Junio 2018

Cristina Gil Martínez

ÁRBOLES DE DECISIÓN Y MÉTODOS DE ENSEMBLE

Apuntes personales sobre modelos basados en árboles de regresión y clasificación y métodos de ensemble (bagging, random forests y boosting).

CONTENIDO

INTRODUCCIÓN

Los métodos para regresión y clasificación basados en árboles de decisión estratifican o segmentan el espacio del predictor en un número simple de regiones, y para obtener las predicciones se suele usar la media o moda de las observaciones de entrenamiento en la región en la que cada observación a predecir pertenece. Los árboles de decisión son simples y fáciles de interpretar, pero pueden no resultar lo suficientemente competitivos frente a otros métodos de aprendizaje supervisado en cuanto a la precisión de predicción. Los métodos de *bagging*, *random forests* y *boosting* producen múltiples árboles que se combinan para mejorar la predicción, a expensar de una interpretación más complicada.

ÁRBOLES DE DECISIÓN

Los árboles de decisión pueden aplicarse tanto para problemas de regresión como de clasificación, y también pueden contener predictores tanto cuantitativos como cualitativos.

Árboles de regresión

Una de las ventajas de los árboles de decisión en comparación con otros métodos de regresión es su fácil interpretación y útil representación gráfica. A la hora de construir un árbol de regresión, dos son los pasos principales:

1. División del espacio del predictor (conjunto de valores posibles para X_1, X_2, ..., X_p en J regiones distintas y no solapantes o nodos internos, R_1, R_2, ..., R_j). Teóricamente, las regiones podrían tomar cualquier forma, aunque si se opta por regiones rectangulares de múltiples dimensiones se simplifica y facilita la interpretación del modelo final. El objetivo es encontrar las regiones R_1, ..., R_j que minimicen el *Residual Sum of Squares* (*RSS*)

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

   donde y_{Rj} es la media de la variable respuesta en la región *j*.

2. Para cada una de las observaciones pertenecientes a la región R_j se asigna la misma predicción, que es simplemente la media de la variable respuesta de las observaciones de entrenamiento en R_j.

DIVISIÓN BINARIA RECURSIVA

Desafortunadamente, resulta computacionalmente inviable considerar todas las posibles particiones del predictor, por lo que se opta por una **división binaria recursiva** (*recursive binary splitting*), similar al concepto de *stepwise selection*, consiguiendo también obtener buenos resultados. El proceso comienza en el punto más alto del árbol (donde todas las observaciones perteneces a una misma región), y de manera sucesiva se divide el espacio del predictor, donde cada división genera dos nuevas **ramas**. Se optará además por la mejor división en cada paso, sin tener en cuenta si dicha división mejorará el árbol en futuros pasos o divisiones.

Para llevar a cabo la división binaria recursiva, lo primero es seleccionar el **predictor X_j** y el **punto de corte *s*** de manera que al dividir el espacio del predictor en las regiones $\{X|X_j < s\}$ y $\{X|X_j \geq s\}$ se consiga la **máxima reducción del RSS**.

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

Es decir, se consideran todos los predictores X, ..., X_p y todos los posibles puntos de corte para cada uno de los predictores, eligiéndose el predictor *j* y punto de corte *s* que minimicen

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

correspondiente a la suma del *RSS* de la región 1 y el *RSS* de la región 2.

El proceso continúa, pero esta vez en lugar de dividir el espacio completo del predictor, se divide una de las dos regiones creadas en el paso anterior. El proceso continúa hasta que se alcance un criterio de parada, por ejemplo, hasta que ninguna región contenga más de *n* observaciones, hasta alcanzar un máximo de nodos terminales, etc. Una vez las regiones R_1, ..., R_j han sido creadas, predecimos la respuesta para una determinada observación de test como la media de las observaciones de entrenamiento en la región en la cual pertenece dicha observación.

PODADO DEL ÁRBOL (PRUNING)

El proceso de división binaria recursiva puede conseguir buenas predicciones con los datos de entrenamiento, ya que reduce el *training RSS*, lo que implica un sobreajuste a los datos (derivado de la facilidad de ramificación y posible complejidad del árbol resultante), reduciendo la capacidad predictiva para nuevos datos.

Una posible alternativa supone construir un árbol hasta el punto en el que la reducción del *RSS* debido a cada división supere un determinado límite (alto). Con esta estrategia obtendremos árboles más pequeños, con menos ramificaciones, con lo que conseguimos reducir la varianza y mejorar la interpretabilidad a expensas de una pequeña reducción del *bias*. Concretamente, la estrategia consistirá en construir un árbol grande T_0 y luego podarlo (***pruning***) para obtener un **sub-árbol** que consiga el menor *test error*, obtenido mediante validación cruzada o validación simple. Sin embargo, suele resultar muy costoso obtener el error de validación para cada posible sub-árbol debido al gran número de posibles sub-árboles, por lo que se opta por seleccionar un grupo pequeño de sub-árboles a considerar. Una manera de conseguir esto es mediante el ***cost complexity pruning***, al que corresponde el siguiente algoritmo:

1. Aplicar el método de **división binaria recursiva** para obtener un árbol grande (T_0) con los datos de entrenamiento, finalizando el proceso con el cumplimiento de una determinada norma de parada.
2. Aplicar ***cost complexity pruning*** a T_0 para obtener una secuencia de mejores sub-árboles, en función del parámetro de penalización o ***tuning parameter α***, que controla el equilibrio entre la complejidad del árbol y su ajuste a los datos de entrenamiento. Conforme α aumenta, más ramas se podan del árbol. A cada valor de α le corresponde un sub-árbol $T \subset T_0$ tal que se minimice

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

   donde |T| es el número de **nodos terminales** u **hojas** del árbol T, y R_m es el rectángulo correspondiente al nodo terminal *m*. Siendo α = 0, T será igual a T_0.
3. Usar *k-fold cross validation* para escoger α. Esto consiste en dividir las observaciones de entrenamiento en K grupos de manera que para cada *k* = 1, ..., *K*:
   a) Repetir pasos 1 y 2 en los *k* – 1 grupos.
   b) Evaluar el *mean squared error* MSE con el grupo k_i de observaciones, en función de α (conveniente graficarlo en función de |T|).

   Promediar los resultados para cada valor de α, escogiendo el valor que minimice el *test error*.
4. Seleccionar el sub-árbol del paso 2 correspondiente al valor escogido de α por validación cruzada.

Árboles de clasificación

Los árboles de clasificación son muy similares a los de regresión, con la diferencia de que se usan para predecir una variable respuesta cualitativa, asignando la predicción para cada observación como la clase

más común (moda) de observaciones de entrenamiento en la región o nodo terminal al que pertenece dicha observación de test.

Al igual que con los árboles de regresión, se aplica la división binaria recursiva para generar el árbol, pero el *RSS* no puede usarse como criterio para estas divisiones, sino otras alternativas como:

ERROR DE CLASIFICACIÓN

Fracción de observaciones de entrenamiento en una región o nodo que no pertenece a la clase más frecuente

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

donde $\hat{\mathbf{p}}_{mk}$ es la proporción de observaciones de entrenamiento en la región *m* que pertenecen a la clase *k*.

El error de clasificación no suele ser lo suficientemente sensible en medir la pureza de los nodos para el crear el árbol (es más recomendable para hacer el *pruning* posterior), por lo que en la práctica se opta por otras dos medidas, el índice de Gini o el *cross-entropy*. Aun así, cualquiera de los tres puede usarse para la poda del árbol. El error de clasificación es preferible si el objetivo es conseguir la máxima predicción en las predicciones del árbol podado final.

ÍNDICE DE GINI

El índice de Gini se define como una medida de la varianza total en el conjunto de las *K* clases, o pureza de los nodos

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

Este índice será pequeño cuando todos los $\hat{p}_{mk}$ estén próximos a 0 o 1. Un valor bajo indica que el nodo contiene predominantemente observaciones de una sola clase.

CROSS-ENTROPY

El *cross-entropy* viene dado por

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

Al igual que el índice de Gini, el valor del *cross-entropy* será pequeño cuando todos los $\hat{p}_{mk}$ estén próximos a 0 o 1. Un valor bajo también indica que el nodo contiene predominantemente observaciones de una sola clase.

Árboles de decisión vs modelos lineales

Mientras que la regresión lineal asume un modelo con la forma

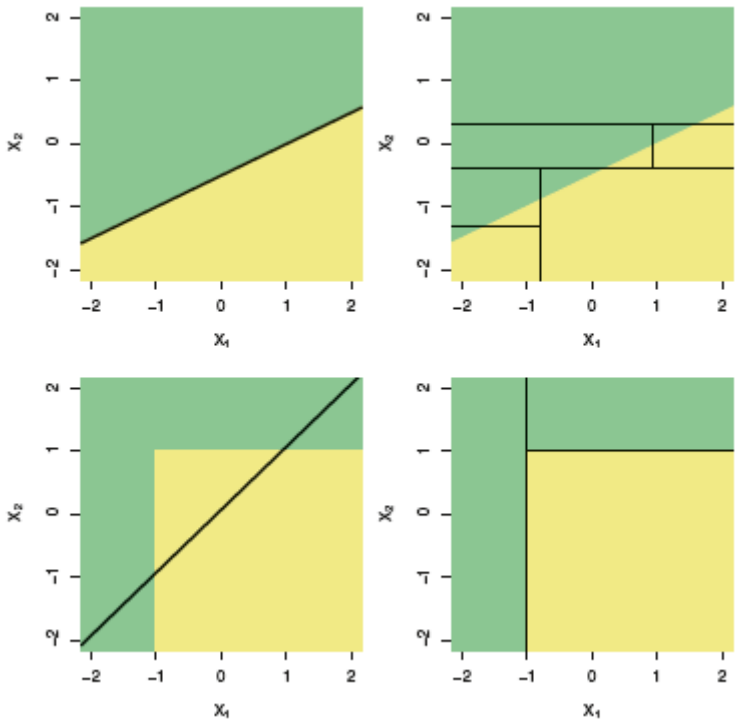
$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j$$

los árboles de regresión asumen un modelo con la forma

$$f(X) = \sum_{m=1}^{M} c_m \cdot 1_{(X \in R_m)}$$

donde R1, ..., R_M representa la partición del espacio del predictor.

El modelo que pueda dar mejores resultados dependerá del problema en cuestión: si la relación entre los predictores y la variable respuesta es aproximadamente lineal, entonces un modelo de regresión lineal dará buenos resultados.



(Imagen obtenida del libro ISLR)

Si de lo contrario existe una relación altamente no lineal y compleja, los árboles de decisión pueden superar los métodos más clásicos. En ambos casos, el rendimiento de unos modelos u otros puede medirse mediante la estimación del *test error* obtenido por validación cruzada o validación simple, aunque otras consideraciones pueden tenerse en cuenta, como por ejemplo el nivel de interpretabilidad o visualización.

Ventajas y desventajas de los árboles de decisión

- Fácil interpretación
- Representación gráfica muy intuitiva y posible aun cuando hay más de 3 predictores
- Fácil manejo de predictores cualitativos sin la necesidad de crear variables *dummy*
- No hay necesidad de asumir a priori ninguna relación entre las variables, pudiendo introducir relaciones muy complejas entre las mismas
- Seleccionan los predictores automáticamente
- Capacidad predictiva superable por otros métodos de regresión/clasificación. Sin embargo, mediante la agregación de varios árboles de decisión, métodos como bagging, *random forests* y *boosting*, la capacidad predictiva de los árboles puede mejorarse sustancialmente.

