Árboles de predicción: tree-based, bagging, random forest y boosting

Joaquín Amat Rodrigo j.amatrodrigo@gmail.com

Febrero, 2017

Índice

Introducción

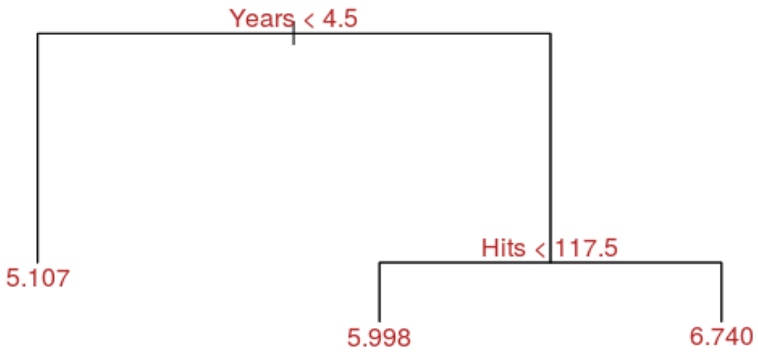
Los métodos predictivos como la regresión lineal o polinómica generan modelos globales en los que una única ecuación se aplica a todo el espacio muestral. Cuando el estudio implica múltiples predictores que interaccionan entre ellos de forma compleja y no lineal, es muy difícil encontrar un modelo global que sea capaz de reflejar la relación entre las variable. Existen métodos de ajuste no lineal que combinan múltiples funciones y que realizan ajustes locales, sin embargo, suelen ser difíciles de interpretar.

Los métodos estadísticos basados en árboles engloban a un conjunto de técnicas no paramétricas que consiguen segmentar el espacio de los predictores en regiones simples, dentro de las cuales es más sencillo manejar las interacciones. Se pueden emplear con dos finalidades: regresión y clasificación. En ocasiones se les conoce como *adaptative nearest-neighbor methods* por que guardan similitud con el método k-nearest-neigbors, que consiste en identificar en el espacio muestral las k observaciones más cercanas y promediar su valor. Esta aproximación sufre dos limitaciones: que solo tiene en cuenta el valor de los predictores y no el de la variable respuesta, y que el valor k elegido es constante para todas las regiones, independientemente de si están más o menos concurridas. Los arboles de predicción consiguen superar estos dos problemas gracias a que las ramificaciones agrupan observaciones acorde a similitudes en los predictores y en la variable respuesta, y porque cada ramificación puede contener un número distinto de observaciones.

Árboles de regresión

Idea intuitiva

La forma más sencilla de entender la idea detrás de los árboles de regresión es a través de un ejemplo simplificado. El set de datos `Hitter` contiene información sobre 322 jugadores de béisbol de la liga profesional. Entre las variables registradas para cada jugador se encuentran: el salario (*Salary*), años de experiencia (*Years*) y el número de bateos durante los últimos años (*Hits*). Utilizando estos datos, se quiere predecir el salario (en unidades logarítmicas) de un jugador en base a su experiencia y número de bateos. El árbol resultante se muestra en la siguiente imagen.



La interpretación del árbol se hace en sentido descendente, por lo que la primera división separa a los jugadores en función de si superan o no los 4.5 años de experiencia. En la rama izquierda quedan aquellos con menos de 4.5 años y su salario predicho se corresponde con el salario promedio de todos los jugadores de este grupo ($\epsilon^{5.107}$). Los jugadores con más de 4.5 años se asignan a la rama derecha, que a su vez se subdivide en función de si superan o no 117.5 bateos. Como resultado de las estratificaciones se han generado 3 regiones que pueden identificarse con la siguiente nomenclatura:

- $R_1 = \{X|Year < 4.5\}$: jugadores que han jugado menos de 4.5 años.
- $R_2 = \{X|Year \geq 4.5, Hits < 117.5\}$: jugadores que han jugado 4.5 años o más y que han conseguido menos de 117.5 bateos.
- $R_3 = \{X|Year \geq 4.5, Hits \geq 117.5\}$: jugadores que han jugado 4.5 años o más y que han conseguido 117.5 o más bateos.

A las regiones R_1, R_2 y R_3 se les conoce como *terminal nodes* o *leaves* del árbol, a los puntos en los que el espacio de los predictores sufre una división como *internal nodes* y a los segmentos que conectan dos nodos *branches* o ramas.

La interpretación del árbol mostrado en la imagen es la siguiente: el factor más importante a la hora de determinar el salario de un jugador es el número de años jugados, los jugadores con más experiencia ganan más. Entre los jugadores con pocos años de experiencia, el número de bateos logrados en los años previos no tiene mucho impacto en el salario, sin embargo, sí lo tiene para jugadores con cuatro años y medio o más de experiencia. Para estos últimos, a mayor número de bateos logrados mayor salario. Queda patente que la principal ventaja de los árboles de decisión frente a otros métodos de regresión es su fácil interpretación y la gran utilidad de su representación gráfica.

Construcción del árbol

El proceso de construcción de un árbol de regresión se divide en dos etapas:

- División del espacio de los predictores (nodos internos) generando regiones no solapantes R_1, R_2, R_3, ..., R_j.
- Predicción: la predicción de todas las observaciones que caen en la región R_j es la misma, la media de la variable respuesta en esa región.

A pesar de la sencillez con la que se puede resumir el proceso de construcción de un árbol, es necesario establecer una metodología que permita crear las regiones R_1, R_2, R_3, ..., R_j, o lo que es equivalente, decidir donde se introducen nodos internos. Aunque desde el punto de vista teórico las regiones podrían tener cualquier forma, si se limitan a regiones rectangulares (de múltiples dimensiones) se simplifica en gran medida el proceso de construcción y se facilita la interpretación. Una vez impuesta esta restricción, el objetivo es encontrar las J regiones (R_1,..., R_j) que minimizan el *Residual Sum of Squares (RSS)*:

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

donde $\hat{y}_{R_j}$ es la media de la variable respuesta en la región R_j. Una descripción menos técnica equivale a decir que se busca una distribución de regiones tal que, el sumatorio de las desviaciones al cuadrado entre las observaciones y la media de la región a la que pertenecen sea lo menor posible.

Desafortunadamente, no es posible considerar todas las posibles particiones del espacio de los predictores. Por esta razón se recurre a lo que se conoce como *recursive binary splitting* (división binaria recursiva). Esta solución sigue la misma idea que la selección de predictores *stepwise (backward o fordward)* en regresión lineal múltiple, no evalúa todas las posibles regiones pero alcanza un buen balance computación-resultado.

RECURSIVE BINARY SPLITTING

El objetivo del método *recursive binary splitting* es encontrar en cada iteración el predictor X_j y el punto de corte s tale que, si se distribuyen las observaciones en las regiones $\{X|X_j < s\}$ y $\{X|X_j \geq s\}$, se consigue la mayor reducción posible en el *RSS*. Está metodología conlleva dos hechos. 1) Que cada división óptima se identifica acorde al impacto que tiene en ese momento, no se tiene en cuanta si es la división que dará lugar a mejores árboles en futuras divisiones. 2) Que se generan dos nuevas ramas del árbol en cada división.

El proceso se inicia en lo más alto del árbol, donde todas las observaciones pertenecen a la misma región. Se identifican todos los posibles puntos de corte *s* para todos los predictores (X_1, X_2,..., X_p). Con cada una de las posibles combinaciones (predictor + punto de corte), se calcula la disminución en *RSS* que se consigue si se escoge esa división. Una vez evaluadas se selecciona el predictor X_j y punto de corte S con los que la disminución es mayor. En términos matemáticos, se busca el predictor *j* y el valor *s* que minimizan la ecuación

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

donde el primer término es el *RSS* de la región 1 y el segundo término es el *RSS* de la región 2, siendo cada una de las regiones el resultado de separar las observaciones acorde a *j* y *s*.

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

En la siguiente iteración se repite el proceso, pero esta vez de forma separada para cada una de las regiones que se han creado en el paso anterior. Las interaciones continúan hasta que se alcanza alguna norma de *stop*; como por ejemplo, que ninguna región contenga un mínimo de *n* observaciones, que el árbol tenga un máximo de nodos terminales, que la incorporación del nodo reduzca el error en al menos un % mínimo...

En la versión clásica de árboles de predicción, cada división evalúa un único predictor haciendo preguntas binarias (si, no). A pesar de que es posible evaluar divisiones más complejas, hacer una pregunta sobre múltiples variables a la vez es equivalente a hacer múltiples preguntas sobre variables individuales. Si en alguna iteración existen dos o más divisiones que consiguen la misma mejora, la elección entre ellas es aleatoria.

A modo de ejemplo ilustrativo, supóngase un escenario muy simplificado en el que se quiere predecir el salario de los jugadores en base a los años de experiencia y del número de bateos. El predictor *años* es de tipo ordinario y sus valores van de 0 a 10. El predictor *bateos* es también de tipo ordinario y va de 70 a 80. Para la primera división hay un total de 22 los posibles puntos de corte:

```
##  [1] "años < 0"    "años < 1"    "años < 2"    "años < 3"    "años < 4"
##  [6] "años < 5"    "años < 6"    "años < 7"    "años < 8"    "años < 9"
## [11] "años < 10"   "bateos < 70" "bateos < 71" "bateos < 72" "bateos < 73"
## [16] "bateos < 74" "bateos < 75" "bateos < 76" "bateos < 77" "bateos < 78"
## [21] "bateos < 79" "bateos < 80"
```

Se calcula el *RSS* obtenido con cada uno de los posibles puntos de corte y se selecciona el que consigue menor *RSS*, supóngase que es *años < 4*. En el siguiente paso se repite el proceso pero esta vez de forma separada para cada región. En la región 1 los posibles puntos de corte serán:

```
##  [1] "años < 0"    "años < 1"    "años < 2"    "años < 3"    "bateos < 70"
##  [6] "bateos < 71" "bateos < 72" "bateos < 73" "bateos < 74" "bateos < 75"
## [11] "bateos < 76" "bateos < 77" "bateos < 78" "bateos < 79" "bateos < 80"
```

Y en la región 2:

```
##  [1] "años < 4"    "años < 5"    "años < 6"    "años < 7"    "años < 8"
##  [6] "años < 9"    "años < 10"   "bateos < 70" "bateos < 71" "bateos < 72"
## [11] "bateos < 73" "bateos < 74" "bateos < 75" "bateos < 76" "bateos < 77"
## [16] "bateos < 78" "bateos < 79" "bateos < 80"
```

Este proceso se repite hasta alcanzar una determinada condición de *stop*.

Tree pruning

El proceso de construcción de árboles descrito en la sección anterior tiende a reducir rápidamente el *training RSS*, es decir, se ajusta muy bien a las observaciones empleadas como entrenamiento. Como consecuencia, se genera un *overfit* del modelo que reduce su capacidad predictiva al aplicarlo a nuevos datos. La razón de este comportamiento radica en la facilidad con la que los árboles se ramifican adquiriendo estructuras complejas.

