Análisis de datos y machine learning con R (caret)

Versión 0.105 (pre-release)

Este tutorial está en proceso de mejora y actualización. Comprueba a menudo la versión del aula virtual.

Si encuentras fallos se agradece que los notifiques por el aula virtual.

Aprendizaje Computacional
Curso -

Contents

1 Introducción

Las primeras prácticas las vamos a dedicar a aprender cómo familiarizarnos con conjuntos de datos. La asignatura trata de cómo analizar de manera inteligente (con técnicas distintas a las que encontramos en la estadística convencional) conjuntos de datos que responden a muestras (i.e. observaciones) de un mismo fenómeno. Primero veremos cómo con simples herramientas de visualización de datos y proceso estadístico, podemos obtener, de manera sencilla, informes basados en estadística descriptiva, que nos ayudarán a entender mejor el fenómeno que estamos estudiando.

Este tipo de informe inical no es el objetivo último del análisis inteligente de datos. Debemos verlo como el primer paso. Como una manera de tener más información para entender el problema de manera inicial y poder atacarlo con las mayores garantías posibles.

Como nota previa indicaros como se presentarán las secuencias de comandos:

- Cuando aparezcan precedidas por `'>'` lo que se muestra es el comando seguido de la salida que produce dicho comando en la consola de R. Para introducir el comando en la consola debes ignorar dicho símbolo `'>'`.
- Cuando aparezcan en una caja, es una secuencia de comandos a introducir en la consola, tal cual (lo que aparece tras un `#` son comentarios en R, y se pueden ignorar).
- También pueden aparecer en una caja con el membrete `'Definición de Comando'` que, como el nombre indica, es tan solo la definición de dicho comando cuyos detalles se pueden obtener con `help("`*Comando*`")`.

Igualmente vamos a utilizar una librería muy útil en Machine Learning, la librería `caret` que proviene de **Cl**assification and **RE**gression **T**raining. Aunque en los laboratorios debería estar instalada sería aconsejable ejecutar el siguiente comando (podría tardar bastante en ejecutarse, no lo hagas en el laboratorio de prácticas), que instala la librería y todos los componentes de los que depende y también los que se sugiere que se incluyan, así como después cargarla en el intérprete:

```
if(!require("caret")) {
  install.packages("caret", dependencies = c("Depends", "Suggests"))
  require(caret)
}
```

1.1 Gráficos en R

Como se ha mencionado se va a trabajar con gráficos en R, aunque no vamos a entrar en detenimiento en ellos. Se aconseja seguir un par de tutoriales para familiarizarse con el tipo de gráficos que usaremos en esta práctica. Los podréis encontrar en:

`http://www.isid.ac.in/~deepayan/R-tutorials/labs/03_rgraphics_lab.pdf`
`http://www.isid.ac.in/~deepayan/R-tutorials/labs/04_lattice_lab.pdf`

El primero trata de los gráficos tradicionales de R usando `plot()` y el segundo trata los gráficos usando parrillas con la librería `lattice`. La mayor parte de ejemplos de este doble tutorial usarán dichos gráficos. Hay un tercer sistema más moderno `ggplot2` del que es interesante tener las "chuletas" disponibles en:

`https://raw.githubusercontent.com/rstudio/cheatsheets/master/data-visualization-2.1.pdf`
`https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf`

Se proporcionarán algunos ejemplos en `ggplot2` pero no es el objetivo del tutorial y hay numerosa documentación en la red para estos sistemas.

El sistema tradicional de gráficos en R funciona de forma *acumulativa*, partiendo de funciones de dibujo de *alto nivel* a las que ir luego añadiendo elementos a los gráficos mediante funciones de *bajo nivel*.

El sistema en `grid` de `lattice` nos permite visualizar de forma fácil y potente datos multivariable. En particular, mediante el uso de `formula` y grupos permite visualizar hasta cuatro dimensiones (con sus diferentes formas de representación, como histogramas, gráficos de densidades, scatter plots, etc.). El tutorial arriba sugerido explica con facilidad esas capacidades.

Otras librerias ofrecen sus propias funciones de representación gráfica para los elementos con los que trabajan. Muchos de esos gráficos específicos en realidad hacen uso de estas librerias y son `wrappers` o elaboraciones de dichas funciones.

2 El problema IRIS

La flor que aparece a la izquierda de la figura 1 es una Iris,del tipo siberiano[1]. Supongamos que queremos estudiar esta flor, a partir de una plantación de tres variedades de la misma, Setosa, Versicolor y Virgínica. Para ello se nos ha facilitado una serie de tuplas de datos de cada flor disponibles en un invernadero que usaremos como datos fuente. Si denotamos con

$$D = \{(\bar{x}, y) | \bar{x} \in R^4\},$$

en donde las dos primeras características de $\bar{x}$ se refieren a la longitud y anchura del pétalo, y las otras dos a la longitud y anchura del sépalo, junto con que y se refiere al tipo de flor, esa es toda la información que nos dan.

El problema genérico se trata de responder a la pregunta de si es posible distinguir un tipo de Iris de las otras dos, simplemente mirando a las dimensiones de pétalo y sépalo correspondientes. O formulado de otra forma, si cada una de estas tres variedades se diferencia de manera significativa del resto por sus pétalos y sépalos.

3 Algunas estadísticas descriptivas sobre el cjto. de datos

Lo primero que debemos tener claro es que estamos ante un problema de clasificación. Es decir, debemos generar una hipótesis h, a partir de los datos D, tal que esta nos permita clasificar nuevas observaciones $\bar{x}$ como pertenecientes a uno de los tres tipos de Iris conocidas. La denominamos hipótesis para hacer énfasis en el hecho de que no nos referimos a un modelo concreto, ya sea de red neuronal o árbol de decisión, por poner dos ejemplos de sobra conocidos. De momento no nos interesa saber qué tipo concreto tendrá el modelo.

Al ser un problema de clasificación, lo primero que deberíamos hacer es preguntarnos por la proporción de ejemplares de cada clase que tenemos. ¿Cómo hacer esto de manera sencilla? Aquí es donde entra R. Arrancaremos RStudio y empezaremos a trabajar.

[1] http://home.att.net/~nnthom266/1999/iris.htm

Figure 1: Detalle de una Iris siberiana (izquierda) y pétalo y sépalo de una flor (derecha).

El conjunto de datos iris es uno de los que R tiene ya preparados para nosotros. Se puede obtener mediante el comando `data`:

Definición de Comando

```
data(..., list = character(), package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"), envir = .GlobalEnv)
```

Al escribir "`data(`" RStudio ya proporciona una lista con muchas bases de datos disponibles, entre ellas irirs. Una vez ejecutado `data("iris")` podemos comprobar que tenemos un data frame de nombre `iris` con los datos. Podemos ver los primeros elementos con el comando `head()`.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

No obstante es interesante el practicar la carga de la base de datos desde un fichero. Eliminamos la variable `iris` para no confundirnos (mediante el comando `rm(iris)`) y utilizaremos el fichero "iris.data" que descargaremos de un repositorio de datos de la red. Asumimos ahora que el conjunto de datos (el fichero de texto `iris.data`) está accesible y que las tuplas del mismo tienen la forma[2]

[2]Mirar en `https://archive.ics.uci.edu/ml/index.php` donde podreis encontrar este ejemplo y otros típicos de problemas de

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-versicolor
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-virginica
```

La primera tupla corresponde a los valores 5.1, 3.5, 1.4 y 0.2 para $\bar{x}$, e `Iris-setosa` para la etiqueta de clase y. El primer paso es intentar que R lea ese fichero y lo haga disponible en memoria principal en forma de variable para que lo podamos manipular desde ahí. Si hacemos (suponiendo que el conjunto de datos esté en el directorio desde el que hemos invocado a R),

```
# Cargar un fichero de datos en un data.frame
iris <- read.table(
"https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data")
```

ahora, en la variable `iris` tenemos almacenado el cjto. de datos. Para ver el contenido de la variable hacemos

```
# Mostrar los primeros elementos de un data.frame
head(iris)
```

Obtendremos esto:

```
                          V1
1 5.1,3.5,1.4,0.2,Iris-setosa
2 4.9,3.0,1.4,0.2,Iris-setosa
3 4.7,3.2,1.3,0.2,Iris-setosa
4 4.6,3.1,1.5,0.2,Iris-setosa
5 5.0,3.6,1.4,0.2,Iris-setosa
6 5.4,3.9,1.7,0.4,Iris-setosa
```

que nos muestra todo el conjunto de datos tal y como está contenido en la variable `iris`. Obsérvese que en la primera línea aparece una etiqueta `V1`, y como primera columna una secuencia de números. La etiqueta hace referencia al nombre de la columna almacenada, lo cual quiere decir que no se han separado en características diferentes los cinco valores que debe haber por tupla. Esto ha sucedido porque el separador que asume por defecto la función `read.table` es el espacio. En cambio, en este fichero de datos,, es una coma lo que distingue unos valores de otros. En otras palabras, los datos probablemente se han cargado mal. Es un error bastante común y no siempre tan obvio como en este caso. El otro típico error al cargar es no especificar la etiqueta de los datos desconocidos.

La sospecha de que hemos cargado los datos de forma incorrecta se termina de confirmar si comprobamos las dimensiones de los datos. Lo hacemos con el comando `dim()`, es decir:

```
> dim(iris)
[1] 150   1
```

que nos indica que tenemos 150 ejemplos con ¡un solo atributo! Por lo tanto debemos corregir el error en la carga de datos. Si echamos mano del manual de referencia de R, veremos que en al entrada para `read.table` aparece un modificador `sep`, que aplicamos tal que así

aprendizaje.

```
# Cargar un fichero en un data.frame, indicando el separador y sin encabezamiento
iris <- read.table(
"https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data", header=F,
sep=",")
```

con lo que si, ahora, le pedimos visualizar las primeras cinco filas del data frame, con el siguiente comando:

```
iris[1:5,]
```

obtenemos:

```
    V1  V2  V3  V4             V5
1   5.1 3.5 1.4 0.2     Iris-setosa
2   4.9 3.0 1.4 0.2     Iris-setosa
3   4.7 3.2 1.3 0.2     Iris-setosa
4   4.6 3.1 1.5 0.2     Iris-setosa
5   5.0 3.6 1.4 0.2     Iris-setosa
...
```

con lo que comprobamos que se han separado correctamente los valores en cinco columnas. Para trabajar más cómodamente con el conjunto, vamos a ponerle un nombre a las columnas. Hay varias formas de hacerlo. Se pueden primero leer los datos y luego cambiar los nombres de las columnas modificándo del data.frame (accesibles, y modificables mediante el comando `names()`), pero también se puede hacer al cargar la base de datos. Así que el comando final para leer los datos será:

```
# Cargar un fichero de datos en un data.frame indicando nombres de variables
iris <- read.table(
"https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data", header=F,
sep=",",col.names=c('longitud sepalo','anchura sepalo','longitud petalo',
'anchura petalo', 'clase'))
```

Simplemente le decimos a R que los nombres de las columnas de los datos que va a leer son los que le indicamos. Con `header=F` le decimos que los datos no vienen con encabezamiento. Con este comando, ahora tenemos:

```
> iris[1:5,]
  longitud.sepalo anchura.sepalo longitud.petalo anchura.petalo       clase
1             5.1            3.5             1.4            0.2 Iris-setosa
2             4.9            3.0             1.4            0.2 Iris-setosa
3             4.7            3.2             1.3            0.2 Iris-setosa
4             4.6            3.1             1.5            0.2 Iris-setosa
5             5.0            3.6             1.4            0.2 Iris-setosa
```

Vamos a hacer una última comprobación sencilla de que hemos cargado los datos correctamente comprobando a la vez sus primeros valores y los tipos de las columnas. También nos sirve como un primer estudio de los datos. Usaremos el comando `str()` que nos muestra de forma compacta la estructura de un objeto en R y que incluye el número de observaciones (ejemplos) y el número de variables (atributos) de la base de datos:

```
# Mostrar la estructura del data.frame, número de observaciones y variables,
# clase de cada variable y los primeros valores de cada variable.
str(iris)
```

que nos devolvería:

```
'data.frame': 150 obs. of  5 variables:
 $ longitud.sepalo: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ anchura.sepalo : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ longitud.petalo: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ anchura.petalo : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ clase          : Factor w/ 3 levels "Iris-setosa",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Así podemos comprobar que las primeras 4 columnas son valores numéricos (reales) y que la clase es un factor (con 3 niveles/clases). Es importante que el atributo que sea la salida en problemas de clasificación esté como factor si vamos a utilizar la librería Caret, pues es la forma en que distingue la librería entre un problema de regresión y otro de clasificación.

Por último una forma alternativa de ver las clases sería:

```
# Mostrar la clase de cada atributo
sapply(iris, class)
```

que nos diría:

```
 longitud.sepalo  anchura.sepalo longitud.petalo  anchura.petalo           clase
      "numeric"       "numeric"       "numeric"       "numeric"        "factor"
```

3.1 Análisis monovariable

Vamos a empezar a visualizar y analizar un poco los datos. La visualización normalmente se hace con gráficas monovariable (para entender mejor cada atributo de manera individual) o con gráficas multivariable (para comprender mejor las relaciones entre atributos). Comenzaremos primero con las monovariables.

El comando `summary()` nos permite tener resúmenes estadísticos sobre los datos. El tipo de resumen depende de la clase del dato del que queremos hacer el `summary()`. Si tratásemos de obtener un resumen de una columna tipo factor, como es la de clase, hacemos

```
> summary(iris$clase)
    Iris-setosa Iris-versicolor  Iris-virginica
             50              50              50
```

y vemos que nos cuenta el número de ocurrencias de cada nivel (clase). Para verlo en forma de porcentaje del total podríamos hacer:

```
porcent  <- prop.table(table(iris$clase)) * 100
cbind(total=table(iris$clase), porcentaje=porcent)
```

Si, por otro lado, utilizamos `summary()` para columnas numéricas, tendremos:

```
> summary(iris[[1]])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
> summary(iris[[2]])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.800   3.000   3.054   3.300   4.400
> summary(iris[[3]])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.600   4.350   3.759   5.100   6.900
> summary(iris[[4]])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.100   0.300   1.300   1.199   1.800   2.500
```

que corresponde a una lista de seis parámetros de estadística descriptiva más bien básicos y que nos sirven para darnos una idea superficial aunque rápida de las características de nuestros datos.

El primero de ellos es el valor mínimo, el segundo el límite por debajo del cual están el 25% de valores más bajos si los ordenamos de menor a mayor, el tercero es la mediana, que es el valor que se encuentra a la mitad de esa lista (no es la media), el cuarto es la media, el quinto el tercer cuartil o el valor por debajo del cual está el 75% de valores menores que el mismo y, por último, el valor máximo.

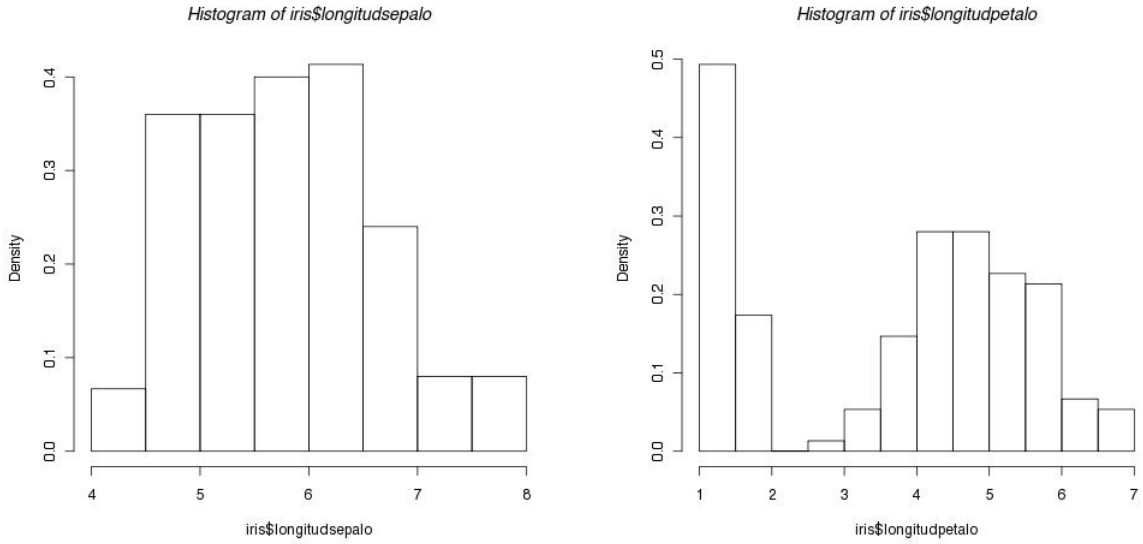
Aunque también podíamos haberlo hecho de una vez

```
> summary(iris)
 longitud.sepalo anchura.sepalo  longitud.petalo anchura.petalo
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.054   Mean   :3.759   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
            clase
 Iris-setosa    :50
 Iris-versicolor:50
 Iris-virginica :50
```

Es interesante notar que, por ejemplo, la diferencia entre media y mediana nos puede dar una idea del nivel de *skewness* de la muestra. Por ejemplo, si representamos un histograma para la longitud del sépalo, con el comando

```
> hist(iris$longitud.sepalo,probability="T")
```

tendremos la figura 2 (a), con la que podemos comprobar que se aproxima ligeramente a una normal (media y mediana son similares). Si ahora echamos la vista a la longitud del pétalo, podemos comprobar con la figura 2 (b) que la distribución está *torcida*:



(a) (b)

Figure 2: Histograma de longitud del Sépalo (a) y el Pétalo (b)

A su vez, comprobamos que media y mediana son diferentes.

El comando `hist()` nos permite hacer un histograma con las frecuencias de aparición de valores dentro de unos intervalos. El comando completo sería:

Definición de Comando

```
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, warn.unused = TRUE, ...)
```

Podemos representar, mediante una versión del histograma bastante enriquecido, el gráfico de la figura 3. y lo hacemos mediante la secuencia de comandos siguiente

```
# Histograma enriquecido con la pdf
dens<-density(iris$longitud.sepalo,na.rm=T)
hist(iris$longitud.sepalo, xlab="",
```

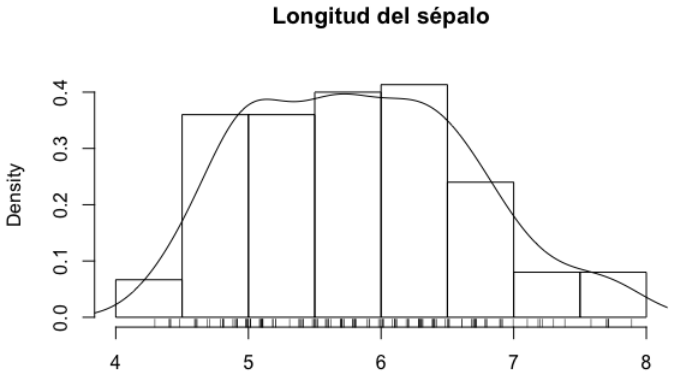


Figure 3: Histograma enriquecido de longitud del sépalo

```
    main="Longitud del sépalo", ylim=c(0,max(dens$y)*1.1),probability=T)
lines(dens)
rug(jitter(iris$longitud.sepalo))
```

en el que añadimos, mediante el uso de kernels, una estimación de la función de densidad de probabilidad (pdf). Además, también se le puede añadir al gráfico, mediante el comando `rug()`, la representación de los valores reales del atributo,bajo el eje x (usamos `jitter()` para añadir un poco de ruido aleatorio a los valores verdaderos, por si hubiese valores repetidos y, de este modo, aparezcan como una línea más gruesa).

Con este gráfico se puede visualizar con cierta facilidad la existencia de algún outlier. Aquí vemos que no hay ningún valor que destaque significativamente del resto (lo cual es normal ya que este cjto. no es precisamente muy irregular). También se puede observar que los valores están parcialmente "redondeados" y tienen poca precisión (están casi "discretizados" y agrupados con espacios entre ellos).

Podríamos usar los gráficos de `lattice` para obtener también un histograma enriquecido. No obstante primero tendríamos que crear una estructura de datos que nos facilite trabajar con fórmulas y grupos.

```
# creando una estructura de datos para usar lattice con facilidad
dt<-data.frame(clase=factor(levels=levels(iris[[5]])),
               variable=factor(levels=names(iris[,1:4])),
               value=vector(mode="numeric")
)
for (i in seq_along(iris[,1:4])) {
  dt<-rbind(dt,data.frame(clase=iris[[5]],variable=names(iris)[i],value=iris[[i]]))
}
```

Pero es más sencillo usar una instrucción de la librería `reshape2`, en particular la instrucción `melt()`.

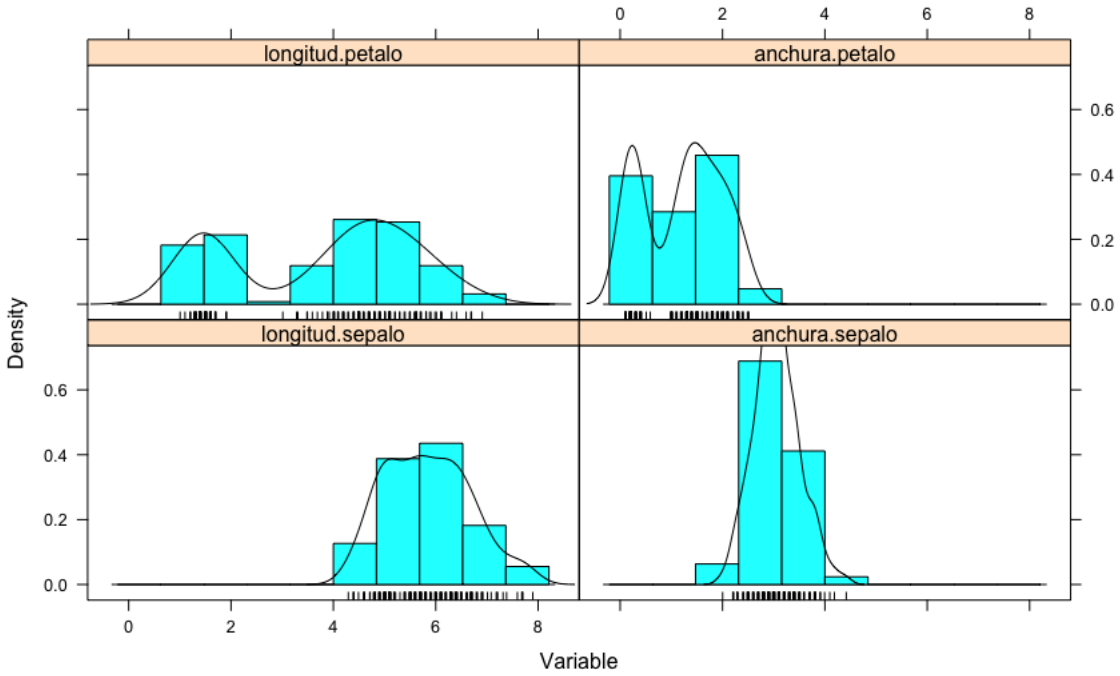


Figure 4: Histogramas "enriquecidos" de todas las variables del problema Iris

```
# creando una estructura de datos para usar lattice con facilidad con "melt()"
melt(iris,id.vars=5)
```

Ahora hacemos un histograma enriquecido. Como queremos añadir a cada gráfico elementos extras usamos varios paneles:

```
# histograma "enriquecido" usando lattice y mostrando todas las variables
histogram( ~ value | variable, data = melt(iris,id.vars=5), layout=c(2,2),
           xlab = "Variable", type = "density",
           panel = function(x, ...) {
             panel.histogram(x, ...)
             panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
             panel.rug(x=jitter(x),col="black")
           })
```

Y el resultado se puede ver en la figura 4.

Sin embargo, si lo que queremos es realmente ver cómo influye cada uno de los atributos en el problema de clasificación, atendiendo a cada una de las tres clases, podemos usar las gráficas de la figura 5. Son gráficas que muestran las distribuciones de probabilidad, para cada clase, representada por un color diferente. En este caso, tres grises. Los diferentes valores del vector numérico correspondiente se muestran en el eje horizontal y

las probabilidades de cada clase, según el rango de valores utilizado en el eje horizontal, aparecen en vertical. Obśervese como parece que, a priori, los atributos *Longitud Pétalo* y *Anchura Pétalo* discriminan bastante bien las tres clases (alto número de intervalos en el eje horizontal, con probabilidad 1). En la figura 5 (c) vemos que se puede clasificar toda la clase *Iris-Setosa* comprobando que la longitud del pétalo es menor que 2, y una buena parte de la clase *Iris-Virginica* si la longitud del pétalo es mayor que 5.5, por ejemplo.

Las gráficas de la figura 5 se obtienen mediante los comandos:

```
# spinegramas de los diferentes atributos

spineplot(clase ~ longitud.sepalo,data=iris)
spineplot(clase ~ anchura.sepalo,data=iris)
spineplot(clase ~ longitud.petalo,data=iris)
spineplot(clase ~ anchura.petalo,data=iris)
```

Veamos otro de los *feature plots* interesantes, el que muestra la densidad de probabilidad. Si utilizamos el comando básico se usa la misma escala para todos los diagramas de la parrilla así que podemos indicarle mediante `scales` que use para ambos ejes escalas diferentes en cada diagrama (se puede simplificar indicando directamente `scales=list(relation="free")` sin indicar eje y lo aplica a los dos).

```
# gráficos de densidad por clase, escala individual
escalas <- list(x=list(relation="free"), y=list(relation="free"))
featurePlot(x=iris[,1:4],y=iris[,5],plot="density", scales=escalas)
```

Si os fijáis en el resultado (figura 6), las distribuciones tienen más o menos la forma de gausianas, que es lo esperable cuando se refieren a medidas de alguna carácteristica física (en este caso tamaños) de una población de individuos.

Analizando precisamente la figura 6 respecto a la longitud del Pétalo se entiende como es que el histograma que aparecía en la figura 2 (b) estaba "skewed" (era asimétrico), ya que comprobamnos que es la suma de tres poblaciones diferentes (las tres clases de orquídeas, que generan tres gausianas), una de ellas bastante diferenciada de las otras dos. Cuando en medidas de carácteristicas vemos una distribución multimodal (que tiene varios picos, o modos) se puede sospechar que hay varias clases o algún atributo oculto.

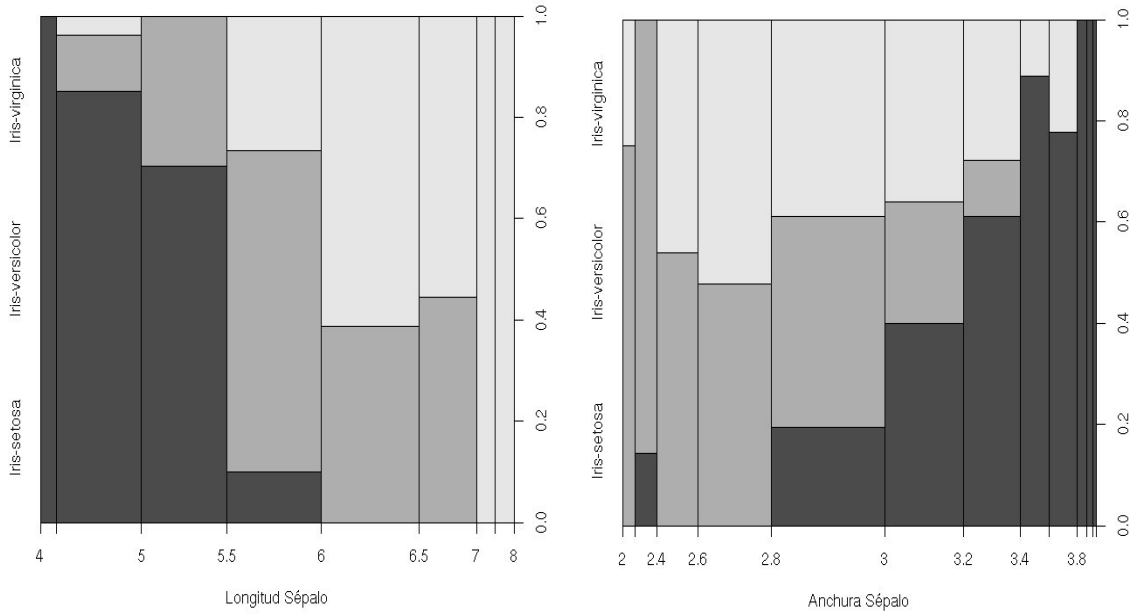
Ejercicio

Obtener el mismo gráfico que la figura 6 usando `lattice`.

Hay un tipo de gráfico que muestra bastante información sobre la dispersión, asimetría estadística (*skewness*) y posibles valores atípicos (*outliers*) sin asumir mucho sobre el tipo de distribución subyacente. Son los diagramas box-and-whisker que se pueden obtener mediante:

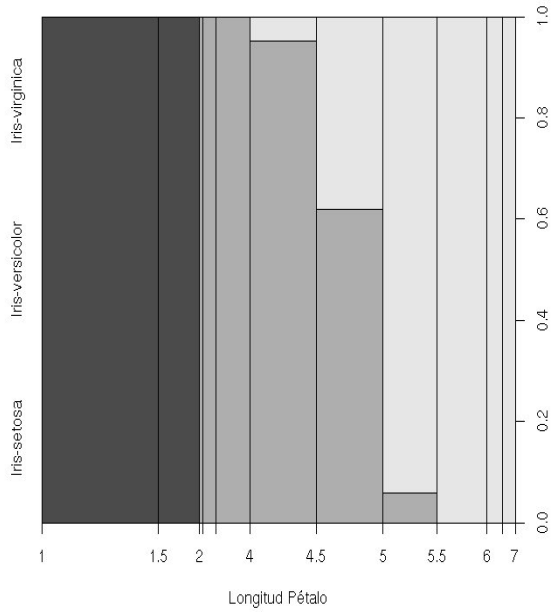
```
# Whisker-Plot usando gráficos tradicionales
boxplot(iris[[1]],iris[[2]],iris[[3]],iris[[4]],names = c("Long. Sépalo",
"Anchura Sépalo", "Long. Pétalo", "Anchura Pétalo"))

# Whisker-Plot usando lattice
```

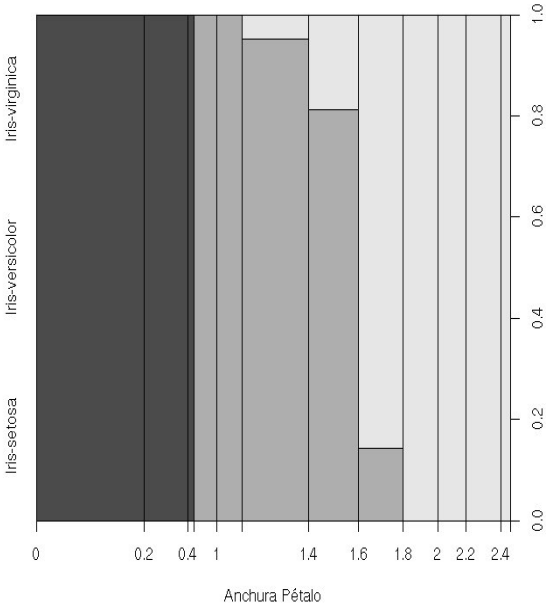


(a)

(b)



(c)



(d)

Figure 5: Gráficas de distribución de probabilidad de las tres clases del conjunto Iris, para cada uno de los cuatro vectores de entrada.

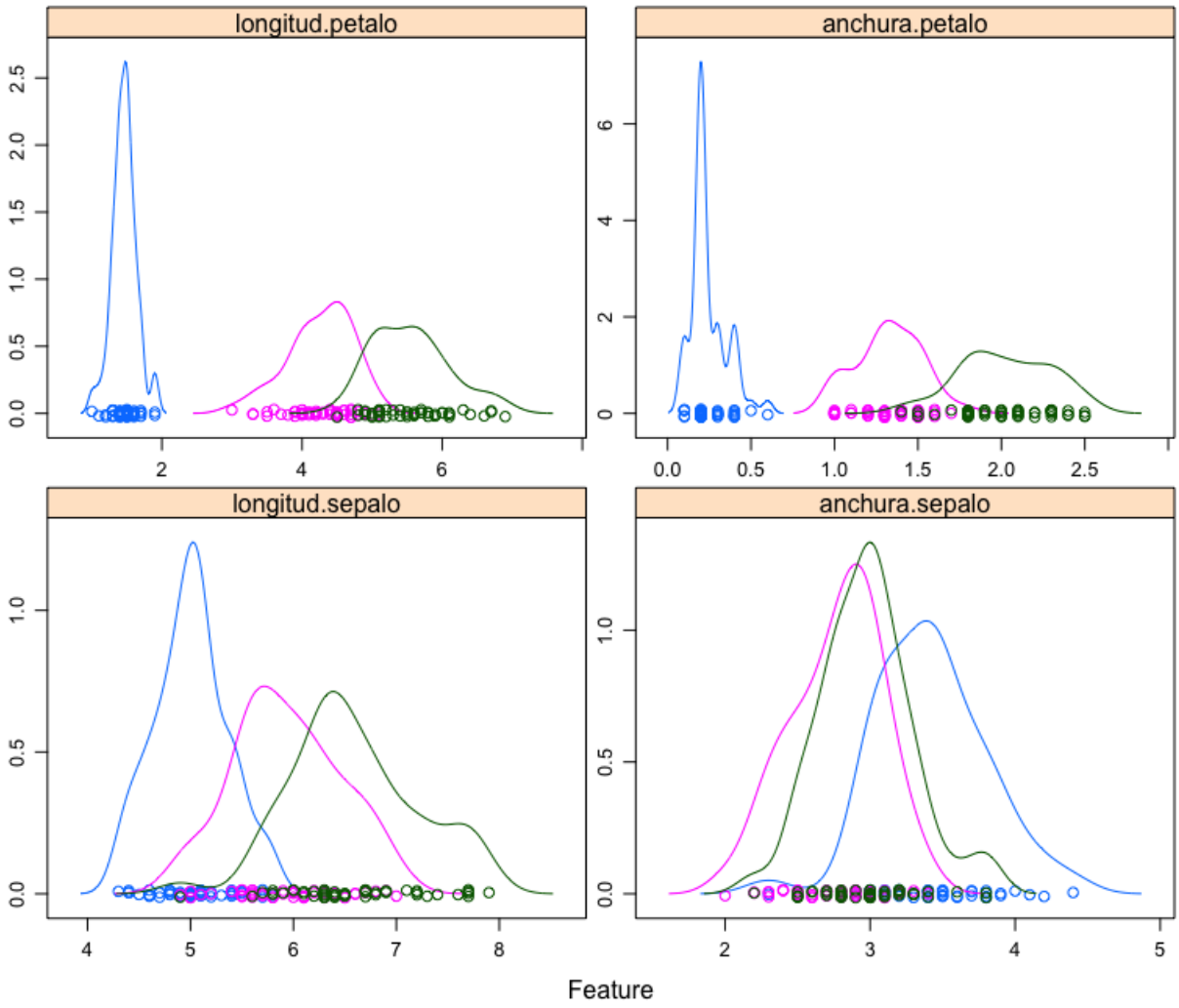


Figure 6: Gráfico de densidad por clase para el conjunto iris usando `featurePlot()`

```
bwplot(~value | variable, data = melt(iris,id.vars=5))
```

que nos daría el diagrama tipo Box-Whisker que aparece en la figura 7. a la derecha, en el que aparecen el máximo y mínimo que no están fuera de rango (extremos de los segmentos), el percentil 25% (i.e. Q_1) y el 75% (i.e. Q_3) que sería la caja, la mediana (línea horizontal que divide la caja) y, por último, los valores fuera de rango (i.e. *outliers*, que aparecen como puntos/círculos por encima y/o debajo de los máximos/mínimos). La mediana se obtiene con el valor en medio de la lista de valores, si el número es impar, o con la media aritmética de los dos valores en el centro.

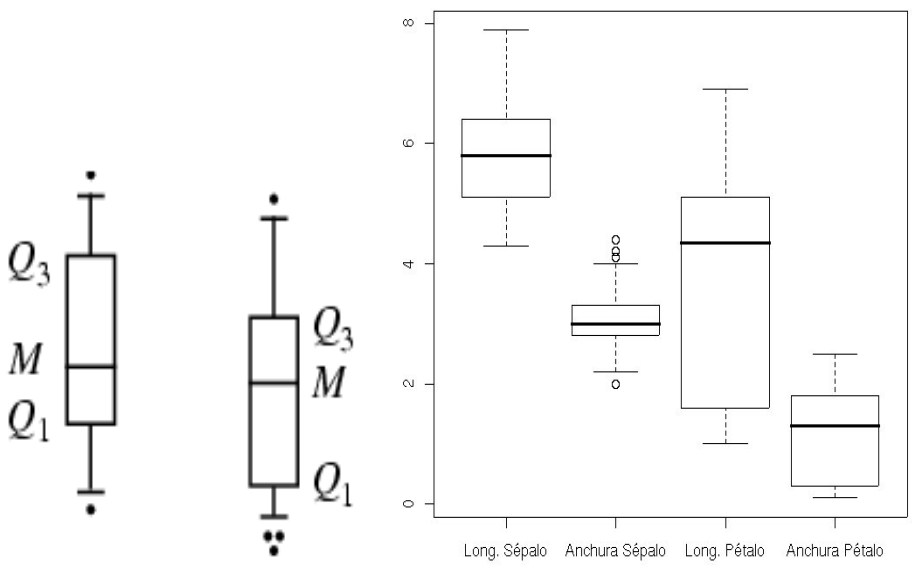


Figure 7: Diagramas Whisker-Plot típicos para dos vectores de datos (izquierda) y los correspondientes a los cuatro atributos de entrada del conjunto Iris (derecha).

Importante: En este típo de gráfico se usa, para considerar que un dato es un outlier (valor extremo) se usa el criterio de Tukey (cerca de Tukey), que se basa en la longitud del rango intercuartil (la diferencia entre los valores Q_3 y Q_1), y en este caso particular decide que es un *outlier* si está más allá de de 3/2 de la diferencia entre los valores Q_3 y Q_1.

¿Qué podemos decir a la luz de los diagramas box-Whisker de la derecha de la figura 7 ? Individualmente, podemos ver que en la anchura del sépalo hay valores bastante extremos (recordemos que los puntos/círculos indican la existencia y cuantos *outliers* hay), lo que podría dificultar el proceso de aprendizaje con algunos algoritmos (los sensibles a *outliers*). Por otro lado, tanto la longitud como la anchura del sépalo tienen una mediana más o menos centrada lo que puede llevar a pensar en una distribución normal de los datos o, al menos, no asimétrica estadísticamente. No podemos decir lo mismo de los vectores relativos al pétalo.

De todos modos los diagrams Box-Whisker (y los qqplots que veremos a continuación), en realidad, son útiles cuando se utilizan en distribuciones más o menos simétricas y unimodales (un solo "máximo"), siendo engañosos en otros casos. Precisamente en este ejemplo, como se puede ver en los histogramas enriquecidos, tenemos distribuciones con varios modos (cada uno asociado a la clase subyacente) así que sería interesante ver los whiskers desagregados por clase. Es muy fácil hacerlo con `lattice`:

```
# Whisker-Plot por clase usando lattice
bwplot(clase ~ value | variable, data = melt(iris,id.vars=5))
```

Así, en la figura 8 tenemos otros *outliers* diferentes a los de la figura previa. Es importante recordar que considerar un valor como atípico (outlier) es siempre un riesgo porque podría no serlo en realidad. Cada dato que se "arregla" introduce un potencial sesgo en los datos (que ya no son los originales) y podríamos estar, en realidad, dañándolos. Es imposible escapar a esta incertidumbre. También cabe destacar que una

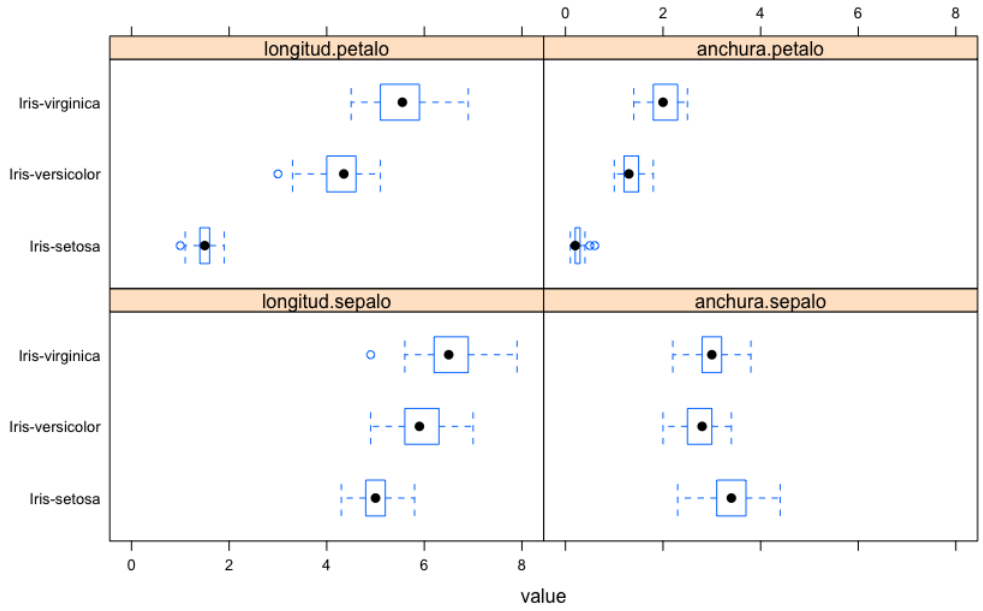


Figure 8: Whisker plot desagregado por clases.

cosa es "arreglar" los datos y otra el hacer determinadas transformaciones sobre ellos (veremos algunas en la sección 10.2) que solo cambian escalas o sistemas de referencia, puesto que en cierto sentido no cambian el modelo subyacente del que los datos son una muestra. En cambio eliminar/modificar valores atípicos o desconocidos (missing) si que cambia el modelo subyacente.

Aunque no entraremos en el uso de ggplot2, que es un paquete gráfico para R que trata de poner cierto orden en como mostrar gráficos con R, incluiremos también la forma alternativa usando ese paquete de algunos de los gráficos de esta práctica. Para poder mostrar los boxplot de ggplot2 en una misma figura hay que transformar los datos usando el comando `melt()` de nuevo. El Whisker-Plot usando ggplot2 se obtendría:

```
# Boxplots de múltiples variables usando ggplot2
g<-ggplot(melt(iris,measure.vars=1:4),aes(x=variable, y=value))
g+geom_boxplot()
```

Para poder ver si una muestra sigue una determinada distribución estadística se pueden usar los diagramas del tipo $Q-Q$ como las que aparecen en la figura 9. En estas gráficas podemos ver una representación de los percentiles de cada uno de los atributos, con respecto a los de una normal centrada en cero (u otras distribuciones si se le indica así en el comando). Como referencia, se incluye una linea con una orientación de 45^0 con respecto al eje orizontal. Cada punto (x, y) hace referencia a un percentil concreto del 0 al 100, de tal forma que si el del percentil 25% es (a, b) significa que el de la normal es a y el del vector de nuestro conjunto de datos es b. Cuanto más se ajusten a la línea representada, másse parecerá a una distribución normal (que es la utilizada en este caso).

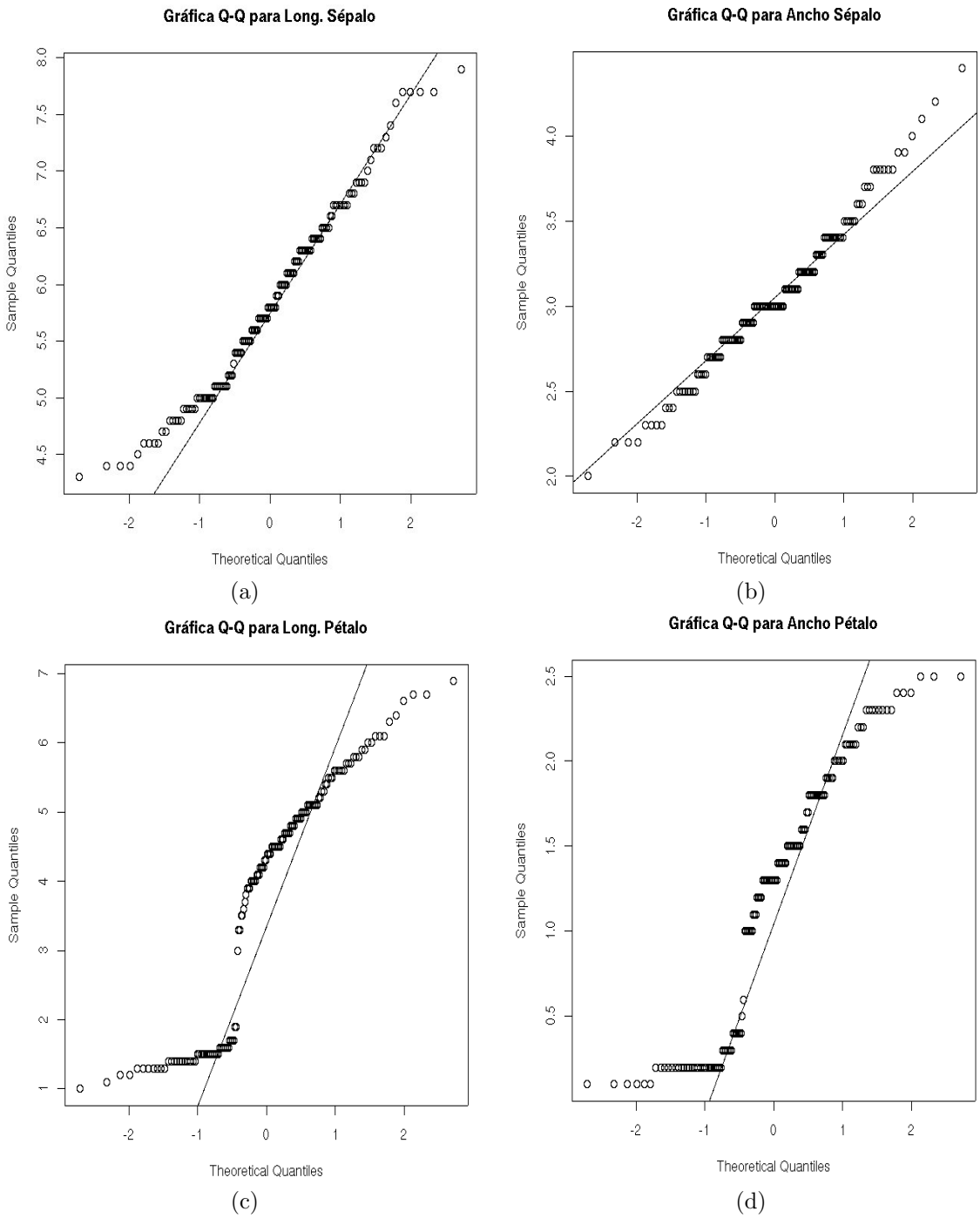


Figure 9: Comprobación visual para determinar si los vectores se ajustan a una normal

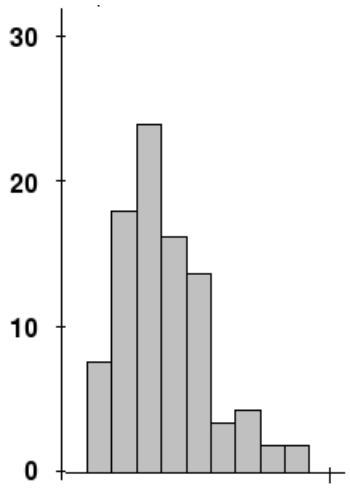


Figure 10: Distribución desplazada a la izquierda

Dichas gráficas las vamos a obtener con los siguientes comandos:

```
# QQ plot usando gráficos clásicos
qqnorm(iris[[1]], main="Gráfica Q-Q para Long. Sépalo")
qqline(iris[[1]], main="Gráfica Q-Q para Long. Sépalo")
qqnorm(iris[[2]], main="Gráfica Q-Q para Ancho Sépalo")
qqline(iris[[2]], main="Gráfica Q-Q para Ancho Sépalo")
qqnorm(iris[[3]], main="Gráfica Q-Q para Long. Pétalo")
qqline(iris[[3]], main="Gráfica Q-Q para Long. Pétalo")
qqnorm(iris[[4]], main="Gráfica Q-Q para Ancho Pétalo")
qqline(iris[[4]], main="Gráfica Q-Q para Ancho Pétalo")
```

Hay que tener en cuenta que los percentiles quedan a la izquierda de la línea trazada como referencia, se dice que la distribución de probabilidad está desplazada a la izquierda. Una distribución desplazada a izquierda la tenemos en la figura 10.

Para conseguir esos gráficos con `lattice` de nuevo usamos varios paneles y el comando sería:

```
# QQ plot usando lattice
qqmath(~ value | variable, data = melt(iris,id.vars=5),layout=c(2,2),
       type = c("p", "g"),
       distribution=qnorm,
       prepanel = prepanel.qqmathline,
       panel = function(x, ...) {
         panel.qqmathline(x, ...)
         panel.qqmath(x, ...)
       })
```

Para poder mostrar los qq plots en ggplot2 incluyendo la línea teórica perfecta (la que añade `qqline()` necesitamos crear la función que nos la calcule y añadir esa línea al gráfico. En particular

```
# Función para mostrar un qqnorm + qqline usando ggplot2
# Extraída de https://stackoverflow.com/questions/4357031/qqnorm-and-qqline-in-ggplot2

qqplot.data <- function (vec) # argument: vector of numbers
{  # following four lines from base R's qqline()
  y <- quantile(vec[!is.na(vec)], c(0.25, 0.75))
  x <- qnorm(c(0.25, 0.75))
  slope <- diff(y)/diff(x)
  int <- y[1L] - slope * x[1L]

  d <- data.frame(resids = vec)

  ggplot(d, aes(sample = resids)) + stat_qq() + geom_abline(slope = slope,
  intercept = int)
}
```

Una vez definida la función `qqplot.data` se utilizaría de la siguiente forma:

```
# QQ plot usando gráficos ggplot2

qqplot.data(iris[[1]])+ggtitle("Gráfica Q-Q para Long. Sépalo")
qqplot.data(iris[[2]])+ggtitle("Gráfica Q-Q para Ancho Sépalo")
qqplot.data(iris[[3]])+ggtitle("Gráfica Q-Q para Long. Pétalo")
qqplot.data(iris[[4]])+ggtitle("Gráfica Q-Q para Ancho Sépalo")
```

No obstante, para más ejemplos sobre como transformar gráficos de `lattice` a `ggplot2` se pueden ver una serie de posts del siguiente blog cuya serie completa es:

https://learnr.wordpress.com/2009/06/28/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-1/
https://learnr.wordpress.com/2009/06/29/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-2/
https://learnr.wordpress.com/2009/07/02/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-3/
https://learnr.wordpress.com/2009/07/02/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-4/
https://learnr.wordpress.com/2009/07/15/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-5/
https://learnr.wordpress.com/2009/07/20/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-6/
https://learnr.wordpress.com/2009/07/27/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-7/
https://learnr.wordpress.com/2009/08/03/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-8/

https://learnr.wordpress.com/2009/08/10/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-9/
https://learnr.wordpress.com/2009/08/11/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-10/
https://learnr.wordpress.com/2009/08/13/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-11/
https://learnr.wordpress.com/2009/08/18/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-13/
https://learnr.wordpress.com/2009/08/20/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-13-2/

3.2 Análisis multivariable

Hasta ahora hemos tratado cada uno de los atributos independientemente con respecto del resto. Pero también podemos relacionarlos entre ellos para ver correlaciones y demás detalles interesantes. Podemos representar, en una simple gráfica de puntos en dos dimensiones, los cuatro vectores por pares, mediante el comando:

```
pairs(iris[1:4],col=as.numeric(iris$clase))
```

Alternativamente la librería caret tiene una función `featurePlot()` que nos permite visualizar las relaciones dos a dos entre variables con diferentes tipos de gráficos. Veamos la descripción del comando:

Definición de Comando

```
featurePlot(x, y, plot = if (is.factor(y)) "strip" else "scatter",
  labels = c("Feature", ""), ...)
```

Arguments

x	a matrix or data frame of continuous feature/probe/spectra data.
y	a factor indicating class membership.
plot	the type of plot. For classification: *box*, *strip*, *density*, *pairs* or *ellipse*. For regression, *pairs* or *scatter*
labels	a bad attempt at pre-defined axis labels
...	options passed to lattice calls.

