R_lab Arboles de Desición

my

26/10/2020

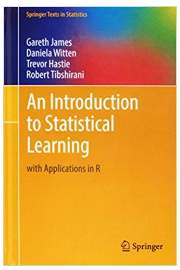


Figure 1: A nice image.

(Tomado de Introduction to Statistical Learning with Applications in R, by G. James,D. Witten, T.Hastie, R. Tibshirani)

La libreria `tree` es utilizada para construir árboles de clasificación y regresión.

```r
install.packages("ISLR")
install.packages("tree")
install.packages("MASS")
install.packages("randomForest")
install.packages("gbm")
```

```r
library(ISLR)
library(tree)
```



```r
install.packages("ISLR")
install.packages("tree")
install.packages("MASS")
install.packages("randomForest")
install.packages("gbm")
```

```r
library(ISLR)
library(tree)
```


La base de datos consiste en un conjunto simulado de ventas de asientos para niño en 400 tiendas distintas. Acá se creará una variable denominada **High**, la cual toma el valor de "Yes" si la variable **Sales** excede el valor 8 y toma el valor de "No" en cualquier otro caso.

```r
#data("Carseats")
#View(Carseats)
attach(Carseats)
High = ifelse(Sales <= 8,"No","Yes")
```

Se crea un único conjunto de datos.

Se ajusta un arbol de clasificación, para explicar la variable **High** utilizando todas las variables a excepción de la variable **Sales**.

```r
Carseats = data.frame(Carseats,High)

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

Carseats

	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	High
1	9.50	138	73	11	276	120	Bad	42	17	Yes	Yes	Yes
2	11.22	111	48	16	260	83	Good	65	10	Yes	Yes	Yes
3	10.06	113	35	10	269	80	Medium	59	12	Yes	Yes	Yes
4	7.40	117	100	4	466	97	Medium	55	14	Yes	Yes	No
5	4.15	141	64	3	340	128	Bad	38	13	Yes	No	No
6	10.81	124	113	13	501	72	Bad	78	16	No	Yes	Yes
7	6.63	115	105	0	45	108	Medium	71	15	Yes	No	No
8	11.85	136	81	15	425	120	Good	67	10	Yes	Yes	Yes
9	6.54	132	110	0	108	124	Medium	76	10	No	No	No
10	4.69	132	113	0	131	124	Medium	76	17	No	Yes	No

Figure 2: A nice image.

La función `summary()` enlista las variables que son utilizadas como nodos internos en el árbol, el número de nodos terminales u hojas y la tasa de error (de entrenamiento)

```r
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
```

La tasa de error(de entrenamiento) es del 9%.

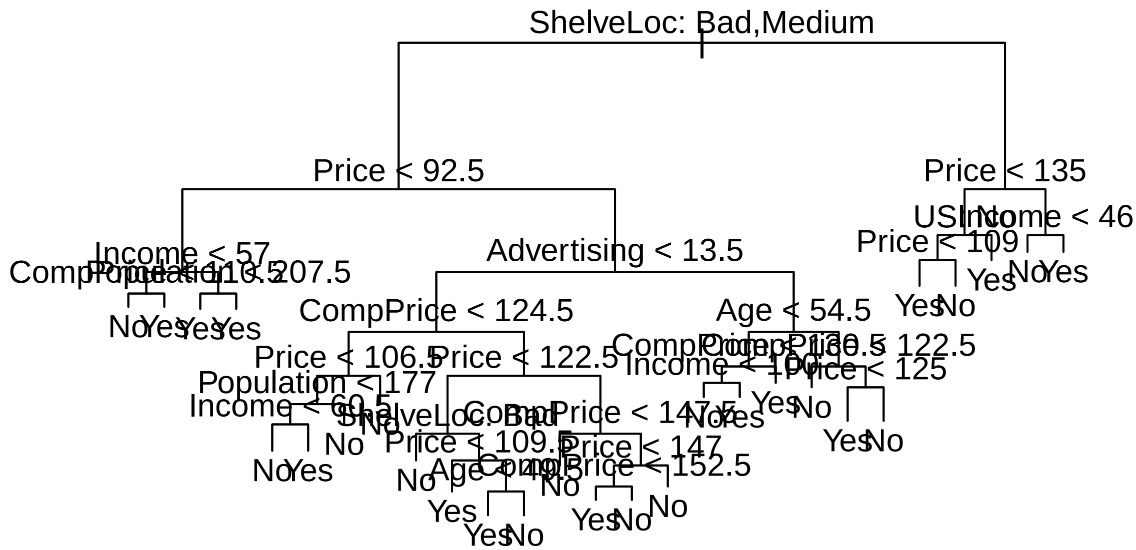
Para arboles de clasificación. La desviación (deviance) reportada en la salida de `summary()` está dada por

$$-2\sum_m \sum_k n_{mk} \log \hat{p}_{mk} \tag{1}$$

donde n_{mk} es el número de observaciones en el m-ésimo nodo terminal que pertenecen a la k-ésima categoria. Una desviación pequeña indica un arbol que proporciona un buen ajuste a los datos de entrenamiento

La desviación residual media reportada es simplemente la ***desviasión*** dividida por $n - |T_0|$ que en este caso es 400-27=373.

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



El indicador más importante de `Sales` parece ser un "nivel de calidad" debido a que la primera rama separa la categoria `Good` de las categorias `Bad` y `Medium`

Si solo escribimos el nombre del objeto del árbol, R imprime la salida correspondiente a cada rama del árbol. R muestra el criterio de división (por ejemplo, Price <92.5), el número de observaciones en esa rama, la desviación, la predicción general para la rama (Sí o No), y la fracción de observaciones en esa rama que toman valores de Sí y No. Las ramas que conducen a nodos terminales son indicado con asteriscos.

```
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 ) *
```

Para evaluar adecuadamente el desenpeño de un árbol de clasificación en estos datos, debemos estimar el error de prueba en lugar de simplemente calcular El error de entrenamiento. Dividimos las observaciones en un conjunto de entrenamiento y uno de prueba. Se construye el árbol utilizando el conjunto de entrenamiento y se evalua su desempeño en Los datos de prueba. La función `predict()` se puede utilizar para este propósito. En el caso de un árbol de clasificación, el argumento type = “class” indica a R que debe retornar La predicción de clase real. Este enfoque lleva a predicciones correctas para alrededor del 71.5% de las ubicaciones en el conjunto de datos de prueba.

