Aprendizaje Estadístico

Rubén Fernández Casal (ruben.fcasal@udc.es)
Julián Costa Bouzas (julian.costa@udc.es)
Manuel Oviedo de la Fuente (manuel.oviedo@udc.es)

Edición: Septiembre de 2021. Impresión: 2024-09-09

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.

Este libro tiene asociado el paquete de R `mpae` (*Métodos Predictivos de Aprendizaje Estadístico*, Fernandez-Casal et al., 2024), que incluye funciones y conjuntos de datos utilizados a lo largo del texto. Este paquete está disponible en CRAN y puede instalarse ejecutando el siguiente código[1]:

```
install.packages("mpae")
```

Sin embargo, para poder ejecutar todos los ejemplos mostrados en el libro, es necesario instalar también los siguientes paquetes: `caret`, `gbm`, `car`, `leaps`, `MASS`, `RcmdrMisc`, `lmtest`, `glmnet`, `mgcv`, `np`, `NeuralNetTools`, `pdp`, `vivid`, `plot3D`, `AppliedPredictiveModeling`, `ISLR`. Para ello, en lugar del código anterior, bastaría con ejecutar:

```
install.packages("mpae", dependencies = TRUE)
```

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.

[1] Alternativamente, se puede instalar la versión en desarrollo disponible en el repositorio rubenfcasal/mpae de GitHub. Por ejemplo, el comando `remotes::install_github("rubenfcasal/mpae", INSTALL_opts = "--with-keep.source")` instala el paquete incluyendo los comentarios en el código y opcionalmente las dependencias.

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.

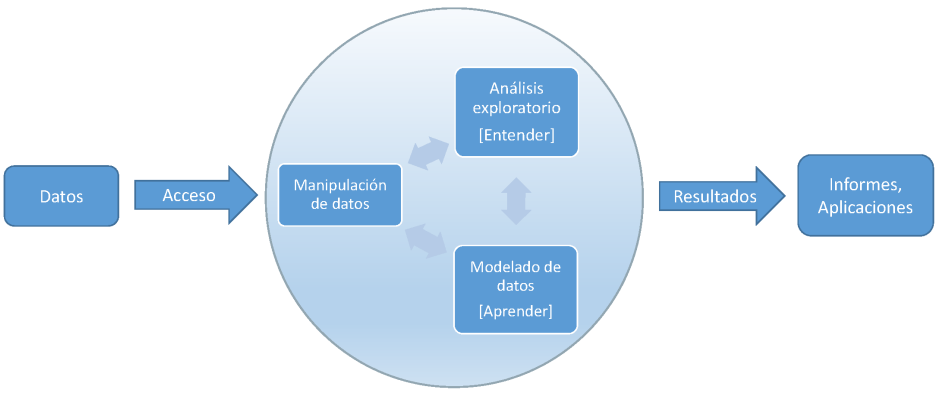


Figura 1.1: Etapas en el análisis de datos.

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. Esto es lo que se conoce como *aleatoriedad aparente*:

> "Nothing in Nature is random… a thing appears random only through the incompleteness of our knowledge."
>
> — Spinoza, Baruch (Ethics, 1677)

Aunque hoy en día gana peso la idea de la física cuántica de que en el fondo hay una *aleatoriedad intrínseca*:

> "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/repetibilidad.
- 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

[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".

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 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, especialmente de selección[2]). 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, 2001b) 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) también expone las diferencias entre ambas culturas, por ejemplo en investigación (la forma en que evolucionan):

- "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,

[2]También es importante detectar la presencia de algún tipo de error de medición, al menos como primer paso para tratar de predecir la respuesta libre de ruido.

> intellectual as opposed to primarily performance driven, emphasizing theoretical support (e.g., through asymptotic properties), under-stated, and conservative".

también en los principales campos de aplicación y en el tipo de datos que manejan:

- "*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".

En las conclusiones, además de alertar de los peligros:

- "Big data that are subject to substantial selection bias and measurement errors, without information in the data about the magnitude,sources and types of errors, should not be used to inform important decisions without substantial care and skepticism".
- "There is vast interest in automated methods for complex data analysis. However, there is a lack of consideration of (1) interpretability, (2) uncertainty quantification, (3) applications with limited training data, and (4) selection bias. Statistical methods can achieve (1)-(4) with a change in focus" (Resumen del artículo).

destaca la importancia de tener en cuenta el punto de vista estadístico.

> "Such developments will likely require a close collaboration between the Stats and ML-communities and mindsets. The emerging field of data science provides a key opportunity to forge a new approach for analyzing and interpreting large and complex data merging multiple fields."
>
> — Dunson, D.B. (2018).

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 $\mathbf{X} = (X_1, X_2, \dots, X_p)$ 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[3]. 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:

$$\left\{\left(x_{1i}, \dots, x_{pi}, y_i\right) : i = 1, \dots, n\right\},$$

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

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 de diversas formas, por ejemplo, asumiendo que la distribución del error depende de $\mathbf{X}$ (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[4].

Como ya se comentó se podría considerar clasificación como un caso particular, por ejemplo definiendo $Y(\mathbf{X})$ de forma que tome los valores $1, 2, \dots, K$, 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, $i = 1, \dots, n$, 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*)[5]. 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, lo que se conoce como la *regla de Bayes*, 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)[6]:

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

de forma que $\hat{p}(\mathbf{x}) = \text{sigmoid}(\hat{Y}(\mathbf{x}))$. Reciprocamente, empleando su inversa, la *función logit*:

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

se pueden transformar las probabilidades a la escala de puntuaciones.

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

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

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

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

Lo anterior se puede generalizar para el caso de múltiples categorías, considerando variables indicadoras de cada categoría $Y_1, \ldots, Y_K$ (para cada caso se agrupan las demás como una sola), lo que se conoce como la estrategia de "uno contra todos" (*One-vs-Rest*, OVR). En este caso típicamente:

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

Otra posible estrategia es la denominada "uno contra uno" (*One-vs-One*, OVO) o también conocido por "votación mayoritaria" (*majority voting*), que requiere entrenar un clasificador para cada par de categorías (se consideran $K(K-1)/2$ subproblemas de clasificación binaria). En este caso se suele seleccionar como predicción la categoría que recibe más votos (la que resultó seleccionada por el mayor número de los clasificadores binarios).

Otros métodos (como por ejemplo los árboles de decisión, que se tratarán en el Tema 2) permiten la estimación directa de las probabilidades de cada clase.

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:

- 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`…
- Regresión spline adaptativa multivariante (MARS): `earth`
- Regresión por *projection pursuit* (incluyendo *single index model*): `caret::ppr()`, `np::npindex()`…
- Redes neuronales: `nnet`, `neuralnet`…

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` (Kuhn, 2023; ver también Kuhn y Johnson, 2013) 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`), `rattle` (Williams, 2022; ver también Williams, 2011) y `radiant`.

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), jackknife 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) puede tener 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))
```

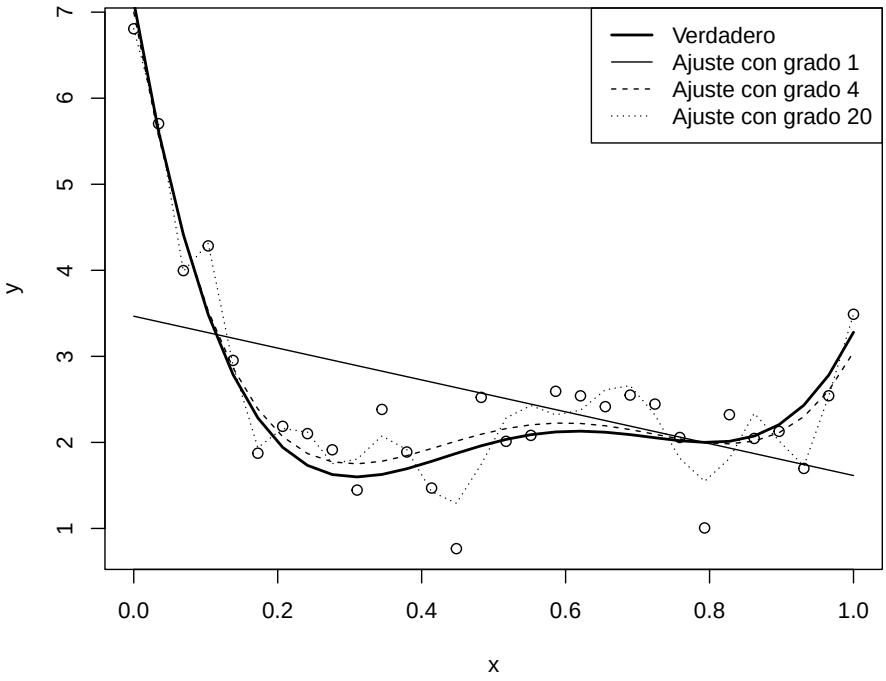


Figura 1.2: Muestra (simulada) y ajustes polinómicos con distinta complejidad.

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 (ver Tabla 1.1).

```
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)))
```

Tabla 1.1: Medidas de bondad de ajuste de los modelos polinómicos.

	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 (ver Figura 1.3):

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

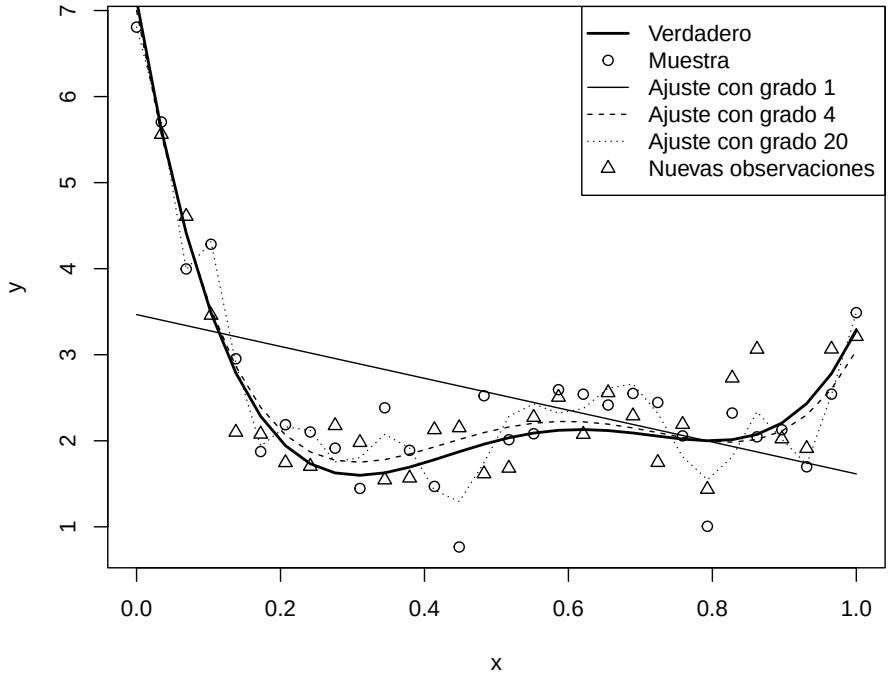


Figura 1.3: Muestra con ajustes polinómicos con distinta complejidad y nuevas observaciones.

```
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 evalúa 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))
    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))
```

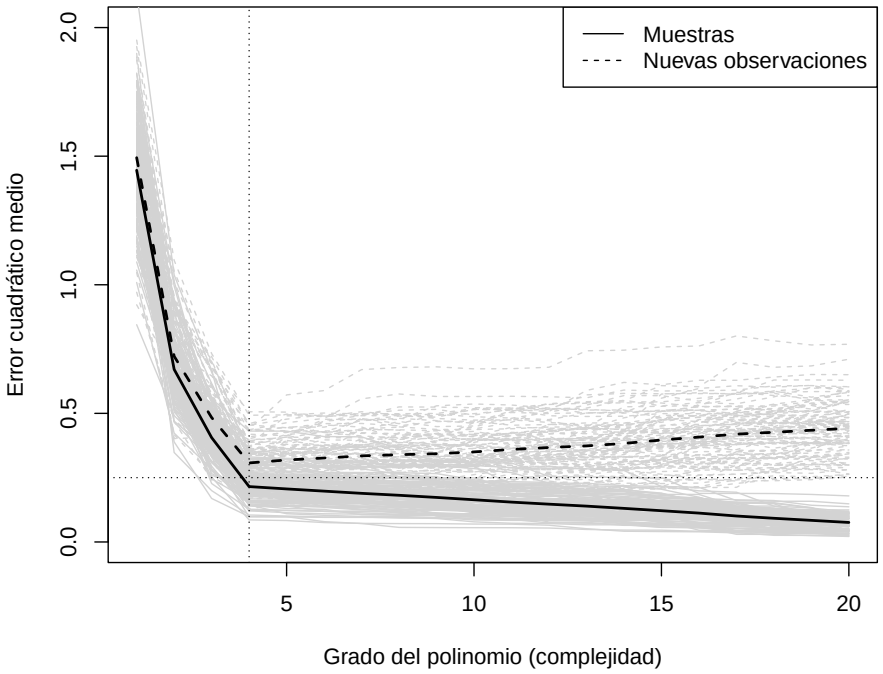


Figura 1.4: Precisiones (errores cuadráticos medios) de ajustes polinómicos variando la complejidad, en las muestras empleadas en el ajuste y en nuevas observaciones (simulados).

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, el predictor óptimo (desconocido) sería la media condicional $m(\mathbf{x}) = E\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$[7]. Por tanto los

[7]Se podrían considerar otras funciones de pérdida, por ejemplo con la distancia L_1 sería la mediana condicional, pero

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 (ver Figura 1.5).

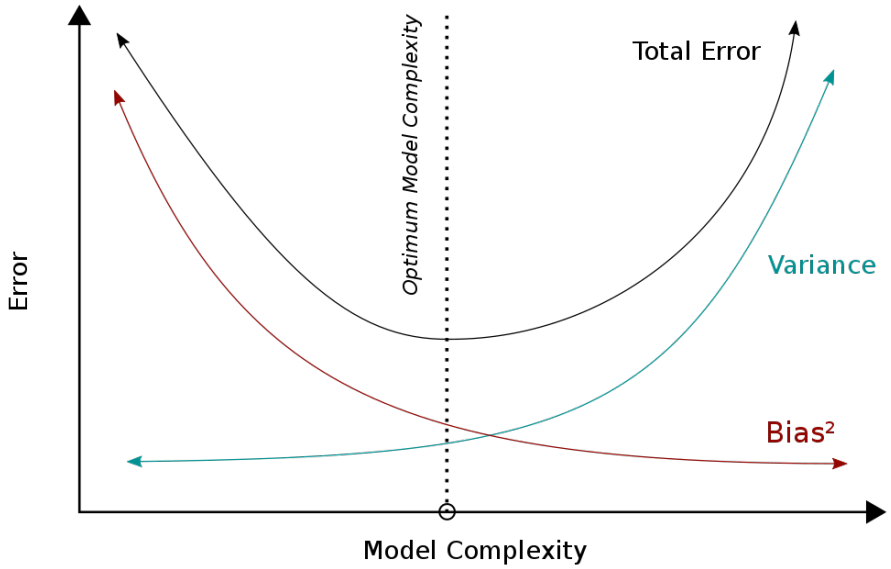


Figura 1.5: Equilibrio entre sesgo y varianza.

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). 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 observaciones 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 errores observados en esta muestra servirán para aproximar lo que ocurriría con nuevas observaciones).

Típicamente se selecciona al azar el 80% de los datos como muestra de entrenamiento y el 20% restante como muestra de test, aunque esto dependería del número de datos (los resultados serán aleatorios, aunque su variabilidad dependerá principalmente del tamaño de las muestras). En R se puede realizar

las consideraciones serían análogas.

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).

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 construir 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, ]
```

Los datos de test deberían utilizarse únicamente para evaluar los modelos finales, no se deberían emplear para seleccionar hiperparámetros. Para seleccionarlos 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.

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)[8]. 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 a partir del "estatus" de los residentes (ver Figura 1.6). 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) {
    respuesta <- as.character(formula)[2] # extraer nombre variable respuesta
    n <- nrow(datos)
```

[8] La partición en k-fold CV se suele realizar al azar. Hay que tener en cuenta la aleatoriedad al emplear k-fold CV, algo que no ocurre con LOOCV.

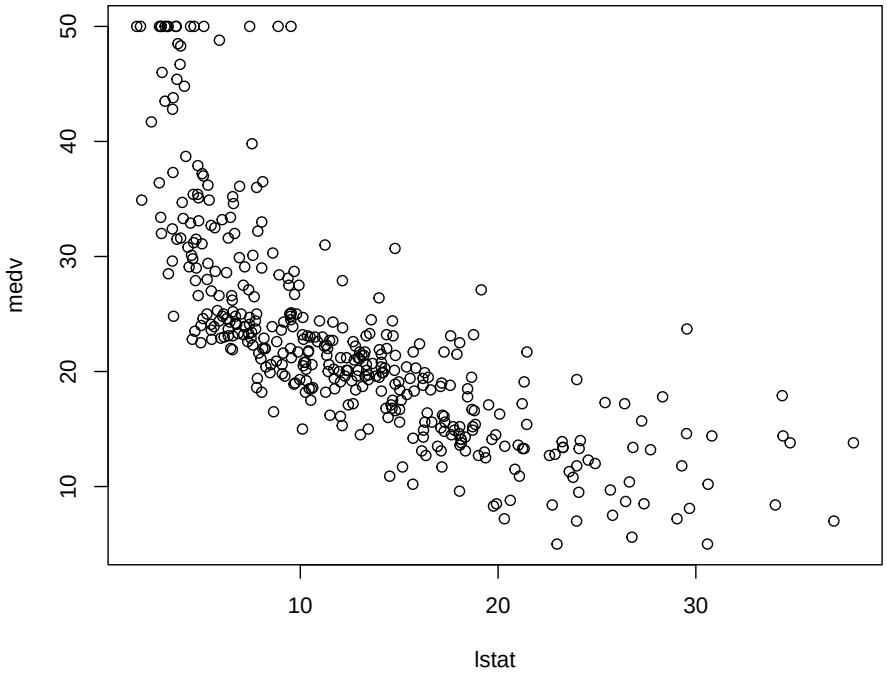


Figura 1.6: Gráfico de dispersión de las valoraciones de las viviendas (`medv`) frente al porcentaje de población con "menor estatus" (`lstat`).

```
    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, respuesta]
    }
    return(cv.res)
}
```

La función anterior no es muy eficiente, pero podría modificarse fácilmente para emplear otros métodos de regresión[9]. En el caso de regresión lineal múltiple (y de otros predictores 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 cruzada (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){
  # Tiempo de computación elevado!
  # cv.res <- cv.lm0(medv ~ poly(lstat, grado), train)
  cv.res <- cv.lm(medv ~ poly(lstat, grado), train)
  se <- cv.res^2
```

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

```
  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")
# Valor óptimo
imin.mse <- which.min(cv.mse)
grado.min <- grados[imin.mse]
points(grado.min, cv.mse[imin.mse], pch = 16)
```

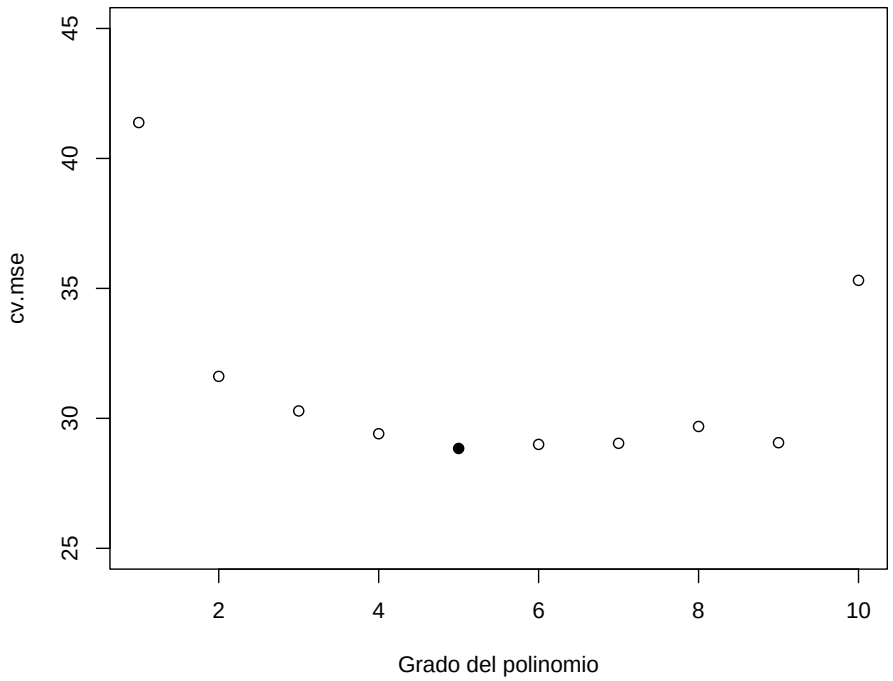


Figura 1.7: Error cuadrático medio de validación cruzada dependiendo del grado del polinomio (complejidad) y valor óptimo.

```
grado.min
```

```
  ## [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 (si cambiamos la muestra o cambiamos la partición los resultados seguramente cambiarán), por lo que la sugerencia es seleccionar el modelo más simple[10] 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).

```
plot(grados, cv.mse, ylim = c(25, 45),
  xlab = "Grado del polinomio")
segments(grados, cv.mse - cv.mse.sd, grados, cv.mse + cv.mse.sd)
# Límite superior "oneSE rule"
upper.cv.mse <- cv.mse[imin.mse] + cv.mse.sd[imin.mse]
abline(h = upper.cv.mse, lty = 2)
# Complejidad mínima por debajo del límite
imin.1se <- min(which(cv.mse <= upper.cv.mse))
grado.1se <- grados[imin.1se]
```

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

```
points(grado.1se, cv.mse[imin.1se], pch = 16)
```

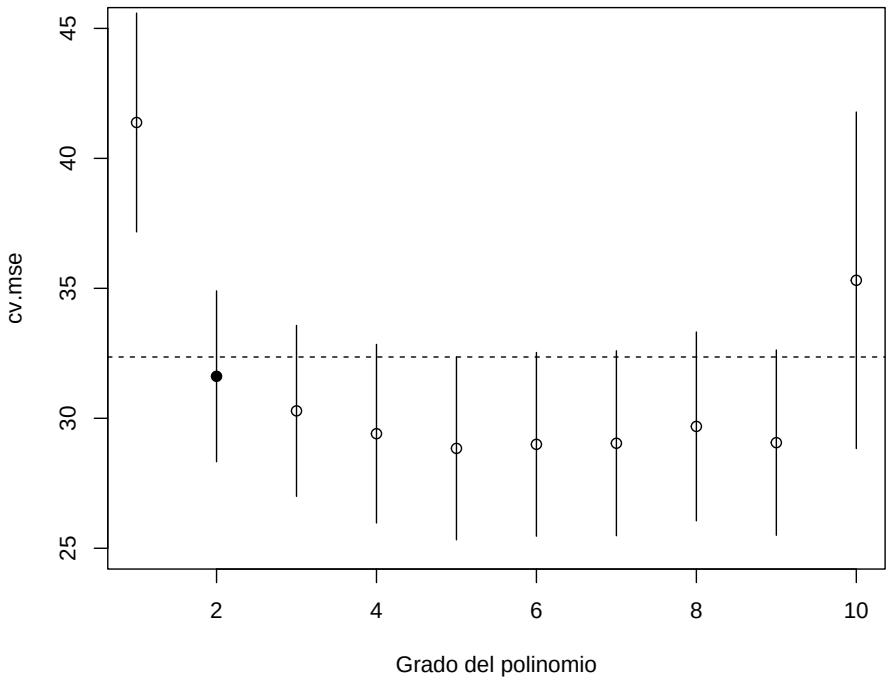


Figura 1.8: Error cuadrático medio de validación cruzada dependiendo del grado del polinomio (complejidad) y valor seleccionado con el criterio de un error estándar.

```
grado.1se
```

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

```
plot(medv ~ lstat, data = train)
fit.min <- lm(medv ~ poly(lstat, grado.min), train)
fit.1se <- lm(medv ~ poly(lstat, grado.1se), train)
newdata <- data.frame(lstat = seq(0, 40, len = 100))
lines(newdata$lstat, predict(fit.min, newdata = newdata))
lines(newdata$lstat, predict(fit.1se, newdata = newdata), lty = 2)
legend("topright", legend = c(paste("Grado óptimo:", grado.min),
       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. Los resultados servirán como medidas globales de la calidad de las predicciones con nuevas observaciones.

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

Si generamos un gráfico de dispersión de observaciones frente a predicciones[11], los puntos deberían estar en torno a la recta $y = x$ (ver Figura 1.10).

```
plot(pred, obs, xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
res <- lm(obs ~ pred)
```

[11] Otras implementaciones, como la función `caret::plotObsVsPred()`, intercambian los ejes, generando un gráfico de dispersión de predicciones sobre observaciones.

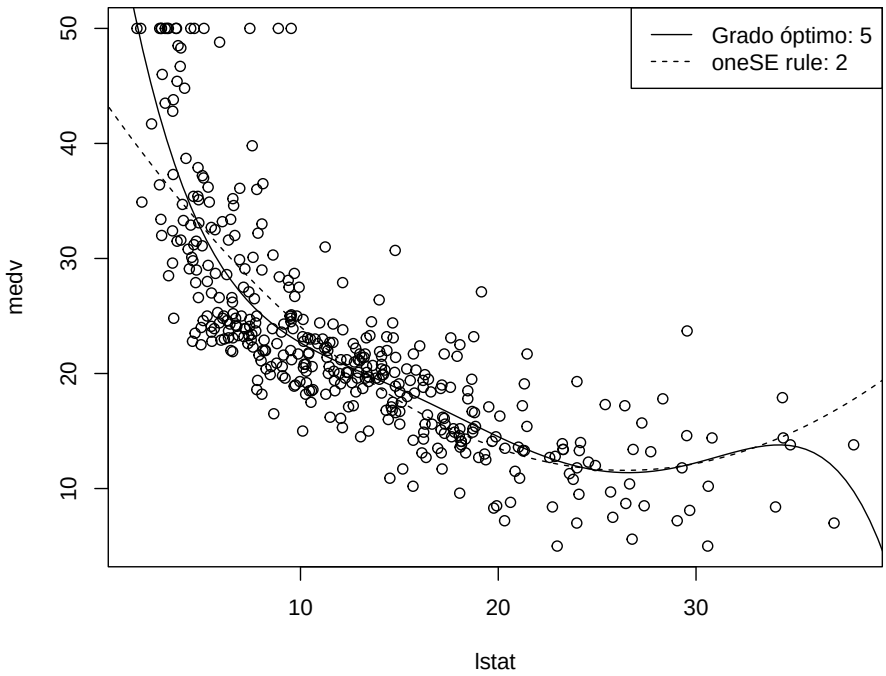


Figura 1.9: Ajuste de los modelos finales, empleando el valor óptimo y el criterio de un error estándar para seleccionar el grado del polinomio mediante validación cruzada.

```
# summary(res)
abline(res, lty = 2)
```

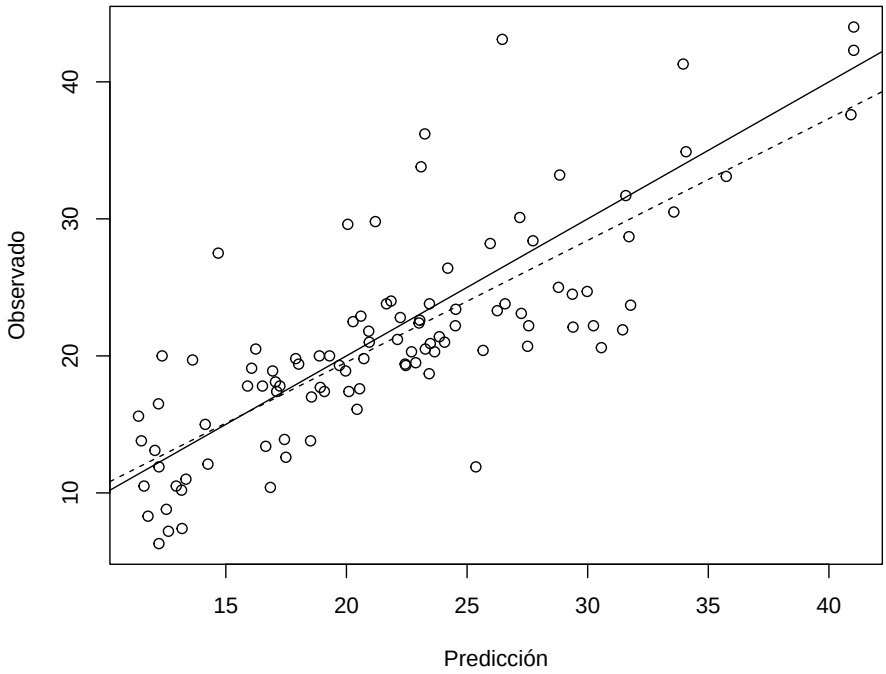


Figura 1.10: Gráfico de dispersión de observaciones frente a predicciones (incluyendo la identidad, línea continua, y el ajuste lineal, línea discontinua).

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 (en nuevas observaciones) explicada por el modelo, algo que en general no es cierto con el anterior[12]. La recomendación sería emplear:

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

(que sería equivalente al coeficiente de determinación ajustado en regresión múltiple pero sin depender de hipótesis estructurales del modelo) 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(
    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
  ))
}
accu.min <- accuracy(pred, obs)
accu.min
```

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

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

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

En este caso concreto (con la semilla establecida anteriormente), estimaríamos que el ajuste polinómico con el grado óptimo (seleccionado minimizando el error cuadrático medio de validación cruzada) explicaría un 60.9% de la variabilidad de la respuesta en nuevas observaciones (un 7.2% más que el modelo seleccionado con el criterio de un error estándar de Breiman).

Ejercicio 1.1

Considerando de nuevo el ejemplo anterior, particionar la muestra en datos de entrenamiento (70%),

[12] Por ejemplo obtendríamos el mismo valor si desplazamos las predicciones sumando una constante (i.e. no tiene en cuenta el sesgo). Lo que interesaría sería medir la proximidad de los puntos a la recta $y = x$.

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[13]:

```
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, 1)
```

```
## List of 3
##  $ train   :'data.frame':  356 obs. of  14 variables:
##  $ validate:'data.frame':  75 obs. of  14 variables:
##  $ test    :'data.frame':  75 obs. of  14 variables:
```

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
Positivo	Verdaderos positivos (TP)	Falsos negativos (FN)
Negativo	Falsos positivos (FP)	Verdaderos negativos (TN)

A partir de esta tabla se pueden obtener distintas medidas de la precisión de las predicciones (serían medidas globales de la calidad de la predicción de nuevas observaciones). 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):

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

[13] Versión "simplificada" (más eficiente computacionalmente) de una propuesta en el post https://stackoverflow.com/questions/36068963). En el caso de que la longitud del factor `f` no coincida con el número de filas (por redondeo), se generaría un *warning* (suprimido) y se reciclaría.

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,              # levels = c('FALSE', 'TRUE')
                      labels = c("Bajo", "Alto"))
# 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 la Sección 6.9 se darán más detalles sobre este tipo de modelos):

```
# Particionado de los datos
set.seed(1)
```

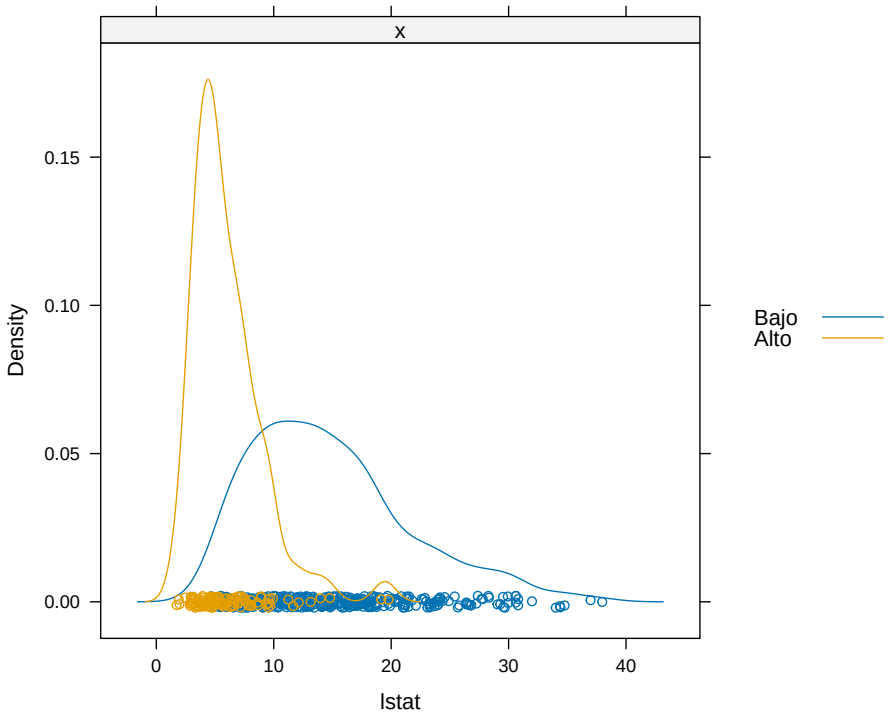


Figura 1.11: Distribución del estatus de la población dependiendo del nivel de valoración de las viviendas.

```
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)
##
## 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"))
```

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 análisis 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")
```

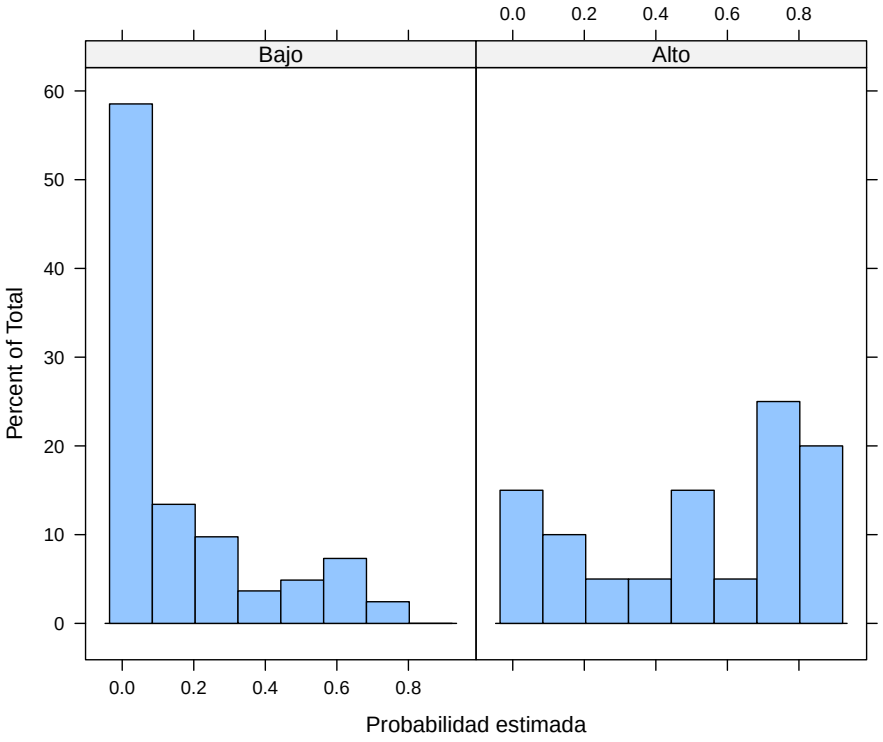


Figura 1.12: Distribución de las probabilidades estimadas de valoración alta de la vivienda dependiendo de la categoría observada.

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 (*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)
plot(roc_glm)
```

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.

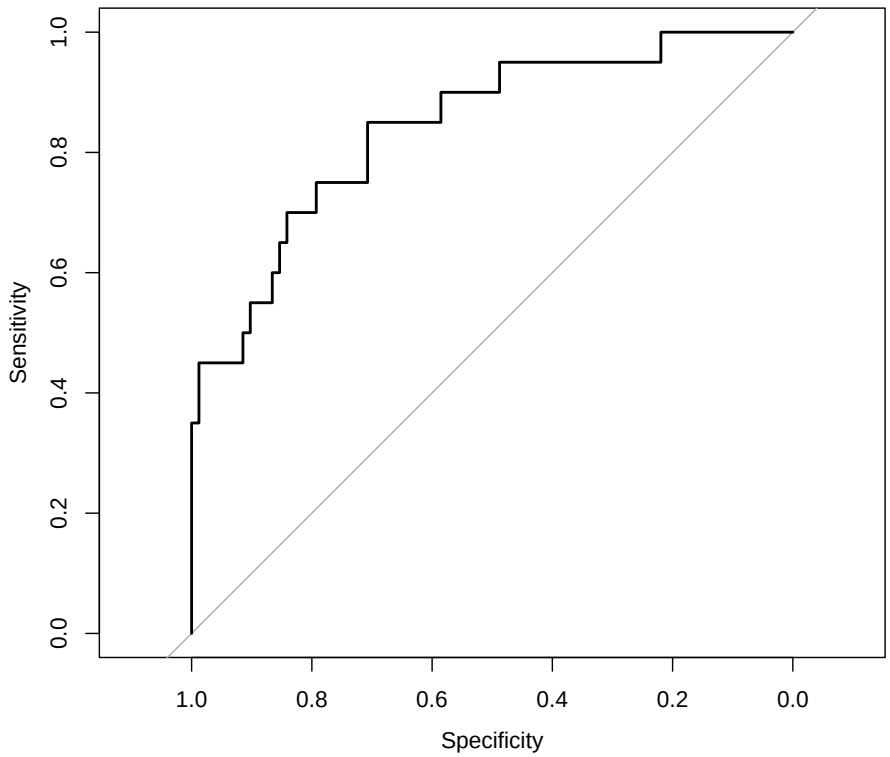


Figura 1.13: Curva ROC correspondiente al modelo de regresión logística.

```
# 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$Species)
f <- factor(levels, levels = levels)
# Nota: Al estar por orden alfabético se podría haber empleado factor(levels)
pred.rand <- sample(f, nrow(test), replace = TRUE, prob = prevalences)
# Evaluación
caret::confusionMatrix(pred.rand, test$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).

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[14] KNN promedia las observaciones 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). Este método está implementado en numerosos paquetes, por ejemplo en la función `knnreg()` del paquete `caret`.

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):

```
# 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)
# Representar
plot(x, y)
lines(x, mu, lwd = 2, col = "lightgray")
# Ajuste de los modelos
library(caret)
# k = número de observaciones más cercanas
fit1 <- knnreg(y ~ x, data = datos, k = 5) # 5% de los datos (n = 100)
fit2 <- knnreg(y ~ x, data = datos, k = 10)
fit3 <- knnreg(y ~ x, data = datos, k = 20)
# Añadir predicciones y leyenda
newdata <- data.frame(x = x)
lines(x, predict(fit1, newdata), lwd = 2, lty = 3)
lines(x, predict(fit2, newdata), lwd = 2, lty = 2)
```

[14] 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.

```
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))
```

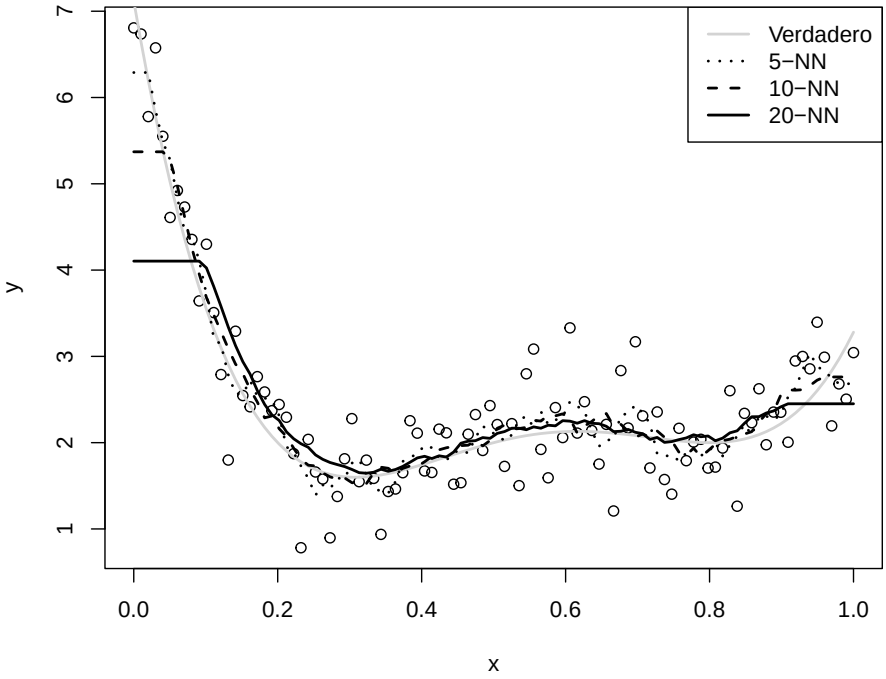


Figura 1.14: Predicciones con el método KNN y distintos vecindarios.

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 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.

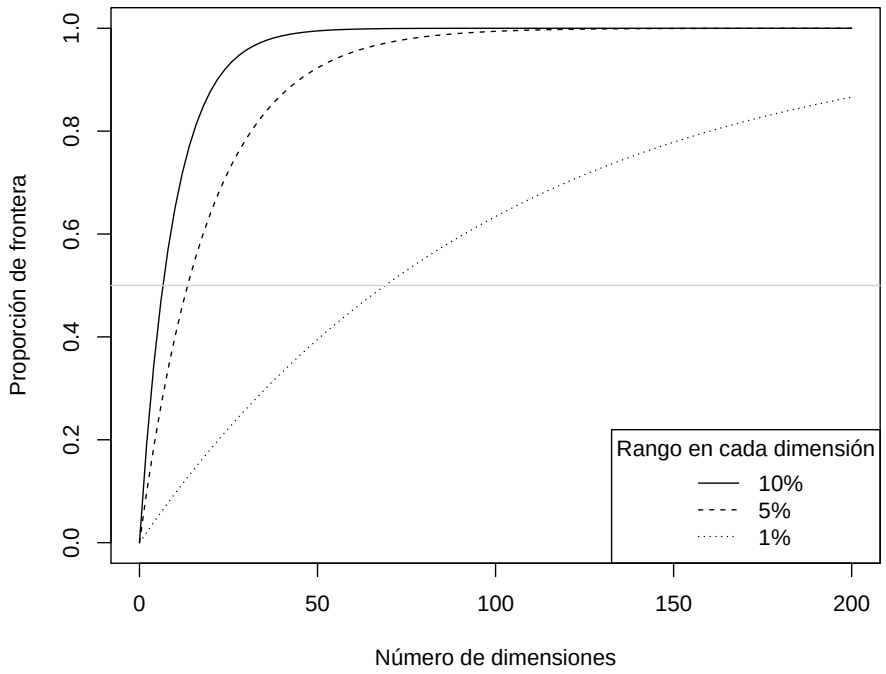


Figura 1.15: Proporción de "frontera" dependiendo del número de dimensiones y del porcentaje de valores considerados extremos en cada dimensión.

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*, ver Goldstein et al. (2015). Estos métodos tratan de estimar el efecto marginal de las variables explicativas. En ese sentido son similares a los gráficos parciales de residuos (habitualmente empleados en los modelos lineales o aditivos; ver p.e. las funciones `termplot()`, `car::crPlots()` o `car::avPlots()`, Sección 6.4, y `mgcv::plot.gam()`, Sección 7.3), que muestran la variación en la predicción a medida que varía una variable explicativa manteniendo constantes el resto (algo que tiene sentido si asumimos que los predictores son independientes), pero en este caso se admite que el resto de predictores también pueden variar.

En el caso de los gráficos PDP se tiene en cuenta el efecto marginal de los demás predictores del modelo. Suponiendo que estamos interesados en un conjunto $\mathbf{X}^S$ de predictores, de forma que $\mathbf{X} = [\mathbf{X}^S, \mathbf{X}^C]$ y $f_{\mathbf{X}^C}(\mathbf{x}^C) = \int f(\mathbf{x}) d\mathbf{x}^S$ es la densidad marginal de $\mathbf{X}^C$, se trata de aproximar:

$$\hat{Y}_S(\mathbf{x}^S) = E_{\mathbf{X}^C}\left[\hat{Y}(\mathbf{x}^S, \mathbf{X}^C)\right] = \int \hat{Y}(\mathbf{x}^S, \mathbf{x}^C) f_{\mathbf{X}^C}(\mathbf{x}^C) d\mathbf{x}^C$$

mediante:

$$\hat{y}_{\mathbf{x}^S}(\mathbf{x}^S) = \frac{1}{n}\sum_{i=1}^{n} \hat{y}(\mathbf{x}^S, \mathbf{x}_i^C)$$

donde n en el tamaño de la muestra de entrenamiento y $\mathbf{x}_i^C$ son los valores observados de las variables explicativas en las que no estamos interesados. La principal diferencia con los gráficos ICE es que, en lugar de mostrar una única curva promedio de la respuesta, estos muestran una curva para cada observación (para más detalles ver las referencias anteriores). En la Sección 3.3.2 se incluyen algunos ejemplos.

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 (ver p.e. `iml::Shapley()`), 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
- `vivid`: Variable Importance and Variable Interaction Displays
- `ICEbox` ICEbox: Individual Conditional Expectation Plot Toolbox.
- `plotmo`: Plot a Model's Residuals, Response, and Partial Dependence Plots.

- `randomForestExplainer`: Explaining and Visualizing Random Forests in Terms of Variable Importance.

En este caso también puede ser de utilidad `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` (*Classification And REgression Training*, Kuhn, 2008) proporciona una interfaz unificada que simplifica el proceso de modelado empleando la mayoría de los métodos de AE implementados en R (actualmente admite 239 métodos; ver el Capítulo 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.

En esta sección se describirán de forma esquemática las principales herramientas disponibles en este paquete, para más detalles se recomendaría consultar el manual del paquete caret. También está disponible una pequeña introducción en la *vignette* del paquete: A Short Introduction to the caret Package y una "chuleta": Caret Cheat Sheet.

1.6.1 Métodos implementados

La función principal es `train()` (descrita en la siguiente subsección), 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 los métodos disponibles
```

```
##  chr [1:239] "ada" "AdaBag" "AdaBoost.M1" "adaboost" ...
```

```
# getModelInfo() devuelve coincidencias parciales por defecto
# names(getModelInfo("knn")) # 2 métodos
modelLookup("knn")  # Información sobre hiperparámetros
```

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

En la versión online del libro se incluye una tabla dinámica con los métodos actualmente disponibles.

1.6.2 Herramientas

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()`). Estimará los parámetros de las transformaciones con la muestra de entrenamiento y permitirá aplicarlas posteriormente de forma automática 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"`
    - Selección de predictores y extracción de componentes: `"corr"`, `"nzv"`, `"zv"`, `"conditionalX"`, `"pca"`, `"ica"`
  - Dispone de múltiples funciones auxiliares, como `dummyVars()` o `rfe()` (*recursive feature elimination*).
- 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[15] `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()`…
- Análisis de la importancia de los predictores:
  - `varImp()`: interfaz a las medidas específicas de los métodos de aprendizaje supervisado (Sección 15.1 del manual) o medidas genéricas (Sección 15.2).

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

1.6.3 Ejemplo

Como ejemplo consideraremos el problema de regresión anterior empleando KNN en caret:

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

Particionamos los datos:

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

Entrenamiento, con preprocesado de los datos (se almacenan las transformaciones para volver a aplicarlas en la predicción con nuevos datos) y empleando validación cruzada con 10 grupos para la 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) # Alternativamente: ggplot(knn, highlight = TRUE)
```

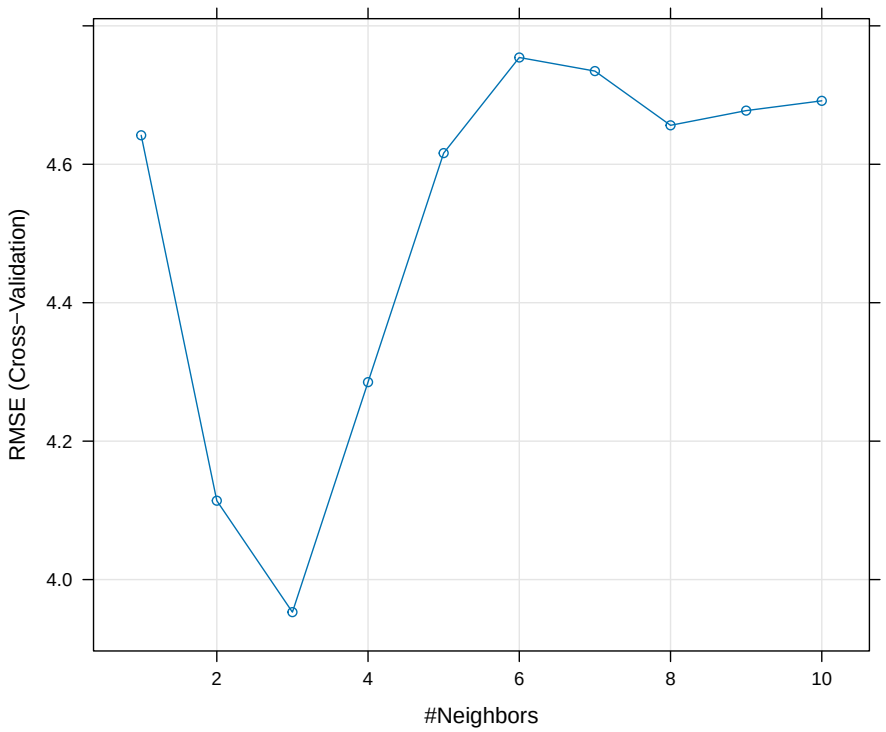


Figura 1.16: Raíz del error cuadrático medio de validación cruzada dependiendo del valor del hiperparámetro.

```
knn$bestTune
```

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

```
knn$finalModel
```

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

Importancia de las variables (interpretación del modelo final):

```
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 del modelo final en la muestra de test:

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

```
##     RMSE Rsquared      MAE
## 4.960971 0.733945 2.724242
```

1.6.4 Desarrollo futuro

Como comenta el autor del paquete `caret` (y coautor en Kuhn y Johnson, 2013):

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

este paquete ha dejado de desarrollarse de forma activa, aunque consideramos que la alternativa `tidymodels` (Kuhn y Wickham, 2023) todavía está en fase de desarrollo[16] y su uso requiere de más tiempo de aprendizaje. Este es uno de los motivos por los que se ha optado por mantener el uso de `caret` en este libro, aunque la intención es incluir apéndices adicionales en próximas ediciones ilustrando el uso de otras herramientas (como `tidymodels`, ver Kuhn y Silge, 2022; o incluso `mlr3`, Becker et al., 2021).

[16] Sin embargo, desde la publicación del libro Kuhn y Silge (2022), disponible en línea en https://www.tmwr.org, ya podríamos considerar que ha superado la fase inicial de desarrollo.

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. Como se muestra en la Figura 2.1 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 (ver Figura 2.2). 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.

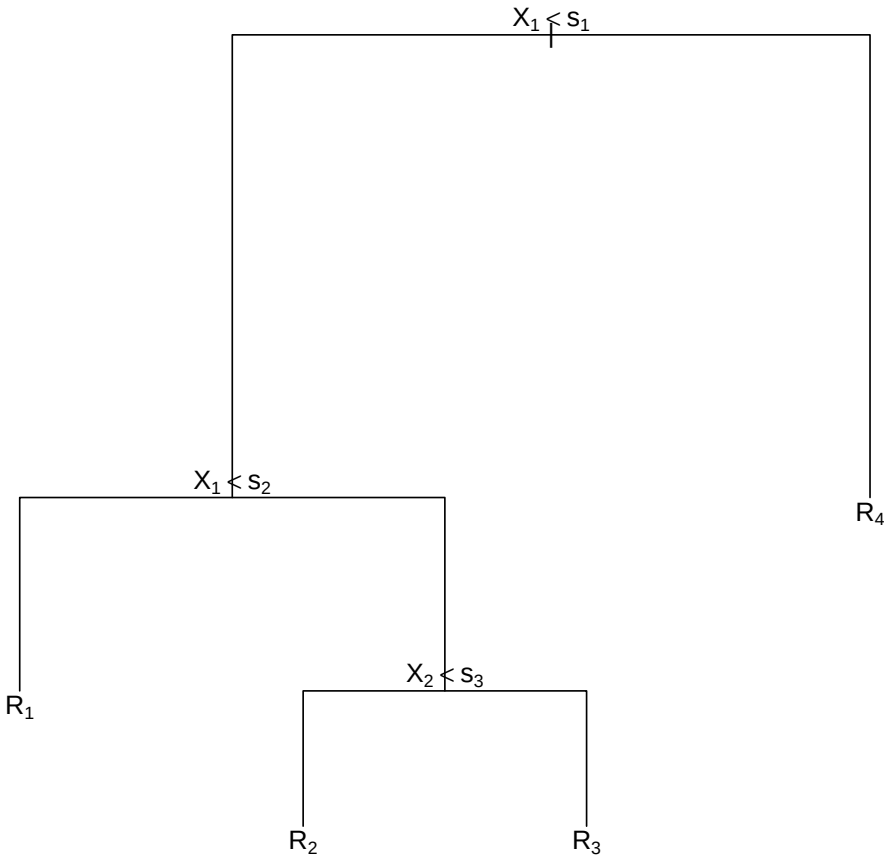


Figura 2.1: Ejemplo de un árbol de decisión obtenido al realizar una partición binaria recursiva de un espacio bidimensional.

- 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 *árboles 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 $\mathbf{X} = (X_1, X_2, \dots, X_p)$ 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 $R_1, R_2, \dots, R_J$, 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.

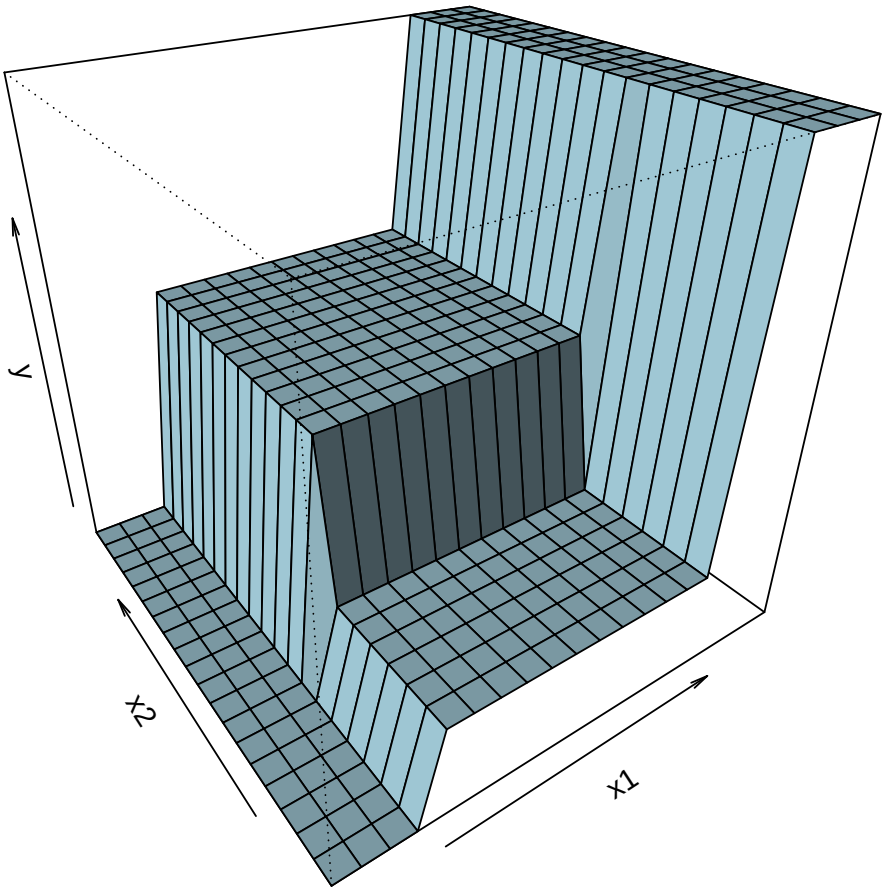


Figura 2.2: Ejemplo de la superficie de predicción correspondiente a un árbol de decisión.

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 $\hat{y}_{R_j}$. Por tanto, se deben seleccionar las regiones $R_1, R_2, \ldots, R_J$ que minimicen

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

(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

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

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 *óptimo* (para los datos de los que disponemos, claro). Veamos la idea.

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

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

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 la proporción de variabilidad residual y a partir de ella podríamos definir:

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

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 $\hat{p}_k$, 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:

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

- Índice de Gini:

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

- Entropía[1] (*cross-entropy*):

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

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 $max_k(\hat{p}_k)$, mientras que en las medidas alternativas intervienen las proporciones $\hat{p}_k$ 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:

[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).

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

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) como puede observarse en el gráfico de barras de la Figura 2.3.

```
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 0.2..
##  $ citric.acid         : num  0.47 0.34 0.13 0.49 0.26 0.63 0.07 0.36 0.27 0.2..
##  $ 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..
##  $ 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 3.4..
##  $ 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), xlab = "Calidad", ylab = "Frecuencia")
```

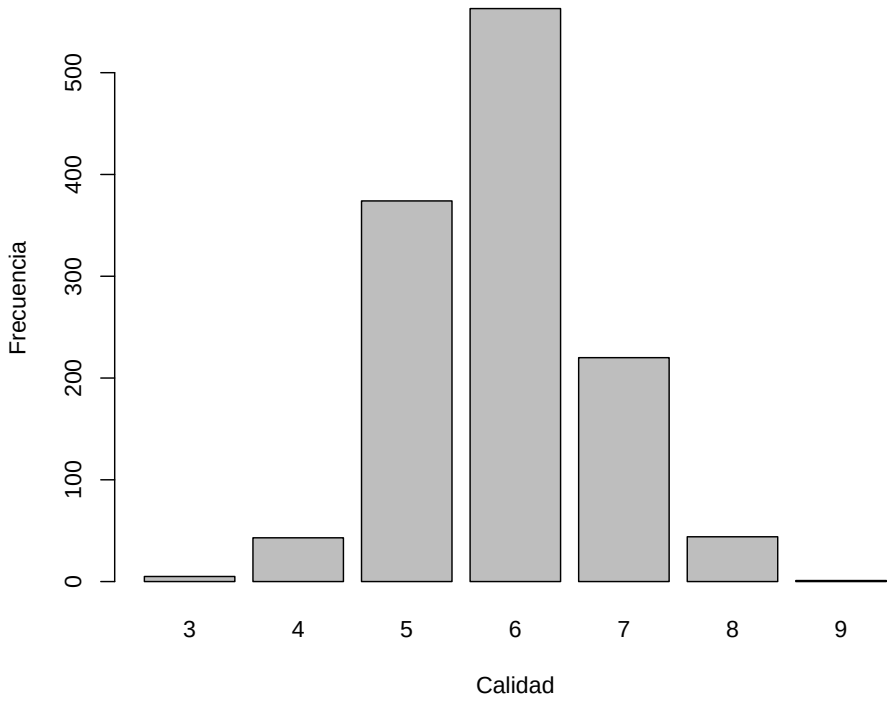


Figura 2.3: Distribución de las evaluaciones de la calidad del vino (`winequality$quality`).

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 árbol de decisión 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` (ver Figura 2.4):

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

Pero puede ser preferible emplear el paquete `rpart.plot` (ver Figura 2.5):

```
library(rpart.plot)
rpart.plot(tree)
```

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
##
```

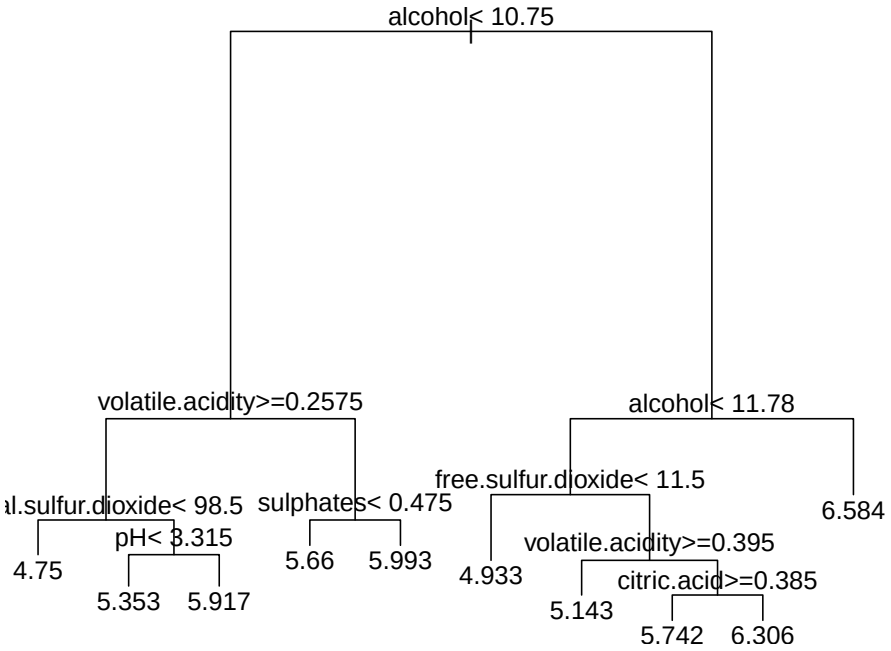


Figura 2.4: Árbol de regresión para predecir `winequality$quality` (obtenido con las opciones por defecto de `rpart()`).

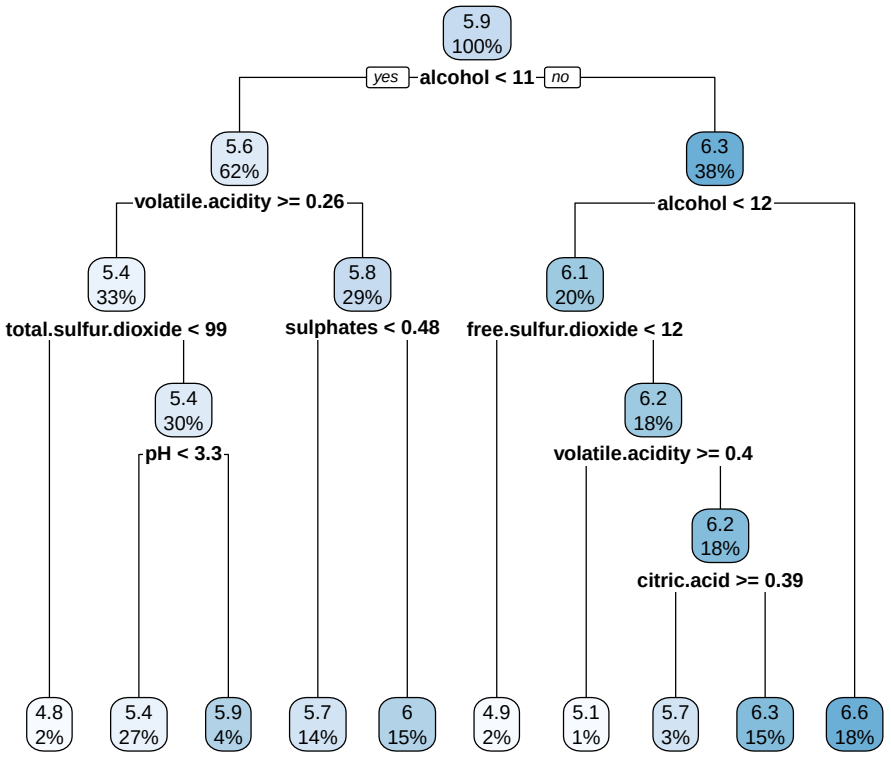


Figura 2.5: Representación del árbol de regresión generada con `rpart.plot()`.

```
## 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 árbol 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)[4]:

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

[4] Realmente en la tabla de texto se muestra el valor mínimo de CP, ya que se obtendría la misma solución para un rango de valores de CP (desde ese valor hasta el anterior, sin incluirlo), mientras que en el gráfico generado por `plotcp()` se representa la media geométrica de los extremos de ese intervalo (ver Figura 2.6).