MÉTODOS DE *ENSEMBLE*

Los métodos de *bagging, random forests* y *boosting* nos permiten mejorar sustancialmente el rendimiento predictivo de modelos basados en árboles, aunque con el inconveniente de una considerable reducción en la facilidad de interpretación del modelo final. Estos métodos también se conocen como métodos de *ensemble* o **métodos combinados**, que son los que utilizan múltiples algoritmos de aprendizaje para obtener predicciones que mejoren las que se podrían obtener por medio de cualquiera de los algoritmos individuales. Son aplicables a muchos métodos de aprendizaje estadísticos (no solo árboles de decisión) para regresión o clasificación.

Bagging

Los árboles de decisión sufren de alta varianza, lo que significa que, si dividiéramos al azar los datos de entrenamiento en dos grupos y ajustáramos un árbol de decisión a cada mitad, los resultados que obtendríamos podrían ser bastante diferentes. Por el contrario, un procedimiento o método con baja

varianza dará resultados parecidos aun aplicándose sobre sets de datos distintos. El método de *bagging* o ***bootstrap aggregation*** es un procedimiento utilizado para reducir la varianza de un método de aprendizaje estadístico, usado muy frecuentemente con árboles de decisión.

REGRESIÓN:

Dado un set de *n* observaciones independientes Z_1, ..., Z_n, cada una con varianza σ^2, la varianza del promedio de las *Z* observaciones viene dado por σ^2/n. En otras palabras, promediar un grupo de observaciones reduce la varianza. De esta forma, un método para reducir la varianza y aumentar por lo tanto la precisión de las predicciones de un método de aprendizaje estadístico implica tomar muchos sets de observaciones de la población, ajustar un modelo de predicción por separado para cada set, y promediar las predicciones resultantes. De manera más formal, calcularíamos $\hat{f}^1(x), \hat{f}^2(x), \ldots, \hat{f}^B(x)$ usando B sets de entrenamiento, promediándolos obteniendo

$$\hat{f}_{\mathrm{avg}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^b(x)$$

En la práctica, no se suele tener acceso a múltiples sets de observaciones, por lo que podemos optar en este caso por el *bootstrap*, tomando repetidamente muestras (**b*) de nuestro único set de datos de entrenamiento

$$\hat{f}_{\mathrm{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Por tanto, la aplicación del *bagging* a árboles de regresión consiste en crear *B* árboles de regresión usando los *B* sets de entrenamiento generados por *bootstrapping*, promediando finalmente las predicciones resultantes. Estos árboles pueden crecer bastante ya que apenas se aplican restricciones, además de que no son podados. De esta manera cada árbol individual tiene alta varianza y poco *bias*, pero promediando los B árboles se contrarresta la varianza.

CLASIFICACIÓN:

En un problema de clasificación, existen varias posibilidades para aplicar *bagging*, pero la más simple es la siguiente: dada una observación de test, podemos obtener la clase predicha por cada uno de los *B* árboles, y escoger como predicción final para dicha observación la clase más común de entre las *B* predicciones (predicción de cada árbol).

El número de árboles (*B*) a crear no es un parámetro crítico a la hora de aplicar *bagging*. Ajustar un gran número de árboles no aumentará el riesgo de *overfitting*, por lo que usaremos un número lo suficientemente alto como para alcanzar la estabilización en la reducción del *test error*.

ESTIMACIÓN DEL ERROR *OUT-OF-BAG*

Una manera directa de estimar el test error de un modelo al que se aplica *bagging* sin necesidad de aplicar la validación cruzada o simple: en promedio, cada árbol generado por *bootstrapping* usa en torno a 2/3 de las observaciones. El 1/3 restante de observaciones no usadas para ajustar cada árbol se conocen como *out-of-bag* (***OOB***). Por lo tanto, podemos obtener la predicción para la *i*-ésima observación de test usando cada uno de los árboles en los cuales dicha observación sea OOB. Esto resultará en *B*/3 predicciones para cada observación, por lo que, para obtener un solo valor, se opta por una medida distinta según nos encontremos en un problema de regresión o clasificación:

REGRESIÓN:

Se promedian los *B*/3 valores. La estimación del *test error* se corresponde al OOB MSE.

CLASIFICACIÓN:

Se escoge la clase mayoritaria, o se clasifica en base al promedio de los valores de probabilidad $P(Y = y \mid X)$. La estimación del *test error* se corresponde con el error de clasificación.

Con un número suficientemente elevado de árboles (*B*) el error *OOB* puede llegar a ser equivalente al error de validación *leave-one-out*. Además, el método basado en OOB para estimar el test error resulta conveniente cuando se aplica *bagging* en sets de datos grandes, para los cuales aplicar la validación cruzada sería computacionalmente muy costoso.

IMPORTANCIA DE LAS VARIABLES

Cuando se combinan múltiples árboles mediante *bagging*, no es posible representar gráficamente el modelo resultante mediante un árbol, y no es identificable de manera inmediata qué variables son las más importantes. Por tanto, *bagging* mejora la predicción del modelo a expensas de la pérdida de interpretabilidad. Aun así, un modo de poder identificar qué predictores tienen mayor importancia es

REGRESIÓN:

Cantidad total reducida del RSS como resultado de las divisiones sobre cada predictor, promediada sobre todos los *B* árboles. Un predictor importante será el que consiga una reducción promedio mayor del RSS.

CLASIFICACIÓN:

Mismo proceso que en el caso de árboles de regresión, pero empleando el índice de Gini.

Random forests

Supóngase que se cuenta con un set de datos cuenta con un predictor muy importante o influyente que destaca sobre el resto. En este caso, todos o casi todos los árboles generados por *bagging* usarán este predictor en la primera ramificación, por lo que acabarán siendo similares unos a otros, y las predicciones entre ellos estarán altamente correlacionadas. En este escenario, la aplicación de *bagging* promediando valores correlacionados no consigue una reducción sustancial de la varianza con respecto a un solo árbol.

El método de *random forests* proporciona una mejora a los árboles combinados por *bagging* en cuanto a que los **decorrelaciona**, teniendo en cuenta solo un subgrupo de predictores en cada división. Al igual que en el *bagging*, se construyen un número de árboles de decisión a partir de pseudo-muestras generadas por *bootstrapping*. Esta vez, se escogen de entre todos los *p* predictores una muestra aleatoria de *m* predictores como candidatos antes de cada división, generalmente $m \approx \sqrt{p}$ (si $m = p$, *bagging* y *random forests* darían resultados equivalentes). Solo se aplica la división a uno de los *m* predictores. Esto hace que, de media, $(p - m) / p$ divisiones no tengan en cuenta el predictor más influyente, dando más oportunidades al resto.

Usar un número pequeño de *m* para aplicar *random forests* puede resultar útil cuando contamos con un gran número de predictores correlacionados.

Al igual que con *bagging*, aumentar el número de pseudo-árboles no incrementará el riesgo de *overfitting*, por lo que en la práctica usamos un valor lo suficientemente alto para conseguir una estabilización del *test error*.

Boosting

Boosting funciona de manera parecida al *bagging* en cuanto a que combina un gran número de árboles, a excepción de que **los árboles se construyen de manera secuencial**: cada árbol se genera usando información, concretamente los residuos, de árboles previamente generados, en lugar de utilizar la variable

respuesta (por ello suelen ser suficientes árboles más pequeños, en lugar de un gran árbol que pueda sobreajustarse a los datos). Otra diferencia es que *boosting* no utiliza remuestreo por *bootstrapping*, sino que cada árbol se genera utilizando una versión modificada del set de datos original. Los parámetros principales del algoritmo para generar árboles de regresión son:

- **Número de árboles** (B). A diferencia del *bagging* y *random forests*, *boosting* puede sobreajustarse a los datos si el número de árboles es muy alto. B se selecciona por validación cruzada.
- **Número de divisiones** (*d*) en cada árbol, que controla el nivel de complejidad. Un valor de $d = 1$ (cada árbol contiene una única división, es decir, un único predictor) suele dar buenos resultados.
- **Parámetro de penalización** (λ), que controla el ritmo con el que *boosting* aprende. Valores comunes para este parámetro suelen ser 0,01 o 0,001, aunque la decisión depende del problema en cuestión. Por ejemplo, un valor muy pequeño de λ puede requerir un número elevado de árboles para conseguir buenos resultados.

siendo el algoritmo el siguiente:

1. Ajustar $\hat{f}(x) = 0$ y $r_i = y_i$ para cada observación en el set de datos de entrenamiento.

2. Para $b = 1, 2, ..., B$, repetir:

a) Ajustar un árbol $\hat{f}^b$ con *d* divisiones ($d + 1$ nodos terminales) sobre los datos de entrenamiento (X, r).

b) Actualizar $\hat{f}$ añadiendo una versión reducida del nuevo árbol:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

c) Actualizar los residuos,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. Obtener el modelo final por *boosting*

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

El proceso es algo más complejo para árboles de clasificación.

EJEMPLOS EN R

Árboles simples: library(tree)

- **tree()** -> Ajusta árboles de decisión mediante partición binaria recursiva.
- **tree.control()** -> Selección de parámetros para el árbol. Se aplica al argumento *control* de la función *tree()*.

Pruning:

- **cv.tree()** -> Lleva a cabo *k-fold cross validation* para estimar la *deviance* o error de clasificación en función del parámetro *cost-complexity k*. Especificar argumento `FUN = prune.tree` para árboles de regresión y `FUN = prune.misclass` para árboles de clasificación.
- **prune.tree()** -> Genera sub-árboles a partir del árbol original mediante *cost- complexity pruning*.

Bagging y random forests: library(RandomForest)

- **randomForest()** -> Implementación del algoritmo de *bagging* (m = p) y *random forests* (m < p), para regresión y clasificación. Con el argumento *mtry* se especifica el número de variables, y con ello el método a seguir.
- **importance()** -> Calcula dos medidas (%IncMSE y IncNodePurity) que identifican la importancia de los predictores del modelo generado por la función *ranfomForest()*.
- **varImpPlot()** -> Diagrama de representación de la importancia de las variables.

Boosting: library(gbm)

- **gbm()** -> Implementación del algoritmo de *boosting*. Especificar argumento *distribution = "gaussian"* para árboles de regresión, y *distribution = "Bernoulli"* para árboles de clasificación. El argumento *interaction.depth* limita la profundidad (complejidad) de cada árbol. El número de árboles *ntrees* son por defecto 5000, y el parámetro de penalización $\lambda = 0{,}01$.

Ejemplo de regresión

En este ejemplo utilizaremos el set de datos `Hitters` del paquete `ISLR`, que contiene información sobre jugadores de beisbol de la *Major League Baseball (MLB)* de las temporadas 1986-1987. Aplicando todos los métodos descritos en la sección teórica, intentaremos predecir los salarios de los jugadores.

NOTA: Para ajustar un árbol de regresión, la variable respuesta ha de ser de tipo `numeric`.

```
library(ISLR)
str(Hitters)
```

```
 ## 'data.frame':    322 obs. of  20 variables:
##  $ AtBat    : int  293 315 479 496 321 594 185 298 323 401 ...
##  $ Hits     : int  66 81 130 141 87 169 37 73 81 92 ...
##  $ HmRun    : int  1 7 18 20 10 4 1 0 6 17 ...
##  $ Runs     : int  30 24 66 65 39 74 23 24 26 49 ...
##  $ RBI      : int  29 38 72 78 42 51 8 24 32 66 ...
##  $ Walks    : int  14 39 76 37 30 35 21 7 8 65 ...
##  $ Years    : int  1 14 3 11 2 11 2 3 2 13 ...
##  $ CAtBat   : int  293 3449 1624 5628 396 4408 214 509 341 5206 ...
##  $ CHits    : int  66 835 457 1575 101 1133 42 108 86 1332 ...
##  $ CHmRun   : int  1 69 63 225 12 19 1 0 6 253 ...
##  $ CRuns    : int  30 321 224 828 48 501 30 41 32 784 ...
##  $ CRBI     : int  29 414 266 838 46 336 9 37 34 890 ...
##  $ CWalks   : int  14 375 263 354 33 194 24 12 8 866 ...
##  $ League   : Factor w/ 2 levels "A","N": 1 2 1 2 2 1 2 1 2 1 ...
##  $ Division : Factor w/ 2 levels "E","W": 1 2 2 1 1 2 1 2 2 1 ...
##  $ PutOuts  : int  446 632 880 200 805 282 76 121 143 0 ...
##  $ Assists  : int  33 43 82 11 40 421 127 283 290 0 ...
##  $ Errors   : int  20 10 14 3 4 25 7 9 19 0 ...
##  $ Salary   : num  NA 475 480 500 91.5 750 70 100 75 1100 ...
##  $ NewLeague: Factor w/ 2 levels "A","N": 1 2 1 2 2 1 1 1 2 1 ...
```

Primero eliminaremos las observaciones para la que no hay información sobre la variable respuesta *Salary*.