En realidad es un `wrapper` para los gráficos de `lattice`. Probaremos varias de ellas relacionadas con clasificación y nos quedaremos con la última:

```
featurePlot(x=iris[,1:4],y=iris[,5],plot="box")
featurePlot(x=iris[,1:4],y=iris[,5],plot="strip")
featurePlot(x=iris[,1:4],y=iris[,5],plot="strip", jitter=T)
featurePlot(x=iris[,1:4],y=iris[,5],plot="density")
featurePlot(x=iris[,1:4],y=iris[,5],plot="pairs")
```

```
featurePlot(x=iris[,1:4],y=iris[,5],plot="ellipse")
```

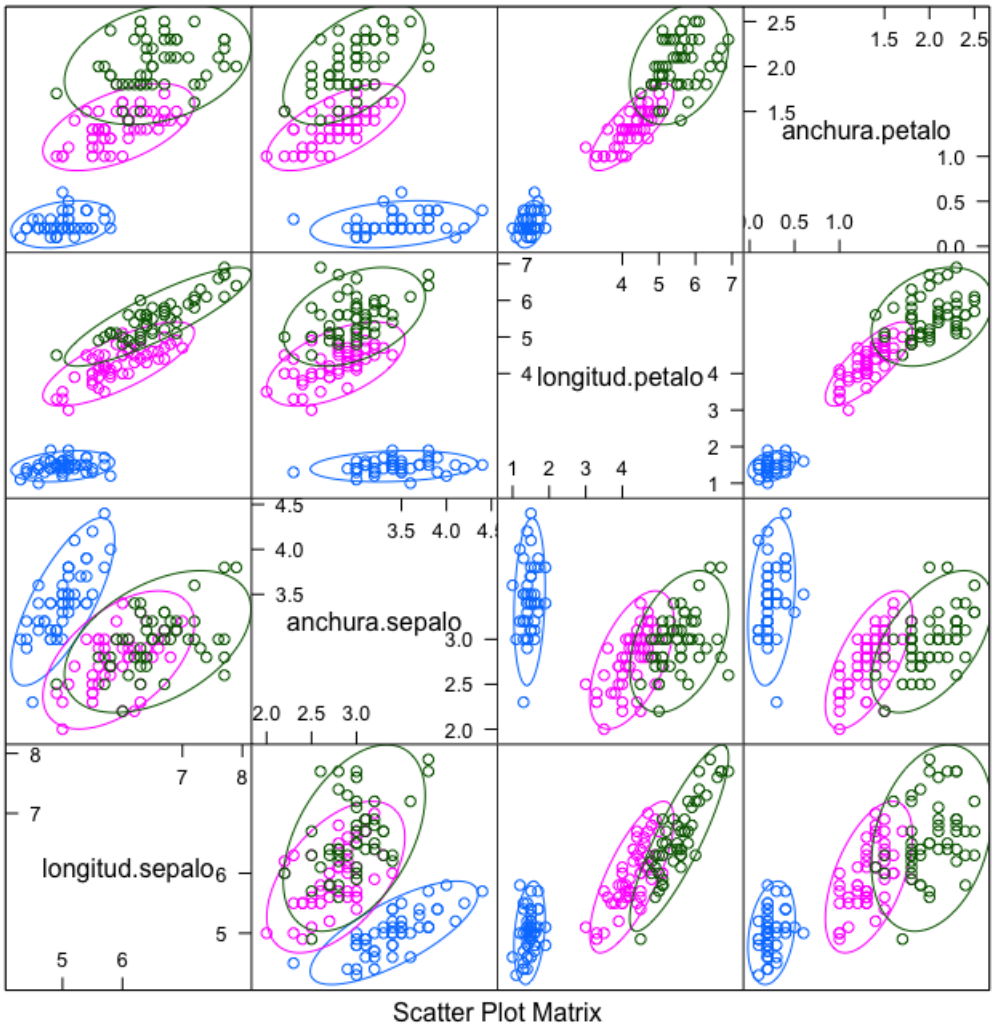


Figure 11: Representación del scatter plot para el conjunto iris usando `pairs()`

La panorámica mostrada en la figura 11 es muy interesante y útil para ver cómo se relacionan las variables por pares. Obsérvese que la diagonal principal de la matriz formada por las gráficas está formada por las etiquetas de los vectores que aparecen en el eje horizontal, para todas las gráficas de la columa. Análogamente, aparecen en el eje vertical, para todas las gráficas de la fila. Por ejemplo, la gráfica inferior derecha hace referencia a la gráfica *(longitud sépalo, anchura pétalo)*. Podemos comprobar como los atributos *anchura pétalo* o *longitud pétalo*, combinado con cualquiera de los 3 restantes sirven para clasificar, en cierta

medida, los puntos, al menos la una de las clases (azul) frente a las otras dos. Otras gráficas llaman la atención por la alta correlación entre sus atributos (e.g. *longitud pétalo* con *anchura pétalo*), pues pueden verse que la forma de la nube de puntos (independientemente de la clase) sigue más o menos una diagonal.

4 Trabajando con datos categóricos

A menudo, en el contexto del aprendizaje nos vamos a encontrar con conjuntos de datos en los que tanto $\bar{x}$ como y son categóricos. Es, por ejemplo, el caso del conjuto de datos `breast-cancer`[3]. En este conjunto de datos hay 10 atributos categóricos, y se corresponde con un problema de clasificación binaria. En este caso, lo cargamos del UCI repository directamente. Fíjaos que, como la URL es tan larga que no cabe en una línea, se usa el comando `paste()` para que quepa en la caja de texto (no sería necesario si se teclea directamente en el interprete R) y que aun funcione un copy/paste del contenido de la caja.

```
breast <- read.table(paste("https://archive.ics.uci.edu/ml/",
   "machine-learning-databases/breast-cancer-wisconsin/",
   "breast-cancer-wisconsin.data", sep=""),
 sep=",",header=F)
head(breast)
```

mostrará el siguiente contenido

```
        V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11
1 1000025  5  1  1  1  2  1  3  1   1   2
2 1002945  5  4  4  5  7 10  3  2   1   2
3 1015425  3  1  1  1  2  2  3  1   1   2
4 1016277  6  8  8  1  3  4  3  7   1   2
5 1017023  4  1  1  3  2  1  3  1   1   2
6 1017122  8 10 10  8  7 10  9  7   1   4
```

pero si comprobamos la estructura con `str(breast)` comprobaremos que es un desastre la importación de datos puesto que los vectores han sido importados como vectores numéricos cuando son datos categóricos y deberían ser factores.

```
'data.frame': 699 obs. of  11 variables:
 $ V1 : int  1000025 1002945 1015425 1016277 1017023 1017122 1018099 1018561 1033078 1033078 ...
 $ V2 : int  5 5 3 6 4 8 1 2 2 4 ...
 $ V3 : int  1 4 1 8 1 10 1 1 1 2 ...
 $ V4 : int  1 4 1 8 1 10 1 2 1 1 ...
 $ V5 : int  1 5 1 1 3 8 1 1 1 1 ...
 $ V6 : int  2 7 2 3 2 7 2 2 2 2 ...
 $ V7 : Factor w/ 11 levels "1","10","2","3",..: 1 2 3 5 1 2 2 1 1 1 ...
 $ V8 : int  3 3 3 3 3 9 3 3 1 2 ...
 $ V9 : int  1 2 1 7 1 7 1 1 1 1 ...
 $ V10: int  1 1 1 1 1 1 1 1 5 1 ...
 $ V11: int  2 2 2 2 2 4 2 2 2 2 ...
```

[3] http://www.ics.uci.edu/∼mlearn/databases/breast-cancer/

Lo podemos arreglar transformando dichos vectores numéricos en factores con:

```
breast <- data.frame(lapply(breast,FUN=as.factor))
str(breast)
```

```
'data.frame': 699 obs. of  11 variables:
 $ V1 : Factor w/ 645 levels "61634","63375",..: 173 176 177 178 180 181 182 183 187 187 ...
 $ V2 : Factor w/ 10 levels "1","2","3","4",..: 5 5 3 6 4 8 1 2 2 4 ...
 $ V3 : Factor w/ 10 levels "1","2","3","4",..: 1 4 1 8 1 10 1 1 1 2 ...
 $ V4 : Factor w/ 10 levels "1","2","3","4",..: 1 4 1 8 1 10 1 2 1 1 ...
 $ V5 : Factor w/ 10 levels "1","2","3","4",..: 1 5 1 1 3 8 1 1 1 1 ...
 $ V6 : Factor w/ 10 levels "1","2","3","4",..: 2 7 2 3 2 7 2 2 2 2 ...
 $ V7 : Factor w/ 11 levels "1","10","2","3",..: 1 2 3 5 1 2 2 1 1 1 ...
 $ V8 : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
 $ V9 : Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
 $ V10: Factor w/ 9 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
 $ V11: Factor w/ 2 levels "2","4": 1 1 1 1 1 2 1 1 1 1 ...
```

Todavía tenemos que modificar más cosas. Si nos vamos de nuevo a `https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.names`, donde aparece la descripción de los datos, veremos que la clase es la variable 11, y que el valor 2 significa tumor benigno y el 4 tumor maligno. Podemos cambiar las etiquetas de ese factor para que sean mas legibles.

```
levels(breast$V11)<-c("benign","malignant")
```

En la información detallada por `str(breast)` nos muestra otras anomalías. La V7 tiene 11 niveles (y en la descripción de `breast-cancer-wisconsin.names` dice que son variables con 10 niveles). Si obtenemos los niveles de dicha variable vemos cual es el problema.

```
> levels(breast$V7)
 [1] "1"  "10" "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "?"
```

que muestra que hay un valor "?", es decir, la base de datos tiene datos incompletos y, dichos datos, están etiquetados con el carácter "?" (que no es el habitual que sería "NA"). Pero antes de ello vemos otro problema con la variable V10 que solo tiene 9 niveles, es decir, hay algún valor que nunca aparece en los datos. Vemos cual es mostrando los niveles de dicha variable.

```
> levels(breast$V10)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "10"
```

que es el valor "9". Se le puede añadir fácilmente con el comando:

```
> levels(breast$V10)<-c(levels(breast$V10),"9")
> str(breast$V10)
 Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
> levels(breast$V10)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "10" "9"
```

Si nos desagrada que ahora los niveles no estén "ordenados" podemos solucionarlo con

```
> breast$V10<-factor(breast$V10,levels=sort(as.integer(levels(breast$V10))))
> levels(breast$V10)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Si seguimos quisquillosos podemos comprobar que todos los niveles están "ordenados":

```
> lapply(breast,FUN=levels)
$`Clump Thickness`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Uniformity of Cell Size `
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Uniformity of Cell Shape`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Marginal Adhesion`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Single Epithelial Cell Size`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Bare Nuclei`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Bland Chromatin`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$`Normal Nucleoli`
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$Mitoses
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$Class
[1] "benign"    "malignant"
```

Pero recordemos que tenemos que volver a cargar bien la base de datos con el "missing data" bien codificado como "NA" en vez de "¿". También hay que arreglar el asunto de ese valor que no aparece nunca en los datos de entrenchment (pero es un valor legítimo), y luego volver a hacer las transformaciones.

```
breast <- read.table(paste("https://archive.ics.uci.edu/ml/",
   "machine-learning-databases/breast-cancer-wisconsin/",
   "breast-cancer-wisconsin.data", sep=""),
 sep=",",header=F, na.strings ="?")
```

Antes de pasarlos a factores vamos a ver cuantos datos, y cuales de ellos, tienen "missing data", usaremos el comando `complete.cases()` que veremos más adelante en la sección de como tratar los datos nulos.

Igualmente, y por ahora, no vamos a hacer nada con esos datos perdidos (ya veremos que hacer en dicha sección).

```
>sum(!complete.cases(breast))
[1]  16
> breast[!complete.cases(breast),]
         V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11
24  1057013  8  4  5  1  2 NA  7  3   1   4
41  1096800  6  6  6  9  6 NA  7  8   1   2
140 1183246  1  1  1  1  1 NA  2  1   1   2
146 1184840  1  1  3  1  2 NA  2  1   1   2
159 1193683  1  1  2  1  3 NA  1  1   1   2
165 1197510  5  1  1  1  2 NA  3  1   1   2
236 1241232  3  1  4  1  2 NA  3  1   1   2
250  169356  3  1  1  1  2 NA  3  1   1   2
276  432809  3  1  3  1  2 NA  2  1   1   2
293  563649  8  8  8  1  2 NA  6 10   1   4
295  606140  1  1  1  1  2 NA  2  1   1   2
298   61634  5  4  3  1  2 NA  2  3   1   2
316  704168  4  6  5  6  7 NA  4  9   1   2
322  733639  3  1  1  1  2 NA  3  1   1   2
412 1238464  1  1  1  1  1 NA  2  1   1   2
618 1057067  1  1  1  1  1 NA  1  1   1   2
```

Repetimos todos los pasos para convertir en factores, corregir etiquetas, añadir niveles y ordenarlos.

```
# Transformamos todo a factores
breast <- data.frame(lapply(breast,FUN=as.factor))
# Arreglamos etiquetas de V11
levels(breast$V11)<-c("benign","malignant")
# Añadimos etiqueta no usada y ordenamos etiquetas de V10
levels(breast$V10)<-c(levels(breast$V10),"9")
breast$V10<-factor(breast$V10,levels=sort(as.integer(levels(x))))
```

por último vemos que V1 es un identificador individual para cada dato, que no tiene utilidad para el aprendizaje, con lo que podríamos eliminarlo.

```
# Eliminamos V1
breast$V1<-NULL
```

También podríamos cambiarle las etiquetas a las variables para que se correspondan con los nombres que hay en la documentación.

```
names(breast)<-c("Clump Thickness","Uniformity of Cell Size ","Uniformity of Cell Shape",
"Marginal Adhesion", "Single Epithelial Cell Size", "Bare Nuclei", "Bland Chromatin",
"Normal Nucleoli","Mitoses","Class")
```

Y así finalmente ya tenemos la base de datos preparada para usarla:

```
> str(breast)
'data.frame': 699 obs. of  10 variables:
 $ Clump Thickness            : Factor w/ 10 levels "1","2","3","4",..: 5 5 3 6 4 8 1 2 2 4 ...
 $ Uniformity of Cell Size    : Factor w/ 10 levels "1","2","3","4",..: 1 4 1 8 1 10 1 1 1 2 ...
 $ Uniformity of Cell Shape   : Factor w/ 10 levels "1","2","3","4",..: 1 4 1 8 1 10 1 2 1 1 ...
 $ Marginal Adhesion          : Factor w/ 10 levels "1","2","3","4",..: 1 5 1 1 3 8 1 1 1 1 ...
 $ Single Epithelial Cell Size: Factor w/ 10 levels "1","2","3","4",..: 2 7 2 3 2 7 2 2 2 2 ...
 $ Bare Nuclei                : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
 $ Bland Chromatin            : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
 $ Normal Nucleoli            : Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
 $ Mitoses                    : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
 $ Class                      : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

Si queremos obtener un resumen estadístico de datos categóricos como estos, obtendremos lo siguiente:

```
> summary(breast)
```

y obtenemos esa descripción básica de la distribución de las distintas categorías (muestran las 6 más frecuentes y agrega las restantes).

```
Clump Thickness Uniformity of Cell Size  Uniformity of Cell Shape Marginal Adhesion
 1      :145     1      :384              1      :353              1      :407
 5      :130     10     : 67              2      : 59              2      : 58
 3      :108     3      : 52              10     : 58              3      : 58
 4      : 80     2      : 45              3      : 56              10     : 55
 10     : 69     4      : 40              4      : 44              4      : 33
 2      : 50     5      : 30              5      : 34              8      : 25
 (Other):117     (Other): 81              (Other): 95              (Other): 63
 Single Epithelial Cell Size  Bare Nuclei  Bland Chromatin Normal Nucleoli    Mitoses
 2      :386                  1      :402   2      :166     1      :443     1      :579
 3      : 72                  10     :132   3      :165     10     : 61     2      : 35
 4      : 48                  2      : 30   1      :152     3      : 44     3      : 33
 1      : 47                  5      : 30   7      : 73     2      : 36     10     : 14
 6      : 41                  3      : 28   4      : 40     8      : 24     4      : 12
 5      : 39                  (Other): 61   5      : 34     6      : 22     7      :  9
 (Other): 66                  NA's   : 16   (Other): 69     (Other): 69     (Other): 17
       Class
 benign   :458
 malignant:241
```

La verdad es que para este conjunto de datos particular nos podríamos haber ahorrado todo este trabajo, puesto que es uno de los incluidos en el paquete `mlbench` y podríamos tener esta misma base de datos con:

```
library(mlbench)
data(BreastCancer)
str(BreastCancer)
```

Y observando los datos de esta base de datos cargada de la libreria vemos que las primeras cinco variables son factores ordenados:

```
 $ Id             : chr  "1000025" "1002945" "1015425" "1016277" ...
 $ Cl.thickness   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 5 5 3 6 4 8 1 2 2 4 ...
 $ Cell.size      : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 1 1 2 ...
 $ Cell.shape     : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 2 1 1 ...
 $ Marg.adhesion  : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 5 1 1 3 8 1 1 1 1 ...
 $ Epith.c.size   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 7 2 3 2 7 2 2 2 2 ...
 $ Bare.nuclei    : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
 $ Bl.cromatin    : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
 $ Normal.nucleoli: Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
 $ Mitoses        : Factor w/ 9 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
 $ Class          : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

Es fácil arreglar nuestra versión de esta base de datos transformando esos factores en factores ordenados:

```
> for (i in 1:5) breast[,i]<-factor(breast[,i],ordered=T)
> str(breast)
'data.frame': 699 obs. of  10 variables:
 $ Clump Thickness            : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 5 5 3 6 4 8 1 2 2 4 ...
 $ Uniformity of Cell Size    : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 1 1 2 ...
 $ Uniformity of Cell Shape   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 2 1 1 ...
 $ Marginal Adhesion          : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 5 1 1 3 8 1 1 1 1 ...
 $ Single Epithelial Cell Size: Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 7 2 3 2 7 2 2 2 2 ...
 $ Bare Nuclei                : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
 $ Bland Chromatin            : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
 $ Normal Nucleoli            : Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
 $ Mitoses                    : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
 $ Class                      : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

Por último veamos como hacer un scatter plot de estos datos. Recordemos que se necesitarían vectores numéricos en los atributos, y estos son factores, por lo que tendremos que convertirlos y usaremos `lapply()`:

```
featurePlot(x=lapply(breast[,-10],FUN=as.integer),y=breast$Class,"strip",jitter=T)
```

Ejercicio

Realicemos los anteriores pasos con una base de datos nueva, `Soybean` presente tanto en la librería `mlbench` como en el UCI Repository en `https://archive.ics.uci.edu/ml/machine-learning-databases/soybean`. Trata de importar los datos desde el UCI (fíjate que está dividida en dos ficheros, `soybean-large.data` y `soybean-large.test` y tendrás que combinarlos) de forma que queden como la base de datos proporcinada

por `mlbench` mediante el comando `data(Soybean`. Trata de arreglar los datos y anímate a estudiarlos un poco.

Con esto acabamos la primera sesión. En la próxima empezaremos con el pre-procesado de datos.

---

5 Datasets utilizados

En el tutorial usaremos algunos datasets bastante conocidos y frecuentes en el mundo de machine learning que, por ellos, se usan para hacer benchmarking (poner a prueba) los diferentes modelos, y que también nos permiten comprobar si estamos aplicando más o menos bien las técnicas.

En particular utilizaremos:

- Iris Data Set (`iris`)
- Wisconsin Breast Cancer Data Set (`Breast-Cancer`)
- Diabetes en la tribu de los indios Pima (`PimaIndianDiabetes`)
- Composición de cristal para investigación forense (`Glass`)
- Datos sobre viviendas en Boston del censo de 1970. La versión 2 está corregida del original y tiene información adicional. (`BostonHousing2`)
- Adult Data Set o Census Income (`adult`)
- Wine Data Set (`wine`)
- Algas, utilizado en el libro de L. Torgo de la bibliografía (`algae`)

La mayoría de ellos ya están disponibles en R con las librerías `mlbench` y la propia `caret`. Cargarlos es tán facil como usar el comando `data()`.

```
library(caret)
library(mlbench)
install.packages("DMwR2")      # Contiene el dataset de las "algas"

data(iris)
data(BreastCancer)
data(PimaIndiansDiabetes)
data(Glass)
data(BostonHousing2)
data(algae,package="DMwR2") # otra forma de cargar los datos sin cargar la librería
```

Otros datasets, como Adult, los podemos cargar directamente del UCI repository, por ejemplo:

```
# Base de datos adult, necesita parámetros específicos para cargar bien
adult<- read.table(
         "http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
         sep=",",header=F,col.names=c("age", "type_employer", "fnlwgt", "education",
        "education_num","marital", "occupation", "relationship", "race","sex",
        "capital_gain", "capital_loss", "hr_per_week","country", "income"),
        fill=FALSE,strip.white=T,na.strings = c("?"))

# Base de datos white wine. Está en formato csv
wine<- read.csv2(
```

```
  paste("https://archive.ics.uci.edu/ml/machine-learning-databases/",
        "wine-quality/winequality-white.csv",
       sep="")
  ,dec="."
)
wine$class<-factor(wine$quality) # La clase como un factor
wine$quality<-NULL               # y se elimina la quality (es la clase)
# Las etiquetas de los niveles no deben ser números
levels(wine$class)<-make.names(levels(wine$class))
```

Fijate que la clase de algunos datasets de clasificación está codificada con números y la hemos de convertir en factor para que caret funcione adecuadamente y cree modelos de clasificación para dichos datasets. Por otro lado mencionar que, a veces, se usan modelos de regresión para clasificar codificando las clases como enteros y redondeando después la salida para convertir el número real que produce el modelo de regresión en el código entero de la clase.

Información específica de cada dataset se puede encontrar en el UCI o en la documentación de las librerías que las proporcionan.

6 Pre-procesado de datos (I): Tratamiento de outliers y nulos

6.1 Tratamiento de valores fuera de rango

Para este apartado, nos vamos a basar en el excelente libro de Luis Torgo que podeis encontrar en la Web de la asignatura. En este se ofrecen algunos ejemplos sencillos de cómo tratar los valores fuera de rango de un conjunto con *outliers* típico. Usaremos el conjunto de datos `algae` donde los ejemplares son muestras de agua de un determinado rio, que nos ayudará a estimar, mediante la medición de determinados compuestos químicos y la proporción de determinados tipos de alga en el agua, la probabilidad de aparición de focos de crecimiento de algas dañinas para el rio.

Cargamos los datos. Para el problema de las algas hay tres ficheros: de aprendizaje, de test (solo con las variables de entrada) y el tercero con las soluciones del conjunto de test (las soluciones del conjunto de test). Los datos también se pueden cargar directamente de la librería `DMwR2`, como hemos indicado en la sección anterior, mediante el comando `data` .

```
# Si lo cargamos desde el fichero de datos 'algas_a.txt'
algae.df <- read.table("algas_a.txt", header=F, dec='.',
         col.names=c('season','size','speed','mxPH','mnO2','Cl','NO3',
         'NH4','oPO4','PO4','Chla','a1','a2','a3','a4','a5','a6','a7'),
         na.strings=c("XXXXXXX"))
# Se puede cargar desde la librería DMwR2
# data(algae) carga 3 data.frames: algae, test.algae y algae.sols. Solo usamos algae
data(algae,package="DMwR2")
head(algae)
```

Obsérvese que, dado que el cjto. tiene valores nulos y aparecen en el fichero `algas_a.txt` como una serie de caracteres 'X', hemos de indicarle a `read.table ()` que los nulos siguen esa nomenclatura y de esa forma R los tratará como nulos cuando los lea.

Si hemos cargado `algae` desde el fichero `algas_a.txt`, el comando `head(algae.df)` nos dará el siguiente resultado:

```
>head(algae.df)
  season  size  speed mxPH mnO2     C1    NO3     NH4    oPO4     PO4 Chla   a1   a2
1 winter small medium 8.00  9.8 60.800  6.238 578.000 105.000 170.000 50.0  0.0  0.0
2 spring small medium 8.35  8.0 57.750  1.288 370.000 428.750 558.750  1.3  1.4  7.6
3 autumn small medium 8.10 11.4 40.020  5.330 346.667 125.667 187.057 15.6  3.3 53.6
4 spring small medium 8.07  4.8 77.364  2.302  98.182  61.182 138.700  1.4  3.1 41.0
5 autumn small medium 8.06  9.0 55.350 10.416 233.700  58.222  97.580 10.5  9.2  2.9
6 winter small   high 8.25 13.1 65.750  9.248 430.000  18.250  56.667 28.4 15.1 14.6
    a3  a4   a5   a6  a7
1  0.0 0.0 34.2  8.3 0.0
2  4.8 1.9  6.7  0.0 2.1
3  1.9 0.0  0.0  0.0 9.7
4 18.9 0.0  1.4  0.0 1.4
5  7.5 0.0  7.5  4.1 1.0
6  1.4 0.0 22.5 12.6 2.9
```

En cambio, si hemos cargado `algae` desde la librería `DMwR2`, mediante el comando `data()`, el comando `head(algae)` tendrá el siguiente resultado:

```
>head(algae)
# A tibble: 6 x 18
  season   size  speed  mxPH  mnO2     Cl    NO3     NH4    oPO4     PO4  Chla    a1    a2
  <fctr> <fctr> <fctr> <dbl> <dbl>  <dbl>  <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl>
1 winter  small medium  8.00   9.8 60.800  6.238 578.000 105.000 170.000  50.0   0.0   0.0
2 spring  small medium  8.35   8.0 57.750  1.288 370.000 428.750 558.750   1.3   1.4   7.6
3 autumn  small medium  8.10  11.4 40.020  5.330 346.667 125.667 187.057  15.6   3.3  53.6
4 spring  small medium  8.07   4.8 77.364  2.302  98.182  61.182 138.700   1.4   3.1  41.0
5 autumn  small medium  8.06   9.0 55.350 10.416 233.700  58.222  97.580  10.5   9.2   2.9
6 winter  small   high  8.25  13.1 65.750  9.248 430.000  18.250  56.667  28.4  15.1  14.6
# ... with 5 more variables: a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>, a7 <dbl>
```

La salida de `head()` es diferente puesto que el objeto `algae` cargado desde `data()` no es un `data.frame`, sino que es un `tibble`. Un `tibble` es una versión más moderna de `data.frame` que trata de ser más cómoda y eficiente (mas información en `https://cran.r-project.org/web/packages/tibble/vignettes/tibble.html`). Luego veremos algunas particularidades de tibble a las que hay que poner atención.

Vamos a fijarnos ahora en el atributo que nos indica en máximo valor de ph medido en el agua,. Hacemos uso de la representación de histograma enriquecido que hemos usado antes:

```
# histograma enriquecido para mxPH
hist(algae$mxPH, xlab="",
    main="Máximo valor de PH", ylim=c(0,1),probability=T)
lines(density(algae$mxPH,na.rm=T))
rug(jitter(algae$mxPH))
```

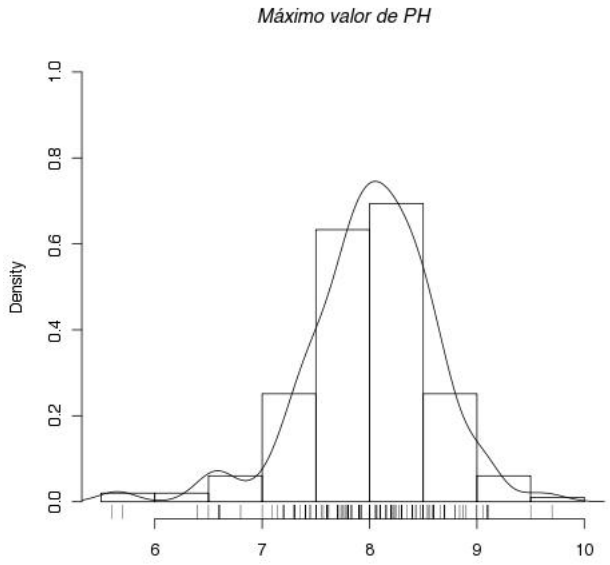


Figure 12: Histograma enriquecido para lecturas de PH máximo

El resultado es el que aparece en la figura 12. Si nos fijamos en los valores de este atributo de los distintos ejemplares podremos comprobar que hay valores que parecen *otuliers*, es decir, se perciben tanto valores excesivamente pequeños pequeños como otros valores muy grandes de ph. Vamos a asegurarnos con un Wisker plot. Utilicemos los comandos:

```
boxplot(algae$mxPH,boxwex=0.15,ylab="Max PH(oPO4)")
rug(jitter(algae$mxPH),side=2)
abline(h=mean(algae$mxPH,na.rm=T),lty=2)
```

De esta forma, además, representamos los ejemplares en el eje de ordenadas, le añadimos también la media mediante una línea horizontal punteada, y obtendremos la figura 13. Podemos observar que, efectivamente, hay potenciales *outliers* (usando el criterio de Tukey) tanto por arriba como por abajo.

Repetimos el whisker plot para otra variable, el ortofosfato, que nos va a mostrar una variable donde hay *outliers* solo por uno de los lados (lo que distorsionará la media).

```
boxplot(algae$oPO4,boxwex=0.15,ylab="Orthophosphate(oPO4)")
rug(jitter(algae$oPO4),side=2)
abline(h=mean(algae$oPO4,na.rm=T),lty=2)
```

Ahora, en la figura 14, podemos ver claramente que hay una gran cantidad de *outliers* con valores excesivamente grandes. Otro dato que nos apuntaría a esto es que la media está por encima de la mediana, indicada por la caja dispuesta verticalmente. El valor de la media está distorsionado debido a la presencia de esos valores. Si se van a usar procedimientos donde la media es la guía para tomar ciertas decisiones (lo

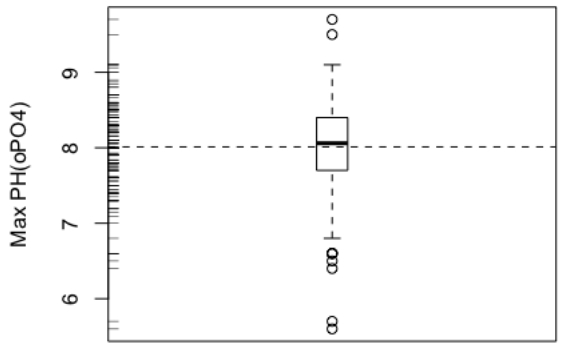


Figure 13: Whisker plot, media y jitter para el máximo PH

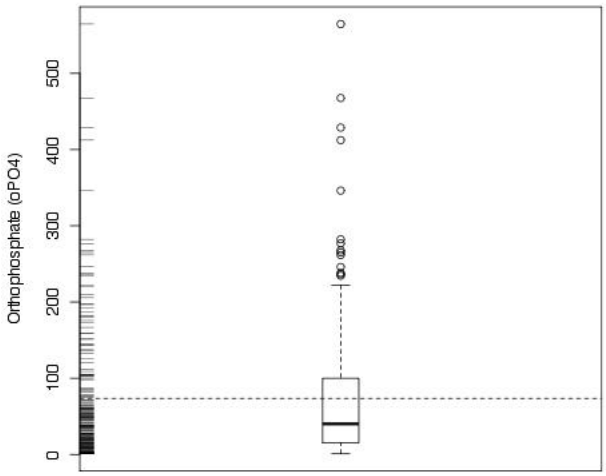


Figure 14: Whisker plot, media y jitter para el ortofosfato

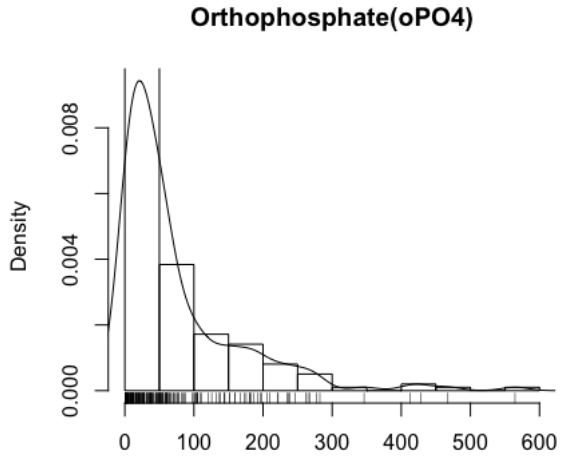


Figure 15: Histograma enriquecido para lecturas de ortofosfato

veremos más adelante, por ejemplo, con la técnica de reducción de dimensionalidad basada en PCA), este tipo de distorsiones puede tener efectos bastante perjudiciales y puede ser muy conveniente eliminar estos posibles *outliers*.

En el histograma también podemos ver que hay unos cuantos datos que se alejan del grueso de la distribución. Con los comandos siguientes obtenemos la figura 15:

```
# histograma enriquecido para oPO4
hist(algae$oPO4, xlab="",
     main="Orthophosphate(oPO4)",
     ylim=c(0,1.2*max(density(algae$oPO4,na.rm=T)$y)),probability=T)
lines(density(algae$oPO4,na.rm=T))
rug(jitter(algae$oPO4))
```

Si queremos identificar exactamente qué puntos cumplen la regla de superar en 3/2 la distancia intercuartil podemos usar el comando `boxplot()` accediendo a su componente `out`, es decir:

```
> boxplot(algae$oPO4,boxwex=0.15,ylab="Orthophosphate(oPO4)") $out
 [1] 428.750 564.600 467.500 246.000 264.900 276.850 412.333 282.167 267.750 261.600 238.200
[12] 234.500 236.400 346.167
```

que nos muestra los valores reales de los potenciales *outliers* de la muestra. A partir de ellos podemos encontrar los valores de corte para eliminar esos valores exageradamente grandes (o pequeños).

```
> extremos<-boxplot(algae$oPO4,boxwex=0.15,ylab="Orthophosphate(oPO4)")$out
> porArriba<-min(extremos[extremos > median(na.omit(algae$oPO4))])
```

```
> porAbajo<-max(extremos[extremos < median(na.omit(algae$oPO4))])
Warning message:
In max(extremos[extremos < median(na.omit(algae$oPO4))]) :
  ningun argumento finito para max; retornando -Inf
> porArriba
[1] 234.5
```

El *Warning* nos está indicando que no hay outliers por abajo. Ahora podemos ver los ejemplos completos indexándolos por esos valores de corte:

```
> na.omit(algae[algae$oPO4 >= porArriba,])
# A tibble: 14 x 18
   season   size  speed  mxPH  mnO2      Cl   NO3      NH4    oPO4     PO4   Chla    a1
   <fctr> <fctr> <fctr> <dbl> <dbl>   <dbl> <dbl>    <dbl>   <dbl>   <dbl>  <dbl> <dbl>
 1 spring  small medium  8.35   8.0  57.750 1.288  370.000 428.750 558.750  1.300   1.4
 2 spring  small medium  7.79   3.2  64.000 2.822 8777.600 564.600 771.600  4.500   0.0
 3 winter  small medium  7.83  10.7  88.000 4.825 1729.000 467.500 586.000 16.000   0.0
 4 winter  small   high  8.10  10.3  26.000 3.780   60.000 246.000 304.000  2.800   6.9
 5 winter  small   high  8.30   7.7  50.000 8.543   76.000 264.900 344.600 22.500   0.0
 6 spring  small   high  8.30   8.8  54.143 7.830   51.429 276.850 326.857 11.840   4.1
 7 winter medium medium  7.80   3.6  48.667 4.030 5738.330 412.333 607.167  4.300   0.0
 8 summer medium medium  7.60   9.7  53.102 7.160 4073.330 282.167 624.733  6.800   0.0
 9 spring medium medium  8.70   9.4 173.750 3.318  101.250 267.750 391.750  3.500   0.0
10 spring medium    low  8.40   5.3  74.667 3.900  131.667 261.600 432.909 24.917   1.9
11 summer medium    low  8.20   6.6 131.400 4.188   92.000 238.200 320.400  6.800   1.2
12 summer  large    low  7.60   4.9  69.000 3.685 1495.000 234.500 236.000 22.500  32.5
13 winter  large medium  8.24   6.1  95.367 3.561 1168.000 236.400 272.222 20.578   2.5
14 summer  large medium  7.91   6.2 151.833 3.923 1081.660 346.167 388.167  5.083   1.7
# ... with 6 more variables: a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>, a7 <dbl>
```

y tendremos todas aquellas muestras con un valor exageradamente grande (por arriba).

Si os fijáis en el número de fila que devuelve `na.omit()` veréis que la lista es consecutiva, lo cual parece extraño (ya sería casualidad que los primeros 14 ejemplos de la base de datos fuesen los que cumpliesen la condición). ¿Qué está pasando aquí? En realidad es un problema sutil que tiene que ver con que no estamos trabajando en este ejemplo con `data.frames` (que sí mostrarían el número del ejemplo de manera correcta) sino con `tibbles`. Los `tibbles` descartan los números (e incluso las etiquetas) de fila de los elementos de manera automática (y silente), y tampoco los indican cuando se accede a subconjuntos indexados. Si vamos a trabajar de manera que necesitamos identificar el número de fila tenemos antes que añadir una columna al `tibble` con esa información. Para ello usamos un comando especial de `tibble`, el comando `tibble::rowid_to_column()` (usa `help(rowid_to_column)` para obtener más información sobre tibbles y nombres/identificadores de filas).

Repitamos el comando, esta vez añadiéndole el número de fila como una nueva columna, `rowid`:

```
> na.omit(tibble::rowid_to_column(algae)[algae$oPO4 >= porArriba,])
# A tibble: 14 x 19
   rowid season   size  speed  mxPH  mnO2      Cl   NO3      NH4    oPO4     PO4   Chla
   <int> <fctr> <fctr> <fctr> <dbl> <dbl>   <dbl> <dbl>    <dbl>   <dbl>   <dbl>  <dbl>
```

```
 1     2 spring  small medium  8.35   8.0  57.750 1.288  370.000 428.750 558.750  1.300
 2    20 spring  small medium  7.79   3.2  64.000 2.822 8777.600 564.600 771.600  4.500
 3    21 winter  small medium  7.83  10.7  88.000 4.825 1729.000 467.500 586.000 16.000
 4    32 winter  small   high  8.10  10.3  26.000 3.780   60.000 246.000 304.000  2.800
 5    43 winter  small   high  8.30   7.7  50.000 8.543   76.000 264.900 344.600 22.500
 6    44 spring  small   high  8.30   8.8  54.143 7.830   51.429 276.850 326.857 11.840
 7    88 winter medium medium  7.80   3.6  48.667 4.030 5738.330 412.333 607.167  4.300
 8    89 summer medium medium  7.60   9.7  53.102 7.160 4073.330 282.167 624.733  6.800
 9    91 spring medium medium  8.70   9.4 173.750 3.318  101.250 267.750 391.750  3.500
10   119 spring medium    low  8.40   5.3  74.667 3.900  131.667 261.600 432.909 24.917
11   120 summer medium    low  8.20   6.6 131.400 4.188   92.000 238.200 320.400  6.800
12   157 summer  large    low  7.60   4.9  69.000 3.685 1495.000 234.500 236.000 22.500
13   171 winter  large medium  8.24   6.1  95.367 3.561 1168.000 236.400 272.222 20.578
14   172 summer  large medium  7.91   6.2 151.833 3.923 1081.660 346.167 388.167  5.083
# ... with 7 more variables: a1 <dbl>, a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>,
#   a7 <dbl>
```

Ahora disponemos de la información referente a qué ejemplo (fila) es el que cumple la condición. Si usasemos un data.frame nos lo daría sin tener que añadir la columna. P.e.:

```
> na.omit(algae.df[algae$oPO4 >= porArriba,])
    season   size  speed mxPH mnO2      Cl   NO3      NH4    oPO4     PO4   Chla   a1   a2
2   spring  small medium 8.35  8.0  57.750 1.288  370.000 428.750 558.750  1.300  1.4  7.6
20  spring  small medium 7.79  3.2  64.000 2.822 8777.600 564.600 771.600  4.500  0.0  0.0
21  winter  small medium 7.83 10.7  88.000 4.825 1729.000 467.500 586.000 16.000  0.0  0.0
32  winter  small   high 8.10 10.3  26.000 3.780   60.000 246.000 304.000  2.800  6.9 17.1
43  winter  small   high 8.30  7.7  50.000 8.543   76.000 264.900 344.600 22.500  0.0 40.9
44  spring  small   high 8.30  8.8  54.143 7.830   51.429 276.850 326.857 11.840  4.1  3.1
88  winter medium medium 7.80  3.6  48.667 4.030 5738.330 412.333 607.167  4.300  0.0  0.0
89  summer medium medium 7.60  9.7  53.102 7.160 4073.330 282.167 624.733  6.800  0.0  0.0
91  spring medium medium 8.70  9.4 173.750 3.318  101.250 267.750 391.750  3.500  0.0  5.5
119 spring medium    low 8.40  5.3  74.667 3.900  131.667 261.600 432.909 24.917  1.9 12.7
120 summer medium    low 8.20  6.6 131.400 4.188   92.000 238.200 320.400  6.800  1.2  1.9
157 summer  large    low 7.60  4.9  69.000 3.685 1495.000 234.500 236.000 22.500 32.5 12.0
171 winter  large medium 8.24  6.1  95.367 3.561 1168.000 236.400 272.222 20.578  2.5 13.2
172 summer  large medium 7.91  6.2 151.833 3.923 1081.660 346.167 388.167  5.083  1.7 12.0
      a3   a4   a5   a6  a7
2    4.8  1.9  6.7  0.0 2.1
20   0.0 44.6  0.0  0.0 1.4
21   0.0  6.8  6.1  0.0 0.0
32  20.2  0.0  4.0  0.0 2.9
43   7.5  0.0  2.4  1.5 0.0
44   0.0  0.0 19.7 17.0 0.0
88   2.6  2.4  5.0  0.0 2.4
89   0.0  1.0 35.6  9.9 0.0
```

```
91   3.3  0.0 20.8 12.4 0.0
119 25.9  0.0  0.0  0.0 6.8
120 22.9  0.0  8.1  0.0 0.0
157  0.0  5.0  0.0  0.0 1.9
171  0.0  2.0  7.4 17.2 0.0
172  4.9  2.7  0.0  5.9 1.7
```

Ejercicio

Repite el proceso para las otras variables numéricas. Te pueden ayudar diversos diagramas que permiten visualizar a la vez todas las variables numéricas con `lattice`. Por un lado los histogramas:

```
# Histograma enriquecido para las variables numéricas
# Fíjate en la diferencia si data=na.omit(algae)
histogram( ~ mxPH+mnO2+Cl+NO3+NH4+oPO4+PO4+Chla , data = algae,
           breaks=NULL,
           xlab = "Variable",
           type = "density",
           scales=  list(x=list(relation="free"),y=list(relation="free")),
           panel = function(x, ...) {
             panel.histogram(x, ...)
             panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
             panel.rug(x=jitter(x),col="black")
           })
```

Por otro lado los Box-Whisker:

```
# Box-Whisker plot de las variables numéricas
bwplot(value ~ variable  | variable,
       data = na.omit(melt(algae,measure.vars=c("mxPH","mnO2","Cl","NO3","NH4","PO4",
                                                "Chla"))),
       xlab = "Variable",
       scales=  list(x=list(relation="free",draw=F),y=list(relation="free")),
       panel = function(x,y, ...) {
         panel.bwplot(x,y, ...)
         panel.rug(y=jitter(y),col="black")
         panel.abline(h=mean(y,na.rm=T),lty=2,col="green")
       })
```

Ahora supongamos que queremos ahondar en la relación entre dos variables, en una primera aproximación al análisis multivariable. Veamos, por ejemplo, cómo se distribuyen las diferentes concentraciones de alga `a1` para los tres tipos (tamaños) de rios que hay en la base de datos.

```
bwplot(size ~ a1,data=algae,ylab="Tamaño del rio",xlab="Alga A1")
# con ggplot2
# ggplot(algae)+geom_boxplot(aes(size,a1))
# +coord_flip()
```

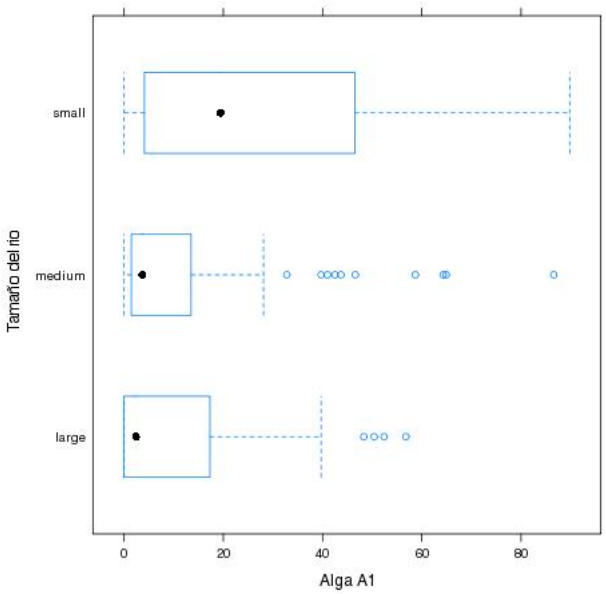


Figure 16: Concentraciones del alga a1, según el tamaño del rio

```
# +ylab("Alga A1")+xlab("Tamaño del rio")
```

y obtenemos el sencillo Whisker plot discriminado para cada uno de los valores, que aparece en la figura 16, con la que podemos deducir que, para el alga `a1`, tendremos altas concentraciones de la misma en rios pequeños. Lo cual puede ser bastante valioso como conocimiento en nuestro análisis. Obsérvese que hemos contrastado un valor discreto (i.e. un factor), con una variable real. Podemos hacerlo también, con este tipo de diagrama, usando dos variables reales, siempre que una de ellas la discreticemos.

Veamos cómo discretizar una variable. Por ejemplo, la variable `mnO2`, concentración mínima de oxígeno:

```
> minO2 <- equal.count(na.omit(algae$mnO2),number=4, overlap=1/5)
> minO2

Data:
  [1]  9.80  8.00 11.40  4.80  9.00 13.10 10.30 10.60  3.40  9.90 10.20 11.70
 [13]  9.60 11.80  9.60 11.50 12.00  9.80 10.40  3.20 10.70  9.20 10.30  8.50
 [25]  9.40 10.70  8.40 11.10  9.80 11.30 12.50 10.30 11.30  9.90  7.80  8.40
 ....

Intervals:
     min    max count
1  1.495  8.205    60
2  7.595  9.905    62
3  9.695 10.805    60
```

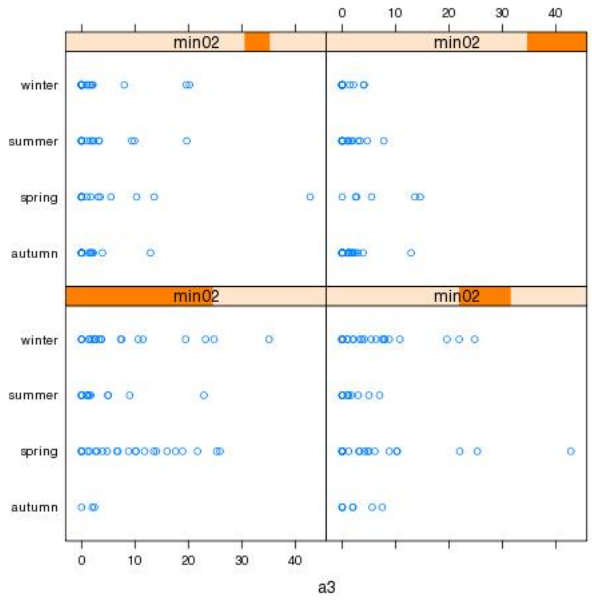


Figure 17: Concentraciones del alga a3, según temporada y niveles de O2 mínimos

```
4 10.695 13.405    61

Overlap between adjacent intervals:
[1] 14 16 15
```

Como vimos en una práctica anterior, se ha utilizando el comando `equal.count()` para crear cuatro bins (grupos) del atributo `mnO2`, al que previamente se le han eliminado los elementos nulos. El algoritmo utilizado ha sido el `equal.count` que tiene un valor de solapamiento de 1/5, lo cual significa que seguramente habrá valores que se dupliquen en intervalos contiguos (ya que los bins han de crearse de tal forma que tengan un número igual de ejemplares y no siempre puede hacerse). Si ahora representamos la variable discreta `season` con respecto a la concentración del tipo 3 de alga agrupándose respecto a estos grupos creados sobre los valores mínimos de O2, tenemos la figura 17.

```
stripplot(season ~ a3|minO2,data=algae[!is.na(algae$mnO2),],layout=c(2,2))
```

En la figura se ve, en la barra donde aparece la etiqueta minO2, un bloque de naranja más oscuro que está en distinta posición en cada uno de los 4 diagramas. Esos bloques indican los intervalos (dentro del rango de valores mínimo y máximo) de los grupos creados sobre la variable `mnO2`. De menor a mayor se muestran de izquierda a derecha y de abajo a arriba.