```
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 ) *
```

Para evaluar adecuadamente el desenpeño de un árbol de clasificación en estos datos, debemos estimar el error de prueba en lugar de simplemente calcular El error de entrenamiento. Dividimos las observaciones en un conjunto de entrenamiento y uno de prueba. Se construye el árbol utilizando el conjunto de entrenamiento y se evalua su desempeño en Los datos de prueba. La función `predict()` se puede utilizar para este propósito. En el caso de un árbol de clasificación, el argumento type = "class" indica a R que debe retornar La predicción de clase real. Este enfoque lleva a predicciones correctas para alrededor del 71.5% de las ubicaciones en el conjunto de datos de prueba.

```
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
```

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

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

A continuación, consideramos si podar el árbol podría mejorar resultados. La función `cv.tree()` realiza validación cruzada para determinar el nivel óptimo de complejidad del árbol; Se utiliza poda con costo de complejidad para seleccionar una secuencia de árboles para su consideración. Se utiliza el argumento `FUN = prune.misclass` para indicar que sea el índice de error de clasificación el que guie el proceso de validación cruzada y poda, en lugar del valor predeterminado para la función `cv.tree()`, que es la desviación. La función `cv.tree()` reporta el número de nodos terminales de cada árbol considerado (tamaño), así como la tasa de error correspondiente y el valor de parámetro de costo-complejidad utilizado (k, que corresponde a α en (8.4)).

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

```
set.seed(3)
cv.carseats =cv.tree(tree.carseats ,FUN=prune.misclass)
names(cv.carseats)
```

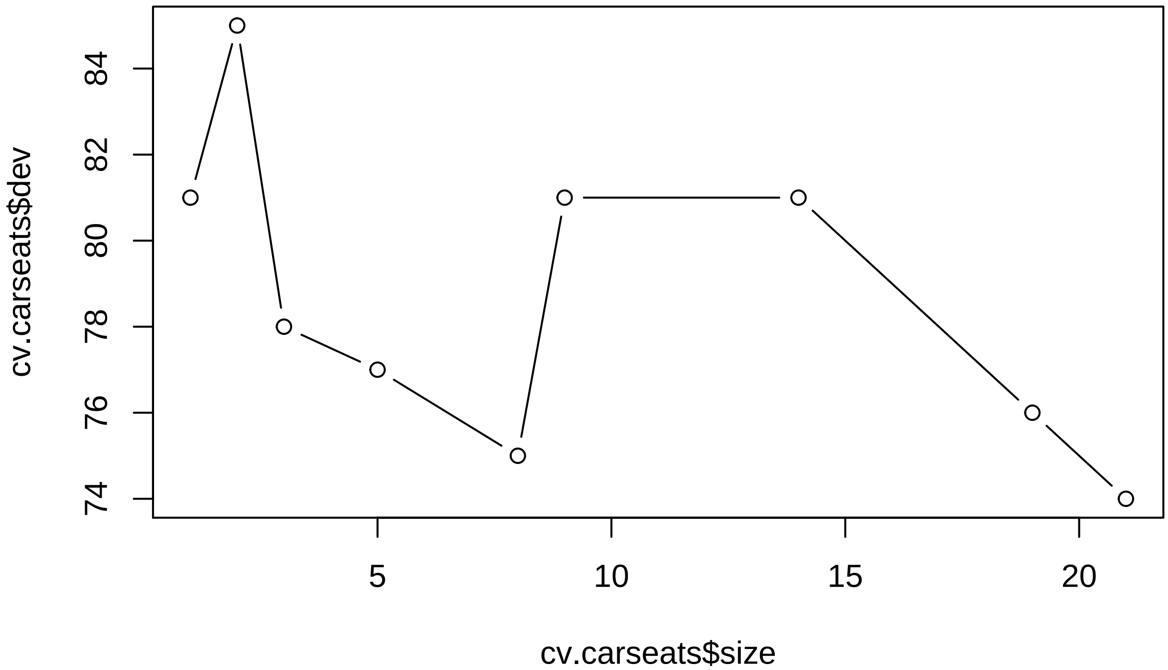
```
## [1] "size"   "dev"    "k"      "method"
```

```
cv.carseats
```

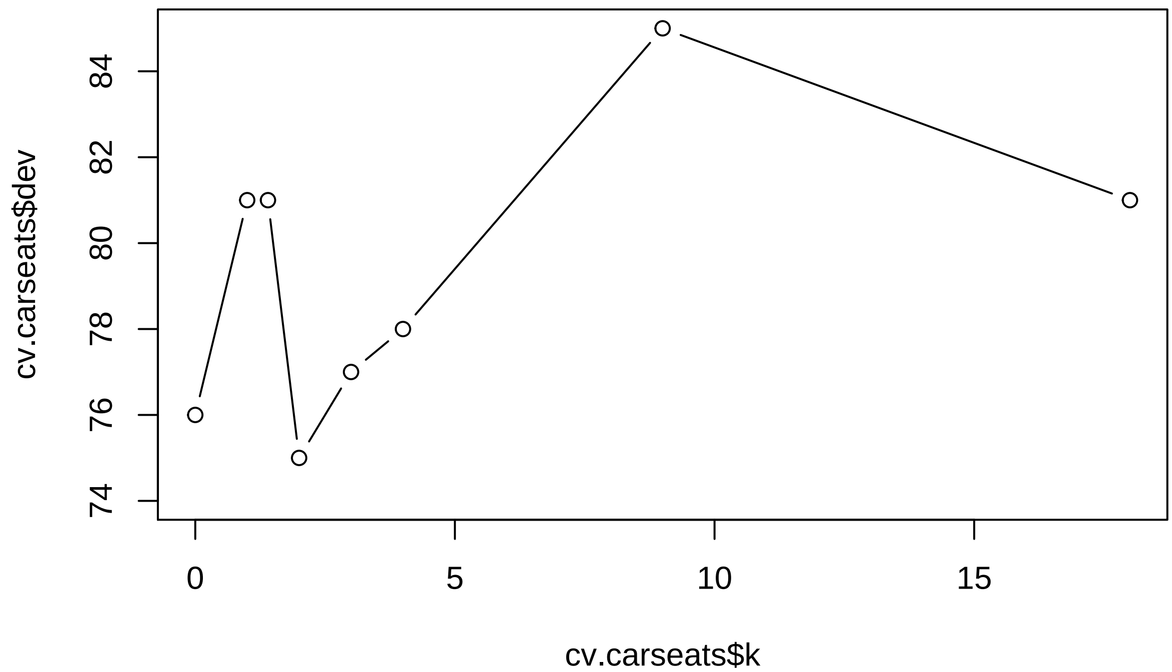
```
## $size
## [1] 21 19 14  9  8  5  3  2  1
##
## $dev
## [1] 74 76 81 81 75 77 78 85 81
##
## $k
## [1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Tengase en cuenta que, a pesar del nombre, **dev** corresponde a la tasa de error de validación cruzada en este caso. El árbol con 9 nodos terminales da como resultado la tasa de error de validación cruzada más baja, con 50 errores de validación cruzada. Trazamos el error en función de tamaño y k.

```
plot(cv.carseats$size ,cv.carseats$dev ,type="b")
```