(Hay otras técnicas para manejar los valores ausentes, pero para este ejemplo se decide eliminarlos)

```
 sum(is.na(Hitters$Salary))
```

```
 ## [1] 59
```

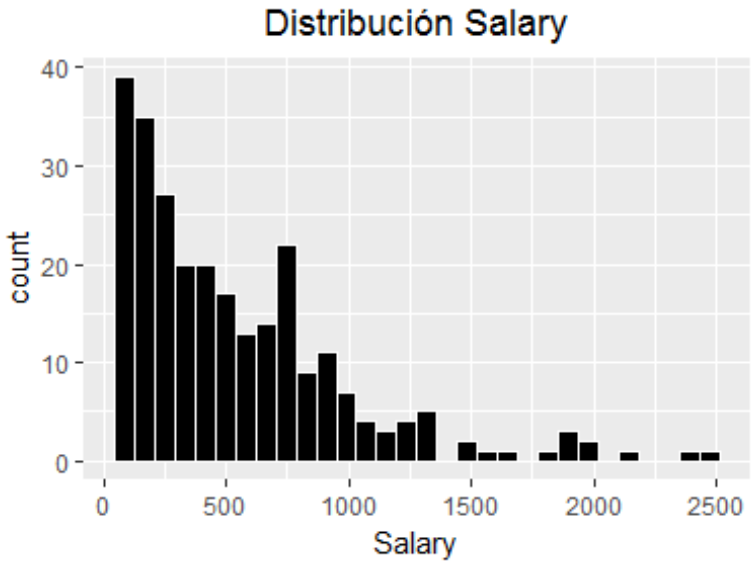
```
 # Omisión de NAs
datos.Hitters <- na.omit(Hitters)
sum(is.na(datos.Hitters$Salary))
```

```
 ## [1] 0
```

```
 dim(datos.Hitters)
```

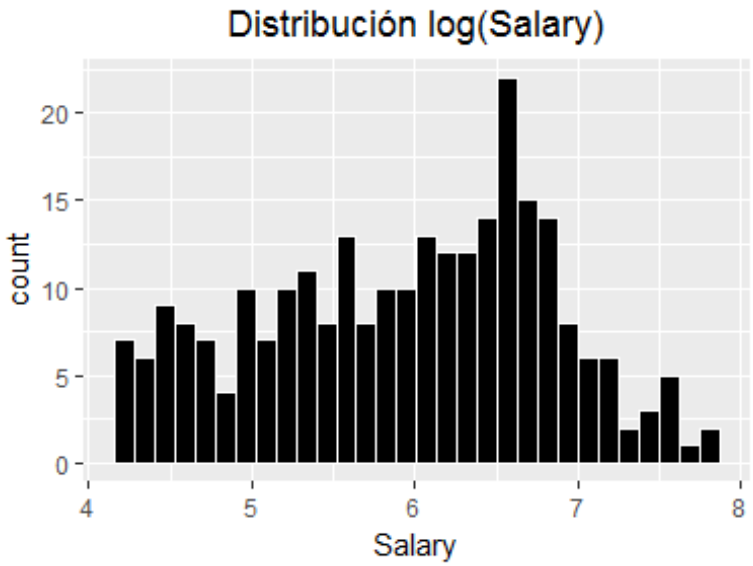
```
 ## [1] 263  20
```

```
 # Distribución variable respuesta
library(ggplot2)



ggplot(data = datos.Hitters, aes(x = Salary)) +
geom_histogram(color = "white", fill = "black") +
labs(title = "Distribución Salary") +
theme(plot.title = element_text(hjust = 0.5))
```



La variable respuesta posee una distribución sesgada. Aplicaremos una transformación logarítmica para hacer su distribución más normal.

```
datos.Hitters$Salary <- log(datos.Hitters$Salary)

ggplot(data = datos.Hitters, aes(x = Salary)) +
geom_histogram(color = "white", fill = "black") +
labs(title = "Distribución log(Salary)") +
theme(plot.title = element_text(hjust = 0.5))
```



Antes de pasar a generar los modelos, dividimos el set de datos en un grupo de entrenamiento (para el ajuste de los modelos) y otro de test (para la evaluación de los mismos). Esta división dependerá de la cantidad de observaciones con las que contemos y la seguridad con la que queramos obtener la estimación del *test error*.

```
# Índice observaciones de entrenamiento
train <- 1:200
# Datos entrenamiento
datos.H.train <- datos.Hitters[train, ]
# Datos test
datos.H.test <- datos.Hitters[-train, ]
```

ÁRBOL DE REGRESIÓN SIMPLE

AJUSTE DEL MODELO

Por defecto, las condiciones de *stop* que regulan el crecimiento del árbol, y que hacen que no todas las variables incluidas en la fórmula tengan por qué ser utilizadas en el árbol final, son:

- `mincut = 5`: Cantidad ponderada del nº de observaciones mínimas de al menos uno de los nodos hijos resultantes para permitir la división.
- `minsize = 10`: Cantidad ponderada del nº de observaciones mínimas en un nodo para permitir su división.
- `mindev = 0,01`: *Deviance* mínima para proceder con una división. Para obtener un árbol más pequeño, reducir el valor.

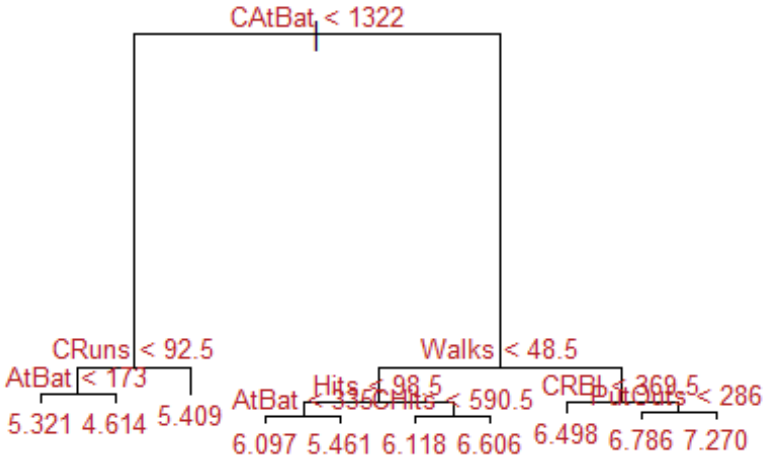
Estos valores pueden ser cambiados por el usuario. A continuación, se muestra el ajuste, pero usando los valores por defecto:

```
library(tree)
# Selección de parámetros para el árbol
setup <- tree.control(nobs = nrow(datos.H.train),
                      mincut = 5,
                      minsize = 10,
                      mindev = 0.01)
# Modelo árbol de regresión
modelo.arbol.r <- tree(Salary ~ ., data = datos.H.train,
                       split = "deviance",
                       control = setup)

summary(modelo.arbol.r)

##
## Regression tree:
## tree(formula = Salary ~ ., data = datos.H.train, control = setup,
```

```
##     split = "deviance")
## Variables actually used in tree construction:
## [1] "CAtBat"  "CRuns"   "AtBat"   "Walks"   "Hits"    "CHits"   "CRBI"
## [8] "PutOuts"
## Number of terminal nodes:  10
## Residual mean deviance:  0.1665 = 31.64 / 190
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.003000 -0.221300  0.009429  0.000000  0.246800  2.342000
```

Según la información devuelta, del conjunto de las 19 variables, el modelo ha utilizado para generar el árbol solo 8 (nodos internos): *CAtBat, CRuns, AtBat, Walks, Hits, CHits, CRBI* y *PutOuts*. Con estas variables el número de nodos terminales es 10. El ***residual mean deviance*** se corresponde con el *training* RSS (31,64) dividido entre el nº de observaciones – nº de nodos terminales (200 – 10 = 190). Cuanto menor es este valor, mejor se ajusta el modelo a los datos de entrenamiento.

```
plot(modelo.arbol.r, type = "proportional")
# pretty = 0 incluye los nombres de los niveles para las variables cualitativas, e
n lugar de mostrar solo una letra
text(modelo.arbol.r, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```



La variable que se encuentra en lo más alto del árbol (primer nodo), y por tanto, la más importante, es *CAtBat*. Esta primera rama diferencia a los jugadores según el número de bateos a lo largo de la trayectoria profesional del jugador. En este caso, los que superan 1322 bateos tienen un salario más alto. De los que tienen menor salario, la variable *CRuns* es más determinante como segunda variable más importante que

en el grupo de jugadores que ganan más, donde *Walks* juega un papel más determinante a este nivel. Además, la altura de las ramas indica la efectividad de la división en cuanto a reducción del RSS.

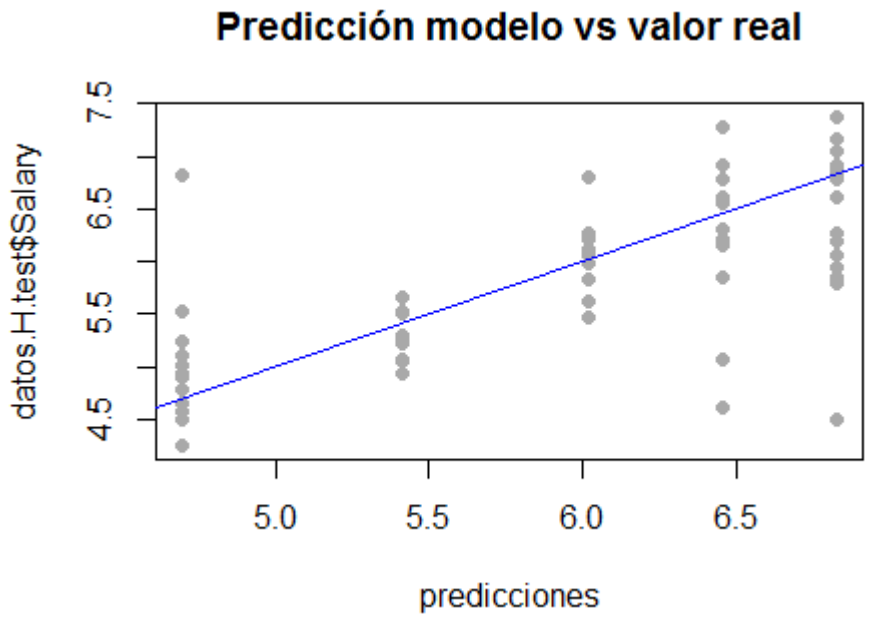
Si además inspeccionamos el objeto `tree`, R nos muestra información correspondiente a cada rama del árbol: criterio de división, número de observaciones en cada rama antes de la división (n), el error o *deviance* (*RSS*), y la media de la predicción final para cada rama (yval). Con asteriscos se indican los nodos terminales.

