Lab: 决策树

拟合分类树

`tree` 包用于构建分类和回归树。

```
library(tree)
```

我们首先用分类树来分析 `Carseats` 数据集。 在这个数据集中， `Sales` 是一个连续变量，所以我们从将其转换为二元变量开始。我们使用 `ifelse()` 函数来创建一个变量，并将其命名为 `High` ，如果 `Sales` 变量超过了8，就把它的值赋为 `Yes` ，否则为 `No` 。

```
library(ISLR2)
attach(Carseats)
High <- factor(ifelse(Sales <= 8, "No", "Yes"))
```

最后，我们使用 `data.frame()` 函数来将 `High` 与 `Carseats` 中剩余的数据合并。

```
Carseats <- data.frame(Carseats, High)
```

我们现在使用 `tree()` 函数来拟合一个分类树，从而通过除了 `Sales` 之外的变量预测 `High` 。 `tree()` 函数的语法规则与 `lm()` 函数非常相似。

```
tree.carseats <- tree(High ~ . - Sales, Carseats)
```

`summary()` 函数列出了树中用作内部节点的变量、终端节点的数量和（训练）错误率。

```
summary(tree.carseats)
```

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

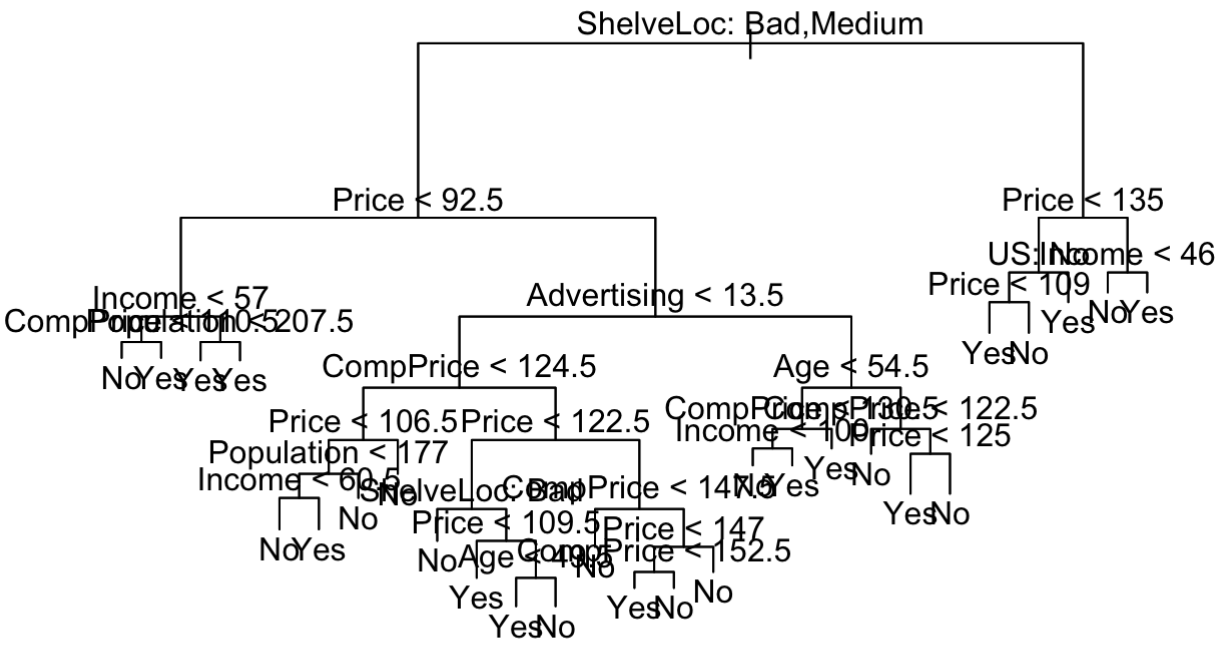
可以看见训练错误率为9%。 对于分类树， `summary()` 中的输出的偏差为

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

其中n_{mk}为第m个终端节点的属于第k个类的观测值的数量。这与熵密切相关。 一个小的偏差表示一个树对训练数据的拟合效果很好。 *residual mean deviance*就是偏差除以$n-|T_0|$，在这个例子中为除以$400-27=373$。