Una estrategia que consigue evitar el exceso de complejidad consiente en generar árboles grandes para después podarlos (*pruning*), y así mantener únicamente la estructura robusta que consigue un *test error* bajo. La selección del *sub-árbol* óptimo puede hacerse mediante *cross-validation*, sin embargo, no suele ser posible estimar el *RSS* de todas las posibles sub-estructuras que se pueden generar a partir de un árbol medianamente grande. Por esta razón, se recurre al *cost complexity pruning* o *weakest link pruning*. *Cost complexity pruning* es un método de penalización de tipo *Loss + Penalty*, similar al empleado en *ridge regression* o *lasso*. En este caso, se busca el *sub-árbol T* que minimiza la ecuación:

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

donde $|T|$ es el número de nodos terminales del árbol. El primer término de la ecuación se corresponde con el sumatorio total de los residuos cuadrados *RSS*. Por definición, cuantos más nodos terminales tenga el modelo menor será esta parte de la ecuación. El segundo término es la restricción, que penaliza al modelo en función del número de nodos terminales (a mayor número penalización). El grado de penalización se determina mediante el *tunning parameter*

α. Cuando $\alpha = 0$, la penalización es nula y el árbol resultante es equivalente al árbol original. A medida que se incrementa α la penalización es mayor y, como consecuencia, los árboles resultantes son de menor tamaño. El valor óptimo de α puede identificarse mediante *cross validation*.

ALGORITMO PARA CREAR UN ÁRBOL DE REGRESIÓN CON PRUNING

1. Se emplea *recursive binary split* para crear un árbol grande y complejo (T_o) empleando los datos de *training*. El proceso se finaliza cuando cada nodo contiene menos de un número *n* de observaciones establecido por el usuario u otra norma de *stop*.

2. Se aplica el *cost complexity pruning* al árbol T_o para obtener el mejor *sub-árbol* en función de α. Es decir, se obtiene el mejor *sub-árbol* para un rango de valores de α.

3. Identificación del valor óptimo de α mediante *k-cross-validation*. Se divide el *training data set* en K grupos. Para $k = 1, ..., k = K$:
    a. Repetir pasos 1 y 2 empleando todas las observaciones excepto las del grupo k_i.
    b. Evaluar el *mean squared error MSE* para el rango de valores de α empleando el grupo k_i.
    c. Obtener el promedio de los K *mean squared error* calculados para cada valor α.

4. Seleccionar el *sub-árbol* del paso 2 que se corresponde con el valor α que ha conseguido el menor *cross-validation mean squared error*.

Ejemplo

El set de datos `Boston` *contiene información sobre viviendas de la ciudad de Boston así como información sobre el barrio en el que se encuentran. Se pretende ajustar un árbol de regresión que permita predecir el precio medio de una vivienda (*`medv`*) en función de las variables disponibles.*

```
library(MASS)
data("Boston")
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio  black
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
```

```
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
##   lstat medv
## 1  4.98 24.0
## 2  9.14 21.6
## 3  4.03 34.7
## 4  2.94 33.4
## 5  5.33 36.2
## 6  5.21 28.7
```

La función `tree()` del paquete `tree` ajusta árboles de predicción. La elección entre árbol de regresión o árbol de clasificación se hace automáticamente dependiendo de si la variable respuesta es de tipo `numeric` o `factor`. Es importante tener en cuenta que solo estos dos tipos de vectores están permitidos, si se pasa uno de tipo `character` se devuelve un error. Su sintaxis es muy similar a la empleada en la función `lm()`.

A continuación se ajusta un árbol de regresión empleando como variable respuesta `medv` y como predictores todas las variables disponibles. La función `tree()` crece el árbol hasta que encuentra una condición de *stop*. Por defecto, estas condiciones son:

- `mincut=5`: Número mínimo de observaciones que debe de tener al menos uno de los nodos hijos para que se produzca la división.
- `minsize=10`: Número mínimo de observaciones que debe de tener un nodo para que pueda dividirse.
- `deepth=31`: Profundidad máxima que puede alcanzar el árbol.

Esto implica que, no todas las variables pasadas como predictores en el argumento `formula`, necesariamente tienen que acabar incluidas en el árbol.

Como en cualquier estudio de regresión, no solo es importante ajustar el modelo, sino también cuantificar su capacidad para predecir nuevas observaciones. Para poder hacer la posterior evaluación, se dividen los datos en dos grupos, uno se emplea como *training data set* y el otro como *test data set*.

```
library(tree)
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)
arbol_regresion <- tree(formula = medv ~ ., data = Boston, subset = train)
summary(arbol_regresion)
```

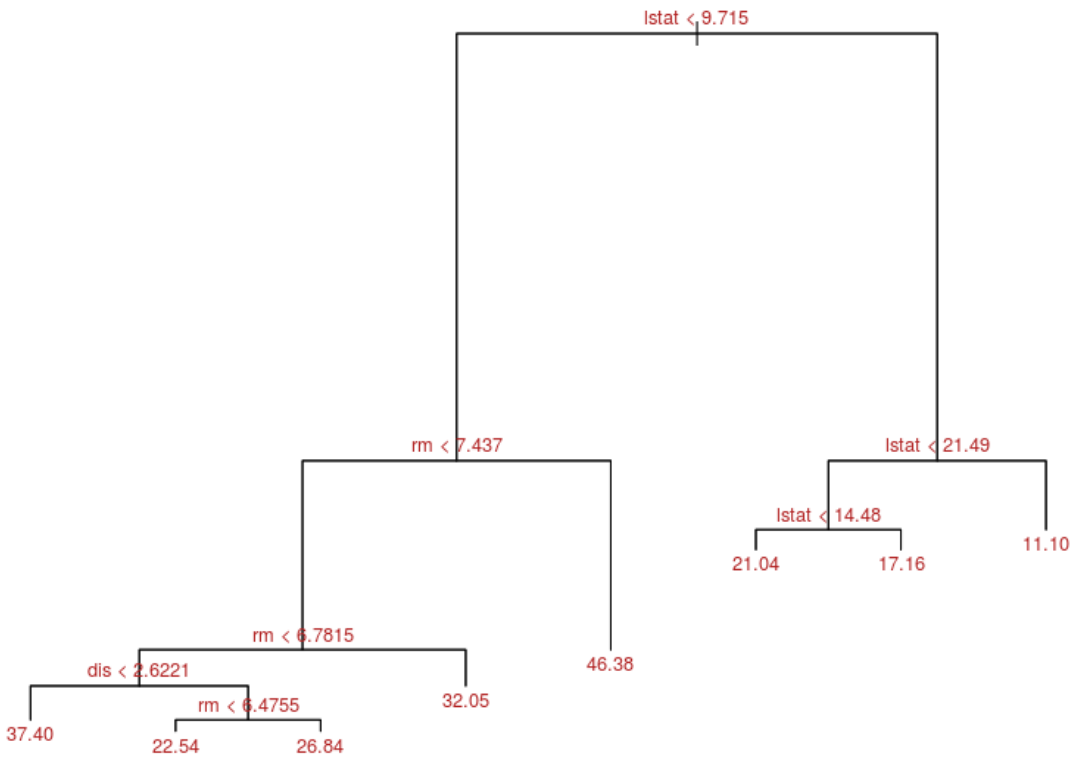
```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
```

```
## Variables actually used in tree construction:
## [1] "lstat" "rm"    "dis"
## Number of terminal nodes:  8
## Residual mean deviance:  12.65 = 3099 / 245
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -14.10000  -2.04200  -0.05357   0.00000   1.96000  12.60000
```

La función `summary()` muestra que el árbol ajustado tiene un total de 8 nodos terminales y que se han empleado como predictores las variables `lstat`, `rm` y `dis`. En el contexto de árboles de regresión, el termino *Residual mean deviance* es simplemente la suma de cuadrados residuales MSE dividida entre (número de observaciones - número de nodos terminales). Cuanto menor es la *deviance*, mejor es el ajuste del árbol a las observaciones de entrenamiento.

Una vez creado el árbol, se puede representar mediante la combinación de las funciones `plot()` y `text()`. La función `plot()` dibuja la estructura del árbol: las ramas y los nodos. Mediante su argumento `type` se puede especificar si se quiere que todas las ramas tengan el mismo tamaño (`type="uniform"`) o que su longitud sea proporcional a la reducción de impureza (heterogeneidad) de los nodos terminales (`type="proportional"`). Esta segunda opción permite identificar visualmente el impacto de cada división en el modelo. La función `text()` añade la descripción de cada nodo interno y el valor de cada nodo terminal.

```
plot(x = arbol_regresion, type = "proportional")
text(x = arbol_regresion, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```



La variable *lstat*, que mide el porcentaje de personas en estado de pobreza en la zona, ha resultado ser el predictor más importante (primer nodo). El árbol indica que, valores bajos de *lstat*, se corresponden con precios de vivienda más elevados. Por ejemplo, el modelo predice un precio de 46380 dólares para viviendas que están en una zona con un porcentaje $lstat < 9.715$ y un número de habitaciones $rm \geq 7.437$.

Para obtener una descripción más detallada del árbol, se puede imprimir el objeto por pantalla. R muestra el criterio de división de cada nodo, el número de observaciones que hay en esa rama (antes de dividirse), la *deviance*, la predicción promedio de esa rama (en el caso de clasificación se muestra el grupo más frecuente) y la proporción de cada grupo. Cuando se trata de nodo terminal, se indica con un asterisco.

```
arbol_regresion
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 253 20890.0 22.67
```

```
##    2) lstat < 9.715 103  7765.0 30.13
##      4) rm < 7.437 89  3310.0 27.58
##        8) rm < 6.7815 61  1995.0 25.52
##         16) dis < 2.6221 5   615.8 37.40 *
##         17) dis > 2.6221 56   610.3 24.46
##           34) rm < 6.4755 31   136.4 22.54 *
##           35) rm > 6.4755 25   218.3 26.84 *
##        9) rm > 6.7815 28   496.6 32.05 *
##      5) rm > 7.437 14   177.8 46.38 *
##    3) lstat > 9.715 150  3465.0 17.55
##      6) lstat < 21.49 120  1594.0 19.16
##       12) lstat < 14.48 62   398.5 21.04 *
##       13) lstat > 14.48 58   743.3 17.16 *
##      7) lstat > 21.49 30   311.9 11.10 *
```

PRUNING

Con la finalidad de reducir la varianza del modelo y así disminuir el *test error*, se somete al árbol a un proceso de *pruning*. Tal como se describió con anterioridad, el proceso de *pruning* intenta encontrar el árbol más sencillo (menor tamaño) que consigue explicar las observaciones. La función `cv.tree()` emplea *cross validation* para identificar el valor óptimo de penalización α. Por defecto, esta función emplea la *deviance* para guiar el proceso de *pruning*.