```
modelo.arbol.r

## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 200 166.4000 5.940
##    2) CAtBat < 1322 70  23.3000 4.971
##      4) CRuns < 92.5 43  12.6000 4.696
##        8) AtBat < 173 5   7.2760 5.321 *
##        9) AtBat > 173 38   3.1220 4.614 *
##      5) CRuns > 92.5 27   2.2610 5.409 *
##    3) CAtBat > 1322 130  42.0400 6.462
##      6) Walks < 48.5 80  20.2600 6.232
##       12) Hits < 98.5 41   9.8330 6.019
##         24) AtBat < 335 36   6.8960 6.097 *
##         25) AtBat > 335 5   1.1580 5.461 *
##       13) Hits > 98.5 39   6.6220 6.456
##         26) CHits < 590.5 12   1.8470 6.118 *
##         27) CHits > 590.5 27   2.7950 6.606 *
##      7) Walks > 48.5 50  10.8100 6.829
##       14) CRBI < 369.5 16   0.9873 6.498 *
##       15) CRBI > 369.5 34   7.2280 6.985
##         30) PutOuts < 286 20   2.6210 6.786 *
##         31) PutOuts > 286 14   2.6770 7.270 *
```

Por ejemplo, el primer nodo contiene todas las observaciones (200), con un *RSS* de 166,4 y una predicción para *Salary* de 1000$ x $\exp^{5,94}$ = 379934,9$ (o 5,94 si nos referimos al promedio del *log Salary*). La primera división (R_1) está definida por la condición *CAtBat* < 1322, habiendo 70 jugadores que cumplen esta condición (sub-árbol por debajo de este nodo). El *RSS* para este grupo es de 23,3, siendo el salario medio de estos jugadores de 1000$ x $\exp^{4,971}$ = 144171$.

EVALUACIÓN DEL MODELO

```
pred.arbol <- predict(object = modelo.arbol.r, newdata = datos.H.test)
plot(x = pred.arbol, y = datos.H.test$Salary,
     main = "Predicción modelo vs valor real",
     xlab = "predicciones",
     col = "darkgrey", pch = 19)
abline(a = 0, b = 1, col = "blue")
```




```
test.MSE.arbol.r <- mean((pred.arbol - datos.H.test$Salary)^2)


test.MSE.arbol.r

## [1] 0.3116231
```

PRUNING

AJUSTE DEL MODELO

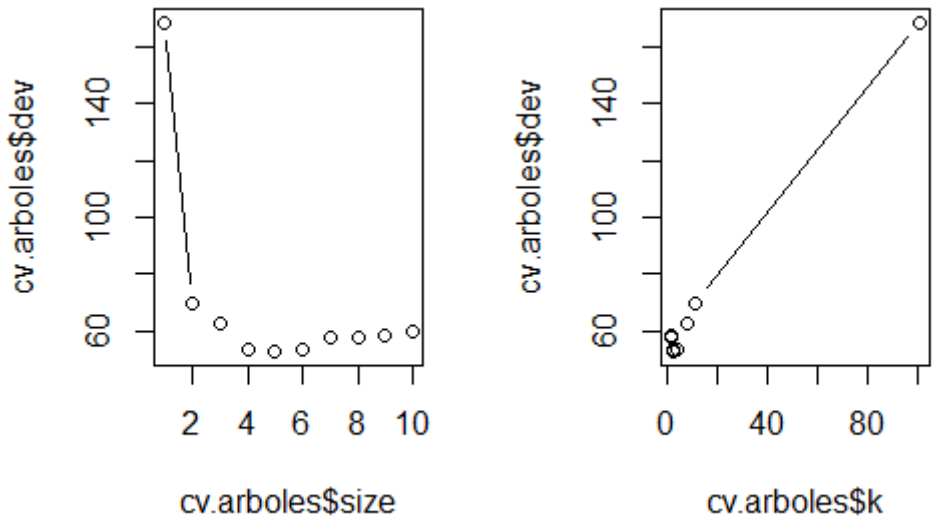
El siguiente paso es considerar si la reducción de la varianza y complejidad mediante la poda del árbol (mediante *cost complexity pruning*) puede mejorar las predicciones del modelo. Para ello hacemos uso de la función `cv.tree()` para encontrar le parámetro de penalización α óptimo por validación cruzada. Por defecto la función utiliza la *deviance* o *RSS* (`FUN = prune.tree`) como método para guiar el proceso de validación cruzada y poda (puede cambiarse con el argumento `FUN =`).

```
set.seed(3)
# 10-fold cross validation (con 10 folds se dividen de manera exacta las observaci
ones: 200 es múltiplo exacto de 10)
cv.arboles <- cv.tree(modelo.arbol.r, K = 10, FUN = prune.tree)
cv.arboles

## $size
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```
##
## $dev
##  [1]  56.96620  58.50210  58.09298  58.09298  53.82112  52.91235  53.65221
##  [8]  65.10143  71.37031 167.33873
##
## $k
##  [1]       -Inf   1.779047   1.930350   1.980293   2.206202   2.590201
##  [7]   3.801387   8.432809  10.977019 101.065841
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Entre la información devuelta por el objeto `cv.tree()` se encuentra: número de nodos terminales de cada árbol considerado (*size*), con su correspondiente error de validación (*dev*) y parámetro de penalización (*k*). Con esta información, podemos representar el error de validación (*deviance*) en función del tamaño del árbol y el parámetro de penalización para identificar el mejor sub-árbol:

```
par(mfrow = c(1, 2))
# Cost complexity pruning
plot(cv.arboles$size, cv.arboles$dev, xlab = "nodos terminales", ylab = "RSS",
     type = "b", pch = 19)
plot(cv.arboles$k, cv.arboles$dev, xlab = "alpha", ylab = "RSS", type = "b")
```

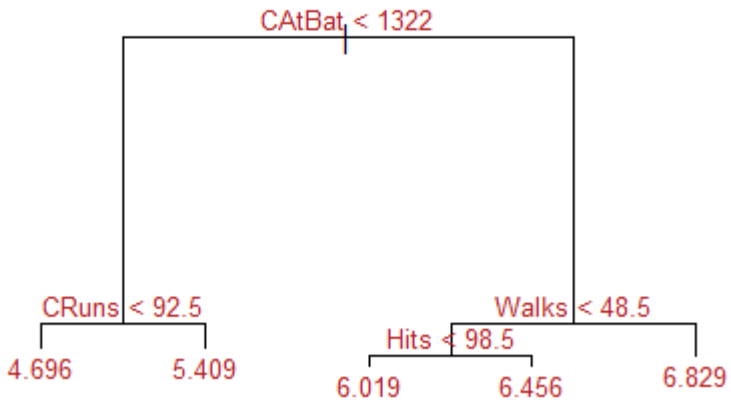



```
cv.arboles$size[which.min(cv.arboles$dev)]

## [1] 5
```

El número de nodos terminales que consigue minimizar el error de validación es 5, por lo que se reducen a la mitad con respecto al árbol inicial. El valor de α correspondiente es 2,20.

NOTA: Puede haber casos en los que la poda no consiga mejorar el modelo, escogiéndose por validación cruzada el mayor tamaño de modelo.

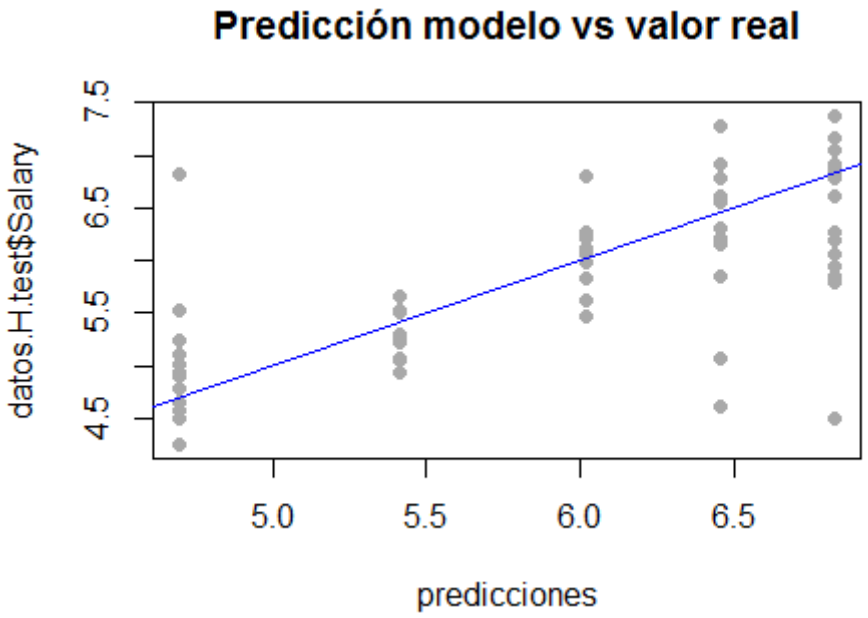
```
# Poda del arbol
modelo.arbol.r.p <- prune.tree(tree = modelo.arbol.r, best = 5)

par(mfrow = c(1,1))
plot(x = modelo.arbol.r.p, type = "proportional")
text(modelo.arbol.r.p, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```



EVALUACIÓN DEL MODELO

Hacemos uso del set de datos de test para evaluar la capacidad predictiva del sub-árbol obtenido.

```
pred.arbol <- predict(object = modelo.arbol.r.p, newdata = datos.H.test)
plot(x = pred.arbol, y = datos.H.test$Salary,
     main = "Predicción modelo pruned vs valor real",
     xlab = "predicciones",
     col = "darkgrey",
     pch = 19)
abline(a = 0, b = 1, col = "blue")
```




```
test.MSE.p <- mean((pred.arbol - datos.H.test$Salary)^2)

test.MSE.p

## [1] 0.3983201
```

El *test MSE* correspondiente a este árbol de regresión es de 0,398 unidades, lo cual equivale a decir que las predicciones del modelo se alejan de los valores reales en √0,3983201 = 0,631 unidades, en promedio. Si recordamos que la variable respuesta está transformada logarítmicamente, esto equivale a decir que las predicciones se desvían de los valores reales en ($exp^{0,631}$ x 1000$ = 1879,48$) en promedio.

BAGGING

AJUSTE DEL MODELO

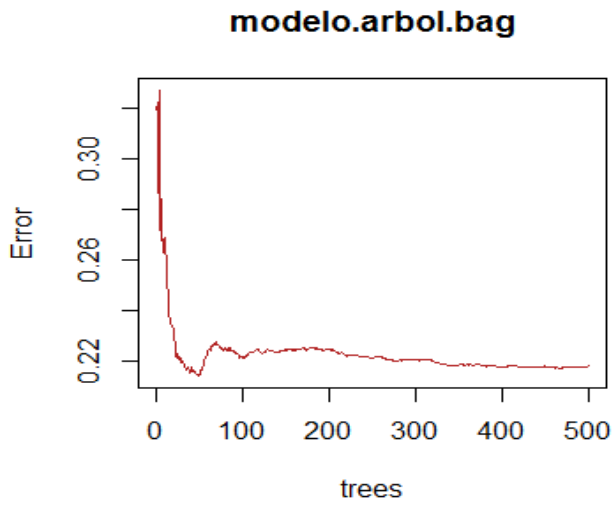
Para intentar mejorar la capacidad predictiva del árbol obtenido en el paso inicial, aplicaremos el método de *bagging*. El set de datos cuenta con 20 variables, 19 predictores y 1 variable respuesta. Por tanto, para aplicar *bagging* todos los predictores han de considerarse en cada partición. El número de pseudo-árboles

que se generan por defecto son 500 (no se recomienda usar un número pequeño, para asegurar que todas o casi todas las observaciones tengan participación):

```
library(randomForest)


# Bagging
set.seed(1)
modelo.arbol.bag <- randomForest(Salary ~ ., data = datos.H.train,
                                 mtry = 19,
                                 importance = TRUE,
                                 ntree = 500)
# con importance = T se evalúa la importancia de los predictores en el proceso.
modelo.arbol.bag