Si no hacemos nosotros mismos la discretización, el propio sistema lo hará con valores por defecto (con posibles efectos indeseados). Lo puedes comprobar usando:

```
stripplot(season ~ a3|mnO2,data=algae[!is.na(algae$mnO2),],layout=c(2,2))
```

Volviendo a la figura 17, se observa el valor de concentraciones del tipo de alga `a3` scon respecto a la variable `season`. Se muestra para cada bin diferente de la variable `minO2`. Repetimos, los diferentes intervalos dentro de valores mínimos de O_2 se reflejan en el bloque naranja oscuro de la barra etiquetada como min02. Las gráficas se disponen por defecto para los distintos bins desde izquierda a derecha y de abajo a arriba. Obsérvese el uso de la función `is.na(algae$mnO2)` para evitar que los valores nulos introdujeran ruido en el análisis.

Por último, como se mencionó antes, con `lattice` se pueden representar hasta 4 variables a la vez con una fórmula de la forma: `Var1 ~ Var2 | Var 3` y el parámetro `groups = Var4`. Eso haría un diagrama con `Var1` en el eje y, `Var2` en el eje x, tantos paneles como niveles tenga `Var3` si es un factor (o intervalos en forma de `shinge` si es numérica), y se pueden sobreimpresionar dentro de un mismo panel tantos grupos como niveles tenga `Var4` (que debe ser un factor) de una forma que se distingan entre dichos grupos. Obviamente no con todos los diagramas disponibles tiene sentido el usar las 4 variables puesto que alguno de los elementos se fija (p.e. en los density plots el eje y refleja la densidad y sería ignorada una variable `Var1`).

```
# Ejemplo de diagrama con 4 Variables, incluye una regresión
# mediante "smooth", que podría ser regresión lineal con "r"
# Ejemplo de diagrama con 4 Variables
xyplot(a3 ~ mxPH | season, groups=size,
       data=na.omit(algae),
       type=c("p","g","smooth"),
       lwd = 3,auto.key=T)
```

Este comando nos produciría el gráfico de la figura 18. Es importante no escalar por libre las variables de los ejes x e y puesto que, ya que se refieren a las mismas variables en todos los paneles, se pueden apreciar las diferencias entre ellas sin la distorsión que existiría si cada diagrama tuviese escalas diferentes (libres).

Por ultimo mencionar que, en realidad, se pueden incluir más paneles que los indicados por los niveles de `Var2` si dicho elemento de la fórmula se expande como `Var21+Var22+...+Var2n`. Conviene recalcar que, si se hace esto último, si tiene sentido que la escala del eje x se autoajuste en cada panel (esté "free"), puesto que en ese caso el eje x representa diferentes variables en varios paneles.

```
# Ejemplo de diagrama con 4+ Variables,
# Esta vez, al mezclar varias Var2i es mejor que el
# eje x autoajuste su escala
xyplot(a3 ~ mxPH+mnO2+NO3 | season, groups=size,
       data=na.omit(algae),
       type=c("p","g","smooth"),
       scales=list(x="free"),
       lwd = 3,auto.key=T)
```

6.2 Tratamiento de valores nulos

Ya hemos visto en clase cómo se pueden manejar los valores nulos. La forma más inmediata de tratarlos es deshacerse de ellos. Sin embargo, dichos nulos, o las tuplas que los incluyen, pueden tener información importante que, si los eliminamos sin más, se perdería. En realidad hay alternativas a la mera eliminación.

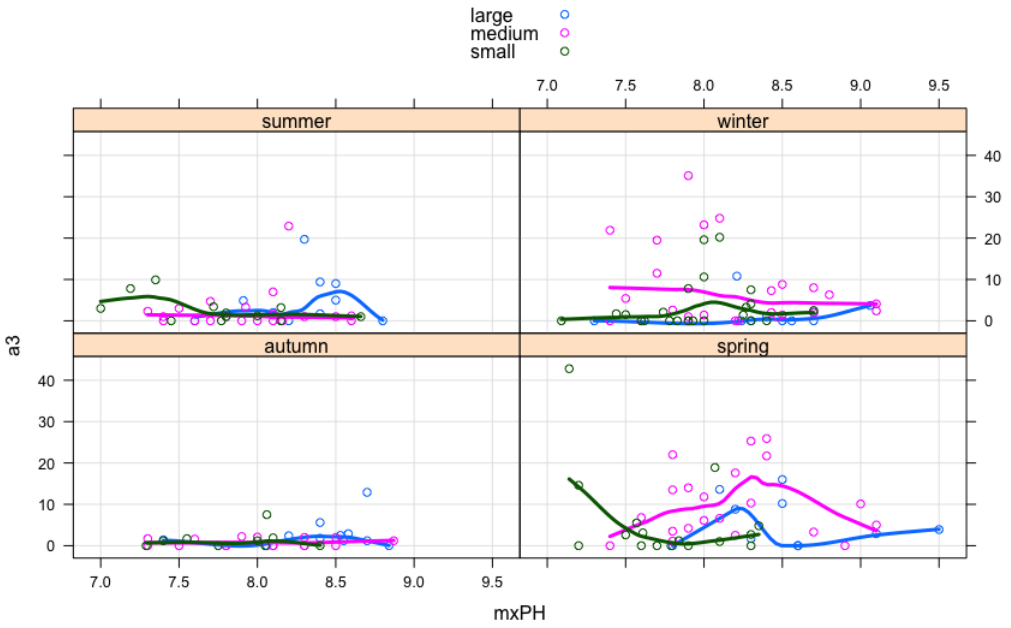


Figure 18: xplot mostrando dos variables agregadas por otras dos variables.

Así por ejemplo podemos sustituirlos con los valores más frecuentes, o también podemos hacer uso de información proveniente de correlaciones entre variables, (la modelamos, y reproducimos el nulo), o también podemos explorar la similitud entre diferentes casos, etc. Todos estos casos los vamos a ver en este apartado.

6.2.1 Eliminación de observaciones con nulos

Para tomar medida de cuán serio es el problema de los valores nulos, debemos contar los casos que tienen valores nulos, y lo hacemos con `complete.cases()`. Como queremos localizar qué ejemplos debemos tratar, y estamos usando tibbles tendremos que añadirle el número de fila como antes hemos aprendido.

```
> tibble::rowid_to_column(algae)[!complete.cases(algae),]
# A tibble: 16 x 19
   rowid season   size  speed  mxPH  mnO2    Cl   NO3   NH4    oPO4     PO4  Chla    a1
   <int> <fctr> <fctr> <fctr> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
 1    28 autumn  small   high  6.80  11.1 9.000 0.630    20   4.000      NA  2.70  30.3
 2    38 spring  small   high  8.00    NA 1.450 0.810    10   2.500   3.000  0.30  75.8
 3    48 winter  small    low    NA  12.6 9.000 0.230    10   5.000   6.000  1.10  35.5
 4    55 winter  small   high  6.60  10.8    NA 3.245    10   1.000   6.500    NA  24.3
 5    56 spring  small medium  5.60  11.8    NA 2.220     5   1.000   1.000    NA  82.7
 6    57 autumn  small medium  5.70  10.8    NA 2.550    10   1.000   4.000    NA  16.8
 7    58 spring  small   high  6.60   9.5    NA 1.320    20   1.000   6.000    NA  46.8
 8    59 summer  small   high  6.60  10.8    NA 2.640    10   2.000  11.000    NA  46.9
```

```
 9    60 autumn  small medium  6.60  11.3    NA 4.170    10   1.000   6.000    NA  47.1
10    61 spring  small medium  6.50  10.4    NA 5.970    10   2.000  14.000    NA  66.9
11    62 summer  small medium  6.40    NA    NA    NA    NA      NA  14.000    NA  19.4
12    63 autumn  small   high  7.83  11.7 4.083 1.328    18   3.333   6.667    NA  14.4
13   116 winter medium   high  9.70  10.8 0.222 0.406    10  22.444  10.111    NA  41.0
14   161 spring  large    low  9.00   5.8    NA 0.900   142 102.000 186.000 68.05   1.7
15   184 winter  large   high  8.00  10.9 9.055 0.825    40  21.083  56.091    NA  16.8
16   199 winter  large medium  8.00   7.6    NA    NA    NA      NA      NA    NA   0.0
# ... with 6 more variables: a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>, a7 <dbl>
> nrow(algae[!complete.cases(algae),])
[1] 16
```

La función `complete.cases()` devuelve un `TRUE` para cada fila con un caso completo (i.e. sin valores nulos). Si lo negamos, tendremos un `TRUE` para cada caso **con** nulos. Con el indexado lógico obtenemos aquellas filas con nulos. Finalmente, con `nrow()` contamos las filas. Tenemos un total de 16, de 200, un 8% de casos. Esto es un porcentaje importante que conviene tratar.

En el caso de que decidieramos deshacernos de los nulos, sería suficiente con hacer:

```
algae.noNA <- na.omit(algae)
```

Incluso si queremos conservar lo máximo posible es aconsejable descartar aquellos ejemplares que tengan valores nulos en muchas de las variables, algo que apuntaría a que son inservibles. Por ejemplo, las tuplas 62 y 199 tienen un total de seis valores nulos (tienen nulo en seis de las variables). Eliminar solo esos casos es tan sencillo como hacer:

```
algae.fix1 <- algae[-c(62,199),]
```

6.2.2 Sustitución mediante valores representativos

Es interesante, en determinados casos, utilizar los valores centrales de atributos como valores sustitutos de los nulos. Como ya sabemos por las clases de teoría, dependiendo de lo similar que la función de distribución de la probabilidad (pdf) de una muestra sea a la distribución normal, el valor de la media es una buena medida de su centralidad, y sería una buena candidata para ese valor sustituto. Si, por el contrario, la muestra tiene una distribución desplazada *skewed*, la mediana es mejor opción como valor central.

Por tanto, antes de tomar una decisión, habría que echar un vistazo a los datos, para determinar la pdf de la muestra. En este ejemplo concreto, el ejemplar 48 no tiene valor en la variable `mxPH`. Dado que dicha variable tiene una distribución parecida a la normal, hacemos:

```
# iremos copiando varios arreglos en diferentes variables para poder volver atrás
algae.fix2<-algae
algae.fix2[48,"mxPH"] <- mean(algae.fix2$mxPH,na.rm=T)
```

con la que el valor nulo de la celda correspondiente se sustituye por la media de la columna correspondiente (recordad usar el parámetro `na.rm=T` para calcular la media sin utilizar valores nulos).

Otras veces necesitaremos trabajar de manera intensiva con una columna (hacer varios cambios a la vez). Por tanto, en esos casos ,un acceso directo no va a ser la mejor manera de proceder. Podemos usar una indexación lógica (preguntamos directamente todos los casos que sean nulos y los sustituimos a la vez). Lo vamos a hacer para la variable `Chla`. Como esta variable tiene una distribución *skewed* utilizaremos mejor la mediana como valor central sustituto.

```
algae.fix2[is.na(algae.fix2$Chla),"Chla"] <- median(algae.fix2$Chla,na.rm=T)
```

Nótese que estos métodos mencionados por ahora de eliminación/corrección, en realidad, introducen un sesgo muy importante (i.e. todos se sustituyen por un valor más o menos representativo). Esto puede influenciar el análisis posterior (centra más aún las distribuciones de lo que realmente son). Son interesantes porque su coste computacional es muy bajo, sin embargo, métodos sin sesgo serían deseables cuando el tamaño del problema lo permita. Veamos alguno de ellos.

6.2.3 Sustitución mediante estudio de correlaciones

La idea es simple. Si uno de los atributos con valores nulos tiene una fuerte correlación con otro, podemos aprovechar ese hecho para así generar sustitutos para los valores nulos, mediante una técnica que introduce poco sesgo y sigue teniendo un bajo coste computacional.

Si recordamos el atributo `mxPH` y la muestra 48, si encontráramos un atributo altamente correlado con él, esa muestra, su valor nulo, podría ser sustituido sin problemas. Podemos probar a comprobar las correlaciones entre las variables con el comando `cor()`:

```
> algae.fix3<-algae
> cor(algae.fix3[,4:18], use="complete.obs")
            mxPH        mnO2          Cl         NO3         NH4         oPO4         PO4
mxPH  1.00000000 -0.10269374  0.14709539 -0.17213024 -0.15429757  0.090229085  0.10132957
mnO2 -0.10269374  1.00000000 -0.26324536  0.11790769 -0.07826816 -0.393752688 -0.46396073
Cl    0.14709539 -0.26324536  1.00000000  0.21095831  0.06598336  0.379255958  0.44519118
NO3  -0.17213024  0.11790769  0.21095831  1.00000000  0.72467766  0.133014517  0.15702971
NH4  -0.15429757 -0.07826816  0.06598336  0.72467766  1.00000000  0.219311206  0.19939575
oPO4  0.09022909 -0.39375269  0.37925596  0.13301452  0.21931121  1.000000000  0.91196460
PO4   0.10132957 -0.46396073  0.44519118  0.15702971  0.19939575  0.911964602  1.00000000
Chla  0.43182377 -0.13121671  0.14295776  0.14549290  0.09120406  0.106914784  0.24849223
a1   -0.16262986  0.24998372 -0.35923946 -0.24723921 -0.12360578 -0.394574479 -0.45816781
a2    0.33501740 -0.06848199  0.07845402  0.01997079 -0.03790296  0.123811068  0.13266789
a3   -0.02716034 -0.23522831  0.07653027 -0.09182236 -0.11290467  0.005704557  0.03219398
a4   -0.18435348 -0.37982999  0.14147281 -0.01448875  0.27452000  0.382481433  0.40883951
a5   -0.10731230  0.21001174  0.14534877  0.21213579  0.01544458  0.122027482  0.15548900
a6   -0.17273795  0.18862656  0.16904394  0.54404455  0.40119275  0.003340366  0.05320294
a7   -0.17027088 -0.10455106 -0.04494524  0.07505030 -0.02539279  0.026150420  0.07978353
             Chla          a1           a2           a3          a4          a5           a6
mxPH  0.43182377 -0.16262986  0.335017401 -0.027160336 -0.18435348 -0.10731230 -0.172737947
mnO2 -0.13121671  0.24998372 -0.068481989 -0.235228307 -0.37982999  0.21001174  0.188626555
Cl    0.14295776 -0.35923946  0.078454019  0.076530269  0.14147281  0.14534877  0.169043945
NO3   0.14549290 -0.24723921  0.019970786 -0.091822358 -0.01448875  0.21213579  0.544044553
NH4   0.09120406 -0.12360578 -0.037902958 -0.112904666  0.27452000  0.01544458  0.401192749
oPO4  0.10691478 -0.39457448  0.123811068  0.005704557  0.38248143  0.12202748  0.003340366
PO4   0.24849223 -0.45816781  0.132667891  0.032193981  0.40883951  0.15548900  0.053202942
Chla  1.00000000 -0.26601088  0.366724647 -0.063301128 -0.08600540 -0.07342837  0.010325497
a1   -0.26601088  1.00000000 -0.262665485 -0.108177581 -0.09338072 -0.26972709 -0.261564023
a2    0.36672465 -0.26266549  1.000000000  0.009759915 -0.17628704 -0.18675894 -0.133518480
```

```
a3   -0.06330113 -0.10817758  0.009759915  1.000000000  0.03336910 -0.14161095 -0.196900051
a4   -0.08600540 -0.09338072 -0.176287038  0.033369102  1.00000000 -0.10131827 -0.084884259
a5   -0.07342837 -0.26972709 -0.186758940 -0.141610948 -0.10131827  1.00000000  0.388608955
a6    0.01032550 -0.26156402 -0.133518480 -0.196900051 -0.08488426  0.38860896  1.000000000
a7    0.01760782 -0.19306384  0.036206205  0.039060248  0.07114638 -0.05149346 -0.030334277
              a7
mxPH -0.17027088
mnO2 -0.10455106
Cl   -0.04494524
NO3   0.07505030
NH4  -0.02539279
oPO4  0.02615042
PO4   0.07978353
Chla  0.01760782
a1   -0.19306384
a2    0.03620621
a3    0.03906025
a4    0.07114638
a5   -0.05149346
a6   -0.03033428
a7    1.00000000
```

pero podemos comprobar que, al ser tantos atributos, la matriz no es demasiado informativa. Podríamos buscar un atributo realmente correlado de una forma más cómodo y visual usando la siguiente forma:

```
> symnum(cor(algae.fix3[,4:18],use="complete.obs"))
     mP mO Cl NO NH o P Ch a1 a2 a3 a4 a5 a6 a7
mxPH 1
mnO2    1
Cl         1
NO3           1
NH4           ,  1
oPO4    .  .        1
PO4     .  .        * 1
Chla .                  1
a1          .        . .    1
a2   .                   .     1
a3                                  1
a4      .           . .             1
a5                                     1
a6            .  .                     .  1
a7                                           1
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
```

El comando `symnum()` nos ofrece una representación simbólica de esta matriz. Si nos fijamos en la leyenda que se nos proporciona, y buscamos marcas como comas o asteriscos (no hay signo más o letra B),

encontramos que las parejas (NH4, NO) y (PO4, oPO4) están correladas. Nos quedamos con la correlación más fuerte, el par (PO4, oPO4). Esta vez queremos que, al imprimir el tibble, nos imprima todas las columnas (no solo las que caben en la pantalla, que es el método de imprimirlas por defecto). Queremos ver todas las columnas para ver si hay ejemplos particulares que tienen muchos valores nulos (lo que haría que no mereciese mucho la pena arreglarlos ya que apenas contendrían información). Para mostrar todas las columnas usaremos el comando print.data.frame(). Como queremos saber el número de ejemplo de nuevo usaremos rowid_to_column(), aunque esta vez lo añadimos una vez al tibble y trabajamos con la tabla que ya incluya esa información (también se podría convertir el tibble a data.frame y trabajar con el tipo de dato antiguo):

```
> algae.fix3<-tibble::rowid_to_column(algae.fix3)
> print.data.frame(algae.fix3[is.na(algae.fix3$PO4),])
  rowid season  size  speed mxPH mnO2 Cl  NO3 NH4 oPO4 PO4 Chla   a1   a2  a3 a4  a5  a6
1    28 autumn small   high  6.8 11.1  9 0.63  20    4  NA  2.7 30.3  1.9 0.0  0 2.1 1.4
2   199 winter large medium  8.0  7.6 NA   NA  NA   NA  NA   NA  0.0 12.5 3.7  1 0.0 0.0
   a7
1 2.1
2 4.9
> print.data.frame(algae.fix3[is.na(algae.fix3$oPO4),])
  rowid season  size  speed mxPH mnO2 Cl NO3 NH4 oPO4 PO4 Chla   a1   a2  a3 a4 a5  a6  a7
1    62 summer small medium  6.4   NA NA  NA  NA   NA  14   NA 19.4  0.0 0.0  2  0 3.9 1.7
2   199 winter large medium  8.0  7.6 NA  NA  NA   NA  NA   NA  0.0 12.5 3.7  1  0 0.0 4.9
```

Ahora se puede apreciar qué ejemplos podemos reparar (la correlación es bidireccional, podemos arreglar las de uno con el otro y las del otro con el uno). Vemos que hay un par de ejemplos con un valor nulo en PO4 y otro par con valor nulo en oPO4. Vemos que solamente la instancia 28 (el número de instancia está en rowid) es subsceptible de areglo ya que, para ambas variables, el resto de instancias con nulos tienen demasiados nulos en otras variables y hay que eliminarlas. Así que podremos arreglar el atributo PO4 de la instancia 28.

Podemos encontrar un modelo de correlación lineal entre las dos variables de una forma extremadamente fácil, con

```
> lm(PO4 ~ oPO4,data=algae.fix3)

Call:
lm(formula = PO4 ~ oPO4, data = algae.fix3)

Coefficients:
(Intercept)         oPO4
     42.752        1.294
```

Lo que estamos haciendo es pedirle a R que genere un modelo lineal con lm(), en el que se aproxime PO4 con oPO4, utilizando los datos fuente. Así, el modelo lineal que se obtiene es el siguiente

$$PO4 = 1.294 \times oPO4 + 42.752$$

Por tanto, solo nos resta aplicar dicho modelo a la sustitución del valor nulo en el ejemplar 28, de la siguiente forma:

```
algae.fix3[algae.fix3$rowid==28,'PO4'] <- algae.fix3[algae.fix3$rowid==28,'oPO4']*1.294 + 42.752
```

aunque para hacerlo genérico podríamos automatizarlo crenado una función y luego aplicando `sapply()`:

```
# Función que arregla PO4 en función de oPO4 (si oPO4 no es NA, por supuesto)
fixPO4 <- function(oPO4) {
  if (is.na(oPO4))
    return(NA)
  else return(42.897 + 1.293 * oPO4)
}
# Arregla todos los NA de PO4
algae.fix3[is.na(algae.fix3$PO4), "PO4"] <-
        sapply(algae.fix3[is.na(algae.fix3$PO4),"oPO4"], fixPO4)
```

6.2.4 Sustitución de variables numéricas mediante preProcess de caret (clustering)

La biblioteca caret proporciona varios mecanismos sencillos para reemplazar los valores `NA` utilizando diversos algoritmos,como por ejemplo el algoritmo de clustering `Knn` (K-vecinos), o la sustitución por la mediana arriba mencionado. Eso sí, solo funcionan con variables numéricas. A continuación veremos como hacerlo.

Lo primero que tenemos que asegurarnos es que no existe ningún `NA` en las variables de salida. Si el valor missing de un ejemplo es la clase entonces el valor a inferir no es "arreglable" porque no tiene sentido. Al fin y al cabo pretendemos hacer precisamente un modelo para hallar dicha clase o valor, y no se puede usar un ejemplo para entrenarlo si desconocemos precisamente el valor de dicho ejemplo. En el problema de las algas las variables de salida son las $\mathtt{a}_i$ y el comando que lo comprobaría sería:

```
> sum(is.na(algae[,c("a1","a2","a3","a4","a5","a6","a7")]))
[1] 0
```

Veamos también el número de `NA` en el resto de variables (aunque se puede hacer con todo el frame puesto que ya sabemos que no los hay en las variables de salida). Por cierto, cuidado si indexas las columnas por número de posición y le has añadido temporalmente una columna al tibble para saber el número de fila (no es el caso ahora, porque volvemos a algae) porque te la añade al principio y cambiará la posición en número de todas las columnas. En muchos casos es más seguro indexarlas por nombre/etiqueta, o bien tener mucho cuidado.

```
> sum(is.na(algae[,1:11]))
[1] 33
> sum(is.na(algae))
[1] 33
```

Ahora utilizaremos el comando `preProcess()` de caret para generar un objeto que nos permite modificar los datos para facilitar el entrenamiento de modelos. Cuando aprendamos a usar caret veremos que este paso lo podemos incluir en el entrenamiento. Veamos ahora los comandos para asignar los valores nulos o *missing* usando *Knn*.

```
# Aviso: knnImpute fuerza automáticamente "center" y "scale" (normaliza),
# mientras que "bagimpute" o "medianImpute" NO normaliza automáticamente
ppKnn<-preProcess(algae,
           method = c("knnImpute"),   # o "bagImpute" / "medianImpute"
# equivale a method=c("knnImpute","center","scale"),
           na.remove = TRUE,
           k = 5,
           knnSummary = mean,
           outcome = NULL,
           fudge = .2,
           numUnique = 3,
           verbose = TRUE,
)
ppbag<-preProcess(algae,
           method = c("bagImpute"),
           na.remove = TRUE,
           k = 5,
           knnSummary = mean,
           outcome = NULL,
           fudge = .2,
           numUnique = 3,
           verbose = TRUE,
)
ppmedian<-preProcess(algae,
           method = c("medianImpute"),
           na.remove = TRUE,
           k = 5,
           knnSummary = mean,
           outcome = NULL,
           fudge = .2,
           numUnique = 3,
           verbose = TRUE,
)
```

Estos preprocesos todavía no se han aplicado a los datos, han construido un objeto que nos lo permite. Para aplicar finalmente los cambios hay que acudir a la función `predict()` y ejecutamos los siguientes comandos:

```
algae.fix4Knn<-predict(ppKnn,algae)
algae.fix4bag<-predict(ppbag,algae)
algae.fix4median<-predict(ppmedian,algae)
```

Finalmente vamos a comprobar los cambios que dicho preproceso ha hecho sobre los datos y será fácil ver que el método `"KnnImpute"` fuerza que todos los datos sean normalizados automáticamente a una distribución gausian normal. Es decir, que se transforman adicionalmente todas las variables numéricas con los métodos "center" y "scale". El primero le resta la media, y "scale" divide por la desviación estándar, es decir, que

ambas normalizan a una distribución gausiana. Si lo que se busca es una normalización al rango $[0, 1]$ (restar el valor mínimo y dividir por $max - min$) el método de preproceso a usar es "range".

Los métodos `"bagImpute", "medianImpute"` dejan el resto de datos como estaban. Es importante saber si los datos originales han sido "transformados" con normalizaciones o usando otros métodos como `PCA` (que ahora veremos como usarlo) porque el modelo se entrenará sobre los datos transformados y no sobre los originales (se hace un ajuste del modelo sobre el sistema "transformado" y no sobre el original). Eso implica que cuando se vayan a inferir nuevos ejemplos no vistos, estos ejemplos estarán en el sistema de referencia original y deben sufrir la misma transformación antes de pasarselos al modelo. Por fortuna caret se encarga de ello automáticamente si el preproceso se incluye en el entrenamiento del modelo (el modelo de caret contiene información sobre las transformaciones y las aplica cuando es necesario). Eso sí, deben llamarse correctamente los comandos `train()` y `predict()` sobre los modelos entrenados y no usar, por ejemplo, elementos de dichos modelos (como el tentador "finalModel", que contiene el modelo final, pero sobre los datos "transformados" y al que, si se usa directamente para predecir, caret no le preprocesará antes los datos de entrada y dará una respuesta inadecuada, pues). Ya lo comentaremos más adelante.

Por lo general las transformaciones (y "retoques") solo se realizan sobre las variables de entrada. Transformar la variable de salida solo tiene sentido por motivos de hacerla más comprensible (o algunos casos raros donde la escala puede dar errores de representación de los valores por la precisión de la representación) pero nunca porque el ajuste de los parámetros modelo lo necesite.

Veamos pues las diferencias entre los distintos algoritmos para imputar los valores desconocidos. Solo mostraremos, obviamente, aquellos casos donde había `NA`s. Como nos interesa conocer el número de ejemplo le aplicaremos `tibble:rowid_to_column()` a los tibbles antes de indexarlos.

```
tibble::rowid_to_column(algae)[!complete.cases(algae),]
tibble::rowid_to_column(algae.fix4Knn)[!complete.cases(algae),]
tibble::rowid_to_column(algae.fix4bag)[!complete.cases(algae),]
tibble::rowid_to_column(algae.fix4median)[!complete.cases(algae),]
```

El resultado obtenido por esos comandos es:

```
> tibble::rowid_to_column(algae)[!complete.cases(algae),]
# A tibble: 16 x 19
   rowid season   size  speed  mxPH  mnO2    Cl   NO3   NH4    oPO4     PO4  Chla    a1
   <int> <fctr> <fctr> <fctr> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
 1    28 autumn  small   high  6.80  11.1 9.000 0.630    20   4.000      NA  2.70  30.3
 2    38 spring  small   high  8.00    NA 1.450 0.810    10   2.500   3.000  0.30  75.8
 3    48 winter  small    low    NA  12.6 9.000 0.230    10   5.000   6.000  1.10  35.5
 4    55 winter  small   high  6.60  10.8    NA 3.245    10   1.000   6.500    NA  24.3
 5    56 spring  small medium  5.60  11.8    NA 2.220     5   1.000   1.000    NA  82.7
 6    57 autumn  small medium  5.70  10.8    NA 2.550    10   1.000   4.000    NA  16.8
 7    58 spring  small   high  6.60   9.5    NA 1.320    20   1.000   6.000    NA  46.8
 8    59 summer  small   high  6.60  10.8    NA 2.640    10   2.000  11.000    NA  46.9
 9    60 autumn  small medium  6.60  11.3    NA 4.170    10   1.000   6.000    NA  47.1
10    61 spring  small medium  6.50  10.4    NA 5.970    10   2.000  14.000    NA  66.9
11    62 summer  small medium  6.40    NA    NA    NA    NA      NA  14.000    NA  19.4
12    63 autumn  small   high  7.83  11.7 4.083 1.328    18   3.333   6.667    NA  14.4
13   116 winter medium   high  9.70  10.8 0.222 0.406    10  22.444  10.111    NA  41.0
```

```
14   161 spring  large    low  9.00   5.8    NA 0.900   142 102.000 186.000 68.05   1.7
15   184 winter  large   high  8.00  10.9 9.055 0.825    40  21.083  56.091    NA  16.8
16   199 winter  large medium  8.00   7.6    NA    NA    NA      NA      NA    NA   0.0
# ... with 6 more variables: a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>, a7 <dbl>
> tibble::rowid_to_column(algae.fix4Knn)[!complete.cases(algae),]
# A tibble: 16 x 19
   rowid season   size  speed       mxPH       mnO2         Cl          NO3        NH4
   <int> <fctr> <fctr> <fctr>      <dbl>      <dbl>      <dbl>        <dbl>      <dbl>
 1    28 autumn  small   high -2.0252762  0.8289472 -0.7395966 -0.702345318 -0.2452406
 2    38 spring  small   high -0.0196115  0.6616709 -0.9008135 -0.654681811 -0.2503360
 3    48 winter  small    low  0.1575555  1.4562335 -0.7395966 -0.808264224 -0.2503360
 4    55 winter  small   high -2.3595537  0.7034899 -0.8083925 -0.009900475 -0.2503360
 5    56 spring  small medium -4.0309410  1.1216808 -0.7710456 -0.281317670 -0.2528837
 6    57 autumn  small medium -3.8638022  0.7034899 -0.6801236 -0.193934573 -0.2503360
 7    58 spring  small   high -2.3595537  0.1598418 -0.5242407 -0.519635207 -0.2452406
 8    59 summer  small   high -2.3595537  0.7034899 -0.7941200 -0.170102820 -0.2503360
 9    60 autumn  small medium -2.3595537  0.9125854 -0.7940089  0.235036993 -0.2503360
10    61 spring  small medium -2.5266924  0.5362136 -0.7710456  0.711672066 -0.2503360
11    62 summer  small medium -2.6938311  0.4358478 -0.7809535 -0.418429693 -0.2001533
12    63 autumn  small   high -0.3037473  1.0798617 -0.8445905 -0.517516829 -0.2462597
13   116 winter medium   high  2.8217469  0.7034899 -0.9270353 -0.761659905 -0.2503360
14   161 spring  large    low  1.6517758 -1.3874643  0.3009764 -0.630850057 -0.1830764
15   184 winter  large   high -0.0196115  0.7453090 -0.7384222 -0.650709852 -0.2350498
16   199 winter  large medium -0.0196115 -0.6347208  0.6633878 -0.031984567 -0.1129330
# ... with 10 more variables: oPO4 <dbl>, PO4 <dbl>, Chla <dbl>, a1 <dbl>, a2 <dbl>,
#   a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>, a7 <dbl>
> tibble::rowid_to_column(algae.fix4bag)[!complete.cases(algae),]
# A tibble: 16 x 19
   rowid season   size  speed     mxPH     mnO2        Cl      NO3      NH4       oPO4
   <int> <fctr> <fctr> <fctr>    <dbl>    <dbl>     <dbl>    <dbl>    <dbl>      <dbl>
 1    28 autumn  small   high 6.800000 11.10000  9.000000 0.630000  20.0000   4.00000
 2    38 spring  small   high 8.000000 10.64158  1.450000 0.810000  10.0000   2.50000
 3    48 winter  small    low 7.809656 12.60000  9.000000 0.230000  10.0000   5.00000
 4    55 winter  small   high 6.600000 10.80000 12.477823 3.245000  10.0000   1.00000
 5    56 spring  small medium 5.600000 11.80000 12.477823 2.220000   5.0000   1.00000
 6    57 autumn  small medium 5.700000 10.80000 15.376901 2.550000  10.0000   1.00000
 7    58 spring  small   high 6.600000  9.50000  9.386827 1.320000  20.0000   1.00000
 8    59 summer  small   high 6.600000 10.80000 15.376901 2.640000  10.0000   2.00000
 9    60 autumn  small medium 6.600000 11.30000 15.141373 4.170000  10.0000   1.00000
10    61 spring  small medium 6.500000 10.40000 41.654308 5.970000  10.0000   2.00000
11    62 summer  small medium 6.400000 10.49779  9.386827 1.864939 194.3916  14.24556
12    63 autumn  small   high 7.830000 11.70000  4.083000 1.328000  18.0000   3.33300
13   116 winter medium   high 9.700000 10.80000  0.222000 0.406000  10.0000  22.44400
14   161 spring  large    low 9.000000  5.80000 85.058944 0.900000 142.0000 102.00000
15   184 winter  large   high 8.000000 10.90000  9.055000 0.825000  40.0000  21.08300
16   199 winter  large medium 8.000000  7.60000 84.765117 4.182252 234.5670  75.09208
```

```
# ... with 9 more variables: PO4 <dbl>, Chla <dbl>, a1 <dbl>, a2 <dbl>, a3 <dbl>, a4 <dbl>,
#   a5 <dbl>, a6 <dbl>, a7 <dbl>
> tibble::rowid_to_column(algae.fix4median)[!complete.cases(algae),]
# A tibble: 16 x 19
   rowid season   size  speed  mxPH  mnO2     Cl   NO3      NH4    oPO4      PO4   Chla
   <int> <fctr> <fctr> <fctr> <dbl> <dbl>  <dbl> <dbl>    <dbl>   <dbl>    <dbl>  <dbl>
 1    28 autumn  small   high  6.80  11.1  9.000 0.630  20.0000   4.000 103.2855  2.700
 2    38 spring  small   high  8.00   9.8  1.450 0.810  10.0000   2.500   3.0000  0.300
 3    48 winter  small    low  8.06  12.6  9.000 0.230  10.0000   5.000   6.0000  1.100
 4    55 winter  small   high  6.60  10.8 32.730 3.245  10.0000   1.000   6.5000  5.475
 5    56 spring  small medium  5.60  11.8 32.730 2.220   5.0000   1.000   1.0000  5.475
 6    57 autumn  small medium  5.70  10.8 32.730 2.550  10.0000   1.000   4.0000  5.475
 7    58 spring  small   high  6.60   9.5 32.730 1.320  20.0000   1.000   6.0000  5.475
 8    59 summer  small   high  6.60  10.8 32.730 2.640  10.0000   2.000  11.0000  5.475
 9    60 autumn  small medium  6.60  11.3 32.730 4.170  10.0000   1.000   6.0000  5.475
10    61 spring  small medium  6.50  10.4 32.730 5.970  10.0000   2.000  14.0000  5.475
11    62 summer  small medium  6.40   9.8 32.730 2.675 103.1665  40.150  14.0000  5.475
12    63 autumn  small   high  7.83  11.7  4.083 1.328  18.0000   3.333   6.6670  5.475
13   116 winter medium   high  9.70  10.8  0.222 0.406  10.0000  22.444  10.1110  5.475
14   161 spring  large    low  9.00   5.8 32.730 0.900 142.0000 102.000 186.0000 68.050
15   184 winter  large   high  8.00  10.9  9.055 0.825  40.0000  21.083  56.0910  5.475
16   199 winter  large medium  8.00   7.6 32.730 2.675 103.1665  40.150 103.2855  5.475
# ... with 7 more variables: a1 <dbl>, a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>,
#   a7 <dbl>
```

Como hemos adelantado, el `"imputeKnn"` transforma todas las variables numéricas con `"center"` y `"scale"` mientras que los dos otros métodos las dejan como estaban. También comprobamos que cada método reemplaza los `NA` por diversos valores. A pesar de no tener la seguridad de cual de ellos es mejor es siempre buena idea comprobar si arreglándolos se obtienen mejores modelos y, desde luego, es una necesidad para utilizar aquellos modelos que no permiten tener valores desconocidos en los datos de entrada.

Ejercicio

Utilizar algún método de los anteriores (que no sea eliminarlos) para "arreglar" los `NA` del conjunto `BreastCancer`.

Ejercicio

Intentar reproducir el análisis simple que hemos realizado en clase para los conjuntos de datos `wine.data`, `waveform.data`, `covtype.data`.

Con esto acabamos la segunda sesión. En la próxima empezaremos con la librería Caret propiamente dicha.

---

7 Introducción a Caret

Esta práctica la vamos a dedicar al uso de la librería caret para hacer modelos de *machine learning* (aprendizaje computacional). La librería CARET (**C**l**A**ssification and **RE**gression **T**raining) es una librería que proporciona un interfaz común a cientos de modelos de machine learning disponibles en R. Es importante fijarse que, es un **interfaz** y que, por debajo, están los algoritmos implementados por diversos autores(¡y que tienen muchas más peculiaridades de las que el interfaz de alto nivel que es Caret suele proporcionar!). Es muy habitual que las técnicas y modelos que ofrece caret tengan muchos más hiper-parámetros que los que refleja dicho interfaz a alto nivel. Con esto quiero hacer hincapié en que, para hacer modelos profesionales, es muy probable que haya que meterse a bajo nivel y retocar o trabajar con los hiper-parámetros que caret no ofrece directamente. Por fortuna se puede hacer esto mientras seguimos usando caret ya que la librería permite pasar dichos hiper-parámetros en las funciones que proporciona, aunque para conocer dichos hiper-parámetros se debe acudir a la documentación original del modelo de machine learning que encapsula.

Caret, además, proporciona y estandariza muchas otras tareas asociadas a machine learning, como el preprocesado de los datos, diversas técnicas de selección de variables, formas de dividir los conjuntos de datos, estimaciones de la importancia de los atributos, y diagramas para visualizar los resultados de los modelos obtenidos.

Como se ha dicho Caret se desarrolló para proporcinar un interfaz unificado para el modelado y predicción. En Octubre de 2018 el interfaz está disponible para 237 modelos diferentes. Caret simplifica el ajuste de modelos usando remuestreo (resampling) y proporciona diversas funciones y clases "helper" para facilitar las tareas de creación de modelos de machine learning. Es capaz, también, de incrementar la eficiencia computacional puesto que utiliza procesamiento en paralelo. Es, sin duda, una de las librerías más utiles de R para hacer machine learning.

Aunque ya la tendrás instalada si has seguido la primera parte del tutorial, volvemos a comprobar su instalación y la cargaremos en el intérprete R con los siguientes comandos:

```
install.packages("caret", dependencies=c("Depends", "Suggests"))
library(caret)
```

Una vez hecho esto ya podremos empezar a trabajar con esta fantástica librería. Si no las tienes ya cargadas volvemos a cargar todas las bases de datos que usamos en el tutorial:

```
library(mlbench)
install.packages("DMwR2")      # Contiene el dataset de las "algas"

data(iris)
data(BreastCancer)
data(PimaIndiansDiabetes)
data(Glass)
data(BostonHousing2)
data(algae,package="DMwR2") # otra forma de cargar los datos sin cargar la librería
# Base de datos adult, necesita parámetros específicos para cargar bien
adult<- read.table(
         "http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
         sep=",",header=F,col.names=c("age", "type_employer", "fnlwgt", "education",
```

```
        "education_num","marital", "occupation", "relationship", "race","sex",
        "capital_gain", "capital_loss", "hr_per_week","country", "income"),
        fill=FALSE,strip.white=T,na.strings = c("?"))

# Base de datos white wine. Está en formato csv
wine<- read.csv2(
  paste("https://archive.ics.uci.edu/ml/machine-learning-databases/",
         "wine-quality/winequality-white.csv",
        sep="")
  ,dec="."
)
wine$class<-factor(wine$quality) # La clase como un factor
# Las etiquetas de los niveles no deben ser números
levels(wine$class)<-make.names(levels(wine$class))
```

8 Dividiendo datos en Entrenamiento/Test

Como se ha mencionado en clase los datos de los que se dispone se deben dividir en dos grupos. Uno para entrenar el sistema y otro grupo para hacer una estimación del rendimiento de los modelos entrenados con datos nuevos.

Es importante que el conjunto de test (o de publicación) se mantenga apartado y no se utilice la información que contiene para tomar decisiones sobre el entrenamiento puesto que hacerlo implica introducir sesgos en dicho entrenamiento. Así que lo primero que debemos hacer es dividir los datos y dejar de un lado los datos de test.

Caret proporciona varias formas de dividir los datos en dichos conjuntos de entrenamiento y de test. Recordemos que luego el conjunto de entrenamiento será utilizado tanto para obtener los hiperparámetros de los modelos mediante remuestreo como para el ajuste de un modelo final una vez hallados los mejores hiperparámetros.

La forma más habitual y fácil es usar el comando createDataPartition(), que nos permite hacer una partición aleatoria estratificada (que preserva las proporciones de las clases o la distribución de los valores del conjunto original en los subconjuntos obtenidos). Es importante recordar que, a partir de ahora, como se empiezan a utilizar elementos aleatorios, es necesario controlar la semilla del generador de números pseudo-aleatorios para poder reproducir los resultados exactos en posteriores ejecuciones, algo muy importante para depurar errores.

Hagamos una partición para el problema adult, el comando `createDataPartition()` necesita la variable que se usará de salida como parámetro para poder hacer la estratificación, también el tanto por uno de datos que contendrá la partición. Por defecto devuelve una lista pero es más cómodo tener un vector (para así dividir el data.frame con facilidad indexándo por dicho vector):

```
# Ponemos la semilla para números aleatorios
set.seed(1234)
adult.Datos.Todo<-adult
adult.Var.Salida.Usada<-c("income")
# Índices para el cjto de entrenamiento (80% dataset "adult",variable de salida "income")
```

```
adult.trainIdx<- createDataPartition(adult.Datos.Todo[[adult.Var.Salida.Usada]],
                                     p=0.8,
                                     list = FALSE,
                                     times = 1)
# Creamos los subconjuntos en base a trainIdx
adult.Datos.Train<-adult.Datos.Todo[adult.trainIdx,]
adult.Datos.Test<-adult.Datos.Todo[-adult.trainIdx,]
```

Y podemos comprobar que se mantienen fielmente las proporciones de las clases de salida en el conjunto original y los dos subconjuntos de entrenamiento y test:

```
> prop.table(table(adult.Datos.Todo$income))*100

   <=50K     >50K
75.91904 24.08096
> prop.table(table(adult.Datos.Train$income))*100

   <=50K     >50K
75.91846 24.08154
> prop.table(table(adult.Datos.Test$income))*100

   <=50K     >50K
75.92138 24.07862
```

Si lo hacemos sobre un problema de regresión vemos que las densidades se mantienen (esta vez usamos un 70%-30% sobre el problema `algae` para el primer tipo de alga):

```
# Ponemos la semilla para números aleatorios
set.seed(1234)
algae.Datos.Todo<-algae
algae.VarSalida.Usada<-c("a1")
# Índices para el cjto de entrenamiento (70% dataset "algae",variable de salida "a1")
algae.trainIdx<- createDataPartition(algae.Datos.Todo[[algae.VarSalida.Usada]],
                                     p=0.7,
                                     list = FALSE,
                                     times = 1)
# Creamos los subconjuntos en base a trainIdx
algae.Datos.Train<-algae.Datos.Todo[algae.trainIdx,]
algae.Datos.Test<-algae.Datos.Todo[-algae.trainIdx,]

# Creamos datos para mostrar en un diagrama las densidades de los 3 cjtos
toplot<-data.frame(dataset="Original",Var=algae.Datos.Todo[[algae.VarSalida.Usada]])
toplot<-rbind(toplot,data.frame(dataset="Train",
                                Var=algae.Datos.Train[[algae.VarSalida.Usada]]))
toplot<-rbind(toplot,data.frame(dataset="Test",
                                Var=algae.Datos.Test[[algae.VarSalida.Usada]]))
```

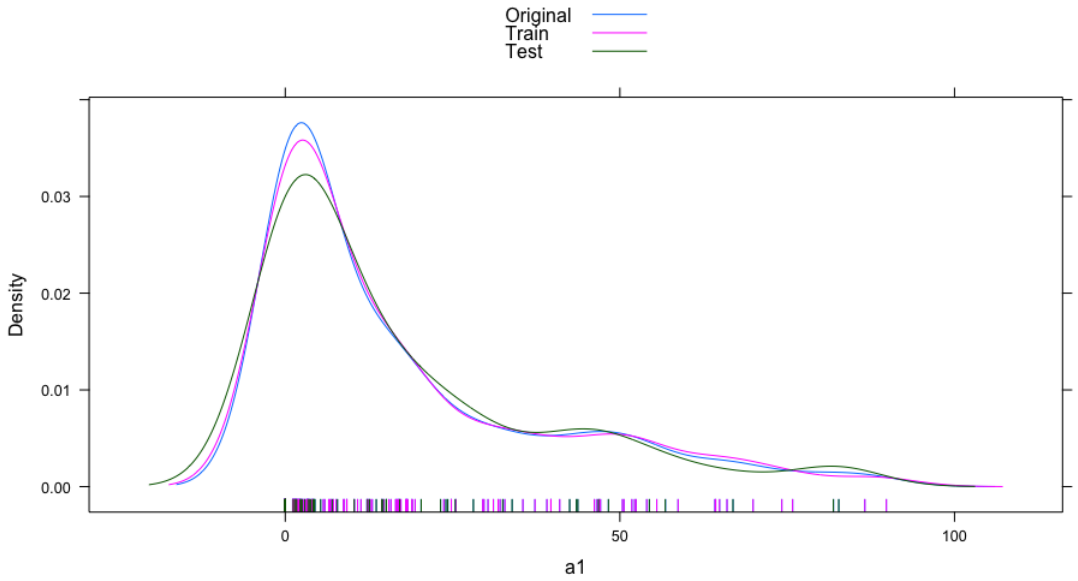


Figure 19: Distribución de valores en conjuntos Original/Training/Testing

```
# Box-Whisker Plot que muestra su similitud
bwplot(dataset~Var,data=toplot,xlab = algae.VarSalida.Usada)
densityplot(~Var,groups=dataset,data=toplot,auto.key = T,
            xlab=algae.VarSalida.Usada,
            panel=panel.superpose,
            panel.groups = function (x,y,col.symbol,lty,  ...) {
              panel.densityplot(x, col=col.symbol, lty=lty,plot.points = F)
              panel.rug(x=jitter(x),col=col.symbol)
            }
)
# col.symbol nos permite dibujar bien los colores de los grupos y
# lty hacerlo en blanco y negro
```

Podemos ver en el diagrama de densidad de la figura 19 que las distribuciones de valores de los tres conjuntos son muy similares. Caret lo consigue dividiendo primero los valores en cuatro grupos delimitados por sus cuartiles y luego extraer de cada uno de ellos las proporciones indicadas. En el caso de clases lo consigue de manera similar, esta vez particionando los datos clase a clase.

Caret también proporciona formas especiales para particionar datos de series temporales (ventanas) o estableciendo diferentes importancias a subgrupos de datos, pero no entraremos en esos detalles aquí. Más información en `http://topepo.github.io/caret/data-splitting.html`.

9 Pre-procesado de datos (II): Eliminar predictores correlados o de poca Varianza

En secciones anteriores de este tutorial hemos visto algunas de las formas de preprocesado de datos. Hemos visto como tratar los valores nulos, o como discretizar variables. También vimos como hacer un análisis exploratorio de datos básico. En la sección 6, en manejo de valores nulos, ya vimos una aplicación de las funcionalidades del preproceso de datos de caret a la hora de "arreglar" *NA*s de valores numéricos, pero `preProcess()` tiene más posibilidades. No obstante primero vamos a ver algunos otros elementos de preprocesado que puede ser interesante hacer antes de pasar a la función `preProcess()`.

Nota: Por comodidad se va a trabajar sobre los conjuntos de datos completos, pero en realidad **se deben realizar estos análisis solo sobre los elementos del conjunto de ENTRENAMIENTO**. Eso sí, **las decisiones tomadas** (transformaciones a realizar, variables a eliminar, etc.) **deben aplicarse a AMBOS conjuntos**, tanto al de **entrenamiento** como al de **test**.

9.1 Eliminar variables con poca Varianza

Que una variable tenga poca varianza indica que carece de mucha información para crear distinciones entre los datos. Cuando práctimente todos los ejemplos son iguales en una característica, dicha característica dice poco de la generalidad de los individuos. Por otro lado, cuando existe poca varianza en alguna variable generamos problemas a muchos de los algoritmos de machine learning. Ten en cuenta que, cuando solo unos pocos ejemplos tienen valores diferentes ,es posible que al dividir los datos de entrenamiento/Validación/Test nos encontremos con pocos o ninguno de dichos ejemplos en alguno de dichos conjuntos, y eso genera modelos inestables. Además, algunos algoritmos (que trabajan con varianzas) pueden dar errores. En resumen, no es buena idea, en general, tener en el modelo variables con poca Varianza, así que podemos eliminarlas. Caret proporciona la función `nearZeroVar()` que las identifica.

Si probamos con algunos de los datasets habituales veremos que no tienen columnas con este problema (por defecto `nearZeroVar()` da como salida un vector con las columnas "conflictivas"), lo cual no es de extrañar porque bastantes de estos conjuntos tradicionales están ya bastante *"limpios"*.

```
> nearZeroVar(iris)
integer(0)
> nearZeroVar(BreastCancer)
integer(0)
> nearZeroVar(BostonHousing)
integer(0)
> nearZeroVar(wine)
integer(0)
> nearZeroVar(adult)
[1] 11 12 14
```

Vemos que solamente `adult` tiene 3 variables con Varianza cero o muy cerca de cero. Vamos a analizar como son dichas variables. Para ello ejecutamos nearZeroVar con el parámetro saveMetrics=T. También hacemos una métrica sencilla de las columnas con problemas.

```
> names(adult)[nearZeroVar(adult)]
[1] "capital_gain" "capital_loss" "country"
```

```
> nzv.adult<-nearZeroVar(adult,saveMetrics = T)
> nzv.adult[nzv.adult$nzv | nzv.adult$zeroVar,]
             freqRatio percentUnique zeroVar  nzv
capital_gain  86.02017     0.3654679   FALSE TRUE
capital_loss 153.67327     0.2825466   FALSE TRUE
country       45.36547     0.1259175   FALSE TRUE
> dim(adult)
[1] 32561    15
> cbind(total=table(adult[["capital_gain"]]),
+                porcentaje=prop.table(table(adult[["capital_gain"]]))*100)
      total   porcentaje
0     29849 91.671017475
114       6  0.018426952
401       2  0.006142317
594      34  0.104419397
914       8  0.024569270
991       5  0.015355794
...
```

Vemos los nombres de las columnas que dan problemas y comprobamos que son las mismas que en las métricas de `nearZeroVar` tienen verdaderas la columna `ZeroVar` o `nzv`. Las otras columnas son los criterios que utiliza `nearZeroVar` para determinar que son columnas a evitar. Por un lado la frecuencia del valor más prevalente sobre el segundo valor más frecuente (el "frequent ratio") que suele ser muy alto en variables muy descompensadas y cercano a 1 en variables con datos balanceados. Por otro lado el porcentaje de valores únicos, que es el número de valores únicos dividido por el total (y multiplicado por 100) y que se acerca a cero a medida que aumenta la granularidad (más valores diferentes en la población). Cuando ambos criterios superan ciertos umbrales se pone a verdadero `nzv`. Se usan ambos criterios a la vez para evitar, por ejemplo, que datos con poca granularidad pero balanceados (lo habitual en "buenas" variables discretas con distribuciones más o menos uniformes) sean catalogadas como problemáticas. En el caso de `adult` vemos que el problema está principalmente en que el valor 0 para `capital gain` es muy frecuente frente al resto. (en `capital loss` es el mismo problema, y en `country` igual, puesto que *United States* es muchísimo más frecuente que el resto de países).

