Aprendizaje Estadístico

Rubén Fernández Casal (ruben.fcasal@udc.es), Julián Costa (julian.costa@udc.es)

2020-10-20

Índice general

Prólogo

Este libro contiene los apuntes de la asignatura de Aprendizaje Estadístico del Máster en Técnicas Estadísticas.

Este libro ha sido escrito en R-Markdown empleando el paquete `bookdown` y está disponible en el repositorio Github: rubenfcasal/aprendizaje_estadistico. Se puede acceder a la versión en línea a través del siguiente enlace:

https://rubenfcasal.github.io/aprendizaje_estadistico.

donde puede descargarse en formato pdf.

Para ejecutar los ejemplos mostrados en el libro sería necesario tener instalados los siguientes paquetes: `caret`, `rattle`, `car`, `leaps`, `MASS`, `RcmdrMisc`, `lmtest`, `glmnet`, `mgcv`, `AppliedPredictiveModeling`, `ISLR`. Por ejemplo mediante los siguientes comandos:

```
pkgs <- c("caret", "rattle", "car", "leaps", "MASS", "RcmdrMisc",
          "lmtest", "glmnet", "mgcv",
          "AppliedPredictiveModeling", "ISLR")

install.packages(setdiff(pkgs, installed.packages()[,"Package"]), dependencies = TRUE)
# Si aparecen errores (normalmente debidos a incompatibilidades con versiones ya instaladas),
# probar a ejecutar en lugar de lo anterior:
# install.packages(pkgs, dependencies=TRUE) # Instala todos...
```

Para generar el libro (compilar) serán necesarios paquetes adicionales, para lo que se recomendaría consultar el libro de "Escritura de libros con bookdown" en castellano.

Este obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional (esperamos poder liberarlo bajo una licencia menos restrictiva más adelante…).

Capítulo 1

Introducción al Aprendizaje Estadístico

La denominada *Ciencia de Datos* (Data Science; también denominada *Science of Learning*) se ha vuelto muy popular hoy en día. Se trata de un campo multidisciplicar, con importantes aportaciones estadísticas e informáticas, dentro del que se incluirían disciplinas como *Minería de Datos* (Data Mining), *Aprendizaje Automático* (Machine Learning), *Aprendizaje Profundo* (Deep Learning), *Modelado Predictivo* (Predictive Modeling), *Extracción de Conocimiento* (Knowlegde Discovery) y también el *Aprendizaje Estadístico* (Statistical Learning).

Podríamos definir la Ciencia de Datos como el conjunto de conocimientos y herramientas utilizados en las distintas etapas del análisis de datos (ver Figura 1.1). Otras definiciones podrían ser:

- El arte y la ciencia del análisis inteligente de los datos.
- El conjunto de herramientas para entender y modelizar conjuntos (complejos) de datos.
- El proceso de descubrir patrones y obtener conocimiento a partir de grandes conjuntos de datos (*Big Data*).

Aunque esta ciencia incluiría también la gestión (sin olvidarnos del proceso de obtención) y la manipulación de los datos.

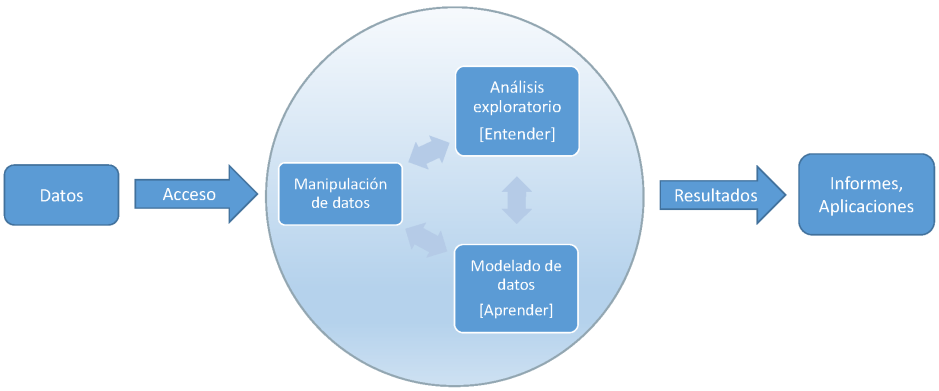


Figura 1.1: Etapas del proceso

Una de estas etapas (que están interrelacionadas) es la construcción de modelos a partir de los datos para aprender y predecir. Podríamos decir que el Aprendizaje Estadístico (AE) se encarga de este problema desde el punto de vista estadístico.

En Estadística se consideran modelos estocásticos (con componente aleatoria), para tratar de tener en cuenta la incertidumbre debida a que no se disponga de toda la información (sobre las variables

que influyen en el fenómeno de interés).

> "Nothing in Nature is random… a thing appears random only through the incompleteness of our knowledge."
>
> — Spinoza, Baruch (Ethics, 1677)
>
> "To my mind, although Spinoza lived and thought long before Darwin, Freud, Einstein, and the startling implications of quantum theory, he had a vision of truth beyond what is normally granted to human beings."
>
> — Shirley, Samuel (Complete Works, 2002). Traductor de la obra completa de Spinoza al inglés.

La Inferencia Estadística proporciona herramientas para ajustar este tipo de modelos a los datos observados (seleccionar un modelo adecuado, estimar sus parámetros y contrastar su validez). Sin embargo, en la aproximación estadística clásica como primer objetivo se trata de explicar por completo lo que ocurre en la población y suponiendo que esto se puede hacer con modelos tratables analíticamente, emplear resultados teóricos (típicamente resultados asintóticos) para realizar inferencias (entre ellas la predicción). Los avances en computación han permitido el uso de modelos estadísticos más avanzados, principalmente métodos no paramétricos, muchos de los cuales no pueden ser tratados analíticamente (por lo menos no por completo o no inicialmente), este es el campo de la Estadística Computacional[1]. Desde este punto de vista, el AE se enmarcaría dentro del campo de la Estadística Computacional.

Cuando pensamos en AE pensamos en:

- Flexibilidad (hay menos suposiciones sobre los datos).
- Procesamiento automático de datos.
- Big Data (en el sentido amplio, donde "big" puede hacer referencia a datos complejos).
- Predicción.

Por el contrario, muchos de los métodos del AE no se preocupan (o se preocupan poco) por:

- Reproducibilidad.
- Cuantificación de la incertidumbre (en términos de probabilidad).
- Inferencia.

La idea es "dejar hablar a los datos" y no "encorsetarlos" a priori, dándoles mayor peso que a los modelos. Sin embargo, esta aproximación puede presentar diversos inconvenientes:

- Algunos métodos son poco interpretables (se sacrifica la interpretabilidad por la precisión de las predicciones).
- Pueden aparecer problemas de sobreajuste (*overfitting*; en los métodos estadísticos clásicos es más habitual que aparezcan problemas de infraajuste, *underfitting*).
- Pueden presentar más problemas al extrapolar o interpolar (en comparación con los métodos clásicos).

1.1 Aprendizaje Estadístico vs. Aprendizaje Automático

El término *Machine Learning* (ML; Aprendizaje Automático) se utiliza en el campo de la *Intelingencia Artificial* desde 1959 para hacer referencia, fundamentalmente, a algoritmos de predicción (inicialmente para reconocimiento de patrones). Muchas de las herramientas que utilizan provienen del campo de la Estadística y, en cualquier caso, la Estadística (y por tanto las Matemáticas) es la base de todos estos enfoques para analizar datos (y no conviene perder la base formal). Por este motivo desde la

[1] Lauro (1996) definió la Estadística Computacional como la disciplina que tiene como objetivo "diseñar algoritmos para implementar métodos estadísticos en computadoras, incluidos los impensables antes de la era de las computadoras (por ejemplo, bootstrap, simulación), así como hacer frente a problemas analíticamente intratables".

Estadística Computacional se introdujo el término *Statistical Learning* (Aprendizaje Estadístico) para hacer referencia a este tipo de herramientas, pero desde el punto de vista estadístico (teniendo en cuenta la incertidumbre debida a no disponer de toda la información).

Tradicionalmente ML no se preocupa del origen de los datos e incluso es habitual que se considere que un conjunto enorme de datos es equivalente a disponer de toda la información (i.e. a la población).

> "The sheer volume of data would obviate the need of theory and even scientific method"
>
> — Chris Anderson, físico y periodista, 2008

Por el contrario en el caso del AE se trata de comprender, si es posible, el proceso subyacente del que provienen los datos y si estos son representativos de la población de interés (i.e. si tienen algún tipo de sesgo). No obstante, en este libro se considerará en general ambos términos como sinónimos.

ML/AE hacen un importante uso de la programación matemática, ya que muchos de sus problemas se plantean en términos de la optimización de funciones bajo restricciones. Recíprocamente, en optimización también se utilizan algoritmos de ML/AE.

1.1.1 Machine Learning vs. Data Mining

Mucha gente utiliza indistintamente los nombres ML y *Data Mining* (DM). Sin embargo, aunque tienen mucho solapamiento, lo cierto es que hacen referencia a conceptos ligeramente distintos.

ML es un conjunto de algoritmos principalmente dedicados a hacer predicciones y que son esencialmente automáticos minimizando la intervención humana.

DM intenta *entender* conjuntos de datos (en el sentido de encontrar sus patrones), requiere de una intervención humana activa (al igual que la Inferencia Estadística tradicional), pero utiliza entre otras las técnicas automáticas de ML. Por tanto podríamos pensar que es más parecido al AE.

1.1.2 Las dos culturas (Breiman, 2001)

Breiman diferencia dos objetivos en el análisis de datos, que él llama *información* (en el sentido de *inferencia*) y *predicción*. Cada uno de estos objetivos da lugar a una cultura:

- *Modelización de datos*: desarrollo de modelos (estocásticos) que permitan ajustar los datos y hacer inferencia. Es el trabajo habitual de los estadísticos académicos.
- *Modelización algorítmica* (en el sentido de predictiva): esta cultura no está interesada en los mecanismos que generan los datos, sólo en los algoritmos de predicción. Es el trabajo habitual de muchos estadísticos industriales y de muchos ingenieros informáticos. El ML es el núcleo de esta cultura que pone todo el énfasis en la precisión predictiva (así, un importante elemento dinamizador son las competiciones entre algoritmos predictivos, al estilo del Netflix Challenge).

1.1.3 Machine Learning vs. Estadística (Dunson, 2018)

- "Machine learning: The main publication outlets tend to be peer-reviewed conference proceedings and the style of research is very fast paced, trendy, and driven by performance metrics in prediction and related tasks".
- "Statistical community: The main publication outlets are peer-reviewed journals, most of which have a long drawn out review process, and the style of research tends to be careful, slower paced, intellectual as opposed to primarily performance driven, emphasizing theoretical support (e.g., through asymptotic properties), under-stated, and conservative".
- "*Big data* in ML typically means that the number of examples (i.e. sample size) is very large".

- "In statistics (…) it has become common to collect high dimensional, complex and intricately structured data. Often the dimensionality of the data vastly exceeds the available sample size, and the fundamental challenge of the statistical analysis is obtaining new insights from these huge data, while maintaining reproducibility/replicability and reliability of the results".

1.2 Métodos de Aprendizaje Estadístico

Dentro de los problemas que aborda el Aprendizaje Estadístico se suelen diferenciar dos grandes bloques: el aprendizaje no supervisado y el supervisado. El *aprendizaje no supervisado* comprende los métodos exploratorios, es decir, aquellos en los que no hay una variable respuesta (al menos no de forma explícita). El principal objetivo de estos métodos es entender las relaciones entre los datos y su estructura, y pueden clasificarse en las siguientes categorías:

- Análisis descriptivo.
- Métodos de reducción de la dimensión (análisis de componentes principales, análisis factorial…).
- Clúster.
- Detección de datos atípicos.

El *aprendizaje supervisado* engloba los métodos predictivos, en los que una de las variables está definida como variable respuesta. Su principal objetivo es la construcción de modelos que posteriormente se utilizarán, sobre todo, para hacer predicciones. Dependiendo del tipo de variable respuesta se diferencia entre:

- Clasificación: respuesta categórica (también se emplea la denominación de variable cualitativa, discreta o factor).
- Regresión: respuesta numérica (cuantitativa).

En este libro nos centraremos únicamente en el campo del aprendizaje supervisado y combinaremos la terminología propia de la Estadística con la empleada en AE (por ejemplo, en Estadística es habitual considerar un problema de clasificación como un caso particular de regresión).

1.2.1 Notación y terminología

Denotaremos por $\mathbf{X} = (X_1, X_2, \ldots, X_p)$ al vector formado por las variables predictoras (variables explicativas o variables independientes; también *inputs* o *features* en la terminología de ML), cada una de las cuales podría ser tanto numérica como categórica[2]. En general (ver comentarios más adelante), emplearemos $Y(\mathbf{X})$ para referirnos a la variable objetivo (variable respuesta o variable dependiente; también *output* en la terminología de ML), que como ya se comentó puede ser una variable numérica (regresión) o categórica (clasificación).

Supondremos que el objetivo principal es, a partir de una muestra:

$$\{(x_{1i}, \ldots, x_{pi}, y_i) : i = 1, \ldots, n\},$$

obtener (futuras) predicciones $\hat{Y}(\mathbf{x})$ de la respuesta para $\mathbf{X} = \mathbf{x} = (x_1, \ldots, x_p)$.

En regresión consideraremos como base el siguiente modelo general (podría ser después de una transformación de la respuesta):

$$Y(\mathbf{X}) = m(\mathbf{X}) + \varepsilon, \tag{1.1}$$

donde $m(\mathbf{x}) = E\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$ es la media condicional, denominada función de regresión (o tendencia), y ε es un error aleatorio de media cero y varianza σ^2, independiente de $\mathbf{X}$. Este modelo puede generalizarse

[2]Aunque hay que tener en cuenta que algunos métodos están diseñados para predictores numéricos, otros para categóricos y algunos para ambos tipos.

de diversas formas, por ejemplo, asumiendo que la distribución del error depende de X (considerando $\varepsilon(\mathbf{X})$ en lugar de ε) podríamos incluir dependencia y heterocedasticidad. En estos casos normalmente se supone que lo hace únicamente a través de la varianza (error heterocedástico independiente), denotando por $\sigma^2(\mathbf{x}) = Var\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$ la varianza condicional[3].

Como ya se comentó se podría considerar clasificación como un caso particular, por ejemplo definiendo $Y(\mathbf{X})$ de forma que tome los valores $1, 2, \ldots, K$, etiquetas que identifican las K posibles categorías (también se habla de modalidades, niveles, clases o grupos). Sin embargo, muchos métodos de clasificación emplean variables auxiliares (variables *dummy*), indicadoras de las distintas categorías, y emplearemos la notación anterior para referirnos a estas variables (también denominadas variables *target*). En cuyo caso, denotaremos por $G(\mathbf{X})$ la respuesta categórica (la clase verdadera; g_i, $i = 1, \ldots, n$, serían los valores observados) y por $\hat{G}(\mathbf{X})$ el predictor.

Por ejemplo, en el caso de dos categorías, se suele definir Y de forma que toma el valor 1 en la categoría de interés (también denominada *éxito* o *resultado positivo*) y 0 en caso contrario (*fracaso* o *resultado negativo*)[4]. Además, en este caso, los modelos típicamente devuelven estimaciones de la probabilidad de la clase de interés en lugar de predecir directamente la clase, por lo que se empleará $\hat{p}$ en lugar de $\hat{Y}$. A partir de esa estimación se obtiene una predicción de la categoría. Normalmente se predice la clase más probable, i.e. "éxito" si $\hat{p}(\mathbf{x}) > c = 0.5$ y "fracaso" en caso contrario (con probabilidad estimada $1 - \hat{p}(\mathbf{x})$).

Resulta claro que el modelo base general (1.1) puede no ser adecuado para modelar variables indicadoras (o probabilidades). Muchos de los métodos de AE emplean (1.1) para una variable auxiliar numérica (denominada puntuación o *score*) que se transforma a escala de probabilidades mediante la función logística (denominada función sigmoidal, *sigmoid function*, en ML)[5]:

$$p(s) = \frac{1}{1 + e^{-s}},$$

cuya inversa es la *función logit*:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right).$$

Lo anterior se puede generalizar para el caso de múltiples categorías, considerando variables indicadoras de cada categoría $Y_1, \ldots, Y_K$ (es lo que se conoce como la estrategia de "uno contra todos"). En este caso típicamente:

$$\hat{G}(\mathbf{x}) = \underset{k}{\text{argmax}} \left\{\hat{p}_k(\mathbf{x}) : k = 1, 2, \ldots, K\right\}.$$

1.2.2 Métodos (de aprendizaje supervisado) y paquetes de R

Hay una gran cantidad de métodos de aprendizaje supervisado implementados en centenares de paquetes de `R` (ver por ejemplo CRAN Task View: Machine Learning & Statistical Learning). A continuación se muestran los principales métodos y algunos de los paquetes de R que los implementan (muchos son válidos para regresión y clasificación, como por ejemplo los basados en árboles, aunque aquí aparecen en su aplicación habitual).

Métodos de Clasificación:

- Análisis discriminante (lineal, cuadrático), Regresión logística, multinomial…: `stats`, `MASS`…
- Árboles de decisión, *bagging*, *random forest*, *boosting*: `rpart`, `party`, `C50`, `Cubist`, `randomForest`, `adabag`, `xgboost`…
- *Support vector machines* (SVM): `kernlab`, `e1071`…

Métodos de regresión:

[3] Por ejemplo considerando en el modelo base $\sigma(\mathbf{X})\varepsilon$ como termino de error y suponiendo adicionalmente que ε tiene varianza uno.

[4] Otra alternativa sería emplear 1 y -1, algo que simplifica las expresiones de algunos métodos.

[5] De especial interés en regresión logística y en redes neuronales artificiales.

- Modelos lineales:
  - Regresión lineal: `lm()`, `lme()`, `biglm`…
  - Regresión lineal robusta: `MASS::rlm()`…
  - Métodos de regularización (Ridge regression, Lasso): `glmnet`, `elasticnet`…
- Modelos lineales generalizados: `glm()`, `bigglm`, ..
- Modelos paramétricos no lineales: `nls()`, `nlme`…
- Regresión local (vecinos más próximos y métodos de suavizado): `kknn`, `loess()`, `KernSmooth`, `sm`, `np`…
- Modelos aditivos generalizados (GAM): `mgcv`, `gam`…
- Redes neuronales: `nnet`…

También existen paquetes de `R` que permiten utilizar plataformas de ML externas, como por ejemplo `h2o` o `RWeka`.

Como todos estos paquetes emplean opciones, estructuras y convenciones sintácticas diferentes, se han desarrollado paquetes que proporcionan interfaces unificadas a muchas de estas implementaciones. Entre ellos podríamos citar `caret`,`mlr3` y `tidymodels`. En la Sección 1.6 se incluye una breve introducción al paquete `caret` que será empleado en diversas ocasiones a lo largo del presente libro.

Adicionalmente hay paquetes de R que disponen de entornos gráficos que permiten emplear estos métodos evitando el uso de comandos. Entre ellos estarían R-Commander con el plugin FactoMineR (`Rcmdr`, `RcmdrPlugin.FactoMineR`) y `rattle`.

1.3 Construcción y evaluación de los modelos

En Inferencia Estadística clásica el procedimiento habitual es emplear toda la información disponible para construir un modelo válido (que refleje de la forma más fiel posible lo que ocurre en la población) y asumiendo que el modelo es el verdadero (lo que en general sería falso) utilizar métodos de inferencia para evaluar su precisión. Por ejemplo, en el caso de regresión lineal múltiple, el coeficiente de determinación ajustado sería una medida del la precisión del modelo para predecir nuevas observaciones (no se debería emplear el coeficiente de determinación sin ajustar; aunque, en cualquier caso, su validez dependería de la de las suposiciones estructurales del modelo).

Alternativamente, en Estadística Computacional es habitual emplear técnicas de remuestreo para evaluar la precisión (entrenando también el modelo con todos los datos disponibles), principalmente validación cruzada (leave-one-out, k-fold), jackniffe o bootstrap.

Por otra parte, como ya se comentó, algunos de los modelos empleados en AE son muy flexibles (están hiperparametrizados) y pueden aparecer problemas si se permite que se ajusten demasiado bien a las observaciones (podrían llegar a interpolar los datos). En estos casos habrá que controlar el procedimiento de aprendizaje, típicamente a traves de parámetros relacionados con la complejidad del modelo (ver sección siguiente).

En AE se distingue entre parámetros estructurales, los que van a ser estimados al ajustar el modelo a los datos (en el entrenamiento), e hiperparámetros (*tuning parameters* o parámetros de ajuste), que imponen restricciones al aprendizaje del modelo (por ejemplo determinando el número de parámetros estructurales). Si los hiperparámetros seleccionados producen un modelo demasiado complejo aparecerán problemas de sobreajuste (*overfitting*) y en caso contrario de infraajuste (*undefitting*).

Hay que tener en cuenta también que al aumentar la complejidad disminuye la interpretabilidad de los modelos. Se trataría entonces de conseguir buenas predicciones (habrá que evaluar la capacidad predictiva) con el modelo más sencillo posible.

1.3.1 Equilibrio entre sesgo y varianza: infraajuste y sobreajuste

La idea es que queremos aprender más allá de los datos empleados en el entrenamiento (en Estadística diríamos que queremos hacer inferencia sobre nuevas observaciones). Como ya se comentó, en AE hay que tener especial cuidado con el sobreajuste. Este problema ocurre cuando el modelo se ajusta demasiado bien a los datos de entrenamiento pero falla cuando se utiliza en un nuevo conjunto de datos (nunca antes visto).

Como ejemplo ilustrativo emplearemos regresión polinómica, considerando el grado del polinomio como un hiperparámetro que determina la complejidad del modelo. En primer lugar simulamos una muestra y ajustamos modelos polinómicos con distintos grados de complejidad.

```
# Simulación datos
n <- 30
x <- seq(0, 1, length = n)
mu <- 2 + 4*(5*x - 1)*(4*x - 2)*(x - 0.8)^2 # grado 4
sd <- 0.5
set.seed(1)
y <- mu + rnorm(n, 0, sd)
plot(x, y)
lines(x, mu, lwd = 2)
# Ajuste de los modelos
fit1 <- lm(y ~ x)
lines(x, fitted(fit1))
fit2 <- lm(y ~ poly(x, 4))
lines(x, fitted(fit2), lty = 2)
fit3 <- lm(y ~ poly(x, 20))
# NOTA: poly(x, degree, raw = FALSE) tiene un problema de desbordamiento si degree > 25
lines(x, fitted(fit3), lty = 3)
legend("topright", legend = c("Verdadero", "Ajuste con grado 1",
                              "Ajuste con grado 4", "Ajuste con grado 20"),
       lty = c(1, 1, 2, 3), lwd = c(2, 1, 1, 1))
```

Como se observa en la Figura 1.2 al aumentar la complejidad del modelo se consigue un mejor ajuste a los datos observados (empleados en el entrenamiento), a costa de un incremento en la variabilidad de las predicciones, lo que puede producir un mal comportamiento del modelo a ser empleado en un conjunto de datos distinto del observado.

Si calculamos medidas de bondad de ajuste, como el error cuadrático medio (MSE) o el coeficiente de determinación, se obtienen mejores resultados al aumentar la complejidad. Como se trata de modelos lineales, podríamos obtener también el coeficiente de determinación ajustado, que sería preferible (en principio, ya que dependería de la validez de las hipótesis estructurales del modelo) para medir la precisión al emplear los modelos en un nuevo conjunto de datos.

```
knitr::kable(t(sapply(list(fit1 = fit1, fit2 = fit2, fit3 = fit3),
       function(x) with(summary(x),
                        c(MSE = mean(residuals^2), R2 = r.squared, R2adj = adj.r.squared)))), digi
```

	MSE	R2	R2adj
fit1	1.22	0.20	0.17
fit2	0.19	0.87	0.85
fit3	0.07	0.95	0.84

Por ejemplo, si generamos nuevas respuestas de este proceso, la precisión del modelo más complejo empeorará considerablemente:

```
y.new <- mu + rnorm(n, 0, sd)
plot(x, y)
points(x, y.new, pch = 2)
lines(x, mu, lwd = 2)
```

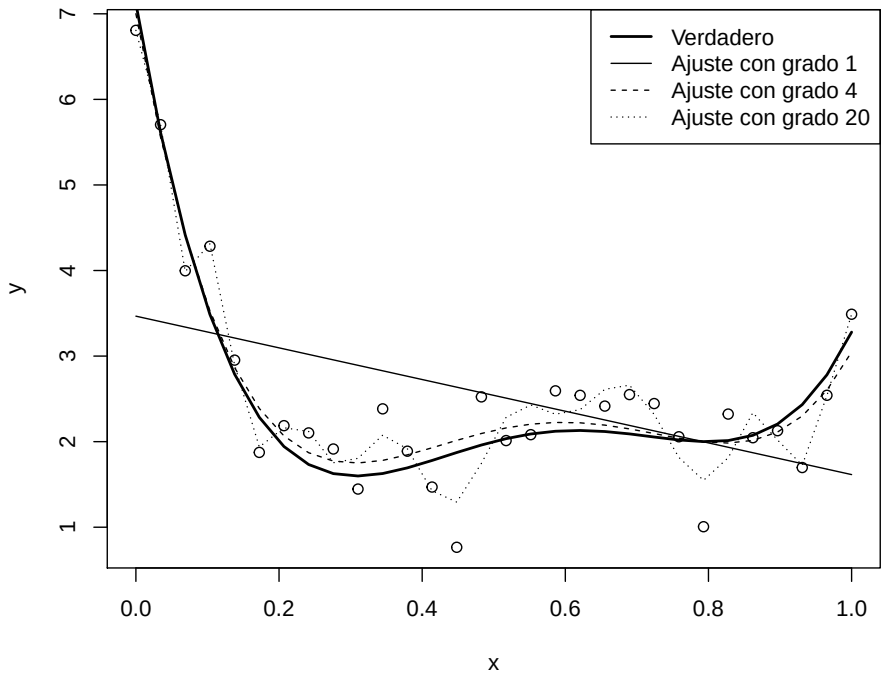


Figura 1.2: Muestra (simulada) y ajustes polinómicos con distinta complejidad.

```
lines(x, fitted(fit1))
lines(x, fitted(fit2), lty = 2)
lines(x, fitted(fit3), lty = 3)
legend("topright", legend = c("Verdadero", "Muestra", "Ajuste con grado 1", "Ajuste con grado 4",
                              "Ajuste con grado 20", "Nuevas observaciones"),
       lty = c(1, NA, 1, 2, 3, NA), lwd = c(2, NA, 1, 1, 1, NA), pch = c(NA, 1, NA, NA, NA, 2))
```

```
MSEP <- sapply(list(fit1 = fit1, fit2 = fit2, fit3 = fit3),
               function(x) mean((y.new - fitted(x))^2))
MSEP
```

```
##      fit1      fit2      fit3
## 1.4983208 0.1711238 0.2621064
```

Como ejemplo adicional, para evitar el efecto de la aleatoriedad de la muestra, en el siguiente código se simulan 100 muestras del proceso anterior a las que se les ajustan modelos polinómicos variando el grado de 1 a 20. Posteriormente se evalua la precisión en la muestra empleada en el ajuste y en un nuevo conjunto de datos procedente de la misma población.

```
nsim <- 100
set.seed(1)
grado.max <- 20
grados <- seq_len(grado.max)
mse <- mse.new <- matrix(nrow = grado.max, ncol = nsim) # Error cuadrático medio
for(i in seq_len(nsim)) {
  y <- mu + rnorm(n, 0, sd)
  y.new <- mu + rnorm(n, 0, sd)
  for (grado in grados) { # grado <- 1
    fit <- lm(y ~ poly(x, grado))
```

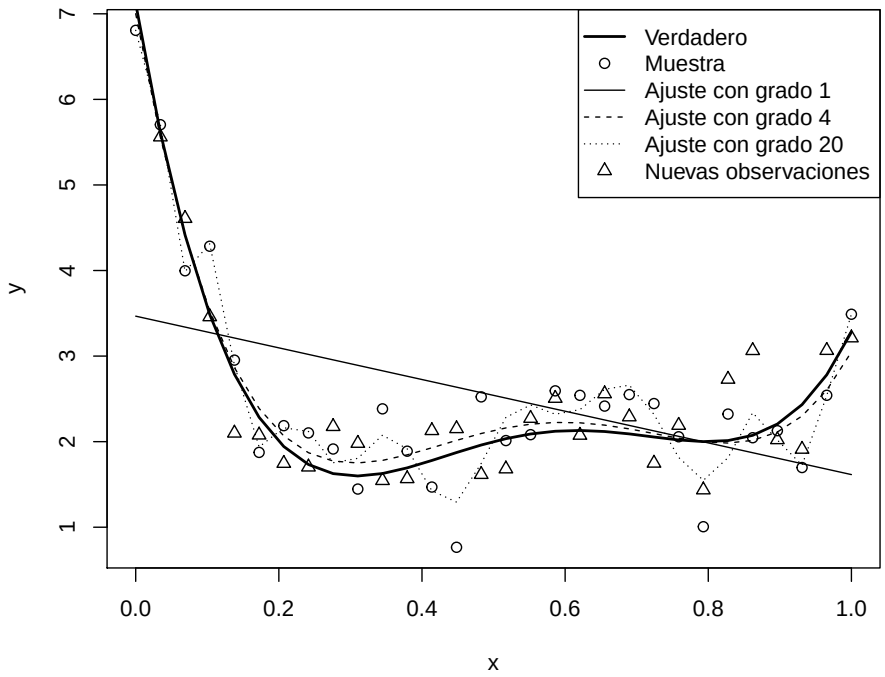


Figura 1.3: Muestra con ajustes polinómicos con distinta complejidad y nuevas observaciones.

```
    mse[grado, i] <- mean(residuals(fit)^2)
    mse.new[grado, i] <- mean((y.new - fitted(fit))^2)
  }
}
# Simulaciones
matplot(grados, mse, type = "l", col = "lightgray", lty = 1, ylim = c(0, 2),
        xlab = "Grado del polinomio (complejidad)",
        ylab = "Error cuadrático medio")
matlines(grados, mse.new, type = "l", lty = 2, col = "lightgray")
# Global
precision <- rowMeans(mse)
precision.new <- rowMeans(mse.new)
lines(grados, precision, lwd = 2)
lines(grados, precision.new, lty = 2, lwd = 2)
abline(h = sd^2, lty = 3)
abline(v = 4, lty = 3)
legend("topright", legend = c("Muestras", "Nuevas observaciones"), lty = c(1, 2))
```

Como se puede observar en la Figura 1.4 los errores de entrenamiento disminuyen a medida que aumenta la complejidad del modelo. Sin embargo los errores de predicción en nuevas observaciones primero disminuyen hasta alcanzar un mínimo, marcado por la línea de puntos vertical que se corresponde con el modelo de grado 4, y después aumentan (la línea de puntos horizontal es la varianza del proceso; el error cuadrático medio de predicción asintótico). La línea vertical representa el equilibrio entre el sesgo y la varianza. Considerando un valor de complejidad a la izquierda de esa línea tendríamos infraajuste (mayor sesgo y menor varianza) y a la derecha sobreajuste (menor sesgo y mayor varianza).