##
## Call:
##  randomForest(formula = Salary ~ ., data = datos.H.train, mtry = 19,      impor
tance = TRUE, ntree = 500)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 19
##
##           Mean of squared residuals: 0.2178554
##                     % Var explained: 73.82
```

El porcentaje de la varianza explicada por el modelo aplicando *bagging* es del 73,82%. El ***Mean of squared resisuals*** se corresponde con el *out-of-bag-MSE*, que puede entenderse como un *test error*. Su evolución respecto al número de árboles puede representarse de la siguiente manera:

```
plot(modelo.arbol.bag, col = "firebrick")
```



El error se estabiliza al alcanzar los 100 árboles aproximadamente.

Identificación de los predictores más importantes

Al combinar múltiples árboles perdemos la posibilidad de obtener una representación gráfica como en el caso de un árbol único. Sin embargo, podemos utilizar la función `importance()` para obtener la importancia de cada predictor:

```
importance(modelo.arbol.bag)
```

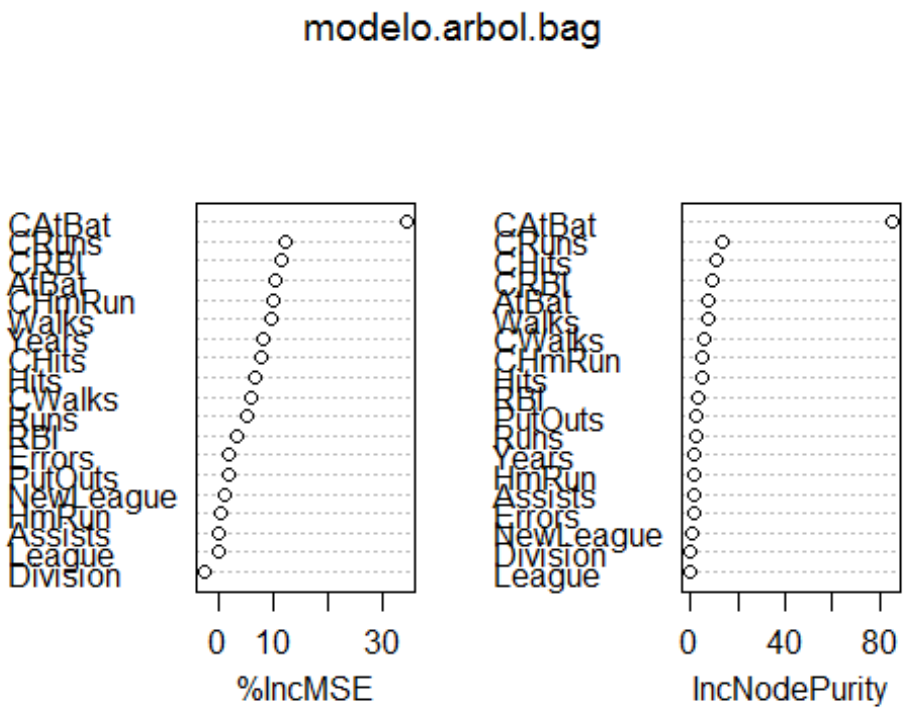
```
##               %IncMSE IncNodePurity
## AtBat     10.36037677    7.59899089
## Hits       6.69476752    4.61244229
## HmRun      0.45962166    1.67420019
## Runs       5.19809979    2.58312025
## RBI        3.21835382    3.02919843
## Walks      9.49509870    7.25161868
## Years      7.94438982    1.78466935
## CAtBat    34.61473697   85.80782159
## CHits      7.80595151   11.13937572
## CHmRun     9.87901114    4.95951018
## CRuns     12.33865863   13.49631939
## CRBI      11.44789911    9.52385064
## CWalks     5.96869309    5.70105949
## League    -0.25504777    0.09686047
## Division  -2.56240113    0.15964731
## PutOuts    1.72924797    2.69742165
## Assists   -0.02584561    1.36241940
## Errors     1.83362063    1.16664726
## NewLeague  0.99642182    0.21617428
```

Para cada predictor se devuelven dos valores:

- *%IncMSE*: disminución media de la precisión de las predicciones sobre las muestras OOB cuando la variable dada se excluye del modelo.
- *IncNodePurity*: medida de la disminución total de impureza de los nodos (medida por el *training RSS*) que resulta de la división de la variable dada.

Los predictores más importantes se corresponderán a aquellos con mayor *%IncMSE* y *IncNodePurity*.

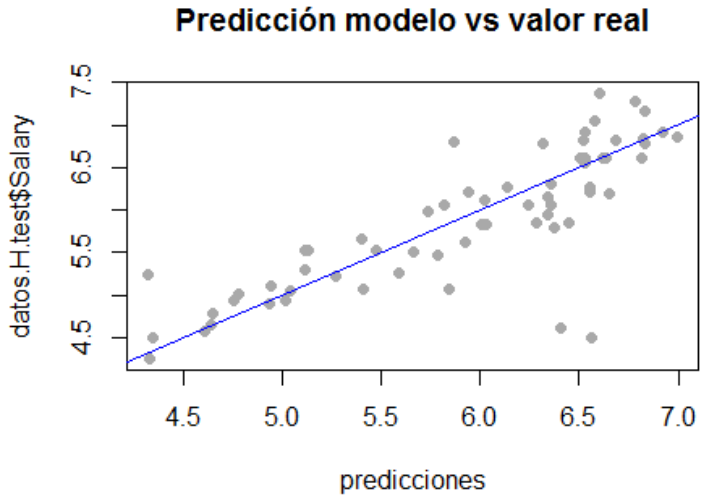
```
varImpPlot(modelo.arbol.bag)
```



En este ejemplo la variable más importante es *CAtBat* (al igual que en el árbol simple).

EVALUACIÓN DEL MODELO

```
pred.arbol <- predict(object = modelo.arbol.bag, newdata = datos.H.test)
plot(x = pred.arbol, y = datos.H.test$Salary,
     main = "Predicción modelo vs valor real",
     xlab = "predicciones",
     col = "darkgrey",
     pch = 19)
abline(a = 0, b = 1, col = "blue")
```



```
test.MSE.bag <- mean((pred.arbol - datos.H.test$Salary)^2)

test.MSE.bag
## [1] 0.2301184
```

Se ha conseguido reducir el test error de 0,398 unidades con el método de *pruning* a 0,23 ($exp^{v0,23}$ x 1000$ = 1615,4$) con el método de *bagging*.

RANDOM FORESTS

El método de *random forests* se aplica de la misma forma que anteriormente con *bagging*, pero especificando en el argumento *mtry* un número menor de predictores respecto al total (m < p). Con este método, la función *randomForest()* usa por defecto *p*/3 variables para árboles de regresión, y √*p* con árboles de clasificación. En lugar de utilizar *p*/3 en este caso, es mejor determinar el valor óptimo en función de un error de validación. Con el paquete *caret* podemos llevar a cabo esta búsqueda, aunque existen otras opciones varias. Contamos con *p* = 19 variables, por lo que como máximo tenemos que evaluar *p*-1 para que el proceso sea considerado *random forest*.

AJUSTE DEL MODELO

Optimización de hiperparámetros

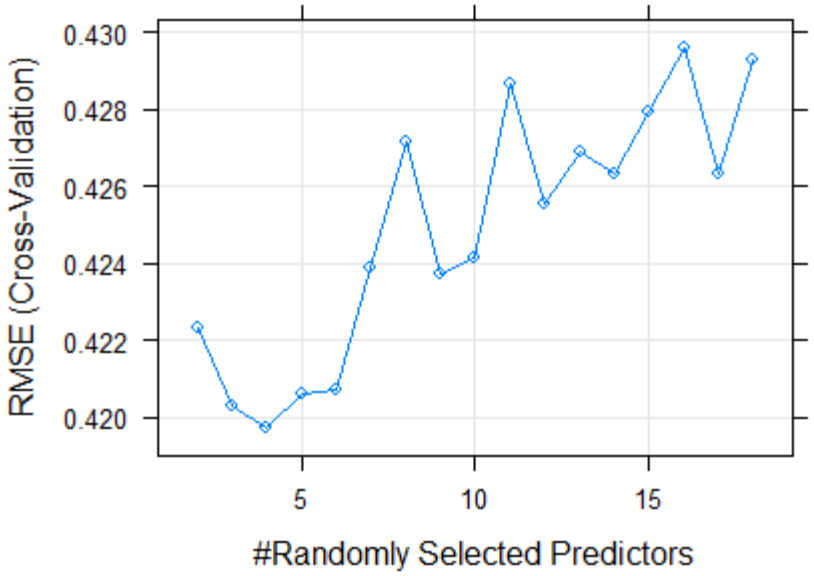
NOTA: `nodesize` también es otro hiperparámetro que se puede optimizar antes de llevar a cabo el ajuste del modelo. La forma correcta es optimizar los diferentes parámetros es de manera no secuencial debido a la interacción de los mismos. En este ejemplo solo se optimiza *mtry*.

```
library(caret)

# Método de validación
cv <- trainControl(method="cv", number=10, search="grid")

# Hiperparámetro a evaluar: número de predictores aleatorios en cada ramificación.
mtry <- expand.grid(.mtry=c(2:18))
```

```r
set.seed(5)
# Evaluación del mejor mtry
mtry_eval <- train(Salary ~., data = datos.H.train,
                   method="rf",
                   tuneGrid= mtry,
                   trControl= cv)

print(mtry_eval)
```

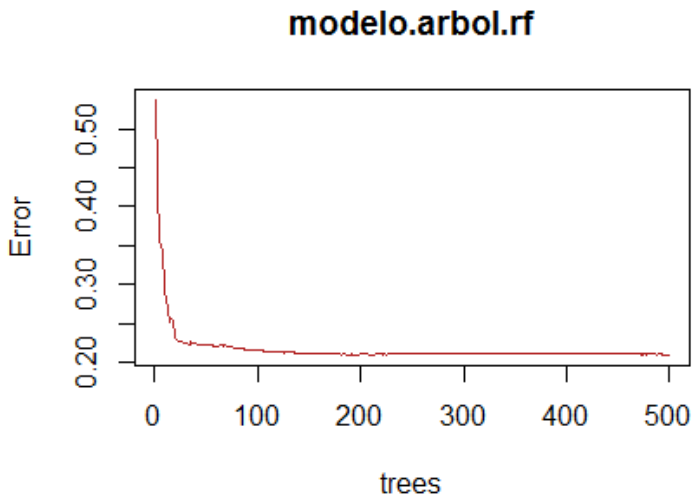
```
 ## Random Forest
##
## 200 samples
##  19 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 180, 180, 179, 180, 180, 180, ...
## Resampling results across tuning parameters:
##
##   mtry  RMSE       Rsquared   MAE
##    2    0.4223142  0.7857987  0.3073154
##    3    0.4202702  0.7857565  0.3037300
##    4    0.4196977  0.7853338  0.3025999
##    5    0.4205946  0.7842937  0.3039913
##    6    0.4207111  0.7835767  0.3037085
##    7    0.4238738  0.7812753  0.3044696
##    8    0.4271681  0.7777777  0.3086185
##    9    0.4237009  0.7814243  0.3052706
##   10    0.4241327  0.7803874  0.3063225
##   11    0.4286657  0.7768568  0.3080168
##   12    0.4255422  0.7787848  0.3069750
##   13    0.4269252  0.7783457  0.3072854
##   14    0.4263076  0.7785399  0.3073513
##   15    0.4279470  0.7773520  0.3094263
##   16    0.4296285  0.7757457  0.3094342
##   17    0.4263104  0.7785275  0.3062605
##   18    0.4293091  0.7757710  0.3101301
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 4.
```

```r
plot(mtry_eval)
```



El valor más óptimo de *mtry* en cuanto a menor *RMSE* es de 4 predictores.