```
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
## 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
##  [ reached getOption("max.print") -- omitted 48 rows ]
```

```
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
```

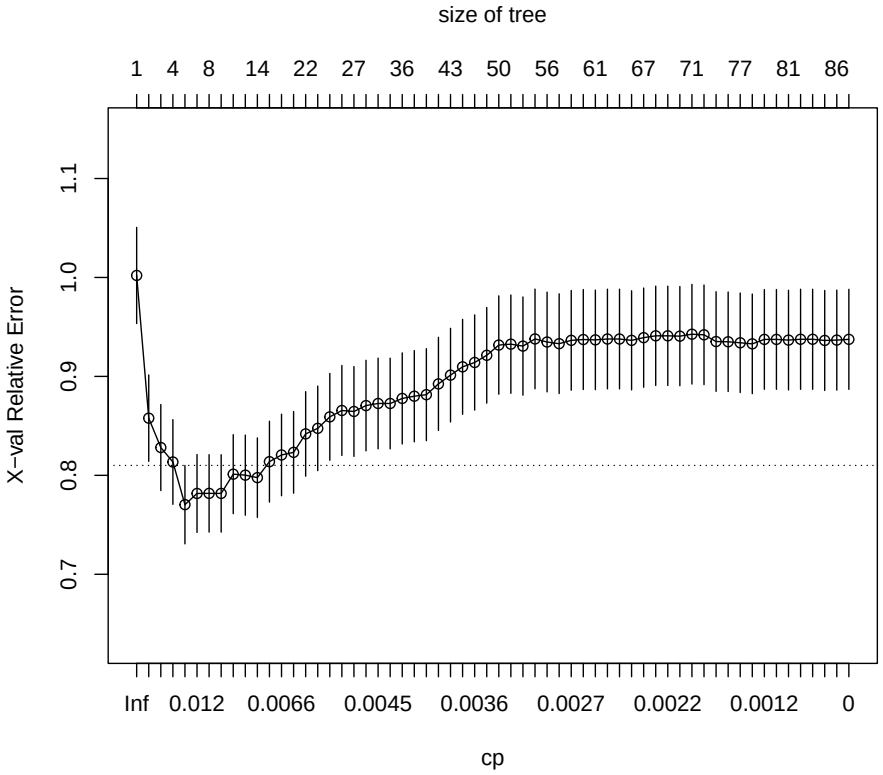


Figura 2.6: Error de validación cruzada (reescalado) dependiendo del parámetro de complejidad CP empleado en el ajuste del árbol de decisión.

```
## 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 (ver Figura 2.7) podamos el árbol 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)
```

Podríamos estudiar el modelo final, por ejemplo mediante el método `summary.rpart()`, 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
```

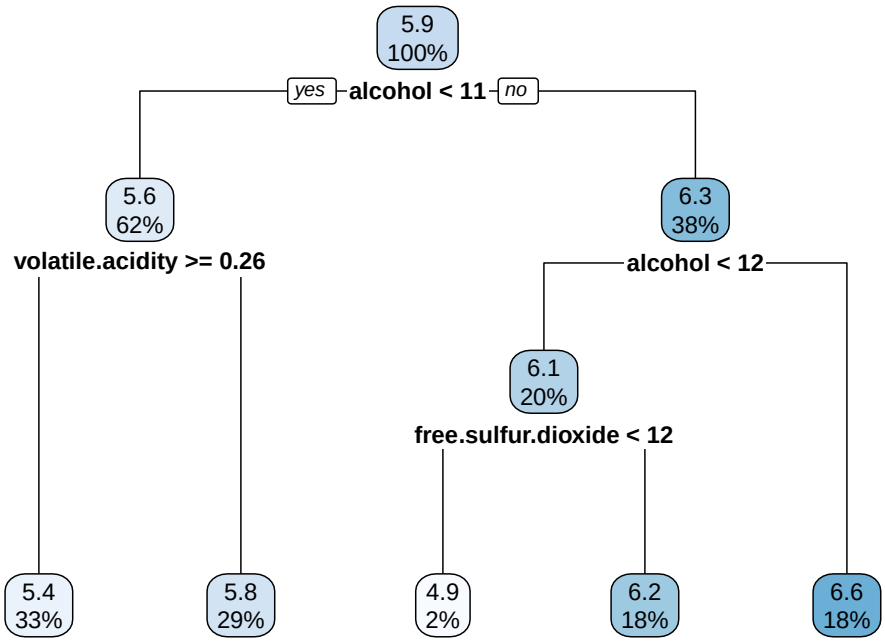


Figura 2.7: Árbol de regresión resultante después de la poda (modelo final).

```
##                  1.1
```

El último paso sería evaluarlo en la muestra de test siguiendo los pasos descritos en la Sección 1.3.4 (ver Figura 2.8):

```
obs <- test$quality
pred <- predict(tree, newdata = test)
# plot(pred, obs, xlab = "Predicción", ylab = "Observado")
plot(jitter(pred), jitter(obs), xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
```

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

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

```
# 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)
  ))
```

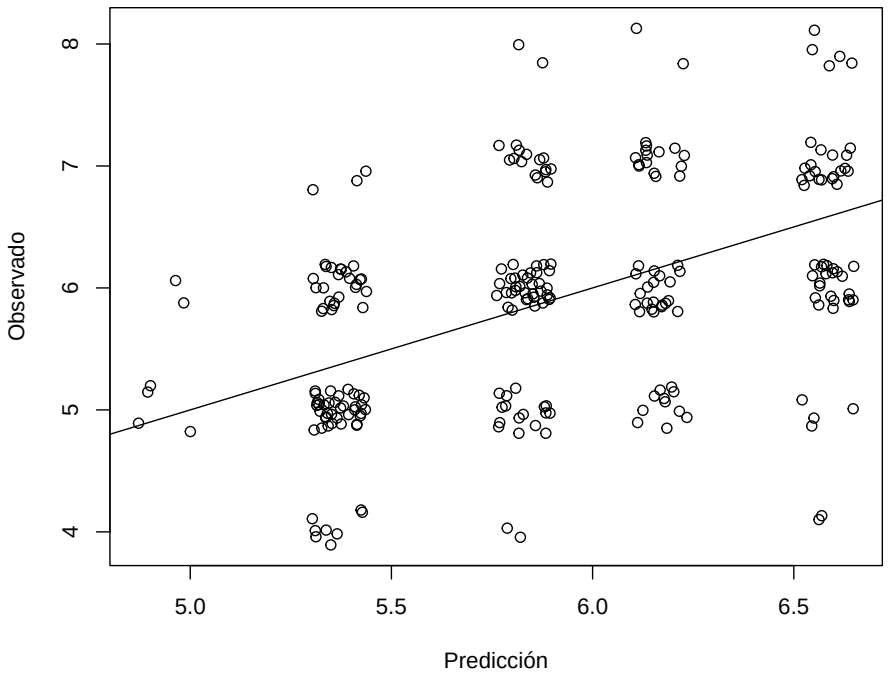


Figura 2.8: Gráfico de observaciones frente a predicciones (`test$quality`; se añade una perturbación para mostrar la distribución de los valores)

```
}
accuracy(pred, test$quality)
```

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

Como se puede observar el ajuste del modelo es bastante malo, como ya se comentó esto es habitual en árboles de regresión (especialmente si son tan pequeños) y normalmente solo se utilizan en un análisis exploratorio inicial (o como base para modelos más avanzados como los mostrados en el siguiente capítulo). En problemas de clasificación es más habitual que se puedan llegar a obtener buenos ajustes con árboles de decisión.

Ejercicio 2.1

Como se comentó en la introducción del Capítulo 1 al emplear el procedimiento habitual en AE de particionar los datos no se garantiza la reproducibilidad/repetibilidad de los resultados ya que dependen de la semilla. El modelo ajustado puede cambiar al variar la semilla (sobre todo si el conjunto de entrenamiento es pequeño; además, en algunos modelos el método de ajuste depende también de la semilla) pero normalmente no hay grandes cambios en las predicciones.

Podemos ilustrar el efecto de la semilla en los resultados empleando el ejemplo anterior. Habría que repetir el ajuste de un árbol de regresión considerando distintas semillas y comparar los resultados obtenidos.

La dificultad podría estar en como comparar los resultados. Una posible solución sería mantener fija la muestra de test (que forma que no dependa de las semillas). Por comodidad podríamos considerar las primeras `ntest` observaciones del conjunto de datos. Posteriormente, para cada semilla, seleccionaríamos la muestra de entrenamiento de la forma habitual y ajustaríamos un árbol. Finalmente evaluaríamos los resultados en la muestra de test.

Como base se podría considerar el siguiente código:

```
ntest <- 10
test <- winequality[1:ntest, ]
df <- winequality[-(1:ntest), ]
```

```
nobs <- nrow(df)
# Para las distintas semillas
set.seed(semilla)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
# tree <- ...
```

Como comentario final, en este caso el conjunto de datos no es muy grande y tampoco se obtuvo un buen ajuste con un árbol de regresión, por lo que sería de esperar que se observaran más diferencias.

Ejercicio 2.2

Como ya se mostró, el paquete `rpart` implementa la selección del parámetro de complejidad mediante validación cruzada. Como alternativa, siguiendo la idea del Ejercicio 1.1, y considerando de nuevo el ejemplo anterior, particionar la muestra en datos de entrenamiento (70%), de validación (15%) y de test (15%), para ajustar los árboles de decisión, seleccionar el parámetro de complejidad (el hiperparámetro) y evaluar las predicciones del modelo final, respectivamente.

Ejercicio 2.3

Una alternativa a particionar en entrenamiento y validación sería emplear bootstrap. La idea es emplear una remuestra bootstrap del conjunto de datos de entrenamiento para ajustar el modelo y utilizar las observaciones no seleccionadas (se suelen denominar datos *out of bag*) como conjunto de validación.

```
set.seed(1)
nobs <- nrow(winequality)
itrain <- sample(nobs, 0.8 * nobs)
train <- winequality[itrain, ]
test <- winequality[-itrain, ]
# Indice muestra de entrenamiento bootstrap
set.seed(1)
ntrain <- nrow(train)
itrain.boot <- sample(ntrain, replace = TRUE)
train.boot <- train[itrain.boot, ]
```

La muestra bootstrap va a contener muchas observaciones repetidas y habrá observaciones no seleccionadas. La probabilidad de que una observación no sea seleccionada es $(1 - 1/n)^n \approx e^{-1} \approx 0.37$.

```
# Número de casos "out of bag"
ntrain - length(unique(itrain.boot))
```

```
## [1] 370
```

```
# Muestra "out of bag"
# oob <- train[-unique(itrain.boot), ]
oob <- train[-itrain.boot, ]
```

El resto sería igual que el caso anterior cambiando `train` por `train.boot` y `validate` por `oob`.

Como comentario final, lo recomendable sería repetir el proceso un número grande de veces y promediar los errores (esto está relacionado con el método de *bagging* descrito en el siguiente capítulo), especialmente cuando el tamaño muestral es pequeño, pero por simplicidad consideraremos únicamente una muestra boostrap.

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, # levels = c('FALSE', 'TRUE')
                          labels = c('good', 'bad'))
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 0.2..
##  $ citric.acid         : num  0.47 0.34 0.13 0.49 0.26 0.63 0.07 0.36 0.27 0.2..
##  $ 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..
##  $ 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 3.4..
##  $ 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 (ver Figura 2.9):

```
library(rpart.plot)
rpart.plot(tree) # Alternativa: rattle::fancyRpartPlot
```

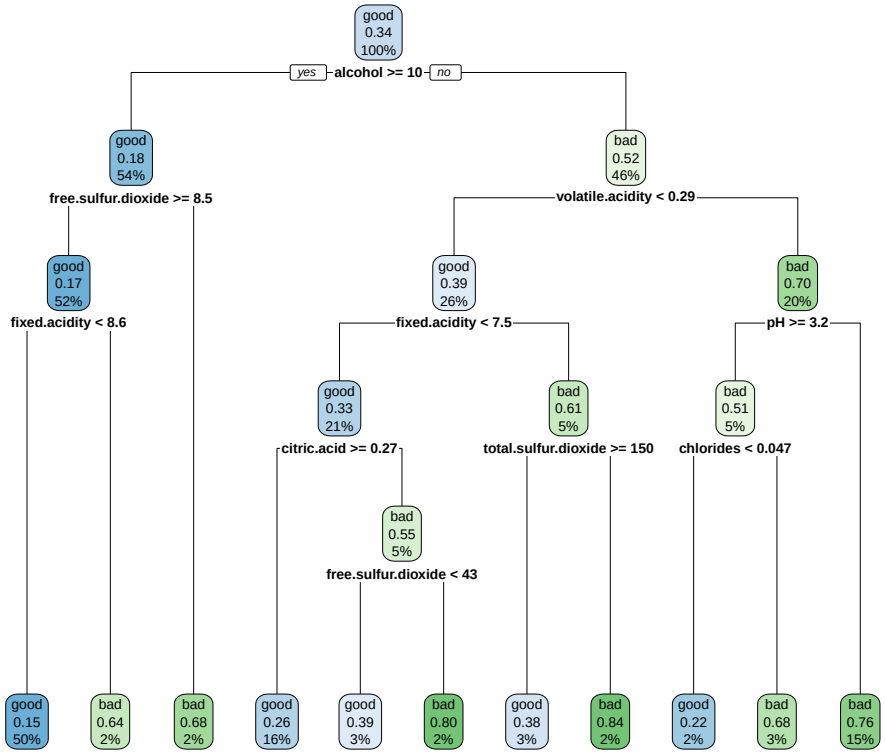


Figura 2.9: Árbol de clasificación de `winetaste$taste` (obtenido con las opciones por defecto).

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). Para ello se puede modificar la información que se muestra en cada nodo (ver Figura 2.10):

```
rpart.plot(tree,
           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
```

Al igual que en el caso de regresión, puede ser de utilidad imprimir las reglas:

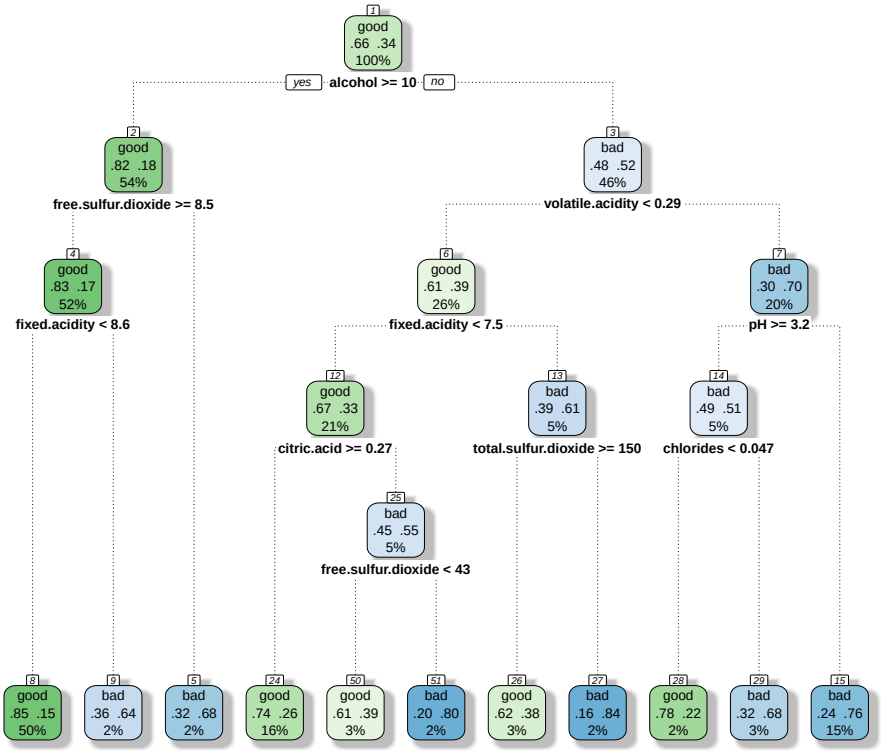


Figura 2.10: Representación del árbol de clasificación de `winetaste$taste` con opciones adicionales.

```
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
```

También se suele emplear el mismo procedimiento para seleccionar un valor óptimo del (hiper)parámetro de complejidad, se construye un árbol de decisión completo y se emplea 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[5], 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 (ver Figura 2.11)

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

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 árbol (ver Figura 2.12).

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

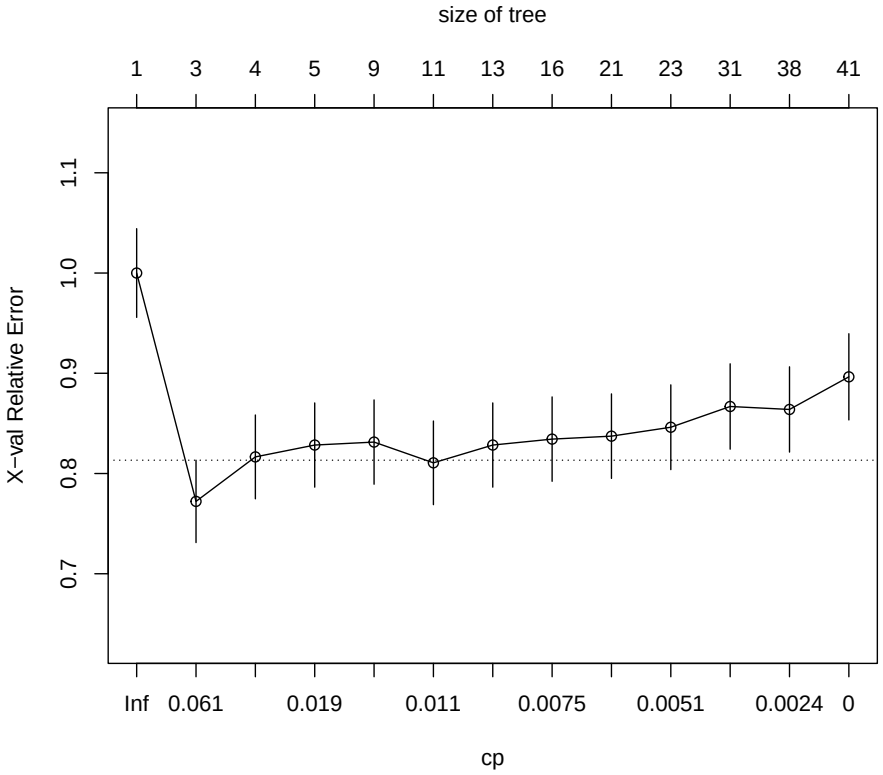


Figura 2.11: Evolución del error (reescalado) de validación cruzada en función del parámetro de complejidad.

```
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.rpart()` devuelve por defecto (`type = "prob"`) una matriz con las probabilidades de cada clase, por lo que habrá que establecer `type = "class"` (para más detalles consultar la ayuda de esta función).

```
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
```

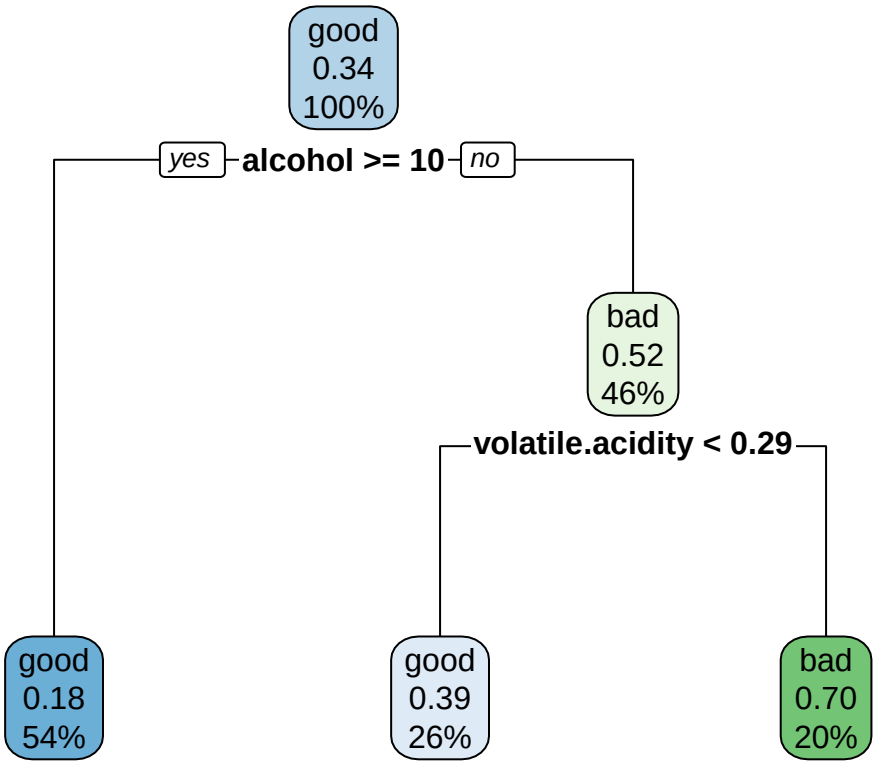


Figura 2.12: Árbol de clasificación de `winetaste$taste` obtenido después de la poda (modelo final).

```
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` (ver Figura 2.13).

```
library(caret)
# 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)
```

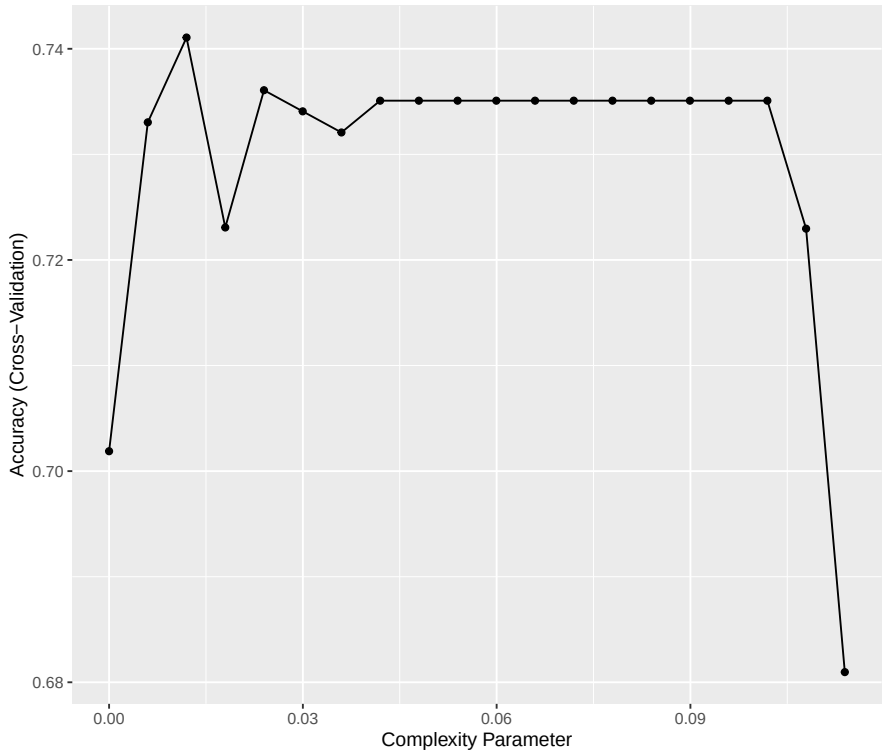


Figura 2.13: Evolución de la precisión (obtenida mediante validación cruzada) dependiendo del parámetro de complejidad.

El modelo final se devuelve en la componente `$finalModel` (ver Figura 2.14):

```
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)
```

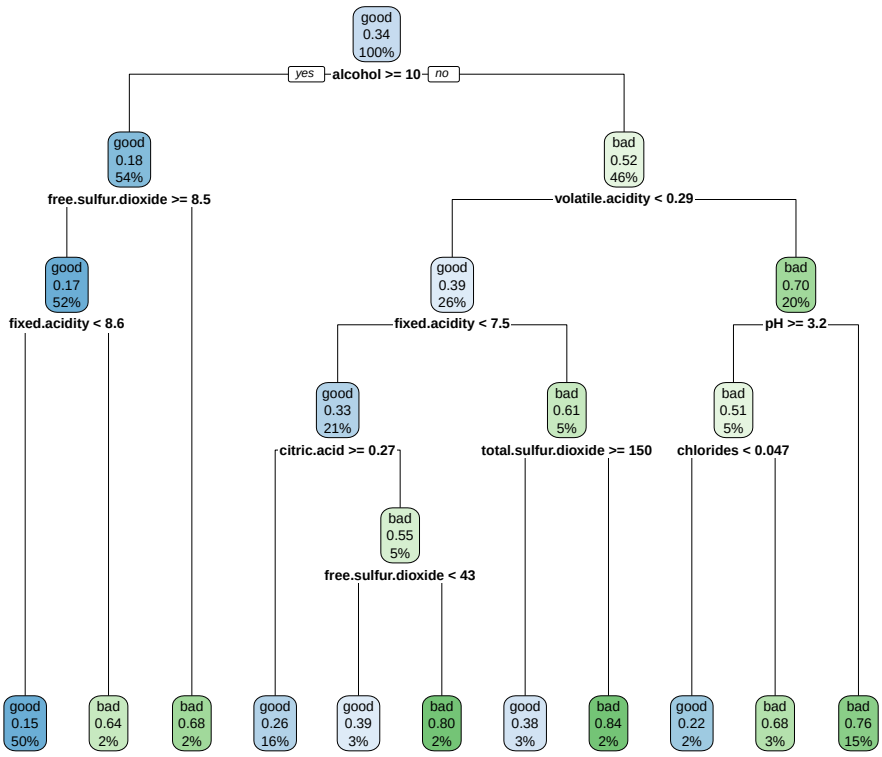


Figura 2.14: Árbol de clasificación de `winetaste$taste`, obtenido con la complejidad "óptima" (empleando `caret`).

Para utilizar la regla de "un error estándar" se puede añadir `selectionFunction = "oneSE"` en las opciones de entrenamiento[6](ver Figura 2.15):

```
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, ...
```

[6]En principio también se podría utilizar la regla de un error estándar estableciendo `method = "rpart1SE"` en la llamada a `train()`, pero `caret` implementa internamente este método y en ocasiones no se obtienen los resultados esperados.

```
## 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)
```

```
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
```

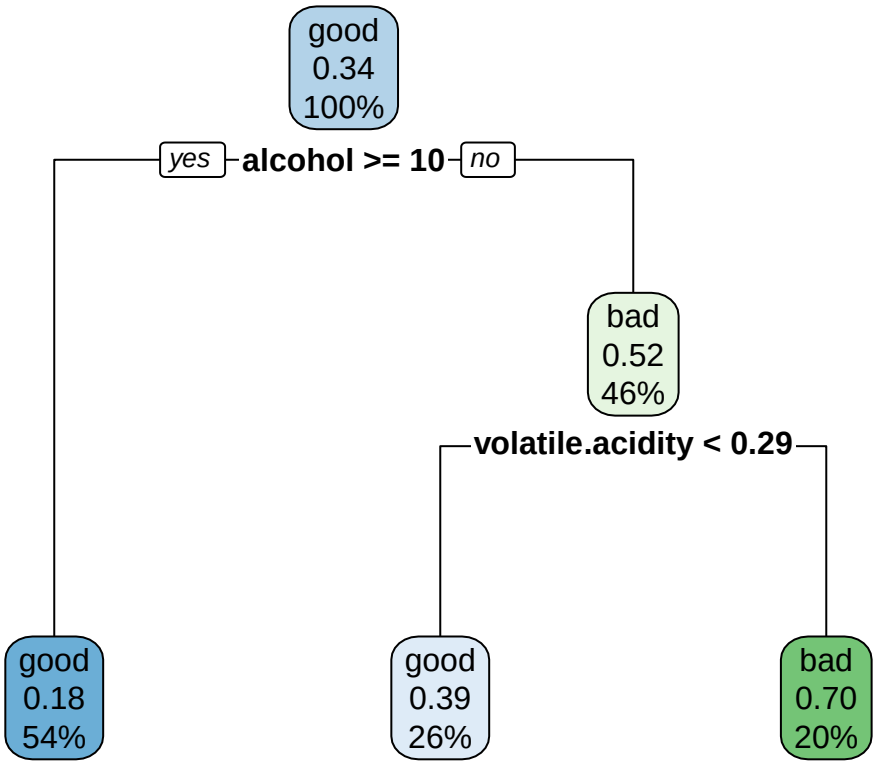


Figura 2.15: Árbol de clasificación de `winetaste$taste`, obtenido con la regla de un error estándar para seleccionar la complejidad (empleando `caret`).

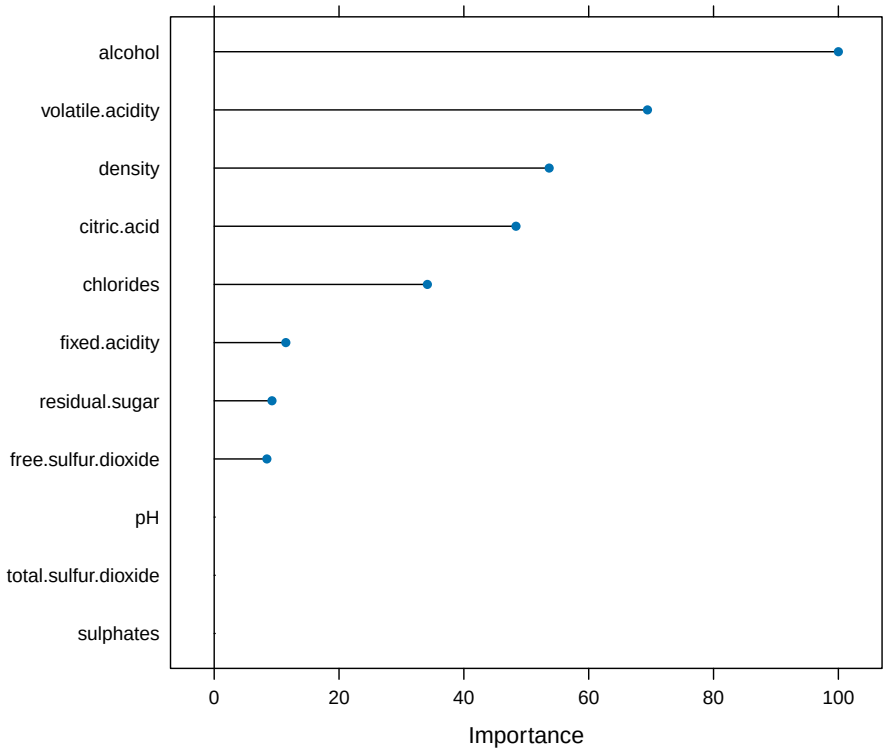


Figura 2.16: Importancia de los (posibles) predictores según el modelo obtenido con la regla de un error estándar.

```
##
```

```
##                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
##
```

2.4 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 et al., 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)[7], 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

Siguiendo con el problema de clasificación anterior, podríamos ajustar un árbol de decisión empleando la metodología de *inferencia condicional* mediante el siguiente código:

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

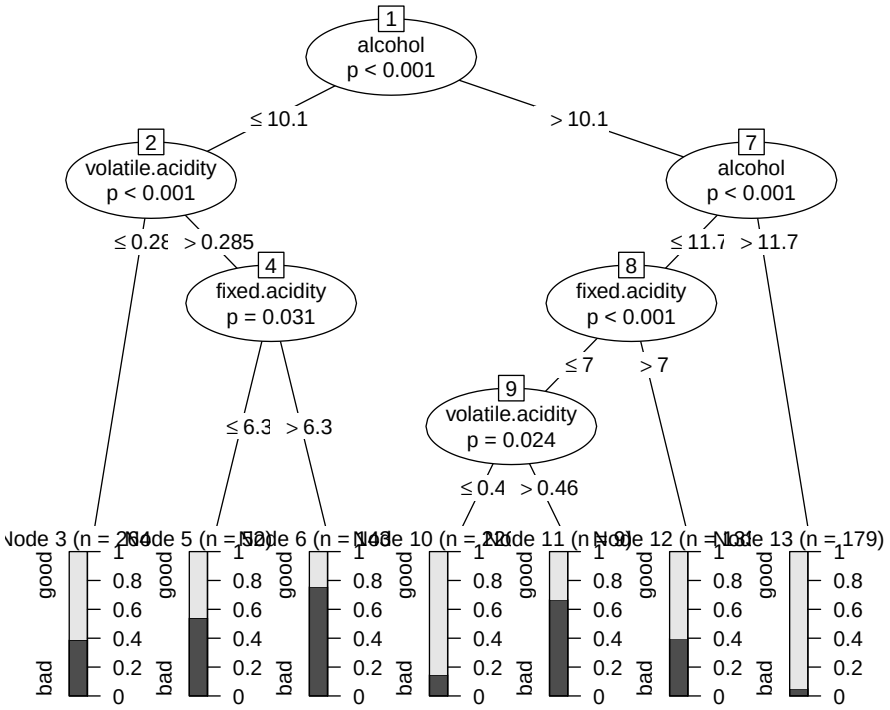


Figura 2.17: Árbol de decisión para clasificar la calidad del vino (`winetaste$taste`) obtenido con el método condicional.

Para más detalles ver la vignette del paquete *party: A Laboratory for Recursive Partytioning*.

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

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 margen 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 et al., 2023, pp. 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 construc-

ción 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 (2001b) 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, 2001b) se puede medir el incremento en el error de predicción OOB al permutar aleatoriamente los valores de la variable explicativa en las muestras 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`). Si la respuesta es un factor asumirá que se trata de un problema de clasificación y de regresión en caso contrario.
- `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, disminuir 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 estableciendo `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
##
##         OOB estimate of  error rate: 23.5%
## Confusion matrix:
##      good bad class.error
## good  565  97   0.1465257
```

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

```
## bad   138 200   0.4082840
```

Con el método `plot()` podemos examinar la convergencia del error en las muestras OOB (simplemente emplea `matplot()` para representar la componente `$err.rate` como se muestra en la Figura 3.1):

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

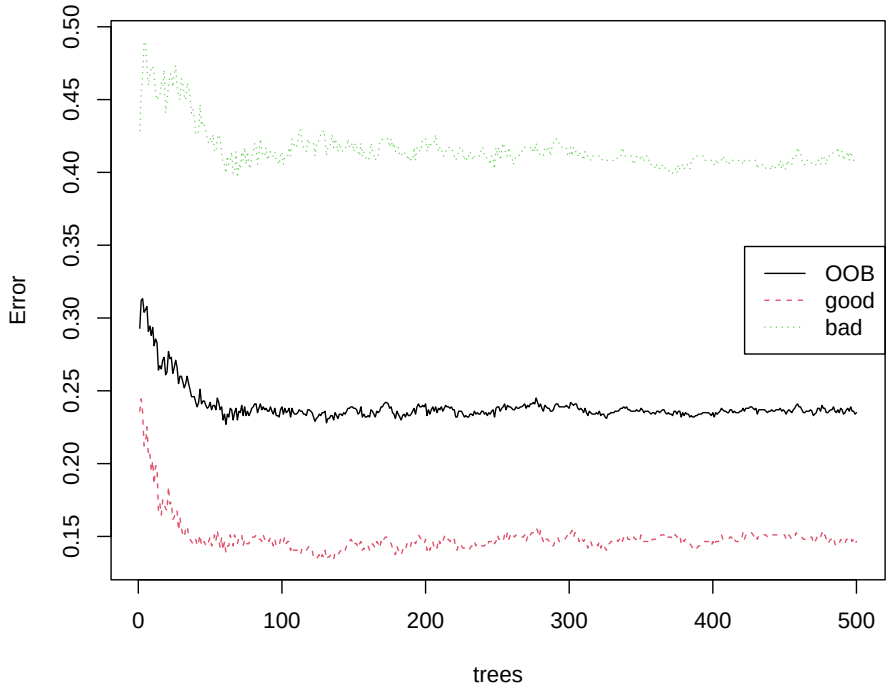


Figura 3.1: Tasas de error OOB al usar bagging para la predicción de `winetaste$taste` (realizado empleando `randomForest()` con `mtry` igual al número de predictores).

Como vemos que los errores se estabilizan podríamos pensar que aparentemente hay convergencia (aunque situaciones de alta dependencia entre los árboles 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:

```
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

Continuando con el ejemplo anterior, empleamos la función `randomForest()` con las opciones por defecto para ajustar un bosque aleatorio (a la muestra de entrenamiento generada anteriormente):

```
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 en la Figura 3.2 que aparentemente hay convergencia y tampoco sería necesario incrementar el número de árboles:

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

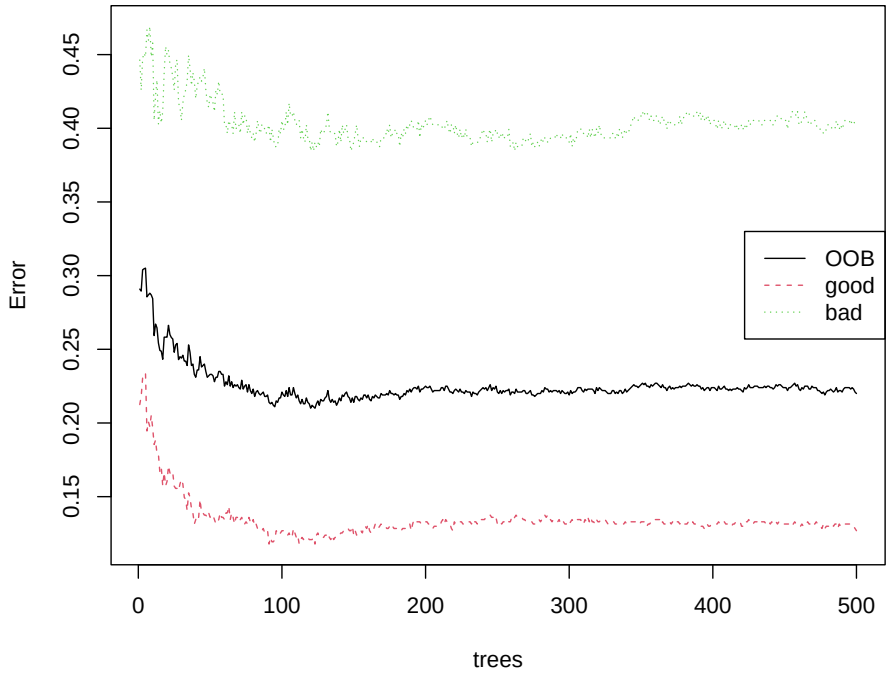


Figura 3.2: Tasas de error OOB al usar bosques aleatorios para la predicción de `winetaste$taste` (empleando `randomForest()` con las opciones por defecto).

Podemos mostrar la importancia de las variables predictoras (utilizadas en el bosque aleatorio y sus sustituas) con la función `importance()` o representarlas con `varImpPlot()` (ver Figura 3.3):

```
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
##
```

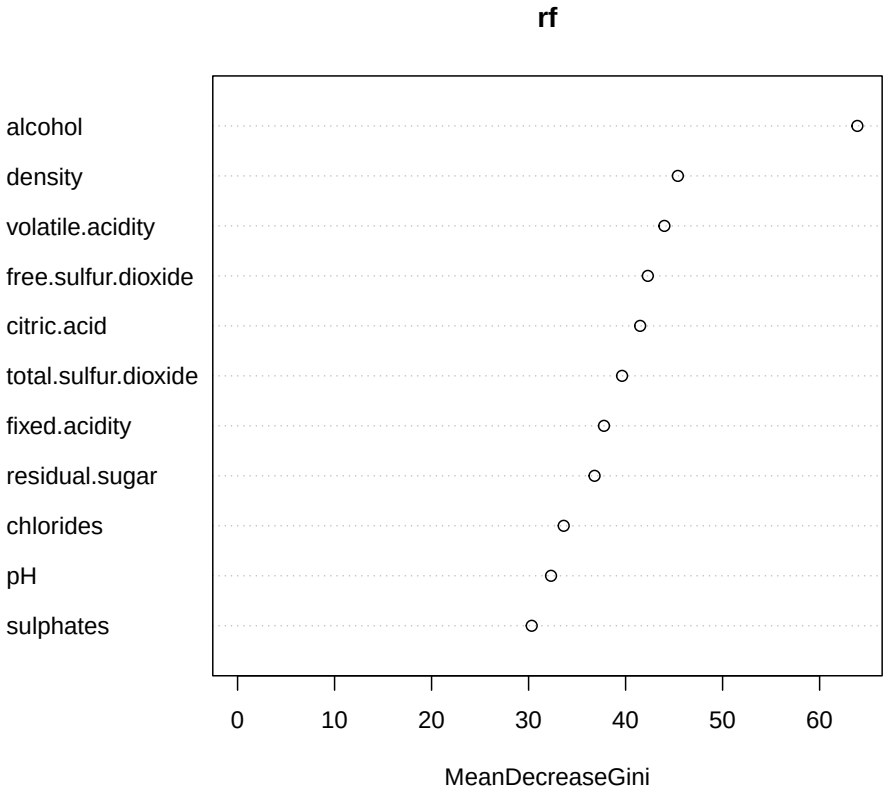


Figura 3.3: Importancia de las variables predictoras al emplear bosques aleatorios para la predicción de `winetaste$taste`.

```
##                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
##
```

Esta mejora sería 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
```

El análisis e interpretación del modelo puede resultar más complicado en este tipo de métodos. Para estudiar el efecto de los predictores en la respuesta se suelen emplear alguna de las herramientas descritas en la Sección 1.5. Por ejemplo, empleando la función `pdp::partial()`, podemos generar gráficos PDP estimando los efectos individuales de los predictores (ver Figura 3.4):

```
library(pdp)
library(gridExtra)
pdp1 <- partial(rf, "alcohol")
p1 <- plotPartial(pdp1)
pdp2 <- partial(rf, c("density"))
p2 <- plotPartial(pdp2)
grid.arrange(p1, p2, ncol = 2)
```

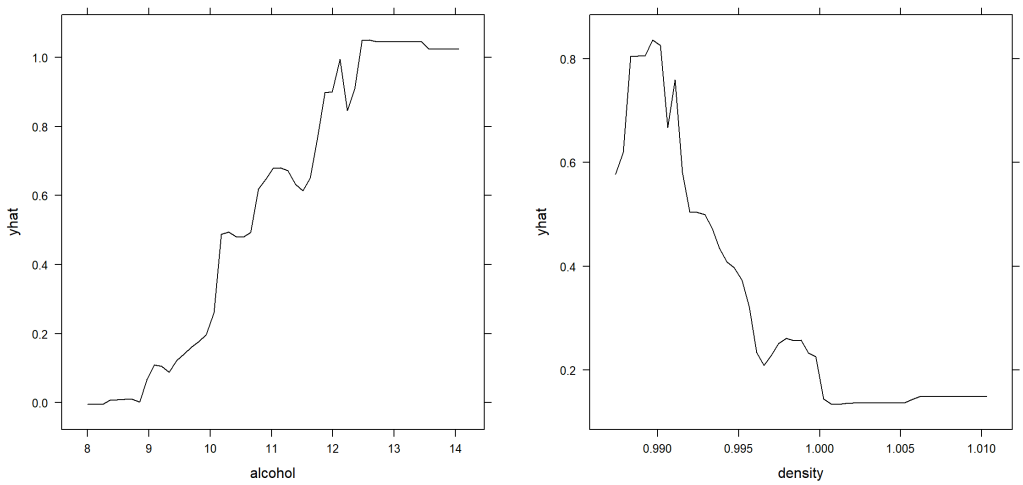


Figura 3.4: Efecto parcial del alcohol (panel izquierdo) y la densidad (panel derecho) sobre la respuesta.

O gráficos PDP considerando la interacción entre dos predictores (ver Figura 3.5) (cuidado, puede requerir de mucho tiempo de computación):

```
pdp12 <- partial(rf, c("alcohol", "density"))
plotPartial(pdp12)
```

Adicionalmente, estableciendo `ice = TRUE` se calculan las curvas de expectativa condicional individual (ICE). Estos gráficos ICE extienden los PDP, ya que además de mostrar la variación del promedio (ver línea roja en la Figura 3.6), también muestra la variación de los valores predichos para cada observación (ver líneas en negro en la Figura 3.6).

```
ice1 <- partial(rf, pred.var = "alcohol", ice = TRUE)
ice2 <- partial(rf, pred.var = "density", ice = TRUE)
p1 <- plotPartial(ice1, alpha = 0.5)
p2 <- plotPartial(ice2, alpha = 0.5)
gridExtra:::grid.arrange(p1, p2, ncol = 2)
```

Se pueden crear gráficos similares utilizando los otros paquetes indicados en la Sección 1.5.

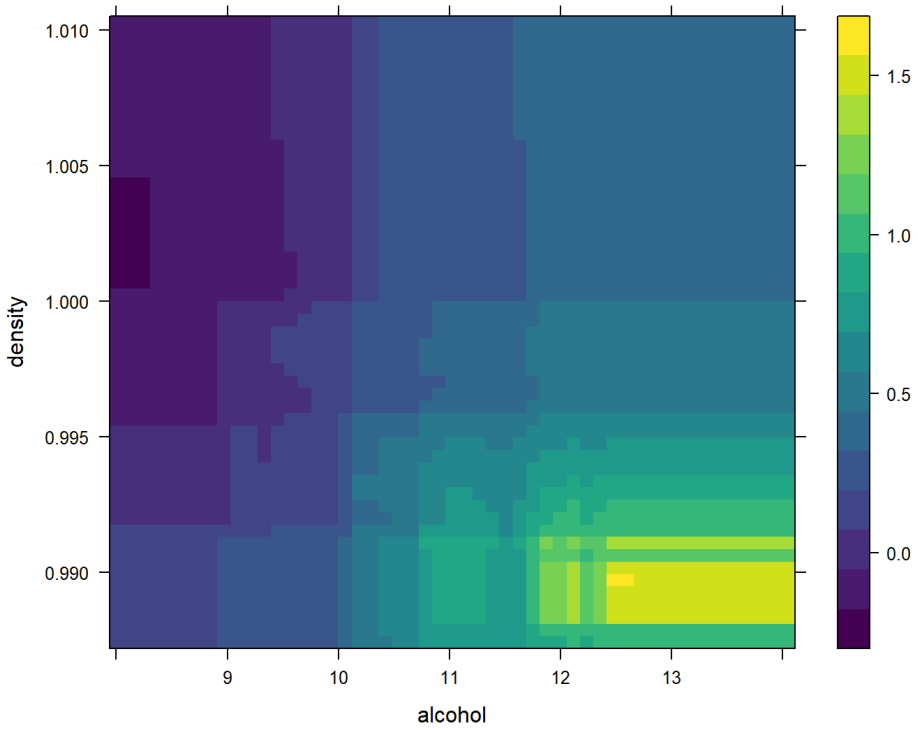


Figura 3.5: Efecto parcial de la interacción del alcohol y la densidad sobre la respuesta.

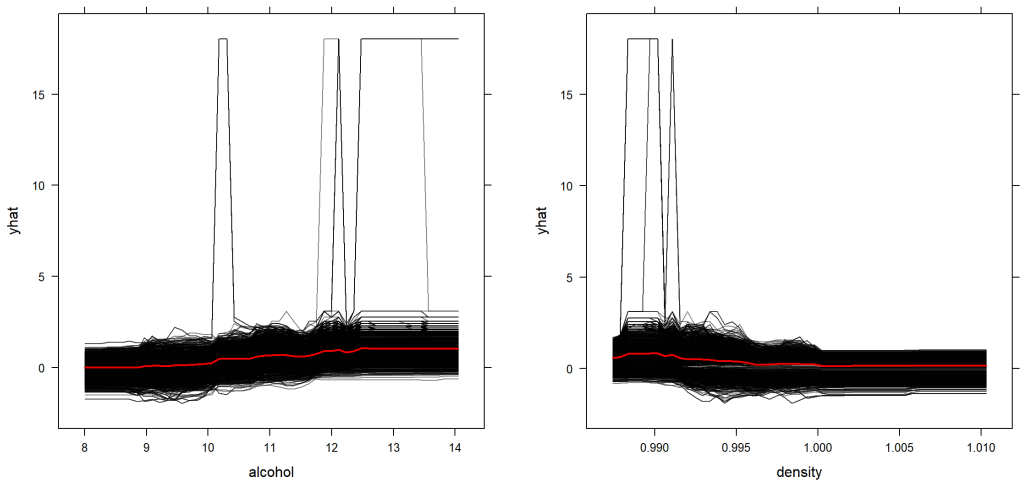


Figura 3.6: Efecto individual de cada observación de alcohol (panel izquierdo) y densidad (panel derecho) sobre la respuesta.

Por ejemplo, la Figura 3.7, generada con el paquete `vivid`, muestra medidas de la importancia de los predictores (*Vimp*) en la diagonal y de la fuerza de las interacciones (*Vint*) fuera de la diagonal.

```
library(vivid)
fit_rf <- vivi(data = train, fit = rf, response = "taste",
               importanceType = "%IncMSE")
viviHeatmap(mat = fit_rf[1:5,1:5])
```

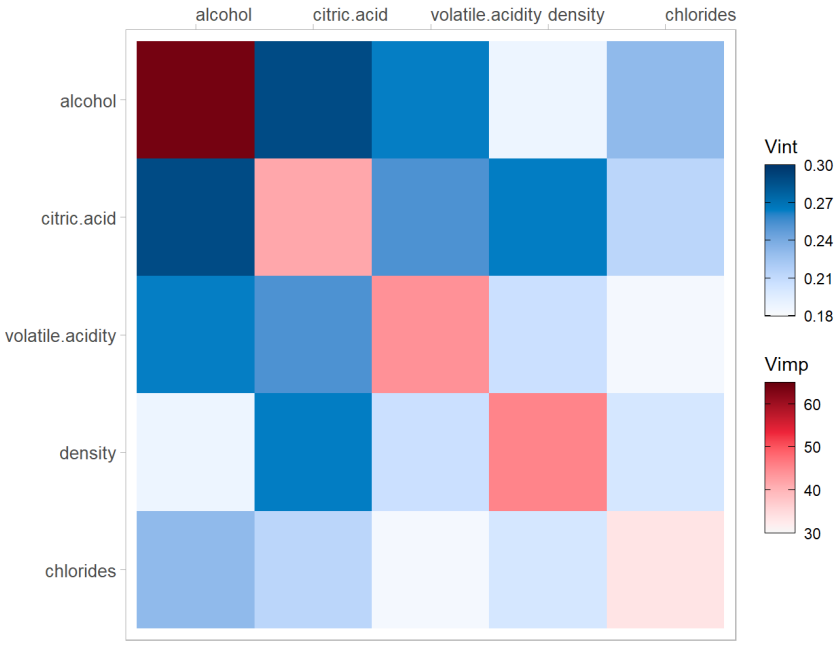


Figura 3.7: Mapa de calor de la importancia e interaciones de los predictores del ajuste mediante bosques aleatorios.

Alternativamente, también se pueden visualizar las relaciones mediante un gráfico de red (ver Figura 3.7).

```
require(igraph)
viviNetwork(mat = fit_rf)
```

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
```

Con las opciones por defecto únicamente evalúa tres valores posibles del hiperparámetro (ver Figura 3.9). Opcionalmente se podría aumentar el número valores a evaluar con `tuneLength` o directamente especificarlos con `tuneGrid`. En cualquier caso el tiempo de computación puede ser demasiado alto, por lo que puede ser recomendable reducir el valor de `nodesize`, paralelizar los cálculos o emplear otros paquetes con implementaciones 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.

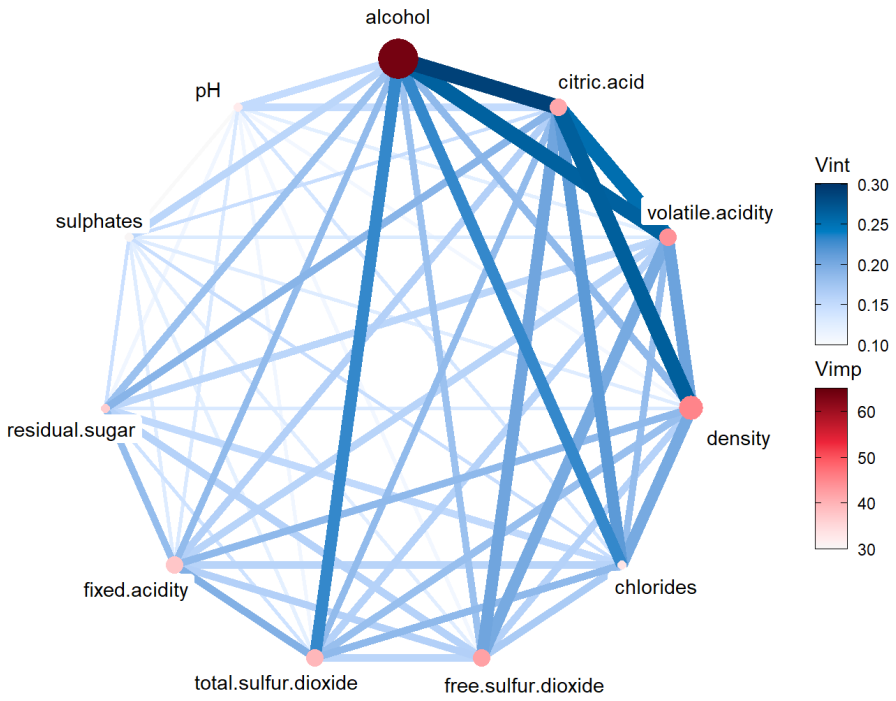


Figura 3.8: Gráfico de red para la importancia e interaciones del ajuste mediante bosques aleatorios.

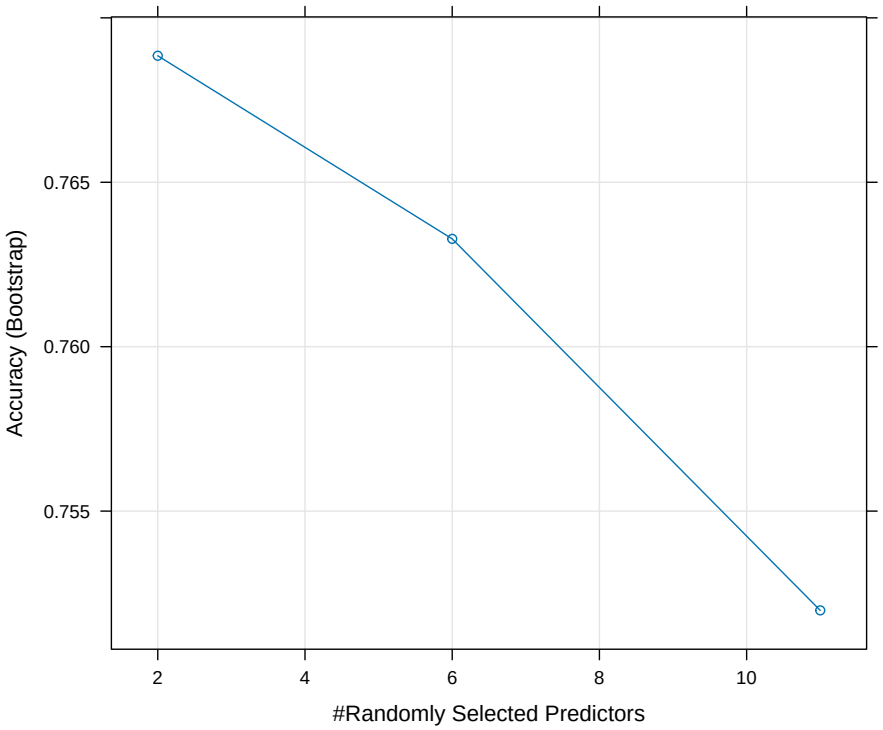


Figura 3.9: Evolución de la precisión de un bosque aleatorio dependiendo del número de predictores seleccionados.

Breiman (2001a) sugiere emplear el valor por defecto para `mtry`, la mitad y el doble (ver Figura 3.10):

```
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)
```

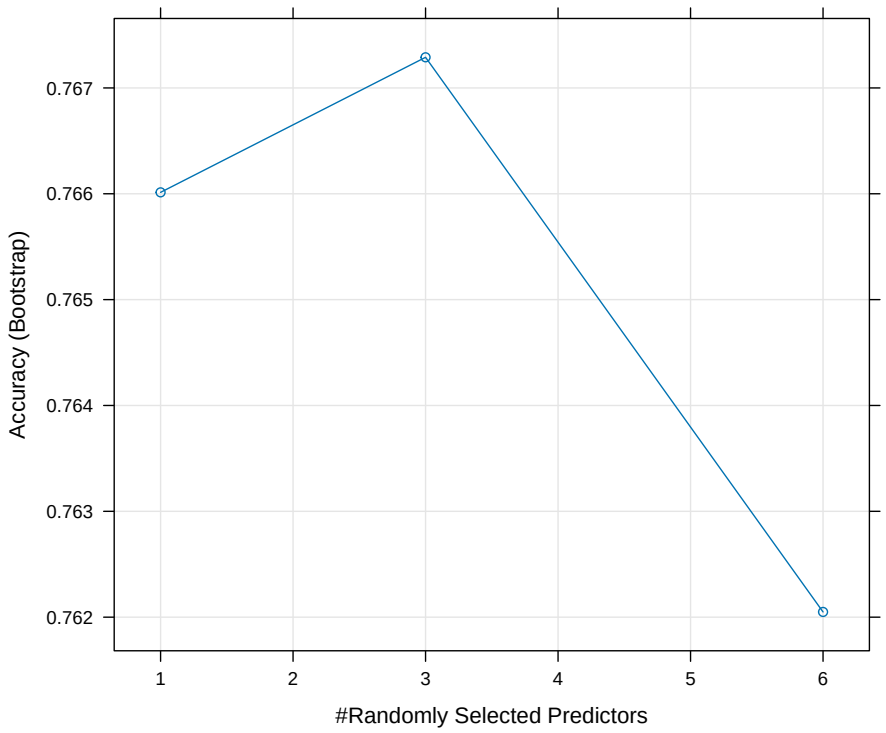


Figura 3.10: Evolución de la precisión de un bosque aleatorio con `caret` usando el argumento `tuneGrid`.

Ejercicio 3.1

Como acabamos de ver, `caret` permite ajustar un bosque aleatorio considerando `mtry` como único hiperparámetro, pero nos podría interesar buscar también valores adecuados para otros parámetros, como por ejemplo `nodesize`. Esto se puede realizar fácilmente empleando directamente la función `randomForest()`. En primer lugar habría que construir la rejilla de búsqueda, con las combinaciones de los valores de los hiperparámetros que se quieren evaluar (para ello se puede utilizar la función `expand.grid()`). Posteriormente se ajustaría un bosque aleatorio en la muestra de entrenamiento con cada una de las combinaciones (por ejemplo utilizando un bucle `for`) y se emplearía el error OOB para seleccionar la combinación óptima (al que podemos acceder empleando `with(fit, err.rate[ntree, "OOB"])` suponiendo que `fit` contiene el bosque aleatorio ajustado).

Continuando con el mismo conjunto de datos de calidad de vino, emplear la función `randomForest()` para ajustar un bosque aleatorio con el fin de clasificar la calidad del vino `taste`, considerando 500 árboles y empleando el error OOB para seleccionar los valores "óptimos" de los hiperparámetros considerando las posibles combinaciones de `mtry = floor(c(mtry.class/2, mtry.class, 2*mtry.class))` (siendo `mtry.class <- sqrt(ncol(train) - 1)`) y `nodesize = c(1, 3, 5, 10)`.

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 (1994), 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 $b = 1, 2, ..., B$, 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 $\widehat{y}_b(\mathbf{x})$ su clasificación utilizando árbol b-ésimo, entonces $\widehat{y}(\mathbf{x}) = signo\left(\sum_b s_b \widehat{y}_b(\mathbf{x})\right)$ (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:
$$\hat{m}(x) = 0,\ r_i = y_i$$

1. 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:
   $$r_i \leftarrow r_i - \lambda \hat{m}^b(x_i)$$

2. 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 (Friedman, 2002), inspirado por la técnica bagging de Breiman. Esta variante, conocida como *stochastic gradient boosting* (SGB), 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`).

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.).

SGB incorpora dos ventajas importantes: reduce la varianza y reduce los tiempos de cómputo. En terminos de rendimiento tanto el método SGB 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

Estos métodos son también de los más populares en AE y están implementados en numerosos paquetes de R: `ada`, `adabag`, `mboost`, `gbm`, `xgboost`…

3.5.1 Ejemplo: clasificación con el paquete `ada`

La función `ada()` del paquete `ada` (Culp et al., 2006) implementa diversos métodos boosting (incluyendo el algoritmo original AdaBoost). Emplea `rpart` para la construcción de los árboles, aunque solo admite respuestas dicotómicas y dos funciones de pérdida (exponencial y logística). Además, un posible problema al emplear esta función es que ordena alfabéticamente los niveles del factor, lo que puede llevar a una mala interpretación de los resultados.

Los principales parámetros son los siguientes:

```
ada(formula, data, loss = c("exponential", "logistic"),
    type = c("discrete", "real", "gentle"), iter = 50,
    nu = 0.1, bag.frac = 0.5, ...)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (típicamente `respuesta ~ .`; también admite matrices `x` e `y` en lugar de fórmulas).
- `loss`: función de pérdida; por defecto `"exponential"` (algoritmo AdaBoost).

- `type`: algoritmo boosting; por defecto `"discrete"` que implementa el algoritmo AdaBoost original que predice la variable respuesta. Otras alternativas son `"real"`, que implementa el algoritmo *Real AdaBoost* (J. Friedman et al., 2000) que permite estimar las probabilidades, y `"gentle"` , versión modificada del anterior que emplea un método Newton de optimización por pasos (en lugar de optimización exacta).
- `iter`: número de iteraciones boosting; por defecto 50.
- `nu`: parámetro de regularización λ; por defecto 0.1 (disminuyendo este parámetro es de esperar que se obtenga una mejora en la precisión de las predicciones pero requería aumentar `iter` aumentando notablemente el tiempo de computación y los requerimientos de memoria).
- `bag.frac`: proporción de observaciones seleccionadas al azar para crecer cada árbol; por defecto 0.5.
- `...`: argumentos adicionales para `rpart.control`; por defecto `rpart.control(maxdepth = 1, cp = -1, minsplit = 0, xval = 0)`.

