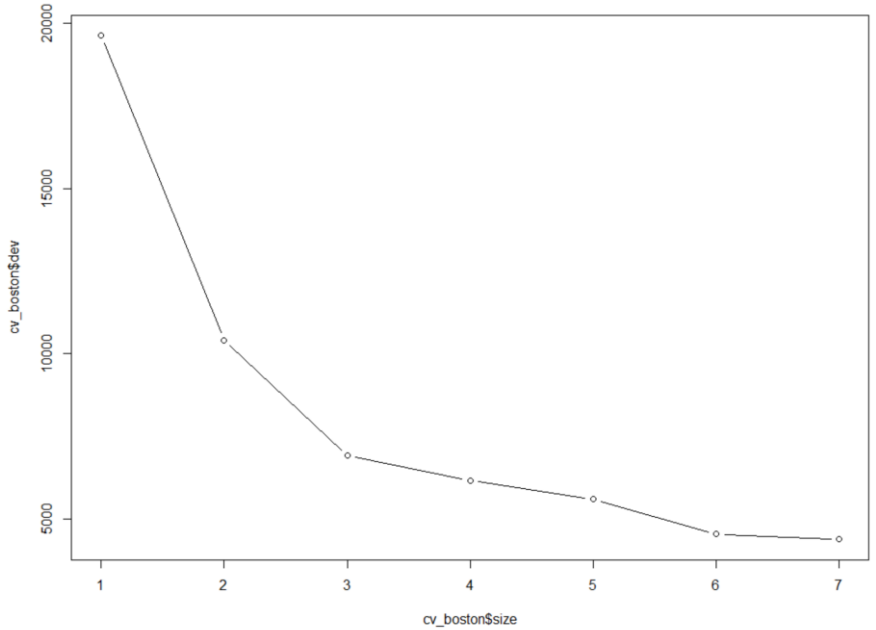


对于回归树：deviance就是RSS。

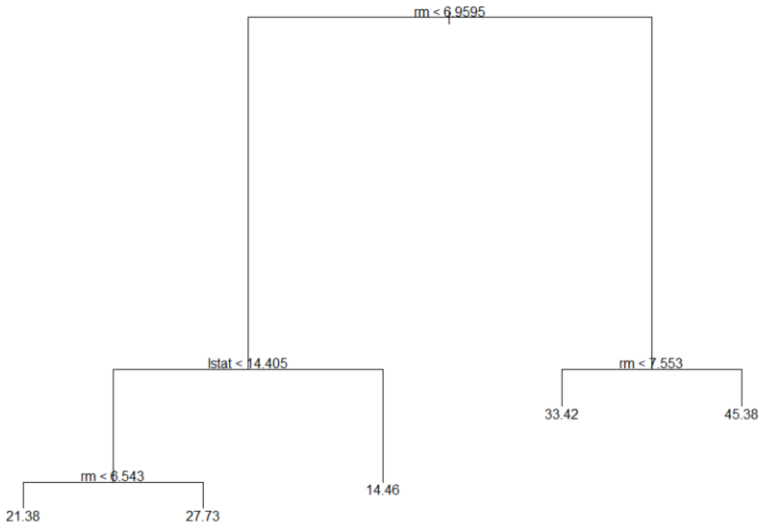
1.1 回归树(Regression Trees)

回归树分析案例：Boston数据集

```
## 决策树修剪评价
cv_boston<- cv.tree(tree_boston)
plot(cv_boston$size,cv_boston$dev,type="b")
```




```
## 修剪决策树
prune_boston<- prune.tree(tree_boston,best = 5)
plot(prune_boston)
text(prune_boston,pretty=0)
```

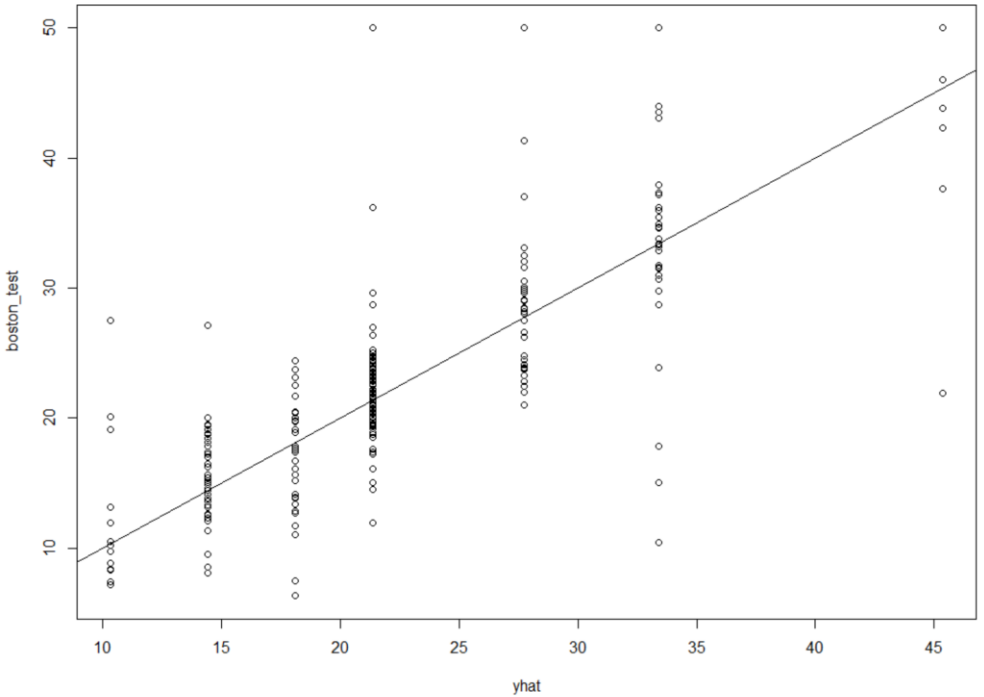


1.1 回归树(Regression Trees)

回归树分析案例：Boston数据集

```
## 决策树预测
yhat<- predict(tree_boston,newdata = Boston[-train,])
boston_test<- Boston[-train,"medv"]
plot(yhat,boston_test)
abline(0,1)
mean((yhat-boston_test)^2)
```

```
> mean((yhat-boston_test)^2)
[1] 35.28688
```



1.2 分类树(Classification Trees)

分类树与回归树类似，只是它预测的是分类变量。对于落在某个区域的数据，我们预测它的响应变量值为该区域内最常发生的类别。

分割标准：对于分类树，RSS不再适用。

- 分类错误率(Classification error rate)：指某个区域的训练数据中，不属于最常见类别的数据比例。

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

式中：$\hat{p}_{mk}$是指落在第m个区域的训练数据中，来自于第k类的数据比例。

分类错误对于决策树生长并不敏感，因此实际中我们常用另外两个指标。

- 基尼指数(Gini index)：测量K个类别的总方差，用来测量节点的纯度(Purity)。值越小，代表纯度越高，也就是所有观测值来自于单一类别。

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

- 熵(Entropy, information)：$D = -\sum_{k=1}^{K} \hat{p}_{mk} log(\hat{p}_{mk})$

由于$0 < \hat{p}_{mk} < 1$，所以$0 \leq -\hat{p}_{mk} log(\hat{p}_{mk})$。与基尼指数类似，熵值越小，代表节点纯度越高。