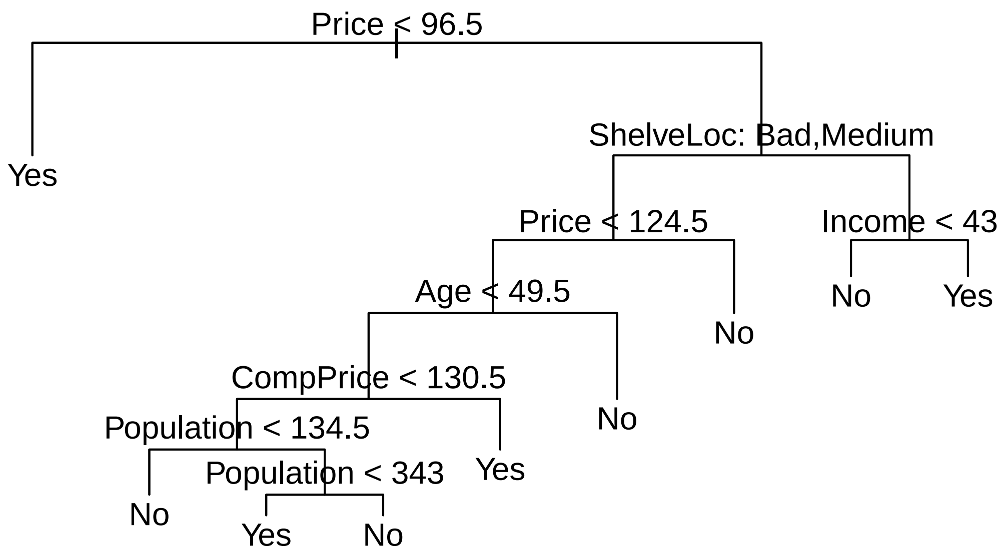



```
plot(cv.carseats$k ,cv.carseats$dev ,type="b")
```



Ahora se aplica la función **`prune.misclass()`** para podar el árbol y obtener el árbol de 9 nodos.

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



Que tan bien se desempeña el árbol podado en el cojunto de prueba?. Una vez más se aplica la función `predict()`

```
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
```

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

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

Ajuste de Arboles de Regresión

Acá se ajustará un árbol de regresión sobre el conjunto de datos **Boston**.

Este conjunto de datos se decribe como **"Housing Values in Suburbs of Boston""**

- Primero se crea un conjunto de entrenamiento y se entrena el árbol en el mismo.

```
library (MASS)
#data("Boston")
set.seed (1)
train = sample (1: nrow(Boston ), nrow(Boston )/2)
tree.boston =tree(medv~.,Boston ,subset =train)
```

```r
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
```


Boston
Filter

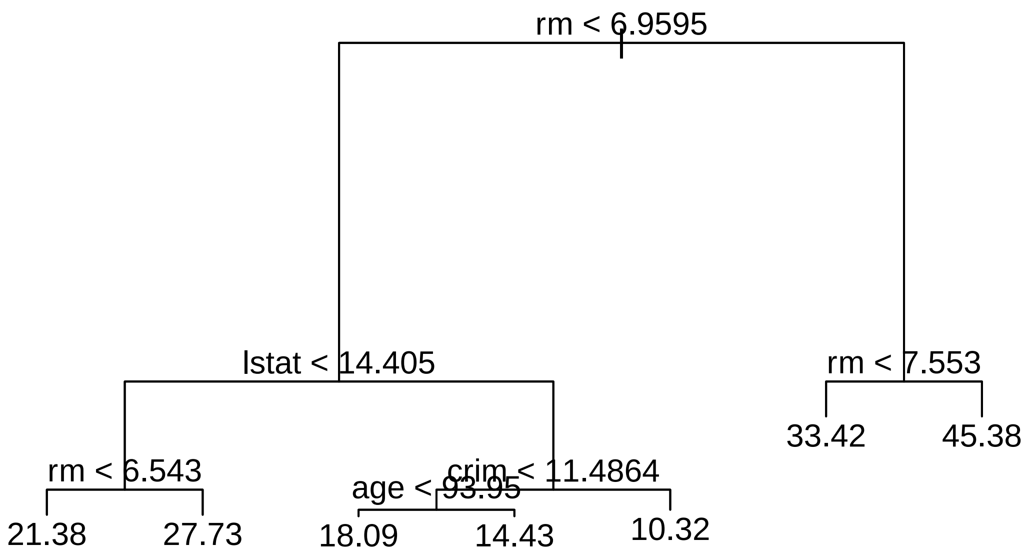
	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
1	0.00632	18.0	2.31	0	0.5380	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
2	0.02731	0.0	7.07	0	0.4690	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
3	0.02729	0.0	7.07	0	0.4690	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
4	0.03237	0.0	2.18	0	0.4580	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
5	0.06905	0.0	2.18	0	0.4580	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
6	0.02985	0.0	2.18	0	0.4580	6.430	58.7	6.0622	3	222	18.7	394.12	5.21	28.7
7	0.08829	12.5	7.87	0	0.5240	6.012	66.6	5.5605	5	311	15.2	395.60	12.43	22.9
8	0.14455	12.5	7.87	0	0.5240	6.172	96.1	5.9505	5	311	15.2	396.90	19.15	27.1
9	0.21124	12.5	7.87	0	0.5240	5.631	100.0	6.0821	5	311	15.2	386.63	29.93	16.5
10	0.17004	12.5	7.87	0	0.5240	6.004	85.9	6.5921	5	311	15.2	386.71	17.10	18.9

Figure 3: A nice image.