树最吸引人的特性之一是可以用图形显示。我们使用 `plot()` 函数来展示树的结构，并用 `text()` 函数来展示节点标签。参数 `pretty = 0` 使得 `R` 包括了所有定性预测变量的类别名，而不是简单地展示每个变量的一个字母。

```
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



对 `Sales` 影响最大的因素看起来应该是 `ShelveLoc` （shelving location），因为第一个分支将 `Good` 与 `Bad` 和 `Medium` 分开了。

如果我们只键入树对象的名称， `R` 的输出将对应树的每个分支。 `R` 将展示分类的准则 （例如， `Price < 92.5` ），节点中的观测值数量，偏差，节点的总体预测（ `Yes` 或 `No` ），以及节点中观测值取 `Yes` 或 `No` 的比例。叶节点使用星号表示。

```
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 ) *
```

为了正确评估分类树对这些数据的表现，我们必须估计测试误差，而不是简单地计算训练误差。

我们将数据集分为测试集和训练集，利用训练集构建树，使用测试集来评估表现。可以使用 `predict()` 函数来完成这一目的。

在分类树的情况，参数 `type = "class"` 使得 R 返回实际的预测类别。这个方法得到的测试集的预测准确率大约为77 %。

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  104  33
##       Yes  13  50
```

```
(104 + 50) / 200
```

```
## [1] 0.77
```

（如果重新执行 `predict()` ，可能得到略有不同的结果，这是因为"ties"：例如，当对应于终端节点的训练观测值在 `Yes` 和 `No` 响应值之间均匀地分类时，这种情况就可能发生。）

下面，我们考虑是否可以通过剪枝来改善结果。 函数 `cv.tree()` 使用交叉验证来确定最优的树的复杂度的水平；使用代价复杂度剪枝来选择一颗子树。 我们使用参数 `FUN = prune.misclass` 来表明我们希望使用分类错误率来主导交叉验证和剪枝过程，而不是 `cv.tree()` 中的默认的偏差。 `cv.tree()` 函数输出了所考虑的每个树的终端节点的数量（ `size` ）以及相应的错误率和所使用的代价复杂度的值（ `k` ，对应代价复杂度剪枝中的α）。

```
set.seed(7)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)
```

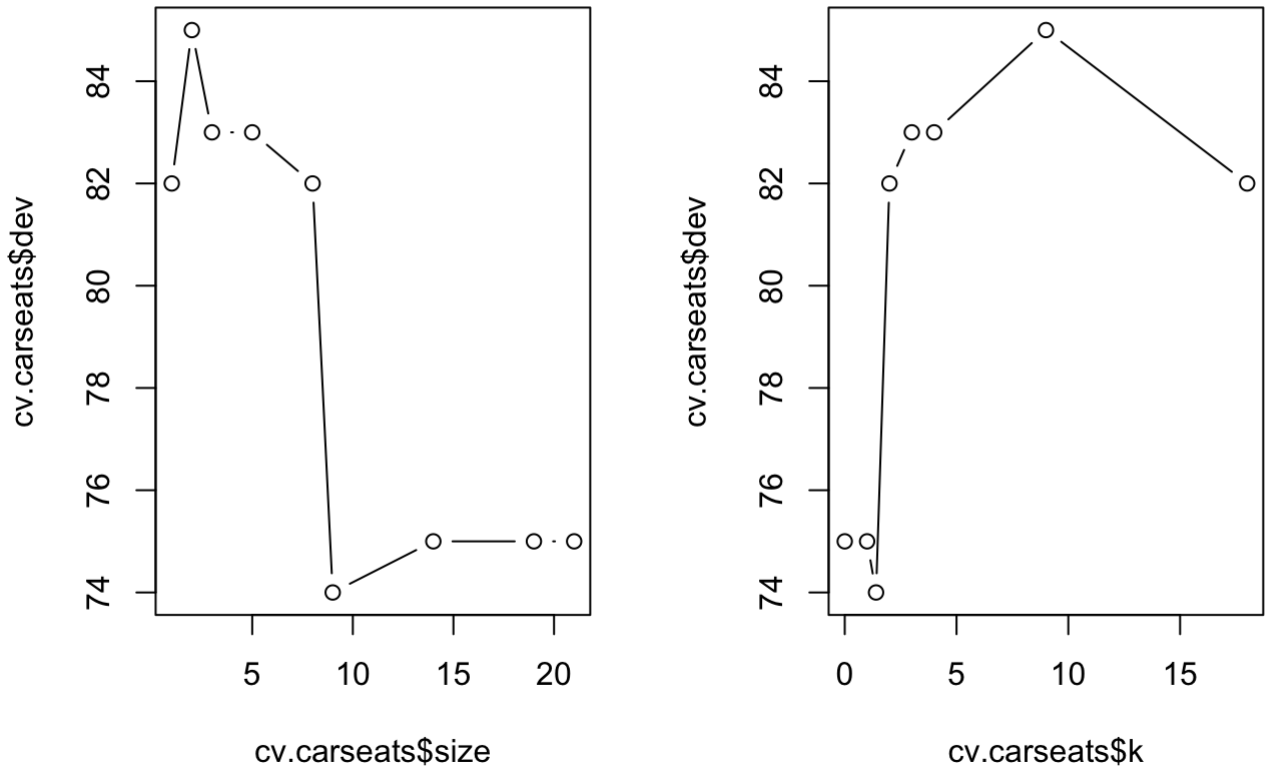
```
## [1] "size"   "dev"    "k"      "method"
```

```
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"
```