1.2 分类树(Classification Trees)

分类树案例：Carseats数据集

```
library(tree)
library(ISLR2)

## 创造新变量：必须转换成factor变量
Carseats$High<-
factor(ifelse(Carseats$Sales<=8,"No","Yes"))
## 建立分类树模型
tree_Carseats<- tree(High~.-Sales,Carseats)
print(summary(tree_Carseats))
```

```
> print(summary(tree_Carseats))

Classification tree:
tree(formula = High ~ . - Sales, data = Carseats)
Variables actually used in tree construction:
[1] "ShelveLoc"   "Price"       "Income"
[4] "CompPrice"   "Population"  "Advertising"
[7] "Age"         "US"
Number of terminal nodes:  27
Residual mean deviance:  0.4575 = 170.7 / 373
Misclassification error rate: 0.09 = 36 / 400
```

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

式中n_{mk}是第m个终端节点中属于第k类的数据个数，$log\hat{p}_{mk}$是第m个终端节点中属于第k类的数据比例。

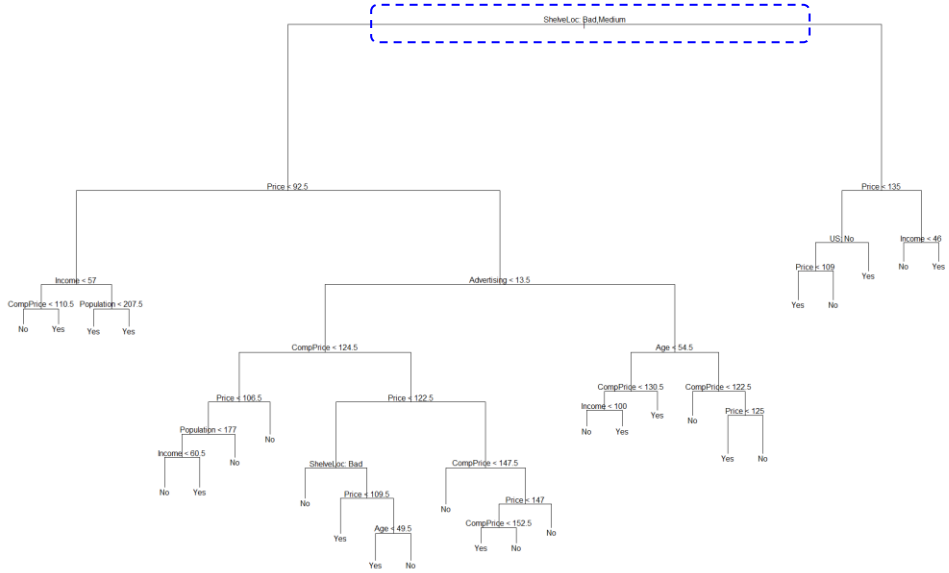
Deviance越小，表明模型拟合越好。

$Residual\ mean\ deviance\ =\ deviance\ /\ (n - |T_0|)$，$|T_0|$是终端节点个数。本例中，$|T_0|$=27

1.2 分类树(Classification Tree

分类树案例：Carseats数据集

```
### 展示决策树
print(tree_Carseats)
### 画出决策树
plot(tree_Carseats)
text(tree_Carseats,pretty = 0)
```




```
> print(tree_Carseats)
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 400 541.500 No ( 0.59000 0.41000 )
   2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
     4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
       8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
        16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
        17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
       9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
        18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
        19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
     5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
      10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
        20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
          40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
            80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
             160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
             161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
            81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
          41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
        21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
          42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
            84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
            85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
             170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
             171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
               342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
               343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
          43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
            86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
            87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
             174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
               348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
               349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
             175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
      11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
        22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
          44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
            88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
            89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
          45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
        23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
          46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
          47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
            94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
            95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
   3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
     6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
      12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
        24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
        25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
      13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
     7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
      14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
      15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

1.2 分类树(Classification Trees)

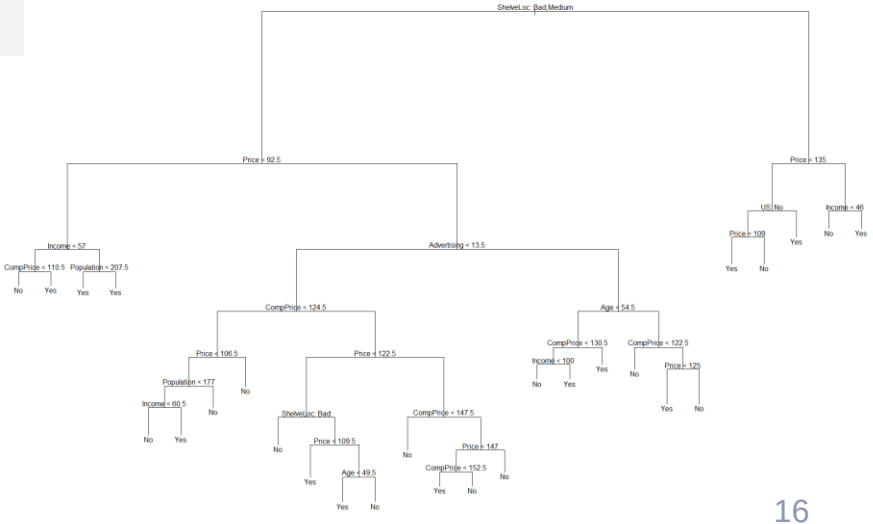
分类树案例：Carseats数据集

```
### 3.3 决策树模型评价
set.seed(2)
# 分割训练和检验数据集
train<- sample(1:nrow(Carseats),200)
train_Carseats<- Carseats[train,]
test_Carseats<- Carseats[-train,]
# 建立决策树模型
tree_Carseats<- tree(High~. - Sales, data = train_Carseats)
print(summary(tree_Carseats))

# 预测检验数据集
test_Carseats$pred_tree<- predict(tree_Carseats,newdata = test_Carseats,type = "class")
# 计算混淆矩阵
table(test_Carseats$pred_tree,test_Carseats$High)
# 计算准确率
print(sum(test_Carseats$pred_tree == test_Carseats$High)/nrow(test_Carseats))
```

```
> table(test_Carseats$pred_tree,test_Carseats$High)

      No Yes
  No  104  33
  Yes  13  50
> # 计算准确率
> print(sum(test_Carseats$pred_tree == test_Carseats$High)/
[1] 0.77
```

对于训练数据集，准确率是多少？
比检验数据集大吗？

```
> summary(tree_Carseats)