Observese que la salida de `summary()` indica que solo tres de las variables han sido utilizados en la construcción del árbol. En el contexto de un árbol de regresión. , la desviación (**deviance**) es simplemente la suma de los errores al cuadrado del árbol.

rm	average number of rooms per dwelling.
lstat	lower status of the population (percent).
crim	per capita crime rate by town.
age	proportion of owner-occupied units built prior to 1940.

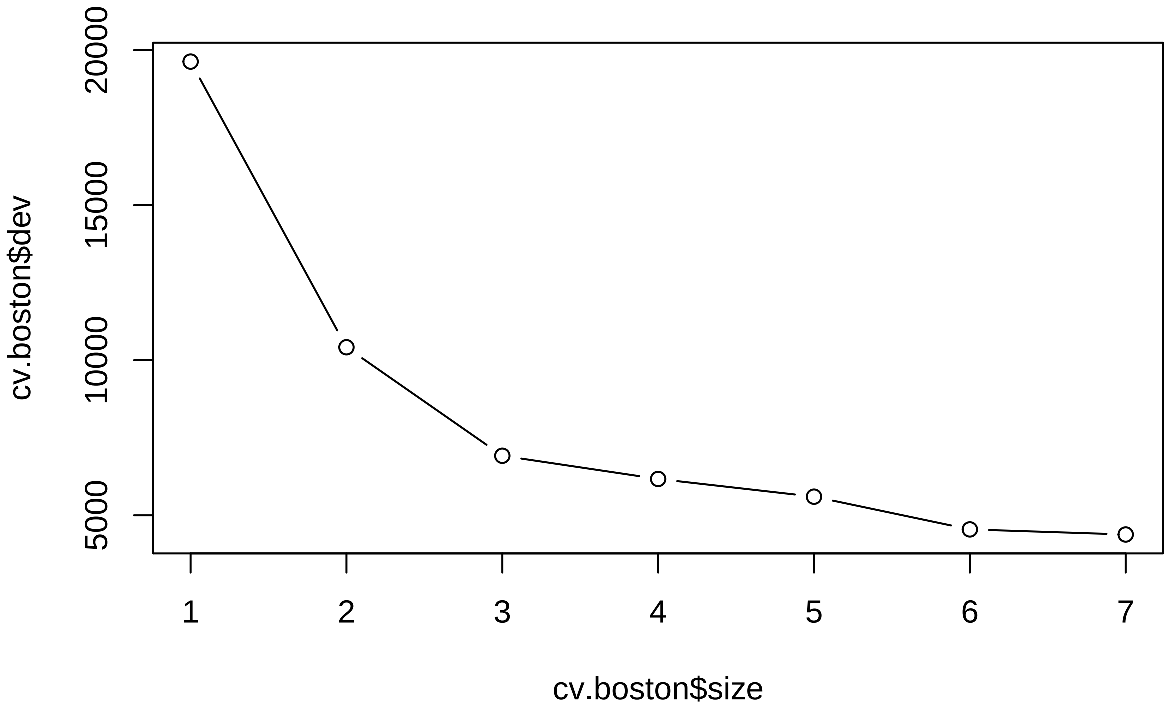
```r
plot(tree.boston )
text(tree.boston ,pretty =0)
```



La variable **lstat** mide el porcentaje de individuos con menor Estatus socioeconómico. El árbol indica que los valores más bajos de **lstat** corresponden a casas más caras. El árbol predice un precio medio de la vivienda. de $27730 para hogares más grandes en suburbios en los que los residentes tienen un alto nivel socioeconómico ($rm \geq 6543$ y $lstat < 14405$).

Ahora utilizamos la función **cv.tree()** para ver si una poda del árbol mejora su desempeño

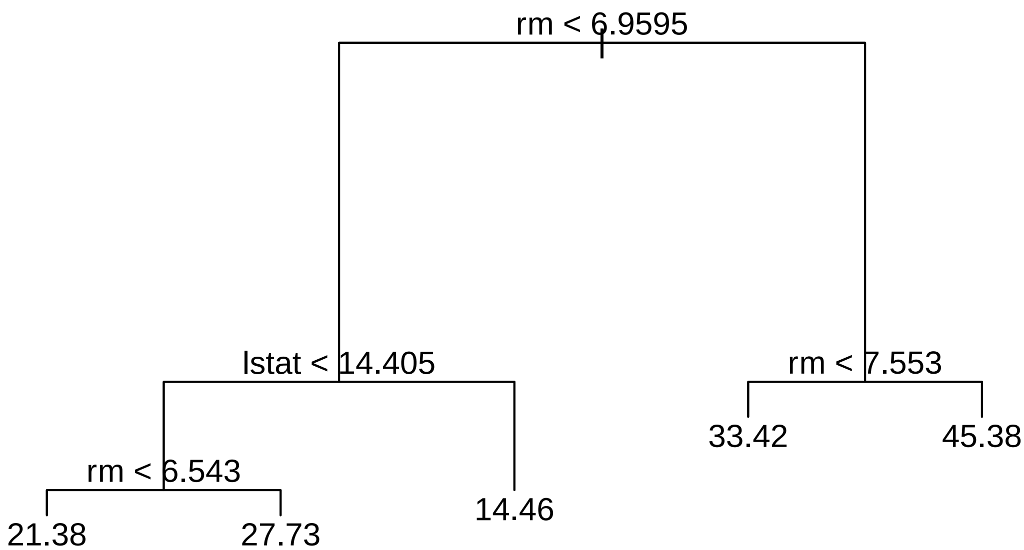
```
cv.boston =cv.tree(tree.boston )
plot(cv.boston$size ,cv.boston$dev,type='b')
```



Según la gráfica, por la validación cruzada se debe seleccionar el árbol más complejo