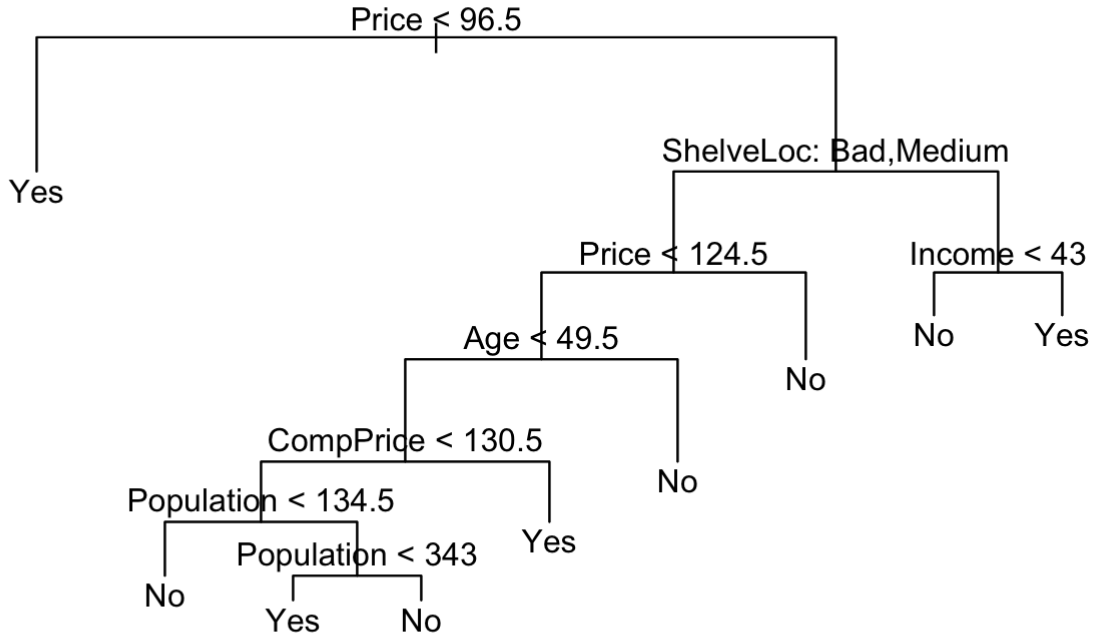
尽管名为 `dev` ，但它与交叉验证错误的数量相对应。具有9个终端节点的树只产生了74个交叉验证错误。我们将错误率绘制为 `size` 和 `k` 的函数。