Classification tree:
tree(formula = High ~ . - Sales, data = train_Carseats)
Variables actually used in tree construction:
[1] "Price"       "Population"  "ShelveLoc"   "Age"
[8] "Income"      "US"
Number of terminal nodes:  21
Residual mean deviance:  0.5543 = 99.22 / 179
Misclassification error rate: 0.115 = 23 / 200
```



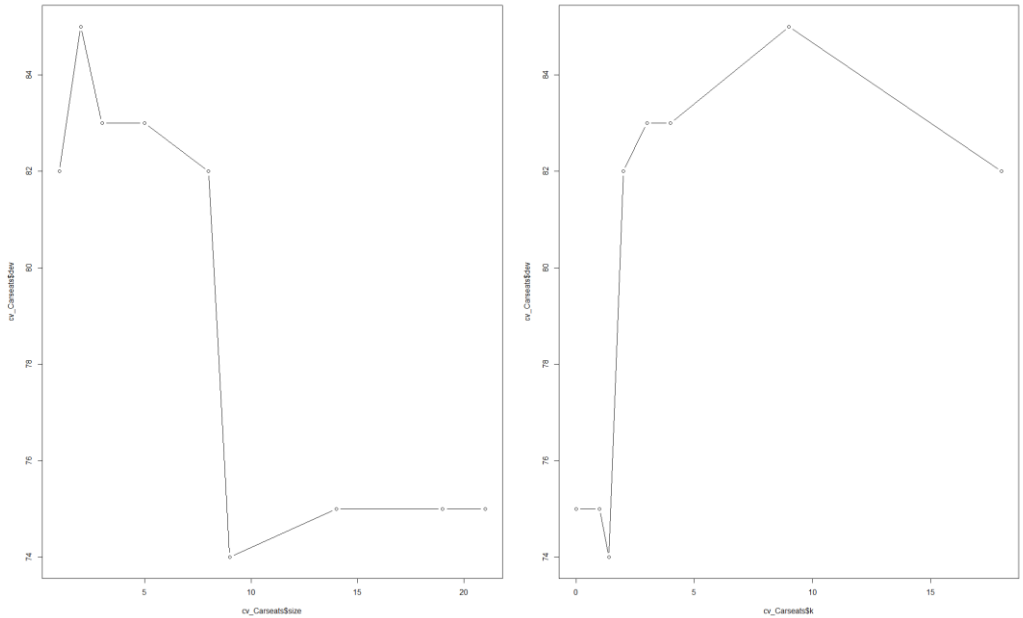
1.2 分类树(Classification Trees)

分类树案例：Carseats数据集

```
### 3.4 决策树修剪
set.seed(7)
# 基于misclassification error来进行交叉验证和决策树修剪
cv_Carseats<- cv.tree(tree_Carseats,FUN = prune.misclass)
print(cv_Carseats)

par(mfrow=c(1,2))
plot(cv_Carseats$size,cv_Carseats$dev,type = "b")
plot(cv_Carseats$k,cv_Carseats$dev,type = "b")
par(mfrow=c(1,1))
```

```
> print(cv_Carseats)
$size
[1] 21 19 14  9  8  5  3  2  1

$dev
[1] 75 75 75 74 82 83 83 85 82

$k
[1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0

$method
[1] "misclass"

attr(,"class")
[1] "prune"         "tree.sequence"
```

The tree with 9 terminal nodes results in only 74 cv errors.



1.2 分类树(Classification Trees)

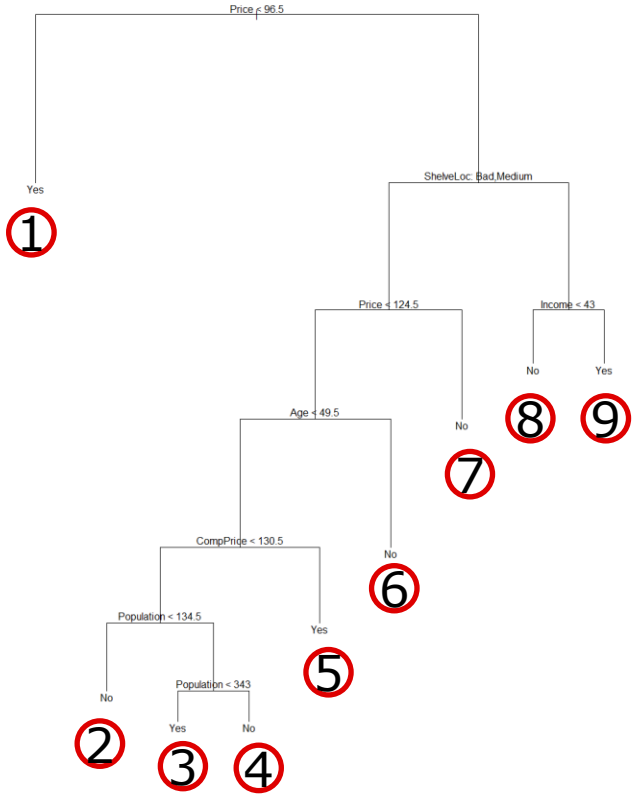
分类树案例：Carseats数据集

```
# 有9个节点的模型表现最优，因此修剪得到有9个终端节点的决策树
prune_Carseats<- prune.misclass(tree_Carseats,best = 9)
# 画出修剪后的决策树
plot(prune_Carseats)
text(prune_Carseats,pretty=0)
```

```
# 修剪决策树评价
# 预测检验数据集
test_Carseats$pred_tree_prune<- predict(prune_Carseats,
                                        newdata = test_Carseats,
                                        type = "class")
# 计算混淆矩阵
table(test_Carseats$pred_tree_prune,test_Carseats$High)
# 计算准确率
print(prop.table(table(test_Carseats$pred_tree_prune,test_Carseats$High)))
```

```
> # 计算混淆矩阵
> table(test_Carseats$pred_tree_prune,test_Carseats$High)

     No Yes
  No  97  25
  Yes 20  58
> # 计算准确率
> print(prop.table(table(test_Carseats$pred_tree_prune,test_Carseats$High)))

