Árboles de predicción: bagging, random forest, boosting y C5.0

Joaquín Amat Rodrigo j.amatrodrigo@gmail.com

Febrero, 2017

Tabla de contenidos

Formato PDF: Github

Introducción

Los métodos predictivos como la regresión lineal o polinómica generan modelos globales en los que una única ecuación se aplica a todo el espacio muestral. Cuando el estudio implica múltiples predictores, que interaccionan entre ellos de forma compleja y no lineal, es muy difícil encontrar un modelo global que sea capaz de reflejar la relación entre las 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. A lo largo de este capítulo se explora la forma en que se construyen los árboles así como métodos más complejos que combinan multitud de ellos.

Ventajas y desventajas

Ventajas

- Los árboles son fáciles de interpretar aun cuando las relaciones entre predictores son complejas. Su estructura se asemejan 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.

Árboles de predicción en R

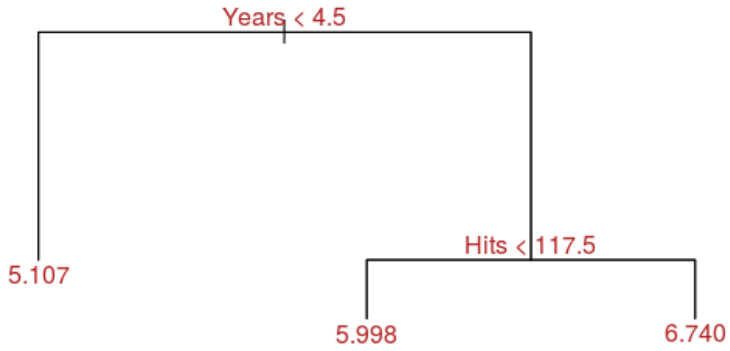
Existen diferentes paquetes en `R` que permiten la creación de modelos basados en árboles. A lo largo de este tutorial se emplean los siguientes:

- `tree` y `rpart`: contiene funciones para la creación y representación de árboles de regresión y clasificación.
- `rpart.plot`: permite crear representaciones detalladas de modelos creados con `rpart`.
- `randomForest`: implementa algoritmos para crear modelos *random forest*.
- `gbm`: implementa algoritmos de *boosting*.
- `C50`: contiene los algoritmos *C5.0* para árboles de clasificación.

Árboles de regresión

Idea intuitiva

Los árboles de regresión son el subtipo de árboles de predicción que trabaja con variables respuesta continuas. 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. En la rama izquierda quedan aquellos con menos de 4.5 años, y su salario predicho se corresponde con el salario promedio de todos los jugadores de este grupo ($\epsilon^{5.107}$). Los jugadores con 4.5 o más años se asignan a la rama derecha, que a su vez, se subdivide en función de si superan o no 117.5 bateos. Como resultado de las estratificaciones se han generado 3 regiones que pueden identificarse con la siguiente nomenclatura:

- $R_1 = \{X|Year < 4.5\}$: jugadores que han jugado menos de 4.5 años.

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

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

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

Viendo el árbol de la imagen 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 jugados, los jugadores con más experiencia ganan más. Entre los jugadores con pocos años de experiencia, el número de bateos logrados en los años previos no tiene mucho impacto en el salario, sin embargo, sí lo tiene para jugadores con cuatro años y medio o más de experiencia. Para estos últimos, a mayor número de bateos logrados mayor salario. 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.

Construcción del árbol

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

- División sucesiva del espacio de los predictores generando regiones no solapantes (nodos terminales) R_1, R_2, R_3, ..., R_j. 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 R_1, R_2, R_3, ..., R_j, o lo que es equivalente, decidir donde se introducen las divisiones: en que predictores y en que valores de los mismos. Es en este punto donde se diferencian lo algoritmos de árboles de regresión y clasificación.

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 (R_1,..., R_j) que minimizan el *Residual Sum of Squares (RSS)* total:

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

donde $\hat{y}_{R_j}$ es la media de la variable respuesta en la región R_j. Una descripción menos técnica equivale a decir que se busca una distribución de regiones tal que, el sumatorio de las desviaciones al cuadrado entre las observaciones y la media de la región a la que pertenecen sea lo menor posible. Desafortunadamente, no es posible considerar todas las posibles particiones del espacio de los predictores. Por esta razón, se recurre a lo que se conoce como *recursive binary splitting* (división binaria recursiva). Esta solución sigue la misma idea que la selección de predictores *stepwise (backward o fordward)* en regresión lineal múltiple, no evalúa todas las posibles regiones pero, alcanza un buen balance computación-resultado.

Recursive binary splitting

El objetivo del método *recursive binary splitting* es encontrar en cada iteración el predictor X_j y el punto de corte (umbral) s tal que, si se distribuyen las observaciones en las regiones $\{X|X_j < s\}$ y $\{X|X_j \geq s\}$, se consigue la mayor reducción posible en el *RSS*. 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 (X_1, X_2,..., X_p). 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.

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

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

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

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

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

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

Y en la región 2:

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

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

Nota: Los algoritmos que implementan recursive binary splitting suelen incorporar estrategias para evitar evaluar todos los posibles puntos de corte.

Predicción

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...).

Árboles de clasificación

Idea intuitiva

Los árboles de clasificación se asemejan mucho a los árboles de regresión, con la diferencia de que predicen variables respuesta cualitativas en lugar de continuas.

Construcción del árbol

Para construir un árbol de clasificación se emplea el mismo método *recursive binary splitting* descrito en los árboles de regresión. Sin embargo, como la variable respuesta en cualitativa, no es posible emplear el *RSS* como criterio de selección de las divisiones óptimas. Existen varias alternativas, todas ellas con el objetivo de encontrar nodos lo más puros/homogéneos posible. Las más empleadas son:

Classification error rate

Se define como la proporción de observaciones que no pertenecen a la clase más común en el nodo.

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

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

Gini index

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

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

Cuando $\hat{p}_{mk}$ es cercano a 0 o a 1 (el nodo contiene mayoritariamente observaciones de una clase), el término $\hat{p}_{mk}(1-\hat{p}_{mk})$ es muy pequeño. Como consecuencia, cuanto mayor sea la pureza del nodo, menor el valor del índice Gini G. El algoritmo *CART* (*Classification and regression trees*) emplea Gini como criterio de división.

Information gain: cross entropy

La entropía es otra forma de cuantificar el desorden de un sistema. En el caso de los nodos, el desorden se corresponde con la impureza. Si un nodo es puro, contiene únicamente observaciones de una clase, su entropía es cero. Por el contrario, si la frecuencia de cada clase es la misma, el valor de la entropía alcanza el valor máximo de 1.

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

Los algoritmos C4.5 y C5.0 emplean *information gain* como criterio de división.

Chi-square

Esta aproximación consiste en identificar si existe una diferencia significativa entre los nodos hijos y el nodo parental, es decir, si hay evidencias de que la división consigue una mejora. Para ello, se aplica un test estadístico chi-square goodness of fit empleando como distribución esperada H_0 la frecuencia de cada clase en el nodo parental. Cuanto mayor el estadístico χ^2, mayor la evidencia estadística de que existe una diferencia.

$$\chi^2 = \sum_{k} \frac{(\text{observado}_k - \text{esperado}_k)^2}{\text{esperado}_k}$$

Los árboles generados con este criterio de división reciben el nombre de CHAID (*Chi-square Automatic Interaction Detector*).

Independientemente de la medida empleada como criterio de selección de las divisiones, el proceso siempre es el mismo:

1. Para cada posible división se calcula el valor de la medida en cada uno de los dos nodos resultantes.

2. Se suman los dos valores ponderando cada uno por la fracción de observaciones que contiene cada nodo. Este paso es muy importante, ya que no es lo mismo dos nodos puros con 2 observaciones, que dos nodos puros con 100 observaciones.

$$\frac{\text{n observaciones nodo A}}{\text{n observaciones totales}} \times \text{pureza A} + \frac{\text{n observaciones nodo B}}{\text{n observaciones totales}} \times \text{pureza B}$$

3. La división con menor o mayor valor (dependiendo de la medida empleada) se selecciona como división óptima.

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

Predicción

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 del valor de sus predictores hasta llegar a uno de los nodos terminales. En el caso de clasificación, suele emplearse la moda de la variable respuesta como valor de predicción, es decir, la clase más frecuente del nodo. Además, puede acompañarse con el porcentaje de cada clase en el nodo terminal, lo que aporta información sobre la confianza de la predicción.

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 y el proceso de podado (*pruning*).

Controlar el tamaño del árbol

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.

Todos estos parámetros son lo que se 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 hasta alcanzar una condición de parada. 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:

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

donde $|T|$ es el número de nodos terminales del árbol.

El primer término de la ecuación se corresponde con el sumatorio total de los residuos cuadrados *RSS*. Por definición, cuantos más nodos terminales tenga el modelo menor será esta parte de la ecuación. El segundo término es la restricción, que penaliza al modelo en función del número de nodos terminales (a mayor número, mayor penalización). El grado de penalización se determina mediante el *tunning parameter* α. Cuando $\alpha = 0$, la penalización es nula y el árbol resultante es equivalente al árbol original. A medida que se incrementa α la penalización es mayor y, como consecuencia, los árboles resultantes son de menor tamaño. El valor óptimo de α puede identificarse mediante *cross validation*.

Algoritmo para crear un árbol de regresión con pruning

1. Se emplea *recursive binary splitting* para crear un árbol grande y complejo (T_o) 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 T_o para obtener el mejor *sub-árbol* en función de α. Es decir, se obtiene el mejor *sub-árbol* para un rango de valores de α.

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

Ejemplo regresión

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

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

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

Ajuste del modelo

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

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

- `mincut = 5`: número mínimo de observaciones que debe de tener al menos uno de los nodos hijos para que se produzca la división.

- `minsize = 10`: número mínimo de observaciones que debe de tener un nodo para que pueda dividirse.
- `deepth = 31`: profundidad máxima que puede alcanzar el árbol.

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

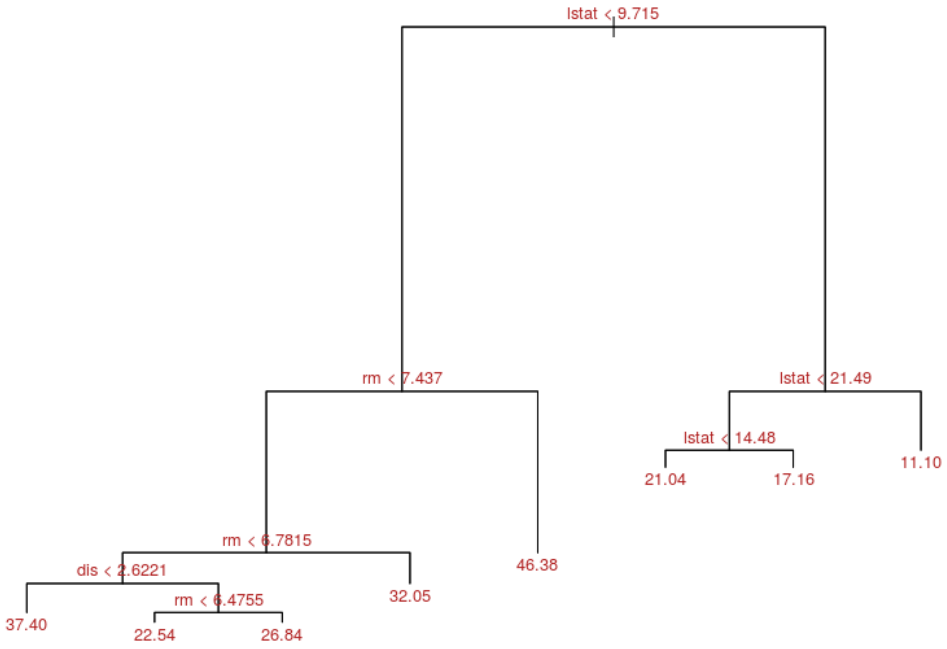
Como en cualquier estudio de regresión, no solo es importante ajustar el modelo, sino también cuantificar su capacidad para predecir nuevas observaciones. Para poder hacer la posterior evaluación, se dividen los datos en dos grupos, uno se emplea como *training data set* y el otro como *test data set*.

```
library(tree)
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)

# Ajuste del árbol
arbol_regresion <- tree(formula = medv ~ ., data = Boston, subset = train,
                        split = "deviance")
```

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

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



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

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

```
arbol_regresion
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 253 20890.0 22.67
##    2) lstat < 9.715 103  7765.0 30.13
##      4) rm < 7.437 89  3310.0 27.58
##        8) rm < 6.7815 61  1995.0 25.52
##         16) dis < 2.6221 5   615.8 37.40 *
```

```
##         17) dis > 2.6221 56   610.3 24.46
##           34) rm < 6.4755 31   136.4 22.54 *
##           35) rm > 6.4755 25   218.3 26.84 *
##        9) rm > 6.7815 28   496.6 32.05 *
##      5) rm > 7.437 14   177.8 46.38 *
##    3) lstat > 9.715 150  3465.0 17.55
##      6) lstat < 21.49 120  1594.0 19.16
##       12) lstat < 14.48 62   398.5 21.04 *
##       13) lstat > 14.48 58   743.3 17.16 *
##      7) lstat > 21.49 30   311.9 11.10 *
```

Podado del árbol (pruning)

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

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

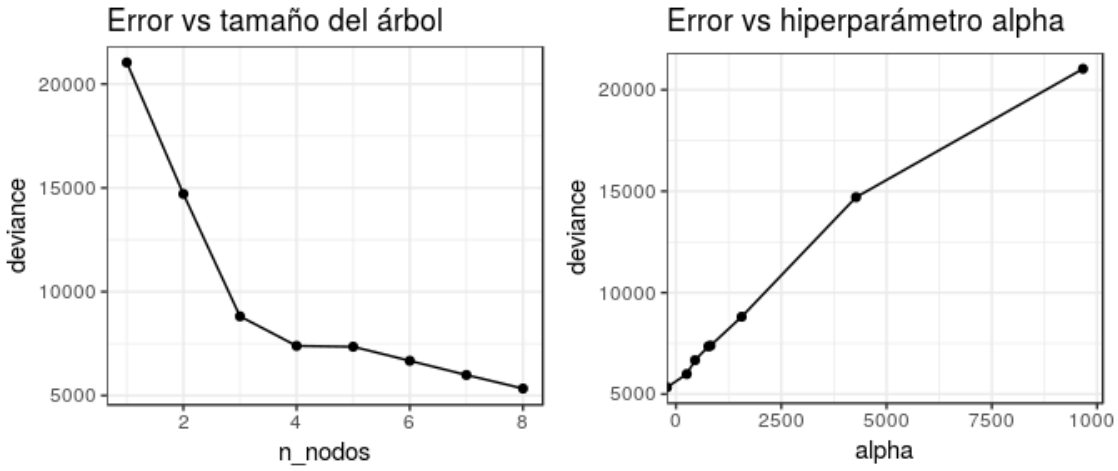
```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  5335.721  5988.901  6671.685  7349.550  7388.893  8812.276 14707.294
## [8] 21035.642
##
## $k
## [1]      -Inf  255.6581  451.9272  768.5087  818.8885 1559.1264 4276.5803
## [8] 9665.3582
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1]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

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

- `size`: el tamaño (número de nodos terminales) de cada árbol.
- `dev`: la estimación de *cross-validation test error* para cada tamaño de árbol.
- `k`: El rango de valores de penalización α evaluados.
- `method`: El criterio empleado para seleccionar el mejor árbol.

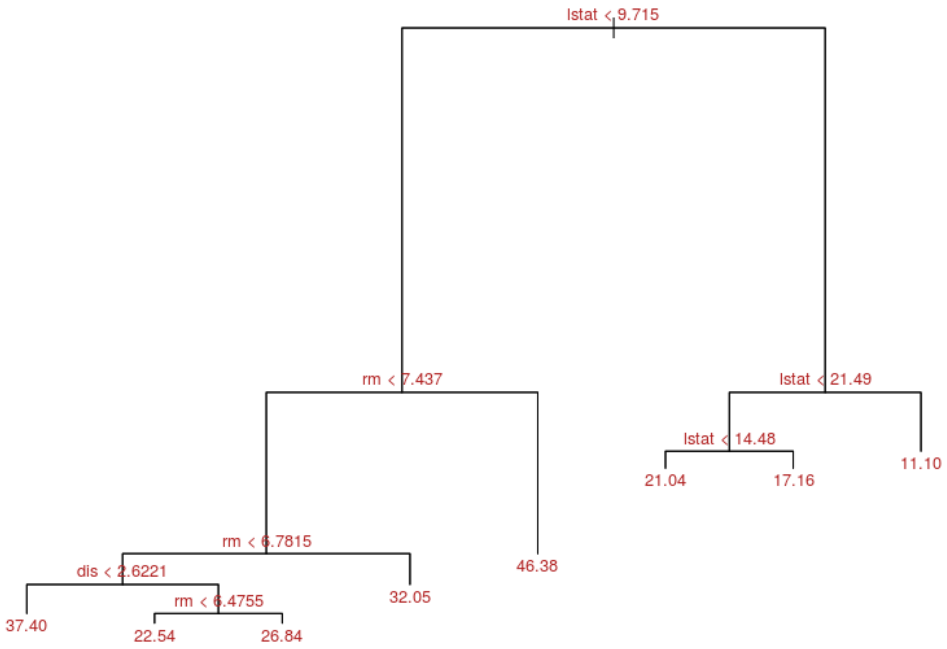
El objetivo del proceso es encontrar el valor α con el que se consigue el menor *cross-validation test error*. Dado que α es quien determina el tamaño del árbol, la frase anterior equivale a decir que se busca el tamaño del árbol que minimiza el *cross-validation test error*. En este caso, el mejor árbol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