Como ejemplo consideraremos el conjunto de datos de calidad de vino empleado en las secciones 2.3.2 y 3.3, pero para evitar problemas reordenamos alfabéticamente los niveles de la respuesta.

```
load("data/winetaste.RData")
# Reordenar alfabéticamente los niveles de winetaste$taste
# winetaste$taste <- factor(winetaste$taste, sort(levels(winetaste$taste)))
winetaste$taste <- factor(as.character(winetaste$taste))
# Partición de los datos
set.seed(1)
df <- winetaste
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
```

Por ejemplo, el siguiente código llama a la función `ada()` con la opción para estimar probabilidades (`type = "real"`, Real AdaBoost), considerando interacciones (de orden 2) entre los predictores (`maxdepth = 2`), disminuyendo ligeramente el valor del parámetro de aprendizaje y aumentando el número de iteraciones:

```
library(ada)
ada.boost <- ada(taste ~ ., data = train, type = "real",
                 control = rpart.control(maxdepth = 2, cp = 0, minsplit = 10, xval = 0),
                 iter = 100, nu = 0.05)
ada.boost
```

```
## Call:
## ada(taste ~ ., data = train, type = "real", control = rpart.control(maxdepth = 2,
##     cp = 0, minsplit = 10, xval = 0), iter = 100, nu = 0.05)
##
## Loss: exponential Method: real   Iteration: 100
##
## Final Confusion Matrix for Data:
##           Final Prediction
## True value bad good
##       bad  162  176
##       good  46  616
##
## Train Error: 0.222
##
## Out-Of-Bag Error:  0.233  iteration= 99
##
## Additional Estimates of number of iterations:
```

```
##
## train.err1 train.kap1
##         93          93
```

Con el método `plot()` podemos representar la evolución del error de clasificación al aumentar el número de iteraciones (ver Figura 3.11):

```
plot(ada.boost)
```

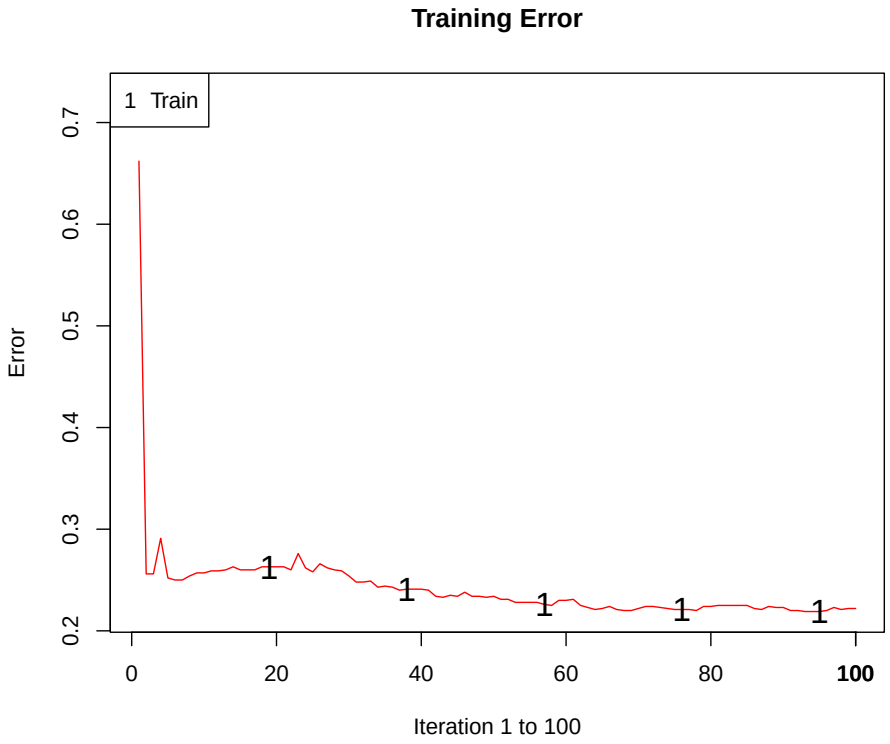


Figura 3.11: Evolución de la tasa de error utilizando `ada()`.

Podemos evaluar la precisión en la muestra de test empleando el procedimiento habitual:

```
pred <- predict(ada.boost, newdata = test)
caret::confusionMatrix(pred, test$taste, positive = "good")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction bad good
##       bad   34   16
##       good  50  150
##
##                Accuracy : 0.736
##                  95% CI : (0.6768, 0.7895)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.008615
##
##                   Kappa : 0.3426
##
##  Mcnemar's Test P-Value : 4.865e-05
##
##             Sensitivity : 0.9036
##             Specificity : 0.4048
##          Pos Pred Value : 0.7500
```

```
##          Neg Pred Value : 0.6800
##              Prevalence : 0.6640
##          Detection Rate : 0.6000
##    Detection Prevalence : 0.8000
##       Balanced Accuracy : 0.6542
##
##        'Positive' Class : good
##
```

Para obtener las estimaciones de las probabilidades, habría que establecer `type = "probs"` al predecir (devolverá una matriz con columnas correspondientes a los niveles):

```
p.est <- predict(ada.boost, newdata = test, type = "probs")
head(p.est)
```

```
##          [,1]      [,2]
## 1  0.49877103 0.5012290
## 4  0.30922187 0.6907781
## 9  0.02774336 0.9722566
## 10 0.04596187 0.9540381
## 12 0.44274407 0.5572559
## 16 0.37375910 0.6262409
```

Este procedimiento también está implementado en el paquete `caret` seleccionando el método `"ada"`, que considera como hiperparámetros:

```
library(caret)
modelLookup("ada")
```

```
##   model parameter          label forReg forClass probModel
## 1   ada      iter         #Trees  FALSE     TRUE      TRUE
## 2   ada  maxdepth Max Tree Depth  FALSE     TRUE      TRUE
## 3   ada        nu  Learning Rate  FALSE     TRUE      TRUE
```

Aunque por defecto la función `train()` solo considera nueve combinaciones de hiperparámetros:

```
set.seed(1)
caret.ada0 <- train(taste ~ ., method = "ada", data = train,
                    trControl = trainControl(method = "cv", number = 5))
caret.ada0
```

```
## Boosted Classification Trees
##
## 1000 samples
##   11 predictor
##    2 classes: 'bad', 'good'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 801, 800, 800, 799
## Resampling results across tuning parameters:
##
##   maxdepth  iter  Accuracy   Kappa
##   1          50   0.7100121  0.2403486
##   1         100   0.7220322  0.2824931
##   1         150   0.7360322  0.3346624
##   2          50   0.7529774  0.3872880
##   2         100   0.7539673  0.4019619
##   2         150   0.7559673  0.4142035
##   3          50   0.7570024  0.4112842
```

```
##   3         100   0.7550323  0.4150030
##   3         150   0.7650024  0.4408835
##
## Tuning parameter 'nu' was held constant at a value of 0.1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were iter = 150, maxdepth = 3 and nu = 0.1.
```

```
confusionMatrix(predict(caret.ada0, newdata = test), test$taste, positive = "good")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction bad good
##       bad   37   22
##       good  47  144
##
##                Accuracy : 0.724
##                  95% CI : (0.6641, 0.7785)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.024724
##
##                   Kappa : 0.3324
##
##  Mcnemar's Test P-Value : 0.003861
##
##             Sensitivity : 0.8675
##             Specificity : 0.4405
##          Pos Pred Value : 0.7539
##          Neg Pred Value : 0.6271
##              Prevalence : 0.6640
##          Detection Rate : 0.5760
##    Detection Prevalence : 0.7640
##       Balanced Accuracy : 0.6540
##
##        'Positive' Class : good
##
```

Se puede aumentar el número de combinaciones empleando `tuneLength` o `tuneGrid` pero la búsqueda en una rejilla completa puede incrementar considerablemente el tiempo de computación. Por este motivo se suelen seguir distintos procedimientos de búsqueda. Por ejemplo, fijar la tasa de aprendizaje (inicialmente a un valor alto) para seleccionar primero un número de interaciones y la complejidad del árbol, y posteriormente fijar estos valores para seleccionar una nueva tasa de aprendizaje (repitiendo el proceso, si es necesario, hasta convergencia).

```
set.seed(1)
caret.ada1 <- train(taste ~ ., method = "ada", data = train,
                    tuneGrid = data.frame(iter =  150, maxdepth = 3,
                                          nu = c(0.3, 0.1, 0.05, 0.01, 0.005)),
                    trControl = trainControl(method = "cv", number = 5))
caret.ada1
```

```
## Boosted Classification Trees
##
## 1000 samples
##   11 predictor
##    2 classes: 'bad', 'good'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
```

```
## Summary of sample sizes: 800, 801, 800, 800, 799
## Resampling results across tuning parameters:
##
##   nu     Accuracy   Kappa
##   0.005  0.7439722  0.3723405
##   0.010  0.7439822  0.3725968
##   0.050  0.7559773  0.4116753
##   0.100  0.7619774  0.4365242
##   0.300  0.7580124  0.4405127
##
## Tuning parameter 'iter' was held constant at a value of 150
## Tuning
##  parameter 'maxdepth' was held constant at a value of 3
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were iter = 150, maxdepth = 3 and nu = 0.1.
```

```
confusionMatrix(predict(caret.ada1, newdata = test), test$taste, positive = "good")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction bad good
##       bad   40   21
##       good  44  145
##
##                Accuracy : 0.74
##                  95% CI : (0.681, 0.7932)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.005841
##
##                   Kappa : 0.375
##
##  Mcnemar's Test P-Value : 0.006357
##
##             Sensitivity : 0.8735
##             Specificity : 0.4762
##          Pos Pred Value : 0.7672
##          Neg Pred Value : 0.6557
##              Prevalence : 0.6640
##          Detection Rate : 0.5800
##    Detection Prevalence : 0.7560
##       Balanced Accuracy : 0.6748
##
##        'Positive' Class : good
##
```

3.5.2 Ejemplo: regresión con el paquete `gbm`

El paquete `gbm` implementa el algoritmo SGB de Friedman (2002) y admite varios tipos de respuesta considerando distintas funciones de pérdida (aunque en el caso de variables dicotómicas éstas deben tomar valores en $\{0,1\}$[3]). La función principal es `gbm()` y se suelen considerar los siguientes argumentos:

```
gbm( formula, distribution = "bernoulli", data, n.trees = 100,
     interaction.depth = 1, n.minobsinnode = 10,
     shrinkage = 0.1, bag.fraction = 0.5,
```

[3]Se puede evitar este inconveniente empleando la interfaz de `caret`.

```
    cv.folds = 0, n.cores = NULL)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (típicamente `respuesta ~ .`; también está disponible una interfaz con matrices `gbm.fit()`).
- `distribution` (opcional): texto con el nombre de la distribución (o lista con el nombre en `name` y parámetros adicionales en los demás componentes) que determina la función de pérdida. Si se omite se establecerá a partir del tipo de la respuesta: `"bernouilli"` (regresión logística) si es una variable dicotómica 0/1, `"multinomial"` (regresión multinomial) si es un factor (no se recomienda) y `"gaussian"` (error cuadrático) en caso contrario. Otras opciones que pueden ser de interés son: `"laplace"` (error absoluto), `"adaboost"` (pérdida exponencial para respuestas dicotómicas 0/1), `"huberized"` (pérdida de Huber para respuestas dicotómicas 0/1), `"poisson"` (regresión de Poisson) y `"quantile"` (regresión cuantil).
- `ntrees`: iteraciones/número de árboles que se crecerán; por defecto 100 (se puede emplear la función `gbm.perf()` para seleccionar un valor "óptimo").
- `interaction.depth`: profundidad de los árboles; por defecto 1 (modelo aditivo).
- `n.minobsinnode`: número mínimo de observaciones en un nodo terminal; por defecto 10.
- `shrinkage`: parámetro de regularización λ; por defecto 0.1.
- `bag.fraction`: proporción de observaciones seleccionadas al azar para crecer cada árbol; por defecto 0.5.
- `cv.folds`: número grupos para validación cruzada; por defecto 0 (no se hace validación cruzada). Si se asigna un valor mayor que 1 se realizará validación cruzada y se devolverá el error en la componente `$cv.error` (se puede emplear para seleccionar hiperparámetros).
- `n.cores`: número de núcleos para el procesamiento en paralelo.

Como ejemplo consideraremos el conjunto de datos *winequality.RData*:

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

library(gbm)
gbm.fit <- gbm(quality ~ ., data = train)
```

```
  ## Distribution not specified, assuming gaussian ...
```

```
gbm.fit
```

```
  ## gbm(formula = quality ~ ., data = train)
  ## A gradient boosted model with gaussian loss function.
  ## 100 iterations were performed.
  ## There were 11 predictors of which 11 had non-zero influence.
```

El método `summary()` calcula las medidas de influencia de los predictores y las representa gráficamente (ver Figura 3.12):

```
summary(gbm.fit)
```

```
  ##                                     var   rel.inf
  ## alcohol                         alcohol 40.907998
  ## volatile.acidity       volatile.acidity 13.839083
  ## free.sulfur.dioxide free.sulfur.dioxide 11.488262
```

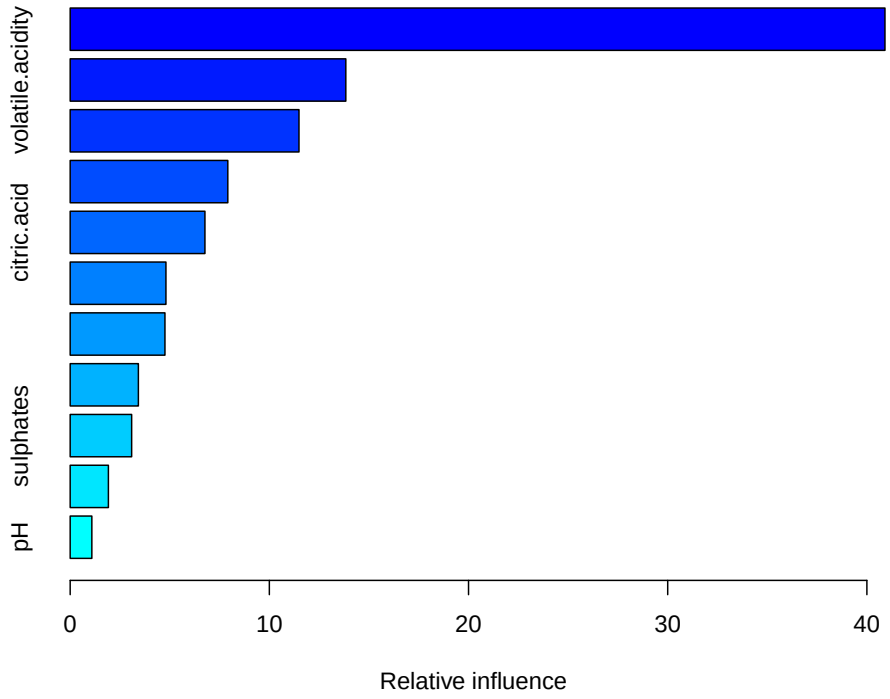


Figura 3.12: Importancia de las variables predictoras (con los valores por defecto de `gbm()`).

```
## fixed.acidity              fixed.acidity  7.914742
## citric.acid                  citric.acid  6.765875
## total.sulfur.dioxide total.sulfur.dioxide  4.808308
## residual.sugar            residual.sugar  4.758566
## chlorides                      chlorides  3.424537
## sulphates                      sulphates  3.086036
## density                          density  1.918442
## pH                                    pH  1.088152
```

Para estudiar el efecto de un predictor se pueden generar gráficos de los efectos parciales mediante el método `plot()` (ver Figura 3.13):

```
p1 <- plot(gbm.fit, i = "alcohol")
p2 <- plot(gbm.fit, i = "density")
# plot(gbm.fit, i = c("alcohol", "density")) # interacción
gridExtra::grid.arrange(p1, p2, ncol = 2)
```

Finalmente podemos evaluar la precisión en la muestra de test empleando el código habitual:

```
pred <- predict(gbm.fit, newdata = test)
obs <- test$quality

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

```
##      RMSE  Rsquared       MAE
## 0.7586208 0.3001401 0.6110442
```

```
# 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)
```

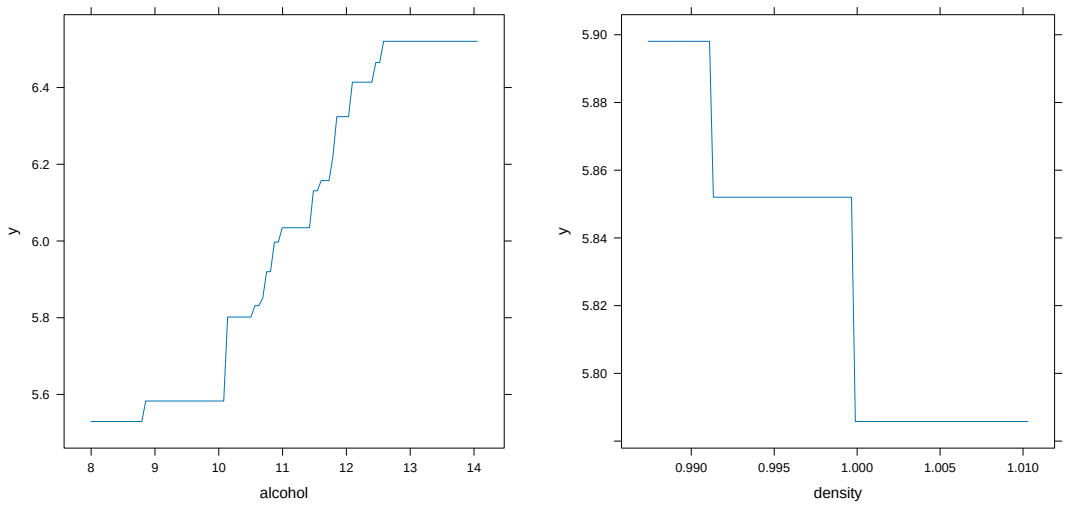


Figura 3.13: Efecto parcial del alcohol (panel izquierdo) y la densidad (panel derecho) sobre la respuesta (con `gbm()`).

```
    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, obs)
```

```
##          me        rmse         mae         mpe        mape   r.squared
## -0.01463661  0.75862081  0.61104421 -2.00702056 10.69753668  0.29917590
```

Este procedimiento también está implementado en el paquete `caret` seleccionando el método `"gbm"`, que considera como hiperparámetros:

```
library(caret)
modelLookup("gbm")
```

```
##   model         parameter                   label forReg forClass probModel
## 1   gbm           n.trees  # Boosting Iterations   TRUE     TRUE      TRUE
## 2   gbm interaction.depth          Max Tree Depth   TRUE     TRUE      TRUE
## 3   gbm         shrinkage               Shrinkage   TRUE     TRUE      TRUE
## 4   gbm    n.minobsinnode Min. Terminal Node Size   TRUE     TRUE      TRUE
```

Aunque por defecto la función `train()` solo considera nueve combinaciones de hiperparámetros. Para hacer una búsqueda más completa se podría seguir un procedimiento análogo al empleado con el método anterior:

```
set.seed(1)
caret.gbm0 <- train(quality ~ ., method = "gbm", data = train,
                    trControl = trainControl(method = "cv", number = 5))
```

```
caret.gbm0
```

```
## Stochastic Gradient Boosting
##
## 1000 samples
##   11 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 801, 800, 800, 799
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE       Rsquared   MAE
##   1                   50      0.7464098  0.2917796  0.5949686
##   1                  100      0.7258319  0.3171046  0.5751816
##   1                  150      0.7247246  0.3197241  0.5719404
##   2                   50      0.7198195  0.3307665  0.5712468
##   2                  100      0.7175006  0.3332903  0.5647409
##   2                  150      0.7258174  0.3222006  0.5713116
##   3                   50      0.7241661  0.3196365  0.5722590
##   3                  100      0.7272094  0.3191252  0.5754363
##   3                  150      0.7311429  0.3152905  0.5784988
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 100, interaction.depth =
##  2, shrinkage = 0.1 and n.minobsinnode = 10.
```

```
caret.gbm1 <- train(quality ~ ., method = "gbm", data = train,
   tuneGrid = data.frame(n.trees =  100, interaction.depth = 2,
                         shrinkage = c(0.3, 0.1, 0.05, 0.01, 0.005),
                         n.minobsinnode = 10),
   trControl = trainControl(method = "cv", number = 5))
```

```
caret.gbm1
```

```
## Stochastic Gradient Boosting
##
## 1000 samples
##   11 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 800, 801, 799, 800
## Resampling results across tuning parameters:
##
##   shrinkage  RMSE       Rsquared   MAE
##   0.005      0.8154916  0.2419131  0.6245818
##   0.010      0.7844257  0.2602989  0.6128582
##   0.050      0.7206972  0.3275463  0.5707273
##   0.100      0.7124838  0.3407642  0.5631748
##   0.300      0.7720844  0.2613835  0.6091765
##
## Tuning parameter 'n.trees' was held constant at a value of 100
## Tuning
```

```
##  parameter 'interaction.depth' was held constant at a value of 2
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 100, interaction.depth =
##  2, shrinkage = 0.1 and n.minobsinnode = 10.
```

```
varImp(caret.gbm1)
```

```
## gbm variable importance
##
##                       Overall
## alcohol              100.0000
## volatile.acidity      28.4909
## free.sulfur.dioxide   24.5158
## residual.sugar        16.8406
## fixed.acidity         12.5623
## density               10.1917
## citric.acid            9.1542
## total.sulfur.dioxide   7.2659
## chlorides              4.5106
## pH                     0.1096
## sulphates              0.0000
```

```
postResample(predict(caret.gbm1, newdata = test), test$quality)
```

```
##      RMSE  Rsquared       MAE
## 0.7403768 0.3329751 0.6017281
```

3.5.3 Ejemplo: XGBoost con el paquete `caret`

El método boosting implementado en el paquete `xgboost` es uno de los más populares hoy en día. Esta implementación proporciona parámetros adicionales de regularización para controlar la complejidad del modelo y tratar de evitar el sobreajuste. También incluye criterios de parada, para detener la evaluación del modelo cuando los árboles adicionales no ofrecen ninguna mejora. Dispone de una interfaz simple `xgboost()` y otra más avanzada `xgb.train()`, que admite funciones de pérdida y evaluación personalizadas. Normalmente es necesario un preprocesado de los datos antes de llamar a estas funciones, ya que requieren de una matriz para los predictores y de un vector para la respuesta (además en el caso de que sea dicotómica debe tomar valores en $\{0, 1\}$). Por tanto es necesario recodificar las variables categóricas como numéricas. Por este motivo puede ser preferible emplear la interfaz de `caret`.

El algoritmo estándar *XGBoost*, que emplea árboles como modelo base, está implementado en el método `"xgbTree"` de `caret`[4]:

```
library(caret)
# names(getModelInfo("xgb"))
modelLookup("xgbTree")
```

```
##     model        parameter                          label forReg forClass
## 1 xgbTree          nrounds          # Boosting Iterations   TRUE     TRUE
## 2 xgbTree        max_depth                 Max Tree Depth   TRUE     TRUE
## 3 xgbTree              eta                     Shrinkage   TRUE     TRUE
## 4 xgbTree            gamma         Minimum Loss Reduction   TRUE     TRUE
## 5 xgbTree colsample_bytree     Subsample Ratio of Columns   TRUE     TRUE
## 6 xgbTree min_child_weight Minimum Sum of Instance Weight   TRUE     TRUE
```

[4]Otras alternativas son: `"xgbDART"` que también emplean árboles como modelo base, pero incluye el método DART (Vinayak y Gilad-Bachrach, 2015) para evitar sobreajuste (básicamente descarta árboles al azar en la secuencia), y `"xgbLinear"` que emplea modelos lineales.

```
## 7 xgbTree        subsample           Subsample Percentage   TRUE     TRUE
##   probModel
## 1      TRUE
## 2      TRUE
## 3      TRUE
## 4      TRUE
## 5      TRUE
## 6      TRUE
## 7      TRUE
```

Este método considera los siguientes hiperparámetros:

- `"nrounds"`: número de iteraciones boosting.
- `"max_depth"`: profundidad máxima del árbol; por defecto 6.
- `"eta"`: parámetro de regularización λ; por defecto 0.3.
- `"gamma"`: mínima reducción de la pérdida para hacer una partición adicional en un nodo del árbol; por defecto 0.
- `"colsample_bytree"`: proporción de predictores seleccionados al azar para crecer cada árbol; por defecto 1.
- `"min_child_weight"`: suma mínima de peso (hessiana) para hacer una partición adicional en un nodo del árbol; por defecto 1.
- `"subsample"`: proporción de observaciones seleccionadas al azar en cada iteración boosting; por defecto 1.

Para más información sobre parámetros adicionales se puede consultar la ayuda de `xgboost::xgboost()` o la lista detallada disponible en la Sección XGBoost Parameters del Manual de XGBoost.

Como ejemplo consideraremos el problema de clasificación empleando el conjunto de datos de calidad de vino:

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

En este caso la función `train()` considera por defecto 108 combinaciones de hiperparámetros y el tiempo de computación puede ser excesivo[5]:

```
caret.xgb <- train(taste ~ ., method = "xgbTree", data = train,
                   trControl = trainControl(method = "cv", number = 5),
                   verbosity = 0)
caret.xgb
```

```
## eXtreme Gradient Boosting
##
## 1000 samples
##   11 predictor
##    2 classes: 'good', 'bad'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 799, 801, 801, 799, 800
```

[5]Además, se establece `verbosity = 0` para evitar (cientos de) warnings: `WARNING: src/c_api/c_api.cc:935: "ntree_limit" is deprecated, use "iteration_range" instead.`

```
## Resampling results across tuning parameters:
##
##   eta  max_depth  colsample_bytree  subsample  nrounds  Accuracy   Kappa
##   0.3  1          0.6               0.50        50      0.7479499  0.3997718
##   0.3  1          0.6               0.50       100      0.7509649  0.4226367
##   0.3  1          0.6               0.50       150      0.7480199  0.4142399
##   0.3  1          0.6               0.75        50      0.7389498  0.3775707
##   0.3  1          0.6               0.75       100      0.7499600  0.4178857
##   0.3  1          0.6               0.75       150      0.7519900  0.4194354
##   0.3  1          0.6               1.00        50      0.7479450  0.3933223
##   0.3  1          0.6               1.00       100      0.7439499  0.3946755
##  [ reached getOption("max.print") -- omitted 100 rows ]
##
## Tuning parameter 'gamma' was held constant at a value of 0
## Tuning
##  parameter 'min_child_weight' was held constant at a value of 1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were nrounds = 100, max_depth = 2, eta
##  = 0.4, gamma = 0, colsample_bytree = 0.8, min_child_weight = 1 and subsample
##  = 1.
```

```
caret.xgb$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 89     100         2 0.4     0              0.8                1         1
```

```
varImp(caret.xgb)
```

```
## xgbTree variable importance
##
##                      Overall
## alcohol              100.000
## volatile.acidity      27.693
## citric.acid           23.788
## free.sulfur.dioxide   23.673
## fixed.acidity         20.393
## residual.sugar        15.734
## density               10.956
## chlorides              8.085
## sulphates              3.598
## pH                     2.925
## total.sulfur.dioxide   0.000
```

```
confusionMatrix(predict(caret.xgb, newdata = test), test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  147  46
##       bad    19  38
##
##                Accuracy : 0.74
##                  95% CI : (0.681, 0.7932)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.005841
##
##                   Kappa : 0.3671
##
```

```
##  Mcnemar's Test P-Value : 0.001260
##
##             Sensitivity : 0.8855
##             Specificity : 0.4524
##          Pos Pred Value : 0.7617
##          Neg Pred Value : 0.6667
##              Prevalence : 0.6640
##          Detection Rate : 0.5880
##    Detection Prevalence : 0.7720
##       Balanced Accuracy : 0.6690
##
##        'Positive' Class : good
##
```

Se podría seguir una estrategia de búsqueda similar a la empleada en los métodos anteriores.

Capítulo 4

Máquinas de soporte vectorial

Las máquinas de soporte vectorial (*support vector machines*, SVM) son métodos estadísticos que Vladimir Vapnik empezó a desarrollar a mediados de 1960, inicialmente para problemas de clasificación binaria (problemas de clasificación con dos categorias), basados en la idea de separar los datos mediante hiperplanos. Actualmente existen extensiones dentro de esta metodología para clasificación con más de dos categorías, para regresión y para detección de datos atípicos. El nombre proviene de la utilización de vectores que hacen de soporte para maximizar la separación entre los datos y el hiperplano.

La popularidad de las máquinas de soporte vectorial creció a partir de los años 90 cuando los incorpora la comunidad informática. Se considera una metodología muy flexible y con buen rendimiento en un amplio abanico de situaciones, aunque por lo general no es la que consigue los mejores rendimientos. Dos referencias ya clásicas son Vapnik (2000) y Vapnik (2013).

Siguiendo a James et al. (2021) distinguiremos en nuestra exposición entre clasificadores de máximo margen (*maximal margin classifiers*), clasificadores de soporte vectorial (*support vector classifiers*) y máquinas de soporte vectorial (*support vector machines*).

4.1 Clasificadores de máximo margen

Los clasificadores de máximo margen (*maximal margin classifiers*; también denominados *hard margin classifiers*) son un método de clasificación binaria que se utiliza cuando hay una frontera lineal que separa perfectamente los datos de entrenamiento de una categoría de los de la otra. Por conveniencia, etiquetamos las dos categorías como +1/-1, es decir, los valores de la variable respuesta $Y \in \{-1, 1\}$. Y suponemos que existe un hiperplano

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p = 0,$$

donde p es el número de variables predictoras, que tiene la propiedad de separar los datos de entrenamiento según la categoría a la que pertenecen, es decir,

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi}) > 0$$

para todo $i = 1, 2, ..., n$, siendo n el número de datos de entrenamiento.

Una vez tenemos el hiperplano, clasificar una nueva observación $\mathbf{x}$ se reduce a calcular el signo de

$$m(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$$

Si el signo es positivo, se clasifica como perteneciente a la categoría +1, y si es negativo a la categoría -1. Además, el valor absoluto de $m(\mathbf{x})$ nos da una idea de la distancia entre la observación y la frontera que define el hiperplano. En concreto

$$\frac{y_i}{\sqrt{\sum_{j=1}^{p} \beta_j^2}}(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi})$$

sería la distancia de la observación i-ésima al hiperplano. Por supuesto, aunque clasifique los datos de entrenamiento sin error, no hay ninguna garantía de que clasifique bien nuevas observaciones, por ejemplo los datos de test. De hecho, si p es grande es fácil que haya un sobreajuste.

Realmente, si existe al menos un hiperplano que separa perfectamente los datos de entrenamiento de las dos categorías, entonces va a haber infinitos. El objetivo es seleccionar un hiperplano que separe los datos lo mejor posible. Para ello, dado un hiperplano de separación, se calculan sus distancias a todos los datos de entrenamiento y se define el *margen* como la menor de esas distancias. El método *maximal margin classifier* lo que hace es seleccionar, de los infinitos hiperplanos, aquel que tiene el mayor margen. Fijémonos en que siempre va a haber varias observaciones que equidistan del hiperplano de máximo margen, y cuya distancia es precisamente el margen. Esas observaciones reciben el nombre de *vectores soporte* (podemos obtener el hiperplano a partir de ellas) y son las que dan nombre a esta metodología (ver Figura 4.1).

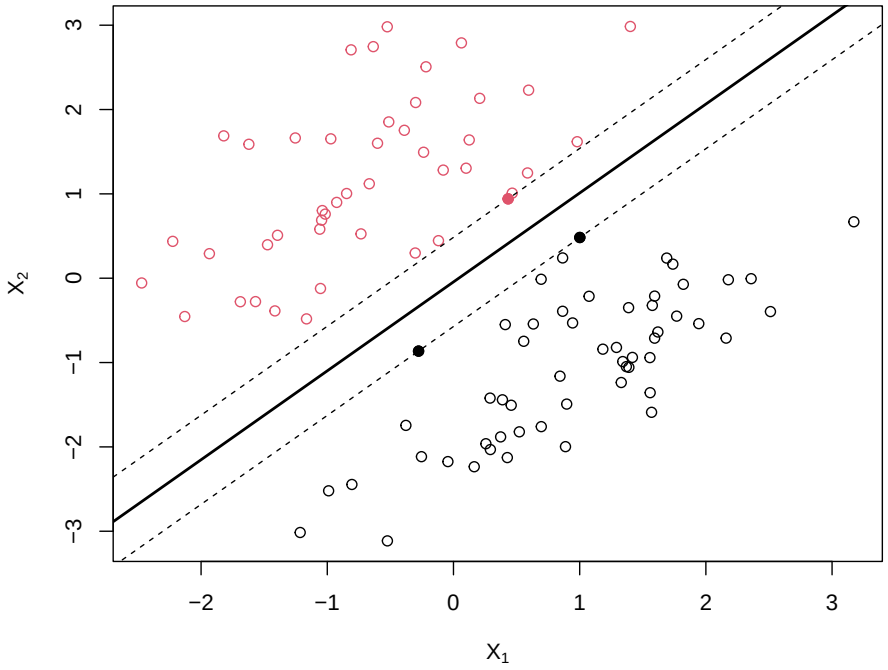


Figura 4.1: Ilustración de un clasificador de máximo margen con dos predictores (con datos simulados; los puntos se corresponden con las observaciones y el color, negro o rojo, con la clase). La línea sólida es el hiperplano de máximo margen y los puntos sólidos los vectores de soporte (las líneas discontinuas se corresponden con la máxima distancia del hiperplano a las observaciones).

Matemáticamente, dadas las n observaciones de entrenamiento $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, el clasificador de máximo margen es la solución del problema de optimización

$$max_{\beta_0,\beta_1,\ldots,\beta_p} M$$

sujeto a

$$\sum_{j=1}^{p} \beta_j^2 = 1$$

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \ldots + \beta_p x_{pi}) \geq M \quad \forall i$$

Si, como estamos suponiendo en esta sección, los datos de entrenamiento son perfectamente separables mediante un hiperplano, entonces el problema anterior va a tener solución con $M > 0$, y M va a ser el margen.

Una forma equivalente (y mas conveniente) de formular el problema anterior, utilizando $M = 1/\|\beta\|$ con $\beta = (\beta_1, \beta_2, \ldots, \beta_p)$, es

$$\min_{\beta_0,\beta} \|\beta\|$$

sujeto a

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi}) \geq 1 \quad \forall i$$

El problema anterior de optimización es convexo (función objetivo cuadrática con restricciones lineales).

Hay una característica de este método que es de destacar: así como en otros métodos, si se modifica cualquiera de los datos se modifica también el modelo, en este caso el modelo solo depende de los (pocos) datos que son vector soporte, y la modificación de cualquier otro dato no afecta a la construcción del modelo (siempre que, al *moverse* el dato, no cambie el margen).

4.2 Clasificadores de soporte vectorial

Los clasificadores de soporte vectorial (*support vector classifiers*; también denominados *soft margin classifiers*) fueron introducidos en Cortes y Vapnik (1995). Son una extensión del problema anterior que se utiliza cuando se desea clasificar mediante un hiperplano pero no existe ninguno que separe perfectamente los datos de entrenamiento según su categoría. En este caso no queda más remedio que admitir errores en la clasificación de algunos datos de entrenamiento (como hemos visto que pasa con todas las metodologías), que van a estar en el lado equivocado del hiperplano. Y en lugar de hablar de un margen se habla de un margen débil (*soft margin*).

Este enfoque, consistente en aceptar que algunos datos de entrenamiento van a estar mal clasificados, puede ser preferible aunque exista un hiperplano que resuelva el problema de la sección anterior, ya que los clasificadores de soporte vectorial son más robustos que los clasificadores de máximo margen.

Veamos la formulación matemática del problema:

$$\max_{\beta_0,\beta_1,...,\beta_p,\epsilon_1,...,\epsilon_n} M$$

sujeto a

$$\sum_{j=1}^{p} \beta_j^2 = 1$$

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi}) \geq M(1 - \epsilon_i) \quad \forall i$$

$$\sum_{i=1}^{n} \epsilon_i \leq K$$

$$\epsilon_i \geq 0 \quad \forall i$$

Las variables ϵ_i son las variables de holgura (*slack variables*). Quizás resultase más intuitivo introducir las holguras en términos absolutos, como $M - \epsilon_i$, pero eso daría lugar a un problema no convexo, mientras que escribiendo la restricción en términos relativos como $M(1 - \epsilon_i)$ el problema pasa a ser convexo. Pero en esta formulación el elemento clave es la introducción del hiperparámetro K, necesariamente no negativo, que se puede interpretar como la tolerancia al error. De hecho, es fácil ver que no puede haber más de K datos de entrenamiento incorrectamente clasificados, ya que si un dato está mal clasificado entonces $\epsilon_i > 1$. En el caso extremo de utilizar $K = 0$, estaríamos en el caso de un *hard margin classifier*. La elección del valor de K también se puede interpretar como una penalización por la complejidad del modelo, y por tanto en términos del balance entre el sesgo y la varianza: valores pequeños van a dar lugar a modelos muy complejos, con mucha varianza y poco sesgo (con el consiguiente riesgo de sobreajuste); y valores grandes a modelos con mucho sesgo y poca varianza. El hiperparámetro K se puede seleccionar de modo óptimo por los procedimientos ya conocidos, tipo bootstrap o validación cruzada.

Una forma equivalente de formular el problema (cuadrático con restricciones lineales) es

$$\min_{\beta_0,\beta} \|\beta\|$$

sujeto a

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi}) \geq 1 - \epsilon_i \quad \forall i$$

$$\sum_{i=1}^{n} \epsilon_i \leq K$$
$$\epsilon_i \geq 0 \quad \forall i$$

En la práctica, por una conveniencia de cálculo, se utiliza la siguiente formulación, también equivalente,

$$\min_{\beta_0,\beta} \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{n} \epsilon_i$$

sujeto a

$$y_i(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_p x_{pi}) \geq 1 - \epsilon_i \quad \forall i$$
$$\epsilon_i \geq 0 \quad \forall i$$

Aunque el problema a resolver es el mismo, y por tanto también la solución, hay que tener cuidado con la interpretación, pues el hiperparámetro K se ha sustituido por C. Este nuevo parámetro es el que nos vamos a encontrar en los ejercicios prácticos y tiene una interpretación inversa a K. El parámetro C es la penalización por mala clasificación (coste que supone que un dato de entrenamiento esté mal clasificado), y por tanto el *hard margin classifier* se obtiene para valores muy grandes ($C = \infty$ se corresponde con $K = 0$), véase la Figura 4.2. Esto es algo confuso, ya que no se corresponde con la interpretación habitual de *penalización por complejidad*.

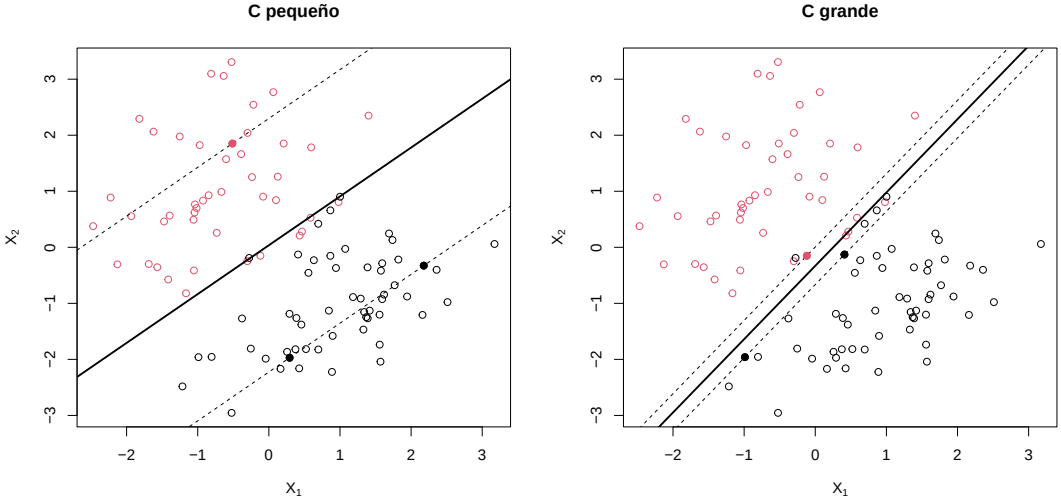


Figura 4.2: Ejemplo de clasificadores de soporte vectorial (margen débil), con parámetro de coste "pequeño" (izquierda) y "grande" (derecha).

En este contexto, los vectores soporte van a ser no solo los datos de entrenamiento que están (correctamente clasificados) a una distancia M del hiperplano, sino también aquellos que están incorrectamente clasificados e incluso los que están a una distancia inferior a M. Como se comentó en la sección anterior, estos son los datos que definen el modelo, que es por tanto robusto a las observaciones que están lejos del hiperplano.

Aunque no vamos a entrar en detalles sobre como se obtiene la solución del problema de optimización, sí resulta interesante destacar que el clasificador de soporte vectorial

$$m(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$$

puede representarse como

$$m(\mathbf{x}) = \beta_0 + \sum_{i=1}^{n} \alpha_i \mathbf{x}^t \mathbf{x}_i$$

donde $\mathbf{x}^t\mathbf{x}_i$ es el producto escalar entre el vector $\mathbf{x}$ del dato a clasificar y el vector $\mathbf{x}_i$ del dato de entrenamiento i-ésimo. Asimismo, los coeficientes $\beta_0, \alpha_1, ..., \alpha_n$ se obtienen (exclusivamente) a partir

de los productos escalares $\mathbf{x}_i^t\mathbf{x}_j$ de los distintos pares de datos de entrenamiento y de las respuestas y_i. Y más aún, el sumatorio anterior se puede reducir a los índices que corresponden a vectores soporte ($i \in S$), al ser los demás coeficientes nulos:

$$m(\mathbf{x}) = \beta_0 + \sum_{i \in S} \alpha_i \mathbf{x}^t \mathbf{x}_i$$

4.3 Máquinas de soporte vectorial

De la misma manera que en el Capítulo 2 dedicado a árboles se comentó que estos serán efectivos en la medida en la que los datos se separen adecuadamente utilizando particiones basadas en rectángulos, los dos métodos de clasificación que hemos visto hasta ahora serán efectivos si hay una frontera lineal que separe los datos de las dos categorías. En caso contrario, un clasificador de soporte vectorial resultará inadecuado. Una solución natural es sustituir el hiperplano, lineal en esencia, por otra función que dependa de las variables predictoras $X_1, X_2, \ldots, X_n$, utilizando por ejemplo una expresión polinómica o incluso una expresión que no sea aditiva en los predictores. Pero esta solución puede resultar muy compleja computacionalmente.

En Boser *et al.* (1992) se propuso sustituir, en todos los cálculos que conducen a la expresión

$$m(\mathbf{x}) = \beta_0 + \sum_{i \in S} \alpha_i \mathbf{x}^t \mathbf{x}_i$$

los productos escalares $\mathbf{x}^t\mathbf{x}_i$, $\mathbf{x}_i^t\mathbf{x}_j$ por funciones alternativas de los datos que reciben el nombre de funciones *kernel*, obteniendo la máquina de soporte vectorial

$$m(\mathbf{x}) = \beta_0 + \sum_{i \in S} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$$

Algunas de las funciones kernel más utilizadas son:

- Kernel lineal
$$K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^t\mathbf{y}$$
- Kernel polinómico
$$K(\mathbf{x}, \mathbf{y}) = (1 + \gamma\mathbf{x}^t\mathbf{y})^d$$
- Kernel radial
$$K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma\|\mathbf{x} - \mathbf{y}\|^2)$$
- Tangente hiperbólica
$$K(\mathbf{x}, \mathbf{y}) = \tanh(1 + \gamma\mathbf{x}^t\mathbf{y})$$

Antes de construir el modelo, es recomendable centrar y reescalar los datos para evitar que los valores grandes *ahoguen* al resto de los datos. Por supuesto, tiene que hacerse la misma transformación a todos los datos, incluidos los datos de test. La posibilidad de utilizar distintos kernels da mucha flexibilidad a esta metodología, pero es muy importante seleccionar adecuadamente los parámetros de la función kernel (γ, d) y el parámetro C para evitar sobreajustes como se puede ver en la Figura 4.3.

La metodología *support vector machine* está específicamente diseñada para clasificar cuando hay exactamente dos categorías. En la literatura se pueden encontrar varias propuestas para extenderla al caso de más de dos categorías, aunque las dos más populares son las comentadas en la Sección 1.2.1: "uno contra todos" (*One-vs-Rest*, OVR) y "uno contra uno" (*One-vs-One*, OVO)[1].

[1]Esta última es la que implementa la función `kernlab::ksvm()`, empleada como ejemplo en la Sección 4.4.

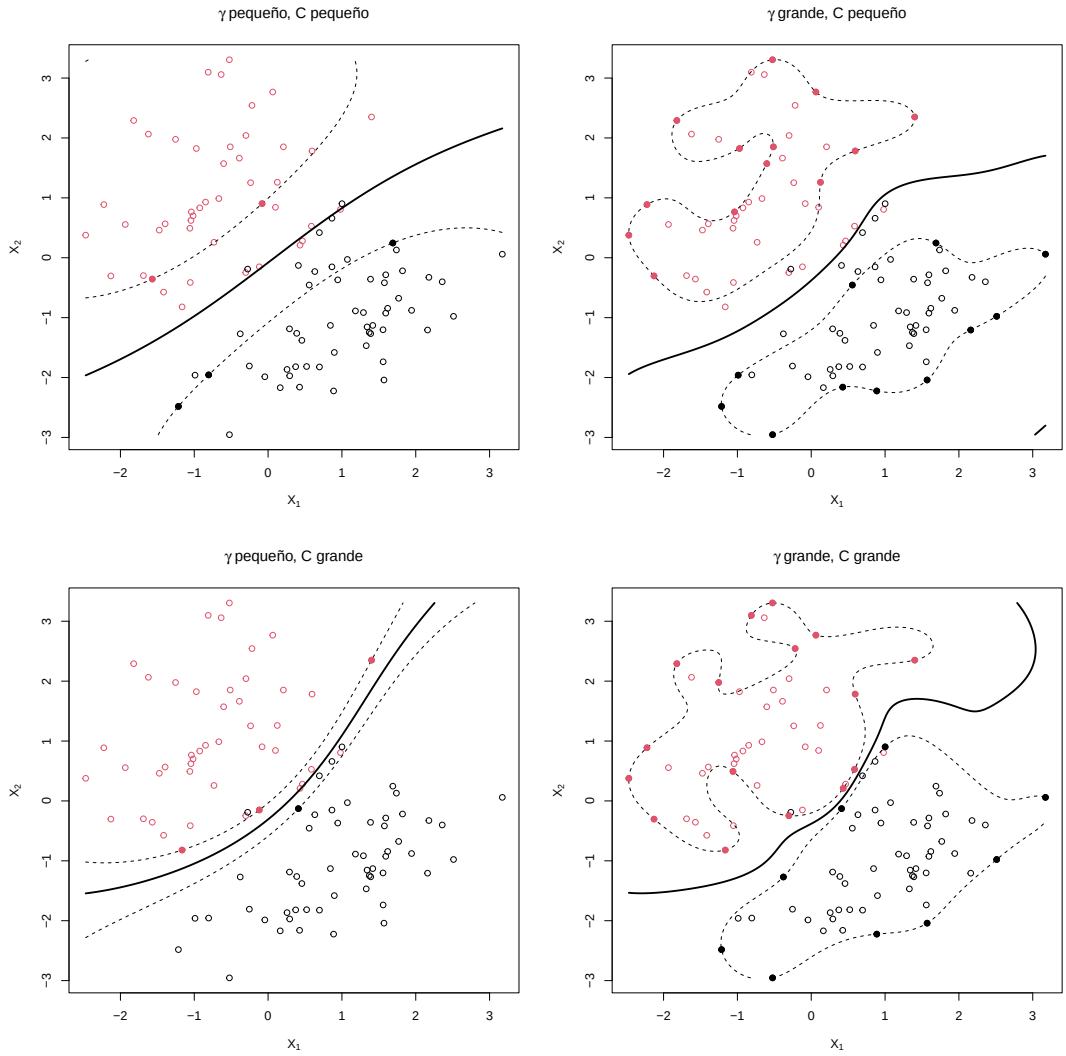


Figura 4.3: Ejemplos de máquinas de soporte vectorial con diferentes valores de los hiperparámetros (γ inverso de la ventana de la función kernel y coste C).

4.3.1 Regresión con SVM

Aunque la metodología SVM está concebida para problemas de clasificación, ha habido varios intentos de adaptar su filosofía a problemas de regresión. En esta sección vamos a comentar muy por encima el enfoque seguido en Drucker et al. (1997), con un fuerte enfoque en la robustez. Recordemos que, en el contexto de la clasificación, el modelo SVM va a depender de unos pocos datos: los vectores soporte. En regresión, si se utiliza RSS como criterio de error, todos los datos van a influir en el modelo y además, al estar los errores al cuadrado, los valores atípicos van a tener mucha influencia, muy superior a la que se tendría si se utilizase, por ejemplo, el valor absoluto. Una alternativa, poco intuitiva pero efectiva, es fijar los hiperparámetros $\epsilon, c > 0$ como umbral y coste, respectivamente, y definir la función de pérdidas

$$L_{\epsilon,c}(x) = \begin{cases} 0 & \text{si } |x| < \epsilon \\ (|x| - \epsilon)c & \text{en otro caso} \end{cases}$$

En un problema de regresión lineal, SVM estima los parámetros del modelo

$$m(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$$

minimizando

$$\sum_{i=1}^{n} L_{\epsilon,c}(y_i - \hat{y}_i) + \sum_{j=1}^{p} \beta_j^2$$

Para hacer las cosas aún más confusas, hay autores que utilizan una formulación, equivalente, en la que el parámetro aparece en el segundo sumando como $\lambda = 1/c$. En la práctica, es habitual fijar el valor de ϵ y seleccionar el valor de c (equivalentemente, λ) por validación cruzada, por ejemplo.

El modelo puede escribirse en función de los vectores soporte, que son aquellas observaciones cuyo residuo excede el umbral ϵ:

$$m(\mathbf{x}) = \beta_0 + \sum_{i \in S} \alpha_i \mathbf{x}^t \mathbf{x}_i$$

Finalmente, utilizando una función kernel, el modelo de regresión SVM es

$$m(\mathbf{x}) = \beta_0 + \sum_{i \in S} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$$

4.3.2 Ventajas e incovenientes

Ventajas:

- Son muy flexibles (pueden adaptarse a fronteras no lineales complejas), por lo que en muchos casos se obtienen buenas predicciones (en otros pueden producir malos resultados).
- Al suavizar el margen, utilizando un parámetro de coste C, son relativamente robustas frente a valores atípicos.

Inconvenientes:

- Los modelos ajustados son difíciles de interpretar (caja negra), habrá que recurrir a herramientas generales como las descritas en la Sección 1.5.
- Pueden requerir mucho tiempo de computación cuando $n >> p$, ya que hay que estimar (en principio) tantos parámetros como número de observaciones en los datos de entrenamiento, aunque finalmente la mayoría de ellos se anularán (en cualquier caso habría que factorizar la matriz $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ de dimensión $n \times n$).
- Están diseñados para predictores numéricos (emplean distancias), por lo que habrá que realizar un preprocesado de las variables explicativas categóricas (para transformarlas en variables indicadoras).

4.4 SVM con el paquete `kernlab`

Hay varios paquetes que implementan este procedimiento (e.g. `e1071`, Meyer et al., 2020; `svmpath`, ver T. Hastie et al., 2004), aunque se considera que el más completo es `kernlab` (Karatzoglou et al., 2004).

La función principal es `ksvm()` y se suelen considerar los siguientes argumentos:

```
ksvm(formula, data, scaled = TRUE, type, kernel ="rbfdot", kpar = "automatic",
     C = 1, epsilon = 0.1, prob.model = FALSE, class.weights, cross = 0)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (e.g. `respuesta ~ .`; también admite matrices).
- `scaled`: vector lógico indicando que predictores serán reescalados; por defecto se reescalan todas las variables no binarias (y se almacenan los valores empleados para ser usados en posteriores predicciones).
- `type` (opcional): cadena de texto que permite seleccionar los distintos métodos de clasificación, de regresión o de detección de atípicos implementados (ver `?ksvm`); por defecto se establece a partir del tipo de la respuesta: `"C-svc"`, clasificación con parámetro de coste, si es un factor y `"eps-svr"`, regresión épsilon, si la respuesta es numérica.
- `kernel`: función núcleo. Puede ser una función definida por el usuario o una cadena de texto que especifique una de las implementadas en el paquete (ver `?kernels`); por defecto `"rbfdot"`, kernel radial gausiano.

- `kpar`: lista con los hiperparámetros del núcleo. En el caso de `"rbfdot"`, además de una lista con un único componente `"sigma"` (inversa de la ventana), puede ser `"automatic"` (valor por defecto) e internamente emplea la función `sigest()` para seleccionar un valor "adecuado".
- `C`: (hiper)parámetro C que especifica el coste de la violación de las restricciones; por defecto 1.
- `epsilon`: (hiper)parámetro ϵ empleado en la función de pérdidas de los métodos de regresión; por defecto 0.1.
- `prob.model`: si se establece a `TRUE` (por defecto es `FALSE`), se emplean los resultados de la clasificación para ajustar un modelo para estimar las probabilidades (y se podrán calcular con el método `predict()`).
- `class.weights`: vector (con las clases como nombres) con los pesos de una mala clasificación en cada clase.
- `cross`: número grupos para validación cruzada; por defecto 0 (no se hace validación cruzada). Si se asigna un valor mayor que 1 se realizará validación cruzada y se devolverá el error en la componente `@cross` (se puede acceder con la función `cross()`; y se puede emplear para seleccionar hiperparámetros).

Como ejemplo consideraremos el problema de clasificación con los datos de calidad de vino:

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


library(kernlab)
set.seed(1)
# Selección de sigma = mean(sigest(taste ~ ., data = train)[-2])
# (depende de la semilla)
svm <- ksvm(taste ~ ., data = train,
            kernel = "rbfdot", prob.model = TRUE)
svm
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0751133799772488
##
## Number of Support Vectors : 594
##
## Objective Function Value : -494.1409
## Training error : 0.198
## Probability model included.
```

```
# plot(svm, data = train) produce un error # packageVersion("kernlab") '0.9.29'
```

Podemos evaluar la precisión en la muestra de test empleando el procedimiento habitual:

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  147  45
##       bad    19  39
##
##                Accuracy : 0.744
##                  95% CI : (0.6852, 0.7969)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.003886
##
##                   Kappa : 0.3788
##
##  Mcnemar's Test P-Value : 0.001778
##
##             Sensitivity : 0.8855
##             Specificity : 0.4643
##          Pos Pred Value : 0.7656
##          Neg Pred Value : 0.6724
##              Prevalence : 0.6640
##          Detection Rate : 0.5880
##    Detection Prevalence : 0.7680
##       Balanced Accuracy : 0.6749
##
##        'Positive' Class : good
##
```

Para obtener las estimaciones de las probabilidades, habría que establecer `type = "probabilities"` al predecir (devolverá una matriz con columnas correspondientes a los niveles)[2]:

```
p.est <- predict(svm, newdata = test, type = "probabilities")
head(p.est)
```

```
##           good       bad
## [1,] 0.4761934 0.5238066
## [2,] 0.7089338 0.2910662
## [3,] 0.8893454 0.1106546
## [4,] 0.8424003 0.1575997
## [5,] 0.6640875 0.3359125
## [6,] 0.3605543 0.6394457
```

Este procedimiento está implementado en el método `"svmRadial"` de `caret` y considera como hiperparámetros:

```
library(caret)
# names(getModelInfo("svm")) # 17 métodos
modelLookup("svmRadial")
```

```
##       model parameter label forReg forClass probModel
## 1 svmRadial     sigma Sigma   TRUE     TRUE      TRUE
## 2 svmRadial         C  Cost   TRUE     TRUE      TRUE
```

En este caso la función `train()` por defecto evaluará únicamente tres valores del hiperparámetro `C = c(0.25, 0.5, 1)` y fijará el valor de `sigma`. Alternativamente podríamos establecer la rejilla de búsqueda, por ejemplo:

[2] Otras opciones son `"votes"` y `"decision"` para obtener matrices con el número de votos o los valores de $m(\mathbf{x})$.

```
tuneGrid <- data.frame(sigma = kernelf(svm)@kpar$sigma, # Emplea clases S4
                       C = c(0.5, 1, 5))
set.seed(1)
caret.svm <- train(taste ~ ., data = train, method = "svmRadial",
                   preProcess = c("center", "scale"),
                   trControl = trainControl(method = "cv", number = 5),
                   tuneGrid = tuneGrid, prob.model = TRUE)
caret.svm
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 1000 samples
##   11 predictor
##    2 classes: 'good', 'bad'
##
## Pre-processing: centered (11), scaled (11)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 801, 800, 800, 799
## Resampling results across tuning parameters:
##
##   C    Accuracy   Kappa
##   0.5  0.7549524  0.4205204
##   1.0  0.7599324  0.4297468
##   5.0  0.7549374  0.4192217
##
## Tuning parameter 'sigma' was held constant at a value of 0.07511338
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.07511338 and C = 1.
```

```
varImp(caret.svm)
```

```
## ROC curve variable importance
##
##                       Importance
## alcohol                  100.000
## density                   73.616
## chlorides                 60.766
## volatile.acidity          57.076
## total.sulfur.dioxide      45.500
## fixed.acidity             42.606
## pH                        34.972
## sulphates                 25.546
## citric.acid                6.777
## residual.sugar             6.317
## free.sulfur.dioxide        0.000
```

```
confusionMatrix(predict(caret.svm, newdata = test), test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  147  45
##       bad    19  39
##
##                Accuracy : 0.744
##                  95% CI : (0.6852, 0.7969)
##     No Information Rate : 0.664
```

```
##     P-Value [Acc > NIR] : 0.003886
##
##                   Kappa : 0.3788
##
##  Mcnemar's Test P-Value : 0.001778
##
##             Sensitivity : 0.8855
##             Specificity : 0.4643
##          Pos Pred Value : 0.7656
##          Neg Pred Value : 0.6724
##              Prevalence : 0.6640
##          Detection Rate : 0.5880
##    Detection Prevalence : 0.7680
##       Balanced Accuracy : 0.6749
##
##        'Positive' Class : good
##
```

Capítulo 5

Otros métodos de clasificación

En los métodos de clasificación que hemos visto en los capítulos anteriores, uno de los objetivos era estimar la probabilidad a posteriori $P(Y = k|\mathbf{X} = \mathbf{x})$ de que la observación $\mathbf{x}$ pertenezca a la categoría k, pero en ningún caso nos preocupábamos por la distribución de las variables predictoras. En la terminología de ML estos métodos se conocen con el nombre de discriminadores (*discriminative methods*). Otro ejemplo de método discriminador es la regresión logística.

En este capítulo vamos a ver métodos que reciben el nombre genérico de métodos generadores (*generative methods*). Se caracterizan porque calculan las probabilidades a posteriori utilizando la distribución conjunta de $(\mathbf{X}, Y)$ y el teorema de Bayes:

$$P(Y = k|\mathbf{X} = \mathbf{x}) = \frac{P(Y = k)f_k(\mathbf{x})}{\sum_{l=1}^{K} P(Y = l)f_l(\mathbf{x})}$$

donde $f_k(\mathbf{x})$ es la función de densidad del vector aleatorio $\mathbf{X} = (X_1, X_2, \dots, X_p)$ para una observación perteneciente a la clase k, es decir, es una forma abreviada de escribir $f(\mathbf{X} = \mathbf{x}|Y = k)$. En la jerga bayesiana a esta función se la conoce como *verosimilitud* (es la función de verosimilitud sin más que considerar que la observación muestral $\mathbf{x}$ es fija y la variable es k) y resumen la fórmula anterior como

$$posterior \propto prior \times verosimilitud$$

Una vez estimadas las probabilidades a priori $P(Y = k)$ y las densidades (verosimilitudes) $f_k(\mathbf{x})$, tenemos las probabilidades a posteriori. Para estimar las funciones de densidad se puede utilizar un método paramétrico o un método no paramétrico. En el primer caso, lo más habitual es modelizar la distribución del vector de variables predictoras como normales multivariantes.

A continuación vamos a ver tres casos particulares de este enfoque, siempre suponiendo normalidad.

5.1 Análisis discriminate lineal

El análisis lineal discrimintante (LDA) se inicia en Fisher (1936) pero es Welch (1939) quien lo enfoca utilizando el teorema de Bayes. Asumiendo que $X|Y = k \sim N(\mu_k, \Sigma)$, es decir, que todas las categorías comparten la misma matriz Σ, se obtienen las funciones discriminantes, lineales en $\mathbf{x}$,

$$\mathbf{x}^t\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^t\Sigma^{-1}\mu_k + \log(P(Y = k))$$

La dificultad técnica del método LDA reside en el cálculo de Σ^{-1}. Cuando hay más variables predictoras que datos, o cuando las variables predictoras están fuertemente correlacionadas, hay un problema. Una solución pasa por aplicar análisis de componentes principales (PCA) para reducir la dimensión y tener predictores incorrelados antes de utilizar LDA. Aunque la solución anterior se utiliza mucho, hay que tener en cuenta que la reducción de la dimensión se lleva a cabo sin tener en cuenta la información de

las categorías, es decir, la estructura de los datos en categorías. Una alternativa consiste en utilizar *partial least squares discriminant analysis* (PLSDA, Berntsson y Wold, 1986). La idea consiste en realizar una regresión PLS siendo las categorías la respuesta, con el objetivo de reducir la dimensión a la vez que se maximiza la correlación con las respuestas.

Una generalización de LDA es el *mixture discriminant analysis* (T. Hastie y Tibshirani, 1996) en el que, siempre con la misma matriz Σ, se contempla la posibilidad de que dentro de cada categoría haya múltiples subcategorías que únicamente difieren en la media. Las distribuciones dentro de cada clase se agregan mediante una mixtura de las distribuciones multivariantes.

A continuación se muestra un ejemplo de análisis discriminante lineal empleando la función `MASS::lda()`, considerando como respuesta la variable `taste` del conjunto de datos `wintaste`.