        No   Yes
  No  0.485 0.125
  Yes 0.100 0.290
```

准确率为0.485+0.290=0.775，比修剪前0.77略有提高!



1.3 决策树与线性模型对比

线性模型：$y = \beta_0 + \sum_{j=1}^{P} X_j \beta_j$

回归树：$y = \sum_{m=1}^{M} c_m * 1_{(X \in R_m)}$

式中：R_1,..., R_m表示各个分割的特征空间。

选择哪个模型取决于数据。

优缺点

- ▲ Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- ▲ Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous chapters.
- ▲ Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small).
- ▲ Trees can easily handle qualitative predictors without the need to create dummy variables.
- ▼ Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches seen in this book.
- ▼ Additionally, trees can be very non-robust. In other words, a small change in the data can cause a large change in the final estimated tree.

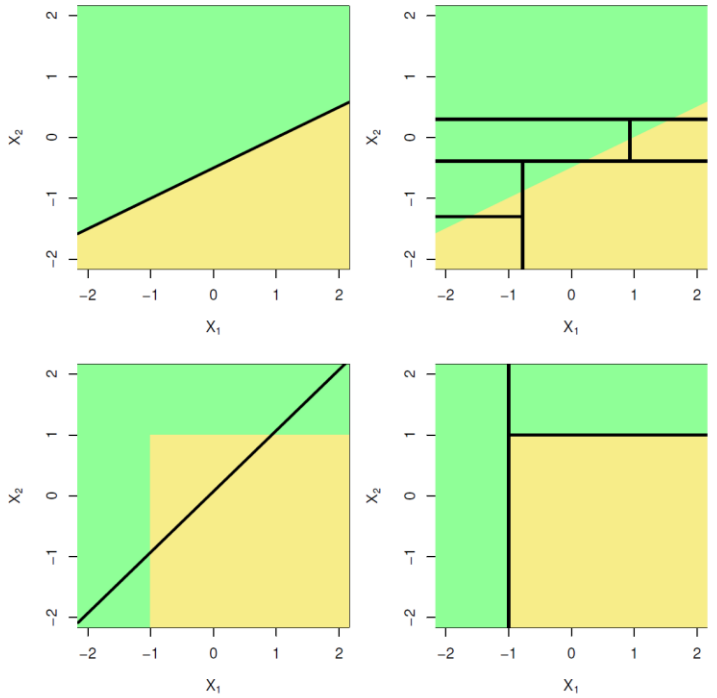


FIGURE 8.7. Top Row: *A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right).* Bottom Row: *Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right).*

2. 集成学习(Ensemble Learning)

所谓集成学习就是将很多简单的模型，合并组成一个更强大的模型。
每个简单的模型也称为弱学习器(Weak learner)。

我们主要学习三种集成学习方法：

- Bagging
- Random forest：随机森林
- Boosting

本部分中，每个简单的模型是一个回归树或者分类树。

2.1 Bagging

决策树的一个缺点就是方差很大，因为其是随机选择分隔点。我们希望模型能够有较小的方差，也就是预测结果比较稳定。

Bootstrap aggregation，又称为bagging，是一种常用的减少统计方法方差的方法。

假设我们有n个独立的观测数据$Z_1,\ldots, Z_n$，每个观测数据的方差为σ^2，那么

$$Var(\bar{z}) = Var\left(\frac{\sum_n Z_i}{n}\right) = \frac{\sigma^2}{n}$$

也就是说，对一系列数据求平均值可以减少方差。

因此，如果从总体中提取多个训练数据集，对每个训练数据集建立一个预测模型并预测检验数据集，然后计算这些模型预测结果的平均值，可以减少预测结果的方差。

2.1 Bagging

假设我们利用B个不一样的训练数据集，建立了B个预测模型$\hat{f}^1(x), \hat{f}^2(x), \ldots, \hat{f}^B(x)$，那么多这些模型求平均值可以得到一个方差更小的模型。

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

由于实际情况中，我们不太可能会获得多个训练数据集，可采用bootstrap得到B个抽样训练数据集。针对每个抽样训练数据集，建立相应的预测模型$\hat{f}^{*b}(x)$。

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

这种方法就称为bagging。一般每个建立的决策树模型都是没有修剪得，事实证明bagging可以有效减小预测结果得方差。

对于分类预测，我们则用多数票决(Majority Vote)法来作为总体预测结果，也就是B个预测结果中出现最多的类别作为总体预测结果。

2.1 Bagging

（1）包外误差估计(Out-of-Bag Error Estimation)

对于bagging模型来说，无需进行交叉验证或者用验证集法来估计其检验误差(test error)。我们在第七次课已经学过，平均来说，每个bagged树利用了大约2/3的数据来训练建模，剩余1/3的数据被称为包外(Out-of-Bag, OOB)数据。因此，

- 对于第i个观测数据，大约有$B/3$个bagged树没有利用该数据，利用这些bagged树对该数据进行预测，产生$B/3$个预测值。
- 对于第i个观测数据，可以计算这些预测值的平均值（对于回归树）或者多数投票值（对于分类树），作为最终预测结果。
- 在计算出来每个观测数据的OOB预测结果之后，计算所有数据的MSE（回归）或分类错误（分类），这个值即是bagged树模型检验误差的有效估计。

当数据集较大，完成交叉验证计算比较繁琐时，OOB是一个非常方便的用来估计检验误差的方法。

2.1 Bagging

（2）变量重要性评价(Variable Importance Measures)

与单个决策树相比，bagging虽然可以有效提高预测精度，但其解释能力较差。因此，bagging模型提高预测精度的代价就是解释能力下降。

尽管如此，我们依然可以用残差平方和（RSS，回归树）或者基尼指数（Gini index，分类树）来评价预测变量的重要性。

对于bagged回归树：对于任一变量，计算由于将其作为分隔值减少的RSS，然后求所有B个RSS的平均值。值越大，表示变量越重要。

对于bagged分类树：对于任一变量，计算由于将其作为分隔值减少的Gini index，然后求所有B个Gini index的平均值。值越大，表示变量越重要。

2.1 Bagging

Bagging例子1：Boston数据集

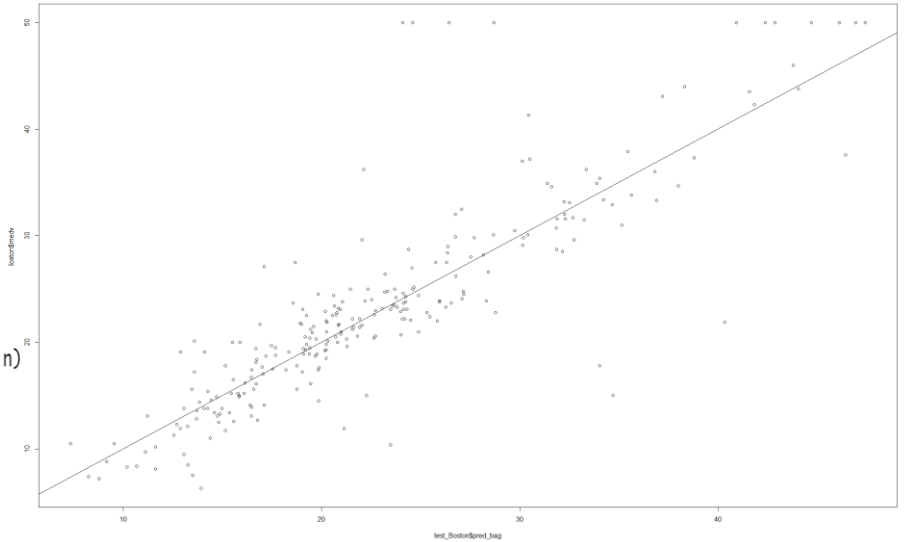
```
###### 1. Bagging and Random forest ########
### 注意：bagging是random forest的一个特殊案例，也就是m = p。
library(randomForest)

### 1.1 数据准备
# 设置随机种子
set.seed(1)
# 创造训练数据集和检验数据集
train<- sample(1:nrow(Boston),nrow(Boston)/2)
train_Boston<- Boston[train,]
test_Boston<- Boston[-train,]