```
# Modelo random forest
modelo.arbol.rf <- randomForest(Salary ~ ., data = datos.H.train,
                                mtry = 4,
                                importance = TRUE,
                                ntree = 500)

plot(modelo.arbol.rf, col = "firebrick")
```



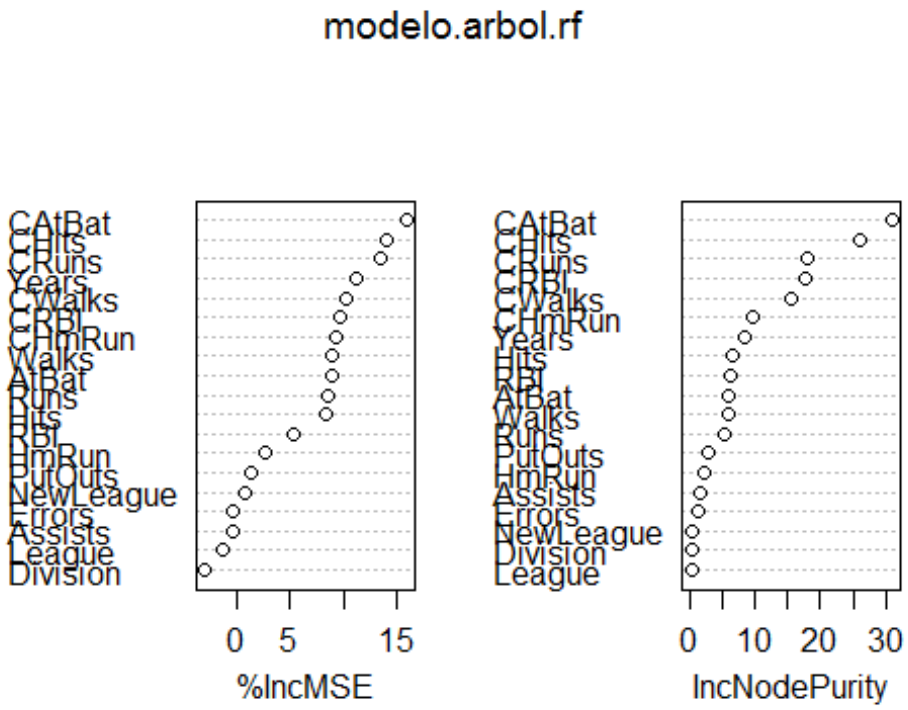
Ajustando el modelo por *random forest*, el error se estabiliza en torno a 200 árboles.

Identificación de los predictores más importantes

```
importance(modelo.arbol.rf)
```

```
##               %IncMSE IncNodePurity
## AtBat       8.8701439     5.8096334
## Hits        8.3788599     6.3341066
## HmRun       2.6877858     2.1891012
## Runs        8.5374700     5.0486537
## RBI         5.2466752     5.9952610
## Walks       8.9142867     5.7330607
## Years      11.1393669     8.3483567
## CAtBat     16.0486330    31.1634181
## CHits      14.1666774    26.0486088
## CHmRun      9.3602967     9.6929400
## CRuns      13.5488301    18.0476783
## CRBI        9.6941698    17.7697372
## CWalks     10.2342932    15.5031767
## League     -1.4150799     0.1746901
## Division   -2.9964941     0.2344381
## PutOuts     1.2948486     2.6774901
## Assists    -0.3822444     1.6070584
## Errors     -0.3209988     1.2312354
## NewLeague   0.7883069     0.2499387
```

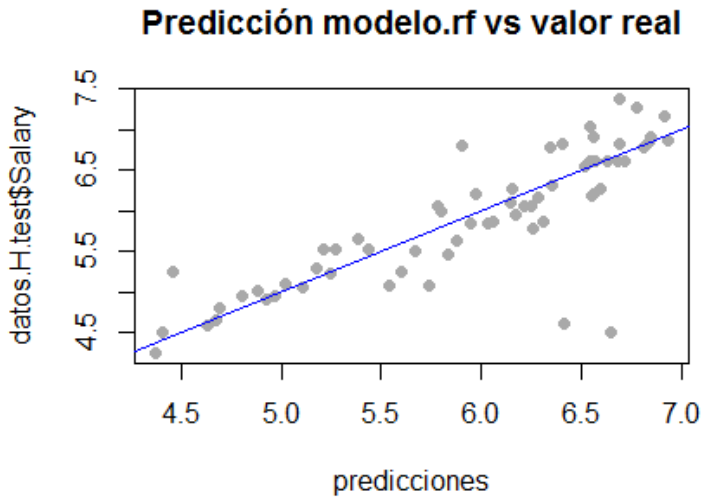
```
varImpPlot(modelo.arbol.rf)
```



En este caso, las dos variables con mayor importancia son *CAtBat* y *CHits*.

EVALUACIÓN DEL MODELO

```
pred.arbol <- predict(object = modelo.arbol.rf, newdata = datos.H.test)
plot(x = pred.arbol, y = datos.H.test$Salary,
     main = "Predicción modelo.rf vs valor real",
     xlab = "predicciones",
     col = "darkgrey",
     pch = 19)
abline(a = 0, b = 1, col = "blue")
```




```
test.MSE.rf <- mean((pred.arbol - datos.H.test$Salary)^2)


test.MSE.rf

## [1] 0.2130045
```

El *test error* se ha reducido levemente respecto al modelo ajustado por *bagging*. Por lo general, *random forest* ofrece mejores resultados que *bagging* al decorrelacionar los árboles.

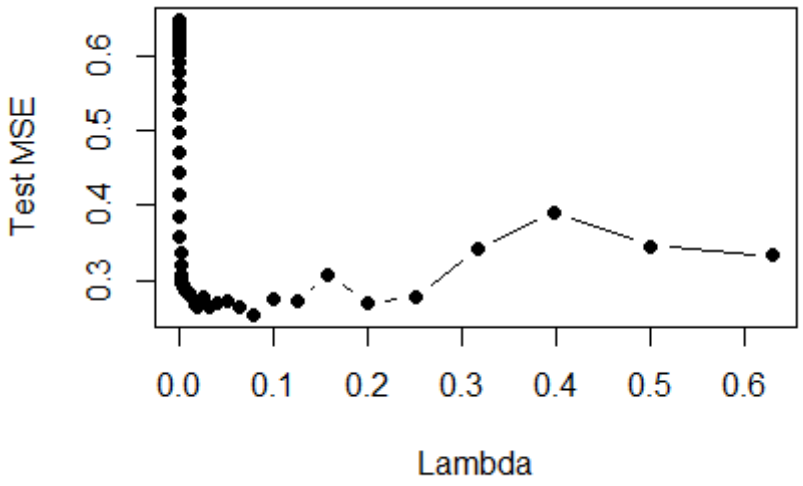
BOOSTING

AJUSTE DEL MODELO

Optimización de hiperparámetros

En lugar de utilizar el valor por defecto de λ = 0,01, evaluaremos cual es el valor óptimo para nuestros datos.

```
library(gbm)

set.seed(1)
potencia <- seq(-10, -0.2, by = 0.1)
# Valores de lambda a evaluar
lambdas <- 10^potencia
test.error <- rep(NA, length(lambdas))

# For loop para identificar el mejor lambda
for (i in 1:length(lambdas)) {
modelo.boost <- gbm(Salary ~ ., data = datos.H.train,
                    distribution = "gaussian",
                    n.trees = 5000,
                    shrinkage = lambdas[i])
pred <- predict(modelo.boost, datos.H.test, n.trees = 5000)
test.error[i] <- mean((pred - datos.H.test$Salary)^2)
}

plot(x= lambdas, y = test.error, type = "b", xlab = "Lambda", ylab = "Test MSE",
     pch = 19)
```




```
lambdas[which.min(test.error)]

## [1] 0.01258925
```

El valor escogido que minimiza el *test MSE* es λ = 0,012.

En el ejemplo anterior solo se ha optimizado un solo hiperparámetro, λ, pero es aconsejable optimizar también:

- `interaction.depth` (complejidad del árbol)
- `n.minobsinnode` (nº mínimo de observaciones por nodo para llevar a cabo la división)
- `n.trees` (nº de iteraciones, o en este caso, árboles)

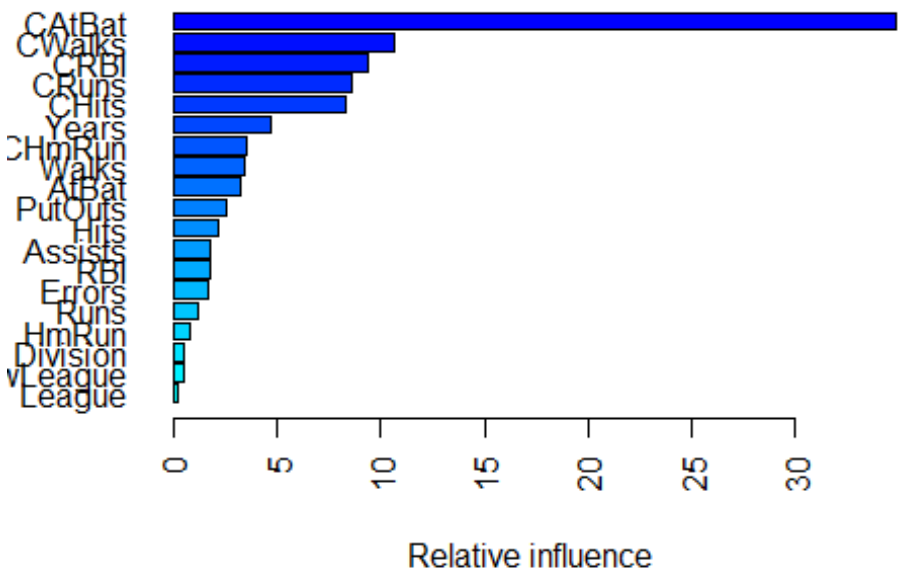
Con el paquete `caret` podemos optimizar todos estos hiperparámetros de manera conjunta:

```
# Método de evaluación de los modelos
set.seed(10)
cv <- trainControl(method="cv", number=10, search = "grid")
# Hiperparámetros a evaluar
opt.par <- expand.grid(interaction.depth = c(1, 5, 9),
                       n.trees = c(1000, 2000, 3000, 4000),
                       shrinkage = c(0.1, 0.01, 0.001),
                       n.minobsinnode = c(5, 10, 15))
set.seed(10)
# Evaluación del major conjunto de hiperparámetros
mejores.par <- train(Salary ~., data = datos.H.train,
                     method="gbm",
                     tuneGrid= opt.par,
                     trControl= cv,
                     verbose = FALSE)


mejores.par$bestTune

##    n.trees interaction.depth shrinkage n.minobsinnode
## 36    4000                 9     0.001             15
```

Procedemos a ajustar el modelo con los hiperparámetros optimizados:

```
# Modelo boost con hiperparámetros optimizados
modelo.arbol.boost <- gbm(Salary ~ ., data = datos.H.train,
                          distribution = "gaussian",
                          n.trees = 4000,
                          shrinkage = 0.001,
                          interaction.depth = 9,
                          n.minobsinnode = 15)

summary(modelo.arbol.boost, las = 2)