```
library(ggplot2)
library(ggpubr)

resultados_cv <- data.frame(n_nodos = cv_arbol$size, deviance = cv_arbol$dev,
                            alpha = cv_arbol$k)
p1 <- ggplot(data = resultados_cv, aes(x = n_nodos, y = deviance)) +
      geom_line() +
      geom_point() +
      labs(title = "Error vs tamaño del árbol") + theme_bw()


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

ggarrange(p1, p2)
```



Con la función `prune.tree()` se obtiene el mejor árbol de regresión del tamaño identificado como óptimo. Esta función también acepta el valor de α óptimo en lugar del tamaño.

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



Evaluación del modelo

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

```
predicciones <- predict(arbol_pruning, newdata = Boston[-train,])
test_mse     <- mean((predicciones - Boston[-train, "medv"])^2)
paste("Error de test (mse) del árbol de regresión tras podado:", round(test_mse,2))
```

```
## [1] "Error de test (mse) del árbol de regresión tras podado: 25.05"
```

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

Ejemplo clasificación

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

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

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

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

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

Ajuste del modelo

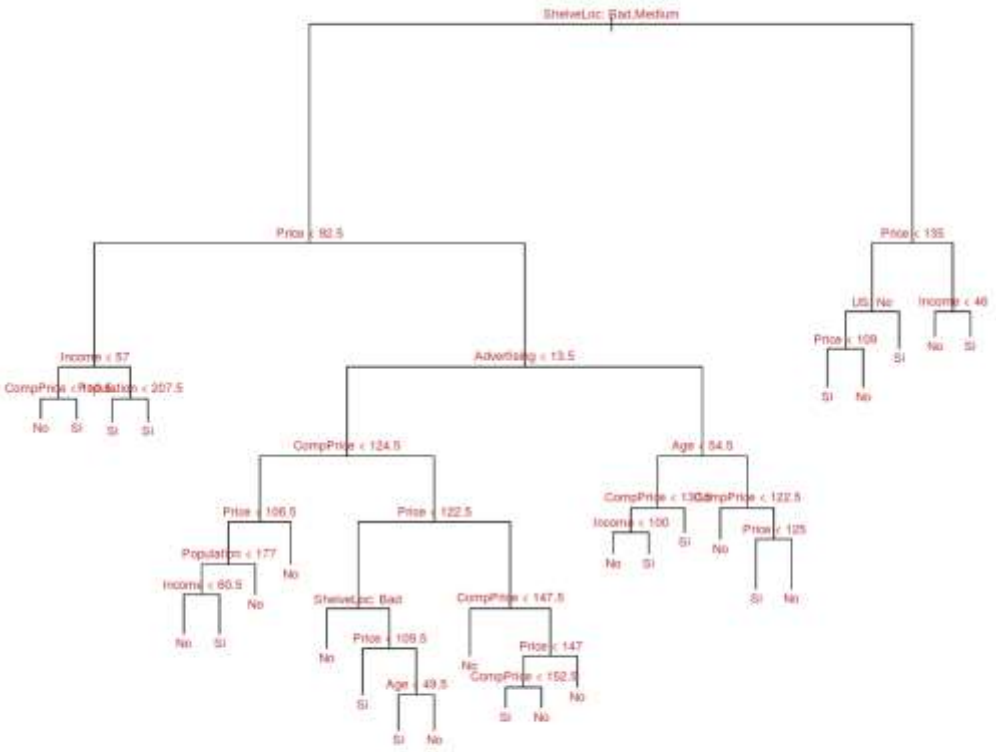
Se ajusta un árbol de clasificación empleando como variable respuesta *ventas_altas* y como predictores todas las variables disponibles excepto Sales.

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

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

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

La función `summary()` muestra que el árbol ajustado tiene un total de 27 nodos terminales y un *classification error rate* del 9%. La *deviance* de los árboles de clasificación se calcula como $-2\sum_m\sum_k n_{mk}\, log(\hat{p}_{mk})$, donde n_{mk} es el número de observaciones en el nodo terminal m que pertenecen a la clase k. El término *Residual mean deviance* mostrado en el `summary` es simplemente la *deviance* dividida entre (número de observaciones - número de nodos terminales). Cuanto menor es la *deviance* mejor es el ajuste del árbol a las observaciones de entrenamiento.

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



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

```
arbol_clasificacion
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Si ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Si ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Si ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Si ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Si ( 0.05000 0.95000 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
```

```
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Si ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Si ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Si ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Si ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Si ( 0.30769 0.69231 ) *
##                 343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##               174) Price < 147 12  16.300 Si ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Si ( 0.14286 0.85714 ) *
##                 349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##               175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##        11) Advertising > 13.5 45  61.830 Si ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Si ( 0.20000 0.80000 )
##            44) CompPrice < 130.5 14  18.250 Si ( 0.35714 0.64286 )
##              88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##              89) Income > 100 5   0.000 Si ( 0.00000 1.00000 ) *
##            45) CompPrice > 130.5 11   0.000 Si ( 0.00000 1.00000 ) *
##          23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##            46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##            47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##              94) Price < 125 5   0.000 Si ( 0.00000 1.00000 ) *
##              95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##     3) ShelveLoc: Good 85  90.330 Si ( 0.22353 0.77647 )
##       6) Price < 135 68  49.260 Si ( 0.11765 0.88235 )
##        12) US: No 17  22.070 Si ( 0.35294 0.64706 )
##          24) Price < 109 8   0.000 Si ( 0.00000 1.00000 ) *
##          25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##        13) US: Yes 51  16.880 Si ( 0.03922 0.96078 ) *
##       7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##        14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##        15) Income > 46 11  15.160 Si ( 0.45455 0.54545 ) *
```

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

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

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

El *test error* es 28.5%, El modelo es capaz de predecir correctamente un 71.5 % de las observaciones del *test set*.

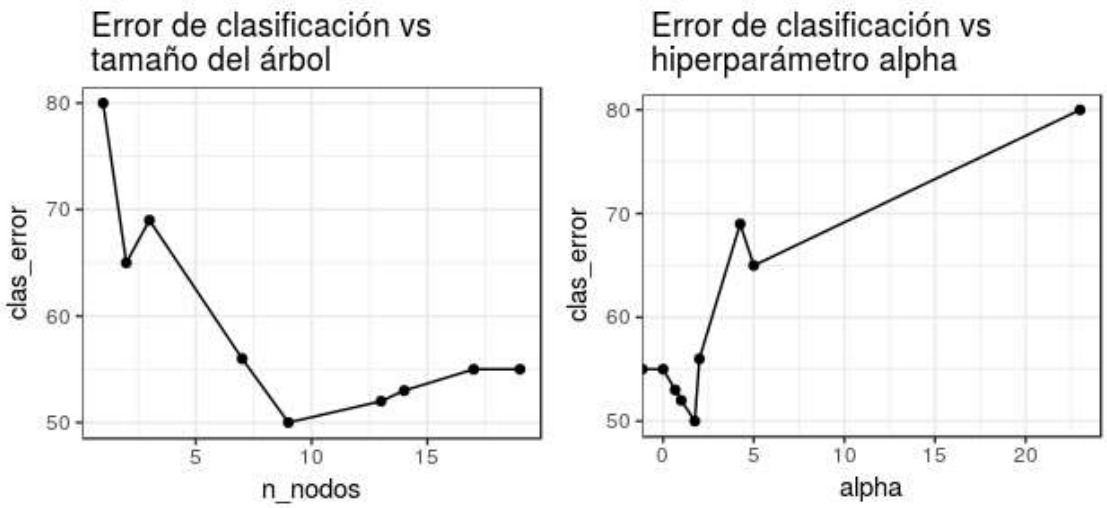
Podado del árbol (pruning)

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

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

```
## $size
## [1] 19 17 14 13  9  7  3  2  1
##
## $dev
## [1] 55 55 53 52 50 56 69 65 80
##
## $k
## [1]       -Inf  0.0000000  0.6666667  1.0000000  1.7500000  2.0000000
## [7]  4.2500000  5.0000000 23.0000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
library(ggplot2)
library(ggpubr)

resultados_cv <- data.frame(n_nodos = cv_arbol$size, clas_error = cv_arbol$dev,
                            alpha = cv_arbol$k)
p1 <- ggplot(data = resultados_cv, aes(x = n_nodos, y = clas_error)) +
      geom_line() +
      geom_point() +
      labs(title = " Error de clasificación vs \n tamaño del árbol") + theme_bw()

p2 <- ggplot(data = resultados_cv, aes(x = alpha, y = clas_error)) +
      geom_line() +
      geom_point() +
      labs(title = " Error de clasificación vs \n hiperparámetro alpha") +
      theme_bw()

ggarrange(p1, p2)
```




```
cv_arbol$size[which.min(cv_arbol$dev)]
```

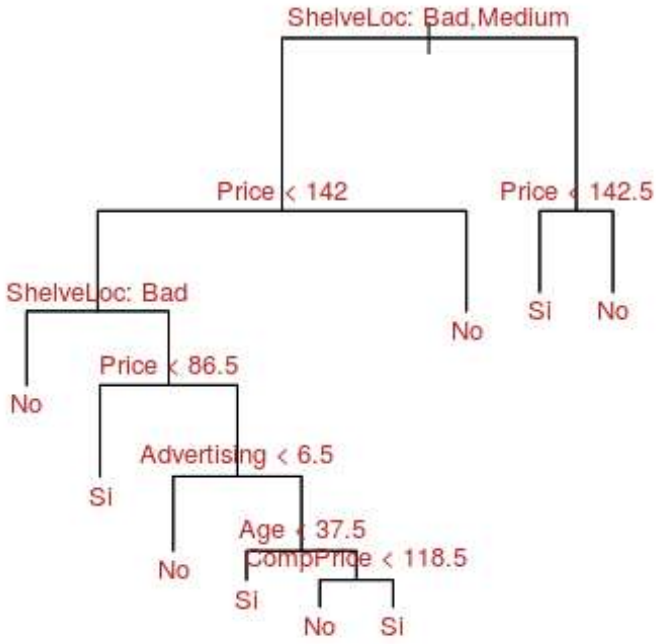
```
## [1] 9
```

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

Con la función `prune.missclas()` se obtiene el mejor árbol de clasificación del tamaño identificado como óptimo (no confundir con la función `prune.tree()` para árboles de

regresión). A esta función también se le puede indicar el valor de *alpha* óptimo en lugar del tamaño.

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



Predicción

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

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

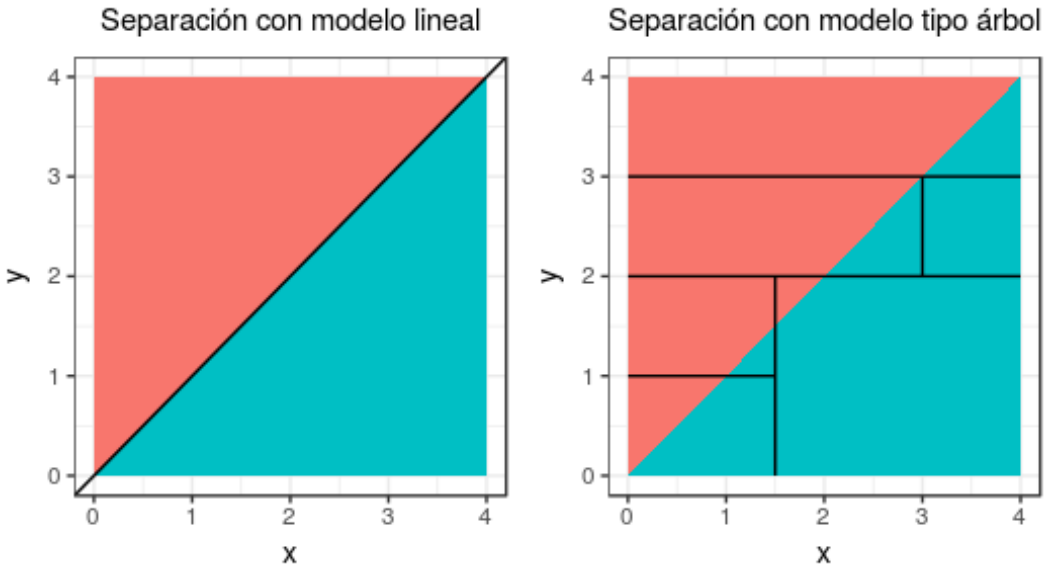
```
paste("El porcentaje de acierto es de",
      100 * ((94 + 60) / (94 + 24 + 22 + 60)), "%")
```

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

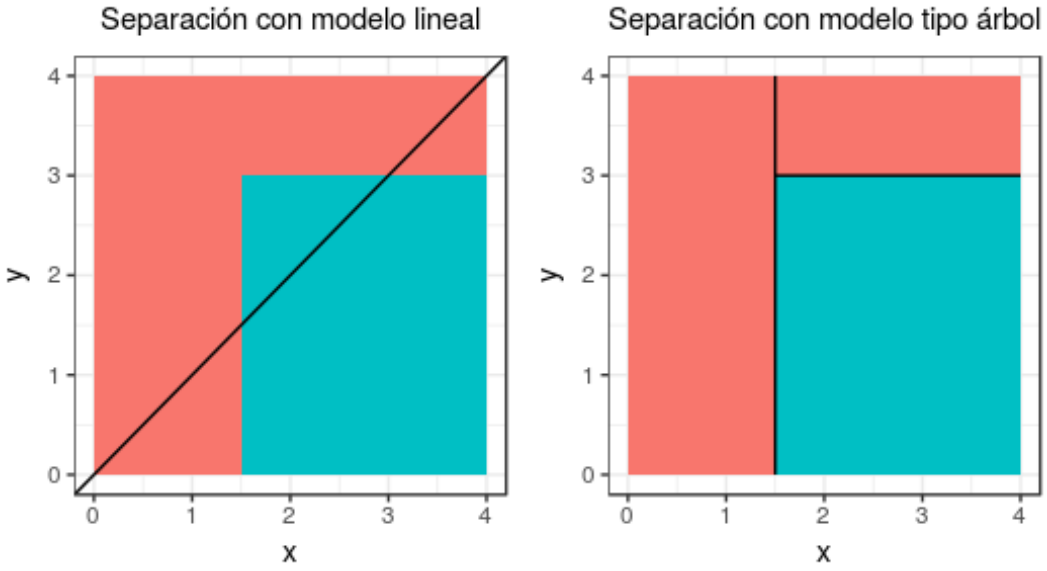
Gracias al proceso de *pruning* el porcentaje de acierto ha pasando de 71.5% a 77%.

Comparación de árboles frente a modelos lineales

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



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

Métodos de *ensemble*

Al igual que todo modelo estadístico, los árboles de predicción sufren el problema del equilibrio bias-varianza. El término bias hace referencia a cuánto se alejan en promedio las predicciones de un modelo respecto a los valores reales, es decir, cómo de bien se aproxima el modelo a la relación real entre las variables. La varianza hace referencia a cuánto varía el modelo dependiendo de la muestra empleada en el entrenamiento. A medida que se aumenta la complejidad de un modelo, se dispone de mayor flexibilidad para adaptarlo 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 equilibro óptimo entre bias y varianza.

¿Cómo se controlan el bias y varianza en los árboles de predicción? Por lo general, los árboles pequeños (pocas ramificaciones) no representarán bien la relación entre las variables, por lo que tienen alto bias, pero poca varianza. Los árboles grandes se ajustan mucho a los datos de entrenamiento, por lo que tienen muy poco bias pero mucha varianza. Los métodos de *ensemble* abarcan un conjunto de técnicas que combinan múltiples modelos predictivos para lograr un equilibro entre bias y varianza. Aunque pueden emplearse con multitud de métodos de aprendizaje estadístico (K-NN, redes neuronales...), con los árboles de predicción, dan muy buenos resultados. A continuación se describen 2 de los más utilizados:

- **Bagging**: En lugar de ajustar un único árbol, se ajustan muchos de ellos en paralelo formando un "bosque". En cada nueva predicción, todos los árboles que forman el "bosque" participan aportando su predicción. Como valor final, se toma la media de todas las predicciones (variables continuas) o la clase más frecuente (variables cualitativas). Uno de los métodos de *bagging* más conocidos es *Random Forest*.

- **Boosting**: Consiste en ajustar 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 diferencias importantes entre *bagging* y *boosting*:

- La estrategia seguida para reducir el error total: El error total de un modelo puede descomponerse como $bias + varianza + \epsilon$. En *bagging*, se emplean modelos con muy poco bias pero mucha varianza, agregando muchos de estos modelos se consigue reducir la varianza sin apenas inflar el bias. En *boosting*, se emplean modelos con muy poca varianza pero mucho bias, ajustando secuencialmente muchos modelos se reduce el bias.

- Forma en que se crean los distintos modelos que forman el *ensemble* final. En el caso de *bagging*, cada modelo es distinto de los otros porque se entrenan con diferentes muestras obtenidas por *bootstrapping* a partir de la muestra original. También se le conoce como *parallel ensemble* porque cada modelo se ajusta independientemente de los otros. En *boosting*, los modelos se ajustan secuencialmente y la importancia (peso) de las observaciones varía en cada iteración, dando lugar a diferentes ajustes. También se les conoce como *sequential ensemble*.

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. 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 sabe un poco de todos los temas. A continuación, se explican con detalle cada una de las estrategias.

Bagging

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

Dado n muestras de observaciones independientes Z_1, ..., Z_n, cada una con varianza σ^2, la varianza de la media de todas las observaciones $\overline{Z}$ es σ^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. Si bien 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, *bagging* ha demostrado incrementar en gran medida la precisión de las predicciones. La forma de aplicarlo es:

1. Generar B *pseudo-training sets* mediante *bootstrapping* a partir de la muestra de entrenamiento original.

2. Entrenar un árbol con cada una de las B 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* que, como se explica más adelante, puede interpretarse de la misma forma que el error de validación cruzada. También puede emplearse validación cruzada.

3. Para cada nueva observación, obtener la predicción de cada uno de los B árboles. El valor final de la predicción se obtiene como la media de las B predicciones en el caso de variables cuantitativas y como la clase predicha más frecuente (moda) para las 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.

Out-of-Bag Error

Dada la naturaleza del proceso de *bagging*, resulta posible estimar de forma directa el *test error* sin necesidad de recurrir a *cross-validation* o a un *test set* independiente. Sin entrar en demostraciones matemáticas, el hecho de que los árboles se ajusten de forma repetida empleando muestras generadas por *bootstrapping* conlleva que, en promedio, cada ajuste usa solo aproximadamente dos tercios de las observaciones originales. Al tercio restante se le llama *out-of-bag* (*OOB*). Si para cada árbol ajustado en el proceso de *bagging* se registran las observaciones empleadas, se puede predecir la respuesta de la observación i haciendo uso de aquellos árboles en los que esa observación ha sido excluida (*OOB*) y promediándolos (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 *test-error*. De hecho, si el número de árboles es suficientemente alto, el *OOB-error* es prácticamente equivalente al *leave-one-out cross-validation error*. Esta es una ventaja añadida de los métodos de *bagging*, ya que evita tener que recurrir al proceso de *cross-validation* (computacionalmente costoso) para la optimización de los hiperparámetros.

Importancia de los predictores

Si bien es cierto que el proceso de *bagging* consigue mejorar la capacidad predictiva en comparación a los modelos basados en un único árbol, esto tiene un coste asociado, la interpretabilidad del modelo se reduce. 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 que predictores son más importantes. Sin embargo, el tener múltiples árboles, abre la posibilidad de nuevas estrategias para cuantificar la importancia de los predictores que hacen de los modelos de *bagging* una herramienta muy potente, no solo para predecir, sino también para el análisis exploratorio. Dos de estas medidas son: el incremento del *MSE* y el incremento de la pureza de nodos.

Incremento del MSE: Identifica la influencia que tiene cada predictor en el *Mean Squared Error* del modelo (estimado por *out-of-bag error*). El valor asociado con cada predictor se obtiene de la siguiente forma:

1. Crear el conjunto de árboles que forman el modelo de *bagging*.
2. Calcular el *out-of-bag MSE* del modelo. Este es el error de referencia (mse_0).
3. Para cada predictor j:
    - Permutar en todos los árboles del modelo los valores del predictor j manteniendo el resto constante.
    - Recalcular el *out-of-bag MSE* tras la permutación, llámese (mse_j).
    - Calcular el incremento en el *MSE* debido a la permutación del predictor j.

$$\%\text{IncMSE}_j = (mse_j - mse_0)/mse_0 * 100$$

Si el predictor permutado estaba contribuyendo al modelo, es de esperar que el modelo aumente su *MSE*, ya que se pierde la información que proporcionaba esa variable. El porcentaje en que se incrementa el *MSE* 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 $(mse_j - mse_0)$ es negativo. A modo general, se puede considerar que estas variables tiene una importancia próxima a cero. En el caso de los árboles de clasificación, en lugar de *MSE* se emplea el *Mean Classification Error*.

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, entropía o MSE). 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.

Aunque este análisis es muy útil, 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.

Ejemplo regresión

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

La función `randomForest()` del paquete `randomForest` permite ajustar modelos *ensemble* basados en árboles mediante el proceso de *bagging* y *random forest*. La misma función puede emplearse para ambos procesos, ya que, como se describe más adelante, *random forest* es un caso particular de *bagging*. Para obtener un modelo básico de *bagging* se indica que el número de predictores empleados por los árboles es igual al número de predictores disponibles (`mtry = p`).

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

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