### 1.2 Bagging1
bag_Boston<- randomForest(medv ~ .,
                          data = Boston,
                          subset = train,
                          mtry = 12,
                          importance = TRUE)
print(bag_Boston)
```

```
> print(bag_Boston)

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      subset = train)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 12

          Mean of squared residuals: 11.19605
                    % Var explained: 85.43
```

```
test_Boston$pred_bag<- predict(bag_Boston,
                               newdata = test_Boston)
plot(test_Boston$pred_bag,test_Boston$medv)
abline(0,1)
print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
```

```
> print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
[1] 23.31685
```

注意：你的结果与我的可能有所不同，这个与R和randomForest工具包的版本有关。



2.1 Bagging

Bagging例子1：Boston数据集

```
### 1.2 Bagging1
bag_Boston<- randomForest(medv ~ .,
                          data = Boston,
                          subset = train,
                          mtry = 12,
                          importance = TRUE)
print(bag_Boston)

test_Boston$pred_bag<- predict(bag_Boston,
                               newdata = test_Boston)
plot(test_Boston$pred_bag,test_Boston$medv)
abline(0,1)
print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
```

VS

```
### 1.3 Bagging2
## 设置tree的个数为25个
bag_Boston<- randomForest(medv ~ .,
                          data = Boston,
                          subset = train,
                          mtry = 12,
                          importance = TRUE,
                          ntree = 25)
print(bag_Boston)

test_Boston$pred_bag<- predict(bag_Boston,
                               newdata = test_Boston)
plot(test_Boston$pred_bag,test_Boston$medv)
abline(0,1)
print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
```

```
> print(bag_Boston)

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      subset = train)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 12

          Mean of squared residuals: 11.19605
                    % Var explained: 85.43
```

```
> print(bag_Boston)

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      ntree = 25, subset = train)

               Type of random forest: regression
                     Number of trees: 25
No. of variables tried at each split: 12

          Mean of squared residuals: 11.05672
                    % Var explained: 85.62
```

2.1 Bagging

Bagging例子2：Carseats数据集

```
### 1.3 Bagging1: Carseats
# 设置随机种子
set.seed(2)
# Carseats: 创造训练数据集和检验数据集
Carseats$High<- factor(ifelse(Carseats$Sales<=8,"No","Yes"))
train2<- sample(1:nrow(Carseats),nrow(Carseats)/2)
train_Carseats<- Carseats[train2,]
test_Carseats<- Carseats[-train2,]
bag_Carseats<- randomForest(High ~ .-Sales,
                            data = Carseats,
                            subset = train2,
                            mtry = 10,
                            importance = TRUE)
print(bag_Carseats)

# 预测检验数据集
test_Carseats$pred_bag<- predict(bag_Carseats,
                                 newdata = test_Carseats)
# 计算Accuracy
print(table(test_Carseats$pred_bag,test_Carseats$High))
print(sum(test_Carseats$pred_bag == test_Carseats$High)/nrow(test_Carseats))
```

	No	Yes
No	104	24
Yes	13	59

2.2 随机森林(Random Forest)

与bagging相比，随机森林方法规定决策树在每次分割时，只能从所有p个预测变量中随机选取m个预测变量作为预测变量。典型情况下，$m \approx \sqrt{p}$。为什么?

假设数据集中有1个很强的预测变量，以及其他一些中等强度的预测变量。对于bagged树，可以预见大部分甚至所有决策树都会将这个很强的预测变量作为最高级的分割参数。所以，组成bagged树的所有单个决策树会非常相近，关联性较高。与关联性较弱的决策树的平均值相比，关联性很强的决策树的平均值并不会极大的减小预测结果方差。也就是与单个决策树相比，bagging并不能有效减少预测结果方差。

随机森林通过强制每次分割(split)只能考虑部分预测变量克服了这一问题。平均来说，有$(p-m)/p$个分割点不会考虑那个很强的预测变量，这就给了其他预测变量更多的机会。这个过程使得各个决策树关联性大大降低，因此再求解平均值可以提高最终结果的可靠性，减少其波动性。

当$m = p$时，随机森林就变成了bagging模型。当数据中有大量密切相关的预测变量时，选择较小的m可以有效提高random forest的预测效果。

2.2 随机森林(Random Forest)

RF例子1：Boston数据集

```
### 1.4 Random forest
## 设置相应变量个数为6
rf_Boston<- randomForest(medv ~ .,
                         data = Boston,
                         subset = train,
                         mtry = 6,
                         importance = TRUE)
print(rf_Boston)

test_Boston$pred_rf<- predict(rf_Boston,
                              newdata = test_Boston)
print(mean((test_Boston$pred_rf - test_Boston$medv)^2))
```

```
> print(rf_Boston)

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = 6, importance = TRUE,      subset = train)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 6

          Mean of squared residuals: 9.905574
                    % Var explained: 87.11
```

```
> print(mean((test_Boston$pred_rf - test_Boston$medv)^2))
[1] 19.84431
```

VS

```
### 1.2 Bagging1
bag_Boston<- randomForest(medv ~ .,
                          data = Boston,
                          subset = train,
                          mtry = 12,
                          importance = TRUE)
print(bag_Boston)

test_Boston$pred_bag<- predict(bag_Boston,
                               newdata = test_Boston)
plot(test_Boston$pred_bag,test_Boston$medv)
abline(0,1)
print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
```

```
> print(bag_Boston)

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      subset = train)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 12

          Mean of squared residuals: 11.19605
                    % Var explained: 85.43
```

```
> print(mean((test_Boston$pred_bag - test_Boston$medv)^2))
[1] 23.31685
```

从MSE来看，随机森林模型表现更优!

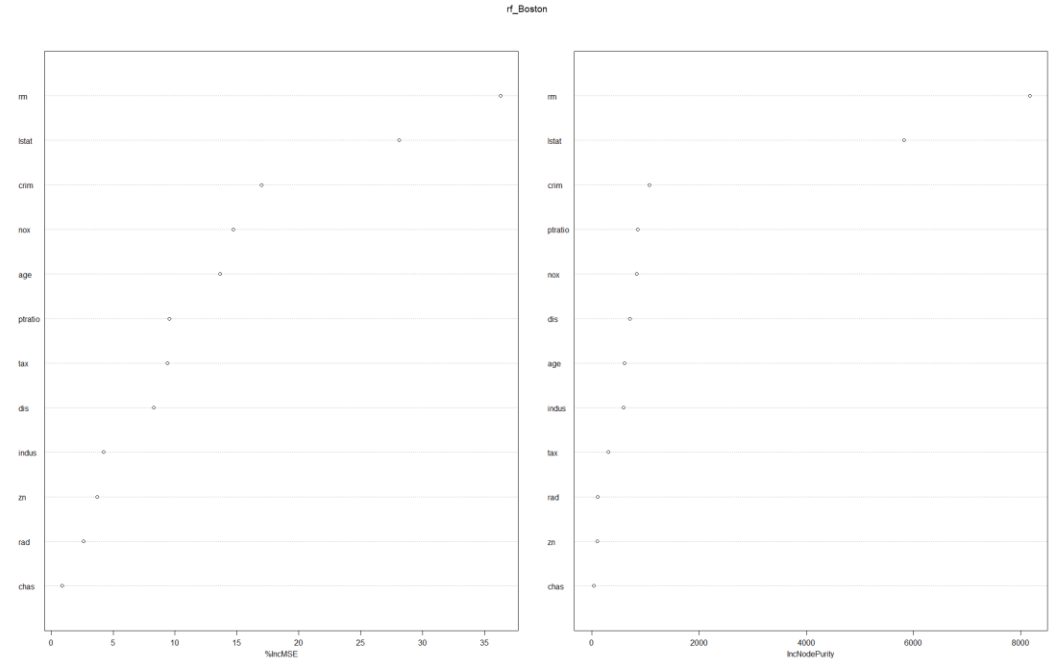
2.2 随机森林(Random Forest)

RF例子1：Boston数据集