```
set.seed(3)
cv_arbol <- cv.tree(arbol_regresion, K = 10)
cv_arbol
```

```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  5335.721  5988.901  6671.685  7349.550  7388.893  8812.276 14707.294
## [8] 21035.642
##
## $k
## [1]      -Inf  255.6581  451.9272  768.5087  818.8885 1559.1264 4276.5803
## [8] 9665.3582
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

El objeto devuelto por `cv.tree()` contiene:

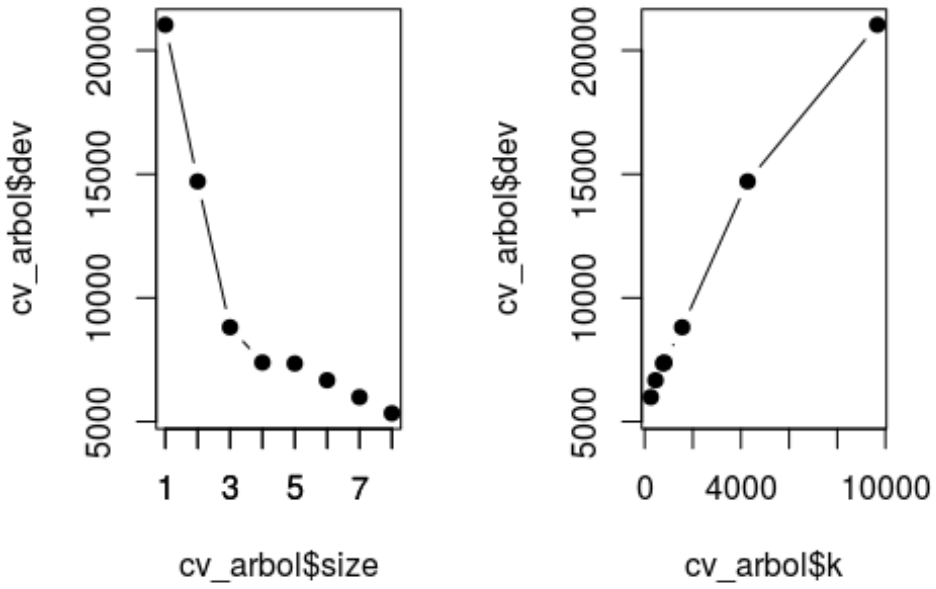
- Size: el tamaño (número de nodos terminales) de cada árbol.
- Dev: la estimación de *cross-validation test error* para cada tamaño de árbol.
- K: El rango de valores de penalización α evaluados.

El objetivo del proceso es encontrar el valor α con el que se consigue el menor *cross-validation test error*. Dado que α es quien determina el tamaño del árbol, la frase anterior equivale a decir que se busca el tamaño del árbol que minimiza el *cross-validation test error*. En este caso, el mejor árbol está formado por 8 nodos, por lo que no es necesario reducir el tamaño original.

```
par(mfrow = c(1, 2))
cv_arbol$size[which.min(cv_arbol$dev)]
```

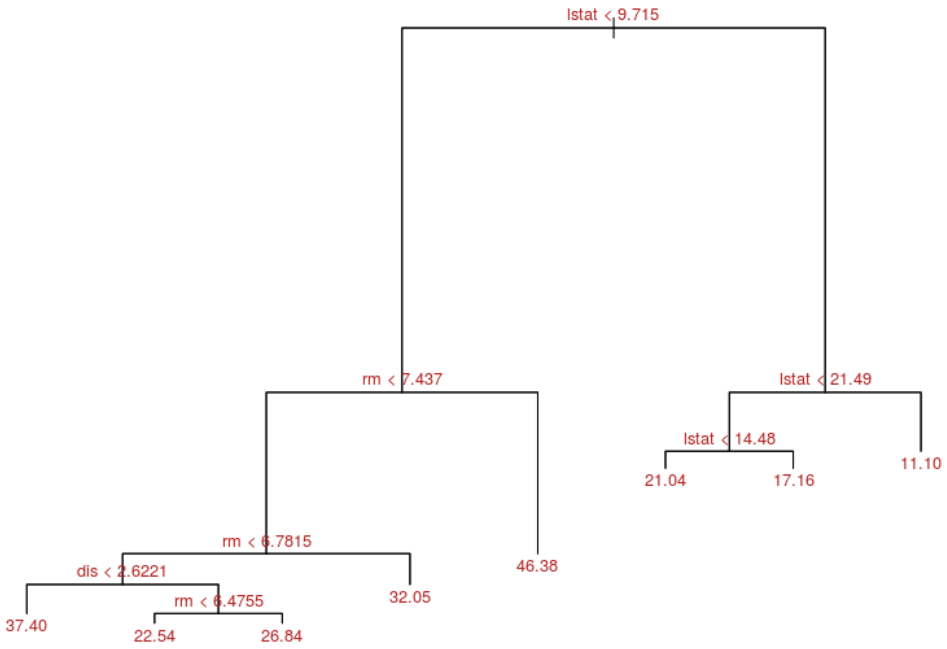
```
## [1] 8
```

```
plot(x = cv_arbol$size, y = cv_arbol$dev, type = "b", pch = 19)
axis(side = 1, at = seq(from = min(cv_arbol$size), to = max(cv_arbol$size)))
plot(x = cv_arbol$k, y = cv_arbol$dev, type = "b", pch = 19)
```



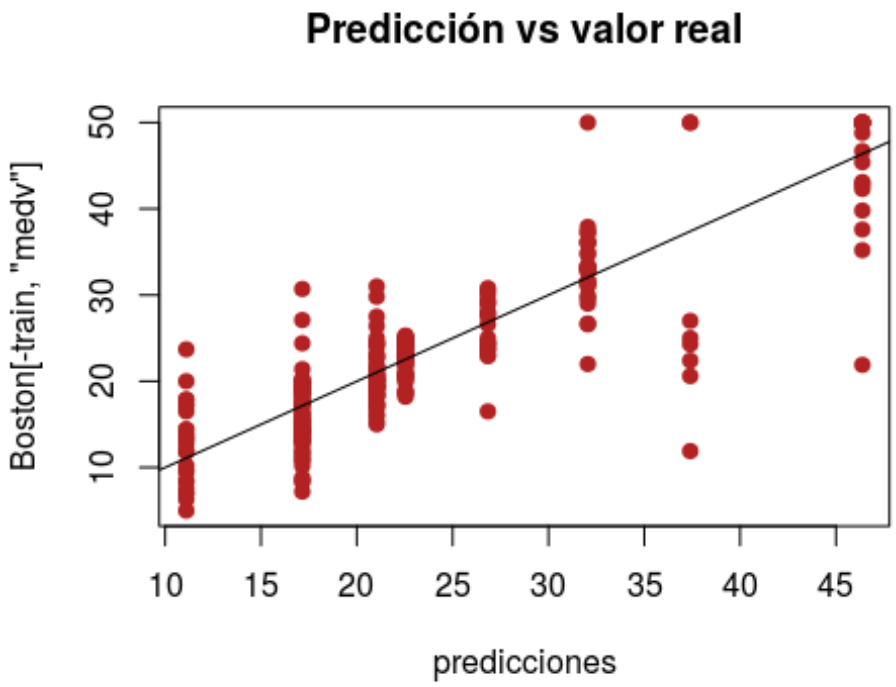
Con la función `prune.tree()` se obtiene el árbol de regresión del tamaño identificado como óptimo. Ha esta función también se le puede indicar el valor de α óptimo en lugar del tamaño.

```
arbol_pruning <- prune.tree(tree = arbol_regresion, best = 8)
plot(x = arbol_pruning, type = "proportional")
text(x = arbol_pruning, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```



Por último, se evalúa la precisión del árbol empleando el *test data set*.

```
predicciones <- predict(arbol_pruning, newdata = Boston[-train, ])
plot(predicciones, Boston[-train, "medv"], col = "firebrick",
     main = "Predicción vs valor real", pch = 19)
abline(0, 1)
```




```
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
test_mse
```

```
## [1] 25.04559
```

El *Mean Square Test Error* asociado con el árbol de regresión es de 25.05 unidades. La raíz cuadrada del *Mean Square Test Error* es 5.005, lo que significa que las predicciones se alejan en promedio 5.005 unidades (5005 dólares) del valor real.

Árboles de clasificación

Los árboles de clasificación se asemejan mucho a los árboles de regresión, excepto que se emplean para predecir una variable respuesta cualitativa en lugar de una variable respuesta continua. En los árboles de regresión, el valor predicho para una observación es la media de la variable respuesta de todas las observaciones que pertenecen al mismo nodo terminal. En los árboles de clasificación, la clase predicha se corresponde con la clase más frecuente en el nodo terminal al que pertenece la observación en cuestión. A la hora de interpretar un árbol de

clasificación, además de la clase más frecuente, suele ser interesante mostrar el porcentaje de cada una.

Para construir un árbol de clasificación se emplea el mismo método *recursive binary split* que en los árboles de regresión. Sin embargo, no es posible emplear el *RSS* como criterio de selección en las divisiones binarias. Las alternativas más empleadas son:

CLASSIFICATION ERROR RATE

Se define como la proporción de observaciones que no pertenecen a la clase más común de la región (nodo).

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

donde $\hat{p}_{mk}$ representa la proporción de observaciones de la región m que pertenecen a la clase k. A pesar de la sencillez de esta medida, no es suficientemente sensible para crear buenos árboles, por lo que en la práctica suelen emplearse otras dos medidas.

GINI INDEX

Es una medida de la varianza total en el conjunto de las K clases de la región (nodo) m. Se considera una medida de "pureza" del nodo.

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

Cuando $\hat{p}_{mk}$ es cercano a 0 o a 1 (la región contiene mayoritariamente observaciones de una clase), el término $\hat{p}_{mk}\,(1 - \hat{p}_{mk})$ es muy pequeño. Como consecuencia, cuanto mayor sea la homogeneidad interna del nodo, menor el valor de G.

CROSS ENTROPY

Al igual que el índice *Gini*, la *cross-entropy* toma valores pequeños cuanto más homogéneo (puro) es el nodo.

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

Para el proceso de construcción del árbol, acorde al libro *Introduction to Statistical Learning*, tanto el *Gini index* como *cross-entropy* son más adecuados que el *classification error rate* debido a su mayor sensibilizad a la homogeneidad de los nodos. Para el proceso de *pruning* los tres son adecuados, aunque, si el objetivo es conseguir la máxima precisión en las predicciones, mejor emplear el *classification error rate*.

Ejemplo

El set de datos `Carseats` *contiene información sobre la venta de sillas infantiles en 400 tiendas distintas. Para cada una de las 400 tiendas se han registrado 11 variables. Se pretende generar un modelo de clasificación que permita predecir si una tienda tiene ventas altas (*Sales > 8*) o bajas (*Sales <= 8*) en función de todas las variables disponibles.*

```
library(ISLR)
data("Carseats")
head(Carseats)
```

```
##   Sales CompPrice Income Advertising Population Price ShelveLoc Age
## 1  9.50       138     73          11        276   120       Bad  42
## 2 11.22       111     48          16        260    83      Good  65
## 3 10.06       113     35          10        269    80    Medium  59
## 4  7.40       117    100           4        466    97    Medium  55
## 5  4.15       141     64           3        340   128       Bad  38
## 6 10.81       124    113          13        501    72       Bad  78
##   Education Urban  US
## 1        17   Yes Yes
## 2        10   Yes Yes
## 3        12   Yes Yes
## 4        14   Yes Yes
## 5        13   Yes  No
## 6        16    No Yes
```

`Sales` es una variable continua. Como el objetivo del estudio es clasificar las tiendas según si venden mucho o poco, se crea una nueva variable dicotómica (Si, No) llamada `ventas_altas`.