```
# El data set tiene 14 columnas: 1 variables respuesta y 13 predictores.
# 13 es el valor que se tiene que indicar en el argumento mtry.
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)
modelo_bagging <- randomForest(medv ~ ., data = Boston, subset = train, mtry = 13)
modelo_bagging
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 10.63887
##                     % Var explained: 87.12
```

Al imprimir el resultado por pantalla se muestra: el número de árboles generados en el proceso, el número de predictores evaluados en cada división (en este caso todos los disponibles), el *Mean of Squared Residuals* y el porcentaje de varianza que el modelo es capaz

de explicar. El término *Mean of Squared Residuals* hace referencia al *out-of-bag MSE*, que como se ha indicado anteriormente, puede interpretarse como una estimación del *test error*.

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

```
predicciones <- predict(object = modelo_bagging, newdata = Boston[-train,])
test_mse     <- mean((predicciones - Boston[-train, "medv"])^2)
paste("Error de test (mse) del modelo obtenido por bagging es:", round(test_mse,2))
```

```
## [1] "Error de test (mse) del modelo obtenido por bagging es: 13.18"
```

El *test-MSE* asociado al árbol de regresión obtenido por *bagging* es de 13.18, casi la mitad del obtenido anteriormente con un árbol generado por *pruning* (23). Esto pone de manifiesto la superioridad de los modelos obtenidos por *bagging* en comparación a los obtenidos por *pruning*.

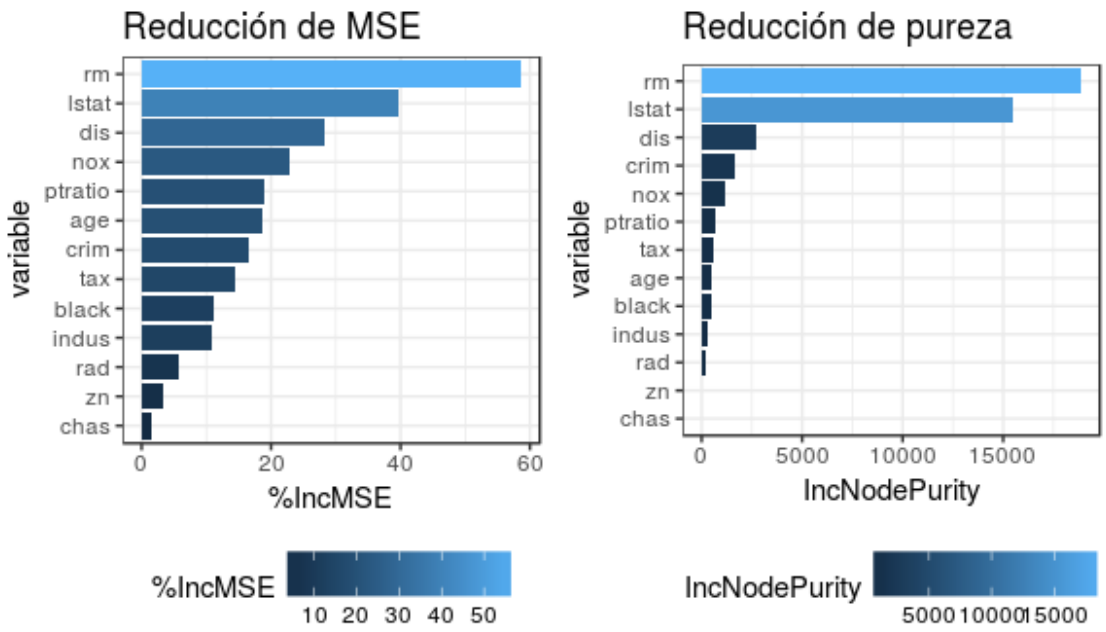
Gracias a la estimación del *out-of-bag MSE* y su similitud con el *leave-one-out error*, realmente no es necesario dividir las observaciones disponibles en *training* y *test* para evaluar el modelo. Se pueden emplear todas ellas en el proceso de *bagging* y considerar el *out-of-bag MSE* como una estimación de *test-MSE*. Esto es una ventaja importante, sobretodo, cuando no se dispone de muchas observaciones. Sin embargo, para comparar un modelo de *bagging* con otros modelos distintos, sí se necesita disponer de un set de test para que las comparaciones sean estrictamente válidas.

```
modelo_bagging <- randomForest(medv ~ ., data = Boston, mtry = 13, ntree = 500,
                               importance = TRUE)
modelo_bagging
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, ntree = 500,
importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 10.4683
##                     % Var explained: 87.6
```

Como se ha comentado en las definiciones teóricas anteriores, una de las principales ventajas de los modelos de *bagging* es su capacidad para cuantificar la importancia de los predictores. La función `importance()` permite extraer esta información de objetos creados mediante `randomforest()`. El resultado es una matriz que contiene dos medidas de importancia: *%IncMSE* y *IncNodePurity*.

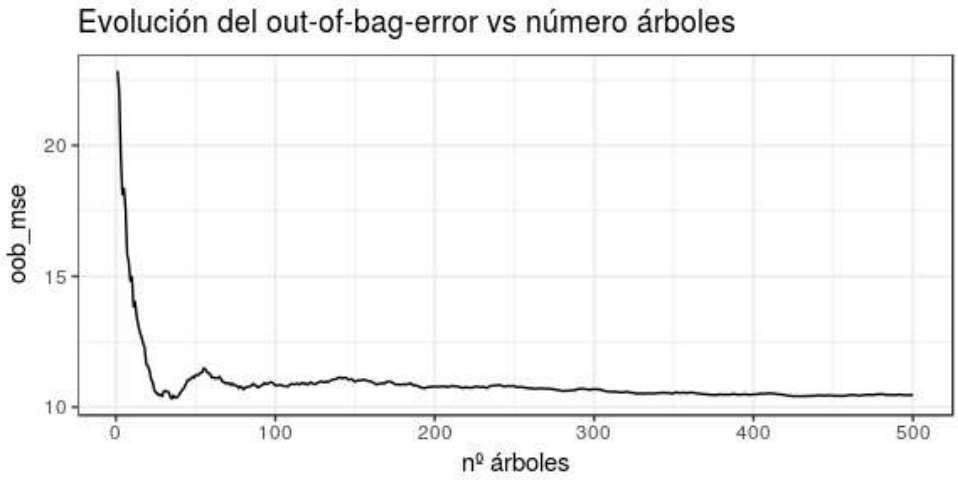
```
library(tidyverse)
library(ggpubr)
importancia_pred <- as.data.frame(importance(modelo_bagging, scale = TRUE))
importancia_pred <- rownames_to_column(importancia_pred, var = "variable")
p1 <- ggplot(data = importancia_pred, aes(x = reorder(variable, `%IncMSE`),
                                          y = `%IncMSE`,
                                          fill = `%IncMSE`)) +
    labs(x = "variable", title = "Reducción de MSE") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")

p2 <- ggplot(data = importancia_pred, aes(x = reorder(variable, IncNodePurity),
                                          y = IncNodePurity,
                                          fill = IncNodePurity)) +
    labs(x = "variable", title = "Reducción de pureza") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")
ggarrange(p1, p2)
```



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

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

```
oob_mse <- data.frame(oob_mse = modelo_bagging$mse,
                      arboles = seq_along(modelo_bagging$mse))
ggplot(data = oob_mse, aes(x = arboles, y = oob_mse )) +
  geom_line() +
  labs(title ="Evolución del out-of-bag-error vs número árboles", x="nº árboles") +
  theme_bw()
```



En este caso, a partir de aproximadamente 100 árboles la precisión del modelo se estabiliza. De hecho, tanto el *out-of-bag-MSE* como el *test error* que se obtienen si en lugar de 500 árboles se emplean solo 100 es casi el mismo.

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

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, ntree = 100,
subset = train)
##                Type of random forest: regression
##                      Number of trees: 100
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 10.53909
##                     % Var explained: 87.24
```

```
# Cálculo del test error
predicciones <- predict(object = modelo_bagging, newdata = Boston[-train,])
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
paste("Error de test (mse) del modelo obtenido por bagging es:",
      round(test_mse,2))
```

```
## [1] "Error de test (mse) del modelo obtenido por bagging es: 13.54"
```

Random Forest

El método de *random forest* es una modificación del proceso de *bagging* que consigue mejores resultados gracias a que *decorrelaciona* los árboles generados en el proceso. Recordando el apartado anterior, el proceso de *bagging* se basa en el hecho de que, promediando un conjunto de 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 m predictores antes de evaluar cada división. De esta forma, un promedio de $(p - m)/p$ divisiones no contemplarán el predictor influyente, permitiendo que otros predictores puedan ser seleccionados. Solo con añadir este paso extra se consigue *decorrelacionar* los árboles, por 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 m predictores. La diferencia en el resultado dependerá del valor m escogido. Si $m = p$ los resultados de *random forest* y *bagging* son equivalentes. Un valor recomendado del hiperparámetro es $m \approx \sqrt{p}$, siendo p el número de predictores totales. Sin embargo, la mejor forma para encontrar el valor óptimo de m es evaluar el *out-of-bag-MSE* para diferentes valores de m. Por lo general, si los predictores están muy correlacionados, valores pequeños de m consiguen mejores resultados.

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

Ejemplo regresión

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

Con la función `randomForest()` se puede generar un modelo *random forest* de la misma forma descrita en el ejemplo anterior de *bagging*. La única diferencia es que, en el argumento `mtry`, se indica un número menor que el total de predictores. Por defecto, la función emplea $p/3$ predictores cuando aplica *random forest* a árboles de regresión y $\sqrt{p}$ cuando son árboles de clasificación. Sin embargo, en lugar de emplear estos valores, es mejor identificar el valor óptimo estudiando el *out-of-bag-MSE* en función del número de predictores evaluados en cada división.

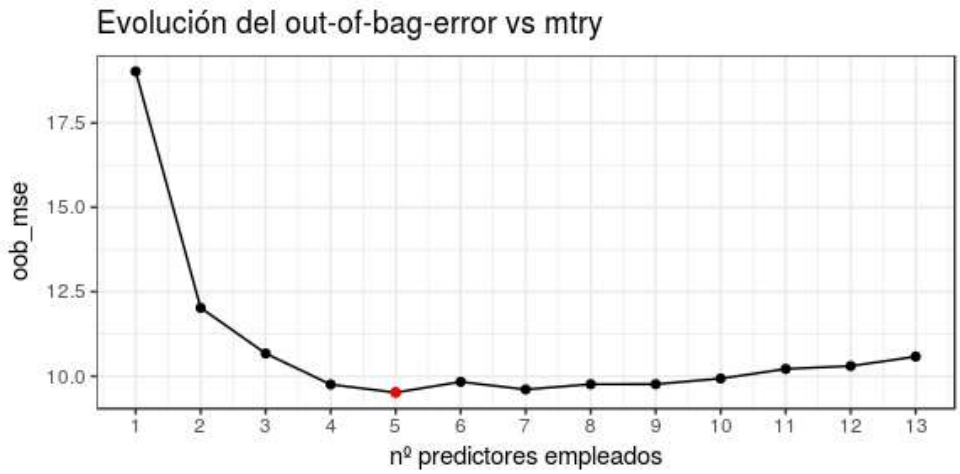
Optimización de hiperparámetros

```
tuning_rf_mtry <- function(df, y, ntree = 500){
  # Esta función devuelve el out-of-bag-MSE de un modelo RandomForest en función
  # del número de predictores evaluados (mtry)

  # Argumentos:
  #   df = data frame con los predictores y variable respuesta
  #   y  = nombre de la variable respuesta
  #   ntree = número de árboles creados en el modelo randomForest

  require(dplyr)
  max_predictores <- ncol(df) - 1
  n_predictores   <- rep(NA, max_predictores)
  oob_mse         <- rep(NA, max_predictores)
```

```
  for (i in 1:max_predictores) {
    set.seed(123)
    f <- formula(paste(y,"~ ."))
    modelo_rf <- randomForest(formula = f, data = df, mtry = i, ntree = ntree)
    n_predictores[i] <- i
    oob_mse[i] <- tail(modelo_rf$mse, n = 1)
  }
  results <- data_frame(n_predictores, oob_mse)
  return(results)
}

hiperparametro_mtry <-  tuning_rf_mtry(df = Boston, y = "medv")
hiperparametro_mtry %>% arrange(oob_mse)
```

```
## # A tibble: 13 x 2
##    n_predictores   oob_mse
##            <int>     <dbl>
##  1             5  9.515977
##  2             7  9.606680
##  3             4  9.753594
##  4             8  9.761492
##  5             9  9.762749
##  6             6  9.832155
##  7            10  9.930386
##  8            11 10.215321
##  9            12 10.301175
## 10            13 10.580720
## 11             3 10.672321
## 12             2 12.022802
## 13             1 19.026398
```

```
ggplot(data = hiperparametro_mtry, aes(x = n_predictores, y = oob_mse)) +
  scale_x_continuous(breaks = hiperparametro_mtry$n_predictores) +
  geom_line() +
  geom_point() +
  geom_point(data = hiperparametro_mtry %>% arrange(oob_mse) %>% head(1),
             color = "red") +
  labs(title = "Evolución del out-of-bag-error vs mtry",
       x = "nº predictores empleados") +
  theme_bw()
```



Se identifica 5 como número óptimo de predictores a evaluar en cada división.