Desde un punto de vista más formal, considerando el modelo (1.1) y una función de pérdidas cuadrática,

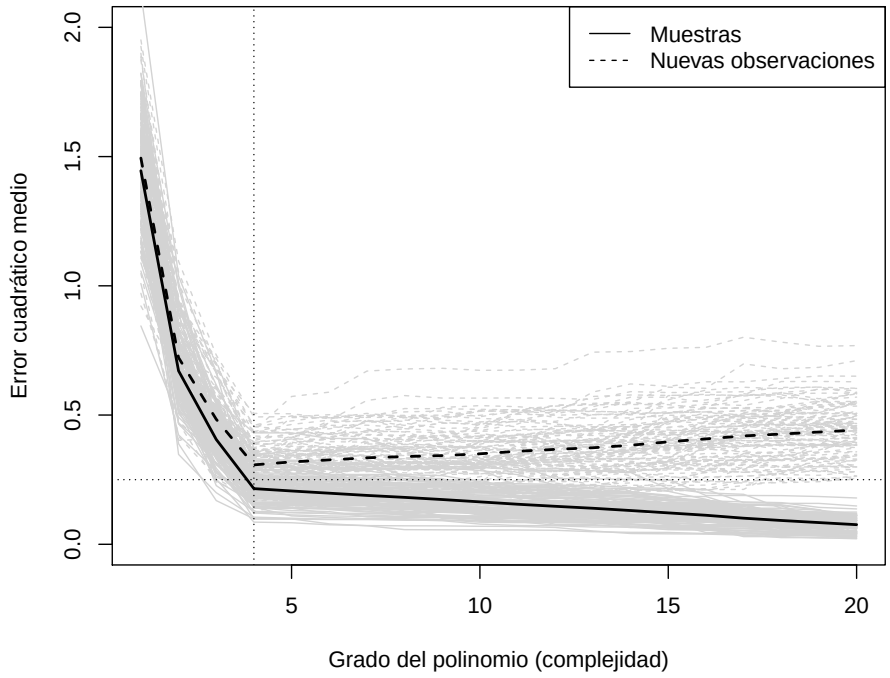


Figura 1.4: Precisiones (errores cuadráticos medios) de ajustes polinómicos variando la complejidad, en las muestras empleadas en el ajuste y en nuevas observaciones (simulados).

el predictor óptimo (desconocido) sería la media condicional $m(\mathbf{x}) = E\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$[6]. Por tanto los predictores serían realmente estimaciones de la función de regresión, $\hat{Y}(\mathbf{x}) = \hat{m}(\mathbf{x})$ y podemos expresar la media del error cuadrático de predicción en términos del sesgo y la varianza:

$$\begin{aligned} E\left(Y(\mathbf{x}_0) - \hat{Y}(\mathbf{x}_0)\right)^2 &= E\left(m(\mathbf{x}_0) + \varepsilon - \hat{m}(\mathbf{x}_0)\right)^2 = E\left(m(\mathbf{x}_0) - \hat{m}(\mathbf{x}_0)\right)^2 + \sigma^2 \\ &= E^2\left(m(\mathbf{x}_0) - \hat{m}(\mathbf{x}_0)\right) + Var\left(\hat{m}(\mathbf{x}_0)\right) + \sigma^2 \\ &= \text{sesgo}^2 + \text{varianza} + \text{error irreducible} \end{aligned}$$

donde $\mathbf{x}_0$ hace referencia al vector de valores de las variables explicativas de una nueva observación (no empleada en la construcción del predictor).

En general, al aumentar la complejidad disminuye el sesgo y aumenta la varianza (y viceversa). Esto es lo que se conoce como el dilema o compromiso entre el sesgo y la varianza (*bias-variance tradeoff*). La recomendación sería por tanto seleccionar los hiperparámetros (el modelo final) tratando de que haya un equilibrio entre el sesgo y la varianza.

1.3.2 Datos de entrenamiento y datos de test

Como se mostró en la sección anterior hay que tener mucho cuidado si se pretende evaluar la precisión de las predicciones empleando la muestra de entrenamiento.

Si el número de observaciones no es muy grande, se puede entrenar el modelo con todos los datos y emplear técnicas de remuestreo para evaluar la precisión (típicamente validación cruzada o bootstrap).

[6]Se podrían considerar otras funciones de pérdida, por ejemplo con la distancia L_1 sería la mediana condicional, pero las consideraciones serían análogas.

Habría que asegurase de que el procedimiento de remuestreo empleado es adecuado (por ejemplo, la presencia de dependencia requeriría de métodos más sofisticados).

Sin embargo, si el número de obervaciones es grande, se suele emplear el procedimiento tradicional en ML, que consiste en particionar la base de datos en 2 (o incluso en 3) conjuntos (disjuntos):

- Conjunto de datos de entrenamiento (o aprendizaje) para construir los modelos.
- Conjunto de datos de test para evaluar el rendimiento de los modelos.

Los datos de test deberían utilizarse únicamente para evaluar los modelos finales, no se deberían emplear para seleccionar hiperparámetros. Para seleccionalos se podría volver a particionar los datos de entrenamiento, es decir, dividir la muestra en tres subconjuntos: datos de entrenamiento, de validación y de test (por ejemplo considerando un 70%, 15% y 15% de las observaciones, respectivamente). Para cada combinación de hiperparámetros se ajustaría el correspondiente modelo con los datos de entrenamiento, se emplearían los de validación para evaluarlos y posteriormente seleccionar los valores "óptimos". Por último, se emplean los datos de test para evaluar el rendimiento del modelo seleccionado. No obstante, lo más habitual es seleccionar los hiperparámetros empleando validación cruzada (o otro tipo de remuestreo) en la muestra de entrenamiento, en lugar de considerar una muestra adicional de validación. En la siguiente sección se describirá esta última aproximación.

En R se puede realizar el particionamiento de los datos empleando la función `sample()` del paquete base (otra alternativa sería emplear la función `createDataPartition` del paquete `caret` como se describe en la Sección 1.6). Típicamente se selecciona el 80% de los datos como muestra de entrenamiento y el 20% restante como muestra de test, aunque esto dependería del número de datos.

Como ejemplo consideraremos el conjunto de datos `Boston` del paquete `MASS` que contiene, entre otros datos, la valoración de las viviendas (`medv`, mediana de los valores de las viviendas ocupadas, en miles de dólares) y el porcentaje de población con "menor estatus" (`lstat`) en los suburbios de Boston. Podemos contruir las muestras de entrenamiento (80%) y de test (20%) con el siguiente código:

```
data(Boston, package = "MASS")
# ?Boston
set.seed(1)
nobs <- nrow(Boston)
itrain <- sample(nobs, 0.8 * nobs)
train <- Boston[itrain, ]
test <- Boston[-itrain, ]
```

1.3.3 Validación cruzada

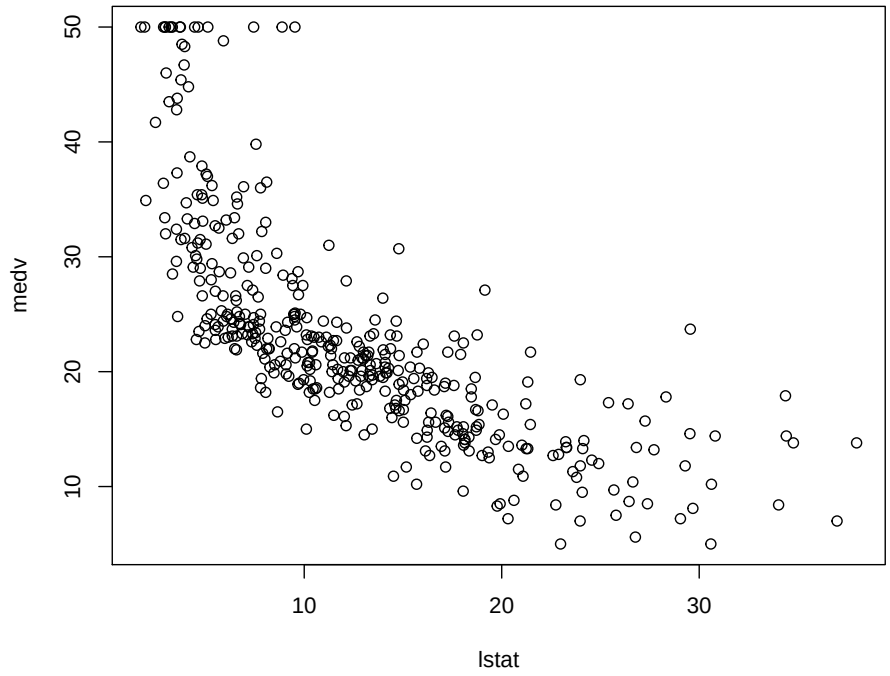
Como ya se comentó, una herramienta para evaluar la calidad predictiva de un modelo es la *validación cruzada*, que permite cuantificar el error de predicción utilizando una única muestra de datos.

En su versión más simple, validación cruzada dejando uno fuera (*Leave-one-out cross-validation*, LOOCV), para cada observación de la muestra se realiza un ajuste empleando el resto de observaciones, y se mide el error de predicción en esa observación (único dato no utilizado en el ajuste del modelo). Finalmente, combinando todos los errores individuales se puede obtener medidas globales del error de predicción (o aproximar características de su distribución).

El método de LOOCV requeriría, en principio (ver comentarios más adelante), el ajuste de un modelo para cada observación por lo que pueden aparecer problemas computacionales si el conjunto de datos es grande. En este caso se suele emplear grupos de observaciones en lugar de observaciones individuales. Si se particiona el conjunto de datos en k grupos, típicamente 10 o 5 grupos, se denomina *k-fold cross-validation* (LOOCV sería un caso particular considerando un número de grupos igual al número de observaciones). Hay muchas variaciones de este método, entre ellas particionar repetidamente de forma aleatoria los datos en un conjunto de entrenamiento y otro de validación (de esta forma algunas observaciones podrían aparecer repetidas veces y otras ninguna en las muestras de validación).

Continuando con el ejemplo anterior, supongamos que queremos emplear regresión polinómica para explicar la valoración de las viviendas (`medv`) a partir del "estatus" de los residentes (`lstat`). Al igual que se hizo en la Sección 1.3.1, consideraremos el grado del polinomio como un hiperparámetro.

```
plot(medv ~ lstat, data = train)
```



Podríamos emplear la siguiente función que devuelve para cada observación (fila) de una muestra de entrenamiento, el error de predicción en esa observación ajustando un modelo lineal con todas las demás observaciones:

```
cv.lm0 <- function(formula, datos) {
    n <- nrow(datos)
    cv.res <- numeric(n)
    for (i in 1:n) {
        modelo <- lm(formula, datos[-i, ])
        cv.pred <- predict(modelo, newdata = datos[i, ])
        cv.res[i] <- cv.pred - datos[i, ]
    }
    return(cv.res)
}
```

La función anterior no es muy eficiente, pero podría modificarse fácilmente para emplear otros métodos de regresión[7]. En el caso de regresión lineal múltiple (y de otros modelos lineales), se pueden obtener fácilmente las predicciones eliminando una de las observaciones a partir del ajuste con todos los datos. Por ejemplo, en lugar de la anterior sería preferible emplear la siguiente función (ver `?rstandard`):

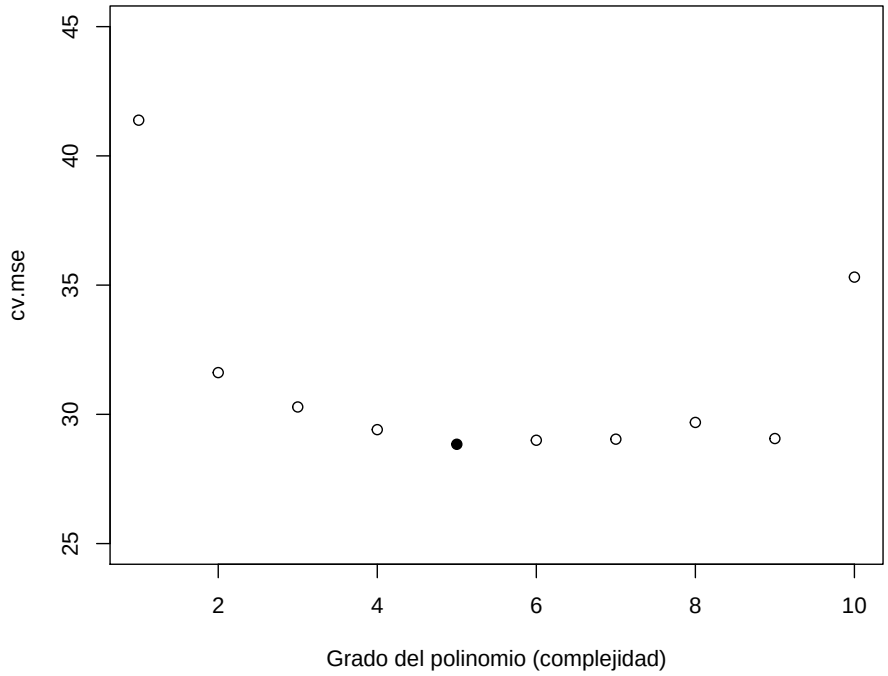
```
cv.lm <- function(formula, datos) {
    modelo <- lm(formula, datos)
    return(rstandard(modelo, type = "predictive"))
}
```

Empleando esta función, podemos calcular una medida del error de predicción de validación cruza-

[7] También puede ser de interés la función `cv.glm()` del paquete `boot`.

da (en este caso el *error cuadrático medio*) para cada valor del hiperparámetro (grado del ajuste polinómico) y seleccionar el que lo minimiza.

```
grado.max <- 10
grados <- seq_len(grado.max)
cv.mse <- cv.mse.sd <- numeric(grado.max)
for(grado in grados){
  cv.res <- cv.lm(medv ~ poly(lstat, grado), train)
  se <- cv.res^2
  cv.mse[grado] <- mean(se)
  cv.mse.sd[grado] <- sd(se)/sqrt(length(se))
}
plot(grados, cv.mse, ylim = c(25, 45),
  xlab = "Grado del polinomio (complejidad)")
# Valor óptimo
imin.mse <- which.min(cv.mse)
grado.op <- grados[imin.mse]
points(grado.op, cv.mse[imin.mse], pch = 16)
```



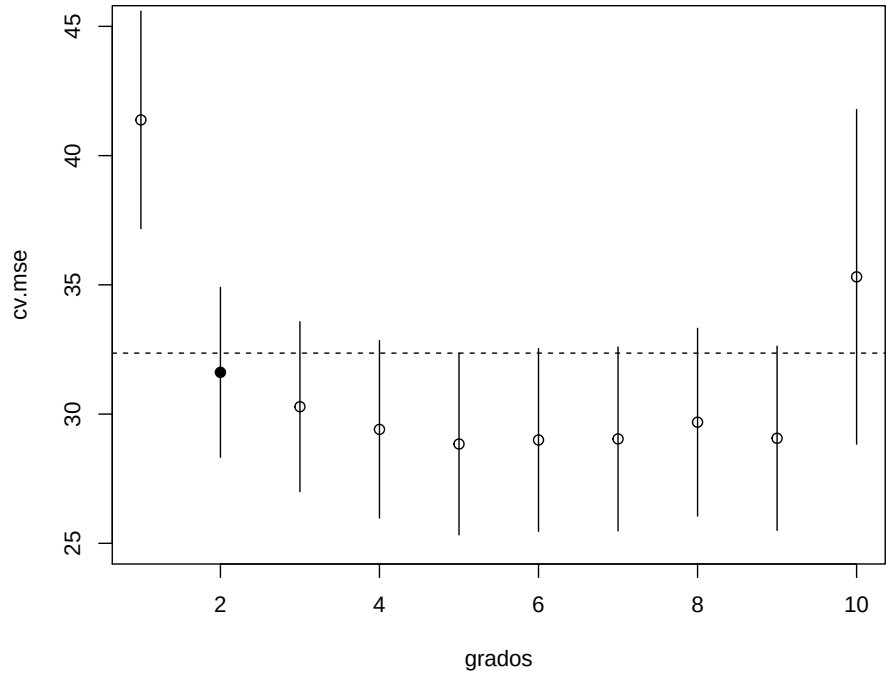

```
grado.op
```

```
## [1] 5
```

En lugar de emplear los valores óptimos de los hiperparámetros, Breiman *et al.* (1984) propusieron la regla de "un error estándar" para seleccionar la complejidad del modelo. La idea es que estamos trabajando con estimaciones de la precisión y pueden presentar variabilidad, por lo que la sugerencia es seleccionar el modelo más simple[8] dentro de un error estándar de la precisión del modelo correspondiente al valor óptimo (se consideraría que no hay diferencias significativas en la precisión; además, se mitigaría el efecto de la variabilidad debida a aleatoriedad/semilla).

[8] Suponiendo que los modelos se pueden ordenar del más simple al más complejo.

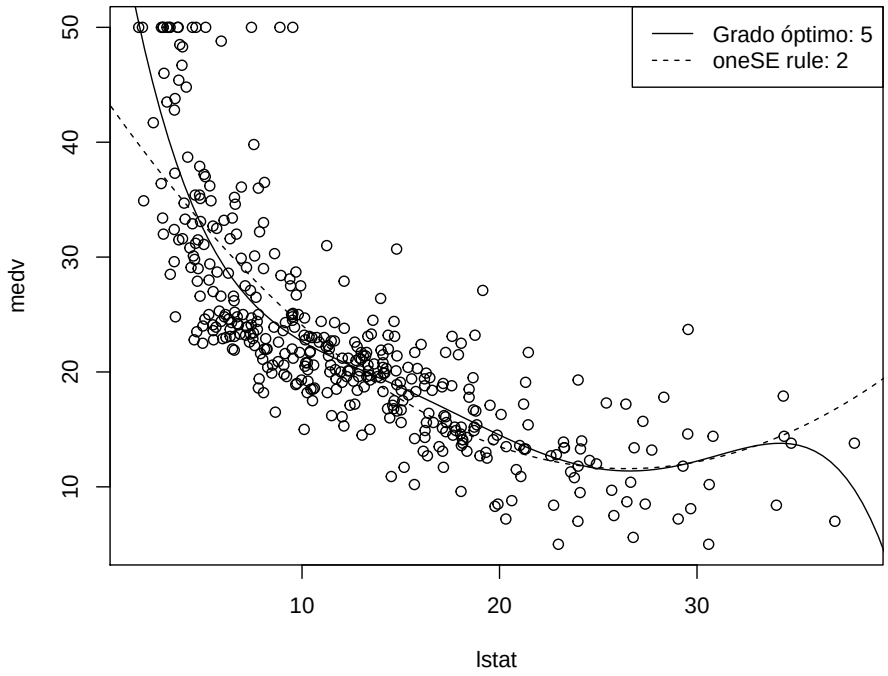
```
plot(grados, cv.mse, ylim = c(25, 45))
segments(grados, cv.mse - cv.mse.sd, grados, cv.mse + cv.mse.sd)
# Límite superior "oneSE rule" y complejidad mínima por debajo de ese valor
upper.cv.mse <- cv.mse[imin.mse] + cv.mse.sd[imin.mse]
abline(h = upper.cv.mse, lty = 2)
imin.1se <- min(which(cv.mse <= upper.cv.mse))
grado.1se <- grados[imin.1se]
points(grado.1se, cv.mse[imin.1se], pch = 16)
```




```
grado.1se
```

```
## [1] 2
```

```
plot(medv ~ lstat, data = train)
fit.op <- lm(medv ~ poly(lstat, grado.op), train)
fit.1se <- lm(medv ~ poly(lstat, grado.1se), train)
newdata <- data.frame(lstat = seq(0, 40, len = 100))
lines(newdata$lstat, predict(fit.op, newdata = newdata))
lines(newdata$lstat, predict(fit.1se, newdata = newdata), lty = 2)
legend("topright", legend = c(paste("Grado óptimo:", grado.op), paste("oneSE rule:", grado.1se)),
       lty = c(1, 2))
```

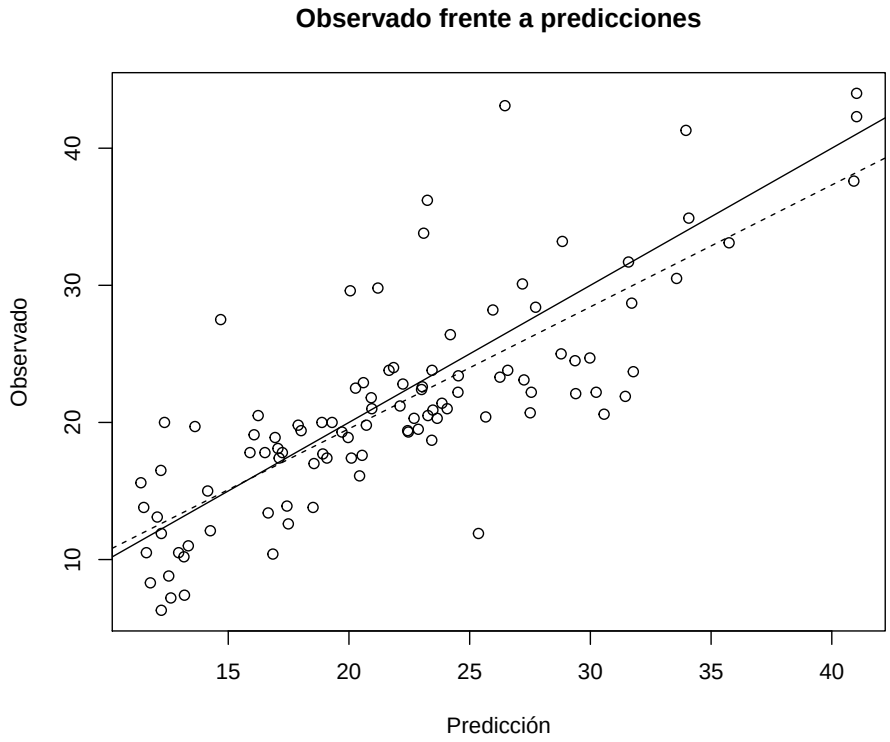


1.3.4 Evaluación de un método de regresión

Para estudiar la precisión de las predicciones de un método de regresión se evalúa el modelo en el conjunto de datos de test y se comparan las predicciones frente a los valores reales.

Si generamos un gráfico de dispersión de observaciones frente a predicciones, los puntos deberían estar en torno a la recta $y = x$ (línea continua).

```
obs <- test$medv
pred <- predict(fit.op, newdata = test)

plot(pred, obs, main = "Observado frente a predicciones",
     xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
res <- lm(obs ~ pred)
# summary(res)
abline(res, lty = 2)
```



También es habitual calcular distintas medidas de error. Por ejemplo, podríamos emplear la función `postResample()` del paquete `caret`:

```
caret::postResample(pred, obs)
```

```
##      RMSE  Rsquared       MAE
## 4.8526718 0.6259583 3.6671847
```

La función anterior, además de las medidas de error habituales (que dependen en su mayoría de la escala de la variable respuesta) calcula un *pseudo R-cuadrado*. En este paquete (también en `rattle`) se emplea uno de los más utilizados, el cuadrado del coeficiente de correlación entre las predicciones y los valores observados (que se corresponde con la línea discontinua en la figura anterior). Estos valores se interpretarían como el coeficiente de determinación en regresión lineal, debería ser próximo a 1. Hay otras alternativas (ver Kvålseth, 1985), pero la idea es que deberían medir la proporción de variabilidad de la respuesta explicada por el modelo, algo que en general no es cierto con el anterior[9]. La recomendación sería emplear:

$$\tilde{R}^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}$$

implementado junto con otras medidas en la siguiente función:

```
accuracy <- function(pred, obs, na.rm = FALSE,
                     tol = sqrt(.Machine$double.eps)) {
  err <- obs - pred     # Errores
  if(na.rm) {
    is.a <- !is.na(err)
    err <- err[is.a]
    obs <- obs[is.a]
  }
  perr <- 100*err/pmax(obs, tol)  # Errores porcentuales
  return(c(
```

[9]Por ejemplo obtendríamos el mismo valor si desplazamos las predicciones sumando una constante (i.e. no tiene en cuenta el sesgo).

```
    me = mean(err),           # Error medio
    rmse = sqrt(mean(err^2)), # Raíz del error cuadrático medio
    mae = mean(abs(err)),     # Error absoluto medio
    mpe = mean(perr),         # Error porcentual medio
    mape = mean(abs(perr)),   # Error porcentual absoluto medio
    r.squared = 1 - sum(err^2)/sum((obs - mean(obs))^2) # Pseudo R-cuadrado
  ))
}
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
## -0.6731294  4.8526718  3.6671847 -8.2322506 19.7097373  0.6086704
```

```
accuracy(predict(fit.1se, newdata = test), obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
## -0.9236280  5.2797360  4.1252053 -9.0029771 21.6512406  0.5367608
```

Ejercicio 1.1.

Considerando de nuevo el ejemplo anterior, particionar la muestra en datos de entrenamiento (70%), de validación (15%) y de test (15%), para entrenar los modelos polinómicos, seleccionar el grado óptimo (el hiperparámetro) y evaluar las predicciones del modelo final, respectivamente.

Podría ser de utilidad el siguiente código (basado en la aproximación de `rattle`), que particiona los datos suponiendo que están almacenados en el data.frame `df`:

```
df <- Boston
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.7 * nobs)
inotrain <- setdiff(seq_len(nobs), itrain)
ivalidate <- sample(inotrain, 0.15 * nobs)
itest <- setdiff(inotrain, ivalidate)
train <- df[itrain, ]
validate <- df[ivalidate, ]
test <- df[itest, ]
```

Alternativamente podríamos emplear la función `split()` creando un factor que divida aleatoriamente los datos en tres grupos (versión "simplificada" de una propuesta en este post):

```
set.seed(1)
p <- c(train = 0.7, validate = 0.15, test = 0.15)
f <- sample( rep(factor(seq_along(p), labels = names(p)),
                 times = nrow(df)*p/sum(p)) )
samples <- suppressWarnings(split(df, f))
str(samples)
```

```
## List of 3
##  $ train   :'data.frame':    356 obs. of  14 variables:
##   ..$ crim   : num [1:356] 0.00632 0.02731 0.02729 0.02985 0.08829 ...
##   ..$ zn     : num [1:356] 18 0 0 0 12.5 12.5 12.5 12.5 12.5 0 ...
##   ..$ indus  : num [1:356] 2.31 7.07 7.07 2.18 7.87 7.87 7.87 7.87 7.87 8.14 ...
##   ..$ chas   : int [1:356] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ nox    : num [1:356] 0.538 0.469 0.469 0.458 0.524 0.524 0.524 0.524 0.524 0.538 ...
##   ..$ rm     : num [1:356] 6.58 6.42 7.18 6.43 6.01 ...
##   ..$ age    : num [1:356] 65.2 78.9 61.1 58.7 66.6 100 85.9 82.9 39 56.5 ...
##   ..$ dis    : num [1:356] 4.09 4.97 4.97 6.06 5.56 ...
##   ..$ rad    : int [1:356] 1 2 2 3 5 5 5 5 5 4 ...
##   ..$ tax    : num [1:356] 296 242 242 222 311 311 311 311 311 307 ...
```

```
##   ..$ ptratio: num [1:356] 15.3 17.8 17.8 18.7 15.2 15.2 15.2 15.2 15.2 21 ...
##   ..$ black  : num [1:356] 397 397 393 394 396 ...
##   ..$ lstat  : num [1:356] 4.98 9.14 4.03 5.21 12.43 ...
##   ..$ medv   : num [1:356] 24 21.6 34.7 28.7 22.9 16.5 18.9 18.9 21.7 19.9 ...
##  $ validate:'data.frame':    75 obs. of  14 variables:
##   ..$ crim   : num [1:75] 0.0324 0.6298 0.9884 0.9558 1.0025 ...
##   ..$ zn     : num [1:75] 0 0 0 0 0 0 0 75 75 0 ...
##   ..$ indus  : num [1:75] 2.18 8.14 8.14 8.14 8.14 8.14 5.96 2.95 2.95 6.91 ...
##   ..$ chas   : int [1:75] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ nox    : num [1:75] 0.458 0.538 0.538 0.538 0.538 0.538 0.499 0.428 0.428 0.448 ...
##   ..$ rm     : num [1:75] 7 5.95 5.81 6.05 6.67 ...
##   ..$ age    : num [1:75] 45.8 61.8 100 88.8 87.3 95 41.5 21.8 15.8 6.5 ...
##   ..$ dis    : num [1:75] 6.06 4.71 4.1 4.45 4.24 ...
##   ..$ rad    : int [1:75] 3 4 4 4 4 4 5 3 3 3 ...
##   ..$ tax    : num [1:75] 222 307 307 307 307 307 279 252 252 233 ...
##   ..$ ptratio: num [1:75] 18.7 21 21 21 21 21 19.2 18.3 18.3 17.9 ...
##   ..$ black  : num [1:75] 395 397 395 306 380 ...
##   ..$ lstat  : num [1:75] 2.94 8.26 19.88 17.28 11.98 ...
##   ..$ medv   : num [1:75] 33.4 20.4 14.5 14.8 21 13.1 21 30.8 34.9 24.7 ...
##  $ test    :'data.frame':    75 obs. of  14 variables:
##   ..$ crim   : num [1:75] 0.069 0.1446 0.2249 0.638 0.6719 ...
##   ..$ zn     : num [1:75] 0 12.5 12.5 0 0 0 0 90 0 0 ...
##   ..$ indus  : num [1:75] 2.18 7.87 7.87 8.14 8.14 ...
##   ..$ chas   : int [1:75] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ nox    : num [1:75] 0.458 0.524 0.524 0.538 0.538 0.538 0.499 0.403 0.413 0.413 ...
##   ..$ rm     : num [1:75] 7.15 6.17 6.38 6.1 5.81 ...
##   ..$ age    : num [1:75] 54.2 96.1 94.3 84.5 90.3 94.1 68.2 21.9 6.6 7.8 ...
##   ..$ dis    : num [1:75] 6.06 5.95 6.35 4.46 4.68 ...
##   ..$ rad    : int [1:75] 3 5 5 4 4 4 5 5 4 4 ...
##   ..$ tax    : num [1:75] 222 311 311 307 307 307 279 226 305 305 ...
##   ..$ ptratio: num [1:75] 18.7 15.2 15.2 21 21 21 19.2 17.9 19.2 19.2 ...
##   ..$ black  : num [1:75] 397 397 393 380 377 ...
##   ..$ lstat  : num [1:75] 5.33 19.15 20.45 10.26 14.81 ...
##   ..$ medv   : num [1:75] 36.2 27.1 15 18.2 16.6 12.7 18.9 35.4 24.2 22.8 ...
```

1.3.5 Evaluación de un método de clasificación

Para estudiar la eficiencia de un método de clasificación supervisada típicamente se obtienen las predicciones para el conjunto de datos de test y se genera una tabla de contingencia, denominada *matriz de confusión*, con las predicciones frente a los valores reales.