Si decidieramos eliminar directamente esas columnas la forma más sencilla sería:

```
adult.nzv<-adult[,-nearZeroVar(adult)]
```

Pero a veces se puede intentar arreglar de otra manera cuando tenemos muchos valores diferentes, y es agregando algunos de los valores menos frecuentes en categorías más genéricas (lo que aumenta su frecuencia). Así por ejemplo, en el caso de adult, puede ser interesante agrupar los países por zonas geográficas en vez de por países, y las ganancias en tres o cuatro categorías.

Como ejemplo vamos a dividir las ganancias y pérdidas en 3 clases cada una, "None", "Low" y "High". Crearemos los cortes como $[-Inf, 0], (0, Mediana(>0)], (Mediana(>0), Inf]$ (el corte entre "Low" y "high" lo hacemos como la mediana de aquellos valores mayores que 0) y luego dividimos los datos en un nuevo factor ordenado.

```
# Lo arreglaremos en una copia de los datos
adult.discCGCL<-adult
# Intervalos donde discretizar para capital gain
cortes.cg<-c(-Inf, 0,median(adult[["capital_gain"]][adult[["capital_gain"]] >0],
                            na.rm=T),Inf)
# Intervalos donde discretizar para capital loss
cortes.cl<-c(-Inf, 0,median(adult[["capital_loss"]][adult[["capital_loss"]] >0],
                            na.rm=T),Inf)
# Se discretizan capital gain y loss
adult.discCGCL[["capital_gain"]] <- ordered(cut(adult$capital_gain,cortes.cg),
                                        labels = c("None", "Low", "High"))
adult.discCGCL[["capital_loss"]] <- ordered(cut(adult$capital_loss,cortes.cl),
                                        labels = c("None", "Low", "High"))
```

```
> str(adult.discCGCL)
'data.frame': 32561 obs. of  15 variables:
 $ age          : int  39 50 38 53 28 37 49 52 31 42 ...
 $ type_employer: Factor w/ 8 levels "Federal-gov",..: 7 6 4 4 4 4 4 6 4 4 ...
 $ fnlwgt       : int  77516 83311 215646 234721 338409 284582 160187 209642 45781 159449 ...
 $ education    : Factor w/ 16 levels "10th","11th",..: 10 10 12 2 10 13 7 12 13 10 ...
 $ education_num: int  13 13 9 7 13 14 5 9 14 13 ...
 $ marital      : Factor w/ 7 levels "Divorced","Married-AF-spouse",..: 5 3 1 3 3 3 4 3 5 3 ...
 $ occupation   : Factor w/ 14 levels "Adm-clerical",..: 1 4 6 6 10 4 8 4 10 4 ...
 $ relationship : Factor w/ 6 levels "Husband","Not-in-family",..: 2 1 2 1 6 6 2 1 2 1 ...
 $ race         : Factor w/ 5 levels "Amer-Indian-Eskimo",..: 5 5 5 3 3 5 3 5 5 5 ...
 $ sex          : Factor w/ 2 levels "Female","Male": 2 2 2 2 1 1 1 2 1 2 ...
 $ capital_gain : Ord.factor w/ 3 levels "None"<"Low"<"High": 2 1 1 1 1 1 1 1 3 2 ...
 $ capital_loss : Ord.factor w/ 3 levels "None"<"Low"<"High": 1 1 1 1 1 1 1 1 1 1 ...
 $ hr_per_week  : int  40 13 40 40 40 40 16 45 50 40 ...
 $ country      : Factor w/ 41 levels "Cambodia","Canada",..: 39 39 39 39 5 39 23 39 39 39 ...
 $ income       : Factor w/ 2 levels "<=50K",">50K": 1 1 1 1 1 1 1 2 2 2 ...
> cbind(total=table(adult.discCGCL[["capital_gain"]]),
+       porcentaje=prop.table(table(adult.discCGCL[["capital_gain"]]))*100)
     total porcentaje
None 29849  91.671017
Low   1559   4.787936
High  1153   3.541046
> cbind(total=table(adult.discCGCL[["capital_loss"]]),
+       porcentaje=prop.table(table(adult.discCGCL[["capital_loss"]]))*100)
     total porcentaje
None 31042  95.334910
Low    782   2.401646
High   737   2.263444
> names(adult.discCGCL)[nearZeroVar(adult.discCGCL)]
[1] "capital_gain" "capital_loss" "country"
```

Vemos que ahora sus frecuencias están un poco más compensadas. No obstante seguimos dentro de los criterios de nearZeroVar para ser eliminadas. Esto nos permite insistir de nuevo en que en el preprocesado de datos, con la eliminación o la transformación de variables, se pueden estar cambiando los datos sin tener nunca la seguridad de que con ello se facilite el modelado. En muchos casos son decisiones que más tarde pueden mostrarse como erróneas, y entonces se debe volver atrás y deshacer los cambios. En el caso particular de la base de datos adult podría ser interesante mantener las variables. Vamos a ver un gráfico sobre dichas variables, para ver si alguno de los valores está especialmente relacionado con una de las clases en función de esta (lo que haría interesante mantenerla al ser muy discriminante para esa clase):

```
> library(reshape2)
> prop.table(table(adult.discCGCL$income))*100

   <=50K     >50K
75.91904 24.08096
> histogram(~income|value+variable, data=melt(adult.discCGCL,id.vars="income",
+           measure.vars =c("capital_gain","capital_loss")))
```

La figura 20 muestra los histogramas de las variables modificadas "Capital Gain" y "Capital Loss" y vemos que mientras la distribución de las clases es 75% para ejemplos que ganan menos de 50.000 dólares, cuando hay "High" Capital Loss o "High" Capital Gain, esas proporciones se invierten y hay elevados porcentajes de casos que ganan más de 50.000 dólares. Esto podría ayudar bastante a clasificar con facilidad esos ejemplos (en especial "High" Capital Gain).

El ejemplo de modificación mostrado arriba se llama bloqueo (coger una variable categórica con algunos niveles poco frecuentes, o variables numéricas con unos pocos valores únicos y algunos de estos poco frecuentes, y reagrupar algunos de ellos en otros niveles) y, aunque supone una pérdida de información, permite simplificar los modelos, alivia el desbalanceo de niveles, y suele dar mejores resultados. Además, como veremos más adelante, las variables categóricas generan "dummy variables" en los modelos, y reducir el número de niveles simplifica los modelos.

Como conclusión es buena idea, en general, eliminar las variables con poca o nula varianza. No obstante a veces merece la pena tratar de agregar varios niveles y mantenerlas.

9.2 Eliminar variables correladas

Cuando dos variables están muy correladas entre ellas básicamente representan la misma información. Es por ello que suele ser interesante eliminar una de dichas variables y simplificar el modelo al tener menos variables de entrada. Cabe destacar que algunos modelos, como "`pls`" pueden funcionar mejor cuando hay variables correladas por lo que, de nuevo, eliminar estas variables suele ser una buena idea pero no siempre mejora los resultados.

Para encontrar con facilidad variables correladas que están por encima de un umbral podemos usar `findCorrelation()`. Se le tiene que pasar una matriz de correlación y un corte para que devuelva una lista con variables que tienen, al menos, dicha correlación con otra variable. Recordemos que las correlaciones se calculan sobre atributos numéricos, y que solo tiene sentido eliminar variables de entrada. La secuencia de comandos para eliminar variables correladas con un corte de 0.85 para el conjunto `algae` sería:

```
# Seleccionamos las variables de entrada
datos.input<-algae[,1:11]
```

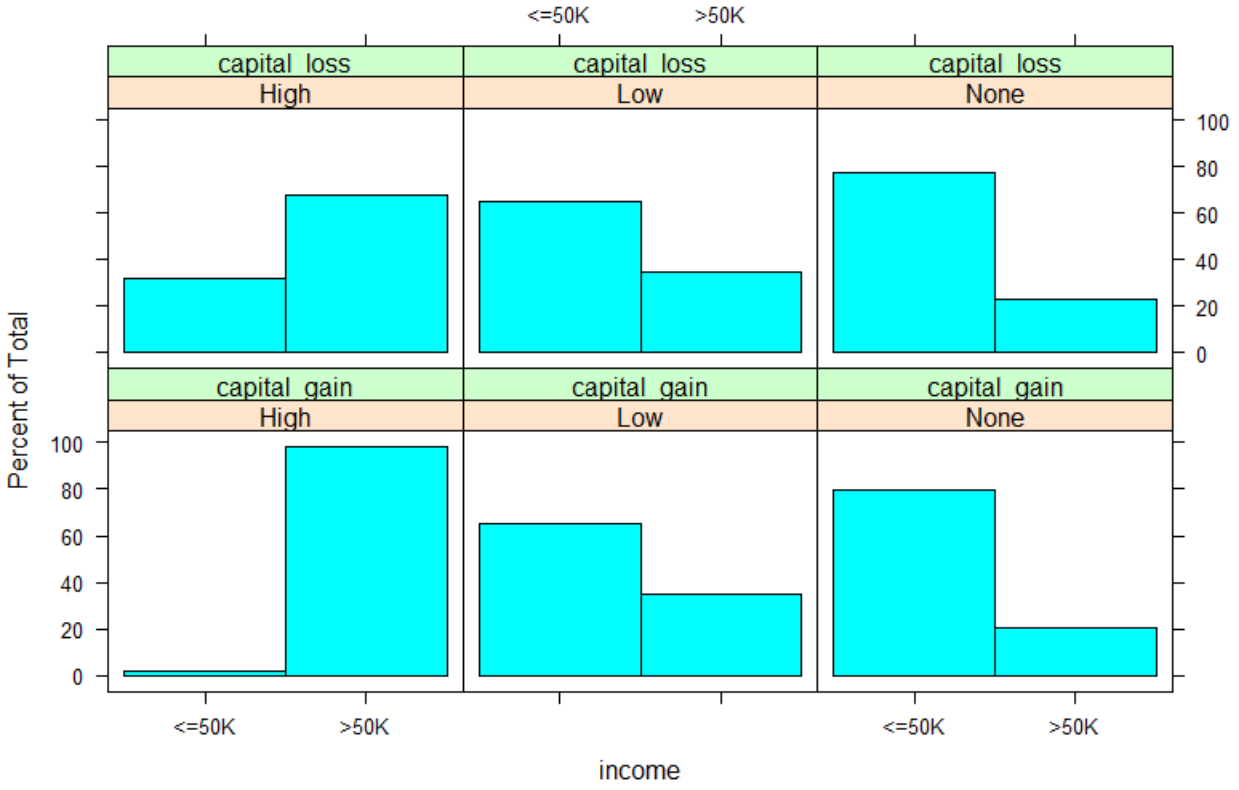


Figure 20: Histogramas de Capital Gain y Loss de Adult Data Set.

```
# Obtenemos la correlación solo de las variables numéricas de las variables de entrada
datos.cor<-cor(na.omit(datos.input[,sapply(datos.input,FUN=is.numeric)]))
# Obtenemos los nombres de las columnas numéricas de entrada correladas
colsToRemove<-labels(datos.cor)[[1]][findCorrelation(datos.cor,cutoff=0.85)]
# Eliminamos las columnas correladas del conjunto completo
algae.nocorr<-algae[setdiff(names(algae),colsToRemove)]
```

Podemos comprobar facilmente que el resultado es el adecuado:

```
> symnum(datos.cor)
     mP mO C1 NO NH o P Ch
mxPH 1
mnO2    1
C1         1
NO3           1
NH4           ,  1
oPO4    .  .        1
PO4     .  .        * 1
Chla .                  1
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
> colsToRemove
[1] "PO4"
```

9.3 Crear Dummy Variables

Las Variables Dummy es una forma de transformar factores en un conjunto de variables numéricas donde cada nivel es ortogonal al resto de variables dummy para ese factor. Dicho de otro modo, para cada nivel del factor se crea una variable numérica que tendrá valor 1 si el valor del ese ejemplo para ese factor es ese nivel y 0 si su valor es cualquier otro. Esto permite separar facilmente una categoría del resto con un hiperplano a 0.5 de esa variable (pues los miembros, con valor 1, estarían a un lado y el resto, con valor 0, al otro lado). Las variables dummy son necesarias en algunos modelos matemáticos que trabajan con la matriz de atributos y valores de entrada.

Caret transforma por defecto las variables de entrada que son factores creando dummy variables (para los modelos que lo requieren, por supuesto). No obstante se puede hacer de manera manual con:

```
# El dataset a ponerle dummy
adult.Data.To.Dummy<-adult.discCGCL
# Hay que indicar qué variables son de salida y entrada
adult.Vars.Salida<-c("income")
adult.Var.Salida.Usada<-c("income")
adult.Vars.Entrada.Usadas<-setdiff(names(adult.Data.To.Dummy),adult.Vars.Salida)
# También qué factores serán Ranked (el resto de factores son no Ranked)
adult.Ranked.Factors<-c("sex")
# Se separan los factores ordenados (no hay que hacer Dummy)
# y los que no se vaya a hacer ranked
```

```
adult.All.Factors<-names(adult.Data.To.Dummy[,adult.Vars.Entrada.Usadas])[sapply(
  adult.Data.To.Dummy[,adult.Vars.Entrada.Usadas], FUN=is.factor)]
adult.Ordered.Factors<-names(adult.Data.To.Dummy[,adult.Vars.Entrada.Usadas])[sapply(
  adult.Data.To.Dummy[,adult.Vars.Entrada.Usadas], FUN=is.ordered)]
adult.Non.Ordered.Factors<-setdiff(adult.All.Factors,adult.Ordered.Factors)
adult.No.Ranked<-setdiff(adult.Non.Ordered.Factors,adult.Ranked.Factors)

# Se calculan las dummy de los ranked y no ranked
adult.Cols.Ranked<-NULL
if(length(adult.Ranked.Factors)>0) {
  adult.Dummy.Ranked<-dummyVars(
    paste("~",paste(adult.Ranked.Factors,sep="",collapse =" + "), collapse=""),
    data=adult.Data.To.Dummy,fullRank=T)
  adult.Cols.Ranked<-data.frame(predict(adult.Dummy.Ranked,
                                        newdata=adult.Data.To.Dummy))
}
adult.Cols.No.Ranked<-NULL
if(length(adult.No.Ranked)>0) {
  adult.Dummy.No.Ranked<-dummyVars(
    paste("~",paste(adult.No.Ranked,sep="",collapse =" + "), collapse=""),
    data=adult.Data.To.Dummy)
  adult.Cols.No.Ranked<-data.frame(predict(adult.Dummy.No.Ranked,
                                           newdata=adult.Data.To.Dummy))
}

# Los factores ordenados se transforman en numéricos
adult.Cols.Ordered.Factors<-NULL
if(length(adult.Ordered.Factors)>0)
  adult.Cols.Ordered.Factors<-data.frame(
    lapply(adult.Data.To.Dummy[,adult.Ordered.Factors],FUN=as.numeric))


adult.Cols.Salida<-data.frame(adult.Data.To.Dummy[,adult.Var.Salida.Usada])
names(adult.Cols.Salida)<-adult.Var.Salida.Usada

#Se eliminan de los datos originales todas las columnas a reemplazar
adult.Data.With.Dummy<-adult.Data.To.Dummy[,setdiff(names(adult.Data.To.Dummy),
                                                    adult.Vars.Salida)]
adult.Data.With.Dummy<-adult.Data.With.Dummy[,setdiff(names(adult.Data.With.Dummy),
                                                      adult.No.Ranked)]
adult.Data.With.Dummy<-adult.Data.With.Dummy[,setdiff(names(adult.Data.With.Dummy),
                                                      adult.Ranked.Factors)]
adult.Data.With.Dummy<-adult.Data.With.Dummy[,setdiff(names(adult.Data.With.Dummy),
                                                      adult.Ordered.Factors)]
# Se añaden todas las columnas nuevas.
```

```
if(!is.null(adult.Cols.Ranked))
  adult.Data.With.Dummy<-cbind(adult.Data.With.Dummy,adult.Cols.Ranked)
if(!is.null(adult.Cols.No.Ranked))
  adult.Data.With.Dummy<-cbind(adult.Data.With.Dummy,adult.Cols.No.Ranked)
if(!is.null(adult.Cols.Ordered.Factors))
  adult.Data.With.Dummy<-cbind(adult.Data.With.Dummy,adult.Cols.Ordered.Factors)
adult.Data.With.Dummy<-cbind(adult.Data.With.Dummy,adult.Cols.Salida)
```

Este código permite seleccionar los grupos de factores a los que aplicar un tipo de dummy ranked o no, y también permite ignorar factores ordered (que tiene más sentido transformarlos a numéricos con su codificación interna que ya está ordenada).

En general, al crear las variables dummy se puede usar `fullRank=T` para crear *n-1* variables (siendo n el número de categorias/niveles) para evitar una colinearidad perfecta (se la denomina **trampa de la variable dummy**). Se trata así de evitar crear variables con alta correlación (y además ahorra espacio) puesto que una variable con dos valores, por ejemplo, crearía dos columnas siendo una la inversa de la otra (perfectamente correladas de forma inversa). Así el sexo es una variable que es aconsejable usar fullRank (es más, si se usan modelos que necesitan calcular la inversa de la matriz de predictores entonces es absolutamente necesario usar fullRank o fallarán), puesto que al tener (normalmente) dos niveles "hombre" y "mujer", no es necesario transformarlo en dos columnas. Sencillamente se coge una columna donde el valor 1 es "hombre" y el valor 0 "mujer" (o viceversa).

En casos donde hay varios niveles ,y cuando no se usen modelos que necesiten que no haya dependencias lineales, se puede usar `fullRank=F` que permite que cada categoría se pueda separar de todas las otras con facilidad preguntando si es valor de dicha columna es igual a uno. Fíjate que si se usa `fullRanked=T` en esos casos habrá una categoría (la que no tiene columna propia) que, para diferenciarla, se debe preguntar que todas las otras columnas dummy creadas a partir de su factor original sean igual a cero.

Ejercicio

Modifica el código anterior para permitir que no se modifiquen un conjunto predeterminado de variables.

10 Pre-procesado de datos (III): Transformando y construyendo variables (Feature Engineering).

Muchas veces los datos poseen información valiosa pero, sencillamente, no están en el formato adecuado para extraerla o para que sea útil. Feature Engineering es la ciencia, o también podría considerarse el arte, de extraer más información de los datos. En realidad no se añade (ni elimina) ninguna información nueva sino que se transforman los datos para hacerlos más útiles.

Por lo habitual se aplican dos tipos de técnicas en feature engineering:

- Transformación de Variables.
- Creación de variables/características.

10.1 Sobre transformaciones de las variables de salida

Como ya se mencionó anteriormente, por lo general las transformaciones solo se realizan sobre las variables de entrada. Transformar la variable de salida, desde este punto de vista, solo tiene sentido por motivos de

hacer, previamente, más comprensible la interpretación de la variable, o algunos casos raros donde la escala puede dar errores de representación de los valores por la precisión de la representación (tener en los modelos floating point overflows u obtener valores $\pm$`Inf` puede ser un indicador de este problema) pero nunca porque el ajuste de los parámetros del modelo lo necesite.

No hay que caer en el error de pensar que se mejora el resultado de una regresión cuando se normaliza la variable de salida, puesto que lo que se hace realmente es escalar también el *MSE* (Mean Square Error, los veremos en la sección 17.3.2), que se debe siempre interpretar en base a la magnitud en la que está medida la variable de salida. En todo caso, si se decide escalar una salida numérica por interpretabilidad es interesante comentar que, en ese caso, el $MSE > 1$ ya nos indicaría que se está haciendo un modelo peor que el modelo ingénuo de predecir siempre una constante.

En todo caso si se transforma la variable de salida, y se desea obtener la predicción en el sistema de medida original, nos tendremos que preocupar de realizar la transformación inversa.

Dicho esto, cuando se trata con ciertos tipos de problemas reales es bastante probable que tengamos que tratar la variable de salida, especialmente en regresión o con series temporales. Es muy importante comprender que con ello estamos transformando realmente el problema base. Por ejemplo, no es lo mismo crear un modelo para predecir el paro sin desestacionalizar que desestacionalizado (el paro tiene una componente oscilatoria estacional que se le puede quitar a los valores brutos). Estas transformaciones harán que no se puedan comparar los resultados de los modelos que se hagan con diferentes transformaciones en la variable de salida (variable depediente), puesto que los problemas base se modifican y esas modificaciones cambian la varianza y las unidades en que se mide esta. Muchas medidas del rendimiento de los modelos de regresión se basan en mediciones de la varianza, como R-square, que mide el tanto por uno de la varianza que sería explicada por el modelo sobre la que se mide.

10.2 Transformación de Variables

Existes varios tipos de transformación que se le pueden aplicar a los datos, entre los que destacaría: Cambio de escala, transformaciones de relaciones entre variables, transformaciones de distribuciones.

10.2.1 Escalado

El cambio de escala se aplica a las variables para normalizarlas y ponerlas todas en una escala común. Esto se hace tanto para mejorar la comprensión sobre su distribución y poder compararlas más facilmente evitándo la distorsión de diferencia de escalas como por el hecho de que de esta manera se evitan problemas con los algoritmos de ajuste de modelos que no posean la propiedad de invarianza al escalado como, por ejemplo, los algoritmos basados en gradiente descente (como las redes neuronales que usen backpropagation).

De forma intuitiva es fácil ver que aplicándo un escalando se evita que alguna de las características o variables de entrada dominen a otras en magnitud y dificulte al algoritmo de ajuste del modelo el que estas características dominadas contribuyan, incluso si son variables importantes. Es por tanto aconsejable, en general, normalizar.

Caret ofrece los 3 tipos básicos de escalado mediante el comando `preProcess()`:

`center` Le resta la media de los valores.

`scale` Divide los valores por la desviación estándar. Es decir, que aplicar "center" y "scale" estandarizaría una distribución normal o gaussiana.

`range` Normaliza las variables a una distribución uniforme a un intervalo [0,1]. Es decir la transforma según: $(x_i - min(X))/(max(X) - min(X))$.

A continuación vemos algunos ejemplos de como aplicarlos:

```
# Usamos el dataset iris y hacemos una partición al 80%
data(iris)
iris.Datos.Todo<-iris
iris.Var.Salida.Usada<-c("Species")
iris.Vars.Entrada.Usadas<-setdiff(names(iris.Datos.Todo),iris.Var.Salida.Usada)
iris.Vars.Entrada.Escaladas<-iris.Vars.Entrada.Usadas

set.seed(1234)
iris.trainIdx.80<- createDataPartition(iris.Datos.Todo[[iris.Var.Salida.Usada]],
                               p=0.8,
                               list = FALSE,
                               times = 1)
iris.Datos.Train<-iris.Datos.Todo[iris.trainIdx.80,]
iris.Datos.Test<-iris.Datos.Todo[-iris.trainIdx.80,]


# Estimamos los valores para normalizar usando el conjunto de entrenamiento
# Solo usamos las variables que vayamos a preprocesar
iris.preProc.CS.Mod<-preProcess(iris.Datos.Train[iris.Vars.Entrada.Escaladas],
                      method=c("center","scale"))

# Aplicamos las transformaciones a ambos conjuntos
iris.Datos.Train.Transf.CS<-predict(iris.preProc.CS.Mod,iris.Datos.Train)
iris.Datos.Test.Transf.CS<-predict(iris.preProc.CS.Mod,iris.Datos.Test)


# Dibujar un diagrama de densidad para las variables que han sido escaladas
VarToPlot<-iris.Vars.Entrada.Escaladas
d1<-densityplot(
  formula(paste("~",paste(VarToPlot,sep="",collapse =" + "),collapse="")),
  data=iris.Datos.Train,main="Variables sin Normalizar",plot.points=F)
d2<-densityplot(
  formula(paste("~",paste(VarToPlot,sep="",collapse =" + "),collapse="")),
  data=iris.Datos.Train.Transf.CS, main="Variables Normalizadas",plot.points=F)

# Gráficos en blanco y negro (mejor para imprimir)
trellis.par.set(theme = standard.theme("pdf",color=FALSE))

print(d1,position=c(0,0,0.5,1),more=T)
print(d2,position=c(0.5,0,1,1))

# Volvemos a gráficos normales
trellis.par.set(theme = standard.theme("pdf"))
```

Es importante recordar que los parámetros de las transformaciones (medias, desviaciones máximas, máximos o mínimos) se deben estimar sobre el conjunto de datos de **entrenamiento**, ya que se supone

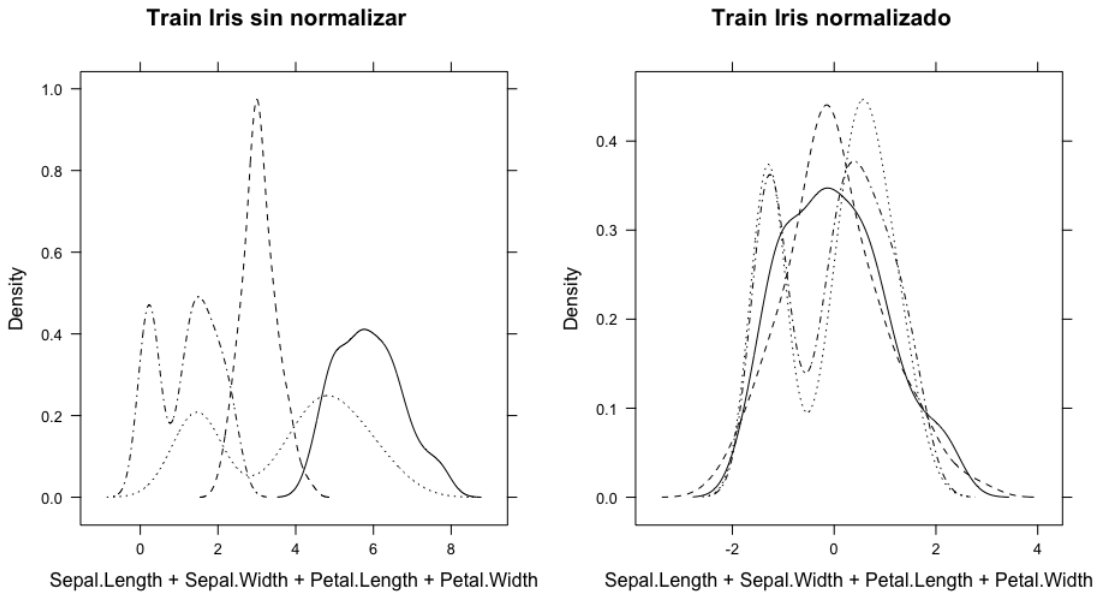


Figure 21: Distribución de las variables del problema `iris` antes y después de normalizarlas.

que no se conoce ninguna información del conjunto de **test**. Por ello el comando `preProcess()` se hace usando dichos datos de entrenamiento.

En la figura 21 se puede ver el efecto que tiene el escalado sobre las variables de entrada del dataset `iris` y comprobamos que ahora todas las variables tienen la misma escala y están centradas, es decir, que sus valores tienen la misma magnitud y se mueven en los mismos rangos.

La función `preProcess()` solo escala variables numéricas (ignora los factores) con lo que en el problema `iris`, al indicar que queremos un preproceso sobre todo el dataset (incluyéndo la salida), ha ignorado la variable de salida, pero esto no sería así si nuestra variable de salida hubiese sido numérica, puesto que también la hubiese transformado. Es decir, que tenemos que tener cuidado al hacer `preProcess()` e incluir solo las columnas que deseemos sean tratadas en dicho preProceso. Al hacer el `predict()` se puede hacer sobre todo el conjunto puesto que ignora las variables que no aparecían en el conjunto sobre el que se hizo el `preProcess()`.

Aquí muestro un código que permite escoger que variables se van a escalar:

```
# Semilla de números aleatorios
set.seed(1234)

# Dataset a utilizar. Sus variables de entrada y salida.
algae.Datos.Todo<-algae
algae.Vars.Salida<-c("a1","a2","a3","a4","a5","a6","a7")
algae.Vars.Entrada<-setdiff(names(algae.Datos.Todo),algae.Vars.Salida)
# Variable a predecir, variables de entrada a usar y variables que se escalarán.
```

```
algae.Var.Salida.Usada<-c("a1")
algae.Vars.Entrada.Usadas<-algae.Vars.Entrada       # Por si algunas se ignoran
algae.Vars.Entrada.Escaladas<-algae.Vars.Entrada.Usadas  # Vars a escalar

algae.Datos.Usados<-cbind(algae.Datos.Todo[algae.Vars.Entrada.Usadas],
                          algae.Datos.Todo[algae.Var.Salida.Usada])
algae.TrainIdx.80<- createDataPartition(algae.Datos.Usados[[algae.Var.Salida.Usada]],
                                p=0.8,
                                list = FALSE,
                                times = 1)
algae.Datos.Train<-algae.Datos.Usados[algae.TrainIdx.80,]
algae.Datos.Test<-algae.Datos.Usados[-algae.TrainIdx.80,]

# Solo indicamos las variables que vayamos a transformar (y usamos TRAIN)
algae.preProc.CS.Mod<-preProcess(algae.Datos.Train[algae.Vars.Entrada.Escaladas],
                             method=c("center","scale"))

# Obtenemos la transformación
algae.Datos.Train.Transf.CS<-predict(algae.preProc.CS.Mod,algae.Datos.Train)
algae.Datos.Test.Transf.CS<-predict(algae.preProc.CS.Mod,algae.Datos.Test)
```

10.2.2 Transformaciones de relaciones entre variables

Dentro de este tipo de transformaciones de los datos encontramos aquellas en que se crean nuevas variables combinando de alguna manera algunas de (o todas) las variables originales.

10.2.2.1 Análisis de Componentes Principales (PCA)

El análisis de componentes principales es un tipo de transformación que nos permite hacer una reducción de dimensionalidad de los datos de entrada. PCA nos crea nuevas variables de entrada (con nombres PC1, PC2, etc.) a partir de una combinación lineal de las variables de entrada originales. Estas nuevas variables no están correladas las unas con las otras (son ortogonales). Las variables están ordenadas de mayor a menor varianza, y es habitual ignorar aquellas que tienen poca, consiguiendo así esa reducción de dimensionalidad mencionada antes.

Caret incluye en `preProcess()` la posibilidad de aplicar PCA, indicando el tanto por uno de la varianza original que se mantiene en el nuevo conjunto de variables (en el parámetro `thresh` de `preProcess()`). De primeras no se conoce el número de variables que se reducirán puesto que no se conoce de forma previa cuanto aporta de la varianza cada una de las nuevas variables que produce PCA.

PCA solo funciona sobre variables numéricas y es muy sensible a valores atípicos (outliers) por lo que se aconseja eliminarlos primero. A continuación un ejemplo sencillo de como aplicarla. No obstante recordad que se debe calcular sobre los datos de entrenamiento.

```
# Sobre iris (se calcula sobre train)
iris.PreProc.Pca.Mod<-preProcess(iris.Datos.Train[,1:4],method=c("pca"),thresh = 0.9)
iris.Datos.Train.Transf.PCA<-predict(iris.PreProc.Pca.Mod,iris.Datos.Train)
```

```
iris.Datos.Test.Transf.PCA<-predict(iris.PreProc.Pca.Mod,iris.Datos.Test)
iris.Datos.Todo.Transf.PCA<-predict(iris.PreProc.Pca.Mod,iris.Datos.Todo)

# Sobre Pima Diabetes (se debería calcular sobre train)
pima.PreProc.Pca.Mod<-preProcess(PimaIndiansDiabetes[,1:8],
                                 method=c("pca"),thresh = 0.85)
pima.Datos.Todo.Transf.PCA<-predict(pima.PreProc.Pca.Mod,PimaIndiansDiabetes)

# Sobre algae (si has ejecutado el último bloque)
# SOLO indicamos las variables que vayamos a transformar
algae.PreProc.Pca.Mod<-preProcess(algae.Datos.Train[algae.Vars.Entrada.Escaladas],
                                  method=c("pca"), thresh = 0.85)

# Obtenemos la transformación
algae.Datos.Train.Transf.PCA<-predict(algae.PreProc.Pca.Mod,algae.Datos.Train)
algae.Datos.Test.Transf.PCA<-predict(algae.PreProc.Pca.Mod,algae.Datos.Test)
algae.Datos.Todo.Transf.PCA<-predict(algae.PreProc.Pca.Mod,algae.Datos.Todo)
```

Los resultados serían:

```
> print(iris.PreProc.Pca.Mod)
Created from 120 samples and 4 variables

Pre-processing:
  - centered (4)
  - ignored (0)
  - principal component signal extraction (4)
  - scaled (4)

PCA needed 2 components to capture 90 percent of the variance
> summary(iris.Datos.Todo.Transf.PCA)
       Species        PC1                PC2
 setosa    :50   Min.   :-2.7332   Min.   :-2.55880
 versicolor:50   1st Qu.:-2.0910   1st Qu.:-0.54182
 virginica :50   Median : 0.3837   Median : 0.01635
                 Mean   :-0.0220   Mean   : 0.03394
                 3rd Qu.: 1.2982   3rd Qu.: 0.61167
                 Max.   : 3.2172   Max.   : 2.58845
> print(pima.PreProc.Pca.Mod)
Created from 768 samples and 8 variables

Pre-processing:
  - centered (8)
  - ignored (0)
  - principal component signal extraction (8)
  - scaled (8)
```

```
PCA needed 6 components to capture 85 percent of the variance
> print(algae.PreProc.Pca.Mod)
Created from 149 samples and 11 variables

Pre-processing:
  - centered (8)
  - ignored (3)
  - principal component signal extraction (8)
  - scaled (8)

PCA needed 5 components to capture 85 percent of the variance
```

10.2.2.2 Análisis de componentes independientes (ICA)

Es similar a PCA pero las nuevas variables son combinaciones lineales de las variables originales dondelas nuevas variables son variables independientes (en vez de no correladas como el PCA). Los nombres de las variables son ICA1, ICA2, etc. Por último, a diferencia de PCA sí se indica el número exacto de componentes a utilizar con el parámetro `n.comp`.

```
# Sobre iris (se calcula sobre train)
iris.PreProc.Ica.Mod<-preProcess(iris.Datos.Train[,1:4],method=c("ica"),n.comp = 2)
iris.Datos.Train.Transf.ICA<-predict(iris.PreProc.Ica.Mod,iris.Datos.Train)
iris.Datos.Test.Transf.ICA<-predict(iris.PreProc.Ica.Mod,iris.Datos.Test)
iris.Datos.Todo.Transf.ICA<-predict(iris.PreProc.Ica.Mod,iris.Datos.Todo)

# Sobre Pima Diabetes (se debería calcular sobre train)
pima.PreProc.Ica.Mod<-preProcess(PimaIndiansDiabetes[,1:8],
                                 method=c("ica"),n.comp = 5)
pima.Datos.Todo.Transf.ICA<-predict(pima.PreProc.Ica.Mod,PimaIndiansDiabetes)

# Sobre algae (si has ejecutado el último bloque)
# SOLO indicamos las variables que vayamos a transformar
algae.PreProc.Ica.Mod<-preProcess(algae.Datos.Train[algae.Vars.Entrada.Escaladas],
                                  method=c("ica"), n.comp=7)
# Obtenemos la transformación
algae.Datos.Train.Transf.ICA<-predict(algae.PreProc.Ica.Mod,algae.Datos.Train)
algae.Datos.Test.Transf.ICA<-predict(algae.PreProc.Ica.Mod,algae.Datos.Test)
algae.Datos.Todo.Transf.ICA<-predict(algae.PreProc.Ica.Mod,algae.Datos.Todo)
```

Los resultados serían:

```
> print(iris.PreProc.Ica.Mod)
Created from 120 samples and 4 variables

Pre-processing:
  - centered (4)
```

```
  - independent component signal extraction (4)
  - ignored (0)
  - scaled (4)

ICA used 2 components
> summary(iris.Datos.Todo.Transf.ICA)
       Species        ICA1               ICA2
 setosa    :50   Min.   :-2.69874   Min.   :-1.89792
 versicolor:50   1st Qu.:-0.54541   1st Qu.:-0.77037
 virginica :50   Median : 0.03190   Median :-0.21515
                 Mean   : 0.03494   Mean   : 0.01398
                 3rd Qu.: 0.63892   3rd Qu.: 1.22635
                 Max.   : 2.69019   Max.   : 1.60076
> print(pima.PreProc.Ica.Mod)
Created from 768 samples and 8 variables

Pre-processing:
  - centered (8)
  - independent component signal extraction (8)
  - ignored (0)
  - scaled (8)

ICA used 5 components
> print(algae.PreProc.Ica.Mod)
Created from 149 samples and 11 variables

Pre-processing:
  - centered (8)
  - independent component signal extraction (8)
  - ignored (3)
  - scaled (8)

ICA used 7 components
```

10.2.2.3 Spatial Sign

Esta transformación proyecta los datos de un grupo de variables de entrada (numéricas) en un círculo unitario de tantas dimensiones como variables de entrada transformada. Hay ocasiones en que es más fácil discriminar tras esta transformación. Un par de ejemplos de como usarlos se obtienen con los siguentes comandos, que incluyen unos diagramas para ver los efectos de esta transformación:

```
# Sobre iris tras aplicar el PCA
iris.PreProc.spa.Mod<-preProcess(iris.Datos.Train.Transf.PCA[,2:3],
                                 method=c("spatialSign"))
iris.Datos.Train.Transf.PCA.SPA<-predict(iris.PreProc.spa.Mod,
```

```
                                          iris.Datos.Train.Transf.PCA)
iris.Datos.Test.Transf.PCA.SPA<-predict(iris.PreProc.spa.Mod,
                                        iris.Datos.Test.Transf.PCA)
iris.Datos.Todo.Transf.PCA.SPA<-predict(iris.PreProc.spa.Mod,
                                        iris.Datos.Todo.Transf.PCA)

# Sobre Pima Diabetes (se debería calcular sobre train)
pima.PreProc.Spa.Mod<-preProcess(PimaIndiansDiabetes[,1:8],method=c("spatialSign"))
pima.Datos.Todo.Transf.SPA<-predict(pima.PreProc.Spa.Mod,PimaIndiansDiabetes)

p1<-xyplot(PC1~PC2,group=Species,data=iris.Datos.Todo.Transf.PCA,
           pch=c("+","O","X"),cex=1.5)
p2<-xyplot(PC1~PC2,group=Species,data=iris.Datos.Todo.Transf.PCA.SPA,
           pch=c("+","O","X"),cex=1.5)

require("ellipse")  #featurePlot necesita esta librería para hacer elipses
p3<-featurePlot(x=iris.Datos.Todo.Transf.PCA[,2:3],
                y=iris.Datos.Todo.Transf.PCA[,1],plot="ellipse")
p4<-featurePlot(x=iris.Datos.Todo.Transf.PCA.SPA[,2:3],
                y=iris.Datos.Todo.Transf.PCA.SPA[,1],plot="ellipse")

print(p1,position=c(0,0.5,0.5,1),more=T)
print(p2,position=c(0.5,0.5,1,1),more=T)
print(p3,position=c(0,0,0.5,0.5),more=T)
print(p4,position=c(0.5,0,1,0.5))
```

Los diagramas de los comandos anteriores se ven en la figura 22 y muestran como se distribuyen los datos en forma de círculo de radio 1.

Los modelos resultantes serían:

```
> print(iris.PreProc.spa.Mod)
Created from 120 samples and 2 variables

Pre-processing:
  - centered (2)
  - ignored (0)
  - scaled (2)
  - spatial sign transformation (2)

> summary(iris.Datos.Todo.Transf.PCA.SPA)
       Species        PC1                PC2
 setosa    :50   Min.   :-0.9999   Min.   :-0.95700
 versicolor:50   1st Qu.:-0.7830   1st Qu.:-0.45340
 virginica :50   Median : 0.3229   Median : 0.02413
                 Mean   : 0.1222   Mean   : 0.09572
                 3rd Qu.: 0.8332   3rd Qu.: 0.74891
```

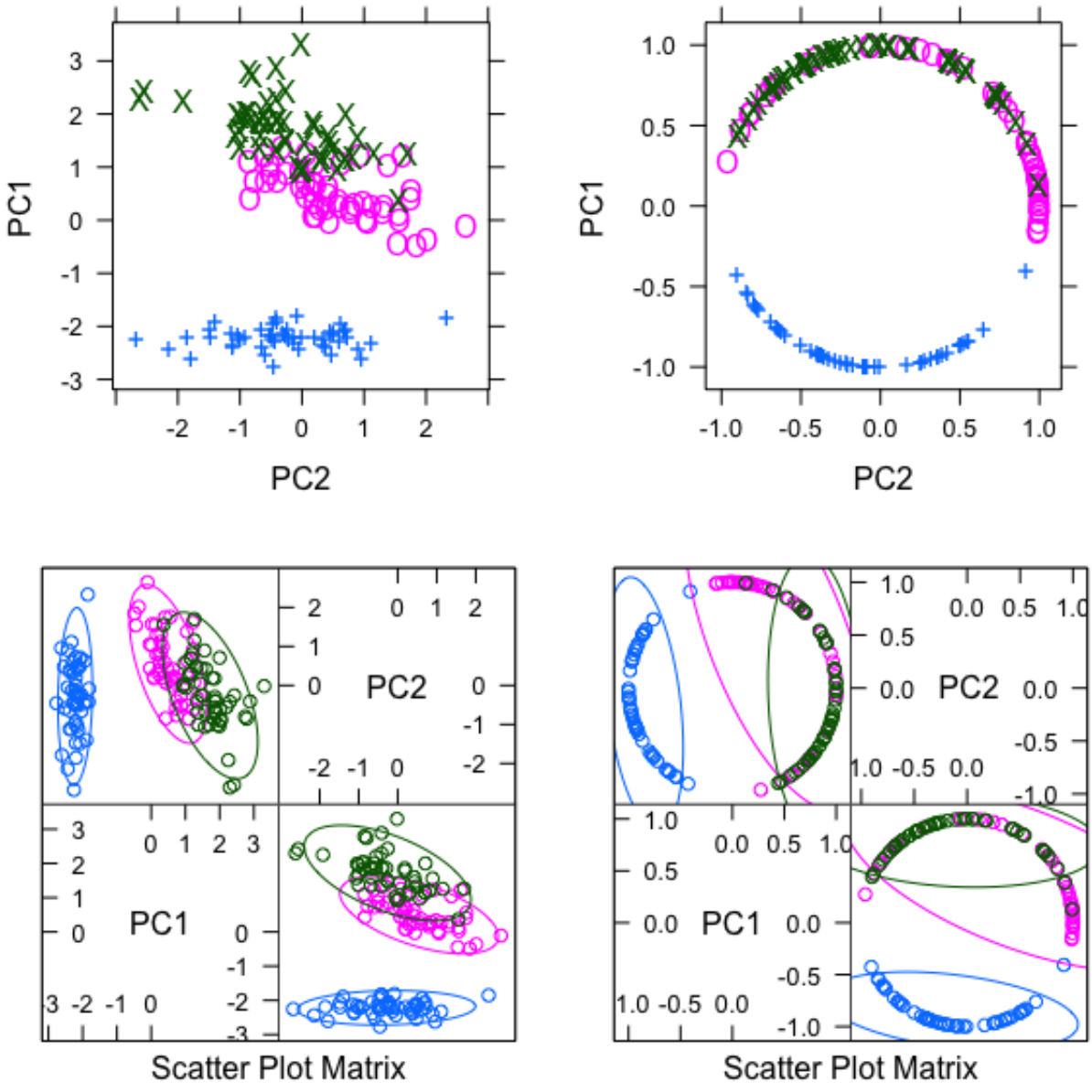


Figure 22: Diagramas que muestran la transformación Spatial

```
                Max.   : 1.0000   Max.   : 0.99997
> print(pima.PreProc.Spa.Mod)
Created from 768 samples and 8 variables

Pre-processing:
  - centered (8)
  - ignored (0)
  - scaled (8)
  - spatial sign transformation (8)
```

10.2.3 Transformaciones de distribuciones asimétricas

Por lo general es mejor trabajar con distribuciones que sean simétricas ya que muchos modelos estadísticos funcionan mejor con datos lo más "gausianos" posible. Si encontramos variables que presentan alguna asimetría (skewedness) se las puede transformar de manera directa con las siguientes fórmulas.

	Asimetría	
Grado de Asimetría	Izquierda	Derecha
Pequeña	x^2	$\sqrt{x}$
Media	x^3	$\sqrt[3]{x}$
Grande	$exp(x)$	$log(x)$

También se pueden usar otras transformaciones que proporciona Caret dentro de `preProcess()`. En particular se pueden utilizar la transformadas Box-Cox (con el parámetro `method=c("BoxCox")`), la Yeo-Johnson (con el parámetro `method=c( "YeoJohnson")`) o la exponencial (con el parámetro `method=c("expoTrans"`).

Con el siguiente código puedes ver un ejemplo del tipo de transformación que realizan esos métodos.

```
Var<-"pedigree"
dx<-PimaIndiansDiabetes[Var]
preProcValuesBC <- preProcess(dx, method = "BoxCox")
dxBoxCox <- predict(preProcValuesBC, dx)
preProcValuesYJ <- preProcess(dx, method = "YeoJohnson")
dxYJ <- predict(preProcValuesYJ, dx)
preProcValuesT <- preProcess(dx, method = "expoTrans")
dxExpoTrans <- predict(preProcValuesT, dx)
d1<-histogram(formula(paste("~",Var,collapse=" ")), data=dx,
              xlab = "Variable Original", type = "density",
              panel = function(x, ...) {
                panel.histogram(x, ...)
                panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
                panel.rug(x=jitter(x),col="black")
              })
d2<-histogram(formula(paste("~",Var,collapse=" ")), data=dxBoxCox,
              xlab = "Variable Transformada BoxCox", type = "density",
              panel = function(x, ...) {
                panel.histogram(x, ...)
```

```
                panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
                panel.rug(x=jitter(x),col="black")
              })
d3<-histogram(formula(paste("~",Var,collapse=" ")), data=dxYJ,
              xlab = "Variable Transformada Yeo-Johnson", type = "density",
              panel = function(x, ...) {
                panel.histogram(x, ...)
                panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
                panel.rug(x=jitter(x),col="black")
              })
d4<-histogram(formula(paste("~",Var,collapse=" ")), data=dxExpoTrans,
              xlab = "Variable Transformada Exponential", type = "density",
              panel = function(x, ...) {
                panel.histogram(x, ...)
                panel.densityplot(x=na.omit(x),col = "black", plot.points=F)
                panel.rug(x=jitter(x),col="black")
              })
print(d1,position=c(0,0.5,0.5,1),more=T)
print(d2,position=c(0.5,0.5,1,1),more=T)
print(d3,position=c(0,0,0.5,0.5),more=T)
print(d4,position=c(0.5,0,1,0.5))
```

El gráfico que genera se puede ver en la figura 23. No todos los métodos son aplicables a todas las posibles distribuciones, por ejemplo, el método Box-Cox,al igual que la raíz cuadrada, necesita valores positivos para poderse aplicar. El método Teo-Johnson o la raíz cúbica si pueden trabajar con valores negativos.

10.2.4 Binning (Categorizar)

Binning, o categorizar una variable numérica, es también un tipo de transformación de variable (que ya hemos mencionado antes aunque ahora la explicamos en profundidad). Se aplica de diversas formas, tanto directamente sobre los valores, como determinando los puntos de corte a intervalos fijos (se divide el rango en un número fijo de contenedores de igual tamaño), o encontrando puntos de corte que generan intervalos de diferentes tamaños pero que contengan un número similar de datos (usando p.e. percentiles o quartiles). También se pueden hacer categorías permitiendo el solapamiento. es decir, basados en frecuencias, o puntos de corte establecidos en valores distinguidos (como la mediana, es decir, el percentil 50). Las razones que justificarían un binning es simplificar los datos y hacerlos más fáciles de interpretar.

El decidir la forma y los puntos sobre los que realizar el binning muchas veces depende más del sentido que le da la transformación a los datos que a algún criterio científico (aunque sigue manteniéndose que para la mayoría de modelos es mejor tener categorías equilibradas). Insisto en que su valor es aumentar la interpretabilidad de los datos. También se aplica cuando se usan algoritmos que, precisamente, trabajan solo con categorías (como son los árboles de decisión más sencillos).