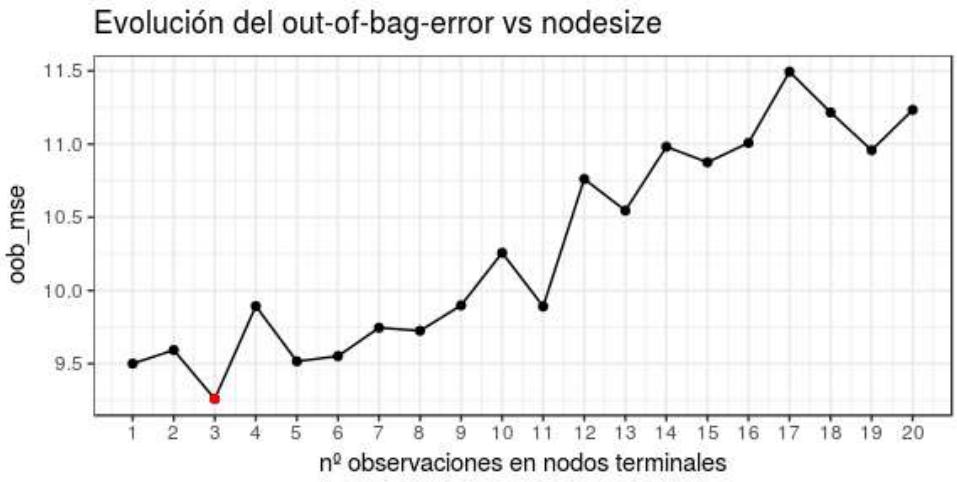
Además del número de predictores evaluados en cada división, el método de *ranfom forest* tiene otros dos hiperparámetros: el número mínimo de observaciones que deben tener los nodos terminales (nodesize) y el número de árboles ajustados (ntree). La función `randomForest()` emplea como valores por defecto `nodesize = 1` en problemas de clasificación, `nodesize = 5` en regresión y `ntree = 500`, sin embargo, ambos hiperparámetros pueden optimizarse empleando el *out of bag error*.

```
tuning_rf_nodesize <- function(df, y, size = NULL, ntree = 500){
  # Esta función devuelve el out-of-bag-MSE de un modelo randomForest
  # en función del tamaño mínimo de los nodos terminales (nodesize).
  # Argumentos:
  #   df = data frame con los predictores y variable respuesta
  #   y  = nombre de la variable respuesta
  #   sizes = tamaños evaluados
  #   ntree = número de árboles creados en el modelo randomForest

  require(dplyr)
  if (is.null(size)){
    size <- seq(from = 1, to = nrow(df), by = 5)
  }
  oob_mse <- rep(NA, length(size))
  for (i in seq_along(size)) {
    set.seed(123)
    f <- formula(paste(y,"~ ."))
    modelo_rf <- randomForest(formula = f, data = df, mtry = 5, ntree = ntree,
                              nodesize = i)
    oob_mse[i] <- tail(modelo_rf$mse, n = 1)
  }
  results <- data_frame(size, oob_mse)
  return(results)
}
```

```
hiperparametro_nodesize <-  tuning_rf_nodesize(df = Boston, y = "medv",
                                               size = c(1:20))
hiperparametro_nodesize %>% arrange(oob_mse)
```

```
## # A tibble: 20 x 2
##     size   oob_mse
##    <int>     <dbl>
##  1     3  9.259399
##  2     1  9.500649
##  3     5  9.515977
##  4     6  9.551852
##  5     2  9.593219
##  6     8  9.725140
##  7     7  9.745709
##  8    11  9.891092
##  9     4  9.893276
## 10     9  9.898667
## 11    10 10.257612
## 12    13 10.545811
## 13    12 10.761486
## 14    15 10.875378
## 15    19 10.958240
## 16    14 10.981766
## 17    16 11.008234
## 18    18 11.216736
## 19    20 11.233598
## 20    17 11.494297
```

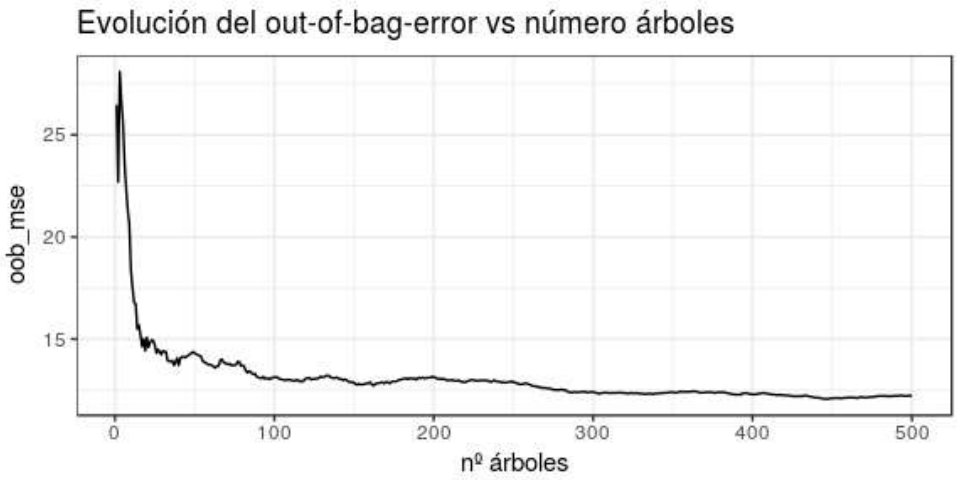
```
ggplot(data = hiperparametro_nodesize, aes(x = size, y = oob_mse)) +
  scale_x_continuous(breaks = hiperparametro_nodesize$size) +
  geom_line() +
  geom_point() +
  geom_point(data = hiperparametro_nodesize %>% arrange(oob_mse) %>% head(1),
             color = "red") +
  labs(title = "Evolución del out-of-bag-error vs nodesize",
       x = "nº observaciones en nodos terminales") +
  theme_bw()
```



Se identifica el 3 como número óptimo de observaciones mínimas que deben contener los nodos terminales.

Por último, se estudia la evolución del *out of bag error* en función del número de árboles. En el caso de *random forest*, este hiperparámetro no afecta al *overfitting*, por lo que el principal objetivo de su optimización es evitar consumir recursos computacionales innecesarios.

```
modelo_randomforest <- randomForest(medv ~ ., data = Boston, subset = train,
                                    mtry = 5 , ntree = 500, nodesize = 3,
                                    importance = TRUE)
oob_mse <- data.frame(oob_mse = modelo_randomforest$mse,
                      arboles = seq_along(modelo_randomforest$mse))
ggplot(data = oob_mse, aes(x = arboles, y = oob_mse )) +
  geom_line() +
  labs(title = "Evolución del out-of-bag-error vs número árboles",
       x = "nº árboles") +
  theme_bw()
```



A partir de aproximadamente 100 árboles el error del modelo se estabiliza.

Nota: La búsqueda de mejores hiperparámetros no debe hacerse de forma secuencial, ya que cada hiperparámetro interacciona con los demás. Sin embargo, para no añadir una capa de complejidad extra he optado por la búsqueda más sencilla. El paquete caret implementa múltiples formas adecuadas de encontrar los hiperparámetros óptimos.

Ajuste final y error de test

```
set.seed(123)
modelo_randomforest <- randomForest(medv ~ ., data = Boston, subset = train,
                                    mtry = 5 , ntree = 100, nodesize = 3,
                                    importance = TRUE)
modelo_randomforest
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 5, ntree = 100,
nodesize = 3, importance = TRUE, subset = train)
##                Type of random forest: regression
##                      Number of trees: 100
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 11.55709
##                     % Var explained: 86.01
```

```
predicciones <- predict(object = modelo_randomforest, newdata = Boston[-train, ])
test_mse     <- mean((predicciones - Boston[-train, "medv"])^2)
paste("Error de test (mse) del modelo:", round(test_mse, 2))
```

```
## [1] "Error de test (mse) del modelo: 10.89"
```

```
# Empleando todas las observaciones en el proceso de randomforest
set.seed(123)
modelo_randomforest <- randomForest(medv ~ ., data = Boston, mtry = 5, ntree = 100,
                                    nodesize = 3, importance = TRUE)
modelo_randomforest
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 5, ntree = 100,
nodesize = 3, importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 100
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 9.619532
##                     % Var explained: 88.61
```

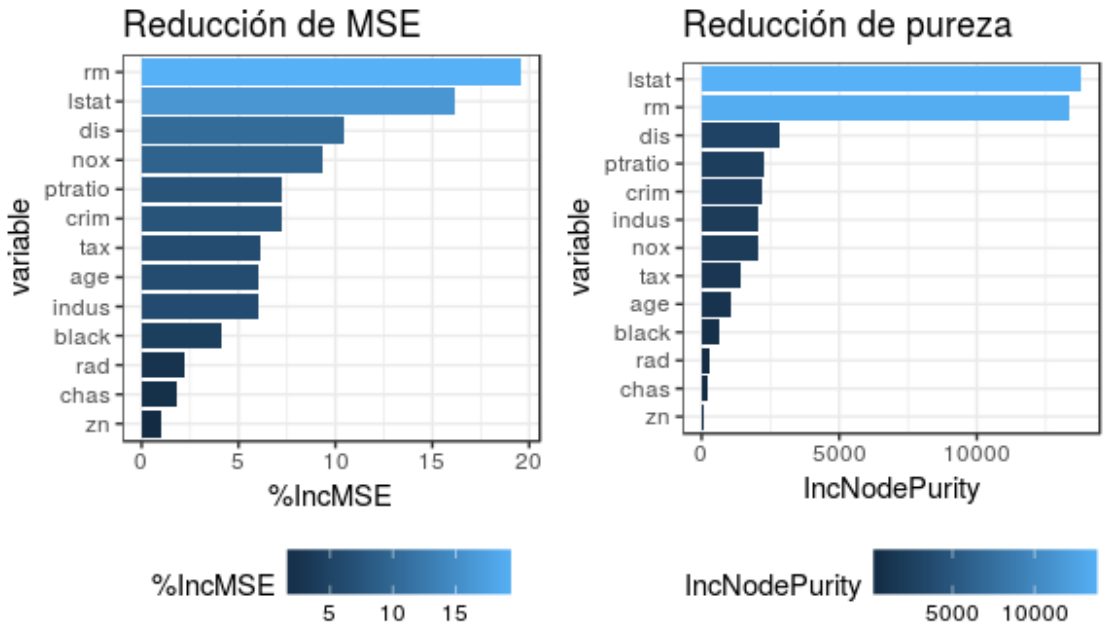
El *test MSE* (10.89) obtenido con el *test set* y el *out-of-bag-MSE* (9.62) estimado empleando todas las observaciones, son similares y ambos menores los obtenidos por *bagging* (13.18 y 10.63). Gracias a la decorrelación, el método *random forest* suele superar al de *bagging*.

Identificación de los predictores más influyentes

Con la función `importance()` se extrae la importancia de las variables.

```
library(tidyverse)
library(ggpubr)
importancia_pred <- as.data.frame(importance(modelo_randomforest, scale = TRUE))
importancia_pred <- rownames_to_column(importancia_pred, var = "variable")
p1 <- ggplot(data = importancia_pred, aes(x = reorder(variable, `%IncMSE`),
                                          y = `%IncMSE`,
                                          fill = `%IncMSE`)) +
    labs(x = "variable", title = "Reducción de MSE") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")
```

```
p2 <- ggplot(data = importancia_pred, aes(x = reorder(variable, IncNodePurity),
                                          y = IncNodePurity,
                                          fill = IncNodePurity)) +
    labs(x = "variable", title = "Reducción de pureza") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")
ggarrange(p1, p2)
```



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

Ejemplo clasificación

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

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

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

Optimización de hiperparámetros

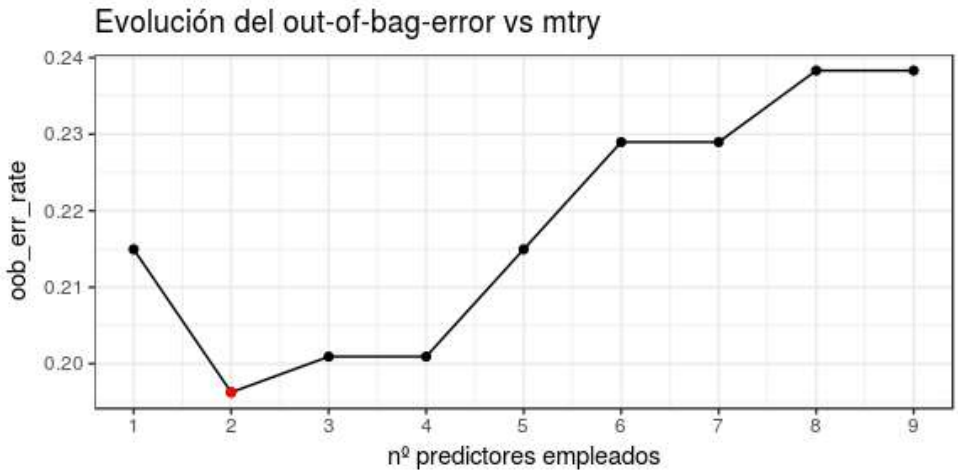
Identificación del valor óptimo del hiperparámetro `mtry`.

```
tuning_rf_mtry <- function(df, y, ntree = 500){
  # Esta función devuelve el out-of-bag clasification error de un modelo
  # randomforest en función del número de predictores evaluados (mtry)

  # Argumentos:
  #   df = data frame con los predictores y variable respuesta
  #   y  = nombre de la variable respuesta
  #   ntree = número de árboles creados en el modelo randomForest

  require(dplyr)
  max_predictores <- ncol(df) - 1
  n_predictores   <- rep(NA, max_predictores)
  oob_err_rate    <- rep(NA, max_predictores)
  for (i in 1:max_predictores) {
    set.seed(123)
    f <- formula(paste(y,"~ ."))
    modelo_rf <- randomForest(formula = f, data = df, mtry = i, ntree = ntree)
    n_predictores[i] <- i
    oob_err_rate[i] <- tail(modelo_rf$err.rate[, 1], n = 1)
  }
```

```
  results <- data_frame(n_predictores, oob_err_rate)
  return(results)
}

hiperparametro_mtry <-  tuning_rf_mtry(df = fgl, y = "type")
hiperparametro_mtry %>% arrange(oob_err_rate)
```

```
## # A tibble: 9 x 2
##   n_predictores oob_err_rate
##           <int>        <dbl>
## 1             2    0.1962617
## 2             3    0.2009346
## 3             4    0.2009346
## 4             1    0.2149533
## 5             5    0.2149533
## 6             6    0.2289720
## 7             7    0.2289720
## 8             8    0.2383178
## 9             9    0.2383178
```

```
ggplot(data = hiperparametro_mtry, aes(x = n_predictores, y = oob_err_rate)) +
  scale_x_continuous(breaks = hiperparametro_mtry$n_predictores) +
  geom_line() +
  geom_point() +
  geom_point(data = hiperparametro_mtry %>% arrange(oob_err_rate) %>% head(1),
             color = "red") +
  labs(title = "Evolución del out-of-bag-error vs mtry",
       x = "nº predictores empleados") +
  theme_bw()
```



Se identifica el valor 2 como número óptimo de predictores a evaluar en cada división.

