Contenido

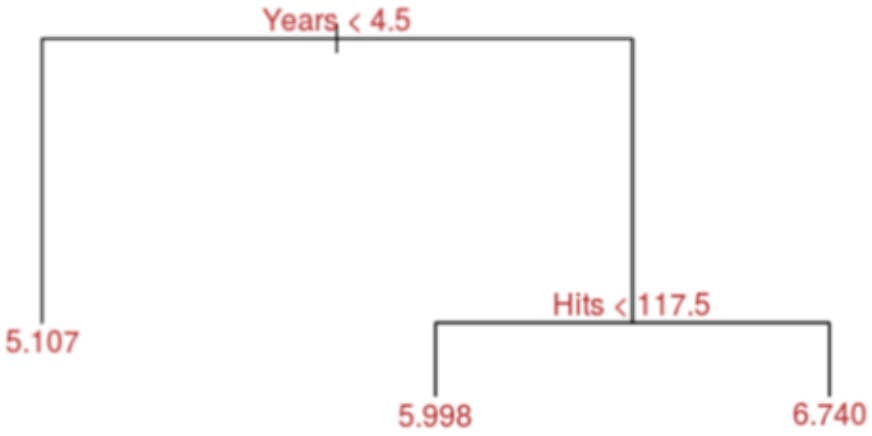
Introducción.

1. 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 variables. Existen métodos de ajuste no lineal (step functions, splines,...) 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 supervisadas no paramétricas que consiguen segmentar el espacio de los predictores en regiones simples, dentro de las cuales es más sencillo manejar las interacciones. Los métodos basados en árboles se han convertido en uno de los referentes dentro del ámbito predictivo debido a los buenos resultados que generan en ámbitos muy diversos.

2. Árboles de regresión

Los árboles de regresión son el subtipo de árboles de predicción que se aplica cuando la variable respuesta es continua. En términos generales, en el entrenamiento de un árbol de regresión, las observaciones se van distribuyendo por bifurcaciones (nodos) generando la estructura del árbol hasta alcanzar un nodo terminal. Cuando se quiere predecir una nueva observación, se recorre el árbol acorde al valor de sus predictores hasta alcanzar uno de los nodos terminales. La predicción del árbol es la media de la variable respuesta de las observaciones de entrenamiento que están en ese mismo nodo terminal.

La forma más sencilla de entender la idea detrás de los árboles de regresión es mediante 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, la primera división es la que separa a los jugadores en función de si superan o no los 4.5 años de experiencia, la segunda división está en función de si superan o no los 117.5 bateos.

- $ 5.107. Es el salario promedio de todos los jugadores que no superan los 4.5 años de experiencia.
- $ 5.998. Es el salario promedio de todos los jugadores que tienen o superan los 4.5 años de experiencia y han conseguido menos de 117.5 bateos.
- $ 6.740. Es el salario promedio de todos los jugadores que tienen o superan los 4.5 años de experiencia y han conseguido igual o más de 117.5 bateos.

Los resultados de las estratificaciones han generado 3 regiones que pueden identificarse con la siguiente nomenclatura:

- $R1=\{X|Year<4.5\}$: jugadores que han jugado menos de 4.5 años.
- $R2=\{X|Year\geq 4.5,\ Hits<117.5\}$: jugadores que han jugado 4.5 años o más y que han conseguido menos de 117.5 bateos.
- $R3=\{X|Year\geq 4.5,\ Hits\geq 117.5\}$: jugadores que han jugado 4.5 años o más y que han conseguido 117.5 o más bateos.

A las regiones $R1$, $R2$ y $R3$ se les conoce como nodos terminales o leaves del árbol, a los puntos en los que el espacio de los predictores sufre una división como internal nodes o splits y a los segmentos que conectan dos nodos como branches o ramas.

Descripción de los resultados: Viendo el árbol anterior, la interpretación del modelo es, la variable más importante a la hora de determinar el salario de un jugador es el número de años de experiencia, los jugadores con más experiencia ganan más. Entre los jugadores con menos 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. Con este ejemplo, 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.

3. Entrenamiento del árbol

El proceso de construcción de un árbol de predicción o clasificación se divide en dos etapas:

- División sucesiva del espacio de los predictores generando regiones no solapantes (nodos terminales) $R1$, $R2$, $R3$, …, Rj. 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.
- Predicción de la variable respuesta en cada 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 $R1$, $R2$, $R3$, …, Rj, o lo que es equivalente, decidir donde se introducen las divisiones.

En el caso de los árboles de regresión, el criterio más frecuentemente empleado para **identificar las divisiones es el Residual Sum of Squares (RSS).** El objetivo es encontrar las J regiones ($R1$,…, Rj) que minimizan el Residual Sum of Squares (RSS) total:

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

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.

4. Recursive binary splitting

El objetivo del método recursive binary splitting es encontrar en cada iteración el predictor X_j y el punto de corte (umbral) s tal que, si se distribuyen las observaciones en las regiones $\{X|Xj<s\}$ y $\{X|Xj\geq s\}$, se consigue la mayor reducción posible en el RSS. El algoritmo seguido es:

1. El proceso se inicia en lo más alto del árbol, donde todas las observaciones pertenecen a la misma región.
2. Se identifican todos los posibles puntos de corte (umbrales) s para cada uno de los predictores ($X1$, $X2$,..., Xp). En el caso de predictores cualitativos, los posibles puntos de corte son cada uno de sus niveles. Para predictores continuos, se ordenan de menor a mayor sus valores, el punto intermedio entre cada par de valores se emplea como punto de corte.
3. Se calcula el RSS total que se consigue con cada posible división identificada en el paso 2.

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

   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 acordes al predictor j y valor s.

4. Se selecciona el predictor X_j y el punto de corte S que resulta en el menor RSS total, es decir, que da lugar a las divisiones más homogéneas posibles. Si existen dos o más divisiones que consiguen la misma mejora, la elección entre ellas es aleatoria.
5. Se repiten de forma iterativa los pasos 1 a 4 para cada una de las regiones que se han creado en la iteración anterior hasta que se alcanza alguna norma de stop. Algunas de las más empleadas son: que ninguna región contenga un mínimo de n observaciones, que el árbol tenga un máximo de nodos terminales o que la incorporación del nodo reduzca el error en al menos un % mínimo.

Está metodología conlleva dos hechos:

- Que cada división óptima se identifica acorde al impacto que tiene en ese momento. No se tiene en cuenta si es la división que dará lugar a mejores árboles en futuras divisiones.
- En cada división se evalúa un único predictor haciendo preguntas binarias (si, no), lo que genera dos nuevas ramas del árbol por división. 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.

Nota: Los algoritmos que implementan recursive binary splitting suelen incorporar estrategias para evitar evaluar todos los posibles puntos de corte. Por ejemplo, para predictores continuos, primero se crea un histograma que agrupa los valores y luego se evalúan los puntos de corte de cada región del histograma (bin).

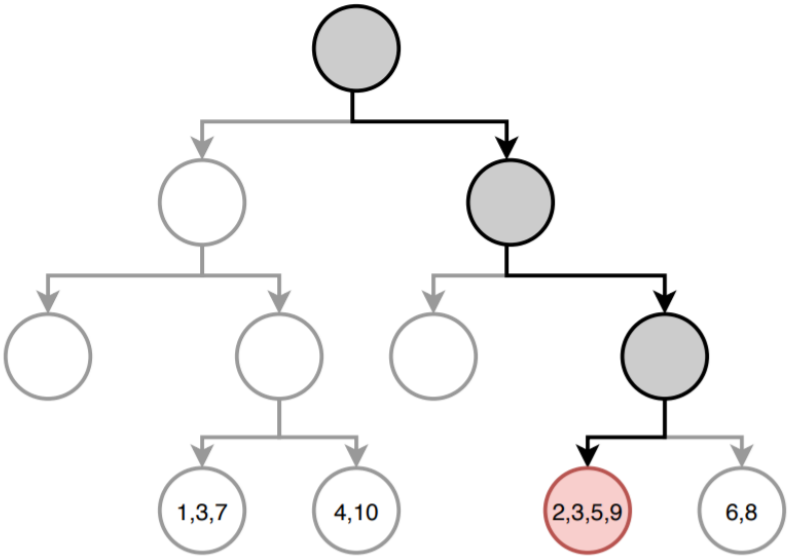
5. Predicción del árbol

Tras la creación de un árbol, las observaciones de entrenamiento quedan agrupadas en los nodos terminales. Para predecir una nueva observación, se recorre el árbol en función de los valores que tienen sus predictores hasta llegar a uno de los nodos terminales. En el caso de regresión, el valor predicho suele ser la media de la variable respuesta de las observaciones de entrenamiento que están en ese mismo nodo. Si bien la media es valor más empleado, se puede utilizar cualquier otro (mediana, cuantil...).

Supóngase que se dispone de 9 observaciones, cada una con un valor de variable respuesta Y y unos predictores X.

<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	
id	1	2	3	4	5	6	7	8	9
Y	10	18	24	8	2	9	16	10	20
X	...	...	...	...	...	...	...	...	...

La siguiente imagen muestra cómo sería la predicción del árbol para una nueva observación. El camino hasta llegar al nodo final está resaltado. En cada nodo terminal se detalla el índice de las observaciones de entrenamiento que forman parte.



El valor predicho por el árbol es la media de la variable respuesta Y de las observaciones con id: 2, 3, 5, 9.

$$\hat{u} = \frac{18 + 24 + 2 + 20}{4} = 16$$

Aunque la anterior es la forma más común de obtener las predicciones de un árbol de regresión, existe otra aproximación. La predicción de un árbol de regresión puede verse como una variante de vecinos cercanos en la que, solo las observaciones que forman parte del mismo nodo terminal que la observación predicha, tienen influencia. Siguiendo esta aproximación, la predicción del árbol se define como la media ponderada de todas las observaciones de entrenamiento, donde el peso de cada observación depende únicamente de si forma parte o no del mismo nodo terminal.

$$\hat{u} = \sum_{i=1}^{n} w_i Y_i$$

El valor de las posiciones del vector de pesos w es 1 para las observaciones que están en el mismo nodo y 0 para el resto. En este ejemplo sería:

$$w = (0,1,1,0,1,0,0,0,1,0)$$

Para que la suma de todos los pesos sea 1, se dividen por el número total de observaciones en el nodo terminal seleccionado, en este caso 4.

$$w = (0,\frac{1}{4},\frac{1}{4},0,\frac{1}{4},0,0,0,\frac{1}{4},0)$$

Así pues, el valor predicho es:

$$w = (0x10) + \left(\frac{1}{4}x18\right) + \left(\frac{1}{4}x24\right) + (0x8) + \left(\frac{1}{4}x2\right) + (0x9) + (0x16) + (0x19) + \left(\frac{1}{4}x20\right) + (0x14) = 16$$

Evitar el overfitting

El proceso de construcción de árboles descrito en las secciones anteriores tiende a reducir rápidamente el error de entrenamiento, es decir, el modelo se ajusta muy bien a las observaciones empleadas como entrenamiento. Como consecuencia, se genera un overfitting 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. De hecho, si no se limitan las divisiones, todo árbol termina ajustándose perfectamente a las observaciones de entrenamiento creando un nodo terminal por observación. Existen dos estrategias para prevenir el problema de overfitting de los árboles: limitar el tamaño del árbol (parada temprana o early stopping) y el proceso de podado (pruning).

Controlar el tamaño del árbol (parada temprana)

El tamaño final que adquiere un árbol puede controlarse mediante reglas de parada que detengan la división de los nodos dependiendo de si se cumplen o no determinadas condiciones. El nombre de estas condiciones puede variar dependiendo del software o librería empleada, pero suelen estar presentes en todos ellos.

- **Observaciones mínimas para división**: define el número mínimo de observaciones que debe tener un nodo para poder ser dividido. Cuanto mayor el valor, menos flexible es el modelo.
- **Observaciones mínimas de nodo terminal:** define el número mínimo de observaciones que deben tener los nodos terminales. Su efecto es muy similar al de observaciones mínimas para división.
- **Profundidad máxima del árbol**: define la profundidad máxima del árbol, entendiendo por profundidad máxima el número de divisiones de la rama más larga (en sentido descendente) del árbol.
- **Número máximo de nodos terminales:** define el número máximo de nodos terminales que puede tener el árbol. Una vez alcanzado el límite, se detienen las divisiones. Su efecto es similar al de controlar la profundidad máxima del árbol.
- **Reducción mínima de error**: define la reducción mínima de error que tiene que conseguir una división para que se lleve a cabo.

A todos estos parámetros se les conoce como hiperparámetros porque no se aprenden durante el entrenamiento del modelo. Su valor tiene que ser especificado por el usuario en base a su conocimiento del problema y mediante el uso de validación cruzada.

Tree pruning

La estrategia de controlar el tamaño del árbol mediante reglas de parada tiene un inconveniente, el árbol se crece seleccionando la mejor división en cada momento.

Al evaluar las divisiones sin tener en cuenta las que vendrán después, nunca se elige la opción que resulta en el mejor árbol final, a no ser que también sea la que genera en ese momento la mejor división. A este tipo de estrategias se les conoce como greedy. Un ejemplo que ilustra el problema de este tipo de estrategia es el siguiente: supóngase que un coche circula por el carril izquierdo de una carretera de dos carriles en la misma dirección. En el carril que se encuentra hay muchos coches circulando a 100 km/h, mientras que el otro carril se encuentra vacío. A cierta distancia se observa que hay un vehículo circulando por el carril derecho a 20 km/h. Si el objetivo del conductor es llegar a su destino lo antes posible tiene dos opciones: cambiarse de carril o mantenerse en el que está. Una aproximación de tipo greedy evaluaría la situación en ese instante y determinaría que la mejor opción es cambiarse de carril y acelerar a más de 100 km/h, sin embargo, a largo plazo, esta no es la mejor solución, ya que una vez alcance al vehículo lento, tendrá que reducir mucho su velocidad.

Una alternativa no greedy que consigue evitar el overfitting consiste en generar árboles grandes, sin condiciones de parada más allá de las necesarias por las limitaciones computacionales, para después podarlos (pruning), manteniendo únicamente la estructura robusta que consigue un test error bajo. La selección del sub-árbol óptimo puede hacerse mediante cross-validation, sin embargo, dado que los árboles se crecen lo máximo posible (tienen muchos nodos terminales) no suele ser viable estimar el test error de todas las posibles sub-estructuras que se pueden generar. En su lugar, 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

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

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, mayor penalización). El grado de penalización se determina mediante el tuning parameter α. Cuando α=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**.**

6. Algoritmo para crear un árbol de regresión con pruning

1. Se emplea recursive binary splitting para crear un árbol grande y complejo (To) empleando los datos de training y reduciendo al máximo posible las

condiciones de parada. Normalmente se emplea como única condición de parada el número mínimo de observaciones por nodo terminal.

2. Se aplica el cost complexity pruning al árbol To 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:
   – Repetir pasos 1 y 2 empleando todas las observaciones excepto las del grupo ki.
   – Evaluar el mean squared error para el rango de valores de α empleando el grupo ki.
   – 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 en el paso 3.

En el caso de los árboles de clasificación, en lugar de emplear la suma de residuos cuadrados como criterio de selección, se emplea alguna de las medidas de homogeneidad vistas en apartados anteriores.

Ventajas y desventajas

Ventajas

- Los árboles son fáciles de interpretar aun cuando las relaciones entre predictores son complejas. Su estructura se asemeja a la forma intuitiva en que clasificamos y predecimos las personas, además, no se requieren conocimientos estadísticos para comprenderlos.
- Los modelos basados en un solo árbol (no es el caso de random forest, boosting…) se pueden representar gráficamente aun cuando el número de predictores es mayor de 3.
- Los árboles pueden manejar tanto predictores cuantitativos como cualitativos sin tener que crear variables dummy.
- Al tratarse de métodos no paramétricos, no es necesario que se cumpla ningún tipo de distribución específica.
- Por lo general, requieren mucha menos limpieza y pre procesado de los datos en comparación a otros métodos de aprendizaje estadístico.
- No se ven muy influenciados por outliers.
- 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 último nodo alcanzado. La precisión de la predicción se verá reducida pero al menos podrá obtenerse.

- Son muy útiles en la exploración de datos, permiten identificar de forma rápida y eficiente las variables más importantes.
- Son capaces de seleccionar predictores de forma automática.