En primer lugar consideraremos el caso de dos categorías. La matriz de confusión será de la forma:

Observado\Predicción	Positivo	Negativo
Verdadero	Verdaderos positivos (TP)	Falsos negativos (FN)
Falso	Falsos positivos (FP)	Verdaderos negativos (TN)

A partir de esta tabla se pueden obtener distintas medidas de la precisión de las predicciones. Por ejemplo, dos de las más utilizadas son la tasa de verdaderos positivos y la de verdaderos negativos (tasas de acierto en positivos y negativos), también denominadas *sensibilidad* y *especificidad*:

- Sensibilidad (*sensitivity*, *recall*, *hit rate*, *true positive rate*; TPR):

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

- Especificidad (*specificity*, *true negative rate*; TNR):

$$TNR = \frac{TN}{TN + FP}$$

La precisión global o tasa de aciertos (*accuracy*; ACC) sería:

$$ACC = \frac{TP + TN}{P + N} = \frac{TP + TN}{TP + TN + FP + FN}$$

Sin embargo hay que tener cuidado con esta medida cuando las clases no están balanceadas. Otras medidas de la precisión global que tratan de evitar este problema son la *precisión balanceada* (*balanced accuracy*, BA):

$$BA = \frac{TPR + TNR}{2}$$

(media aritmética de TPR y TNR) o la *puntuación F1* (*F1 score*; media armónica de TPR y el valor predictivo positivo, PPV, descrito más adelante):

$$F_1 = \frac{2TP}{2TP + FP + FN}$$

Otra medida global es el coeficiente kappa de Cohen, que compara la tasa de aciertos con la obtenida en una clasificación al azar (un valor de 1 indicaría máxima precisión y 0 que la precisión es igual a la que obtendríamos clasificando al azar; empleando la tasa de positivos, denominada *prevalencia*, para predecir positivo).

También hay que tener cuidado las medidas que utilizan como estimación de la probabilidad de positivo (*prevalencia*) la tasa de positivos en la muestra de test, como el valor (o índice) predictivo positivo (*precision*, *positive predictive value*; PPV):

$$PPV = \frac{TP}{TP + FP}$$

(que no debe ser confundido con la precisión global ACC) y el valor predictivo negativo negativo (NPV):

$$NPV = \frac{TN}{TN + FN},$$

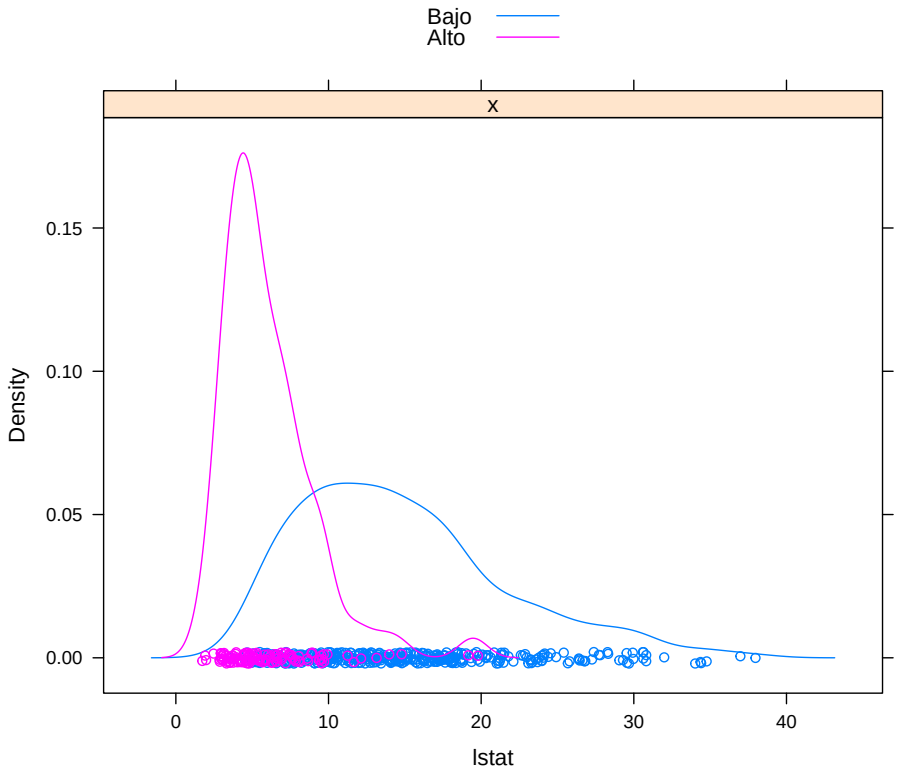
si la muestra de test no refleja lo que ocurre en la población (por ejemplo si la clase de interés está sobrerrepresentada en la muestra). En estos casos habrá que recalcularlos empleando estimaciones válidas de las probabilidades de la clases (por ejemplo, en estos casos, la función `caret::confusionMatrix()` permite establecer estimaciones válidas mediante el argumento `prevalence`).

Como ejemplo emplearemos los datos anteriores de valoraciones de viviendas y estatus de la población, considerando como respuesta una nueva variable `fmedv` que clasifica las valoraciones en "Bajo" o "Alto" dependiendo de si `medv > 25`.

```
# data(Boston, package = "MASS")
datos <- Boston
datos$fmedv <- factor(datos$medv > 25, labels = c("Bajo", "Alto")) # levels = c('FALSE', 'TRUE')
# En este caso las clases no están balanceadas
table(datos$fmedv)
```

```
##
## Bajo Alto
##  382  124
```

```
caret::featurePlot(datos$lstat, datos$fmedv, plot = "density",
            labels = c("lstat", "Density"), auto.key = TRUE)
```



El siguiente código realiza la partición de los datos y posteriormente ajusta un modelo de regresión logística en la muestra de entrenamiento considerando `lstat` como única variable explicativa (en el Capítulo 5 se darán más detalles sobre este tipo de modelos):

```
# Particionado de los datos
set.seed(1)
nobs <- nrow(datos)
itrain <- sample(nobs, 0.8 * nobs)
train <- datos[itrain, ]
test <- datos[-itrain, ]
# Ajuste modelo
modelo <- glm(fmedv ~ lstat, family = binomial, data = train)
summary(modelo)
```

```
##
## Call:
## glm(formula = fmedv ~ lstat, family = binomial, data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.9749  -0.4161  -0.0890   0.3785   3.6450
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  3.74366    0.47901   7.815 5.48e-15 ***
## lstat       -0.54231    0.06134  -8.842  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

```
##     Null deviance: 460.84  on 403  degrees of freedom
## Residual deviance: 243.34  on 402  degrees of freedom
## AIC: 247.34
##
## Number of Fisher Scoring iterations: 7
```

En este caso podemos obtener las estimaciones de la probabilidad de la segunda categoría empleando `predict()` con `type = "response"`, a partir de las cuales podemos establecer las predicciones como la categoría más probable:

```
obs <- test$fmedv
p.est <- predict(modelo, type = "response", newdata = test)
pred <- factor(p.est > 0.5, labels = c("Bajo", "Alto")) # levels = c('FALSE', 'TRUE')
```

Finalmente podemos obtener la matriz de confusión con el siguiente código:

```
tabla <- table(obs, pred)
# addmargins(tabla, FUN = list(Total = sum))
tabla
```

```
##       pred
## obs    Bajo Alto
##   Bajo   71   11
##   Alto    8   12
```

```
# Porcentajes respecto al total
print(100*prop.table(tabla), digits = 2)
```

```
##       pred
## obs    Bajo Alto
##   Bajo 69.6 10.8
##   Alto  7.8 11.8
```

```
# Porcentajes (de aciertos y fallos) por categorías
print(100*prop.table(tabla, 1), digits = 3)
```

```
##       pred
## obs    Bajo Alto
##   Bajo 86.6 13.4
##   Alto 40.0 60.0
```

Alternativamente se podría emplear la función `confusionMatrix()` del paquete `caret` que permite obtener distintas medidas de la precisión:

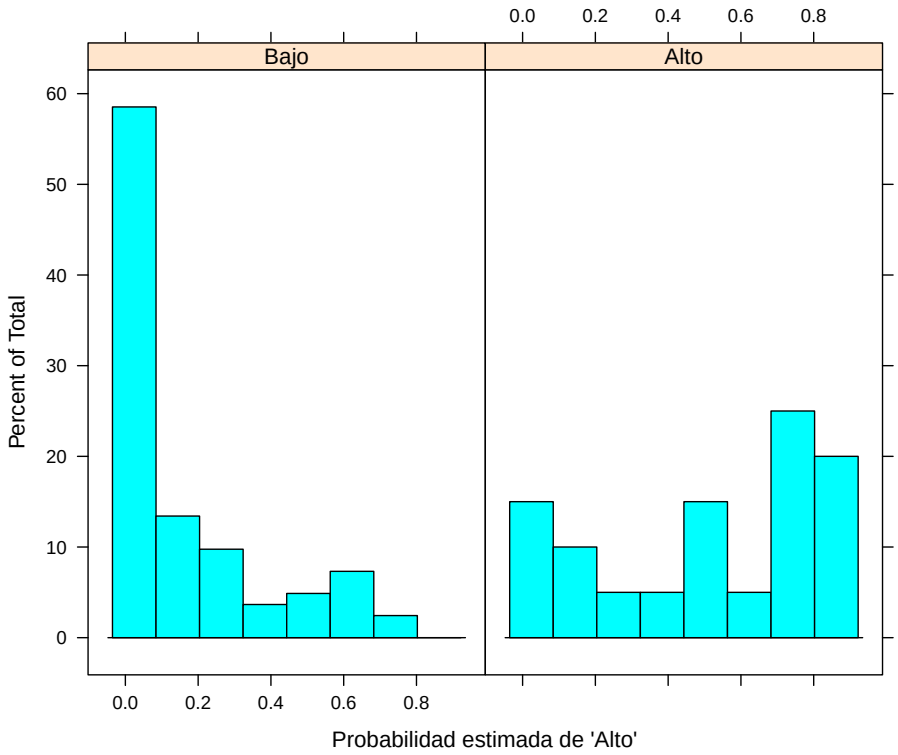
```
caret::confusionMatrix(pred, obs, positive = "Alto", mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Bajo Alto
##       Bajo   71    8
##       Alto   11   12
##
##                Accuracy : 0.8137
##                  95% CI : (0.7245, 0.884)
##     No Information Rate : 0.8039
##     P-Value [Acc > NIR] : 0.4604
##
##                   Kappa : 0.4409
##
##  Mcnemar's Test P-Value : 0.6464
```

```
##
##             Sensitivity : 0.6000
##             Specificity : 0.8659
##          Pos Pred Value : 0.5217
##          Neg Pred Value : 0.8987
##               Precision : 0.5217
##                  Recall : 0.6000
##                      F1 : 0.5581
##              Prevalence : 0.1961
##          Detection Rate : 0.1176
##    Detection Prevalence : 0.2255
##       Balanced Accuracy : 0.7329
##
##        'Positive' Class : Alto
##
```

Si el método de clasificación proporciona estimaciones de las probabilidades de las categorías, disponemos de más información en la clasificación que también podemos emplear en la evaluación del rendimiento. Por ejemplo, se puede realizar un analisis descriptivo de las probabilidades estimadas y las categorías observadas en la muestra de test:

```
# Imitamos la función caret::plotClassProbs()
library(lattice)
histogram(~ p.est | obs, xlab = "Probabilidad estimada de 'Alto'")
```



Para evaluar las estimaciones de las probabilidades se suele emplear la curva ROC (*receiver operating characteristics*, característica operativa del receptor; diseñada inicialmente en el campo de la detección de señales). Como ya se comentó, normalmente se emplea $c = 0.5$ como punto de corte para clasificar en la categoría de interés (es lo que se conoce como *regla de Bayes*), aunque se podrían considerar otros valores (por ejemplo para mejorar la clasificación en una de las categorías, a costa de empeorar la precisión global). En la curva ROC se representa la sensibilidad (TPR) frente a la tasa de falsos negativos (FNR = 1 - TNR = 1 - especificidad) para distintos valores de corte. Para ello se puede

emplear el paquete `pROC`:

```
library(pROC)
roc_glm <- roc(response = obs, predictor = p.est)
# View((as.data.frame(roc_glm[2:4])))
plot(roc_glm)
```

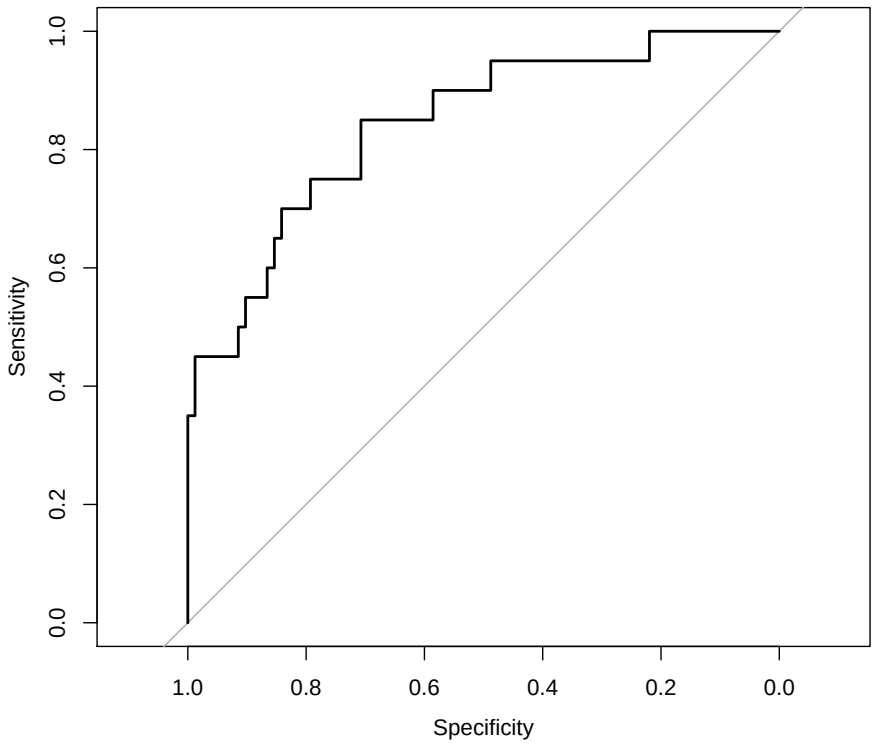


Figura 1.5: Curva ROC correspondiente al modelo de regresión logística.

```
# plot(roc_glm, legacy.axes = TRUE)
```

Lo ideal sería que la curva se aproximase a la esquina superior izquierda (máxima sensibilidad y especificidad). La recta diagonal se correspondería con un clasificador aleatorio. Una medida global del rendimiento del clasificador es el área bajo la curva ROC (AUC; equivalente al estadístico U de Mann-Whitney o al índice de Gini). Un clasificador perfecto tendría un valor de 1 y 0.5 uno aleatorio.

```
# roc_glm$auc
roc_glm
```

```
##
## Call:
## roc.default(response = obs, predictor = p.est)
##
## Data: p.est in 82 controls (obs Bajo) < 20 cases (obs Alto).
## Area under the curve: 0.8427
```

```
ci.auc(roc_glm)
```

```
## 95% CI: 0.7428-0.9426 (DeLong)
```

Como comentario adicional, aunque se puede modificar el punto de corte para mejorar la clasificación en la categoría de interés (de hecho, algunas herramientas como `h2o` lo modifican por defecto; en este caso concreto para maximizar F_1 en la muestra de entrenamiento), muchos métodos de clasificación

(como los basados en árboles descritos en el Capítulo 2) admiten como opción una matriz de pérdidas que se tendrá en cuenta para medir la eficiencia durante el aprendizaje y normalmente esta sería la aproximación recomendada.

En el caso de más de dos categorías podríamos generar una matriz de confusión de forma análoga, aunque en este caso en principio solo podríamos calcular medidas globales de la precisión como la tasa de aciertos o el coeficiente kappa de Cohen. Podríamos obtener también medidas por clase, como la sensibilidad y la especificidad, siguiendo la estrategia "uno contra todos" descrita en la Sección 1.2.1. Esta aproximación es la que sigue la función `confusionMatrix()` del paquete `caret` (devuelve las medidas comparando cada categoría con las restantes en el componente `$byClass`).

Como ejemplo ilustrativo consideraremos el conocido conjunto de datos `iris` (Fisher, 1936) en el que el objetivo es clasificar flores de lirio en tres especies (`Species`) a partir del largo y ancho de sépalos y pétalos, aunque en este caso emplearemos un clasificador aleatorio.

```
data(iris)
# Partición de los datos
datos <- iris
set.seed(1)
nobs <- nrow(datos)
itrain <- sample(nobs, 0.8 * nobs)
train <- datos[itrain, ]
test <- datos[-itrain, ]
# Entrenamiento
prevalences <- table(train$Species)/nrow(train)
prevalences
```

```
##
##     setosa versicolor  virginica
##  0.3250000  0.3166667  0.3583333
```

```
# Calculo de las predicciones
levels <- names(prevalences) # levels(train$Species)
f <- factor(levels, levels = levels) # factor(levels) valdría en este caso al estar por orden alfabé
pred.rand <- sample(f, nrow(test), replace = TRUE, prob = prevalences)
# Evaluación
caret::confusionMatrix(pred.rand, test$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa          3          3         1
##   versicolor      4          2         5
##   virginica       4          7         1
##
## Overall Statistics
##
##                Accuracy : 0.2
##                  95% CI : (0.0771, 0.3857)
##     No Information Rate : 0.4
##     P-Value [Acc > NIR] : 0.9943
##
##                   Kappa : -0.1862
##
##  Mcnemar's Test P-Value : 0.5171
##
## Statistics by Class:
##
```

```
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 0.2727           0.16667          0.14286
## Specificity                 0.7895           0.50000          0.52174
## Pos Pred Value              0.4286           0.18182          0.08333
## Neg Pred Value              0.6522           0.47368          0.66667
## Prevalence                  0.3667           0.40000          0.23333
## Detection Rate              0.1000           0.06667          0.03333
## Detection Prevalence        0.2333           0.36667          0.40000
## Balanced Accuracy           0.5311           0.33333          0.33230
```

1.4 La maldición de la dimensionalidad

Podríamos pensar que al aumentar el número de variables explicativas se mejora la capacidad predictiva de los modelos. Lo cual, en general, sería cierto si realmente los predictores fuesen de utilidad para explicar la respuesta. Sin embargo, al aumentar el número de dimensiones se pueden agravar notablemente muchos de los problemas que ya pueden aparecer en dimensiones menores, esto es lo que se conoce como la *maldición de la dimensionalidad* (*curse of dimensionality*, Bellman, 1961).

Uno de estos problemas es el denominado *efecto frontera* que ya puede aparecer en una dimensión, especialmente al trabajar con modelos flexibles (como ajustes polinómicos con grados altos o los métodos locales que trataremos en el Capítulo 6). La idea es que en la "frontera" del rango de valores de una variable explicativa vamos a disponer de pocos datos y los errores de predicción van a tener gran variabilidad (se están haciendo extrapolaciones de los datos, más que interpolaciones, y van a ser menos fiables).

Como ejemplo consideraremos un problema de regresión simple, con un conjunto de datos simulados (del proceso ya considerado en la Sección 1.3.1) con 100 observaciones (que ya podríamos considerar que no es muy pequeño).

```r
# Simulación datos
n <- 100
x <- seq(0, 1, length = n)
mu <- 2 + 4*(5*x - 1)*(4*x - 2)*(x - 0.8)^2 # grado 4
sd <- 0.5
set.seed(1)
y <- mu + rnorm(n, 0, sd)
datos <- data.frame(x = x, y = y)
plot(x, y)
lines(x, mu, lwd = 2, col = "lightgray")
```

Cuando el número de datos es más o menos grande podríamos pensar en predecir la respuesta a partir de lo que ocurre en las observaciones cercanas a la posición de predicción, esta es la idea de los métodos locales (Capítulo 6). Uno de los métodos de este tipo más conocidos es el de los *k-vecinos más cercanos* (*k-nearest neighbors*; KNN). Se trata de un método muy simple, pero que puede ser muy efectivo, que se basa en la idea de que localmente la media condicional (la predicción óptima) es constante. Concretamente, dados un entero k (hiperparámetro) y un conjunto de entrenamiento $\mathcal{T}$, para obtener la predicción correspondiente a un vector de valores de las variables explicativas $\mathbf{x}$, el método de regresión[10] KNN promedia las obsevaciones en un vecindario $\mathcal{N}_k(\mathbf{x}, \mathcal{T})$ formado por las k observaciones más cercanas a $\mathbf{x}$:

$$\hat{Y}(\mathbf{x}) = \hat{m}(\mathbf{x}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}, \mathcal{T})} Y_i$$

(sería necesario definir una distancia, normalmente la distancia euclídea de los predictores estandarizados).

[10] En el caso de clasificación se considerarían las variables indicadoras de las categorías y se obtendrían las frecuencias relativas en el vecindario como estimaciones de las probabilidades de las clases.

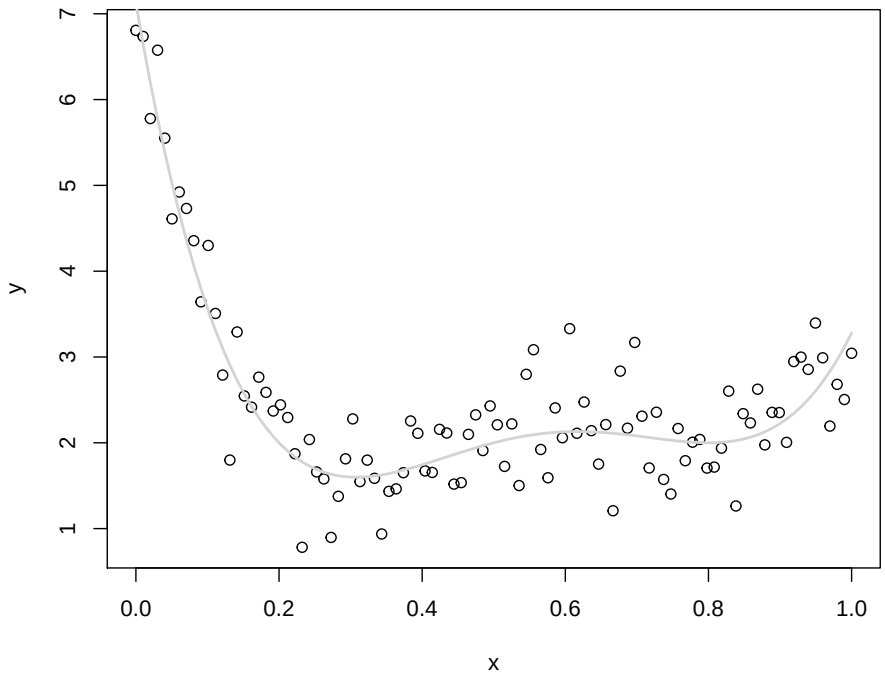


Figura 1.6: Muestra simulada y tendencia teórica.

Este método está implementado en numerosos paquetes, por ejemplo en la función `knnreg()` del paquete `caret`:

```
library(caret)

# Ajuste de los modelos
fit1 <- knnreg(y ~ x, data = datos, k = 5) # 5 observaciones más cercanas (5% de los datos)
fit2 <- knnreg(y ~ x, data = datos, k = 10)
fit3 <- knnreg(y ~ x, data = datos, k = 20)

plot(x, y)
lines(x, mu, lwd = 2, col = "lightgray")
newdata <- data.frame(x = x)
lines(x, predict(fit1, newdata), lwd = 2, lty = 3)
lines(x, predict(fit2, newdata), lwd = 2, lty = 2)
lines(x, predict(fit3, newdata), lwd = 2)
legend("topright", legend = c("Verdadero", "5-NN", "10-NN", "20-NN"),
       lty = c(1, 3, 2, 1), lwd = 2, col = c("lightgray", 1, 1, 1))
```

A medida que aumenta k disminuye la complejidad del modelo y se observa un incremento del efecto frontera. Habría que seleccionar un valor óptimo de k (buscando un equilibro entre sesgo y varianza, como se mostró en la Sección 1.3.1 y se ilustrará en la última sección de este capítulo empleando este método con el paquete `caret`), que dependerá de la tendencia teórica y del número de datos. En este caso, para $k = 5$, podríamos pensar que el efecto frontera aparece en el 10% más externo del rango de la variable explicativa (con un número mayor de datos podría bajar al 1%). Al aumentar el número de variables explicativas, considerando que el 10% más externo del rango de cada una de ellas constituye la "frontera" de los datos, tendríamos que la proporción de frontera sería $1 - 0.9^d$, siendo d el número de dimensiones. Lo que se traduce que con $d = 10$ el 65% del espacio predictivo sería frontera y en

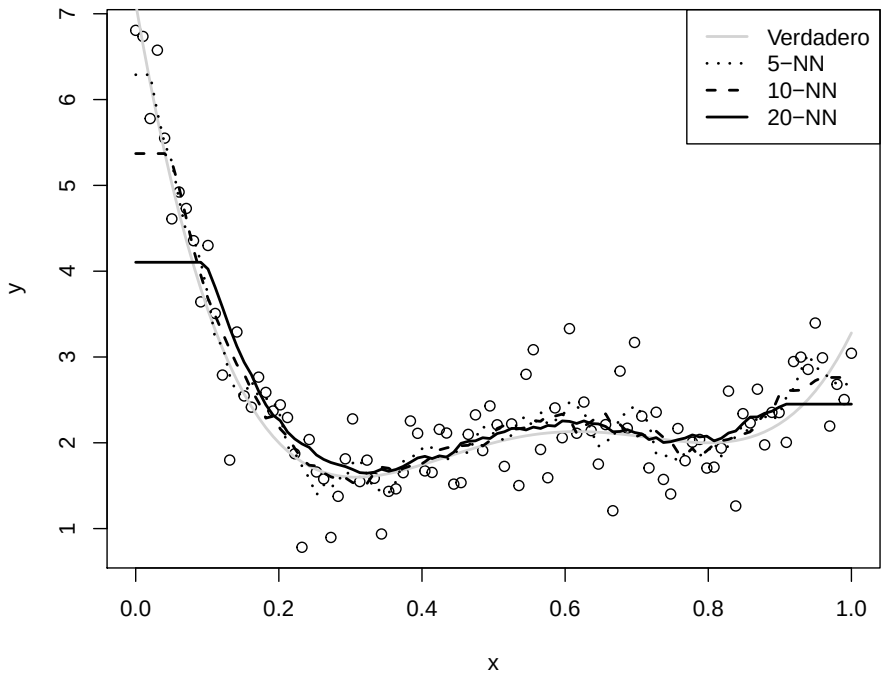
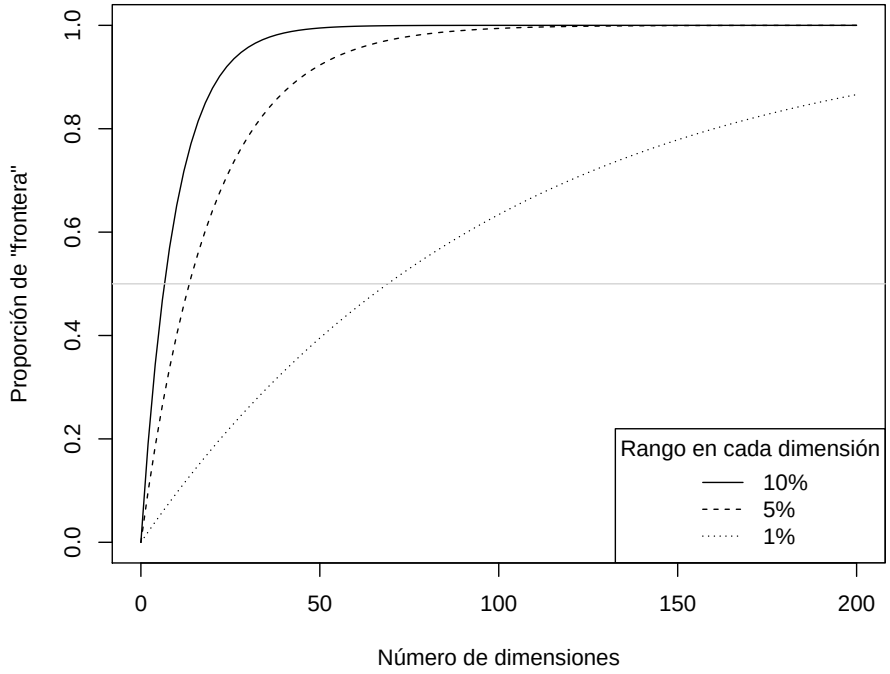


Figura 1.7: Predicciones con el método KNN y distintos vecindarios

torno al 88% para $d = 20$, es decir, al aumentar el número de dimensiones el problema del efecto frontera será generalizado.