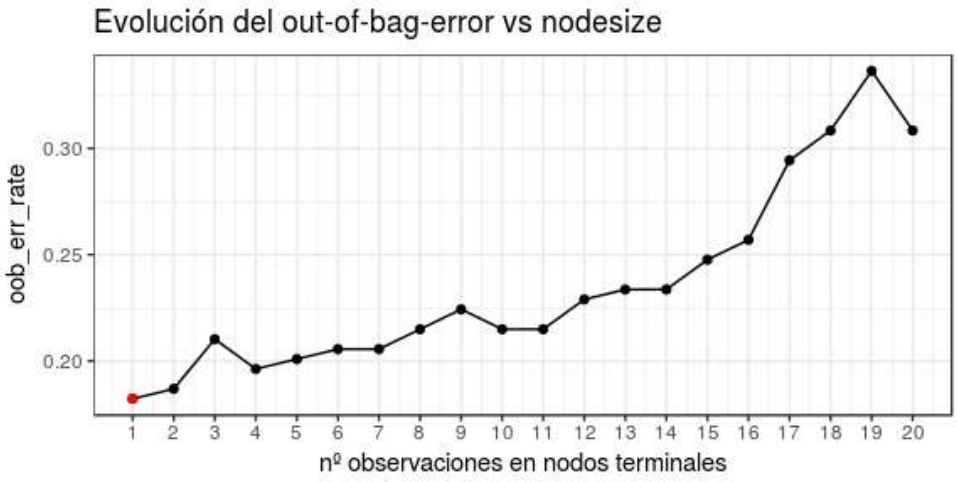
Identificación del valor óptimo del hiperparámetro `nodesize`.

```
tuning_rf_nodesize <- function(df, y, size = NULL, ntree = 500){
  # Esta función devuelve el out-of-bag clasification error de un modelo
RandomForest
  # en función del tamaño mínimo de los nodos terminales (nodesize).

  # Argumentos:
  #   df = data frame con los predictores y variable respuesta
  #   y  = nombre de la variable respuesta
  #   sizes = tamaños evaluados
  #   ntree = número de árboles creados en el modelo randomForest

  require(dplyr)
  if (is.null(size)){
    size <- seq(from = 1, to = nrow(df), by = 5)
  }
  oob_err_rate <- rep(NA, length(size))
  for (i in seq_along(size)) {
    set.seed(321)
    f <- formula(paste(y,"~ ."))
    modelo_rf <- randomForest(formula = f, data = df, mtry = 2, ntree = ntree,
                              nodesize = i)
    oob_err_rate[i] <- tail(modelo_rf$err.rate[, 1], n = 1)
  }
  results <- data_frame(size, oob_err_rate)
  return(results)
}

hiperparametro_nodesize <-  tuning_rf_nodesize(df = fgl, y = "type",
                                               size = c(1:20))
hiperparametro_nodesize %>% arrange(oob_err_rate)
```

```
## # A tibble: 20 x 2
##     size oob_err_rate
##    <int>        <dbl>
##  1     1    0.1822430
##  2     2    0.1869159
##  3     4    0.1962617
##  4     5    0.2009346
##  5     6    0.2056075
##  6     7    0.2056075
##  7     3    0.2102804
##  8     8    0.2149533
##  9    10    0.2149533
## 10    11    0.2149533
## 11     9    0.2242991
## 12    12    0.2289720
## 13    13    0.2336449
```

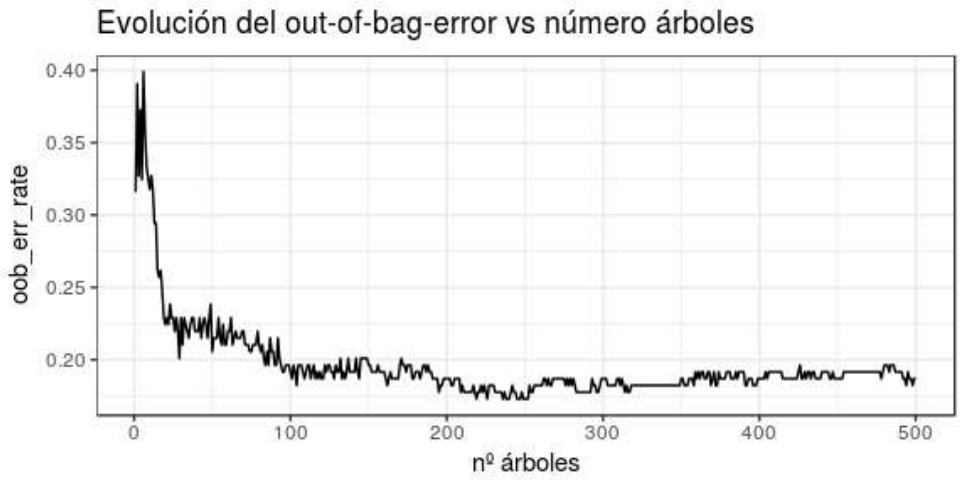
```
## 14    14    0.2336449
## 15    15    0.2476636
## 16    16    0.2570093
## 17    17    0.2943925
## 18    18    0.3084112
## 19    20    0.3084112
## 20    19    0.3364486
```

```
ggplot(data = hiperparametro_nodesize, aes(x = size, y = oob_err_rate)) +
  scale_x_continuous(breaks = hiperparametro_nodesize$size) +
  geom_line() +
  geom_point() +
  geom_point(data = hiperparametro_nodesize %>% arrange(oob_err_rate) %>% head(1),
             color = "red") +
  labs(title = "Evolución del out-of-bag-error vs nodesize",
       x = "nº observaciones en nodos terminales") +
  theme_bw()
```



Se identifica el 1 como número óptimo de observaciones mínimas que deben contener los nodos terminales.

```
modelo_randomforest <- randomForest(type ~ ., data = fgl, mtry = 2, ntree = 500,
                                    importance = TRUE, nodesize = 1)

oob_err_rate <- data.frame(oob_err_rate = modelo_randomforest$err.rate[, 1],
                           arboles = seq_along(modelo_randomforest$err.rate[, 1]))
ggplot(data = oob_err_rate, aes(x = arboles, y = oob_err_rate )) +
  geom_line() +
  labs(title = "Evolución del out-of-bag-error vs número árboles",
       x = "nº árboles") +
  theme_bw()
```



Alcanzados en torno a 300 árboles el error del modelo se estabiliza.

Ajuste final y error de test

```
set.seed(17)
modelo_randomforest <- randomForest(type ~ ., data = fgl, mtry = 2, ntree = 300,
                                    importance = TRUE, nodesize = 1,
                                    norm.votes = TRUE )
modelo_randomforest
```

```
##
## Call:
##  randomForest(formula = type ~ ., data = fgl, mtry = 2, ntree = 300,
importance = TRUE, nodesize = 1, norm.votes = TRUE)
##                Type of random forest: classification
##                      Number of trees: 300
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 20.56%
## Confusion matrix:
##       WinF WinNF Veh Con Tabl Head class.error
## WinF    63     6   1   0    0    0   0.1000000
## WinNF   11    58   1   3    2    1   0.2368421
## Veh      9     2   6   0    0    0   0.6470588
## Con      0     2   0  10    0    1   0.2307692
## Tabl     1     1   0   0    7    0   0.2222222
## Head     1     2   0   0    0   26   0.1034483
```

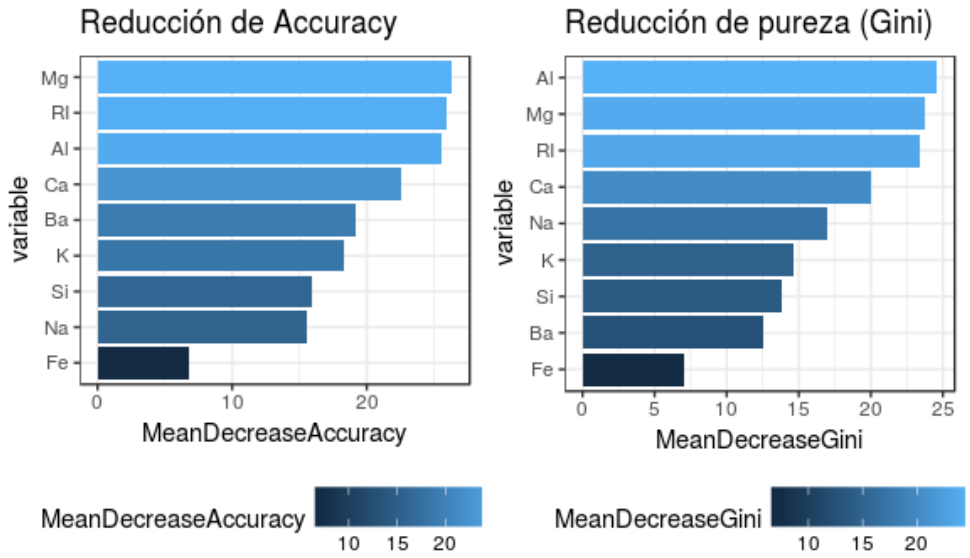
Al tratarse de un problema de clasificación, es conveniente no solo mostrar la clase predicha (la más votada) sino también la fracción de votos para cada clase.

```
head(modelo_randomforest$votes)
```

```
##         WinF     WinNF         Veh Con        Tabl        Head
## 1 0.48979592 0.3163265 0.193877551   0 0.000000000 0.000000000
## 2 0.57407407 0.3611111 0.046296296   0 0.009259259 0.009259259
## 3 0.04237288 0.9322034 0.025423729   0 0.000000000 0.000000000
## 4 0.49532710 0.4299065 0.074766355   0 0.000000000 0.000000000
## 5 0.72072072 0.2792793 0.000000000   0 0.000000000 0.000000000
## 6 0.18018018 0.8108108 0.009009009   0 0.000000000 0.000000000
```

Identificación de los predictores más influyentes

```
library(tidyverse)
library(ggpubr)
importancia_pred <- as.data.frame(importance(modelo_randomforest, scale = TRUE))
importancia_pred <- rownames_to_column(importancia_pred, var = "variable")
p1 <- ggplot(data=importancia_pred, aes(x=reorder(variable, MeanDecreaseAccuracy),
                                        y = MeanDecreaseAccuracy,
                                        fill = MeanDecreaseAccuracy)) +
    labs(x = "variable", title = "Reducción de Accuracy") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")

p2 <- ggplot(data = importancia_pred, aes(x = reorder(variable, MeanDecreaseGini),
                                          y = MeanDecreaseGini,
                                          fill = MeanDecreaseGini)) +
    labs(x = "variable", title = "Reducción de pureza (Gini)") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")
ggarrange(p1, p2)
```



El modelo obtenido tiene un *OOB estimated error rate* del 20.56%. De entre todos los predictores empleados, *RI, Mg, Al* y *Ca* son los más influyentes. De hecho, si se genera un modelo más simple que solo incluya estos predictores, la estimación del error aumenta poco.

```
set.seed(17)
arbol_randomforest <- randomForest(type ~ RI + Mg + Al + Ca, data = fgl, mtry = 2,
                                   nodesize = 1, ntree = 300, importance = TRUE)
arbol_randomforest
```

```
##
## Call:
##  randomForest(formula = type ~ RI + Mg + Al + Ca, data = fgl,      mtry = 2,
nodesize = 1, ntree = 300, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 300
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 23.83%
## Confusion matrix:
##       WinF WinNF Veh Con Tabl Head class.error
## WinF    62     6   2   0    0    0   0.1142857
## WinNF   11    57   3   2    2    1   0.2500000
## Veh      6     2   9   0    0    0   0.4705882
## Con      0     1   0   8    1    3   0.3846154
## Tabl     0     2   0   3    3    1   0.6666667
## Head     1     4   0   0    0   24   0.1724138
```

Extremely randomized trees

Los métodos de *extremely randomized trees* llevan la decorrelación de los árboles un paso más allá que *random forest*. En cada división de los nodos, solo evalúa un subconjunto aleatorio de los predictores disponibles y, además, dentro de cada predictor seleccionado solo se evalúa un subconjunto aleatorio de los posibles puntos de corte. En determinados escenarios, este método consigue reducir un poco más la varianza. El paquete `ExtraTrees` implementa algoritmos de este tipo.

Boosting

Boosting es otra estrategia de *ensemble* que se puede emplear con un amplio grupo de métodos de *statistical learning*, entre ellos los árboles de predicción. La idea detrás de *boosting* es ajustar, de forma secuencial, múltiples *weak learners* (modelos sencillos que predicen solo ligeramente mejor que lo esperado por azar). Cada nuevo modelo emplea información del modelo anterior para aprender de sus errores, mejorando iteración a iteración. En el caso de los árboles de predicción, un *weak learners* se consigue utilizando árboles con una o pocas ramificaciones. A diferencia del método de *bagging*, el *boosting* no hace uso de muestreo repetido (*bootstrapping*), por lo que cada árbol construido depende en gran medida de los árboles previos. Tres de los algoritmos de *boosting* más empleados son *AdaBoost*, *Gradient Boosting* y *Stochastic Gradient Boosting*.

Al igual que en *bagging*, es posible identificar la influencia que tiene cada predictor sobre un modelo *boosting*. En cada división de los árboles, se registra el descenso conseguido en la medida empleada como criterio de división (índice Gini, entropía o MSE). Para cada uno de los predictores, se calcula el descenso medio conseguido en el total de *weak learners* que forman el *ensemble*.

Los algoritmos de *boosting* se caracterizan por tener una cantidad considerable de hiperparámetros, cuyo valor óptimo se identifica mediante validación cruzada. Tres de los más comunes son:

- El número de *weak learners* o número de iteraciones: A diferencia del *bagging* y *random forest*, el *boosting* puede sufrir *overfitting* si este valor es excesivamente alto. Para evitarlo se emplea un término de regularización conocido como *learning rate*.

- *Learning rate* (λ): Controla el ritmo al que aprenden los modelos. Suelen recomendarse valores de 0.01 o 0.001, aunque la elección correcta puede variar dependiendo del

problema. Cuanto menor sea λ, más árboles se necesitan para alcanzar buenos resultados pero menor es el riesgo de *overfitting*.

- Si los *weak learners* son árboles, el número de divisiones *d* de cada árbol. Suelen emplearse valores pequeños, entre 1 y 10.

AdaBoost

La publicación en 1999 del algoritmo *AdaBoost (Adaptative Boosting)* por parte de Yoav Freund y Robert Schapire supuso un avance muy importante en el campo del aprendizaje estadístico, ya que hizo posible aplicar la estrategia de *boosting* a multitud de problemas. A continuación, se introduce *AdaBoost* con un problema de clasificación binaria.

Para el funcionamiento de *AdaBoost* es necesario establecer:

- Un tipo de modelo, normalmente llamado *weak learner* o *base learner*, que sea capaz de predecir la variable respuesta con un porcentaje de acierto ligeramente superior a lo esperado por azar. En el caso de los árboles de regresión, este *weak learner* suele ser un árbol con apenas unos pocos nodos.
- Codificar las dos clases de la variable respuesta como *+1* y *-1*.
- Un peso inicial e igual para todas las observaciones que forman el set de entrenamiento.

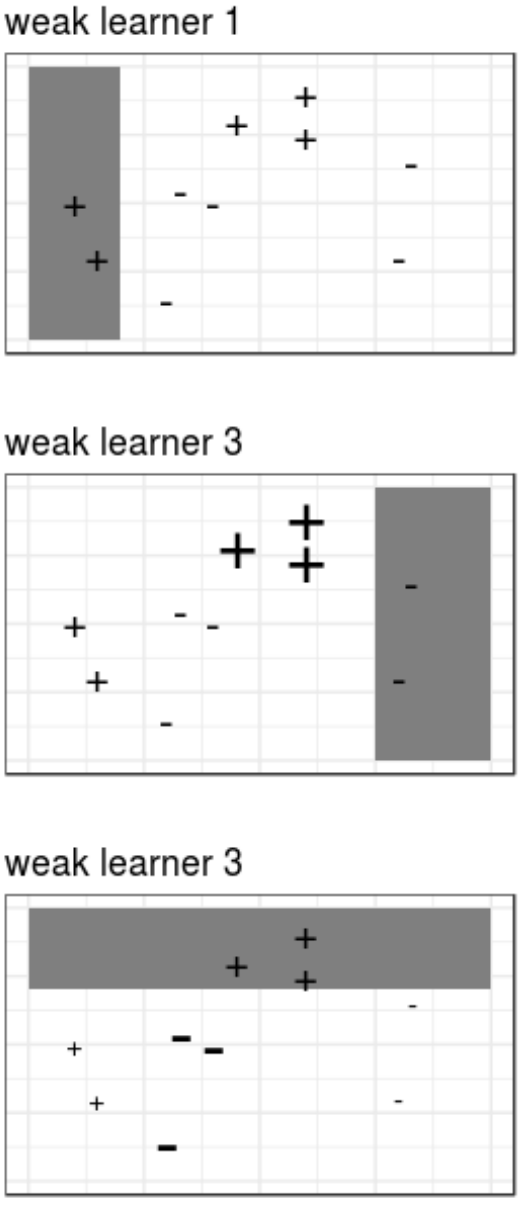
Una vez que estos tres puntos se han establecido, se inicia un proceso iterativo. En la primera iteración, se ajusta el *weak learner* empleando los datos de entrenamiento y los pesos iniciales (todos iguales). Con el *weak learner* ajustado y almacenado, se predicen las observaciones de entrenamiento y se identifican aquellas bien y mal clasificadas. Con esta información:

- Se actualizan los pesos de las observaciones, disminuyendo el de las que están bien clasificadas y aumentando el de las mal clasificadas.
- Se asigna un peso total al *weak learner*, proporcional al total de aciertos. Cuantos más aciertos consiga el *weak learner*, mayor su influencia en el conjunto del *ensemble*.

En la siguiente iteración, se llama de nuevo al *weak learner* y se ajusta, esta vez, empleando las observaciones de entrenamiento y los pesos actualizados en la iteración anterior. El nuevo *weak learner* se almacena, obteniendo así un nuevo modelo para el *ensemble*. Este proceso se repite *M* veces, generando un total de *M weak learner*. Para clasificar nuevas observaciones, se obtiene la predicción de cada uno de los *weak learners* que forman el *ensemble* y se agregan sus resultados, ponderando el peso de cada uno acorde al peso que se le ha asignado en el

ajuste. El objetivo detrás de esta estrategia es que cada nuevo *weak learner* se centra en predecir correctamente las observaciones que los anteriores no han sido capaces.

La siguiente imagen muestra como varían tres *weak learner* consecutivos dependiendo del peso asignado a las observaciones. En cada iteración, los pesos de las observaciones (representados con el tamaño) se ajustan dependiendo de si, en el *weak learner* anterior, fueron clasificadas de forma correcta o incorrecta.



Algoritmo AdaBoost

n: número de observaciones de entrenamiento

M: número de iteraciones de aprendizaje (número total de *weak learners*)

G_m: *weak learner* de la iteración m

w_i: peso de la observación i

α_m: peso del *weak learner* m

---

1. Inicializar los pesos de las observaciones, asignando el mismo valor a todas ellas:

$$w_i = \frac{1}{N}, \quad i = 1,2,\ldots,N$$

2. Para $m = 1$ hasta M:
   a. Ajustar el *weak learner* G_m utilizando las observaciones de entrenamiento y los pesos w_i.
   b. Calcular el error del *weak learner* como:

$$err_m = \frac{\sum_{i=1}^{N} w_i \, I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

   c. Calcular el peso asignado al *weak learner* G_m:

$$\alpha_m = log(\frac{1 - err_m}{err_m})$$

   d. Actualizar los pesos de las observaciones:

$$w_i = w_i exp[\alpha_m I(y_i \neq G_m(x_i))], \quad i = 1,2,\ldots,N$$

3. Predicción de nuevas observaciones agregando todos los *weak learners* y ponderándolos por su peso:

$$G(x) = sign[\sum_{m=1}^{M} \alpha_m \, G_m(x)]$$

Gradient Boosting

Gradient Boosting es una generalización del algoritmo *AdaBoost* que permite emplear cualquier función de coste, siempre que esta sea diferenciable. La flexibilidad de este algoritmo ha hecho posible aplicar *boosting* a multitud de problemas (regresión, clasificación con más de dos clases...). Para cada uno de ellos, el algoritmo de *Gradient Boosting* es ligeramente distinto, pero, para todos, la idea es la misma: dada una función de coste (por ejemplo, residuos cuadrados para regresión) y un *weak learner* (por ejemplo, árboles), el algoritmo trata de encontrar el modelo que minimiza la función de coste. Suele iniciarse con la mejor aproximación de la variable respuesta (la media en el caso de regresión), se calculan los residuos y con ellos se ajusta un nuevo *weak learner* que intente minimizar la función de coste. Este proceso se repite M veces, de forma que cada nuevo modelo minimiza los residuos (errores) del anterior.