Desventajas

- La capacidad predictiva de los modelos de regresión y clasificación basados en un único árbol es bastante inferior a la conseguida con otros modelos debido a su tendencia al overfitting. 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.
- Cuando tratan con variables continuas, pierden parte de su información al categorizarlas en el momento de la división de los nodos. Por esta razón, suelen ser modelos que consiguen mejores resultados en clasificación que en regresión.
- Tal y como se describe más adelante, la creación de las ramificaciones de los árboles se consigue mediante el algoritmo de recursive binary splitting. Este algoritmo identifica y evalúa las posibles divisiones de cada predictor acorde a una determinada medida (RSS, Gini, entropía…). Los predictores continuos o predictores cualitativos con muchos niveles tienen mayor probabilidad de contener, solo por azar, algún punto de corte óptimo, por lo que suelen verse favorecidos en la creación de los árboles.

7. Paquetes R

Debido a sus buenos resultados, Random Forest y Gradient Boosting, se han convertido en los algoritmos de referencia cuando se trata con datos tabulares, de ahí que se hayan desarrollado múltiples implementaciones. Cada una tiene unas características que las hacen más adecuadas dependiendo del caso de uso. Algunos de los paquetes más empleados en R son:

- tree y rpart: permiten crear y representar árboles de regresión y clasificación.
- rpart.plot: permite crear representaciones detalladas de modelos creados con rpart.
- randomForest: dispone de los principales algoritmos para crear modelos random forest. Destaca por su fácil uso, pero no por su rapidez.

- ranger: algoritmos para crear modelos random forest. Es similar a randomForest pero mucho más rápido. Además, incorpora extremely randomized trees y quantile regression forests.
- gbm: dispone de los principales algoritmos de boosting. Este paquete ya no está mantenido, aunque es útil para explicar los conceptos, no se recomienda su uso en producción.
- XGBoost: esta librería permite acceder al algoritmo XGboost (Extra Gradient boosting). Una adaptación de gradient boosting que destaca por su eficiencia y rapidez.
- H2O: implementaciones muy optimizadas de los principales algoritmos de machine learning, entre ellos random forest, gradient boosting y XGBoost. Para conocer más detalles sobre cómo utilizar esta librería consultar Machine Learning con H2O y R.
- C50: implementación de los algoritmos C5.0 para árboles de clasificación.

8. Ejemplo regresión:

El data set 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.

Descripción de variables

El marco de datos de Boston tiene 506 filas y 14 columnas.

crim: Tasa de criminalidad per cápita por poblado.

zn: proporción de tierra residencial zonificada para lotes de más de 25,000 pies cuadrados.

indus: Proporción de acres de negocios no minoristas por ciudad.

chas: Variable ficticia del río Charles (= 1 si el trecho limita al río; 0 en caso contrario).

nox: Concentración de óxidos de nitrógeno (partes por 10 millones).

rm: promedio de habitaciones por vivienda.

age: Proporción de unidades ocupadas por el propietario construidas antes de 1940.

dis: media ponderada de distancias a cinco centros de empleo de Boston.

rad: Indice de accesibilidad a autopistas.

tax: tasa de impuesto a la propiedad de valor total por \ $ 10,000.

ptratio: Proporción alumno-profesor por ciudad.

black: 1000 (Bk - 0,63) ^ 2 donde Bk es la proporción de negros por ciudad.

lstat: porcentaje de personas en estado de pobreza (porcentaje).

medv: precio medio de las viviendas ocupadas en \ $ 1000s. (variable de respuesta)

```
# arboles de regresion
# Tratamiento de datos
library(MASS) # data  Boston
library("easypackages")
paq <- c("dplyr", "tidyr", "ggplot2", "ggpubr")
libraries(paq)
```

```
library(tree)
# importando y conociendo datos
Boston <- read.csv("Boston.csv",head=T, sep=";")
head(Boston)
```

```
##      crim    zn indus chas   nox    rm   age    dis  rad  tax ptratio  black lstat  medv
## 1 0.00632 18  2.31    0     0.538  6.575  65.2  4.0900    1  296    15.3  396.90  4.98  24.0
## 2 0.02731  0  7.07    0     0.469  6.421  78.9  4.9671    2  242    17.8  396.90  9.14  21.6
## 3 0.02729  0  7.07    0     0.469  7.185  61.1  4.9671    2  242    17.8  392.83  4.03  34.7
## 4 0.03237  0  2.18    0     0.458  6.998  45.8  6.0622    3  222    18.7  394.63  2.94  33.4
## 5 0.06905  0  2.18    0     0.458  7.147  54.2  6.0622    3  222    18.7  396.90  5.33  36.2
## 6 0.02985  0  2.18    0     0.458  6.430  58.7  6.0622    3  222    18.7  394.12  5.21  28.7
```

```
str (Boston)
```

```
## 'data.frame':    506 obs. of  14 variables:
##  $ crim    : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ zn      : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
##  $ indus   : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
##  $ chas    : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ nox     : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524 0.524 ...
##  $ rm      : num  6.58 6.42 7.18 7 7.15 ...
##  $ age     : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
##  $ dis     : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ rad     : int  1 2 2 3 3 3 5 5 5 5 ...
##  $ tax     : int  296 242 242 222 222 222 311 311 311 311 ...
##  $ ptratio : num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
```

```
##  $ black   : num  397 397 393 395 397 ...
##  $ lstat   : num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv    : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

La data está compuesta por 506 observaciones y 14 variables, todas reconocidas numéricas (int y num). La variable Chas es cualitativa, podríamos excluirla o tomarla como variable dummy. Tomemos la decisión de tomarla como dummy.

```
# Boston$chas <- as.factor(Boston$chas)
# str(Boston)
# la variable Chas se ubica en la posicion 4. eliminado
Boston <-Boston [,-4]
str(Boston)
```

```
## 'data.frame':    506 obs. of  13 variables:
##  $ crim    : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ zn      : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
##  $ indus   : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
##  $ nox     : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524
0.524 ...
##  $ rm      : num  6.58 6.42 7.18 7 7.15 ...
##  $ age     : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
##  $ dis     : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ rad     : int  1 2 2 3 3 3 5 5 5 5 ...
##  $ tax     : int  296 242 242 222 222 222 311 311 311 311 ...
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
##  $ black   : num  397 397 393 395 397 ...
##  $ lstat   : num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv    : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

```
# resumen de NAs y descriptiva
library(skimr)
skim(Boston)
```

Data summary

Name	Boston
Number of rows	506
Number of columns	13
_____	
Column type frequency:	
factor	0
numeric	13
_____	
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
chas	0	1	FALSE	2	0: 471, 1: 35

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
crim	0	1	3.61	8.60	0.01	0.08	0.26	3.68	88.98	▇▁▁▁
zn	0	1	11.36	23.32	0.00	0.00	0.00	12.50	100.00	▇▁▁▁
indus	0	1	11.14	6.86	0.46	5.19	9.69	18.10	27.74	▇▆▁▇▁
nox	0	1	0.55	0.12	0.38	0.45	0.54	0.62	0.87	▇▇▆▅▁
rm	0	1	6.28	0.70	3.56	5.89	6.21	6.62	8.78	▁▂▅▂▁
age	0	1	68.57	28.15	2.90	45.02	77.50	94.07	100.00	▂▂▂▃▇
dis	0	1	3.80	2.11	1.13	2.10	3.21	5.19	12.13	▇▅▂▁▁
rad	0	1	9.55	8.71	1.00	4.00	5.00	24.00	24.00	▇▂▁▁▃
tax	0	1	408.24	168.54	187.00	279.00	330.00	666.00	711.00	▇▆▂▁▇
ptratio	0	1	18.46	2.16	12.60	17.40	19.05	20.20	22.00	▁▃▅▅▇
black	0	1	356.67	91.29	0.32	375.38	391.44	396.22	396.90	▁▁▁▁▇
lstat	0	1	12.65	7.14	1.73	6.95	11.36	16.96	37.97	▇▇▅▂▁
medv	0	1	22.53	9.20	5.00	17.02	21.20	25.00	50.00	▂▅▇▂▁

Ajuste del modelo

La función tree() del paquete tree permite ajustar árboles. La elección entre árbol de regresión o árbol de clasificación se hace automáticamente dependiendo de si la variable respuesta es de tipo numérico 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.

A continuación, se ajusta un árbol de regresión empleando como variable respuesta medv y como predictores todas las variables disponibles. Como en todo 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 de entrenamiento y otro de test.

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.

```
# Ajuste del modelo
library(tree)
set.seed(123) # para reproducir los mismos resultados
# creando data training (80%) y data test 20%
train <- sample(1:nrow(Boston), size = nrow(Boston)*0.8)
data_train <- Boston[train,]
data_test <- Boston[-train,]
```

```
dim(data_train)
```

```
## [1] 404  13
```

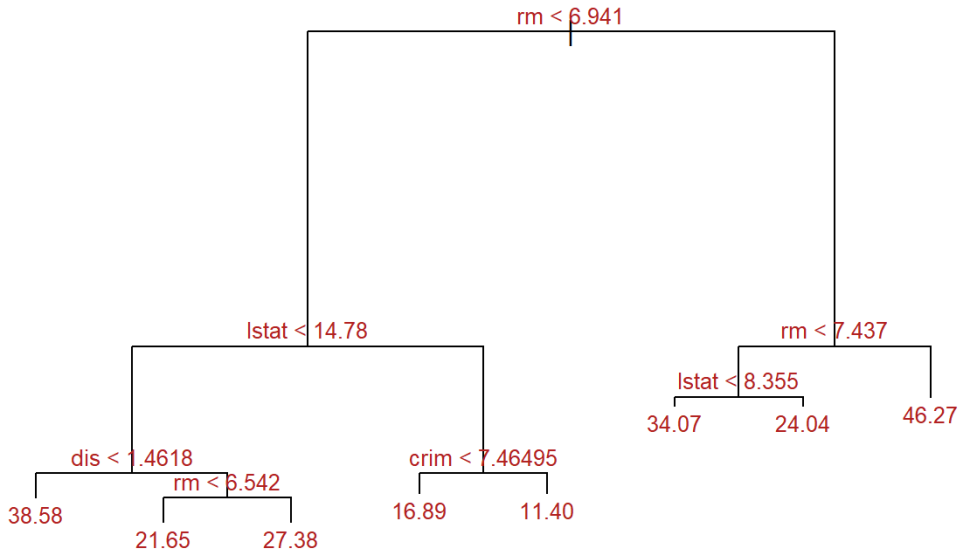
```
dim(data_test)
```

```
## [1] 102  13
```

```r
# entrenamiento de un arbol con valores por defecto
arbol_regresion1 <- tree(formula = medv ~ .,
                         data = data_train, split = "deviance")
arbol_regresion1
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 34110.0 22.51
##    2) rm < 6.941 348 14110.0 20.02
##      4) lstat < 14.78 218  5458.0 23.30
##        8) dis < 1.4618 5  1061.0 38.58 *
##        9) dis > 1.4618 213  3203.0 22.94
##         18) rm < 6.542 165  1445.0 21.65 *
##         19) rm > 6.542 48   537.2 27.38 *
##      5) lstat > 14.78 130  2384.0 14.53
##       10) crim < 7.46495 74   830.7 16.89 *
##       11) crim > 7.46495 56   589.9 11.40 *
##    3) rm > 6.941 56  4462.0 37.97
##      6) rm < 7.437 34  1529.0 32.59
##       12) lstat < 8.355 29   653.8 34.07 *
##       13) lstat > 8.355 5   446.6 24.04 *
##      7) rm > 7.437 22   433.1 46.27 *
```

```r
plot(x =arbol_regresion1, type = "proportional")
text(x = arbol_regresion1, splits = TRUE, pretty = 0, cex = 0.80, col = "firebrick")
```




```
# A continuación la correlación entre Y e y ajustada
y_hat <- predict(object=arbol_regresion1, newdata=data_train)
cor(y_hat, data_train$medv)
```

```
## [1] 0.9078344
```

```
# prediciendo con el modelo
predict(object=arbol_regresion1,
        newdata=data.frame(crim=2,zn=0.025,indus=20,nox=0,rm=0.25,
                           age=7,dis=62,rad=3,tax=265,ptratio=20,black=400,
                           lstat=4.98))
```

```
##        1
## 21.65212
```

```
# error
predicciones <- predict(arbol_regresion1, newdata = data_test)
test_rmse    <- sqrt(mean((predicciones - data_test$medv)^2))
paste("Error de test del árbol inicial:", round(test_rmse,2))
```

```
"Error de test del árbol inicial: 4.72"
```

Controlando algunos criterios

```
# controlando algunos criterios
# variable respuesta medv
# method = "recursive.partition"
set.seed(123)
arbol_regresion <- tree::tree(
  formula = medv ~ .,  # Variables de respuesta y predictoras
  data    = data_train, # datos de entrenamiento
  split   = "deviance",# criterio de division, puede usar tambien "gini"
  mincut  = 20, # Número mínimo de observaciones para que se produzca la división
  minsize = 50  # Número mínimo de observaciones para que un nodo pueda ramificarse
)
summary(arbol_regresion)
```

```
## Regression tree:
## tree::tree(formula = medv ~ ., data = data_train, split = "deviance",
##     mincut = 20, minsize = 50)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "dis"   "crim"
## Number of terminal nodes:  8
## Residual mean deviance:  17.16 = 6794 / 396
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -22.1900  -2.1930   0.1406   0.0000   2.1240  23.2300
```

La función summary() muestra que el árbol ajustado tiene un total de 8 nodos terminales y que se han empleado como predictores las variables rm, lstat. dis y crim. En el contexto de árboles de regresión, el término Residual mean deviance (17.16) es la suma de cuadrados residuales 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

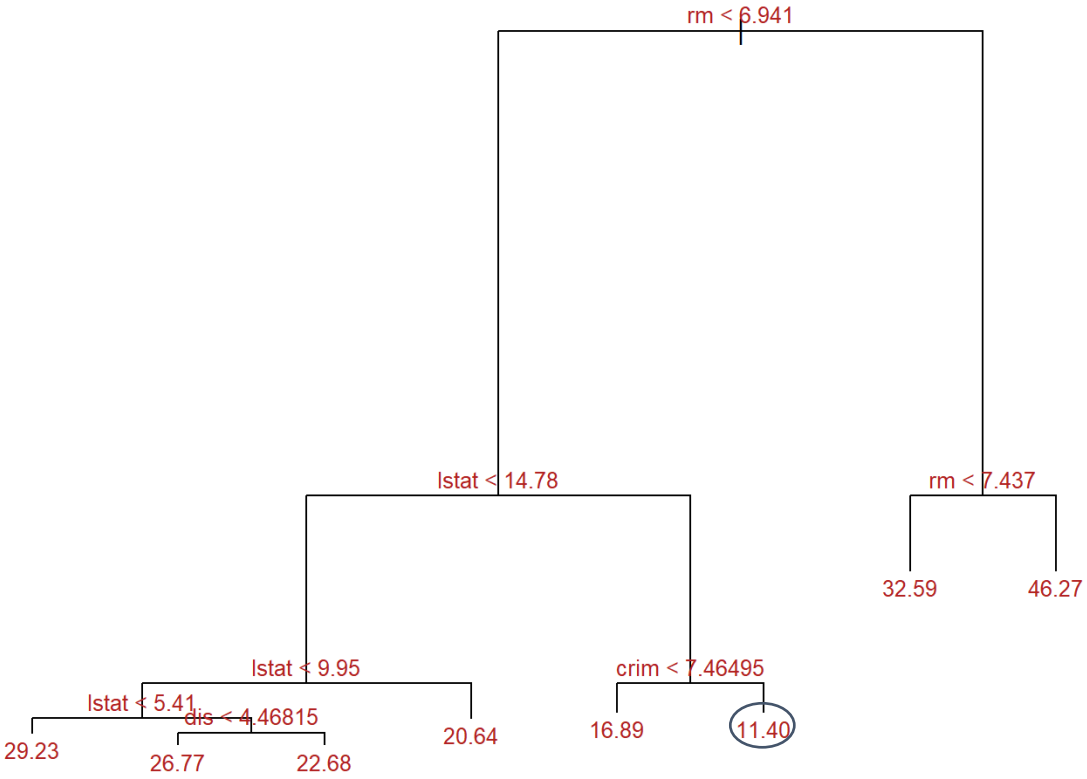
impacto de cada división en el modelo. La función text() añade la descripción de cada nodo interno y el valor de cada nodo terminal.

```
arbol_regresion
```

```
# estructura del arbol
plot(x =arbol_regresion, type = "proportional")
text(x = arbol_regresion, splits = TRUE, pretty = 0, cex = 0.80, col = "f
irebrick")
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 34110.0 22.51
##    2) rm < 6.941 348 14110.0 20.02
##      4) lstat < 14.78 218  5458.0 23.30
##        8) lstat < 9.95 124  3678.0 25.32
##         16) lstat < 5.41 25   728.6 29.23 *
##         17) lstat > 5.41 99  2471.0 24.33
##           34) dis < 4.46815 40  1777.0 26.77 *
##           35) dis > 4.46815 59   293.9 22.68 *
##        9) lstat > 9.95 94   611.8 20.64 *
##      5) lstat > 14.78 130  2384.0 14.53
##       10) crim < 7.46495 74   830.7 16.89 *
##       11) crim > 7.46495 56   589.9 11.40 *
##    3) rm > 6.941 56  4462.0 37.97
##      6) rm < 7.437 34  1529.0 32.59 *
##      7) rm > 7.437 22   433.1 46.27 *
```

```
# estructura del arbol
par(mar = c(1,1,1,1)) # Márgenes del gráfico
plot(x =arbol_regresion, type = "proportional")
text(x = arbol_regresion, splits = TRUE, pretty = 0, cex = 0.80, col = "f
irebrick")
```



La variable *rm* (promedio de habitaciones por vivienda) ha resultado ser el predictor más importante (primer nodo). Por ejemplo, el nodo 11.40 (precio de $ 11,30) indica que valores bajos de *rm* se corresponden con valores altos de *lstat* y *crim* o dicho de otra manera: el modelo predice un precio de $ 11,40 para viviendas con menores a 7 habitaciones (rm < 6.941), mayor a 14.78 % de personas en estado de pobreza (lstat > 14.78) y tasa de criminalidad per cápita mayor a 7 (crim > 7.46495)
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.