```
curve(1 - 0.9^x, 0, 200, ylab = 'Proporción de "frontera"', xlab = 'Número de dimensiones')
curve(1 - 0.95^x, lty = 2, add = TRUE)
curve(1 - 0.99^x, lty = 3, add = TRUE)
abline(h = 0.5, col = "lightgray")
legend("bottomright", title = "Rango en cada dimensión", legend = c("10%" , "5%", "1%"),
       lty = c(1, 2, 3))
```



Desde otro punto de vista, suponiendo que los predictores se distribuyen de forma uniforme, la densidad de las observaciones es proporcional a $n^{1/d}$, siendo n el tamaño muestral. Por lo que si consideramos que una muestra de tamaño $n = 100$ es suficientemente densa en una dimensión, para obtener la misma densidad muestral en 10 dimensiones tendríamos que disponer de un tamaño muestral de $n = 100^{10} = 10^{20}$. Por tanto, cuando el número de dimensiones es grande no va a haber muchas observaciones en el entorno de la posición de predicción y puede haber serios problemas de sobreajuste si se pretende emplear un modelo demasiado flexible (por ejemplo KNN con k pequeño). Hay que tener en cuenta que, en general, fijado el tamaño muestral, la flexibilidad de los modelos aumenta al aumentar el número de dimensiones del espacio predictivo.

Para concluir, otro de los problemas que se agravan notablemente al aumentar el número de dimensiones es el de colinealidad (o concurvidad) que puede producir que muchos métodos (como los modelos lineales o las redes neuronales) sean muy poco eficientes o inestables (llegando incluso a que no se puedan aplicar), además de que complica notablemente la interpretación de cualquier método. Esto está relacionado también con la dificultad para determinar que variables son de interés para predecir la respuesta (i.e. no son ruido). Debido a la aleatoriedad, predictores que realmente no están relacionados con la respuesta pueden ser tenidos en cuenta por el modelo con mayor facilidad (KNN con las opciones habituales tiene en cuenta todos los predictores con el mismo peso). Lo que resulta claro es que si se agrega ruido se producirá un incremento en el error de predicción. Incluso si las variables añadidas resultan de interés, si el número de observaciones es pequeño en comparación, el incremento en la variabilidad de las predicciones puede no compensar la disminución del sesgo de predicción.

Como conclusión, en el caso multidimensional habrá que tratar de emplear métodos que minimicen estos problemas.

1.5 Análisis e interpretación de los modelos

El análisis e interpretación de modelos es un campo muy activo en AE/ML, para el que recientemente se ha acuñado el término de *interpretable machine learning* (IML). A continuación se resumen brevemente algunas de las principales ideas, para más detalles ver por ejemplo Molnar (2020).

Como ya se comentó, a medida que aumenta la complejidad de los modelos generalmente disminuye su interpretabilidad, por lo que normalmente interesa encontrar el modelo más simple posible que resulte de utilidad para los objetivos propuestos. Aunque el principal objetivo sea la predicción, una vez obtenido el modelo final suele interesar medir la importancia de cada predictor en el modelo y si es posible como influyen en la predicción de la respuesta, es decir, estudiar el efecto de las variables explicativas. Esto puede presentar serias dificultades especialmente en modelos complejos en los que hay interacciones entre los predictores (el efecto de una variable explicativa depende de los valores de otras).

La mayoría de los métodos de aprendizaje supervisado permiten obtener medidas de la importancia de las variables explicativas en la predicción (ver p.e. la ayuda de la función `caret::varImp()`; algunos, como los basados en árboles, incluso de las no incluidas en el modelo final). Muchos de los métodos de clasificación, en lugar de proporcionar medidas globales, calculan medidas para cada categoría. Alternativamente también se pueden obtener medidas de la importancia de las variables mediante procedimientos generales (en el sentido de que se pueden aplicar a cualquier modelo), pero suelen requerir de mucho más tiempo de computación (ver p.e. Molnar, 2020, Capítulo 5).

En algunos de los métodos se modela explícitamente los efectos de los distintos predictores y estos se pueden analizar con (mas o menos) facilidad. Hay que tener en cuenta que, al margen de las interacciones, la colinealidad/concurvidad dificulta notablemente el estudio de los efectos de las variables explicativas. Otros métodos son más del tipo "caja negra" (*black box*) y precisan de aproximaciones más generales, como los gráficos PDP (*Partial Dependence Plots*; Friedman y Popescu, 2008; ver también Greenwell, 2017) o las curvas ICE (*Individual Conditional Expectation*; Goldstein *et al.* , 2015). Estos métodos[11] tratan de estimar el efecto marginal de las variables explicativas, es decir, la variación en la predicción a medida que varía una variable explicativa manteniendo constantes el resto. La principal diferencia entre ambas aproximaciones es que los gráficos PDP muestran una única curva con el promedio de la respuesta mientras que las curvas ICE muestran una curva para cada observación (para más detalles ver las referencias anteriores).

En problemas de clasificación también se están empleando la teoría de juegos cooperativos y las técnicas de optimización de Investigación Operativa para evaluar la importancia de las variables predictoras y determinar las más influyentes. Por citar algunos, Strumbelj y Kononenko (2010) propusieron un procedimiento general basado en el valor de Shapley de juegos cooperativos, y en Agor y Özaltın (2019) se propone el uso de algoritmos genéticos para determinar los predictores más influyentes.

Paquetes y funciones de R:

- `pdp`: Partial Dependence Plots

  (también implementa curvas ICE y es compatible con `caret`)
- `iml`: Interpretable Machine Learning
- `DALEX`: moDel Agnostic Language for Exploration and eXplanation
- `lime`: Local Interpretable Model-Agnostic Explanations
- `vip`: Variable Importance Plots
- `caret::varImp()`, `h2o::h2o.partialPplot()`…

En los siguientes capítulos se mostrarán ejemplos empleando algunas de estas herramientas.

1.6 Introducción al paquete `caret`

Como ya se comentó en la Sección 1.2.2, el paquete `caret` (abreviatura de *Classification And REgression Training*) proporciona una interfaz unificada que simplifica el proceso de modelado empleando la mayoría de los métodos de AE implementados en R (actualmente admite 238 métodos; ver el Capítulo

[11] Similares a los gráficos parciales de residuos de los modelos lineales o aditivos (ver p.e. las funciones `termplot()`, `car::crPlots()` o `car::avPlots()`).

6 del manual de este paquete). Además de proporcionar rutinas para los principales pasos del proceso, incluye también numerosas funciones auxiliares que permitirían implementar nuevos procedimientos.

Enlaces:

- Manual
  - 3. Pre-Processing
  - 5. Model Training and Tuning
  - 6. Available Models
  - 17. Measuring Performance
- Vignette
- Cheat Sheet

La función principal es `train()` (descrita más adelante), que incluye un parámetro `method` que permite establecer el modelo mediante una cadena de texto. Podemos obtener información sobre los modelos disponibles con las funciones `getModelInfo()` y `modelLookup()` (puede haber varias implementaciones del mismo método con distintas configuraciones de hiperparámetros; también se pueden definir nuevos modelos, ver el Capítulo 13 del manual).

```
library(caret)
str(names(getModelInfo())) # Listado de todos los métodos disponibles
```

```
##  chr [1:238] "ada" "AdaBag" "AdaBoost.M1" "adaboost" "amdai" "ANFIS" ...
```

```
# names(getModelInfo("knn", regex = TRUE)) # Por defecto devuelve coincidencias parciales
modelLookup("knn")  # Información sobre hiperparámetros
```

```
##   model parameter      label forReg forClass probModel
## 1   knn         k #Neighbors   TRUE     TRUE      TRUE
```

Este paquete permite, entre otras cosas:

- Partición de los datos
  - `createDataPartition(y, p = 0.5, list = TRUE, ...)`: crea particiones balanceadas de los datos.
    - En el caso de que la respuesta `y` sea categórica realiza el muestreo en cada clase. Para respuestas numéricas emplea cuantiles (definidos por el argumento `groups = min(5, length(y))`).
    - `p`: proporción de datos en la muestra de entrenamiento.
    - `list`: lógico; determina si el resultado es una lista con las muestras o un vector (o matriz) de índices
  - Funciones auxiliares: `createFolds()`, `createMultiFolds()`, `groupKFold()`, `createResample()`, `createTimeSlices()`
- Análisis descriptivo: `featurePlot()`
- Preprocesado de los datos:
  - La función principal es `preProcess(x, method = c("center", "scale"), ...)`, aunque se puede integrar en el entrenamiento (función `train()`) para estimar los parámetros de las transformaciones a partir de la muestra de entrenamiento y posteriormente aplicarlas automáticamente al hacer nuevas predicciones (p.e. en la muestra de test).
  - El parámetro `method` permite establecer una lista de procesados:
    - Imputación: `"knnImpute"`, `"bagImpute"` o `"medianImpute"`

    * Creación y transformación de variables explicativas: `"center"`, `"scale"`, `"range"`, `"BoxCox"`, `"YeoJohnson"`, `"expoTrans"`, `"spatialSign"`

      Funciones auxiliares: `dummyVars()`…

    * Selección de predictores y extracción de componentes: `"corr"`, `"nzv"`, `"zv"`, `"conditionalX"`, `"pca"`, `"ica"`

      Funciones auxiliares: `rfe()`…

- Entrenamiento y selección de los hiperparámetros del modelo:
  - La función principal es `train(formula, data, method = "rf", trControl = trainControl(), tuneGrid = NULL, tuneLength = 3, ...)`
    * `trControl`: permite establecer el método de remuestreo para la evaluación de los hiperparámetros y el método para seleccionar el óptimo, incluyendo las medidas de precisión. Por ejemplo `trControl = trainControl(method = "cv", number = 10, selectionFunction = "oneSE")`.

      Los métodos disponibles son: `"boot"`, `"boot632"`, `"optimism_boot"`, `"boot_all"`, `"cv"`, `"repeatedcv"`, `"LOOCV"`, `"LGOCV"`, `"timeslice"`, `"adaptive_cv"`, `"adaptive_boot"` o `"adaptive_LGOCV"`
    * `tuneLength` y `tuneGrid`: permite establecer cuantos hiperparámetros serán evaluados (por defecto 3) o una rejilla con las combinaciones de hiperparámetros.
    * `...` permite establecer opciones específicas de los métodos.
  - También admite matrices `x`, `y` en lugar de fórmulas (o *recetas*: `recipe()`).
  - Si se imputan datos en el preprocesado será necesario establecer `na.action = na.pass`.
- Predicción: Una de las ventajas es que incorpora un único método `predict()` para objetos de tipo `train` con dos únicas opciones[12] `type = c("raw", "prob")`, la primera para obtener predicciones de la respuesta y la segunda para obtener estimaciones de las probabilidades (en los métodos de clasificación que lo admitan).

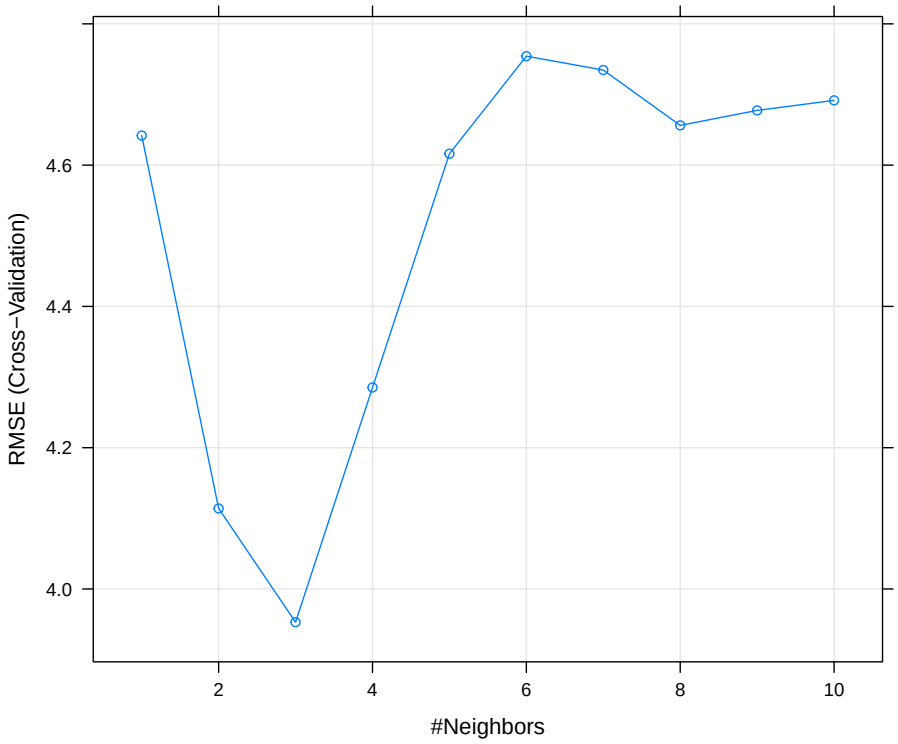
  Además, si se incluyo un preprocesado en el entrenamiento, se emplearán las mismas transformaciones en un nuevo conjunto de datos `newdata`.
- Evaluación de los modelos
  - `postResample(pred, obs, ...)`: regresión
  - `confusionMatrix(pred, obs, ...)`: clasificación
    * Funciones auxiliares: `twoClassSummary()`, `prSummary()`…
- Analisis de la importancia de los predictores:
  - `varImp()`: interfaz a las medidas específicas de los métodos de aprendizaje supervisado (Sección 15.1) o medidas genéricas (Sección 15.2).

Ejemplo regresión con KNN:

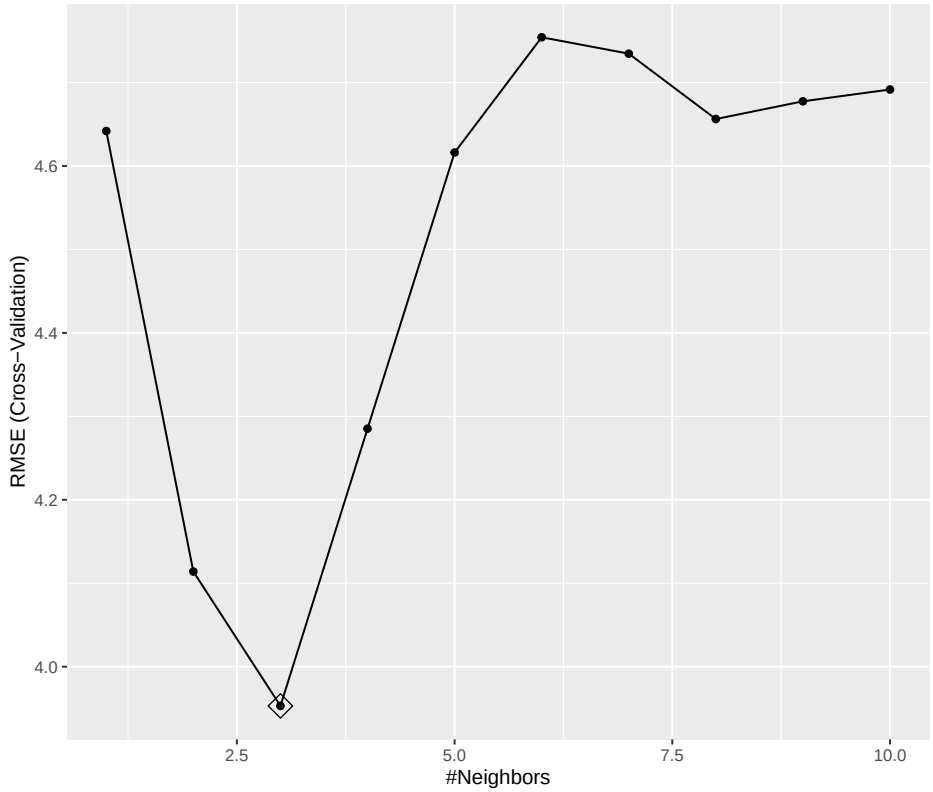
```
# caret
data(Boston, package = "MASS")

library(caret)
# Partición
set.seed(1)
itrain <- createDataPartition(Boston$medv, p = 0.8, list = FALSE)
train <- Boston[itrain, ]
test <- Boston[-itrain, ]
```

[12] En lugar de la variedad de opciones que emplean los distintos paquetes (e.g.: `type = "response"`, `"class"`, `"posterior"`, `"probability"`…).

```
# Entrenamiento y selección de hiperparámetros
set.seed(1)
knn <- train(medv ~ ., data = train,
             method = "knn",
             preProc = c("center", "scale"),
             tuneGrid = data.frame(k = 1:10),
             trControl = trainControl(method = "cv", number = 10))
plot(knn)
```




```
ggplot(knn, highlight = TRUE)
```




```
knn$bestTune
```

```
##   k
## 3 3
```

```
knn$finalModel
```

```
## 3-nearest neighbor regression model
```

```
# Interpretación
varImp(knn)
```

```
## loess r-squared variable importance
##
##         Overall
## lstat    100.00
## rm        88.26
## indus     36.29
## ptratio   33.27
## tax       30.58
## crim      28.33
## nox       23.44
## black     21.29
## age       20.47
## rad       17.16
## zn        15.11
## dis       14.35
## chas       0.00
```

```
# Evaluación
postResample(predict(knn, newdata = test), test$medv)
```

```
##     RMSE Rsquared      MAE
```

```
## 4.960971 0.733945 2.724242
```

Un comentario final:

> "While I'm still supporting caret, the majority of my development effort has gone into the tidyverse modeling packages (called tidymodels)".
>
> — Max Kuhn, autor del paquete `caret` (actualmente ingeniero de software en RStudio).

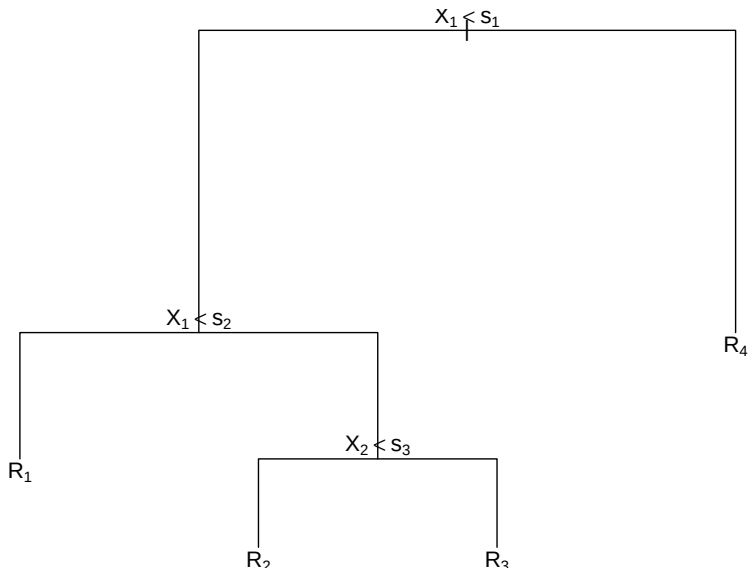
Kuhn, M. y Wickham, H. (2020). *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles.* Version 0.1.1 (2020-07-14). https://www.tidymodels.org.

Capítulo 2

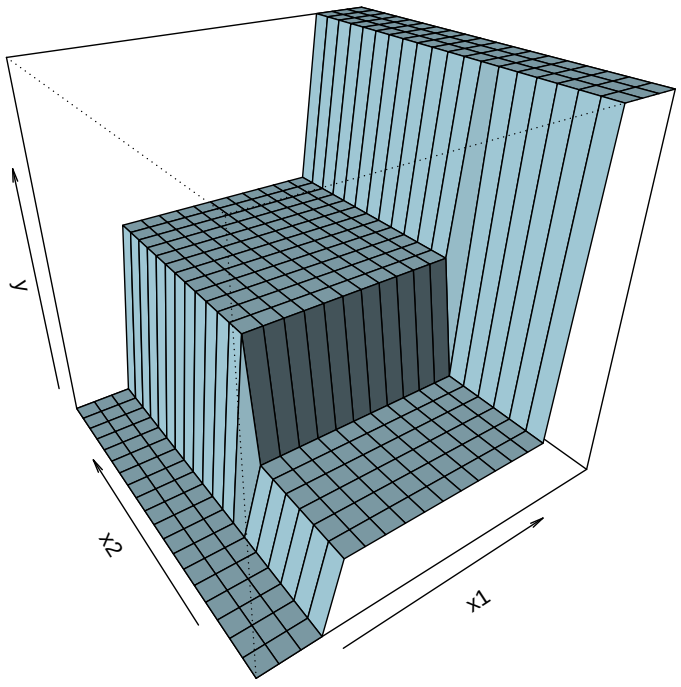
Árboles de decisión

Los *árboles de decisión* son uno de los métodos más simples y fáciles de interpretar para realizar predicciones en problemas de clasificación y de regresión. Se desarrollan a partir de los años 70 del siglo pasado como una alternativa versátil a los métodos clásicos de la estadística, fuertemente basados en las hipótesis de linealidad y de normalidad, y enseguida se convierten en una técnica básica del aprendizaje automático. Aunque su calidad predictiva es mediocre (especialmente en el caso de regresión), constituyen la base de otros métodos altamente competitivos (bagging, bosques aleatorios, boosting) en los que se combinan múltiples árboles para mejorar la predicción, pagando el precio, eso sí, de hacer más difícil la interpretación del modelo resultante.

La idea de este método consiste en la segmentación (partición) del *espacio predictor* (es decir, del conjunto de posibles valores de las variables predictoras) en regiones tan simples que el proceso se pueda representar mediante un árbol binario. Se parte de un nodo inicial que representa a toda la muestra (se utiliza la muestra de entrenamiento), del que salen dos ramas que dividen la muestra en dos subconjuntos, cada uno representado por un nuevo nodo. Este proceso se repite un número finito de veces hasta obtener las hojas del árbol, es decir, los nodos terminales, que son los que se utilizan para realizar la predicción. Una vez construido el árbol, la predicción se realizará en cada nodo terminal utilizando, típicamente, la media en un problema de regresión y la moda en un problema de clasificación.



Al final de este proceso iterativo el espacio predictor se ha particionado en regiones de forma rectangular en la que la predicción de la respuesta es constante. Si la relación entre las variables predictoras y la variable respuesta no se puede describir adecuadamente mediante rectángulos, la calidad predictiva del árbol será limitada. Como vemos, la simplicidad del modelo es su principal argumento, pero también su talón de Aquiles.



Como se ha dicho antes, cada nodo padre se divide, a través de dos ramas, en dos nodos hijos. Esto se hace seleccionando una variable predictora y dando respuesta a una pregunta dicotómica sobre ella. Por ejemplo, ¿es el sueldo anual menor que 30000 euros?, o ¿es el género igual a *mujer*? Lo que se persigue con esta partición recursiva es que los nodos terminales sean homogéneos respecto a la variable respuesta Y.

Por ejemplo, en un problema de clasificación, la homogeneidad de los nodos terminales significaría que en cada uno de ellos sólo hay elementos de una clase (categoría), y diríamos que los nodos son *puros*. En la práctica, esto siempre se puede conseguir construyendo árboles suficientemente profundos, con muchas hojas. Pero esta solución no es interesante, ya que va a dar lugar a un modelo excesivamente complejo y por tanto sobreajustado y de difícil interpretación. Será necesario encontrar un equilibrio entre la complejidad del árbol y la pureza de los nodos terminales.

En resumen:

- Métodos simples y fácilmente interpretables.
- Se representan mediante árboles binarios.
- Técnica clásica de apendizaje automático (computación).
- Válidos para regresión y para clasificación.
- Válidos para predictores numéricos y categóricos.

La metodología CART (Classification and Regresion Trees, Breiman *et al.*, 1984) es la más popular para la construcción de árboles de decisión y es la que se va a explicar con algo de detalle en las siguientes secciones.

En primer lugar se tratarán los *árboles de regresión* (árboles de decisión en un problema de regresión, en el que la variable respuesta Y es numérica) y después veremos los *arboles de clasificación* (respuesta categórica) que son los más utilizados en la práctica (los primeros se suelen emplear únicamente como métodos descriptivos o como base de métodos más complejos). Las variables predictoras $\mathbf{X} = (X_1, X_2, \ldots, X_p)$ pueden ser tanto numéricas como categóricas. Además, con la metodología CART, las variables explicativas podrían contener datos faltantes. Se pueden establecer "particiones sustitutas" (*surrogate splits*), de forma que cuando falta un valor en una variable que determina una división, se usa una variable alternativa que produce una partición similar.

2.1 Árboles de regresión CART

Como ya se comentó, la construcción del modelo se hace a partir de la muestra de entrenamiento, y consiste en la partición del espacio predictor en J regiones $R_1, R_2, \ldots, R_J$, para cada una de las cuales se va a calcular una constante: la media de la variable respuesta Y para las observaciones de entranamiento que caen en la región. Estas constantes son las que se van a utilizar para la predicción de nuevas observaciones; para ello solo hay que comprobar cuál es la región que le corresponde.

La cuestión clave es cómo se elige la partición del espacio predictor, para lo que vamos a utilizar como criterio de error el RSS (suma de los residuos al cuadrado). Como hemos dicho, vamos a modelizar la respuesta en cada región como una constante, por tanto en la región R_j nos interesa el $min_{c_j} \sum_{i \in R_j} (y_i - c_j)^2$, que se alcanza en la media de las respuestas y_i (de la muestra de entrenamiento) en la región R_j, a la que llamaremos $\widehat{y}_{R_j}$. Por tanto, se deben seleccionar las regiones $R_1, R_2, \ldots, R_J$ que minimicen

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

(Obsérvese el abuso de notación $i \in R_j$, que significa las observaciones $i \in N$ que verifican $x_i \in R_j$).

Pero este problema es, en la práctica, intratable y vamos a tener que simplificarlo. El método CART busca un compromiso entre rendimiento, por una parte, y sencillez e interpretabilidad, por otra, y

por ello en lugar de hacer una búsqueda por todas las particiones posibles sigue un proceso iterativo (recursivo) en el que va realizando cortes binarios. En la primera iteración se trabaja con todos los datos:

- Una variable explicativa X_j y un punto de corte s definen dos hiperplanos $R_1 = \{X \mid X_j \leq s\}$ y $R_2 = \{X \mid X_j > s\}$.
- Se seleccionan los valores de j y s que minimizen

$$\sum_{i \in R_1} (y_i - \widehat{y}_{R_1})^2 + \sum_{i \in R_2} (y_i - \widehat{y}_{R_2})^2$$

A diferencia del problema original, este se soluciona de forma muy rápida. A continuación se repite el proceso en cada una de las dos regiones R_1 y R_2, y así sucesivamente hasta alcanzar un criterio de parada.

Fijémonos en que este método hace dos concesiones importantes: no solo restringe la forma que pueden adoptar las particiones, sino que además sigue un criterio de error *greedy*: en cada iteración busca minimizar el RSS de las dos regiones resultantes, sin preocuparse del error que se va a cometer en iteraciones sucesivas. Y fijémonos también en que este proceso se puede representar en forma de árbol binario (en el sentido de que de cada nodo salen dos ramas, o ninguna cuando se llega al final), de ahí la terminología de *hacer crecer* el árbol.

¿Y cuándo paramos? Se puede parar cuando se alcance una profundidad máxima, aunque lo más habitual es, para dividir un nodo (es decir, una región), exigirle un número mínimo de observaciones.

- Si el árbol resultante es demasiado grande, va a ser un modelo demasiado complejo, por tanto va a ser difícil de interpretar y, sobre todo, va a provocar un sobreajuste de los datos. Cuando se evalúe el rendimiento utilizando la muestra de validación, los resultados van a ser malos. Dicho de otra manera, tendremos un modelo con poco sesgo pero con mucha varianza y en consecuencia inestable (pequeños cambios en los datos darán lugar a modelos muy distintos). Más adelante veremos que esto justifica la utilización del *bagging* como técnica para reducir la varianza.
- Si el árbol es demasiado pequeño, va a tener menos varianza (menos inestable) a costa de más sesgo. Más adelante veremos que esto justifica la utilización del *boosting*. Los árboles pequeños son más fáciles de interpretar ya que permiten identificar las variables explicativas que más influyen en la predicción.

Sin entrar por ahora en métodos combinados (métodos *ensemble*, tipo *bagging* o *boosting*), vamos a explicar cómo encontrar un equilibrio entre sesgo y varianza. Lo que se hace es construir un árbol grande para a continuación empezar a *podarlo*. Podar un árbol significa colapsar cualquier cantidad de sus nodos internos (no terminales), dando lugar a otro árbol más pequeño al que llamaremos *subárbol* del árbol original. Sabemos que el árbol completo es el que va a tener menor error si utilizamos la muestra de entrenamiento, pero lo que realmente nos interesa es encontrar el subárbol con un menor error al utilizar la muestra de validación. Lamentablemente, no es una buena estrategia el evaluar todos los subárboles: simplemente, hay demasiados. Lo que se hace es, mediante un hiperparámetro (*tuning parameter* o parámetro de ajuste) controlar el tamaño del árbol, es decir, la complejidad del modelo, seleccionando el subárbol *optimo* (para los datos de los que disponemos, claro). Veamos la idea.

Dado un subárbol T con $R_1, R_2, \ldots, R_t$ nodos terminales, consideramos como medida del error el RSS más una penalización que depende de un hiperparámetro no negativo $\alpha \geq 0$

$$RSS_\alpha = \sum_{j=1}^{t} \sum_{i \in R_j} (y_i - \widehat{y}_{R_j})^2 + \alpha t \tag{2.1}$$

Para cada valor del parámetro α existe un único subárbol *más pequeño* que minimiza este error (obsérvese que aunque hay un continuo de valores distinos de α, sólo hay una cantidad finita de subárboles). Evidentemente, cuando $\alpha = 0$, ese subárbol será el árbol completo, algo que no nos

interesa. Pero a medida que se incrementa α se penalizan los subárboles con muchos nodos terminales, dando lugar a una solución más pequeña. Encontrarla puede parecer muy costoso computacionalmente, pero lo cierto es que no lo es. El algoritmo consistente en ir colapsando nodos de forma sucesiva, de cada vez el nodo que produzca el menor incremento en el RSS (corregido por un factor que depende del tamaño), da lugar a una sucesión finita de subárboles que contiene, para todo α, la solución.

Para finalizar, sólo resta seleccionar un valor de α. Para ello, como se comentó en la Sección 1.3.2, se podría dividir la muestra en tres subconjuntos: datos de entrenamiento, de validación y de test. Para cada valor del parámetro de complejidad α hemos utilizado la muestra de entrenamiento para obtener un árbol (en la jerga, para cada valor del hiperparámetro α se entrena un modelo). Se emplea la muestra independiente de validación para seleccionar el valor de α (y por tanto el árbol) con el que nos quedamos. Y por último emplearemos la muestra de test (independiente de las otras dos) para evaluar el rendimiento del árbol seleccionado. No obstante, lo más habitual para seleccionar el valor del hiperparámetro α es emplear validación cruzada (o otro tipo de remuestreo) en la muestra de entrenamiento en lugar de considerar una muestra adicional de validación.

Hay dos opciones muy utilizadas en la práctica para seleccionar el valor de α: se puede utilizar directamente el valor que minimice el error; o se puede forzar que el modelo sea un poco más sencillo con la regla *one-standard-error*, que selecciona el árbol más pequeño que esté a una distancia de un error estándar del árbol obtenido mediante la opción anterior.

También es habitual escribir la Ecuación (2.1) reescalando el parámetro de complejidad como $\tilde{\alpha} = \alpha/RSS_0$, siendo $RSS_0 = \sum_{i=1}^{n}(y_i - \bar{y})^2$ la variabilidad total (la suma de cuadrados residual del árbol sin divisiones):

$$RSS_{\tilde{\alpha}} = RSS + \tilde{\alpha} RSS_0 t$$

De esta forma se podría interpretar el hiperparámetro $\tilde{\alpha}$ como una penalización en la proporción de variabilidad explicada, ya que dividiendo la expresión anterior por RSS_0 obtendríamos:

$$R^2_{\tilde{\alpha}} = R^2 + \tilde{\alpha} t$$

2.2 Árboles de clasificación CART

En un problema de clasificación la variable respuesta puede tomar los valores $1, 2, \ldots, K$, etiquetas que identifican las K categorías del problema. Una vez construido el árbol, se comprueba cuál es la categoría modal de cada región: considerando la muestra de entrenamiento, la categoría más frecuente. Dada una observación, se predice que pertenece a la categoría modal de la región a la que pertenece.

El resto del proceso es idéntico al de los árboles de regresión ya explicado, con una única salvedad: no podemos utilizar RSS como medida del error. Es necesario buscar una medida del error adaptada a este contexto. Fijada una región, vamos a denotar por $\widehat{p}_k$, con $k = 1, 2, \ldots, K$, a la proporción de observaciones (de la muestra de entrenamiento) en la región que pertenecen a la categoría k. Se utilizan tres medidas distintas del error en la región:

- Proporción de errores de clasificación:

$$1 - max_k(\widehat{p}_k)$$

- Índice de Gini:

$$\sum_{k=1}^{K} \widehat{p}_k(1 - \widehat{p}_k)$$

- Entropía[1] (*cross-entropy*):

$$-\sum_{k=1}^{K} \widehat{p}_k \log(\widehat{p}_k)$$

[1] La entropía es un concepto básico de la teoría de la información (Shannon, 1948) y se mide en *bits* (cuando en la definición se utilizan log_2).

Aunque la proporción de errores de clasificación es la medida del error más intuitiva, en la práctica sólo se utiliza para la fase de poda. Fijémonos que en el cálculo de esta medida sólo interviene $max_k(\widehat{p}_k)$, mientras que en las medidas alternativas intervienen las proporciones $\widehat{p}_k$ de todas las categorías. Para la fase de crecimiento se utilizan indistintamente el índice de Gini o la entropía. Cuando nos interesa el error no en una única región sino en varias (al romper un nodo en dos, o al considerar todos los nodos terminales), se suman los errores de cada región previa ponderación por el número de observaciones que hay en cada una de ellas.

En la introducción de este tema se comentó que los árboles de decisión admiten tanto variables predictoras numéricas como categóricas, y esto es cierto tanto para árboles de regresión como para árboles de clasificación. Veamos brevemente como se tratarían los predictores categóricos a la hora de incorporarlos al árbol. El problema radica en qué se entiende por hacer un corte si las categorías del predictor no están ordenadas. Hay dos soluciones básicas:

- Definir variables predictoras *dummy*. Se trata de variables indicadoras, una por cada una de las categorías que tiene el predictor. Este criterio de *uno contra todos* tiene la ventaja de que estas variables son fácilmente interpretables, pero tiene el inconveniente de que puede aumentar mucho el número de variables predictoras.
- Ordenar las categorías de la variable predictora. Lo ideal sería considerar todas las ordenaciones posibles, pero eso es desde luego poco práctico: el incremento es factorial. El truco consiste en utilizar un único órden basado en algún criterio *greedy*. Por ejemplo, si la variable respuesta Y también es categórica, se puede seleccionar una de sus categorías que resulte especialmente interesante y ordenar las categorías del predictor según su proporción en la categoría de Y. Este enfoque no añade complejidad al modelo, pero puede dar lugar a resultados de difícil interpretación.

2.3 CART con el paquete `rpart`

La metodología CART está implementada en el paquete `rpart` (Recursive PARTitioning)[2]. La función principal es `rpart()` y habitualmente se emplea de la forma:

```
rpart(formula, data, method, parms, control, ...)
```

- `formula`: permite especificar la respuesta y las variables predictoras de la forma habitual, se suele establecer de la forma `respuesta ~ .` para incluir todas las posibles variables explicativas.
- `data`: `data.frame` (opcional; donde se evaluará la fórmula) con la muestra de entrenamiento.
- `method`: método empleado para realizar las particiones, puede ser `"anova"` (regresión), `"class"` (clasificación), `"poisson"` (regresión de Poisson) o `"exp"` (supervivencia), o alternativamente una lista de funciones (con componentes `init`, `split`, `eval`; ver la vignette *User Written Split Functions*). Por defecto se selecciona a partir de la variable respuesta en `formula`, por ejemplo si es un factor (lo recomendado en clasificación) emplea `method = "class"`.
- `parms`: lista de parámetros opcionales para la partición en el caso de clasificación (o regresión de Poisson). Puede contener los componentes `prior` (vector de probabilidades previas; por defecto las frecuencias observadas), `loss` (matriz de pérdidas; con ceros en la diagonal y por defecto 1 en el resto) y `split` (criterio de error; por defecto `"gini"` o alternativamente `"information"`).
- `control`: lista de opciones que controlan el algoritmo de partición, por defecto se seleccionan mediante la función `rpart.control`, aunque también se pueden establecer en la llamada a la función principal, y los principales parámetros son:

  ```
  rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01, xval =
  10, maxdepth = 30, ...)
  ```

  - `cp` es el parámetro de complejidad $\tilde{\alpha}$ para la poda del árbol, de forma que un valor de 1 se corresponde con un árbol sin divisiones y un valor de 0 con un árbol de profundidad

[2]El paquete `tree` es una traducción del original en S.

máxima. Adicionalmente, para reducir el tiempo de computación, el algoritmo empleado no realiza una partición si la proporción de reducción del error es inferior a este valor (valores más grandes simplifican el modelo y reducen el tiempo de computación).

- `maxdepth` es la profundidad máxima del árbol (la profundidad de la raíz sería 0).

- `minsplit` y `minbucket` son, respectivamente, los números mínimos de observaciones en un nodo intermedio para particionarlo y en un nodo terminal.

- `xval` es el número de grupos (folds) para validación cruzada.

Para más detalles consultar la documentación de esta función o la vignette *Introduction to Rpart*.

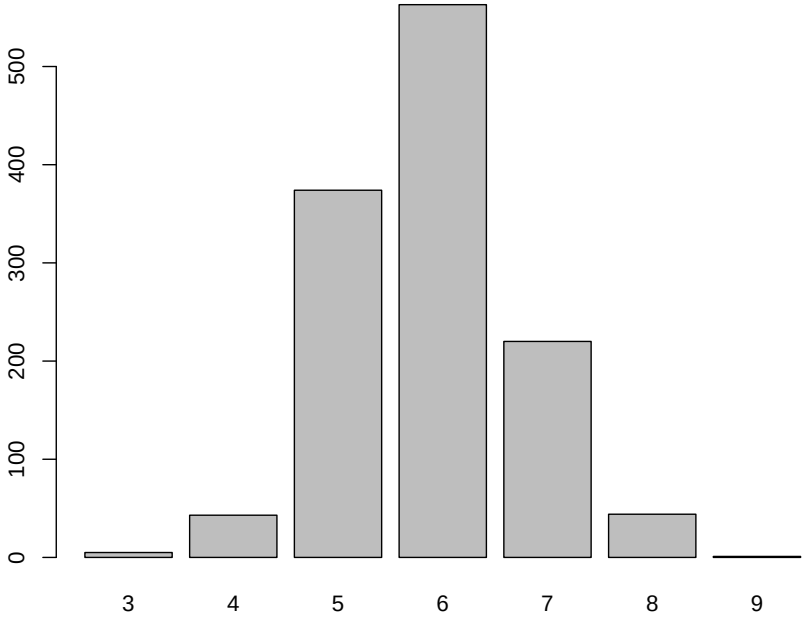
2.3.1 Ejemplo: regresión

Emplearemos el conjunto de datos *winequality.RData* (ver Cortez et al., 2009), que contiene información fisico-química (`fixed.acidity`, `volatile.acidity`, `citric.acid`, `residual.sugar`, `chlorides`, `free.sulfur.dioxide`, `total.sulfur.dioxide`, `density`, `pH`, `sulphates` y `alcohol`) y sensorial (`quality`) de una muestra de 1250 vinos portugueses de la variedad *Vinho Verde*. Como respuesta consideraremos la variable `quality`, mediana de al menos 3 evaluaciones de la calidad del vino realizadas por expertos, que los evaluaron entre 0 (muy malo) y 10 (muy excelente).

```
load("data/winequality.RData")
str(winequality)
```

```
## 'data.frame':    1250 obs. of  12 variables:
##  $ fixed.acidity       : num  6.8 7.1 6.9 7.5 8.6 7.7 5.4 6.8 6.1 5.5 ...
##  $ volatile.acidity    : num  0.37 0.24 0.32 0.23 0.36 0.28 0.59 0.16 0.28 0.28 ...
##  $ citric.acid         : num  0.47 0.34 0.13 0.49 0.26 0.63 0.07 0.36 0.27 0.21 ...
##  $ residual.sugar      : num  11.2 1.2 7.8 7.7 11.1 11.1 7 1.3 4.7 1.6 ...
##  $ chlorides           : num  0.071 0.045 0.042 0.049 0.03 0.039 0.045 0.034 0.03 0.032 ...
##  $ free.sulfur.dioxide : num  44 6 11 61 43.5 58 36 32 56 23 ...
##  $ total.sulfur.dioxide: num  136 132 117 209 171 179 147 98 140 85 ...
##  $ density             : num  0.997 0.991 0.996 0.994 0.995 ...
##  $ pH                  : num  2.98 3.16 3.23 3.14 3.03 3.08 3.34 3.02 3.16 3.42 ...
##  $ sulphates           : num  0.88 0.46 0.37 0.3 0.49 0.44 0.57 0.58 0.42 0.42 ...
##  $ alcohol             : num  9.2 11.2 9.2 11.1 12 8.8 9.7 11.3 12.5 12.5 ...
##  $ quality             : int  5 4 5 7 5 4 6 6 8 5 ...
```

```
barplot(table(winequality$quality))
```



En primer lugar se selecciona el 80% de los datos como muestra de entrenamiento y el 20% restante como muestra de test:

```
set.seed(1)
nobs <- nrow(winequality)
itrain <- sample(nobs, 0.8 * nobs)
train <- winequality[itrain, ]
test <- winequality[-itrain, ]
```

Podemos obtener el arbol con las opciones por defecto con el comando:

```
tree <- rpart(quality ~ ., data = train)
```

Al imprimirlo se muestra el número de observaciones e información sobre los distintos nodos (número de nodo, condición que define la partición, número de observaciones en el nodo, función de pérdida y predicción), marcando con un * los nodos terminales.

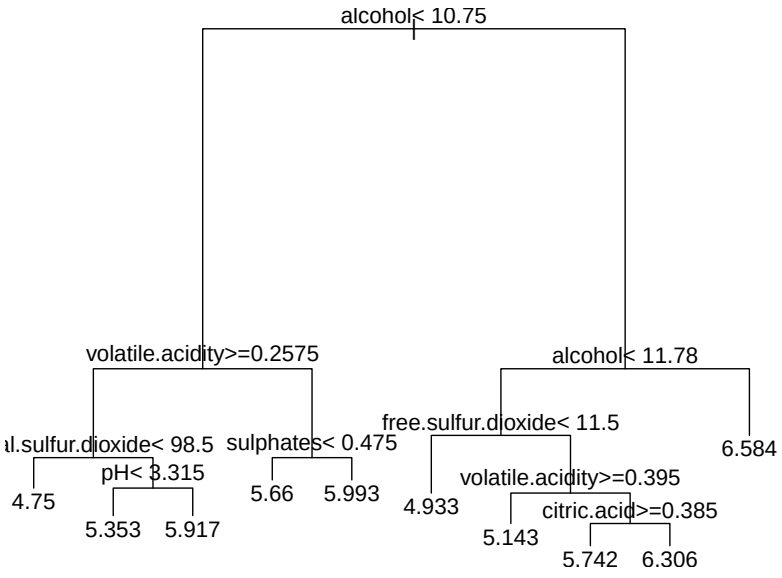
```
tree
```

```
## n= 1000
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 1000 768.95600 5.862000
##    2) alcohol< 10.75 622 340.81190 5.586817
##      4) volatile.acidity>=0.2575 329 154.75990 5.370821
##        8) total.sulfur.dioxide< 98.5 24  12.50000 4.750000 *
##        9) total.sulfur.dioxide>=98.5 305 132.28200 5.419672
##         18) pH< 3.315 269 101.44980 5.353160 *
##         19) pH>=3.315 36  20.75000 5.916667 *
##      5) volatile.acidity< 0.2575 293 153.46760 5.829352
##       10) sulphates< 0.475 144  80.32639 5.659722 *
```

```
##       11) sulphates>=0.475 149  64.99329 5.993289 *
##    3) alcohol>=10.75 378 303.53700 6.314815
##      6) alcohol< 11.775 200 173.87500 6.075000
##       12) free.sulfur.dioxide< 11.5 15  10.93333 4.933333 *
##       13) free.sulfur.dioxide>=11.5 185 141.80540 6.167568
##         26) volatile.acidity>=0.395 7  12.85714 5.142857 *
##         27) volatile.acidity< 0.395 178 121.30900 6.207865
##           54) citric.acid>=0.385 31  21.93548 5.741935 *
##           55) citric.acid< 0.385 147  91.22449 6.306122 *
##      7) alcohol>=11.775 178 105.23600 6.584270 *
```

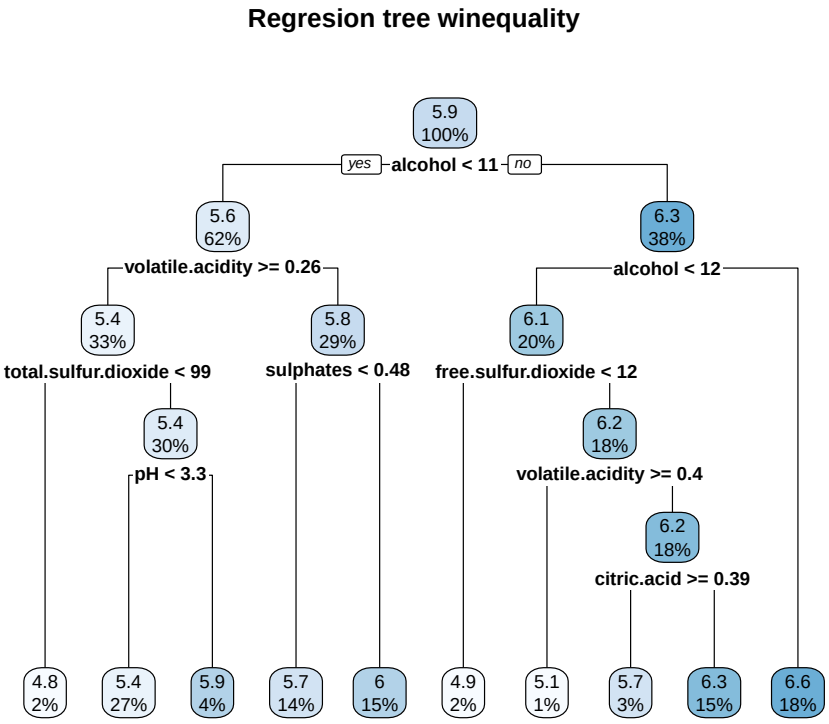
Para representarlo se puede emplear las herramientas del paquete `rpart`:

```
plot(tree)
text(tree)
```



Pero puede ser preferible emplear el paquete `rpart.plot`

```
library(rpart.plot)
rpart.plot(tree, main="Regresion tree winequality")
```



Nos interesa como se clasificaría a una nueva observación en los nodos terminales (en los nodos intermedios solo nos interesarían las condiciones, y el orden de las variables consideradas, hasta llegar a las hojas) y las correspondientes predicciones (la media de la respuesta en el correspondiente nodo terminal). Para ello, puede ser de utilidad imprimir las reglas:

```
rpart.rules(tree, style = "tall")
```

```
## quality is 4.8 when
##     alcohol < 11
##     volatile.acidity >= 0.26
##     total.sulfur.dioxide < 99
##
## quality is 4.9 when
##     alcohol is 11 to 12
##     free.sulfur.dioxide < 12
##
## quality is 5.1 when
##     alcohol is 11 to 12
##     volatile.acidity >= 0.40
##     free.sulfur.dioxide >= 12
##
## quality is 5.4 when
##     alcohol < 11
##     volatile.acidity >= 0.26
##     total.sulfur.dioxide >= 99
##     pH < 3.3
##
## quality is 5.7 when
##     alcohol < 11
##     volatile.acidity < 0.26
##     sulphates < 0.48
```

```
##
## quality is 5.7 when
##     alcohol is 11 to 12
##     volatile.acidity < 0.40
##     free.sulfur.dioxide >= 12
##     citric.acid >= 0.39
##
## quality is 5.9 when
##     alcohol < 11
##     volatile.acidity >= 0.26
##     total.sulfur.dioxide >= 99
##     pH >= 3.3
##
## quality is 6.0 when
##     alcohol < 11
##     volatile.acidity < 0.26
##     sulphates >= 0.48
##
## quality is 6.3 when
##     alcohol is 11 to 12
##     volatile.acidity < 0.40
##     free.sulfur.dioxide >= 12
##     citric.acid < 0.39
##
## quality is 6.6 when
##     alcohol >= 12
```

Por defecto se poda el arbol considerando `cp = 0.01`, que puede ser adecuado en muchos casos. Sin embargo, para seleccionar el valor óptimo de este (hiper)parámetro se puede emplear validación cruzada. En primer lugar habría que establecer `cp = 0` para construir el árbol completo, a la profundidad máxima (determinada por los valores de `minsplit` y `minbucket`, que se podrían seleccionar "a mano" dependiendo del número de observaciones o también considerándolos como hiperparámetos; esto último no está implementado en `rpart`, ni en principio en `caret`)[3].

```
tree <- rpart(quality ~ ., data = train, cp = 0)
```

Posteriormente podemos emplear las funciones `printcp()` (o `plotcp()`) para obtener (representar) los valores de CP para los árboles (óptimos) de menor tamaño junto con su error de validación cruzada `xerror` (reescalado de forma que el máximo de `rel error` es 1):

```
printcp(tree)
```

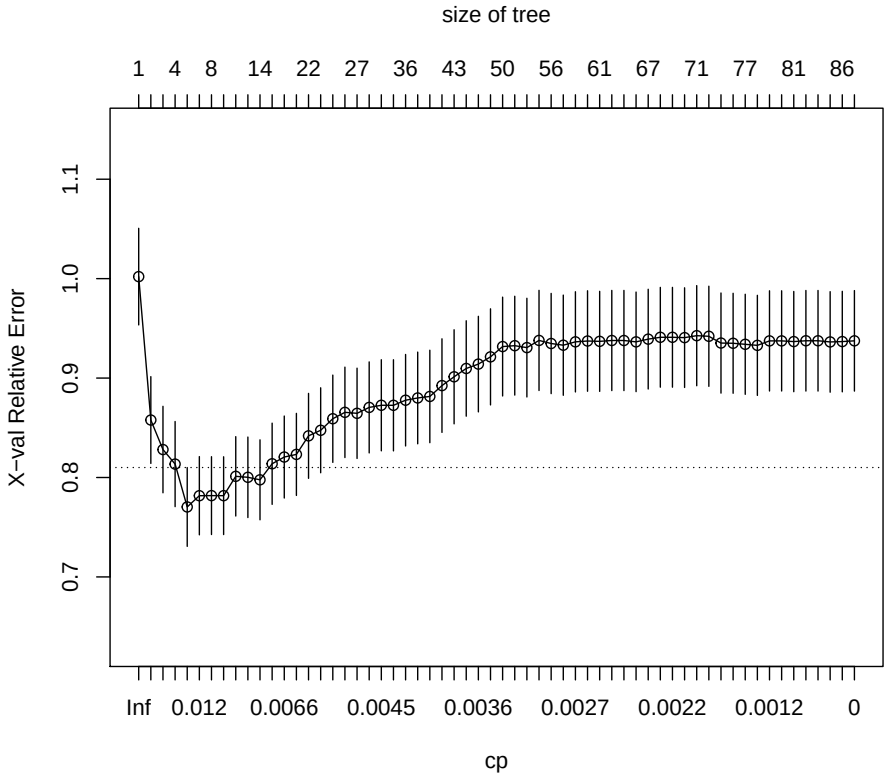
```
##
## Regression tree:
## rpart(formula = quality ~ ., data = train, cp = 0)
##
## Variables actually used in tree construction:
##  [1] alcohol              chlorides            citric.acid
##  [4] density              fixed.acidity        free.sulfur.dioxide
##  [7] pH                   residual.sugar       sulphates
## [10] total.sulfur.dioxide volatile.acidity
##
## Root node error: 768.96/1000 = 0.76896
##
## n= 1000
##
##            CP nsplit rel error  xerror     xstd
```

[3] Los parámetros `maxsurrogate`, `usesurrogate` y `surrogatestyle` serían de utilidad si hay datos faltantes.

```
##  1  0.16204707      0   1.00000 1.00203 0.048591
##  2  0.04237491      1   0.83795 0.85779 0.043646
##  3  0.03176525      2   0.79558 0.82810 0.043486
##  4  0.02748696      3   0.76381 0.81350 0.042814
##  5  0.01304370      4   0.73633 0.77038 0.039654
##  6  0.01059605      6   0.71024 0.78168 0.039353
##  7  0.01026605      7   0.69964 0.78177 0.039141
##  8  0.00840800      9   0.67911 0.78172 0.039123
##  9  0.00813924     10   0.67070 0.80117 0.039915
##  10 0.00780567     11   0.66256 0.80020 0.040481
##  11 0.00684175     13   0.64695 0.79767 0.040219
##  12 0.00673843     15   0.63327 0.81381 0.040851
##  13 0.00643577     18   0.61305 0.82059 0.041240
##  14 0.00641137     19   0.60662 0.82323 0.041271
##  15 0.00549694     21   0.59379 0.84187 0.042714
##  16 0.00489406     23   0.58280 0.84748 0.042744
##  17 0.00483045     24   0.57791 0.85910 0.043897
##  18 0.00473741     25   0.57308 0.86553 0.045463
##  19 0.00468372     26   0.56834 0.86455 0.045413
##  20 0.00450496     28   0.55897 0.87049 0.045777
##  21 0.00448365     32   0.54095 0.87263 0.045824
##  22 0.00437484     33   0.53647 0.87260 0.045846
##  23 0.00435280     35   0.52772 0.87772 0.046022
##  24 0.00428623     36   0.52337 0.87999 0.046124
##  25 0.00412515     37   0.51908 0.88151 0.046505
##  26 0.00390866     39   0.51083 0.89242 0.047068
##  27 0.00375301     42   0.49910 0.90128 0.047319
##  28 0.00370055     43   0.49535 0.90965 0.047991
##  29 0.00351987     45   0.48795 0.91404 0.048079
##  30 0.00308860     47   0.48091 0.92132 0.048336
##  31 0.00305781     49   0.47473 0.93168 0.049699
##  32 0.00299018     51   0.46862 0.93258 0.049701
##  33 0.00295148     52   0.46563 0.93062 0.049644
##  34 0.00286138     54   0.45972 0.93786 0.050366
##  35 0.00283972     55   0.45686 0.93474 0.050404
##  36 0.00274809     56   0.45402 0.93307 0.050390
##  37 0.00273457     58   0.44853 0.93642 0.050406
##  38 0.00260607     59   0.44579 0.93726 0.050543
##  39 0.00252978     60   0.44318 0.93692 0.050323
##  40 0.00252428     62   0.43813 0.93778 0.050381
##  41 0.00250804     64   0.43308 0.93778 0.050381
##  42 0.00232226     65   0.43057 0.93642 0.050081
##  43 0.00227625     66   0.42825 0.93915 0.050166
##  44 0.00225146     67   0.42597 0.94101 0.050195
##  45 0.00224774     68   0.42372 0.94101 0.050195
##  46 0.00216406     69   0.42147 0.94067 0.050124
##  47 0.00204851     70   0.41931 0.94263 0.050366
##  48 0.00194517     72   0.41521 0.94203 0.050360
##  49 0.00188139     73   0.41326 0.93521 0.050349
##  50 0.00154129     75   0.40950 0.93500 0.050277
##  51 0.00143642     76   0.40796 0.93396 0.050329
##  52 0.00118294     77   0.40652 0.93289 0.050325
##  53 0.00117607     78   0.40534 0.93738 0.050406
##  54 0.00108561     79   0.40417 0.93738 0.050406
##  55 0.00097821     80   0.40308 0.93670 0.050406
##  56 0.00093107     81   0.40210 0.93752 0.050589
```

```
## 57 0.00090075      82    0.40117 0.93752 0.050589
## 58 0.00082968      83    0.40027 0.93634 0.050561
## 59 0.00048303      85    0.39861 0.93670 0.050557
## 60 0.00000000      86    0.39813 0.93745 0.050558
```

```
plotcp(tree)
```



La tabla con los valores de las podas (óptimas, dependiendo del parámetro de complejidad) está almacenada en la componente `$cptable`:

```
head(tree$cptable, 10)
```

```
##             CP nsplit rel error    xerror       xstd
## 1  0.162047069      0 1.0000000 1.0020304 0.04859127
## 2  0.042374911      1 0.8379529 0.8577876 0.04364585
## 3  0.031765253      2 0.7955780 0.8281010 0.04348571
## 4  0.027486958      3 0.7638128 0.8134957 0.04281430
## 5  0.013043701      4 0.7363258 0.7703804 0.03965433
## 6  0.010596054      6 0.7102384 0.7816774 0.03935308
## 7  0.010266055      7 0.6996424 0.7817716 0.03914071
## 8  0.008408003      9 0.6791102 0.7817177 0.03912344
## 9  0.008139238     10 0.6707022 0.8011719 0.03991498
## 10 0.007805674     11 0.6625630 0.8001996 0.04048088
```

A partir de la que podríamos seleccionar el valor óptimo de forma automática, siguiendo el criterio de un error estándar de Breiman et al. (1984):