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



我们现在应用 `prune.misclass()` 函数，以修剪树从而获得九节点树。

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



这个修剪的树在测试数据集上的表现如何？我们再次使用 `predict()` 函数。

```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred No Yes
##       No  97  25
##       Yes 20  58
```

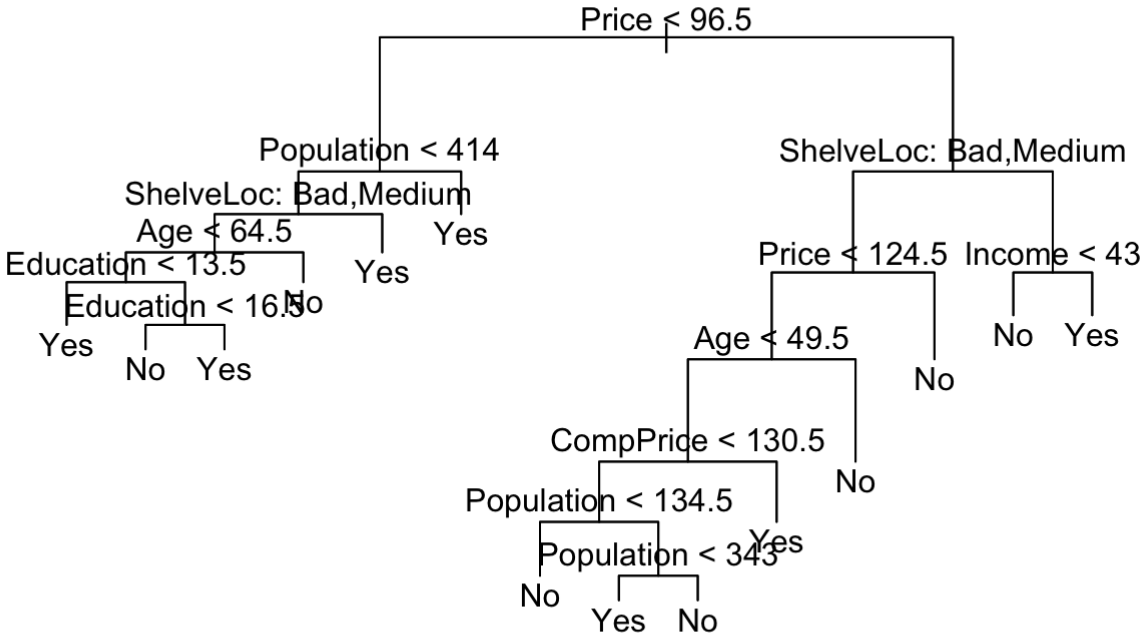
```
(97 + 58) / 200
```

```
## [1] 0.775
```

这样77.5 %的测试观测值被正确地分类，因此，修剪过程不仅产生了一个可解释性更好的树，而且还略微提高了分类精度。

如果我们增加 `best` 的值，我们将获得更大的修剪树，分类精度更低：

```
prune.carseats <- prune.misclass(tree.carseats, best = 14)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```




```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  102  31
##       Yes  15  52
```