```
## 检查变量重要性
importance(rf_Boston)
varImpPlot(rf_Boston)
```

这里面用来两个指标：
(1) The mean decrease of accuracy in predictions on the out of bag samples when a given variable is permuted.
(2) The total decrease in node impurity that results from splits over that variable, averaged over all trees. It is the training RSS for regression, and deviance for classification.

```
> importance(rf_Boston)
          %IncMSE IncNodePurity
crim    17.0128392    1072.65851
zn       3.7151491      97.75289
indus    4.2403982     591.41915
chas     0.8952874      33.61372
nox     14.7195570     838.36081
rm      36.3039145    8174.18981
age     13.6347005     607.51021
dis      8.2892814     712.40915
rad      2.6196531     105.31781
tax      9.4015933     310.25811
ptratio  9.5298854     859.88140
lstat   28.1228388    5822.10921
```



2.2 随机森林(Random Forest)

RF例子2：Carseats数据集

```
### 2.2 Random forest: Carseats
## 设置相应变量个数为6
rf_Carseats<- randomForest(High ~ .-Sales,
                           data = Carseats,
                           subset = train2,
                           mtry = 6,
                           importance = TRUE)
print(rf_Carseats)
```

```
> print(rf_Carseats)

Call:
 randomForest(formula = High ~ . - Sales, data = Carseats, mtry = 6
= train2)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 6

        OOB estimate of  error rate: 28.5%
Confusion matrix:
    No Yes class.error
No  97  22   0.1848739
Yes 35  46   0.4320988
```

```
# 计算Accuracy
test_Carseats$pred_rf<- predict(rf_Carseats,
                                newdata = test_Carseats)
print(table(test_Carseats$pred_rf,test_Carseats$High))
print(sum(test_Carseats$pred_rf == test_Carseats$High)/nrow(test_Carseats))

## 检查变量重要性
importance(rf_Carseats)
varImpPlot(rf_Carseats)
```

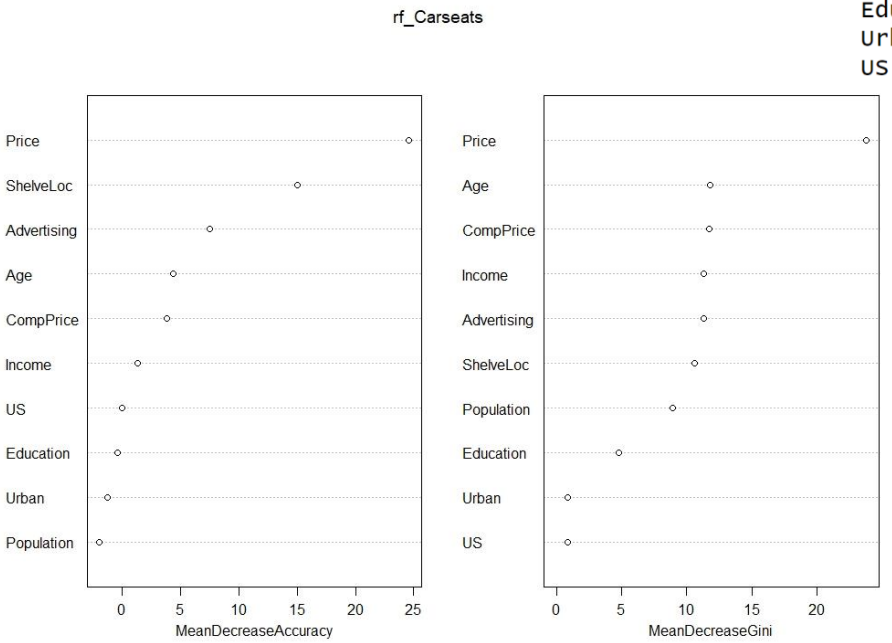
	No	Yes
No	105	21
Yes	12	62

2.2 随机森林(Random Forest)

RF例子2：Carseats数据集