```
xerror <- tree$cptable[,"xerror"]
imin.xerror <- which.min(xerror)
# Valor óptimo
tree$cptable[imin.xerror, ]
```

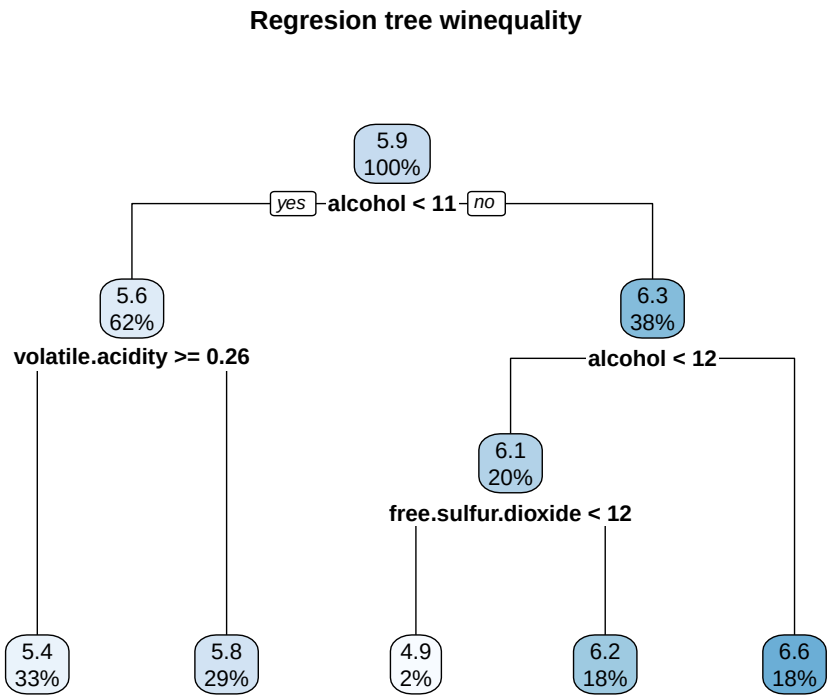
```
##         CP     nsplit  rel error     xerror       xstd
```

```
## 0.01304370 4.00000000 0.73632581 0.77038039 0.03965433
```

```
# Límite superior "oneSE rule" y complejidad mínima por debajo de ese valor
upper.xerror <- xerror[imin.xerror] + tree$cptable[imin.xerror, "xstd"]
icp <- min(which(xerror <= upper.xerror))
cp <- tree$cptable[icp, "CP"]
```

Para obtener el modelo final podamos el arbol con el valor de complejidad obtenido 0.0130437 (que en este caso coincide con el valor óptimo):

```
tree <- prune(tree, cp = cp)
rpart.plot(tree, main="Regresion tree winequality")
```



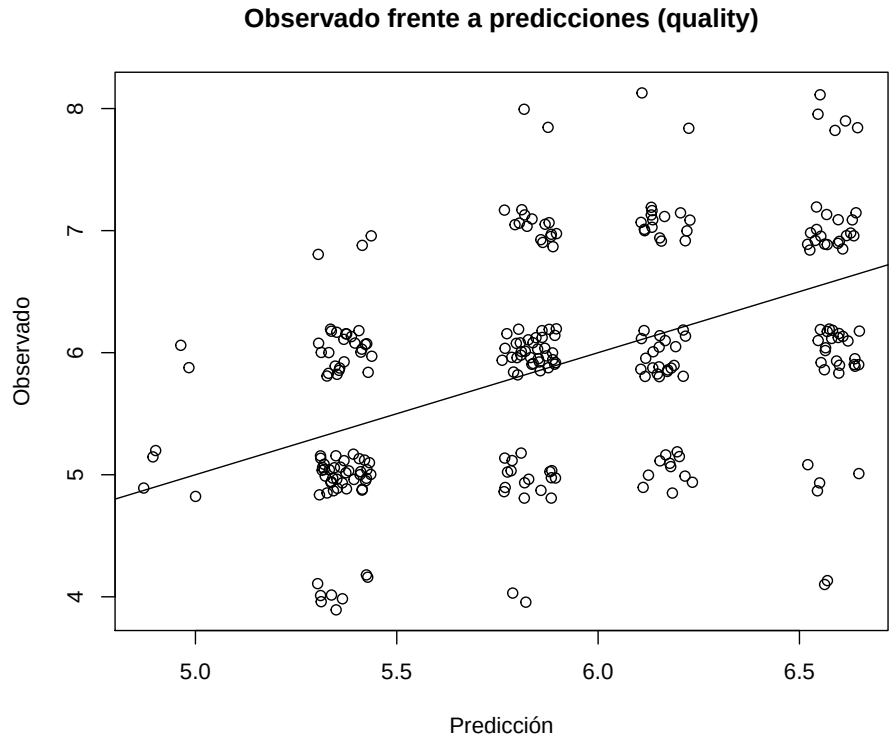
Podríamos estudiar el modelo final, por ejemplo mediante el método `summary()`, que entre otras cosas muestra una medida (en porcentaje) de la importancia de las variables explicativas para la predicción de la respuesta (teniendo en cuenta todas las particiones, principales y secundarias, en las que se emplea cada variable explicativa). Alternativamente podríamos emplear el siguiente código:

```
# summary(tree)
importance <- tree$variable.importance # Equivalente a caret::varImp(tree)
importance <- round(100*importance/sum(importance), 1)
importance[importance >= 1]
```

```
##              alcohol              density            chlorides
##                 36.1                 21.7                 11.3
##     volatile.acidity total.sulfur.dioxide  free.sulfur.dioxide
##                  8.7                  8.5                  5.0
##       residual.sugar            sulphates          citric.acid
##                  4.0                  1.9                  1.1
##                   pH
##                  1.1
```

El último paso sería evaluarlo en la muestra de test siguiendo los pasos descritos en la Sección 1.3.4:

```r
obs <- test$quality
pred <- predict(tree, newdata = test)

# plot(pred, obs, main = "Observado frente a predicciones (quality)",
#      xlab = "Predicción", ylab = "Observado")
plot(jitter(pred), jitter(obs), main = "Observado frente a predicciones (quality)",
     xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
```




```r
# Empleando el paquete caret
caret::postResample(pred, obs)
```

```
##      RMSE  Rsquared       MAE
## 0.8145614 0.1969485 0.6574264
```

```r
# Con la función accuracy()
accuracy <- function(pred, obs, na.rm = FALSE,
                     tol = sqrt(.Machine$double.eps)) {
  err <- obs - pred     # Errores
  if(na.rm) {
    is.a <- !is.na(err)
    err <- err[is.a]
    obs <- obs[is.a]
  }
  perr <- 100*err/pmax(obs, tol)  # Errores porcentuales
  return(c(
    me = mean(err),           # Error medio
    rmse = sqrt(mean(err^2)), # Raíz del error cuadrático medio
    mae = mean(abs(err)),     # Error absoluto medio
    mpe = mean(perr),         # Error porcentual medio
    mape = mean(abs(perr)),   # Error porcentual absoluto medio
```

```
    r.squared = 1 - sum(err^2)/sum((obs - mean(obs))^2)
  ))
}
accuracy(pred, test$quality)
```

```
##           me         rmse          mae          mpe         mape    r.squared
## -0.001269398  0.814561435  0.657426365 -1.952342173 11.576716037  0.192007721
```

2.3.2 Ejemplo: modelo de clasificación

Para ilustrar los árboles de clasificación CART, podemos emplear los datos anteriores de calidad de vino, considerando como respuesta una nueva variable `taste` que clasifica los vinos en "good" o "bad" dependiendo de si `winequality$quality >= 5` (este conjunto de datos está almacenado en el archivo *winetaste.RData*).

```
# load("data/winetaste.RData")
winetaste <- winequality[, colnames(winequality)!="quality"]
winetaste$taste <- factor(winequality$quality < 6, labels = c('good', 'bad')) # levels = c('FALSE',
str(winetaste)
```

```
## 'data.frame':    1250 obs. of  12 variables:
##  $ fixed.acidity       : num  6.8 7.1 6.9 7.5 8.6 7.7 5.4 6.8 6.1 5.5 ...
##  $ volatile.acidity    : num  0.37 0.24 0.32 0.23 0.36 0.28 0.59 0.16 0.28 0.28 ...
##  $ citric.acid         : num  0.47 0.34 0.13 0.49 0.26 0.63 0.07 0.36 0.27 0.21 ...
##  $ residual.sugar      : num  11.2 1.2 7.8 7.7 11.1 11.1 7 1.3 4.7 1.6 ...
##  $ chlorides           : num  0.071 0.045 0.042 0.049 0.03 0.039 0.045 0.034 0.03 0.032 ...
##  $ free.sulfur.dioxide : num  44 6 11 61 43.5 58 36 32 56 23 ...
##  $ total.sulfur.dioxide: num  136 132 117 209 171 179 147 98 140 85 ...
##  $ density             : num  0.997 0.991 0.996 0.994 0.995 ...
##  $ pH                  : num  2.98 3.16 3.23 3.14 3.03 3.08 3.34 3.02 3.16 3.42 ...
##  $ sulphates           : num  0.88 0.46 0.37 0.3 0.49 0.44 0.57 0.58 0.42 0.42 ...
##  $ alcohol             : num  9.2 11.2 9.2 11.1 12 8.8 9.7 11.3 12.5 12.5 ...
##  $ taste               : Factor w/ 2 levels "good","bad": 2 2 2 1 2 2 1 1 1 2 ...
```

```
table(winetaste$taste)
```

```
##
## good  bad
##  828  422
```

Como en el caso anterior, se contruyen las muestras de entrenamiento (80%) y de test (20%):

```
# set.seed(1)
# nobs <- nrow(winetaste)
# itrain <- sample(nobs, 0.8 * nobs)
train <- winetaste[itrain, ]
test <- winetaste[-itrain, ]
```

Al igual que en el caso anterior podemos obtener el árbol de clasificación con las opciones por defecto (`cp = 0.01` y `split = "gini"`) con el comando:

```
tree <- rpart(taste ~ ., data = train)
```

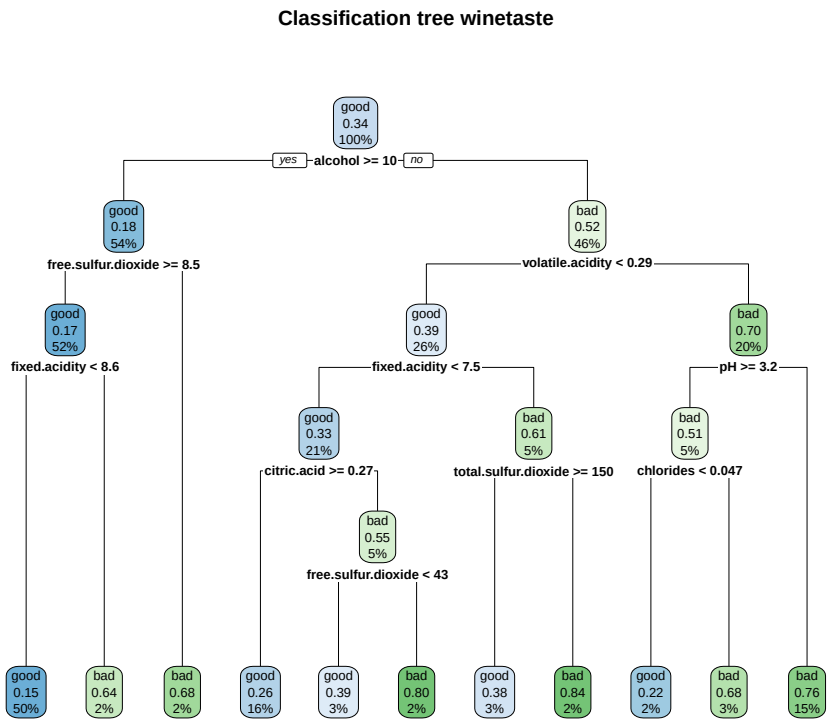
En este caso al imprimirlo como información de los nodos se muestra (además del número de nodo, la condición de la partición y el número de observaciones en el nodo) el número de observaciones mal clasificadas, la predicción y las proporciones estimadas (frecuencias relativas en la muestra de entrenamiento) de las clases:

```
tree
```

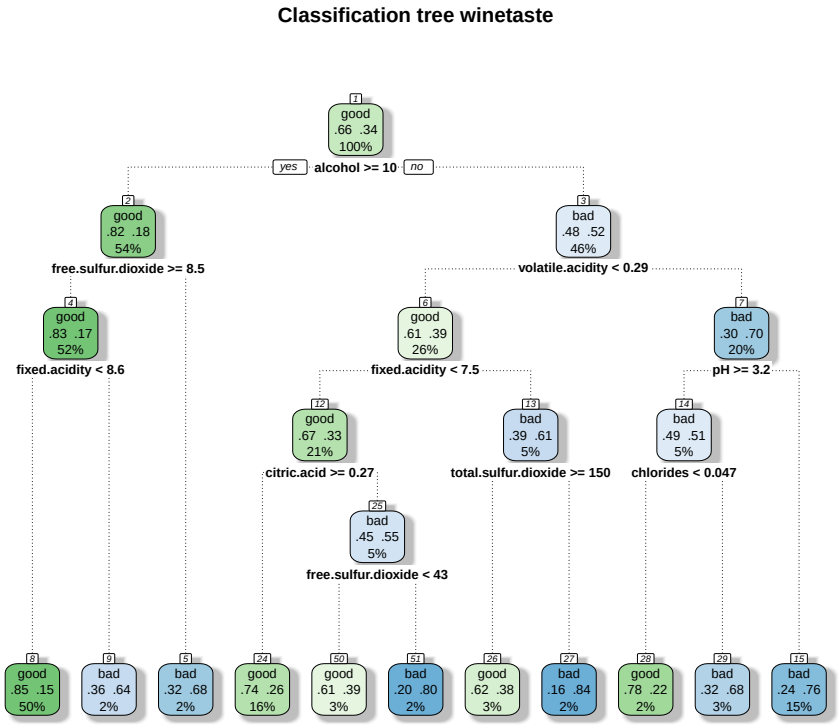
```
## n= 1000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 1000 338 good (0.6620000 0.3380000)
##    2) alcohol>=10.11667 541 100 good (0.8151571 0.1848429)
##      4) free.sulfur.dioxide>=8.5 522  87 good (0.8333333 0.1666667)
##        8) fixed.acidity< 8.55 500  73 good (0.8540000 0.1460000) *
##        9) fixed.acidity>=8.55 22   8 bad (0.3636364 0.6363636) *
##      5) free.sulfur.dioxide< 8.5 19   6 bad (0.3157895 0.6842105) *
##    3) alcohol< 10.11667 459 221 bad (0.4814815 0.5185185)
##      6) volatile.acidity< 0.2875 264 102 good (0.6136364 0.3863636)
##       12) fixed.acidity< 7.45 213  71 good (0.6666667 0.3333333)
##         24) citric.acid>=0.265 160  42 good (0.7375000 0.2625000) *
##         25) citric.acid< 0.265 53  24 bad (0.4528302 0.5471698)
##           50) free.sulfur.dioxide< 42.5 33  13 good (0.6060606 0.3939394) *
##           51) free.sulfur.dioxide>=42.5 20   4 bad (0.2000000 0.8000000) *
##       13) fixed.acidity>=7.45 51  20 bad (0.3921569 0.6078431)
##         26) total.sulfur.dioxide>=150 26  10 good (0.6153846 0.3846154) *
##         27) total.sulfur.dioxide< 150 25   4 bad (0.1600000 0.8400000) *
##      7) volatile.acidity>=0.2875 195  59 bad (0.3025641 0.6974359)
##       14) pH>=3.235 49  24 bad (0.4897959 0.5102041)
##         28) chlorides< 0.0465 18   4 good (0.7777778 0.2222222) *
##         29) chlorides>=0.0465 31  10 bad (0.3225806 0.6774194) *
##       15) pH< 3.235 146  35 bad (0.2397260 0.7602740) *
```

También puede ser preferible emplear el paquete `rpart.plot` para representarlo:

```
library(rpart.plot)
rpart.plot(tree, main="Classification tree winetaste") # Alternativa: rattle::fancyRpartPlot
```




```
rpart.plot(tree, main="Classification tree winetaste",
           extra = 104,           # show fitted class, probs, percentages
           box.palette = "GnBu", # color scheme
           branch.lty = 3,        # dotted branch lines
           shadow.col = "gray",  # shadows under the node boxes
           nn = TRUE)             # display the node numbers
```



Nos interesa como se clasificaría a una nueva observación (como se llega a los nodos terminales) y su probabilidad estimada (la frecuencia relativa de la clase más frecuente en el correspondiente nodo terminal). Al igual que en el caso de regresión, puede ser de utilidad imprimir las reglas:

```
rpart.rules(tree, style = "tall")
```

```
## taste is 0.15 when
##     alcohol >= 10
##     fixed.acidity < 8.6
##     free.sulfur.dioxide >= 8.5
##
## taste is 0.22 when
##     alcohol < 10
##     volatile.acidity >= 0.29
##     pH >= 3.2
##     chlorides < 0.047
##
## taste is 0.26 when
##     alcohol < 10
##     volatile.acidity < 0.29
##     fixed.acidity < 7.5
##     citric.acid >= 0.27
##
## taste is 0.38 when
##     alcohol < 10
##     volatile.acidity < 0.29
##     fixed.acidity >= 7.5
##     total.sulfur.dioxide >= 150
##
## taste is 0.39 when
##     alcohol < 10
```

```
##     volatile.acidity < 0.29
##     fixed.acidity < 7.5
##     free.sulfur.dioxide < 42.5
##     citric.acid < 0.27
##
## taste is 0.64 when
##     alcohol >= 10
##     fixed.acidity >= 8.6
##     free.sulfur.dioxide >= 8.5
##
## taste is 0.68 when
##     alcohol < 10
##     volatile.acidity >= 0.29
##     pH >= 3.2
##     chlorides >= 0.047
##
## taste is 0.68 when
##     alcohol >= 10
##     free.sulfur.dioxide < 8.5
##
## taste is 0.76 when
##     alcohol < 10
##     volatile.acidity >= 0.29
##     pH < 3.2
##
## taste is 0.80 when
##     alcohol < 10
##     volatile.acidity < 0.29
##     fixed.acidity < 7.5
##     free.sulfur.dioxide >= 42.5
##     citric.acid < 0.27
##
## taste is 0.84 when
##     alcohol < 10
##     volatile.acidity < 0.29
##     fixed.acidity >= 7.5
##     total.sulfur.dioxide < 150
```

Al igual que en el caso anterior, para seleccionar un valor óptimo del (hiper)parámetro de complejidad, se puede construir un árbol de decisión completo y emplear validación cruzada para podarlo. Además, si el número de observaciones es grande y las clases están más o menos balanceadas, se podría aumentar los valores mínimos de observaciones en los nodos intermedios y terminales[4], por ejemplo:

```
tree <- rpart(taste ~ ., data = train, cp = 0, minsplit = 30, minbucket = 10)
```

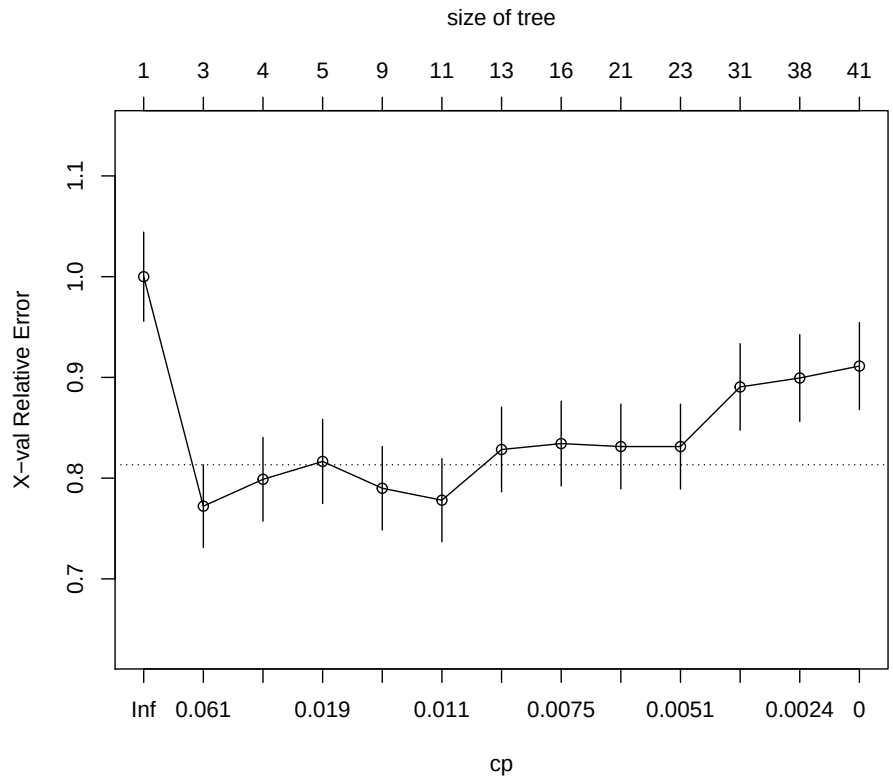
En este caso mantenemos el resto de valores por defecto:

```
tree <- rpart(taste ~ ., data = train, cp = 0)
```

Representamos los errores (reescalados) de validación cruzada:

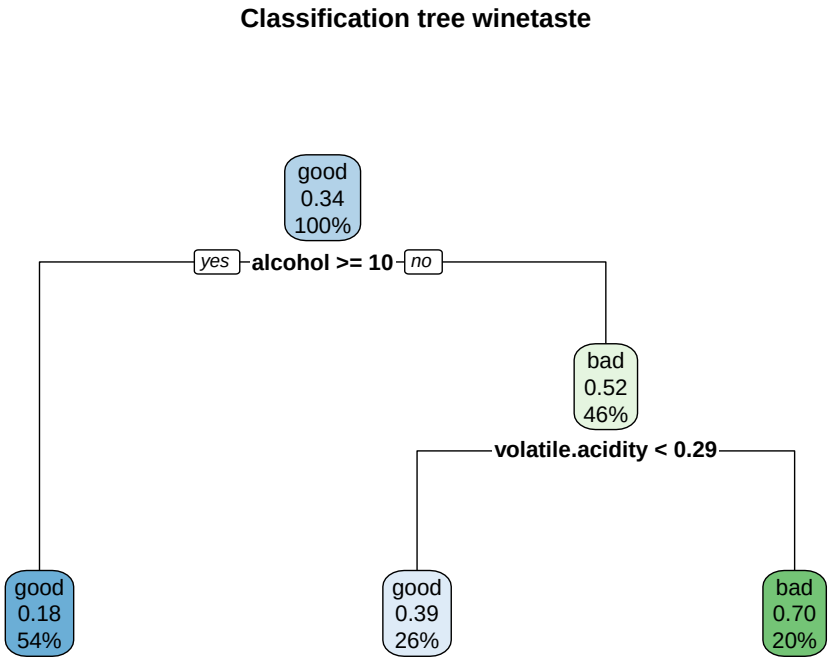
```
# printcp(tree)
plotcp(tree)
```

[4]Otra opción, más interesante para regresión, sería considerar estos valores como hiperparámetros.



Para obtener el modelo final, seleccionamos el valor óptimo de complejidad siguiendo el criterio de un error estándar de Breiman et al. (1984) y podamos el arbol:

```
xerror <- tree$cptable[,"xerror"]
imin.xerror <- which.min(xerror)
upper.xerror <- xerror[imin.xerror] + tree$cptable[imin.xerror, "xstd"]
icp <- min(which(xerror <= upper.xerror))
cp <- tree$cptable[icp, "CP"]
tree <- prune(tree, cp = cp)
# tree
# summary(tree)
# caret::varImp(tree)
# importance <- tree$variable.importance
# importance <- round(100*importance/sum(importance), 1)
# importance[importance >= 1]
rpart.plot(tree, main="Classification tree winetaste")
```



El último paso sería evaluarlo en la muestra de test siguiendo los pasos descritos en la Sección 1.3.5. El método `predict()` por defecto (`type = "prob"`) devuelve una matriz con las probabilidades de cada clase, habrá que establecer `type = "class"` (para más detalles consultar la ayuda de `predic.rpart()`).

```
obs <- test$taste
head(predict(tree, newdata = test))
```

```
##         good       bad
## 1  0.3025641 0.6974359
## 4  0.8151571 0.1848429
## 9  0.8151571 0.1848429
## 10 0.8151571 0.1848429
## 12 0.8151571 0.1848429
## 16 0.8151571 0.1848429
```

```
pred <- predict(tree, newdata = test, type = "class")
table(obs, pred)
```

```
##       pred
## obs    good bad
##   good  153  13
##   bad    54  30
```

```
caret::confusionMatrix(pred, obs)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  153  54
##       bad    13  30
##
```

```
##                Accuracy : 0.732
##                  95% CI : (0.6725, 0.7859)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.01247
##
##                   Kappa : 0.3171
##
##  Mcnemar's Test P-Value : 1.025e-06
##
##             Sensitivity : 0.9217
##             Specificity : 0.3571
##          Pos Pred Value : 0.7391
##          Neg Pred Value : 0.6977
##              Prevalence : 0.6640
##          Detection Rate : 0.6120
##    Detection Prevalence : 0.8280
##       Balanced Accuracy : 0.6394
##
##        'Positive' Class : good
##
```

2.3.3 Interfaz de caret

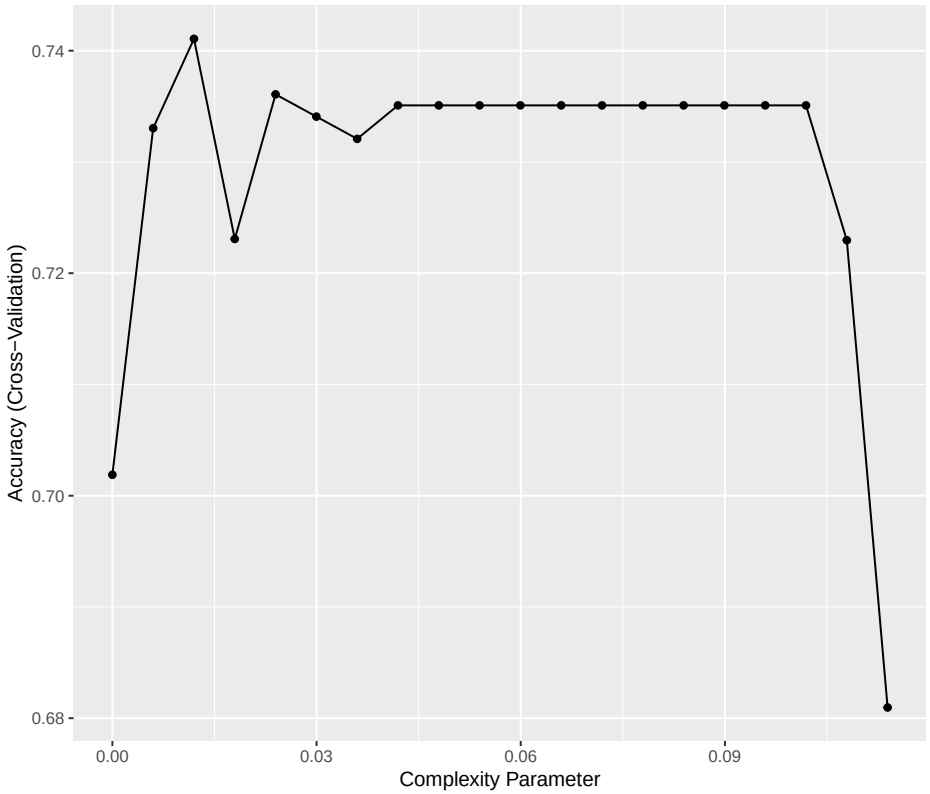
En `caret` podemos ajustar un árbol CART seleccionando `method = "rpart"`. Por defecto emplea bootstrap de las observaciones para seleccionar el valor óptimo del hiperparámetro `cp` (considerando únicamente tres posibles valores). Si queremos emplear validación cruzada como en el caso anterior podemos emplear la función auxiliar `trainControl()` y para considerar un mayor rango de posibles valores, el argumento `tuneLength`.