Primer root:

rm < 6,941, lstat < 14.78 , lstat < 9.95, lstat < 5.41 es 29.23*

Podado del árbol (pruning)

Con la finalidad de reducir la varianza del modelo y así mejorar la capacidad predictiva, 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 los mejores resultados de predicción.

Para aplicar el proceso de pruning es necesario indicar el grado de penalización por complejidad α. Cuanto mayor es este valor, más agresivo es el podado y menor el

tamaño del árbol resultante. Dado que no hay forma de conocer de antemano el valor óptimo de α, se recurre a validación cruzada para identificarlo.

La función cv.tree() emplea cross validation para identificar el valor óptimo de penalización. Por defecto, esta función emplea la deviance para guiar el proceso de pruning.

```
# Pruning (podado y coste- complejidad del arbol)

# el objetivo de obtener predicciones significativamente buenas, contribu
yendo

# a su vez a la disminución de la variabilidad del modelo y evitando un s
obreajuste

# del modelo. Para esto, se aplica primeramente el proceso de validación
cruzada

# sobre el árbol de tamaño completo:


# Se hace crecer el árbol tanto como sea posible

arbol_regresion <- tree(

  formula = medv ~ .,

  data    = data_train,

  split   = "deviance",

  mincut  = 1,

  minsize = 2,

  mindev  = 0

)
```

```
# Aplicación del proceso de validación cruzada

set.seed(123)

cv_arbol <- cv.tree(arbol_regresion, K = 5)

cv_arbol
```

```
## $size

##   [1] 321 319 305 303 294 293 292 289 288 281 274 272 267 262 259 258
255 254

##  [19] 249 243 242 240 239 238 237 232 231 229 228 226 225 222 221 220
217 216

##  [37] 215 212 210 209 208 206 205 204 203 200 199 198 197 196 195 194
192 191

…….

…….
```

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.
- method: El criterio empleado para seleccionar el mejor árbol.

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.

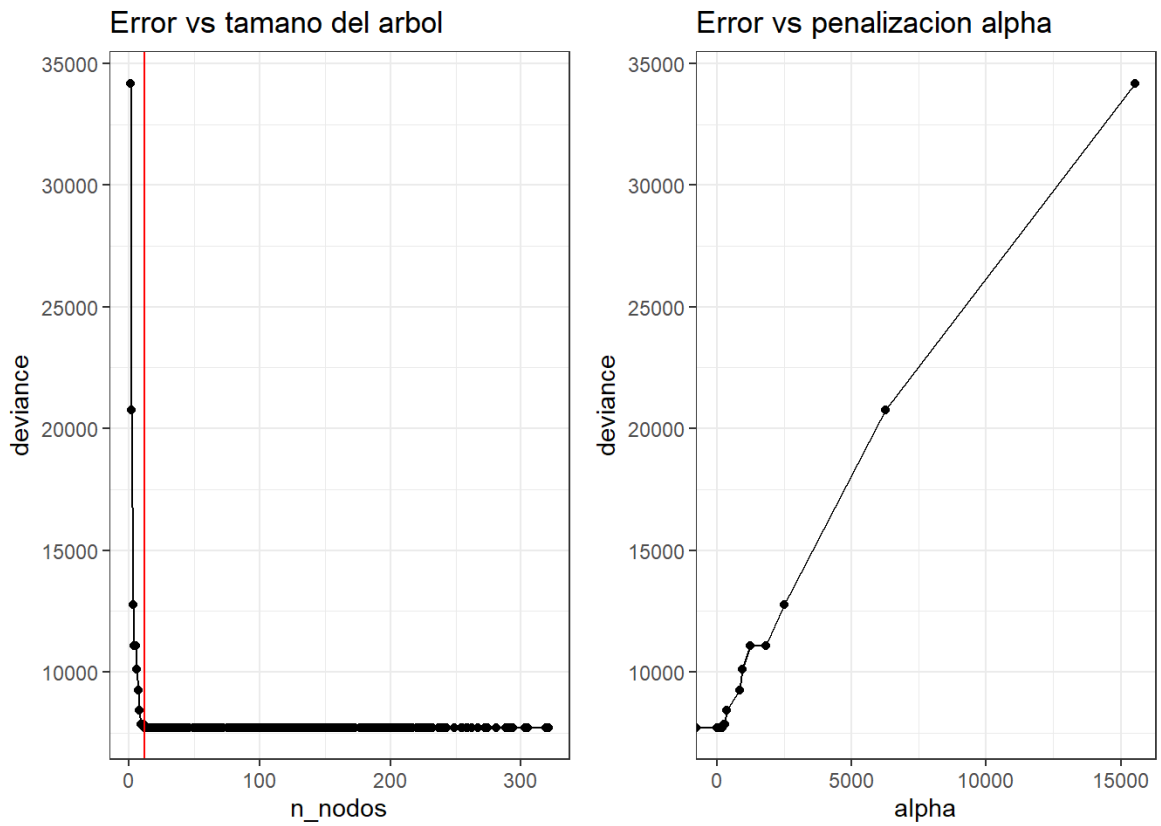
```
# Así, el número de nodos terminales por árbol (size),
# error por validación cruzada de cada árbol (dev), el rango de valores
# de penalización y el método empleado para la construcción del árbol.
# Tamano optimo encontrado
# ========================
size_optimo <- rev(cv_arbol$size)[which.min(rev(cv_arbol$dev))]
paste("Tamano optimo del numero de nodos terminales", size_optimo)
```

```
## [1] "Tamano optimo del numero de nodos terminales 12"
```

```
library(ggplot2)
library(magrittr)
library(ggpubr)
```

```
resultados_cv <- data.frame(
  n_nodos  = cv_arbol$size,
  deviance = cv_arbol$dev,
  alpha    = cv_arbol$k
)


# Gráfico del decremento del error en relación con el número de nodos ter
minales
p1 <- ggplot(data = resultados_cv, aes(x = n_nodos, y = deviance)) +
  geom_line() +
```

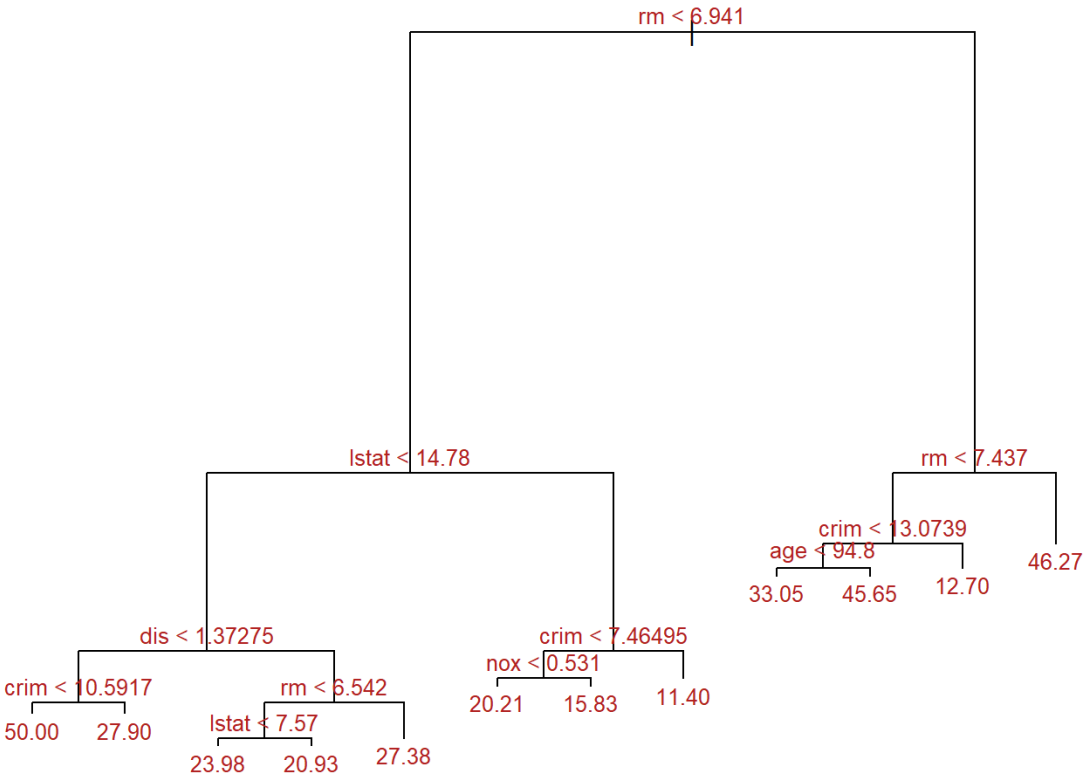
```
  geom_point() +
  geom_vline(xintercept = size_optimo, color = "red") +
  labs(title = "Error vs tamano del arbol") +
  theme_bw()

# Gráfico que contrasta los valores del parámetro de ajuste y el error ob
tenido con ellos

p2 <- ggplot(data = resultados_cv, aes(x = alpha, y = deviance)) +
  geom_line() +
  geom_point() +
  labs(title = "Error vs penalizacion alpha") +
  theme_bw()

ggarrange(p1, p2)
```



Una vez identificado el valor óptimo de, con la función prune.tree() se aplica el podado final. Esta función también acepta el valor de α óptimo en lugar del tamaño.

```
# Estructura del arbol creado final
```

```
# Poda del árbol de decisión por coste-complejidad
arbol_final <- prune.tree(
  tree = arbol_regresion,# Árbol de tamaño completo
  best = size_optimo # Número de nodos terminales óptimo
)
# Gráfico del árbol de decisión posterior al proceso de podado
# par(mar = c(1,1,1,1))
plot(x = arbol_final, type = "proportional")
text(x = arbol_final, splits = TRUE, pretty = 0, cex = 0.8, col = "firebr
ick")
```



Evaluación y Predicción del modelo

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

```
# Prediccion y evaluacion del modelo
# con data test
predicciones <- predict(arbol_regresion, newdata = data_test)
test_mse     <- mean((predicciones - data_test$medv)^2)
```

```
paste("Error de test (mse) del arbol de regresion tras podado:", round(te
st_mse,2))
```

```
## [1] "Error de test (mse) del arbol de regresion tras podado: 31.53"
```

```
predicciones <- predict(arbol_regresion, newdata = data_test)
test_rmse    <- sqrt(mean((predicciones - data_test$medv)^2))
paste("Error de test (rmse) del arbol inicial:", round(test_rmse,2))
```

```
## [1] "Error de test (rmse) del arbol inicial: 5.62"
```

El Mean Square Test Error asociado con el árbol de regresión es de 26.29 unidades. La raíz cuadrada del Mean Square Test Error es 5.62, lo que significa que las predicciones se alejan en promedio 5.62 unidades (5062 dólares) del valor real.

```
# predicciones y error de predicicion del modelo de tamaño completo
predicciones1 <- predict(arbol_final, newdata = data_test)
test_mse     <- mean((predicciones - data_test$medv)^2)
paste("Error de test (mse) del arbol de regresion tras podado:", round(te
st_mse,2))
```

```
## [1] "Error de test (mse) del arbol de regresion tras podado: 31.53"
```

```
predicciones <- predict(arbol_final, newdata = data_test)
test_rmse    <- sqrt(mean((predicciones - data_test$medv)^2))
paste("Error de test (rmse) del arbol final:", round(test_rmse,2))
```

```
## [1] "Error de test (rmse) del arbol final: 5.3"
```

El proceso de pruning no consigue reducir el error (rmse) del modelo.

Random Forest

Un modelo Random Forest está formado por un conjunto (ensemble) de árboles de decisión individuales, cada uno entrenado con una muestra aleatoria extraída de los datos de entrenamiento originales mediante bootstrapping. Esto implica que cada árbol se entrena con unos datos ligeramente distintos. En cada árbol individual, las observaciones se van distribuyendo por bifurcaciones (nodos) generando la estructura del árbol hasta alcanzar un nodo terminal. La predicción de una nueva observación se obtiene agregando las predicciones de todos los árboles individuales que forman el modelo.
Para entender cómo funcionan los modelos Random Forest es necesario conocer primero los conceptos de ensemble y bagging

Métodos de *ensemble*

Todos los modelos de aprendizaje estadístico y *machine learning* sufren el problema de equilibrio entre bias y varianza.
El término bias (sesgo) hace referencia a cuánto se alejan en promedio las predicciones de un modelo respecto a los valores reales. Refleja cómo de capaz es el modelo de aprender la relación real que existe entre los predictores y la variable respuesta. Por ejemplo, si la relación sigue un patrón no lineal, por muchos datos de los que se disponga, un modelo de regresión lineal no podrá modelar correctamente la relación, por lo que tendrá un bias alto.
El término varianza hace referencia a cuánto cambia el modelo dependiendo de los datos utilizados en su entrenamiento. Idealmente, un modelo no debería modificarse demasiado por pequeñas variaciones en los datos de entrenamiento, si esto ocurre, es porque el modelo está memorizando los datos en lugar de aprender la verdadera relación entre los predictores y la variable respuesta. Por ejemplo, un modelo de árbol con muchos nodos, suele variar su estructura con que apenas cambien unos pocos datos de entrenamiento, tiene mucha varianza.

A medida que aumenta la complejidad de un modelo, este dispone de mayor flexibilidad para adaptarse a las observaciones, reduciendo así el bias y mejorando su capacidad predictiva. Sin embargo, alcanzado un determinado grado de flexibilidad, aparece el problema de *overfitting*, el modelo se ajusta tanto a los datos de entrenamiento que es incapaz de predecir correctamente nuevas observaciones. El mejor modelo es aquel que consigue un **equilibrio óptimo entre bias y varianza**.

¿Cómo se controlan el bias y varianza en los modelos basados en árboles? Por lo general, los árboles pequeños (pocas ramificaciones) tienen poca varianza pero no consiguen representar bien la relación entre las variables, es decir, tienen bias alto. En contraposición, los árboles grandes se ajustan mucho a los datos de entrenamiento, por lo que tienen muy poco bias pero mucha varianza. Una forma de solucionar este problema son los métodos de ensemble.
Los métodos de ensemble combinan múltiples modelos en uno nuevo con el objetivo de lograr un equilibro entre bias y varianza, consiguiendo así mejores predicciones que

cualquiera de los modelos individuales originales. Dos de los tipos de ensemble más utilizados son:

- **Bagging**: Se ajustan múltiples modelos, cada uno con un subconjunto distinto de los datos de entrenamiento. Para predecir, todos los modelos que forman el agregado participan aportando su predicción. Como valor final, se toma la media de todas las predicciones (variables continuas) o la clase más frecuente (variables categóricas). Los modelos *Random Forest* están dentro de esta categoría.
- **Boosting**: Se ajustan secuencialmente múltiples modelos sencillos, llamados *weak learners*, de forma que cada modelo aprende de los errores del anterior. Como valor final, al igual que en *bagging*, se toma la media de todas las predicciones (variables continuas) o la clase más frecuente (variables cualitativas). Tres de los métodos de *boosting* más empleados son *AdaBoost*, *Gradient Boosting* y *Stochastic Gradient Boosting*.