```
Carseats$ventas_altas <- ifelse(test = Carseats$Sales > 8, yes = "Si", no = "No")
```

A continuación se ajusta un árbol de clasificación empleando como variable respuesta `ventas_altas` y como predictores todas las variables disponibles excepto `Sales`.

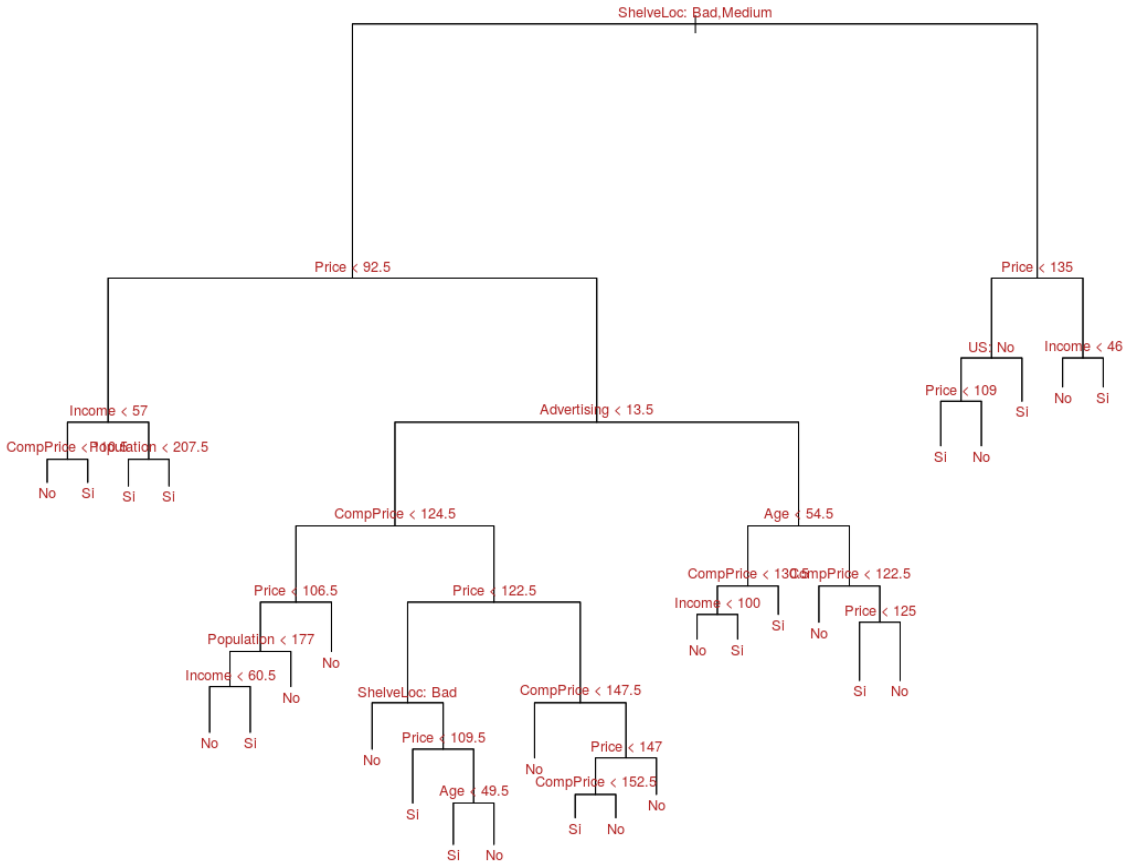
```
library(tree)
# Conversión de la variable respuesta a tipo factor
Carseats$ventas_altas <- as.factor(Carseats$ventas_altas)

arbol_clasificacion <- tree(formula = ventas_altas ~ CompPrice + Income +
                            Advertising + Population + Price + ShelveLoc + Age +
                            Education + Urban + US, data = Carseats)
# equivalente a tree( ventas_altas ~ . - Sales, Carseats)
summary(arbol_clasificacion)
```

```
##
## Classification tree:
## tree(formula = ventas_altas ~ CompPrice + Income + Advertising +
##     Population + Price + ShelveLoc + Age + Education + Urban +
##     US, 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 función `summary()` muestra que el árbol ajustado ha incluido todos los predictores disponibles, que tiene un total de 27 nodos terminales y un *classification error rate* del 9%. La *deviance* de los árboles de clasificación se calcula como $-2\sum_m\sum_k n_{mk}\, log(\hat{p}_{mk})$, donde n_{mk} es el número de observaciones en el nodo terminal m que pertenecen a la clase k. El termino *Residual mean deviance* mostrado en el summary es simplemente la *deviance* dividida entre (número de observaciones - número de nodos terminales). Cuanto menor es la *deviance* mejor es el ajuste del árbol a las observaciones de entrenamiento.

```
plot(x = arbol_clasificacion, type = "proportional")
text(x = arbol_clasificacion, splits = TRUE, pretty = 0, cex = 0.8,
     col = "firebrick")
```



Analizando la representación del árbol, el predictor más influyente sobre las ventas es el lugar que ocupa el producto en los estantes de las tiendas (*ShelveLoc*). Este hecho queda reflejado en la primera división del árbol, que separa las posiciones buenas de las malas e intermedias.

```
arbol_clasificacion
```

```
## 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 Si ( 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 Si ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Si ( 0.19444 0.80556 )
```

```
##          18) Population < 207.5 16  21.170 Si ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Si ( 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 Si ( 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 Si ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Si ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Si ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Si ( 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 Si ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Si ( 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 Si ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Si ( 0.20000 0.80000 )
##            44) CompPrice < 130.5 14  18.250 Si ( 0.35714 0.64286 )
##              88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##              89) Income > 100 5   0.000 Si ( 0.00000 1.00000 ) *
##            45) CompPrice > 130.5 11   0.000 Si ( 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 Si ( 0.00000 1.00000 ) *
##              95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##     3) ShelveLoc: Good 85  90.330 Si ( 0.22353 0.77647 )
##       6) Price < 135 68  49.260 Si ( 0.11765 0.88235 )
##        12) US: No 17  22.070 Si ( 0.35294 0.64706 )
##          24) Price < 109 8   0.000 Si ( 0.00000 1.00000 ) *
##          25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##        13) US: Yes 51  16.880 Si ( 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 Si ( 0.45455 0.54545 ) *
```

En la práctica, no solo es interesante ajustar el árbol sino también evaluar su capacidad predictiva mediante la estimación del *test error*. La forma más sencilla de hacerlo es dividiendo las observaciones disponibles en dos grupos, uno se emplea para ajustar el árbol *training data set* y el otro para evaluarlo *test data set*. La función `predict()` puede aplicarse a modelos de tipo árbol. En el caso de árboles de clasificación, hay que indicar el argumento `type="class"` para que R devuelva la clase predicha.

```
set.seed(2)
train <- sample(1:nrow(Carseats), nrow(Carseats)/2, replace = FALSE)
Carseats_train <- Carseats[train, ]
Carseats_test <- Carseats[-train, ]
arbol_clasificacion <- tree(formula = ventas_altas ~ . - Sales,
                            data = Carseats_train)
predicciones <- predict(arbol_clasificacion, newdata = Carseats_test,
                        type = "class")
table(predicciones, Carseats_test$ventas_altas)
```

```
##
## predicciones No Si
##           No 86 27
##           Si 30 57
```

El *test error* es 0.285. El modelo es capaz de predecir correctamente un 71.5 % de las observaciones del *test set*.

PRUNING

Con la finalidad de reducir la varianza del modelo y así disminuir el *test error*, se somete al árbol a un proceso de *pruning*. A diferencia del ejemplo anterior, al ser este un árbol de clasificación, se indica en la función `cv.tree()` que `FUN=prune.misclass`. De esta manera se emplea el *clasification error rate* para guiar el proceso de *pruning*.

```
set.seed(3)
cv_arbol <- cv.tree(arbol_clasificacion, FUN = prune.misclass, K = 10)
cv_arbol
```

```
## $size
## [1] 19 17 14 13  9  7  3  2  1
##
## $dev
## [1] 55 55 53 52 50 56 69 65 80
##
## $k
## [1]       -Inf  0.0000000  0.6666667  1.0000000  1.7500000  2.0000000
## [7]  4.2500000  5.0000000 23.0000000
```

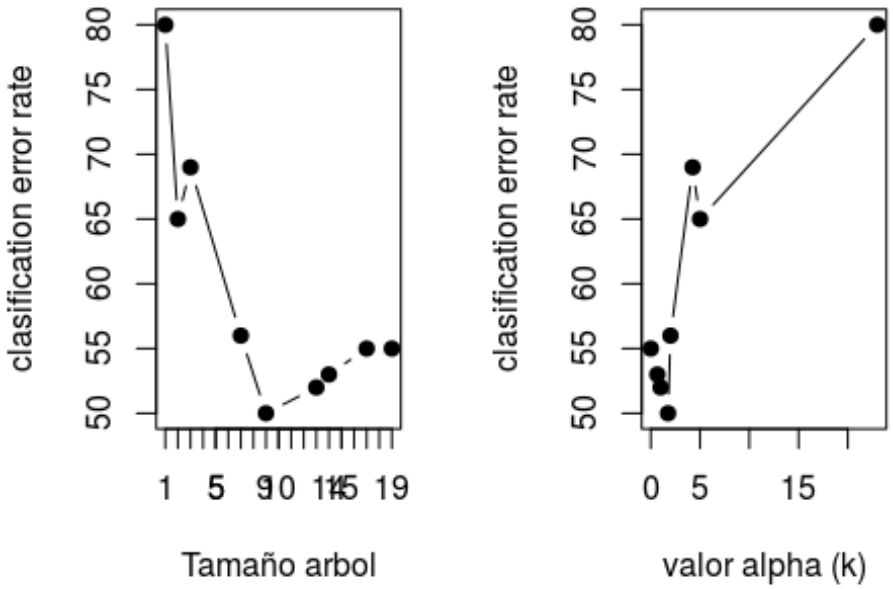
```
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
par(mfrow = c(1, 2))
cv_arbol$size[which.min(cv_arbol$dev)]
```

```
## [1] 9
```