En principio, al **aplicar binning**, se está perdiendo información y, por lo general, **no está aconsejado**. Hay una serie de problemas conocidos asociados a binning (p.e. ver `http://biostat.mc.vanderbilt.edu/wiki/Main/CatContinuous` para una lista de ellos en el ámbito médico, donde es habitual hacerlo[4]) pero

[4]Esta tendencia en el ámbito médico tiene mucho que ver con el hecho de que la forma habitual de diagnóstico se basa en reglas de decisión que trabajan mucho con categorías, (más que con valores numéricos) puesto que les son mucho más fáciles y

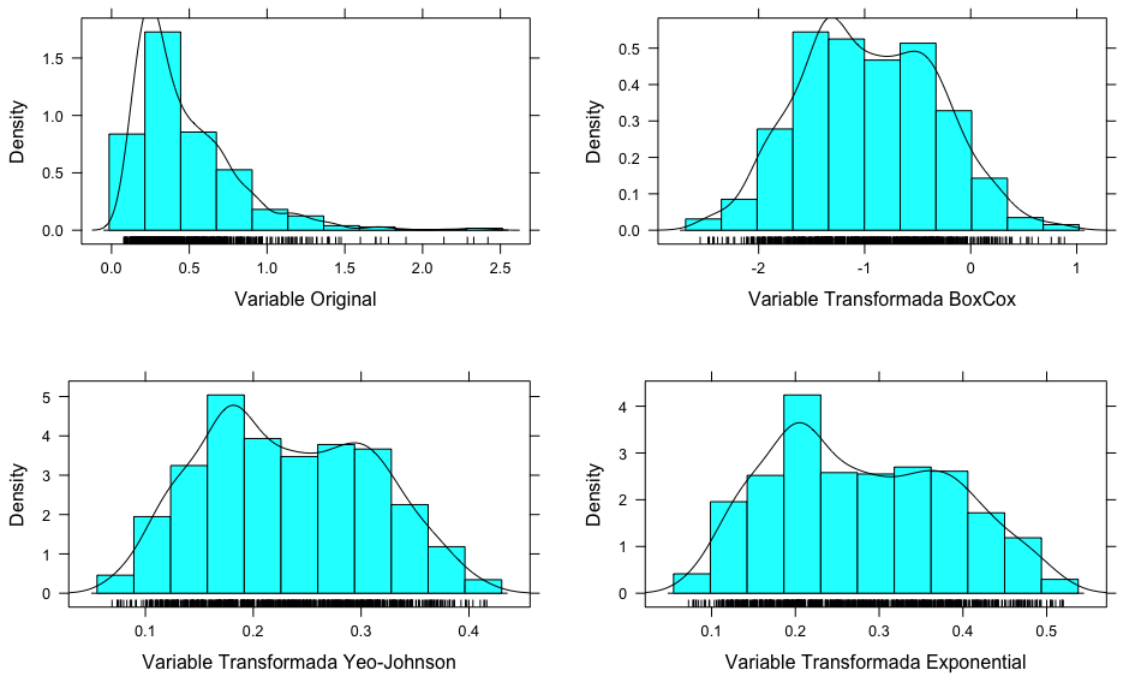


Figure 23: Transformaciones sobre la variable "pedigree" del dataset Diabetes Pima Indians, que presenta una asimetría (skeweedness) hacia la izquierda.

también ha existido cierta presión a transformar los datos a estructuras de representación que sean más fáciles de comprender por los humanos. A los seres humanos les gustan mucho las categorías, categorizamos de manera automática y continua todo lo que percibimos. Para complicar las cosas, además, la mayor parte de categorías que manejan los seres humanos son "fuzzy" (difusas/borrosas), lo que significa que la separación entre categorías adyacentes no es precisa y concluyente y que hay zonas "grises" donde hay una transición gradual de una categoría a la otra. Para los humanos es más fácil comprender un modelo si toma decisiones en función de categorías (un árbol de decisión es un ejemplo claro de modelo cuya estructura fácil de interpretar para los humanos, especialmente si los atributos (variables) son categóricos), pero por desgracia estos modelos suelen tener peores resultados.

Hoy en día no hay una presión tan grande en que los modelos de machine learning sean interpretables. Hay confianza en las "máquinas" y ya no hace falta ni mostrar los modelos ni los datos a los clientes. Aun así se puede hacer binning en el análisis previo de datos para ser más fáciles de interpretar por el analista de datos, pero no usar dicho binning en el entrenamiento del modelo.

Por último, para terminar de confundir, mencionar que los árboles de decisión son parte fundamental de los random forests (una de las (familias de) técnicas que suele dar mejores resultados en datos no perceptuales (no basados en imágenes) y que son más rápidos de crear cuando los datos son categóricos, por lo que lo mismo binning puede, en esos casos, ser de utilidad. Como se ha repetido en varias ocasiones, para saberlo con seguridad hay que comprobarlo empíricamente.

Ejercicio Categoriza las variable numérica indicadas en los conjuntos que se te indiquen:

- `adult[["age"]]` en 6 conjuntos de similar tamaño y sobrelapamiento de aproximadamente 5%.
- `iris[["Sepal.Length"]]` en 5 conjuntos en los percentiles 20-40-60-80.
- `algae[["a2"]` en 3 conjuntos: los valores igual a 0, de 0 a mediana de valores diferentes de 0,y de mediana al máximo.
- `adult[["education_num"]]` en 5 conjuntos más o menos iguales, "a ojo".

10.2.5 Bloqueo (Blocking)

Existen casos donde nos encontraremos con variables categóricas con muchas categorías diferentes o con categorías muy descompensadas respecto a su frecuencia (seguimos hablando sobre variables/atributos de entrada, cuando hay descompensación de clases en un problema de clasifiación para la variable de salida hay que aplicar ciertas técnicas que veremos en la sección 13 para paliar posibles efectos de dicha descompensación). No es poco habitual, además, que algunas categorizaciones sean confusas, p.e., que algunos niveles no están bien definidos, o que existan niveles a diferentes niveles de abstracción, o que haya niveles que estén incluídos en otros o tengan intersección con otros (lo que hace difícil distinguirlos con claridad). En estos casos puede ser interesante reagrupar las categorías.

Agrupar algunas categorías con escasos representantes en categorías más generales o reducir el número de niveles (uniendo varias categorías en una más general) es otra forma de transformación que permite equilibrar mejor categorías que estén descompensadas o clarificar/simplificar variables categóricas.

Merece la pena llamar la atención sobre el hecho de que haciendo bloqueo también estamos potencialmente perdiendo información que podría ser relevante. En algunos casos se hará porque la información sea confusa o no esté muy clara. En otros casos se hará, sencillamente, por no tener suficientes datos dentro de un grupo

naturales a los humanos (los médicos).

para tomar una decisión sobre dicho grupo al estar pobremente representado, así que se bloquean en grupos similares. En todo caso el bloqueo simplifica drásticamente los modelos a usar. En métodos basados en matemáticas, cuando se usan variables categóricas, el cálculo debe hacerse sobre las variables dummy que hemos visto antes, lo que implica la creación una variable (dummy) por cada nivel de la variable categórica. Es decir, que si tienes 2 variables categóricas, con 6 y 4 categorías respectivamente, los cálculos añaden 10 nuevas variables a la ecuación. En bases de datos donde la mayoría de las variables son categóricas, y cada una con bastantes niveles, es fácil ver como el número de variables en las ecuaciones del modelo se incrementan drásticamente. El bloqueo reduce este problema.

10.3 Creando nuevas variables

Se podría considerar que alguna de las técnicas que hemos mencionado en transformación de variables son, en cierto modo, nuevas variables (p.e. PCA) pero aquí mencionaremos alguna técnica que crea nuevas variables sin reemplazar las variables de las que proviene (aunque en muchos casos pueden terminar reemplazándolas).

10.3.1 Cálculo de distancia de clases

Caret incluye una función ge genera nuevas variables basadas en las distancias a los centroides de las clases (parecido a como funciona el análisis de discriminador lineal). El método funciona calculando el centroide de clase y la matrix de covarianza sobre cada nivel de una variable factor. Las nuevas variables son las distancias de Mahalanobis a cada uno de esos centroides de clase. En la documentación de caret indica que estos predictores ayudan a sistemas no lineales cuando la frontera de decisión es lineal.

A continuación un ejemplo de como crear esas variables. Los nombres de las variables siguen el formato `dis.Etiqueta_de_clase`.

```
# Ejemplo de crear nuevas variables con Class Distribution
iris.preProc.ClssDtrb.Mod<-classDist(iris.Datos.Train[,1:4],iris.Datos.Train[,5])
# Añadimos las nuevas variables
iris.xtra.Vars<-cbind(iris,predict(iris.preProc.ClssDtrb.Mod,iris[,1:4]))
# Dibujamos las nuevas variables y las antiguas
p1<-featurePlot(x=predict(iris.preProc.ClssDtrb.Mod,iris[,1:4]),
                iris[,5],plot="ellipse")
p2<-featurePlot(x=iris[,1:4],iris[,5],plot="ellipse")
print(p1,position=c(0,0,0.5,1),more=T)
print(p2,position=c(0.5,0,1,1))
```

En los diagramas de la figura 24 se muestra la distribución espacial dos a dos de las nuevas variables.

10.4 Combinando todo el preproceso

Hemos visto toda una batería de posibles arreglos, eliminaciones, transformaciones, conversiones, escalados, discretizaciones y creaciones de nuevas variables. Hay algunos otros que se verán más adelante (como el sobre y submuestreo de la sección 13). Muchas de estos procesos se llevan a cabo sobre algunas de las variables, mientras otras quedan sin tocar.

En principio todo el proceso de obtención de un modelo de predicción seguiría el algoritmo descrito en Algorithm 1 :

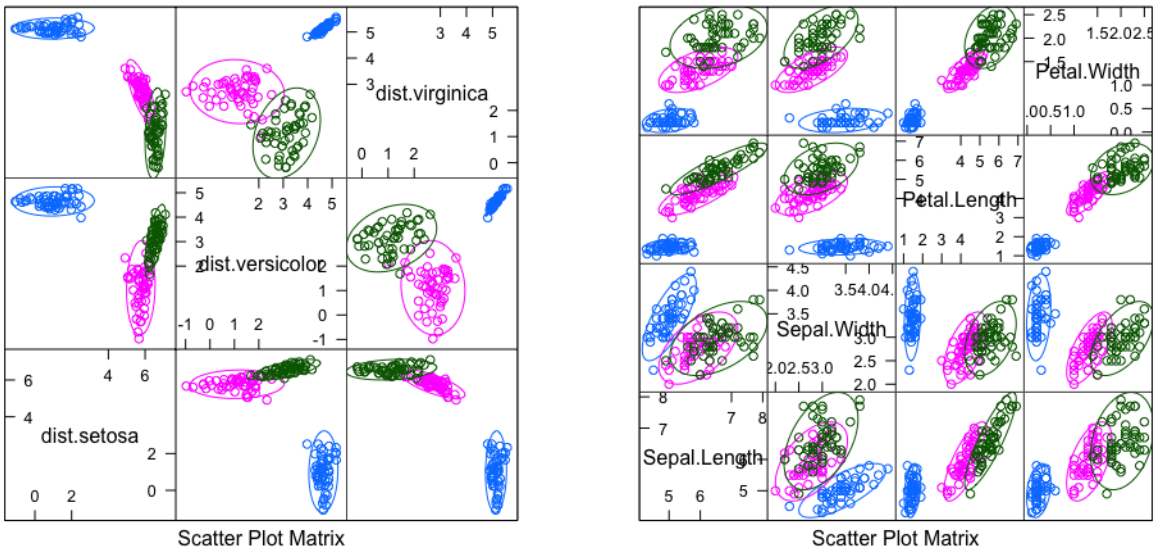


Figure 24: Diagramas que muestran nuevas variables creadas por distancia a centroide de clase

Como se indica entre las líneas 5 y 11, el preproceso es la repetición de escoger una o varias variables a preprocesar, realizar la transformación correspondiente, aplicarla a los datos de entrenamiento y test, y guardarla para aplicarla luego a nuevos datos que se necesiten predecir. Cuando más adelante se necesite el modelo final para predecir un nuevo dato, como se dice en la línea 15, el dato nuevo debe sufrir el mismo preproceso, y en el mismo orden, antes de pasárselo al modelo final escogido (ya que el modelo final se ha hecho sobre los datos preprocesados y no sobre los originales).

No obstante hay un elemento de sesgo que, inevitablemente, se cuela si se sigue el proceso descrito. Recordemos que algunas de las transformaciones se basan en información que proporciona la propia muestra de datos de dicha variable, lo que obliga, también para evitar sesgos, a calcular la transformación en base solamente al conjunto de entrenamiento (aunque luego se aplique tanto a entrenamiento como a test). Esta misma idea genera un pequeño problema cuando, como veremos ahora, se haga un remuestreo con entrenamiento y validación. El problema consiste en que, siendo puristas, cuando usamos validación, la información contenida en validación no debería haberse usado para hacer los calculos de algunos de los preprocesos, de manera análoga a entrenamiento y test. Caret, de hecho, sugiere que el preproceso no se haga antes de hacer el entrenamiento (que se explica en la sección 11) sino que se le pasen los métodos de pre-procesado como uno de los parámetros de dicho entrenamiento (se le pasa al comando `train()`, en el parámetro `preProcess`, una lista de los métodos de preproceso a usar idénticos a los admitidos en el comando `preProcess()`). En el proceso de remuestreo Caret calcula los valores del (los) método(s) de pre-proceso indicado(s) sobre cada remuestreo de entrenamiento (y se lo aplica al conjunto de validación de ese remuestreo). El problema es que la forma en que caret recibe el parámetro de pre-proceso no permite (al menos que yo conozca) un manejo tan fino como para seleccionar individualmente a qué variable aplicar qué transformación (o qué conjunto de transformaciones) sino que aplica los métodos indicados a todas las variables elegibles para dicho método. Así , p.e., si se le indica que escale y centre los datos lo hace sobre

Algorithm 1: Proceso de obtención de un modelo de predicción.

```
1  begin
2      Se dividen los datos en un conjunto de entrenamiento y otro de test.
3      repeat
4          Se restauran los conjuntos de entrenamiento y test.
5          repeat
6              Se toma decisión sobre el elemento de preproceso a aplicar (estudiado sobre el conjunto
                de entrenamiento solamente).
7              if se necesita calcular algo para aplicar el elemento de preproceso then
8                  Se calcula utilizando solamente el conjunto de entrenamiento
9              Se aplica el elemento de preproceso tanto al conjunto de entrenamiento como al de test.
10             Se guarda la información que permita aplicar este elemento en datos futuros.
11         until Se decide que los datos ya están suficientemente (pre)procesados
12         Se entrena/n uno o varios modelos (se buscan los mejores hiper-parámetros de cada modelo y
            se ajusta un modelo final con los mejores hiper-parámetros)
13         Se comparan los modelos finales y se escoge uno.
14         Se evalúa el modelo final escogido con el conjunto de test
15     until Se considera que el modelo es aceptable
16     if se necesita predecir un nuevo dato then
17         Se preprocesan los predictores del nuevo dato en el mismo orden, y de la misma forma, que el
            conjunto de entrenamiento que generó el modelo final escogido.
18         Se predice conforme al modelo final escogido.
```

todas las variables numéricas de entrada, sin poder indicarle un subconjunto de ellas. Tampoco se le podría indicar que haga una secuencia determinada de transformaciones (aunque puede hacer varias de ellas en un orden pre-establecido). El valor por defecto de este parámetro `preProcess` de `train()` es `NULL` (no hace transformaciones).

En principio creo que merece la pena sacrificar algo de sesgo a costa de tener mayor flexibilidad en las transformaciones a aplicar. No obstante, si se van a hacer transformaciones globales o simples, y que puede manejar el parámetro `preProcess` , es mejor pasárselas al comando `train()` puesto que los aplica de manera que evita ese sesgo en el remuestreo.

10.4.1 Ejemplo de transformación del conjunto de datos adult

Por último vamos a ver un ejemplo completo de como hacer un preproceso básico inicial (antes de comprobar si, realmente, mejora el modelado) con el conjunto `adult`.Veremos que se incluyen en el ejemplo algunas de las transformaciones vistas. El siguiente ejemplo está basado en una interesante página `http://scg.sdsu.edu/dataset-adult_r/` donde hace una transformación inicial bastante completa sobre `adult` que merece la pena repasar.

Primero vamos a trabajar sobre una variable auxiliar donde almacenaremos todos los cambios que realizaremos sobre la base de datos original. El tipo de transformaciones utilizadas nos permite trabajar directamente sobre todos los ejemplos (sin necesidad de dividir previamente en training y test).

```
# Guardaremos en adult.data el preproceso de este ejemplo
adult.data<-adult
```

Ahora vamos a analizar todas las variables que tiene el conjunto de datos `adult`:

Variable	Descripción
`age`	La edad del individuo
`type_employer`	El tipo de empleo que tiene el individuo, ya sea funcionario del gobierno, militar, privado, etc.
`fnlwgt`	Es una medida del número de personas que los que hicieron el censo consideraban que esta observación representaba. Se puede usar para ponderar el efecto individual de este ejemplo particular (en algoritmos que lo permiten). Esta variable se ignorará.
`education`	El nivel educativo más alto que alcanzó este individuo.
education_num	El nivel más alto de educación pero en forma numérica
`marital`	Estado civil del individuo.
`occupation`	El puesto de trabajo del individuo.
`relationship`	Incluye valores de relación como marido, padre, etc. aunque no hace referencia a quien/es. Es posible que sea un resto del origen de la base de datos, que incluiría información relacional entre individuos de la base de datos.
`race`	La raza de los individuos: Blanco, Negro, Esquimal, etc.
`sex`	Sexo biológico.
`capital_gain`	Ganancias de capital (en bolsa) declaradas por el individuo.
`capital_loss`	Pérdidas de capital (en bolsa) declaradas por el individuo.
`hr_per_week`	Horas trabajadas por semana.
`country`	País de origen del individuo.
`income`	Variable booleana (Clase). Si la persona gana o no más de 50.000$ al año.

Una vez hecho esto pasamos al pre-proceso propiamente dicho. Lo primero es eliminar dos variables:: `fnlwgt`, and `education_num`. La razón es que entorpecen el análisis. `education_num`, por ejemplo, es simplemente una copia de la información de `education`. Por otro lado `fnlwght` es un peso a utilizar para ponderar de manera diferente los ejemplos (hacer que unos tengan más influencia que otros) que podrían usarse en algunos algoritmos particulares pero no en los ejemplos generales que vamos a usar. Así que, sencillamente, se eliminan de los datos del data frame. Hay dos formas de eliminar una columna de un data frame, o bien indexando con $ o bien con [[]] (dobles corchetes cuadrados). Es muy importante usar el doble corchete cuadrado, y no el sencillo, porque devuelven cosas distintas (el simple devuelve **una lista con un elemento** que es la columna, y las dobles devuelven **la columna**) o se comete un error.

```
# Dos formas de eliminar una columna de un data frame
adult.data[["education_num"]]<-NULL
adult.data$fnlwgt<-NULL
```

Cuando se cargaron los datos con `read.table()` se transformaron en factores algunas variables que estaban almacenadas como cadenas de caracteres. Lo podemos comprobar:

```
> str(adult.data)
'data.frame': 32561 obs. of  13 variables:
 $ age          : int  39 50 38 53 28 37 49 52 31 42 ...
 $ type_employer: Factor w/ 8 levels "Federal-gov",..: 7 6 4 4 4 4 4 6 4 4 ...
 $ education    : Factor w/ 16 levels "10th","11th",..: 10 10 12 2 10 13 7 12 13 10 ...
```

```
 $ marital      : Factor w/ 7 levels "Divorced","Married-AF-spouse",..: 5 3 1 3 3 3 4 3 5 3 ...
 $ occupation   : Factor w/ 14 levels "Adm-clerical",..: 1 4 6 6 10 4 8 4 10 4 ...
 $ relationship : Factor w/ 6 levels "Husband","Not-in-family",..: 2 1 2 1 6 6 2 1 2 1 ...
 $ race         : Factor w/ 5 levels "Amer-Indian-Eskimo",..: 5 5 5 3 3 5 3 5 5 5 ...
 $ sex          : Factor w/ 2 levels "Female","Male": 2 2 2 2 1 1 1 2 1 2 ...
 $ capital_gain : int  2174 0 0 0 0 0 0 0 14084 5178 ...
 $ capital_loss : int  0 0 0 0 0 0 0 0 0 0 ...
 $ hr_per_week  : int  40 13 40 40 40 40 16 45 50 40 ...
 $ country      : Factor w/ 41 levels "Cambodia","Canada",..: 39 39 39 39 5 39 23 39 39 39 ...
 $ income       : Factor w/ 2 levels "<=50K",">50K": 1 1 1 1 1 1 1 2 2 2 ...
```

Como se van a hacer cambios en algunas de esas variables y se van a buscar patrones dentro de los textos, habrá que transformar esos factores (con los que vayamos a trabajar) en cadenas de caracteres para poder trabajar con ellas. Después las reconvertiremos de nuevo a factores.

```
adult.data$type_employer <- as.character(adult.data$type_employer)
adult.data$occupation <- as.character(adult.data$occupation)
adult.data$country <- as.character(adult.data$country)
adult.data$education <- as.character(adult.data$education)
adult.data$race <- as.character(adult.data$race)
adult.data$marital <- as.character(adult.data$marital)
```

Vamos a echar un vistazo a la frecuencia relativa de algunos valores dentro de una variable para ver si deberíamos bloquearlas (reorganizar las categorías para hacerlas más equilibradas).

```
> table(adult.data$type_employer)

     Federal-gov        Local-gov     Never-worked          Private     Self-emp-inc
             960             2093                7            22696             1116
Self-emp-not-inc        State-gov      Without-pay
            2541             1298               14
```

Se puede ver que tanto “Never-worked” como “Without-Pay” son grupos muy pequeños en comparación con los otros y además se parecen mucho en cuanto a lo que representan, quizá deban integrarse en una categoría más genérica “Not-Working”. También hay categorías que son un mismo grupo separados de manera más específica y quizá no merezca la pena tenerlos disgregados (como Self-emp). Los funcionarios quizá también se pueden unir en un solo grupo sin diferenciarlos entre federales, estatales y locales, o quizá en dos grupos, los federales por un lado, y los estatales y locales por otro. De esta forma se reducen bastante las categorías y, lo más importante, estas nuevas categorías están un poco más equilibradas respecto a su frecuencia. Tener variables con categorías muy poco representadas suele dar resultados pobres (aunque no siempre).

```
adult.data$type_employer <- gsub("^Federal-gov","Federal-Govt",adult.data$type_employer)
adult.data$type_employer <- gsub("^Local-gov","Other-Govt",adult.data$type_employer)
adult.data$type_employer <- gsub("^State-gov","Other-Govt",adult.data$type_employer)
adult.data$type_employer <- gsub("^Private","Private",adult.data$type_employer)
adult.data$type_employer <- gsub("^Self-emp-inc","Self-Employed",adult.data$type_employer)
```

```
adult.data$type_employer <- gsub("^Self-emp-not-inc","Self-Employed",adult.data$type_employer)
adult.data$type_employer <- gsub("^Without-pay","Not-Working",adult.data$type_employer)
adult.data$type_employer <- gsub("^Never-worked","Not-Working",adult.data$type_employer)
```

El comando `gsub()` busca, mediante expresiones regulares, subcadenas determinadas en una cadena de caracteres y las sustitye por otras subcadenas. Podemos ver ahora el resultado de nuestros cambios:

```
> table(adult.data$type_employer)

 Federal-Govt   Not-Working    Other-Govt       Private Self-Employed
          960            21          3391         22696          3657
```

Las categorías están ahora más equilibradas y tenemos menos. Veamos ahora otras categorías más y repitamos el proceso. Esta vez con `occupation`:

```
> table(adult.data$occupation)

     Adm-clerical      Armed-Forces      Craft-repair   Exec-managerial   Farming-fishing
             3770                 9              4099              4066               994
Handlers-cleaners Machine-op-inspct     Other-service   Priv-house-serv    Prof-specialty
             1370              2002              3295               149              4140
  Protective-serv             Sales      Tech-support  Transport-moving
              649              3650               928              1597
```

Para `occupation` quizá una forma sencilla de bloquear las categorías sería crear unas categorías que distinguieran entre trabajadores de oficina ("White-Collar" en inglés, por el color del cuello de la camisa) y trabajadores manuales ("Blue-Collar" en inglés, por el color del cuello del mono de trabajo). También se reagrupan los trabajos en la administración, en servicios, etc. Los que trabajan en las fuerzas armadas (un grupo de trabajadores muy especial en Estados Unidos) son muy pocos y, desgraciadamente, no se acomodan bien en otras categorías. Si se considera que deben permanecer como categoría (y no pasarlos a 'Òther-Occupations", p.e.) habrá que considerar aplicar sub-muestreos (los veremos en la sección 13), en particular up-sampling, para aumentar (artificialmente) su número y paliar el efecto de su bajísima frecuencia relativa.

```
adult.data$occupation <- gsub("^Adm-clerical","Admin",adult.data$occupation)
adult.data$occupation <- gsub("^Armed-Forces","Military",adult.data$occupation)
adult.data$occupation <- gsub("^Craft-repair","Blue-Collar",adult.data$occupation)
adult.data$occupation <- gsub("^Exec-managerial","White-Collar",adult.data$occupation)
adult.data$occupation <- gsub("^Farming-fishing","Blue-Collar",adult.data$occupation)
adult.data$occupation <- gsub("^Handlers-cleaners","Blue-Collar",adult.data$occupation)
adult.data$occupation <- gsub("^Machine-op-inspct","Blue-Collar",adult.data$occupation)
adult.data$occupation <- gsub("^Other-service","Service",adult.data$occupation)
adult.data$occupation <- gsub("^Priv-house-serv","Service",adult.data$occupation)
adult.data$occupation <- gsub("^Prof-specialty","Professional",adult.data$occupation)
adult.data$occupation <- gsub("^Protective-serv","Other-Occupations",adult.data$occupation)
adult.data$occupation <- gsub("^Sales","Sales",adult.data$occupation)
adult.data$occupation <- gsub("^Tech-support","Other-Occupations",adult.data$occupation)
```

```
adult.data$occupation <- gsub("^Transport-moving","Blue-Collar",adult.data$occupation)
```

El resultado que tenemos ahora es:

```
> table(adult.data$occupation)

           Admin       Blue-Collar          Military Other-Occupations      Professional
            3770             10062                 9              1577              4140
           Sales           Service      White-Collar
            3650              3444              4066
```

Ahora veamos si debemos bloquear la variable `country`:

```
> table(adult.data$country)

                  Cambodia                   Canada                    China
                        19                      121                       75
                  Columbia                     Cuba       Dominican-Republic
                        59                       95                       70
                   Ecuador                El-Salvador                  England
                        28                      106                       90
                    France                  Germany                   Greece
                        29                      137                       29
                 Guatemala                    Haiti       Holand-Netherlands
                        64                       44                        1
                  Honduras                     Hong                   Hungary
                        13                       20                       13
                     India                     Iran                   Ireland
                       100                       43                       24
                     Italy                  Jamaica                     Japan
                        73                       81                       62
                      Laos                   Mexico                 Nicaragua
                        18                      643                       34
Outlying-US(Guam-USVI-etc)                     Peru                Philippines
                        14                       31                      198
                    Poland                 Portugal                Puerto-Rico
                        60                       37                      114
                  Scotland                    South                   Taiwan
                        12                       80                       51
                  Thailand          Trinadad&Tobago            United-States
                        18                       19                    29170
                   Vietnam                Yugoslavia
                        67                       16
```

Es evidente que la variable country tiene un pequeño problema, y es que los Estados Unidos representa una inmensa mayoría de los individuos mientras que el resto de países tienen tan pocos ejemplos que sus contribuciones podrían no ser significativas. De nuevo parece necesario bloquear los países, agrupándolos

por zonas geográficas o geo-económicas (puesto que estamos tratando con un problema de base económica y quizá merezca la pena introducir ese matiz en escoger las nuevas categorías del bloqueo):

```
adult.data$country[adult.data$country=="Cambodia"] <- "SE-Asia"
adult.data$country[adult.data$country=="Canada"] <- "British-Commonwealth"
adult.data$country[adult.data$country=="China"] <- "China"
adult.data$country[adult.data$country=="Columbia"] <- "South-America"
adult.data$country[adult.data$country=="Cuba"] <- "Other"
adult.data$country[adult.data$country=="Dominican-Republic"] <- "Latin-America"
adult.data$country[adult.data$country=="Ecuador"] <- "South-America"
adult.data$country[adult.data$country=="El-Salvador"] <- "South-America"
adult.data$country[adult.data$country=="England"] <- "British-Commonwealth"
adult.data$country[adult.data$country=="France"] <- "Euro_1"
adult.data$country[adult.data$country=="Germany"] <- "Euro_1"
adult.data$country[adult.data$country=="Greece"] <- "Euro_2"
adult.data$country[adult.data$country=="Guatemala"] <- "Latin-America"
adult.data$country[adult.data$country=="Haiti"] <- "Latin-America"
adult.data$country[adult.data$country=="Holand-Netherlands"] <- "Euro_1"
adult.data$country[adult.data$country=="Honduras"] <- "Latin-America"
adult.data$country[adult.data$country=="Hong"] <- "China"
adult.data$country[adult.data$country=="Hungary"] <- "Euro_2"
adult.data$country[adult.data$country=="India"] <- "British-Commonwealth"
adult.data$country[adult.data$country=="Iran"] <- "Other"
adult.data$country[adult.data$country=="Ireland"] <- "British-Commonwealth"
adult.data$country[adult.data$country=="Italy"] <- "Euro_1"
adult.data$country[adult.data$country=="Jamaica"] <- "Latin-America"
adult.data$country[adult.data$country=="Japan"] <- "Other"
adult.data$country[adult.data$country=="Laos"] <- "SE-Asia"
adult.data$country[adult.data$country=="Mexico"] <- "Latin-America"
adult.data$country[adult.data$country=="Nicaragua"] <- "Latin-America"
adult.data$country[adult.data$country=="Outlying-US(Guam-USVI-etc)"] <- "Latin-America"
adult.data$country[adult.data$country=="Peru"] <- "South-America"
adult.data$country[adult.data$country=="Philippines"] <- "SE-Asia"
adult.data$country[adult.data$country=="Poland"] <- "Euro_2"
adult.data$country[adult.data$country=="Portugal"] <- "Euro_2"
adult.data$country[adult.data$country=="Puerto-Rico"] <- "Latin-America"
adult.data$country[adult.data$country=="Scotland"] <- "British-Commonwealth"
adult.data$country[adult.data$country=="South"] <- "Euro_2"
adult.data$country[adult.data$country=="Taiwan"] <- "China"
adult.data$country[adult.data$country=="Thailand"] <- "SE-Asia"
adult.data$country[adult.data$country=="Trinadad&Tobago"] <- "Latin-America"
adult.data$country[adult.data$country=="United-States"] <- "United-States"
adult.data$country[adult.data$country=="Vietnam"] <- "SE-Asia"
adult.data$country[adult.data$country=="Yugoslavia"] <- "Euro_2"
```

El criterio usado aquí, como acabamos de decir, es una combinación de localización geográfica, organi-

zación política y zonas económicas. `Euro_1` son países de la Eurozona que son más fuertes económicamente y, por tanto, la gente de esos países debería ser más rica. `Euro_2` serían países de la Eurozona menos prósperos o que tengan problemas financieros importantes como Portugal o Grecia. También se incluyen en esta categoría países europeos eslavos o que tuvieron gran influencia de la antigua Unión Soviética, como Polonia. Las antiguas colonias Británicas todavía conservan grandes lazos de unión con Gran Bretaña y se agruparían todas en "British-Commonwealth".

También se bloqueará la variable education. El objetivo es reducir el número de categorías en las variables categóricas. En algunos métodos esto simplifica mucho los cálculos (aparte de hacer más claros e inteligibles los modelos). Se puden bloquear (agrupar) todos los diferentes tipos de abandono escolar juntos. También se pueden agrupar a la gente que hizo el instituto (High School) con los que fueron algunos años a la Universidad (College) pero no la terminaron. Los distintos tipos de graduados se agruparán juntos igualmente.

Respecto a estado civil se bloquean distinguiendo solo entre personas que nunca se casaron, las casadas, las que estuvieron casdas y ya no lo están, y las personas viudas.

Las razas se mantienen igual pero se simplifican sus nombres:

```
adult.data$education <- gsub("^10th","Dropout",adult.data$education)
adult.data$education <- gsub("^11th","Dropout",adult.data$education)
adult.data$education <- gsub("^12th","Dropout",adult.data$education)
adult.data$education <- gsub("^1st-4th","Dropout",adult.data$education)
adult.data$education <- gsub("^5th-6th","Dropout",adult.data$education)
adult.data$education <- gsub("^7th-8th","Dropout",adult.data$education)
adult.data$education <- gsub("^9th","Dropout",adult.data$education)
adult.data$education <- gsub("^Assoc-acdm","Associates",adult.data$education)
adult.data$education <- gsub("^Assoc-voc","Associates",adult.data$education)
adult.data$education <- gsub("^Bachelors","Bachelors",adult.data$education)
adult.data$education <- gsub("^Doctorate","Doctorate",adult.data$education)
adult.data$education <- gsub("^HS-Grad","HS-Graduate",adult.data$education)
adult.data$education <- gsub("^Masters","Masters",adult.data$education)
adult.data$education <- gsub("^Preschool","Dropout",adult.data$education)
adult.data$education <- gsub("^Prof-school","Prof-School",adult.data$education)
adult.data$education <- gsub("^Some-college","HS-Graduate",adult.data$education)
adult.data$marital[adult.data$marital=="Never-married"] <- "Never-Married"
adult.data$marital[adult.data$marital=="Married-AF-spouse"] <- "Married"
adult.data$marital[adult.data$marital=="Married-civ-spouse"] <- "Married"
adult.data$marital[adult.data$marital=="Married-spouse-absent"] <- "Not-Married"
adult.data$marital[adult.data$marital=="Separated"] <- "Not-Married"
adult.data$marital[adult.data$marital=="Divorced"] <- "Not-Married"
adult.data$marital[adult.data$marital=="Widowed"] <- "Widowed"

adult.data$race[adult.data$race=="White"] <- "White"
adult.data$race[adult.data$race=="Black"] <- "Black"
adult.data$race[adult.data$race=="Amer-Indian-Eskimo"] <- "Amer-Indian"
adult.data$race[adult.data$race=="Asian-Pac-Islander"] <- "Asian"
adult.data$race[adult.data$race=="Other"] <- "Other"
```

Ahora se categorizarán las ganancias y las pérdidas. Este ejemplo es el mismo que vimos antes en la

sección 9.1. Se decide que una variable numérica se discretice, en vez de hacer algún tipo de transformación numérica. La razón es que la variable tiene una distribución tan asimétrica que posiblemente una transformación para reducirla no vaya a aportar mucho ni ser lo más apropiado. Entonces se puede discretizar la variable y bloquearla en tres categorías "None". "Low" y "High". Para ambas variables `None` significa que no tiene inversiones en bolsa. `Low` indicaría que tiene algunas inversiones, y `High` que tiene mucho dinero metido en bolsa. Hay que tener en cuenta, como vimos ya antes, que el tener grandes ganancias o pérdidas en bolsa está muy asociado a tener ingresos altos, básicamente porque para invertir en bolsa primero tienes que tener el dinero para hacerlo.

Quizá en este preproceso se debiera calcular la mediana sobre los datos del conjunto de entrenamiento solamente, pero para no complicar el ejemplo lo haremos sobre todo el conjunto.

```
# Intervalos donde discretizar para capital gain
cortes.cg<-c(-Inf, 0,median(adult.data[["capital_gain"]]
                                    [adult.data[["capital_gain"]] >0], na.rm=T),Inf)
# Intervalos donde discretizar para capital loss
cortes.cl<-c(-Inf, 0,median(adult.data[["capital_loss"]]
                                    [adult.data[["capital_loss"]] >0], na.rm=T),Inf)
# Se discretizan capital gain y loss
adult.data[["capital_gain"]] <- ordered(cut(adult.data$capital_gain,cortes.cg),
                                      labels = c("None", "Low", "High"))
adult.data[["capital_loss"]] <- ordered(cut(adult.data$capital_loss,cortes.cl),
                                      labels = c("None", "Low", "High"))
```

Merece la pena llamar la atención sobre el hecho de que haciendo todo este bloqueo estamos potencialmente perdiendo información que podría ser relevante. En algunos casos esa información parecía confusa o no estaba muy clara. En otros casos se ha optado por hacerlo al no tener, sencillamente, suficientes datos dentro de un grupo para tomar una decisión sobre dicho grupo al estar pobremente representado, así que se bloquean en grupos similares. En todo caso el bloqueo simplifica drásticamente los modelos a usar. En métodos basados en matemáticas, cuando se usan variables categóricas, el cálculo debe hacerse sobre las variables dummy que hemos visto antes, lo que implica una variable por cada nivel de la variable categórica. Es decir, que si tienes 2 variables categóricas, con 6 y 4 categorías respectivamente, los cálculos añaden 10 nuevas variables a la ecuación. En bases de datos como esta, donde la mayoría de las variables son categóricas, y cada una con bastantes niveles, es fácil ver como el número de variables en las ecuaciones del modelo se incrementan drásticamente. El bloqueo reduce este problema.

Este conjunto de datos tiene datos incompletos (missing data). Ya vimos en la sección 6.2 como se tratan. En este ejemplo sencillamente los ignoraremos. Hay algunos modelos que son capaces de trabajar con datos nulos o incompletos, pero hay otros que no. Si se van a comparar los dos tipos de modelos es mejor tratar los datos nulos (de cualquiera de las formas descritas) y usar el mismo conjunto de entrenamiento (ya sin nulos) para evitar sesgos en la comparación al estar, en realidad, entrenándose sobre dos conjuntos de datos diferentes (uno con nulos y otro sin ellos). Si solo trabajamos con modelos que acepten nulos no será necesario eliminarlos.

```
adult.data <- na.omit(adult.data)
```

Ya hemos acabado de modificar las variables categóricas, así que hay que reconvertirlas de nuevo a factores.

```
adult.data$marital <- factor(adult.data$marital)
adult.data$education <- factor(adult.data$education)
adult.data$country <- factor(adult.data$country)
adult.data$type_employer <- factor(adult.data$type_employer)
adult.data$occupation <- factor(adult.data$occupation)
adult.data$race <- factor(adult.data$race)
adult.data$sex <- factor(adult.data$sex)
adult.data$relationship <- factor(adult.data$relationship)
```

También vamos a cambiar las etiquetas de la variable income ya que están almacenadas como "¿=50k" y "¡50k" y podría dar problemas la etiqueta al empezar con un símbolo de mayor o menor (p.e., no se podrían usar comandos condicionales). Así que cambiaremos las etiquetas accediendo a sus nombres mediante el comando levels. Primero comprobamos las etiquetas que ya tiene y las sustituimos por otras más adecuadas (en este caso usaremos "No" para indicar que no se gana más de 50.000$ y "Yes" para indicar que si).

```
> levels(adult.data$income)
[1] "<=50K" ">50K"
> levels(adult.data$income)<-c('No','Yes')
```

For the two remaining variables in the dataset, I choose to scale them. This applies a normal transformation. Each value minus its mean over the sample standard deviation. Of course, each sample will have a different mean and standard deviation, so if we wanted to apply any trained model to new data, we would have to make note of what this particular sample's mean and standard deviation are so as to apply the same transformation. It's worth noting that neural networks require numerical input variables to be scaled.

Por último vamos a transformar un par de variables más. La primera es `age`, que si la visualizamos con un histograma vemos que sigue una distribución normal (pero con un corte en el ala izquierda al no incluir valores inferiores a 17 años). Eso sugiere que se le aplique la transformación *scale*, que recordemos que normalizaría la distribución normal subyacente.

```
adult.data.age.preProc.Scale<-preProcess(adult.data['age'],method=c("center","scale"))
adult.data<- predict(adult.data.age.preProc.Scale,adult.data)
```

Insisto en que no debemos olvidar que, a partir de ahora, cuando nos llegase un nuevo dato a los posibles modelos obtenidos a partir de este conjunto transformado de datos, deberemos aplicar los mismos cambios que hemos estado aplicando. Deberemos aplicar este `predict()` al nuevo dato antes de pasárselo al modelo, al igual que todos los bloqueos y la categorización de los capital Gain/Loss.

Para acabar vamos a ver otra variable curiosa en su comportamiento, la de `hr_per_week`. Si le hacemos un histograma enriquecido (Figura 25) veremos que su densidad tiene unos curiosos máximos a ciertos intervalos. Esto apunta que que el histograma por defecto no tiene suficiente precisión para describir realmente como están distribuídas las frecuencias.

```
hist(adult.data$hr_per_week, xlab="",
     main="Hrs per week",
     ylim=c(0,1.2*max(density(adult.data$hr_per_week,na.rm=T)$y)),
     probability=T)
```

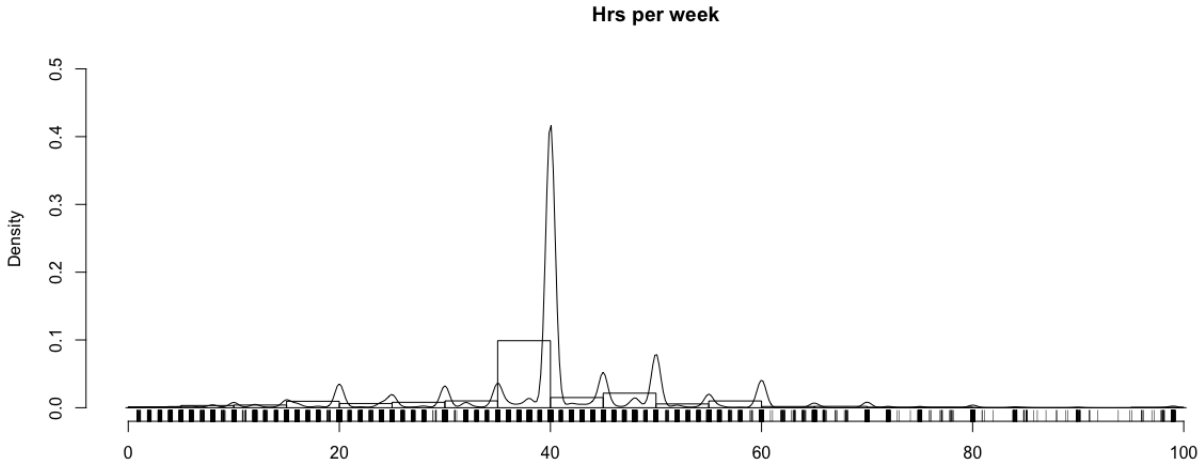


Figure 25: Histograma enriquecido de horas semanales de trabajo en el dataset Adult (granularidad gruesa).

```
lines(density(adult.data$hr_per_week,na.rm=T))
rug(jitter(adult.data$hr_per_week))
```

Echemos un vistazo a sus frecuencias por valor usando el comando `table()`.

```
> table(adult.data$hr_per_week)

   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
  20   32   39   54   60   64   26  145   18  278   11  173   23   34  404
  16   17   18   19   20   21   22   23   24   25   26   27   28   29   30
 205   29   75   14 1224   24   44   21  252  674   30   30   86    7 1149
  31   32   33   34   35   36   37   38   39   40   41   42   43   44   45
   5  266   39   28 1297  220  149  476   38 15217   36  219  151  212 1824
  46   47   48   49   50   51   52   53   54   55   56   57   58   59   60
  82   49  517   29 2819   13  138   25   41  694   97   17   28    5 1475
  61   62   63   64   65   66   67   68   70   72   73   74   75   76   77
   2   18   10   14  244   17    4   12  291   71    2    1   66    3    6
  78   80   81   82   84   85   86   87   88   89   90   91   92   94   95
   8  133    3    1   45   13    2    1    2    2   29    3    1    1    2
  96   97   98   99
   5    2   11   85
```

No es tan fácil de apreciar examinando la tabla numérica, pero si os fijáis, se observa que en ciertos valores hay varios picos de frecuencia. Vamos a hacer un histograma que tenga un bin para cada valor diferente (vemos que el rango va de 1 a 99 horas trabajadas) que sería el grano más fino posible (aunque podríamos ejecutar el comando `barplot(table(adult.data$hr_per_week))` para ver rápidamente lo mismo).

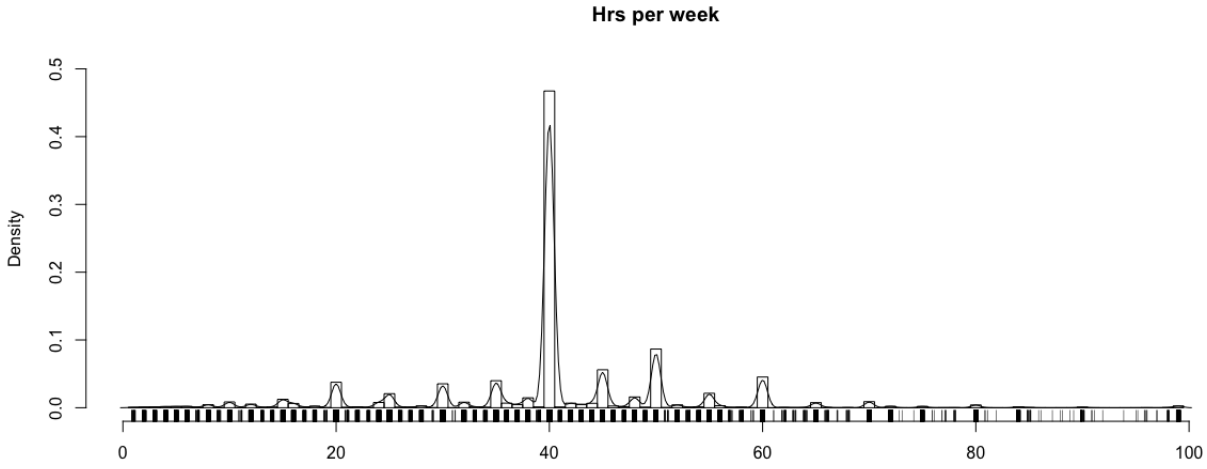


Figure 26: Histograma enriquecido de horas semanales de trabajo en el dataset Adult (granularidad fina).

```
hist(adult.data$hr_per_week, xlab="",
     breaks = c(0.5:99.5),
     main="Hrs per week",
     ylim=c(0,1.2*max(density(adult.data$hr_per_week,na.rm=T)$y)),
     probability=T)
lines(density(adult.data$hr_per_week,na.rm=T))
rug(jitter(adult.data$hr_per_week))
```

Ahora, como se ve en la figura 26, se aprecian más facilmente los picos que nos sugería la función de densidad. Hay un claro máximo a las 40 horas semanales (que será la jornada semanal general en Estados Unidos) y luego vemos ciertos picos en 20, 25, 30, 35, 45, 50, 55 y 60 horas. Eso nos indica que los contratos tienden a redondear las horas. Si decidiesemos, por ejemplo, categorizar esta variable, esta información sería muy valiosa para decidir los puntos de corte y las categorías. Por ejemplo, podríamos crear la siguiente categorización:

Valor	Descripción
<= 20	Menos de media jornada
>20 y <40	Entre media jornada y jornada completa
=40	Jornada completa
>40 y <=50	Hasta 50 horas semanales
>50 y <=60	Hasta 60 horas semanales
>60	Más de 60 horas semanales

Si queremos hacer ahora la categorización usaremos un pequeño truco para usar con comodidad el comando `cut()`. Como tenemos la categoría "especial" de jornada completa (con exactamente 40 horas) y el `cut()` hace la división con todos los intervalos semiabiertos, cambiamos las 40 horas a 40.5 horas y hacemos los cortes de modo que nos cree una categoría entre 40 y 40.5 (cerrada a la derecha) y así transformamos

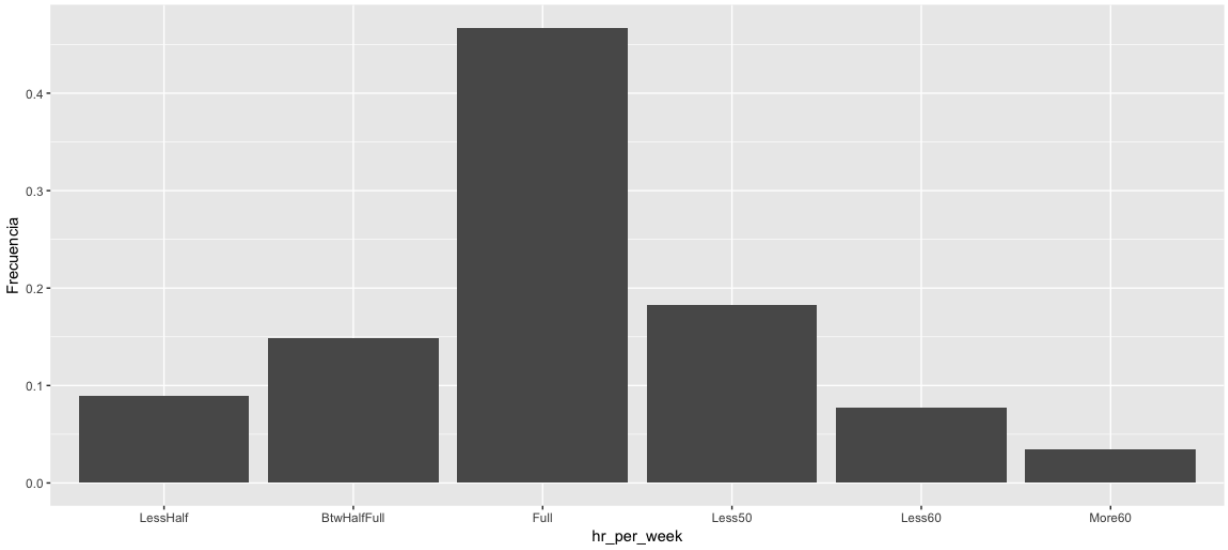


Figure 27: Histograma de horas semanales de trabajo en el dataset Adult (categorizado).

con facilidad. Hacemos el factor ordenado y ponemos etiquetas a nuestro gusto.

```
trick<-adult.data$hr_per_week
trick[trick==40] <- 40.5
adult.data$hr_per_week<- ordered(cut(trick,breaks=c(0,20,40,40.5,50,60,100)),
        labels=c("LessHalf","BtwHalfFull","Full","Less50","Less60","More60"))
```

Si ahora hacemos un histograma rápido de la variable observaremos una forma bastante familiar, como se aprecia en la figura 27.

```
plot(adult.data$hr_per_week)
# Comandos alternativos (con el total de ejemplos)
barplot(table(adult.data$hr_per_week))
ggplot(adult.data, aes(x=hr_per_week))+geom_bar()
# Comandos alternativos (con la frecuencia)
barplot(table(adult.data$hr_per_week)/nrow(adult.data))
ggplot(adult.data, aes(x=hr_per_week))+geom_bar(aes(y=..count../sum(..count..)))+labs(
   y="Frecuencia")
```

Con esto acabamos este ejemplo y la tercera sesión. En la próxima empezaremos a entrenar modelos.

---

11 Entrenando el modelo y ajustando hiperparámetros

Ahora entramos en la parte de entrenar modelos potencialmente útiles para predecir la variable de salida. En principio hay cuatro pasos a llevar a cabo:

1. Decidir que algoritmos de Machine Learning utilizar.
2. Encontrar cuales son los mejores hiper-parámetros de dichos algoritmos para nuestros datos
3. Ajustar un modelo con dichos mejores hiper-parámetros.
4. Comparar los modelos obtenidos con cada algoritmo y decidir con cual nos quedamos.

11.1 Algoritmos de Machine Learning (Models) que proporciona caret

Caret proporciona, en el momento de escribir este tutorial , 238 algoritmos de Machine Learning con un interfaz común. Se puede ver un listado de todos ellos con el comando `names(getModelInfo())`. También se puede encontrar información adicional en `http://topepo.github.io/caret/available-models.html`.