Aunque el objetivo final es el mismo, lograr un balance óptimo entre bias y varianza, existen dos diferencias importantes:

- Forma en que consiguen reducir el error total. El error total de un modelo puede descomponerse como bias+varianza+$\epsilon$$bias+varianza+\epsilon$. En *bagging*, se emplean modelos con muy poco bias pero mucha varianza, agregándolos se consigue reducir la varianza sin apenas inflar el bias. En *boosting*, se emplean modelos con muy poca varianza pero mucho bias, ajustando secuencialmente los modelos se reduce el bias. Por lo tanto, cada una de las estrategias reduce una parte del error total.
- Forma en que se introducen variaciones en los modelos que forman el *ensemble*. En *bagging*, cada modelo es distinto del resto porque cada uno se entrena con una muestra distinta obtenida mediante bootstrapping. En *boosting*, los modelos se ajustan secuencialmente y la importancia (peso) de las observaciones va cambiando en cada iteración, dando lugar a diferentes ajustes.

La clave para que los métodos de ensemble consigan mejores resultados que cualquiera de sus modelos individuales es que, los modelos que los forman, sean lo más diversos posibles (sus errores no estén correlacionados). Una analogía que refleja este concepto es la siguiente: supóngase un juego como el trivial en el que los equipos tienen que acertar preguntas sobre temáticas diversas. Un equipo formado por muchos jugadores, cada uno experto en un tema distinto, tendrá más posibilidades de ganar que un equipo formado por jugadores expertos en un único tema o por un único jugador que sepa un poco de todos los temas.
A continuación, se describe con más detalle la estrategia de bagging, sobre la que se fundamenta el modelo Random Forest.

Bagging

El término *bagging* es el diminutivo de *bootstrap aggregation*, y hace referencia al empleo del muestreo repetido con reposición bootstrapping con el fin de reducir la varianza de algunos modelos de aprendizaje estadístico, entre ellos los basados en árboles.

Dadas nn muestras de observaciones independientes $Z_1,\ldots, Zn$, cada una con varianza σ2, la varianza de la media de las observaciones $\bar{Z}$ es $\sigma 2/n$. En otras palabras, promediando un conjunto de observaciones se reduce la varianza. Basándose en esta idea, una forma de reducir la varianza y aumentar la precisión de un método predictivo es obtener múltiples muestras de la población, ajustar un modelo distinto con cada una de ellas, y hacer la media (la moda en el caso de variables cualitativas) de las predicciones resultantes. Como en la práctica no se suele tener acceso a múltiples muestras, se puede simular el proceso recurriendo a bootstrapping, generando así *pseudo-muestras* con los que ajustar diferentes modelos y después agregarlos. 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 de decisión, dada su naturaleza de bajo bias y alta varianza, bagging ha demostrado tener muy buenos resultados. La forma de aplicarlo es:

1. Generar BB pseudo-training sets mediante bootstrapping a partir de la muestra de entrenamiento original.
2. Entrenar un árbol con cada una de las BB muestras del paso 1. Cada árbol se crea sin apenas restricciones y no se somete a pruning, por lo que tiene varianza alta pero poco bias. En la mayoría de casos, la única regla de parada es el número mínimo de observaciones que deben tener los nodos terminales. El valor óptimo de este hiperparámetro puede obtenerse comparando el out of bag error o por validación cruzada.
3. Para cada nueva observación, obtener la predicción de cada uno de los BB árboles. El valor final de la predicción se obtiene como la media de las BB predicciones en el caso de variables cuantitativas y como la clase predicha más frecuente (moda) para variables cualitativas.

En el proceso de bagging, el número de árboles creados no es un hiperparámetro crítico en cuanto a que, por mucho que se incremente el número, no se aumenta el riesgo de overfitting. Alcanzado un determinado número de árboles, la reducción de test error se estabiliza. A pesar de ello, cada árbol ocupa memoria, por lo que no conviene almacenar más de los necesarios.

Entrenamiento de Random Forest

El algoritmo de *Random Forest* es una modificación del proceso de *bagging* que consigue mejorar los resultados gracias a que *decorrelaciona* aún más los árboles generados en el proceso.
Recordando el apartado anterior, los beneficios del *bagging* se basan en el hecho de que, promediando un conjunto de modelos, se consigue reducir la varianza. Esto es cierto siempre y cuando los modelos agregados no estén correlacionados. Si la correlación 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* estarán dominados por el mismo predictor y serán muy parecidos entre ellos. Como consecuencia de la alta correlación entre los árboles, el proceso de *bagging* apenas conseguirá disminuir la varianza y, por lo tanto, tampoco mejorar el modelo. *Random forest* evita este problema haciendo una selección aleatoria de mm predictores antes de evaluar cada división. De esta forma, un promedio de p$(p-m)/p$ divisiones no contemplarán el predictor influyente, permitiendo que otros predictores puedan ser seleccionados. Añadiendo este paso extra se consigue *decorrelacionar* los árboles todavía más, con lo que su agregación consigue una mayor reducción de la 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 mm predictores. La diferencia en el resultado dependerá del valor mm escogido. Si $m=p$ los resultados de *random forest* y *bagging* son equivalentes. Algunas recomendaciones son:

- La raíz cuadrada del número total de predictores para problemas de clasificación. m≈p$\sqrt{m\approx p}$
- Un tercio del número de predictores para problemas de regresión. m≈p3$m\approx p3$
- Si los predictores están muy correlacionados, valores pequeños de mm consiguen mejores resultados.

Sin embargo, la mejor forma para encontrar el valor óptimo de mm es evaluar el *out-of-bag-error* o recurrir a validación cruzada.

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

Predicción de Random Forest

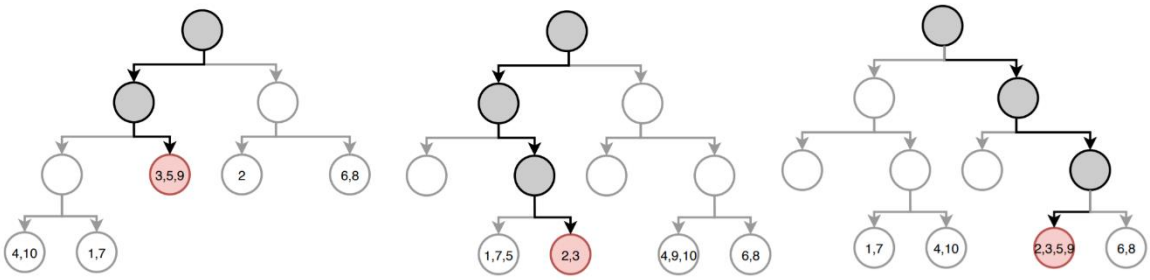
La predicción de un modelo *Random Forest* es la media de las predicciones de todos los árboles que lo forman.

Supóngase que se dispone de 10 observaciones, cada una con un valor de variable respuesta Y y unos predictores X.

	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
id	1	2	3	4	5	6	7	8	9
Y	10	18	24	8	2	9	16	10	20
X	…	…	…	…	…	…	…	…	…

3 rows | 1-10 of 11 columns

La siguiente imagen muestra cómo sería la predicción del modelo para una nueva observación. En cada árbol, el camino hasta llegar al nodo final está resaltado. En cada nodo terminal se detalla el índice de las observaciones de entrenamiento que forman parte.



Predicción con random forest: en cada árbol, el camino hasta llegar al nodo final está resaltado. En cada nodo terminal se detalla el índice de las observaciones de entrenamiento que forman parte de él.

El valor predicho por cada árbol es la media de la variable respuesta *Y* en el nodo terminal. Acorde a la imagen, las predicciones de cada uno de los tres árboles (de izquierda a derecha) es:

$$\hat{y}_{arbol_1} = \frac{24 + 2 + 20}{3} = 15.33333$$

$$\hat{y}_{arbol_2} = \frac{18 + 24}{2} = 21$$

$$\hat{y}_{arbol_3} = \frac{18 + 24 + 2 + 20}{4} = 16$$

La predicción final del modelo es la media de todas las predicciones individuales:

$$\hat{\mu} = \frac{15.33333 + 21 + 16}{3} = 17.4$$

Aunque la anterior es la forma más común de obtener las predicciones de un modelo Random Forest, existe otra aproximación. La predicción de un árbol de regresión puede verse como una variante de vecinos cercanos en la que, solo las observaciones que forman parte del mismo nodo terminal que la observación predicha, tienen influencia. Siguiendo esta aproximación, la predicción del árbol se define como la media ponderada de todas las observaciones de entrenamiento, donde el peso de cada observación depende únicamente de si forma parte o no del mismo nodo terminal. Para Random Forest esto equivale a la media ponderada de todas las observaciones, empleando como pesos ww la media de los vectores de pesos de todos los árboles.
Acorde a la imagen anterior, el vector de pesos para cada uno de los tres árboles (de izquierda a derecha) es:

$$\mathbf{w}_{arbol_1} = (0, 0, \frac{1}{3}, 0, \frac{1}{3}, 0, 0, 0, \frac{1}{3}, 0)$$

$$\mathbf{w}_{arbol_2} = (0, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0)$$

$$\mathbf{w}_{arbol_3} = (0, \frac{1}{4}, \frac{1}{4}, 0, \frac{1}{4}, 0, 0, 0, \frac{1}{4}, 0)$$

La media de todos los vectores de pesos es:

$$\overline{\mathbf{w}} = \frac{1}{3}(\mathbf{w}_{arbol_1} + \mathbf{w}_{arbol_2} + \mathbf{w}_{arbol_3}) =$$
$$(0, \frac{1}{4}, \frac{13}{36}, 0, \frac{7}{36}, 0, 0, 0, \frac{7}{36}, 0)$$

Una vez obtenido el vector de pesos promedio, se puede calcular la predicción con la media ponderada de todas las observaciones de entrenamiento:

$$\hat{\mu} = \sum_{i=1}^{n} \overline{\mathbf{w}}_i Y_i$$

$$\hat{\mu} = (0 \times 10) + (\frac{1}{4} \times 18) + (\frac{13}{36} \times 24) + (0 \times 8) + (\frac{1}{4} \times 2) +$$
$$(0 \times 9) + (0 \times 16) + (0 \times 10) + (\frac{1}{4} \times 20) + (0 \times 14) = 17.4$$

Out-of-Bag Error

Dada la naturaleza del proceso de *bagging*, resulta posible estimar el error de test sin necesidad de recurrir a métodos de validación cruzada (*cross-validation*). El hecho de que los árboles se ajusten empleando muestras generadas por *bootstrapping* 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 y promediándolos (la moda en el caso de los árboles de clasificación). Siguiendo este proceso, se pueden obtener las predicciones para las *n* observaciones y con ellas calcular el *OOB-mean square error* (para regresión) o el *OOB-classification error* (para árboles de clasificación). Como la variable respuesta de cada observación se predice empleando únicamente los árboles en cuyo ajuste no participó dicha observación, el *OOB-error* sirve como estimación del error de test. De hecho, si el número de árboles es suficientemente alto, el *OOB-error* es prácticamente equivalente al *leave-one-out cross-validation error*.

Esta es una ventaja añadida de los métodos de *bagging*, y por lo tanto de *Random Forest* ya que evita tener que recurrir al proceso de validación cruzada (computacionalmente costoso) para la optimización de los hiperparámetros.
Dos limitaciones en el uso *Out-of-Bag Error*:

- El *Out-of-Bag Error* no es adecuado cuando las observaciones tienen una relación temporal (series temporales). Como la selección de las observaciones que participan en cada entrenamiento es aleatoria, no respetan el orden temporal y se estaría introduciendo información a futuro.
- El preprocesado de los datos de entrenamiento se hace de forma conjunta, por lo que las observaciones *out-of-bag* pueden sufrir *data leakage*. De ser así, las estimaciones del *OOB-error* son demasiado optimistas.

En un muestreo por *bootstrapping*, si el tamaño de los datos de entrenamiento es n, cada observación tiene una probabilidad de ser elegida de $1n$. Por lo tanto, la probabilidad de no ser elegida en todo el proceso es de $(1-1/n)n$, lo que converge en $1/\epsilon$, que es aproximadamente un tercio.

Importancia de los predictores

Si bien es cierto que el proceso de *bagging* (*Random Forest*) 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. Al tratarse de una combinación de múltiples árboles, no es posible obtener una representación gráfica sencilla del modelo y no es inmediato identificar de forma visual qué predictores son más importantes. Sin embargo, se han desarrollado nuevas estrategias para cuantificar la importancia de los predictores que hacen de los modelos de *bagging* (*Random Forest*) una herramienta muy potente, no solo para predecir, sino también para el análisis exploratorio. Dos de estas medidas son: importancia por permutación e impureza de nodos.

Importancia por permutación
Identifica la influencia que tiene cada predictor sobre una determinada métrica de evaluación del modelo (estimada por out-of-bag error o validación cruzada). El valor asociado con cada predictor se obtiene de la siguiente forma:

1. Crear el conjunto de árboles que forman el modelo.
2. Calcular una determinada métrica de error (mse, classification error, …). Este es el valor de referencia ($error0$).
3. Para cada predictor j:
4. Permutar en todos los árboles del modelo los valores del predictor j manteniendo el resto constante.
5. Recalcular la métrica tras la permutación, llámese (errorj).

6. Calcular el incremento en la métrica debido a la permutación del predictor j.
   $\%Incrementoj=(errorj-error0)/error0*100$

Si el predictor permutado estaba contribuyendo al modelo, es de esperar que el modelo aumente su error, ya que se pierde la información que proporcionaba esa variable. El porcentaje en que se incrementa el error debido a la permutación del predictor j puede interpretarse como la influencia que tiene j sobre el modelo. Algo que suele llevar a

confusiones es el hecho de que este incremento puede resultar negativo. Si la variable no contribuye al modelo, es posible que, al reorganizarla aleatoriamente, solo por azar, se consiga mejorar ligeramente el modelo, por lo que ($errorj - error0$) es negativo. A modo general, se puede considerar que estas variables tienen una importancia próxima a cero.
Aunque esta estrategia suele ser la más recomendada, cabe tomar algunas precauciones en su interpretación. Lo que cuantifican es la influencia que tienen los predictores sobre el modelo, no su relación con la variable respuesta. ¿Por qué es esto tan importante? Supóngase un escenario en el que se emplea esta estrategia con la finalidad de identificar qué predictores están relacionados con el peso de una persona, y que dos de los predictores son: el índice de masa corporal (IMC) y la altura. Como IMC y altura están muy correlacionados entre sí (la información que aportan es redundante), cuando se permute uno de ellos, el impacto en el modelo será mínimo, ya que el otro aporta la misma información. Como resultado, estos predictores aparecerán como poco influyentes aun cuando realmente están muy relacionados con la variable respuesta. Una forma de evitar problemas de este tipo es, siempre que se excluyan predictores de un modelo, comprobar el impacto que tiene en su capacidad predictiva.

Incremento de la pureza de nodos

Cuantifica el incremento total en la pureza de los nodos debido a divisiones en las que participa el predictor (promedio de todos los árboles). La forma de calcularlo es la siguiente: en cada división de los árboles, se registra el descenso conseguido en la medida empleada como criterio de división (índice Gini, mse entropía, …). Para cada uno de los predictores, se calcula el descenso medio conseguido en el conjunto de árboles que forman el ensemble. Cuanto mayor sea este valor medio, mayor la contribución del predictor en el modelo.

Ajuste del modelo

Se ajusta un modelo empleando como variable respuesta medv y como predictores todas las otras variables disponibles.
La función ranger del paquete ranger permite entrenar modelos random forest para problemas de regresión. Los parámetros e hiperparámetros empleados por defecto son:

- `formula = NULL`
- `data = NULL`
- `num.trees = 500`
- `mtry = NULL`
- `importance = "none"`
- `write.forest = TRUE`
- `probability = FALSE`
- `min.node.size = NULL`
- `max.depth = NULL`
- `replace = TRUE`
- `sample.fraction = ifelse(replace, 1, 0.632)`
- `case.weights = NULL`
- `class.weights = NULL`
- `splitrule = NULL`
- `num.random.splits = 1`
- `alpha = 0.5`
- `minprop = 0.1`

- split.select.weights = NULL
- always.split.variables = NULL
- respect.unordered.factors = NULL
- scale.permutation.importance = FALSE
- local.importance = FALSE
- regularization.factor = 1
- regularization.usedepth = FALSE
- keep.inbag = FALSE
- inbag = NULL
- holdout = FALSE
- quantreg = FALSE
- oob.error = TRUE
- num.threads = NULL
- save.memory = FALSE
- verbose = TRUE
- seed = NULL
- dependent.variable.name = NULL
- status.variable.name = NULL
- classification = NULL
- x = NULL
- y = NULL

De entre todos ellos, destacan aquellos que detienen el crecimiento de los árboles, los que controlan el número de árboles y predictores incluidos, y los que gestionan la paralelización:

- num.trees; número de árboles incluidos en el modelo.
- max.depth: profundidad máxima que pueden alcanzar los árboles. Por defecto (0) crece al máximo los árboles.
- min.node.size: número mínimo de observaciones que debe de tener cada uno de los nodos hijos para que se produzca la división. Por defecto emplea 1 para problemas de clasificación y 10 para problemas de regresión.
- mtry: número de predictores considerados en cada división. Por defecto, se emplea como valor la raíz cuadrada del número total de predictores disponible (redondeado a la baja).
- oob.error: Si se calcula o no el out-of-bag error (accuracy, mean squared error o R2$R2$). Por defecto es FALSE ya que aumenta el tiempo de entrenamiento.
- num.threads: número de cores empleados para el entrenamiento. En random forest los árboles se ajustan de forma independiente, por lo que la paralelización reduce notablemente el tiempo de entrenamiento. Por defecto se utilizan todos los cores disponibles.
- seed: semilla para que los resultados sean reproducibles.