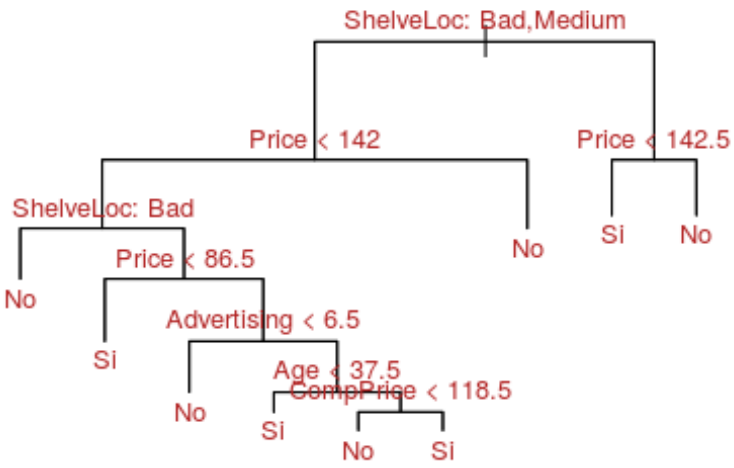
En este caso, un árbol con 9 nodos terminales minimiza el *test error*.

```
plot(x = cv_arbol$size, y = cv_arbol$dev, type = "b", pch = 19,
     xlab = "Tamaño arbol", ylab = "clasification error rate")
axis(side = 1, at = seq(from = min(cv_arbol$size), to = max(cv_arbol$size)))
plot(x = cv_arbol$k, y = cv_arbol$dev, type = "b", pch = 19,
     xlab = "valor alpha (k)", ylab = "clasification error rate")
```



Finalmente, con la función `prune.missclas()` se obtiene el árbol de clasificación del tamaño identificado como óptimo (no confundir con la función `prune.tree()` para árboles de regresión). Ha esta función también se le puede indicar el valor de *alpha* óptimo en lugar del tamaño.

```
arbol_pruning <- prune.misclass(tree = arbol_clasificacion, best = 9)
plot(x = arbol_pruning, type = "proportional")
text(x = arbol_pruning, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```




```
predicciones <- predict(arbol_pruning, newdata = Carseats_test, type = "class")
table(predicciones, Carseats_test$ventas_altas)
```

```
##
## predicciones No Si
##           No 94 24
##           Si 22 60
```

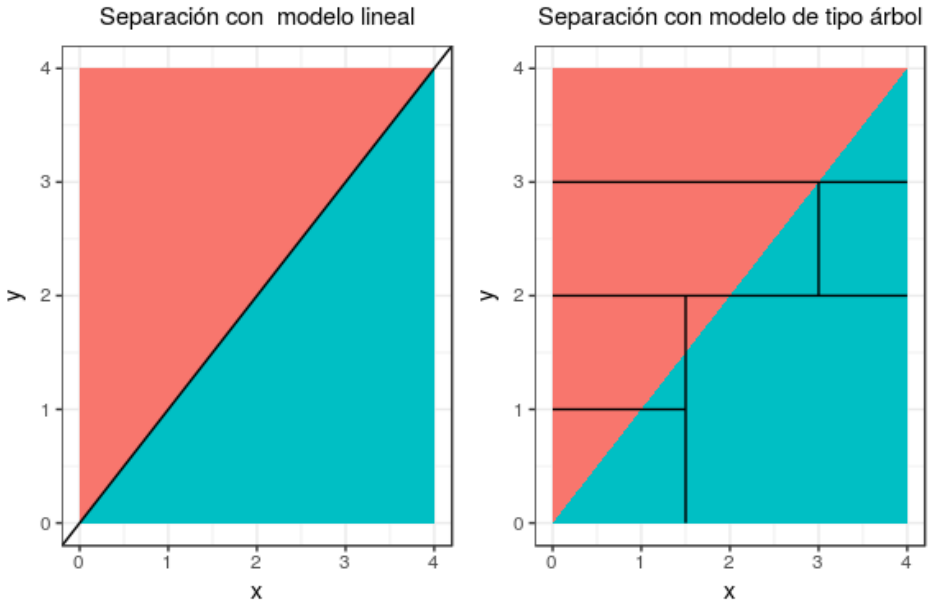
```
paste("El porcentaje de acierto es de", 100 * ((94 + 60)/(94 + 24 + 22 + 60)), "%")
```

```
## [1] "El porcentaje de acierto es de 77 %"
```

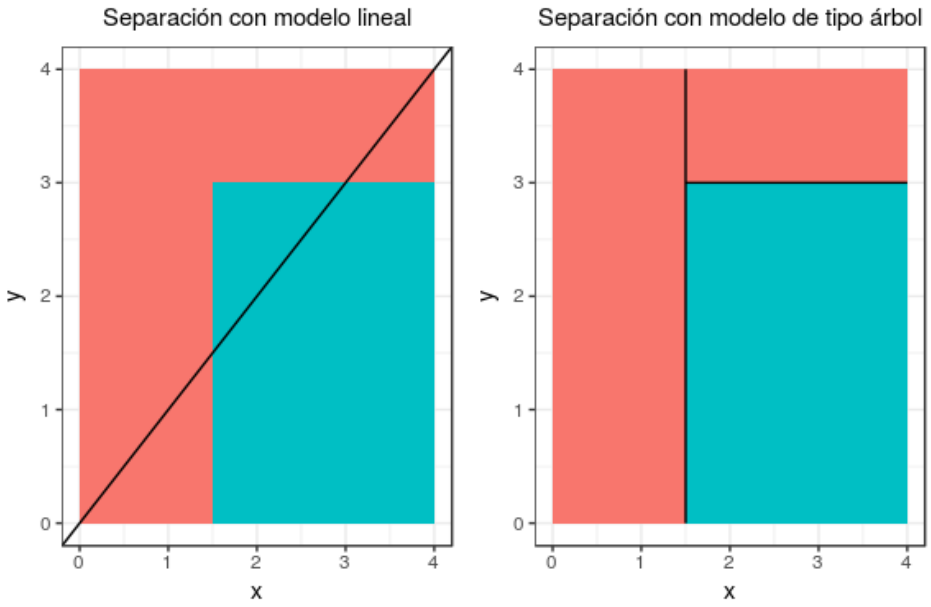
Gracias al proceso de *pruning* el *test error* se ha reducido y el modelo es capaz de predecir correctamente el 77% de las observaciones del *test set*.

Comparación de árboles frente a modelos lineales

La superioridad de los métodos basados en árboles frente a los métodos lineales depende del problema en cuestión. Cuando la relación entre los predictores y la variable respuesta es aproximadamente lineal, un modelo de tipo *regresión lineal* funciona bien y supera a los árboles de regresión.



Por contra, si la relación entre los predictores y la variable respuesta es de tipo no lineal y compleja, los métodos basados en árboles suelen superar a las aproximaciones clásicas.



El procedimiento adecuado para encontrar el mejor modelo (lineal, no lineal, árboles...) consiste en estudiar el problema en cuestión, seleccionar los modelos para los que se cumplen las condiciones y comparar la estimación del *test error* mediante *cross-validation* o validación simple.

Ventajas y desventajas de los árboles de regresión y clasificación

Ventajas

- Los árboles son muy fáciles de interpretar aun cuando las relaciones entre predictores son complejas.
- Los árboles de decisión se asemejan en mayor medida a la forma en que piensa el cerebro humano en comparación a los métodos de regresión lineal, polinómica, splines...
- Los modelos basados en árboles se pueden representar gráficamente aun cuando el número de predictores es mayor de 3.
- Los árboles pueden manejar fácilmente predictores cualitativos sin tener que crear variables *dummy*.
- Si para alguna observación el valor de un predictor no está disponible, a pesar de no poder llegar a ningún nodo terminal, se puede conseguir una predicción empleando todas las observaciones que pertenecen al nodo alcanzado. La precisión de la predicción se verá reducida pero al menos podrá obtenerse.

Desventajas

- La capacidad predictiva de los árboles de regresión y clasificación es inferior a la conseguida con otro tipo de modelos. Sin embargo, existen técnicas más complejas que, haciendo uso de la combinación de múltiples árboles, (*bagging*, *random forest*, *boosting*) consiguen mejorar en gran medida este problema.

Bagging

El término *bagging* o *bootstrap aggregation* hace referencia al empleo del muestreo repetido (*bootstrapping*) con el fin de reducir la varianza de algunos métodos de *statistical learning*, entre ellos los árboles de predicción.

Los árboles de regresión y clasificación descritos en los apartados anteriores tienen el inconveniente de sufrir una alta varianza. Esto significa que, dependiendo del conjunto de datos empleado para construir el árbol, se obtienen resultados muy distintos.

Dado un conjunto de observaciones independientes Z_1,..., Z_n, cada una con varianza σ^2, la varianza de la media de las observaciones $\overline{Z}$ es σ^2/n. En otras palabras, promediando un conjunto de observaciones se reduce su varianza. Basándose en esta idea, una forma de reducir la varianza y aumentar la precisión de un método predictivo es obtener múltiples *trainig sets* de la población, ajustar un modelo distinto con cada uno de ellos, y hacer la media de las predicciones resultantes. Si bien en la práctica no se suele tener acceso a múltiples *training sets*, recurriendo a *bootstrapping,* se pueden generar *pseudo-training sets* con los que ajustar diferentes modelos y después promediarlos. A este proceso se le conoce como *bagging* y es aplicable a una gran variedad de métodos de regresión.

En el caso particular de los árboles, *bagging* ha demostrado incrementar en gran medida la precisión de las predicciones. La forma de aplicarlo es ajustando B árboles utilizando B *pseudo-training sets*. Cada árbol se crea sin apenas restricciones y no se somete a *pruning*, por lo que tiene varianza alta pero poco bias. Al hacer el promedio de todos los árboles se consigue contrarrestar la varianza. Cuando se trata de árboles de clasificación, en los que la variable respuesta es cualitativa, no se puede hacer la media de las predicciones. En su lugar, una de las formas de proceder (hay otras) es registrar la clase predicha para cada observación en los B árboles y seleccionar como predicción final la más frecuente.

En el proceso de *bagging*, el número de árboles creados no es un parámetro crítico en cuanto a que, por mucho que se incremente el número, no se aumenta el riesgo de *overfiting*. Alcanzado un determinado número de árboles, la reducción de *test error* se estabiliza.

Out-of-Bag Error Estimation

Dada la naturaleza del proceso de *bagging*, resulta posible estimar de forma directa el *test error* sin necesidad de recurrir a *cross-validation* o a un *test set* de evaluación. Sin entrar en demostraciones matemáticas, el hecho de que los árboles se ajusten de forma repetida empleando muestras generadas por *boostrapping* conlleva que, en promedio, cada ajuste use solo aproximadamente dos tercios de las observaciones originales. Al tercio restante se le llama *out-of-bag* (*OOB*). Si para cada árbol ajustado en el proceso de *bagging* se registran las observaciones empleadas, se puede predecir la respuesta de la observación *i* haciendo uso de aquellos árboles en los que esa observación ha sido excluida (*OOB*) y promediándolos. En el caso de los árboles de clasificación, en lugar de la media se emplea la clase más frecuente. Siguiendo este proceso, se pueden obtener las predicciones para las *n* observaciones y con ellas obtener el *OOB-mean square error* (para regresión) o el *OOB-classification error* (para árboles de clasificación). Como la respuesta de cada observación se predice empleando únicamente los árboles en cuyo ajuste no participó dicha observación, el *OOB-error* sirve como estimación del *test-error*. De hecho, si el número de árboles es suficientemente alto, el *OOB-error* es prácticamente equivalente al *leave-one-out cross-validation error*.

Importancia de los predictores después de bagging

Si bien es cierto que el proceso de *bagging* consigue mejorar la capacidad predictiva en comparación a los modelos basados en un único árbol, esto tiene un coste asociado, la interpretabilidad del modelo se reduce. Tras combinar múltiples árboles ya no es posible obtener una representación gráfica sencilla del modelo y no es inmediato identificar que predictores son más importantes. Para lograr esto último, se puede almacenar de forma separada la reducción total en *RSS* debida a divisiones hechas en cada predictor y después promediar el valor de todos los árboles. Cuando mayor sea la reducción promedio conseguida por el predictor, mayor su importancia. En el caso de árboles de clasificación, el proceso es el mismo pero empleando el *Gini index* o *cross-entropy* en lugar de *RSS*.

Ejemplo regresión

El set de datos `Boston` *contiene información sobre viviendas de la ciudad de Boston así como información sobre el barrio en el que se encuentran. Se pretende generar un árbol de regresión que permita predecir el precio medio de una vivienda (medv) en función de las variables disponibles. Para mejorar la capacidad predictiva del modelo se recurre al proceso de bagging.*

La función `randomForest()` del paquete de R `randomForest` permite ajustar árboles de decisión mediante el proceso de *bagging* y *random forest*. La misma función puede emplearse para ambos procesos, ya que *bagging* es un caso particular de *random forest* en el que $m = p$. Con el argumento `mtry` se especifica el número de predictores que se tienen que considerar en cada división del árbol. Si su valor es igual al número total de predictores, el método aplicado es *bagging*.