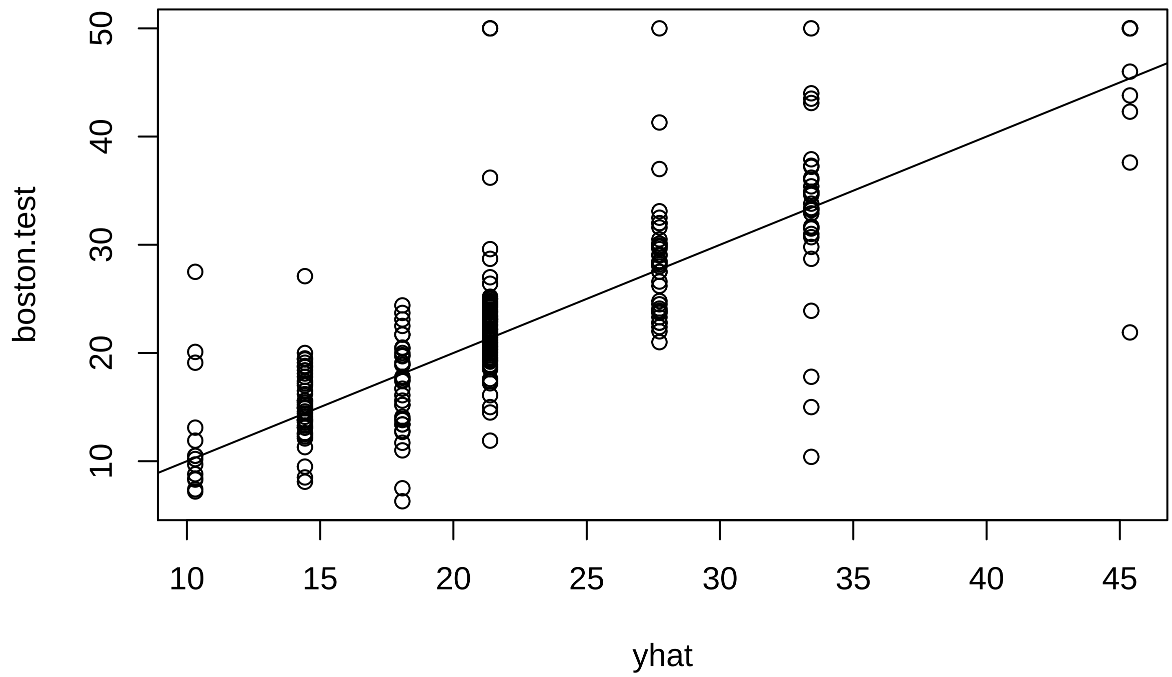
Ahora bien si se quiere podar el árbol, se utiliza la función `prune.tree()`

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



Teniendo en cuenta los resultados de la validación cruzada, se utiliza el árbol para hacer predicciones en el conjunto de prueba

```
yhat = predict(tree.boston ,newdata = Boston[-train ,])
boston.test = Boston[-train ,"medv"]
plot(yhat,boston.test)
abline(0,1)
```




```
mean((yhat-boston.test)^2)
```

```
## [1] 35.28688
```

El MSE en el conjunto de prueba asociado con el árbol de regresión es de 35.29. La raiz cuadrada de 35.29 es 5.94 o aproximadamente 6.0. Indicando que en este modelo las predicciones están alrededor de $6000.0 del verdadero valor medio para una casa en los suburvios.

Bagging y Bosque aleatorio.

Ahora se aplicará *bagging* y bosque aleatorio (**random forest**) al conjunto de datos **Boston**, utilizando el paquete **randomForest** de R.

Los resultados exactos obtenidos en esta sección pueden depender de la versión de R y la versión del paquete **randomForest** instalados en el computador. Recordemos que el *bagging* es simplemente un caso especial de un bosque aleatorio con $m = p$. Por lo tanto, la función `randomForest()` puede ser utilizada para realizar tanto *bosques aleatorios* como *bagging*. Ahora se lleva a cabo **bagging** con la función `randomForest()` como sigue:

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

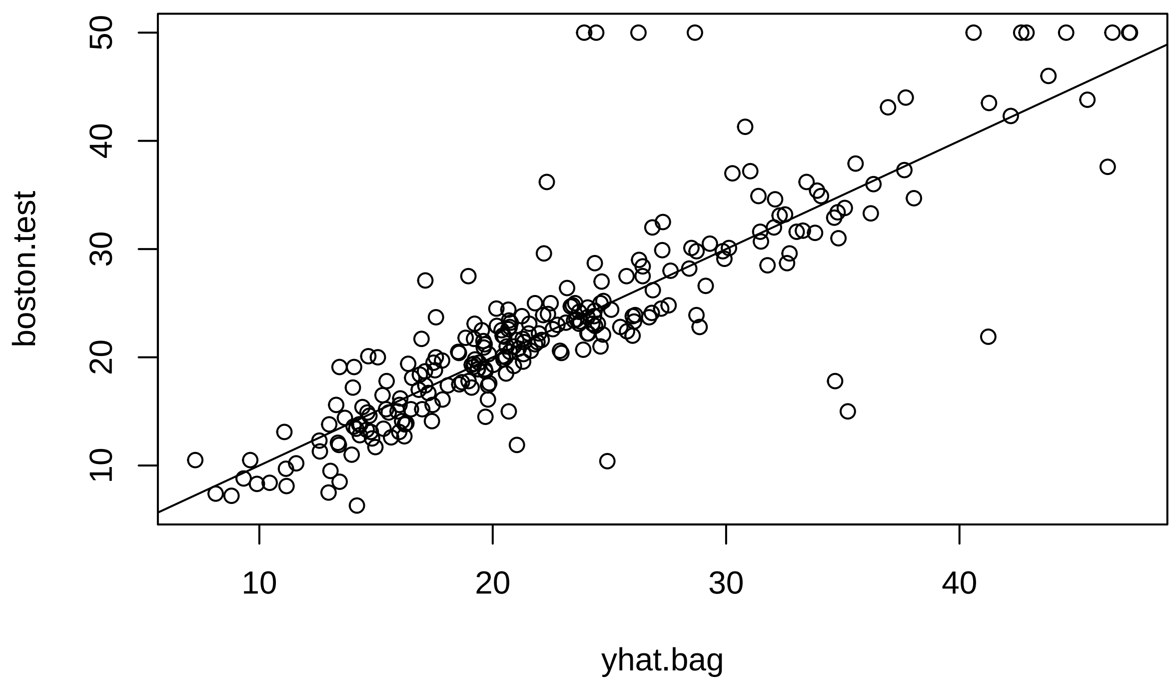
```
set.seed(1)
bag.boston = randomForest(medv~.,data=Boston ,subset =train,mtry=13, importance =TRUE)
bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE,      subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.39601
##                     % Var explained: 85.17
```

El argumento `mtry=13` especifica que todos los 13 predictores deben ser utilizados en cada split o segmentación en la construcción del árbol. Considerese ahora la siguiente cuestión

Si se debe llevar a cabo un bagging, que tan bien se desepeña el modelo **Bagged**?

```r
yhat.bag = predict(bag.boston ,newdata = Boston[-train ,])
plot(yhat.bag , boston.test)
abline(0,1)
```




```r
mean(( yhat.bag -boston.test)^2)
```

```
## [1] 23.59273
```

El MSE de prueba asociado con el árbol *Bagged* es de 23.6, mucho menor que con el
árbol simpelmente podado de forma óptima (approx 35/%).