##                 var    rel.inf
## CAtBat       CAtBat 34.8632434
## CWalks       CWalks 10.6094840
## CRBI           CRBI  9.3708698
```

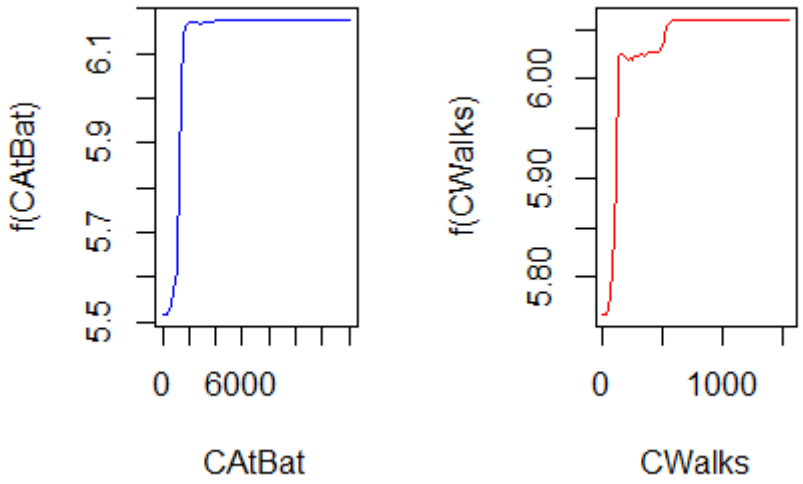
```
## CRuns          CRuns  8.6525466
## CHits          CHits  8.3199699
## Years          Years  4.6932290
## CHmRun        CHmRun  3.5224585
## Walks          Walks  3.4751841
## AtBat          AtBat  3.2282573
## PutOuts      PutOuts  2.6038701
## Hits            Hits  2.1383607
## Assists      Assists  1.8018555
## RBI              RBI  1.7424711
## Errors        Errors  1.6956301
## Runs            Runs  1.2250718
## HmRun          HmRun  0.7718587
## Division    Division  0.5335554
## NewLeague NewLeague  0.5142247
## League        League  0.2378593
```



CAtBat es, con diferencia, el predictor más influyente del modelo ajustado por *boosting*, seguido de *CWalks*.

Podemos, además, representar la influencia de estos predictores sobre la variable respuesta cuando se mantienen constantes el resto de predictores (*partial dependence plots*).

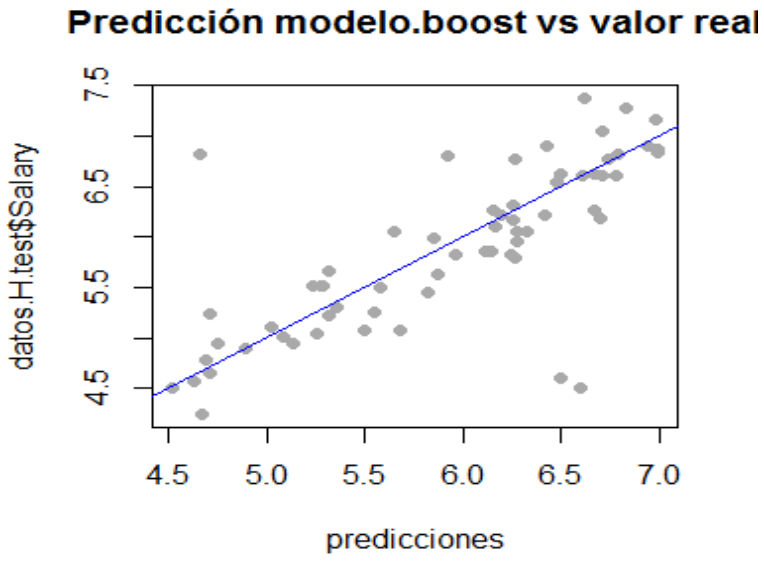
```
par(mfrow = c(1, 2))
plot(modelo.arbol.boost, i = "CAtBat", col = "blue")
plot(modelo.arbol.boost, i = "CWalks", col = "red")
```



El salario medio de los jugadores aumenta conforme aumenta *CAtBat* y CWalks.

EVALUACIÓN DEL MODELO

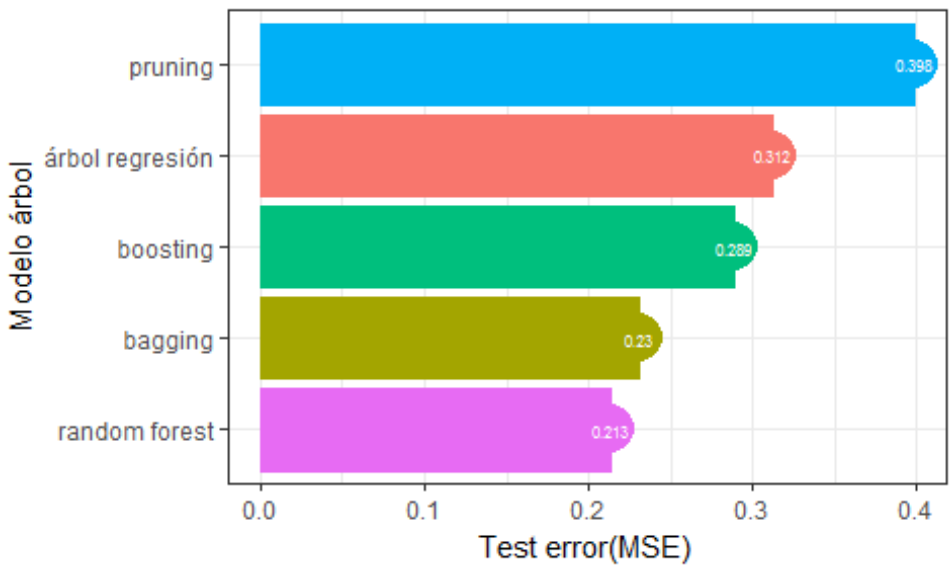
```
pred.arbol <- predict(object = modelo.arbol.boost, newdata = datos.H.test,
                      n.trees = 4000)

plot(x = pred.arbol, y = datos.H.test$Salary,
     main = "Predicción modelo.boost vs valor real", xlab = "predicciones",
     col = "darkgrey", pch = 19)
abline(a = 0, b = 1, col = "blue")
```



```
test.MSE.boost <- mean((pred.arbol - datos.H.test$Salary)^2)


test.MSE.boost

## [1] 0.2885397
```

COMPARACIÓN DE MODELOS

```
modelo <- c("árbol regresión", "pruning", "bagging", "random forest",
            "boosting")
test.MSE <- c(test.MSE.arbol.r, test.MSE.p, test.MSE.bag, test.MSE.rf,
              test.MSE.boost)
comparacion <- data.frame(modelo = modelo, test.MSE = test.MSE)



ggplot(data = comparacion, aes(x = reorder(x = modelo, X = test.MSE),
                               y = test.MSE, color = modelo,
                               label = round(test.MSE,3))) +
geom_bar(stat = "identity", aes(fill = modelo)) +
geom_point(size = 8) +
geom_text(color = "white", size = 2) +
labs(x = "Modelo árbol", y = "Test error(MSE)") +
theme_bw() +
coord_flip() +
theme(legend.position = "none")
```



Random forest es el método que ha conseguido superar al resto, con un menor error de predicción cuando se ha aplicado a nuevos datos. En conjunto, los métodos de *pruning*, *bagging*, y *boosting* también mejoran el modelo frente al árbol simple, lo que refleja la superioridad de estos métodos.

Ejemplo de clasificación

Para este ejemplo de clasificación utilizaremos el set de datos *OJ*, también parte del paquete *ISLR*. Contiene información sobre compra de dos tipos de bebida (*Citrus Hill* y *Minute Maid Orange Juice*) por parte de 1070 clientes (las variables registran distintas características del cliente y el producto). En este caso, generaremos árboles de clasificación que predigan que tipo de bebida (*Purchase*) se compra, en función del conjunto de predictores.

```
str(OJ)
```

```
## 'data.frame':    1070 obs. of  18 variables:
##  $ Purchase      : Factor w/ 2 levels "CH","MM": 1 1 1 2 1 1 1 1 1 1 ...
##  $ WeekofPurchase: num  237 239 245 227 228 230 232 234 235 238 ...
##  $ StoreID       : num  1 1 1 1 7 7 7 7 7 7 ...
##  $ PriceCH       : num  1.75 1.75 1.86 1.69 1.69 1.69 1.69 1.75 1.75 1.75 ...
##  $ PriceMM       : num  1.99 1.99 2.09 1.69 1.69 1.99 1.99 1.99 1.99 1.99 ...
##  $ DiscCH        : num  0 0 0.17 0 0 0 0 0 0 0 ...
##  $ DiscMM        : num  0 0.3 0 0 0 0 0.4 0.4 0.4 0.4 ...
##  $ SpecialCH     : num  0 0 0 0 0 0 1 1 0 0 ...
##  $ SpecialMM     : num  0 1 0 0 0 1 1 0 0 0 ...
##  $ LoyalCH       : num  0.5 0.6 0.68 0.4 0.957 ...
##  $ SalePriceMM   : num  1.99 1.69 2.09 1.69 1.69 1.99 1.59 1.59 1.59 1.59 ...
##  $ SalePriceCH   : num  1.75 1.75 1.69 1.69 1.69 1.69 1.69 1.75 1.75 1.75 ...
##  $ PriceDiff     : num  0.24 -0.06 0.4 0 0 0.3 -0.1 -0.16 -0.16 -0.16 ...
##  $ Store7        : Factor w/ 2 levels "No","Yes": 1 1 1 1 2 2 2 2 2 2 ...
##  $ PctDiscMM     : num  0 0.151 0 0 0 ...
##  $ PctDiscCH     : num  0 0 0.0914 0 0 ...
##  $ ListPriceDiff : num  0.24 0.24 0.23 0 0 0.3 0.3 0.24 0.24 0.24 ...
##  $ STORE         : num  1 1 1 1 0 0 0 0 0 0 ...
```

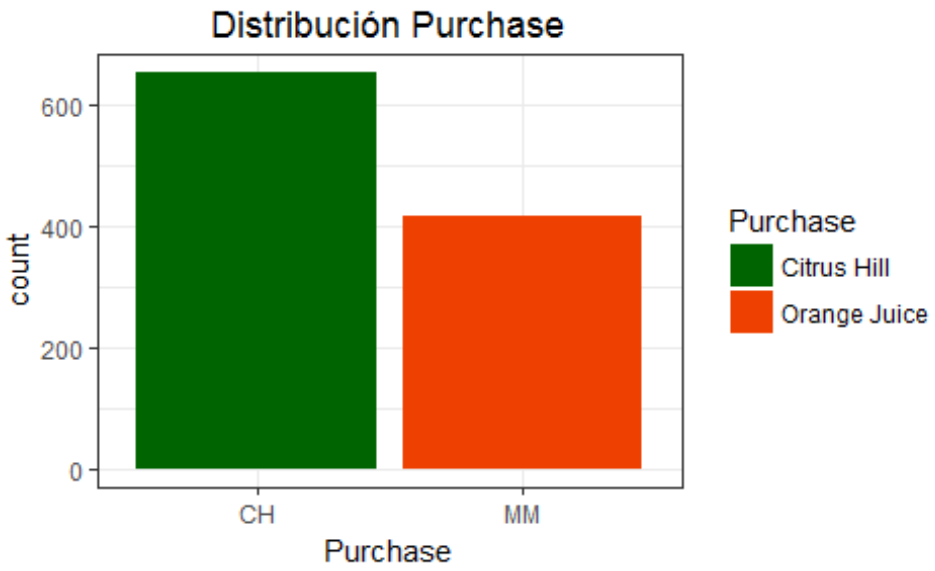
```
sum(is.na(OJ$Purchase))
```

```
## [1] 0
```

El set de datos no contiene observaciones ausentes para la variable respuesta *Purchase*.