```
library(MASS)
library(randomForest)
data("Boston")
ncol(Boston)
```

```
## [1] 14
```

```
# El data set tiene 14 columnas: 1 varible respuesta y 13 predictores
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)
arbol_bagging <- randomForest(medv ~ ., data = Boston, subset = train, mtry = 13,
                              importance = TRUE)
arbol_bagging
```

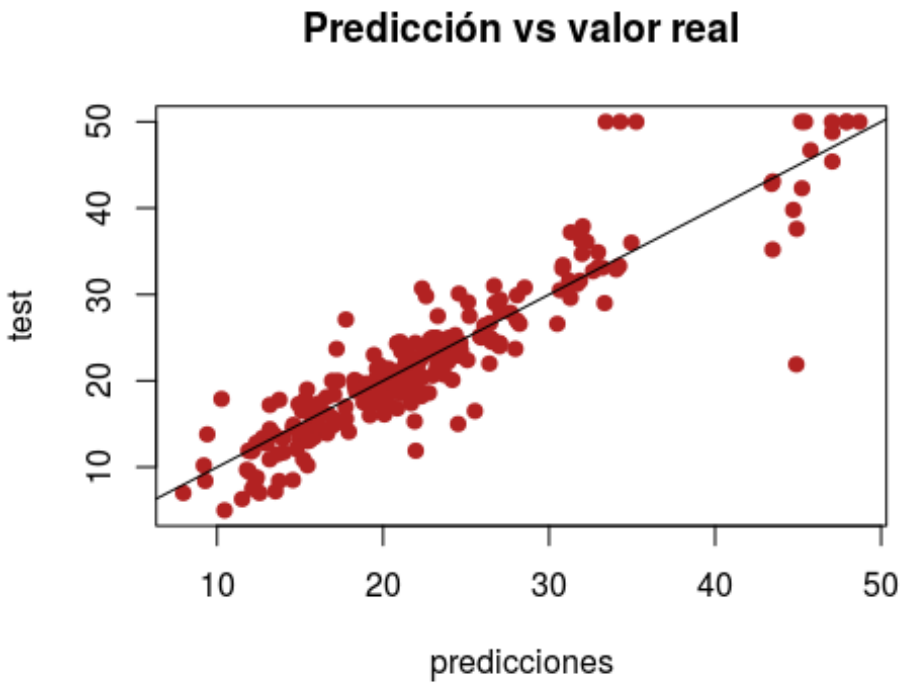
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE,
subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.08966
##                     % Var explained: 86.57
```

Al imprimir el resultado por pantalla se muestra: el número de pseudo-árboles generados en el proceso, el número de predictores evaluados en cada división (en este caso todos los disponibles), el *Mean of squared residuals* y el porcentaje de varianza que el modelo es capaz de explicar. El término *Mean of squared residuals* hace referencia al *Out-of-bag MSE*,

que como se ha indicado anteriormente puede interpretarse como una estimación del *test error*.

Para comprobar la precisión del modelo al aplicarlo a nuevos datos, se predicen las observaciones no utilizadas en el ajuste del modelo.

```
predicciones <- predict(object = arbol_bagging, newdata = Boston[-train, ])
plot(predicciones, Boston[-train, "medv"], col = "firebrick",
     main = "Predicción vs valor real", pch = 19, xlab = "predicciones",
     ylab = "test")
abline(0, 1)
```




```
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
test_mse
```

```
## [1] 13.33831
```

El *test-MSE* asociado al árbol de regresión obtenido por *bagging* es de 13.33, casi la mitad del obtenido anteriormente con un árbol generado por *pruning* (25.05). Esto pone de manifiesto la superioridad de los modelos obtenidos por *bagging* en comparación a los obtenidos por *pruning*.

Gracias a la estimación del *Out-of-bag MSE* y su similitud con el *leave one out error*, realmente no es necesario dividir las observaciones disponibles en *training* y *test* para evaluar

el modelo. Se pueden emplear todas ellas en el proceso de *bagging* y considerar el *Out-of-bag MSE* como una estimación de *test-MSE*.

```
arbol_bagging <- randomForest(medv ~ ., data = Boston, mtry = 13,
                              importance = TRUE)
arbol_bagging
```

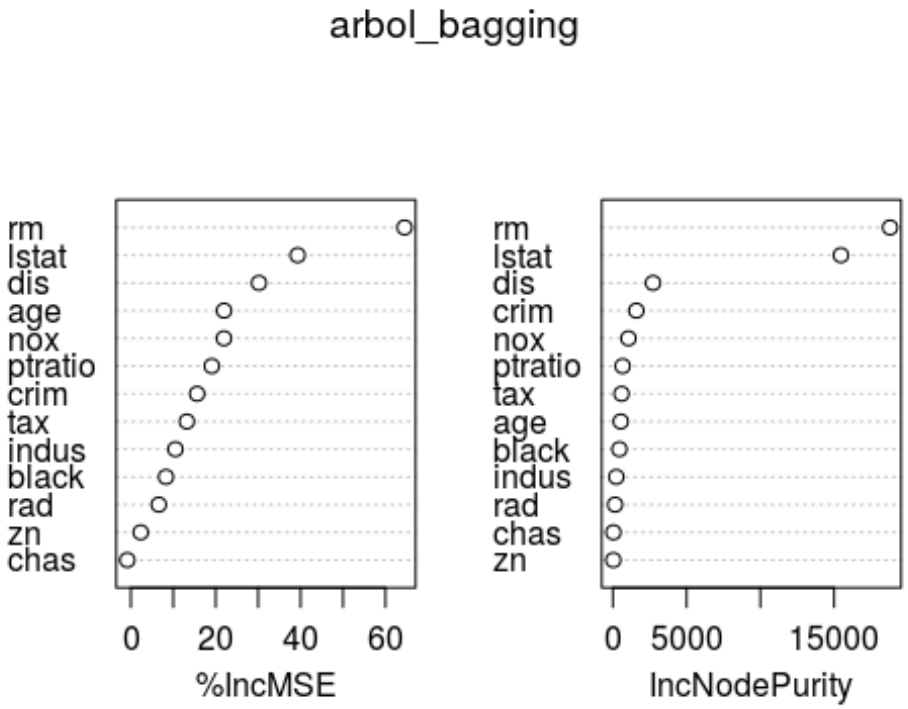
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 10.65845
##                     % Var explained: 87.37
```

A diferencia de los métodos basados en un único árbol, al combinar múltiples árboles se pierde la posibilidad de obtener una representación simple. Sin embargo, se puede evaluar la importancia de cada predictor mediante la función `importance()`. El resultado es una matriz que contiene dos medidas de importancia. *%IncMSE* cuantifica el incremento en *Out-of-bag-MSE* cuando el predictor se excluye del modelo, es decir, mide la reducción en la capacidad predictiva del modelo debido a su exclusión. *IncNodePurity* cuantifica el incremento total en la pureza de los nodos debido a divisiones en las que participa el predictor (promedio de todos los árboles).

```
importance(arbol_bagging)
```

```
##            %IncMSE IncNodePurity
## crim    15.6193035    1594.96350
## zn       2.3360992      34.90368
## indus   10.5054806     236.73931
## chas    -0.8184611      41.34098
## nox     21.9489714    1059.39138
## rm      64.4979984   18792.58558
## age     21.9701288     534.74002
## dis     30.1958008    2723.10831
## rad      6.5953549     152.59322
## tax     13.2077555     594.14821
## ptratio 19.0811020     663.26309
## black    8.2840367     446.45641
## lstat   39.3112216   15464.66585
```

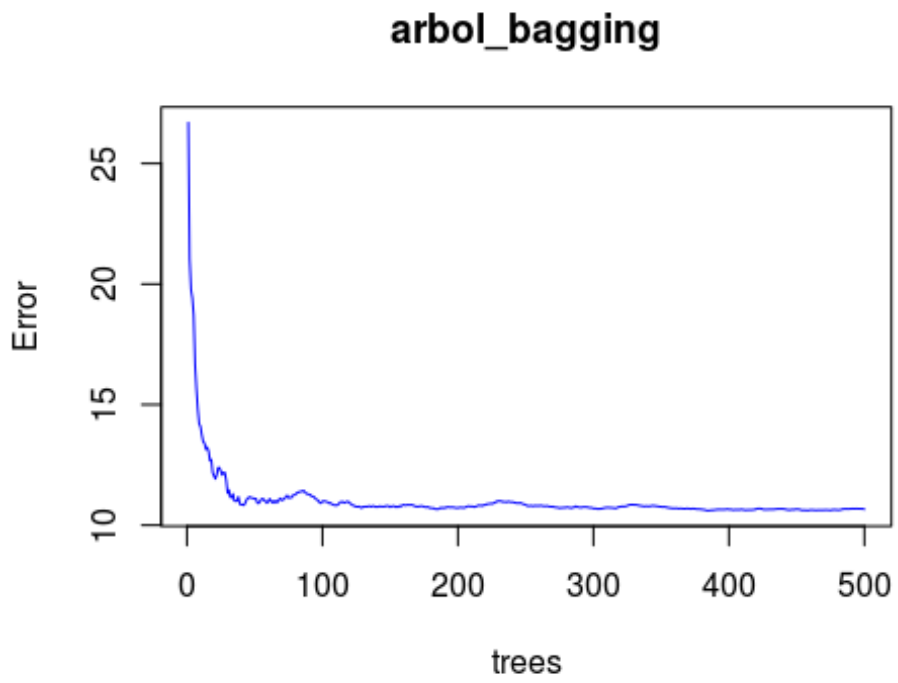
```
varImpPlot(arbol_bagging)
```



Los resultados muestran que, teniendo en cuenta todos los árboles generados en el proceso de *bagging*, las variables *lstat* y *rm* son con diferencia las más importantes en el modelo.

Por defecto, la función `randomForest()` crea 500 árboles. Este valor se puede modificar mediante el argumento `ntree`, siendo recomendable no sustituirlo por valores muy pequeños para que todas las observaciones participen un mínimo de veces en el proceso. Haciendo `plot()` de un objeto `randomForest` se puede visualizar la evolución del *out-of-bag-MSE* en función del número de árboles.