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

library(MASS)
ld <- lda(taste ~ ., data = train)
ld
```

```
## Call:
## lda(taste ~ ., data = train)
##
## Prior probabilities of groups:
##  good   bad
## 0.662 0.338
##
## Group means:
##      fixed.acidity volatile.acidity citric.acid residual.sugar  chlorides
## good      6.726888        0.2616994   0.3330211       6.162009 0.04420242
## bad       7.030030        0.3075148   0.3251775       6.709024 0.05075740
##      free.sulfur.dioxide total.sulfur.dioxide   density       pH sulphates
## good            34.75831             132.7568 0.9935342 3.209668 0.4999396
## bad             35.41124             147.4615 0.9950789 3.166331 0.4763905
##        alcohol
## good 10.786959
## bad   9.845611
##
## Coefficients of linear discriminants:
##                                 LD1
## fixed.acidity        -4.577255e-02
## volatile.acidity      5.698858e+00
## citric.acid          -5.894231e-01
## residual.sugar       -2.838910e-01
## chlorides            -6.083210e+00
## free.sulfur.dioxide   1.039366e-03
## total.sulfur.dioxide -8.952115e-04
## density               5.642314e+02
## pH                   -2.103922e+00
## sulphates            -2.400004e+00
## alcohol              -1.996112e-01
```

En este caso, al haber solo dos categorías se construye una única función discriminante lineal. Podemos

examinar la distribución de los valores que toma esta función en la muestra de entrenamiento mediante el método `plot.lda()`:

```
plot(ld)
```

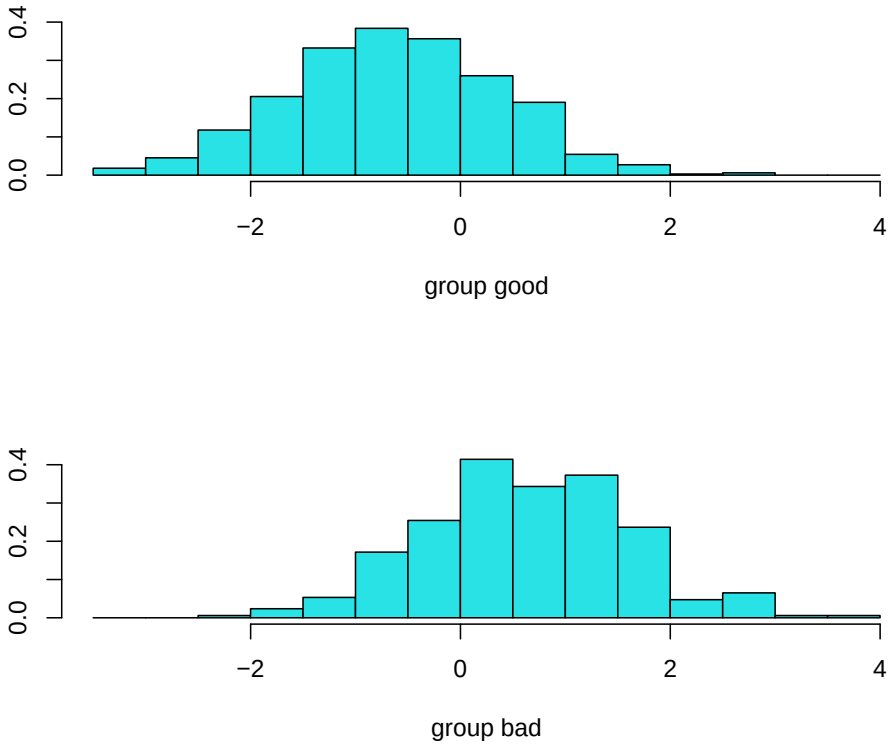


Figura 5.1: Distribución de los valores de la función discriminante lineal en cada clase.

Podemos evaluar la precisión en la muestra de test empleando la matriz de confusión:

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

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction good bad
##       good  146  49
##       bad    20  35
##
##                Accuracy : 0.724
##                  95% CI : (0.6641, 0.7785)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.0247239
##
##                   Kappa : 0.3238
##
##  Mcnemar's Test P-Value : 0.0007495
##
##             Sensitivity : 0.8795
##             Specificity : 0.4167
##          Pos Pred Value : 0.7487
##          Neg Pred Value : 0.6364
##              Prevalence : 0.6640
```

```
##          Detection Rate : 0.5840
##    Detection Prevalence : 0.7800
##       Balanced Accuracy : 0.6481
##
##        'Positive' Class : good
##
```

También podríamos examinar las probabilidades estimadas:

```
p.est <-ld.pred$posterior
```

5.2 Análisis discriminante cuadrático

El análisis discriminante cuadrático (QDA) relaja la suposición de que todas las categorías tengan la misma estructura de covarianzas, es decir, $X|Y = k \sim N(\mu_k, \Sigma_k)$, obteniendo como solución

$$-\frac{1}{2}(\mathbf{x} - \mu_k)^t \Sigma_k^{-1}(\mathbf{x} - \mu_k) - \frac{1}{2}\log(|\Sigma_k|) + \log(P(Y = k))$$

Vemos que este método da lugar a fronteras discriminantes cuadráticas.

Si el número de variables predictoras es próximo al tamaño muestral, en la prácticas QDA se vuelve impracticable, ya que el número de variables predictoras tiene que ser menor que el numero de datos en cada una de las categorías. Una recomendación básica es utilizar LDA y QDA únicamente cuando hay muchos más datos que predictores. Y al igual que en LDA, si dentro de las clases los predictores presentan mucha colinealidad el modelo va a funcionar mal.

Al ser QDA una generalización de LDA podemos pensar que siempre va a ser preferible, pero eso no es cierto, ya que QDA requiere estimar muchos más parámetros que LDA y por tanto tiene más riesgo de sobreajustar. Al ser menos flexible, LDA da lugar a modelos más simples: menos varianza pero más sesgo. LDA suele funcionar mejor que QDA cuando hay pocos datos y es por tanto muy importante reducir la varianza. Por el contrario, QDA es recomendable cuando hay muchos datos.

Una solución intermedia entre LDA y QDA es el análisis discriminante regularizado (RDA, Friedman, 1989), que utiliza el hiperparámetro λ para definir la matriz

$$\Sigma'_{k,\lambda} = \lambda\Sigma_k + (1 - \lambda)\Sigma$$

También hay una versión con dos hiperparámetros, λ y γ,

$$\Sigma'_{k,\lambda,\gamma} = (1 - \gamma)\Sigma'_{k,\lambda} + \gamma\frac{1}{p}\text{tr}(\Sigma'_{k,\lambda})I$$

De modo análogo al caso lineal, podemos realizar un análisis discriminante cuadrático empleando la función `MASS::qda()`:

```
qd <- qda(taste ~ ., data = train)
qd
```

```
## Call:
## qda(taste ~ ., data = train)
##
## Prior probabilities of groups:
##  good   bad
## 0.662 0.338
##
## Group means:
##      fixed.acidity volatile.acidity citric.acid residual.sugar  chlorides
## good      6.726888        0.2616994   0.3330211       6.162009 0.04420242
```

```
## bad       7.030030        0.3075148   0.3251775       6.709024 0.05075740
##      free.sulfur.dioxide total.sulfur.dioxide   density       pH sulphates
## good            34.75831             132.7568 0.9935342 3.209668 0.4999396
## bad             35.41124             147.4615 0.9950789 3.166331 0.4763905
##        alcohol
## good 10.786959
## bad   9.845611
```

```
qd.pred <- predict(qd, newdata = test)
pred <- qd.pred$class
# p.est <- qd.pred$posterior
caret::confusionMatrix(pred, test$taste)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  147  40
##       bad    19  44
##
##                Accuracy : 0.764
##                  95% CI : (0.7064, 0.8152)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.0003762
##
##                   Kappa : 0.4363
##
##  Mcnemar's Test P-Value : 0.0092202
##
##             Sensitivity : 0.8855
##             Specificity : 0.5238
##          Pos Pred Value : 0.7861
##          Neg Pred Value : 0.6984
##              Prevalence : 0.6640
##          Detection Rate : 0.5880
##    Detection Prevalence : 0.7480
##       Balanced Accuracy : 0.7047
##
##        'Positive' Class : good
##
```

En este caso vemos que se obtienen mejores métricas (en la muestra test) que con el discriminante lineal del ejemplo anterior.

5.3 Naive Bayes

El método *naive Bayes* (Bayes ingenuo) es una simplificación de los métodos anteriores en la que se asume que las variables explicativas son *independientes*. Esta es una suposición extremadamente fuerte y en la práctica difícilmente nos encontraremos con un problema en el que los predictores sean independientes, pero a cambio se va a reducir mucho la complejidad del modelo. Esta simplicidad del modelo le va a permitir manejar un gran número de predictores, incluso con un tamaño muestral moderado, situaciones en las que puede ser imposible utilizar LDA o QDA. Otra ventaja asociada con su simplicidad es que el cálculo de las predicciones va a poder hacer muy rápido incluso para tamaños muestrales muy grandes. Además, y quizás esto sea lo más sorprendente, en ocasiones su rendimiento es muy competitivo.

Asumiendo normalidad, este modelo no es más que un caso particular de QDA con matrices Σ_k diagonales. Cuando las variables predictoras son categóricas, lo más habitual es modelizar su distribución

utilizando distribuciones multinomiales. Siguiendo con los ejemplos anteriores, empleamos la función `e1071::naiveBayes()` para realizar la clasificación:

```
library(e1071)
nb <- naiveBayes(taste ~ ., data = train)
nb
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##  good   bad
## 0.662 0.338
##
## Conditional probabilities:
##       fixed.acidity
## Y          [,1]      [,2]
##   good 6.726888 0.8175101
##   bad  7.030030 0.9164467
##
##       volatile.acidity
## Y           [,1]       [,2]
##   good 0.2616994 0.08586935
##   bad  0.3075148 0.11015113
##
##       citric.acid
## Y           [,1]      [,2]
##   good 0.3330211 0.1231345
##   bad  0.3251775 0.1334682
##
##       residual.sugar
## Y          [,1]     [,2]
##   good 6.162009 4.945483
##   bad  6.709024 5.251402
##
##       chlorides
## Y            [,1]       [,2]
##   good 0.04420242 0.02237654
##   bad  0.05075740 0.03001672
##
##       free.sulfur.dioxide
## Y          [,1]     [,2]
##   good 34.75831 14.87336
##   bad  35.41124 19.26304
##
##       total.sulfur.dioxide
## Y          [,1]     [,2]
##   good 132.7568 38.05871
##   bad  147.4615 47.34668
##
##       density
## Y           [,1]       [,2]
##   good 0.9935342 0.00285949
```

```
##   bad  0.9950789 0.00256194
##
##       pH
## Y          [,1]      [,2]
##   good 3.209668 0.1604529
##   bad  3.166331 0.1472261
##
##       sulphates
## Y           [,1]       [,2]
##   good 0.4999396 0.11564067
##   bad  0.4763905 0.09623778
##
##       alcohol
## Y           [,1]      [,2]
##   good 10.786959 1.2298425
##   bad   9.845611 0.8710844
```

En las tablas correspondientes a los predictores[1], se muestran la media y la desviación típica de sus distribuciones condicionadas a las distintas clases.

En este caso los resultados obtenidos en la muestra de test son peores que con los métodos anteriores:

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  136  47
##       bad    30  37
##
##                Accuracy : 0.692
##                  95% CI : (0.6307, 0.7486)
##     No Information Rate : 0.664
##     P-Value [Acc > NIR] : 0.19255
##
##                   Kappa : 0.2734
##
##  Mcnemar's Test P-Value : 0.06825
##
##             Sensitivity : 0.8193
##             Specificity : 0.4405
##          Pos Pred Value : 0.7432
##          Neg Pred Value : 0.5522
##              Prevalence : 0.6640
##          Detection Rate : 0.5440
##    Detection Prevalence : 0.7320
##       Balanced Accuracy : 0.6299
##
##        'Positive' Class : good
##
```

[1] Aunque al imprimir los resultados aparece `Naive Bayes Classifier for Discrete Predictors`, se trata de un error. En este caso todos los predictores son continuos.

Capítulo 6

Modelos lineales y extensiones

En los modelo lineales se supone que la función de regresión es lineal[1]:

$$E(Y|\mathbf{X}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

Es decir, que el efecto de las variables explicativas sobre la respuesta es muy simple, proporcional a su valor, y por tanto la interpretación de este tipo de modelos es (en principio) muy fácil. El coeficiente β_j representa el incremento medio de Y al aumentar en una unidad el valor de X_j, manteniendo fijos el resto de las covariables. En este contexto las variables predictoras se denominan habitualmente variables independientes, pero en la práctica es de esperar que no haya independencia entre ellas, por lo que puede no ser muy razonable pensar que al variar una de ellas el resto va a permanecer constante.

El ajuste de este tipo de modelos en la práctica se suele realizar empleando el método de mínimos cuadrados (ordinarios), asumiendo (implícitamente o explícitamente) que la distribución condicional de la respuesta es normal, lo que se conoce como el modelo de regresión lineal múltiple (siguiente sección).

Los modelos lineales generalizados son una extensión de los modelos lineales para el caso de que la distribución condicional de la variable respuesta no sea normal (por ejemplo discreta: Bernouilli, Binomial, Poisson…). En los modelos lineales generalizados se introduce una función invertible g, denominada función enlace (o link):

$$g\left(E(Y|\mathbf{X})\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

y su ajuste en la práctica se realiza empleando el método de máxima verosimilitud.

Ambos son modelos clásicos de la inferencia estadística y, aunque pueden ser demasiado simples en muchos casos, pueden resultar muy útiles en otros por lo que también se emplean habitualmente en AE. Además, como veremos más adelante (en las secciones finales de este capítulo y en los siguientes), sirve como punto de partida para procedimientos más avanzados. En este capítulo se tratarán estos métodos desde el punto de vista de AE (descrito en el Capítulo 1), es decir, con el objetivo de predecir en lugar de realizar inferencias (y preferiblemente empleando un procedimiento automático y capaz de manejar grandes volúmenes de datos).

En consecuencia, se supondrá que se dispone de unos conocimientos básicos de los métodos clásicos de regresión lineal y regresión lineal generalizada. Para un tratamiento más completo de este tipo de métodos se puede consultar Faraway (2016), que incluye su aplicación en la práctica con R (también el Capítulo 8 de Fernández-Casal et al., 2019). Además por simplicidad, en las siguientes secciones nos centraremos principalmente en el caso de modelos lineales, pero los distintos procedimientos y comentarios se extienden de forma análoga al caso de modelos generalizados (básicamente habría que sustituir la suma de cuadrados residual por el logaritmo negativo de la verosimilitud), que serán tratados en la última sección.

[1] Algunos predictores podrían corresponderse con interacciones, $X_i = X_j X_k$, o transformaciones (e.g. $X_i = X_j^2$) de las variables explicativas originales. También se podría haber transformado la respuesta.

6.1 Regresión lineal múltiple

Como ya se comentó, el método tradicional considera el siguiente modelo:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon, \tag{6.1}$$

donde $\left(\beta_0, \beta_1, \ldots, \beta_p\right)^t$ es un vector de parámetros (desconocidos) y ε es un error aleatorio normal de media cero y varianza σ^2.

Por tanto las hipótesis estructurales del modelo son:

- Linealidad
- Homocedasticidad (varianza constante del error)
- Normalidad (y homogeneidad: ausencia de valores atípicos y/o influyentes)
- Independencia de los errores

Hipótesis adicional en regresión múltiple:

- Ninguna de las variables explicativas es combinación lineal de las demás.

En el caso de regresión múltiple es de especial interés el fenómeno de la colinealidad (o multicolinealidad) relacionado con la última de estas hipótesis (que se tratará en la Sección 6.2). Además se da por hecho que el número de observaciones disponible es como mínimo el número de parámetros, $n \geq p+1$.

El procedimiento habitual para ajustar un modelo de regresión lineal a un conjunto de datos es emplear mínimos cuadrados (ordinarios):

$$\min_{\beta_0,\beta_1,\ldots,\beta_p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta_1 x_{1i} - \cdots - \beta_p x_{pi}\right)^2$$

En R podemos emplear la función `lm()`:

```
ajuste <- lm(formula, data, subset, weights, na.action)
```

- `formula`: fórmula que especifica el modelo.
- `data`: data.frame (opcional) con las variables de la formula.
- `subset`: vector (opcional) que especifica un subconjunto de observaciones.
- `weights`: vector (opcional) de pesos (mínimos cuadrados ponderados, WLS).
- `na.action`: opción para manejar los datos faltantes; por defecto `na.omit`.

Alternativamente se puede emplear la función `biglm()` del paquete `biglm` para ajustar modelos lineales a grandes conjuntos de datos (especialmente cuando el número de observaciones es muy grande, incluyendo el caso de que los datos excedan la capacidad de memoria del equipo). También se podría utilizar la función `rlm()` del paquete `MASS` para ajustar modelos lineales empleando un método robusto cuando hay datos atípicos.

Como ejemplo consideraremos el conjunto de datos *hbat.RData* que contiene observaciones de clientes de la compañía de distribución industrial HBAT (Hair et al., 1998). Las variables se pueden clasificar en tres grupos: las 6 primeras (categóricas) son características del comprador, las variables de la 7 a la 19 (numéricas) miden percepciones de HBAT por parte del comprador y las 5 últimas son posibles variables de interés (respuestas).

```
load("data/hbat.RData")
as.data.frame(attr(hbat, "variable.labels"))
```

```
##             attr(hbat, "variable.labels")
## empresa                           Empresa
## tcliente                 Tipo de cliente
```

```
## tindustr                  Tipo Industria
## tamaño               Tamaño de la empresa
## region                            Región
## distrib           Sistema de distribución
## calidadp              Calidad de producto
## web      Actividades comercio electrónico
## soporte                   Soporte técnico
## quejas               Resolución de quejas
## publi                          Publicidad
## producto               Línea de productos
## imgfvent       Imagen de fuerza de ventas
## precio                    Nivel de precios
## garantia         Garantía y reclamaciones
## nprod                     Nuevos productos
## facturac            Encargo y facturación
## flexprec          Flexibilidad de precios
## velocida             Velocidad de entrega
## satisfac            Nivel de satisfacción
## precomen          Propensión a recomendar
## pcompra              Propensión a comprar
## fidelida      Porcentaje de compra a HBAT
## alianza  Consideraría alianza estratégica
```

Consideraremos como respuesta la variable *fidelida* y, por comodidad, únicamente las variables continuas correspondientes a las percepciones de HBAT como variables explicativas (para una introducción al tratamiento de variables predictoras categóricas ver por ejemplo la Sección 8.5 de Fernández-Casal et al., 2019).

Como ya se comentó, se trata de un método clásico de Estadística y el procedimiento habitual es emplear toda la información disponible para construir el modelo y posteriormente (asumiendo que es el verdadero) utilizar métodos de inferencia para evaluar su precisión. Sin embargo seguiremos el procedimiento habitual en AE y particionaremos los datos en una muestra de entrenamiento y en otra de test.

```
df <- hbat[c(23, 7:19)]  # Nota: realmente no copia el objeto...
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
```

El primer paso antes del modelado suele ser realizar un análisis descriptivo. Por ejemplo podemos generar un gráfico de dispersión matricial y calcular la matriz de correlaciones (lineal de Pearson). Sin embargo en muchos casos el número de variables es grande y en lugar de emplear gráficos de dispersión puede ser preferible representar gráficamente las correlaciones mediante un mapa de calor o algún gráfico similar. Por ejemplo en la Figura 6.1 se combinan elipses con colores para representar las correlaciones.

```
# plot(train) # gráfico de dispersión matricial
mcor <- cor(train)
corrplot::corrplot(mcor, method = "ellipse")
```

```
print(mcor, digits = 1)
```

```
##          fidelida calidadp    web soporte quejas publi producto imgfvent precio
## fidelida     1.00     0.55  0.219   0.070   0.61  0.27     0.67     0.21  -0.19
## calidadp     0.55     1.00 -0.088   0.051   0.05 -0.05     0.51    -0.15  -0.43
## web          0.22    -0.09  1.000  -0.009   0.14  0.53     0.03     0.79   0.20
## soporte      0.07     0.05 -0.009   1.000   0.17  0.03     0.17     0.04  -0.11
```

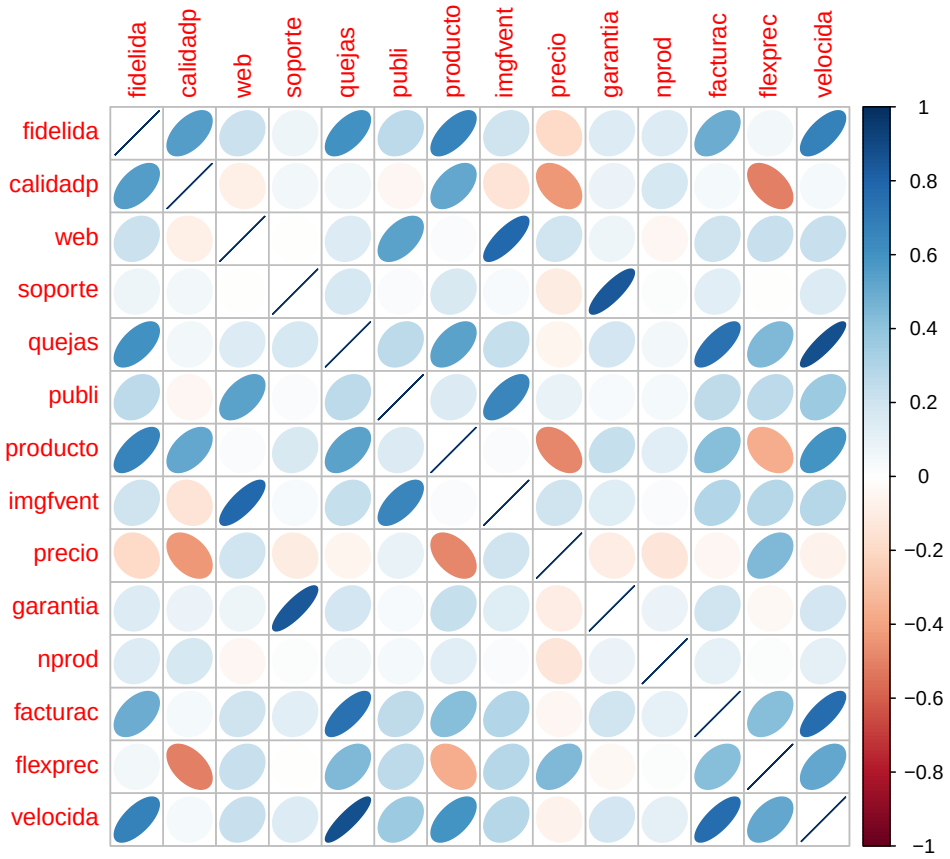


Figura 6.1: Representación de las correlaciones lineales entre las variables continuas del conjunto de datos HBAT, generada con la función `corrplot::corrplot()`.

```
##           garantia nprod facturac flexprec velocida
## fidelida      0.14   0.14     0.50    0.055     0.68
## calidadp      0.09   0.17     0.04   -0.509     0.04
## web           0.08  -0.05     0.21    0.221     0.23
## soporte       0.84   0.02     0.13   -0.005     0.14
##  [ reached getOption("max.print") -- omitted 10 rows ]
```

En este caso observamos que aparentemente hay una relación (lineal) entre la respuesta y algunas de las variables explicativas (que en principio no parece razonable suponer que son independientes). Vemos también que, si consideramos un modelo de regresión lineal simple, el mejor ajuste se obtendría empleando `velocida` como variable explicativa (ver Figura 6.2):

```
modelo <- lm(fidelida ~ velocida, train)
summary(modelo)
```

```
##
## Call:
## lm(formula = fidelida ~ velocida, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -13.8349  -4.3107   0.3677   4.3413  12.3677
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  27.5486     2.6961   10.22   <2e-16 ***
## velocida      7.9736     0.6926   11.51   <2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.403 on 158 degrees of freedom
## Multiple R-squared:  0.4562,   Adjusted R-squared:  0.4528
## F-statistic: 132.6 on 1 and 158 DF,  p-value: < 2.2e-16
```

```
plot(fidelida ~ velocida, train)
abline(modelo)
```

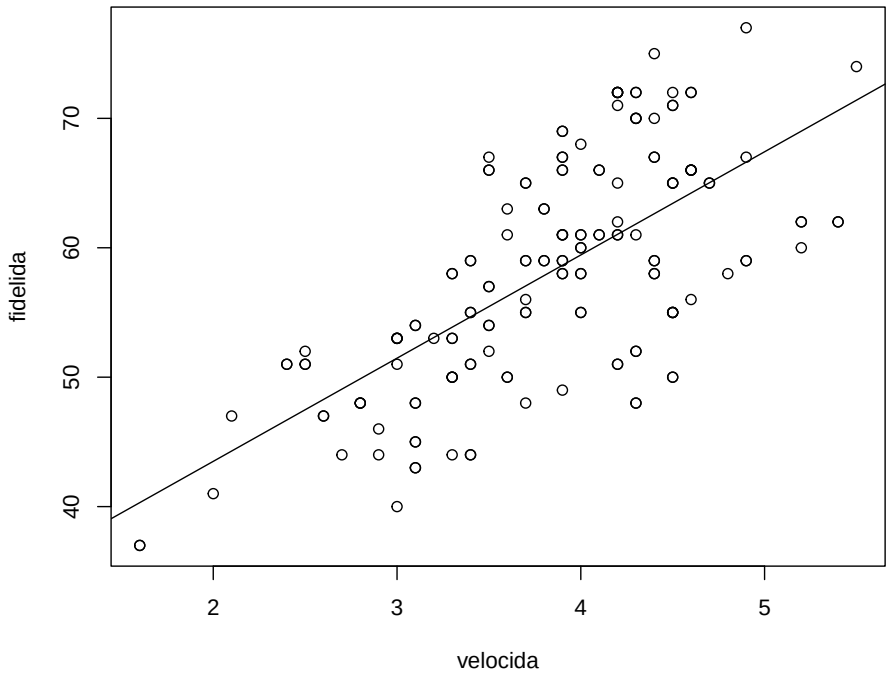


Figura 6.2: Gráfico de dispersión y recta de regresión ajustada para `fidelida` en función de la `velocida`.

Para calcular predicciones (estimaciones de la media condicionada), también intervalos de confianza o de predicción, se puede emplear el correspondiente método `predict()` (consultar la ayuda de `predict.lm()` para ver todas las opciones disponibles).

```
valores <- seq(1, 6, len = 100)
newdata <- data.frame(velocida = valores)
pred <- predict(modelo, newdata = newdata, interval = c("confidence"))
# head(pred)
```

Por ejemplo, en la Figura 6.3 se representan las predicciones, los intervalos de confianza para la media condicional y los intervalos de predicción para nuevas observaciones.

```
plot(fidelida ~ velocida, train)
matlines(valores, pred, lty = c(1, 2, 2), col = 1)
pred2 <- predict(modelo, newdata = newdata, interval = c("prediction"))
matlines(valores, pred2[, -1], lty = 3, col = 1)
legend("topleft", c("Ajuste", "Int. confianza", "Int. predicción"), lty = 1:3)
```

Para la extracción de información se pueden acceder a los componentes del modelo ajustado o emplear funciones (genéricas; muchas de ellas válidas para otro tipo de modelos: rlm, glm…). Algunas de las más utilizadas se muestran en la Tabla 6.1.

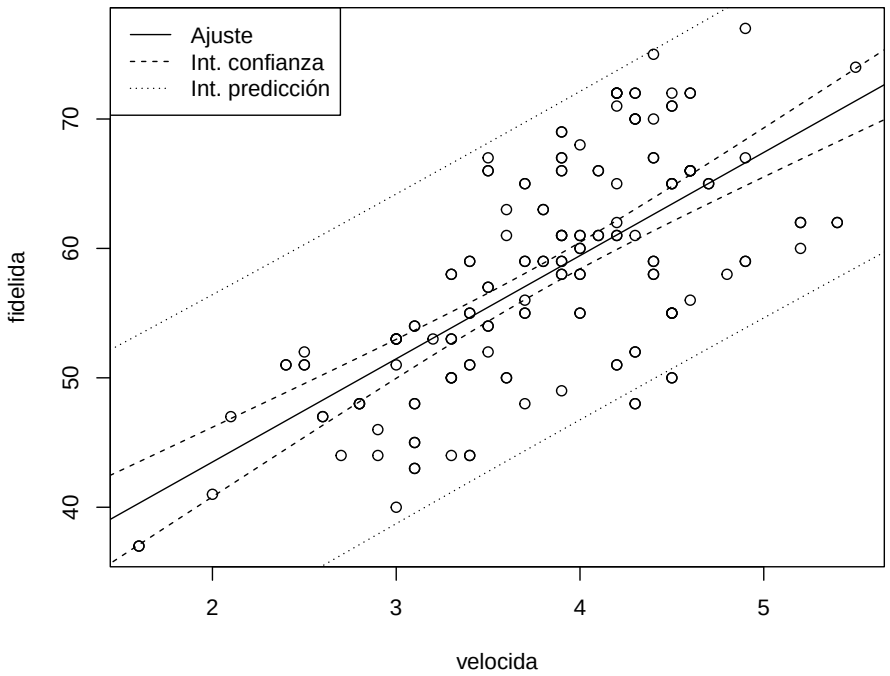


Figura 6.3: Ajuste lineal (predicciones) e intervalos de confianza y predicción (puntuales).

Tabla 6.1: Listado de las principales funciones auxiliares para modelos ajustados.

Función	Descripción
`fitted()`	valores ajustados
`coef()`	coeficientes estimados (y errores estándar)
`confint()`	intervalos de confianza para los coeficientes
`residuals()`	residuos
`plot()`	gráficos de diagnóstico
`termplot()`	gráfico de efectos parciales
`anova()`	calcula tablas de análisis de varianza (también permite comparar modelos)
`influence.measures()`	calcula medidas de diagnóstico ("dejando uno fuera"; LOOCV)
`update()`	actualiza un modelo (p.e. eliminando o añadiendo variables)

Ejemplos (no evaluados):

```
modelo2 <- update(modelo, . ~ . + calidadp)
summary(modelo2)
confint(modelo2)
anova(modelo2)
anova(modelo, modelo2)
oldpar <- par(mfrow=c(1,2))
termplot(modelo2, partial.resid = TRUE)
par(oldpar)
```

6.2 El problema de la colinealidad

Si alguna de las variables explicativas no aporta información relevante sobre la respuesta puede aparecer el problema de la colinealidad. En regresión múltiple se supone que ninguna de las variables explicativas es combinación lineal de las demás. Si una de las variables explicativas (variables independientes) es combinación lineal de las otras, no se pueden determinar los parámetros de forma única (sistema singular). Sin llegar a esta situación extrema, cuando algunas variables explicativas

estén altamente correlacionadas entre sí, tendremos una situación de alta colinealidad. En este caso las estimaciones de los parámetros pueden verse seriamente afectadas:

- Tendrán varianzas muy altas (serán poco eficientes).
- Habrá mucha dependencia entre ellas (al modificar ligeramente el modelo, añadiendo o eliminando una variable o una observación, se producirán grandes cambios en las estimaciones de los efectos).

Consideraremos un ejemplo de regresión lineal bidimensional con datos simulados en el que las dos variables explicativas están altamente correlacionadas (en este caso además solo una de las variables explicativas tiene un efecto lineal sobre la respuesta):

```
set.seed(1)
n <- 50
rand.gen <- runif # rnorm
x1 <- rand.gen(n)
rho <- sqrt(0.99) # coeficiente de correlación
x2 <- rho*x1 + sqrt(1 - rho^2)*rand.gen(n)
fit.x2 <- lm(x2 ~ x1)
# plot(x1, x2)
# summary(fit.x2)

# Rejilla x-y para predicciones:
x1.range <- range(x1)
x1.grid <- seq(x1.range[1], x1.range[2], length.out = 30)
x2.range <- range(x2)
x2.grid <- seq(x2.range[1], x2.range[2], length.out = 30)
xy <- expand.grid(x1 = x1.grid, x2 = x2.grid)

# Modelo teórico:
model.teor <- function(x1, x2) x1
# model.teor <- function(x1, x2) x1 - 0.5*x2
y.grid <- matrix(mapply(model.teor, xy$x1, xy$x2), nrow = length(x1.grid))
y.mean <- mapply(model.teor, x1, x2)
```

Los valores de las variables explicativas y la tendencia teórica se muestran en la Figura 6.4:

```
library(plot3D)
ylim <- c(-2, 3) # range(y, y.pred)
scatter3D(z = y.mean, x = x1, y = x2, pch = 16, cex = 1, clim = ylim, zlim = ylim,
          theta = -40, phi = 20, ticktype = "detailed",
          xlab = "x1", ylab = "x2", zlab = "y",
          surf = list(x = x1.grid, y = x2.grid, z = y.grid, facets = NA))
scatter3D(z = rep(ylim[1], n), x = x1, y = x2, add = TRUE, colkey = FALSE,
           pch = 16, cex = 1, col = "black")
x2.pred <- predict(fit.x2, newdata = data.frame(x1 = x1.range))
lines3D(z = rep(ylim[1], 2), x = x1.range, y = x2.pred, add = TRUE,
        colkey = FALSE, col = "black")
```

Para ilustrar el efecto de la correlación en los predictores, en la Figura 6.5 se muestran ejemplos de simulaciones bajo colinealidad y los correspondientes modelos ajustados. Podemos observar una alta variabilidad en los modelos ajustados (puede haber grandes diferencias en las estimaciones de los coeficientes de los predictores).

Incluso puede ocurrir que el contraste de regresión sea significativo (alto coeficiente de determinación), pero los contrastes individuales sean no significativos. Por ejemplo, en el último ajuste obtendríamos:

```
summary(fit)
```

```
##
```

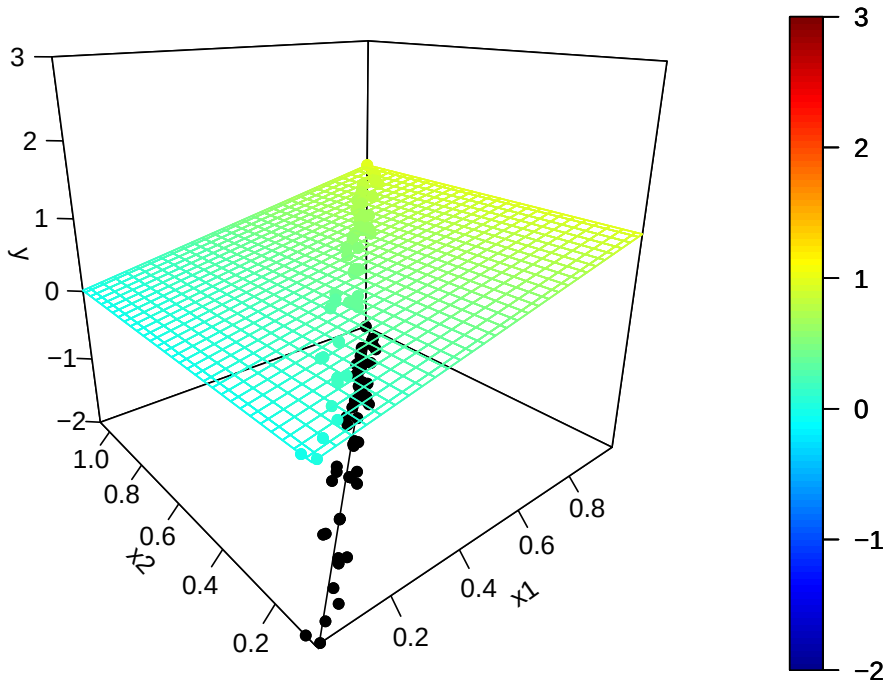


Figura 6.4: Modelo teórico y valores de las variables explicativas (altamente correlacionadas, con un coeficiente de determinación de 0.99).

```
## Call:
## lm(formula = y ~ x1 + x2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.45461 -0.13147  0.01428  0.16316  0.36616
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.11373    0.08944  -1.272    0.210
## x1           0.87084    1.19929   0.726    0.471
## x2           0.16752    1.19337   0.140    0.889
##
## Residual standard error: 0.2209 on 47 degrees of freedom
## Multiple R-squared:  0.6308,   Adjusted R-squared:  0.6151
## F-statistic: 40.15 on 2 and 47 DF,  p-value: 6.776e-11
```

Podemos comparar los resultados anteriores con el caso de predictores incorrelados (Ver Figura 6.6):

Por ejemplo, en el último ajuste obtendríamos:

```
summary(fit2)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.45800 -0.08645  0.00452  0.15402  0.33662
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

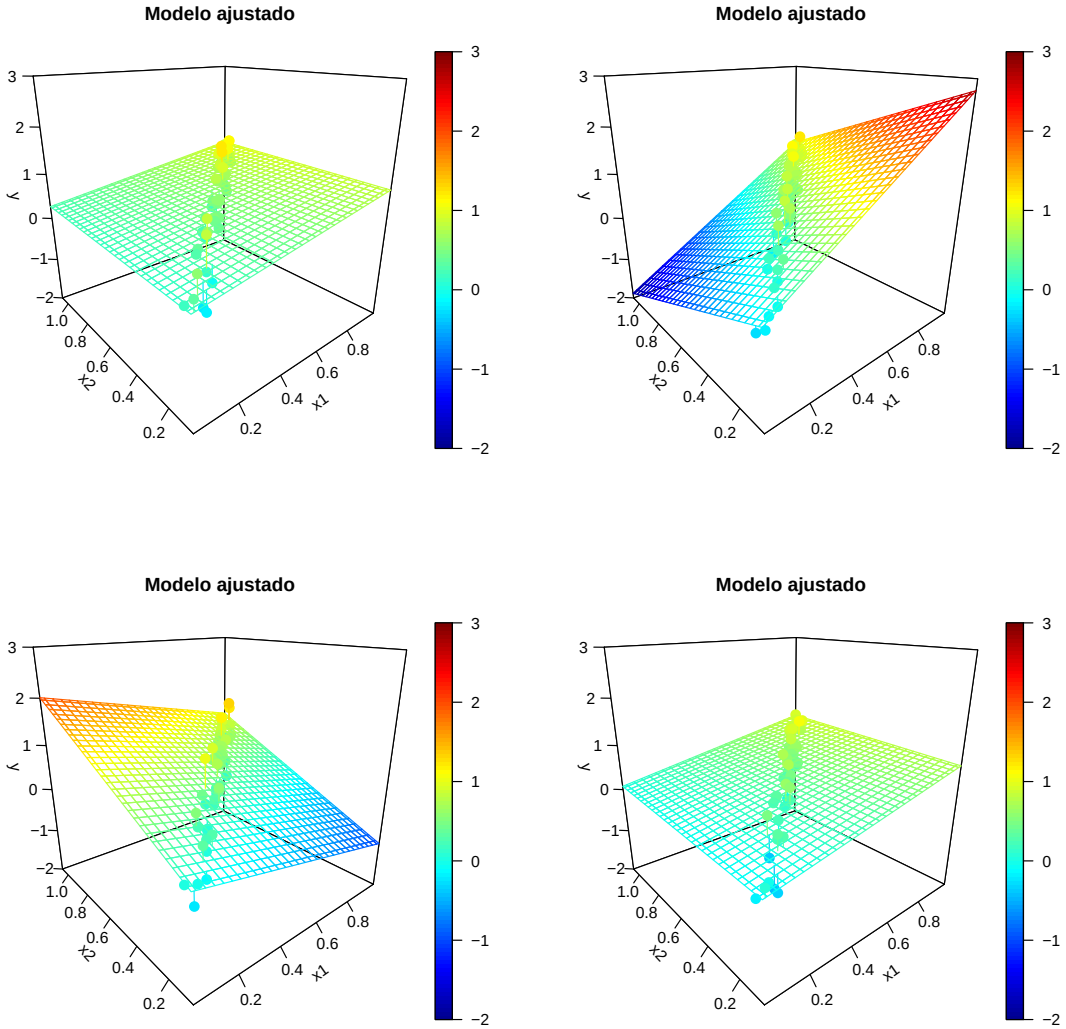


Figura 6.5: Ejemplo de simulaciones bajo colinelidad y correspondientes modelos ajustados.

```
## (Intercept) -0.22365    0.08515  -2.627   0.0116 *
## x1           1.04125    0.11044   9.428 2.07e-12 ***
## x2           0.22334    0.10212   2.187   0.0337 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2105 on 47 degrees of freedom
## Multiple R-squared:  0.6648,   Adjusted R-squared:  0.6505
## F-statistic:  46.6 on 2 and 47 DF,  p-value: 7.016e-12
```

En la práctica, para la detección de colinealidad, se puede emplear la función `vif()` del paquete `car` para calcular los factores de inflación de varianza para las variables del modelo. Por ejemplo, en los últimos ajustes obtendríamos:

```
library(car)
vif(fit)
```

```
##       x1       x2
## 107.0814 107.0814
```

```
vif(fit2)
```

```
##       x1       x2
```

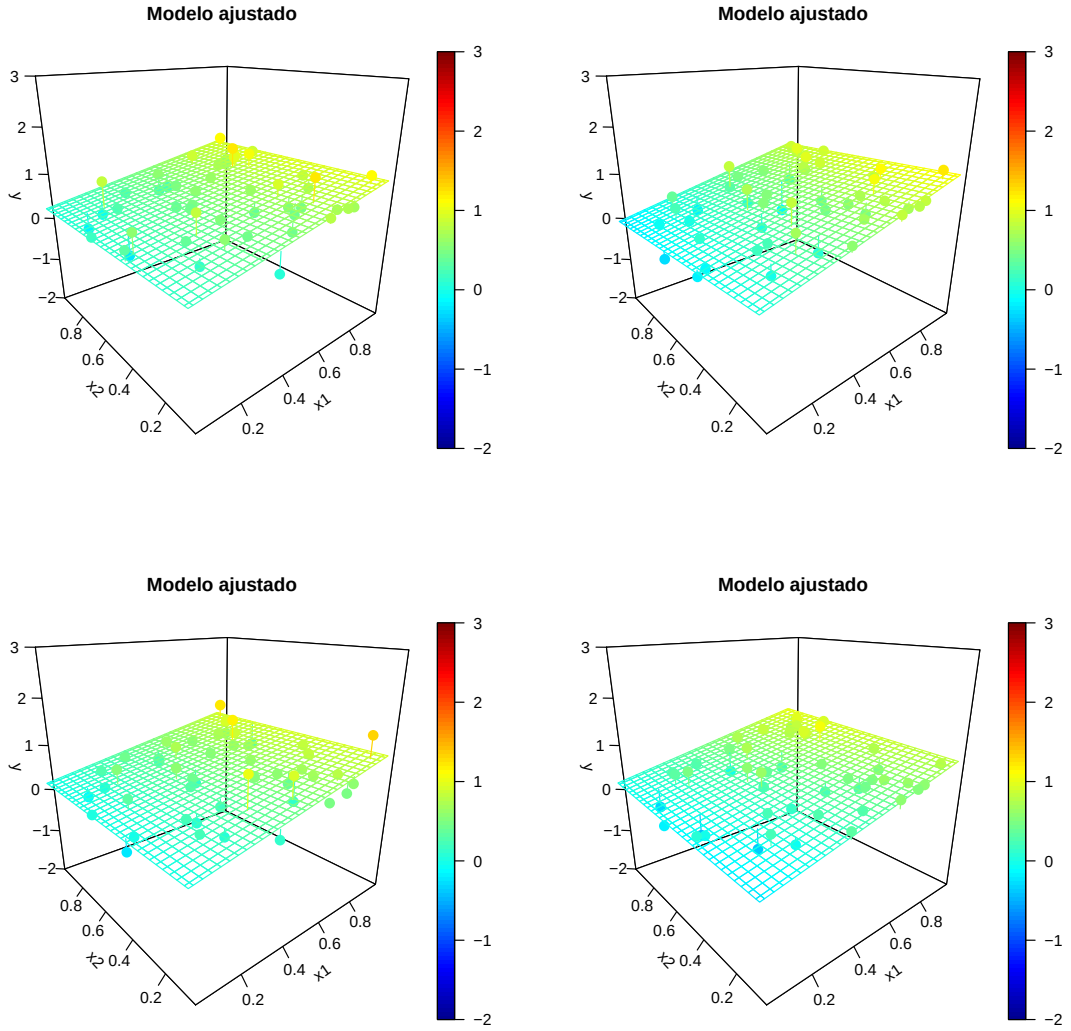


Figura 6.6: Ejemplo de simulaciones bajo independencia y correspondientes modelos ajustados.

```
## 1.000139 1.000139
```

La idea de este estadístico es que la varianza de la estimación del efecto en regresión simple (efecto global) es menor que en regresión múltiple (efecto parcial). El factor de inflación de la varianza mide el incremento debido a la colinealidad. Valores grandes, por ejemplo > 10, indican la posible presencia de colinealidad.

Las tolerancias, proporciones de variabilidad no explicada por las demás covariables, se pueden calcular con `1/vif(modelo)`. Por ejemplo, los coeficientes de tolerancia de los últimos ajustes serían:

```
1/vif(fit)
```

```
##          x1          x2
## 0.009338689 0.009338689
```

```
1/vif(fit2)
```

```
##        x1        x2
## 0.9998606 0.9998606
```

Como ya se comentó en la Sección 1.4, el problema de la colinealidad se agrava al aumentar el número de dimensiones (la maldición de la dimensionalidad). Hay que tener en cuenta también que, además de la dificultad para interpretar el efecto de los predictores, va a resultar más difícil 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, especialmente si se recurre a los contrastes tradicionales para determinar si tienen un efecto significativo.

6.3 Selección de variables explicativas

Cuando se dispone de un conjunto grande de posibles variables explicativas suele ser especialmente importante determinar cuales de estas deberían ser incluidas en el modelo de regresión. Si alguna de las variables no contiene información relevante sobre la respuesta no se debería incluir (se simplificaría la interpretación del modelo, aumentaría la precisión de la estimación y se evitarían problemas como la colinealidad). Se trataría entonces de conseguir un buen ajuste con el menor número de predictores posible.

6.3.1 Búsqueda exhaustiva

Para obtener el modelo "óptimo" lo ideal sería evaluar todas las posibles combinaciones de los predictores. La función `regsubsets()` del paquete `leaps` permite seleccionar los mejores modelos fijando el número máximo de variables explicativas. Por defecto, evalúa todos los modelos posibles con un determinado número de parámetros (variando desde 1 hasta por defecto un máximo de `nvmax = 8`) y selecciona el mejor (`nbest = 1`).

```
library(leaps)
regsel <- regsubsets(fidelida ~ . , data = train)
# summary(regsel)
# names(summary(regsel))
```

Al representar el resultado se obtiene un gráfico con los mejores modelos ordenados según el criterio determinado por el argumento `scale = c("bic", "Cp", "adjr2", "r2")` (para detalles sobre estas medidas ver por ejemplo la Sección 6.1.3 de James et al., 2021). Se representa una matriz en la que las filas se corresponden con los modelo y las columnas con predictores, indicando los incluidos en cada modelo mediante un sombreado. Por ejemplo, en la Figura 6.7 se muestra el resultado obtenido empleando el coeficiente de determinación ajustado.

```
plot(regsel, scale = "adjr2")
```

En este caso, considerando que es preferible un modelo más simple que una mejora del 2% en la proporción de variabilidad explicada, podríamos seleccionar como modelo final el modelo con dos predictores. Podemos obtener fácilmente los coeficientes de este modelo:

```
coef(regsel, 2)
```

```
## (Intercept)    calidadp    velocida
##    3.332511    3.204201    7.700260
```

pero normalmente nos interesará ajustarlo de nuevo:

```
modelo <- lm(fidelida ~ velocida + calidadp, data = train)
```

Notas:

- Si se emplea alguno de los criterios habituales, el mejor modelo con un determinado número de variables no depende del criterio empleado. Aunque estos criterios pueden diferir al comparar modelos con distinto número de variables explicativas.
- Si el número de variables explicativas es grande, en lugar de emplear una búsqueda exhaustiva se puede emplear un criterio por pasos, mediante el argumento `method = c("backward", "forward", "seqrep")`, pero puede ser recomendable emplear el paquete `MASS` para obtener directamente el modelo final.

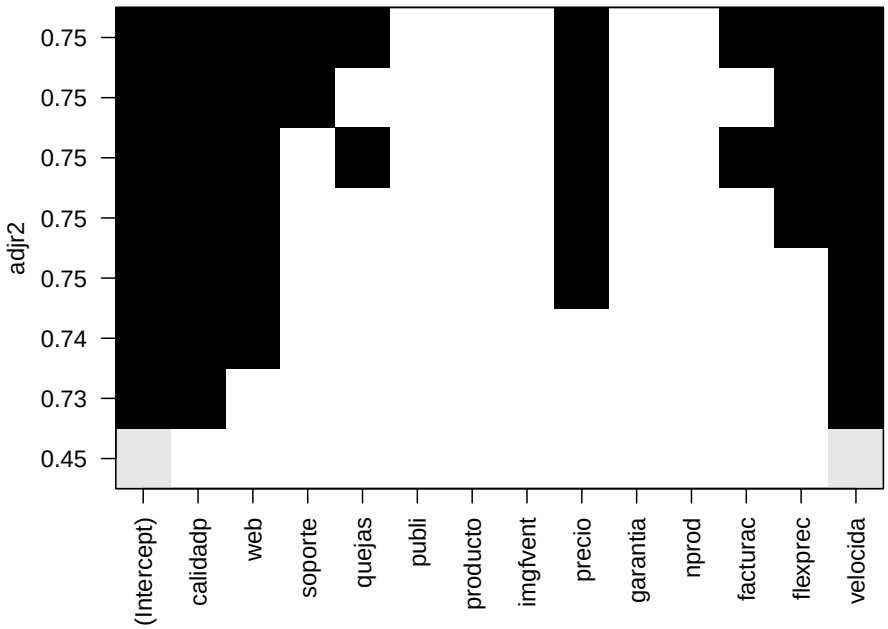


Figura 6.7: Modelos obtenidos mediante búsqueda exhaustiva ordenados según su coeficiente de determinación ajustado.

6.3.2 Selección por pasos

Si el número de variables es grande (no sería práctico evaluar todas las posibilidades) se suele utilizar alguno (o varios) de los siguientes métodos:

- Selección progresiva (*forward*): Se parte de una situación en la que no hay ninguna variable y en cada paso se incluye una aplicando un criterio de entrada (hasta que ninguna de las restantes lo verifican).
- Eliminación progresiva (*backward*): Se parte del modelo con todas las variables y en cada paso se elimina una aplicando un criterio de salida (hasta que ninguna de las incluidas lo verifican).
- Selección paso a paso (*stepwise*): Se combina un criterio de entrada y uno de salida. Normalmente se parte sin ninguna variable y en cada paso puede haber una inclusión y posteriormente la exclusión de alguna de las anteriormente añadidas (*forward/backward*). Otra posibilidad es partir del modelo con todas las variables y en cada paso puede haber una exclusión y posteriormente la inclusión de alguna de las anteriormente eliminadas (*backward/forward*).

Hay que tener en cuenta que se tratan de algoritmos "avariciosos" (también denominados "voraces"), ya que en cada paso tratan de elegir la mejor opción, pero puede que el resultado final no sea la solución global óptima (de hecho es bastante habitual que los modelos finales sean diferentes).

La función `stepAIC()` del paquete `MASS` permite seleccionar el modelo por pasos[2], hacia delante o hacia atrás según criterio AIC (*Akaike Information Criterion*) o BIC (*Bayesian Information Criterion*). La función `stepwise()` del paquete `RcmdrMisc` es una interfaz de `stepAIC()` que facilita su uso. Los métodos disponibles son `"backward/forward"`, `"forward/backward"`, `"backward"` y `"forward"`. Normalmente obtendremos un modelo más simple combinando el método por pasos hacia delante con el criterio BIC:

```
library(MASS)
library(RcmdrMisc)
modelo.completo <- lm(fidelida ~ . , data = train)
modelo <- stepwise(modelo.completo, direction = "forward/backward", criterion = "BIC")
```

```
##
```

[2]También está disponible la función `step()` del paquete base `stats` con menos opciones.

```
## Direction:  forward/backward
## Criterion:  BIC
##
## Start:  AIC=694.72
## fidelida ~ 1
##
##            Df Sum of Sq     RSS    AIC
## + velocida  1    5435.2  6478.5 602.32
## + producto  1    5339.6  6574.2 604.67
## + quejas    1    4405.4  7508.4 625.93
## + calidadp  1    3664.7  8249.1 640.98
## + facturac  1    2962.6  8951.2 654.05
## + publi     1     866.5 11047.3 687.71
## + web       1     572.1 11341.6 691.92
## + imgfvent  1     516.4 11397.4 692.70
## + precio    1     433.4 11480.4 693.87
## <none>                  11913.8 694.72
## + garantia  1     248.7 11665.1 696.42
## + nprod     1     234.1 11679.6 696.62
## + soporte   1      59.0 11854.7 699.00
## + flexprec  1      35.9 11877.9 699.31
##
## Step:  AIC=602.32
## fidelida ~ velocida
##
##            Df Sum of Sq     RSS    AIC
## + calidadp  1    3288.7  3189.9 494.04
## + flexprec  1    1395.7  5082.9 568.58
## + producto  1    1312.1  5166.5 571.19
## + precio    1     254.7  6223.8 600.98
## <none>                  6478.5 602.32
## + web       1      54.4  6424.2 606.05
## + nprod     1      45.1  6433.4 606.28
## + quejas    1      13.5  6465.1 607.06
## + facturac  1       9.6  6468.9 607.16
## + publi     1       8.4  6470.1 607.19
## + soporte   1       7.9  6470.6 607.20
## + garantia  1       4.8  6473.7 607.28
## + imgfvent  1       2.4  6476.1 607.34
## - velocida  1    5435.2 11913.8 694.72
##
## Step:  AIC=494.04
## fidelida ~ velocida + calidadp
##
##            Df Sum of Sq    RSS    AIC
## + web       1     175.4 3014.5 490.06
## + imgfvent  1     125.6 3064.3 492.68
## <none>                  3189.9 494.04
## + precio    1      95.3 3094.6 494.26
## + publi     1      48.1 3141.8 496.68
## + soporte   1      29.4 3160.5 497.63
## + facturac  1      15.3 3174.6 498.34
## + nprod     1       9.7 3180.2 498.63
## + garantia  1       6.2 3183.7 498.80
## + quejas    1       5.2 3184.7 498.85
## + flexprec  1       4.8 3185.0 498.87
```

```
## + producto  1       3.6 3186.3 498.93
## - calidadp  1    3288.7 6478.5 602.32
## - velocida  1    5059.2 8249.1 640.98
##
## Step:  AIC=490.06
## fidelida ~ velocida + calidadp + web
##
##            Df Sum of Sq    RSS    AIC
## <none>                  3014.5 490.06
## + precio    1      53.8 2960.7 492.26
## + soporte   1      24.2 2990.3 493.85
## + facturac  1      21.9 2992.6 493.97
## - web       1     175.4 3189.9 494.04
## + quejas    1      14.7 2999.8 494.36
## + flexprec  1      10.6 3004.0 494.58
## + producto  1      10.3 3004.2 494.59
## + garantia  1       9.7 3004.8 494.62
## + nprod     1       5.3 3009.3 494.86
## + imgfvent  1       2.5 3012.1 495.01
## + publi     1       0.2 3014.3 495.13
## - calidadp  1    3409.7 6424.2 606.05
## - velocida  1    4370.9 7385.4 628.36
```

En la salida de texto de esta función, "<none>" representa el modelo actual en cada paso y se ordenan las posibles acciones dependiendo del criterio elegido (aunque siempre muestra el valor de AIC). El algoritmo se detiene cuando ninguna de ellas mejora el modelo actual. Como resultado devuelve el modelo ajustado final:

```
summary(modelo)
```

```
##
## Call:
## lm(formula = fidelida ~ velocida + calidadp + web, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -12.1533  -1.8588   0.1145   3.0086   7.7625
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -1.2205     3.0258  -0.403  0.68724
## velocida      7.3582     0.4893  15.040  < 2e-16 ***
## calidadp      3.2794     0.2469  13.283  < 2e-16 ***
## web           1.4005     0.4649   3.012  0.00302 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.396 on 156 degrees of freedom
## Multiple R-squared:  0.747,    Adjusted R-squared:  0.7421
## F-statistic: 153.5 on 3 and 156 DF,  p-value: < 2.2e-16
```

Cuando el número de variables explicativas es muy grande (o si el tamaño de la muestra es pequeño en comparación) pueden aparecer problemas al emplear los métodos anteriores (incluso pueden no ser aplicables). Una alternativa son los métodos de regularización (ridge regression, LASSO; Sección 6.7) o los de reducción de la dimensión (regresión con componentes principales o mínimos cuadrados parciales; Sección 6.8).

Por otra parte en los modelos anteriores no se consideraron interacciones entre predictores (para detalles sobre como incluir interacciones en modelos lineales ver por ejemplo la Sección 8.6 de Fernández-

Casal et al., 2019). Por ejemplo podríamos considerar como modelo completo `respuesta ~ .*.`, que incluiría los efectos principales y las interacciones de orden 2 de todos los predictores.

En la práctica se suele comenzar con modelos aditivos y posteriormente se estudian posibles interacciones siguiendo un proceso interactivo (aunque también, por ejemplo, se podría considerar un nuevo modelo completo a partir de las variables seleccionadas en el modelo aditivo, incluyendo todas las posibles interacciones de orden 2, y posteriormente aplicar alguno de los métodos de selección anteriores). Como ya vimos en capítulos anteriores, en AE interesan algoritmos que puedan detectar e incorporar automáticamente efectos de interacción (en el siguiente capítulo veremos extensiones en este sentido).

6.4 Análisis e interpretación del modelo

Al margen de la colinealidad, si no se verifican las otras hipótesis estructurales del modelo (Sección 6.1), los resultados y las conclusiones basadas en la teoría estadística pueden no ser fiables, o incluso totalmente erróneas:

- La falta de linealidad "invalida" las conclusiones obtenidas (cuidado con las extrapolaciones).
- La falta de normalidad tiene poca influencia si el número de datos es suficientemente grande (TCL). En caso contrario la estimación de la varianza, los intervalos de confianza y los contrastes podrían verse afectados.
- Si no hay igualdad de varianzas los estimadores de los parámetros no son eficientes pero sí insesgados. Las varianzas, los intervalos de confianza y contrastes podrían verse afectados.
- La dependencia entre observaciones puede tener un efecto mucho más grave.

Con el método `plot()` se pueden generar gráficos de interés para la diagnosis del modelo (ver Figura 6.8):

```
oldpar <- par(mfrow = c(2,2))
plot(modelo)
```

```
par(oldpar)
```

Por defecto se muestran cuatro gráficos (ver `help(plot.lm)` para más detalles). El primero (residuos frente a predicciones) permite detectar falta de linealidad o heterocedasticidad (o el efecto de un factor omitido: mala especificación del modelo), lo ideal sería no observar ningún patrón. El segundo gráfico (gráfico QQ), permite diagnosticar la normalidad, los puntos del deberían estar cerca de la diagonal. El tercer gráfico de dispersión-nivel permite detectar heterocedasticidad (la pendiente debería ser nula) y ayudar a seleccionar una transformación para corregirla (también se podría emplear la función `boxcox()` del paquete `MASS`). El último gráfico permite detectar valores atípicos o influyentes. Representa los residuos estandarizados en función del valor de influencia (a priori) o leverage (hii que depende de los valores de las variables explicativas, debería ser $< 2(p+1)/2$) y señala las observaciones atípicas (residuos fuera de [-2,2]) e influyentes a posteriori (estadístico de Cook >0.5 y >1).

Si las conclusiones obtenidas dependen en gran medida de una observación (normalmente atípica), esta se denomina influyente (a posteriori) y debe ser examinada con cuidado por el experimentador. Se puede volver a ajustar el modelo eliminando las observaciones influyentes[3], pero puede ser recomendable emplear regresión lineal robusta, por ejemplo mediante la función `rlm()` del paquete `MASS`).

En regresión lineal múltiple, en lugar de generar gráficos de dispersión simple (p.e. gráficos de dispersión matriciales) para analizar los efectos de las variables explicativas y detectar posibles problemas (falta de linealidad…), se pueden generar gráficos parciales de residuos, por ejemplo con el comando:

```
termplot(modelo, partial.resid = TRUE)
```

Aunque puede ser preferible emplear las funciones `crPlots()` ó `avPlots()` del paquete `car`[4]:

[3] Normalmente se sigue un proceso iterativo, eliminando la más influyente cada vez, por ejemplo con `which.max(cooks.distance(modelo))` y `update()`.

[4] Estas funciones permitirían además detectar puntos atípicos o influyentes mediante el argumento `id`.

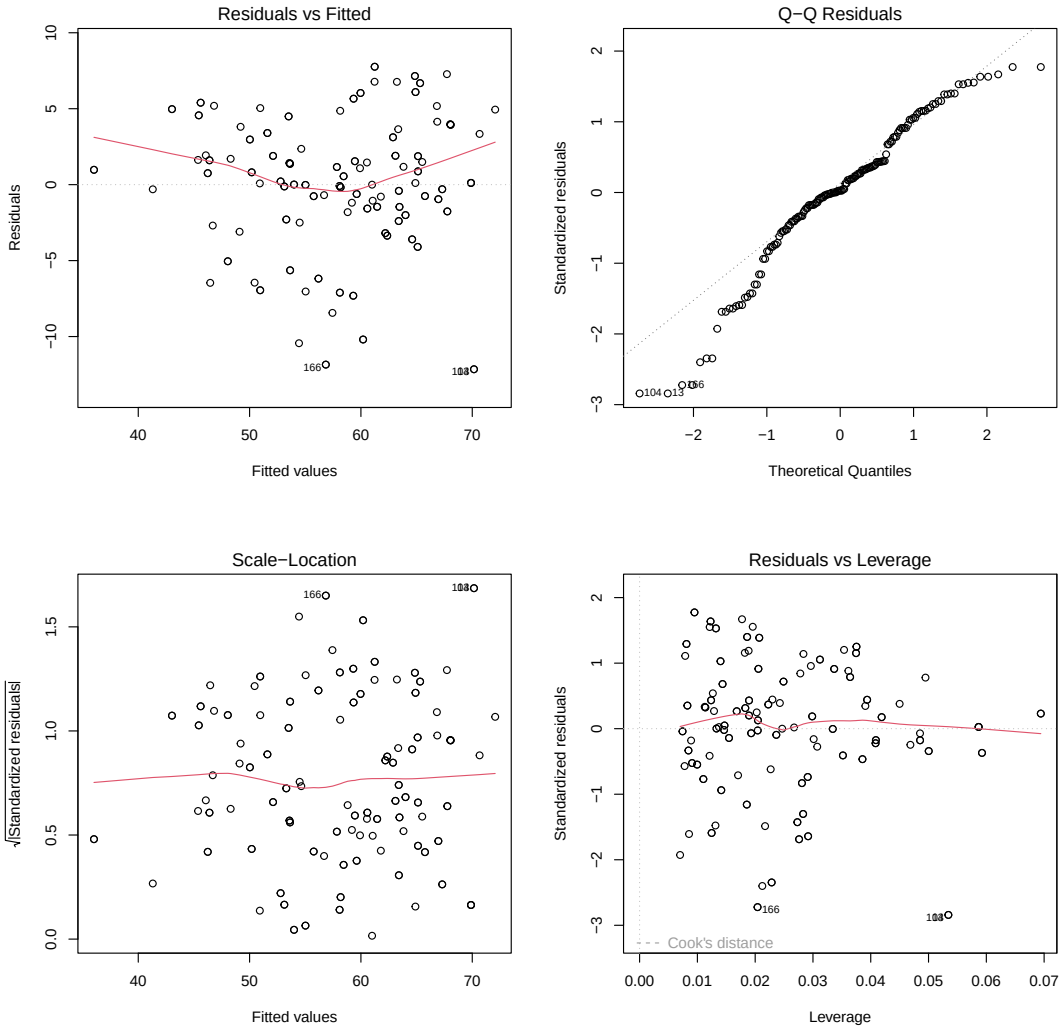


Figura 6.8: Gráficos de diagnóstico del ajuste lineal.

```
library(car)
# avPlots(modelo)
crPlots(modelo, main = "")
```

En la tabla 6.2 se incluyen algunas funciones adicionales que permiten obtener medidas de diagnosis o resúmenes numéricos de interés (ver `help(influence.measures)` para un listado más completo).

Tabla 6.2: Listado de las principales funciones auxiliares para modelos ajustados.

Función	Descripción
`rstandard()`	residuos estandarizados (también eliminados)
`rstudent()`	residuos estudentizados
`cooks.distance()`	valores del estadístico de Cook
`influence()`	valores de influencia, cambios en coeficientes y varianza residual al eliminar cada dato (LOOCV).