```
## 检查变量重要性
importance(rf_Carseats)
varImpPlot(rf_Carseats)
```

```
> importance(rf_Carseats)
                    No         Yes MeanDecreaseAccuracy MeanDecreaseGini
CompPrice    2.4711533   3.2663255           3.82981188        11.7429632
Income       0.5435274   1.6793896           1.32390700        11.3265049
Advertising  3.8133779   6.9518682           7.54064812        11.3116124
Population  -2.2644767  -0.4915801          -1.94080097         8.8951307
Price       18.4756909  19.4364462          24.60900454        23.7932000
ShelveLoc   13.7385528   9.1975732          14.98171246        10.6367188
Age          3.9924030   2.6528449           4.38184461        11.8125788
Education    0.1512083  -0.7480913          -0.37126640         4.7982811
Urban       -1.3406141  -0.2855617          -1.26438377         0.8448764
US          -0.7221044   0.5946624          -0.02719634         0.8177936
```



2.4 Boosting

对于bagging模型，其特点是通过bootstrap抽出多个样本数据，对于每个样本数据单独进行建模，然后将多个建成的模型合并，创造一个最终的模型。在整个过程中，每个模型的建立过程是相互独立，同时进行的。

与bagging模型相比，boosting是顺序(sequentially)建模：每个决策树的构建都要受到之前已经所建决策树的影响。

Boosting并不会涉及到bootstrap抽样，相反，每个决策树都是对前一个数据集进行修改后建模得到。

对数据建立一个较大的决策树可能会出现过拟合(overfitting)的情况，相比之下，boosting方法学习的较慢。

2.4 Boosting

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.

2. For $b = 1, 2, \ldots, B$, repeat:

   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).

   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

- 基于现在的模型，我们对模型的残值（而非响应变量）构建新的决策树。
- 接着，把这个新的决策树加入到拟合函数中来更新残值。
- 对新的残值，构建新的决策树。

与bagging相比，boosting在缓慢提高模型表现。一般来说，学习较慢的统计学习方法表现会更好一些。

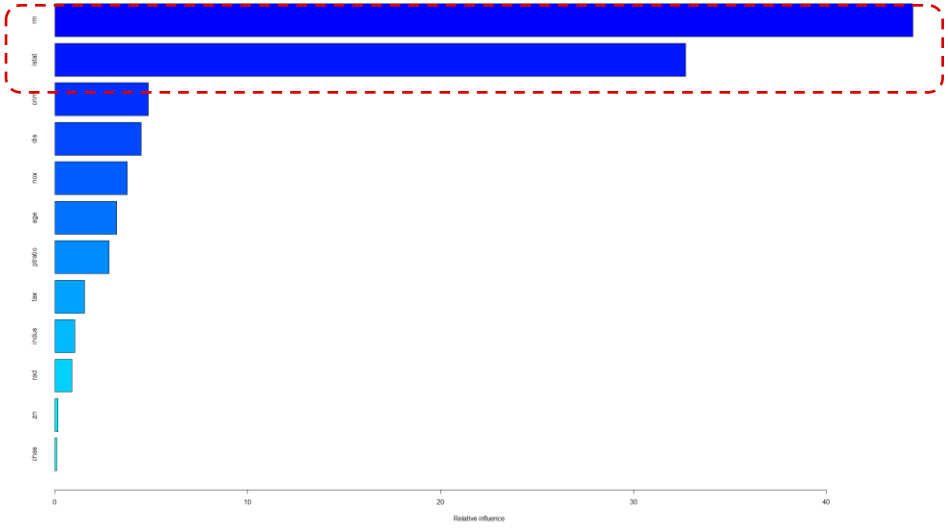
Boosting有三个调节参数(tuning parameters)：

- 决策树的数目B：与bagging和random forest不同，B过大可能会导致boosting过拟合。我们需要用交叉验证的方法来确定B值。
- 缩减参数λ：一个较小的整数，用来控制boosting的学习速率，典型值为0.01或0.001。比较小的λ值需要采用一个很大的B值来达到一个较好的表现。
- 每个决策树分割点数d：控制模型的复杂度。经常$d = 1$表现良好。也就是每个树是1个树桩(stump)，只有1次分割。这种情况下，the boosted ensemble就是一个相加模型(Additive model)，因此每次只涉及到1个预测变量。更一般的来说，d是交互深度(interaction depth)，因为d次分割最多涉及到d个变量。

2.4 Boosting

Boosting例子1：Boston数据集

```
###### 2. Boosting ########
library(gbm)
set.seed(1)
boost_Boston<- gbm(medv ~ .,
                   data = train_Boston,
                   distribution = "gaussian",
                   n.trees = 5000,
                   interaction.depth = 4)
print(boost_Boston)
print(summary(boost_Boston))
```




```
> print(summary(boost_Boston))
             var     rel.inf
rm           rm 44.48249588
lstat     lstat 32.70281223
crim       crim  4.85109954
dis         dis  4.48693083
nox         nox  3.75222394
age         age  3.19769210
ptratio ptratio  2.81354826
tax         tax  1.54417603
indus     indus  1.03384666
rad         rad  0.87625748
zn           zn  0.16220479
chas       chas  0.09671228
```

```
## 预测检验数据集
test_Boston$pred_boost<- predict(boost_Boston,
                                 newdata = test_Boston,
                                 n.trees = 5000)
print(mean((test_Boston$pred_boost - test_Boston$medv)^2))
```

```
> print(mean((test_Boston$pred_boost - test_Boston$medv)^2))
[1] 18.39057
```

2.4 Boosting

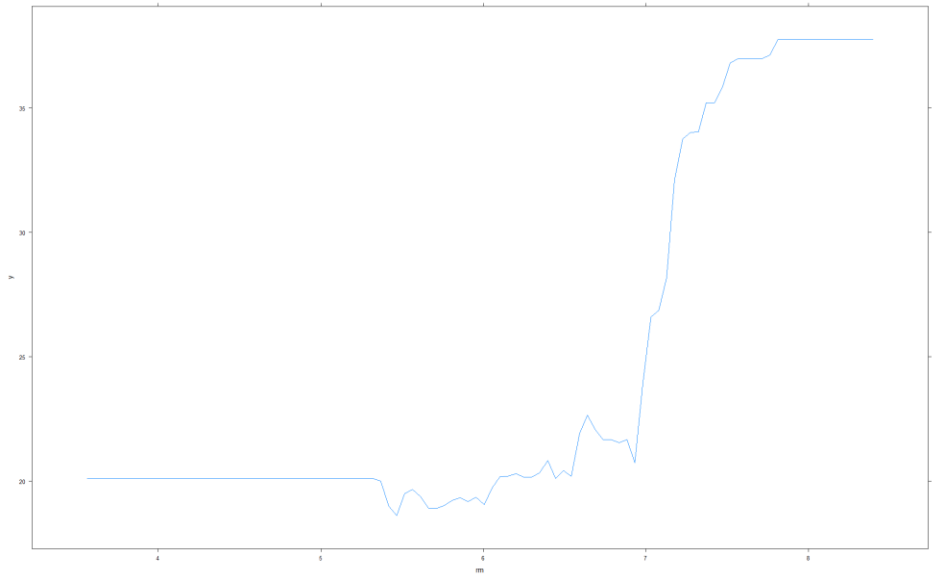
Boosting例子1：Boston数据集

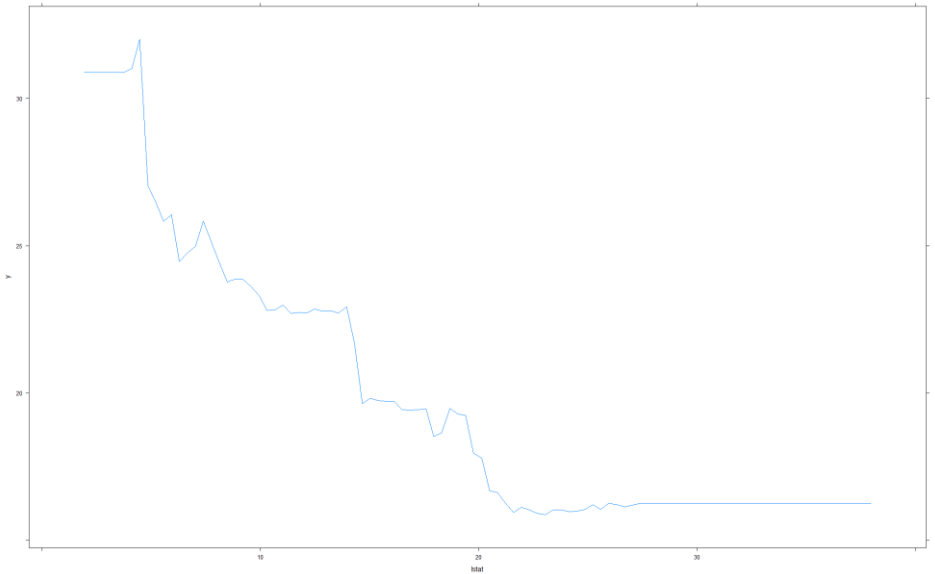
Partial dependence plot