Dado que el objetivo de *Gradient Boosting* es ir minimizando los residuos iteración a iteración, es susceptible de *overfitting*. Una forma de evitar este problema es empleando un valor de regularización, también conocido como *learning rate* (λ), que limite la influencia de cada modelo en el conjunto del *ensemble*. Como consecuencia de esta regularización, se necesitan más modelos para formar el *ensemble* pero se consiguen mejores resultados.

Stochastic Gradient Boosting

Tiempo después de la publicación de algoritmo de *Gradient Boosting*, se le incorporó una de las propiedades de *bagging*, el muestreo aleatorio de observaciones de entrenamiento. En concreto, en cada iteración del algoritmo, el *weak learner* se ajusta empleando únicamente una fracción del set de entrenamiento, extraída de forma aleatoria y sin reemplazo (no con *bootstrapping*). Al resultado de esta modificación se le conoce como *Stochastic Gradient Boosting* y aporta dos ventajas: consigue mejorar la capacidad predictiva y permite estimar el *out of bag error*, lo que facilita mucho la optimización de hiperparámetros.

Ejemplo regresión

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

La función `gbm()` del paquete `gbm` incorpora los algoritmos de *AdaBoost*, *Gradient Boosting* y *Stochastic Gradient Boosting*, empleando árboles de predicción a modo de *weak learners*. Esta función contiene multitud de argumentos, cuya correcta elección puede determinar en gran medida el modelo final. Algunos de los más importantes son:

- `Distribution`: determina la función de coste (*loss function*). Algunas de las más utilizadas son: *gaussian* (*squared loss*) para regresión, *bernulli* para respuestas binarias, *multinomial* para variables respuesta con más de dos clases y *adaboost* para respuestas binarias y que emplea la función exponencial del algoritmo original *AdaBoost*.

- `n.trees`: número de iteraciones del algoritmo de *boosting*, es decir, número de modelos que forman el *ensemble*. Cuanto mayor es este valor, más se reduce el error de entrenamiento, pudiendo llegar generarse *overfitting*.

- `interaction.depth`: determina la complejidad de los árboles empleados como *weak learner*, en concreto, el número total de divisiones que tiene el árbol. Emplear árboles con ente 1 y 6 nodos suele dar muy buenos resultados.

- `shrinkage`: este parámetro, también conocido como *learning rate*, controla la influencia que tiene cada modelo sobre el conjunto del *ensemble*. Es preferible mejorar un modelo mediante muchos pasos pequeños que mediante unos pocos grandes. Por esta razón, se recomienda emplear un valor de shrinkage tan pequeño como sea posible, teniendo en cuenta que, cuanto menor sea, mayor el número de iteraciones necesarias. Por defecto es de 0.001.

- `n.minobsinnode`: número mínimo de observaciones que debe tener un nodo para poder ser dividido. Al igual que `interaction.depth`, permite controlar la complejidad de los *weak learners* basados en árboles.

- `bag.fraction` (subsampling fraction): fracción de observaciones del set de entrenamiento seleccionadas de forma aleatoria para ajustar cada *weak learner*. Si su valor es de 1, se emplea el algoritmo de *Gradient Boosting*, si es menor que 1, se emplea *Stochastic Gradient Boosting*. Por defecto su valor es de 0.5.

- `cv.folds`: la función `gbm()` incorpora la posibilidad de realizar validación cruzada para obtener el error de validación en función del número de árboles.

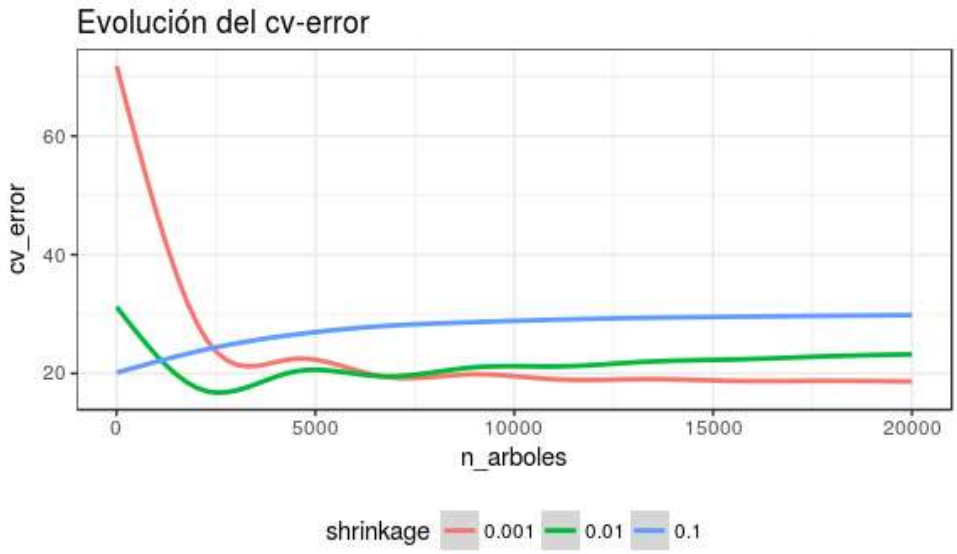
A continuación, se ajusta un modelo mediante *Stochastic Gradient Boosting* y se analiza como influye cada uno de los hiperparámetros en el error de validación cruzada y el error de test.

Nota 1: en lugar del error de validación cruzada, podría emplearse el out of bag error. Este útimo requiere mucha menos computación pero, con frecuencia, subestima la mejora conseguida al añadir más iretaciones.

Nota 2: La optimización de hiperparámetros no debe hacerse de forma secuancial, ya que cada hiperparámetro interacciona con los demás, el paquete caret implementa formas adecuadas de encontrar los hiperparámetros optimos. En este ejemplo, sin embargo, con la finalidad de mostrar como afecta cada hiperparámetro al ensemble final, se procede a analizar cada uno por separado.

Learning rate (shrinkage)

```
library(MASS)
library(gbm)
data("Boston")
set.seed(1)
train <- sample(1:nrow(Boston), size = nrow(Boston)/2)
cv_error  <- vector("numeric")
n_arboles <- vector("numeric")
shrinkage <- vector("numeric")
for (i in c(0.001, 0.01, 0.1)) {
  set.seed(123)
  arbol_boosting <- gbm(medv ~ ., data = Boston[train, ],
                        distribution = "gaussian",
                        n.trees = 20000,
                        interaction.depth = 1,
                        shrinkage = i,
                        n.minobsinnode = 10,
                        bag.fraction = 0.5,
                        cv.folds = 5)
  cv_error  <- c(cv_error, arbol_boosting$cv.error)
  n_arboles <- c(n_arboles, seq_along(arbol_boosting$cv.error))
  shrinkage <- c(shrinkage, rep(i, length(arbol_boosting$cv.error)))
}
error <- data.frame(cv_error, n_arboles, shrinkage)

ggplot(data = error, aes(x = n_arboles, y = cv_error,
                         color = as.factor(shrinkage))) +
  geom_smooth() +
  labs(title = "Evolución del cv-error", color = "shrinkage") +
  theme_bw() +
  theme(legend.position = "bottom")
```

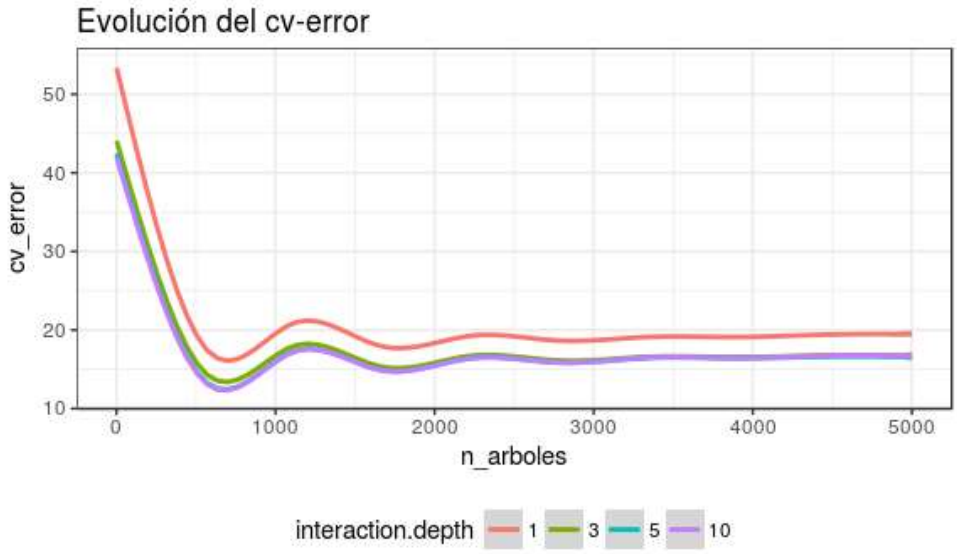


Puede observarse que, cuanto menor es el valor de *shrinkage*, más rápido aprende el modelo pero antes aparece el *overfitting*. Una estrategia seguida con frecuencia es emplear el menor valor *shrinkage* posible (teniendo en cuenta el tiempo de computación requerido) y con él, optimizar el resto de hiperparámetros. En este caso, cabe destacar que, aunque el mínimo error de validación cruzada se alcanza con *shrinkage = 0.001*, con *shrinkage = 0.01* se consigue un mínimo muy próximo pero con varios miles de árboles menos. Teniendo en cuenta todo esto, se selecciona como valor óptimo shrinkage = 0.01.

Complejidad de los árboles

```
cv_error  <- vector("numeric")
n_arboles <- vector("numeric")
interaction.depth <- vector("numeric")
for (i in c(1, 3, 5, 10)) {
  set.seed(123)
  arbol_boosting <- gbm(medv ~ ., data = Boston[train, ],
                        distribution = "gaussian",
                        n.trees = 5000,
                        interaction.depth = i,
                        shrinkage = 0.01,
                        n.minobsinnode = 10,
                        bag.fraction = 0.5,
                        cv.folds = 5)
  cv_error  <- c(cv_error, arbol_boosting$cv.error)
  n_arboles <- c(n_arboles, seq_along(arbol_boosting$cv.error))
  interaction.depth <- c(interaction.depth,
                         rep(i, length(arbol_boosting$cv.error)))
}
error <- data.frame(cv_error, n_arboles, interaction.depth)
```

```
ggplot(data = error, aes(x = n_arboles, y = cv_error,
                         color = as.factor(interaction.depth))) +
  geom_smooth() +
  labs(title = "Evolución del cv-error", color = "interaction.depth") +
  theme_bw() +
  theme(legend.position = "bottom")
```

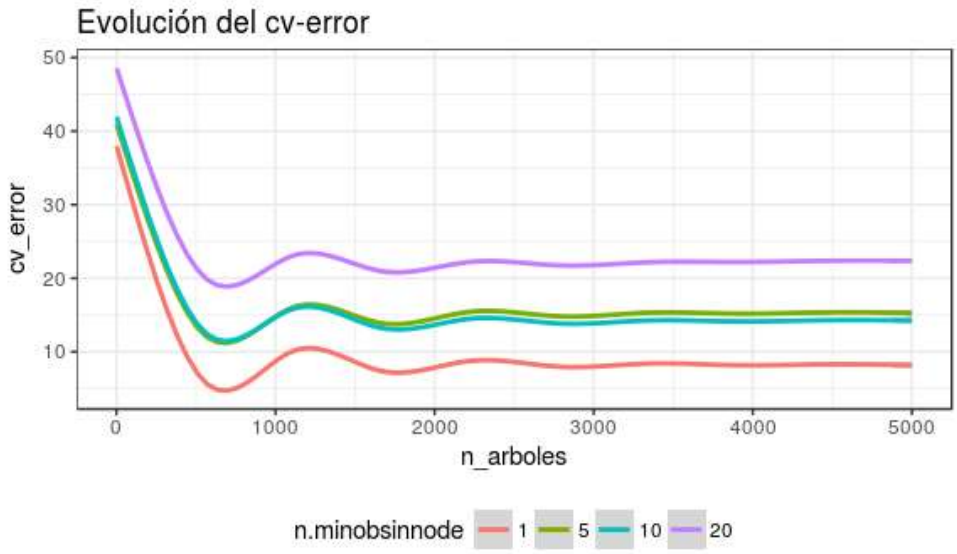


Para este problema en cuestión, empleando un `shrinkage = 0.01`, la complejidad de los árboles apenas influye en el error de validación cruzada. Los mejores resultados se obtienen con valores de 5 y 10. Siguiendo el principio de parsimonia, y para reducir la cantidad de memoria necesaria para almacenar el modelo, se selecciona `interaction.depth = 5`.

Número mínimo de observaciones por nodo

```
cv_error  <- vector("numeric")
n_arboles <- vector("numeric")
n.minobsinnode <- vector("numeric")
for (i in c(1, 5, 10, 20)) {
  arbol_boosting <- gbm(medv ~ ., data = Boston[train, ],
                        distribution = "gaussian",
                        n.trees = 5000,
                        interaction.depth = 5,
                        shrinkage = 0.01,
                        n.minobsinnode = i,
                        bag.fraction = 0.5,
                        cv.folds = 5)
```

```
  cv_error  <- c(cv_error, arbol_boosting$cv.error)
  n_arboles <- c(n_arboles, seq_along(arbol_boosting$cv.error))
  n.minobsinnode <- c(n.minobsinnode,
                      rep(i, length(arbol_boosting$cv.error)))
}
error <- data.frame(cv_error, n_arboles, n.minobsinnode)

ggplot(data = error, aes(x = n_arboles, y = cv_error,
                         color = as.factor(n.minobsinnode))) +
  geom_smooth() +
  labs(title = "Evolución del cv-error", color = "n.minobsinnode") +
  theme_bw() +
  theme(legend.position = "bottom")
```

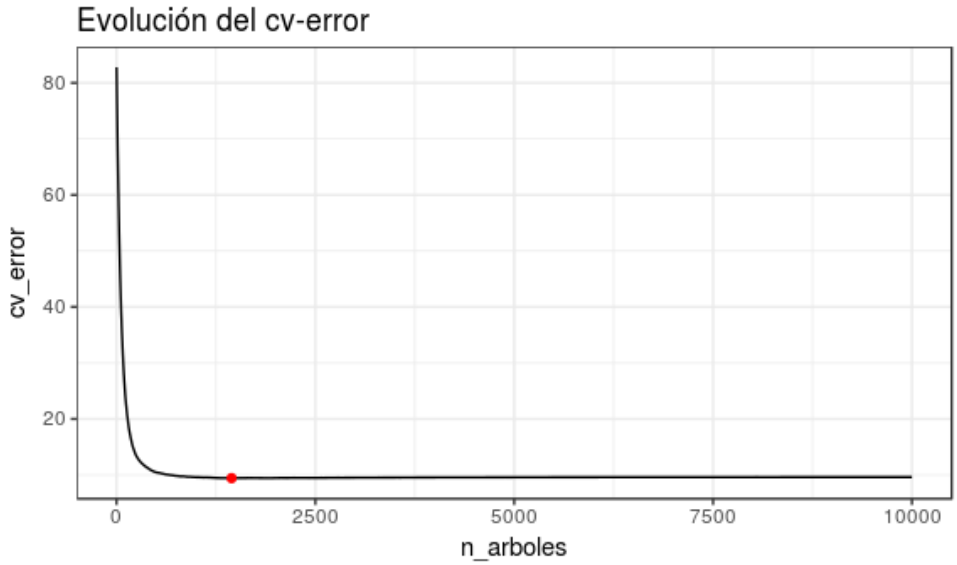


El número óptimo de observaciones para que un nodo pueda ser dividido, siendo `shrinkage = 0.01` y `interaction.depth = 5`, es de 1.