Hay muchas herramientas adicionales disponibles en otros paquetes. Por ejemplo, como ya se comentó, se puede emplear la función `vif()` del paquete `car` para calcular los factores de inflación de varianza, aunque puede ser preferible emplear otras medidas como el *índice de condicionamiento*, implementado en el paquete `mctest`. La librería `lmtest` proporciona herramientas adicionales para la diagnosis de

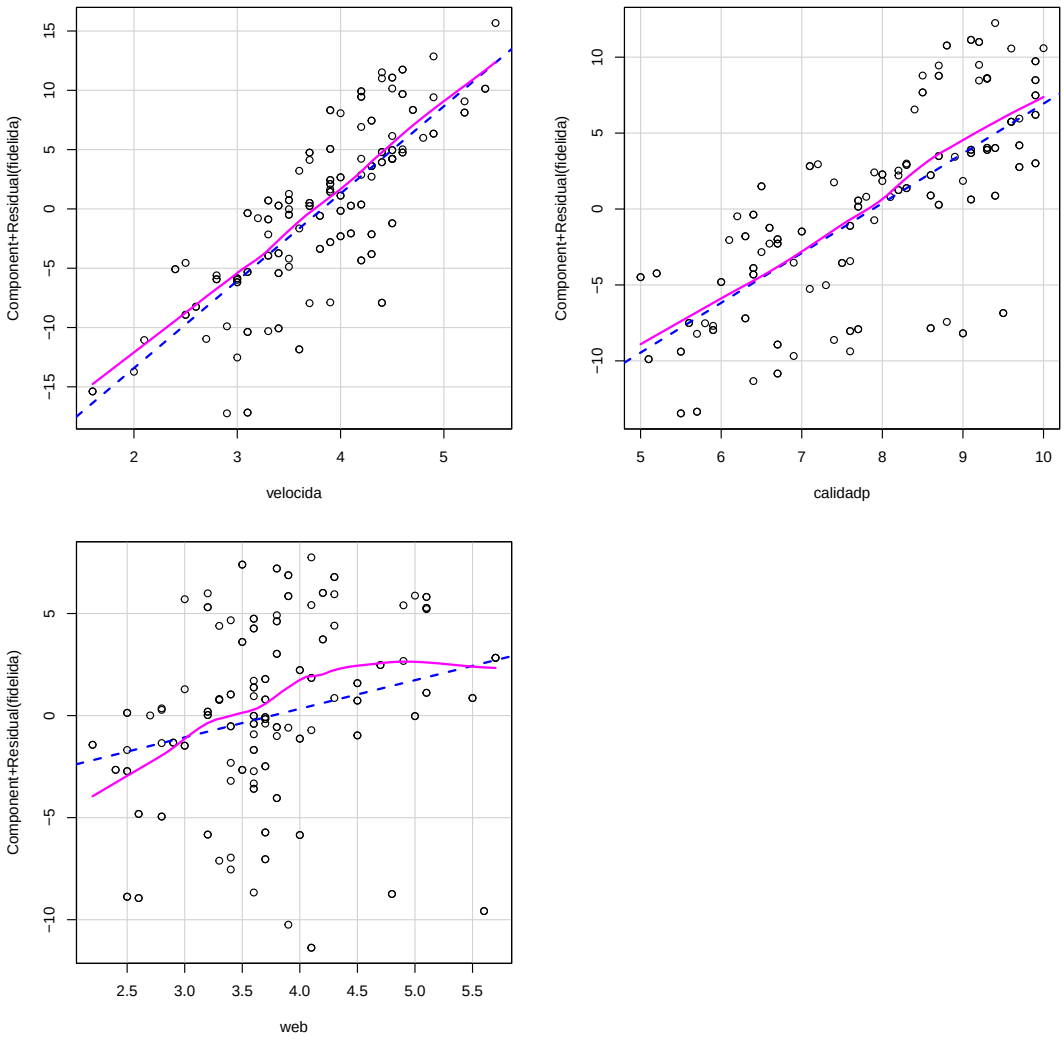


Figura 6.9: Gráficos parciales de residuos (componentes + residuos) del ajuste lineal.

modelos lineales, por ejemplo el test de Breusch-Pagan (para contrastar homocedasticidad) en la función `bptest()` o el de Durbin-Watson (para detectar si hay correlación en serie) en `dwtest()`.

Posibles soluciones cuando no se satisfacen los supuestos básicos:

- Como ya se comentó, pueden llevarse a cabo transformaciones de los datos para tratar de corregir la falta de linealidad, heterocedasticidad y/o normalidad (normalmente estas últimas "suelen ocurrir en la misma escala"). Otra alternativa sería tratar de emplear modelos lineales generalizados.
- Si no se logra corregir la heterocedasticidad puede ser adecuado utilizar mínimos cuadrados ponderados (habría que modelar la varianza).
- Si hay dependencia se puede tratar de modelarla y utilizar mínimos cuadrados generalizados.
- Si no se logra corregir la falta de linealidad se puede pensar en utilizar modelos más flexibles (capítulo siguiente y anteriores).

Otra alternativa es emplear las técnicas de aprendizaje estadístico descritas en la Sección 1.3. Desde este punto de vista podríamos ignorar las hipótesis estructurales y pensar que los procedimientos clásicos, como por ejemplo el ajuste lineal mediante el método por pasos, son simplemente algoritmos de predicción. En ese caso, después de ajustar el modelo en la muestra de entrenamiento, en lugar de emplear medidas como el coeficiente de determinación ajustado, emplearíamos la muestra de test para evaluar la capacidad predictiva en nuevas observaciones.

Trataremos en primer lugar este último paso y posteriormente, en la Sección 6.6, se darán algunas nociones de como se podría haber empleado remuestreo para la selección del modelo.

6.5 Evaluación de la precisión

Para evaluar la precisión de las predicciones podríamos utilizar el coeficiente de determinación ajustado:

```
summary(modelo)$adj.r.squared
```

```
## [1] 0.7421059
```

que estimaría la proporción de variabilidad explicada en una nueva muestra. Sin embargo, hay que tener en cuenta que su validez dependería de la de las hipótesis estructurales (especialmente de la linealidad, homocedasticidad e independencia), ya que se obtiene a partir de estimaciones de las varianzas residual y total:

$$R^2_{ajus} = 1 - \frac{\hat{S}^2_R}{\hat{S}^2_Y} = 1 - \left(\frac{n-1}{n-p-1}\right)(1-R^2)$$

siendo $\hat{S}^2_R = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2/(n-p-1)$. Algo similar ocurriría con otras medidas de bondad de ajuste, como por ejemplo BIC o AIC.

Alternativamente, por si no es razonable asumir estas hipótesis, se pueden emplear el procedimiento tradicional en AE (o alguno de los otros descritos en la Sección 1.3).

Podemos evaluar el modelo ajustado en el conjunto de datos de test y comparar las predicciones frente a los valores reales (ver Figura 6.10):

```
obs <- test$fidelida
pred <- predict(modelo, newdata = test)
plot(pred, obs, xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
res <- lm(obs ~ pred) # summary(res)
abline(res, lty = 2)
```

También podemos obtener medidas de error, por ejemplo empleando la función `accuracy()` de la Sección 1.3.4:

```
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
##  0.4032996  4.1995208  3.3013714 -0.1410512  5.9553699  0.8271449
```

6.6 Selección del modelo mediante remuestreo

De igual forma, los métodos de selección de variables descritos en la Sección 6.3 dependen (en mayor o menor medida) de la validez de las hipótesis estructurales. Por este motivo se podría pensar en emplear alguno de los procedimientos descritos en la Sección 1.3. Por ejemplo, podríamos considerar como hiperparámetros la inclusión no de cada una de las posibles variables explicativas y realizar la selección de variables (la complejidad del modelo) mediante remuestreo. Sin embargo esto puede presentar dificultades computacionales.

Otra posibilidad es emplear remuestreo para escoger entre distintos procedimientos de selección o para escoger el número de predictores incluidos en el modelo. En este caso, el procedimiento de selección debería realizarse también en cada uno de los conjuntos de entrenamiento utilizados en la validación.

Esto último puede hacerse fácilmente con el paquete `caret`. Por ejemplo, este paquete implementa métodos de selección basados en el paquete `leaps`, considerando el número máximo de predictores

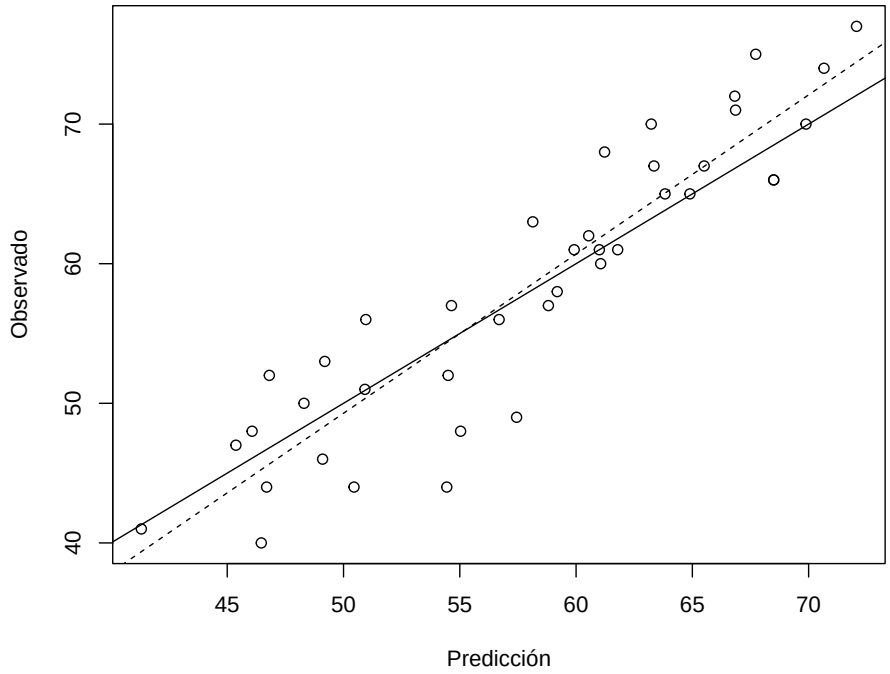


Figura 6.10: Gráfico de dispersión de observaciones frente a predicciones, del ajuste lineal en la muestra de test.

`nvmax` como hiperparámetro y empleando búsqueda: hacia atrás (`"leapBackward"`), hacia adelante (`"leapForward"`) y por pasos (`"leapSeq"`).

```
library(caret)
modelLookup("leapSeq")
```

```
##     model parameter                         label forReg forClass probModel
## 1 leapSeq     nvmax Maximum Number of Predictors   TRUE    FALSE     FALSE
```

```
caret.leapSeq <- train(fidelida ~ ., data = train, method = "leapSeq",
                   trControl = trainControl(method = "cv", number = 10),
                   tuneGrid = data.frame(nvmax = 1:6))
caret.leapSeq
```

```
## Linear Regression with Stepwise Selection
##
## 160 samples
##  13 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 144, 143, 145, 144, 144, 143, ...
## Resampling results across tuning parameters:
##
##   nvmax  RMSE      Rsquared   MAE
##   1      6.585282  0.4344609  5.354280
##   2      6.378169  0.4586041  5.077810
##   3      6.364526  0.4611045  5.085980
##   4      4.407117  0.7505874  3.387448
##   5      4.446449  0.7525050  3.366581
##   6      4.652626  0.7247463  3.553985
##
## RMSE was used to select the optimal model using the smallest value.
```

```
  ## The final value used for the model was nvmax = 4.
```

```
# summary(caret.leapSeq$finalModel)
with(caret.leapSeq, coef(finalModel, bestTune$nvmax))
```

```
  ## (Intercept)    calidadp         web      precio    velocida
  ##  -5.3610807   3.4712306   1.2471721   0.4190875   7.4382584
```

Una vez seleccionado el modelo final[5], estudiaríamos la eficiencia de las predicciones en la muestra de test:

```
pred <- predict(caret.leapSeq, newdata = test)
accuracy(pred, obs)
```

```
  ##         me       rmse        mae        mpe       mape  r.squared
  ##  0.2886512  4.1741220  3.2946370 -0.3413883  5.9590279  0.8292294
```

Además, en el caso de ajustes de modelos de este tipo, puede resultar de interés realizar un preprocesado de los datos para eliminar predictores correlados o con varianza próxima a cero, estableciendo por ejemplo `preProc = c("nzv", "corr")` en la llamada a la función `train()`.

6.7 Métodos de regularización

Como ya se comentó, el procedimiento habitual para ajustar un modelo de regresión lineal es emplear mínimos cuadrados, es decir, utilizar como criterio de error la suma de cuadrados residual

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta^t \mathbf{x}_i\right)^2$$

Si el modelo lineal es razonablemente adecuado, utilizar RSS va a dar lugar a estimaciones con poco sesgo, y si además $n \gg p$, entonces el modelo también va a tener poca varianza (bajo las hipótesis estructurales, la estimación es insesgada y además de varianza mínima entre todas las técnicas insesgadas). Las dificultades surgen cuando p es grande o cuando hay correlaciones altas entre las variables predictoras: tener muchas variables dificulta la interpretación del modelo, y si además hay problemas de colinealidad o se incumple $n \gg p$, entonces la estimación del modelo va a tener muchas varianza y el modelo estará sobreajustado. La solución pasa por forzar a que el modelo tenga menos complejidad para así reducir su varianza. Una forma de conseguirlo es mediante la regularización (*regularization* o *shrinkage*) de la estimación de los parámetros $\beta_1, \beta_2, \ldots, \beta_p$ que consiste en considerar todas las variables predictoras pero forzando a que algunos de los parámetros se estimen mediante valores muy próximos a cero, o directamente con ceros. Esta técnica va a provocar un pequeño aumento en el sesgo pero a cambio una notable reducción en la varianza y una interpretación más sencilla del modelo resultante.

Hay dos formas básicas de lograr esta simplificación de los parámetros (con la consiguiente simplificación del modelo), utilizando una penalización cuadrática (norma L_2) o en valor absoluto (norma L_1):

- *Ridge regression* (Hothorn et al., 2006)

$$\min_{\beta_0,\beta} \text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

  Equivalentemente,

$$\min_{\beta_0,\beta} \text{RSS}$$

  sujeto a

$$\sum_{j=1}^{p} \beta_j^2 \leq s$$

[5] Se podrían haber entrenado distintos métodos de selección de predictores y comparar los resultados (en las mismas muestras de validación) para escoger el modelo final.

- LASSO (*least absolute shrinkage and selection operator*, Tibshirani, 1996)

$$\min_{\beta_0,\beta} RSS + \lambda \sum_{j=1}^{p} |\beta_j|$$

Equivalentemente,

$$\min_{\beta_0,\beta} \text{RSS}$$

sujeto a

$$\sum_{j=1}^{p} |\beta_j| \leq s$$

Una formulación unificada consiste en considerar el problema

$$\min_{\beta_0,\beta} RSS + \lambda \sum_{j=1}^{p} |\beta_j|^d$$

Si $d = 0$, la penalización consiste en el número de variables utilizadas, por tanto se corresponde con el problema de selección de variables; $d = 1$ se corresponde con LASSO y $d = 2$ con *ridge*.

La ventaja de utilizar LASSO es que va a forzar a que algunos parámetros sean cero, con lo cual también se realiza una selección de las variables más influyentes. Por el contrario, *ridge regression* va a incluir todas las variables predictoras en el modelo final, si bien es cierto que algunas con parámetros muy próximos a cero: de este modo va a reducir el riesgo del sobreajuste, pero no resuelve el problema de la interpretabilidad. Otra posible ventaja de utilizar LASSO es que cuando hay variables predictoras correlacionadas tiene tendencia a seleccionar una y anular las demás (esto también se puede ver como un inconveniente, ya que pequeños cambios en los datos pueden dar lugar a distintos modelos), mientras que *ridge* tiende a darles igual peso.

Dos generalizaciones de LASSO son *least angle regression* (LARS, Efron et al., 2004) y *elastic net* (Zou y Hastie, 2005). *Elastic net* combina las ventajas de *ridge* y LASSO, minimizando

$$\min_{\beta_0,\beta} \text{RSS} + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{p} \beta_j^2 + \alpha \sum_{j=1}^{p} |\beta_j| \right)$$

siendo $0 \leq \alpha \leq 1$, un hiperparámetro adicional que determina la combinación lineal de ambas penalizaciones.

Es muy importante estandarizar (centrar y reescalar) las variables predictoras antes de realizar estas técnicas. Fijémonos en que, así como RSS es insensible a los cambios de escala, la penalización es muy sensible. Previa estandarización, el término independiente β_0 (que no interviene en la penalización) tiene una interpretación muy directa, ya que

$$\hat{\beta}_0 = \bar{y} = \sum_{i=1}^{n} \frac{y_i}{n}$$

Los dos métodos de regularización comentados dependen del hiperparámetro λ (equivalentemente, s). Es muy importante seleccionar adecuadamente el valor del hiperparámetro, por ejemplo utilizando *validación cruzada*. Hay algoritmos muy eficientes que permiten el ajuste, tanto de *ridge regression* como de LASSO de forma conjunta (simultánea) para todos los valores de λ.

6.7.1 Implementación en R

Hay varios paquetes que implementan estos métodos: `h2o`, `elasticnet`, `penalized`, `lasso2`, `biglasso`, etc., pero el paquete `glmnet` utiliza una de las más eficientes.

```
library(glmnet)
```

El paquete `glmnet` no emplea formulación de modelos, hay que establecer la respuesta `y` y la matriz numérica `x` correspondiente a las variables explicativas. Por tanto no se pueden incluir directamente predictores categóricos, habrá que codificarlos empleando variables auxiliares numéricas. Se puede emplear la función `model.matrix()` (o `Matrix::sparse.model.matrix()` si el conjunto de datos es muy grande) para construir la matriz de diseño `x` a partir de una fórmula (alternativamente se pueden emplear la herramientas implementadas en el paquete `caret`). Además, tampoco admite datos faltantes.

La función principal es `glmnet()`:

```
glmnet(x, y, family, alpha = 1, lambda = NULL, ...)
```

- `family`: familia del modelo lineal generalizado (ver Sección 6.9); por defecto `"gaussian"` (modelo lineal con ajuste cuadrático), también admite `"binomial"`, `"poisson"`, `"multinomial"`, `"cox"` o `"mgaussian"` (modelo lineal con respuesta multivariante).
- `alpha`: parámetro α de elasticnet $0 \leq \alpha \leq 1$. Por defecto `alpha = 1` penalización LASSO (`alpha = 0` para *ridge regression*).
- `lambda`: secuencia (opcional) de valores de λ; si no se especifica se establece una secuencia por defecto (en base a los argumentos adicionales `nlambda` y `lambda.min.ratio`). Se devolverán los ajustes para todos los valores de esta secuencia (también se podrán obtener posteriormente para otros valores).

Entre los métodos genéricos disponibles del objeto resultante, `coef()` y `predict()` permiten obtener los coeficientes y las predicciones para un valor concreto de λ, que se debe especificar mediante el argumento `s = valor` ("For historical reasons we use the symbol 's' rather than 'lambda' ").

Aunque para seleccionar el un valor "óptimo" del hiperparámetro λ (mediante validación cruzada) se puede emplear `cv.glmnet()`:

```
cv.glmnet(x, y, family, alpha, lambda, type.measure = "default", nfolds = 10, ...)
```

Esta función también devuelve los ajustes con toda la muestra de entrenamiento (en la componente `$glmnet.fit`) y se puede emplear el resultado directamente para predecir o obtener los coeficientes del modelo. Por defecto seleccionando λ mediante la regla de "un error estándar" de Breiman et al. (1984) (componente `$lambda.1se`), aunque también calcula el valor óptimo (componente `$lambda.min`; que se puede seleccionar con estableciendo `s = "lambda.min"`). Para más detalles consultar la vignette del paquete "An Introduction to glmnet".

Continuaremos con el ejemplo de los datos de clientes de la compañía de distribución industrial HBAT empleado en la Sección 6.1 (donde solo se consideraban predictores numéricos y tampoco había datos faltantes):

```
load("data/hbat.RData")
df <- hbat[c(23, 7:19)]
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
x <- as.matrix(train[-1])
y <- train$fidelida
```

6.7.2 Ejemplo: Ridge Regression

Podemos ajustar modelos de regresión ridge (con la secuencia de valores de λ por defecto) con la función `glmnet()` con `alpha=0` (*ridge penalty*). Con el método `plot()` podemos representar la evolución de

los coeficientes en función de la penalización (etiquetando las curvas con el índice de la variable si `label = TRUE`; ver Figura 6.11).

```
fit.ridge <- glmnet(x, y, alpha = 0)
plot(fit.ridge, xvar = "lambda", label = TRUE)
```

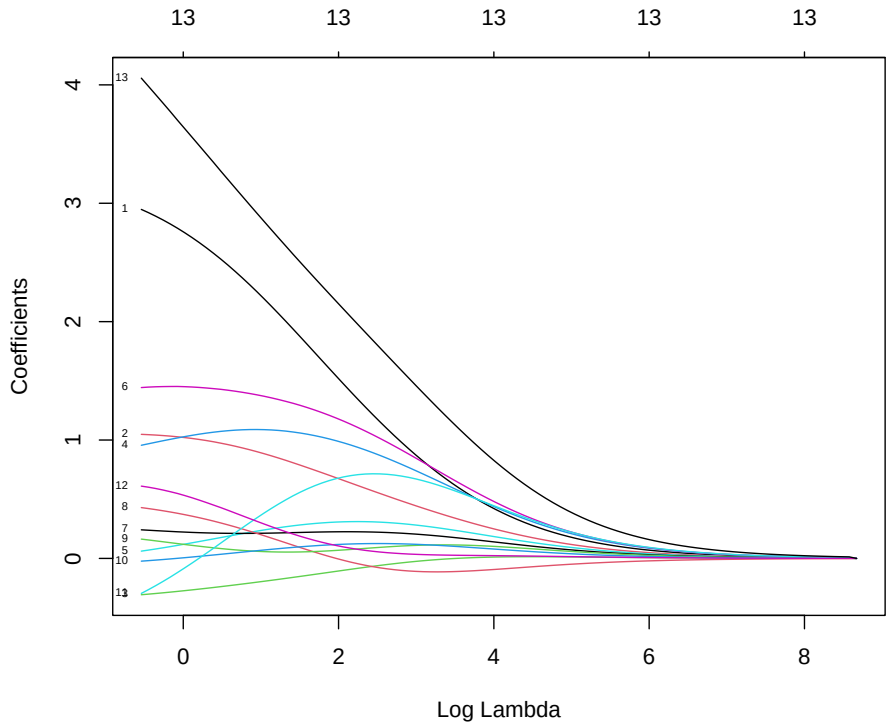


Figura 6.11: Gráfico de perfil de la evolución de los coeficientes en función del logaritmo de la penalización del ajuste ridge.

Podemos obtener el modelo o predicciones para un valor concreto de λ:

```
coef(fit.ridge, s = 2) # lambda = 2
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  3.56806743
## calidadp     2.41027431
## web          0.94414628
## soporte     -0.22183509
## quejas       1.08417665
## publi        0.20121976
## producto     1.41018809
## imgfvent     0.21140360
## precio       0.26171759
## garantia     0.07110803
## nprod        0.04859325
## facturac     0.22695054
## flexprec     0.37732748
## velocida     3.11101217
```

Para seleccionar el parámetro de penalización por validación cruzada empleamos `cv.glmnet()` (normalmente emplearíamos esta función en lugar de `glmnet()`). El correspondiente método `plot()` muestra la evolución de los errores de validación cruzada en función de la penalización, incluyendo las bandas de un error estándar de Breiman (ver Figura 6.12).

```
set.seed(1)
cv.ridge <- cv.glmnet(x, y, alpha = 0)
plot(cv.ridge)
```

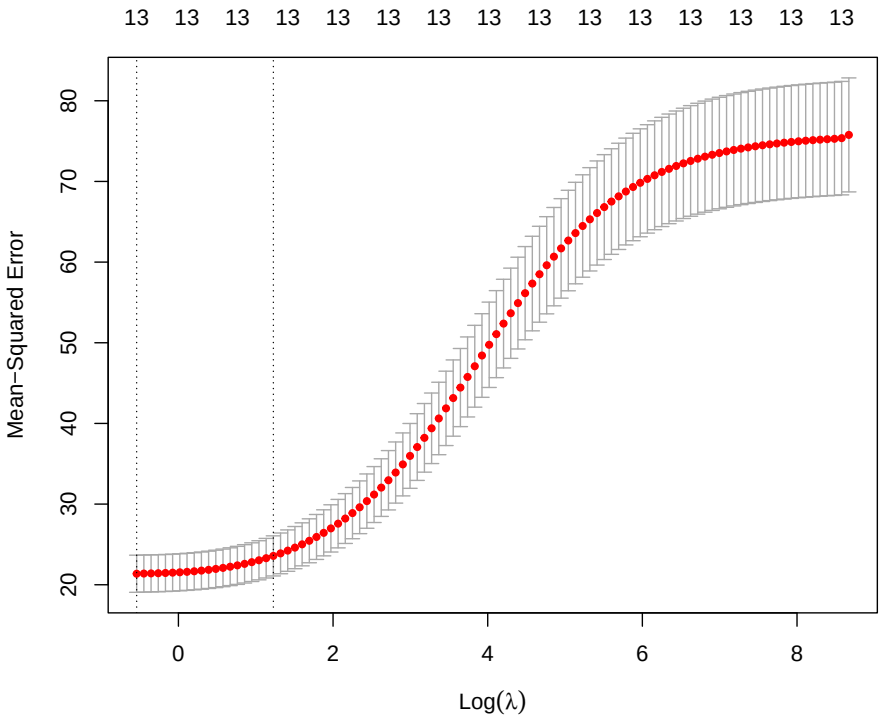


Figura 6.12: Error cuadrático medio de validación cruzada en función del logaritmo de la penalización del ajuste ridge, junto con los intervalos de un error estándar. Las líneas verticales se corresponden con `lambda.min` y `lambda.1se`.

En este caso el parámetro óptimo, según la regla de un error estándar de Breiman, sería[6]:

```
cv.ridge$lambda.1se
```

```
## [1] 3.413705
```

y el correspondiente modelo contiene todas las variables explicativas:

```
coef(cv.ridge) # s = "lambda.1se"
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  8.38314266
## calidadp     2.06713538
## web          0.84771656
## soporte     -0.17674893
## quejas       1.08099022
## publi        0.25926570
## producto     1.34198207
## imgfvent     0.21510001
## precio       0.15194226
## garantia     0.05417865
## nprod        0.08252518
## facturac     0.45964418
## flexprec     0.24646749
```

[6]Para obtener el valor óptimo global podemos emplear `cv.ridge$lambda.min`, y añadir el argumento `s = "lambda.min"` a los métodos `coef()` y `predict()` para obtener los correspondientes coeficientes y predicciones.

```
## velocida     2.70697234
```

Finalmente evaluamos la precisión en la muestra de test:

```
obs <- test$fidelida
newx <- as.matrix(test[, -14])
pred <- predict(cv.ridge, newx = newx) # s = "lambda.1se"
accuracy(pred, obs)
```

```
##        me      rmse       mae       mpe      mape r.squared
## -108.0747  108.9324  108.0747 -186.9472  186.9472 -115.3046
```

6.7.3 Ejemplo: Lasso

También podríamos ajustar modelos LASSO con la opción por defecto de `glmnet()` (`alpha = 1`, *lasso penalty*). Pero en este caso lo haremos al mismo tiempo que seleccionamos el parámetro de penalización por validación cruzada (ver Figura 6.13):

```
set.seed(1)
cv.lasso <- cv.glmnet(x,y)
plot(cv.lasso)
```

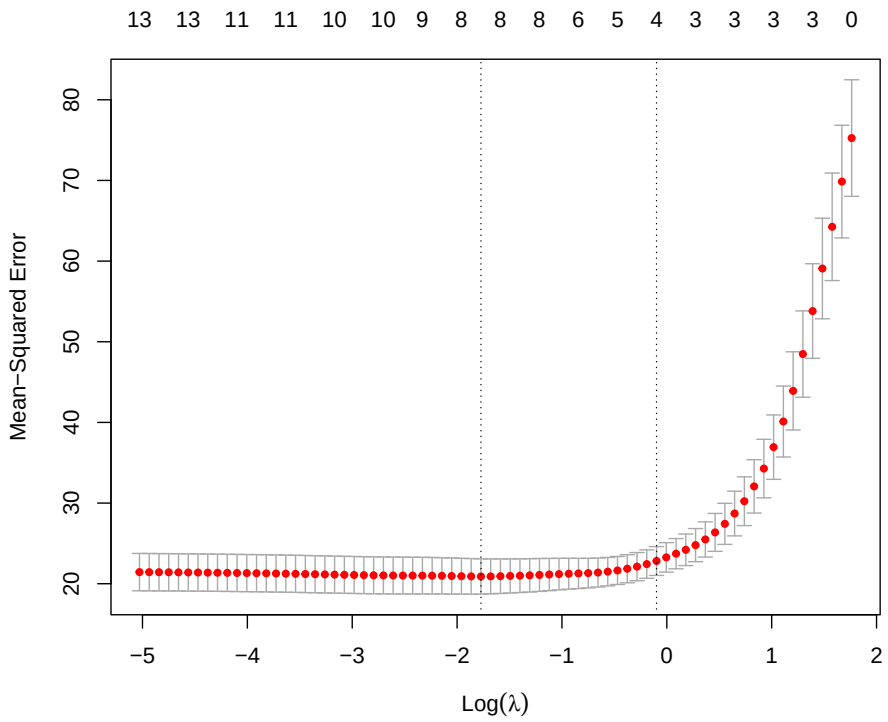


Figura 6.13: Error cuadrático medio de validación cruzada en función del logaritmo de la penalización del ajuste LASSO, junto con los intervalos de un error estándar. Las líneas verticales se corresponden con `lambda.min` y `lambda.1se`.

También podemos generar el gráfico con la evolución de los componentes a partir del ajuste almacenado en la componente `$glmnet.fit`:

```
plot(cv.lasso$glmnet.fit, xvar = "lambda", label = TRUE)
abline(v = log(cv.lasso$lambda.1se), lty = 2)
abline(v = log(cv.lasso$lambda.min), lty = 3)
```

Como podemos observar en la Figura 6.14, la penalización LASSO tiende a forzar que las estimaciones de los coeficientes sean exactamente cero cuando el parámetro de penalización λ es suficientemente grande. En este caso, el modelo resultante (empleando la regla *oneSE*) solo contiene 4 variables

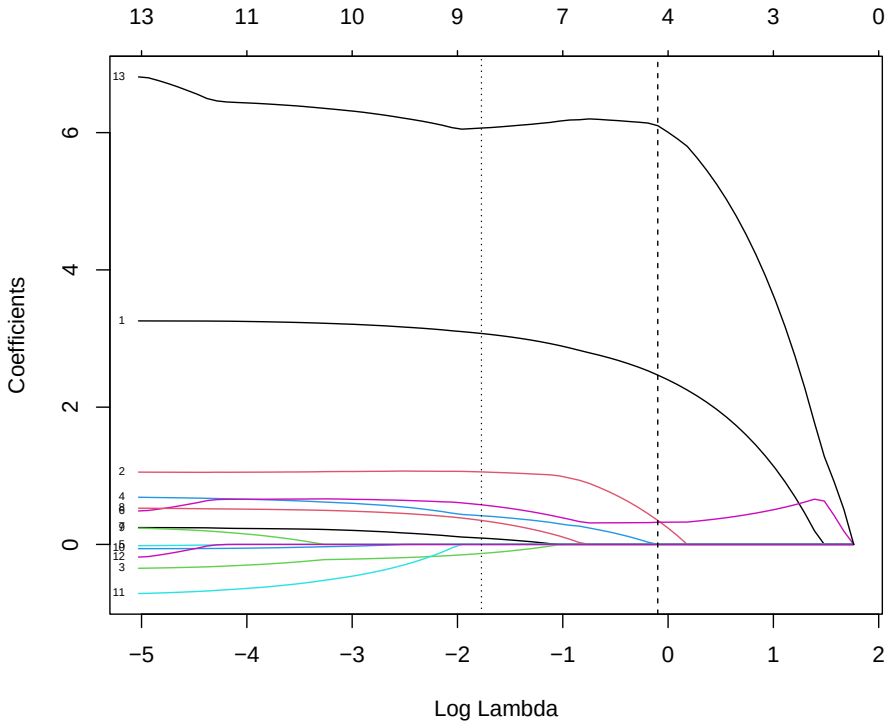


Figura 6.14: Evolución de los coeficientes en función del logaritmo de la penalización del ajuste LASSO. Las líneas verticales se corresponden con `lambda.min` y `lambda.1se`.

explicativas:

```
coef(cv.lasso) # s = "lambda.1se"
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept) 12.0485398
## calidadp     2.4673862
## web          0.3498592
## soporte      .
## quejas       .
## publi        .
## producto     0.3227830
## imgfvent     .
## precio       .
## garantia     .
## nprod        .
## facturac     .
## flexprec     .
## velocida     6.1011015
```

Por tanto este método también podría ser empleando para la selección de variables (puede hacerse automáticamente; estableciendo `relax = TRUE`, en la llamada a `glmnet()` o `cv.glmnet()`, devolverá los modelos ajustados sin regularización en la componente `$relaxed`).

Finalmente evaluamos también la precisión en la muestra de test:

```
pred <- predict(cv.lasso, newx = newx)
accuracy(pred, obs)
```

```
##         me      rmse       mae       mpe      mape r.squared
## -129.5991  130.8208  129.5991 -223.8971  223.8971 -166.7400
```

6.7.4 Ejemplo: Elastic Net

Podemos ajustar modelos *elastic net* para un valor concreto de `alpha` empleando la función `glmnet()`, pero las opciones del paquete no incluyen la selección de este hiperparámetro. Aunque se podría implementar fácilmente (como se muestra en `help(cv.glmnet)`), resulta mucho más cómodo emplear el método `"glmnet"` de `caret`:

```
library(caret)
modelLookup("glmnet")
```

```
##    model parameter                    label forReg forClass probModel
## 1 glmnet     alpha        Mixing Percentage   TRUE     TRUE      TRUE
## 2 glmnet    lambda Regularization Parameter   TRUE     TRUE      TRUE
```

```
set.seed(1)
# Se podría emplear train(fidelida ~ ., data = train, ...)
caret.glmnet <- train(x, y, method = "glmnet",
    preProc = c("zv", "center", "scale"),
    trControl = trainControl(method = "cv", number = 5),
    tuneLength = 5)
caret.glmnet
```

```
## glmnet
##
## 160 samples
##  13 predictor
##
## Pre-processing: centered (13), scaled (13)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 129, 129, 127, 127, 128
## Resampling results across tuning parameters:
##
##   alpha  lambda       RMSE      Rsquared   MAE
##   0.100  0.005410605  4.581364  0.7148069  3.414825
##   0.100  0.025113806  4.576940  0.7153275  3.408862
##   0.100  0.116567961  4.545239  0.7187940  3.361951
##   0.100  0.541060544  4.474562  0.7284099  3.295198
##   0.100  2.511380580  4.704072  0.7187452  3.594686
##   0.325  0.005410605  4.573738  0.7157479  3.408931
##   0.325  0.025113806  4.564560  0.7167890  3.397543
##   0.325  0.116567961  4.500833  0.7241961  3.326005
##   0.325  0.541060544  4.438653  0.7349191  3.306102
##   0.325  2.511380580  4.881621  0.7184709  3.757854
##   0.550  0.005410605  4.573800  0.7157344  3.411370
##   0.550  0.025113806  4.552473  0.7182118  3.386635
##  [ reached getOption("max.print") -- omitted 13 rows ]
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 1 and lambda = 0.116568.
```

Los resultados de la selección de los hiperparámetros α y λ de regularización se muestran en la Figura 6.15:

```
ggplot(caret.glmnet, highlight = TRUE)
```

Finalmente, se evalúan las predicciones en la muestra de test del modelo ajustado (que en esta ocasión mejoran los resultados del modelo LASSO ajustando en la sección anterior):

```
pred <- predict(caret.glmnet, newdata = test)
accuracy(pred, obs)
```

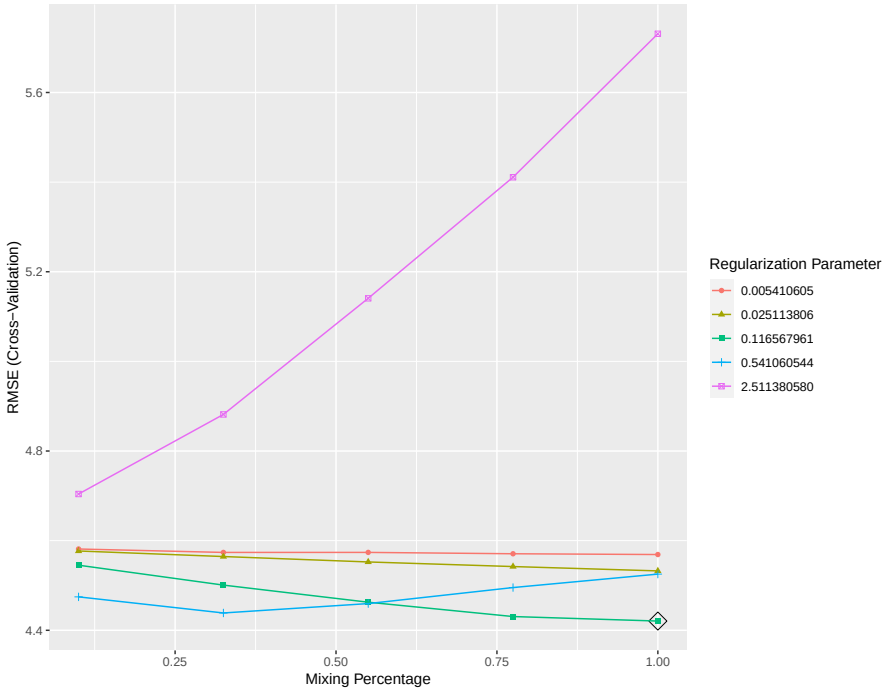


Figura 6.15: Errores RMSE de validación cruzada de los modelos *elastic net* en función de los hiperparámetros de regularización.

```
##          me        rmse         mae         mpe        mape   r.squared
##  0.49843131  4.28230543  3.43805155 -0.02851825  6.15711131  0.82026277
```

6.8 Métodos de reducción de la dimensión

Otra alternativa, para tratar de reducir la varianza de los modelos lineales, es transformar los predictores considerando $k < p$ combinaciones lineales:

$$Z_j = a_{1j}X_1 + a_{2j}X_2 + ... + a_{pj}X_p$$

con $j = 1, ..., k$, denominadas componentes (o variables latentes), y posteriormente ajustar un modelo de regresión lineal empleándolas como nuevos predictores:

$$Y = \alpha_0 + \alpha_1 Z_1 + ... + \alpha_k Z_k + \varepsilon$$

Adicionalmente, si se seleccionan los coeficientes a_{ji} (denominados *cargas* o *pesos*) de forma que

$$\sum_{i=1}^{p} a_{ij}a_{il} = 0, \text{ si } j \neq l,$$

las componentes serán ortogonales y se evitarán posibles problemas de colinealidad. De esta forma se reduce la dimensión del problema, pasando de $p+1$ a $k+1$ coeficientes a estimar, lo cual en principio reducirá la varianza, especialmente si p es grande en comparación con n. Por otra parte, también podríamos expresar el modelo final en función de los predictores originales, con coeficientes:

$$\beta_i = \sum_{j=1}^{k} \alpha_j a_{ij}$$

Es decir, se ajusta un modelo lineal con restricciones, lo que en principio incrementará el sesgo (si $k = p$ sería equivalente a ajustar un modelo lineal sin restricciones). Además, podríamos interpretar

los coeficientes α_j como los efectos de las componentes del modo tradicional, pero resultaría más complicado interpretar los efectos de los predictores originales.

También hay que tener en cuenta que al considerar combinaciones lineales, si las hipótesis estructurales de linealidad, homocedasticidad, normalidad o independencia no son asumibles en el modelo original, es de esperar que tampoco lo sean en el modelo transformado (se podrían emplear las herramientas descritas en la Sección 6.4 para su análisis).

Hay una gran variedad de algoritmos para obtener estas componentes, en esta sección consideraremos las dos aproximaciones más utilizadas: componentes principales y mínimos cuadrados parciales. También hay numerosos paquetes de R que implementan métodos de este tipo (`pls`, `plsRglm`…), incluyendo `caret`.

6.8.1 Regresión por componentes principales (PCR)

Una de las aproximaciones tradicionales, cuando se detecta la presencia de colinealidad, consiste en aplicar el método de componentes principales a los predictores. El análisis de componentes principales (*principal component analysis*, PCA) es un método muy utilizado de aprendizaje no supervisado, que permite reducir el número de dimensiones tratando de recoger la mayor parte de la variabilidad de los datos originales, en este caso de los predictores (para más detalles sobre PCA ver por ejemplo el Capítulo 10 de James et al., 2021).

Al aplicar PCA a los predictores $X_1, \ldots, X_p$ se obtienen componentes ordenados según la variabilidad explicada de forma descendente. El primer componente es el que recoge el mayor porcentaje de la variabilidad total (se corresponde con la dirección de mayor variación de las observaciones). Las siguientes componentes se seleccionan entre las direcciones ortogonales a las anteriores y de forma que recojan la mayor parte de la variabilidad restante. Además estas componentes son normalizadas, de forma que:

$$\sum_{i=1}^{p} a_{ij}^2 = 1$$

(se busca una transformación lineal ortonormal). En la práctica esto puede llevarse a cabo fácilmente a partir de la descomposición espectral de la matriz de covarianzas muestrales, aunque normalmente se estandarizan previamente los datos (i.e., se emplea la matriz de correlaciones). Por tanto, si se pretende emplear estas componentes para ajustar un modelo de regresión, habrá que conservar los parámetros de estas transformaciones para poder aplicarlas a nuevas observaciones.

Normalmente se seleccionan las primeras k componentes de forma que expliquen la mayor parte de la variabilidad de los datos (los predictores en este caso). En PCR (*principal component regression*, Massy, 1965) se confía en que estas componentes recojan también la mayor parte de la información sobre la respuesta, pero podría no ser el caso.

Como ejemplo continuaremos con los datos de clientes de la compañía de distribución industrial HBAT. Aunque podríamos emplear las funciones `printcomp()` y `lm()` del paquete base, emplearemos por comodidad la función `pcr()` del paquete `pls`, ya que incorpora validación cruzada para seleccionar el número de componentes y facilita el cálculo de nuevas predicciones. Los principales argumentos de esta función son:

```
pcr(formula, ncomp, data, scale = FALSE, center = TRUE,
    validation = c("none", "CV", "LOO"), segments = 10,
    segment.type = c("random", "consecutive", "interleaved"), ...)
```

- `ncomp`: número máximo de componentes (ajustará modelos desde 1 hasta `ncomp` componentes).
- `scale`, `center`: normalmente valores lógicos indicando si los predictores serán reescalados (divididos por su desviación estándar y centrados (restando su media).
- `validation`: determina el tipo de validación, puede ser `"none"` (ninguna, se empleará el modelo ajustado con toda la muestra de entrenamiento), `"LOO"` (VC dejando uno fuera) y `"CV"` (VC por grupos). En este último caso los grupos de validación se especifican mediante `segments` (número

de grupos) y `segment.type` (por defecto aleatorios; para más detalles consultar la ayuda de `mvrCv()`).

Como ejemplo continuaremos con los datos de clientes de la compañía de distribución industrial HBAT. Reescalaremos los predictores y emplearemos validación cruzada por grupos para seleccionar el número de componentes:

```
library(pls)
set.seed(1)
pcreg <- pcr(fidelida ~ ., data = train, scale = TRUE, validation = "CV")
```

Podemos obtener un resumen de los resultados de validación (evolución de los errores de validación cruzada) y del ajuste en la muestra de entrenamiento (evolución de la proporción de variabilidad explicada de los predictores y de la respuesta) con el método `summary()`:

```
summary(pcreg)
```

```
## Data:  X dimension: 160 13
##    Y dimension: 160 1
## Fit method: svdpc
## Number of components considered: 13
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           8.683    6.892    5.960    5.695    5.448    5.525    4.901
## adjCV        8.683    6.888    5.954    5.630    5.440    5.517    4.846
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       4.930    4.880    4.550     4.575     4.555     4.579     4.573
## adjCV    4.916    4.862    4.535     4.560     4.539     4.561     4.554
##
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X           29.40    50.38    63.09    75.38    82.93    87.33    91.02
## fidelida    37.89    53.76    58.84    61.79    61.96    70.56    70.97
##           8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## X           94.14    96.39     97.86     99.00     99.93    100.00
## fidelida    72.13    75.12     75.12     75.78     76.00     76.14
```

Aunque suele resultar más cómodo representar gráficamente estos valores (ver Figura 6.16). Por ejemplo empleando `RMSEP()` para acceder a los errores de validación[7]:

```
# validationplot(pcreg, legend = "topright")
rmsep.cv <- RMSEP(pcreg)
plot(rmsep.cv, legend = "topright")
```

```
ncomp.op <- with(rmsep.cv, comps[which.min(val[2, 1, ])]) # mínimo adjCV RMSEP
```

En este caso, empleando el criterio de menor error de validación cruzada se seleccionaría un número elevado de componentes, el mínimo se alcanzaría con 9 componentes (bastante próximo a ajustar un modelo lineal con todos los predictores).

Los coeficientes de los predictores originales con el modelo seleccionado serían[8]:

```
coef(pcreg, ncomp = 9, intercept = TRUE)
```

```
## , , 9 comps
##
```

[7] `"adjCV"` es una estimación de validación cruzada con corrección de sesgo.

[8] También se pueden analizar distintos aspectos del ajuste (predicciones, coeficientes, puntuaciones, cargas, biplots, cargas de correlación o gráficos de validación) con el método `plot.mvr()`.

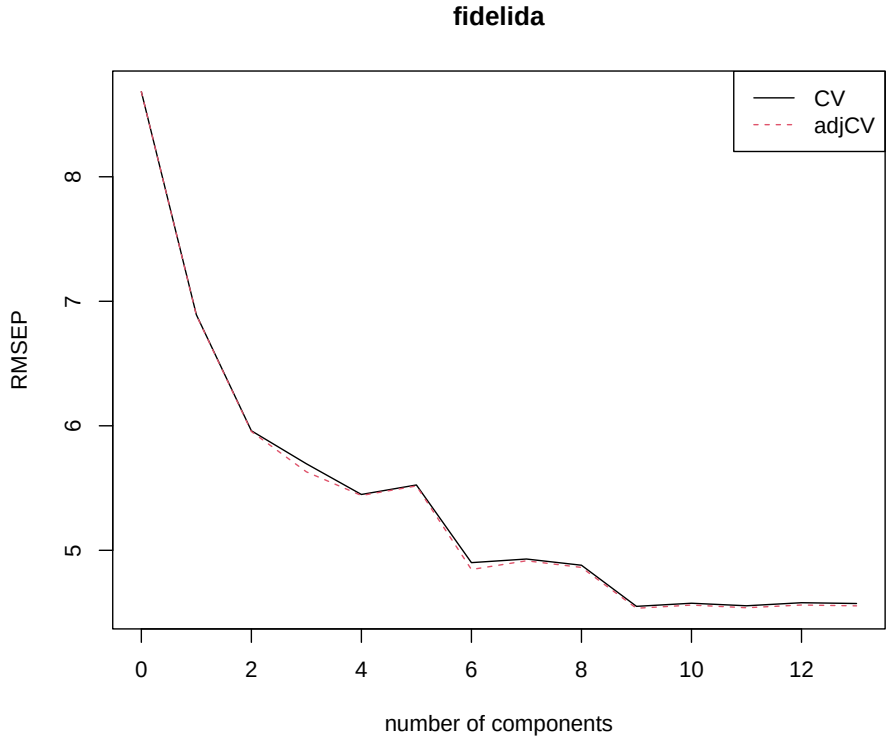


Figura 6.16: Errores de validación cruzada en función del número de componentes en el ajuste mediante PCR.

```
##                fidelida
## (Intercept) -5.33432555
## calidadp     4.53450998
## web          1.36586619
## soporte     -0.08892573
## quejas       2.21875583
## publi        0.16238769
## producto     1.77457778
## imgfvent    -0.20518565
## precio       0.83775389
## garantia    -0.32313633
## nprod        0.07569817
## facturac    -0.45633670
## flexprec     0.72941138
## velocida     2.27911181
```

Finalmente evaluamos su precisión:

```
pred <- predict (pcreg , test, ncomp = 9)
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
## 0.54240746 4.39581553 3.46755308 0.08093351 6.15004687 0.81060798
```

Alternativamente podríamos emplear el método `"pcr"` de `caret`. Por ejemplo seleccionando el número de componentes mediante la regla de un error estándar (ver Figura 6.17):

```
library(caret)
modelLookup("pcr")
```

```
##   model parameter       label forReg forClass probModel
## 1   pcr     ncomp #Components   TRUE    FALSE     FALSE
```

```
set.seed(1)
trControl <- trainControl(method = "cv", number = 10, selectionFunction = "oneSE")
caret.pcr <- train(fidelida ~ ., data = train, method = "pcr",
                   preProcess = c("zv", "center", "scale"),
                   trControl = trControl, tuneGrid = data.frame(ncomp = 1:10))
caret.pcr
```

```
## Principal Component Analysis
##
## 160 samples
##  13 predictor
##
## Pre-processing: centered (13), scaled (13)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 145, 143, 144, 143, 145, 144, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE      Rsquared   MAE
##    1     6.844663  0.3889234  5.680556
##    2     5.894889  0.5446242  4.795170
##    3     5.604429  0.5837430  4.588714
##    4     5.403949  0.6077034  4.374244
##    5     5.476941  0.5984578  4.392604
##    6     4.806541  0.6891701  3.772401
##    7     4.830222  0.6886805  3.780234
##    8     4.825438  0.6895724  3.736469
##    9     4.511374  0.7295536  3.371115
##   10     4.551134  0.7260848  3.405876
##
## RMSE was used to select the optimal model using  the one SE rule.
## The final value used for the model was ncomp = 6.
```

```
ggplot(caret.pcr, highlight = TRUE)
```

```
pred <- predict(caret.pcr, newdata = test)
accuracy(pred, obs)
```

```
##        me      rmse       mae       mpe      mape r.squared
## 0.9287609 5.0118766 4.0669487 0.6031423 7.2300043 0.7538026
```

Al incluir más componentes se aumenta la proporción de variabilidad explicada de los predictores, pero esto no está relacionado con su utilidad para explicar la respuesta. No va a haber problemas de colinealidad aunque incluyamos muchas componentes, pero se tendrán que estimar más coeficientes y va a disminuir su precisión. Sería más razonable obtener las componentes principales y después aplicar un método de selección. Por ejemplo podemos combinar el método de preprocesado `"pca"` de `caret` con un método de selección de variables[9] (ver Figura 6.18):

```
set.seed(1)
caret.pcrsel <- train(fidelida ~ ., data = train, method = "leapSeq",
                   preProcess = c("zv", "center", "scale", "pca"),
                   trControl = trControl, tuneGrid = data.frame(nvmax = 1:10))
caret.pcrsel
```

```
## Linear Regression with Stepwise Selection
##
## 160 samples
```

[9]Esta forma de proceder se podría emplear con otros modelos que puedan tener problemas de colinealidad, como los lineales generalizados.

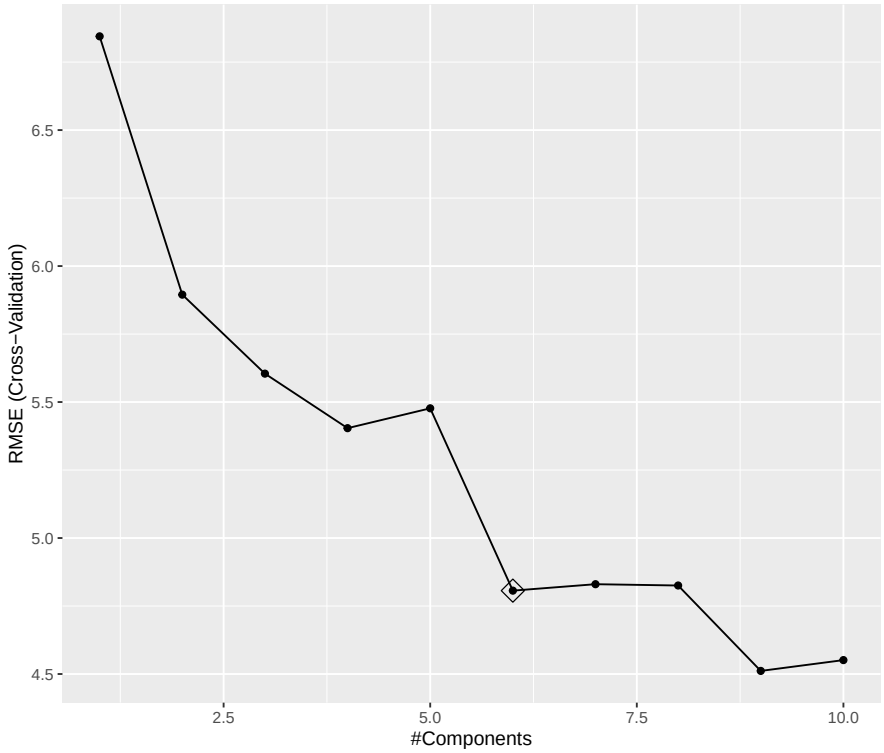


Figura 6.17: Errores de validación cruzada en función del número de componentes en el ajuste mediante PCR y valor óptimo según la regla de un error estándar.

```
##  13 predictor
##
## Pre-processing: centered (13), scaled (13), principal component
##  signal extraction (13)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 145, 143, 144, 143, 145, 144, ...
## Resampling results across tuning parameters:
##
##   nvmax  RMSE      Rsquared   MAE
##    1     6.844663  0.3889234  5.680556
##    2     5.894889  0.5446242  4.795170
##    3     5.626635  0.5780222  4.614693
##    4     4.965728  0.6639455  4.041916
##    5     4.829841  0.6864472  3.782061
##    6     4.666785  0.7085316  3.558379
##    7     4.545961  0.7276881  3.437428
##    8     4.642381  0.7140237  3.518435
##    9     4.511374  0.7295536  3.371115
##   10     4.511374  0.7295536  3.371115
##
## RMSE was used to select the optimal model using  the one SE rule.
## The final value used for the model was nvmax = 6.
```

```
ggplot(caret.pcrsel, highlight = TRUE)
```

```
with(caret.pcrsel, coef(finalModel, bestTune$nvmax))
```

```
## (Intercept)         PC1         PC2         PC3         PC4         PC5
##  58.0375000   2.7256200  -2.0882164   1.5167199  -1.1761229  -0.3588877
##         PC6
```

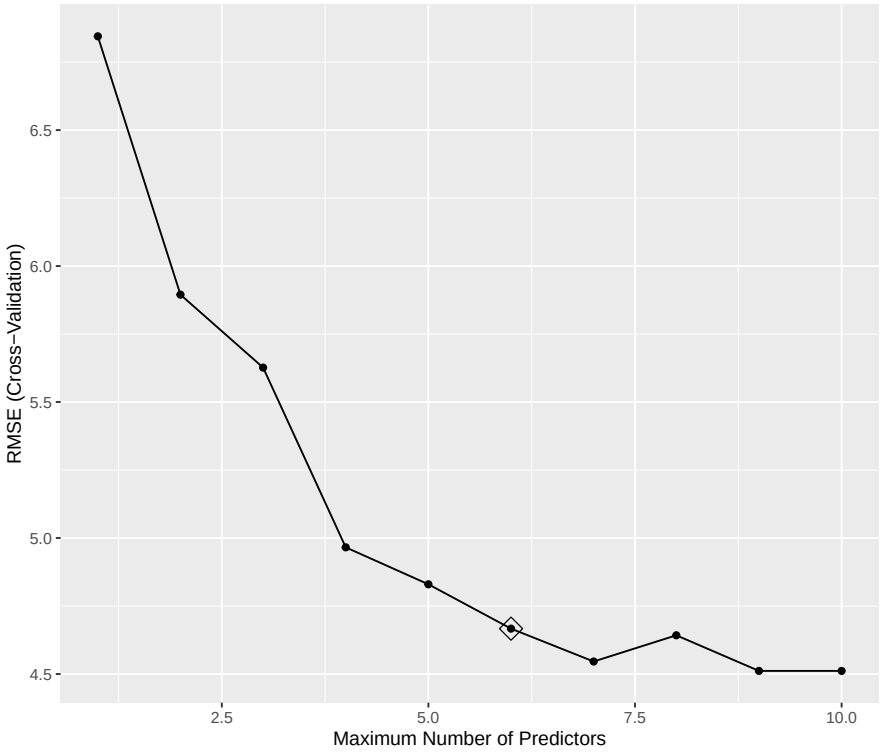


Figura 6.18: Errores de validación cruzada en función del número de componentes en el ajuste mediante PCR con selección por pasos y valor óptimo según la regla de un error estándar.

```
##   3.3571851
```

```
pred <- predict(caret.pcrsel, newdata = test)
accuracy(pred, obs)
```

```
##        me      rmse       mae       mpe      mape r.squared
## 0.9287609 5.0118766 4.0669487 0.6031423 7.2300043 0.7538026
```

6.8.2 Regresión por mínimos cuadrados parciales (PLSR)

Como ya se comentó, en PCR las componentes se determinan con el objetivo de explicar la variabilidad de los predictores, ignorando por completo la respuesta. Por el contrario, en PLSR (*partial least squares regression*, Wold et al., 1983) se construyen las componentes $Z_1, \ldots, Z_k$ teniendo en cuenta desde un principio el objetivo final de predecir linealmente la respuesta.

Hay varios procedimientos para seleccionar los pesos a_{ij}, pero la idea es asignar mayor peso a los predictores que están más correlacionados con la respuesta (o con los correspondientes residuos al ir obteniendo nuevos componentes), considerando siempre direcciones ortogonales (ver por ejemplo la Sección 6.3.2 de James et al., 2021).

Continuando con el ejemplo anterior, emplearemos en primer lugar la función `plsr()` del paquete `pls`, que tiene los mismos argumentos que la función `pcr()` descrita en la sección anterior[10]:

```
set.seed(1)
plsreg <- plsr(fidelida ~ ., data = train, scale = TRUE, validation = "CV")
summary(plsreg)
```

```
## Data:  X dimension: 160 13
```

[10]Realmente ambas funciones llaman internamente a `mvr()` donde están implementadas distintas proyecciones (ver `help(pls.options)`, o Mevik y Wehrens, 2007).

```
##    Y dimension: 160 1
## Fit method: kernelpls
## Number of components considered: 13
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           8.683    5.439    4.952    4.684    4.588    4.560    4.576
## adjCV        8.683    5.433    4.945    4.670    4.571    4.542    4.558
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       4.572    4.572    4.580     4.563     4.584     4.574     4.573
## adjCV    4.555    4.554    4.562     4.545     4.564     4.555     4.554
##
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X           27.32    44.59    56.70    66.70    70.29    79.07    86.27
## fidelida    62.09    69.84    73.96    75.43    75.89    75.96    76.00
##           8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## X           90.98    93.83     95.23     97.59     98.53    100.00
## fidelida    76.03    76.04     76.09     76.12     76.14     76.14
```

```
# validationplot(plsreg, legend = "topright")
rmsep.cv <- RMSEP(plsreg)
plot(rmsep.cv, legend = "topright")
```

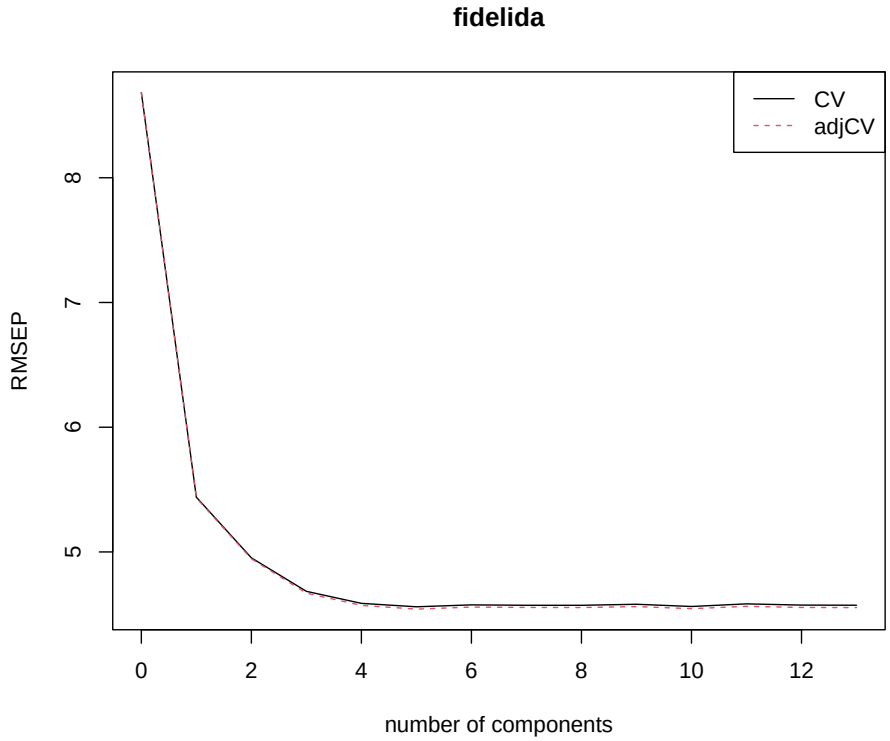


Figura 6.19: Error cuadrático medio de validación cruzada en función del número de componentes en el ajuste mediante PLS.

```
ncomp.op <- with(rmsep.cv, comps[which.min(val[2, 1, ])]) # mínimo adjCV RMSEP
```

En este caso el mínimo se alcanza con 5 componentes pero, a la vista de la Figura 6.19, parece que 4 sería un valor razonable. Podríamos obtener los coeficientes de los predictores del modelo seleccionado:

```
coef(plsreg, ncomp = 4, intercept = TRUE)
```

```
## , , 4 comps
##
##                fidelida
## (Intercept) -8.29699768
## calidadp     4.63190425
## web          0.94145378
## soporte     -0.40668608
## quejas       1.53320821
## publi        0.09443857
## producto     1.94009838
## imgfvent     0.11088335
## precio       1.18534756
## garantia     0.07078050
## nprod       -0.08382899
## facturac    -0.03154511
## flexprec     0.63288585
## velocida     2.53362229
```

y evaluar su precisión:

```
pred <- predict(plsreg , test, ncomp = 4)
accuracy(pred, obs)
```

```
##        me      rmse       mae       mpe      mape r.squared
## 0.5331010 4.4027291 3.4983343 0.0461853 6.2529706 0.8100118
```

Empleamos también el método `"pls"` de `caret` seleccionando el número de componentes mediante la regla de un error estándar(ver Figura 6.20):

```
modelLookup("pls")
```

```
##   model parameter       label forReg forClass probModel
## 1   pls     ncomp #Components   TRUE     TRUE      TRUE
```

```
set.seed(1)
caret.pls <- train(fidelida ~ ., data = train, method = "pls",
                   preProcess = c("zv", "center", "scale"),
                   trControl = trControl, tuneGrid = data.frame(ncomp = 1:10))
caret.pls
```

```
## Partial Least Squares
##
## 160 samples
##  13 predictor
##
## Pre-processing: centered (13), scaled (13)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 145, 143, 144, 143, 145, 144, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE      Rsquared   MAE
##    1     5.385375  0.6130430  4.301648
##    2     4.902373  0.6765146  3.882695
##    3     4.630757  0.7151341  3.472635
##    4     4.516718  0.7278875  3.356058
##    5     4.480285  0.7320425  3.391015
##    6     4.490064  0.7314898  3.376068
```

```
##     7     4.478319  0.7323472  3.365828
##     8     4.490064  0.7312432  3.384096
##     9     4.492672  0.7308291  3.379606
##    10     4.483548  0.7316750  3.368064
##
## RMSE was used to select the optimal model using  the one SE rule.
## The final value used for the model was ncomp = 3.
```

```
ggplot(caret.pls, highlight = TRUE)
```

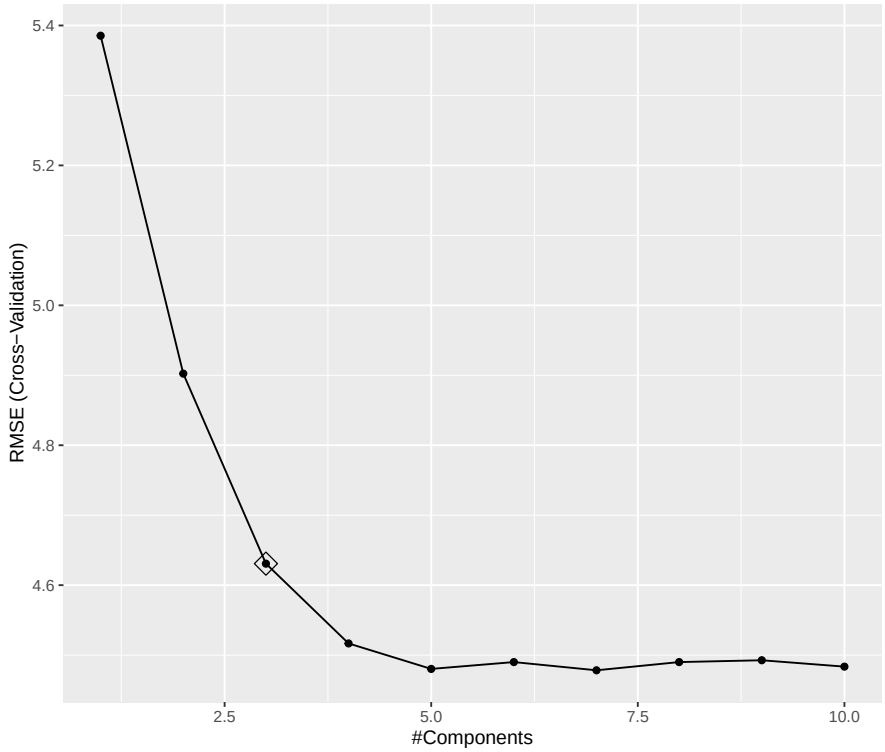


Figura 6.20: Errores de validación cruzada en función del número de componentes en el ajuste mediante PLS.

```
pred <- predict(caret.pls, newdata = test)
accuracy(pred, obs)
```

```
##        me      rmse       mae       mpe      mape r.squared
## 0.8028426 4.6565051 3.7577014 0.4673422 6.6186253 0.7874785
```

Como comentario final, en la práctica se suelen obtener resultados muy similares empleando PCR, PLSR o *ridge regression*.