```
(102 + 52) / 200
```

```
## [1] 0.77
```

拟合回归树

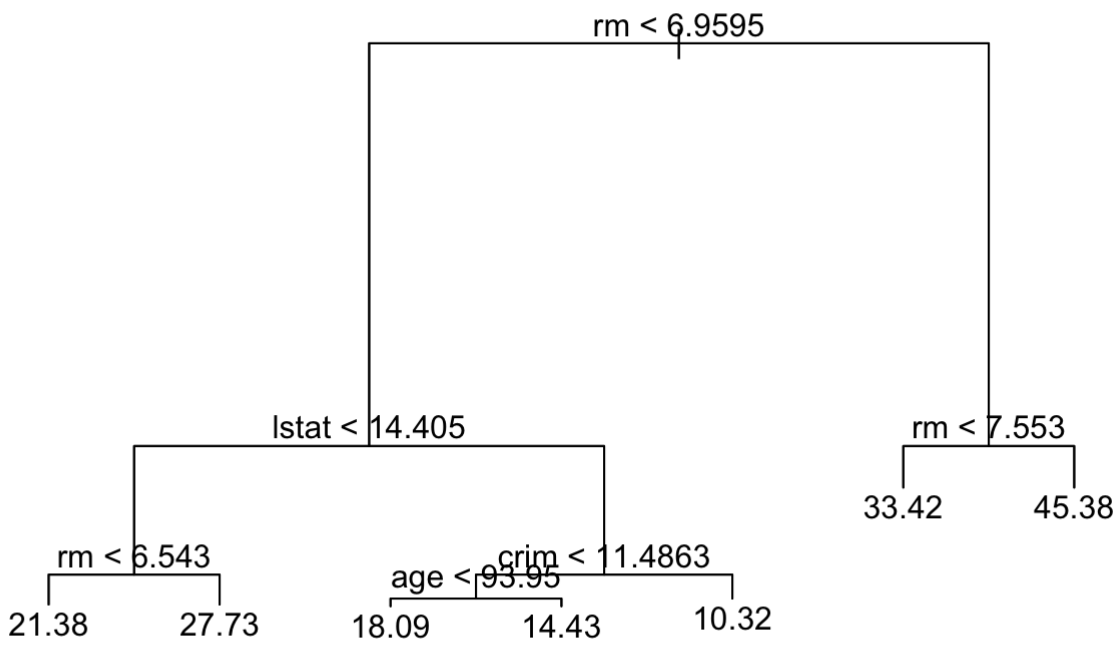
这里我们对 Boston 数据集拟合回归树。首先，我们创建一个训练集，并用训练集拟合树。

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston) / 2)
tree.boston <- tree(medv ~ ., Boston, subset = train)
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
```

注意到 summary() 的输出表明在构建树时只使用了四个变量。在回归树的背景下，偏差仅仅是树的平方误差之和。我们现在绘制树。

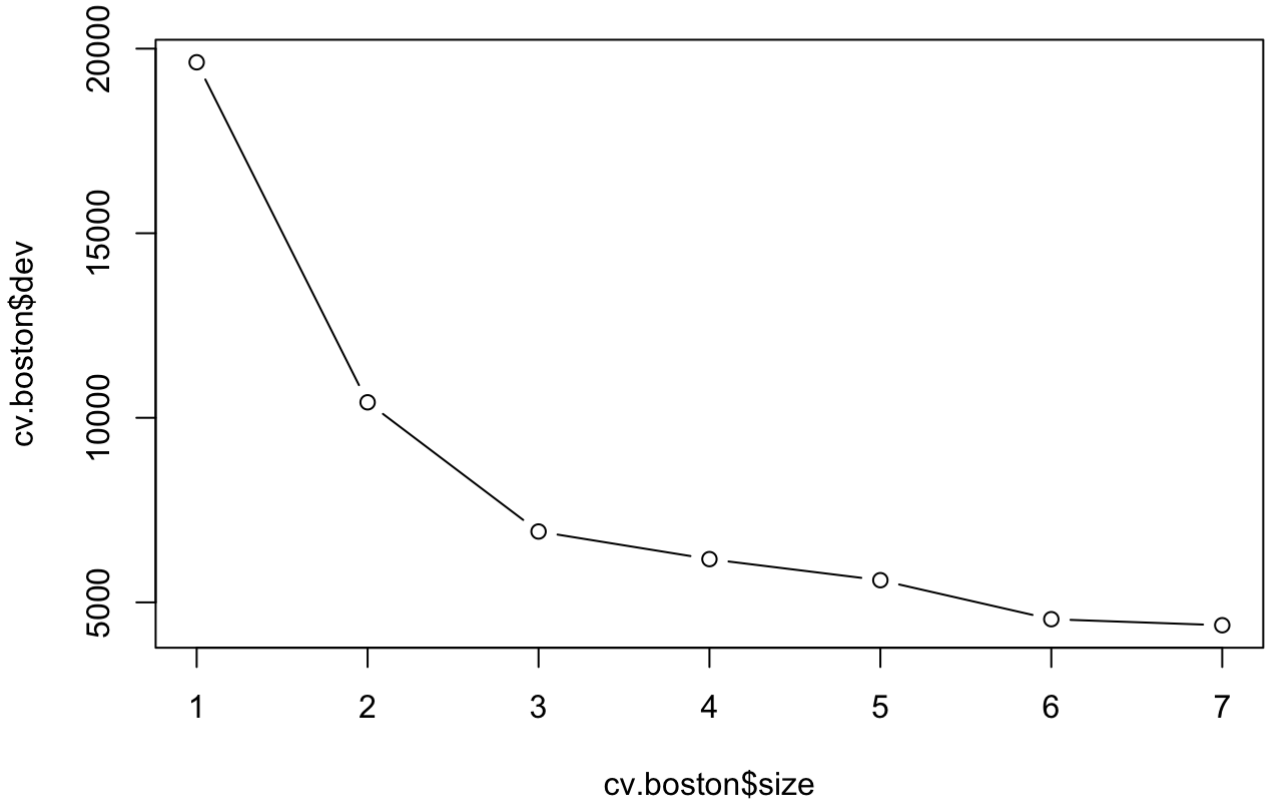
```
plot(tree.boston)
text(tree.boston, pretty = 0)
```