```
library(caret)
# names(getModelInfo()) # Listado de todos los métodos disponibles
# modelLookup("rpart")  # Información sobre hiperparámetros
set.seed(1)
# itrain <- <- createDataPartition(winetaste$taste, p = 0.8, list = FALSE)
# train <- winetaste[itrain, ]
# test <- winetaste[-itrain, ]
caret.rpart <- train(taste ~ ., method = "rpart", data = train,
                     tuneLength = 20,
                     trControl = trainControl(method = "cv", number = 10))
caret.rpart
```

```
## CART
##
## 1000 samples
##   11 predictor
##    2 classes: 'good', 'bad'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 901, 900, 900, 900, 900, 900, ...
## Resampling results across tuning parameters:
##
##   cp           Accuracy   Kappa
##   0.000000000  0.7018843  0.3487338
##   0.005995017  0.7330356  0.3870552
##   0.011990034  0.7410655  0.3878517
```

```
##   0.017985051  0.7230748  0.3374518
##   0.023980069  0.7360748  0.3698691
##   0.029975086  0.7340748  0.3506377
##   0.035970103  0.7320748  0.3418235
##   0.041965120  0.7350849  0.3422651
##   0.047960137  0.7350849  0.3422651
##   0.053955154  0.7350849  0.3422651
##   0.059950171  0.7350849  0.3422651
##   0.065945188  0.7350849  0.3422651
##   0.071940206  0.7350849  0.3422651
##   0.077935223  0.7350849  0.3422651
##   0.083930240  0.7350849  0.3422651
##   0.089925257  0.7350849  0.3422651
##   0.095920274  0.7350849  0.3422651
##   0.101915291  0.7350849  0.3422651
##   0.107910308  0.7229637  0.2943312
##   0.113905325  0.6809637  0.1087694
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.01199003.
```

```
ggplot(caret.rpart)
```

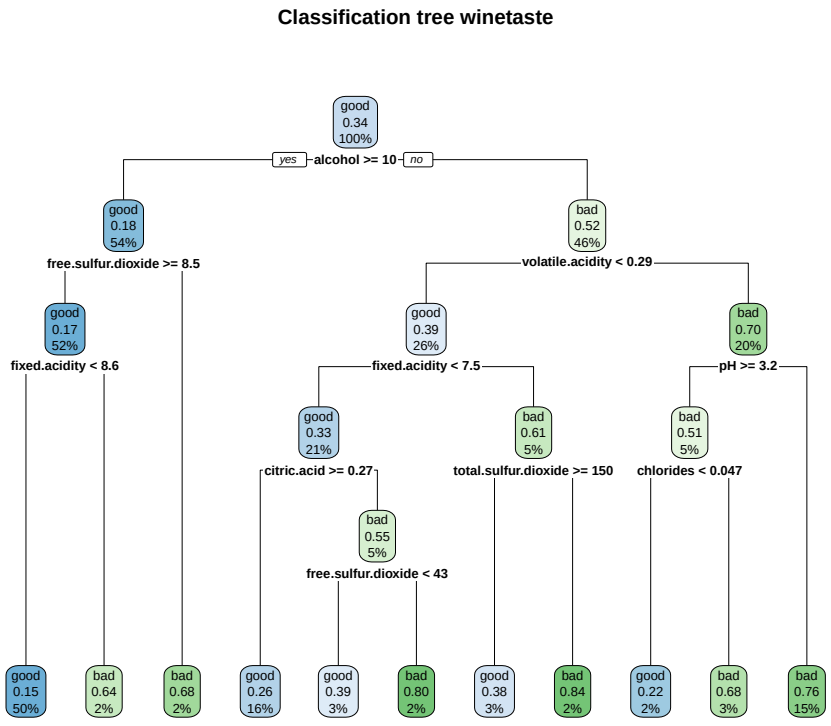



```
caret.rpart$finalModel
```

```
## n= 1000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 1000 338 good (0.6620000 0.3380000)
##    2) alcohol>=10.11667 541 100 good (0.8151571 0.1848429)
```

```
##      4) free.sulfur.dioxide>=8.5 522  87 good (0.8333333 0.1666667)
##        8) fixed.acidity< 8.55 500  73 good (0.8540000 0.1460000) *
##        9) fixed.acidity>=8.55 22   8 bad (0.3636364 0.6363636) *
##      5) free.sulfur.dioxide< 8.5 19   6 bad (0.3157895 0.6842105) *
##    3) alcohol< 10.11667 459 221 bad (0.4814815 0.5185185)
##      6) volatile.acidity< 0.2875 264 102 good (0.6136364 0.3863636)
##       12) fixed.acidity< 7.45 213  71 good (0.6666667 0.3333333)
##         24) citric.acid>=0.265 160  42 good (0.7375000 0.2625000) *
##         25) citric.acid< 0.265 53  24 bad (0.4528302 0.5471698)
##           50) free.sulfur.dioxide< 42.5 33  13 good (0.6060606 0.3939394) *
##           51) free.sulfur.dioxide>=42.5 20   4 bad (0.2000000 0.8000000) *
##       13) fixed.acidity>=7.45 51  20 bad (0.3921569 0.6078431)
##         26) total.sulfur.dioxide>=150 26  10 good (0.6153846 0.3846154) *
##         27) total.sulfur.dioxide< 150 25   4 bad (0.1600000 0.8400000) *
##      7) volatile.acidity>=0.2875 195  59 bad (0.3025641 0.6974359)
##       14) pH>=3.235 49  24 bad (0.4897959 0.5102041)
##         28) chlorides< 0.0465 18   4 good (0.7777778 0.2222222) *
##         29) chlorides>=0.0465 31  10 bad (0.3225806 0.6774194) *
##       15) pH< 3.235 146  35 bad (0.2397260 0.7602740) *
```

```
rpart.plot(caret.rpart$finalModel, main="Classification tree winetaste")
```



Para utilizar la regla de "un error estándar" se puede añadir `selectionFunction = "oneSE"`

```
set.seed(1)
caret.rpart <- train(taste ~ ., method = "rpart", data = train,
                     tuneLength = 20,
                     trControl = trainControl(method = "cv", number = 10,
                                              selectionFunction = "oneSE"))
caret.rpart
```

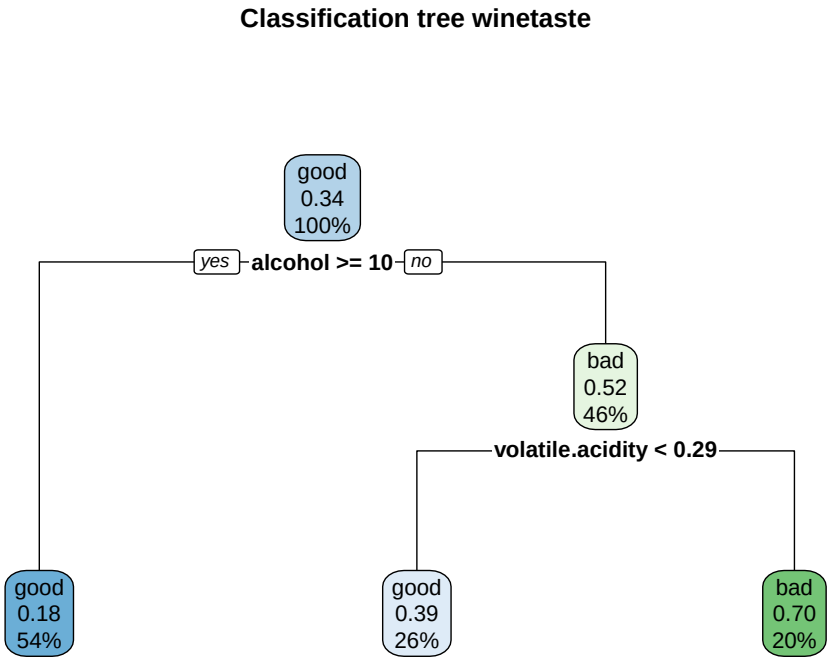
```
## CART
```

```
##
## 1000 samples
##   11 predictor
##    2 classes: 'good', 'bad'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 901, 900, 900, 900, 900, 900, ...
## Resampling results across tuning parameters:
##
##   cp           Accuracy   Kappa
##   0.000000000  0.7018843  0.3487338
##   0.005995017  0.7330356  0.3870552
##   0.011990034  0.7410655  0.3878517
##   0.017985051  0.7230748  0.3374518
##   0.023980069  0.7360748  0.3698691
##   0.029975086  0.7340748  0.3506377
##   0.035970103  0.7320748  0.3418235
##   0.041965120  0.7350849  0.3422651
##   0.047960137  0.7350849  0.3422651
##   0.053955154  0.7350849  0.3422651
##   0.059950171  0.7350849  0.3422651
##   0.065945188  0.7350849  0.3422651
##   0.071940206  0.7350849  0.3422651
##   0.077935223  0.7350849  0.3422651
##   0.083930240  0.7350849  0.3422651
##   0.089925257  0.7350849  0.3422651
##   0.095920274  0.7350849  0.3422651
##   0.101915291  0.7350849  0.3422651
##   0.107910308  0.7229637  0.2943312
##   0.113905325  0.6809637  0.1087694
##
## Accuracy was used to select the optimal model using  the one SE rule.
## The final value used for the model was cp = 0.1019153.
```

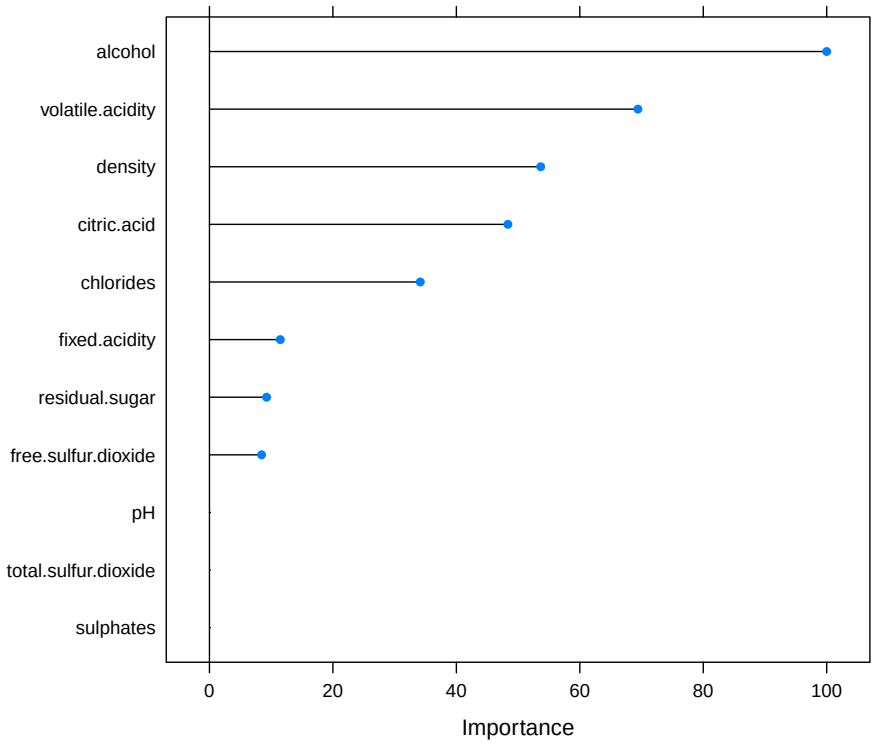
```
# ggplot(caret.rpart)
caret.rpart$finalModel
```

```
## n= 1000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 1000 338 good (0.6620000 0.3380000)
##   2) alcohol>=10.11667 541 100 good (0.8151571 0.1848429) *
##   3) alcohol< 10.11667 459 221 bad (0.4814815 0.5185185)
##     6) volatile.acidity< 0.2875 264 102 good (0.6136364 0.3863636) *
##     7) volatile.acidity>=0.2875 195  59 bad (0.3025641 0.6974359) *
```

```
rpart.plot(caret.rpart$finalModel, main = "Classification tree winetaste")
```




```
var.imp <- varImp(caret.rpart)
plot(var.imp)
```



Para calcular las predicciones (o las estimaciones de las probabilidades) podemos emplear el método `predict.train()` y posteriormente `confusionMatrix()` para evaluar su precisión:

```
pred <- predict(caret.rpart, newdata = test)
# p.est <- predict(caret.rpart, newdata = test, type = "prob")
confusionMatrix(pred, test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  153  54
##       bad    13  30
##
##                Accuracy : 0.732
##                  95% CI : (0.6725, 0.7859)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.01247
##
##                   Kappa : 0.3171
##
##  Mcnemar's Test P-Value : 1.025e-06
##
##             Sensitivity : 0.9217
##             Specificity : 0.3571
##          Pos Pred Value : 0.7391
##          Neg Pred Value : 0.6977
##              Prevalence : 0.6640
##          Detection Rate : 0.6120
##    Detection Prevalence : 0.8280
##       Balanced Accuracy : 0.6394
##
##        'Positive' Class : good
##
```

NOTA: En principio también se podría utilizar la regla de "un error estándar" seleccionando `method = "rpart1SE"` (pero `caret` implementa internamente este método y en ocasiones no se obtienen los resultados esperados).

```
set.seed(1)
caret.rpart <- train(taste ~ ., method = "rpart1SE", data = train)
caret.rpart
printcp(caret.rpart$finalModel)
caret.rpart$finalModel
rpart.plot(caret.rpart$finalModel, main = "Classification tree winetaste")
varImp(caret.rpart)
```

2.4 Alternativas a los árboles CART

Una de las alternativas más populares es la metodología C4.5 (Quinlan, 1993), evolución de ID3 (1986), que en estos momentos se encuentra en la versión C5.0 (y es ya muy similar a CART). C5.0 se utiliza sólo para clasificación e incorpora *boosting* (que veremos en el tema siguiente). Esta metodología está implementada en el paquete `C50`.

Ross Quinlan desarrolló también la metodologia M5 (Quinlan, 1992) para regresión. Su principal característica es que los nodos terminales, en lugar de contener un número, contienen un modelo (de regresión) lineal. El paquete `Cubist` es una evolución de M5 que incorpora un método *ensemble* similar a *boosting*.

La motivación detrás de M5 es que, si la predicción que aporta un nodo terminal se limita a un único número (como hace la metodología CART), entonces el modelo va a predecir muy mal los valores que *realmente* son muy extremos, ya que el número de posibles valores predichos está limitado por el número de nodos terminales, y en cada uno de ellos se utiliza una media. Por ello M5 le asocia a cada nodo un modelo de regresión lineal, para cuyo ajuste se utilizan los datos del nodo y todas las variables que están en la ruta del nodo. Para evaluar los posibles cortes que conducen al siguiente nodo, se utilizan los propios modelos lineales para calcular la medida del error.

Una vez se ha construido todo el árbol, para realizar la predicción se puede utilizar el modelo lineal que está en el nodo terminal correspondiente, pero funciona mejor si se utiliza una combinación lineal del modelo del nodo terminal y de todos sus nodos ascendientes (es decir, los que están en su camino).

Otra opción es CHAID (CHi-squared Automated Interaction Detection, Kass, 1980), que se basa en una idea diferente. Es un método de construcción de árboles de clasificación que se utiliza cuando las variables predictoras son cualitativas o discretas; en caso contrario deben ser categorizadas previamente. Y se basa en el contraste chi-cuadrado de independencia para tablas de contingencia.

Para cada par (X_i, Y), se considera su tabla de contingencia y se calcula el p-valor del contraste chi-cuadrado, seleccionándose la variable predictora que tenga un p-valor más pequeño, ya que se asume que las variables predictoras más relacionadas con la respuesta Y son las que van a tener p-valores más pequeños y darán lugar a mejores predicciones. Se divide el nodo de acuerdo con los distintos valores de la variable predictora seleccionada, y se repite el proceso mientras haya variables *significativas*. Como el método exige que el p-valor sea menor que 0.05 (o el nivel de significación que se elija), y hay que hacer muchas comparaciones es necesario aplicar una corrección para comparaciones múltiples, por ejemplo la de Bonferroni.

Lo que acabamos de explicar daría lugar a árboles no necesariamente binarios. Como se desea trabajar con árboles binarios (si se admite que de un nodo salga cualquier número de ramas, con muy pocos niveles de profundidad del árbol ya nos quedaríamos sin datos), es necesario hacer algo más: forzar a que las variables predictoras tengan sólo dos categorías mediante un proceso de fusión. Se van haciendo pruebas chi-cuadrado entre pares de categorías y la variable respuesta, y se fusiona el par con el p-valor más alto, ya que se trata de fusionar las categorías que sean más similares.

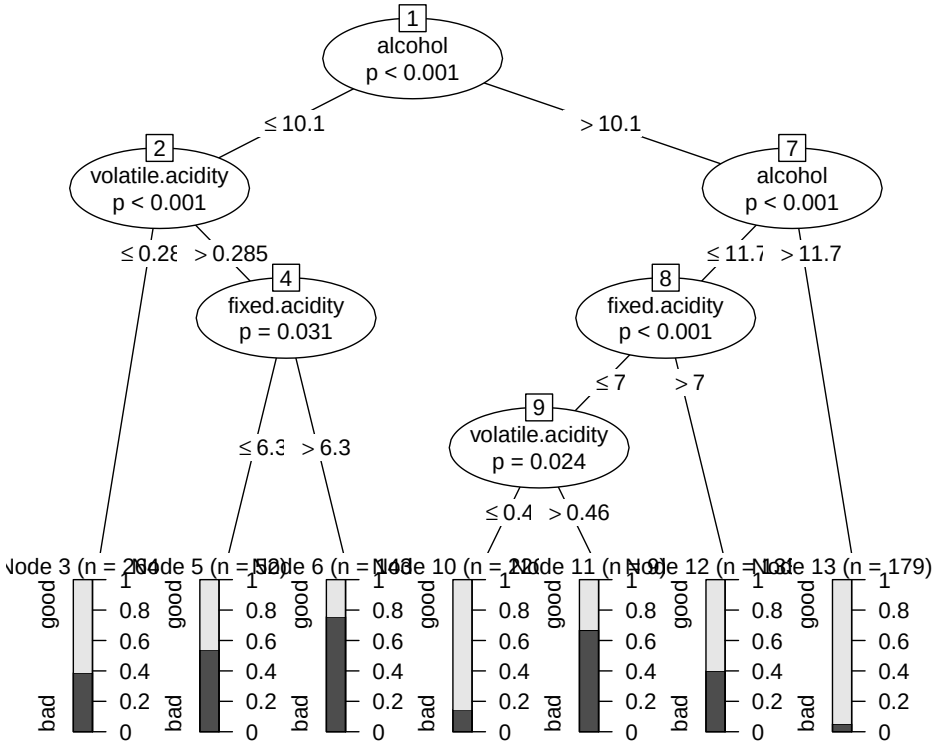
Para árboles de regresión hay metodologías que, al igual que CHAID, se basan en el cálculo de p-valores, en este caso de contrastes de igualdes de medias. Una de las más utilizadas son los *conditional inference trees* (Hothorn *et al.*, 2006)[5], implementada en la función `ctree()` del paquete `party`.

Un problema conocido de los árboles CART es que sufren un sesgo de selección de variables: los predictores con más valores distintos son favorecidos. Esta es una de las motivaciones de utilizar estos métodos basados en contrastes de hipótesis. Por otra parte hay que ser conscientes de que los contrastes de hipótesis y la calidad predictiva son cosas distintas.

2.4.1 Ejemplo

```
library(party)
tree2 <- ctree(taste ~ ., data = train)
plot(tree2)
```

[5]Otra alternativa es GUIDE (Generalized, Unbiased, Interaction Detection and Estimation; Loh, 2002).


1
alcohol
p < 0.001
≤ 10.1
> 10.1
2
volatile.acidity
p < 0.001
7
alcohol
p < 0.001
≤ 0.28
> 0.285
≤ 11.7
> 11.7
4
fixed.acidity
p = 0.031
8
fixed.acidity
p < 0.001
≤ 6.3
> 6.3
9
volatile.acidity
p = 0.024
≤ 7
> 7
≤ 0.4
> 0.46
Node 3 (n = 204)
Node 5 (n = 52)
Node 6 (n = 146)
Node 10 (n = 126)
Node 11 (n = 98)
Node 12 (n = 138)
Node 13 (n = 179)
good
bad
1
0.8
0.6
0.4
0.2
0

Capítulo 3

Bagging y Boosting

Tanto el *bagging* como el *boosting* son procedimientos generales para la reducción de la varianza de un método estadístico de aprendizaje.

La idea básica consiste en combinar métodos de predicción sencillos (débiles), es decir, con poca capacidad predictiva, para obtener un método de predicción muy potente (y robusto). Estas ideas se pueden aplicar tanto a problemas de regresión como de clasificación.

Son muy empleados con árboles de decisión: son predictores débiles y se generan de forma rápida. Lo que se hace es construir muchos modelos (crecer muchos árboles) que luego se combinan para producir predicciones (promediando o por consenso).

3.1 Bagging

En la década de 1990 empiezan a utilizarse los métodos *ensemble* (métodos combinados), esto es, métodos predictivos que se basan en combinar las predicciones de cientos de modelos. Uno de los primeros métodos combinados que se utilizó fue el *bagging* (nombre que viene de *bootstrap aggregation*), propuesto en Breiman (1996). Es un método general de reducción de la varianza que se basa en la utilización del bootstrap junto con un modelo de regresión o de clasificación, como puede ser un árbol de decisión.

La idea es muy sencilla. Si disponemos de muchas muestras de entrenamiento, podemos utilizar cada una de ellas para entrenar un modelo que después nos servirá para hacer una predicción. De este modo tendremos tantas predicciones como modelos y por tanto tantas predicciones como muestras de entrenamiento. El procedimiento consistente en promediar todas las predicciones anteriores tiene dos ventajas importantes: simplifica la solución y reduce mucho la varianza.

El problema es que en la práctica no suele disponerse más que de una única muestra de entrenamiento. Aquí es donde entra en juego el bootstrap, técnica especialmente útil para estimar varianzas, pero que en esta aplicación se utiliza para reducir la varianza. Lo que se hace es generar cientos o miles de muestras bootstrap a partir de la muestra de entrenamiento, y después utilizar cada una de estas muestras bootstrap como una muestra de entrenamiento (*bootstrapped training data set*).

Para un modelo que tenga intrínsecamente poca variabilidad, como puede ser una regresión lineal, aplicar bagging puede ser poco interesante, ya que hay poco márgen para mejorar el rendimiento. Por contra, es un método muy importante para los árboles de decisión, porque un árbol con mucha profundidad (sin podar) tiene mucha variabilidad: si modificamos ligeramente los datos de entrenamiento es muy posible que se obtenga un nuevo árbol completamente distinto al anterior; y esto se ve como un inconveniente. Por esa razón, en este contexto encaja perfectamente la metodología bagging.

Así, para árboles de regresión se hacen crecer muchos árboles (sin poda) y se calcula la media de las predicciones. En el caso de los árboles de clasificación lo más sencillo es sustituir la media por la moda y utilizar el criterio del voto mayoritario: cada modelo tiene el mismo peso y por tanto cada

modelo aporta un voto. Además, la proporción de votos de cada categoría es una estimación de su probabilidad.

Una ventaja adicional del bagging es que permite estimar el error de la predicción de forma directa, sin necesidad de utilizar una muestra de test o de aplicar validación cruzada u, otra vez, remuestreo, y se obtiene un resultado similar al que obtendríamos con estos métodos. Es bien sabido que una muestra bootstrap va a contener muchas observaciones repetidas y que, en promedio, sólo utiliza aproximadamente dos tercios de los datos (para ser más precisos, $1 - (1 - 1/n)^n \approx 1 - e^{-1} = 0.6321$ al aumentar el tamaño del conjunto de datos de entrenamiento). Un dato que no es utilizado para construir un árbol se denomina un dato *out-of-bag* (OOB). De este modo, para cada observación se pueden utilizar los árboles para los que esa observación es *out-of-bag* (aproximadamente una tercera parte de los árboles construidos) para generar una única predicción para ella. Repitiendo el proceso para todas las observaciones se obtiene una medida del error.

Una decisión que hay que tomar es cuántas muestras bootstrap se toman (o lo que es lo mismo, cuántos árboles se construyen). Realmente se trata de una aproximación Monte Carlo, por lo que típicamente se estudia gráficamente la convergencia del error OOB al aumentar el número de árboles (para más detalles ver p.e. Fernández-Casal y Cao, 2020, Sección 4.1). Si aparentemente hay convergencia con unos pocos cientos de árboles, no va a variar mucho el nivel de error al aumentar el número. Por tanto aumentar mucho el número de árboles no mejora las predicciones, aunque tampoco aumenta el riesgo de sobreajuste. Los costes computacionales aumentan con el número de árboles, pero la construcción y evaluación del modelo son fácilmente paralelizables (aunque pueden llegar a requerir mucha memoria si el conjunto de datos es muy grande). Por otra parte si el número de árboles es demasiado pequeño puede que se obtengan pocas (o incluso ninguna) predicciones OOB para alguna de las observaciones de la muestra de entrenamiento.

Una ventaja que ya sabemos que tienen los árboles de decisión es su fácil interpretabilidad. En un árbol resulta evidente cuales son los predictores más influyentes. Al utilizar bagging se mejora (mucho) la predicción, pero se pierde la interpretabilidad. Aún así, hay formas de calcular la importancia de los predictores. Por ejemplo, si fijamos un predictor y una medida del error podemos, para cada uno de los árboles, medir la reducción del error que se consigue cada vez que hay un corte que utilice ese predictor particular. Promediando sobre todos los árboles bagging se obtiene una medida global de la importancia: un valor alto en la reducción del error sugiere que el predictor es importante.

En resumen:

- Se remuestrea repetidamente el conjunto de datos de entrenamiento.
- Con cada conjunto de datos se entrena un modelo.
- Las predicciones se obtienen promediando las predicciones de los modelos (la decisión mayoritaria en el caso de clasificación).
- Se puede estimar la precisión de las predicciones con el error OOB (out-of-bag).

3.2 Bosques aleatorios

Los bosques aleatorios (*random forest*) son una variante de bagging específicamente diseñados para trabajar con árboles de decisión. Las muestras bootstrap que se generan al hacer bagging introducen un elemento de aleatoriedad que en la práctica provoca que todos los árboles sean distintos, pero en ocasiones no son lo *suficientemente* distintos. Es decir, suele ocurrir que los árboles tengan estructuras muy similares, especialmente en la parte alta, aunque después se vayan diferenciando según se desciende por ellos. Esta característica se conoce como correlación entre árboles y se da cuando el árbol es un modelo adecuado para describir la relación ente los predictores y la respuesta, y también cuándo uno de los predictores es muy fuerte, es decir, es especialmente relevante, con lo cual casi siempre va a estar en el primer corte. Esta correlación entre árboles se va a traducir en una correlación entre sus predicciones (más formalmente, entre los predictores).

Promediar variables altamente correladas produce una reducción de la varianza mucho menor que si promediamos variables incorreladas. La solución pasa por añadir aleatoriedad al proceso de construcción de los árboles, para que estos dejen de estar correlados. Hubo varios intentos, entre los que destaca Dietterich (2000) al proponer la idea de introducir aleatorieadad en la selección de las variables de cada corte. Breiman (2001) propuso un algoritmo unificado al que llamó bosques aleatorios. En la construcción de cada uno de los árboles que finalmente constituirán el bosque, se van haciendo cortes binarios, y para cada corte hay que seleccionar una variable predictora. La modificación introducida fue que antes de hacer cada uno de los cortes, de todas las p variables predictoras, se seleccionan al azar $m < p$ predictores que van a ser los candidatos para el corte.

El hiperparámetro de los bosques aleatorios es m, y se puede seleccionar mediante las técnicas habituales. Como puntos de partida razonables se pueden considerar $m = \sqrt{p}$ (para problemas de clasificación) y $m = p/3$ (para problemas de regresión). El número de árboles que van a constituir el bosque también puede tratarse como un hiperparámetro, aunque es más frecuente tratarlo como un problema de convergencia. En general, van a hacer falta más árboles que en bagging.

Los bosques aleatorios son computacionalmente más eficientes que bagging porque, aunque como acabamos de decir requieren más árboles, la construcción de cada árbol es mucho más rápida al evaluarse sólo unos pocos predictores en cada corte.

Este método también puede ser empleado para aprendizaje no supervisado, por ejemplo se puede construir una matriz de proximidad entre observaciones a partir de la proporción de veces que están en un mismo nodo terminal (para más detalles ver Liaw y Wiener, 2002).

En resumen:

- Los bosques aleatorios son una modificación del bagging para el caso de árboles de decisión.
- También se introduce aleatoriedad en las variables, no sólo en las observaciones.
- Para evitar dependencias, los posibles predictores se seleccionan al azar en cada nodo (e.g. $m = \sqrt{p}$).
- Se utilizan árboles sin podar.
- Estos métodos dificultan la interpretación.
- Se puede medir la importancia de las variables (índices de importancia).
  - Por ejemplo, para cada árbol se suman las reducciones en el índice de Gini correspondientes a las divisiones de un predictor y posteriormente se promedian los valores de todos los árboles.
  - Alternativamente (Breiman, 2001) se puede medir el incremento en el error de predicción OOB al permutar aleatoriamente los valores de la variable explicativa en la muestra OOB (manteniendo el resto sin cambios).

3.3 Bagging y bosques aleatorios en R

Estos algoritmos son de los más populares en AE y están implementados en numerosos paquetes de R, aunque la referencia es el paquete `randomForest` (que emplea el código Fortran desarrollado por Leo Breiman y Adele Cutler). La función principal es `randomForest()` y se suele emplear de la forma:

```
randomForest(formula, data, ntree, mtry, nodesize, ...)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (típicamente `respuesta ~ .`), aunque si el conjunto de datos es muy grande puede ser preferible emplear una matriz o un data.frame para establecer los predictores y un vector para la respuesta (sustituyendo estos argumentos por `x` e `y`).
- `ntree`: número de árboles que se crecerán; por defecto 500.

- `mtry`: número de predictores seleccionados al azar en cada división; por defecto `max(floor(p/3), 1)` en el caso de regresión y `floor(sqrt(p))` en clasificación, siendo `p = ncol(x) = ncol(data) - 1` el número de predictores.
- `nodesize`: número mínimo de observaciones en un nodo terminal; por defecto 1 en clasificación y 5 en regresión (puede ser recomendable incrementarlo si el conjunto de datos es muy grande, para evitar posibles problemas de sobreajuste, dismunir el tiempo de computación y los requerimientos de memoria; también podría ser considerado como un hiperparámetro).

Otros argumentos que pueden ser de interés[1] son:

- `maxnodes`: número máximo de nodos terminales (como alternativa para la establecer la complejidad).
- `importance = TRUE`: permite obtener medidas adicionales de importancia.
- `proximity = TRUE`: permite obtener una matriz de proximidades (componente `$proximity`) entre las observaciones (frecuencia con la que los pares de observaciones están en el mismo nodo terminal).
- `na.action = na.fail`: por defecto no admite datos faltantes con la interfaz de fórmulas. Si los hubiese, se podrían imputar estableciento `na.action = na.roughfix` (empleando medias o modas) o llamando previamente a `rfImpute()` (que emplea proximidades obtenidas con un bosque aleatorio).

Más detalles en la ayuda de esta función o en Liaw y Wiener (2002).