Número de árboles (iteraciones)

Una vez identificados el resto hiperparámetros óptimos, el último de ellos suele ser el número de árboles que debe contener el *ensemble*. Dado que cada árbol ocupa memoria y que un número excesivo puede conllevar problemas de *overfitting*, se debe seleccionar como cantidad óptima aquella a partir de la cual el error de validación alcanza la región de mínimos.

```
set.seed(123)
arbol_boosting <- gbm(medv ~ ., data = Boston[train, ],
                      distribution = "gaussian",
                      n.trees = 10000,
                      interaction.depth = 5,
                      shrinkage = 0.01,
                      n.minobsinnode = 1,
                      bag.fraction = 0.5,
                      cv.folds = 5)
error <- data.frame(cv_error = arbol_boosting$cv.error,
                    n_arboles = seq_along(arbol_boosting$cv.error))
ggplot(data = error, aes(x = n_arboles, y = cv_error)) +
  geom_line(color = "black") +
  geom_point(data = error[which.min(error$cv_error),], color = "red") +
  labs(title = "Evolución del cv-error") +
  theme_bw()
```




```
error[which.min(error$cv_error),]
```

```
##      cv_error n_arboles
## 1447 9.424089      1447
```

Un *ensemble* formado por 1447 árboles consigue el menor error de validación cruzada.

```
# Se reajusta el modelo final con los hiperparámetro óptimos
set.seed(123)
arbol_boosting <- gbm(medv ~ ., data = Boston[train, ],
                      distribution = "gaussian",
                      n.trees = 1447,
                      interaction.depth = 5,
                      shrinkage = 0.01,
                      n.minobsinnode = 1,
                      bag.fraction = 0.5)
```

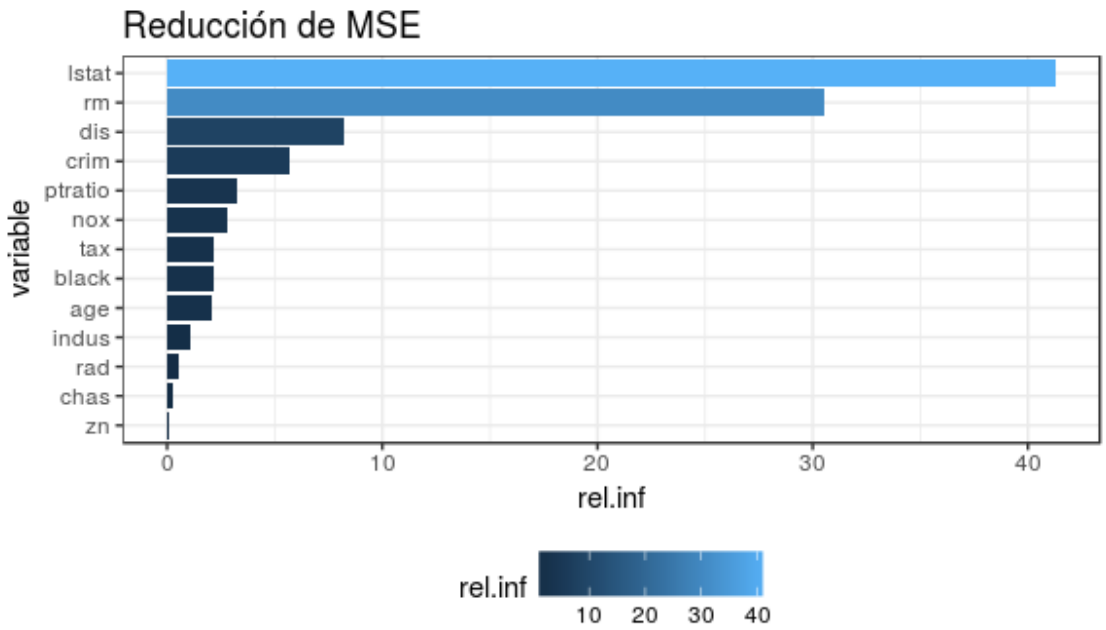
Búsqueda de hiperparámetros con caret

```
library(caret)

set.seed(123)
validacion <- trainControl(## 10-fold CV
                           method = "cv",
                           number = 10)

tuning_grid <-  expand.grid(interaction.depth = c(1, 5, 9),
                            n.trees = c(100, 1000, 2000, 3000),
                            shrinkage = c(0.1, 0.01, 0.001),
                            n.minobsinnode = c(1, 10, 20))

set.seed(123)
mejor_modelo <- train(medv ~ ., data = Boston[train, ],
                      method = "gbm",
                      trControl = validacion,
                      verbose = FALSE,
                      tuneGrid = tuning_grid)

# Se muestran los hiperparámetros del mejor modelo
mejor_modelo$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 51    2000                 5      0.01              1
```

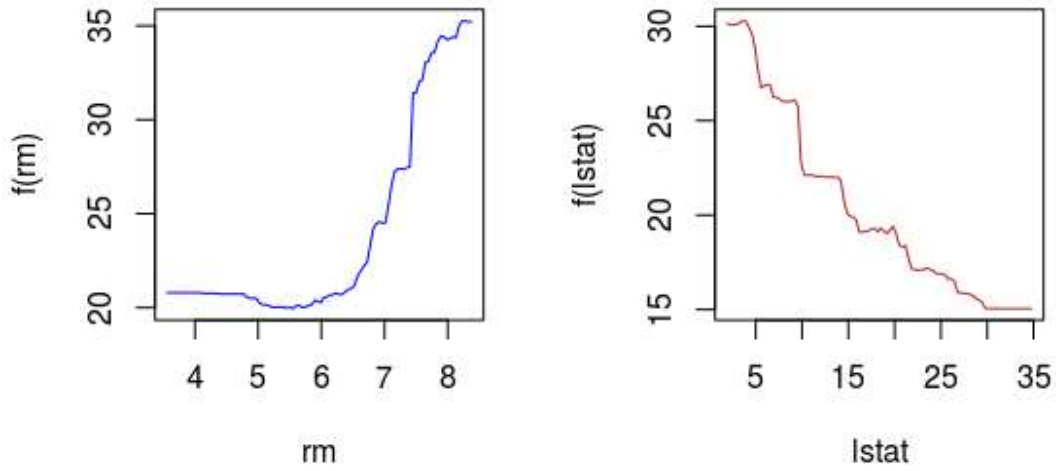
El `summary()` de un objeto `gbm` muestra la influencia relativa de cada predictor incluido en el modelo. De nuevo, las variables *rm* y *lstat* son las más influyentes.

```
importancia_pred <- summary(arbol_boosting, plotit = FALSE)
ggplot(data = importancia_pred, aes(x = reorder(var, rel.inf), y = rel.inf,
                                    fill = rel.inf)) +
  labs(x = "variable", title = "Reducción de MSE") +
  geom_col() +
  coord_flip() +
  theme_bw() +
  theme(legend.position = "bottom")
```



Además, la función `plot()` aplicada a un árbol `gbm` permite generar lo que se conoce como *partial dependence plots*, que muestran la influencia de un predictor sobre la variable respuesta, manteniendo constantes el resto de predictores. En este ejemplo, conforme aumenta *rm* aumenta el precio medio de la vivienda y lo opuesto ocurre con *lstat*.

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



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

```
predicciones <- predict(object = arbol_boosting, newdata = Boston[-train,],
                        n.trees = 1447)
test_mse <- mean((predicciones - Boston[-train, "medv"])^2)
paste("Error de test (mse) del modelo:", round(test_mse, 2))
```

```
## [1] "Error de test (mse) del modelo: 10.32"
```

El *test-error* conseguido es ligeramente inferior al obtenido mediante *random forest*.

Comparación Random Forest y Boosting

Ambos métodos son muy potentes y la superioridad de uno u otro depende del problema al que se apliquen. Algunos aspectos a tener en cuenta son:

- Gracias al *out-of-bag error*, el método de *random forest* no necesita recurrir a validación cruzada para la optimización de sus hiperparámetros. Esto puede ser una ventaja muy importante cuando los requerimientos computacionales son limitantes. Esta característica también está presente en *Stochastic Gradient Boosting* pero no en *AdaBoost* y *Gradient Boosting*.

- *Random forest* tiene menos hiperparámetros, lo que hace más sencillo su correcta implementación.

- Si existe una proporción alta de predictores irrelevantes, *random forest* puede verse perjudicado, sobre todo a medida que se reduce el número de predictores (m) evaluados. Supóngase que, de 100 predictores, 90 de ellos no aportan información (son ruidos). Si el hiperparámetro m es igual a 3 (en cada división se evalúan 3 predictores aleatorios), es muy probable que los 3 predictores seleccionados sean de los que no aportan nada. Aun así, el algoritmo seleccionará el mejor de ellos, incorporándolo al modelo. Cuanto mayor sea el porcentaje de predictores no relevantes, mayor la frecuencia con la que esto ocurre, por lo que los árboles que forman el *random forest* contendrán predictores irrelevantes. Como consecuencia, se reduce su capacidad predictiva. En el caso de *boosting*, siempre se evalúan todos los predictores, por lo que los no relevantes se ignoran.

- En *random forest*, cada modelo que forma el *ensemble* es independiente del resto, esto facilita mucho la paralelización del algoritmo.

- *Random forest* no sufre problemas de *overfitting* por muchos árboles que se agreguen.

C5.0

C5.0 es el algoritmo sucesor de C4.5, ambos publicados por Quinlan, con el objetivo de crear árboles de clasificación. Entre sus características, destacan la capacidad para generar árboles de predicción simples, modelos basados en reglas, *ensembles* basados en *boosting* y asignación de distintos pesos a los errores. Este algoritmo ha resultado de una gran utilidad a la hora de crear modelos de clasificación y todas sus capacidades son accesibles mediante el paquete `C50`. Aunque comparte muchas características con los algoritmos vistos en apartados anteriores, cabe tener en cuenta algunas peculiaridades:

- La medida de pureza empleada para las divisiones del árbol es la entropía.
- El podado de los árboles se realiza por defecto, y el método empleado se conoce como *pessimistic pruning*.
- Los árboles se pueden convertir en modelos basados en reglas.
- Emplea un algoritmo de boosting más próximo a *AdaBoost* que a *Gradient Boosting*.
- Por defecto, el algoritmo de *boosting* se detiene si la incorporación de nuevos modelos no aporta un mínimo de mejora.
- Incorpora una estrategia para la selección de predictores (*Winnowing*) previo ajuste del modelo.
- Permite asignar diferente peso a cada tipo de error.

Ejemplo

Ejemplo obtenido del libro *Machine Learning with R Second Edition by Brett Lantz*.

Supóngase una entidad bancaria que se dedica a conceder préstamos a sus clientes. La compañía ha detectado que parte de sus pérdidas económicas se deben a que, un porcentaje de clientes a los que se les concede un préstamo, nunca lo devuelven. El banco está interesado en un sistema de aprendizaje estadístico que ayude a identificar a qué clientes denegar el préstamo.

Se trata de un problema de clasificación binaria (si/no) que puede resolverse con un número considerable de herramientas (regresión logística, máquinas de vector soporte, redes neuronales…). En este escenario, además de la predicción, es importante poder entender las

razones por las que se concede o no un préstamo. En situaciones de este tipo, en las que la transparencia del modelo añade mucho valor, los sistemas basados en árboles suelen ser la elección adecuada. A continuación, se exploran las capacidades del algoritmo *C5.0* para resolver problemas de clasificación.

Carga y exploración de datos

El set de datos *credit.csv* puede descargarse registrándose de forma gratuita en packtpub.

```
library(tidyverse)
datos <- read_csv(file = "./datos/credit.csv")
dim(datos)
```

```
## [1] 1000   17
```

```
glimpse(datos)
```

```
## Observations: 1,000
## Variables: 17
## $ checking_balance     <chr> "< 0 DM", "1 - 200 DM", "unknown", "< 0 D...
## $ months_loan_duration <int> 6, 48, 12, 42, 24, 36, 24, 36, 12, 30, 12...
## $ credit_history       <chr> "critical", "good", "critical", "good", "...
## $ purpose              <chr> "furniture/appliances", "furniture/applia...
## $ amount               <int> 1169, 5951, 2096, 7882, 4870, 9055, 2835,...
## $ savings_balance      <chr> "unknown", "< 100 DM", "< 100 DM", "< 100...
## $ employment_duration  <chr> "> 7 years", "1 - 4 years", "4 - 7 years"...
## $ percent_of_income    <int> 4, 2, 2, 2, 3, 2, 3, 2, 2, 4, 3, 3, 1, 4,...
## $ years_at_residence   <int> 4, 2, 3, 4, 4, 4, 4, 2, 4, 2, 1, 4, 1, 4,...
## $ age                  <int> 67, 22, 49, 45, 53, 35, 53, 35, 61, 28, 2...
## $ other_credit         <chr> "none", "none", "none", "none", "none", "...
## $ housing              <chr> "own", "own", "own", "other", "other", "o...
## $ existing_loans_count <int> 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 2,...
## $ job                  <chr> "skilled", "skilled", "unskilled", "skill...
## $ dependents           <int> 1, 1, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ phone                <chr> "yes", "no", "no", "no", "no", "yes", "no...
## $ default              <chr> "no", "yes", "no", "no", "yes", "no", "no...
```

El set de datos contiene información sobre 1000 clientes. Para cada uno de ellos, se han registrado 17 variables de tipo cualitativo y cuantitativo. La variable *default* contiene la información sobre si el cliente defraudó o no.

De los 1000 clientes, 700 devolvieron el préstamo y 300 no. Para que un sistema de clasificación se considere útil, debe de tener una capacidad de acierto mayor que la frecuencia de la clase mayoritaria, en este caso *700/1000 = 0.7*.

```
table(datos$default)
```

```
##
##  no yes
## 700 300
```

Separación de las observaciones en entrenamiento y test

Para poder determinar la capacidad predictiva del modelo, se seleccionan aleatoriamente un 90% de las observaciones como entrenamiento y el 10% restante como test.

```
set.seed(123)
indices <- sample(x = 1:1000, size = 900)
train <- datos[indices,]
test  <- datos[-indices,]
```

Cuando se generan particiones de entrenamiento y test, es importante comprobar que las proporciones de los niveles de la variable respuesta son aproximadamente iguales entre ambas particiones e iguales al set original.

```
table(train$default)/nrow(train)
```

```
##
##        no       yes
## 0.7033333 0.2966667
```

```
table(test$default)/nrow(test)
```

```
##
##   no  yes
## 0.67 0.33
```

Árbol de clasificación simple

C5.0 necesita que las variables cualitativas estén almacenadas en forma de factor. Se procede a convertir todas las columnas del *dataframe* que no son numéricas en factor.

```
train <- train %>% map_if(.p = is_character, .f = as.factor) %>%
         as.data.frame()
test <- test %>% map_if(.p = is_character, .f = as.factor) %>%
         as.data.frame()

library(C50)
# Ajuste del árbol
arbol_clasificacion <- C5.0(default ~ ., data = train)
arbol_clasificacion
```

```
##
## Call:
## C5.0.formula(formula = default ~ ., data = train)
##
## Classification Tree
## Number of samples: 900
## Number of predictors: 16
##
## Tree size: 57
##
## Non-standard options: attempt to group attributes
```

El `summary` de un modelo `C50` muestra información detallada sobre el número de observaciones de entrenamiento, el número de predictores empleados, las divisiones del árbol y los errores de entrenamiento. En este caso, 133 de los 900 clientes han sido clasificados incorrectamente, el error de entrenamiento es del 14.8%.

```
summary(arbol_clasificacion)
```

```
##
## Call:
## C5.0.formula(formula = default ~ ., data = train)
##
##
## C5.0 [Release 2.07 GPL Edition]      Fri Dec 29 11:18:54 2017
## -------------------------------
##
## Class specified by attribute `outcome'
##
## Read 900 cases (17 attributes) from undefined.data
##
## Decision tree:
##
## checking_balance in {> 200 DM,unknown}: no (412/50)
```

```
## checking_balance in {< 0 DM,1 - 200 DM}:
## :...credit_history in {perfect,very good}: yes (59/18)
##     credit_history in {critical,good,poor}:
##     :...months_loan_duration <= 22:
##         :...credit_history = critical: no (72/14)
##         :   credit_history = poor:
##         :   :...dependents > 1: no (5)
##         :   :   dependents <= 1:
##         :   :   :...years_at_residence <= 3: yes (4/1)
##         :   :       years_at_residence > 3: no (5/1)
##         :   credit_history = good:
##         :   :...savings_balance in {> 1000 DM,500 - 1000 DM}: no (15/1)
##         :       savings_balance = 100 - 500 DM:
##         :       :...other_credit = bank: yes (3)
##         :       :   other_credit in {none,store}: no (9/2)
##         :       savings_balance = unknown:
##         :       :...other_credit = bank: yes (1)
##         :       :   other_credit in {none,store}: no (21/8)
##         :       savings_balance = < 100 DM:
##         :       :...purpose in {business,car0,renovations}: no (8/2)
##         :           purpose = education:
##         :           :...checking_balance = < 0 DM: yes (4)
##         :           :   checking_balance = 1 - 200 DM: no (1)
##         :           purpose = car:
##         :           :...employment_duration = > 7 years: yes (5)
##         :           :   employment_duration = unemployed: no (4/1)
##         :           :   employment_duration = 1 - 4 years:
##         :           :   :...years_at_residence <= 2: yes (2)
##         :           :   :   years_at_residence > 2: no (6/1)
##         :           :   employment_duration = < 1 year:
##         :           :   :...years_at_residence <= 2: yes (5)
##         :           :   :   years_at_residence > 2: no (3/1)
##         :           :   employment_duration = 4 - 7 years:
##         :           :   :...amount <= 1680: yes (2)
##         :           :       amount > 1680: no (3)
##         :           purpose = furniture/appliances:
##         :           :...job in {management,unskilled}: no (23/3)
##         :               job = unemployed: yes (1)
##         :               job = skilled:
##         :               :...months_loan_duration > 13: [S1]
##         :                   months_loan_duration <= 13:
##         :                   :...housing in {other,own}: no (23/4)
##         :                       housing = rent:
##         :                       :...percent_of_income <= 3: yes (3)
##         :                           percent_of_income > 3: no (2)
##         months_loan_duration > 22:
##         :...savings_balance = > 1000 DM: no (2)
##             savings_balance = 500 - 1000 DM: yes (4/1)
##             savings_balance = 100 - 500 DM:
##             :...credit_history in {critical,poor}: no (14/3)
```

```
##             :   credit_history = good:
##             :   :...other_credit = bank: no (1)
##             :       other_credit in {none,store}: yes (12/2)
##             savings_balance = unknown:
##             :...checking_balance = 1 - 200 DM: no (17)
##             :   checking_balance = < 0 DM:
##             :   :...credit_history = critical: no (1)
##             :       credit_history in {good,poor}: yes (12/3)
##             savings_balance = < 100 DM:
##             :...months_loan_duration > 47: yes (21/2)
##                 months_loan_duration <= 47:
##                 :...housing = other:
##                     :...percent_of_income <= 2: no (6)
##                     :   percent_of_income > 2: yes (9/3)
##                     housing = rent:
##                     :...other_credit = bank: no (1)
##                     :   other_credit in {none,store}: yes (16/3)
##                     housing = own:
##                     :...employment_duration = > 7 years: no (13/4)
##                         employment_duration = 4 - 7 years:
##                         :...job in {management,skilled,
##                         :   :       unemployed}: yes (9/1)
##                         :   job = unskilled: no (1)
##                         employment_duration = unemployed:
##                         :...years_at_residence <= 2: yes (4)
##                         :   years_at_residence > 2: no (3)
##                         employment_duration = 1 - 4 years:
##                         :...purpose in {business,car0,education}: yes (7/1)
##                         :   purpose in {furniture/appliances,
##                         :   :           renovations}: no (7)
##                         :   purpose = car:
##                         :   :...years_at_residence <= 3: yes (3)
##                         :       years_at_residence > 3: no (3)
##                         employment_duration = < 1 year:
##                         :...years_at_residence > 3: yes (5)
##                             years_at_residence <= 3:
##                             :...other_credit = bank: no (0)
##                                 other_credit = store: yes (1)
##                                 other_credit = none:
##                                 :...checking_balance = 1 - 200 DM: no (8/2)
##                                     checking_balance = < 0 DM:
##                                     :...job in {management,skilled,
##                                         :       unemployed}: yes (2)
##                                         job = unskilled: no (3/1)
##
## SubTree [S1]
##
## employment_duration in {1 - 4 years,> 7 years,unemployed}: yes (10)
## employment_duration in {< 1 year,4 - 7 years}: no (4)
##
```

```
##
## Evaluation on training data (900 cases):
##
##      Decision Tree
##    ----------------
##    Size      Errors
##
##      56  133(14.8%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##     598    35    (a): class no
##      98   169    (b): class yes
##
##
##  Attribute usage:
##
##  100.00% checking_balance
##   54.22% credit_history
##   47.67% months_loan_duration
##   38.11% savings_balance
##   14.33% purpose
##   14.33% housing
##   12.56% employment_duration
##    9.00% job
##    8.67% other_credit
##    6.33% years_at_residence
##    2.22% percent_of_income
##    1.56% dependents
##    0.56% amount
##
##
## Time: 0.0 secs
```