6.9 Modelos lineales generalizados

Como ya se comentó, los modelos lineales generalizados son una extensión de los modelos lineales para el caso de que la distribución condicional de la variable respuesta no sea normal, introduciendo una función de enlace (o link) g de forma que

$$g\left(E(Y|\mathbf{X})\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

y su ajuste en la práctica se realiza empleando el método de máxima verosimilitud (habrá que especificar también una familia de distribuciones para la respuesta).

La función link debe ser invertible, de forma que se pueda volver a transformar el modelo ajustado (en la escala lineal de las puntuaciones) a la escala original. Por ejemplo, como se comentó al final de

la Sección 1.2.1, para modelar una variable indicadora, con distribución de Bernouilli (caso particular de la Binomial) donde $E(Y|\mathbf{X}) = p(\mathbf{X})$ es la probabilidad de éxito, podemos considerar la función logit

$$\text{logit}(p(\mathbf{X})) = \log\left(\frac{p(\mathbf{X})}{1-p(\mathbf{X})}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

(que proyecta el intervalo $[0, 1]$ en $\mathbb{R}$), siendo su inversa la función logística

$$p(\mathbf{X}) = \frac{e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_p X_p}}$$

Esto da lugar al modelo de regresión logística (múltiple), que será el que utilizaremos como ejemplo en esta sección. Para un tratamiento más completo de los métodos de regresión lineal generalizada se recomienda consultar alguno de los libros de referencia, como Faraway (2016), T. J. Hastie y Pregibon (2017), Dunn y Smyth (2018) o McCullagh (2019).

Para el ajuste (estimación de los parámetros) de un modelo lineal generalizado a un conjunto de datos (por máxima verosimilitud) se emplea la función `glm()` (la mayoría de los principales parámetros coinciden con los de la función `lm()`):

```
ajuste <- glm(formula, family = gaussian, data, weights, subset, na.action, ...)
```

El parámetro `family` especifica la distribución y opcionalmente la función de enlace. Por ejemplo:

- `gaussian(link = "identity")`, `gaussian(link = "log")`
- `binomial(link = "logit")`, `binomial(link = "probit")`
- `poisson(link = "log")`
- `Gamma(link = "inverse")`

Para cada distribución se toma por defecto una función de enlace (el denominado *enlace canónico*, mostrada en primer lugar en la lista anterior; ver `help(family)` para más detalles). Por ejemplo, en el caso del modelo logístico bastará con establecer `family = binomial`.

También se podría emplear la función `bigglm()` del paquete `biglm` para ajustar modelos lineales generalizados a grandes conjuntos de datos, aunque en este caso los requerimientos computacionales pueden ser mayores.

Como se comentó en la Sección 6.1, muchas de las herramientas y funciones genéricas disponibles para los modelos lineales son válidas también para este tipo de modelos, como por ejemplo las mostradas en las tablas 6.1 y 6.2.

Como ejemplo continuaremos con los datos de clientes de la compañía de distribución industrial HBAT, pero consideraremos como respuesta la variable *alianza* y como predictores las percepciones de HBAT (al igual que en las secciones anteriores consideraremos únicamente variables explicativas continuas, sin interacciones, por comodidad).

```
df <- hbat[c(24, 7:19)]
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
```

En primer lugar se suele realizar un análisis descriptivo. Por ejemplo, si el número de variables no es muy grande, podemos generar un gráfico de dispersión matricial, diferenciando las observaciones pertenecientes a las distintas clases (ver Figura 6.21).

```
plot(train[-1], pch = as.numeric(train$alianza), col = as.numeric(train$alianza))
```

Para ajustar un modelo de regresión logística bastaría con establecer el argumento `family = binomial` en la llamada a `glm()` (por defecto utiliza `link = "logit"`):

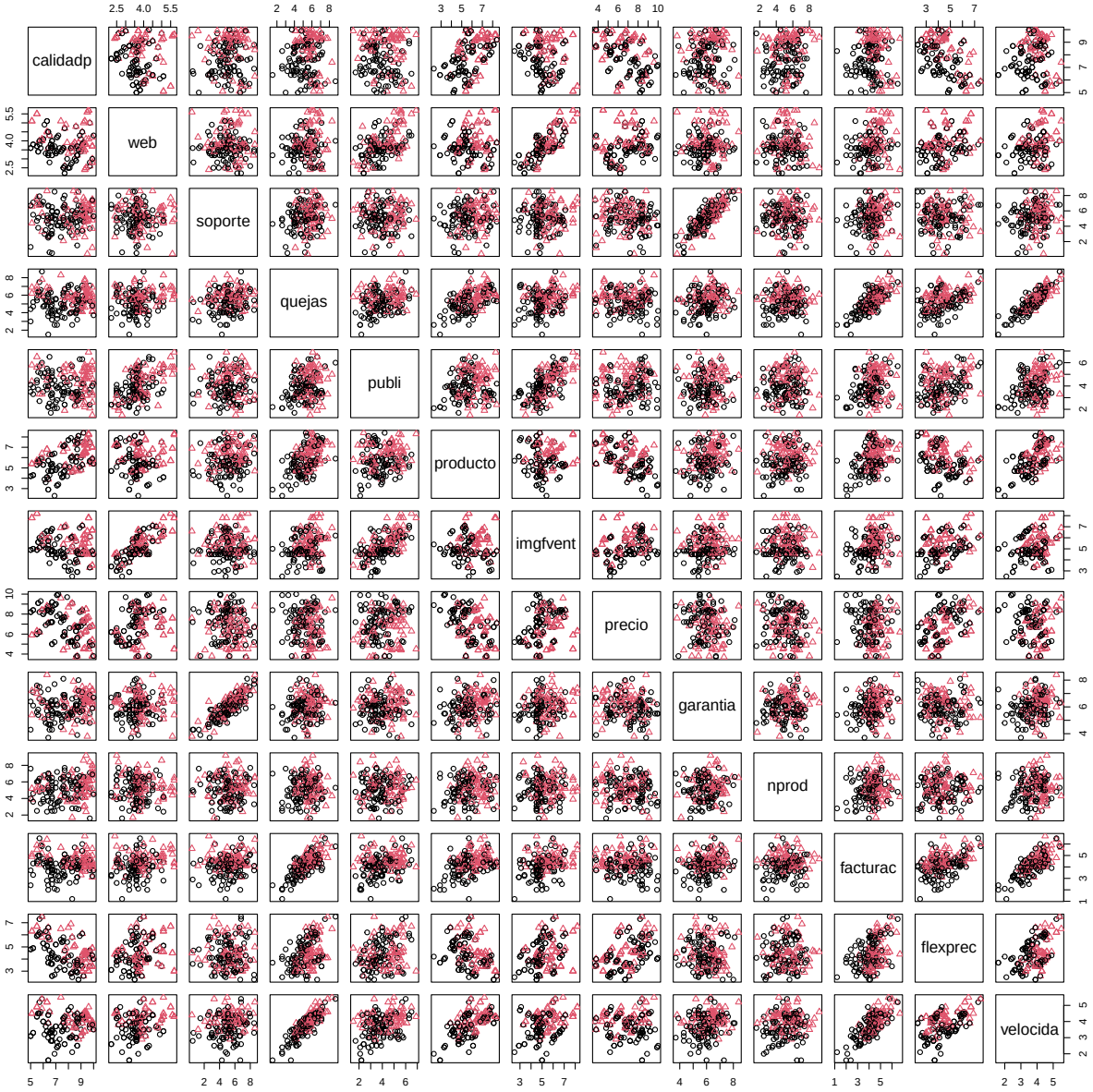


Figura 6.21: Gráfico de dispersión matricial, con colores y símbolos dependiendo de `alianza`.

```
modelo <- glm(alianza ~ velocida + calidadp, family = binomial, data = train)
modelo
```

```
##
## Call:  glm(formula = alianza ~ velocida + calidadp, family = binomial,
##     data = train)
##
## Coefficients:
## (Intercept)     velocida     calidadp
##    -12.5218       1.6475       0.7207
##
## Degrees of Freedom: 159 Total (i.e. Null);  157 Residual
## Null Deviance:     218.2
## Residual Deviance: 160.5   AIC: 166.5
```

La razón de ventajas (OR) permite cuantificar el efecto de las variables explicativas en la respuesta

(incremento proporcional en la razón entre la probabilidad de éxito y la de fracaso, al aumentar una unidad la variable manteniendo las demás fijas):

```
exp(coef(modelo))  # Razones de ventajas ("odds ratios")
```

```
##  (Intercept)     velocida     calidadp
## 3.646214e-06 5.194162e+00 2.055887e+00
```

```
exp(confint(modelo))
```

```
##                    2.5 %       97.5 %
## (Intercept) 4.465945e-08 1.593277e-04
## velocida    2.766629e+00 1.068554e+01
## calidadp    1.557441e+00 2.789897e+00
```

Para obtener un resumen más completo del ajuste también se utiliza `summary()`:

```
summary(modelo)
```

```
##
## Call:
## glm(formula = alianza ~ velocida + calidadp, family = binomial,
##     data = train)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -12.5218     2.0758  -6.032 1.62e-09 ***
## velocida      1.6475     0.3426   4.809 1.52e-06 ***
## calidadp      0.7207     0.1479   4.872 1.11e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 218.19  on 159  degrees of freedom
## Residual deviance: 160.55  on 157  degrees of freedom
## AIC: 166.55
##
## Number of Fisher Scoring iterations: 5
```

La desvianza (deviance) es una medida de la bondad del ajuste de un modelo lineal generalizado (sería equivalente a la suma de cuadrados residual de un modelo lineal; valores más altos indican peor ajuste). La *Null deviance* se correspondería con un modelo solo con la constante y la *Residual deviance* con el modelo ajustado. En este caso hay una reducción de 57.65 con una pérdida de 2 grados de libertad (una reducción significativa).

Para contrastar globalmente el efecto de las covariables también podemos emplear:

```
modelo.null <- glm(alianza ~ 1, binomial, train)
anova(modelo.null, modelo, test = "Chi")
```

```
## Analysis of Deviance Table
##
## Model 1: alianza ~ 1
## Model 2: alianza ~ velocida + calidadp
##   Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
## 1       159     218.19
## 2       157     160.55  2   57.646 3.036e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

6.9.1 Selección de variables explicativas

El objetivo sería conseguir un buen ajuste con el menor número de variables explicativas posible. Al igual que en el caso del modelo de regresión lineal múltiple, se podría seguir un proceso interactivo, eliminando o añadiendo variables con la función `update()`, aunque también están disponibles métodos automáticos de selección de variables.

Para obtener el modelo "óptimo" lo ideal sería evaluar todos los modelos posibles. En este caso no se puede emplear la función `regsubsets` del paquete `leaps` (sólo para modelos lineales), pero por ejemplo el paquete `bestglm` proporciona una herramienta equivalente (`bestglm()`). También se podría emplear la función `stepwise()` del paquete `RcmdrMisc` para seleccionar un modelo por pasos según criterio AIC o BIC:

```
# library(RcmdrMisc)
modelo.completo <- glm(alianza ~ ., family = binomial, data = train)
modelo <- stepwise(modelo.completo, direction='forward/backward', criterion='BIC')
```

```
##
## Direction:  forward/backward
## Criterion:  BIC
##
## Start:  AIC=223.27
## alianza ~ 1
##
##             Df Deviance    AIC
## + velocida   1   189.38 199.53
## + calidadp   1   192.15 202.30
## + facturac   1   193.45 203.60
## + producto   1   196.91 207.06
## + quejas     1   198.10 208.25
## + imgfvent   1   198.80 208.95
## + web        1   204.40 214.55
## + publi      1   209.28 219.43
## + precio     1   211.97 222.12
## + garantia   1   212.37 222.52
## <none>           218.19 223.27
## + nprod      1   213.97 224.12
## + soporte    1   216.50 226.65
## + flexprec   1   216.99 227.14
##
## Step:  AIC=199.53
## alianza ~ velocida
##
##             Df Deviance    AIC
## + calidadp   1   160.55 175.77
## + imgfvent   1   178.43 193.65
## + web        1   181.52 196.74
## + precio     1   183.38 198.61
## <none>           189.38 199.53
## + flexprec   1   185.47 200.69
## + producto   1   185.54 200.77
## + nprod      1   186.92 202.14
## + facturac   1   186.93 202.15
## + garantia   1   187.02 202.25
## + publi      1   187.44 202.67
## + soporte    1   189.06 204.29
## + quejas     1   189.27 204.50
## - velocida   1   218.19 223.27
```

```
##
## Step:  AIC=175.77
## alianza ~ velocida + calidadp
##
##            Df Deviance    AIC
## + imgfvent  1   137.05 157.35
## + web       1   145.63 165.93
## <none>          160.55 175.77
## + publi     1   156.03 176.33
## + facturac  1   157.35 177.66
## + flexprec  1   157.44 177.74
## + producto  1   157.75 178.06
## + garantia  1   158.97 179.27
## + nprod     1   160.18 180.47
## + quejas    1   160.20 180.50
## + soporte   1   160.37 180.67
## + precio    1   160.37 180.67
## - calidadp  1   189.38 199.53
## - velocida  1   192.15 202.30
##
## Step:  AIC=157.35
## alianza ~ velocida + calidadp + imgfvent
##
##            Df Deviance    AIC
## <none>          137.05 157.35
## + precio    1   134.97 160.35
## + flexprec  1   135.39 160.77
## + publi     1   135.65 161.03
## + producto  1   135.72 161.09
## + facturac  1   135.81 161.19
## + garantia  1   136.31 161.69
## + nprod     1   136.63 162.00
## + soporte   1   136.79 162.16
## + quejas    1   136.96 162.33
## + web       1   137.03 162.40
## - velocida  1   160.34 175.57
## - imgfvent  1   160.55 175.77
## - calidadp  1   178.43 193.65
```

```
summary(modelo)
```

```
##
## Call:
## glm(formula = alianza ~ velocida + calidadp + imgfvent, family = binomial,
##     data = train)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -20.5164     3.4593  -5.931 3.02e-09 ***
## velocida      1.6631     0.3981   4.177 2.95e-05 ***
## calidadp      1.0469     0.2014   5.197 2.02e-07 ***
## imgfvent      1.0085     0.2398   4.205 2.61e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

```
##     Null deviance: 218.19  on 159  degrees of freedom
## Residual deviance: 137.05  on 156  degrees of freedom
## AIC: 145.05
##
## Number of Fisher Scoring iterations: 5
```

6.9.2 Análisis e interpretación del modelo

Las hipótesis estructurales del modelo son similares al caso de regresión lineal (aunque algunas como la linealidad se suponen en la escala transformada). Si no se verifican, los resultados basados en la teoría estadística pueden no ser fiables o totalmente erróneos.

Con el método `plot()` se pueden generar gráficos de interés para la diagnosis del modelo (ver Figura 6.22):

```
oldpar <- par( mfrow=c(2,2))
plot(modelo)
```

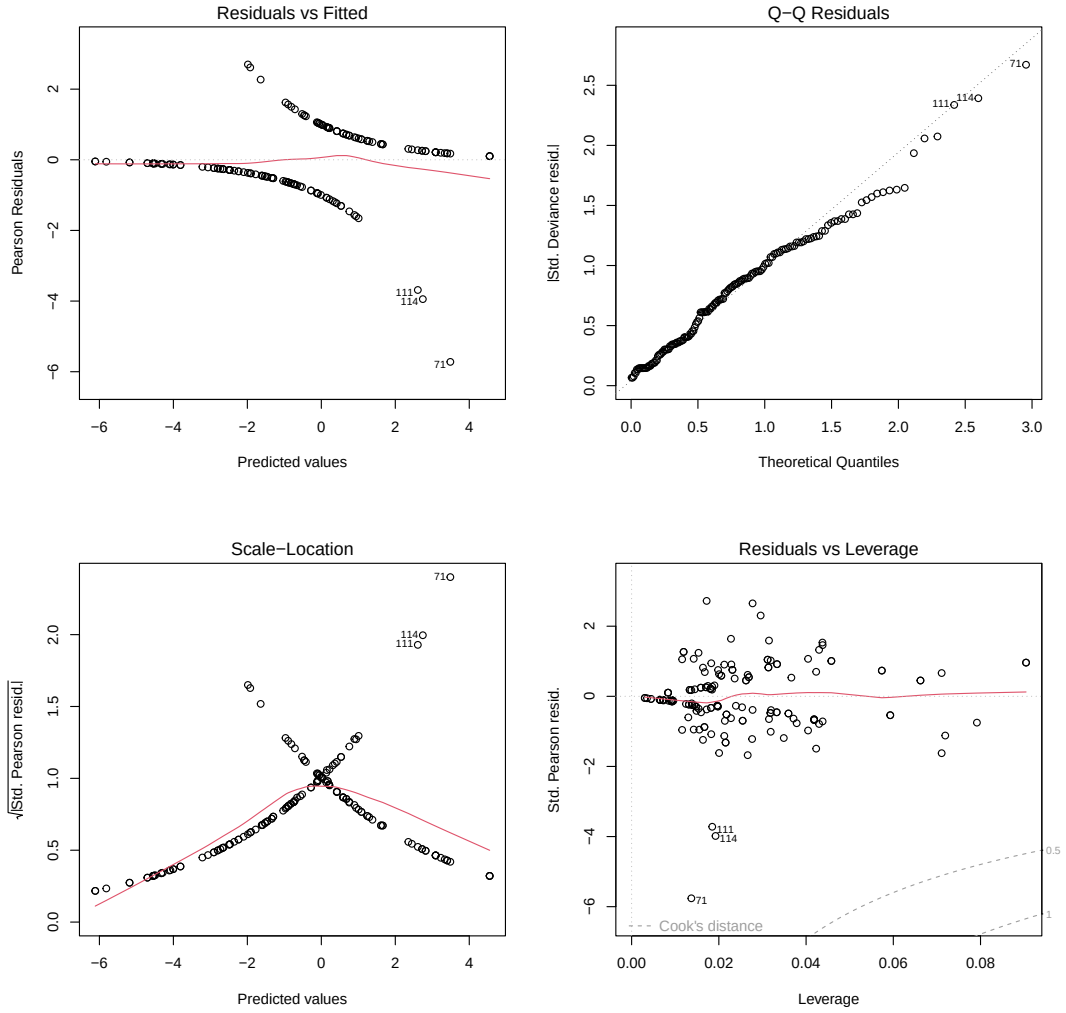


Figura 6.22: Gráficos de diagnóstico del ajuste lineal generalizado.

```
par(oldpar)
```

Su interpretación es similar a la de los modelos lineales (consultar las referencias incluidas al principio de la sección para más detalles). En este caso destacan tres posibles datos atípicos, aunque aparentemente no son muy influyentes a posteriori (en el modelo ajustado). Adicionalmente se pueden generar gráficos parciales de residuos, por ejemplo con la función `crPlots()` del paquete `car` (ver Figura 6.23):

```
# library(car)
id <- list(method = which(abs(residuals(modelo, type = "pearson")) > 3), col = 2)
crPlots(modelo, id = id, main = "")
```

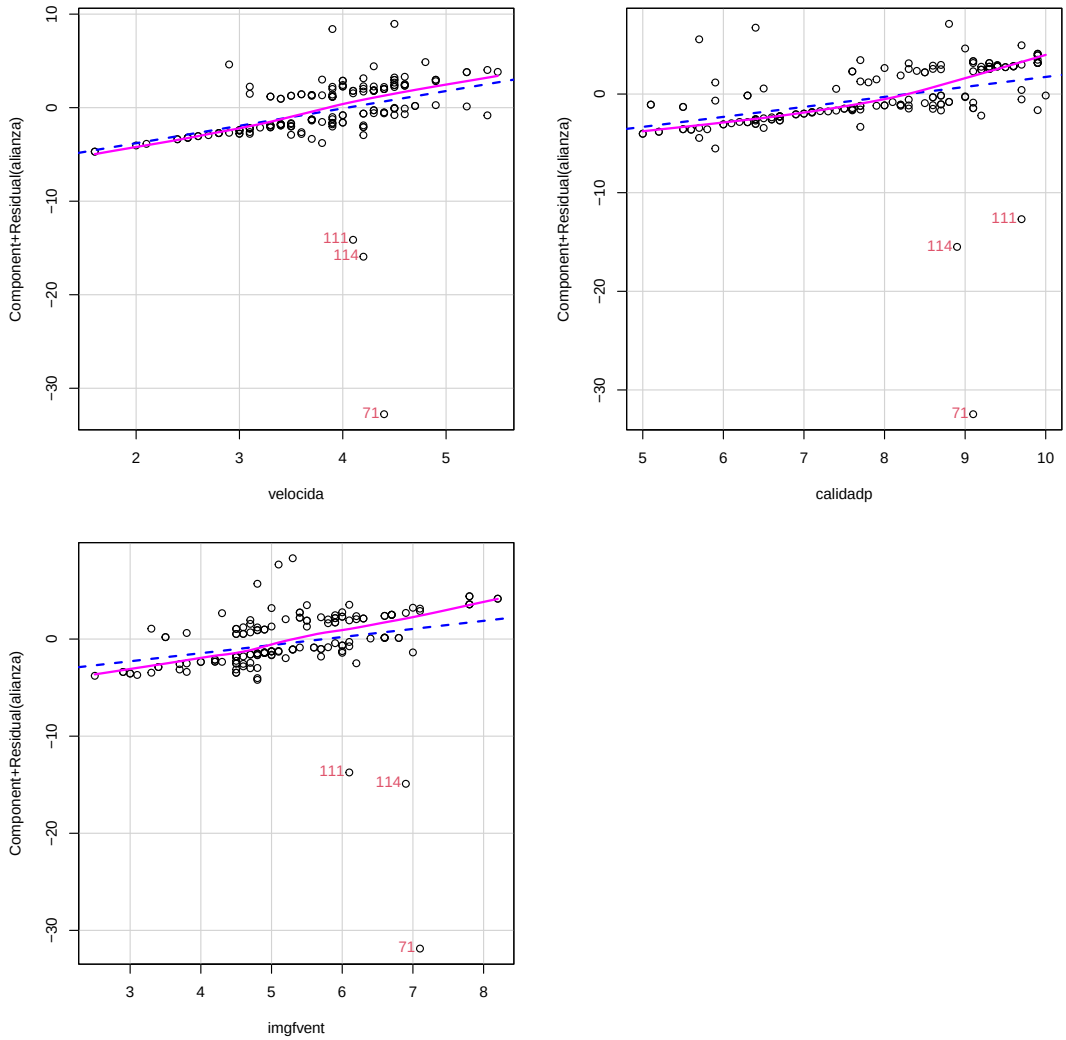


Figura 6.23: Gráficos parciales de residuos del ajuste generalizado.

Se pueden emplear las mismas funciones vistas en los modelos lineales para obtener medidas de diagnosis de interés (tablas 6.1 y 6.2). Por ejemplo `residuals(model, type = "deviance")` proporcionará los residuos *deviance*. Por supuesto también pueden aparecer problemas de colinealidad, y podemos emplear las mismas herramientas para detectarla:

```
# library(car)
vif(modelo)
```

```
## velocida calidadp imgfvent
## 1.193557 1.656649 1.451237
```

Si no se satisfacen los supuestos básicos también se pueden intentar distintas alternativas (se puede cambiar la función de enlace y la familia de distribuciones, que puede incluir parámetros para modelar dispersión, además de las descritas en la Sección 6.4), incluyendo emplear modelos más flexibles o técnicas de aprendizaje estadístico que no dependan de ellas (sustancialmente).

6.9.3 Evaluación de la precisión

Para evaluar la calidad de la predicción en nuevas observaciones podemos seguir los pasos mostrados en la Sección 1.3.5. Obteniendo las estimaciones de la probabilidad (de la segunda categoría) empleando `predict()` con `type = "response"`:

```
p.est <- predict(modelo, type = "response", newdata = test)
pred <- factor(p.est > 0.5, labels = c("No", "Si")) # levels = c('FALSE', 'TRUE')
```

y las medidas de precisión de la predicción (además de los criterios AIC o BIC tradicionales):

```
caret::confusionMatrix(pred, test$alianza, positive = "Si", mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction No Si
##         No 19  5
##         Si  3 13
##
##                Accuracy : 0.8
##                  95% CI : (0.6435, 0.9095)
##     No Information Rate : 0.55
##     P-Value [Acc > NIR] : 0.0008833
##
##                   Kappa : 0.5918
##
##  Mcnemar's Test P-Value : 0.7236736
##
##             Sensitivity : 0.7222
##             Specificity : 0.8636
##          Pos Pred Value : 0.8125
##          Neg Pred Value : 0.7917
##               Precision : 0.8125
##                  Recall : 0.7222
##                      F1 : 0.7647
##              Prevalence : 0.4500
##          Detection Rate : 0.3250
##    Detection Prevalence : 0.4000
##       Balanced Accuracy : 0.7929
##
##        'Positive' Class : Si
##
```

o de las estimaciones de las probabilidades (como el AUC).

6.9.4 Extensiones

Se pueden imponer restricciones a las estimaciones de los parámetros de modo análogo al caso de modelos lineales (secciones 6.7 y 6.8). Por ejemplo, en los métodos de regularización (*ridge*, LASSO o *elastic net*; Sección 6.7) bastaría con cambiar en la función de pérdidas la suma residual de cuadrados por el logaritmo negativo de la función de verosimilitud.

Ejercicio 6.1

Emplear el paquete `glmnet` para ajustar modelos logísticos con penalización *ridge* y LASSO a la muestra de entrenamiento de los datos de clientes de la compañía de distribución industrial HBAT, considerando como respuesta la variable *alianza* y seleccionando un valor "óptimo" del hiperparámetro λ. Ajustar también un modelo con penalización *elastic net* empleando `caret` (seleccionando los valores óptimos de los hiperparámetros).

El método PCR (Sección 6.8) se extendería de forma inmediata al caso de modelos generalizados, simplemente cambiando el modelo ajustado. También están disponibles métodos PLSR para modelos generalizados.

Ejercicio 6.2

Emplear el paquete `caret` para ajustar modelos logísticos con reducción de la dimensión a los datos de clientes de la compañía de distribución industrial HBAT. Comparar el modelo obtenido con preprocesado `"pca"` y el método `"glmStepAIC"`, con el obtenido empleando el método `"plsRglm"`.

Capítulo 7

Regresión no paramétrica

Se trata de métodos que no suponen ninguna forma concreta de la media condicional (i.e. no se hacen suposiciones paramétricas sobre el efecto de las variables explicativas):

$$Y = m\left(X_1, \ldots, X_p\right) + \varepsilon$$

siendo m una función "cualquiera" (se asume que es una función "suave" de los predictores).

La idea detrás de la mayoría de estos métodos es ajustar localmente un modelo de regresión (este capítulo se podría haber titulado "modelos locales"). Suponiendo que disponemos de "suficiente" información en un entorno de la posición de predicción (el número de observaciones debe ser relativamente grande), podríamos pensar en predecir la respuesta a partir de lo que ocurre en las observaciones cercanas.

Nos centraremos principalmente en el caso de regresión, pero la mayoría de estos métodos se pueden extender para el caso de clasificación (por ejemplo considerando una función de enlace y realizando el ajuste localmente por máxima verosimilitud).

Los métodos de regresión basados en: árboles de decisión, bosques aleatorios, bagging, boosting y máquinas de soporte vectorial, vistos en capítulos anteriores, entrarían también dentro de esta clasificación.

7.1 Regresión local

En este tipo de métodos se incluirían: vecinos más próximos, regresión tipo núcleo y loess (o lowess). También se podrían incluir los *splines de regresión* (*regression splines*), pero se tratarán en la siguiente sección, ya que también se pueden ver como una extensión de un modelo lineal global.

Con muchos de estos procedimientos no se obtiene una expresión cerrada del modelo ajustado y (en principio) es necesario disponer de la muestra de entrenamiento para calcular predicciones, por lo que en AE también se denominan *métodos basados en memoria*.

7.1.1 Vecinos más próximos

Uno de los métodos más conocidos de regresión local es el denominado *k-vecinos más cercanos* (*k-nearest neighbors*; KNN), que ya se empleó como ejemplo en la Sección 1.4 (la maldición de la dimensionalidad). Se trata de un método muy simple, pero que en la práctica puede ser efectivo en muchas ocasiones. 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 KNN promedia las observaciones en un vecindario $\mathcal{N}_k(\mathbf{x}, \mathcal{T})$ formado por las k observaciones más cercanas a $\mathbf{x}$:

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

Se puede emplear la misma idea en el caso de clasificación, las frecuencias relativas en el vecindario serían las estimaciones de las probabilidades de las clases (lo que sería equivalente a considerar las variables indicadoras de las categorías) y normalmente la predicción sería la moda (la clase más probable).

Para seleccionar el vecindario es necesario especificar una distancia, por ejemplo:

$$d(\mathbf{x}_0, \mathbf{x}_i) = \left(\sum_{j=1}^{p} \left| x_{j0} - x_{ji} \right|^d \right)^{\frac{1}{d}}$$

Normalmente se considera la distancia euclídea ($d = 2$) o la de Manhatan ($d = 1$) si los predictores son muméricos (también habría distancias diseñadas para predictores categóricos). En cualquier caso la recomendación es estandarizar previamente los predictores para que no influya su escala en el cálculo de las distancias.

Como ya se mostró en al final del Capítulo 1, este método está implementado en la función `knnreg()` (Sección 1.4) y en el método `"knn"` del paquete `caret` (Sección 1.6). Como ejemplo adicional emplearemos el conjunto de datos `MASS::mcycle` que contiene mediciones de la aceleración de la cabeza en una simulación de un accidente de motocicleta, utilizado para probar cascos protectores (considerando el conjunto de datos completo como si fuese la muestra de entrenamiento; ver Figura 7.1):

```
data(mcycle, package = "MASS")
library(caret)
# Ajuste de los modelos
fit1 <- knnreg(accel ~ times, data = mcycle, k = 5) # 5% de los datos
fit2 <- knnreg(accel ~ times, data = mcycle, k = 10)
fit3 <- knnreg(accel ~ times, data = mcycle, k = 20)
# Representación
plot(accel ~ times, data = mcycle, col = 'darkgray')
newx <- seq(1 , 60, len = 200)
newdata <- data.frame(times = newx)
lines(newx, predict(fit1, newdata), lty = 3)
lines(newx, predict(fit2, newdata), lty = 2)
lines(newx, predict(fit3, newdata))
legend("topright", legend = c("5-NN", "10-NN", "20-NN"),
       lty = c(3, 2, 1), lwd = 1)
```

El hiperparámetro k (número de vecinos más cercanos) determina la complejidad del modelo, de forma que valores más pequeños de k se corresponden con modelos más complejos (en el caso extremo $k = 1$ se interpolarían las observaciones). Este parámetro se puede seleccionar empleando alguno de los métodos descritos en la Sección 1.3.3 (por ejemplo mediante validación con k grupos como se mostró en la Sección 1.6).

7.1.2 Regresión polinómica local

En el caso univariante, para cada x_0 se ajusta un polinomio de grado d:

$$\beta_0 + \beta_1 (x - x_0) + \cdots + \beta_d (x - x_0)^d$$

por mínimos cuadrados ponderados, con pesos

$$w_i = K_h(x - x_0) = \frac{1}{h} K \left(\frac{x - x_0}{h} \right)$$

donde K es una función núcleo (normalmente una densidad simétrica en torno al cero) y $h > 0$ es un parámetro de suavizado, llamado ventana, que regula el tamaño del entorno que se usa para llevar a cabo el ajuste (esta ventana también se puede suponer local, $h \equiv h(x_0)$; por ejemplo el método KNN se puede considerar un caso particular, con $d = 0$ y K la densidad de una $\mathcal{U}(-1, 1)$). A partir de este ajuste[1]:

[1] Se puede pensar que se están estimando los coeficientes de un desarrollo de Taylor de $m(x_0)$.

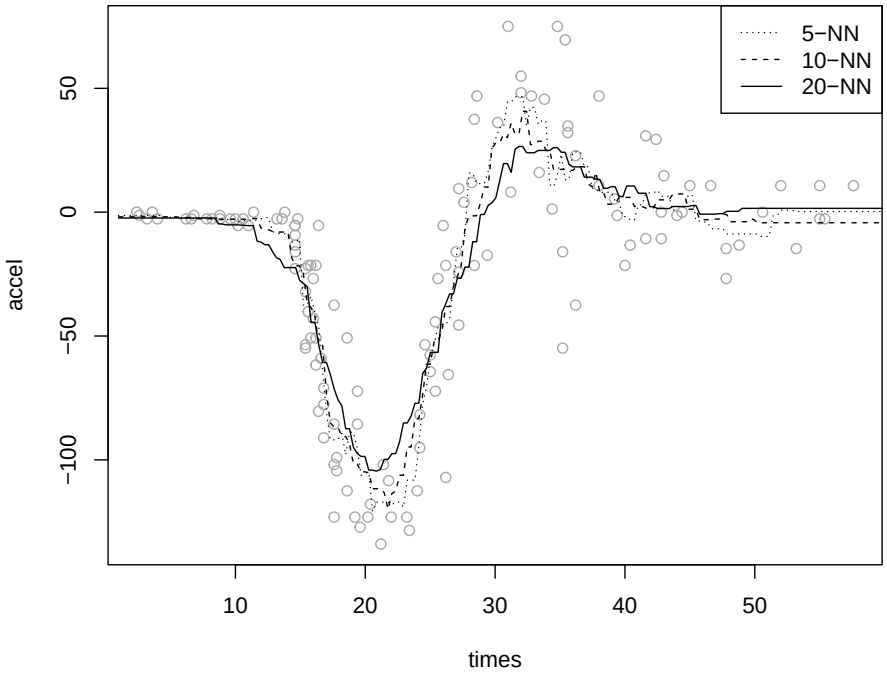


Figura 7.1: Predicciones con el método KNN y distintos vecindarios.

- La estimación en x_0 es $\hat{m}_h(x_0) = \hat{\beta}_0$.
- Podemos obtener también estimaciones de las derivadas: $\widehat{m_h^{(r)}}(x_0) = r!\hat{\beta}_r$.

Por tanto, la estimación polinómica local de grado d, $\hat{m}_h(x) = \hat{\beta}_0$, se obtiene al minimizar:

$$\min_{\beta_0,\beta_1,\ldots,\beta_d} \sum_{i=1}^{n} \left\{Y_i - \beta_0 - \beta_1(x - X_i) - \ldots - \beta_d(x - X_i)^d\right\}^2 K_h(x - X_i)$$

Explícitamente:

$$\hat{m}_h(x) = \mathbf{e}_1^t \left(X_x^t W_x X_x\right)^{-1} X_x^t W_x \mathbf{Y} \equiv s_x^t \mathbf{Y}$$

donde $\mathbf{e}_1 = (1, \cdots, 0)^t$, X_x es la matriz con $(1, x - X_i, \ldots, (x - X_i)^d)$ en la fila i, $W_x = \texttt{diag}\left(K_h(x_1 - x), \ldots, K_h(x_n - x)\right)$ es la matriz de pesos, e $\mathbf{Y} = (Y_1, \cdots, Y_n)^t$ es el vector de observaciones de la respuesta.

Se puede pensar que se obtiene aplicando un suavizado polinómico a (X_i, Y_i):

$$\hat{\mathbf{Y}} = S\mathbf{Y}$$

siendo S la matriz de suavizado con $\mathbf{s}_{X_i}^t$ en la fila i (este tipo de métodos también se denominan *suavizadores lineales*).

Habitualmente se considera $d = 0$, el estimador Nadaraya-Watson, o $d = 1$, estimador lineal local. Desde el punto de vista asintótico ambos estimadores tienen un comportamiento similar[2], pero en la práctica suele ser preferible el estimador lineal local, sobre todo porque se ve menos afectado por el denominado efecto frontera (Sección 1.4).

La ventana h es el (hiper)parámetro de mayor importancia en la predicción y para seleccionarlo se suelen emplear métodos de validación cruzada (Sección 1.3.3) o tipo plug-in (reemplazando las funciones desconocidas que aparecen en la expresión de la ventana asintóticamente óptima por estimaciones;

[2]Asintóticamente el estimador lineal local tiene un sesgo menor que el de Nadaraya-Watson (pero del mismo orden) y la misma varianza (e.g. Fan y Gijbels (1996)).

e.g. función `dpill()` del paquete `KernSmooth`). Por ejemplo, usando el criterio de validación cruzada dejando uno fuera (LOOCV) se trataría de minimizar:

$$CV(h) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{m}_{-i}(x_i))^2$$

siendo $\hat{m}_{-i}(x_i)$ la predicción obtenida eliminando la observación i-ésima. Al igual que en el caso de regresión lineal, este error también se puede obtener a partir del ajuste con todos los datos:

$$CV(h) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{m}(x_i)}{1 - S_{ii}}\right)^2$$

siendo S_{ii} el elemento i-ésimo de la diagonal de la matriz de suavizado (esto en general es cierto para cualquier suavizador lineal).

Alternativamente se podría emplear *validación cruzada generalizada* (Craven y Wahba, 1978):

$$GCV(h) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{m}(x_i)}{1 - \frac{1}{n}tr(S)}\right)^2$$

(sustituyendo S_{ii} por su promedio). Además, la traza de la matriz de suavizado $tr(S)$ es lo que se conoce como el *número efectivo de parámetros* ($n - tr(S)$ sería una aproximación de los grados de libertad del error).

Aunque el paquete base de `R` incluye herramientas para la estimación tipo núcleo de la regresión (`ksmooth()`, `loess()`), recomiendan el uso del paquete `KernSmooth` (**R-KernSmooth?**). Continuando con el ejemplo del conjunto de datos `MASS::mcycle` emplearemos la función `locpoly()` de este paquete para obtener estimaciones lineales locales[3] con una venta seleccionada mediante un método plug-in (ver Figura 7.2):

```
# data(mcycle, package = "MASS")
times <- mcycle$times
accel <- mcycle$accel
library(KernSmooth)
h <- dpill(times, accel) # Método plug-in de Ruppert, Sheather y Wand (1995)
fit <- locpoly(times, accel, bandwidth = h) # Estimación lineal local
plot(times, accel, col = 'darkgray')
lines(fit)
```

Hay que tener en cuenta que el paquete `KernSmooth` no implementa los métodos `predict()` y `residuals()`:

```
pred <- approx(fit, xout = times)$y # pred <- predict(fit)
resid <- accel - pred # resid <- residuals(fit)
```

Tampoco calcula medidas de bondad de ajuste, aunque podríamos calcular medidas de la precisión de las predicciones de la forma habitual (en este caso de la muestra de entrenamiento):

```
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(
```

[3] La función `KernSmooth::locpoly()` también admite la estimación de derivadas.

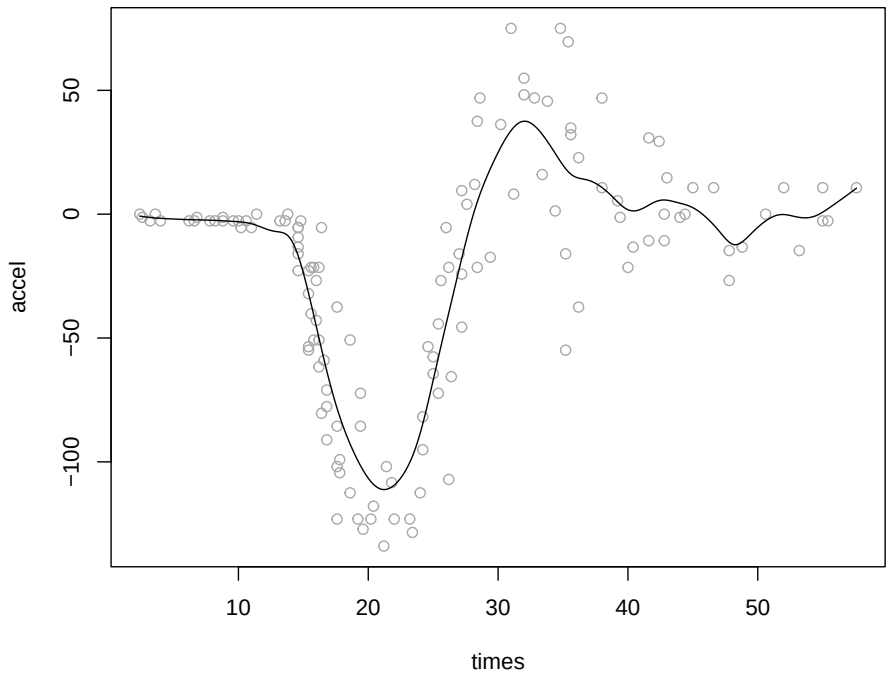


Figura 7.2: Ajuste lineal local con ventana plug-in.

```
    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, accel)
```

```
##            me          rmse           mae           mpe          mape
## -2.712378e-01  2.140005e+01  1.565921e+01 -2.460832e+10  7.559223e+10
##     r.squared
##  8.023864e-01
```

El caso multivariante es análogo, aunque habría que considerar una matriz de ventanas simétrica H. También hay extensiones para el caso de predictores categóricos (nominales o ordinales) y para el caso de distribuciones de la respuesta distintas de la normal (máxima verosimilitud local).

Otros paquetes de R incluyen más funcionalidades (`sm`, `locfit`, `npsp`…), pero hoy en día el paquete `np` es el que se podría considerar más completo.

7.1.3 Regresión polinómica local robusta

También hay versiones robustas del ajuste polinómico local tipo núcleo. Estos métodos surgieron en el caso bivariante ($p = 1$), por lo que también se denominan *suavizado de diagramas de dispersión* (*scatterplot smoothing*; e.g. función `lowess()`, *locally weighted scatterplot smoothing*, del paquete base). Posteriormente se extendieron al caso multivariante (e.g. función `loess()`). Son métodos muy empleados en análisis descriptivo (no supervisado) y normalmente se emplean ventanas locales tipo vecinos más cercanos (por ejemplo a través de un parámetro `spam` que determina la proporción de observaciones empleadas en el ajuste).

Como ejemplo emplearemos la función `loess()` con ajuste robusto (habrá que establecer `family = "symmetric"` para emplear M-estimadores, por defecto con 4 iteraciones, en lugar de mínimos cuadra-

dos ponderados), seleccionando previamente `spam` por validación cruzada (LOOCV) pero empleando como criterio de error la mediana de los errores en valor absoluto (*median absolute deviation*, MAD)[4] (ver Figura 7.3).

```
cv.loess <- function(formula, datos, span, ...) {
  n <- nrow(datos)
  cv.pred <- numeric(n)
  for (i in 1:n) {
    modelo <- loess(formula, datos[-i, ], span = span,
                    control = loess.control(surface = "direct"), ...)
    # control = loess.control(surface = "direct") permite extrapolaciones
    cv.pred[i] <- predict(modelo, newdata = datos[i, ])
  }
  return(cv.pred)
}

ventanas <- seq(0.1, 0.5, len = 10)
np <- length(ventanas)
cv.error <- numeric(np)
for(p in 1:np){
  cv.pred <- cv.loess(accel ~ times, mcycle, ventanas[p], family = "symmetric")
  # cv.error[p] <- mean((cv.pred - mcycle$accel)^2)
  cv.error[p] <- median(abs(cv.pred - mcycle$accel))
}

plot(ventanas, cv.error)
imin <- which.min(cv.error)
span.cv <- ventanas[imin]
points(span.cv, cv.error[imin], pch = 16)
```

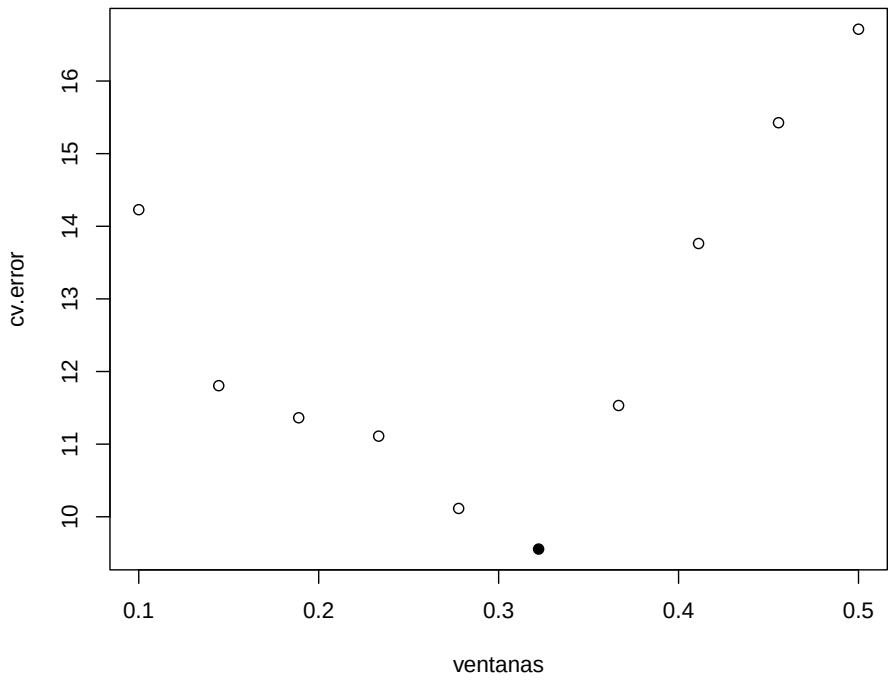


Figura 7.3: Error de predicción de validación cruzada (mediana de los errores absolutos) del ajuste LOWESS dependiendo del parámetro de suavizado.

[4]En este caso habría dependencia entre las observaciones y los criterios habituales como validación cruzada tenderán a seleccionar ventanas pequeñas, i.e. a infrasuavizar.

Empleamos el parámetro de suavizado seleccionado para ajustar el modelo final (ver Figura 7.4):

```
# Ajuste con todos los datos
plot(accel ~ times, data = mcycle, col = 'darkgray')
fit <- loess(accel ~ times, mcycle, span = span.cv, family = "symmetric")
lines(mcycle$times, predict(fit))
```

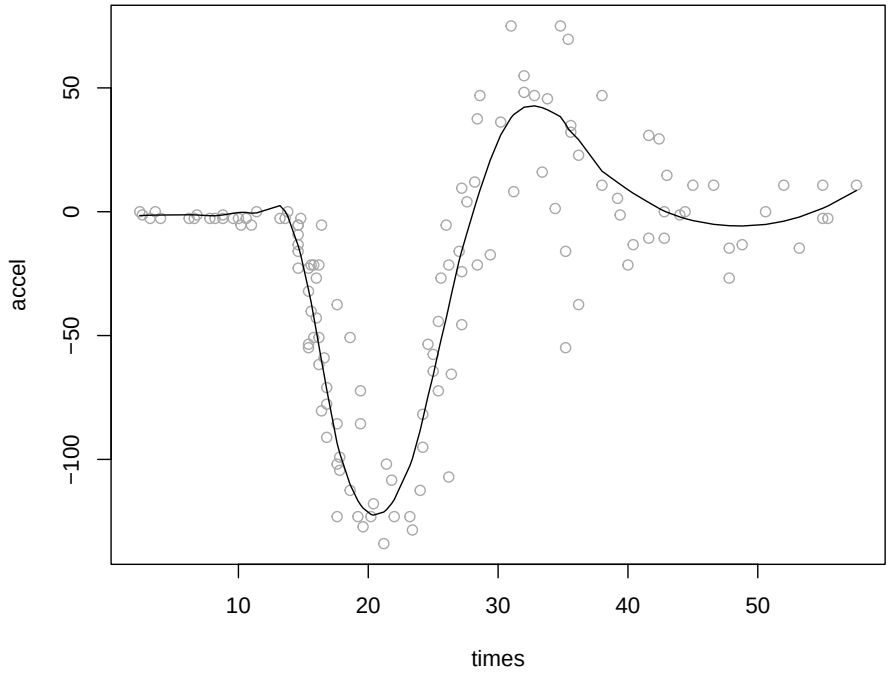


Figura 7.4: Ajuste polinómico local robusto (LOWESS), con el parámetro de suavizado seleccionado mediante validación cruzada.

7.2 Splines

Otra alternativa consiste en trocear los datos en intervalos, fijando unos puntos de corte z_i (denominados nudos; *knots*), con $i = 1, \dots, k$, y ajustar un polinomio en cada segmento (lo que se conoce como regresión segmentada, *piecewise regression*; ver Figura 7.5). De esta forma sin embargo habrá discontinuidades en los puntos de corte, pero podrían añadirse restricciones adicionales de continuidad (o incluso de diferenciabilidad) para evitarlo (e.g. paquete `segmented`).

7.2.1 Splines de regresión

Cuando en cada intervalo se ajustan polinomios de orden d y se incluyen restricciones de forma que las derivadas sean continuas hasta el orden $d-1$ se obtienen los denominados splines de regresión (*regression splines*). Puede verse que este tipo de ajustes equivalen a transformar la variable predictora X, considerando por ejemplo la *base de potencias truncadas* (*truncated power basis*):

$$1, x, \dots, x^d, (x - z_1)_+^d, \dots, (x - z_k)_+^d$$

siendo $(x - z)_+ = \max(0, x - z)$, y posteriormente realizar un ajuste lineal:

$$m(x) = \beta_0 + \beta_1 b_1(x) + \beta_2 b_2(x) + \dots + \beta_{k+d} b_{k+d}(x)$$

Típicamente se seleccionan polinomios de grado $d = 3$, lo que se conoce como splines cúbicos, y nodos equiespaciados. Además, se podrían emplear otras bases equivalentes. Por ejemplo, para evitar posibles problemas computacionales con la base anterior, se suele emplear la denominada base B-spline (De Boor y De Boor, 1978), implementada en la función `bs()` del paquete `splines` (ver Figura 7.6):

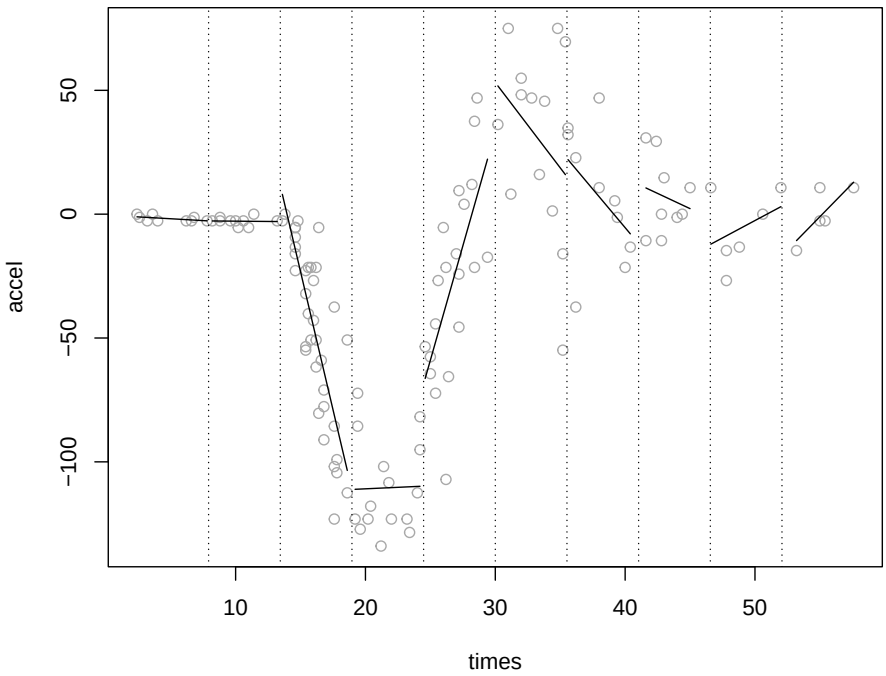


Figura 7.5: Estimación mediante regresión segmentada.

```
nknots <- 9 # nodos internos; 10 intervalos
knots <- seq(min(times), max(times), len = nknots + 2)[-c(1, nknots + 2)]
library(splines)
fit1 <- lm(accel ~ bs(times, knots = knots, degree = 1))
fit2 <- lm(accel ~ bs(times, knots = knots, degree = 2))
fit3 <- lm(accel ~ bs(times, knots = knots)) # degree = 3
# Representar
plot(times, accel, col = 'darkgray')
newx <- seq(min(times), max(times), len = 200)
newdata <- data.frame(times = newx)
lines(newx, predict(fit1, newdata), lty = 3)
lines(newx, predict(fit2, newdata), lty = 2)
lines(newx, predict(fit3, newdata))
abline(v = knots, lty = 3, col = 'darkgray')
legend("topright", legend = c("d=1 (df=11)", "d=2 (df=12)", "d=3 (df=13)"),
       lty = c(3, 2, 1))
```

El grado del polinomio, pero sobre todo el número de nodos, determinarán la flexibilidad del modelo. Se podrían considerar el número de parámetros en el ajuste lineal, los grados de libertad, como medida de la complejidad (en la función `bs()` se puede especificar `df` en lugar de `knots`, y estos se generarán a partir de los cuantiles).

Como ya se comentó, al aumentar el grado de un modelo polinómico se incrementa la variabilidad de las predicciones, especialmente en la frontera. Para tratar de evitar este problema se suelen emplear los *splines naturales*, que son splines de regresión con restricciones adicionales de forma que el ajuste sea lineal en los intervalos extremos (lo que en general produce estimaciones más estables en la frontera y mejores extrapolaciones). Estas restricciones reducen la complejidad (los grados de libertad del modelo), y al igual que en el caso de considerar únicamente las restricciones de continuidad y diferenciabilidad, resultan equivalentes a considerar una nueva base en un ajuste sin restricciones. Por ejemplo, se puede emplear la función `splines::ns()` para ajustar un spline natural (cúbico por defecto; ver Figura 7.7):

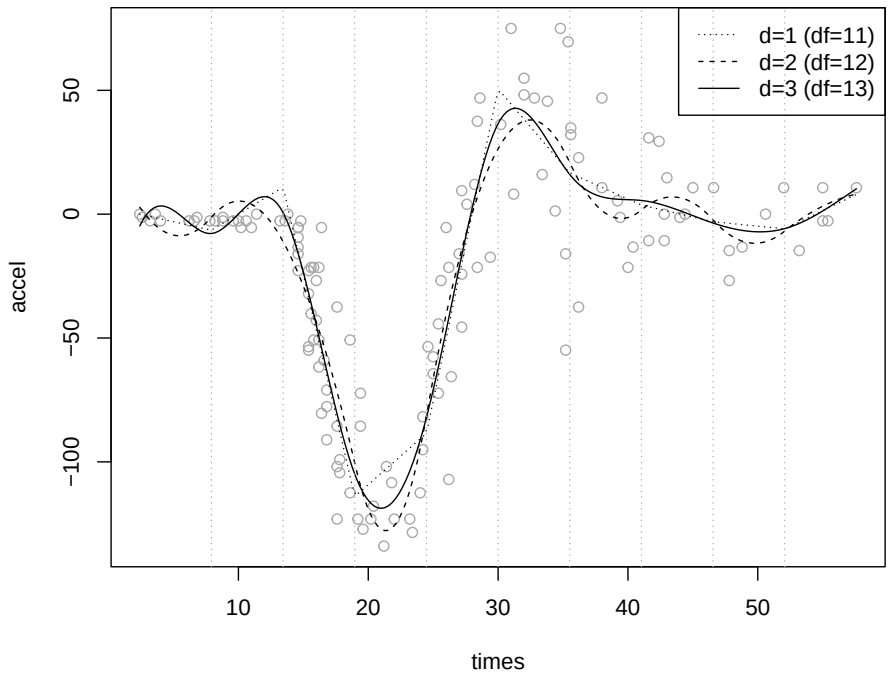


Figura 7.6: Ajustes mediante splines de regresión (de grados 1, 2 y 3).

```
plot(times, accel, col = 'darkgray')
fit4 <- lm(accel ~ ns(times, knots = knots))
lines(newx, predict(fit4, newdata))
lines(newx, predict(fit3, newdata), lty = 2)
abline(v = knots, lty = 3, col = 'darkgray')
legend("topright", legend = c("ns (d=3, df=11)", "bs (d=3, df=13)"), lty = c(1, 2))
```

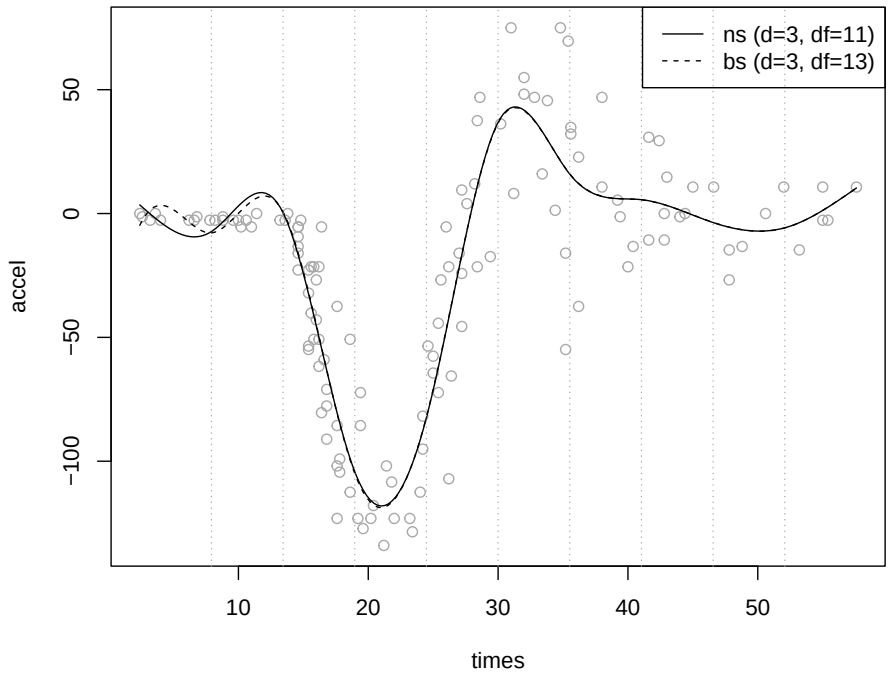


Figura 7.7: Ajuste mediante splines naturales (ns) y B-splines (bs).

La dificultad está en la selección de los nodos z_i. Si se consideran equiespaciados (o se emplea otro

criterio como los cuantiles), se podría seleccionar su número (equivalentemente los grados de libertad) empleando algún método de validación cruzada. Sin embargo, sería preferible considerar más nodos donde aparentemente hay más variaciones en la función de regresión y menos donde es más estable, esta es la idea de la regresión spline adaptativa descrita en la Sección 7.4. Otra alternativa son los splines penalizados, descritos al final de esta sección.

7.2.2 Splines de suavizado

Los splines de suavizado (*smoothing splines*) se obtienen como la función $s(x)$ suave (dos veces diferenciable) que minimiza la suma de cuadrados residual más una penalización que mide su rugosidad:

$$\sum_{i=1}^{n}(y_i - s(x_i))^2 + \lambda \int s''(x)^2 dx$$

siendo $0 \leq \lambda < \infty$ el (hiper)parámetro de suavizado.

Puede verse que la solución a este problema, en el caso univariante, es un spline natural cúbico con nodos en $x_1, \ldots, x_n$ y restricciones en los coeficientes determinadas por el valor de λ (es una versión regularizada de un spline natural cúbico). Por ejemplo si $\lambda = 0$ se interpolarán las observaciones y cuando $\lambda \to \infty$ el ajuste tenderá a una recta (con segunda derivada nula). En el caso multivariante $p > 1$ la solución da lugar a los denominados *thin plate splines*[5].

Al igual que en el caso de la regresión polinómica local (Sección 7.1.2), estos métodos son suavizadores lineales:

$$\hat{\mathbf{Y}} = S_\lambda \mathbf{Y}$$

y para seleccionar el parámetro de suavizado λ podemos emplear los criterios de validación cruzada (dejando uno fuera), minimizando:

$$CV(\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{s}_\lambda(x_i)}{1 - \{S_\lambda\}_{ii}}\right)^2$$

siendo $\{S_\lambda\}_{ii}$ el elemento i-ésimo de la diagonal de la matriz de suavizado, o validación cruzada generalizada (GCV), minimizando:

$$GCV(\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{s}_\lambda(x_i)}{1 - \frac{1}{n}tr(S_\lambda)}\right)^2$$

Análogamente, el número efectivo de parámetros o grados de libertad[6] $df_\lambda = tr(S_\lambda)$ sería una medida de la complejidad del modelo equivalente a λ (muchas implementaciones permiten seleccionar la complejidad empleando df).