Como en todo estudio predictivo, no solo es importante ajustar el modelo, sino también cuantificar su capacidad para predecir nuevas observaciones. Para poder hacer esta evaluación, se dividen los datos en dos grupos, uno de entrenamiento y otro de test

```
###### RANDOMFOREST
# continua Ejemplo regresion
# Tratamiento de datos
```

```
library(MASS)
library(dplyr)
library(tidyr)
library(skimr)
# Gráficos
library(ggplot2)
library(ggpubr)
# Preprocesado y modelado
library(tidymodels)
library(ranger)
library(doParallel)
```

```
Boston <- read.csv("Boston.csv",head=T, sep=";")
library (MASS)
library(ggplot2)
library(randomForest)
```

```
Boston$chas <- as.factor(Boston$chas)
str(Boston)
```

```
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
##  $ chas   : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524
0.524 ...
##  $ rm     : num  6.58 6.42 7.18 7 7.15 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
##  $ dis    : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ rad    : int  1 2 2 3 3 3 5 5 5 5 ...
##  $ tax    : int  296 242 242 222 222 222 311 311 311 311 ...
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
##  $ black  : num  397 397 393 395 397 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

```
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio  black
lstat
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
4.98
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
9.14
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
4.03
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
2.94
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
5.33
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
5.21
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
## 4 33.4
## 5 36.2
## 6 28.7
```

```
# creando data train (80%) y test 20%
set.seed(123)
train <- sample(1:nrow(Boston), size = nrow(Boston)*0.8)
data_train <- Boston[train,]
data_test <- Boston[-train,]
```

```
# Creacion y entrenamiento del modelo
set.seed(123)
require(randomForest)
library(ranger)
modelo  <- ranger(
  formula   = medv ~ .,
  data = data_train,
  num.trees = 12,
```

```
  seed      = 123
)

print(modelo)
```

```
## Ranger result
##
## Call:
## ranger(formula = medv ~ ., data = data_train, num.trees = 12,seed=123)
##
## Type:                             Regression
## Number of trees:                  12
## Sample size:                      404
## Number of independent variables:  13
## Mtry:                             3
## Target node size:                 5
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       16.80881
## R squared (OOB):                  0.8013986
```

```
ncol(Boston)
```

```
## [1] 14
summary(modelo)
```

```
##                           Length Class         Mode
## predictions               404    -none-        numeric
## num.trees                   1    -none-        numeric
## num.independent.variables   1    -none-        numeric
## mtry                        1    -none-        numeric
## min.node.size               1    -none-        numeric
## prediction.error            1    -none-        numeric
## forest                      8    ranger.forest list
## splitrule                   1    -none-        character
## treetype                    1    -none-        character
```

```
## r.squared                    1    -none-        numeric
## call                         5    -none-        call
## importance.mode              1    -none-        character
## num.samples                  1    -none-        numeric
## replace                      1    -none-        logical
## dependent.variable.name      1    -none-        character
```

Predicción y evaluación del modelo

Una vez entrenado el modelo, se evalúa la capacidad predictiva empleando el conjunto de test.

```
# Error de test del modelo
predicciones <- predict(
  modelo,
  data = data_test
)

predicciones <- predicciones$predictions
test_rmse    <- sqrt(mean((predicciones - data_test$medv)^2))
paste("Error de test (rmse) del modelo:", round(test_rmse,2))
```

```
## [1] "Error de test (rmse) del modelo: 3.74"
```

Optimización de hiperparámetros

El modelo inicial se ha entrenado utilizando 10 árboles (num.trees=10) y manteniendo el resto de hiperparámetros con su valor por defecto. Al ser hiperparámetros, no se puede saber de antemano cuál es el valor más adecuado, la forma de identificarlos es mediante el uso de estrategias de validación, por ejemplo validación cruzada.

Los modelos *Random Forest* tienen la ventaja de disponer del *Out-of-Bag error*, lo que permite obtener una estimación del error de test sin recurrir a la validación cruzada, que es computacionalmente costosa. En la implementación de ranger, para problemas de regresión, se calculan dos métricas como *oob_score*: el *mean*

squared error y el $R2R2$. Si se desea otra, se tiene que recurrir a validación cruzada.
Cabe tener en cuenta que, cuando se busca el valor óptimo de un hiperparámetro con dos métricas distintas, el resultado obtenido raramente es el mismo. Lo importante es que ambas métricas identifiquen las mismas regiones de interés. A continuación, se muestran las dos aproximaciones.

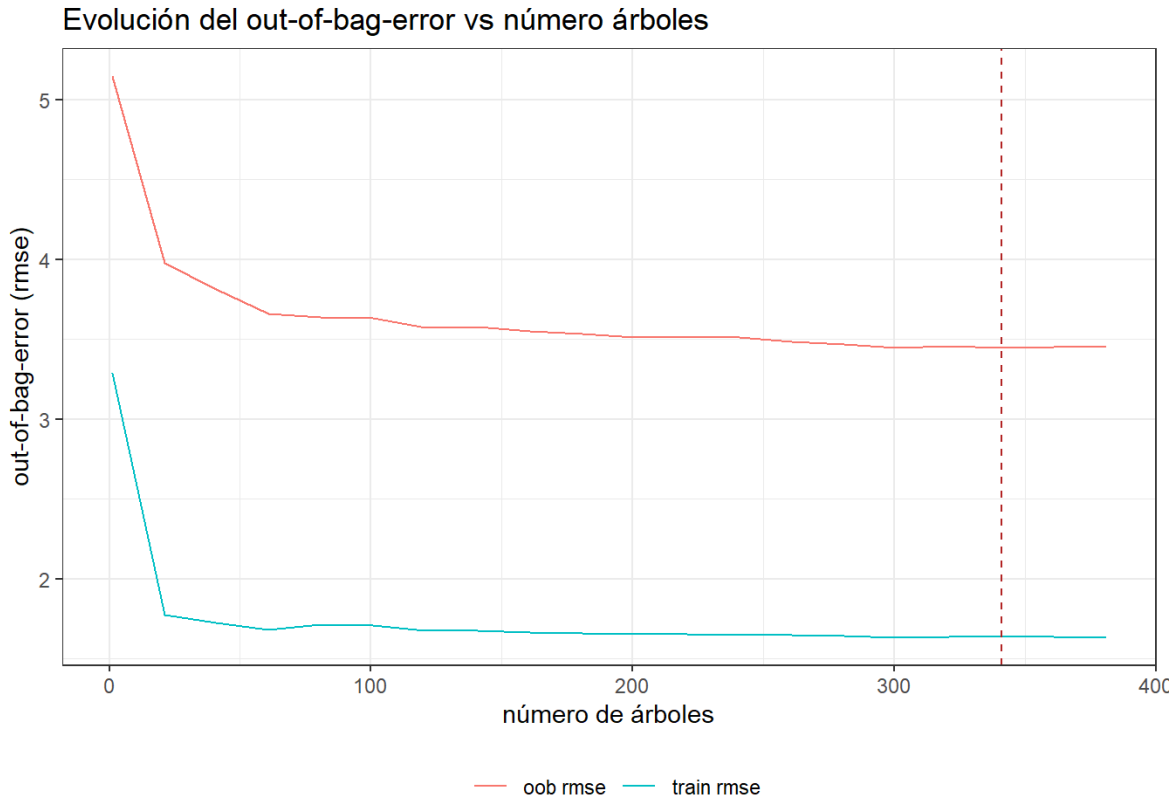
Número de árboles

En *Random Forest*, el número de árboles no es un hiperparámetro crítico en cuanto que, añadir árboles, solo puede hacer que mejorar el resultado. En *Random Forest* no se produce *overfitting* por exceso de árboles sin embargo, añadir árboles una vez que la mejora se estabiliza es una pérdida de recursos computacionales.

```
Validación empleando el Out-of-Bag error (root mean squared error)
```

```
# Optimización de hiperparámetros
# Validación empleando el Out-of-Bag error (root mean squared error)

# Valores evaluados
num_trees_range <- seq(1, 400, 20)

# Bucle para entrenar un modelo con cada valor de num_trees y extraer su error
# de entrenamiento y de Out-of-Bag.

train_errors <- rep(NA, times = length(num_trees_range))
oob_errors   <- rep(NA, times = length(num_trees_range))

for (i in seq_along(num_trees_range)){
  modelo  <- ranger(
    formula   = medv ~ .,
    data      = data_train,
    num.trees = num_trees_range[i],
    oob.error = TRUE,
    seed      = 123
  )

  predicciones_train <- predict(
```

```
    modelo,
    data = data_train
  )
  predicciones_train <- predicciones_train$predictions
  train_error <- mean((predicciones_train - data_train$medv)^2)
  oob_error   <- modelo$prediction.error
  train_errors[i] <- sqrt(train_error)
  oob_errors[i]   <- sqrt(oob_error)
  }


# Gráfico con la evolución de los errores
df_resulados <- data.frame(n_arboles = num_trees_range, train_errors, oob
_errors)
ggplot(data = df_resulados) +
  geom_line(aes(x = num_trees_range, y = train_errors, color = "train rms
e")) +
  geom_line(aes(x = num_trees_range, y = oob_errors, color = "oob rmse"))
+
  geom_vline(xintercept = num_trees_range[which.min(oob_errors)],
             color = "firebrick",
             linetype = "dashed") +
  labs(
    title = "Evolución del out-of-bag-error vs número árboles",
    x     = "número de árboles",
    y     = "out-of-bag-error (rmse)",
    color = ""
  ) +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
paste("Valor óptimo de num.trees:", num_trees_range[which.min(oob_errors)
])
```

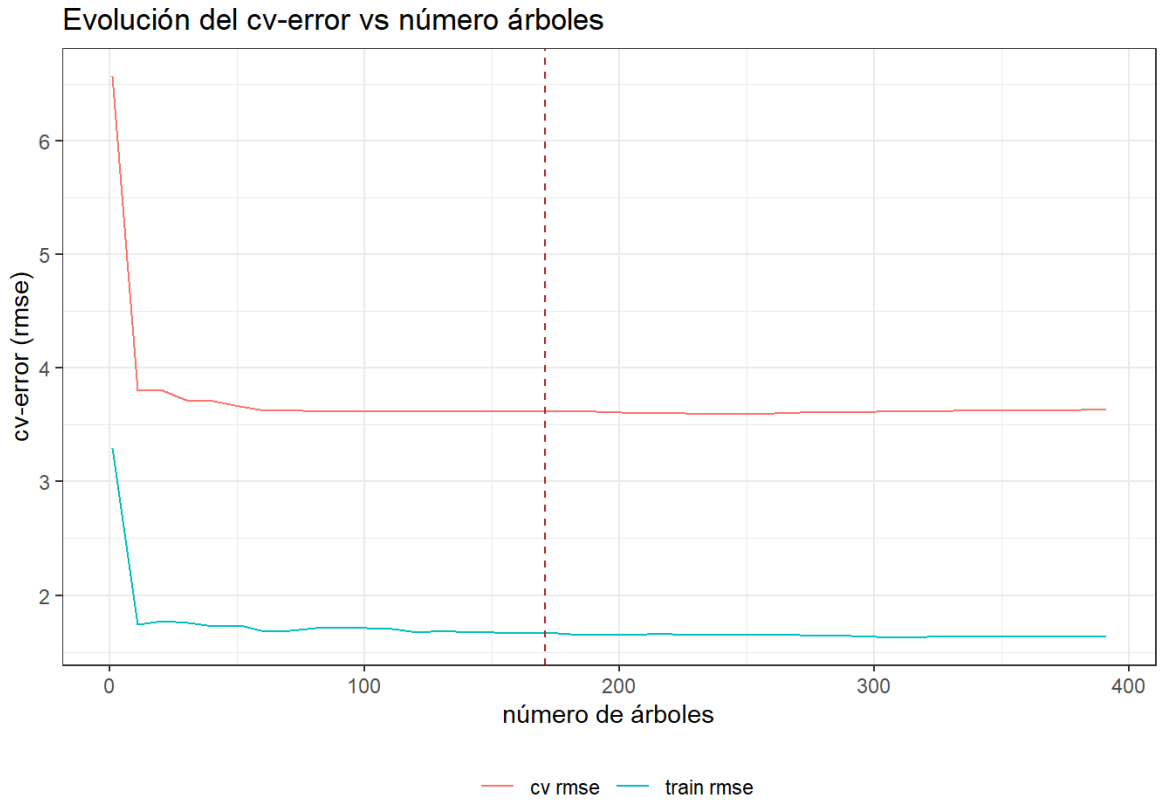
```
## [1] "Valor óptimo de num.trees: 341"
```

El paquete ranger no tiene ninguna funcionalidad propia para realizar validación cruzada, por lo que se recurre a tidymodels. Puede encontrarse una descripción más detallada de este proceso en Machine Learning con R y tidymodels.
Otro *framework* excelente para *machine learning* en **R** es mlr3.

```
# Validación empleando k-cross-validation (root mean squared error)

# Valores evaluados
num_trees_range <- seq(1, 400, 10)


# Bucle para entrenar un modelo con cada valor de num_trees y extraer su
error
# de entrenamiento y de Out-of-Bag.


train_errors <- rep(NA, times = length(num_trees_range))
cv_errors    <- rep(NA, times = length(num_trees_range))
```

```
for (i in seq_along(num_trees_range)){

  # Definición del modelo
  modelo <- rand_forest(
    mode  = "regression",
    trees = num_trees_range[i]
  ) %>%
    set_engine(
      engine = "ranger",
      seed   = 123
    )

  # Particiones validación cruzada
  set.seed(1234)
  cv_folds <- vfold_cv(
    data    = data_train,
    v       = 5,
    repeats = 1
  )

  # Ejecución validación cruzada
  validacion_fit <- fit_resamples(
    preprocessor = medv ~ .,
    object       = modelo,
    resamples    = cv_folds,
    metrics      = metric_set(rmse)
  )

  # Extraer la métrica de validación
  cv_error <- collect_metrics(validacion_fit)$mean

  # Predicción datos train
  modelo_fit <- modelo %>% fit(medv ~ ., data = data_train)
  predicciones_train <- predict(
    modelo_fit,
    new_data = data_train
  )
```

```
  predicciones_train <- predicciones_train$.pred
  train_error <- sqrt(mean((predicciones_train - data_train$medv)^2))


  # Resultados
  train_errors[i] <- train_error
  cv_errors[i]    <- cv_error
  }
# Gráfico con la evolución de los errores
df_resulados <- data.frame(n_arboles = num_trees_range, train_errors, cv_
errors)
ggplot(data = df_resulados) +
  geom_line(aes(x = num_trees_range, y = train_errors, color = "train rms
e")) +
  geom_line(aes(x = num_trees_range, y = cv_errors, color = "cv rmse")) +
  geom_vline(xintercept = num_trees_range[which.min(oob_errors)],
             color = "firebrick",
             linetype = "dashed") +
  labs(
    title = "Evolución del cv-error vs número árboles",
    x     = "número de árboles",
    y     = "cv-error (rmse)",
    color = ""
  ) +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
paste("Valor óptimo de num.trees:", num_trees_range[which.min(cv_errors)]
)
```

```
## [1] "Valor óptimo de num.trees: 231"
```

Ambas métricas indican que, a partir de entre 50 y 100 árboles, el error de validación del modelo se estabiliza.

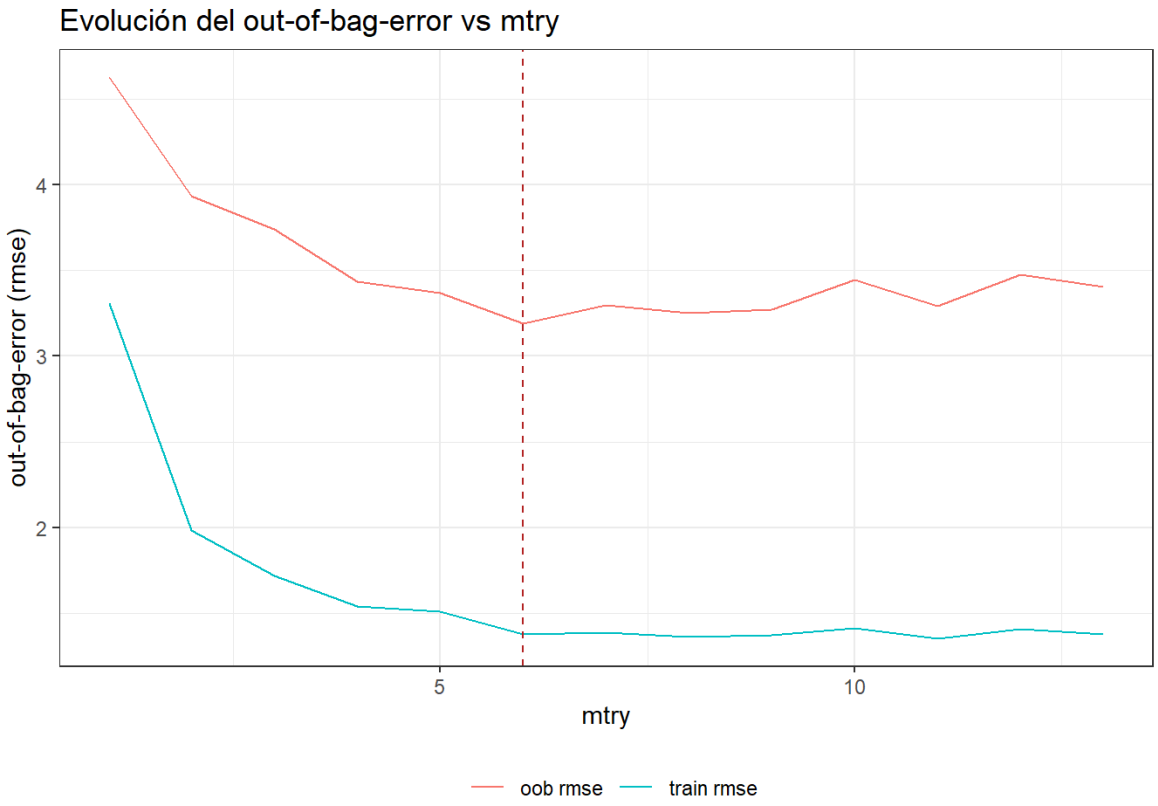
mtry

El valor de mtry es uno de los hiperparámetros más importantes de *random forest*, ya que es el que permite controlar cuánto se decorrelacionan los árboles entre sí.