Error de test

Una vez ajustado el modelo se evalúa su capacidad predictiva con el test de entrenamiento.

```
predicciones <- predict(arbol_clasificacion, newdata = test, type = "class")
table(predicho = predicciones, real = test$default)
```

```
##         real
## predicho no yes
##      no  59  19
##      yes  8  14
```

```
error_clas <- mean(predicciones != test$default)
paste("El error de clasificación es del:", 100 * error_clas, "%.",
      sum(predicciones == test$default),
      "clasificaciones correctas de un total de", length(predicciones))
```

```
## [1] "El error de clasificación es del: 27 %. 73 clasificaciones correctas de un
total de 100"
```

Si se especifica el argumento `type = "prob"`, en lugar de la clase predicha se devuelven las probabilidades de pertenecer a cada clase.

```
predicciones <- predict(arbol_clasificacion, newdata = test, type = "prob")
head(predicciones)
```

```
##          no       yes
## 1 0.8782163 0.1217837
## 2 0.8782163 0.1217837
## 3 0.8041553 0.1958447
## 4 0.3117222 0.6882778
## 5 0.8782163 0.1217837
## 6 0.8041553 0.1958447
```

El modelo de clasificación basado en un árbol simple apenas consigue clasificar correctamente las observaciones mejor de lo que cabría esperar si se asignara a todas a la clase mayoritaria. Como se ha descrito en secciones anteriores, los modelos de árboles simples pueden mejorarse empleando un *pruning* adecuado.

Boosting

C5.0 incorpora un método propio de *boosting* (semejante a *AdaBoost*) para generar modelos de tipo *ensemble* basados en árboles.

```
modelo_boost <- C5.0(default ~ ., data = train, trials = 10)
modelo_boost
```

```
##
## Call:
## C5.0.formula(formula = default ~ ., data = train, trials = 10)
##
## Classification Tree
## Number of samples: 900
## Number of predictors: 16
##
## Number of boosting iterations: 10
## Average tree size: 47.5
##
## Non-standard options: attempt to group attributes
```

```
predicciones <- predict(modelo_boost, newdata = test, type = "class")
table(predicho = predicciones, real = test$default)
```

```
##         real
## predicho no yes
##      no  62  13
##      yes  5  20
```

```
error_clas <- mean(predicciones != test$default)
paste("El error de clasificación es del:", 100 * error_clas, "%.",
      sum(predicciones == test$default),
      "clasificaciones correctas de un total de", length(predicciones))
```

```
## [1] "El error de clasificación es del: 18 %. 82 clasificaciones correctas de un
total de 100"
```

Los resultados muestran que, mediante *boosting*, se mejora considerablemente la capacidad predictiva del modelo.

Identificación de variables más influyentes

C5.0 incorpora dos formas de cuantificar la influencia de los predictores en el modelo final (simple o *ensemble*).

- *Usage*: porcentaje de observaciones de entrenamiento que caen en todos los nodos generados tras una división en el que ha participado el predictor. Por ejemplo, en el caso de un árbol simple, el predictor empleado en la primera decisión recibe automáticamente una importancia del 100%, ya que todas las observaciones de entrenamiento caen en uno de los dos nodos hijos generados. Otros predictores puede que participen con frecuencia en divisiones, pero si en los nodos hijos solo caen unas pocas observaciones, su importancia puede ser próxima a cero. En el caso de *boosting*, se promedia la importancia de cada predictor en todos los árboles que forman el *ensemble*.

- *Splits*: porcentaje de divisiones en las que participa cada predictor. No tiene en cuenta la importancia de la división.

Las dos medidas cuantifican aspectos muy diferentes, por lo que el *ranking* de importancia puede variar mucho dependiendo cual se utilice.

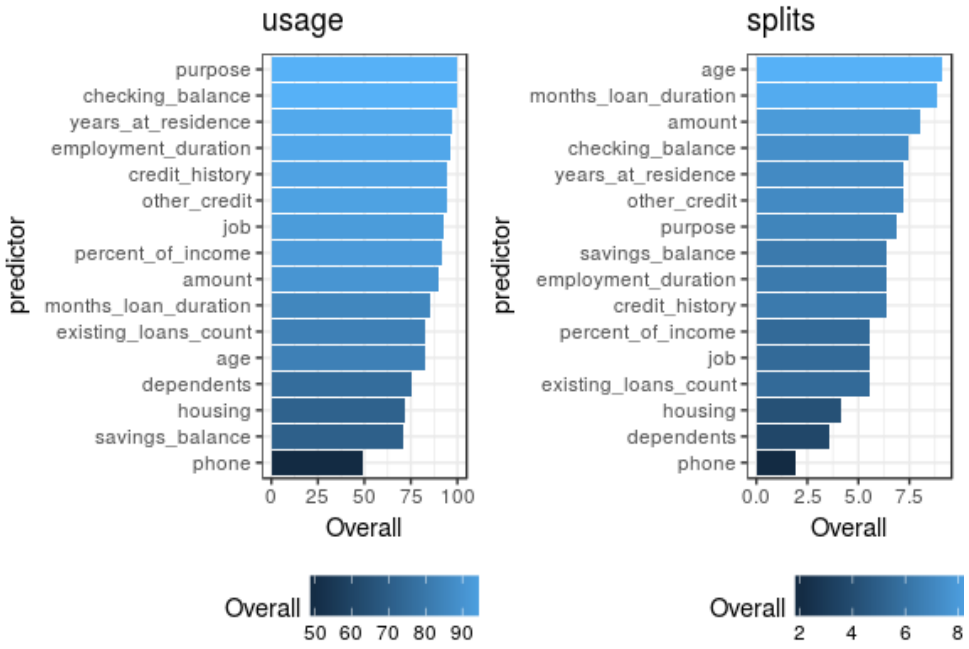
```
importancia_usage <- C5imp(modelo_boost, metric = "usage")
importancia_usage <- importancia_usage %>%
                     rownames_to_column(var = "predictor")
importancia_usage
```

```
##               predictor Overall
## 1      checking_balance  100.00
## 2               purpose  100.00
## 3    years_at_residence   97.11
## 4   employment_duration   96.67
## 5        credit_history   94.78
## 6          other_credit   94.67
## 7                   job   92.56
## 8     percent_of_income   92.11
## 9                amount   90.33
## 10 months_loan_duration   85.11
## 11                  age   82.78
## 12 existing_loans_count   82.78
## 13           dependents   75.78
## 14              housing   71.56
## 15      savings_balance   70.78
## 16                phone   49.22
```

```
importancia_splits <- C5imp(modelo_boost, metric = "splits")
importancia_splits <- importancia_splits %>%
                     rownames_to_column(var = "predictor")
importancia_splits
```

```
##               predictor  Overall
## 1                   age 9.116022
## 2  months_loan_duration 8.839779
## 3                amount 8.011050
## 4      checking_balance 7.458564
## 5          other_credit 7.182320
## 6    years_at_residence 7.182320
## 7               purpose 6.906077
## 8        credit_history 6.353591
## 9   employment_duration 6.353591
## 10      savings_balance 6.353591
## 11 existing_loans_count 5.524862
## 12                  job 5.524862
## 13    percent_of_income 5.524862
## 14              housing 4.143646
## 15           dependents 3.591160
## 16                phone 1.933702
```

```
p1 <- ggplot(data = importancia_usage, aes(x = reorder(predictor, Overall),
                                           y = Overall, fill = Overall)) +
    labs(x = "predictor", title = "usage") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")

p2 <- ggplot(data = importancia_splits, aes(x = reorder(predictor, Overall),
                                            y = Overall, fill = Overall)) +
    labs(x = "predictor", title = "splits") +
    geom_col() +
    coord_flip() +
    theme_bw() +
    theme(legend.position = "bottom")
ggarrange(p1, p2)
```



Definir el peso de errores

En muchos escenarios, las consecuencias de cometer un error de clasificación son distintas dependiendo el tipo que sean. Por ejemplo, en el ámbito biomédico, no es igual de grave confundir un tumor maligno con uno benigno que viceversa, ya que, en el primer caso, la vida del paciente puede estar en peligro. En el problema de los préstamos de este ejercicio, concederle un préstamo a alguien que no lo va a devolver, es mucho más costoso para el banco que no concedérselo a varios clientes que sí lo devolverían. El algoritmo C5.0 permite asignar diferentes penalizaciones a cada tipo de error, lo que fuerza al modelo (árbol) a minimizar determinados tipos de error.

El peso o penalización de cada error se pasa en forma de matriz al argumento `costs`. En la matriz se especifica cuánto de costoso es cada error respecto a los demás. En este problema, dado que la variable respuesta es binaria, se necesita una matriz 2x2.

```
dimensiones <- list(predicho = c("no", "yes"), real = c("no", "yes"))
dimensiones
```

```
## $predicho
## [1] "no"  "yes"
##
## $real
## [1] "no"  "yes"
```

A continuación, se asigna la penalización de cada tipo de error. Se considera que un préstamo no recuperado es 4 veces más costoso para el banco que un cliente perdido. Las predicciones correctas tienen no tienen coste.

```
coste_errores <- matrix(data = c(0, 1, 4, 0), nrow = 2, dimnames = dimensiones)
coste_errores
```

```
##         real
## predicho no yes
##      no   0   4
##      yes  1   0
```

Por último, se reajusta un modelo, esta vez empleando los pesos y se comparan los resultados frente al modelo que emplea el mismo peso para todos los errores.

```
# Mismo peso para todos los errores
arbol_clasificacion <- C5.0(default ~ ., data = train)
predicciones <- predict(arbol_clasificacion, newdata = test, type = "class")
table(predicho = predicciones, real = test$default)
```

```
##         real
## predicho no yes
##      no  59  19
##      yes  8  14
```

```
error_clas <- mean(predicciones != test$default)
paste("El error de clasificación es del:", 100 * error_clas,"%")
```

```
## [1] "El error de clasificación es del: 27 %"
```

```
# Distinto peso para cada tipo de error
arbol_clasificacion_2 <- C5.0(default ~ ., data = train, costs = coste_errores)
predicciones <- predict(arbol_clasificacion_2, newdata = test, type = "class")
table(predicho = predicciones, real = test$default)
```

```
##         real
## predicho no yes
##      no  37   7
##      yes 30  26
```

```
error_clas <- mean(predicciones != test$default)
paste("El error de clasificación es del:", 100 * error_clas,"%")
```

```
## [1] "El error de clasificación es del: 37 %"
```

Como consecuencia de las penalizaciones, el error total del modelo aumenta, pero el tipo de error más costoso, se reduce considerablemente. El número de clientes clasificados como "no fraude" pero que realmente sí defraudan se ha reducido de 19 a 7.

Winnowing

Winnowing es una estrategia incorporada en C5.0 que permite hacer selección de predictores relevantes. Consiste en emplear un algoritmo inicial que identifique los predictores que están relacionados con la variable respuesta para, posteriormente, ajustar el modelo final únicamente con estos predictores. El algoritmo seguido es el siguiente:

1. Se divide aleatoriamente el set de entrenamiento en dos mitades. Con una de ellas se ajusta un árbol (llamado *winnowing tree*) cuya finalidad es evaluar la utilidad de los predictores.

2. Los predictores que no se han utilizado en ninguna división del *winnowing tree* se identifican como no útiles.

3. La otra mitad de las observaciones de entrenamiento, no utilizadas en el ajuste del *winnowing tree*, se emplean para estimar el error del árbol. Se inicia un proceso

iterativo en el que se eliminan (de uno en uno) cada uno de los predictores que forman el *winnowing tree*. Si, como resultado de la eliminación, el error del árbol disminuye, se considera que el predictor solo aporta ruido y se identifica como no útil.

4. Una vez que todos los predictores han sido identificados como útiles o no útiles, se reajusta el *winnowing tree* utilizando únicamente los predictores útiles. Si al hacerlo, el error del árbol (calculado con la otra mitad del set de entrenamiento) no se reduce, el proceso de *winnowing* se descarta y no se excluye ningún predictor. Si por lo contrario, el error sí se reduce, se lanza el ajuste C5.0 convencional, empleando todas las observaciones de entrenamiento pero solo los predictores identificados como útiles.

Optimización de hiperparámetros

El paquete `C50` no incorpora métodos de validación cruzada para la identificación de los valores óptimos de los hiperparámetros. Sin embargo, es posible hacerlo recurriendo al paquete `caret`.

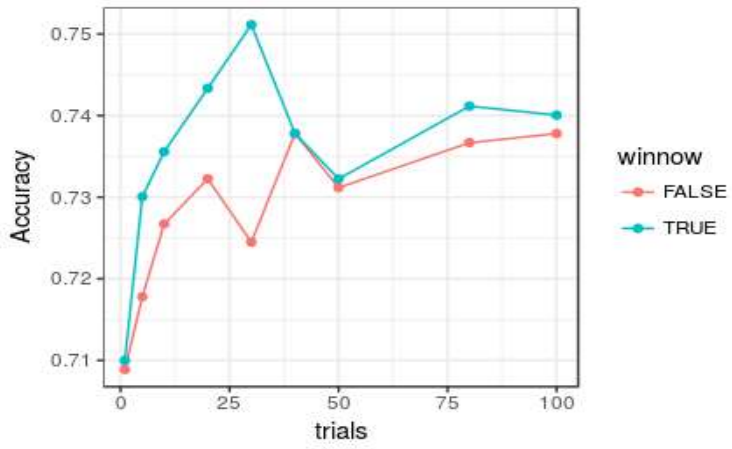
```
library(caret)

# Argumentos que pueden evaluarse con caret
getModelInfo(model = "C5.0")[[2]]$parameters
```

```
##   parameter     class                 label
## 1    trials   numeric # Boosting Iterations
## 2     model character            Model Type
## 3    winnow   logical                Winnow
## 4      cost   numeric                  Cost
```

```
# Estrategia de evaluación de los modelos
ctrl <- trainControl(method = "cv", number = 5)

# Hiperparámetros comparados
grid <- expand.grid(model = "tree",
                    trials = c(1, 5, 10, 20, 30, 40, 50, 80, 100),
                    winnow = c(FALSE, TRUE))

set.seed(123)
modelos <- train(default ~ ., data = train,
                 method = "C5.0",
                 trControl = ctrl,
                 tuneGrid = grid)
modelos$bestTune
```

```
##    trials model winnow
## 14     30  tree   TRUE
```

```
ggplot(data = modelos$results, aes(x = trials, y = Accuracy, color = winnow)) +
  geom_line() +   geom_point() +  theme_bw()
```



Bibliografía

Introduction to Statistical Learning, Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani

Applied Predictive Modeling Max Kuhn, Kjell Johnson

Top 10 algorithms in data mining by Xindong Wu, Vipin Kumar et al.

Machine Learning with R by Brett Lantz

CMU Statistics

Classification and regression trees. Wei-Yin Loh

https://www.youtube.com/watch?v=wPqtzj5VZus

Ensemble Learning to Improve Machine Learning Results

Generalized Boosted Models: A guide to the gbm package

https://en.wikipedia.org/wiki/Gradient_boosting

Classification Using C5.0