Vamos a utilizar algunos de ellos para hacer las pruebas de los siguientes apartados. En particular probaremos:

Modelo	Lineal	Tipo	Descripción
lda	si	Sencillo	Linear Discriminant Analysis
glm	si	Sencillo	Generalized Linear Model
rpart	no	Sencillo	El algoritmo CART de árboles de decisión de clasificación y regresión.
knn	no	Sencillo	Un algoritmo de clustering adaptado para hacer clasificación.
svmRadial	no	Complejo	Un algoritmo de Support Vector Machines
rf	no	Complejo	Random Forests, un algoritmo que combina múltiples árboles de regresión.
gbm	no	Complejo	Stochastic Gradient Boosting
nnet	no	Complejo	Neural Networks

Caret proporciona un interfaz común para todos esos métodos y entrenar cualquier de ellos es tan sencillo como ejecutar:

```
# predictores son los valores de las variables de entrada de los datos de entrenamiento
#    p.e. datos.Train[Vars.Entrada.Usadas]
# valores.Conocidos son los valores de salida de los datos de entrenamiento
#    p.e. datos.Train[[Var.Salida.Usada]]
modelo.lda<-train(predictores, valores.Conocidos, method='lda')
modelo.glm<-train(predictores, valores.Conocidos, method='glm')
modelo.rpart<-train(predictores, valores.Conocidos, method='rpart')
```

Un ejemplo con el dataset de la diabetes de los Indios Pima sería:

```
library(caret)
library(mlbench)
data("PimaIndiansDiabetes")
# Usamos el dataset PimaIndiansDiabetes y hacemos una partición al 80%
pima.Datos.Todo<-PimaIndiansDiabetes
```

```
pima.Var.Salida.Usada<-c("diabetes")
pima.Vars.Entrada.Usadas<-setdiff(names(pima.Datos.Todo),pima.Var.Salida.Usada)

set.seed(1234)
pima.TrainIdx.80<- createDataPartition(pima.Datos.Todo[[pima.Var.Salida.Usada]],
                                p=0.8,
                                list = FALSE,
                                times = 1)
pima.Datos.Train<-pima.Datos.Todo[pima.TrainIdx.80,]
pima.Datos.Test<-pima.Datos.Todo[-pima.TrainIdx.80,]

set.seed(1234)
pima.modelo.bstrp25.lda<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                pima.Datos.Train[[pima.Var.Salida.Usada]],
                                method='lda')
set.seed(1234)
pima.modelo.bstrp25.glm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                pima.Datos.Train[[pima.Var.Salida.Usada]],
                                method='glm')
set.seed(1234)
pima.modelo.bstrp25.rpart<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                  pima.Datos.Train[[pima.Var.Salida.Usada]],
                                  method='rpart')
# El siguiente puede tardar un poco de tiempo en ejecutarse
set.seed(1234)
pima.modelo.bstrp25.rf<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                               pima.Datos.Train[[pima.Var.Salida.Usada]],
                               method='rf')
```

El resultado del modelo final ajustado de dichos modelos, tras evaluar varios hiper-parámetros, se puede visualizar rápidamente con el comando `print()` (también vale directamente escribiéndo el nombre de la variable donde almacenamos el modelo), que muestra el nombre del modelo, información sobre los datos de entrenamiento, si ha existido preproceso (si se le indicó a `train()` en el parámetro `preProcess`), el tipo de remuestreo utilizado (ver sección 11.2.2) con los respectivos tamaños de éstos y el resultado de la evaluación de las distintas configuraciones de hiper-parámetros usando dichos remuestreos. Se muestran dos medidas típicas de clasificación (ver sección 11.2.3), `Accuracy` y `Kappa`. Se informa de cual de las medidas se utilizó para decidir la mejor configuración. Por último, incluye incluye la configuración final de hiper-parámetros utilizada en el modelo final:

```
> pima.modelo.bstrp25.rpart
CART

615 samples
  8 predictor
  2 classes: 'neg', 'pos'
```

```
No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 615, 615, 615, 615, 615, 615, ...
Resampling results across tuning parameters:

  cp          Accuracy   Kappa
  0.02093023  0.7342479  0.3993597
  0.02558140  0.7372226  0.3969645
  0.30232558  0.7019374  0.2126772

Accuracy was used to select the optimal model using  the largest value.
The final value used for the model was cp = 0.0255814.
```

Como los modelos glm y lda no tienen hiper-parámetros (ver sección 11.2) la información sobre configuraciones no aparece.

```
> pima.modelo.bstrp25.lda
Linear Discriminant Analysis

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 615, 615, 615, 615, 615, 615, ...
Resampling results:

  Accuracy   Kappa
  0.7648633  0.4486478
```

Es fácil comprobar que los resultados son la evaluación hecha sobre el total de los remuestreos puesto que si se calcula, p.e., el Accuracy (exactitud) sobre el conjunto de datos de entrenamiento usándo el modelo final nos aparece un valor distinto.

```
> MLmetrics::Accuracy(predict(pima.modelo.bstrp25.lda,
+                             pima.Datos.Train[pima.Vars.Entrada.Usadas]),
+                     pima.Datos.Train[[pima.Var.Salida.Usada]])
[1] 0.7739837
```

11.2 Encontrar los mejores hiper-parámetros y ajustar modelo final

La mayor parte de métodos tienen hiper-parámetros que afectan tanto a la estructura del modelo final como a la forma en que el algoritmo de ajuste funciona. Para conocer qué hiper-parámetros (a los hiper-parámetros, en la terminología de caret, se les llama model parameters) son directamente accesibles por caret se puede usar el comando modelLookup().

```
> 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
```

Este comando nos da información sobre los hiperparámetros a ajustar, su nombre, y si el modelo/algoritmo se puede usar para clasificación y/o regresión En el caso de `gbm` se puede ver que tiene cuatro parámetros controlables directamente por `caret`: `n.trees`, `interaction.depth`, `shrinkage` y `n.minobsinnode`. Para saber qué es cada uno de dichos parámetros debe irse a la documentación del paquete que contiene ese modelo (en este caso el paquete `gbm` cuyo `vignette` se puede conseguir en `https://cran.r-project.org/web/packages/gbm/gbm.pdf`) y localizar la función que realiza el entrenamiento de esa implementación del modelo (en este caso la función `gbm.fit()`).

Es importante comprobar que muchas veces los modelos tienen más hiper-parámetros posibles que los que caret pone a disposición directa del usuario de la librería (p.e. en `gbm` tendríamos, además de los arriba expuestos y accesibles directamente a través de caret, otros hiper-parámetros adicionales como `distribution` o `bag.fraction`). Más adelante veremos como podemos acceder a ellos si también quisieramos controlarlos.

Cuando utilizamos `train()` caret hace primero una optimización de los valores de los hiper-parámetros, usando parte de datos de entrenamiento como conjunto de validación siguiendo alguno de los métodos de remuestreo que veremos en la sección 11.2.2 (hace varios remuestreos por cada combinación para estimar un rendimiento medio). Después, una vez encontrada la mejor combinación o configuración de ellos, hace un modelo final. Este modelo final lo entrena utilizando todos los datos de entrenamiento proporcionados (lo que llama final model). Este proceso sigue el algoritmo que se describe en la figura 28.

```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

Figure 28: Algoritmo de ajuste de hiper-parámetros por remuestreo (resampling), y ajuste de modelo final de Caret.

Como se puede comprobar realiza varias iteraciones con diferente conjunto de parámetros para determinar cual es la mejor combinación de (hiper-)parámetros del modelo. En cada una de esas iteraciones, y para cada configuración particular de hiper-parámetros del modelo, deja de lado un grupo de datos para usarlos de validación, luego ajusta al modelo con el resto de datos de entrenamiento, y calcula una estimación de la calidad del modelo obtenido con el conjunto de validación que dejó apartado (ejemplos que no ha visto

al entrenar esa iteración específica). Después cambia el conjunto de validación y vuelve a ajustar el modelo con la misma configuración de parámetros de modelo pero con diferente conjunto entrenamiento/validación. Estas son las "resampling interation" de la figura 28 con lo que se obtiene una muestra de la calidad de dicha configuración de parámetros del modelo. A partir de esa muestra de "hold-out predictions" se obtiene una estimación de lo bueno que sería un modelo entrenado con dicha configuración de hiper-parámetros. Después repite el proceso para varias configuraciones de hiper-parámetros. Una vez que se han probado todos las combinaciones de hiper-parámetros que queremos estudiar se escoge la mejor configuración en base a las estimaciones calculadas. Finalmente, una vez escogida la mejor configuración de hiper-parámetros, se entrena (ajusta) un único modelo final con dichos hiper-parámetros, pero esta vez usando **todos** los datos de entrenamiento para ello (sin conjuntos de validación alguno).

11.2.1 El comando trainControl

Aunque el proceso de entrenamiento es automático se pueden controlar muchos de sus elementos. Por ejemplo, la técnica por defecto de remuestreo es "bootstrap" (como vimos al mostrar `pima.modelo.rpart`), pero se puede cambiar y utilizar otra, como la popular "repeatedcv" (Crosvalidación de k pliegues con repetición). Esto se consigue con el parámetro `trControl` que se construye con la función `trainControl()`.

Por ejemplo:

```
pima.TrainCtrl.3cv10 <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",
  number = 10,
  ## con 3 repeticiones
  repeats = 3,
  # Que muestre información mientras entrena
  verboseIter=T)

set.seed(1234)
pima.modelo.3cv10.lda<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                             pima.Datos.Train[[pima.Var.Salida.Usada]],
                             method='lda', trControl=pima.TrainCtrl.3cv10)
set.seed(1234)
pima.modelo.3cv10.glm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                             pima.Datos.Train[[pima.Var.Salida.Usada]],
                             method='glm', trControl=pima.TrainCtrl.3cv10)
set.seed(1234)
pima.modelo.3cv10.rpart<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                               pima.Datos.Train[[pima.Var.Salida.Usada]],
                               method='rpart', trControl=pima.TrainCtrl.3cv10)
set.seed(1234)
pima.modelo.3cv10.rf<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                               pima.Datos.Train[[pima.Var.Salida.Usada]],
                               method='rf', trControl=pima.TrainCtrl.3cv10)
```

El comando `trainControl()` se puede utilizar también para tener un control más fino del entrenamiento. En realidad tiene un número considerable de parámetros con los que controlar muchos elementos. Mencionaremos algunos de los más útiles a continuación.

11.2.2 Sobre-entrenamiento (overfitting) y remuestreo (resampling)

Uno de los principales problemas con los que nos podemos encontrar a la hora de encontrar el mejor conjunto de hiperparámetros es caer en el sobre-entrenamiento, que sucede cuando un modelo se adapta demasiado a los datos y captura tendencias individuales que no generalizan a datos nuevos datos (básicamente terminan memorizando los datos individuales más que capturando los patrones comunes que relacionan a grupos de ellos).

Por fortuna este efecto se puede detectar cuando el conjunto de entrenamiento (el que usa para crear el modelo) tiene buenos resultados mientras un conjunto de comprobación (datos que no se usan para entrenar el modelo) tiene resultados muy pobres (es decir, que los resultados de entrenamiento no se reproducen en datos no vistos).

Algunos hiperparámetros son responsables de que se llegue a esa situación de sobre-entrenamiento (p.e. el número de splits de los modelos de árboles, o el parámetro k, de los clusterings) y escoger los hiperparámetros adecuados es fundamental para tener un buen modelo final (y ocupará gran parte del trabajo computacional).

Dado que no podemos hacer uso del conjunto test (si lo hiciésemos se caería también en sobre-entrenamiento) y dado que necesitamos conjuntos de comprobación para detectar el sobre-entrenamiento, (sin olvidar que debemos realizar bastantes pruebas sobre cada conjunto de hiper-parámetros para poder tener una estimación aceptable sobre su "adecuación" para el problema que estamos resolviendo) necesitamos usar algún método de remuestreo (resampling).

Los métodos de remuestreo permiten introducir cierta variación en los conjuntos de entrenamiento (lo que permite estimar mejor el comportamiento general de una configuración particular de hiper-parámetros) mientras que proporciona conjuntos de comprobación (validación) para hacer dicha estimación de como será el rendimiento del modelo sobre ejemplos no vistos. También permite detectar si hay mucha diferencia entre resultados de entrenamiento y validación y desestimar configuraciones con pobre validación (para prevenir el overfitting).

Caret proporciona varias formas de remuestreo que pasamos a describir.

11.2.2.1 Formas de muestreo que ofrece trainControl

Cada una de los siguientes métodos se selecciona con el parámetro `method` del comando `trainControl()`.

K-fold Crossvalidation En esta técnica se dividen los datos en K bloques diferentes (pliegues) de más o menos igual tamaño (preferentemente división estratificada). Después se deja uno de ellos a un lado como conjunto de validación y se entrena el modelo con los *K-1* restantes como datos de entrenamiento. Se evalúa el modelo con el conjunto de validación. Luego se repite el proceso con distintos conjuntos como bloque de validación y se obtienen K modelos con K resultados de validación. El rendimiento se basa en las predicciones hechas sobre esos resultados de validación. Lo más común es que K sea 5 o 10.

El método `repeated K-fold CV` lo que hace es repetir varias veces el proceso y tener varias versiones de los pliegues. Es el método más frecuente.

Bootstrapping Con bootstraping se utiliza una muestra aleatoria con reemplazamiento. La muestra aleatoria es del mismo tamaño que el conjunto original. Como las muestras se pueden seleccionar más de una vez se puede calcular que cada dato original tiene un 63.2% de probablilidades de estar, al menos una vez, en el conjunto seleccionado. Los datos que no son finalmente seleccionados ni una sola vez se utilizan como conjunto de validación. Este proceso se repite varias veces (entre 30 y 100).

Es el método por defecto de caret. Tiene algo menos de varianza que k-fold pero algo de sesgo no-zero.

Leave Group Out En este método se crean aleatoriamente un grupo de entrenamiento (p.e. el 80% del conjunto original) y otro de validación (el 20% restante) cada vez. El proceso se repite entre 20 y 100 veces. Los grupos se escogen de manera estratificada.

Tiene algo menos de varianza que k-fold pero algo de sesgo.

11.2.2.2 Seleccionando manualmente los datos de validación

Los índices de los datos que se apartan para validación dentro de cada remuestreo normalmente se generan dentro de `train()` a partir del generador de números aleatorios. Por lo general esto es suficiente para tener buenas estimaciones. No obstante, a veces puede ser interesante predeterminarlos de antemano (p.e. usar exactamente los mismos subconjuntos de validación para diferentes algoritmos). Así pues se pueden especificar de antemano en `trainControl()` mediante el parámetro `index`.

11.2.2.3 Controlando las semillas de números aleatorios de cada remuestreo

Otro elemento importante de trainControl es el parámetro `seeds` que nos permite controlar las semillas del generador de números aleatorios que se usarán en cada iteración de remuestreo. La utilidad reside en que, como ya hemos comentado, si se quiere reproducir exáctamente el entrenamiento de un modelo, se debe hacer un `set.seed()` justo antes de ejecutar `train()` para inicializar el generador de números pseudo-aleatorio y eso decide, entre otras cosas, tanto los índices de los conjuntos de validación como también el uso de números aleatorios interno de cada algoritmo de ajuste de los modelos de cada remuestreo. Normalmente esto es suficiente para conseguir que se pueda reproducir la misma ejecución en otro momento, siempre y cuando `train()` se ejecute de manera **secuencial**.

El problema de reproducibilidad aparece si se ejecuta en **paralelo**, puesto que algunos de los modelos a ajustar en cada remuestreo irán a un hilo/proceso específico en función de la disponibilidad y ya no se puede saber en cada uno de dichos procesos por donde iba la "semilla". Así que, si se usa la capacidad de ejecución en paralelo (algo muy útil), se perdería la posibilidad de reproducir exactamente los experimentos si lo único que hiciesemos fuese ejecutar un `set.seed()` justo antes de ejecutar `train()`.

La forma de solucionar el problema es proporcionarle las semillas de cada remuestreo a `train()` mediante el parámetro `seeds`. Se debe entender que el ejecución concurrente en caret no se produce dentro de un ajuste (entrenamiento) particular de un modelo, es decir, no hay una ejecución concurrente en la entrenamiento de una configuración de hiper-parámetros del algoritmo de ajuste. Lo que se paraleliza es que, como hay muchas configuraciones a ajustar (además, cada remuestreo es un ajuste diferente), lo que va a un procesador libre es cada llamada independiente al algoritmo de ajuste. Dicho de otro modo, cada entrenamiento de un modelo para remuestreo se ejecutaría en un solo procesador (secuencialmente). Gracias al parámetro `seeds`, cuando se va a usar un nuevo procesador para entrenar una configuración, se le pasa también la semilla a fijar antes de ejecutar el ajuste que va a dicho procesador. De este modo se consigue que, independientemente de como se asignen los remuestreos/entrenamientos a los procesadores, en cada ajuste se podrá controlar que inicialice los números aleatorios de forma controlada y, por tanto, reproducible.

La variable que llamamos `combHParams` tiene que ver con cuantas combinaciones diferentes de hiper-parámetros se van a comprobar en cada experimento. Esto se controla con los parámetros `tuneLegth` (ver sección 11.2.4.1) y `tuneGrid` (ver sección 11.2.4).La variable `SeedsLength` debe tener un valor igual al número de repeticiones por el número de remuestreos más uno (la semilla única del modelo final). Si usamos `n_folds` para almacenar el número de pliegues y `n_reps` para almacenar el número de repeticiones, entonces

`SeedsLength=(n_folds*n_reps) + 1`. La variable `seeds` será una lista de vectores de enteros que se usarán para fijar la semilla aleatoria en cada iteración de remuestreo. Un valor de `NA` hará que no se fije una semilla, mientras que un valor de `NULL` hará que se fijen semillas aleatorias. La lista tiene que tener B+1 elementos, donde B es elnúmero de remuestreos (pliegues * repeticiones) más 1. Los primeros B elementos de la lista deben ser vectores de enteros de longitud `combHParams` que, recordemos es el número de combinaciones de hiperparámetros que se evalúan por remuestreo. El último elemento de la lista tiene que ser un único valor entero (será la semilla del modelo final, tras hacer la búsqueda de hiperparámetros así que solo se necesita una única semilla, y no M).

Aquí se muestra un ejemplo de 10 remuestreos repetidos 3 veces y con 3 combinaciones de hiperparámetros:

```
# Ejemplo de uso de seeds para poder reproducir experimentos usando varios procesadores
library(doParallel); library(caret)
set.seed(1234)
n_folds=10
n_reps=3
# seedsLength es = (numero_pliegues*numero_repeticiones)+1, en este caso (3*10)+1
seedsLength=(n_folds*n_reps)+1
seeds <- vector(mode = "list", length = seedsLength)
# Crearemos unos pliegues para usar los mismos en todos los modelos diferentes
foldIndexes<-createMultiFolds(pima.Datos.Train[[pima.Var.Salida.Usada]],k=n_folds,times=n_reps)
# combHParam es el número de combinaciones de hiper-parámetros a probar
# en el ejemplo,para random forest, solo se varía por defecto el hiperparámetro mtry
# para este problema se prueba mtry={2,5,8}, es decir, 3 combinaciones diferentes.
combHParam=3
for(i in 1:(seedsLength-1)) seeds[[i]]<- sample.int(n=1000, combHParam)
#Hay que crear una semilla única para el modelo final a entrenar.
seeds[[seedsLength]]<-sample.int(1000, 1)
# TrainControl con seeds
pima.TrainCtrl.3cv10.plRF <- trainControl(method='repeatedcv',
                                          number = n_folds,
                                          repeats = n_reps,
                                          index = foldIndexes,
                                          seeds=seeds
)
# TrainControl sin seeds
pima.TrainCtrl.3cv10.plRF.ns <- trainControl(method='repeatedcv',
                                             number = n_folds,
                                             index = foldIndexes,
                                             repeats = n_reps
)
# Ejecutar el modelo en paralelo (se deja un core siempre libre o se "congela" la consola)
cl <- makeCluster(detectCores()-1)
registerDoParallel(cl)
```

```
# Haremos 3 modelos, 2 con la misma semilla en seeds y 1 sin el seeds y veremos la diferencia
set.seed(1234)
pima.modelo.3cv10.rf.pl1<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                pima.Datos.Train[[pima.Var.Salida.Usada]],
                                method='rf', trControl=pima.TrainCtrl.3cv10.plRF)
set.seed(1234)
pima.modelo.3cv10.rf.pl2<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                pima.Datos.Train[[pima.Var.Salida.Usada]],
                                method='rf', trControl=pima.TrainCtrl.3cv10.plRF)
set.seed(1234)
pima.modelo.3cv10.rf.plNS<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                 pima.Datos.Train[[pima.Var.Salida.Usada]],
                                 method='rf', trControl=pima.TrainCtrl.3cv10.plRF.ns)
# Volvemos al modo no paralelo
stopImplicitCluster()
stopCluster(cl)
registerDoSEQ()
```

Comprobaréis que, dependiendo de los procesadores que tenga vuestra máquina, el tiempo de ejecución mejora ostensiblemente (y eso que entrena 3 modelos diferentes). Si ahora comprobamos los resultados de los 3 modelos:

```
> pima.modelo.3cv10.rf.pl1
Random Forest

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 554, 554, 553, 554, 553, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.7570774  0.4452402
  5     0.7608761  0.4589229
  8     0.7581527  0.4509480

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 5.
> pima.modelo.3cv10.rf.pl2
Random Forest

615 samples
  8 predictor
```

```
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 554, 554, 553, 554, 553, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.7570774  0.4452402
  5     0.7608761  0.4589229
  8     0.7581527  0.4509480

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 5.
> pima.modelo.3cv10.rf.plNS
Random Forest

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 554, 554, 553, 554, 553, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.7581791  0.4466374
  5     0.7619602  0.4602648
  8     0.7608408  0.4588072

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 5.
```

se ve que los modelos `pima.modelo.3cv10.rf.1` y `pima.modelo.3cv10.rf.2` son idénticos mientras que el modelo `pima.modelo.3cv10.rf.plNS` no lo es (a pesar de haberse reseteado la semilla de números aleatorios al mismo valor `set.Seed(1234)` antes de entrenar cada uno de los 3 modelos. Así que, para poder reproducir los resultados ,si usamos varios procesadores, es **necesario** usar `seeds`.

11.2.3 Cambiando medidas de rendimiento

En los ejemplos anteriores hemos visto que el modelo final se escogía con la medida de evaluación `Accuracy` (exactitud), pero se puede controlar mediante trainControl() las medidas que se utilizan para medir el rendimiento del modelo.

En realidad, lo que hace Caret, cuando evalua un modelo, es utilizar lo que denomina una *summary function* (una función que se puede especificar mediante el parámetro `summaryFunction` de `trainControl()`),

y que calcula diversas medidas de rendimiento. A `train()` se le indica cual de las medidas de las summary function debe usar para escoger el modelo final mediante el parámetro `metric` de `train()`.

La summary function que utiliza por defecto Caret se llama `postResample()`, a la que se le pasan dos vectores de igual longitud con los valores predichos y los valores verdaderos (observados). Si son numéricos, es decir, si el problema es de regresión, calcula 3 medidas: la media cuadrática del error ("RMSE" Root Mean Square Error), el R^2 simple ("Rsquared"), y la media absoluta del error (MAE Mean Absolute Error). Todas ellas son medidas que se utilizan con frecuencia para evaluar el rendimiento de la regresión.

Para los problemas de clasificación (los reconoce si los vectores que se le pasan son factores) calcula 2 medidas: La exactitud ("Accuracy") y el coeficiente Kappa de Cohen ("Kappa"). La medida "Accuracy" puede dar problemas con clases desbalanceadas (los modelos tienden a ignorar clases con pocos ejemplos) por lo que en esos casos es aconsejable usar el coeficiente Kappa. El valor de Kappa es:

$$\kappa = \frac{O - E}{1 - E} \tag{1}$$

con O la exactitud observada y E la exactitud esperada de acuerdo al azar. No existe acuerdo en categorizar los valores de kappa para interpretar como de "bueno" es el resultado dado un valor. Algunos autores dicen que excelente si es mayor que 0.75, y pobre por debajo de 0.4.

Como hemos mencionado, para seleccionar cual de las medidas de entre las calculadas por la función del parámetro `summaryFunction` se utiliza para seleccionar el modelo se utiliza el parámetro `metric` de `train()` (p.e. `metric = "Kappa"`), indicando en el parametro `maximize` (también de `train()`) si es una medida a maximizar (`maximize = T`) o minimizar (`maximize=F`). El valor por defecto es maximizar.

Veamos un ejemplo con Support Vector Machines:

```
set.seed(1234)
pima.modelo.3cv10.svm.Kappa<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                   pima.Datos.Train[[pima.Var.Salida.Usada]],
                                   method='svmRadial', metric = "Kappa",
                                   trControl=pima.TrainCtrl.3cv10)
```

```
> pima.modelo.3cv10.svm.Kappa
Support Vector Machines with Radial Basis Function Kernel

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 553, 554, 554, 554, 554, ...
Resampling results across tuning parameters:

  C     Accuracy   Kappa
  0.25  0.7543716  0.4144473
  0.50  0.7554204  0.4290174
  1.00  0.7505376  0.4206782
```

```
Tuning parameter 'sigma' was held constant at a value of 0.1200789
Kappa was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.1200789 and C = 0.5.
```

Como se puede suponer se le puede pasar una función definida por el usuario para que se utilicen otras medidas. Caret proporciona varias funciones alternativas que se pueden utilizar. Por ejemplo tiene `twoClassSummary()` que calcula, para problemas de clasificación de dos clases, la Receiver Operating Characteristic ("ROC"), la sensibilidad ("Sens") y la especifidad ("Spec").

La sensibilidad es la medida en la que el sistema predice correctamente los casos positivos (verdaderos positivos/total de positivos), y la especificidad la medida en que predice correctamente los casos negativos (verdaderos negativos/total de negativos). Se podría dar más valor a una u otra medida si no queremos que cuesten igual los falsos positivos o los falsos negativos. Si se valora más la sensibilidad buscaríamos modelos donde no se nos escapen casos positivos (aunque aumentemos los falsos positivos) y si se valora más la especificidad se buscarían modelos donde no se nos escapen casos negativos (aumentando normalmente los falsos negativos). Habría que encontrar un trade-off entre ambos valores (si se hace un 50% tenemos el típico caso en que apreciamos por igual un verdadero positivo como un verdadero negativo). En los casos extremos se llegan a clasificadores que siempre clasifican los datos como positivos (lo que da sensibilidad 1 pues no se nos escapa ningún positivo verdadero, ¡decimos que todos son verdaderos!) o clasifican siempre como negativo (lo que da especificidad 1 pues no se nos escapa ningún verdadero negativo). No trates de maximizar alguna de dichas medidas por separado sino más bien una combinación de ellas (que es lo que hace ROC).

La curva ROC se puede usar para estimar el rendimiento utilizando una combinación de la sensibilidad y la especificidad (por lo general la una aumenta a costa de la otra). Una curva ROC dibuja el ratio de verdaderos positivos (sensibilidad) frente al ratio de falsos positivos (1 menos la especificidad). Estos dos ratios están positivamente correlados puesto que es habitual que para acertar más positivos se aumente el área del espacio de entrada que se considera de la clase positivos y al aumentarla es habitual que, junto con nuevos casos positivos, ahora bien clasificados, se nos "cuelen" algunos casos que son negativos, lo que aumentaría al mismo tiempo el ratio de falsos positivos. Es, por tanto, una forma de visualizar en qué medida el ser más tolerante con falsos positivos mejora la tasa de aciertos de los positivos verdaderos.

Hacer un diagrama donde se dibuja la relación entre estos ratios genera curvas como las mostradas en la figura 29 .El área bajo dicha curva se puede usar para medir el rendimiento de un clasificador de dos clases pues cuanto más se asemeje a una L invertida y más se aleje de una diagonal indica que se es más fino al aumentar el ratio de verdaderos positivos (se cuelan menos negativos). No obstante, tarde o temprano se llega al caso límite de clasificar todo como positivo (lo que genera un ratio de verdaderos positivos de 1, y también de falsos positivos de 1). En la figura 29(b) la curva generada por el modelo A es peor que la B y la C. Los modelos B y C tienen rendimientos más similares (las áreas bajo la curva tienen valores más parecidos) aunque muestran diferentes comportamientos en el trade-off de aumento de verdaderos positivos frente a tolerar más falsos positivos.

```
pima.TrainCtrl.3cv10.2ClssSum <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",number = 10
  ## con 3 repeticiones
  ,repeats = 3
  # Esta vez mejor sin verbose
  ,verboseIter=F
```

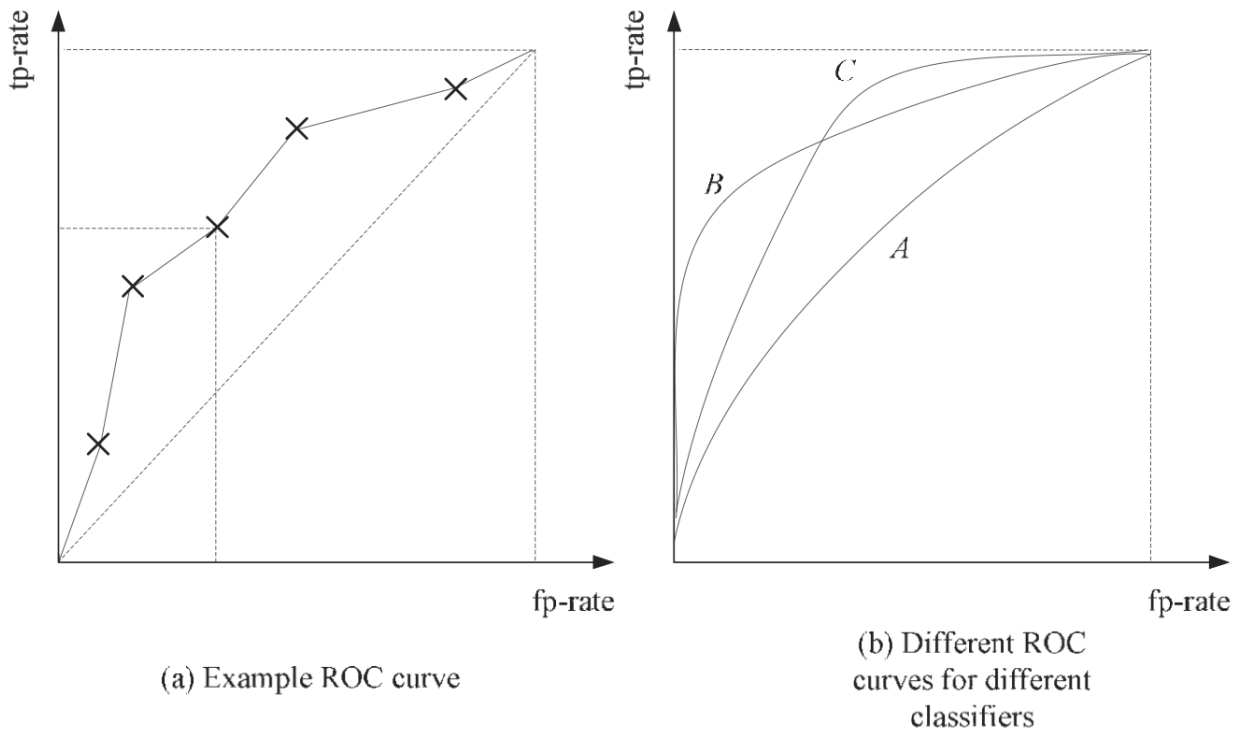


Figure 29: Ejemplos de curvas ROC (a) calculada, (b) ejemplos de formas.

```
  ,summaryFunction = twoClassSummary
  ,classProbs = TRUE  # Para usar ROC necesita calcular las classProbs
)

set.seed(1234)
pima.modelo.3cv10.svm.ROC<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                 pima.Datos.Train[[pima.Var.Salida.Usada]],
                                 method='svmRadial', metric = "ROC",
                                 trControl=pima.TrainCtrl.3cv10.2ClssSum)
```

```
> pima.modelo.3cv10.svm.ROC
Support Vector Machines with Radial Basis Function Kernel

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
```

```
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 553, 554, 554, 554, 554, ...
Resampling results across tuning parameters:

  C     ROC        Sens       Spec
  0.25  0.8285083  0.8566667  0.5592352
  0.50  0.8289899  0.8608333  0.5500000
  1.00  0.8285119  0.8658333  0.5328283

Tuning parameter 'sigma' was held constant at a value of 0.1200789
ROC was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.1200789 and C = 0.5.
```

Si se desea un diagrama ROC se necesita una librería y calcular la curva. Se puede mostrar diversa información en dichos diagramas o indicar si se quiere el punto de threshold donde Sensibilidad y Especifidad se tienen en cuenta de igual manera, o se busca el punto que maximiza ambas. Prueba los siguientes comandos:

```
library(pROC)
pima.preds.Test.3cv10.svm.ROC.probs<-predict(pima.modelo.3cv10.svm.ROC,
                    newdata=pima.Datos.Test[pima.Vars.Entrada.Usadas],type="prob")
pima.Test.3cv10.svm.ROCcurve <- roc(pima.Datos.Test[[pima.Var.Salida.Usada]],
                          pima.preds.Test.3cv10.svm.ROC.probs[, 1])
plot(pima.Test.3cv10.svm.ROCcurve, type = "S", print.auc=T,
     print.auc.y=.3,print.thres = .5)
plot(pima.Test.3cv10.svm.ROCcurve, print.thres="best",
     print.thres.best.method="closest.topleft",add=T)
# Si se quiere obtener el mejor threshold (según el método que se quiera)
pima.Test.3cv10.svm.ROCcurve.coords <- coords(pima.Test.3cv10.svm.ROCcurve, "best",
                              best.method="closest.topleft",
                              ret=c("threshold", "accuracy"))
print(pima.Test.3cv10.svm.ROCcurve.coords)# Muestra threshold y accuracy
```

Lo que produce el diagrama que se puede ver en la figura 30 que muestra tanto el área bajo la curva como distintos puntos de equilibrio usando los datos de test.

Caret también proporciona una función `multiClassSummary()` que calcula varias medidas para clasificación de múltiples clases. Además de exactitud y Kappa calcula la sensibilidad y especificidad medias (hace la media de las medidas de una clase frente al resto, p.e. si hay 3 clases, calcula la sensibilidad de la clase 1 frente a la clase 2+3, la clase 2 frente a 1+3, y la clase 3 frente a 1+2 y luego hace la media), la precisión media, etc.

Vemos un ejemplo de esta función con el dataset iris.

```
# Usamos el dataset iris y hacemos una partición al 80%
data(iris)
iris.Datos.Todo<-iris
iris.Var.Salida.Usada<-c("Species")
```

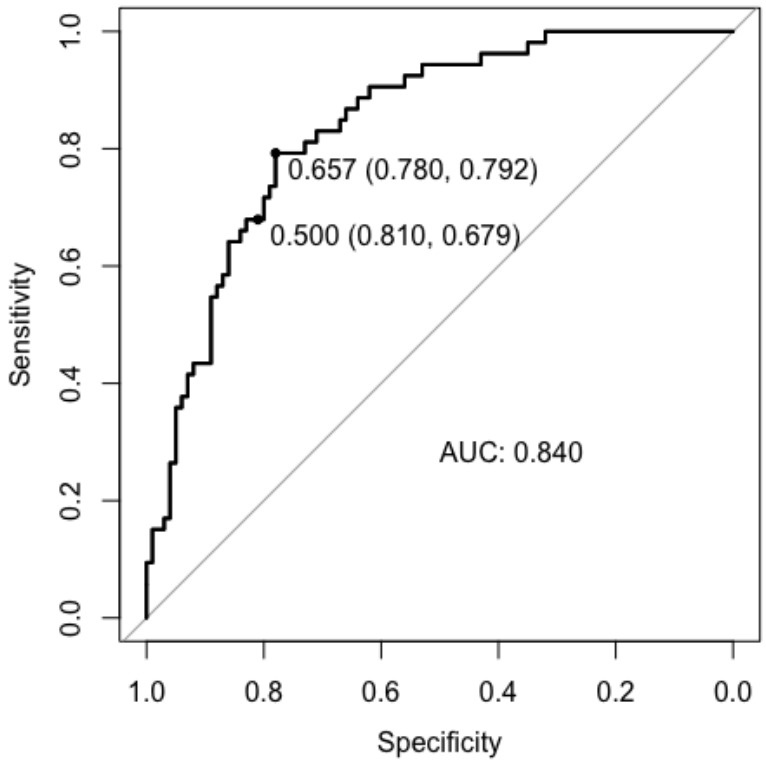


Figure 30: Curva ROC de Support Vector Machine para Pima Indians

```
iris.Vars.Entrada.Usadas<-setdiff(names(iris.Datos.Todo),iris.Var.Salida.Usada)
iris.Vars.Entrada.Escaladas<-iris.Vars.Entrada.Usadas

set.seed(1234)
iris.trainIdx.80<- createDataPartition(iris.Datos.Todo[[iris.Var.Salida.Usada]],
                                       p=0.8,
                                       list = FALSE,
                                       times = 1)
iris.Datos.Train<-iris.Datos.Todo[iris.trainIdx.80,]
iris.Datos.Test<-iris.Datos.Todo[-iris.trainIdx.80,]

iris.TrainCtrl.3cv10.mltClssSmmry <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",number = 10
  ## con 3 repeticiones
  ,repeats = 3
```

```
  # Esta vez mejor sin verbose
  ,verboseIter=F
  ,summaryFunction = multiClassSummary
)

set.seed(1234)
iris.modelo.3cv10.svm.mltClssSmmry<-train(iris.Datos.Train[iris.Vars.Entrada.Usadas],
                                          iris.Datos.Train[[iris.Var.Salida.Usada]],
                                          method='svmRadial',
                                          trControl=iris.TrainCtrl.3cv10.mltClssSmmry)
```

```
> iris.modelo.3cv10.svm.mltClssSmmry
Support Vector Machines with Radial Basis Function Kernel

120 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 108, 108, 108, 108, 108, 108, ...
Resampling results across tuning parameters:

  C     Accuracy   Kappa      Mean_F1    Mean_Sensitivity  Mean_Specificity
  0.25  0.9305556  0.8958333  0.9290300  0.9305556         0.9652778
  0.50  0.9444444  0.9166667  0.9437831  0.9444444         0.9722222
  1.00  0.9472222  0.9208333  0.9464727  0.9472222         0.9736111
  Mean_Pos_Pred_Value  Mean_Neg_Pred_Value  Mean_Precision  Mean_Recall  Mean_Detection_Rate
  0.9414815            0.9685185            0.9414815       0.9305556    0.3101852
  0.9527778            0.9745370            0.9527778       0.9444444    0.3148148
  0.9551852            0.9758818            0.9551852       0.9472222    0.3157407
  Mean_Balanced_Accuracy
  0.9479167
  0.9583333
  0.9604167

Tuning parameter 'sigma' was held constant at a value of 0.5745213
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.5745213 and C = 1.
```

Para ver otras summary functions disponibles se puede mirar la documentación de caret en `http://topepo.github.io/caret/measuring-performance.html#measures-for-class-probabilities`

Si se desea crear una summary function propia se debe crear una función que devuelva un vector de medidas numéricas con sus nombres correspondientes (para poder seleccionar entre ellas, entre otras cosas). La función debe tener tres parámetros:

- `data`, que debe ser un dataframe o una matriz con, al menos, dos columnas llamadas `obs` y `pred` que

contendrán los valores predichos y observados (verdaderos) de los ejemplos usados como validación. Si el parámetro `classProbs` de `trainControl` es `TRUE` hay columnas extra con las probabilidades de las clases. El nombre de esas columnas son los nombres de las clases (las etiquetas de los niveles). Si en la llamada a `train()` hay un vector de `weights` también se pasa esa columna en `data`. También puede tener información adicional en `data` si se usa el método `recipe` en `train()`.

- `lev`, que contiene los nombres de las clases (de los niveles).
- `model`, que contiene el nómbre del método usado (el valor de `method` que se le pasa a `train()`).

Como ejemplo de medida propia vamos a tratar el problema de la calidad del vino blanco. Este es un problema de clasificación que da resultados bastante pobres de Accuracy (alrededor del 50% sobre test). Se trata de distinguir entre 7 calidades de vino (que van de 3 a 9). El problema está bastante descompensado en número de ejemplos de cada clase, aparte de estar bastante mezcladas las clases.

El caso es que en el artículo (disponible en `http://www.sciencedirect.com/science/article/pii/S0167923609001377`) de los propios autores de dicha base de datos (cuya información se puede encontrar en `https://archive.ics.uci.edu/ml/datasets/wine+quality`, indican que utilizan medidas varias medidas alternativas de Accuracy a la que parametrizan con cierta tolerancia. A efectos esas *"Accuracies"* alternativas admiten como correcta una clasificación que se equivoque alguna clase arriba o abajo. La justificación que dan los autores es que, a efectos prácticos, no hace falta hilar tan fino y se puede tolerar un pequeño error de calidad arriba/abajo. Lo que no se puede hacer es confundir un vino bueno con uno malo (o viceversa), es decir, de calidades bastante diferentes (varias clases de "distancia"). Así pues mostraremos una función alternativa que incluya tanto las medidas habituales Accuracy y Kappa para clasificación, como dos accuracies que admitan como acierto un error de una clase arriba/abajo o de hasta dos clases. El código sería:

```
wine1classdif<-function(data, lev = NULL, model = NULL) {
  # calculamos los valores habituales (Accuracy y Kappa)
  out<-postResample(data[, "pred"],data[, "obs"])
  aciertos<-sum(data[, "pred"]==data[, "obs"])
  difs<-abs(as.numeric(data[, "pred"])-as.numeric(data[, "obs"]))
  dif1<-sum(difs<=1)
  dif2<-sum(difs<=2)
  tot<-length(data[,"pred"])
  out<-c(out,dif1=dif1/tot,dif2=dif2/tot)
  out
}

# Base de datos white wine. Está en formato csv
wine<- read.csv2(
  paste("https://archive.ics.uci.edu/ml/machine-learning-databases/",
        "wine-quality/winequality-white.csv",
        sep="")
  ,dec="."
)
wine$class<-factor(wine$quality) # La clase como un factor
wine$quality<-NULL               # y se elimina la quality (es la clase)
```

```
# Las etiquetas de los niveles no deben ser números
levels(wine$class)<-make.names(levels(wine$class))

# Usamos el dataset wine y hacemos una partición al 80%

wine.Datos.Todo<-wine
wine.Var.Salida.Usada<-c("class")
wine.Vars.Entrada.Usadas<-setdiff(names(wine.Datos.Todo),wine.Var.Salida.Usada)

set.seed(1234)
wine.TrainIdx.80<- createDataPartition(wine.Datos.Todo[[wine.Var.Salida.Usada]],
                                       p=0.8,
                                       list = FALSE,
                                       times = 1)
wine.Datos.Train<-wine.Datos.Todo[wine.TrainIdx.80,]
wine.Datos.Test<-wine.Datos.Todo[-wine.TrainIdx.80,]

wine.TrainCtrl.3cv10.wn1ClssDf <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",
  number = 10, # 10 pliegues
  repeats = 3,  ## con 3 repeticiones
  verboseIter=F, # Esta vez mejor sin verbose
  seeds=seeds,  # Podemos reusar seeds porque son 3 combinaciones de hiper-parámetros
  summaryFunction = wine1classdif    # Nuestra función especial
)

cl <- makeCluster(detectCores()-1)
registerDoParallel(cl)

set.seed(1234)
wine.modelo.3cv10.svm.wn1ClssDf<-train(wine.Datos.Train[wine.Vars.Entrada.Usadas],
                                       wine.Datos.Train[[wine.Var.Salida.Usada]],
                                       method='svmRadial',
                                       trControl=wine.TrainCtrl.3cv10.wn1ClssDf
                                       ,metric = "dif1")
stopImplicitCluster()
stopCluster(cl)
registerDoSEQ()
```

Los resultados los vemos a continuación (como tarda un poco en ajustar el modelo de Support Vector Machines hemos usado varios cores, si tarda mucho siempre puedes probar un modelo de árboles de decisión que es más rápido).

```
> wine.modelo.3cv10.svm.wn1ClssDf
Support Vector Machines with Radial Basis Function Kernel
```

```
3920 samples
  11 predictor
   7 classes: 'X3', 'X4', 'X5', 'X6', 'X7', 'X8', 'X9'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 3529, 3528, 3528, 3527, 3527, 3529, ...
Resampling results across tuning parameters:

  C     Accuracy   Kappa      dif1       dif2
  0.25  0.5487255  0.2529637  0.9434568  0.9961765
  0.50  0.5539159  0.2690147  0.9475399  0.9962617
  1.00  0.5664177  0.2970085  0.9504335  0.9960069

Tuning parameter 'sigma' was held constant at a value of 0.08196613
dif1 was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.08196613 and C = 1.
```

Comprobamos que su Accuracy normal es bastante pobre, el 55%, en cambio si se permite una clase arriba/abajo alcanzamos el 95% de Acierto. Veamos si el modelo no está sobre entrenado. Calcularemos el rendimiento sobre el conjunto de Test:

```
# Calculamos las medidas sobre el conjunto de Test
wine1classdif(data.frame(pred=predict(wine.modelo.3cv10.svm.wn1ClssDf,
                                      wine.Datos.Test[wine.Vars.Entrada.Usadas]),
                         obs=wine.Datos.Test[[wine.Var.Salida.Usada]]))
```

```
> # Calculamos las medidas sobre el conjunto de Test
> wine1classdif(data.frame(pred=predict(wine.modelo.3cv10.svm.wn1ClssDf,
+                                       wine.Datos.Test[wine.Vars.Entrada.Usadas]),
+                          obs=wine.Datos.Test[[wine.Var.Salida.Usada]]))
 Accuracy     Kappa      dif1      dif2
0.5797546 0.3177717 0.9498978 0.9959100
```

Comprobamos que los resultados de `dif1` (que es por la métrica que estamos optimizando) son muy similares a los valores obtenidos con el entrenamiento, luego hemos conseguido un modelo final que no está sobreajustado.

Ejercicio

Crear una función summary propia para dos clases que calcule una medida de error en la que los falsos negativos cuenten cinco veces más que los falsos positivos (algo útil, p.e., en tests de enfermedades muy peligrosas) y ajusta un modelo de diabetes en Pima Indians guiado por dicha medida. Recuerda que tendrás que minimizar si evaluas el error (y no el acierto).

Por último mencionar que Caret utiliza por defecto el mejor valor de la medida que se le indique (bien maximiza o minimiza según la medida) pero que se pueden usar otras formas de escoger el mejor modelo.

Por ejemplo, se puede desear tener un modelo con una medida de rendimiento un poco peor pero que el modelo final sea más simple. Esto se controla con el parámetro `selectionFunction` que, por defecto es la función `best` pero que puede tomar otros valores como `oneSE` o `tolerance` que permiten,respectivamente, un estándar error o un porcentaje de tolerancia para obtener modelos más simples. Por cierto, no siempre es fácil de decidir qué se entiende por modelo más simple, aunque en algunos modelos como árboles de decisión o redes neuronales es fácil de decidir, los que contienen menos nodos o neuronas.

11.2.4 Determinando las configuraciones de hiper-parámetros a probar (tuneGrid)

Se ha mencionado que Caret prueba varias configuraciones de hiper-parámetros. Hasta ahora hemos visto que, por defecto, prueba 3 configuraciones, como puedes comprobar al ver los modelos. P.e. en los Support Vector Machines del ejemplo anterior sobre el vino se indica que el parámetro `sigma` se mantuvo constante a `0.08196613`, y probó 3 valores diferentes de `C` (`0.25, 0.5` y `1.0`), escogiendo finalmente `C = 1`. De hecho si se ejecuta un `plot()` sobre el modelo muestra como varía la medida escogida para evaluar el modelo (en ese ejemplo "`dif1`"), ya que ese es el diagrama por defecto de un modelo de caret (mostrar los resultados de los distintos hiper-parámetros probados).

```
plot(wine.modelo.3cv10.svm.wn1ClssDf)
```

En estas ejecuciones no hemos controlado nada sobre los hiper-parámetros a probar. Hay cuatro formas diferentes de decirle a Caret qué valores queremos explorar para esos hiper-parámetros: La forma sencilla (mediante `tuneLength`), la forma completa en forma de parrilla (mediante `tuneGrid`) una búsqueda aleatoria de parámetros y un remuestreo adaptativo de hiper-parámetros.

11.2.4.1 tuneLength

La forma más sencilla es utilizar el parámetro `tuneLength` de `train()`, donde se le indica el número de combinaciones de los parámetros que va a intentar. La función `train()` contiene algoritmos que generan los valores de dichas combinaciones tratando que sean valores razonables. No obstante esta forma es en la que menos control se tiene sobre las combinaciones de hiperparámetros utilizados.

```
# Usamos el dataset PimaIndiansDiabetes
set.seed(1234)
pima.modelo.3cv10.tl10.gbm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                  pima.Datos.Train[[pima.Var.Salida.Usada]],
                                  method="gbm", trControl=pima.TrainCtrl.3cv10
                                  ,tuneLength = 10
                                  ,verbose=F # Este modelo es verbose por defecto
)
```

Habrás podido comprobar que el programa ejecuta bastante más que 10 combinaciones y es que, en el Stochastic Gradient Boosting , que tiene 4 hiper-parámetros accesibles, por defecto Caret fija 2 de ellos y los otros 2 prueba tantos valores diferentes como indique tuneLength, en este caso 10x10=100 combinaciones diferentes.

```
> pima.modelo.3cv10.tl10.gbm
Stochastic Gradient Boosting
```

```
615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 553, 554, 554, 554, 554, ...
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy   Kappa
   1                  50      0.7689847  0.4600583
   1                 100      0.7597391  0.4458086
   1                 150      0.7554557  0.4401469
   1                 200      0.7554557  0.4425357
   1                 250      0.7510929  0.4333757
   1                 300      0.7537987  0.4403691
   1                 350      0.7505729  0.4305700
   1                 400      0.7489864  0.4274067
   1                 450      0.7462542  0.4207809
   1                 500      0.7451877  0.4182657
   2                  50      0.7635554  0.4564090
   2                 100      0.7559580  0.4418088
   2                 150      0.7494712  0.4274036
...
  10                 350      0.7251102  0.3881858
  10                 400      0.7234708  0.3858748
  10                 450      0.7240349  0.3868667
  10                 500      0.7235237  0.3875162

Tuning parameter 'shrinkage' was held constant at a value of 0.1
Tuning
 parameter 'n.minobsinnode' was held constant at a value of 10
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 50, interaction.depth = 1, shrinkage =
 0.1 and n.minobsinnode = 10.
```

Hemos tardado algo en ejecutar los modelos al haberlo hecho sin paralelismo. Si quisieramos usar paralelismo y asegurarnos replicabilidad habría que crear un seeds específico (para las 100 combinaciones). Como ejemplo ponemos el código a usar:

```
# Usamos el dataset PimaIndiansDiabetes
set.seed(1234)
# seedsLength es = (numero_pliegues*numero_repeticiones)+1, en este caso (10*3)+1
n_folds<-10
```

```
n_reps<-3
seedsLength=(n_folds*r_reps)+1
seeds <- vector(mode = "list", length = seedsLength)
# Crearemos unos pliegues para usar los mismos en todos los modelos diferentes
foldIndexes<-createMultiFolds(pima.Datos.Train[[pima.Var.Salida.Usada]],k=n_folds,times=n_reps)
# combHParam es el número de combinaciones de hiper-parámetros a probar
combHParam=100
for(i in 1:(seedsLength-1)) seeds[[i]]<- sample.int(n=1000, combHParam)
# Hay que crear una semilla única para el modelo final a entrenar.
seeds[[seedsLength]]<-sample.int(1000, 1)
# TrainControl con seeds
pima.TrainCtrl.3cv10.plGBM <- trainControl(method='repeatedcv',
                                           number = n_folds,
                                           repeats = n_reps,
                                           index = foldIndexes,
                                           seeds=seeds
)
cl <- makeCluster(detectCores()-1)
registerDoParallel(cl)
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.pl<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                     pima.Datos.Train[[pima.Var.Salida.Usada]],
                                     method="gbm", trControl=pima.TrainCtrl.3cv10.plGBM
                                     ,tuneLength = 10
                                     ,verbose=F # Este modelo es verbose por defecto
)
stopImplicitCluster()
stopCluster(cl)
registerDoSEQ()
```

11.2.4.2 tuneGrid

Se le puede pasar a `train()` una parrilla con todas las combinaciones que se quiere comprobar mediante el parámetro `tuneGrid`. Dicho argumento debe ser un data frame donde las columnas tengan los nombres de los parámetros y las filas sean las combinaciones a comprobar. Lo habitual es crear ese data frame con el comando `expand.grid()`, al que se le pasan vectores con los diferentes valores a probar de cada parámetro y genera las combinaciones de todos con todos. Por supuesto se puede hacer un dataframe con las combinaciones particulares que se quiera.