变量 lstat 衡量的是{lower socioeconomic status}的人的百分比，而变量 rm 对应的是房间的平均数量。树表明 rm 值更大或者 lstat 值更小对应着更贵的房子。例如，该树预测 rm >= 7.553 的区域，房屋中位数为 45,400。

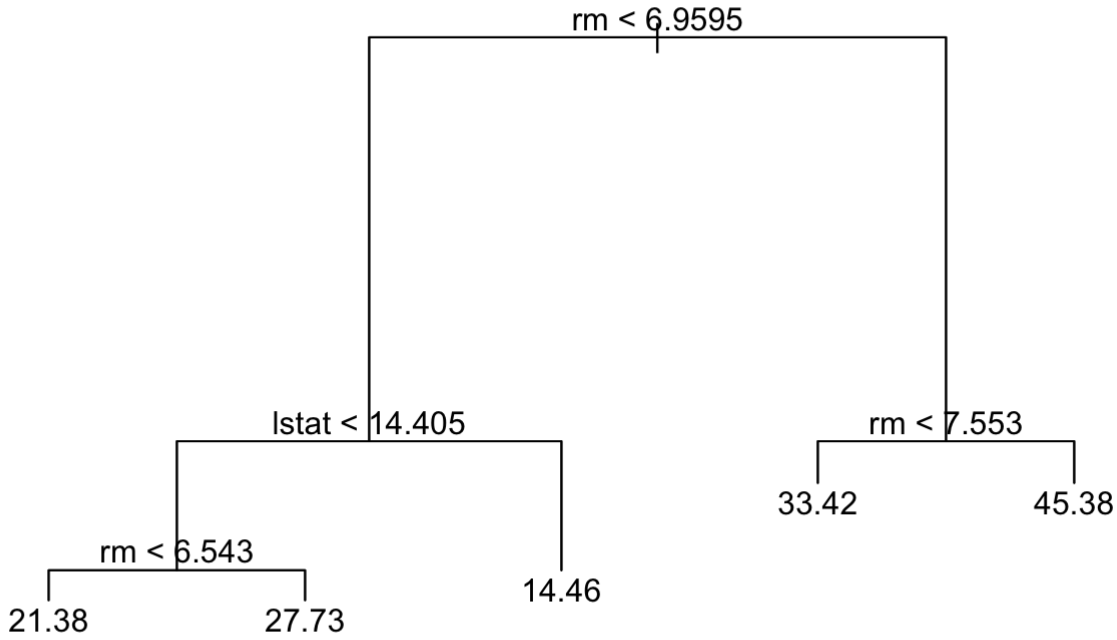
现在我们使用 cv.tree() 函数来看看修剪树是否会提升表现。

```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = "b")
```



在这种情况下，通过交叉验证我们选择所考虑的最复杂的树。然而，如果我们想修剪这棵树，我们可以使用 `prune.tree()` 函数并按如下方式进行，：

```
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
text(prune.boston, pretty = 0)
```


rm < 6.9595
lstat < 14.405
rm < 7.553
33.42
45.38
rm < 6.543
14.46
21.38
27.73