- Se puede cambiar el número de árboles generados por la función `randomForest()` utilizando el argumento `ntree`

```r
bag.boston = randomForest(medv~.,data = Boston ,subset =train ,mtry=13, ntree =25)
yhat.bag = predict(bag.boston ,newdata = Boston[-train ,])
mean(( yhat.bag -boston.test)^2)
```

```
## [1] 23.66716
```

Generar un bosque aleatorio procede exactamente de la misma manera, excepto que se utiliza un valor menor del argumento `mtry`. Por defecto, `randomForest()` utiliza $p/3$ variables al construir un bosque aleatorio de árboles de regresión, y $\sqrt{p}$ variables al construir un bosque aleatorio de árboles de clasificación. Aquí se utilizará `mtry = 6`.

```r
set.seed (1)
rf.boston = randomForest(medv~.,data=Boston ,subset =train,mtry = 6, importance =TRUE)
yhat.rf = predict(rf.boston ,newdata = Boston[-train ,])
mean(( yhat.rf - boston.test)^2)
```

```
## [1] 19.62021
```

El MSE de prueba igual a 19.62 indica que hay una mejora sobre el árbol *Bagged* en este caso.

Utilizando la función "importance()", se puede indagar por la importancia de cada variable

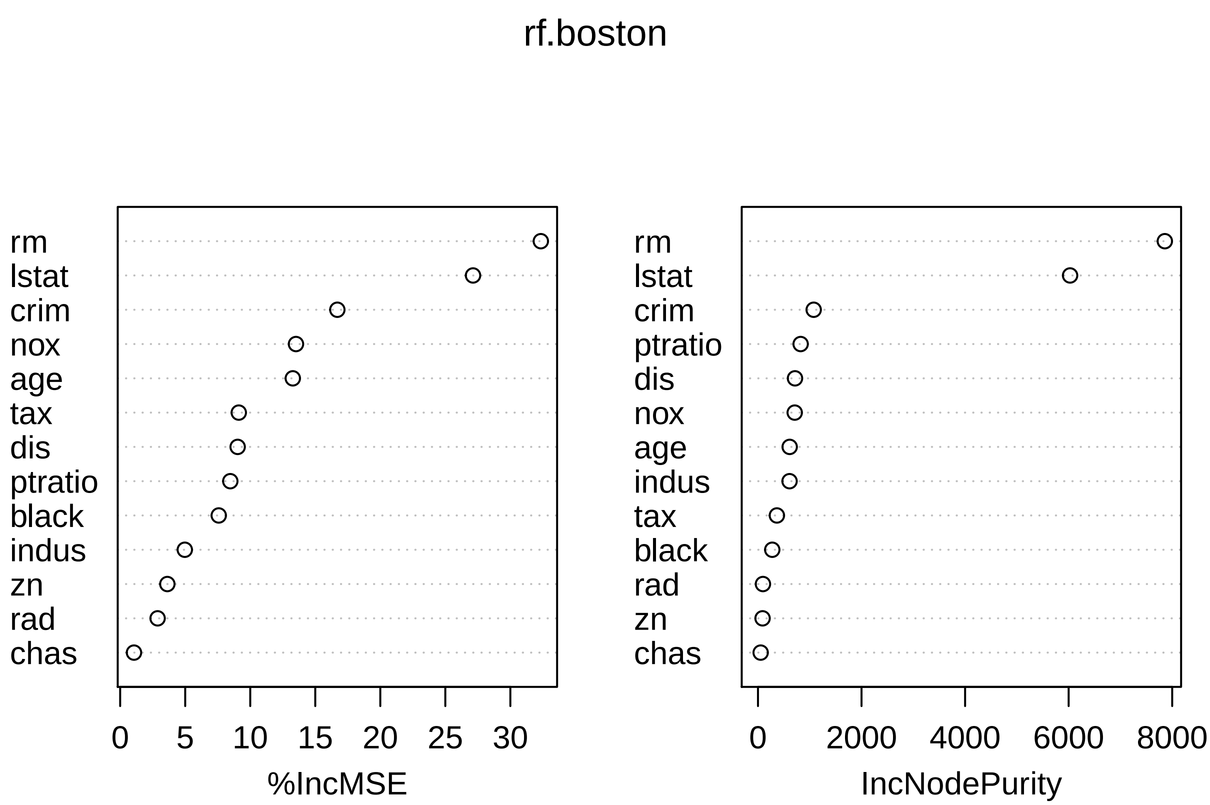
```r
importance(rf.boston)
```

```
##            %IncMSE IncNodePurity
## crim     16.697017    1076.08786
## zn        3.625784      88.35342
## indus     4.968621     609.53356
## chas      1.061432      52.21793
## nox      13.518179     709.87339
## rm       32.343305    7857.65451
## age      13.272498     612.21424
## dis       9.032477     714.94674
## rad       2.878434      95.80598
## tax       9.118801     364.92479
## ptratio   8.467062     823.93341
## black     7.579482     275.62272
## lstat    27.129817    6027.63740
```

Se informan dos medidas de importancia para las variable.