```
plot(arbol_bagging, col = "blue")
```



En este caso, a partir de aproximadamente 100 árboles la precisión del modelo se estabiliza. El *out-of-bag-MSE* que se obtiene si en lugar de 500 árboles se emplean solo 100 es casi el mismo.

```
arbol_bagging <- randomForest(medv ~ ., data = Boston, subset = train, mtry = 13,
                              ntree = 100, importance = TRUE)
arbol_bagging
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, ntree = 100,
importance = TRUE, subset = train)
##                Type of random forest: regression
##                      Number of trees: 100
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.05678
##                     % Var explained: 86.61
```

```
predicciones <- predict(object = arbol_bagging, newdata = Boston[-train, ])
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
test_mse
```

```
## [1] 13.68333
```

Random Forest

El método de *random forest* es una modificación del proceso de *bagging* que consigue mejores resultados gracias a que *decorrelaciona* los árboles generados en el proceso.

Recordando el apartado anterior, el proceso de *bagging* se basa en el hecho de que, promediando un conjunto de observaciones, se consigue reducir la varianza. Esto es cierto siempre y cuando las observaciones no estén correlacionadas. Si la correlación entre ellas es alta, la reducción de varianza que se puede lograr es pequeña. Supóngase un set de datos en el que hay un predictor muy influyente junto con otros moderadamente influyentes. En este escenario, todos o casi todos los árboles creados en el proceso de *bagging* tendrán el mismo predictor en la primera ramificación y serán muy parecidos entre ellos. Como consecuencia de la alta correlación entre los árboles, el proceso de *bagging* no conseguirá una mejora sustancial del modelo por disminución de varianza.

Random forest evita este problema haciendo una selección aleatoria de m predictores antes de evaluar cada división. De esta forma, un promedio de $(p - m)/p$ de las divisiones no contemplarán el predictor influyente, permitiendo que otros predictores tengas mayor posibilidad de ser seleccionados. Solo con añadir este paso extra se consigue *decorrelacionar* los árboles, de modo que el promedio de todos ellos sí puede maximizar la reducción de varianza.

Los métodos de *random forest* y *bagging* siguen el mismo algoritmo con la única diferencia de que, en *random forest*, antes de cada división se seleccionan aleatoriamente m predictores. La diferencia en el resultado dependerá del valor m escogido. Si $m = p$ los resultados de *random forest* y *bagging* son equivalentes. Un valor recomendado es $m \approx \sqrt{p}$, siendo p el número de predictores totales. Cuando los predictores están muy correlacionados, valores pequeños de m consiguen mejores resultados.

Al igual que ocurre con *bagging*, *random forest* no sufre problemas de *overfit* por aumentar el número de pseudo-árboles creados en el proceso. Alcanzado un determinado número de árboles, la reducción de *test error* se estabiliza.

Ejemplo regresión

Tal como se ha descrito en el ejemplo anterior, se pueden generar árboles por *random forest* con la función `randomForest()` de la misma forma que *bagging*. La única diferencia es que, en el argumento `mtry`, se indica un número menor que el total de predictores. Por defecto,

la función emplea $p/3$ predictores cuando aplica *random forest* a árboles de regresión y $\sqrt{p}$ cuando son árboles de clasificación.

```
arbol_randomforest <- randomForest(medv ~ ., data = Boston, subset = train,
                                   mtry = 5, ntree = 500, importance = TRUE)
arbol_randomforest
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 5, ntree = 500,
importance = TRUE, subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 12.46168
##                     % Var explained: 84.91
```

```
# Se ha especificado que se evalúen 5 predictores aleatorios en cada  ramificación
predicciones <- predict(object = arbol_randomforest, newdata = Boston[-train, ])
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
test_mse
```

```
## [1] 11.65919
```

```
# Empleando todas las observaciones en el proceso de randomforest

arbol_randomforest <- randomForest(medv ~ ., data = Boston, mry = 5, ntree = 500,
                                   importance = TRUE)
arbol_randomforest
```

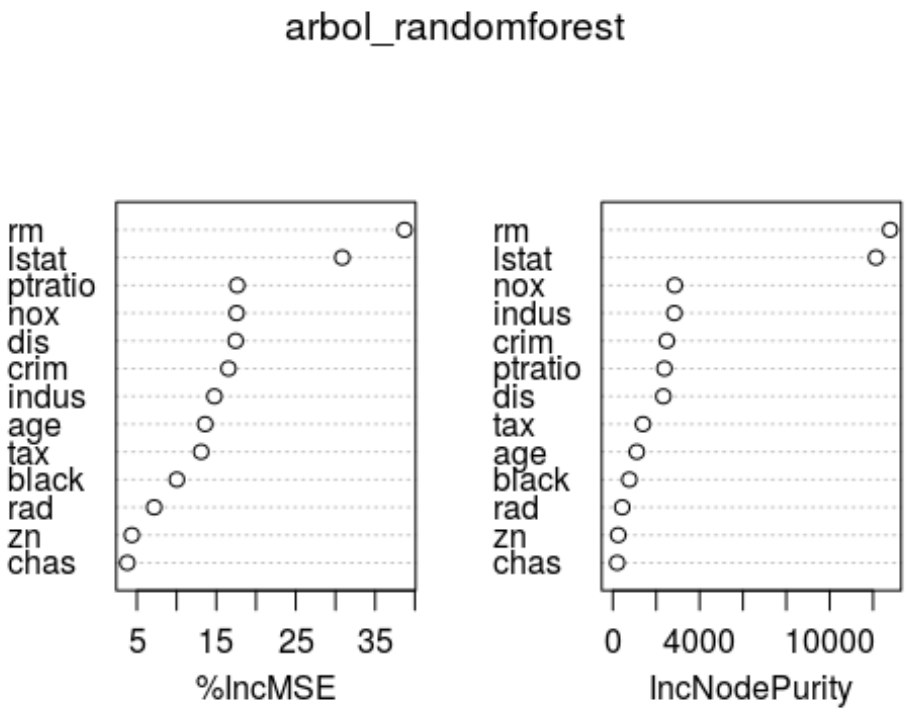
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mry = 5, ntree = 500,
importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 9.848425
##                     % Var explained: 88.33
```

El *test MSE* (11.659) obtenido con el *test set* y el *out-of-bag-MSE* (10.03) estimado empleando todas las observaciones, son similares y ambos menores que los obtenido por *bagging* (13.3 y 10.6). El método *random forest* suele superar al de *bagging*.

```
importance(arbol_randomforest)
```

```
##           %IncMSE IncNodePurity
## crim    16.525555     2490.0791
## zn       4.363408      246.9271
## indus   14.763598     2837.0534
## chas     3.798017      200.2440
## nox     17.554201     2856.5444
## rm      38.701956    12784.0860
## age     13.596186     1104.8145
## dis     17.440580     2328.9725
## rad      7.178942      433.5545
## tax     13.092647     1390.1648
## ptratio 17.637946     2380.2464
## black   10.033578      761.5189
## lstat   30.862249    12136.4284
```

```
varImpPlot(arbol_randomforest)
```



Al igual que con *bagging*, las variables *lstat* y *rm* son con diferencia las más importantes en el modelo.

Ejemplo clasificación

El set de datos `fgl` del paquete `MASS` contiene información sobre 214 muestras de cristal. Para cada una de ellas se han registrado 9 variables relacionadas con su composición y una (type) que indica su procedencia. Se desea ajustar un árbol de clasificación mediante random forest que permita predecir la procedencia del cristal en función de su composición.

```
library(MASS)
library(randomForest)
data("fgl")
head(fgl)
```

```
##      RI    Na   Mg   Al    Si    K   Ca Ba   Fe type
## 1  3.01 13.64 4.49 1.10 71.78 0.06 8.75  0 0.00 WinF
## 2 -0.39 13.89 3.60 1.36 72.73 0.48 7.83  0 0.00 WinF
## 3 -1.82 13.53 3.55 1.54 72.99 0.39 7.78  0 0.00 WinF
## 4 -0.34 13.21 3.69 1.29 72.61 0.57 8.22  0 0.00 WinF
## 5 -0.58 13.27 3.62 1.24 73.08 0.55 8.07  0 0.00 WinF
## 6 -2.04 12.79 3.61 1.62 72.97 0.64 8.07  0 0.26 WinF
```

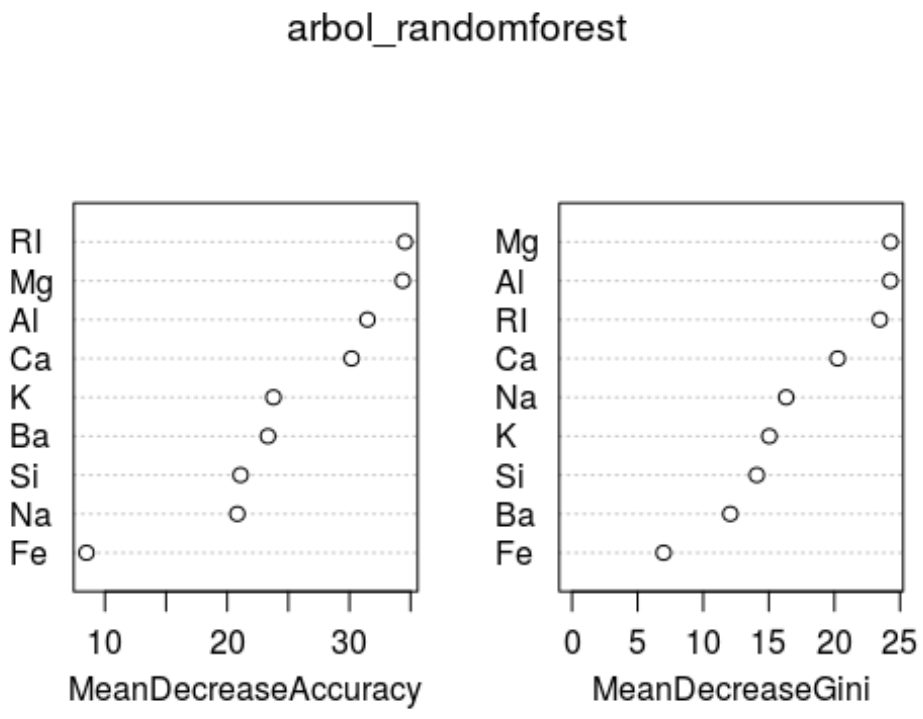
```
set.seed(17)
arbol_randomforest <- randomForest(type ~ ., data = fgl, mtry = 2, ntree = 500,
                                   importance = TRUE)
arbol_randomforest
```

```
##
## Call:
##  randomForest(formula = type ~ ., data = fgl, mtry = 2, ntree = 500,
importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 19.16%
## Confusion matrix:
##       WinF WinNF Veh Con Tabl Head class.error
## WinF    63     6   1   0    0    0   0.1000000
## WinNF   10    61   1   2    1    1   0.1973684
## Veh      8     2   7   0    0    0   0.5882353
## Con      0     2   0  10    0    1   0.2307692
## Tabl     0     2   0   0    7    0   0.2222222
## Head     1     3   0   0    0   25   0.1379310
```

```
importance(arbol_randomforest)
```

```
##         WinF     WinNF       Veh        Con      Tabl      Head
## RI 31.357457 21.180806  8.362283  6.9591617  6.221309  6.479567
## Na 12.171962  9.542229  2.108458 10.0087700 11.883741 12.627974
## Mg 24.651815 24.721311 10.098428 19.9605662 14.538549 17.314601
## Al 25.107118 17.080762  7.210349 15.1273914 -1.351087 13.506572
## Si 16.524970 10.692628  3.563037  4.9856918  5.031331  3.469925
## K  15.395562 12.608011  5.535336  7.8704352 17.688354  8.120308
## Ca 15.483430 23.389299  6.115048 14.8594678  2.444065  6.772105
## Ba  7.952370 11.350104  4.455369  4.2110074  7.456697 24.703466
## Fe  5.658116  4.082619  1.447552  0.7477509  3.986513  4.165731
##    MeanDecreaseAccuracy MeanDecreaseGini
## RI             34.536624        23.467070
## Na             20.835636        16.326536
## Mg             34.358902        24.270211
## Al             31.472662        24.257694
## Si             21.096830        14.088355
## K              23.785589        15.040157
## Ca             30.177209        20.254886
## Ba             23.336005        12.067203
## Fe              8.494595         6.983596
```

```
varImpPlot(arbol_randomforest)
```