```
# Usamos el dataset iris
iris.trainCtrl.3cv10.resampAll <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",number = 10
  ,repeats = 3     ## con 3 repeticiones
  ,verboseIter=T   # Verbose para ver si falla algún valor del grid
  ,returnResamp = "all"  # Guardamos todo para hacer diagramas
```

```
)
gbm.grid <- expand.grid(n.trees=c(10,20,30,40,50,75,100,500,1000),
                    shrinkage=c(0.01,0.05,0.1),
                    n.minobsinnode = c(3,5,10,15),
                    interaction.depth=c(1,5,10)
)

# Otros parámetros de gbm que no aparecen en grid se deben poner directamente
# en train: p.e. distribution, o bag.fraction
set.seed(1234)
iris.modelo.3cv10.grid.gbm<-train(iris.Datos.Train[iris.Vars.Entrada.Usadas],
                            iris.Datos.Train[[iris.Var.Salida.Usada]],
                            method="gbm", trControl=iris.trainCtrl.3cv10.resampAll
                            ,tuneGrid = gbm.grid
                            ,verbose=F # Este modelo es verbose por defecto
# da error                  ,distribution = "gaussian"  # Otros parámetros de gbm
#                           ,bag.fraction=0.75          # Otros parámetros de gbm
)

# Incrementamos el número de lineas que muestra print
options(max.print = 10000)
# Mostramos el resultado del ajuste de hiper-parámetros
iris.modelo.3cv10.grid.gbm
# Usamos escala logaritmica en el eje x
plot(iris.modelo.3cv10.grid.gbm,scales=list(x=list(log=T)))
options(max.print = 1000)
```

Ejercicio Has visto que incluso para un conjunto tan pequeño como el `iris` se tarda bastante en ejecutar en un solo proceso. Modifica el código anterior para usar paralelismo. Recuerda calcular el número de combinaciones diferentes de hiper-parámetros.

```
> iris.modelo.3cv10.grid.gbm
Stochastic Gradient Boosting

120 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 108, 108, 108, 108, 108, 108, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.minobsinnode  n.trees  Accuracy   Kappa
  0.01        1                  3               10     0.9277778  0.8916667
  0.01        1                  3               20     0.9277778  0.8916667
```

```
  0.01        1                   3                30     0.9305556  0.8958333
  0.01        1                   3                40     0.9250000  0.8875000
  0.01        1                   3                50     0.9305556  0.8958333
...
  0.05        1                  15                20     0.9333333  0.9000000
  0.05        1                  15                30     0.9388889  0.9083333
  0.05        1                  15                40     0.9444444  0.9166667
  0.05        1                  15                50     0.9500000  0.9250000
  0.05        1                  15                75     0.9416667  0.9125000
  0.05        1                  15               100     0.9388889  0.9083333
  0.05        1                  15               500     0.9305556  0.8958333
  ...
  0.10       10                  15                75     0.9361111  0.9041667
  0.10       10                  15               100     0.9250000  0.8875000
  0.10       10                  15               500     0.9194444  0.8791667
  0.10       10                  15              1000     0.9166667  0.8750000

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 50, interaction.depth = 1, shrinkage =
 0.05 and n.minobsinnode = 15.
```

En la figura 31 se muestra un diagrama con los resultados anteriores donde se muestra más claramente como varían los resultados con las diferentes combinaciones. Los resultados pueden mostrar zonas donde puede merecer la pena probar un grid más fino en algunos valores (p.e. en varios shrinkages fijando el iteration depth en 1), o descartar otros (p.e. valores altos en el número de iteraciones de boosting).

Los diagramas que muestran los resultados de las distintas combinaciones de hiper-parámetros se pueden dibujar de varias maneras o con otras métricas. Por ejemplo:

```
# Usamos colores standard
trellis.par.set(standard.theme("pdf"))
plot(iris.modelo.3cv10.grid.gbm, metric="Kappa",plotType="level")

plot(iris.modelo.3cv10.grid.gbm, metric="Accuracy",plotType="line")
```

genera el diagrama de la figura 32, que muestra la variación de la métrica `Kappa`. `plotType` también tiene como alternativa los valores line (en la figura 33), y scatter (el tipo por defecto y mostrado en la figura 31). Mientras que con `line` se muestran en el eje x solo los valores probados (independientemente de la distancia que exista entre los valores), con `scatter` se muestra en el eje x la distancia real entre los valores probados.

Por cierto, si al ejecutar el modelo aparece un error como:

```
- Fold10.Rep3: shrinkage=0.50, interaction.depth=10, n.minobsinnode= 5, n.trees=1000
+ Fold10.Rep3: shrinkage=0.50, interaction.depth=10, n.minobsinnode=10, n.trees=1000
predictions failed for Fold10.Rep3: shrinkage=0.50, interaction.depth=10, n.minobsinnode=10,
n.trees=1000 Error in lvl[x] : invalid subscript type 'list'

- Fold10.Rep3: shrinkage=0.50, interaction.depth=10, n.minobsinnode=10, n.trees=1000
Error in { :
```

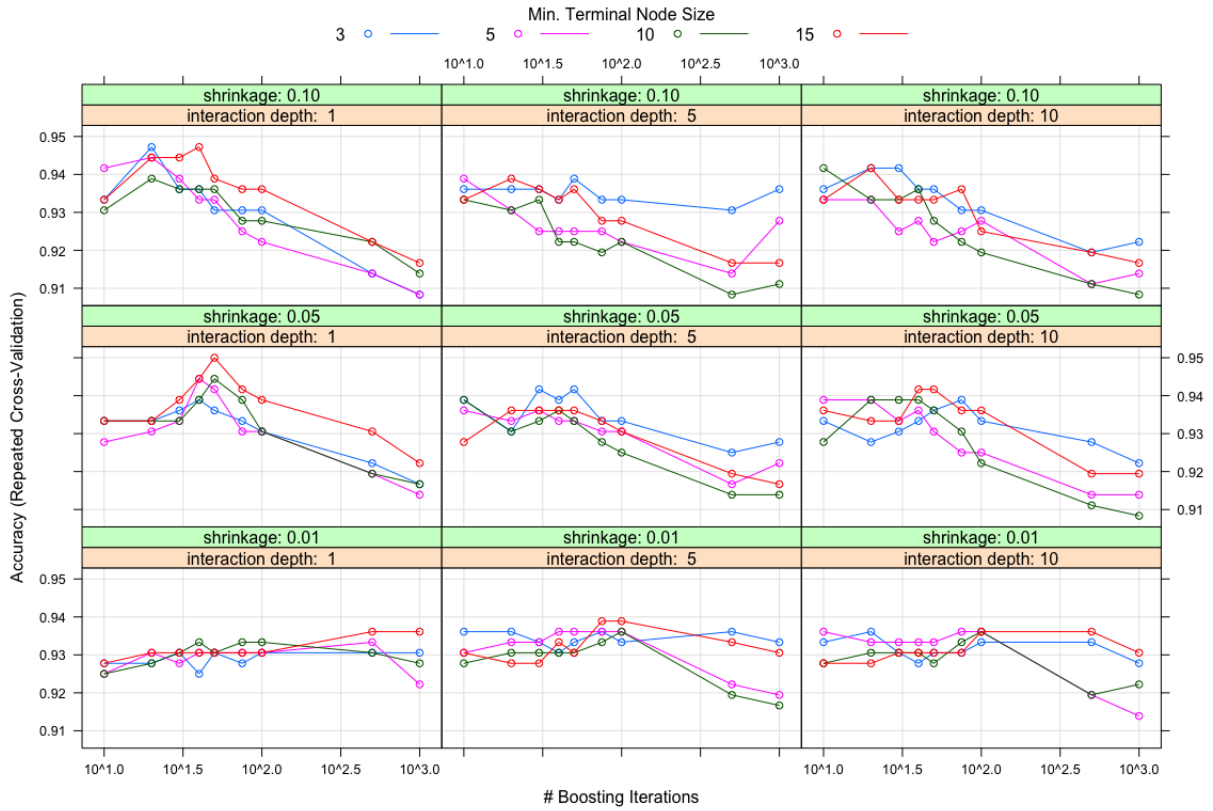


Figure 31: Diagrama de resultados de diferentes hiper-parámetros de gbm en iris usando el grid del ejemplo. Número de árboles está en escala logarítmica.

```
  task 30 failed - "arguments imply differing number of rows: 0, 12"
Además: There were 50 or more warnings (use warnings() to see the first 50)
```

es muy probable que alguno de los valores de los parámetros, tanto del grid o de los parámetros adicionales que se le pasen directamente al método subyacente en el comando `train()`, sean incorrectos. Este error de arriba se produce, por ejemplo, al ejecutar:

```
gbm.grid.bad <- expand.grid(n.trees=c(10,20,50,100,500,1000),
                    shrinkage=c(0.01,0.05,0.1,0.5),
                    n.minobsinnode = c(3,5,10),
                    interaction.depth=c(1,5,10)
)
iris.modelo.3cv10.grid.gbm.bad<-train(iris.Datos.Train[iris.Vars.Entrada.Usadas],
                            iris.Datos.Train[[iris.Var.Salida.Usada]],
                            method="gbm", trControl=iris.trainCtrl.3cv10.resampAll
```

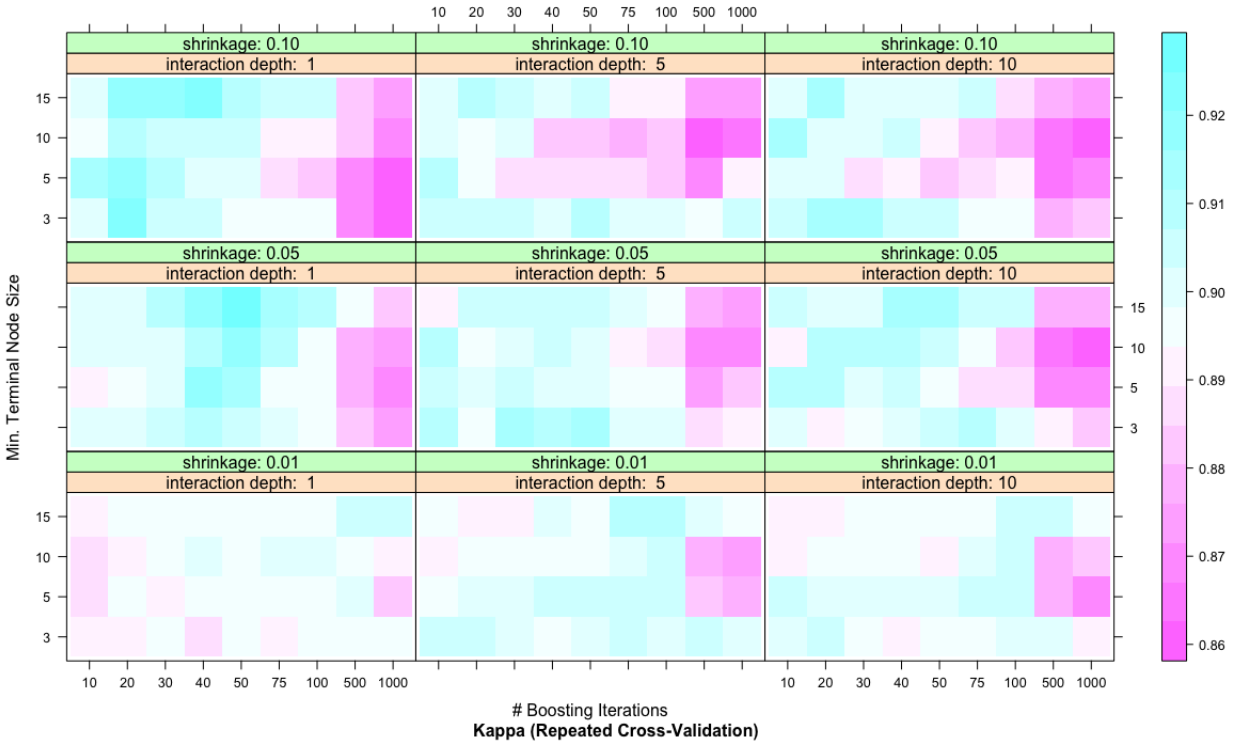


Figure 32: Diagrama de resultados con la metrica Kappa y un diagrama de niveles para diferentes hiperparámetros de gbm en iris usando el grid del ejemplo

```
                        ,tuneGrid = gbm.grid.bad
                        ,verbose=F # Este modelo es verbose por defecto
 )
```

y lo produce el valor 0.5 de `shrinkage` (hay múltiples fallos en los folds con ese valor y por ello, si hay errores, o sospecha de ellos, es interesante activar `verboseIter`). Si ejecutamos:

```
> warnings()
Warning messages:
1: predictions failed for Fold02.Rep1: shrinkage=0.50, interaction.depth= 1,
 n.minobsinnode= 5, n.trees=1000 Error in lvl[x] : invalid subscript type 'list'

2: predictions failed for Fold02.Rep1: shrinkage=0.50, interaction.depth= 5,
 n.minobsinnode= 5, n.trees=1000 Error in lvl[x] : invalid subscript type 'list'

...

49: predictions failed for Fold03.Rep2: shrinkage=0.50, interaction.depth= 1,
```

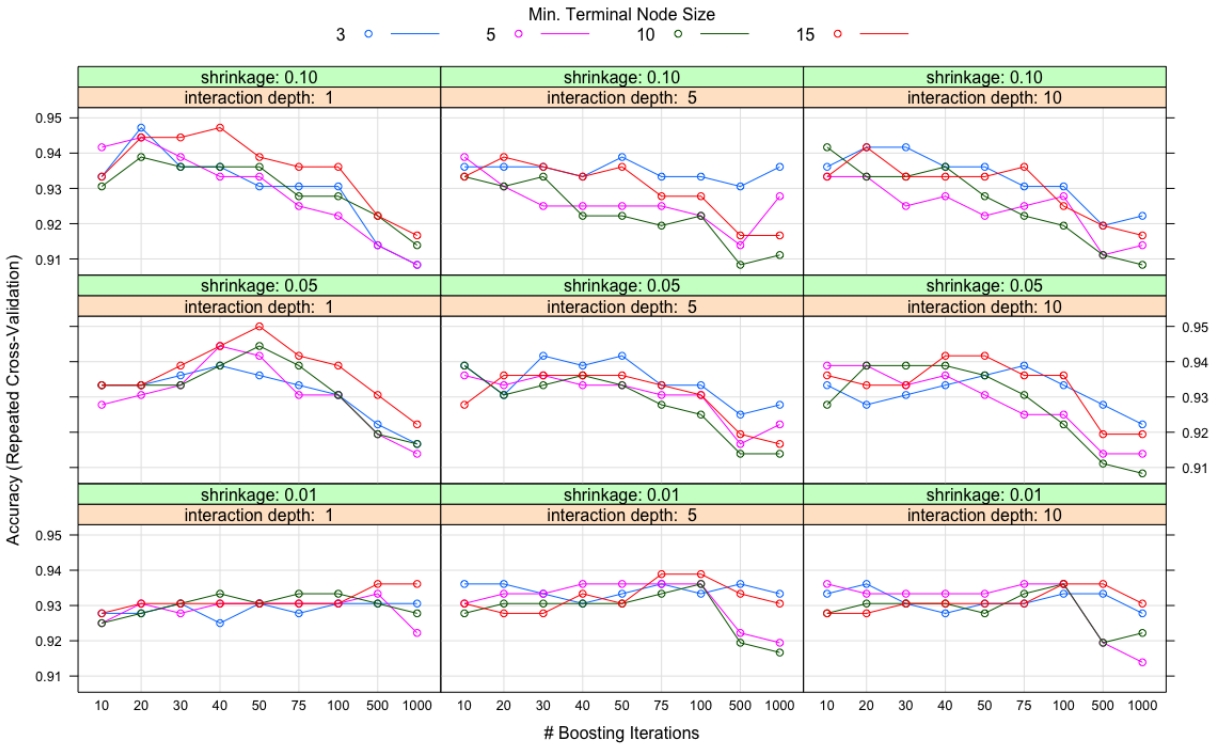


Figure 33: Diagrama de resultados con `plotType = "line"` para diferentes hiper-parámetros de gbm en iris usando el grid del ejemplo.

```
 n.minobsinnode= 3, n.trees=1000 Error in lvl[x] : invalid subscript type 'list'

50: predictions failed for Fold03.Rep2: shrinkage=0.50, interaction.depth= 1,
 n.minobsinnode=10, n.trees=1000 Error in lvl[x] : invalid subscript type 'list'
```

salimos de dudas sobre que `shrinkage = 0.50` es un parámetro que da problemas al algoritmo (y quizá sea interesante averigurar la razón).

Otros problemas pueden aparecer con otro tipo de parámetros. Por ejemplo, si te encuentras una salida del tipo:

```
Aggregating results
Something is wrong; all the Accuracy metric values are missing:
    Accuracy       Kappa
 Min.   : NA   Min.   : NA
 1st Qu.: NA   1st Qu.: NA
 Median : NA   Median : NA
```

```
 Mean   :NaN   Mean   :NaN
 3rd Qu.: NA   3rd Qu.: NA
 Max.   : NA   Max.   : NA
 NA's   :162   NA's   :162
Error: Stopping
Además: There were 50 or more warnings (use warnings() to see the first 50)
```

que nos aparecería si ejecutamos:

```
iris.modelo.3cv10.grid.gbm.bad2<-train(iris.Datos.Train[iris.Vars.Entrada.Usadas],
                       iris.Datos.Train[[iris.Var.Salida.Usada]],
                       method="gbm", trControl=iris.trainCtrl.3cv10.resampAll
                       ,tuneGrid = gbm.grid
                       ,verbose=F # Este modelo es verbose por defecto
                       ,distribution = "gaussian"  # parámetro problemático
)
```

el error lo produce el parámetro extra `distribution` con el valor `gaussian`. De nuevo sería interesante averiguar las razones por las que no funciona.

Para terminar esta sección ejecutaremos un grid sobre el gbm para el dataset Pima Indians. Lo vamos a usar más adelante para comparar resultados. Es una ejecución larga. Si has hecho el ejercicio de paralelizar la ejecución del grid para Iris, puedes reusarlo ahora para el Pima y lo hará más rápido:

```
# Usamos el dataset Pima
set.seed(1234)
pima.modelo.3cv10.grid.gbm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                  pima.Datos.Train[[pima.Var.Salida.Usada]],
                                  method="gbm", trControl=pima.TrainCtrl.3cv10
                                  ,tuneGrid = gbm.grid
                                  ,verbose=F # Este modelo es verbose por defecto
)
```

11.2.4.3 Búsqueda aleatoria de hiperparámetros

También se le puede decir a Caret que busque combinaciones al azar de los hiperparámetros. Caret genera esas combinaciones al azar dentro de rangos "razonables" para cada modelo. La forma de activarlo es mediante el parámetro `search` de `trainControl()`. Por defecto el valor es `"grid"`, es decir, que si se le proporciona una parrilla de hiperparámetros en el parámetro `tuneGrid` de `train()` buscará en dicha parrilla. En cambio si `search` de trainControl es `"random"` buscará tantas combinaciones al azar entre dichos rangos "razonables" como indique el parámetro `tuneLength` de `train()`.

```
# Usamos el dataset Pima
pima.trainCtrl.3cv10.randHP <- trainControl(## Crosvalidación de 10 pliegues
  method = "repeatedcv",
  number = 10
  ,repeats = 3  ## con 3 repeticiones
  ,verboseIter=T   # Verbose para comprobar si falla alguna configuración
```

```
  ,search = "random"
)
set.seed(1234)
pima.modelo.3cv10.randHP.tl5.gbm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                        pima.Datos.Train[[pima.Var.Salida.Usada]],
                                        method="gbm",
                                        trControl=pima.trainCtrl.3cv10.randHP
                                        ,tuneLength =  5
                                        ,verbose=F # Este modelo es verbose por defecto
)
```

Podemos comparar los resultados con el modelo obtenido en la sección 11.2.4.1.

```
max(pima.modelo.3cv10.tl10.gbm$results$Accuracy)
max(pima.modelo.3cv10.grid.gbm$results$Accuracy)
max(pima.modelo.3cv10.randHP.tl5.gbm$results$Accuracy)
# O mejor, usando un patrón de nombres
do.call("rbind", sapply(
  ls(pattern = "pima.modelo.*.gbm"),
  simplify = F,
  FUN = function(x)
    c(MaxAccuracy = eval(parse (
      text = paste("max(", x, "$results$Accuracy)",sep="")
      )))
  ))

# Para saber la mejor combinación de hiper-parámetros
pima.modelo.3cv10.grid.gbm$bestTune
```

```
> max(pima.modelo.3cv10.tl10.gbm$results$Accuracy)
[1] 0.7689847
> max(pima.modelo.3cv10.grid.gbm$results$Accuracy)
[1] 0.7706416
> max(pima.modelo.3cv10.randHP.tl5.gbm$results$Accuracy)
[1] 0.7072448
> # O mejor, usando un patrón de nombres
> do.call("rbind", sapply(
+   ls(pattern = "pima.modelo.*.gbm"),
+   simplify = F,
+   FUN = function(x)
+     c(MaxAccuracy = eval(parse (
+       text = paste("max(", x, "$results$Accuracy)",sep="")
+       )))
+   ))
                                MaxAccuracy
pima.modelo.3cv10.grid.gbm        0.7706416
```

```
pima.modelo.3cv10.randHP.tl5.gbm   0.7072448
pima.modelo.3cv10.tl10.gbm         0.7689847
pima.modelo.3cv10.tl10.gbm.pl      0.7679447
>
> # Para saber la mejor combinación de hiper-parámetros
> pima.modelo.3cv10.grid.gbm$bestTune
    n.trees interaction.depth shrinkage n.minobsinnode
318      30                10       0.1             15
```

que muestra que se ha encontrado la mejor combinación de hiper-parámetros usando `grid` en este caso. En realidad ese modelo ha probado bastante más combinaciones que otros modelos de esa lista (en particular 324). Para ver cuantas combinaciones ha probado cada modelo nos haremos una función auxiliar que permite ejecutar un comando sobre algún campo de una lista de modelos (el comando `do.cal()l` que hemos visto arriba pero parametrizado).

```
# Ejecuta un comando sobre lista de modelos
ejcomen <- function(mods, campo, command,nombre="Valor") {
  lst<-as.list(mods)
  names(lst)<-mods
  out<-do.call("rbind",
  sapply(
    mods,
    simplify = F,
    FUN = function(x, com = command)
      eval(parse (
        text = paste(com, "(", x, campo, ")", sep = "")
      ))
  ))
  colnames(out)<-nombre
  out }
```

y ejecutamos `nrow` sobre el campo `results`, en la lista de de modelos que nos interese. Nos indica cuantas combinaciones de hiper-parámetros ha buscado ese modelo.

```
ejcomen(ls(pattern="pima.modelo.*.gbm"),"$results","nrow","HPcombinations.tried")
```

```
> ejcomen(ls(pattern="pima.modelo.*.gbm"),"$results","nrow","HPcombinations.tried")
                                HPcombinations.tried
pima.modelo.3cv10.grid.gbm                       324
pima.modelo.3cv10.randHP.tl5.gbm                   5
pima.modelo.3cv10.tl10.gbm                       100
pima.modelo.3cv10.tl10.gbm.pl                    100
```

Es fácil ahora comprobar que algunas de estas exploraciones son más exhaustivas que las otras.

11.2.4.4 Remuestreo adaptativo de hiper-parámetros

Existe otra manera de tratar de mejorar la combinación de hiperparámetros. La eficiencia de los modelos suele ser, por desgracia, bastante sensible a los hiper-parámetros y no es fácil saber de antemano que valores

deben tener. Aunque se puede hacer una exploración en parrilla como se menciona antes es muy posible que se tenga que repetir la exploración con una nueva parrilla que haga una búsqueda de grano más fino en aquellas zonas más prometedoras de los primeros experimentos. Forma parte del proceso habitual de machine learning, que consiste en repetir los experimentos con diferentes hiper-parámetros, métodos, preprocesados, etc. hasta encontrar los mejores modelos.

No obstante caret incluye un par de métodos para realizar un remuestreo adaptativo de los hiper-parámetros de forma que a medida que encuentra buenas combinaciones busca nuevas combinaciones en los alrededores de estas. La forma de hacerlo es utilizando el parámetro `adaptive` de `trainControl()`. Se le pasa una lista con los parámetros que necesita este remuestreo adaptativo cuyos detalles se pueden ver en `http://topepo.github.io/caret/adaptive-resampling.html`

Puedes hacer una prueba con el siguiente código (aunque se toma su tiempo):

```
# Usando remuestreo adaptativo
pima.trainCtrl.3cv10.adapHP <- trainControl(## Crosvalidación de 10 pliegues
  method = "adaptive_cv",number = 10
  ,adaptive = list(min =5, alpha = 0.05, method ="gls", complete=TRUE)
  ## con 3 repeticiones
  ,repeats = 3
  # Verbose para comprobar si falla alguna configuración
  ,verboseIter=T
)
set.seed(1234)
pima.modelo.3adaptcv10.grid.gbm<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                               pima.Datos.Train[[pima.Var.Salida.Usada]],
                               method="gbm", trControl=pima.trainCtrl.3cv10.adapHP
                               ,tuneGrid  =  gbm.grid
                               ,verbose=F # Este modelo es verbose por defecto
)
```

y ahora comprobamos si hemos mejorado la versión de grid original.

```
ejcomen(ls(pattern="pima.modelo.*.gbm"),"$results$Accuracy","max","Max.Accuracy")
ejcomen(ls(pattern="pima.modelo.*.gbm"),"$bestTune","",
        getModelInfo("gbm")$gbm$parameters$parameter)
```

```
> ejcomen(ls(pattern="pima.modelo.*.gbm"),"$results$Accuracy","max","Max.Accuracy")
                               Max.Accuracy
pima.modelo.3adaptcv10.grid.gbm    0.7722193
pima.modelo.3cv10.grid.gbm         0.7706416
pima.modelo.3cv10.randHP.tl5.gbm   0.7072448
pima.modelo.3cv10.tl10.gbm         0.7689847
pima.modelo.3cv10.tl10.gbm.pl      0.7679447
> ejcomen(ls(pattern="pima.modelo.*.gbm"),"$bestTune","",
+         getModelInfo("gbm")$gbm$parameters$parameter)
                               n.trees interaction.depth shrinkage n.minobsinnode
pima.modelo.3adaptcv10.grid.gbm     100                 1 0.0500000              5
```

```
pima.modelo.3cv10.grid.gbm            30                10 0.1000000             15
pima.modelo.3cv10.randHP.tl5.gbm    4304                 6 0.1760972              9
pima.modelo.3cv10.tl10.gbm            50                 1 0.1000000             10
pima.modelo.3cv10.tl10.gbm.pl        150                 1 0.1000000             10
```

y comprobamos que parece que la mejora levemente.

11.2.5 Acceder a otros hiperparámetros que CARET enmascara

Hemos visto ya que los hiperparámetros con los que trabaja caret en cada método y con los que trabaja `tuneGrid` son solo algunos de los que en realidad tiene cada modelo particular. También hemos visto que se puede acceder a dichos hiperparámetros añadiéndolos directamente en la lista de parámetros usados en `train()`. Por ejemplo, random forest, en caret, solo maneja el hiperparámetro `mtry`, que indica el número de predictores usados en cada árbol del bosque, pero en el algoritmo del modelo hay otro hiperparámetro que es muy interesante analizar como sería `ntree` que indica el número de árboles que tiene el bosque (por defecto es 500). Tal y como lo hemos mostrado se puede incluir en la llamada a `train()` otro valor de `ntree` de la siguiente forma:

```
set.seed(1234)
pima.modelo.3cv10.ntree1000.rf<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                      pima.Datos.Train[[pima.Var.Salida.Usada]],
                                      method="rf", trControl=pima.TrainCtrl.3cv10
                                      ,ntree=1000
)
```

Ahora podríamos ver que con 1000 árboles por bosque tenemos unos resultados prácticamente iguales.

```
> ejcomen(ls(pattern="pima.modelo.*.rf"),"$results$Accuracy","max","Max.Accuracy")
                               Max.Accuracy
pima.modelo.3cv10.ntree1000.rf    0.7684558
pima.modelo.3cv10.rf              0.7684735
pima.modelo.3cv10.rf.pl1          0.7608761
pima.modelo.3cv10.rf.pl2          0.7608761
pima.modelo.3cv10.rf.plNS         0.7619602
pima.modelo.bstrp25.rf            0.7538253
```

Si quisieramos también probar diferentes combinaciones tendríamos tarde o temprano que ejecutar varias veces el `train()` de caret con los diferentes valores de `ntree` de alguna manera parecida a :

```
# búsqueda manual
pima.models.3cv10.ntree.rf <- list()
for (ntree in c(500,1000, 1500, 2000, 2500)) {
  set.seed(1234)
  model <- train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                 pima.Datos.Train[[pima.Var.Salida.Usada]],
                 method="rf", trControl=pima.TrainCtrl.3cv10
                 , ntree=ntree)
  key <- paste("RF.NTR",toString(ntree),sep="")
```

```
  pima.models.3cv10.ntree.rf [[key]] <- model
}
# compara resultados
# Ya veremos resamples() más adelante
pima.resamps.3cv10.ntree<-resamples(pima.models.3cv10.ntree.rf)
summary(pima.resamps.3cv10.ntree)
dotplot(pima.resamps.3cv10.ntree)
```

```
> summary(pima.resamps.3cv10.ntree)

Call:
summary.resamples(object = pima.resamps.3cv10.ntree)

Models: RF.NTR500, RF.NTR1000, RF.NTR1500, RF.NTR2000, RF.NTR2500
Number of resamples: 30

Accuracy
                Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
RF.NTR500  0.6721311 0.7328794 0.7723427 0.7684735 0.8024194 0.8688525    0
RF.NTR1000 0.6885246 0.7287811 0.7741935 0.7684558 0.8000397 0.8548387    0
RF.NTR1500 0.6885246 0.7287811 0.7723427 0.7673718 0.7903226 0.8688525    0
RF.NTR2000 0.6885246 0.7287811 0.7723427 0.7657412 0.7894632 0.8548387    0
RF.NTR2500 0.6885246 0.7287811 0.7661290 0.7652124 0.7894632 0.8548387    0

Kappa
                Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
RF.NTR500  0.2394015 0.3979090 0.4887243 0.4781612 0.5390313 0.7095238    0
RF.NTR1000 0.2858903 0.3799773 0.4918793 0.4771619 0.5537596 0.6861642    0
RF.NTR1500 0.2858903 0.3799773 0.4899254 0.4750897 0.5348725 0.7095238    0
RF.NTR2000 0.2858903 0.3799773 0.4813612 0.4712895 0.5225768 0.6861642    0
RF.NTR2500 0.2858903 0.3799773 0.4552664 0.4695262 0.5305853 0.6861642    0
```

El `dotplot()` nos mostraría que no hay diferencias significativas entre los distintos valores del parámetro `ntree` para este problema, como se ve en la figura 34.

Ver un mismo modelo tratándolo como si fueran varios diferentes puede ser un engorro así que sería interesante si pudiésemos ampliar el algoritmo en caret para que, en vez del método manual en el que se ejecuta varias veces `train()` con los distintos valores para los hiperparámetros no manejados por defecto se pudiesen también incluir estos otros hiperparámetros en un único `train()`. Es decir, que se extendiera para poder usar `tuneGrid` como se ha visto en la sección 11.2.4. Esto lo podemos hacer creando un modelo adicional basándonos en el modelo que queramos ampliar. En realidad añadiremos un nuevo modelo tal y como se explica en detalle en `https://topepo.github.io/caret/using-your-own-model-in-train.html`.

Antes de comentar como se hace hay que avisar que, al modificar el código original, podríamos modificar de forma sutil, pero sustancial, el comportamiento del modelo original tal y como lo implementa caret (que, a veces, incluye ciertos preprocesos no evidentes sin examinar el código) con lo que no debemos mezclar resultados del `train()` original y el modificado como si fuesen el mismo modelo y deberíamos repetir experimentos con la versión modificada, por si acaso.

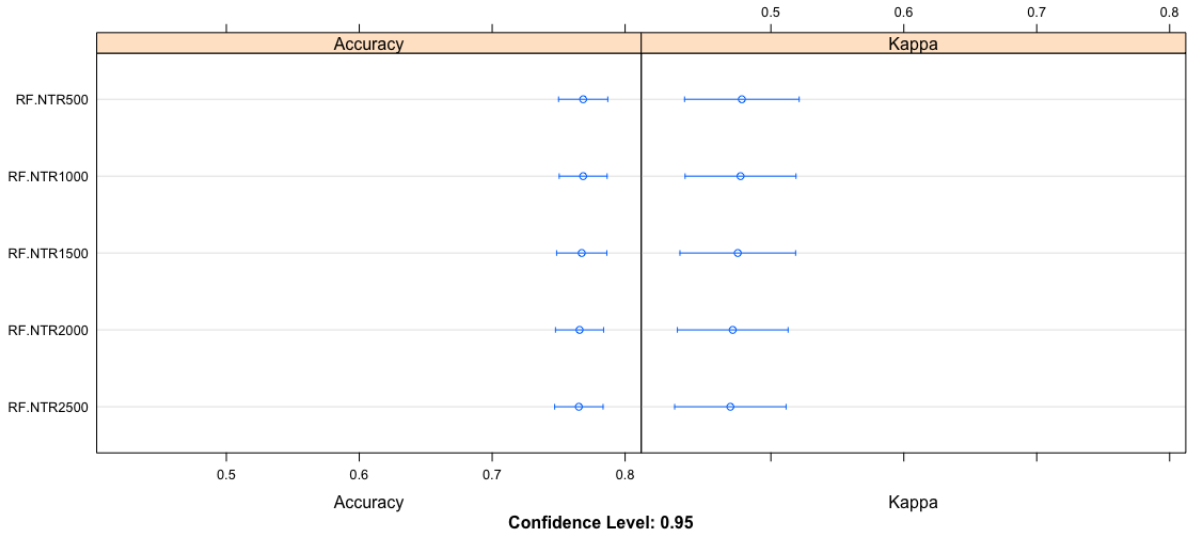


Figure 34: Diagrama de resultados variando "manualmente" el parámetro ntree en varios modelos

Para hacer las cosas bien podríamos primero irnos al código más actualizado de caret sobre el `train` del modelo que vayamos a modificar y basarnos en el. Por ejemplo, el código de los random forests lo encontraríamos en: `https://github.com/topepo/caret/blob/master/models/files/rf.R`. A partir de este código podríamos modificar solo lo que nos interese. Pero para simplificar haremos una versión, valga la redundancia, simplificada con el siguiente código:

```
library(randomForest)
RF2param <- list(
  type = "Classification",
  library = "randomForest",
  loop = NULL,
  parameters =  data.frame(parameter = c("mtry", "ntree"),
                           class = rep("numeric", 2), label = c("mtry", "ntree")),
  grid = function(x, y, len = NULL, search = "grid")
  {},
  fit = function(x, y, wts, param, lev, last, weights, classProbs, ...) {
    require(randomForest)   # Si no se exige puede fallar al usar varios procesadores
    randomForest(x, y, mtry = param$mtry, ntree = param$ntree, ...)
  },
  predict = function(modelFit, newdata, preProc = NULL, submodels = NULL)
    predict(modelFit, newdata),
  prob = function(modelFit, newdata, preProc = NULL, submodels = NULL)
    predict(modelFit, newdata, type = "prob"),
  sort = function(x)
```

```
    x[order(x[, 1]),],
  levels = function(x)
    x$classes,
  label = "Random Forest 2 params")
```

Si lo comparáis con la versión original de caret veréis ciertas diferencias importantes. Algunas están relacionadas con la simplificación del ejemplo y otras con los cambios que queremos introducir.

Así por ejemplo entre los cambios para simplificar el ejemplo vemos que en el tipo (`type`) decimos que es solo de clasificación (en el original es tanto clasificación como regresión). También hacemos que la función del parámetro `grid` no haga nada (en el original hay código para generar un grid automático con valores "razonables") pues le vamos a pasar un grid nosotros. Por último tampoco se incluyen funciones para valorar la importancia de las variables (`predictors` y `varImp`), ni `tags` ni `oob` (out of bag performance).

Por otro lado, entre los cambios para incluir el nuevo parámetro vemos que en el campo `parameters` incluímos también `ntree`, que declaramos como numérico y una etiqueta extendida. El otro cambio sutil es en el campo `fit` (la función que genera un modelo individual) que ahora incluye como uno de sus parámetros `ntree`). Lo demás no se toca.

Ahora ya se puede ejecutar un grid mediante `tuneGrid` que incluya ambos parámetros. Para indicar a caret que use la versión modificada y le pasamos a `train()`, en el parámetro `method`, el objeto `RF2param` y ya tenemos a caret buscando la combinación de ambos hiper-parámetros. De este modo accedemos a todas las ventajas de análisis que caret nos ofrece para un método (p.e. que escoja la mejor combinación de hiper-parámetros, los diagramas de comparación de hiper-parámetros, tratarlo como un solo modelo, etc.) mientras exploramos más parámetros. Veamos el resultado:

```
rf2parGrid<-expand.grid(mtry=c(2,5,8),ntree=c(500,1000,1500,2000,2500))

set.seed(1234)
pima.modelo.3cv10.rf2par<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                pima.Datos.Train[[pima.Var.Salida.Usada]],
                                method=RF2param, tuneGrid = rf2parGrid,
                                trControl=pima.TrainCtrl.3cv10)
```

```
> pima.modelo.3cv10.rf2par
Random Forest 2 params

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 553, 554, 554, 554, 554, ...
Resampling results across tuning parameters:

  mtry  ntree  Accuracy   Kappa
  2      500   0.7565926  0.4446665
  2     1000   0.7576415  0.4460923
```

```
  2     1500   0.7635995  0.4580737
  2     2000   0.7587167  0.4477049
  2     2500   0.7614402  0.4549449
  5      500   0.7598096  0.4589879
  5     1000   0.7657853  0.4708042
  5     1500   0.7652036  0.4693619
  5     2000   0.7646748  0.4679710
  5     2500   0.7673541  0.4741427
  8      500   0.7635643  0.4673900
  8     1000   0.7598184  0.4603787
  8     1500   0.7630619  0.4677201
  8     2000   0.7608761  0.4616728
  8     2500   0.7598008  0.4597035

Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were mtry = 5 and ntree = 2500.
```

cuyo diagrama podemos ver en la figura 35. Esta es una forma bastante más cómoda y práctica de trabajar con varios hiperparámetros ocultos que si lo hacemos de manera "manual". Cuando trabajamos con un modelo específico, o que conozcamos con más profundidad, probablemente merezca la pena hacer esta versión ampliada para trabajar más cómodamente. Eso sí, recuerda no mezclar resultados de la original y la modificada (y revisar si caret ha actualizado el train del original).

Por supuesto, para saber qué otros hiperparámetros podrían utilizarse pero que caret está abstrayendo, uno debe irse a la documentación de la librería original y comprobar los parámetros de la función que se usa para el ajuste de modelos. En este ejemplo la documentación de la librería original está en `https://cran.r-project.org/web/packages/randomForest/randomForest.pdf` y la función que hace el ajuste es `randomForest()`, (fíjate que la función de ajuste es la que se usa en el parámetro `fit` de la lista del objeto `RF2param`)

Os habréis dado cuenta que, realmente, lo que se suele cambiar para añadir nuevos hiperparámetros son, principalmente, tres elementos de la lista: `fit`, `parameters`, y `label`. El elemento `grid`, en principio, se podría dejar como esté (aunque si decides no retocarlo quizá sea mejor opción "anular" la función) puesto que si se va a usar `tuneGrid` no se llamará nunca a `grid` (a `grid` se llama cuando se tiene que generar el `tunegrid` a partir de `tuneLength`) y no pasaría nada.

Entonces lo que podemos hacer es retocar directamente esos elementos sobre el código más actualizado que tengamos del método. Podemos cargar la lista de un modelo buscándola con modelLookup, y luego retocamos solo lo que nos interesa. Vamos a hacerlo de nuevo para Random Forest:

```
RF2paramNew<-getModelInfo("rf",regex =F)[[1]]
RF2paramNew$label = "Random Forest 2 parameters"
RF2paramNew$parameters =  data.frame(parameter = c("mtry", "ntree"),
                                     class = rep("numeric", 2),
                                     label = c("#Randomly Selected Predictors", "#Trees"))
RF2paramNew$fit = function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  require(randomForest)   # Si no se exige puede fallar al usar varios procesadores
  randomForest(x, y, mtry = param$mtry, ntree = param$ntree, ...)
}
```

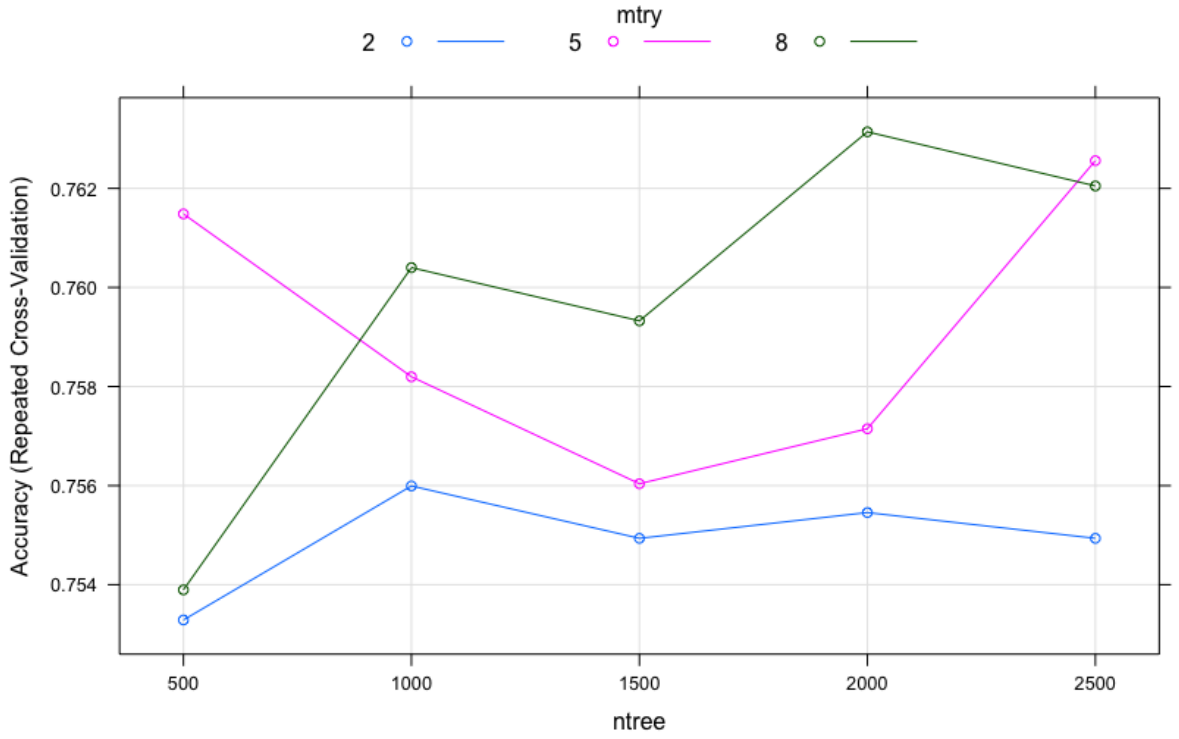


Figure 35: Diagrama de resultados de ajuste de random forest con su versión de 2 hiper-parámetros.