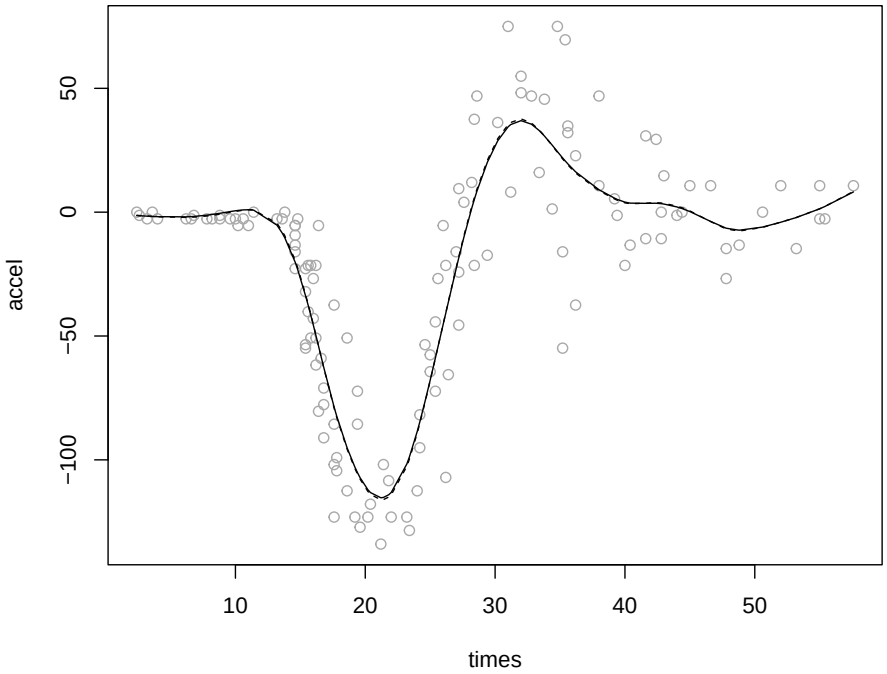
Este método de suavizado está implementado en la función `smooth.spline()` del paquete base y por defecto emplea GCV para seleccionar el parámetro de suavizado (aunque también admite CV y se puede especificar `lambda` o `df`)[7].

```
sspline.gcv <- smooth.spline(times, accel)
sspline.cv <- smooth.spline(times, accel, cv = TRUE)
plot(times, accel, col = 'darkgray')
lines(sspline.gcv)
lines(sspline.cv, lty = 2)
```

[5] Están relacionados con las funciones radiales. También hay versiones con un número reducido de nodos denominados *low-rank thin plate regression splines* empleados en el paquete `mgcv`.

[6] Esto también permitiría generalizar los criterios AIC o BIC.

[7] Además de predicciones, el correspondiente método `predict()` también permite obtener estimaciones de las derivadas.



Cuando el número de observaciones es muy grande, y por tanto el número de nodos, pueden aparecer problemas computacionales al emplear estos métodos.

7.2.3 Splines penalizados

Los splines penalizados (*penalized splines*) combinan las dos aproximaciones anteriores. Incluyen una penalización (que depende de la base considerada) y el número de nodos puede ser mucho menor que el número de observaciones (son un tipo de *low-rank smoothers*). De esta forma se obtienen modelos spline con mejores propiedades, con un menor efecto frontera y en los que se evitan problemas en la selección de los nodos. Unos de los más empleados son los P-splines (Eilers y Marx, 1996) que emplean una base B-spline con una penalización simple (basada en los cuadrados de diferencias de coeficientes consecutivos $(\beta_{i+1} - \beta_i)^2$).

Además, un modelo spline penalizado se puede representar como un modelo lineal mixto, lo que permite emplear herramientas desarrolladas para este tipo de modelos (por ejemplo la implementadas en el paquete `nlme`, del que depende `mgcv`, que por defecto emplea splines penalizados). Para más detalles ver por ejemplo las secciones 5.2 y 5.3 de Wood (2017).

7.3 Modelos aditivos

Se supone que:

$$Y = \beta_0 + f_1(X_1) + f_2(X_2) + ... + f_p(X_p) + \varepsilon$$

con f_i, $i = 1, ..., p$, funciones cualesquiera. De esta forma se consigue mucha mayor flexibilidad que con los modelos lineales pero manteniendo la interpretabilidad de los efectos de los predictores. Adicionalmente se puede considerar una función de enlace, obteniéndose los denominados *modelos aditivos generalizados* (GAM). Para más detalles sobre este tipo modelos ver por ejemplo T. Hastie y Tibshirani (1990) o Wood (2017).

Los modelos lineales (generalizados) serían un caso particular considerando $f_i(x) = \beta_i x$. Además, se podrían considerar cualquiera de los métodos de suavizado descritos anteriormente para construir las componentes no paramétricas (por ejemplo si se emplean splines naturales de regresión el ajuste se reduciría al de un modelo lineal). Se podrían considerar distintas aproximaciones para el modelado de cada componente (modelos semiparamétricos) y realizar el ajuste mediante *backfitting* (se ajusta cada componente de forma iterativa, empleando los residuos obtenidos al mantener las demás fijas). Si en las componentes no paramétricas se emplea únicamente splines de regresión (con o sin penalización),

se puede reformular el modelo como un GLM (regularizado si hay penalización) y ajustarlo fácilmente adaptando herramientas disponibles (*penalized re-weighted iterative least squares*, PIRLS).

De entre todos los paquetes de R que implementan estos modelos destacan:

- `gam`: Admite splines de suavizado (univariantes, `s()`) y regresión polinómica local (multivariante, `lo()`), pero no dispone de un método para la selección automática de los parámetros de suavizado (se podría emplear un criterio por pasos para la selección de componentes). Sigue la referencia T. Hastie y Tibshirani (1990).
- `mgcv`: Admite una gran variedad de splines de regresión y splines penalizados (`s()`; por defecto emplea thin plate regression splines penalizados multivariantes), con la opción de selección automática de los parámetros de suavizado mediante distintos criterios. Además de que se podría emplear un método por pasos, permite la selección de componentes mediante regularización. Al ser más completo que el anterior sería el recomendado en la mayoría de los casos (ver `?mgcv::mgcv.package` para una introducción al paquete). Sigue la referencia Wood (2017).

La función `gam()` del paquete `mgcv` permite ajustar modelos aditivos generalizados empleando suavizado mediante splines:

```
ajuste <- gam(formula, family = gaussian, data, method = "GCV.Cp", select = FALSE, ...)
```

(también dispone de la función `bam()` para el ajuste de estos modelos a grandes conjuntos de datos y de la función `gamm()` para el ajuste de modelos aditivos generalizados mixtos, incluyendo dependencia en los errores). El modelo se establece a partir de la `formula` empleando `s()` para especificar las componentes "suaves" (ver `help(s)` y Sección 7.3.3).

Algunas posibilidades de uso son las que siguen:

- Modelo lineal:
  ```
  ajuste <- gam(y ~ x1 + x2 + x3)
  ```
- Modelo (semiparamétrico) aditivo con efectos no paramétricos para `x1` y `x2`, y un efecto lineal para `x3`:
  ```
  ajuste <- gam(y ~ s(x1) + s(x2) + x3)
  ```
- Modelo no aditivo (con interacción):
  ```
  ajuste <- gam(y ~ s(x1, x2))
  ```
- Modelo (semiparamétrico) con distintas combinaciones :
  ```
  ajuste <- gam(y ~ s(x1, x2) + s(x3) + x4)
  ```

En esta sección utilizaremos como ejemplo el conjunto de datos `Prestige` de la librería `carData`. Se tratará de explicar `prestige` (puntuación de ocupaciones obtenidas a partir de una encuesta) a partir de `income` (media de ingresos en la ocupación) y `education` (media de los años de educación).

```
library(mgcv)
data(Prestige, package = "carData")
modelo <- gam(prestige ~ s(income) + s(education), data = Prestige)
summary(modelo)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## prestige ~ s(income) + s(education)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)  46.8333     0.6889   67.98   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                edf Ref.df     F p-value
## s(income)    3.118  3.877 14.61  <2e-16 ***
## s(education) 3.177  3.952 38.78  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.836   Deviance explained = 84.7%
## GCV = 52.143  Scale est. = 48.414    n = 102
```

```
# coef(modelo)
# El resultado es un modelo lineal en transformaciones de los predictores
```

En este caso el método `plot()` representa los efectos (parciales) estimados de cada predictor (ver Figura 7.8):

```
plot(modelo, shade = TRUE, pages = 1) # residuals = FALSE por defecto
```

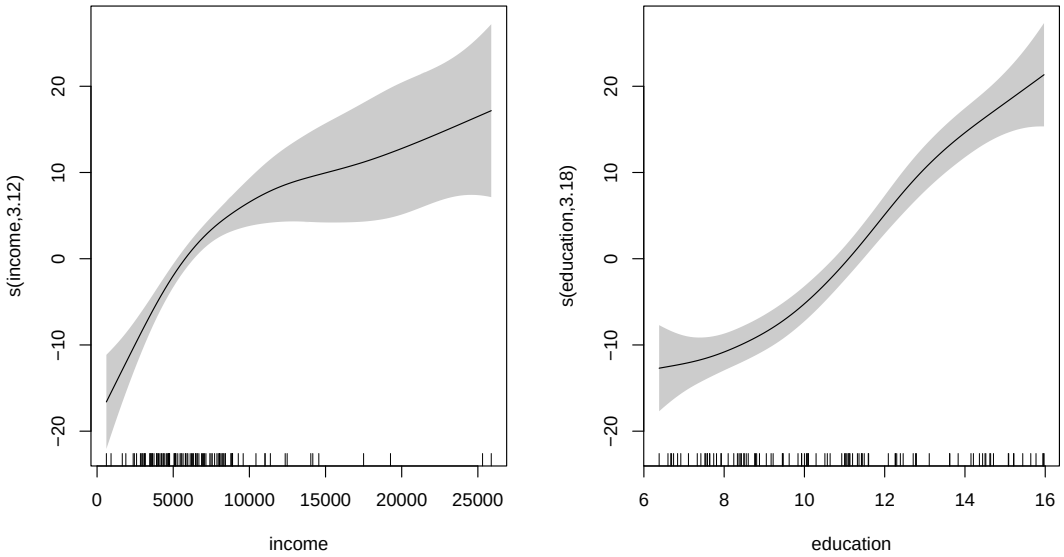


Figura 7.8: Estimaciones de los efectos parciales de `income` (izquierda) y `education` (derecha).

En general se representa cada componente no paramétrica (salvo que se especifique `all.terms = TRUE`), incluyendo gráficos de contorno para el caso de componentes bivariantes (correspondientes a interacciones entre predictores).

Se dispone también de un método `predict()` para calcular las predicciones de la forma habitual (por defecto devuelve las correspondientes a las observaciones `modelo$fitted.values` y para nuevos datos hay que emplear el argumento `newdata`).

7.3.1 Superficies de predicción

En el caso bivariante, para representar las estimaciones (la superficie de predicción) obtenidas con el modelo se pueden utilizar las funciones `persp()` o versiones mejoradas como `plot3D::persp3D()`. Estas funciones requieren que los valores de entrada estén dispuestos en una rejilla bidimensional. Para generar esta rejilla se puede emplear la función `expand.grid(x,y)` que crea todas las combinaciones de los puntos dados en `x` e `y` (ver Figura 7.9):

```
inc <- with(Prestige, seq(min(income), max(income), len = 25))
ed <- with(Prestige, seq(min(education), max(education), len = 25))
newdata <- expand.grid(income = inc, education = ed)
# Representamos la rejilla
plot(income ~ education, Prestige, pch = 16)
abline(h = inc, v = ed, col = "grey")
```

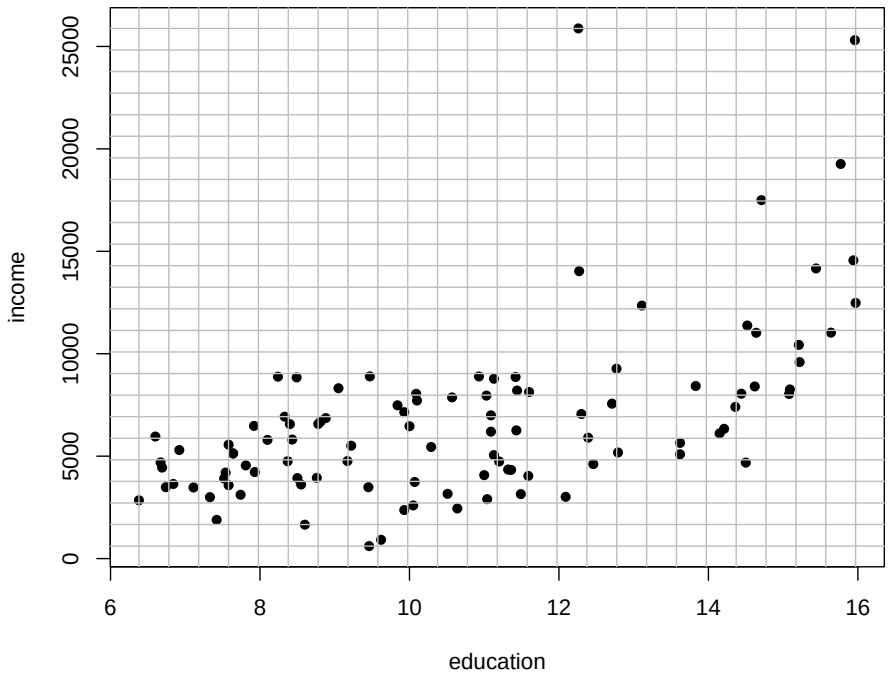


Figura 7.9: Observaciones y rejilla de predicción (para los predictores `education` e `income`).

y usaríamos estos valores para obtener la superficie de predicción, que en este caso[8] representamos con la función `plot3D::persp3D()` (ver Figura 7.10):

```
# Se calculan las predicciones
pred <- predict(modelo, newdata)
# Se representan
pred <- matrix(pred, nrow = 25)
# persp(inc, ed, pred, theta = -40, phi = 30)
# contour(inc, ed, pred, xlab = "Income", ylab = "Education")
# filled.contour(inc, ed, pred, xlab = "Income", ylab = "Education",
#                key.title = title("Prestige"))
plot3D::persp3D(inc, ed, pred, theta = -40, phi = 30, ticktype = "detailed",
                xlab = "Income", ylab = "Education", zlab = "Prestige")
```

Puede ser más cómodo emplear el paquete `modelr` (emplea gráficos `ggplot2`) para trabajar con modelos y predicciones.

7.3.2 Comparación y selección de modelos

Además de las medidas de bondad de ajuste como el coeficiente de determinación ajustado, también se puede emplear la función `anova()` para la comparación de modelos (y seleccionar las componentes por pasos de forma interactiva). Por ejemplo, viendo el gráfico de los efectos se podría pensar que el efecto de `education` podría ser lineal:

[8] Alternativamente se podrían emplear las funciones `contour()`, `filled.contour()`, `plot3D::image2D()` o similares.

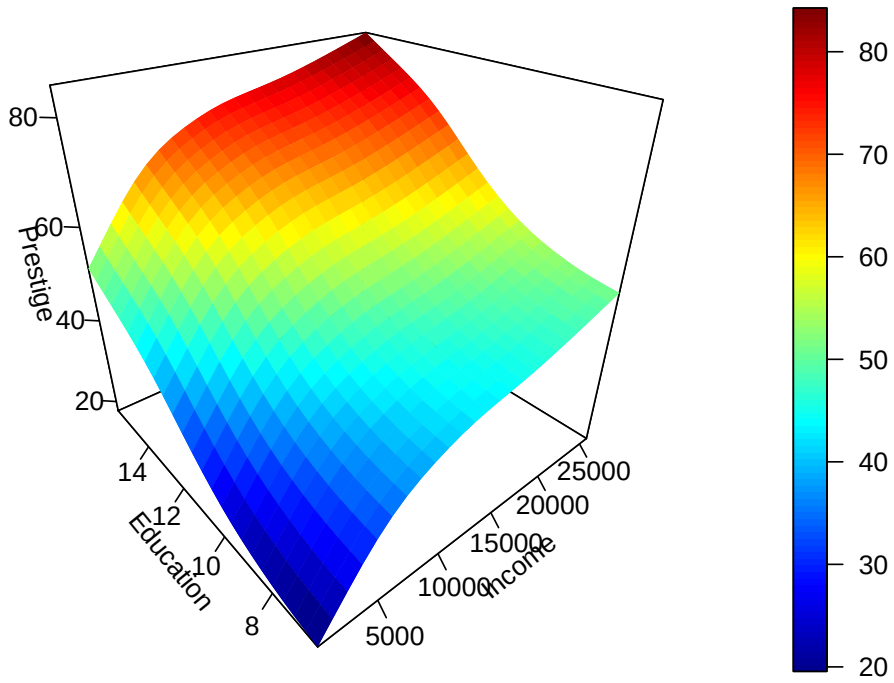


Figura 7.10: Superficie de predicción obtenida con el modelo GAM.

```
# plot(modelo)
modelo0 <- gam(prestige ~ s(income) + education, data = Prestige)
summary(modelo0)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## prestige ~ s(income) + education
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.2240     3.7323   1.132    0.261
## education     3.9681     0.3412  11.630   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##            edf Ref.df    F p-value
## s(income) 3.58  4.441 13.6  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.825   Deviance explained = 83.3%
## GCV = 54.798  Scale est. = 51.8      n = 102
```

```
anova(modelo0, modelo, test="F")
```

```
## Analysis of Deviance Table
##
## Model 1: prestige ~ s(income) + education
```

```
## Model 2: prestige ~ s(income) + s(education)
##   Resid. Df Resid. Dev     Df Deviance      F Pr(>F)
## 1    95.559     4994.6
## 2    93.171     4585.0 2.3886   409.58 3.5418 0.0257 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

En este caso aceptaríamos que el modelo original es significativamente mejor.

Alternativamente, podríamos pensar que hay interacción:

```
modelo2 <- gam(prestige ~ s(income, education), data = Prestige)
summary(modelo2)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## prestige ~ s(income, education)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  46.8333     0.7138   65.61   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                       edf Ref.df     F p-value
## s(income,education) 4.94  6.303 75.41  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.824   Deviance explained = 83.3%
## GCV = 55.188  Scale est. = 51.974    n = 102
```

En este caso el coeficiente de determinación ajustado es menor y ya no tendría sentido realizar el contraste.

Ademas se pueden seleccionar componentes del modelo (mediante regularización) empleando el parámetro `select = TRUE`. Para más detalles consultar la ayuda `help(gam.selection)` o ejecutar `example(gam.selection)`.

7.3.3 Diagnosis del modelo

La función `gam.check()` realiza una diagnosis descriptiva y gráfica del modelo ajustado (ver Figura 7.11):

```
gam.check(modelo)
```

```
##
## Method: GCV   Optimizer: magic
## Smoothing parameter selection converged after 4 iterations.
## The RMS GCV score gradient at convergence was 9.783945e-05 .
## The Hessian was positive definite.
## Model rank =  19 / 19
##
## Basis dimension (k) checking results. Low p-value (k-index<1) may
## indicate that k is too low, especially if edf is close to k'.
##
```

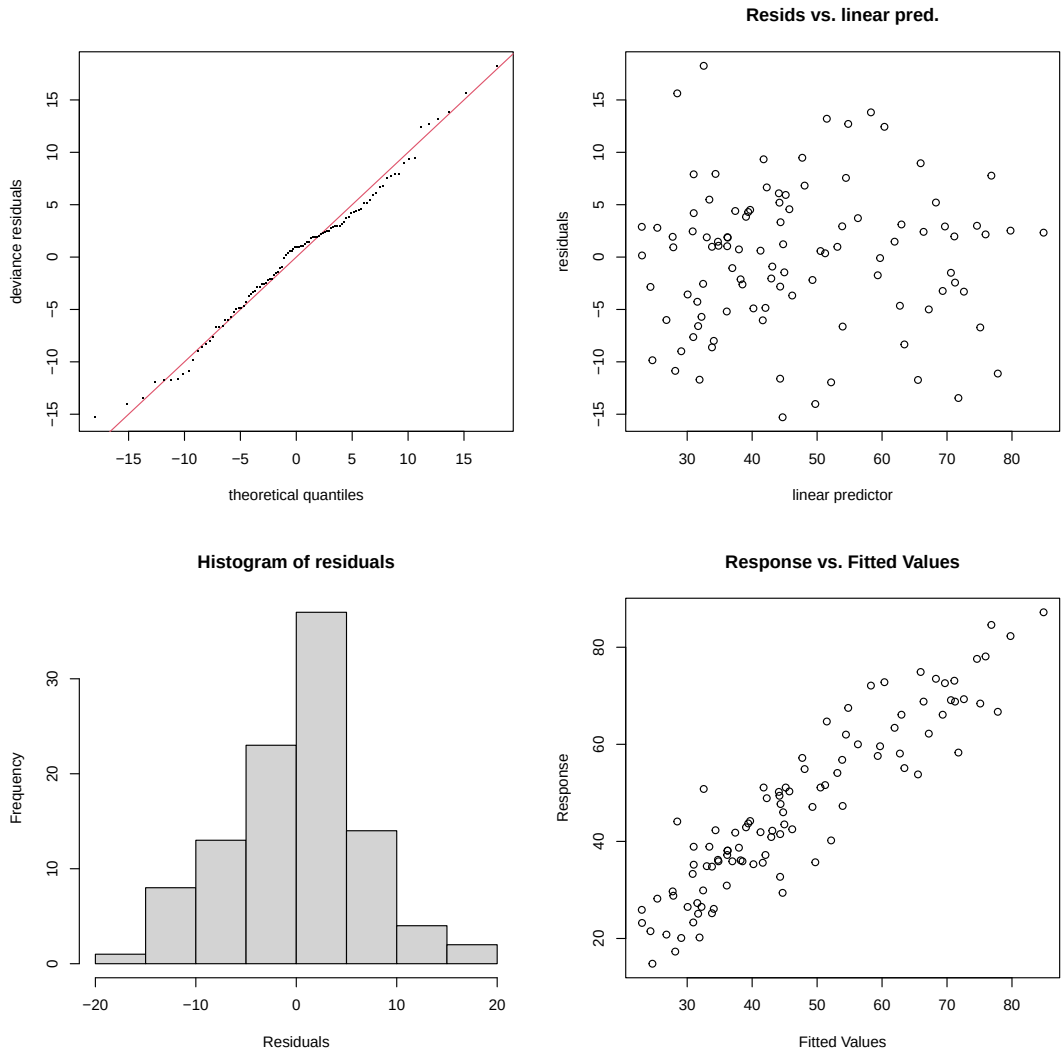


Figura 7.11: Gráficas de diagnóstico del modelo aditivo ajustado.

```
##                k'  edf k-index p-value
## s(income)    9.00 3.12    0.98    0.42
## s(education) 9.00 3.18    1.03    0.59
```

Lo ideal sería observar normalidad en los dos gráficos de la izquierda, falta de patrón en el superior derecho, y ajuste a una recta en el inferior derecho. En este caso parece que el modelo se comporta adecuadamente. Como se deduce del resultado anterior, podría ser recomendable modificar la dimensión `k` de la base utilizada construir la componente no paramétrica, este valor se puede interpretar como el grado máximo de libertad permitido en ese componente, aunque normalmente no influye demasiado en el resultado (puede influir en el tiempo de computación).

También se podría chequear concurvidad (generalización de la colinealidad; función `concurvity()`) entre las componentes del modelo:

```
concurvity(modelo)
```

```
##                  para s(income) s(education)
## worst    3.061188e-23 0.5931528    0.5931528
## observed 3.061188e-23 0.4065402    0.4398639
## estimate 3.061188e-23 0.3613674    0.4052251
```

Esta función devuelve tres medidas por componente, que tratan de medir la proporción de variación de esa componente que está contenida en el resto (similares al complementario de la tolerancia), un

valor próximo a 1 indicaría que puede haber problemas de concurvidad.

También se puede ajustar modelos GAM empleando `caret`. Por ejemplo con los métodos `"gam"` y `"gamLoess"`:

```
library(caret)
# names(getModelInfo("gam")) # 4 métodos
modelLookup("gam")
```

```
##   model parameter             label forReg forClass probModel
## 1   gam    select Feature Selection   TRUE     TRUE      TRUE
## 2   gam    method            Method   TRUE     TRUE      TRUE
```

```
modelLookup("gamLoess")
```

```
##      model parameter  label forReg forClass probModel
## 1 gamLoess      span   Span   TRUE     TRUE      TRUE
## 2 gamLoess    degree Degree   TRUE     TRUE      TRUE
```

Ejercicio 7.1

Continuando con los datos de `MASS:mcycle`, emplear `mgcv::gam()` para ajustar un spline penalizado para predecir `accel` a partir de `times` con las opciones por defecto y representar el ajuste obtenido. Comparar el ajuste con el obtenido empleando un spline penalizado adaptativo (`bs="ad"`; ver `?adaptive.smooth`).

Ejercicio 7.2

Empleando el conjunto de datos `airquality`, crear una muestra de entrenamiento y otra de test, buscar un modelo aditivo que resulte adecuado para explicar `sqrt(Ozone)` a partir de `Temp`, `Wind` y `Solar.R`. ¿Es preferible suponer que hay una interacción entre `Temp` y `Wind`?

7.4 Regresión spline adaptativa multivariante

La regresión spline adaptativa multivariante, en inglés *multivariate adaptive regression splines* [MARS; Friedman (1991)], es un procedimiento adaptativo para problemas de regresión que puede verse como una generalización tanto de la regresión lineal por pasos (*stepwise linear regression*) como de los árboles de decisión CART.

El modelo MARS es un spline multivariante lineal:

$$m(\mathbf{x}) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(\mathbf{x})$$

(es un modelo lineal en transformaciones $h_m(\mathbf{x})$ de los predictores originales), donde las bases $h_m(\mathbf{x})$ se construyen de forma adaptativa empleando funciones *bisagra* (*hinge functions*)

$$h(x) = (x)_+ = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

y considerando como posibles nodos los valores observados de los predictores (en el caso univariante se emplean las bases de potencias truncadas con $d = 1$ descritas en la Sección 7.2.1, pero incluyendo también su versión simetrizada).

Vamos a empezar explicando el modelo MARS aditivo (sin interacciones), que funciona de forma muy parecida a los árboles de decisión CART, y después lo extenderemos al caso con interacciones. Asumimos que todas las variables predictoras son numéricas. El proceso de construcción del modelo es un proceso iterativo *hacia delante* (forward) que empieza con el modelo

$$\widehat{m}(\mathbf{x}) = \widehat{\beta}_0$$

donde $\widehat{\beta}_0$ es la media de todas las respuestas, para a continuación considerar todos los puntos de corte (*knots*) posibles x_{ji} con $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, p$, es decir, todas las observaciones de todas las

variables predictoras de la muestra de entrenamiento. Para cada punto de corte x_{ji} (combinación de variable y observación) se consideran dos bases:

$$h_1(\mathbf{x}) = h(x_j - x_{ji})$$
$$h_2(\mathbf{x}) = h(x_{ji} - x_j)$$

y se construye el nuevo modelo

$$\hat{m}(\mathbf{x}) = \hat{\beta}_0 + \hat{\beta}_1 h_1(\mathbf{x}) + \hat{\beta}_2 h_2(\mathbf{x})$$

La estimación de los parámetros $\beta_0, \beta_1, \beta_2$ se realiza de la forma estándar en regresión lineal, minimizando RSS. De este modo se construyen muchos modelos alternativos y entre ellos se selecciona aquel que tenga un menor error de entrenamiento. En la siguiente iteración se conservan $h_1(\mathbf{x})$ y $h_2(\mathbf{x})$ y se añade una pareja de términos nuevos siguiendo el mismo procedimiento. Y así sucesivamente, añadiendo de cada vez dos nuevos términos. Este procedimiento va creando un modelo lineal segmentado (piecewise) donde cada nuevo término modeliza una porción aislada de los datos originales.

El *tamaño* de cada modelo es el número términos (funciones h_m) que este incorpora. El proceso iterativo se para cuando se alcanza un modelo de tamaño M, que se consigue después de incorporar $M/2$ cortes. Este modelo depende de $M+1$ parámetros β_m con $m = 0, 1, \ldots, M$. El objetivo es alcanzar un modelo lo suficientemente grande para que sobreajuste los datos, para a continuación proceder a su poda en un proceso de eliminación de variables hacia atrás (*backward deletion*) en el que se van eliminando las variables de una en una (no por parejas, como en la construcción). En cada paso de poda se elimina el término que produce el menor incremento en el error. Así, para cada tamaño $\lambda = 0, 1, \ldots, M$ se obtiene el mejor modelo estimado $\hat{m}_\lambda$.

La selección *óptima* del valor del hiperparámetro λ puede realizarse por los procedimientos habituales tipo validación cruzada. Una alternativa mucho más rápida es utilizar validación cruzada generalizada (GCV) que es una aproximación de la validación cruzada *leave-one-out* mediante la fórmula

$$\text{GCV}(\lambda) = \frac{\text{RSS}}{(1 - M(\lambda)/n)^2}$$

donde $M(\lambda)$ es el número de parámetros *efectivos* del modelo, que depende del número de términos más el número de puntos de corte utilizados penalizado por un factor (2 en el caso aditivo que estamos explicando, 3 cuando hay interacciones).

Hemos explicado una caso particular de MARS: el modelo aditivo. El modelo general sólo se diferencia del caso aditivo en que se permiten interacciones, es decir, multiplicaciones entre las variables $h_m(\mathbf{x})$. Para ello, en cada iteración durante la fase de construcción del modelo, además de considerar todos los puntos de corte, también se consideran todas las combinaciones con los términos incorporados previamente al modelo, denominados términos padre. De este modo, si resulta seleccionado un término padre $h_l(\mathbf{x})$ (incluyendo $h_0(\mathbf{x}) = 1$) y un punto de corte x_{ji}, después de analizar todas las posibilidades, al modelo anterior se le agrega

$$\hat{\beta}_{m+1} h_l(\mathbf{x}) h(x_j - x_{ji}) + \hat{\beta}_{m+2} h_l(\mathbf{x}) h(x_{ji} - x_j)$$

Recordando que en cada caso se vuelven a estimar todos los parámetros β_i.

Al igual que λ, también el grado de interacción máxima permitida se considera un hiperparámetro del problema, aunque lo habitual es trabajar con grado 1 (modelo aditivo) o interacción de grado 2. Una restricción adicional que se impone al modelo es que en cada producto no puede aparecer más de una vez la misma variable X_j.

Aunque el procedimiento de construcción del modelo realiza búsquedas exhaustivas y en consecuencia puede parecer computacionalmente intratable, en la práctica se realiza de forma razonablemente rápida, al igual que ocurría en CART. Una de las principales ventajas de MARS es que realiza una selección automática de las variables predictoras. Aunque inicialmente pueda haber muchos predictores, y este método es adecuado para problemas de alta dimensión, en el modelo final van a aparecer muchos menos (pueden aparecer más de una vez). Además, si se utiliza un modelo aditivo su interpretación es directa, e incluso permitiendo interacciones de grado 2 el modelo puede ser interpretado. Otra ventaja

es que no es necesario realizar un prepocesado de los datos, ni filtrando variables ni transformando los datos. Que haya predictores con correlaciones altas no va a afectar a la construcción del modelo (normalmente seleccionará el primero), aunque sí puede dificultar su interpretación. Aunque hemos supuesto al principio de la explicación que los predictores son numéricos, se pueden incorporar variables predictoras cualitativas siguiendo los procedimientos estándar. Por último, se puede realizar una cuantificación de la importancia de las variables de forma similar a como se hace en CART.

En conclusión, MARS utiliza splines lineales con una selección automática de los puntos de corte mediante un algoritmo avaricioso similar al empleado en los árboles CART, tratando de añadir más puntos de corte donde aparentemente hay más variaciones en la función de regresión y menos puntos donde esta es más estable.

7.4.1 MARS con el paquete `earth`

Actualmente el paquete de referencia para MARS es `earth` (*Enhanced Adaptive Regression Through Hinges*)[9].

La función principal es `earth()` y se suelen considerar los siguientes argumentos:

```
earth(formula, data, glm = NULL, degree = 1, ...)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (e.g. `respuesta ~ .`; también admite matrices). Admite respuestas multidimensionales (ajustará un modelo para cada componente) y categóricas (las convierte en multivariantes), también predictores categóricos, aunque no permite datos faltantes.
- `glm`: lista con los parámetros del ajuste GLM (e.g. `glm = list(family = binomial)`).
- `degree`: grado máximo de interacción; por defecto 1 (modelo aditivo).

Otros parámetros que pueden ser de interés (afectan a la complejidad del modelo en el crecimiento, a la selección del modelo final o al tiempo de computación; para más detalles ver `help(earth)`):

- `nk`: número máximo de términos en el crecimiento del modelo (dimensión M de la base); por defecto `min(200, max(20, 2 * ncol(x))) + 1` (puede ser demasiado pequeña si muchos de los predictores influyen en la respuesta).
- `thresh`: umbral de parada en el crecimiento (se interpretaría como `cp` en los árboles CART); por defecto 0.001 (si se establece a 0 la única condición de parada será alcanzar el valor máximo de términos `nk`).
- `fast.k`: número máximo de términos padre considerados en cada paso durante el crecimiento; por defecto 20, si se establece a 0 no habrá limitación.
- `linpreds`: índice de variables que se considerarán con efecto lineal.
- `nprune`: número máximo de términos (incluida la intersección) en el modelo final (después de la poda); por defecto no hay límite (se podrían incluir todos los creados durante el crecimiento).
- `pmethod`: método empleado para la poda; por defecto `"backward"`. Otras opciones son: `"forward"`, `"seqrep"`, `"exhaustive"` (emplea los métodos de selección implementados en paquete `leaps`), `"cv"` (validación cruzada, empleando `nflod`) y `"none"` para no realizar poda.
- `nfold`: número de grupos de validación cruzada; por defecto 0 (no se hace validación cruzada).
- `varmod.method`: permite seleccionar un método para estimar las varianzas y por ejemplo poder realizar contrastes o construir intervalos de confianza (para más detalles ver `?varmod` o la vignette "Variance models in earth").

Utilizaremos como ejemplo inicial los datos de `MASS:mcycle` (ver Figura 7.12):

[9] Desarrollado a partir de la función `mda::mars()` de T. Hastie y R. Tibshirani. Utiliza este nombre porque MARS está registrado para un uso comercial por Salford Systems.

```
# data(mcycle, package = "MASS")
library(earth)
mars <- earth(accel ~ times, data = mcycle)
# mars
summary(mars)
```

```
## Call: earth(formula=accel~times, data=mcycle)
##
##              coefficients
## (Intercept)     -90.992956
## h(19.4-times)     8.072585
## h(times-19.4)     9.249999
## h(times-31.2)   -10.236495
##
## Selected 4 of 6 terms, and 1 of 1 predictors
## Termination condition: RSq changed by less than 0.001 at 6 terms
## Importance: times
## Number of terms at each degree of interaction: 1 3 (additive model)
## GCV 1119.813    RSS 133670.3    GRSq 0.5240328    RSq 0.5663192
```

```
plot(mars)
```

Por defecto, se representa un resumen de los errores de validación en la selección del modelo, la distribución empírica y el gráfico QQ de los residuos, y los residuos frente a las predicciones (en la muestra de entrenamiento).

Podemos representar el ajuste obtenido (ver Figura 7.13):

```
plot(accel ~ times, data = mcycle, col = 'darkgray')
lines(mcycle$times, predict(mars))
```

Como con las opciones por defecto el ajuste no es muy bueno (aunque puede ser suficiente para un análisis preliminar), podríamos forzar la complejidad del modelo en el crecimiento (`minspan = 1` permite que todas las observaciones sean potenciales nodos; ver Figura 7.14):

```
mars2 <- earth(accel ~ times, data = mcycle, minspan = 1, thresh = 0)
summary(mars2)
```

```
## Call: earth(formula=accel~times, data=mcycle, minspan=1, thresh=0)
##
##              coefficients
## (Intercept)      -6.274366
## h(times-14.6)   -25.333056
## h(times-19.2)    32.979264
## h(times-25.4)   153.699248
## h(times-25.6)  -145.747392
## h(times-32)     -30.041076
## h(times-35.2)    13.723887
##
## Selected 7 of 12 terms, and 1 of 1 predictors
## Termination condition: Reached nk 21
## Importance: times
## Number of terms at each degree of interaction: 1 6 (additive model)
## GCV 623.5209    RSS 67509.03    GRSq 0.7349776    RSq 0.7809732
```

```
plot(accel ~ times, data = mcycle, col = 'darkgray')
lines(mcycle$times, predict(mars2))
```

Como siguiente ejemplo consideramos los datos de `carData::Prestige` (ver Figura 7.15):

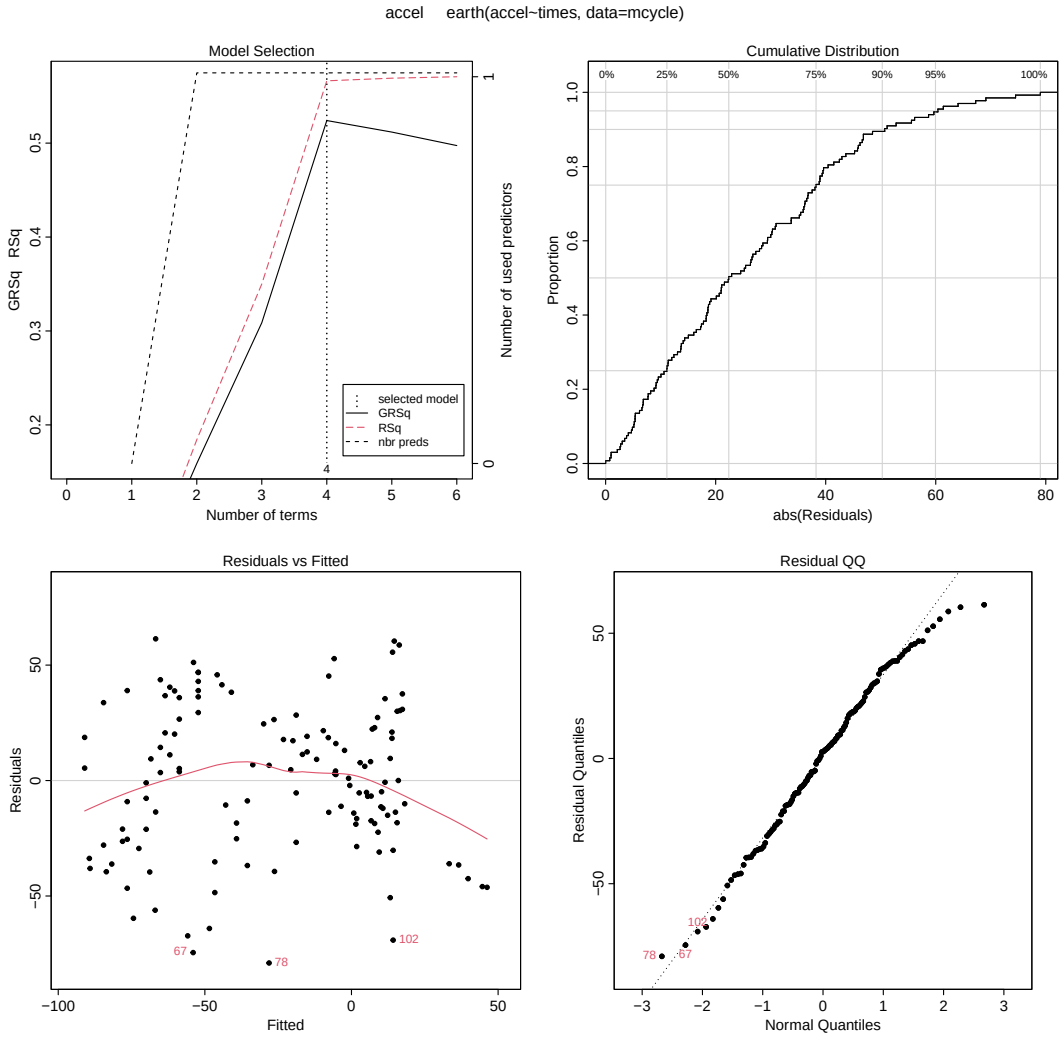


Figura 7.12: Resultados de validación del modelo MARS univariante (empleando la función `earth()` con parámetros por defecto y `MASS:mcycle`).

```
# data(Prestige, package = "carData")
mars <- earth(prestige ~ education + income + women, data = Prestige,
              degree = 2, nk = 40)
summary(mars)
```

```
## Call: earth(formula=prestige~education+income+women, data=Prestige, degree=2,
##             nk=40)
##
##                                coefficients
## (Intercept)                      19.9845240
## h(education-9.93)                 5.7683265
## h(income-3161)                    0.0085297
## h(income-5795)                   -0.0080222
## h(women-33.57)                    0.2154367
## h(income-5299) * h(women-4.14)   -0.0005163
## h(income-5795) * h(women-4.28)    0.0005409
##
## Selected 7 of 31 terms, and 3 of 3 predictors
## Termination condition: Reached nk 40
## Importance: education, income, women
```

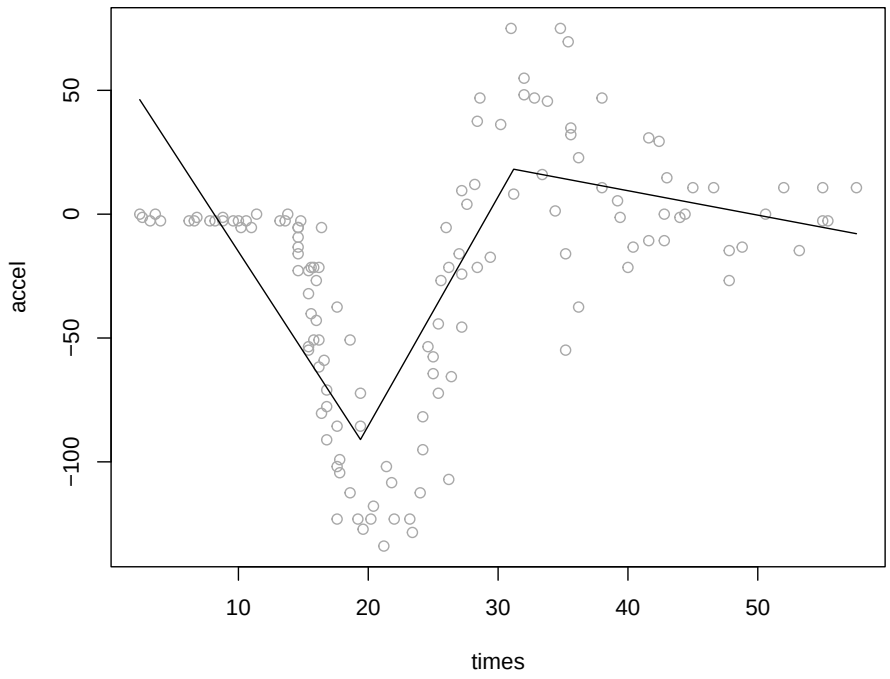


Figura 7.13: Ajuste del modelo MARS univariante (obtenido con la función `earth()` con parámetros por defecto) para predecir `accel` en función de `times`.

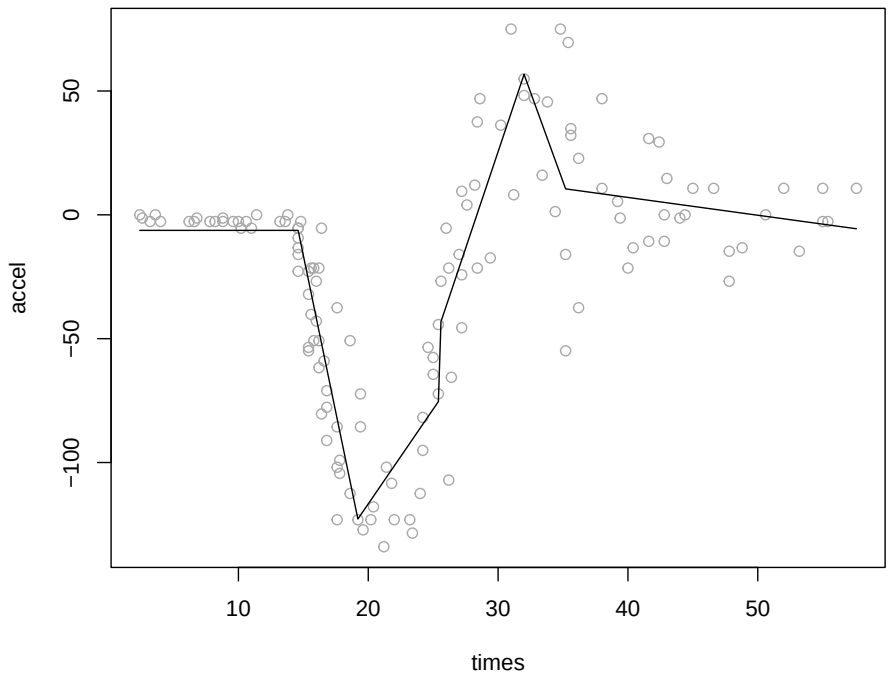


Figura 7.14: Ajuste del modelo MARS univariante (con la función `earth()` con parámetros `minspan = 1` y `thresh = 0`).

```
  ## Number of terms at each degree of interaction: 1 4 2
  ## GCV 53.08737    RSS 3849.355    GRSq 0.8224057    RSq 0.8712393
plot(mars)
```

Para representar los efectos de las variables importa las herramientas del paquete `plotmo` (del mismo autor; válido también para la mayoría de los modelos tratados en este libro, incluyendo `mgcv::gam()`; ver Figura 7.16):

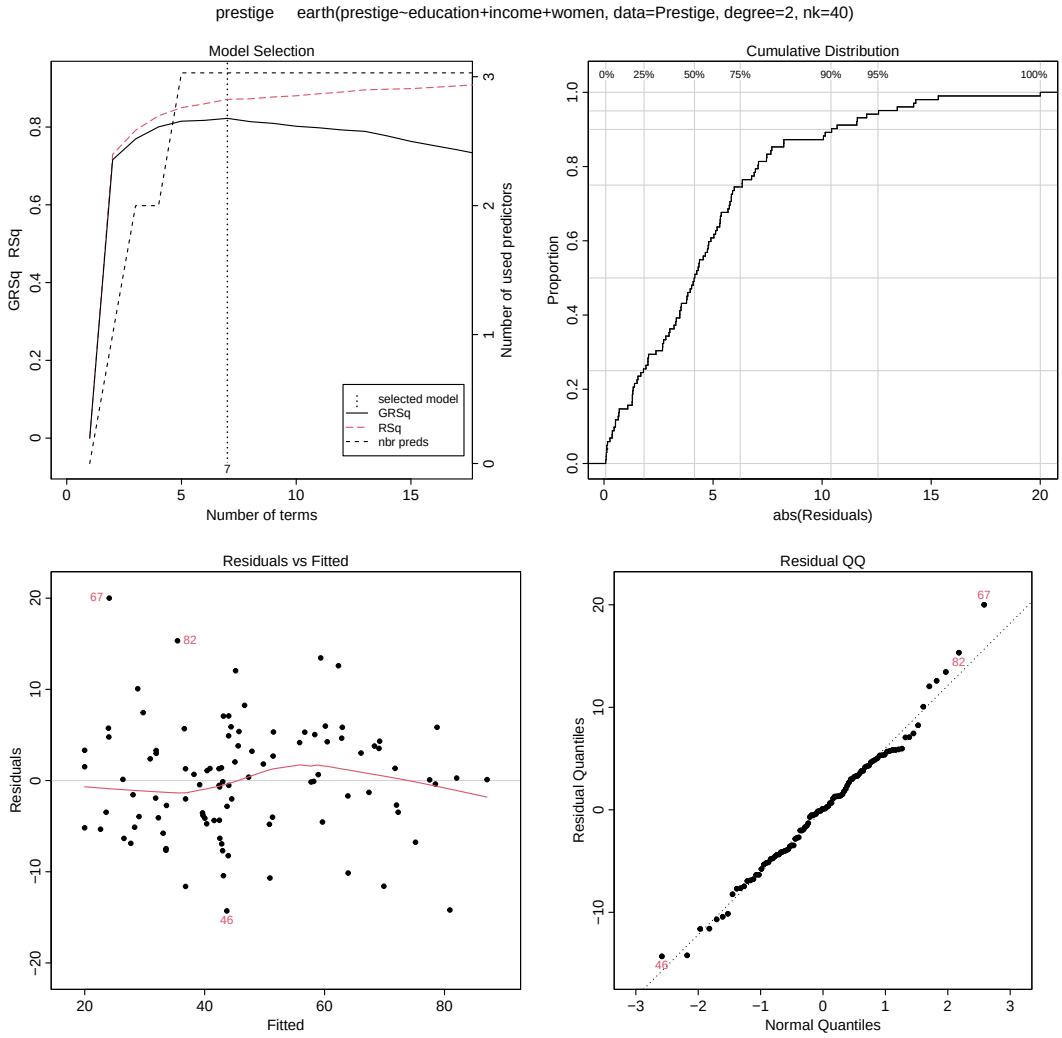


Figura 7.15: Resultados de validación del ajuste del modelo MARS multivariante (para `carData::Prestige`).

```
plotmo(mars)
```

```
##  plotmo grid:    education income women
##                      10.54   5930  13.6
```

También podemos obtener la importancia de las variables (función `evimp()`) y representarla gráficamente (método `plot.evimp()`; ver Figura 7.17):

```
varimp <- evimp(mars)
varimp
```

```
##           nsubsets   gcv    rss
## education        6 100.0  100.0
## income           5  36.0   40.3
## women            3  16.3   22.0
```

```
plot(varimp)
```

Para finalizar, destacar que podríamos tener en cuenta este modelo como punto de partida para ajustar un modelo GAM más flexible (como se mostró en la Sección 7.3). Por ejemplo:

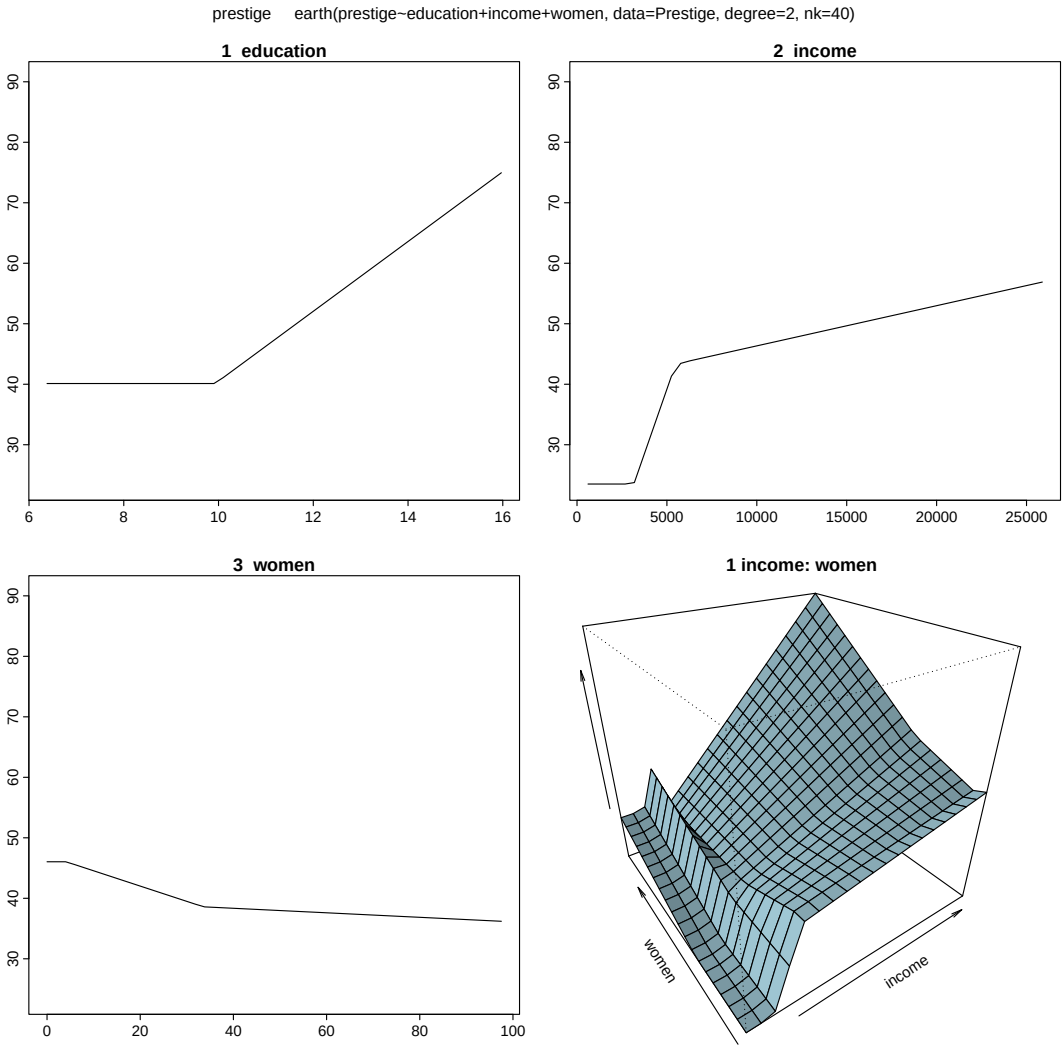


Figura 7.16: Efectos parciales de las componentes del modelo MARS ajustado.

```
# library(mgcv)
gam <- gam(prestige ~ s(education) + s(income) + s(women), data = Prestige, select = TRUE)
summary(gam)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## prestige ~ s(education) + s(income) + s(women)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  46.8333     0.6461   72.49   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                edf Ref.df     F p-value
## s(education) 2.349      9 9.926 < 2e-16 ***
```

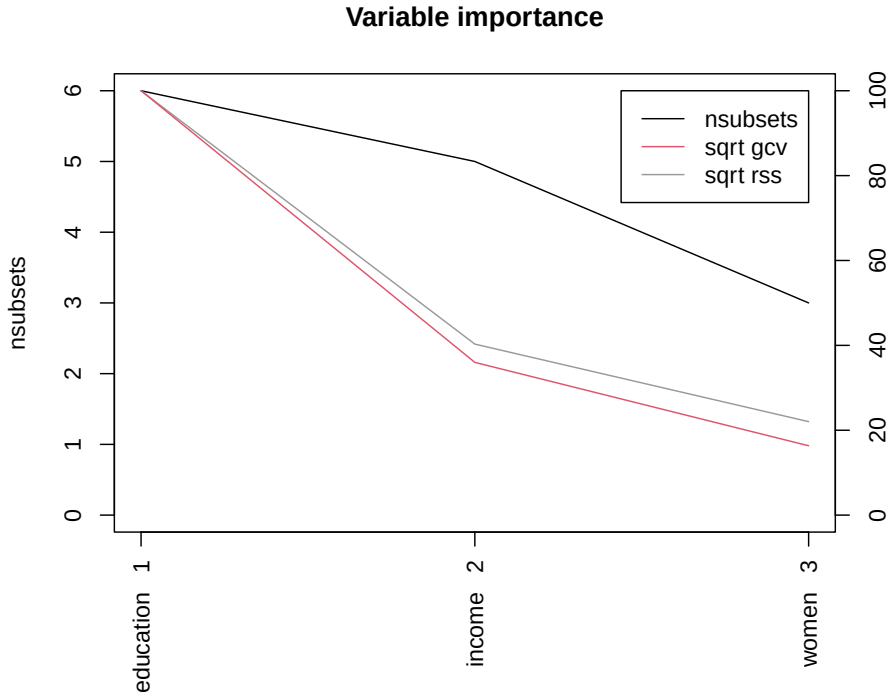


Figura 7.17: Importancia de los predictores incluidos en el modelo MARS.

```
## s(income)    6.289      9 7.420 < 2e-16 ***
## s(women)     1.964      9 1.309 0.00149 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.856   Deviance explained = 87.1%
## GCV = 48.046  Scale est. = 42.58     n = 102
```

```
gam2 <- gam(prestige ~ s(education) + s(income, women), data = Prestige)
summary(gam2)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## prestige ~ s(education) + s(income, women)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   46.833      0.679   68.97   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                   edf Ref.df     F p-value
## s(education)    2.802  3.489 25.09  <2e-16 ***
## s(income,women) 4.895  6.286 10.03  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.841   Deviance explained = 85.3%
```

```
  ## GCV = 51.416  Scale est. = 47.032    n = 102
```

```
anova(gam, gam2, test = "F")
```

```
  ## Analysis of Deviance Table
  ##
  ## Model 1: prestige ~ s(education) + s(income) + s(women)
  ## Model 2: prestige ~ s(education) + s(income, women)
  ##   Resid. Df Resid. Dev      Df Deviance      F  Pr(>F)
  ## 1    88.325     3849.1
  ## 2    91.225     4388.3 -2.9001  -539.16 4.3661 0.00705 **
  ## ---
  ## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
plotmo(gam2)
```

```
  ##  plotmo grid:    education income women
  ##                      10.54   5930  13.6
```

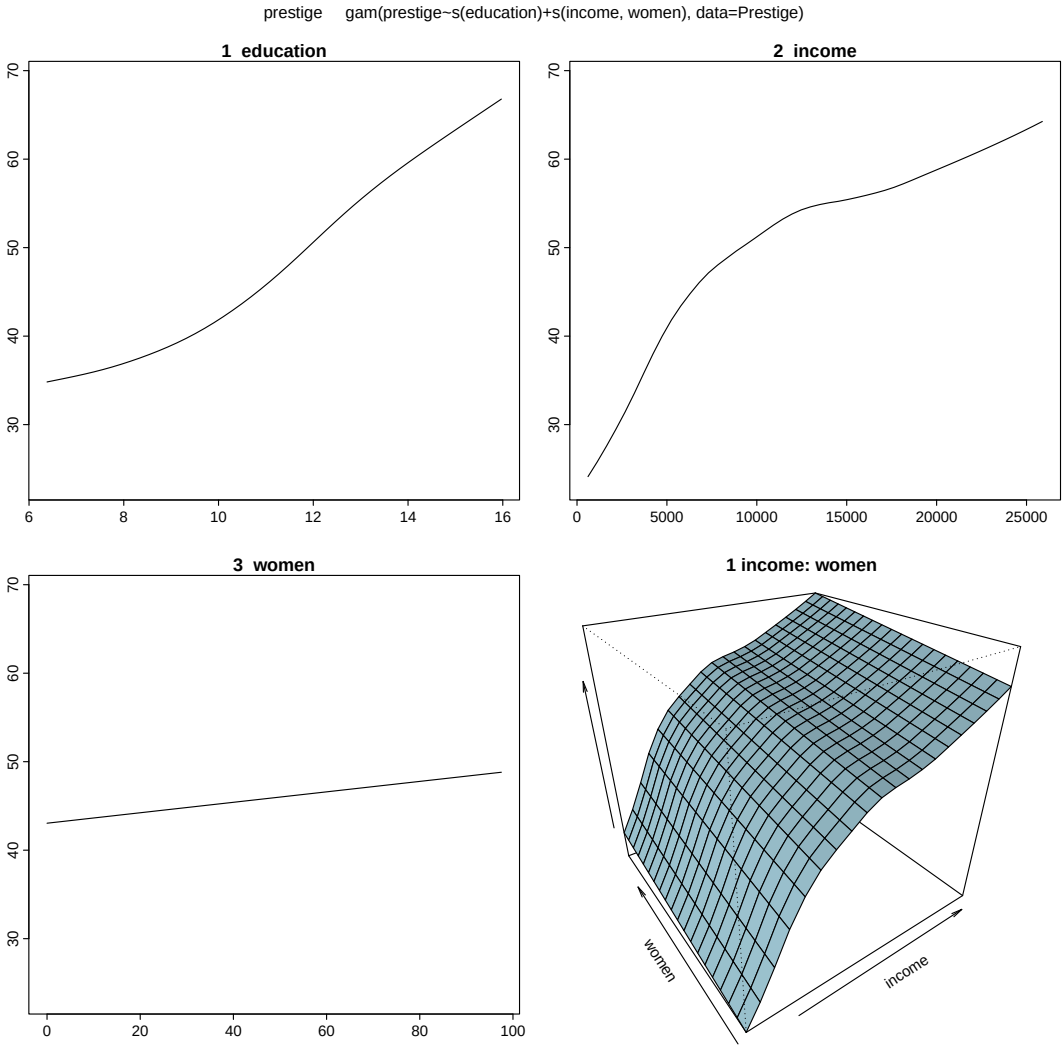


Figura 7.18: Efectos parciales de las componentes del modelo GAM con interacción (para `carData::Prestige`).

En la Figura 7.18 (generada con `plotmo::plotmo()`) se representan los efectos parciales de las componentes, y en la Figura 7.19 el efecto parcial de la interacción (empleando `plot()`):

```
plot(gam2, scheme = 2, select = 2)
```

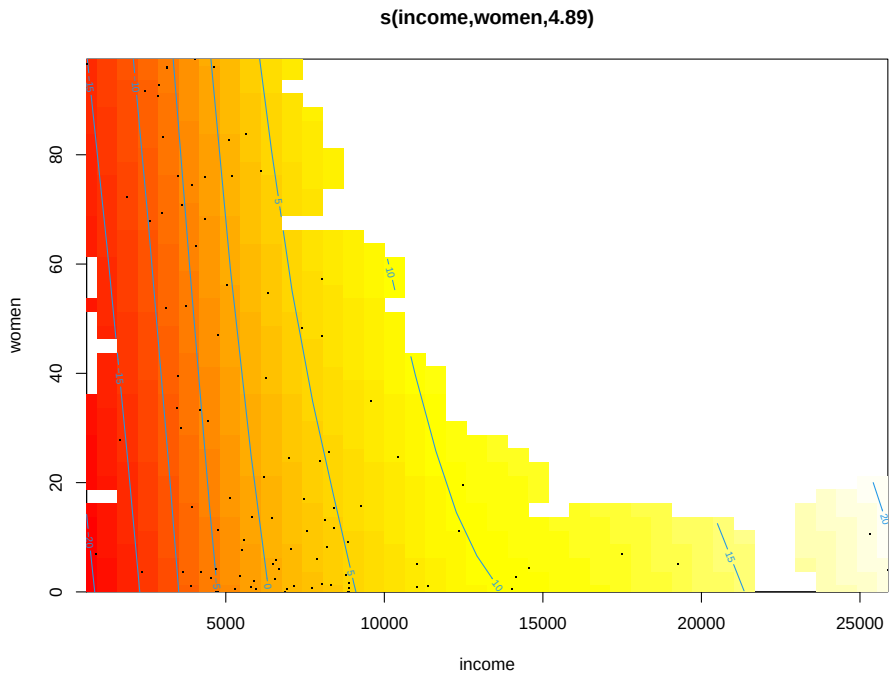


Figura 7.19: Efecto parcial de la interacción `income:women`.

Ejercicio 7.3

Comentar brevemente los resultados del ajuste del modelo GAM del ejemplo anterior. ¿Observas algo extraño en el contraste ANOVA? (Probar a ejecutar `anova(gam2, gam, test = "F")`.)