Entre las numerosas alternativas, además de las implementadas en paquetes que integran colecciones de métodos como `h2o` o `RWeka`, una de las más utilizadas son los bosques aleatorios con *conditional inference trees*, implementada en la función `cforest()` del paquete `party`.

3.3.1 Ejemplo: Clasificación con bagging

Como ejemplo consideraremos el conjunto de datos de calidad de vino empleado en la Sección 2.3.2 (para hacer comparaciones con el ajuste de un único árbol).

```
load("data/winetaste.RData")
set.seed(1)
df <- winetaste
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
```

Al ser bagging con árboles un caso particular de bosques aleatorios cuando $m = p$, también podemos emplear `randomForest`:

```
library(randomForest)
set.seed(4) # NOTA: Fijamos esta semilla para ilustrar dependencia
bagtrees <- randomForest(taste ~ ., data = train, mtry = ncol(train) - 1)
bagtrees
```

```
##
## Call:
##  randomForest(formula = taste ~ ., data = train, mtry = ncol(train) -      1)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 11
##
```

[1] Si se quiere minimizar el uso de memoria, por ejemplo mientras se seleccionan hiperparámetros, se puede establecer `keep.forest=FALSE`.

```
##         OOB estimate of  error rate: 23.5%
## Confusion matrix:
##      good bad class.error
## good  565  97   0.1465257
## bad   138 200   0.4082840
```

Con el método `plot()` podemos examinar la convergencia del error en la muestra OOB (simplemente emplea `matplot()` para representar la componente `$err.rate`):

```
plot(bagtrees, main = "Tasas de error")
legend("topright", colnames(bagtrees$err.rate), lty = 1:5, col = 1:6)
```

Como vemos que los errores se estabilizan podríamos pensar que aparentemente hay convergencia (aunque situaciones de alta dependencia entre los áboles dificultarían su interpretación).

Con la función `getTree()` podemos extraer los árboles individuales. Por ejemplo el siguiente código permite extraer la variable seleccionada para la primera división:

```
# View(getTree(bagtrees, 1, labelVar=TRUE))
split_var_1 <- sapply(seq_len(bagtrees$ntree),
                      function(i) getTree(bagtrees, i, labelVar=TRUE)[1, "split var"])
```

En este caso concreto podemos observar que siempre es la misma, lo que indicaría una alta dependencia entre los distintos árboles:

```
table(split_var_1)
```

```
## split_var_1
##              alcohol            chlorides          citric.acid
##                  500                    0                    0
##              density        fixed.acidity  free.sulfur.dioxide
##                    0                    0                    0
##                   pH       residual.sugar            sulphates
##                    0                    0                    0
```

```
## total.sulfur.dioxide     volatile.acidity
##                    0                    0
```

Por último evaluamos la precisión en la muestra de test:

```
pred <- predict(bagtrees, newdata = test)
caret::confusionMatrix(pred, test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  145  42
##       bad    21  42
##
##                Accuracy : 0.748
##                  95% CI : (0.6894, 0.8006)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.002535
##
##                   Kappa : 0.3981
##
##  Mcnemar's Test P-Value : 0.011743
##
##             Sensitivity : 0.8735
##             Specificity : 0.5000
##          Pos Pred Value : 0.7754
##          Neg Pred Value : 0.6667
##              Prevalence : 0.6640
##          Detection Rate : 0.5800
##    Detection Prevalence : 0.7480
##       Balanced Accuracy : 0.6867
##
##        'Positive' Class : good
##
```

3.3.2 Ejemplo: Clasificación con bosques aleatorios

Como ejemplo llamamos a la función `randomForest()` con las opciones por defecto:

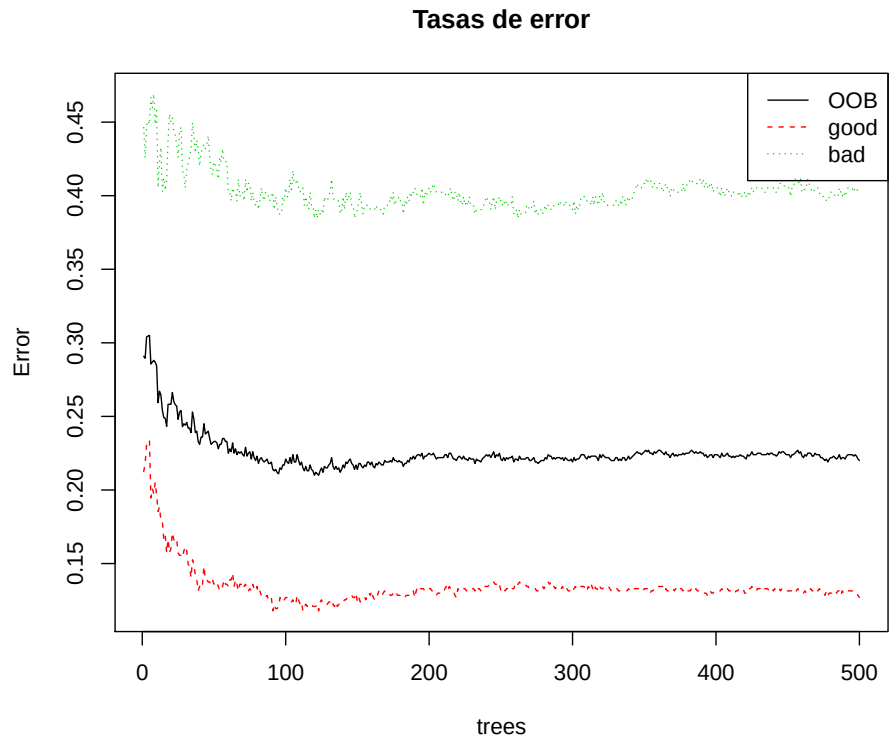
```
# load("data/winetaste.RData")
# set.seed(1)
# df <- winetaste
# nobs <- nrow(df)
# itrain <- sample(nobs, 0.8 * nobs)
# train <- df[itrain, ]
# test <- df[-itrain, ]

set.seed(1)
rf <- randomForest(taste ~ ., data = train)
rf
```

```
##
## Call:
##  randomForest(formula = taste ~ ., data = train)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
```

```
##
##         OOB estimate of  error rate: 22%
## Confusion matrix:
##      good bad class.error
## good  578  84   0.1268882
## bad   136 202   0.4023669
```

En este caso también observamos que aparentemente hay convergencia y tampoco sería necesario incrementar el número de árboles:

```
plot(rf, main = "Tasas de error")
legend("topright", colnames(rf$err.rate), lty = 1:5, col = 1:6)
```

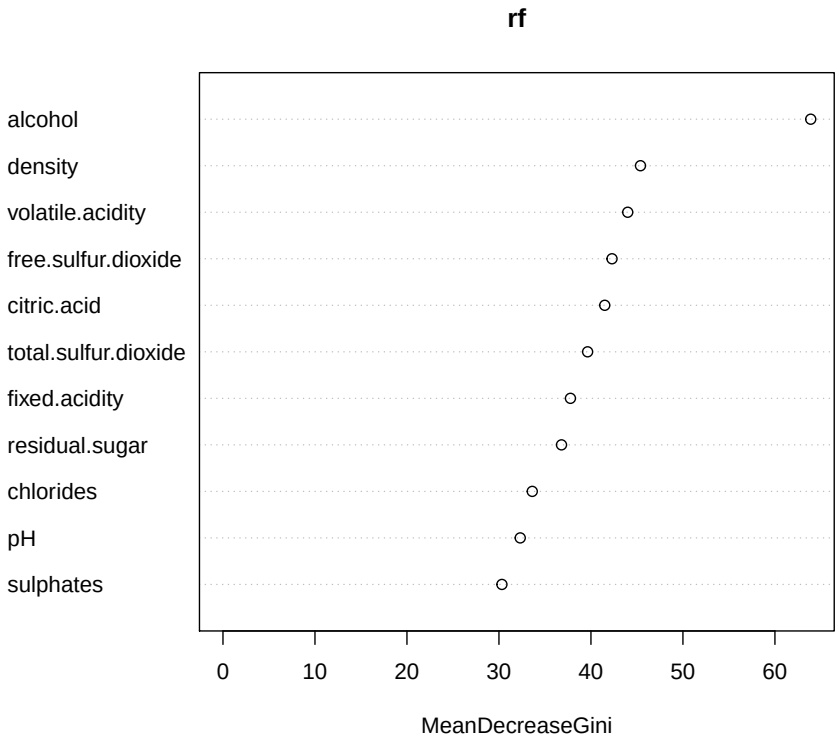


Podemos mostrar la importancia de las variables predictoras con la función `importance()` o representarlas con `varImpPlot()`:

```
importance(rf)
```

```
##                       MeanDecreaseGini
## fixed.acidity                37.77155
## volatile.acidity             43.99769
## citric.acid                  41.50069
## residual.sugar               36.79932
## chlorides                    33.62100
## free.sulfur.dioxide          42.29122
## total.sulfur.dioxide         39.63738
## density                      45.38724
## pH                           32.31442
## sulphates                    30.32322
## alcohol                      63.89185
```

```
varImpPlot(rf)
```



Si evaluamos la precisión en la muestra de test podemos observar un ligero incremento en la precisión en comparación con el método anterior:

```
pred <- predict(rf, newdata = test)
caret::confusionMatrix(pred, test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  153  43
##       bad    13  41
##
##                Accuracy : 0.776
##                  95% CI : (0.7192, 0.8261)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 7.227e-05
##
##                   Kappa : 0.4494
##
##  Mcnemar's Test P-Value : 0.0001065
##
##             Sensitivity : 0.9217
##             Specificity : 0.4881
##          Pos Pred Value : 0.7806
##          Neg Pred Value : 0.7593
##              Prevalence : 0.6640
##          Detection Rate : 0.6120
##    Detection Prevalence : 0.7840
##       Balanced Accuracy : 0.7049
##
```

```
##        'Positive' Class : good
##
```

Debida a que en este caso la dependencia entre los árboles es menor:

```
split_var_1 <- sapply(seq_len(rf$ntree),
                      function(i) getTree(rf, i, labelVar=TRUE)[1, "split var"])
table(split_var_1)
```

```
## split_var_1
##              alcohol            chlorides          citric.acid
##                  150                   49                   38
##              density        fixed.acidity  free.sulfur.dioxide
##                  114                   23                   20
##                   pH      residual.sugar            sulphates
##                   11                    0                    5
## total.sulfur.dioxide     volatile.acidity
##                   49                   41
```

3.3.3 Ejemplo: bosques aleatorios con `caret`

En paquete `caret` hay varias implementaciones de bagging y bosques aleatorios[2], incluyendo el algoritmo del paquete `randomForest` considerando como hiperparámetro el número de predictores seleccionados al azar en cada división `mtry`. Para ajustar este modelo a una muestra de entrenamiento hay que establecer `method = "rf"` en la llamada a `train()`.

```
library(caret)
# str(getModelInfo("rf", regex = FALSE))
modelLookup("rf")
```

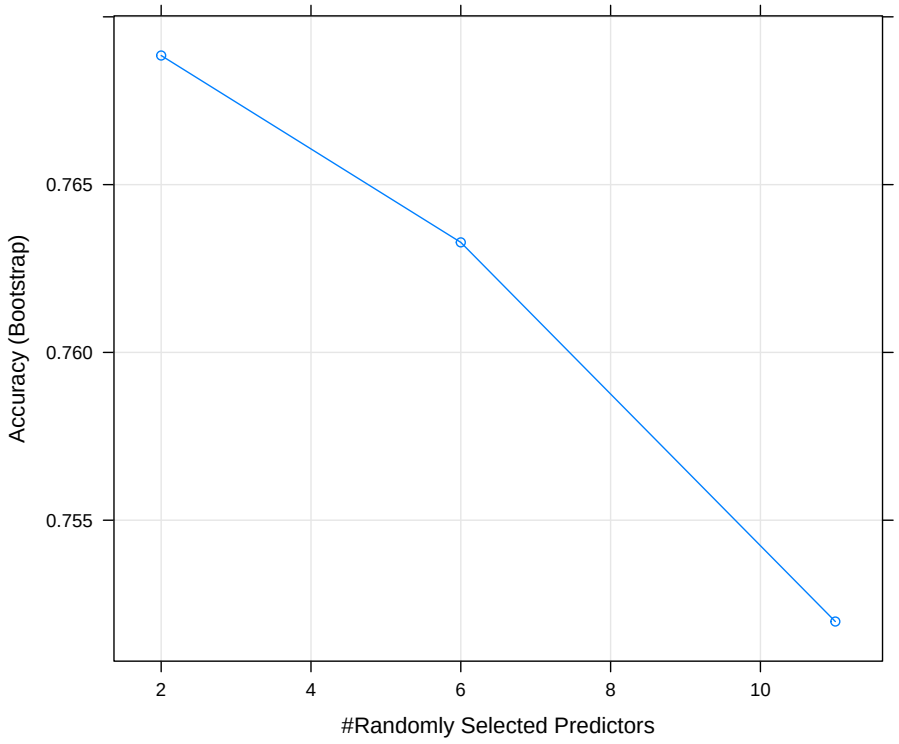
```
##   model parameter                         label forReg forClass probModel
## 1    rf      mtry #Randomly Selected Predictors   TRUE     TRUE      TRUE
```

```
# load("data/winetaste.RData")
# set.seed(1)
# df <- winetaste
# nobs <- nrow(df)
# itrain <- sample(nobs, 0.8 * nobs)
# train <- df[itrain, ]
# test <- df[-itrain, ]
```

Con las opciones por defecto únicamente evalúa tres valores posibles del hiperparámetro (se podría aumentar el número con `tuneLength` o especificarlos con `tuneGrid`), pero aún así el tiempo de computación puede ser alto (puede ser recomendable reducir el valor de `nodesize` o paralelizar los cálculos; otras implementaciones pueden ser más eficientes).

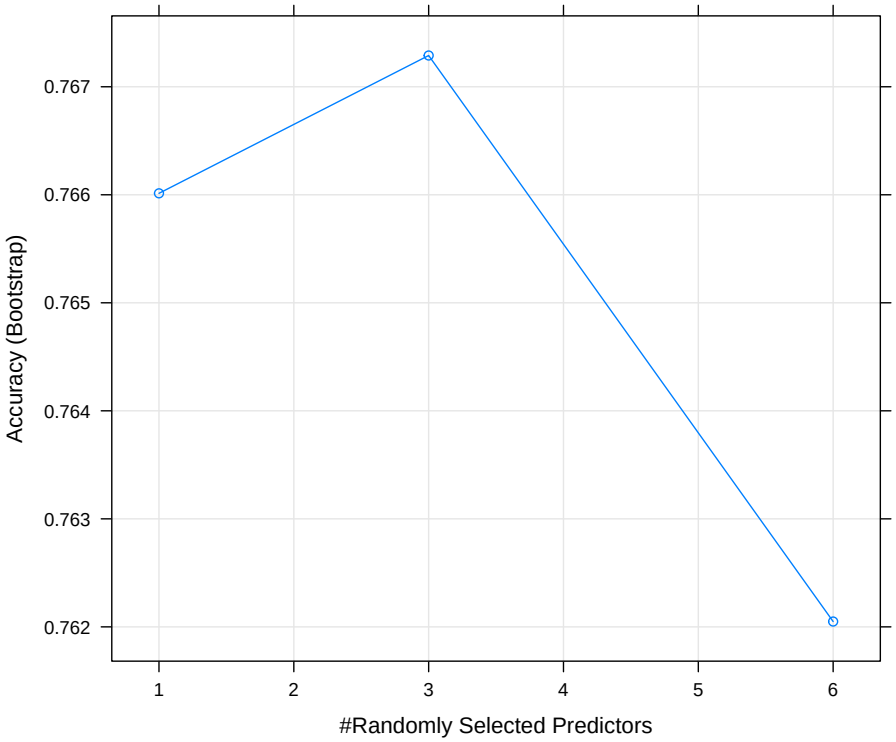
```
set.seed(1)
rf.caret <- train(taste ~ ., data = train, method = "rf")
plot(rf.caret)
```

[2]Se puede hacer una búsqueda en la tabla del Capítulo 6: Available Models del manual.



Breiman (2001) sugiere emplear el valor por defecto, la mitad y el doble:

```
mtry.class <- sqrt(ncol(train) - 1)
tuneGrid <- data.frame(mtry = floor(c(mtry.class/2, mtry.class, 2*mtry.class)))
set.seed(1)
rf.caret <- train(taste ~ ., data = train,
                  method = "rf", tuneGrid = tuneGrid)
plot(rf.caret)
```



3.4 Boosting

La metodología *boosting* es una metodología general de aprendizaje lento en la que se combinan muchos modelos obtenidos mediante un método con poca capacidad predictiva para, *impulsados*, dar lugar a un mejor predictor. Los árboles de decisión pequeños (construidos con poca profundidad) resultan perfectos para esta tarea, al ser realmente malos predictores (*weak learners*), fáciles de combinar y generarse de forma muy rápida.

El boosting nació en el contexto de los problemas de clasificación y tardó varios años en poderse extender a los problemas de regresión. Por ese motivo vamos a empezar viendo el boosting en clasificación.

La idea del boosting la desarrollaron Valiant (1984) y Kearns y Valiant (1989), pero encontrar una implementación efectiva fue una tarea difícil que no se resolvió satisfactoriamente hasta que Freund y Schapire (1996) presentaron el algoritmo *AdaBoost*, que rápidamente se convirtió en un éxito.

Veamos, de forma muy esquemática, en que consiste el algoritmo AdaBoost para un problema de clasificación en el que sólo hay dos categorías y en el que se utiliza como clasificador débil un árbol de decisión con pocos nodos terminales, sólo marginalmente superior a un clasificador aleatorio. En este caso resulta más cómodo recodificar la variable indicadora Y como 1 si éxito y -1 si fracaso.

1. Seleccionar B, número de iteraciones.
2. Se les asigna el mismo peso a todas las observaciones de la muestra de entrenamiento $(1/n)$.
3. Para $b = 1, 2, \ldots, B$, repetir:
    a. Ajustar el árbol utilizando las observaciones ponderadas.
    b. Calcular la proporción de errores en la clasificación e_b.
    c. Calcular $s_b = \log((1 - e_b)/e_b)$.

   d. Actualizar los pesos de las observaciones. Los pesos de las observaciones correctamente clasificadas no cambian; se les da más peso a las observaciones incorrectamente clasificadas, multiplicando su peso anterior por $(1 - e_b)/e_b$.

4. Dada una observación $\mathbf{x}$, si denotamos por $\hat{y}_b(\mathbf{x})$ su clasificación utilizando árbol b-ésimo, entonces $\hat{y}(\mathbf{x}) = signo\,(\sum_b s_b \hat{y}_b(\mathbf{x}))$ (si la suma es positiva, se clasifica la observación como perteneciente a la clase +1, en caso contrario a la clase -1).

Vemos que el algoritmo AdaBoost no combina árboles independientes (como sería el caso de los bosques aleatorios, por ejemplo), sino que estos se van generando en una secuencia en la que cada árbol depende del anterior. Se utiliza siempre el mismo conjunto de datos (de entrenamiento), pero a estos datos se les van poniendo unos pesos en cada iteración que dependen de lo que ha ocurrido en la iteración anterior: se les da más peso a las observaciones mal clasificadas para que en sucesivas iteraciones se clasifiquen bien. Finalmente, la combinación de los árboles se hace mediante una suma ponderada de las B clasificaciones realizadas. Los pesos de esta suma son los valores s_b. Un árbol que clasifique de forma aleatoria $e_b = 0.5$ va a tener un peso $s_b = 0$ y cuando mejor clasifique el árbol mayor será su peso. Al estar utilizando clasificadores débiles (árboles pequeños) es de esperar que los pesos sean en general próximos a cero.

El siguiente hito fue la aparición del método *gradient boosting machine* (Friedman, 2001), perteneciente a la familia de los métodos iterativos de descenso de gradientes. Entre otras muchas ventajas, este método permitió resolver no sólo problemas de clasificación sino también de regresión; y permitió la conexión con lo que se estaba haciendo en otros campos próximos como pueden ser los modelos aditivos o la regresión logística. La idea es encontrar un modelo aditivo que minimice una función de perdida utilizando predictores débiles (por ejemplo árboles).

Si como función de pérdida se utiliza RSS, entonces la pérdida de utilizar $m(x)$ para predecir y en los datos de entrenamiento es

$$L(m) = \sum_{i=1}^{n} L(y_i, m(x_i)) = \sum_{i=1}^{n} (y_i - m(x_i))^2$$

.

Se desea minimizar $L(m)$ con respecto a m mediante el método de los gradientes, pero estos son precisamente los residuos: si $L(m) = \frac{1}{2}(y_i - m(x_i))^2$, entonces

$$-\frac{\partial L(y_i, m(x_i))}{\partial m(x_i)} = y_i - m(x_i) = r_i$$

Una ventaja de esta aproximación es que puede extenderse a otras funciones de pérdida, por ejemplo si hay valores atípicos se puede considerar como función de pérdida el error absoluto.

Veamos el algoritmo para un problema de regresión utilizando árboles de decisión. Es un proceso iterativo en el que lo que se *ataca* no son los datos directamente, sino los residuos (gradientes) que van quedando con los sucesivos ajustes, siguiendo una idea greedy (la optimización se resuelve en cada iteración, no globalmente).

1. Seleccionar el número de iteraciones B, el parámetro de regularización λ y el número de cortes de cada árbol d.

2. Establecer una predicción inicial constante y calcular los residuos de los datos i de la muestra de entrenamiento:
$$\hat{m}(x) = 0,\ r_i = y_i$$

3. Para $b = 1, 2, \ldots, B$, repetir:

   a. Ajustar un árbol de regresión $\hat{m}^b$ con d cortes utilizando los residuos como respuesta: (X, r).

   b. Calcular la versión regularizada del árbol:
$$\lambda \hat{m}^b(x)$$

c. Actualizar los residuos:

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

4. Calcular el modelo boosting:

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

Comprobamos que este método depende de 3 hiperparámetros, B, d y λ, susceptibles de ser seleccionados de forma *óptima*:

- B es el número de árboles. Un valor muy grande podría llegar a provocar un sobreajuste (algo que no ocurre ni con bagging ni con bosques aleatorios, ya que estos son métodos en los que se construyen árboles independientes). En cada iteración, el objetivo es ajustar de forma óptima el gradiente (en nuestro caso, los residuos), pero este enfoque greedy no garantiza el óptimo global y puede dar lugar a sobreajustes.
- Al ser necesario que el aprendizaje sea lento se utilizan árboles muy pequeños. Esto consigue que poco a poco se vayan cubriendo las zonas en las que es más difícil predecir bien. En muchas situaciones funciona bien utilizar $d = 1$, es decir, con un único corte. En este caso en cada $\hat{m}^b$ interviene una única variable, y por tanto $\hat{m}$ es un ajuste de un modelo aditivo. Si $d > 1$ se puede interpretar como un parámetro que mide el órden de interacción entre las variables.
- $0 < \lambda < 1$, parámetro de regularización. Las primeras versiones del algorimo utilizaban un $\lambda = 1$, pero no funcionaba bien del todo. Se mejoró mucho el rendimiento *ralentizando* aún más el aprendizaje al incorporar al modelo el parámetro λ, que se puede interpretar como una proporción de aprendizaje (la velocidad a la que aprende, *learning rate*). Valores pequeños de λ evitan el problema del sobreajuste, siendo habitual utilizar $\lambda = 0.01$ o $\lambda = 0.001$. Como ya se ha dicho, lo ideal es seleccionar su valor utilizando, por ejemplo, validación cruzada. Por supuesto, cuanto más pequeño sea el valor de λ, más lento va a ser el proceso de aprendizaje y serán necesarias más iteraciones, lo cual incrementa los tiempos de cómputo.

El propio Friedman propuso una mejora de su algoritmo, inspirado por la técnica bagging de Breiman. Esta variante, conocida como *stochastic gradient boosting*, es a día de hoy una de las más utilizadas. La única diferencia respecto al algoritmo anterior es en la primera línea dentro del bucle: al hacer el ajuste de (X, r), no se considera toda la muestra de entrenamiento, sino que se selecciona al azar un subconjunto. Esto incorpora un nuevo hiperparámetro a la metodología, la fracción que se utiliza de los datos. Lo ideal es seleccionar un valor por algún método automático (*tunearlo*) tipo validación cruzada; una selección manual típica es 0.5. Hay otras variantes, como por ejemplo la selección aleatoria de predictores antes de crecer cada árbol o antes de cada corte (ver por ejemplo la documentación de `[h2o::gbm](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html)`).

Este sería un ejemplo de un método con muchos hiperparámetros y diseñar una buena estrategia para ajustarlos (*tunearlos*) puede resultar mucho más complicado (puede haber problemas de mínimos locales, problemas computacionales, etc.).

Stochastic gradient boosting incorpora dos ventajas importantes: reduce la varianza y reduce los tiempos de cómputo. En terminos de rendimiento tanto el método *stochastic gradient boosting* como *random forest* son muy competitivos, y por tanto son muy utilizando en la práctica. Los bosques aleatorios tienen la ventaja de que, al construir árboles de forma independiente, es paralelizable y eso puede reducir los tiempos de cómputo.

Otro método reciente que está ganando popularidad es *extreme gradient boosting*, también conocido como *XGBoost* (Chen y Guestrin, 2016). Es un metodo más complejo que el anterior que, entre otras modificaciones, utiliza una función de pérdida con una penalización por complejidad y, para evitar el sobreajuste, regulariza utilizando la hessiana de la función de pérdida (necesita calcular las derivadas parciales de primer y de segundo orden), e incorpora parámetros de regularización adicionales para evitar el sobreajuste.

Por último, la importancia de las variables se puede medir de forma similar a lo que ya hemos visto en otros métodos: dentro de cada árbol se sumas las reducciones del error que consigue cada predictor,

y se promedia entre todos los árboles utilizados.

En resumen:

- La idea es hacer un "aprendizaje lento".
- Los arboles se crecen de forma secuencial, se trata de mejorar la clasificación anterior.
- Se utilizan arboles pequeños.
- A diferencia de bagging y bosques aleatorios puede haber problemas de sobreajuste (si el número de árboles es grande y la tasa de aprendizaje es alta).
- Se puede pensar que se ponderan las observaciones iterativamente, se asigna más peso a las que resultaron más difíciles de clasificar.
- El modelo final es un modelo aditivo (media ponderada de los árboles).

3.5 Boosting en R

En preparación...

ada, adabag, mboost, gbm, xgboost

Referencias

Bibliografía básica

James, G., Witten, D., Hastie, T. y Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Aplications in R*. Springer.

Kuhn, M. y Johnson, K. (2013). *Applied predictive modeling*. Springer.

Williams, G. (2011). *Data Mining with Rattle and R*. Springer.

Bibliografía complementaria

Libros

Bellman, R.E. (1961). *Adaptive Control Processes*, Princeton University Press.

Burger, S.V. (2018). *Introduction to machine learning with R: Rigorous mathematical analysis*. O'Reilly.

Breiman, L., Friedman, J., Stone, C.J. y Olshen, R.A. (1984). *Classification and regression trees*. CRC press.

Efron, B. y Hastie, T. (2016). *Computer age statistical inference*. Cambridge University Press.

Fernández-Casal, R. y Cao, R. (2020). *Simulación Estadística*. https://rubenfcasal.github.io/simbook.

Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Hastie, T., Tibshirani, R. y Wainwright, M. (2015). *Statistical learning with sparsity: the lasso and generalizations*. CRC press.

Irizarry, R.A. (2019). *Introduction to Data Science: Data Analysis and Prediction Algorithms with R*. CRC Press.

Molnar, C. (2020). Interpretable Machine Learning. A Guide for Making Black Box Models Explainable. Lulu.com.

Torgo, L. (2011). *Data Mining with R: Learning with Case Studies*. Chapman & Hall/CRC Press.

Artículos

Agor, J. y Özaltın, O.Y. (2019). Feature selection for classification models via bilevel optimization. *Computers & Operations Research*, 106, 156-168.

Biecek, P. (2018). DALEX: explainers for complex predictive models in R. *The Journal of Machine Learning Research*, 19(1), 3245-3249.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2), 123-140.

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

Breiman, L. (2001). Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author). *Statistical Science*, 16, 199-231.

Dietterich, T.G. (2000). An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. *Machine Learning*, 40(2), 139–158.

Dunson D.B. (2018). Statistics in the big data era: Failures of the machine. *Statistics and Probability Letters*, 136, 4-9.

Fisher, R.A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2), 179-188.

Freund, Y. y Schapire, R. (1996). Experiments with a New Boosting Algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference*, 148–156.

Friedman, J.H. (2001). Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29, 1189–1232.

Friedman, J.H. y Popescu, B.E. (2008). Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3), 916-954. Goldstein, A., Kapelner, A., Bleich, J. y Pitkin, E. (2015). Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1), 44-65.

Greenwell, B.M. (2017). pdp: An R Package for Constructing Partial Dependence Plots. *The R Journal*, 9(1), 421-436.

Kearns, M. y Valiant, L. (1989). Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing.*

Kuhn, M. (2008). Building predictive models in R using the caret package. *Journal of Statistical Software*, 28(5), 1-26.

Lauro, C. (1996). Computational statistics or statistical computing, is that the question?, *Computational Statistics & Data Analysis*, 23 (1), 191-193.

Liaw, A. y Wiener, M. (2002). Classification and regression by randomForest. *R News*, 2(3), 18-22.

Loh, W.Y. (2002). Regression tress with unbiased variable selection and interaction detection. *Statistica Sinica*, 361-386.

Shannon C (1948). A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27, 379–423.

Strumbelj, E. y Kononenko, I. (2010). An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*, 11, 1-18.

Valiant, L. (1984). A Theory of the Learnable. *Communications of the ACM*, 27, 1134–1142.