- La primera está basada en la disminución media de la precisión en las predicciones en las muestras **bagging** cuando una variable dada se excluye del modelo.
- La segunda es una medida de la disminución total en la impureza del nodo que resulta de divisiones sobre esa variable, promediada sobre todos los árboles (esto se trazó en la Figura 8.9).
- En el caso de los árboles de regresión, la impureza del nodo se mide mediante la RSS en el conjunto de prueba, y para árboles de clasificación por la desviación (**deviance**).
- Se pueden obtener gráficos de estas medidas de importancia utilizando la función "varImpPlot()".

```r
varImpPlot(rf.boston)
```



Los resultados indican que en todos los árboles considerados en el bosque aleatorio, el nivel de riqueza de la comunidad (**"lstat"**) y el tamaño de la casa (**rm**) son, con mucho, las dos variables más importantes.

Boosting

Acá se utilizará el paquete **gbm**, y dentro de la función **gbm()**, para ajustar árboles de regresión *boosted* al conjunto de datos de Boston. Ejecutamos **gbm()** con la opción **distribución = "gaussian"** ya que este es un problema de regresión; si fuera un problema de clasificación binario, se utilizaría **distribution = "bernoulli"**. los argumento **n.trees = 5000** indica que queremos 5000 árboles, y la opción **Interaction.depth = 4** limita la profundidad de cada árbol.

```
library(gbm)
```

```
## Loaded gbm 2.1.5
```

```
set.seed(1)
boost.boston = gbm(medv~.,data=Boston[train ,], distribution="gaussian",
n.trees = 5000, interaction.depth =4)
```

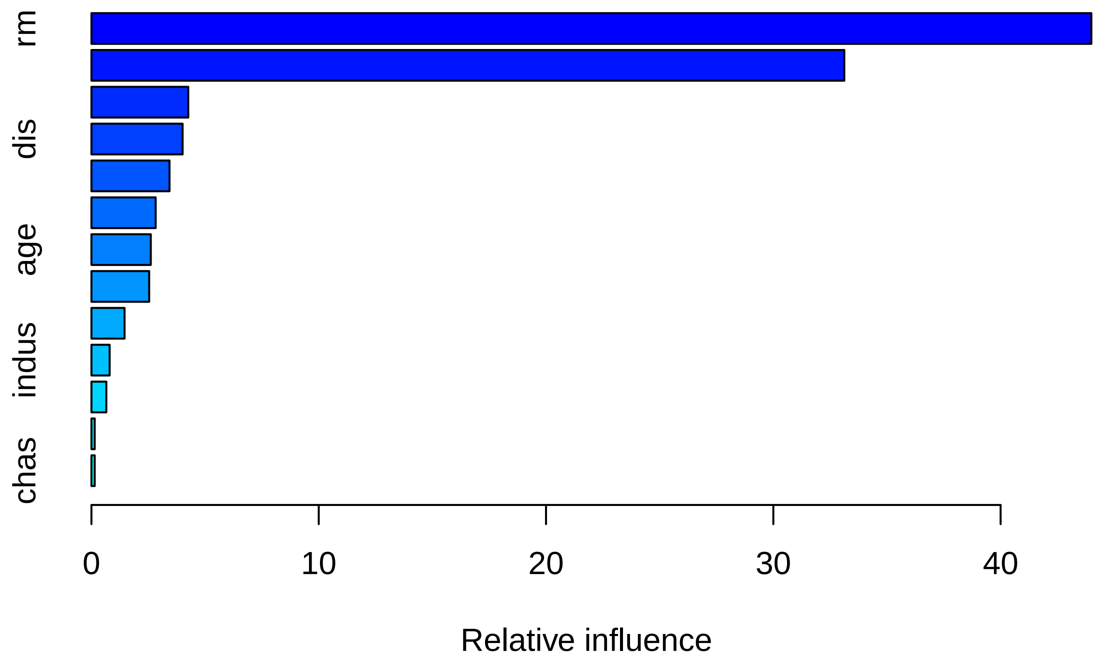
Acá se utilizará el paquete **gbm**, y de la función `gbm()` del mismo, para ajustar árboles de regresión *boosted* al conjunto de datos de Boston. Ejecutamos `gbm()` con la opción **distribución = "gaussian"** ya que este es un problema de regresión; si fuera un problema de clasificación binario, se utilizaría **distribution = "bernoulli"**. los argumento `n.trees = 5000` indica que queremos 5000 árboles, y la opción `Interaction.depth = 4` limita la profundidad de cada árbol.

```
library(gbm)
set.seed(1)
```

```r
boost.boston = gbm(medv~.,data=Boston[train ,], distribution ="gaussian",n.trees = 5000, interaction.dep
```

La función **summary()** produce un gráfico de influencia relativa y también genera estadísticas de influencia relativa.