El modelo obtenido tiene un *OOB estimate of error rate* del 19.16%. De entre todo los predictores empleados, RI, Mg, Al y Ca son los más influyentes. De hecho, si se genera un modelo más simple que solo incluya estos predictores, la estimación del error aumenta poco.

```
set.seed(17)
arbol_randomforest <- randomForest(type ~ RI + Mg + Al + Ca, data = fgl, mtry = 2,
    ntree = 500, importance = TRUE)
arbol_randomforest
```

```
##
## Call:
##  randomForest(formula = type ~ RI + Mg + Al + Ca, data = fgl,      mtry = 2,
ntree = 500, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 22.9%
## Confusion matrix:
##       WinF WinNF Veh Con Tabl Head class.error
## WinF    63     5   2   0    0    0   0.1000000
## WinNF   10    58   3   2    2    1   0.2368421
## Veh      7     1   9   0    0    0   0.4705882
## Con      0     1   0   8    1    3   0.3846154
## Tabl     0     2   0   3    3    1   0.6666667
## Head     1     3   0   1    0   24   0.1724138
```

Boosting

Boosting es otro procedimiento general que se puede aplicar a un amplio grupo de métodos de *statistical learning*, entre ellos los árboles de predicción, con la finalidad de reducir su varianza. Es una alternativa al método de *bagging* visto anteriormente.

La idea detrás de *boosting* es la siguiente. En lugar de crear un único árbol grande que se ajuste bien a todos los datos (potencial *overfiting*), la aproximación por *boosting* intenta aprender poco a poco. El proceso consiste en crear árboles de forma secuencial; cada nuevo árbol emplea información de los árboles anteriores (sus residuos), mejorando así el modelo lentamente. Cada uno de los árboles ajustados suele ser muy pequeño, con solo d nodos. Además, el parámetro λ determina cuanto influye cada nuevo árbol al modelo "acumulado".

Los tres parámetros del método de *boosting* son:

- El número de árboles B: A diferencia del *bagging* y *random forest*, el *boosting* puede sufrir *overfitting* si B es excesivamente alto. Se emplea *cross validation* para identificar el número de árboles óptimo.
- λ: Controla el ritmo al que el proceso de *boosting* aprende. Normalmente se emplean los valores 0.01 o 0.001, aunque la elección correcta puede variar dependiendo del problema. Cuanto menor sea λ, más árboles se necesitan para alcanzar buenos resultados.
- El número de divisiones d de cada árbol: Con frecuencia $d = 1$ genera resultados óptimos. En este caso cada árbol contiene un único nodo, es decir, un único predictor.

A diferencia del método de *bagging* y *random forest*, el *boosting* no hace uso de muestreo repetido (*bootstraping*), por lo que cada árbol construido depende en gran medida de los árboles previos. Acorde al libro de *Introduction to Statistical Learning*, el método de *boosting* puede superar ligeramente a *random forest*.

Ejemplo regresión

De nuevo, se pretende generar un árbol de regresión que permita predecir el precio medio de una vivienda (medv) en función de las variables disponibles, esta vez empleando el proceso de Boosting.

La función `gbm()` del paquete `gbm` ajusta árboles de decisión empleando el método de *boosting*. Entre sus múltiples argumentos cabe destacar: el tipo de distribución, `distribution="gaussian"` para árboles de regresión y `distribution="bernulli"` para árboles

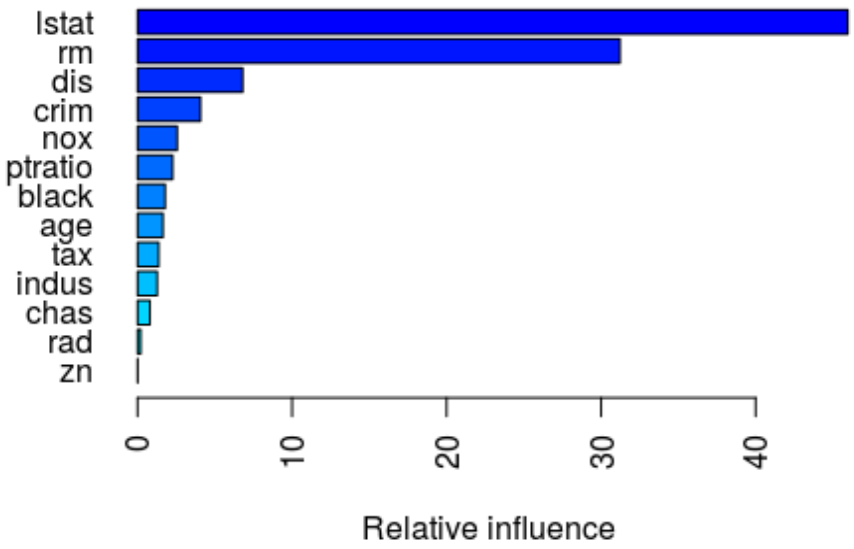
de clasificación. El número de árboles, `n.trees`, que por defecto es 5000. La profundidad máxima de cada árbol, `interaction.depth`, y el parámetro de *shrinkage* α, `shrinkage`, que por defecto es de 0.001.

```
library(MASS)
library(gbm)
data("Boston")
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)
arbol_boosting <- gbm(medv ~ ., data = Boston[train, ], distribution = "gaussian",
                      n.trees = 5000, interaction.depth = 4)
arbol_boosting
```

```
## gbm(formula = medv ~ ., distribution = "gaussian", data = Boston[train,
##     ], n.trees = 5000, interaction.depth = 4)
## A gradient boosted model with gaussian loss function.
## 5000 iterations were performed.
## There were 13 predictors of which 13 had non-zero influence.
```

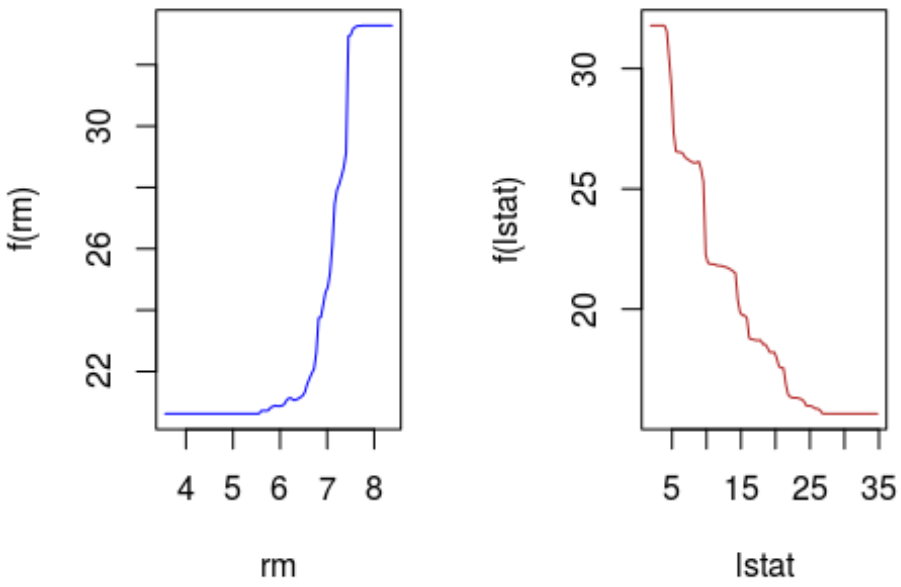
El `summary()` de un objeto `gbm` muestra la influencia relativa de cada predictor incluido en el modelo. De nuevo, las variables *rm* y *lstat* son las más influyentes. Además, la función `plot()` aplicada a un árbol `gbm` permite generar lo que se conoce como *partial dependence plots*, que muestran la influencia de un predictor sobre la variable respuesta, manteniendo constantes el resto de predictores. En este ejemplo, con forme aumenta *rm* aumenta el precio medio de la vivienda y lo opuesto ocurre con *lstat*.

```
summary(arbol_boosting, las = 2)
```



```
##             var     rel.inf
## lstat     lstat 45.96792013
## rm           rm 31.22018272
## dis         dis  6.80567724
## crim       crim  4.07534048
## nox         nox  2.56586166
## ptratio ptratio  2.26983216
## black     black  1.78740116
## age         age  1.64495723
## tax         tax  1.36917603
## indus     indus  1.27052715
## chas       chas  0.80066528
## rad         rad  0.20727091
## zn           zn  0.01518785
```

```
par(mfrow = c(1, 2))
plot(arbol_boosting, i.var = "rm", col = "blue")
plot(arbol_boosting, i.var = "lstat", col = "firebrick")
```



Una vez generado el modelo, se predice el precio medio empleando el *test set* y se evalúa su precisión.

```
Predicciones <- predict(object=arbol_boosting,newdata=Boston[-train,],n.trees=5000)
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
test_mse
```

```
## [1] 11.84694
```

El *test-error* es muy inferior al obtenido mediante *bagging* y ligeramente mejor que el obtenido por *random forest*.

Bibliografía

Introduction to Statistical Learning

Apuntes del curso Statistics 36-350: Data Mining (Fall 2009) - CMU Statistics https://www.stat.cmu.edu/~cshalizi/350/lectures/22/lecture-22.pdf

Classification and Regression by randomForest, Andy Liaw and Mathew Wiener

Classification and regression trees. Wei-Yin Loh