```
####  mtry
# Validación empleando el Out-of-Bag error (root mean squared error)
# Valores evaluados
mtry_range <- seq(1, ncol(data_train)-1)


# Bucle para entrenar un modelo con cada valor de mtry y extraer su error
```

```
# de entrenamiento y de Out-of-Bag.

train_errors <- rep(NA, times = length(mtry_range))
oob_errors   <- rep(NA, times = length(mtry_range))

for (i in seq_along(mtry_range)){
  modelo  <- ranger(
    formula   = medv ~ .,
    data      = data_train,
    num.trees = 50,
    mtry      = mtry_range[i],
    oob.error = TRUE,
    seed      = 123
  )

  predicciones_train <- predict(
    modelo,
    data = data_train
  )
  predicciones_train <- predicciones_train$predictions
  train_error <- mean((predicciones_train - data_train$medv)^2)
  oob_error   <- modelo$prediction.error
  train_errors[i] <- sqrt(train_error)
  oob_errors[i]   <- sqrt(oob_error)
 }
# Gráfico con la evolución de los errores
df_resulados <- data.frame(mtry = mtry_range, train_errors, oob_errors)
ggplot(data = df_resulados) +
  geom_line(aes(x = mtry_range, y = train_errors, color = "train rmse"))
+
  geom_line(aes(x = mtry_range, y = oob_errors, color = "oob rmse")) +
  geom_vline(xintercept =  mtry_range[which.min(oob_errors)],
             color = "firebrick",
             linetype = "dashed") +
  labs(
    title = "Evolución del out-of-bag-error vs mtry",
    x     = "mtry",
    y     = "out-of-bag-error (rmse)",
```

```
    color = ""
  ) +
  theme_bw() +
  theme(legend.position = "bottom")
```



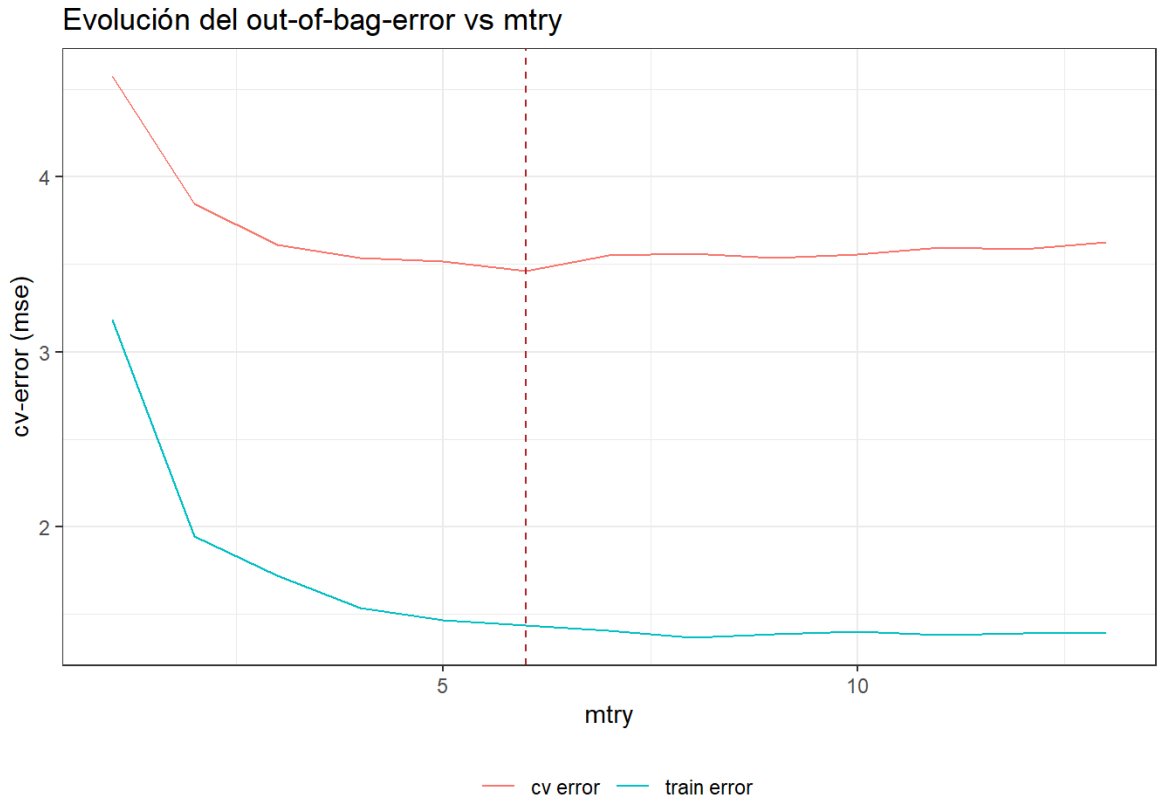

```
paste("Valor óptimo de mtry:", mtry_range[which.min(oob_errors)])
```

```
## [1] "Valor óptimo de mtry: 6"
```

```
# Validación empleando k-cross-validation (root mean squared error)
# Valores evaluados
mtry_range <- seq(1, ncol(data_train)-1)

# Bucle para entrenar un modelo con cada valor de mtry y extraer su error
# de entrenamiento y de Out-of-Bag.

train_errors <- rep(NA, times = length(mtry_range))
cv_errors    <- rep(NA, times = length(mtry_range))
```

```
for (i in seq_along(mtry_range)){

  # Definición del modelo
  modelo <- rand_forest(
    mode  = "regression",
    trees = 100,
    mtry  = mtry_range[i]
  ) %>%
    set_engine(
      engine = "ranger",
      seed   = 123
    )

  # Particiones validación cruzada
  set.seed(1234)
  cv_folds <- vfold_cv(
    data    = data_train,
    v       = 5,
    repeats = 1
  )

  # Ejecución validación cruzada
  validacion_fit <- fit_resamples(
    preprocessor = medv ~ .,
    object       = modelo,
    resamples    = cv_folds,
    metrics      = metric_set(rmse)
  )

  # Extraer datos de validación
  cv_error <- collect_metrics(validacion_fit)$mean

  # Predicción datos train
  modelo_fit <- modelo %>% fit(medv ~ ., data = data_train)
  predicciones_train <- predict(
    modelo_fit,
    new_data = data_train
```

```
  )
  predicciones_train <- predicciones_train$.pred
  train_error <- sqrt(mean((predicciones_train - data_train$medv)^2))

  # Resultados
  train_errors[i] <- train_error
  cv_errors[i]    <- cv_error
}

# Gráfico con la evolución de los errores
df_resulados <- data.frame(mtry = mtry_range, train_errors, cv_errors)
ggplot(data = df_resulados) +
  geom_line(aes(x = mtry_range, y = train_errors, color = "train error"))
+
  geom_line(aes(x = mtry_range, y = cv_errors, color = "cv error")) +
  geom_vline(xintercept =  mtry_range[which.min(cv_errors)],
             color = "firebrick",
             linetype = "dashed") +
  labs(
    title = "Evolución del out-of-bag-error vs mtry",
    x     = "mtry",
    y     = "cv-error (mse)",
    color = ""
  ) +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
paste("Valor óptimo de mtry:", mtry_range[which.min(cv_errors)])
```

```
## [1] "Valor óptimo de mtry: 6"
```

Grid search

Aunque el análisis individual de los hiperparámetros es útil para entender su impacto en el modelo e identificar rangos de interés, la búsqueda final no debe hacerse de forma secuencial, ya que cada hiperparámetro interacciona con los demás. Es preferible recurrir a *grid search* o *random search* para analizar varias combinaciones de hiperparámetros. Puede encontrarse más información sobre las estrategias de búsqueda en Tidymodels y en mlr3.

Grid Search basado en out-of-bag error

```
######  Grid search
# Grid Search basado en out-of-bag error
# Grid de hiperparámetros evaluados
param_grid = expand_grid(
  'num_trees' = c(50, 100, 500, 1000, 5000),
```

```
  'mtry'      = c(3, 5, 7, ncol(data_train)-1),
  'max_depth' = c(1, 3, 10, 20)
)
# Loop para ajustar un modelo con cada combinación de hiperparámetros
oob_error = rep(NA, nrow(param_grid))
for(i in 1:nrow(param_grid)){
  modelo <- ranger(
    formula   = medv ~ .,
    data      = data_train,
    num.trees = param_grid$num_trees[i],
    mtry      = param_grid$mtry[i],
    max.depth = param_grid$max_depth[i],
    seed      = 123
  )
  oob_error[i] <- sqrt(modelo$prediction.error)
}
# Resultados
resultados <- param_grid
resultados$oob_error <- oob_error
resultados <- resultados %>% arrange(oob_error)

# Mejores hiperparámetros por out-of-bag error
head(resultados, 1)
```

```
## # A tibble: 1 × 4
##   num_trees  mtry max_depth oob_error
##       <dbl> <dbl>     <dbl>     <dbl>
## 1      5000     7        20      3.16
```

Grid Search basado en validación cruzada

```
##### Grid Search basado en validación cruzada
# DEFINICIÓN DEL MODELO Y DE LOS HIPERPARÁMETROS A OPTIMIZAR
modelo <- rand_forest(
  mode  = "regression",
```

```
  mtry  = tune(),
  trees = tune()
) %>%
  set_engine(
    engine     = "ranger",
    max.depth  = tune(),
    importance = "none",
    seed       = 123
  )

# DEFINICIÓN DEL PREPROCESADO
# ============================
# En este caso no hay preprocesado, por lo que el transformer solo contie
ne
# la definición de la fórmula y los datos de entrenamiento.
library(recipes)
transformer <- recipe(
  formula = medv ~ .,
  data    =  data_train
)

# DEFINICIÓN DE LA ESTRATEGIA DE VALIDACIÓN Y CREACIÓN DE PARTICIONES
# ============================================================
set.seed(1234)
cv_folds <- vfold_cv(
  data    = data_train,
  v       = 5,
  strata  = medv
)

# WORKFLOW
library(workflows)
workflow_modelado <- workflow() %>%
  add_recipe(transformer) %>%
  add_model(modelo)

# GRID DE HIPERPARÁMETROS
```

```
hiperpar_grid <- expand_grid(
  'trees'     = c(50, 100, 500, 1000, 5000),
  'mtry'      = c(3, 5, 7, ncol(data_train)-1),
  'max.depth' = c(1, 3, 10, 20)
)
# EJECUCIÓN DE LA OPTIMIZACIÓN DE HIPERPARÁMETROS
library(parallel)
library(doParallel)
cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)
grid_fit <- tune_grid(
  object    = workflow_modelado,
  resamples = cv_folds,
  metrics   = metric_set(rmse),
  grid      = hiperpar_grid
)

stopCluster(cl)

# Mejores hiperparámetros por validación cruzada
show_best(grid_fit, metric = "rmse", n = 1)
```

```
## # A tibble: 1 × 9
##    mtry trees max.depth .metric .estimator  mean     n std_err .config
##   <dbl> <dbl>     <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     7  5000        20 rmse    standard    3.16     5   0.240 Preprocessor1_M
o…
```

Una vez identificados los mejores hiperparámetros, se reentrena el modelo indicando los valores óptimos en sus argumentos.

```
# ENTRENAMIENTO FINAL
mejores_hiperpar <- select_best(grid_fit, metric = "rmse")
modelo_final_fit <- finalize_workflow(
  x = workflow_modelado,
  parameters = mejores_hiperpar
) %>%
```

```
  fit(
    data = data_train
  ) %>%
  pull_workflow_fit()
```

```
# Error de test del modelo final
# ===========
predicciones <- modelo_final_fit %>%
  predict(
    new_data = data_test,
    type     = "numeric"
  )
predicciones <- predicciones %>%
bind_cols(data_test %>% dplyr::select(medv))
rmse_test  <- rmse(
  data     = predicciones,
  truth    = medv,
  estimate = .pred,
  na_rm    = TRUE
)
rmse_test
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        3.55
```

Tras optimizar los hiperparámetros, se consigue reducir el error *rmse* del modelo de 4.15 a 3.57. Las predicciones del modelo final se alejan en promedio 3.57 unidades (3570 dólares) del valor real.

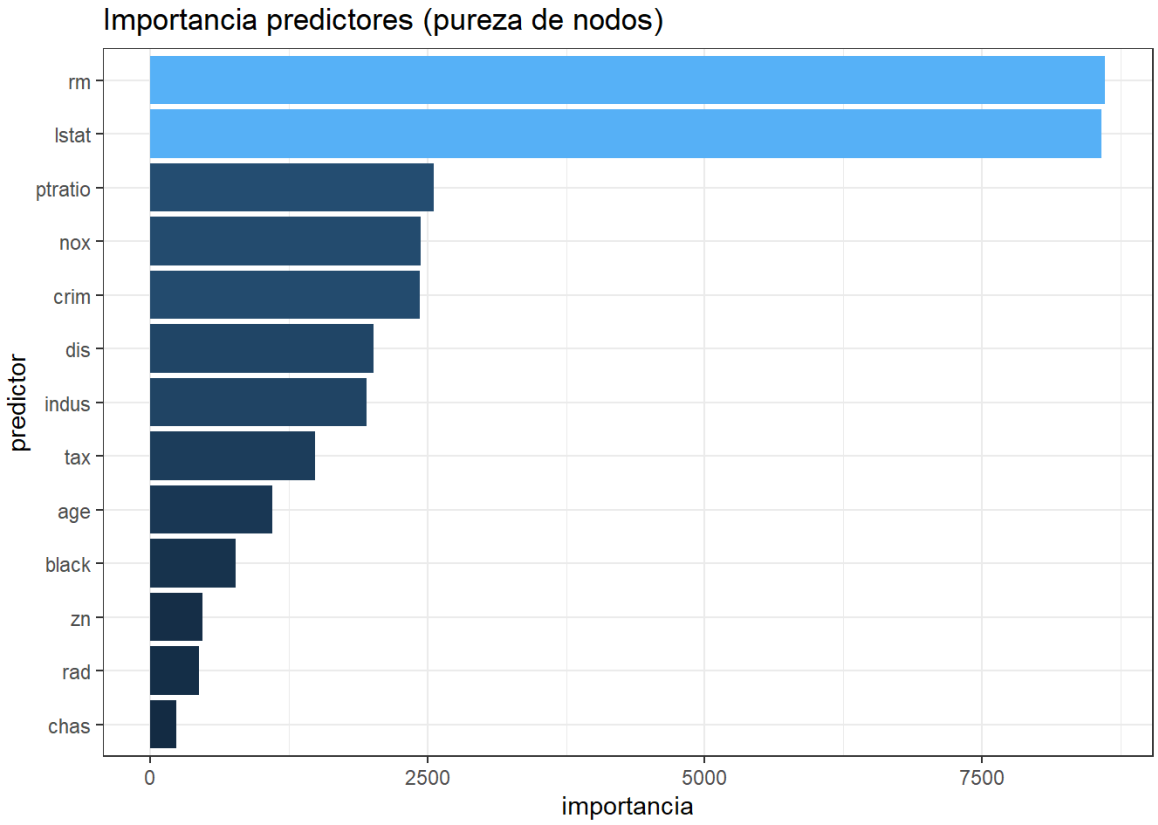
Importancia de predictores

Importancia por pureza de nodos

En los modelos anteriores, el argumento `importance` se deja por defecto como `"none"`. Esto desactiva el cálculo de importancia de predictores para reducir así el tiempo de entrenamiento. Se entrena de nuevo el modelo, con los mejores

hiperparámetros encontrados, pero esta vez indicando importance = "impurity". Los modelos ranger calculan la impureza a partir del índice Gini*Gini* en problemas de clasificación y con la varianza en regresión.