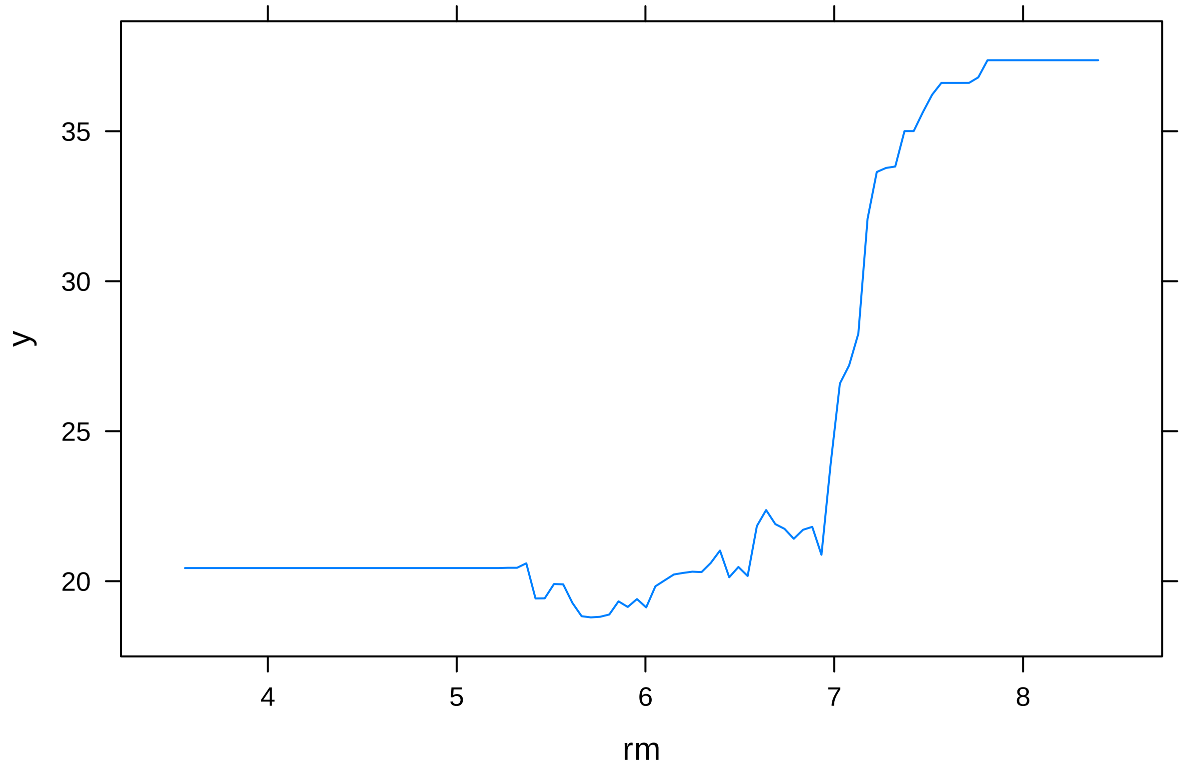
```r
summary(boost.boston)
```



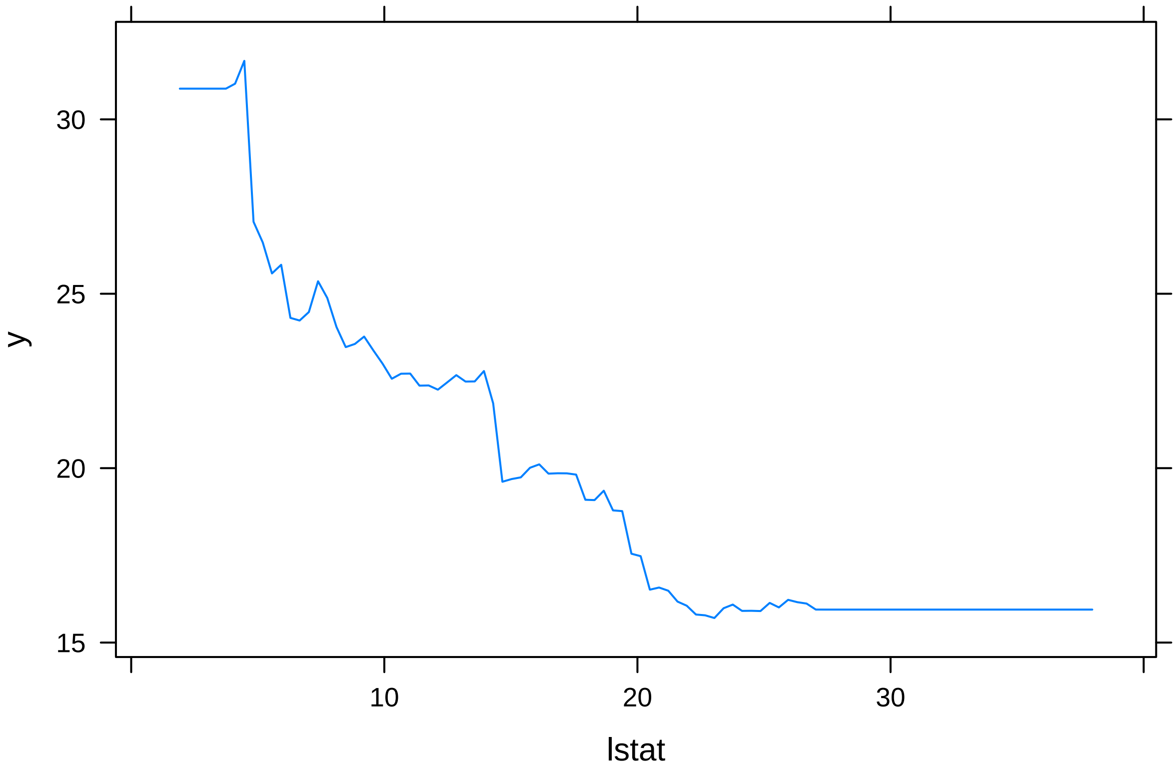

```
##             var    rel.inf
## rm           rm 43.9919329
## lstat     lstat 33.1216941
## crim       crim  4.2604167
## dis         dis  4.0111090
## nox         nox  3.4353017
## black     black  2.8267554
## age         age  2.6113938
## ptratio ptratio  2.5403035
## tax         tax  1.4565654
## indus     indus  0.8008740
## rad         rad  0.6546400
## zn           zn  0.1446149
## chas       chas  0.1443986
```

Vemos que las variables **lstat** y **rm** son, por mucho, las variables más importantes. Se puede también conseguir gráficos de dependencia parcial para estas dos variables. Estos gráficos ilustran el efecto marginal en la respuesta de las variables seleccionadas, aparte del efecto conjunto de las demás. En este caso, como se podría esperar, la mediana de los precios de la vivienda aumentan con **rm** y disminuyen con **lstat**.

```r
plot(boost.boston ,i ="rm")
```




```r
plot(boost.boston ,i ="lstat")
```



Ahora se utiliza el modelo **Boosted** para predecir **medv** en el conjunto de prueba

```
yhat.boost=predict(boost.boston ,newdata = Boston[-train ,],n.trees =5000)
mean(( yhat.boost -boston.test)^2)
```

```
## [1] 18.84709
```

El MSE de 18.85 es parecido al MSE de 19.62 conseguido con el bosque aleatorio y superior que el obtenido con *bagging*.Si se quiere utilizar **Boosting** con un valor diferente del parámetro de contracción λ en (8.10). El valor por defecto el valor es 0.001, pero este se modifica fácilmente. Aquí se tomará $\lambda = 0.2$.

$$\hat{f} = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{3}$$

```
boost.boston =gbm(medv~.,data=Boston [train ,], distribution="gaussian",n.trees =5000 ,
                  interaction.depth =4, shrinkage =0.2,verbose =F)
yhat.boost=predict(boost.boston ,newdata = Boston[-train ,],
n.trees =5000)
mean(( yhat.boost -boston.test)^2)
```

```
## [1] 18.33455
```

Asi, en este caso utilizando $\lambda = 0.2$ se consigue una pequeña mejora en el MSE del conjunto de prueba que cuando se utiliza $\lambda = 0.001$.