```
# Distribución variable respuesta
library(ggplot2)

ggplot(data = OJ, aes(x = Purchase, y = ..count.., fill = Purchase)) +
```

```
geom_bar() +
labs(title = "Distribución Purchase") +
scale_fill_manual(values = c("darkgreen", "orangered2"),
                  labels = c("Citrus Hill", "Orange Juice")) +
theme_bw() + theme(plot.title = element_text(hjust = 0.5))
```




```
# Tabla frecuencias variable respuesta
table(OJ$Purchase)

##
##  CH  MM
## 653 417

# Tabla proporciones variable respuesta
library(dplyr)
prop.table(table(OJ$Purchase)) %>% round(digits = 2)

##
##   CH   MM
## 0.61 0.39
```

La clase mayoritaria (moda) en este caso es la bebida *CH* con el 61% de las compras. Este será el nivel basal a superar por el modelo (este es el porcentaje mínimo de aciertos si siempre se predijera *CH*).

Esta división dependerá de la cantidad de observaciones con las que contemos y la seguridad con la que queramos obtener la estimación del *test error*. Antes de pasar a generar los modelos, dividimos el set de datos en un grupo de entrenamiento (para el ajuste de los modelos) y otro de test (para la evaluación de los mismos). En este caso se opta por una división 80%-20%.

```r
# Índices observaciones de entrenamiento
train <- createDataPartition(y = OJ$Purchase, p = 0.8, list = FALSE, times = 1)


# Datos entrenamiento
datos.OJ.train <- OJ[train, ]


dim(datos.OJ.train)

## [1] 857  18

# Datos test
datos.OJ.test <- OJ[-train, ]


dim(datos.OJ.test)

## [1] 213  18
```

Es importante comprobar que la variable respuesta se distribuye de equitativamente en ambos grupos y con respecto al set de datos completo, aunque la función del paquete *caret* `createDataPartition()` asegura esta distribución:

```r
prop.table(table(datos.OJ.train$Purchase))

##
##        CH        MM
## 0.6102684 0.3897316

prop.table(table(datos.OJ.test$Purchase))

##
##        CH        MM
## 0.6103286 0.3896714
```

ÁRBOL DE CLASIFICACIÓN SIMPLE

AJUSTE DEL MODELO

NOTA: Para ajustar un árbol de regresión, la variable respuesta ha de ser de tipo `factor`.

```r
# Modelo árbol de clasificación
modelo.arbol.c <- tree(Purchase ~ ., data = datos.OJ.train)

summary(modelo.arbol.c)
```

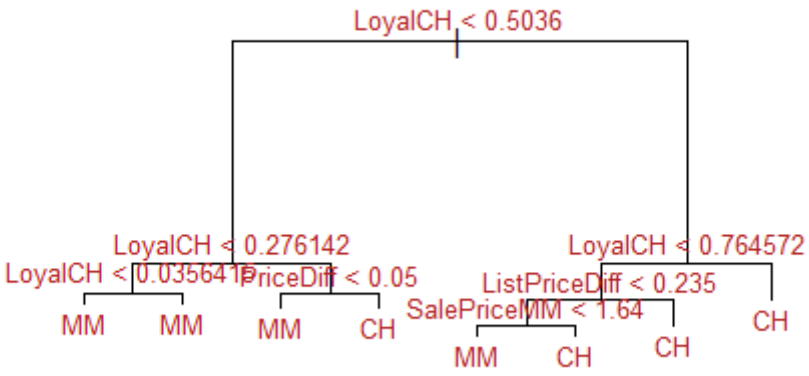
```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = datos.OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff" "SalePriceMM"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7882 = 669.2 / 849
## Misclassification error rate: 0.1739 = 149 / 857
```

El modelo ha utilizado solo 4 variables (nodos internos) del total de 17, con un número final de 8 nodos terminales. El *training error* (observaciones mal clasificadas) es del 17,39%. El *residual mean deviance* se corresponde con la *deviance* (669,2) dividida entre el nº de observaciones – nº de nodos terminales (857 – 18 = 849). Para árboles de clasificación, la *deviance* (índice Gini o *entropy*) viene dada por

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

donde n_{mk} es el número de observaciones en el m_i nodo terminal que pertenece a la clase k_i.

```
# Representación gráfica del árbol
plot(modelo.arbol.c)
# pretty = 0 incluye los nombres de los niveles para las variables cualitativas, e
n lugar de mostrar solo una letra
text(modelo.arbol.c, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```




```
modelo.arbol.c

## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 857 1146.00 CH ( 0.61027 0.38973 )
##    2) LoyalCH < 0.5036 378  455.90 MM ( 0.29101 0.70899 )
```

```
##      4) LoyalCH < 0.276142 183  149.60 MM ( 0.14208 0.85792 )
##        8) LoyalCH < 0.0356415 58   10.10 MM ( 0.01724 0.98276 ) *
##        9) LoyalCH > 0.0356415 125  125.10 MM ( 0.20000 0.80000 ) *
##      5) LoyalCH > 0.276142 195  266.60 MM ( 0.43077 0.56923 )
##       10) PriceDiff < 0.05 76   83.21 MM ( 0.23684 0.76316 ) *
##       11) PriceDiff > 0.05 119  163.50 CH ( 0.55462 0.44538 ) *
##    3) LoyalCH > 0.5036 479  384.10 CH ( 0.86221 0.13779 )
##      6) LoyalCH < 0.764572 197  229.40 CH ( 0.73096 0.26904 )
##       12) ListPriceDiff < 0.235 79  109.50 CH ( 0.50633 0.49367 )
##         24) SalePriceMM < 1.64 22   20.86 MM ( 0.18182 0.81818 ) *
##         25) SalePriceMM > 1.64 57   75.02 CH ( 0.63158 0.36842 ) *
##       13) ListPriceDiff > 0.235 118   85.95 CH ( 0.88136 0.11864 ) *
##      7) LoyalCH > 0.764572 282  105.40 CH ( 0.95390 0.04610 ) *
```

El predictor más importante es la lealtad a la marca del consumidor al producto *CH* (*LoyalCH*). En función del *LoyalCH*, la decisión también depende del *PriceDiff*: Si *LoyalCH* es < 0,276, la predicción es *MM*, y si es mayor, la clasificación depende del *PriceDiff* y *ListPriceDiff* (diferencia de precio entre la bebida *MM* y *CH*).

Ejemplo de interpretación (nodo terminal): si escogemos el nodo terminal indicado por el número 8) y el asterisco, vemos que el criterio de división para la creación de este nodo es *LoyalCH* < 0,035. El número de observaciones en esta rama es de 58 con una *deviance* de 10,10, y una predicción final para dicha rama de *MM* (menos del 2% de observaciones en esta rama pertenecen a la clase *CH*, mientras que el 98,2% pertenecen a la clase *MM*).

EVALUACIÓN DEL MODELO

```r
# Predicción del modelo con los datos de test
pred <- predict(modelo.arbol.c, newdata = datos.OJ.test, type = "class")
table(datos.OJ.test$Purchase, pred)
```

```
##     pred
##       CH  MM
##   CH 124   6
##   MM  19  64
```

```r
# Predicciones correctas
(124+64)/213
```

```
## [1] 0.8826291
```

```r
# Test error
test.MSE.arbol <- 1 - (124+64)/213


test.MSE.arbol
```

```
## [1] 0.1173709
```

El árbol de clasificación es capaz de predecir la clase del 88,26% de las observaciones del *test set* correctamente. El *test error* es por tanto del 11,73%.

PRUNING

AJUSTE DEL MODELO

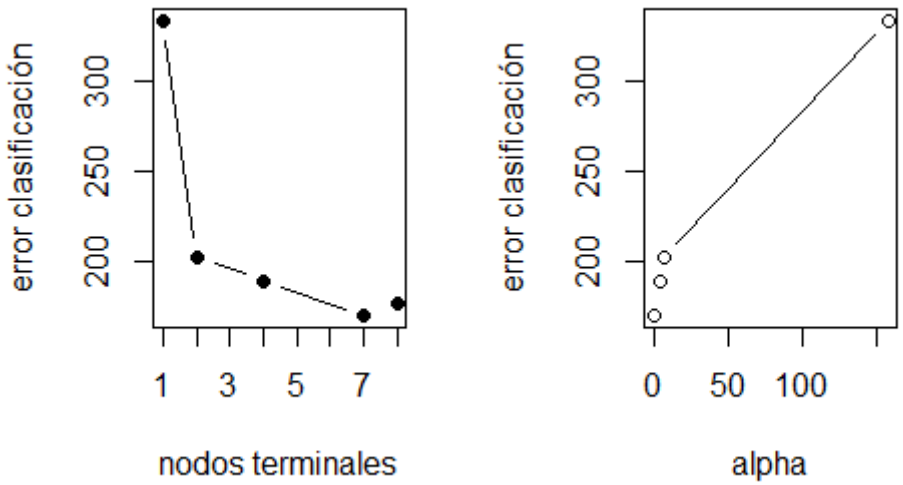
Para la poda de árboles de clasificación, especificamos en la función *cv.tree()* el argumento *FUN = prune.misclass* para indicar que queremos el error de clasificación como método para guiar el proceso.

```
# 10-fold cross validation
set.seed(3)
cv.arboles <- cv.tree(modelo.arbol.c, K = 10, FUN = prune.misclass)
cv.arboles
```

```
## $size
## [1] 8 7 4 2 1
##
## $dev
## [1] 176 170 189 202 333
##
## $k
## [1]       -Inf   0.000000   4.666667   6.500000 158.000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

En este output, la *deviance* se corresponde con el error de validación.

```
par(mfrow = c(1, 2))
# Cost complexity pruning
plot(cv.arboles$size, cv.arboles$dev, xlab = "nodos terminales",
     ylab = "error clasificación", type = "b", pch = 19)
plot(x = cv.arboles$k, y = cv.arboles$dev, xlab = "alpha",
     ylab = "error clasificación", type = "b")
```

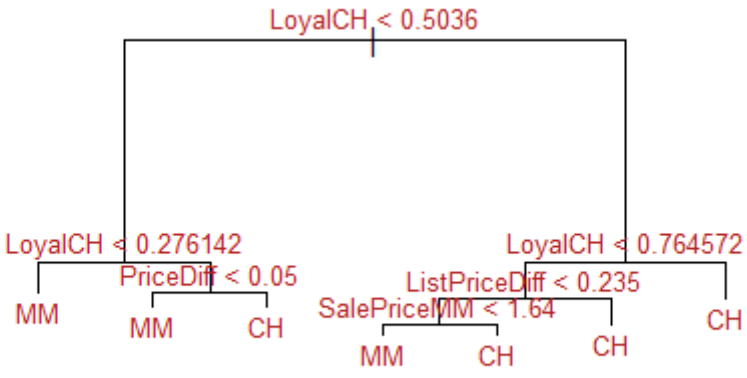



```
cv.arboles$size[which.min(cv.arboles$dev)]

## [1] 7
```

El número de nodos terminales que consigue minimizar el error de validación son 7, uno menos que el árbol sin podar.

Para la poda de un árbol de clasificación se utiliza la función `prune.misclass()`:

```
# Poda del arbol
modelo.arbol.c.p <- prune.misclass(tree = modelo.arbol.c, best = 7)

par(mfrow = c(1,1))
plot(x = modelo.arbol.c.p, type = "proportional")
text(modelo.arbol.c.p, splits = TRUE, pretty = 0, cex = 0.8, col = "firebrick")
```



EVALUACIÓN DEL MODELO

```
pred <- predict(modelo.arbol.c.p, newdata = datos.OJ.test, type = "class")
table(datos.OJ.test$Purchase, pred)
```

```
##     pred
##       CH  MM
##   CH 124   6
##   MM  19  64
```

```
# Predicciones correctas
(124+64)/213
```

```
## [1] 0.8826291
```

```
# Test error
test.MSE.arbol <- 1 - (124+64)/213
```

```
test.MSE.arbol
```

```
## [1] 0.1173709
```

En este caso la poda del árbol no ha modificado las predicciones.

BIBLIOGRAFÍA

An Introduction to Statistical Learning: with Applications in R (Springer Texts in Statistics)

https://topepo.github.io/caret/model-training-and-tuning.html