```
#### Importancia de predictores
# Entrenamiento modelo
modelo <- rand_forest(
  mode  = "regression"
) %>%
  set_engine(
    engine     = "ranger",
    importance = "impurity",
    seed       = 123
  )

modelo <- modelo %>% finalize_model(mejores_hiperpar)
modelo <- modelo %>% fit(medv ~., data = data_train)

# Importancia
importancia_pred <- modelo$fit$variable.importance %>%
  enframe(name = "predictor", value = "importancia")

# Gráfico
ggplot(
  data = importancia_pred,
  aes(x    = reorder(predictor, importancia),
      y    = importancia,
      fill = importancia)
) +
  labs(x = "predictor", title = "Importancia predictores (pureza de nodos
)") +
  geom_col() +
  coord_flip() +
  theme_bw() +
  theme(legend.position = "none")
```

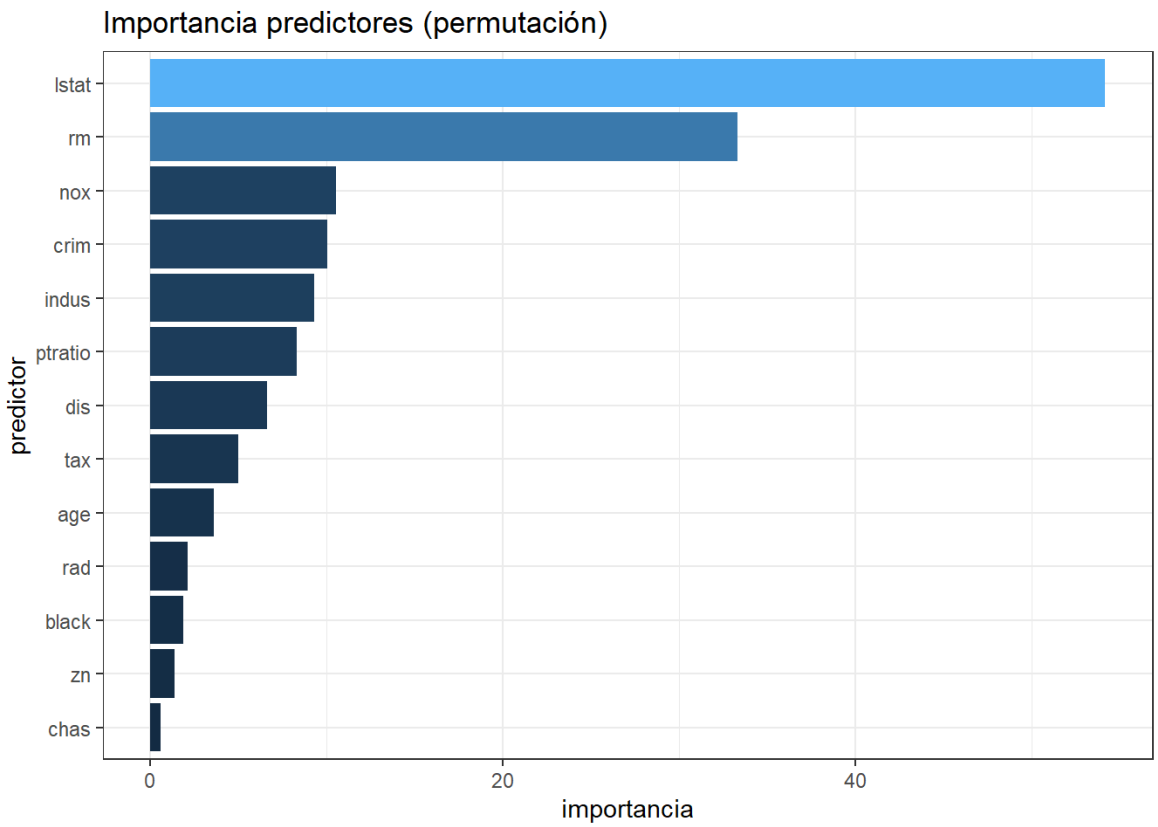


Importancia por permutación

Se entrena de nuevo el modelo, con los mejores hiperparámetros encontrados, pero esta vez indicando importance = "permutation".

```
##Importancia por permutación
# Entrenamiento modelo
modelo <- rand_forest(
  mode  = "regression"
) %>%
  set_engine(
    engine     = "ranger",
    importance = "permutation",
    seed       = 123
  )

modelo <- modelo %>% finalize_model(mejores_hiperpar)
modelo <- modelo %>% fit(medv ~., data = data_train)

# Importancia
```

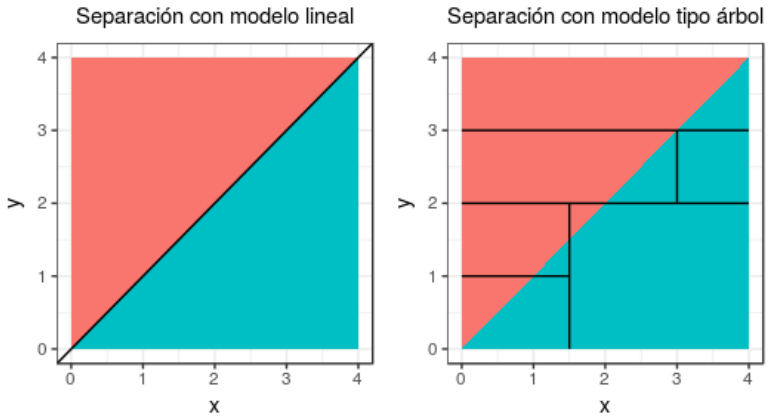
```r
importancia_pred <- modelo$fit$variable.importance %>%
  enframe(name = "predictor", value = "importancia")

# Gráfico
ggplot(
  data = importancia_pred,
  aes(x    = reorder(predictor, importancia),
      y    = importancia,
      fill = importancia)
) +
  labs(x = "predictor", title = "Importancia predictores (permutación)")
+
  geom_col() +
  coord_flip() +
  theme_bw() +
  theme(legend.position = "none")
```



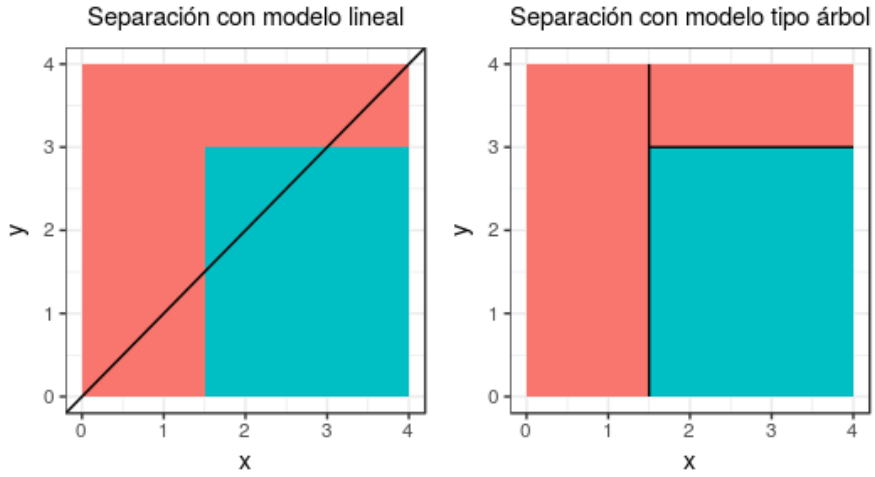
Ambas estrategias identifican lstats y rm como los predictores más influyentes, acorde a los datos de entrenamiento.

9. 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.



Al contrario, 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 lineales 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 el test error.

Ejemplo Regresión + Poda (usando método ANVA)

Para crear el árbol de decisión vamos a usar la misma función rpart como en los ejemplos anteriores, sin embargo se cambia la definición de method = "anova".

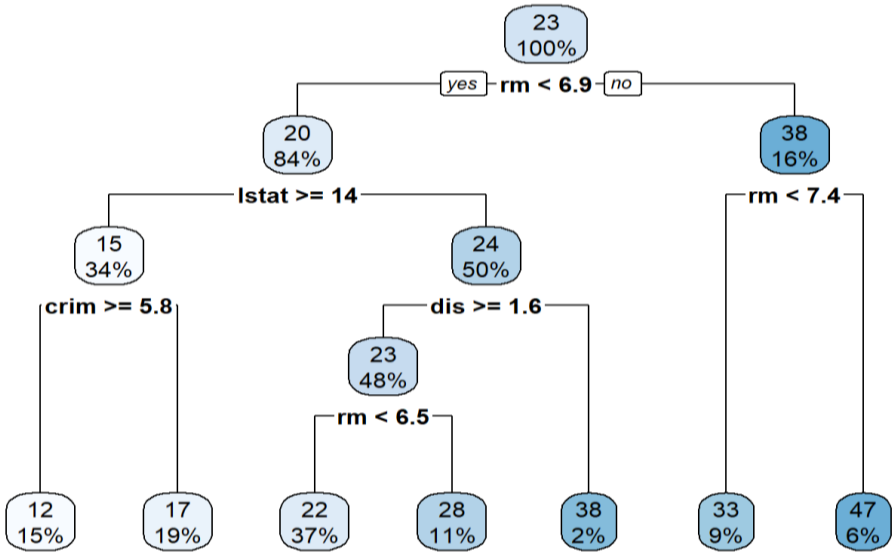
deviance = SSE (Sum Square Error) en el nodo

```
require(devtools)
library(dplyr)       # data wrangling
library(rpart)       # performing regression trees
library(rpart.plot)  # plotting regression trees
library(MLmetrics)
```

```
m1 <- rpart(medv ~ ., data = Boston, subset = train,
  method  = "anova")
m1
```

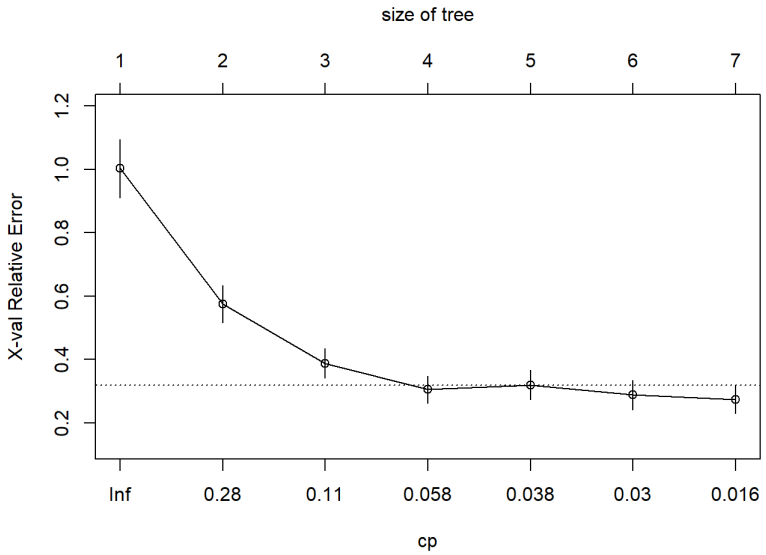
```
## n= 404
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 36363.3300 22.86040
##    2) rm< 6.945 341 14316.0900 20.02141
##      4) lstat>=14.405 139  2598.2430 14.94460
##        8) crim>=5.7819 61   667.3377 12.03443 *
##        9) crim< 5.7819 78  1010.2670 17.22051 *
##      5) lstat< 14.405 202  5670.0350 23.51485
##       10) dis>=1.5511 195  2719.5550 22.99487
##         20) rm< 6.543 150   978.7957 21.60533 *
##         21) rm>=6.543 45   485.7280 27.62667 *
##       11) dis< 1.5511 7  1429.0200 38.00000 *
##    3) rm>=6.945 63  4422.5040 38.22698
##      6) rm< 7.445 38  1106.4140 32.74474 *
##      7) rm>=7.445 25   438.0200 46.56000 *
```

```
rpart.plot(m1)
```



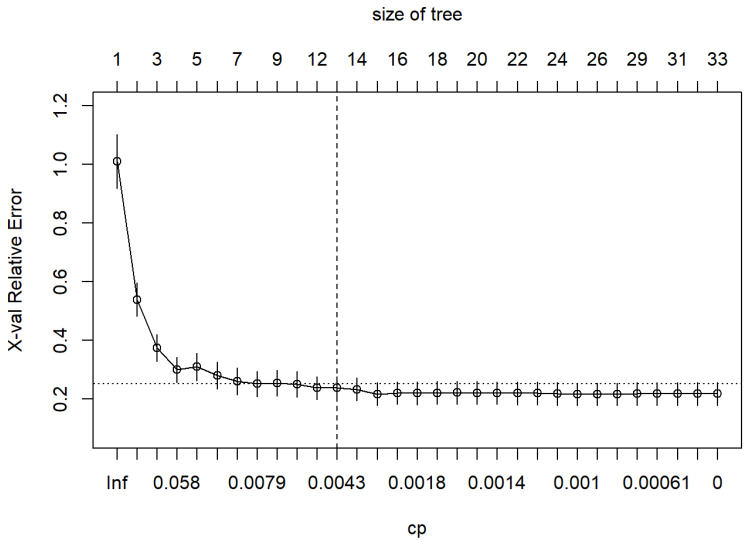
Visualicemos el árbol usando rpart.plot. Este árbol particiona usando 6 variables. Por defecto rpart aplica un proceso para calcular la penalización de acuerdo al número de árboles.

```
plotcp(m1)
```



extendiendo el árbol al máximo, forzando el argumento cp=0.