```
## 检查partial dependence plots
plot(boost_Boston,i="rm")
plot(boost_Boston,i="lstat")
```

排除其他因素之后，选定变量对于相应变量的边际效应：

- 随着rm增加，medv增大
- 随着lstat增大，medv减小





2.4 Boosting

Boosting例子1：Boston数据集

```
### 变换参数: lambda from the default 0.001 to 0.1
boost_Boston<- gbm(medv ~ .,
                   data = train_Boston,
                   distribution = "gaussian",
                   n.trees = 5000,
                   interaction.depth = 4,
                   shrinkage = 0.1,
                   verbose = F)
# 创造一列新数据: pred_boost2
test_Boston$pred_boost2<- predict(boost_Boston,
                                  newdata = test_Boston,
                                  n.trees = 5000)
print(mean((test_Boston$pred_boost2 - test_Boston$medv)^2))
```

```
> print(mean((test_Boston$pred_boost2 - test_Boston$medv)^2))
[1] 17.9821
```

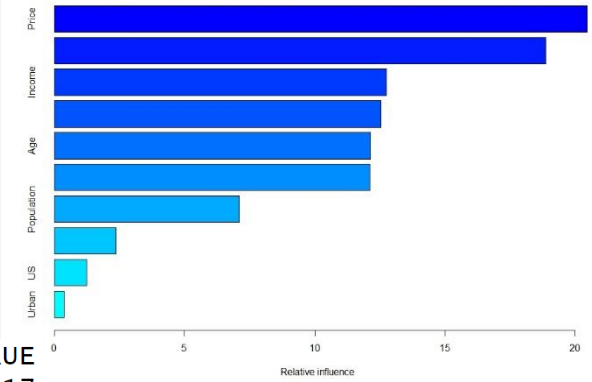
MSE略有下降。

2.4 Boosting

Boosting例子2：Carseats数据集

```
### 3.2 Boost: Carseats
### gbm requires the response variable to be in {0,1}.
### 因此需要把High转换成0和1
train_Carseats$High<- train_Carseats$Sales>8
test_Carseats$High<- test_Carseats$Sales>8
boost_Carseats<- gbm(High ~ .- Sales,
                     data = train_Carseats,
                     distribution = "bernoulli",
                     n.trees = 100,
                     interaction.depth = 4)
print(boost_Carseats)
print(summary(boost_Carseats))

## 检查partial dependence plots
plot(boost_Carseats,i="Price")
plot(boost_Carseats,i="ShelveLoc")

# 预测检验数据集. Predict returns the probability
test_Carseats$pred_boost<- predict(boost_Carseats,
                                   newdata = test_Carseats,
                                   type = "response")>0.5
# 计算Accuracy
print(table(test_Carseats$pred_boost,test_Carseats$High))
print(sum(test_Carseats$pred_boost == test_Carseats$High)/nrow(test_Carseats))
```

```
> print(summary(boost_Carseats))
                       var    rel.inf
Price                Price 20.4697000
ShelveLoc        ShelveLoc 18.8598872
Income              Income 12.7481333
Advertising  Advertising 12.5436220
Age                    Age 12.1527191
CompPrice        CompPrice 12.1279561
Population      Population  7.1105938
Education        Education  2.3707063
US                      US  1.2331688
Urban                Urban  0.3835134
```

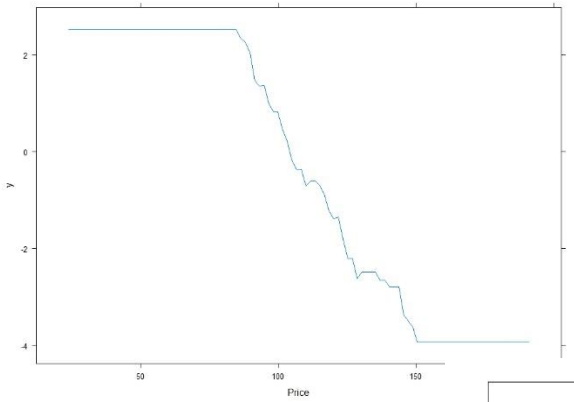


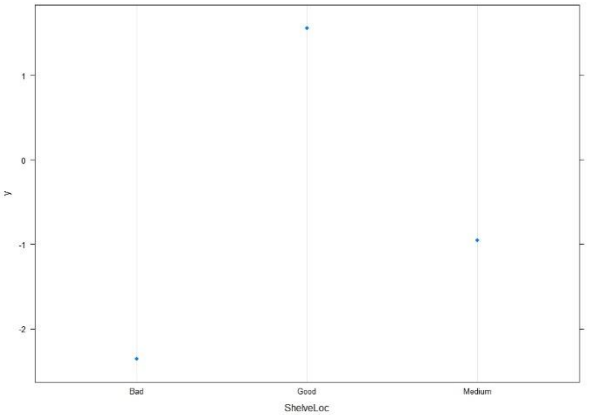

```
       FALSE TRUE
FALSE    108   17
TRUE       9   66
```

2.4 Boosting

Boosting例子2：Carseats数据集

```
### 3.2 Boost: Carseats
### gbm requires the response variable to be in
### 因此需要把High转换成0和1
train_Carseats$High<- train_Carseats$Sales>8
test_Carseats$High<- test_Carseats$Sales>8
boost_Carseats<- gbm(High ~ .- Sales,
                     data = train_Carseats,
                     distribution = "bernoulli",
                     n.trees = 100,
                     interaction.depth = 4)
print(boost_Carseats)
print(summary(boost_Carseats))

## 检查partial dependence plots
plot(boost_Carseats,i="Price")
plot(boost_Carseats,i="ShelveLoc")

# 预测检验数据集. Predict returns the probability
test_Carseats$pred_boost<- predict(boost_Carseats,
                                   newdata = test_Carseats,
                                   type = "response")>0.5
# 计算Accuracy
print(table(test_Carseats$pred_boost,test_Carseats$High))
print(sum(test_Carseats$pred_boost == test_Carseats$High)/nrow(test_Carseats))
```






```
       FALSE TRUE
FALSE    108   17
TRUE       9   66
```

2.5 总结

对比一下三种集成学习方法：

- In *bagging*, the trees are grown independently on random samples of the observations. Consequently, the trees tend to be quite similar to each other. Thus, bagging can get caught in local optima and can fail to thoroughly explore the model space.
- In *random forests*, the trees are once again grown independently on random samples of the observations. However, each split on each tree is performed using a random subset of the features, thereby decorrelating the trees, and leading to a more thorough exploration of model space relative to bagging.
- In *boosting*, we only use the original data, and do not draw any random samples. The trees are grown successively, using a "slow" learning approach: each new tree is fit to the signal that is left over from the earlier trees, and shrunken down before it is used.

谢 谢!