7.4.2 MARS con el paquete `caret`

Emplearemos como ejemplo el conjunto de datos `earth::Ozone1`:

```
# data(ozone1, package = "earth")
df <- ozone1
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
train <- df[itrain, ]
test <- df[-itrain, ]
```

`caret` implementa varios métodos basados en `earth`, en este caso emplearemos el algoritmo original:

```
library(caret)
# names(getModelInfo("[Ee]arth")) # 4 métodos
modelLookup("earth")
```

```
##   model parameter          label forReg forClass probModel
## 1 earth    nprune         #Terms   TRUE     TRUE      TRUE
## 2 earth    degree Product Degree   TRUE     TRUE      TRUE
```

Para selección de los hiperparámetros óptimos consideramos una rejilla de búsqueda personalizada:

```
tuneGrid <- expand.grid(degree = 1:2,
                        nprune = floor(seq(2, 20, len = 10)))
set.seed(1)
caret.mars <- train(O3 ~ ., data = train, method = "earth",
    trControl = trainControl(method = "cv", number = 10),
```

```
    tuneGrid = tuneGrid)
caret.mars
```

```
## Multivariate Adaptive Regression Spline
##
## 264 samples
##   9 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 238, 238, 238, 236, 237, 239, ...
## Resampling results across tuning parameters:
##
##   degree  nprune  RMSE      Rsquared   MAE
##   1        2      4.842924  0.6366661  3.803870
##   1        4      4.558953  0.6834467  3.488040
##   1        6      4.345781  0.7142046  3.413213
##   1        8      4.256592  0.7295113  3.220256
##   1       10      4.158604  0.7436812  3.181941
##   1       12      4.128416  0.7509562  3.142176
##   1       14      4.069714  0.7600561  3.061458
##   1       16      4.058769  0.7609245  3.058843
##   1       18      4.058769  0.7609245  3.058843
##   1       20      4.058769  0.7609245  3.058843
##   2        2      4.842924  0.6366661  3.803870
##   2        4      4.652783  0.6725979  3.540031
##  [ reached getOption("max.print") -- omitted 8 rows ]
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were nprune = 10 and degree = 2.
```

```
ggplot(caret.mars, highlight = TRUE)
```

Podemos analizar el modelo final con las herramientas de `earth`:

```
summary(caret.mars$finalModel)
```

```
## Call: earth(x=matrix[264,9], y=c(4,13,16,3,6,2...), keepxy=TRUE, degree=2,
##             nprune=10)
##
##                                coefficients
## (Intercept)                      11.6481994
## h(dpg-15)                        -0.0743900
## h(ibt-110)                        0.1224848
## h(17-vis)                        -0.3363332
## h(vis-17)                        -0.0110360
## h(101-doy)                       -0.1041604
## h(doy-101)                       -0.0236813
## h(wind-3) * h(1046-ibh)          -0.0023406
## h(humidity-52) * h(15-dpg)       -0.0047940
## h(60-humidity) * h(ibt-110)      -0.0027632
##
## Selected 10 of 21 terms, and 7 of 9 predictors (nprune=10)
## Termination condition: Reached nk 21
## Importance: humidity, ibt, dpg, doy, wind, ibh, vis, temp-unused, ...
## Number of terms at each degree of interaction: 1 6 3
## GCV 13.84161    RSS 3032.585    GRSq 0.7846289    RSq 0.8199031
```

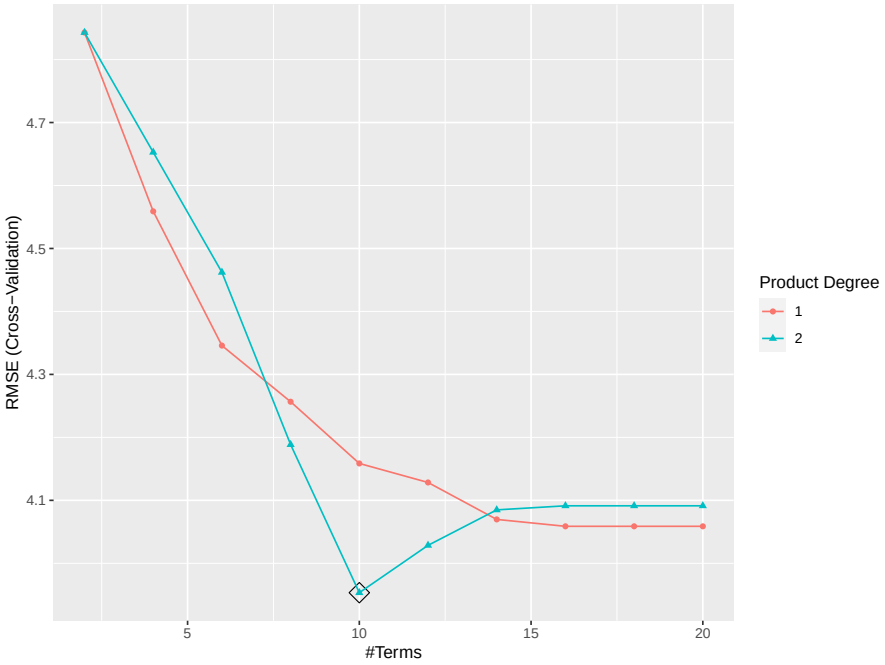


Figura 7.20: Errores RMSE de validación cruzada de los modelos MARS en función del numero de términos `nprune` y del orden máximo de interacción `degree`, resaltando la combinación óptima.

Representamos los efectos parciales de las componentes, separando los efectos principales (ver Figura 7.21) de las interacciones (ver Figura 7.22):

```
# plotmo(caret.mars$finalModel)
plotmo(caret.mars$finalModel, degree2 = 0, caption = "")
```

```
##  plotmo grid:    vh wind humidity temp    ibh dpg   ibt vis   doy
##                5770    5     64.5   62 2046.5  24 169.5 100 213.5
```

```
plotmo(caret.mars$finalModel, degree1 = 0, caption = "")
```

Finalmente evaluamos la precisión de las predicciones en la muestra de test con el procedimiento habitual:

```
pred <- predict(caret.mars, newdata = test)
accuracy(pred, test$O3)
```

```
##           me         rmse          mae          mpe         mape   r.squared
##    0.4817913    4.0952444    3.0764376 -14.1288949   41.2602037    0.7408061
```

7.5 Projection pursuit

Projection pursuit (Friedman y Tukey, 1974) es una técnica de análisis exploratorio de datos multivariantes que busca proyecciones lineales de los datos en espacios de dimensión baja, siguiendo una idea originalmente propuesta en (Kruskal, 1969). Inicialmente se presentó como una técnica gráfica y por ese motivo buscaba proyecciones de dimensión 1 o 2 (proyecciones en rectas o planos), resultando que las direcciones interesantes son aquellas con distribución no normal. La motivación es que cuando se realizan transformaciones lineales lo habitual es que el resultado tenga la apariencia de una distribución normal (por el teorema central del límite), lo cual oculta las singularidades de los datos originales. Se supone que los datos son una trasformación lineal de componentes no gaussianas (variables latentes) y la idea es deshacer esta transformación mediante la optimización de una función objetivo, que en este contexto recibe el nombre de *projection index*. Aunque con orígenes distintos, *projection*

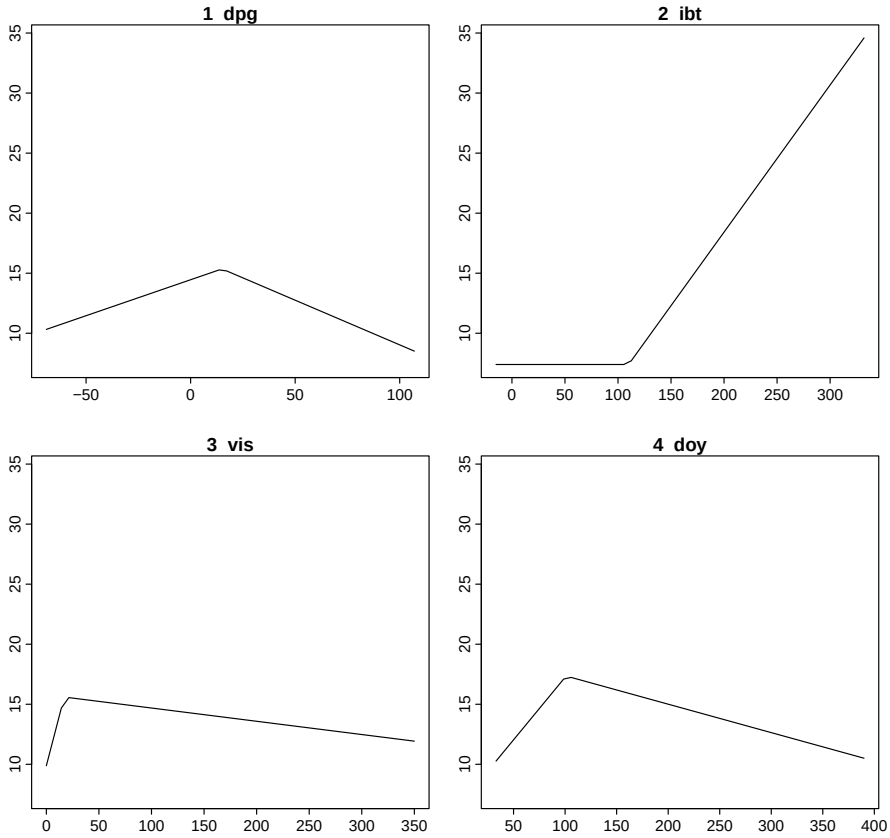


Figura 7.21: Efectos parciales principales del modelo MARS ajustado con `caret`.

pursuit es muy similar a *independent component analysis* (Comon, 1994), una técnica de reducción de la dimensión que, en lugar de buscar como es habitual componentes incorreladas (ortogonales), busca componentes independientes y con distribución no normal (ver por ejemplo la documentación del paquete `fastICA`).

Hay extensiones de *projection pursuit* para regresión, clasificación, estimación de la función de densidad, etc.

7.5.1 Regresión por *projection pursuit*

En el método original de *projection pursuit regression* (PPR, Friedman y Stuetzle, 1981) se considera el siguiente modelo semiparamétrico

$$m(\mathbf{x}) = \sum_{m=1}^{M} g_m(\alpha_{1m}x_1 + \alpha_{2m}x_2 + ... + \alpha_{pm}x_p)$$

siendo $\alpha_m = (\alpha_{1m}, \alpha_{2m}, ..., \alpha_{pm})$ vectores de parámetros (desconocidos) de módulo unitario y g_m funciones suaves (desconocidas), denominadas funciones *ridge*.

Con esta aproximación se obtiene un modelo muy general que evita los problemas de la maldición de la dimensionalidad. De hecho se trata de un *aproximador universal*, con M suficientemente grande y eligiendo adecuadamente las componentes se podría aproximar cualquier función continua. Sin embargo el modelo resultante puede ser muy difícil de interpretar, salvo el caso de $M = 1$ que se corresponde con el denominado *single index model* empleado habitualmente en Econometría, pero que solo es algo más general que el modelo de regresión lineal múltiple.

El ajuste se este tipo de modelos es en principio un problema muy complejo. Hay que estimar las funciones univariantes g_m (utilizando un método de suavizado) y los parámetros α_{im}, utilizando como criterio de error RSS. En la práctica se resuelve utilizando un proceso iterativo en el que se van

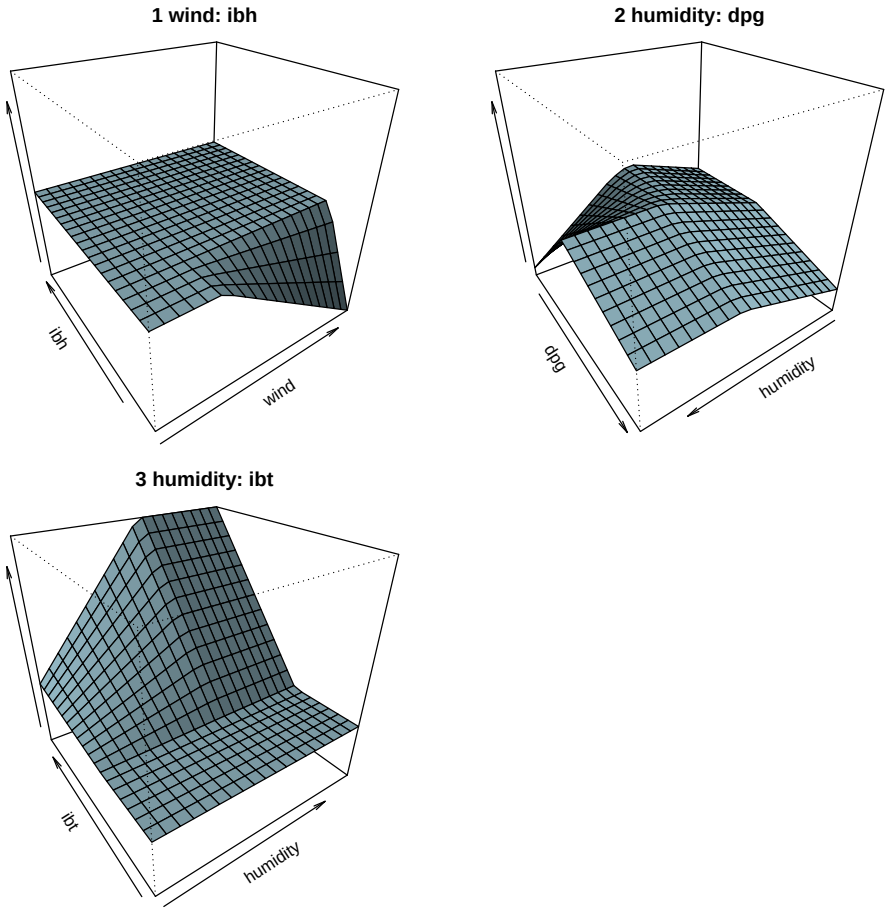


Figura 7.22: Efectos parciales principales de las interacciones del modelo MARS ajustado con `caret`.

fijando sucesivamente los valores de los parámetros y las funciones *ridge* (si son estimadas empleando un método que también proporcione estimaciones de su derivada, las actualizaciones de los parámetros se pueden obtener por mínimos cuadrados ponderados).

También se han desarrollado extensiones del método original para el caso de respuesta multivariante:

$$m_i(\mathbf{x}) = \beta_{i0} + \sum_{m=1}^{M} \beta_{im} g_m(\alpha_{1m} x_1 + \alpha_{2m} x_2 + ... + \alpha_{pm} x_p)$$

reescalando las funciones *rigde* de forma que tengan media cero y varianza unidad sobre las proyecciones de las observaciones.

Este procedimiento de regresión está muy relacionado con las redes de neuronas artificiales que se tratarán en el siguiente capítulo y que han sido de mayor objeto de estudio y desarrollo en los último años.

7.5.2 Implementación en R

El método PPR (con respuesta multivariante) está implementado en la función `ppr()` del paquete base de R[10], y es también la empleada por el método `"ppr"` de `caret`. Esta función:

```
ppr(formula, data, nterms, max.terms = nterms, optlevel = 2,
    sm.method = c("supsmu", "spline", "gcvspline"),
    bass = 0, span = 0, df = 5, gcvpen = 1, ...)
```

[10]Basada en la función `ppreg()` de S-PLUS e implementado en R por B.D. Ripley inicialmente para el paquete `MASS`.

va añadiendo términos *ridge* hasta un máximo de `max.terms` y posteriormente emplea un método hacia atrás para seleccionar `nterms` (el argumento `optlevel` controla como se vuelven a reajustar los términos en cada iteración). Por defecto emplea el *super suavizador* de Friedman (función `supsmu()`, con parámetros `bass` y `spam`), aunque también admite splines (función `smooth.spline()`, fijando los grados de libertad con `df` o seleccionándolos mediante GCV). Para más detalles ver `help(ppr)`.

Continuaremos con el ejemplo del conjunto de datos `earth::Ozone1`. En primer lugar ajustamos un modelo PPR con dos términos (incrementando el suavizado por defecto de `supsmu()` siguiendo la recomendación de Venables y Ripley, 2002):

```
ppreg <- ppr(O3 ~ ., nterms = 2, data = train, bass = 2)
summary(ppreg)
```

```
## Call:
## ppr(formula = O3 ~ ., data = train, nterms = 2, bass = 2)
##
## Goodness of fit:
##  2 terms
## 4033.668
##
## Projection direction vectors ('alpha'):
##          term 1       term 2
## vh       -0.016617786  0.047417127
## wind     -0.317867945 -0.544266150
## humidity  0.238454606 -0.786483702
## temp      0.892051760 -0.012563393
## ibh      -0.001707214 -0.001794245
## dpg       0.033476907  0.285956216
## ibt       0.205536326  0.026984921
## vis      -0.026255153 -0.014173612
## doy      -0.044819013 -0.010405236
##
## Coefficients of ridge terms ('beta'):
##   term 1   term 2
## 6.790447 1.531222
```

Representamos las funciones rigde (ver Figura 7.23):

```
oldpar <- par(mfrow = c(1, 2))
plot(ppreg)
```

```
par(oldpar)
```

Evaluamos las predicciones en la muestra de test:

```
pred <- predict(ppreg, newdata = test)
obs <- test$O3
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
##  0.4819794  3.2330060  2.5941476 -6.1203121 34.8728543  0.8384607
```

Podemos emplear el método `"ppr"` de `caret` para seleccionar el número de términos (ver Figura 7.24):

```
library(caret)
modelLookup("ppr")
```

```
##   model parameter   label forReg forClass probModel
## 1   ppr    nterms # Terms   TRUE    FALSE     FALSE
```

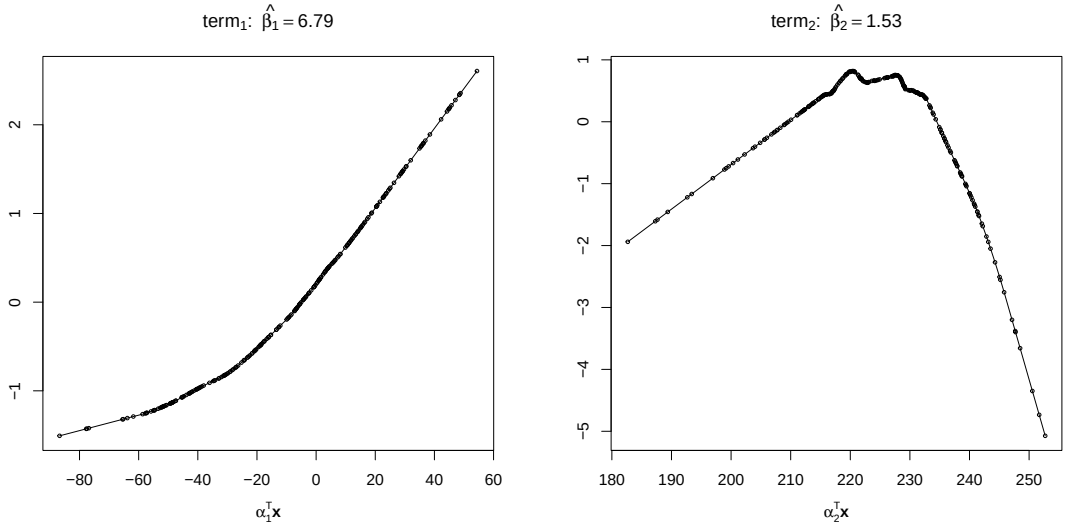


Figura 7.23: Estimaciones de las funciones *ridge* del ajuste PPR.

```
set.seed(1)
caret.ppr <- train(O3 ~ ., data = train, method = "ppr", # bass = 2,
    trControl = trainControl(method = "cv", number = 10))
caret.ppr
```

```
## Projection Pursuit Regression
##
## 264 samples
##   9 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 238, 238, 238, 236, 237, 239, ...
## Resampling results across tuning parameters:
##
##   nterms  RMSE      Rsquared   MAE
##   1       4.366022  0.7069042  3.306658
##   2       4.479282  0.6914678  3.454853
##   3       4.624943  0.6644089  3.568929
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was nterms = 1.
```

```
ggplot(caret.ppr, highlight = TRUE)
```

Analizamos el modelo final ajustado (ver Figura 7.25):

```
summary(caret.ppr$finalModel)
```

```
## Call:
## ppr(x = as.matrix(x), y = y, nterms = param$nterms)
##
## Goodness of fit:
##  1 terms
## 4436.727
##
## Projection direction vectors ('alpha'):
##           vh         wind     humidity         temp          ibh          dpg
```

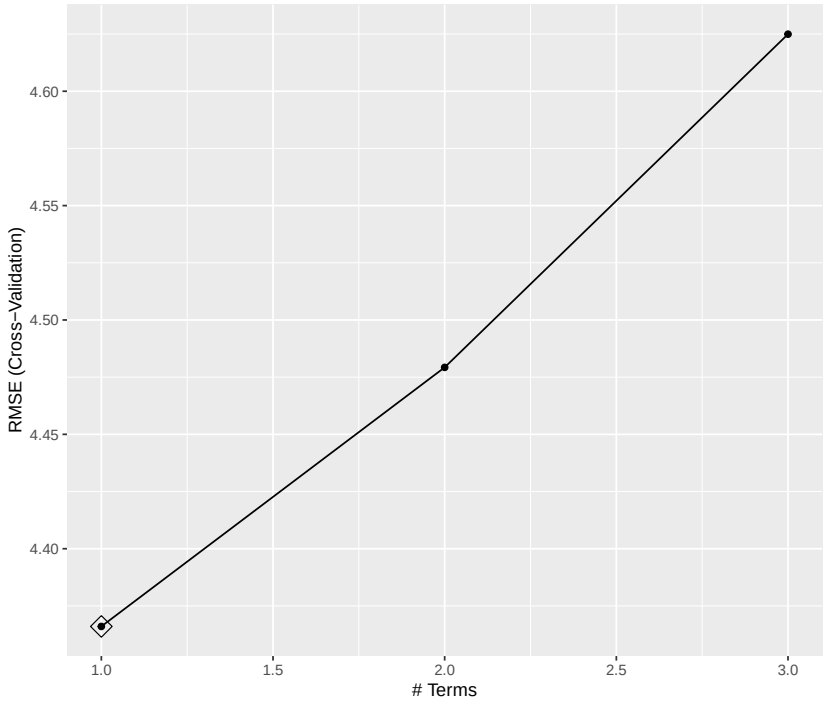


Figura 7.24: Errores RMSE de validación cruzada de los modelos PPR en función del numero de términos `nterms`, resaltando el valor óptimo.

```
## -0.016091543 -0.167891347  0.351773894  0.907301452 -0.001828865  0.026901492
##          ibt          vis          doy
##  0.148021198 -0.026470384 -0.035703896
##
## Coefficients of ridge terms ('beta'):
##   term 1
## 6.853971
```

```
plot(caret.ppr$finalModel)
```

```
# varImp(caret.ppr) # emplea una medida genérica de importancia
pred <- predict(caret.ppr, newdata = test)
accuracy(pred, obs)
```

```
##          me        rmse         mae         mpe        mape   r.squared
##   0.3135877   3.3652891   2.7061615 -10.7532705  33.8333646   0.8249710
```

Para ajustar un modelo *single index* también se podría emplear la función `npindex()` del paquete `np` (que implementa el método de Ichimura, 1993, considerando un estimador local constante), aunque en este caso ni el tiempo de computación ni el resultado es satisfactorio[11]:

```
library(np)
bw <- npindexbw(O3 ~ vh + wind + humidity + temp + ibh + dpg + ibt + vis + doy,
                data = train, optim.method = "BFGS", nmulti = 1) # Por defecto nmulti = 5
# summary(bw)
```

[11] No admite una fórmula del tipo `respuesta ~ .`:

```
bw <- npindexbw(O3 ~ ., data = train)
# Error in terms.formula(formula): '.' in formula and no 'data' argument
formula <- reformulate(setdiff(colnames(train), "O3"), response = "O3") # Escribe la formula explícitamente
```

El valor por defecto de `nmulti = 5` (número de reinicios con punto de partida aleatorio del algoritmo de optimización) incrementa el tiempo de computación. Además, los resultados de texto contienen caracteres inválidos para compilar en LaTeX.

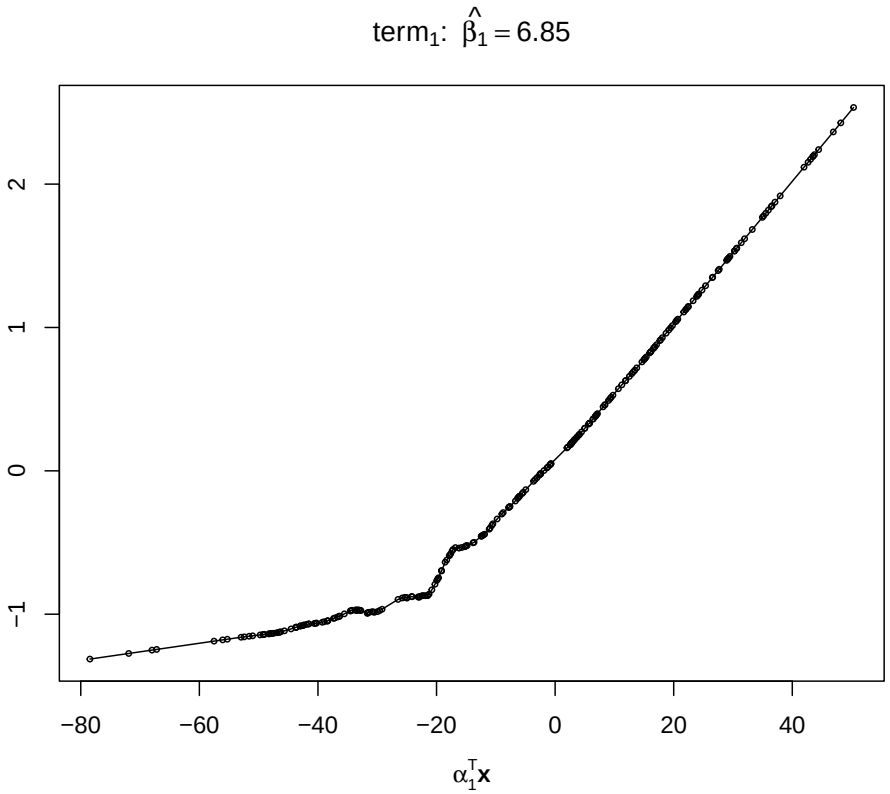


Figura 7.25: Estimación de la función *ridge* del ajuste PPR (con selección óptima del número de componentes).

```
sindex <- npindex(bws = bw, gradients = TRUE)
summary(sindex)
```

```
##
## Single Index Model
## Regression Data: 264 training points, in 9 variable(s)
##
##       vh     wind humidity     temp        ibh      dpg      ibt       vis
## Beta:  1 10.85006 6.264221 8.855986 0.09266013 4.003849 5.662514 -0.661448
##            doy
## Beta: -1.11846
## Bandwidth: 13.79708
## Kernel Regression Estimator: Local-Constant
##
## Residual standard error: 3.261427
## R-squared: 0.8339121
##
## Continuous Kernel Type: Second-Order Gaussian
## No. Continuous Explanatory Vars.: 1
```

Al representar la función *ridge* se observa que aparentemente la ventana seleccionada produce un infrasuavizado (sobreajuste; ver Figura 7.26):

```
plot(bw)
```

```
pred <- predict(sindex, newdata = test)
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
```

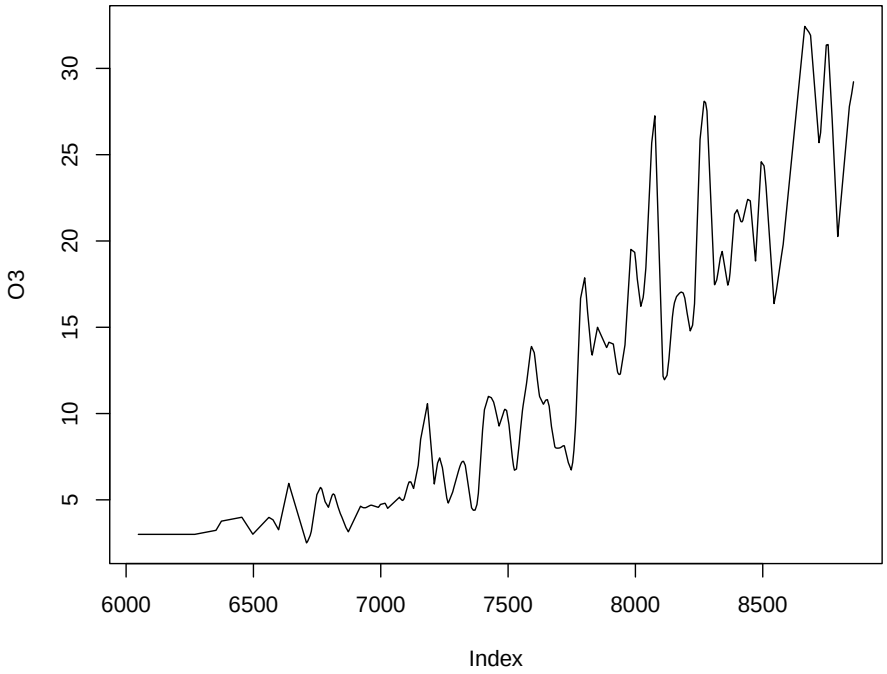


Figura 7.26: Estimación de la función *ridge* del modelo *single index* ajustado.

```
##  0.3502566  4.7723853  3.6367933 -8.8225529 38.2419105  0.6480052
```

Capítulo 8

Redes neuronales

Las redes neuronales (McCulloch y Pitts, 1943), también conocidas como redes de neuronas artificiales (*artificial neural network*; ANN), son una metodología de aprendizaje supervisado que destaca porque da lugar a modelos con un número muy elevado de parámetros, adecuada para abordar problemas con estructuras subyacentes muy complejas, pero de muy difícil interpretación. Con la aparición de los métodos SVM y boosting, ANN perdió popularidad, pero en los últimos años ha vuelto a ganarla, también gracias al aumento de las capacidades de computación. El diseño y el entrenamiento de una ANN suele requerir de más tiempo y experimentación que otros algoritmos de AE/ML. Además el gran número de hiperparámetros lo convierte en un problema de optimización complicado. En este capítulo se va a hacer una breve introducción a estos métodos, para poder emplearlos con solvencia en la práctica sería muy recomendable profundizar más en esta metodología (por ejemplo se podría consultar Chollet y Allaire, 2018, para un tratamiento más detallado).

En los métodos de aprendizaje supervisado se realizan una o varias transformaciones del espacio de las variables predictoras buscando una representación *óptima* de los datos, para así poder conseguir una buena predicción. Los modelos que realizan una o dos transformaciones reciben el nombre de modelos superficiales (*shallow models*). Por el contrario, cuando se realizan muchas transformaciones se habla de aprendizaje profundo (*deep learning*). No nos debemos dejar engañar por la publicidad: que un aprendizaje sea profundo no significa que sea mejor que el superficial. Aunque es verdad que ahora mismo la metodología que está de moda son las redes neuronales profundas (*deep neural networks*), hay que ser muy consciente de que dependiendo del contexto será más conveniente un tipo de modelos u otro. Se trata de una metodología adecuada para problemas muy complejos y no tanto para problemas con pocas observaciones o pocos predictores. Hay que tener en cuenta que no existe ninguna metodología que sea *transversalmente* la mejor (lo que se conoce como el teorema *no free lunch*, Wolpert y Macready, 1997).

Una red neuronal básica, como la representada en la Figura 8.1, va a realizar dos transformaciones de los datos, y por tanto es un modelo con tres capas: una capa de entrada (*input layer*) consistente en las variables originales $\mathbf{X} = (X_1, X_2, \dots, X_p)$, otra capa oculta (*hidden layer*) con M nodos, y la capa de salida (*output layer*) con la predicción (o predicciones) final $m(\mathbf{X})$.

Para que las redes neuronales tengan un rendimiento aceptable se requiere disponer de tamaños muestrales grandes, debido a que son modelos hiperparametrizados (y por tanto de difícil interpretación). El ajuste de estos modelos puede requerir de mucho de tiempo de computación, incluso si están implementados de forma muy eficiente (computación en paralelo con GPUs), y solo desde fechas recientes es viable utilizarlos con un número elevado de capas (*deep neural networks*). También son muy sensibles a la escala de los predictores, por lo que requerirían de un preprocesado en el que se homogeneicen (también podrían tener problemas de colinealidad).

Una de las fortalezas de las redes neuronales es que sus modelos son muy robustos frente a predictores irrelevantes. Esto la convierte en una metodología muy interesante cuando se dispone de datos de dimensión muy alta. Otros métodos requieren un preprocesado muy costoso, pero las redes neuronales lo realizan de forma automática en las capas intermedias, que de forma sucesiva se van centrado en

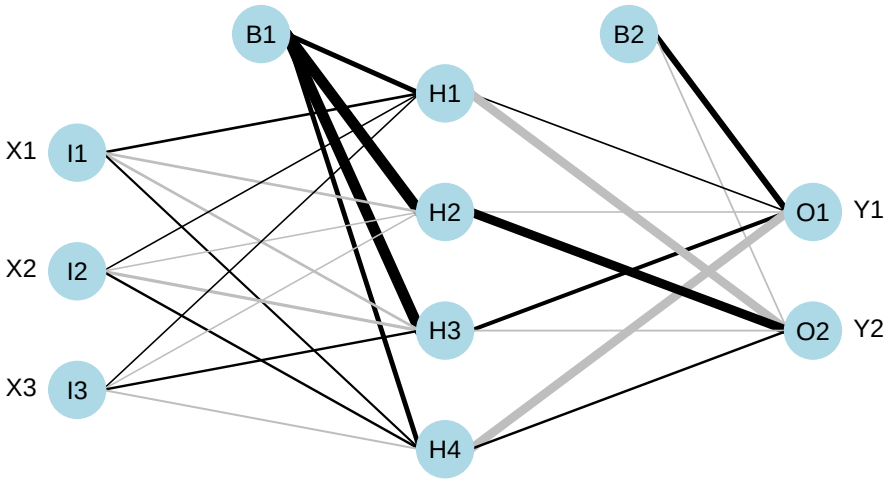


Figura 8.1: Diagrama de una red neuronal.

aspectos relevantes de los datos. Y una de sus debilidades es que conforme aumentan las capas se hace más difícil la interpretación del modelo, hasta convertirse en una auténtica caja negra.

Hay distintas formas de construir redes neuronales. La básica recibe el nombre de *feedforward* (o también *multilayer perceptron*). Otras formas, con sus campos de aplicación principales, son:

- *Convolutional neural networks* para reconocimiento de imagen y vídeo.
- *Recurrent neural networks* para reconocimiento de voz.
- *Long short-term memory neural networks* para traducción automática.

8.1 Single-hidden-layer feedforward network

La red neuronal más simple es la *single-hidden-layer feedforward network*, también conocida como *single layer perceptron*. Se trata de una red *feedforward* con una única capa oculta que consta de M variables ocultas h_m (los nodos que conforman la capa, también llamados unidades ocultas). Cada variable h_m es una combinación lineal de las variables predictoras, con parámetros ω_{jm} (los parámetros ω_{0m} reciben el nombre de parámetros *sesgo*)

$$\omega_{0m} + \omega_{1m}x_1 + \omega_{2m}x_2 + ... + \omega_{pm}x_p$$

transformada por una función no lineal, denominada *función de activación*, típicamente la función logística (denominada función sigmoidal, *sigmoid function*, en este contexto)

$$\phi(u) = \frac{1}{1+e^{-u}} = \frac{e^u}{1+e^u}$$

(la idea es que cada neurona "aprende" un resultado binario). De este modo tenemos que

$$h_m(\mathbf{x}) = \phi\left(\omega_{0m} + \omega_{1m}x_1 + \omega_{2m}x_2 + ... + \omega_{pm}x_p\right)$$

El modelo final es una combinación lineal de las variables ocultas

$$m(\mathbf{x}) = \gamma_0 + \gamma_1 h_1 + \gamma_2 h_2 + ... + \gamma_M h_M$$

aunque también se puede considerar una función de activación en el nodo final para adaptar la predicción a distintos tipos de respuestas (en regresión sería normalmente la identidad) y distintas funciones de activación en los nodos intermedios (ver por ejemplo Wikipedia: Activation function para un listado de distintas funciones de activación).

Por tanto, el modelo m es un modelo de regresión no lineal en dos etapas con $M(p+1)+M+1$ parámetros (también llamados *pesos*). Por ejemplo, con 200 variables predictoras y 10 variables ocultas,

hay nada menos que 2021 parámetros. Como podemos comprobar, incluso con el modelo más sencillo y una cantidad moderada de variables predictoras y ocultas, el número de parámetros a estimar es muy grande. Por eso decimos que estos modelos están hiperparametrizados.

La estimación de los parámetros (el aprendizaje) se realiza minimizando una función de pérdidas, típicamente RSS. La solución exacta de este problema de optimización suele ser imposible en la práctica (es un problema no convexo), por lo que se resuelve mediante un algoritmo heurístico de descenso de gradientes (que utiliza las derivadas de las funciones de activación), llamado en este contexto *backpropagation* (Werbos, 1974), que va a converger a un óptimo local, pero difícilmente al óptimo global. Por este motivo, el modelo resultante va a ser muy sensible a la solución inicial, que generalmente se selecciona de forma aleatoria con valores próximos a cero (si se empezase dando a los parámetros valores nulos, el algoritmo no se movería). El algoritmo va cogiendo los datos de entrenamiento por lotes (de 32, 64…) llamados *batch*, y recibe el nombre de *epoch* cada vez que el algoritmo completa el procesado de todos los datos; por tanto, el número de *epochs* es el número de veces que el algoritmo procesa la totalidad de los datos.

Una forma de mitigar la inestabilidad de la estimación del modelo es generando muchos modelos (que se consiguen con soluciones iniciales diferentes) y promediando las predicciones; una alternativa es utilizar *bagging*. El algoritmo depende de un parámetro en cada iteración, que representa el ratio de aprendizaje (*learning rate*). Por razones matemáticas, se selecciona una sucesión que converja a cero.

Otro problema inherente a la heurística de tipo gradiente es que se ve afectada negativamente por la correlación entre las variables predictoras. Cuando hay correlaciones muy altas, es usual preprocesar los datos, o bien eliminando variables predictoras o bien utilizando PCA.

Naturalmente, al ser las redes neuronales unos modelos con tantos parámetros tienen una gran tendencia al sobreajuste. Una forma de mitigar este problema es implementar la misma idea que se utiliza en la regresión *ridge* de penalizar los parámetros y que en este contexto recibe el nombre de reducción de los pesos (*weight decay*)

$$\min_{\omega,\gamma} \text{RSS} + \lambda \left(\sum_{m=1}^{m} \sum_{j=0}^{p} \omega_{jm}^2 + \sum_{m=0}^{M} \gamma_m^2 \right)$$

En esta modelización del problema, hay dos hiperparámetros cuyos valores deben ser seleccionados: el parámetro regularizador λ (con frecuencia un número entre 0 y 0.1) y el número de nodos M. Es frecuente seleccionar M a mano (un valor alto, entre 5 y 100) y λ por validación cruzada, confiando en que el proceso de regularización forzará a que muchos pesos (parámetros) sean próximos a cero. Además, al depender la penalización de una suma de pesos es imprescindible que sean comparables, es decir, hay que reescalar las variables predictoras antes de empezar a construir el modelo.

La extensión natural de este modelo es utilizar más de una capa de nodos (variables ocultas). En cada capa, los nodos están *conectados* con los nodos de la capa precedente.

Observemos que el modelo *single-hidden-layer feedforward network* tiene la misma forma que el de la *projection pursuit regression* (Sección 7.5.1), sin más que considerar $\alpha_m = \omega_m/\|\omega_m\|$, con $\omega_m = (\omega_{1m}, \omega_{2m}, \ldots, \omega_{pm})$, y

$$g_m(\alpha_{1m}x_1 + \alpha_{2m}x_2 + \ldots + \alpha_{pm}x_p) = \gamma_m \phi(\omega_{0m} + \omega_{1m}x_1 + \omega_{2m}x_2 + \ldots + \omega_{pm}x_p)$$

Sin embargo, hay que destacar una diferencia muy importante: en una red neuronal, el analista fija la función ϕ (lo más habitual es utilizar la función logística), mientras que las funciones *ridge* g_m se consideran como funciones no paramétricas desconocidas que hay que estimar.

8.2 Clasificación con ANN

En un problema de clasificación con dos categorías, si se emplea una variable binaria para codificar la respuesta, bastará con considerar una función logística como función de activación en el nodo final (de esta forma se estará estimando la probabilidad de éxito). En el caso general, en lugar de construir un

único modelo $m(\mathbf{x})$, se construyen tantos como categorías, aunque habrá que seleccionar una función de activación adecuada en los nodos finales.

Por ejemplo, en el caso de una *single-hidden-layer feedforward network*, para cada categoría i, se construye el modelo T_i como ya se explicó antes

$$T_i(\mathbf{x}) = \gamma_{0i} + \gamma_{1i}h_1 + \gamma_{2i}h_2 + ... + \gamma_{Mi}h_M$$

y a continuación se transforman los resultados de los k modelos para obtener estimaciones válidas de las probabilidades

$$m_i(\mathbf{x}) = \tilde{\phi}_i(T_1(\mathbf{x}), T_2(\mathbf{x}), ..., T_k(\mathbf{x}))$$

donde $\tilde{\phi}_i$ es la función *softmax*

$$\tilde{\phi}_i(u_1, u_2, ..., u_k) = \frac{e^{u_i}}{\sum_{j=1}^{k} e^{u_j}}$$

Como criterio de error se suele utilizar la *entropía* aunque se podrían considerar otros. Desde este punto de vista la regresión logística (multinomial) sería un caso particular.

8.3 Implementación en R

Hay numerosos paquetes que implementan métodos de este tipo, aunque por simplicidad consideraremos el paquete `nnet` que implementa redes neuronales *feed fordward* con una única capa oculta y está incluido en el paquete base de R. Para el caso de redes más complejas se puede utilizar por ejemplo el paquete `neuralnet`, pero en el caso de grandes volúmenes de datos o aprendizaje profundo la recomendación sería emplear paquetes computacionalmente más eficientes (con computación en paralelo empleando CPUs o GPUs) como `keras`, `h2o` o `sparlyr`, entre otros.

La función principal `nnet()` se suele emplear con los siguientes argumentos:

```
nnet(formula, data, size, Wts, linout = FALSE, skip = FALSE,
     rang = 0.7, decay = 0, maxit = 100, ...)
```

- `formula` y `data` (opcional): permiten especificar la respuesta y las variables predictoras de la forma habitual (e.g. `respuesta ~ .`; también implementa una interfaz con matrices `x` e `y`). Admite respuestas multidimensionales (ajustará un modelo para cada componente) y categóricas (las convierte en multivariantes si tienen más de dos categorías y emplea *softmax* en los nodos finales). Teniendo en cuenta que por defecto los pesos iniciales se asignan al azar (`Wts <- runif(nwts, -rang, rang)`) la recomendación sería reescalar los predictores en el intervalo $[0, 1]$, sobre todo si se emplea regularización (`decay > 0`).
- `size`: número de nodos en la capa oculta.
- `linout`: permite seleccionar la identidad como función de activación en los nodos finales; por defecto `FALSE` y empleará la función logística o *softmax* en el caso de factores con múltiples niveles (si se emplea la interfaz de fórmula, con matrices habrá que establecer `softmax = TRUE`).
- `skip`: permite añadir pesos adicionales entre la capa de entrada y la de salida (saltándose la capa oculta); por defecto `FALSE`.
- `decay`: parámetro λ de regularización de los pesos (*weight decay*); por defecto 0. Para emplear este parámetro los predictores deberían estar en la misma escala.
- `maxit`: número máximo de iteraciones; por defecto 100.

Como ejemplo consideraremos el conjunto de datos `earth::Ozone1` empleado en el capítulo anterior:

```
data(ozone1, package = "earth")
df <- ozone1
set.seed(1)
nobs <- nrow(df)
itrain <- sample(nobs, 0.8 * nobs)
```

```
train <- df[itrain, ]
test <- df[-itrain, ]
```

En este caso emplearemos el método `"nnet"` de `caret` para preprocesar los datos y seleccionar el número de nodos en la capa oculta y el parámetro de regularización. Como emplea las opciones por defecto de `nnet()` (diseñadas para clasificación), estableceremos `linout = TRUE`[1] y aumentaremos el número de iteraciones (aunque seguramente sigue siendo demasiado pequeño).

```
library(caret)
# Buscar "Neural Network": 10 métodos
# getModelInfo("nnet")
modelLookup("nnet")
```

```
##   model parameter         label forReg forClass probModel
## 1  nnet      size #Hidden Units   TRUE     TRUE      TRUE
## 2  nnet     decay  Weight Decay   TRUE     TRUE      TRUE
```

```
tuneGrid <- expand.grid(size = 2*1:5, decay = c(0, 0.001, 0.01))
set.seed(1)
caret.nnet <- train(O3 ~ ., data = train, method = "nnet",
             preProc = c("range"), # Reescalado en [0,1]
             tuneGrid = tuneGrid,
             trControl = trainControl(method = "cv", number = 10),
             linout = TRUE, maxit = 200, trace = FALSE)
ggplot(caret.nnet, highlight = TRUE)
```

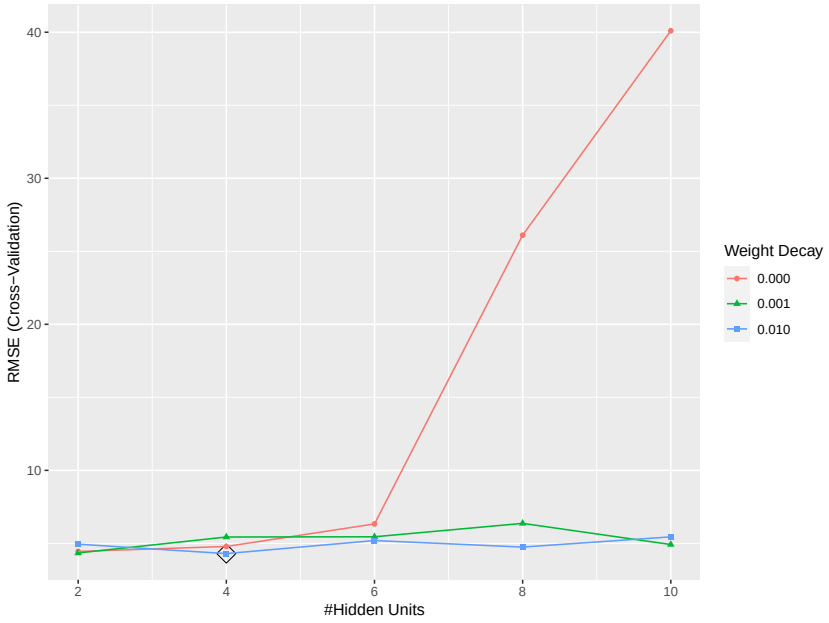


Figura 8.2: Selección de los hiperparámetros asociados a una red neuronal (el número de nodos y el parámetro de regularización) mediante un criterio de error RMSE calculado por validación cruzada.

Analizamos el modelo resultante:

```
summary(caret.nnet$finalModel)
```

```
## a 9-4-1 network with 45 weights
## options were - linear output units  decay=0.01
##  b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1 i8->h1 i9->h1
```

[1] La alternativa sería transformar la respuesta a rango 1.

```
##  -8.66   3.74  -5.50 -18.11 -12.83   6.49  14.39  -4.53  14.48  -1.96
##  b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2 i7->h2 i8->h2 i9->h2
##  -2.98   1.78   0.00   1.58   1.96  -0.60   0.63   2.46   2.36 -19.69
##  b->h3 i1->h3 i2->h3 i3->h3 i4->h3 i5->h3 i6->h3 i7->h3 i8->h3 i9->h3
##  25.23 -50.14   9.74  -3.66  -5.61   4.21 -11.17  39.34 -20.18   0.37
##  b->h4 i1->h4 i2->h4 i3->h4 i4->h4 i5->h4 i6->h4 i7->h4 i8->h4 i9->h4
##  -3.90   4.94  -1.08   1.50   1.52  -0.54   0.14  -1.27   0.98  -1.54
##   b->o  h1->o  h2->o  h3->o  h4->o
##  -5.32   4.19 -14.03   7.50  38.75
```

y lo representamos gráficamente, empleando el paquete `NeuralNetTools` (ver Figura 8.3):

```
library(NeuralNetTools)
old.par <- par(mar = c(bottom = 1, left = 2, top = 2, right = 3), xpd = NA)
plotnet(caret.nnet$finalModel)
```

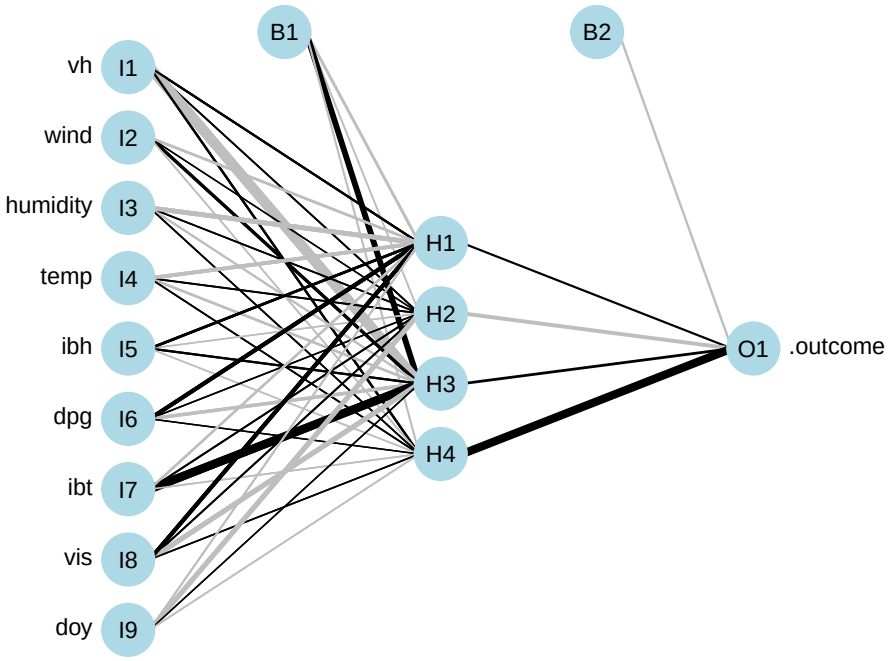


Figura 8.3: Representación de la red neuronal ajustada (generada con el paquete `NeuralNetTools`).

```
par(old.par)
```

Por último evaluamos las predicciones en la muestra de test:

```
pred <- predict(caret.nnet, newdata = test)
obs <- test$O3
accuracy(pred, obs)
```

```
##         me       rmse        mae        mpe       mape  r.squared
##  0.3321276  3.0242169  2.4466958 -7.4095987 32.8000107  0.8586515
```

y las representamos gráficamente (ver Figura 8.4):

```
plot(pred, obs, xlab = "Predicción", ylab = "Observado")
abline(a = 0, b = 1)
abline(lm(obs ~ pred), lty = 2)
```

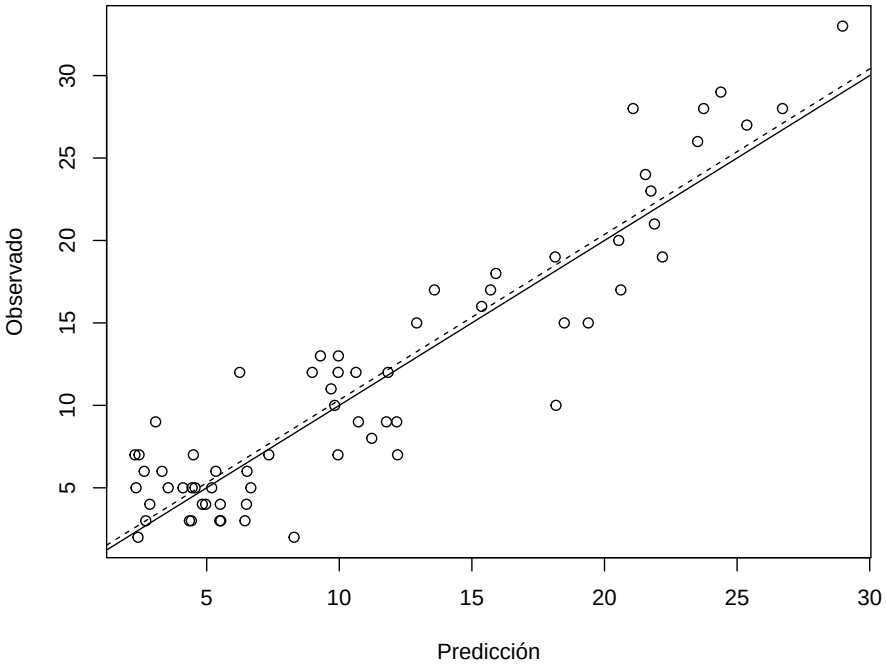


Figura 8.4: Observaciones frente a predicciones (en la muestra de test) con la red neuronal ajustada.

Referencias

Agor, J., y Özaltın, O. Y. (2019). Feature selection for classification models via bilevel optimization. *Computers and Operations Research*, *106*, 156-168. https://doi.org/10.1016/j.cor.2018.05.005

Becker, M., Binder, M., Bischl, B., Lang, M., Pfisterer, F., Reich, N. G., Richter, J., Schratz, P., Sonabend, R., y Pulatov, D. (2021). *mlr3 book*. https://mlr3book.mlr-org.com

Bellman, R. (1961). *Adaptive Control Processes: a guided tour*. Princeton University Press.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123-140. https://doi.org/10.1007/bf00058655

Breiman, L. (2001a). Random forests. *Machine Learning*, *45*(1), 5-32.

Breiman, L. (2001b). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, *16*(3), 199-231. https://doi.org/10.1214/ss/1009213726

Breiman, L., Friedman, J. H., Stone, C. J., y Olshen, R. A. (1984). *Classification and Regression Trees*. Taylor; Francis.

Chen, T., y Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785-794. https://doi.org/10.1145/2939672.2939785

Chollet, F., y Allaire, J. J. (2018). *Deep Learning with R*. Manning Publications.

Cortes, C., y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273-297. https://doi.org/10.1007/bf00994018

Cortez, P., Cerdeira, A., Almeida, F., Matos, T., y Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, *47*(4), 547-553. https://doi.org/10.1016/j.dss.2009.05.016

Craven, P., y Wahba, G. (1978). Smoothing noisy data with spline functions. *Numerische Mathematik*, *31*(4), 377-403. https://doi.org/10.1007/bf01404567

Culp, M., Johnson, K., y Michailidis, G. (2006). ada: An R Package for Stochastic Boosting. *Journal of Statistical Software*, *17*(2), 1-27. https://doi.org/10.18637/jss.v017.i02

De Boor, C., y De Boor, C. (1978). *A practical guide to splines* (Vol. 27). springer-verlag New York. https://doi.org/10.1007/978-1-4612-6333-3

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, *40*(2), 139-157.

Drucker, H., Burges, C. J., Kaufman, L., Smola, A., y Vapnik, V. (1997). Support Vector Regression Machines. *Advances in Neural Information Processing Systems*, *9*, 155-161.

Dunn, P. K., y Smyth, G. K. (2018). *Generalized linear models with examples in R* (Vol. 53). Springer.

Dunson, D. B. (2018). Statistics in the big data era: Failures of the machine. *Statistics and Probability Letters*, *136*, 4-9. https://doi.org/10.1016/j.spl.2018.02.028

Efron, B., Hastie, T., Johnstone, I., y Tibshirani, R. (2004). Least angle regression. *The Annals of Statistics*, *32*(2), 407-499. https://doi.org/10.1214/009053604000000067

Eilers, P. H., y Marx, B. D. (1996). Flexible smoothing with B-splines and penalties. *Statistical Science*, *11*(2), 89-121. https://doi.org/10.1214/ss/1038425655

Fan, J., y Gijbels, I. (1996). *Local Polynomial Modelling and Its Applications*. Chapman; Hall.

Faraway, J. J. (2016). *Linear Models with R* (Second). CRC Press.

Fernandez-Casal, R., Oviedo-de la Fuente, M., y Costa-Bouzas, J. (2024). *mpae: Metodos Predictivos de Aprendizaje Estadistico (Statistical Learning Predictive Methods)*. https://github.com/rubenfcasal/mpae

Fernández-Casal, R., Cao, R., y Costa, J. (2023). *Técnicas de simulación y remuestreo*. https://rubenfcasal.github.io/simbook

Fernández-Casal, R., Roca-Pardiñas, J., y Costa, J. (2019). *Introducción al Análisis de Datos con R*. https://rubenfcasal.github.io/intror

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics*, *7*(2), 179-188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

Freund, Y., y Schapire, R. E. (1996). Schapire R: Experiments with a new boosting algorithm. *in: Thirteenth International Conference on ML*.

Friedman, J. H. (1989). Regularized discriminant analysis. *Journal of the American statistical association*, *84*(405), 165-175. https://doi.org/10.1080/01621459.1989.10478752

Friedman, J. H. (1991). Multivariate Adaptive Regression Splines. *The Annals of Statistics*, *19*(1), 1-67. https://doi.org/10.1214/aos/1176347963

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *The Annals of Statistics*, 1189-1232. https://doi.org/10.1214/aos/1013203451

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & data analysis*, *38*(4), 367-378. https://doi.org/10.1016/S0167-9473(01)00065-2

Friedman, J. H., y Popescu, B. E. (2008). Predictive learning via rule ensembles. *The Annals of Applied Statistics*, *2*(3), 916-954. https://doi.org/10.1214/07-aoas148

Friedman, J. H., y Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American statistical Association*, *76*(376), 817-823. https://doi.org/10.1080/01621459.1981.10477729

Friedman, J. H., y Tukey, J. W. (1974). A projection pursuit algorithm for exploratory data analysis. *IEEE Transactions on computers*, *100*(9), 881-890. https://doi.org/10.1109/t-c.1974.224051

Friedman, J., Hastie, T., y Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The Annals of Statistics*, *28*(2), 337-407. https://doi.org/10.1214/aos/1016218223

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. https://doi.org/10.1080/10618600.2014.907095

Greenwell, B. M. (2017). pdp: An R Package for Constructing Partial Dependence Plots. *The R Journal*, *9*(1), 421-436. https://doi.org/10.32614/RJ-2017-016

Hair, J. F., Anderson, R. E., Tatham, R. L., y Black, W. (1998). *Multivariate Data Analysis*. Prentice Hall.

Hastie, T. J., y Pregibon, D. (2017). *Generalized linear models* (pp. 195-247). Routledge.

Hastie, T., Rosset, S., Tibshirani, R., y Zhu, J. (2004). The entire regularization path for the support vector machine. *Journal of Machine Learning Research*, *5*(Oct), 1391-1415.

Hastie, T., y Tibshirani, R. (1990). *Generalized Additive Models*. Chapman; Hall. https://doi.org/10.1201/9780203753781

Hastie, T., y Tibshirani, R. (1996). Discriminant Analysis by Gaussian Mixtures. *Journal of the Royal Statistical Society. Series B (Methodological)*, *58*(1), 155-176. https://doi.org/10.1111/j.2517-6161.1996.tb02073.x

Hothorn, T., Hornik, K., y Zeileis, A. (2006). Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, *15*(3), 651-674. https://doi.org/10.1198/106186006x133933

Ichimura, H. (1993). Semiparametric least squares (SLS) and weighted SLS estimation of single-index models. *Journal of Econometrics*, *58*(1), 71-120. https://doi.org/10.1016/0304-4076(93)90114-K

James, G., Witten, D., Hastie, T., y Tibshirani, R. (2021). *An Introduction to Statistical Learning: With Applications in R, Second Edition*. Springer. https://www.statlearning.com

Karatzoglou, A., Smola, A., Hornik, K., y Zeileis, A. (2004). kernlab - An S4 Package for Kernel Methods in R. *Journal of Statistical Software, Articles*, *11*(9), 1-20. https://doi.org/10.18637/jss.v011.i09

Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, *29*(2), 119-127. https://doi.org/10.2307/2986296

Kearns, M., y Valiant, L. (1994). Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the ACM*, *41*(1), 67-95. https://doi.org/10.1145/174644.174647

Kruskal, J. B. (1969). Toward a practical method which helps uncover the structure of a set of multivariate observations by finding the linear transformation which optimizes a new «index of condensation». *Statistical Computation*, 427-440. https://doi.org/10.1016/b978-0-12-498150-

8.50024-0
Kuhn, M. (2008). Building Predictive Models in R Using the caret Package. *Journal of Statistical Software*, *28*(5), 1-26. https://doi.org/10.18637/jss.v028.i05
Kuhn, M. (2023). *caret: Classification and Regression Training*. https://github.com/topepo/caret/
Kuhn, M., y Johnson, K. (2013). *Applied predictive modeling* (Vol. 26). Springer. https://doi.org/10.1007/978-1-4614-6849-3
Kuhn, M., y Silge, J. (2022). *Tidy Modeling with R*. O'Reilly Media. https://www.tmwr.org
Kuhn, M., y Wickham, H. (2023). *tidymodels: Easily Install and Load the Tidymodels Packages*. https://tidymodels.tidymodels.org
Kvålseth, T. O. (1985). Cautionary note about R 2. *The American Statistician*, *39*(4), 279-285. https://doi.org/10.1080/00031305.1985.10479448
Lauro, C. (1996). Computational statistics or statistical computing, is that the question? *Computational Statistics & Data Analysis*, *23*(1), 191-193. https://doi.org/10.1016/0167-9473(96)88920-1
Liaw, A., y Wiener, M. (2002). Classification and Regression by randomForest. *R News*, *2*(3), 18-22. https://www.r-project.org/doc/Rnews/Rnews_2002-3.pdf
Loh, W.-Y. (2002). Regression tress with unbiased variable selection and interaction detection. *Statistica Sinica*, 361-386.
Massy, W. F. (1965). Principal components regression in exploratory statistical research. *Journal of the American Statistical Association*, *60*(309), 234-256. https://doi.org/10.1080/01621459.1965.10480787
McCullagh, P. (2019). *Generalized linear models*. Routledge.
McCulloch, W. S., y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133. https://doi.org/10.1007/bf02459570
Mevik, B.-H., y Wehrens, R. (2007). The pls Package: Principal Component and Partial Least Squares Regression in R. *Journal of Statistical Software, Articles*, *18*(2), 1-23. https://doi.org/10.18637/jss.v018.i02
Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., y Leisch, F. (2020). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. https://CRAN.R-project.org/package=e1071
Molnar, C. (2020). *Interpretable Machine Learning*. Lulu.com. https://christophm.github.io/interpretable-ml-book
Quinlan, J. R. et al. (1992). Learning with continuous classes. *5th Australian joint conference on artificial intelligence*, *92*, 343-348.
Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Elsevier Science.
Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379-423. https://doi.org/10.2307/410457
Spinoza, B. (1677). *Ethics*.
Strumbelj, E., y Kononenko, I. (2010). An efficient explanation of individual classifications using game theory. *The Journal of Machine Learning Research*, *11*, 1-18.
Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, *58*(1), 267-288. https://doi.org/10.1111/j.2517-6161.1996.tb02080.x
Valiant, L. G. (1984). A Theory of the Learnable. *Communications of the ACM*, *27*(11), 1134-1142. https://doi.org/10.1145/800057.808710
Vapnik, V. (2000). *Statistical Learning Theory*. Willey.
Vapnik, V. (2013). *The Nature of Statistical Learning Theory*. Springer.
Venables, W. N., y Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer New York. https://doi.org/10.1007/978-0-387-21706-2
Vinayak, R. K., y Gilad-Bachrach, R. (2015). Dart: Dropouts meet multiple additive regression trees. *Artificial Intelligence and Statistics*, 489-497.
Welch, B. L. (1939). Note on Discriminant Functions. *Biometrika*, *31*(1/2), 218-220. https://doi.org/10.2307/2334985
Werbos, P. (1974). New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*.
Williams, G. (2011). *Data mining with Rattle and R: The art of excavating data for knowledge discovery*. Springer Science & Business Media.

Williams, G. (2022). *rattle: Graphical User Interface for Data Science in R*. https://rattle.togaware.com/

Wold, S., Martens, H., y Wold, H. (1983). The multivariate calibration problem in chemistry solved by the PLS method. En *Matrix pencils* (pp. 286-293). Springer. https://doi.org/10.1007/bfb0062108

Wolpert, D. H., y Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67-82. https://doi.org/10.1109/4235.585893

Wood, S. N. (2017). *Generalized Additive Models: An Introduction with R, Second Edition*. CRC Press.

Zou, H., y Hastie, T. (2005). Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society, Series B (Statistical Methodology)*, *67*(2), 301-320. https://doi.org/10.1111/j.1467-9868.2005.00503.x