```
m2 <- rpart(
  formula = medv ~ ., data = Boston, subset = train,
  method  = "anova",
  control = list(cp = 0, xval = 10)
)
plotcp(m2)
abline(v = 12, lty = "dashed")
```




```
m1$cptable
```

```
##           CP nsplit rel error    xerror       xstd
## 1 0.48468417      0 1.0000000 1.0030379 0.09141489
## 2 0.16631633      1 0.5153158 0.5751815 0.05899148
## 3 0.07914760      2 0.3489995 0.3878537 0.04631578
## 4 0.04184052      3 0.2698519 0.3048800 0.04288827
## 5 0.03451365      4 0.2280114 0.3193135 0.04670918
## 6 0.02531778      5 0.1934977 0.2876669 0.04566300
## 7 0.01000000      6 0.1681800 0.2741174 0.04532067
```

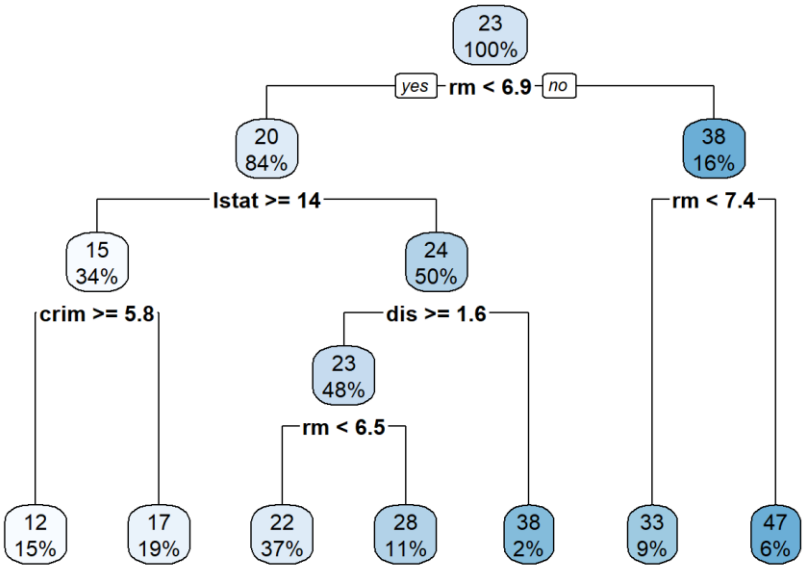
Entonces rpart aplica automáticamente el proceso de optimización para identificar un subárbol óptimo. Para este caso 6 divisiones, 7 nodos con un cp =0.01. Sin embargo, es posible mejorar el modelo a través de un tuning de los parámetros de restricción (e.g., profundidad, número mínimo de observaciones, etc)

```
# optimal model
optimal_tree <- rpart(medv ~ ., data = Boston, subset = train,
  method  = "anova",
  control = list(minsplit = 6, maxdepth = 7, cp = 0.01)
)

pred <- predict(optimal_tree, newdata = Boston, subset = train)
```

```
rmse_gof = (MSE(y_pred = pred, y_true = Boston$medv))^(1/2)
rmse_gof
```

4.24352

```
rpart.plot(optimal_tree)
```




```
optimal_tree
```

```
## n= 404
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 36363.3300 22.86040
##    2) rm< 6.945 341 14316.0900 20.02141
##      4) lstat>=14.405 139  2598.2430 14.94460
##        8) crim>=5.7819 61   667.3377 12.03443 *
##        9) crim< 5.7819 78  1010.2670 17.22051 *
##      5) lstat< 14.405 202  5670.0350 23.51485
##       10) dis>=1.5511 195  2719.5550 22.99487
##         20) rm< 6.543 150   978.7957 21.60533 *
```

```
##          21) rm>=6.543 45   485.7280 27.62667 *
##       11) dis< 1.5511 7  1429.0200 38.00000 *
##    3) rm>=6.945 63  4422.5040 38.22698
##      6) rm< 7.445 38  1106.4140 32.74474 *
##      7) rm>=7.445 25   438.0200 46.56000 *
```

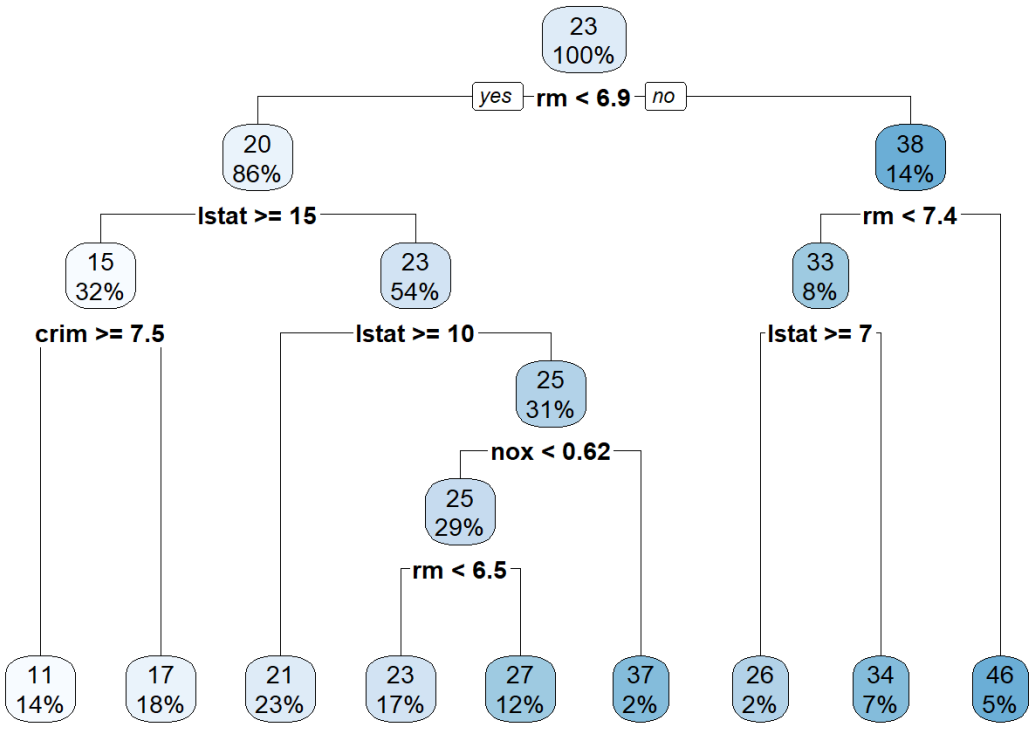
```
################################################
# con el metodo  (rpart)
require(devtools)
library(dplyr)       # data wrangling
library(rpart)       # performing regression trees
library(rpart.plot)  # plotting regression trees
library(MLmetrics)
```

```
m1 <- rpart(medv ~ ., data = Boston, subset = train,
  method  = "anova")
m1
```

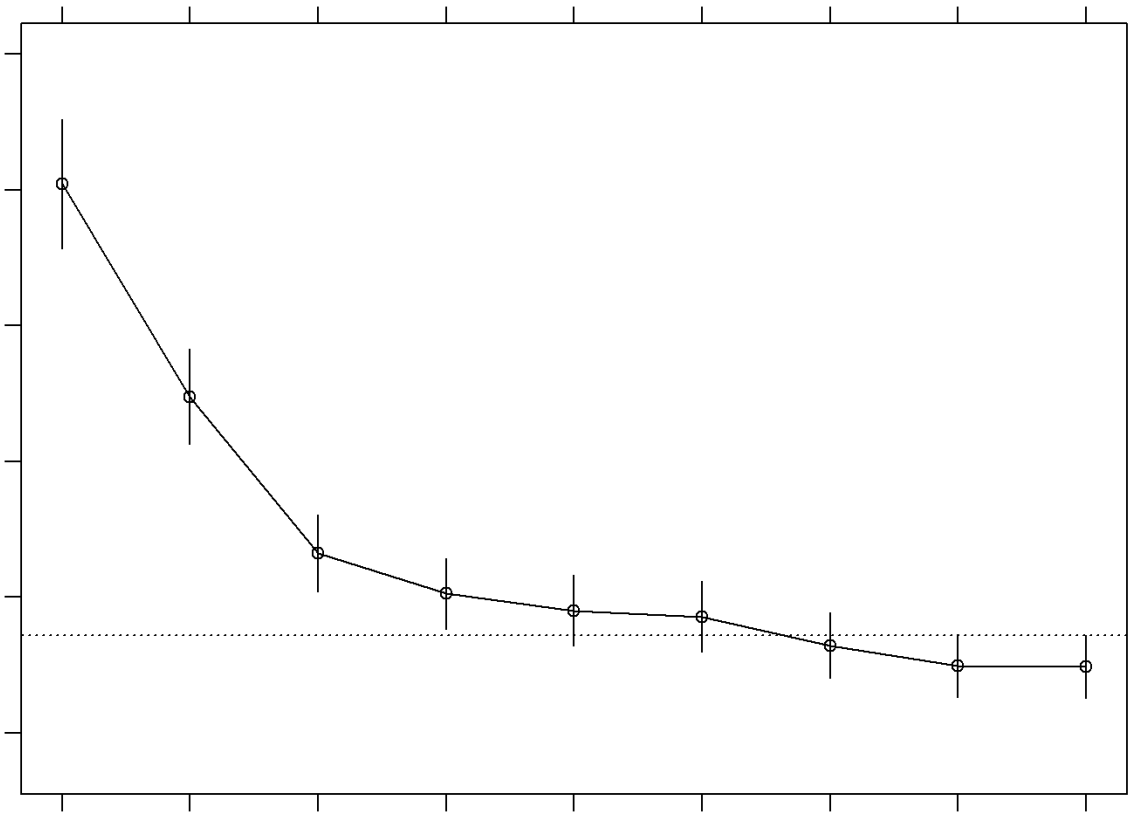
```
## n= 404
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 34108.2700 22.51089
##    2) rm< 6.941 348 14114.2700 20.02356
##      4) lstat>=14.78 130  2384.2310 14.52615
##        8) crim>=7.46495 56   589.9193 11.39643 *
##        9) crim< 7.46495 74   830.6778 16.89459 *
##      5) lstat< 14.78 218  5458.3790 23.30183
##       10) lstat>=9.95 94   611.7698 20.64255 *
##       11) lstat< 9.95 124  3677.9410 25.31774
##         22) nox< 0.618 117  1765.5900 24.64530
```

```
##           44) rm< 6.4815 67   613.7000 22.60000 *
##           45) rm>=6.4815 50   496.0402 27.38600 *
##         23) nox>=0.618 7   975.1771 36.55714 *
##    3) rm>=6.941 56  4461.6020 37.96786
##      6) rm< 7.437 34  1529.3590 32.59412
##       12) lstat>=6.97 7   577.0171 25.95714 *
##       13) lstat< 6.97 27   564.0541 34.31481 *
##      7) rm>=7.437 22   433.0636 46.27273 *
```

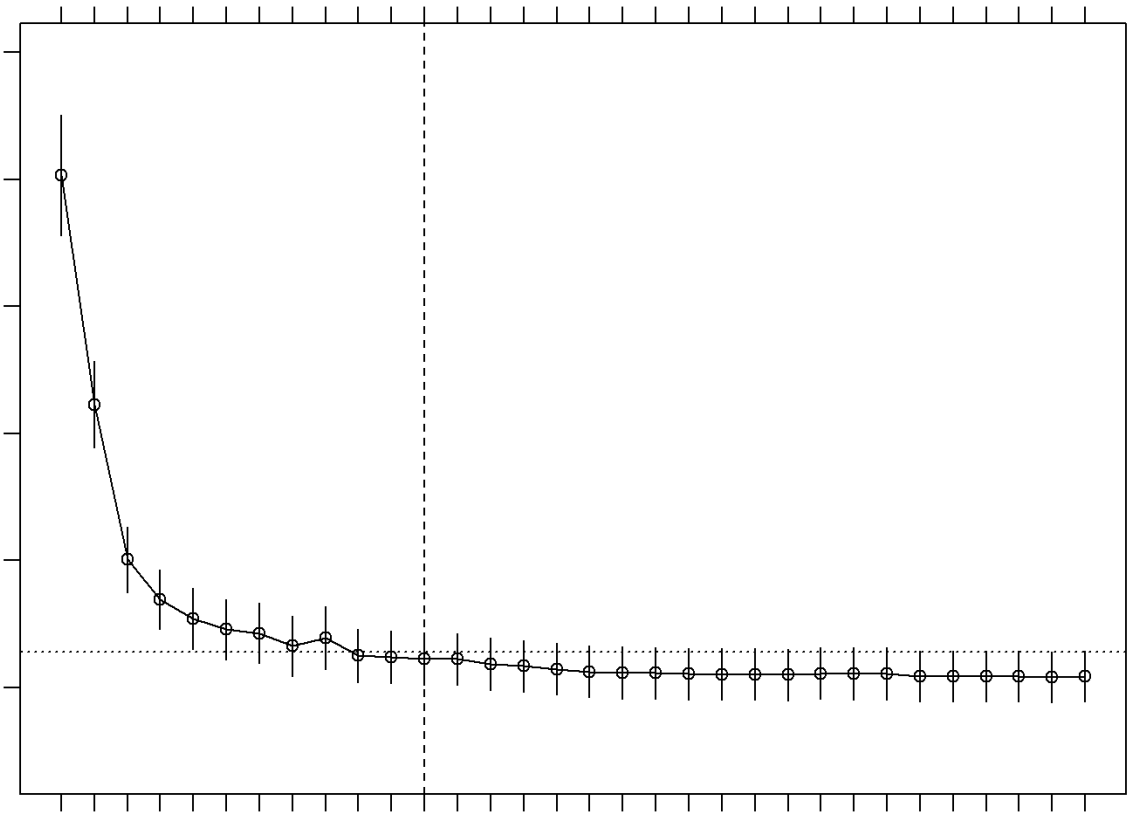
```
rpart.plot(m1)
```




```
plotcp(m1)
```



```
m2 <- rpart(
  formula = medv ~ ., data = Boston, subset = train,
  method  = "anova",
  control = list(cp = 0, xval = 10)
)

plotcp(m2)
abline(v = 12, lty = "dashed")
```



```
m1$cptable
```

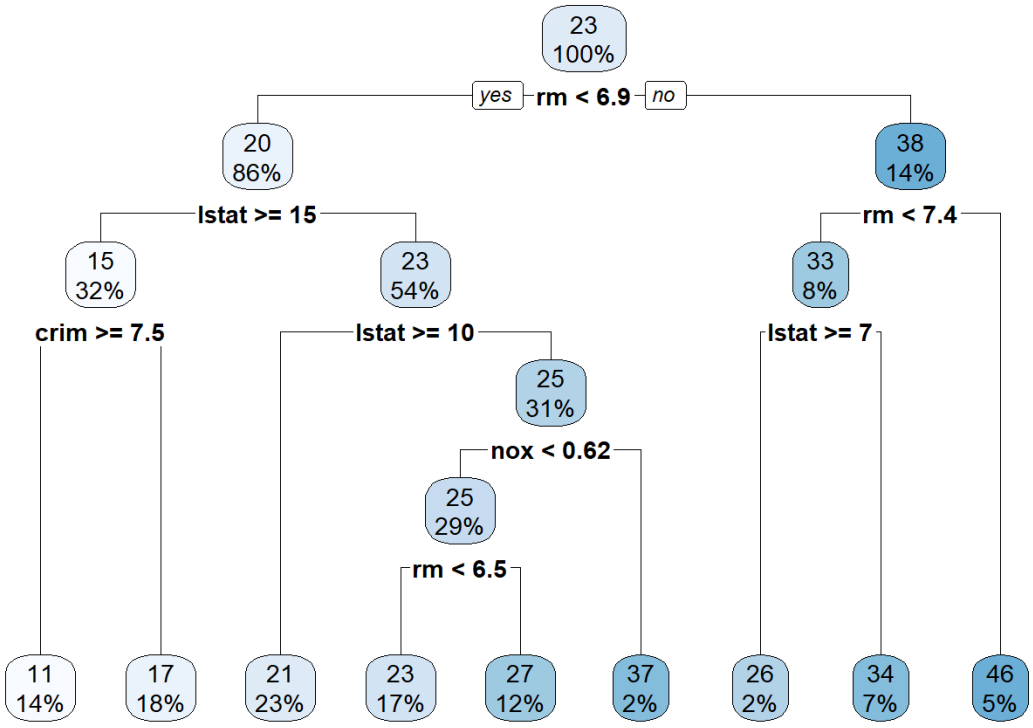
```
##           CP nsplit rel error    xerror       xstd
## 1 0.45538522      0 1.0000000 1.0080260 0.09489404
## 2 0.18387494      1 0.5446148 0.6953490 0.06970349
## 3 0.07327195      2 0.3607398 0.4649681 0.05610785
## 4 0.03426349      3 0.2874679 0.4052210 0.05252032
## 5 0.02825221      4 0.2532044 0.3804136 0.05204680
## 6 0.02747644      5 0.2249522 0.3716279 0.05206332
## 7 0.01922846      6 0.1974757 0.3292507 0.04869726
## 8 0.01138397      7 0.1782473 0.2990305 0.04572224
## 9 0.01000000      8 0.1668633 0.2980001 0.04585310
```

```
# optimal model
optimal_tree <- rpart(medv ~ ., data = Boston, subset = train,
  method  = "anova",
  control = list(minsplit = 6, maxdepth = 7, cp = 0.01)
```

```
)

pred <- predict(optimal_tree, newdata = Boston, subset = train)
rmse_gof = (MSE(y_pred = pred, y_true = Boston$medv))^(1/2)
rmse_gof
```

```
## [1] 4.023519
```

```
rpart.plot(optimal_tree)
```




```
optimal_tree
```

```
## n= 404
##
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 404 34108.2700 22.51089
##    2) rm< 6.941 348 14114.2700 20.02356
##      4) lstat>=14.78 130  2384.2310 14.52615
##        8) crim>=7.46495 56   589.9193 11.39643 *
##        9) crim< 7.46495 74   830.6778 16.89459 *
##      5) lstat< 14.78 218  5458.3790 23.30183
##       10) lstat>=9.95 94   611.7698 20.64255 *
##       11) lstat< 9.95 124  3677.9410 25.31774
##         22) nox< 0.618 117  1765.5900 24.64530
##           44) rm< 6.4815 67   613.7000 22.60000 *
##           45) rm>=6.4815 50   496.0402 27.38600 *
##         23) nox>=0.618 7   975.1771 36.55714 *
##    3) rm>=6.941 56  4461.6020 37.96786
##      6) rm< 7.437 34  1529.3590 32.59412
##       12) lstat>=6.97 7   577.0171 25.95714 *
##       13) lstat< 6.97 27   564.0541 34.31481 *
##      7) rm>=7.437 22   433.0636 46.27273 *
```