```
# Si queremos anular "grid"
# RF2paramNew$grid= function(x, y, len = NULL, search = "grid") {}
# Si queremos modificar grid, para que también podamos usar tuneLength
RF2paramNew$grid = function(x, y, len = NULL, search = "grid") {
  if (search == "grid") {
    if (is.null(len)) {
      ntreevals = c(500)
    }
    else {
      nup = floor((len-1) / 2)
      ndown = len - nup - 1
      lowseq<-seq(0, 500 , length.out = (ndown+2))[c(-1,-(ndown+2))]
      highseq<-c()
      if (nup>0)
        highseq<-seq(500, 4000, length.out = (nup+1))[-1]
      ntreevals<-as.integer(c(lowseq,500,highseq))
    }
```

```
    out <- expand.grid(
      mtry = caret::var_seq( p = ncol(x), classification = is.factor(y), len = len),
      ntree = ntreevals)
  } else {
    out <- expand.grid(
      mtry = unique(sample(1:ncol(x),size = len, replace = TRUE)),
      ntree = ntreevals)
  }
  out
}
```

Ahora se puede usar `RF2paramNew` como `method` para entrenar un Random Forest con `train()` usando tanto `tuneGrid` como `tuneLength` en el `train()` y optimizaría los dos hiperparámetros `mtry` y `ntree` a la vez. Para que se genere automáticamente la parrilla haremos `tuneGrid=NULL` y pondremos en `tuneLength` el número de diferentes niveles da cada uno de los dos parámetros que vamos a explorar. Por ejemplo:

```
# Hacemos una versión paralela para ejecutarlo más rápido. Haremos un tuneLength de 6
# Eso son 6x6 = 36 combinaciones
set.seed(1234)
# seedsLength es = (numero_repeticiones*numero_remuestreos)+1, en este caso (3*10)+1
seedsLength=31
seeds <- vector(mode = "list", length = seedsLength)
# Crearemos unos pliegues para usar los mismos en todos los modelos diferentes
foldIndexes<-createMultiFolds(pima.Datos.Train[[pima.Var.Salida.Usada]],k=10,times=3)
# combHParam es el número de combinaciones de hiper-parámetros a probar (6x6)
combHParam=36
for(i in 1:seedsLength) seeds[[i]]<- sample.int(n=1000, combHParam)
# Hay que crear una semilla única para el modelo final a entrenar.
seeds[[seedsLength+1]]<-sample.int(1000, 1)
# TrainControl con seeds
pima.TrainCtrl.3cv10.rf2parNew.pl <- trainControl(method='repeatedcv',
                                          number = 10,
                                          repeats = 3,
                                          index = foldIndexes,
                                          seeds=seeds,
                                          verboseIter = T
)
# Esta vez hacemos el cluster poniendo un fichero donde van los mensajes por
# si hay errores
cl <- makeCluster(detectCores()-1, outfile = 'debug.txt')
registerDoParallel(cl)
set.seed(1234)
pima.modelo.3cv10.rf2parNew<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                   pima.Datos.Train[[pima.Var.Salida.Usada]],
                                   method=RF2paramNew,
```

```
                                  tuneGrid = NULL, # Será generada por RF2paramNew$grid()
                                  tuneLength = 6,  # Cuantos niveles por cada parametro
                                  trControl=pima.TrainCtrl.3cv10)
 stopImplicitCluster()
 stopCluster(cl)
 registerDoSEQ()

 pima.modelo.3cv10.rf2parNew

 plot(pima.modelo.3cv10.rf2parNew)
```

```
> pima.modelo.3cv10.rf2parNew
Random Forest 2 parameters with autoGrid

615 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 554, 553, 554, 554, 554, 554, ...
Resampling results across tuning parameters:

  mtry  ntree  Accuracy   Kappa
  2      125   0.7603120  0.4527568
  2      250   0.7581879  0.4472949
  2      375   0.7592720  0.4496080
  2      500   0.7619690  0.4559924
  2     2250   0.7592632  0.4490498
  2     4000   0.7614313  0.4540891
  3      125   0.7576415  0.4468920
  3      250   0.7603384  0.4531962
...
  4      500   0.7614137  0.4614795
  4     2250   0.7657765  0.4708201
  4     4000   0.7646660  0.4668638
  5      125   0.7619602  0.4637732
 ...
  8      500   0.7641371  0.4693549
  8     2250   0.7619513  0.4654110
  8     4000   0.7587167  0.4580651

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were mtry = 4 and ntree = 2250.
> ejcomen(ls(pattern="pima.modelo.*.rf"),"$results$Accuracy","max","Max.Accuracy")
                              Max.Accuracy
```

```
pima.modelo.3cv10.ntree1000.rf    0.7684558
pima.modelo.3cv10.rf              0.7684735
pima.modelo.3cv10.rf.pl1          0.7608761
pima.modelo.3cv10.rf.pl2          0.7608761
pima.modelo.3cv10.rf.plNS         0.7619602
pima.modelo.3cv10.rf2par          0.7631412
pima.modelo.3cv10.rf2parNew       0.7657765
pima.modelo.bstrp25.rf            0.7538253
```

Finalmente parece que la mejor combinación se obtuvo con `pima.modelo.3cv10.rf` que, recordemos, se obtuvo con los valores por defecto sin hacer ninguna búsqueda especial (probó 3 valores de mtry con 500 árboles, y tuvo la suerte de dar con un buen par de hiper-parámetros: mtry=5 y ntree=500). No obstante, en problemas reales más complejos, suele ser necesaria una buena búsqueda de hiper-parámetros. Es importante señalar que en sentido estricto no se puede decir que el mejor era `pima.modelo.3cv10.rf`, puesto que se debe hacer una comparación estadística. Más adelante mostraremos como se comparan estadísticamente diferentes modelos (sección 15) pero adelantaremos ahora un dotplot para comparar los resultados de todos los modelos obtenidos con random forest, usando las diferentes estrategias de búsqueda de hiper-parámetros:

```
listaRF<-lapply(as.list(ls(pattern="pima.modelo.3cv10.*.rf")), FUN=function(x) get(x))
names(listaRF)<-as.list(ls(pattern="pima.modelo.3cv10.*.rf"))
dotplot(resamples(listaRF))
```

obtedremos la figura 36 donde prodemos observar que no hay diferencias significativas entre los modelos obtenidos tras usar todas las diferentes estrategias de búsqueda de mejores hiper-parámetros para este caso y que, en realidad, los mejores modelos obtenidos con cada estrategia son básicamente indistinguibles.

Por supuesto no se tienen que usar todas las estrategias aquí mostradas, tan solo usar la que uno considere más adecuada para el problema o con la fase de exploración en que se encuentre uno. Es normal hacer pruebas ligeras (con pocos parámetros) cuando se desea tomar una decisión de pre-proceso o el modelo específico a utilizar, y solo entrar en una búsqueda más extensa de combinaciones una vez decidido todo el preproceso y la técnica/modelo que mejor parecía comportarse en las pruebas preliminares.

12 Usando varios procesadores

Aunque ya hemos usado un par de ejemplos de uso de varios procesadores vamos a explicarlo un poco más detenidamente. La mayoría de ordenadores modernos tienen procesadores con múltiples núcleos y capacidades de computación concurrente. Caret es capaz de paralelizar el proceso de ajuste si los algoritmos que se usan también tienen dicha capacidad. Hay que tener en cuenta algunas cosas respecto al procesamiento paralelo que hace caret.

1. Caret lo que hace es enviar a cada "trabajador" el ajuste de un modelo con unos hiperparámetros determinados, es decir, que no se paraleliza el "interior" de un ajuste/entrenamiento, con lo que es posible ver que en los últimos experimentos haya procesadores inactivos (si hay 10 ajustes por hacer, y tardan más o menos lo mismo cada uno, con 3 procesadores habrá un momento en que el décimo trabajo se haga en un procesador y los otros se queden idle, el último trabajo no se divide en 3 partes).

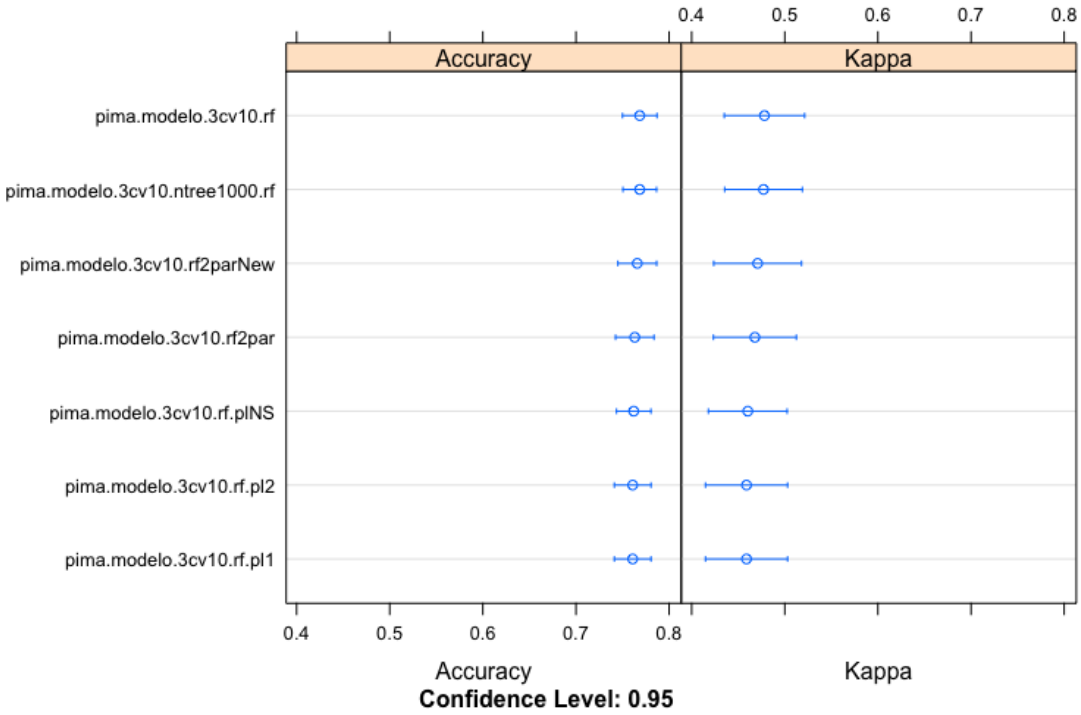


Figure 36: Diagrama de resultados de todos los experimentos de random forest.

2. Trabajar en paralelo hace que se pierda el control sobre la semilla de números aleatorios lo que normalmente arruina la reproductibilidad de los experimentos. Como se ha mencionado en 11.2.2.3 la forma de solucionarlo es pasando las semillas con el agrumento `seeds` de `trainControl()`.

3. La implementación en Windows y otras plataformas del paralelismo es diferente, por lo que tendrás que incluir código que distinga la situación si quieres que tu código sea multiplataforma.

4. Si vas a seguir trabajado con la máquina mientras haces experimentos no es buena idea usar todos lo núcleos. Deja alguno para poder seguir usando la máquina puesto que usará toda la potencia de cada núcleo y te hara que la máquina no responda a otras tareas.

5. Al paralelizar se pierde la opción de ver por la pantalla el progreso (no funciona sobre la consola el verbose) pero se pueden redirigir las salidas de todos los procesos a un fichero al ejecutar el comando `makeCluster(workersToUse, outfile='debut.txt')`. Por supuesto puedes usar el nombre del fichero que desees (incluso redirigirlo).

Como ejemplo de código para manejar multiplataforma Windows/Mac puedes usar:

```
# Para usar múltiple cores

useParallel<-FALSE # No lo usaremos en el tutorial
debugFile <- NULL # Poner una cadena si se quiere volvar a un fichero la salida de los procesos

if (useParallel) {
  require(doParallel)
  systemInfo<-Sys.info()
  logicalCores<-detectCores(logical=T)
  trueCores<-detectCores(logical=F)
  logCporTrueC=logicalCores/trueCores
# No uses todos los cores de la máquina si vas a usar la consola o el GUI
# deja siempre 1 libre o se queda calculando y no responde mientras al teclado
  workersToUse<-(trueCores-1)*logCporTrueC
  if (systemInfo["sysname"]=="Windows")
          require(foreach)
  if (is.null(debugFile))
    cl <- makeCluster(workersToUse)
  else
   cl<-  makeCluster(workersToUse, outfile=debugFile)
    registerDoParallel(cl, cores=workersToUse)
}
```

Cuando termines de usar paralelo deberías desactivarlo.

```
# para parar el uso de paralelo
if (useParallel) {
  require(doParallel)
  stopImplicitCluster()
  stopCluster(cl)
  registerDoSEQ()
}
```

Nota: la última versión de caret parece tener un problema con multiprocesadores para algunos algoritmos. Da el error siguiente:

```
Error in e$fun(obj, substitute(ex), parent.frame(), e$data) :
  unable to find variable "optimismBoot"
```

Se soluciona con el siguiente comando:

```
devtools::install_github('topepo/caret/pkg/caret')
# y quizás necesites ejecutar este también
requireNamespaceQuietStop<-caret:::requireNamespaceQuietStop
```

Si continúas teniendo problemas, reinicia R y no utilices parallel por ahora hasta que solucionen el problema.

Con esto se termina la cuarta sesión de prácticas de Caret.

```
######################
# Si no has grabado las sesiones anteriores...
library(caret)
library(mlbench)
library(doParallel)
library(randomForest)
data("PimaIndiansDiabetes")
# Usamos el dataset PimaIndiansDiabetes y hacemos una partición al 80%
pima.Datos.Todo<-PimaIndiansDiabetes
pima.Var.Salida.Usada<-c("diabetes")
pima.Vars.Entrada.Usadas<-setdiff(names(pima.Datos.Todo),pima.Var.Salida.Usada)

set.seed(1234)
pima.TrainIdx.80<- createDataPartition(pima.Datos.Todo[[pima.Var.Salida.Usada]],
                                       p=0.8,
                                       list = FALSE,
                                       times = 1)
pima.Datos.Train<-pima.Datos.Todo[pima.TrainIdx.80,]
pima.Datos.Test<-pima.Datos.Todo[-pima.TrainIdx.80,]
# Usamos el dataset PimaIndiansDiabetes
set.seed(1234)
# seedsLength es = (numero_repeticiones*numero_remuestreos), en este caso (3*10)+1
seedsLength=30
# Se necesita una semilla adicional para el modelo final escogido
seeds <- vector(mode = "list", length = (seedsLength+1))
# combHParam es el número de combinaciones de hiper-parámetros a probar
combHParam=100
for(i in 1:seedsLength) seeds[[i]]<- sample.int(n=10000, combHParam)
# Hay que crear una semilla única para el modelo final a entrenar.
seeds[[seedsLength+1]]<-sample.int(10000, 1)
# TrainControl con seeds
pima.TrainCtrl.3cv10.2ClssSum <- trainControl(
  method = "repeatedcv"
  ,number = 10 ,repeats = 3 ## Crosvalidación de 10 pliegues, 3 repeticiones
  ,seeds=seeds              # Para usar varios procesadores
  ,verboseIter=F
  ,summaryFunction = twoClassSummary
  ,classProbs = TRUE  # Para usar ROC necesita calcular las classProbs

  )

# Ejecuta un comando sobre lista de modelos
ejcomen <- function(mods, campo, command,nombre="Valor") {
  lst<-as.list(mods)
```

```
  names(lst)<-mods
  out<-do.call("rbind",
               sapply(
                 mods,
                 simplify = F,
                 FUN = function(x, com = command)
                   eval(parse (
                     text = paste(com, "(", x, campo, ")", sep = "")
                   ))
               ))
  colnames(out)<-nombre
  out }

# Hasta aquí si no has grabado sesiones anteriores
#####################################################
```

13 Submuestreo con clases desbalanceadas

Cuando se trabaja con problemas de clasificación no es infrecuente encontrarte con datos donde las clases a inferir están muy desbalanceadas. Por desgracia esto suele tener un impacto muy negativo en el ajuste de los modelos. Una de las técnicas habituales para resolver este problema es submuestrear los datos para tratar de balancear las clases. Normalmente nos encontramos tres tipos:

- ***down-sampling***: Consiste en reducir el número de ejemplos de las clases más frecuentes para igualar la clase menos frecuente. Por ejemplo, si tenemos 100 ejemplos y dos clases, una con 25 casos y la otra con 75, el downsampling lo que haría sería coger aleatoriamente 25 ejemplos de la clase con 75 muestras y usar un dataset con 50 (25 de cada clase). Caret tiene la función `downSample()` que implementa esta técnica.
- ***up-sampling*** : En este caso hace lo contrario, remuestrea (con reemplazamiento) la clase minoritaria para igualarla a la mayoritaria (repite varios datos). En el ejemplo anterior escogería 50 nuevos casos aleatoriamente entre los 25 casos (con reemplazamiento) y se usaría un dataset con 150 ejemplos (75 de cada clase). Caret implementa esto con la función `upSample()`.
- ***Métodos hibridos***: Hacen un poco de cada, p.e. **SMOTE** y **ROSE** son dos procedimientos que *down-samplean* la clase mayoritaria y, al mismo tiempo, crean nuevos puntos partiendo de la minoritaria (evitan duplicados). Los packages **DMwR** y **ROSE** implementan estos métodos.

Estos procedimientos se deben llevar a cabo **solamente** en los datos de **Training**, y **nunca** en los de **Test**. El conjunto de Test se usa para estimar el comportamiento con datos reales y cuando se submuestrea se crea una situación artificial que, si bien es para ayudar a ajustar el modelo, no se corresponde con lo que se quiere modelar.

Como se puede suponer, este submuestreo no sale gratis. Por un lado recordemos que el conjunto de hiper-parámetros se escoge con las estimaciones de los conjuntos de validación y estos están submuestreados, con lo que hay cierto ruido que suele llevar a una estimación optimista del verdadero rendimiento. Por otro lado se entrena sobre datos que ya no son una representación fidedigna del sistema original y eso introduce incertidumbre en el modelo.

```
# Las clases de diabetes están desequilibrados con doble de negativos
# Probamos a equilibrarlos con los diferentes métodos
library(DMwR)
set.seed(12345)
pima.Datos.Train.downsmpld<-downSample(x=pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                       y=pima.Datos.Train[[pima.Var.Salida.Usada]],
                                       yname=pima.Var.Salida.Usada)
set.seed(1234)
pima.Datos.Train.upsmpld<-upSample(x=pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                   y=pima.Datos.Train[[pima.Var.Salida.Usada]],
                                   yname=pima.Var.Salida.Usada)
# Se necesitan los paquetes DMwR y caTools para usar SMOTE
if(!require("DMwR")) {install.packages("DMwR");require("DMwR")}
if(!require("caTools")) {install.packages("caTools");require("caTools")}
# Smote y Rose necesita el formato de fórmula para pasarle los datos
form<-formula(paste(pima.Var.Salida.Usada,paste(pima.Vars.Entrada.Usadas,sep="",
                                                collapse="+"), sep="~",collapse=""))
set.seed(1234)
pima.Datos.Train.smote<-SMOTE(form,data=pima.Datos.Train)
# Se necesita el paquete ROSE para usar ROSE
if(!require("ROSE")) {install.packages("ROSE");require("ROSE")}
set.seed(1234)
pima.Datos.Train.rose<-ROSE(form,data=pima.Datos.Train)$data
######################

cl <- makeCluster(detectCores()-1,outfile='debug.txt')
registerDoParallel(cl)

# Hacemos ahora algunos modelos con estos datos
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.ROC<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="gbm", trControl=pima.TrainCtrl.3cv10.2ClssSum
  ,tuneLength = 10
  ,metric ="ROC"
  ,verbose=F # Este modelo es verbose por defecto
)
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.ROC.ds<-train(
  pima.Datos.Train.downsmpld[pima.Vars.Entrada.Usadas],
  pima.Datos.Train.downsmpld[[pima.Var.Salida.Usada]],
  method="gbm", trControl=pima.TrainCtrl.3cv10.2ClssSum
  ,tuneLength = 10
```

```
  ,metric ="ROC"
  ,verbose=F # Este modelo es verbose por defecto
)
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.ROC.us<-train(
  pima.Datos.Train.upsmpld[pima.Vars.Entrada.Usadas],
  pima.Datos.Train.upsmpld[[pima.Var.Salida.Usada]],
  method="gbm", trControl=pima.TrainCtrl.3cv10.2ClssSum
  ,tuneLength = 10
#  ,metric ="ROC"
  ,verbose=F # Este modelo es verbose por defecto
)
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.ROC.smote<-train(
  pima.Datos.Train.smote[pima.Vars.Entrada.Usadas],
  pima.Datos.Train.smote[[pima.Var.Salida.Usada]],
  method="gbm", trControl=pima.TrainCtrl.3cv10.2ClssSum
  ,tuneLength = 10
  ,metric ="ROC"
  ,verbose=F # Este modelo es verbose por defecto
)
set.seed(1234)
pima.modelo.3cv10.tl10.gbm.ROC.rose<-train(
  pima.Datos.Train.rose[pima.Vars.Entrada.Usadas],
  pima.Datos.Train.rose[[pima.Var.Salida.Usada]],
  method="gbm", trControl=pima.TrainCtrl.3cv10.2ClssSum
  ,tuneLength = 10
  ,metric ="ROC"
  ,verbose=F # Este modelo es verbose por defecto
)
stopImplicitCluster()
stopCluster(cl)
registerDoSEQ()
```

Hay que tener cuidado cuando comparamos los modelos. Podemos tener la tentación de extraer directamente los resultados de entrenamiento de la métrica y compararlos.

```
> ejcomen(ls(pattern="pima.modelo.*.gbm.ROC*"),"$results$ROC","max","Max.ROC")
                                    Max.ROC
pima.modelo.3cv10.tl10.gbm.ROC       0.8336418
pima.modelo.3cv10.tl10.gbm.ROC.ds    0.8344843
pima.modelo.3cv10.tl10.gbm.ROC.rose  0.7817136
pima.modelo.3cv10.tl10.gbm.ROC.smote 0.9790078
pima.modelo.3cv10.tl10.gbm.ROC.us    0.9117917
```

pero esto nos llevaría a error porque los modelos han sido entrenados con conjuntos de datos bastante diferentes y esos valores están calculados sobre dichos conjuntos de entrenamiento diferentes.

La forma correcta de comparar estos modelos que han sido entrenados con versiones diferentes de los datos sería viendo los resultados sobre el conjunto de test. En particular vamos a ejecutar unos comandos para estimar el rendimiento sobre el conjunto de Test (que sí es el mismo para todos los modelos). De todos modos en la sección 15 se verá como comparar modelos, aunque entrenando el mismo conjunto de datos.

Veamos el código para comparar sobre Test usando ROCs (una versión del original disponible en la documentación de caret en `http://topepo.github.io/caret/subsampling-for-class-imbalances.html`.

```
pima.gbm.models <- list(original = pima.modelo.3cv10.tl10.gbm.ROC,
                        down = pima.modelo.3cv10.tl10.gbm.ROC.ds,
                        up = pima.modelo.3cv10.tl10.gbm.ROC.us,
                        SMOTE = pima.modelo.3cv10.tl10.gbm.ROC.smote,
                        ROSE = pima.modelo.3cv10.tl10.gbm.ROC.rose)

resampling <- resamples(pima.gbm.models)

test_roc <- function(model, data, levnames,varEnt,varSal) {
  if(!require("pROC")) {install.packages("pROC");require("pROC")}
  pred<-predict(model, data[varEnt], type = "prob")[, levnames[2]]
  roc_obj <- roc(data[[varSal]], pred, levels = levnames)
  ci(roc_obj)  # Intervalo de confianza al 95%
}

pima.gbm.results.test <- lapply(pima.gbm.models, test_roc, data = pima.Datos.Test,
                            varEnt=pima.Vars.Entrada.Usadas,
                            varSal=pima.Var.Salida.Usada,
                            levnames=levels(pima.Datos.Test[[pima.Var.Salida.Usada]]))
pima.gbm.results.test <- lapply(pima.gbm.results.test, as.vector)
pima.gbm.results.test <- do.call("rbind", pima.gbm.results.test)
colnames(pima.gbm.results.test) <- c("lower CI95%", "ROC", "upper CI95%")
pima.gbm.results.test <- as.data.frame(pima.gbm.results.test)
# Resultado sobre VALIDACIÓN en entrenamiento
summary(resampling, metric = "ROC")
# Resultado sobre TEST
pima.gbm.results.test
```

Con `resampling()` vemos los resultados sobre los conjuntos de validación del entrenamiento (se obtienen a partir delos diferentes resultados de validación de los varios modelos entrenados con los mismos hiperparámetros que el modelo final escogido, recuerda que no se hace sobre el conjunto de datos que se usan para entrenar cada experimento, sino los resultados sobre los que se apartan para validar que, en principio, no ha visto ese modelo en particular), y `pima.bgm.results.test` nos proporciona resultados del único modelo final escogido sobre el único conjunto Test (calculando también el intervalo de confianza al 95% sobre el valor medio de ROC sobre el conjunto de Test):

```
> # Resultado sobre VALIDACIÓN en entrenamiento
> summary(resampling, metric = "ROC")

Call:
```

```
summary.resamples(object = resampling, metric = "ROC")

Models: original, down, up, SMOTE, ROSE
Number of resamples: 30

ROC
              Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
original 0.7250000 0.8108360 0.8346591 0.8336418 0.8651786 0.9136364    0
down     0.5865801 0.8136361 0.8495671 0.8344843 0.8764758 0.9297052    0
up       0.8443750 0.8970312 0.9162500 0.9117917 0.9295313 0.9706250    0
SMOTE    0.9504472 0.9694097 0.9810669 0.9790078 0.9884168 0.9992844    0
ROSE     0.6691810 0.7371767 0.7930871 0.7817136 0.8316272 0.8927083    0

> # Resultado sobre TEST
> pima.gbm.results.test
         lower CI95%       ROC upper CI95%
original   0.7632657 0.8279245   0.8925834
down       0.7676209 0.8322642   0.8969074
up         0.6938034 0.7673585   0.8409135
SMOTE      0.7103017 0.7815094   0.8527172
ROSE       0.7653667 0.8292453   0.8931239
```

Viendo el resultado de remuestreo sobre la validación del conjunto de entrenamiento se concluiría, erróneamente, que el modelo entrenado con tratamiento de clases desbalanceadas SMOTE va a producir unos resultados casi perfectos sobre nuevos datos. En realidad la estimación está desencaminada porque se han aumentado mucho el número de ejemplos de ambas clases y hace que sea muy posible que aparezca el mismo dato tanto en el conjunto de entrenamiento como de validación, con lo que el conjunto de validación no es realmente independiente de su entrenamiento y sobrevalora demasiado el resultado. Algo parecido le pasa al método up-sampling, pues la clase minoritaria replica el mismo dato varias veces y de nuevo es posible que un mismo dato se use para entrenar y validar a la vez (una o varias copias irían a entrenar y alguna al de validación) y de nuevo sobrevalora el resultado de validación.

Viendo los resultados sobre test, y dados los intervalos de confianza, los resultados son similares para todos los métodos pues los intervalos se solapan. En realidad solo el conjunto original muestra un comportamiento esperable respecto a las diferencias entre estimaciones de remuestreo de validación y estimaciones sobre test (un valor ligeramente peor del test frente a validación, en este caso 0.833 frente a 0.827).

14 Pre-Procesado de datos (y IV): Selección de variables.

Incluso cuando hemos preprocesado los datos, y pensamos que todas esas variables de entrada tienen información útil para hacer la predicción, es habitual que cuando se los ofrecemos a los métodos,nos encontremos con que algunas variables son ignoradas o que apenas aportan al modelo. Es más, es posible que la importancia de las variables dependa del propio modelo (y algunas se usen en unos modelos y no en otros).

Caret dispone del método `varImp()` que nos informa de la importancia que tienen las variables de entrada en el modelo final entrenado. Esta información se puede utilizar para decidir repetir el experimento eliminando dichas variables (y obteniendo así modelos más simples).

```
> library(gbm)
Loaded gbm 2.1.4
> varImp(pima.modelo.3cv10.tl10.gbm.ROC)
gbm variable importance

         Overall
glucose  100.000
mass      44.660
age       30.304
pedigree  15.423
insulin    5.222
pregnant   4.820
triceps    3.135
pressure   0.000
```

Así por ejemplo, en el modelo gbm anterior vemos que la variable `pressure` probablemente se pudiera eliminar para ese modelo, así como `insulin`, `triceps` o `pregnant`.

Otro aspecto a tener en cuenta es que algunos modelos hacen una selección de variables internamente como parte de su ajuste. Quizá el caso más claro son los árboles de decisión que, por su propia forma de construirse, tienen una selección implícita de variables (es posible que haya variables sobre las que nunca llega a preguntarse y serían eliminables). Lo habitual es que esos mecanismos de selección sean más eficientes que los que ahora mencionaremos. Muchos de esos métodos tienen un método llamado `predictors()` que devuelve un vector indicando las variables que usa finalmente el modelo.

Siguiendo con el mismo ejemplo podemos ver esa información accediendo directamente al modelo final mediante la columna `finalModel` del modelo que nos devuelve caret, aunque no hace falta extraer `finalModel` y se puede aplicar `predictors()` directamente sobre el modelo de caret:

```
> predictors(pima.modelo.3cv10.tl10.gbm.ROC$finalModel)
[1] "pregnant" "glucose"  "pressure" "triceps"  "insulin"  "mass"     "pedigree"
[8] "age"
> pima.modelo.3cv10.tl10.gbm.ROC$finalModel
A gradient boosted model with bernoulli loss function.
100 iterations were performed.
There were 8 predictors of which 8 had non-zero influence.
```

aunque en este caso nos diría que `gbm` usa todos los predictores. Otros modelos anteriores, como uno de los modelos de árbol de decisión, si que no usan todos los predictores:

```
> predictors(pima.modelo.bstrp25.rpart)
[1] "glucose"  "insulin"  "pedigree" "mass"     "triceps"  "age"
```

Los valores de `predictors()` no tienen porqué coincidir con las estimaciones de `varImp()`, ya que `predictors()` muestra las usadas finalmente en el modelo final mientras que `varImp()` muestra sobre varios modelos de los entrenados.

```
> varImp(pima.modelo.bstrp25.rpart)
rpart variable importance
```

```
         Overall
glucose   100.00
mass       55.11
age        47.68
insulin    23.17
pregnant   22.50
pressure    0.00
triceps     0.00
pedigree    0.00
```

Dejándo de lado la selección de variables inherente o interna de ciertos algoritmos, hay mecanismos generales de selección de variables, como los que hemos visto en clase. La mayoría de algoritmos para seleccionar variables caen dentro de dos tipos:

- Wrappers: Como los vistos en clase. Básicamente son mecanismos que añaden o quitan alguna variable y evalúan el rendimiento del modelo con ese conjunto reducido de variables. Si mejora o si no empeora, significativamente, el modelo final (suele depender de si se añaden o se quitan variables), entonces se siguen eliminando (o añadiendo). En cierto sentido las variables usadas serían como otro hiper-parámetro a ajustar. Los wrappers de los que dispone caret son la eliminación recursiva de características (Recursive Feature Elimination), algoritmos genéticos y simulated annealing.
- Filtros. Estos métodos evaluan la relevancia de las variables de entrada antes de ajustar los modelos predictivos y deciden usar solo los que cumplan algún criterio. Caret dispone de un mecanismo para trabajar con filtros de una variable. Más info en `http://topepo.github.io/caret/feature-selection-using-univariate-filters.html`.

14.1 RFE

Este método de eliminar variables se basa en la selección hacia atrás vista en clase. Si queréis más información podéis consultar en:

`http://topepo.github.io/caret/recursive-feature-elimination.html`

14.2 Algoritmos genéticos

El problema de una selección hacia atrás (o hacia delante) es que es un tipo de búsqueda voraz y que trabaja de manera incremental uno a uno, lo que restringe la búsqueda y hace imposible explorar ciertas combinaciones. La selección de variables es, en realidad, la búsqueda de un subconjunto bueno de variables dentro del espacio de búsqueda de todos los posibles subconjuntos de variables de diferentes tamaños. Dicho de otro modo, es una optimización (en la que no necesariamente se busca el óptimo global pero si que se intenta no quedarse con óptimos locales) en la que se pretende hacer una búsqueda lo más distribuída posible. En resumen, que se pueden utilizar diversos algoritmos de optimización general si se definen algunos de los elementos que necesitan. En particular los algoritmos genéticos pueden llegar a funcionar bastante bien cuando se tiene poca información sobre el espacio de soluciones pero se puede definir una fitness-function. Caret incorpora la posibilidad de usar esta técnica. Más información en:

`http://topepo.github.io/caret/feature-selection-using-genetic-algorithms.html`

14.3 Simulated Annealing

Simulated Annealing es otra técnica bastante utilizada en optimización. Caret también la incluye para seleccionar variables. Más detalles en:

`http://topepo.github.io/caret/feature-selection-using-simulated-annealing.html`

15 Comparando modelos

Ya vimos en la sección 11 que `train()` de caret nos obtiene el mejor modelo para ese método/algoritmo (dentro de la exploración de diferentes hiperparámetros y otros mecanismos y criterios que le hayamos indicado en `trainControl()` o en `train()`). Es posible que, no obstante, querramos conocer cuales han sido los resultados para las distintas posibilidades, por si quisieramos repetir el experimento con otras configuraciones. Para observar esos resultados (explorar dentro del mismo modelo) se pueden utilizar diversas funciones de `lattice` con el propio modelo como parámetro. Ya vimos como usar plot `plot()` con el parámetro `plotType` "level", "scatter" o "line" para ver los resultados de los remuestreos frente a los hiperparámetros a ajustar. Tenemos también la opción de usar `histogram()` y `densityplot()` para ver las distribuciones de los resultados del ajuste de hiperparámetros. No obstante el dibujo obtenido con esos diagramas dependerá de si se ha guardado toda la información de los remuestreos de validación o solo el del ejemplo final. Por ejemplo, vamos a hacer un modelo que solo guarda el ejemplo final:

```
# Usamos el dataset diabetes Pima

set.seed(1234)
pima.modelo.3cv10.tl7.rsFinal.rf<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                        pima.Datos.Train[[pima.Var.Salida.Usada]],
                                        method="rf", trControl=pima.TrainCtrl.3cv10
                                        ,tuneLength = 7
                                        ,verbose=F
)
```

Si ahora ejecutamos los siguientes comandos:

```
trellis.par.set(caretTheme())

d1<-densityplot(pima.modelo.3cv10.tl7.rsFinal.rf, pch = "|")
d2<-histogram(pima.modelo.3cv10.tl7.rsFinal.rf)


print(d2,position=c(0,0,1,0.5),more=T)
print(d1,position=c(0,0.5,1,1))
```

obtendríamos la figura 37.

En cambio si tenemos intención de explorar los resultados de las diferentes combinaciones de hiperparámetros se debe incluir la opción `"all"` en el parámetro `returnResamp` del `trainControl()`, para que almacene toda la información necesaria para los diagramas. Si hacemos ahora un modelo con toda esa información:

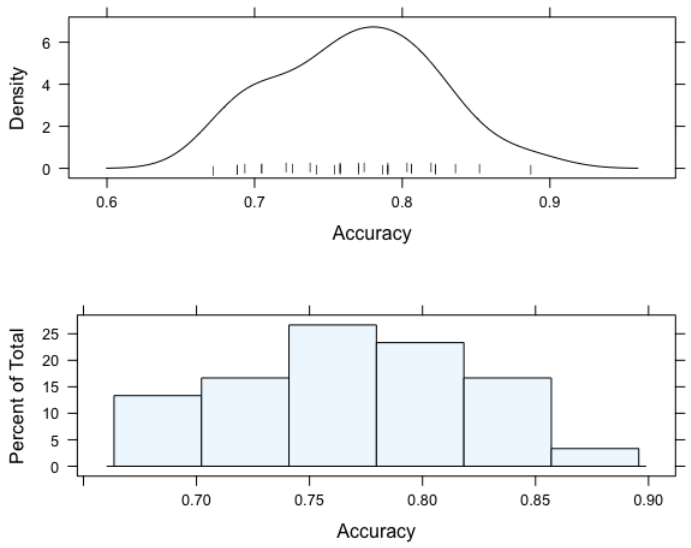


Figure 37: Histograma y densityplot de los resultados de validación del ajuste de hiperparámetros del modelo final de Radial Functions

```
# Hacemos un modelo que guarda TODOS los remuestreos del mejor
# Usamos el dataset diabetes Pima
pima.trainCtrl.3cv10.resampAll <- trainControl(
  method = "repeatedcv"
  ,number = 10 ,repeats = 3, ## Crosvalidación de 10 pliegues, 3 repeticiones
  # Que muestre información mientras entrena
  verboseIter=T,
  returnResamp = "all"  # Guardamos todo para hacer diagramas
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.rf<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                                      pima.Datos.Train[[pima.Var.Salida.Usada]],
                                      method="rf", trControl=pima.trainCtrl.3cv10.resampAll
                                      ,tuneLength = 7
                                      ,verbose=F
)

d1<-densityplot(pima.modelo.3cv10.tl7.rsAll.rf, pch = "|")
d2<-histogram(pima.modelo.3cv10.tl7.rsAll.rf)
```

obtendríamos los diagramas que muestran la figura 38 que muestra todos los hiper-parámetros explorados por train. Fíjate que, aunque `tuneLength=15`, caret ha decidido explorar solo 7 valores del hiper-parámetro `Mtry`.

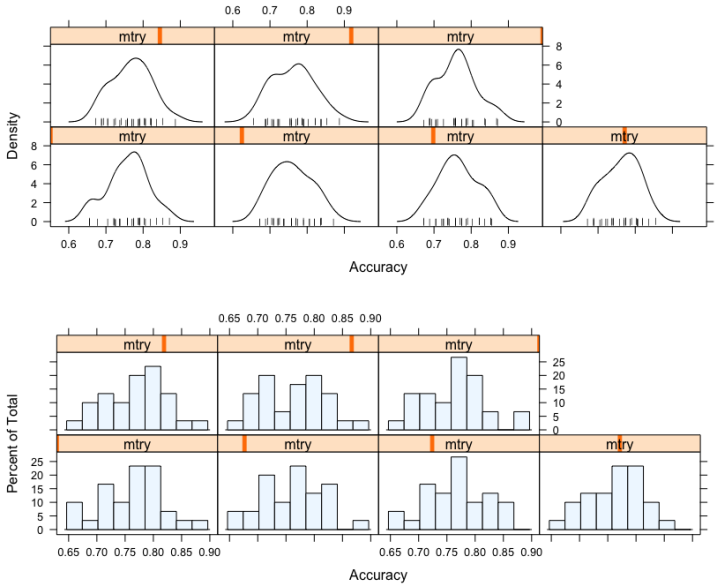


Figure 38: Histograma y densityplot de los resultados de validación del ajuste de hiperparámetros de todas las configuraciones probadas de Radial Functions

Si tenemos toda la información también podemos usar otros gráficos para explorar la búsqueda de hiper-parámetros modelo a modelo. Por ejemplo, `xyplot()` y `stripplot()` muestran las estadísticas de resampling del ajuste de hiper-parámetros (muestra en el eje x el hiper-parámetro que tenga más valores diferentes y los demás actúan como variables condicionantes (paneles y grupos). Veamos un ejemplo sobre radial functions:

```
modToPlot<-pima.modelo.3cv10.tl7.rsAll.rf

d1<-xyplot(modToPlot,
       metric = "Accuracy",
       type = c("p", "a"))

d2<-stripplot(modToPlot,
          horizontal = FALSE,
          jitter = T)

print(d2,position=c(0,0,1,0.5),more=T)
print(d1,position=c(0,0.5,1,1))
```

que genera los diagramas de la figura 39.

No obstante lo habitual es que tratemos de ajustar diferentes modelos y técnicas que son diferentes. Para ver como hacerlo vamos a crear unos cuantos modelos sobre algún conjunto de datos. Usaremos el dataset de diabetes de los indios Pima:

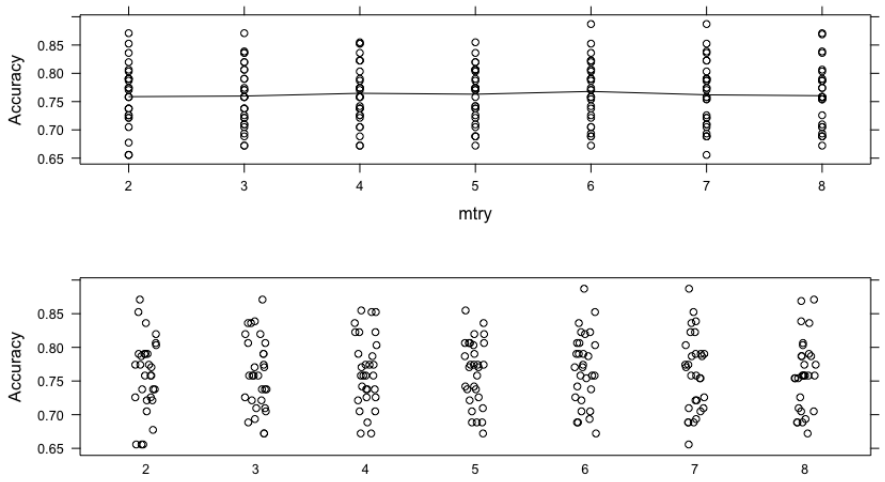


Figure 39: xyplot y stripplot de los resultados de validación del ajuste de hiperparámetros de todas las configuraciones probadas de Radial Functions

```
# Generaremos varios modelos alternativos

# Usamos el dataset diabetes Pima
# Hacemos ahora algunos modelos con estos datos

set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.lda.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="lda", trControl=pima.trainCtrl.3cv10.resampAll
  # sin hiper-parámetros                  ,tuneLength = 10
  ,verbose=F
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.glm.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="glm",
  trControl=pima.trainCtrl.3cv10.resampAll
  # sin hiper-parámetros                  ,tuneLength = 10
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.rpart.Acc<-train(
```

```
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="rpart",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.knn.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="knn",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.svm.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="svmRadial",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7
  ,verbose=F
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.rf.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="rf",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7
  ,verbose=F
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.gbm.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="gbm",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7
  ,verbose=F # Este modelo es verbose por defecto
)
set.seed(1234)
pima.modelo.3cv10.tl7.rsAll.nnet.Acc<-train(
  pima.Datos.Train[pima.Vars.Entrada.Usadas],
```

```
  pima.Datos.Train[[pima.Var.Salida.Usada]],
  method="nnet",
  trControl=pima.trainCtrl.3cv10.resampAll
  ,tuneLength = 7   # Puedes poner menos porque tarda mucho
  ,verbose=F
  ,MaxNWts = 10000 #argumento extra
  ,trace = F       # Evita msgs de convergencia
)
```

Una vez que tenemos todos estos modelos diferentes, con sus hiper-parámetros ajustados, es necesario que podamos comparar entre modelos para decidir cual es el modelo final que nos quedamos. Para hacer esto utilizaremos la función `resamples()`, que toma como parámetro una lista con los modelos a comparar. Los nombres de los elementos de cada lista deberían ser acrónimos o abreviaturas que permitan distinguir con facilidad los modelos.

Al objeto devuelto se le puede aplicar el comando `summary()` para tener un resultado descriptivo de la comparativa.

```
pima.modelList.3cv10.tl7.rsAll.Acc<-list(
  LDA=pima.modelo.3cv10.tl7.rsAll.lda.Acc
  ,GLM=pima.modelo.3cv10.tl7.rsAll.glm.Acc
  ,CART=pima.modelo.3cv10.tl7.rsAll.rpart.Acc
  ,KNN=pima.modelo.3cv10.tl7.rsAll.knn.Acc
  ,SVMrad=pima.modelo.3cv10.tl7.rsAll.svm.Acc
  ,RF=pima.modelo.3cv10.tl7.rsAll.rf.Acc
  ,GBM=pima.modelo.3cv10.tl7.rsAll.gbm.Acc
  ,NNET=pima.modelo.3cv10.tl7.rsAll.nnet.Acc
)

pima.resamps.3cv10.tl7.rsAll.Acc<-resamples(pima.modelList.3cv10.tl7.rsAll.Acc)
summary(pima.resamps.3cv10.tl7.rsAll.Acc)
```

El comando `resamples()` generará varios Warnings. No pasa nada, solo aparecen por usar la opción `returnResamp = "all"` (lo normal es guardar solo el final). El comando funciona bien (se queda con los resultados del modelo final). El `summary()` produce la siguiente información:

```
> summary(pima.resamps.3cv10.tl7.rsAll.Acc)

Call:
summary.resamples(object = pima.resamps.3cv10.tl7.rsAll.Acc)

Models: LDA, GLM, CART, KNN, SVMrad, RF, GBM, NNET
Number of resamples: 30

Accuracy
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
LDA    0.6721311 0.7377049 0.7580645 0.7641460 0.7894632 0.8548387    0
GLM    0.6721311 0.7377049 0.7580645 0.7630531 0.7903226 0.8548387    0
```

```
CART   0.6557377 0.7287811 0.7480169 0.7489424 0.7741935 0.8387097    0
KNN    0.6721311 0.7049180 0.7398202 0.7439979 0.7732681 0.8387097    0
SVMrad 0.6557377 0.7377049 0.7580645 0.7554204 0.7837123 0.8387097    0
RF     0.6721311 0.7287811 0.7704918 0.7679094 0.8056584 0.8870968    0
GBM    0.6885246 0.7387626 0.7704918 0.7663053 0.8032787 0.8548387    0
NNET   0.5483871 0.6451613 0.6910365 0.6936894 0.7258065 0.8387097    0

Kappa
              Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
LDA     0.25700365 0.3821805 0.4519961 0.4496583 0.5186156 0.6658683    0
GLM     0.25700365 0.3821805 0.4518943 0.4489542 0.5223454 0.6658683    0
CART    0.15109344 0.3767561 0.4236535 0.4224139 0.4791620 0.6075949    0
KNN     0.24380165 0.3109873 0.4025593 0.4061069 0.4808692 0.6197007    0
SVMrad  0.19179811 0.3759470 0.4267787 0.4290174 0.4841903 0.6327014    0
RF      0.27380952 0.3936315 0.4769404 0.4775061 0.5598847 0.7559055    0
GBM     0.21210061 0.3781122 0.4553474 0.4497574 0.5378447 0.6729191    0
NNET   -0.09318996 0.1569003 0.2634055 0.2772643 0.3877759 0.6327014    0
```

También se pueden dibujar los resultados de comparar los resultados de validación de los modelos finales con los diagramas disponibles para `lattice`, en particular, `densityplot()` (density plot), `bwplot()` (box-whisker plot), `splom()` (Scatter plot matrices), `parallelplot()` (Parallel Coordinate plot) y `xyplot()` (bivariate scatterplot).

```
# Diversos diagramas para comparar los modelos

trellis.par.set(caretTheme())

densityplot(pima.resamps.3cv10.tl7.rsAll.Acc,
            scales =list(x = list(relation = "free"),
                         y = list(relation = "free")),
            auto.key = list(columns = 4),
            pch = "|")

bwplot(pima.resamps.3cv10.tl7.rsAll.Acc, metric = "Accuracy")

splom(pima.resamps.3cv10.tl7.rsAll.Acc, metric = "Accuracy")

splom(pima.resamps.3cv10.tl7.rsAll.Acc, variables = "metrics",
      auto.key= list(columns=4))

parallelplot(pima.resamps.3cv10.tl7.rsAll.Acc, metric = "Accuracy"
             ,auto.key=T)

xyplot(pima.resamps.3cv10.tl7.rsAll.Acc,
       models = c("GBM","NNET"),
```

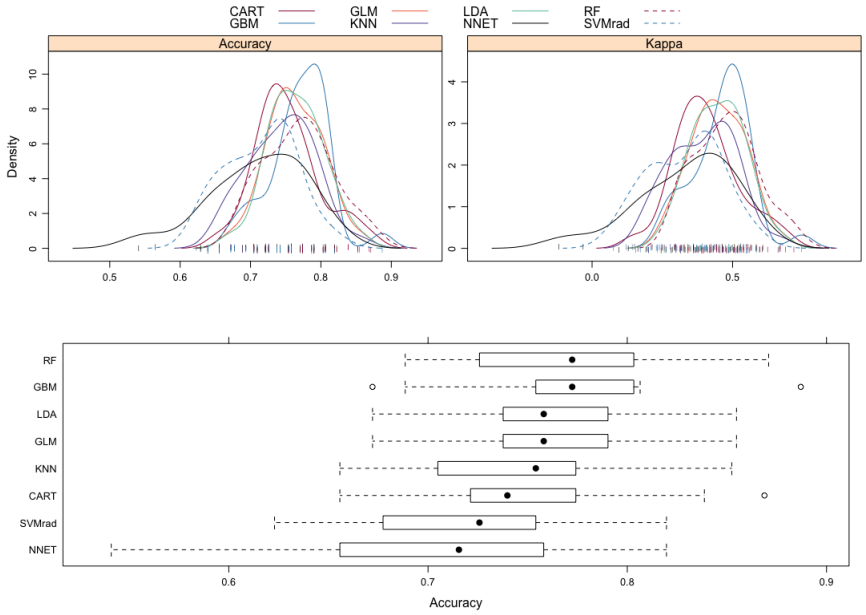


Figure 40: Diagramas `densityplot()` y `bwplot()` de los remuestreos de los modelos comparados.

```
       metric = "Accuracy")

xyplot(pima.resamps.3cv10.tl7.rsAll.Acc,
       models = c("GBM","NNET"),
       metric = "Accuracy"
       ,what="BlandAltman")
```

Aunque probablemente el más útil sea el `dotplot()` (Cleveland dot plot), puestro que muestra el intervalo de confianza al 95% del rendimiento sobre los conjuntos de validación. Es muy, muy importante no confundir esta medida de rendimiento con la estimación final de rendimiento que se evalúa sobre el conjunto test, como veremos en la sección **??**).

```
dotplot(pima.resamps.3cv10.tl7.rsAll.Acc,
        scales =list(x = list(relation = "free"))) # metric="")

dotplot(pima.resamps.3cv10.tl7.rsAll.Acc,
        scales =list(x = list(relation = "free")),
        between = list(x = 2))
```

Lo que nos proporciona el diagrama de la figura 43. En dicha figura se determinaría que todos los modelos, a excepción de NNET, son más o menos iguales. El resultado esperado de los modelos está entre el 74% y el 78% de acierto. No obstante esta estimación es un poco optimista puesto que, aunque está realizada sobre pliegues de validación que no eran vistos al entrenar, suelen dar resultados ligeramente mejores que las estimaciones sobre Test como luego veremos.

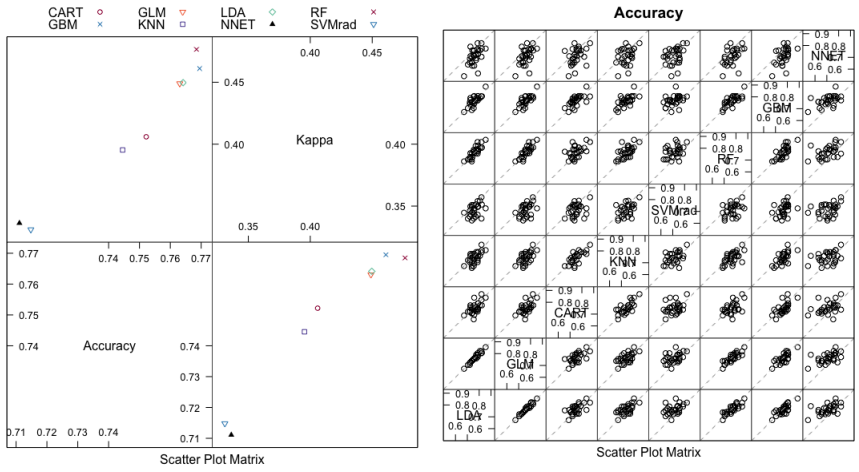


Figure 41: Diagramas `splom()` de los remuestreos de los modelos comparados.

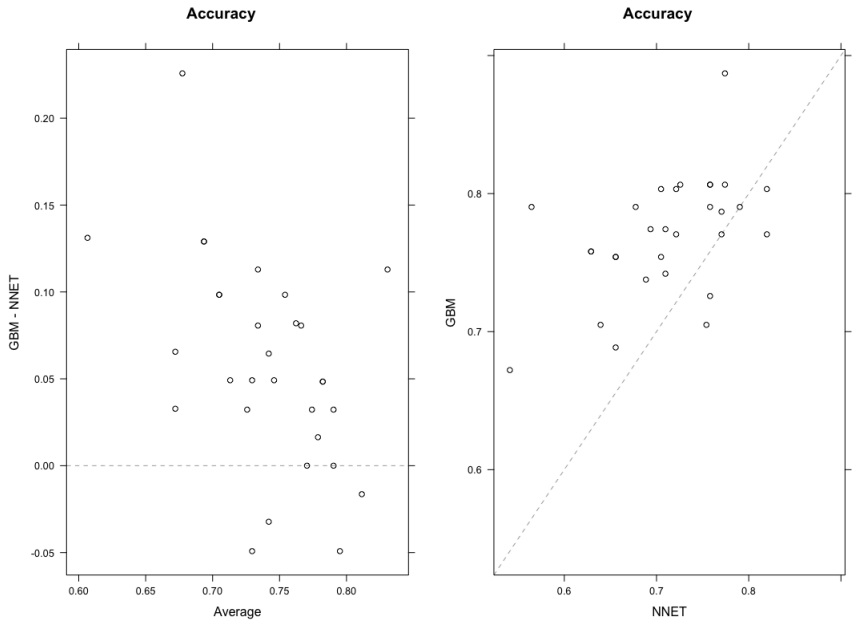


Figure 42: Diagramas `xyplot()` de los remuestreos de los modelos comparados.

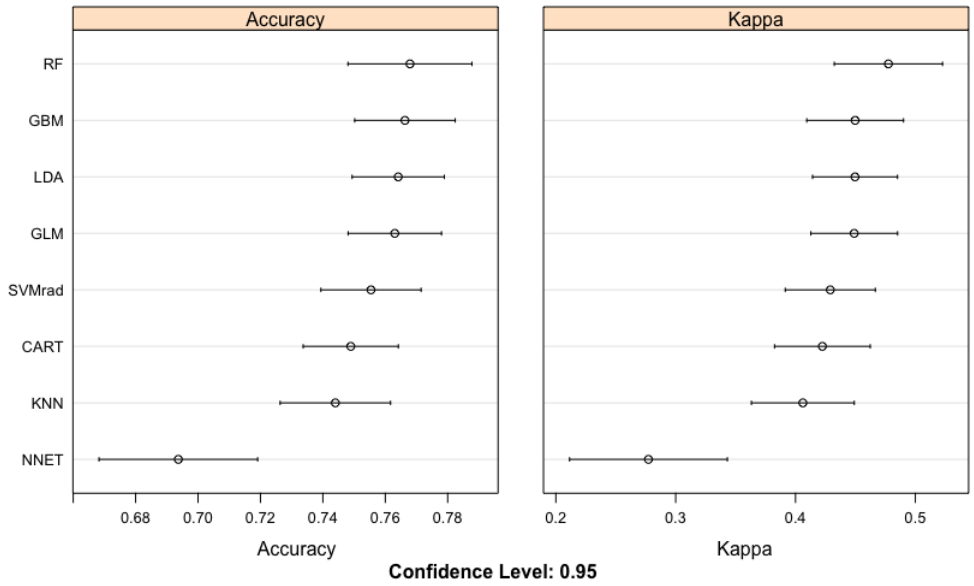


Figure 43: Dotplot que compara los modelos finales con las estimaciones obtenidas de los remuestreos de validación con intervalos de confianza al 95%

Mas información sobre estos gráficos y diversos parámetros se puede encontrar ejecutando el comando `help("xyplot.resamples")` y consultando su documentación.

Para tener una confirmación estadística (aunque ya nos la ha mostrado el comando `dotplot()` sobre resamples) de que existen diferencias entre el rendimiento de los diferentes modelos podemos usar el comando `diff()` sobre el objeto devuelto por `resamples()`. De esta manera se aplica un t-test para evaluar la hipótesis nula (que los modelos tienen el mismo rendimiento desde el punto de vista estadístico). El objeto devuelto por `diff()` también se puede dibujar con `lattice()` y se le puede applicar `summary()` para tener un resumen descriptivo de los resultados. Los diagramas que se pueden usar sobre un `diff` de `resamples` son: `densityplot()` (density plot), `dotplot()` (Cleveland dot plot) y `levelplot()` (diagrama de niveles de falso color).

```
pima.diffs.3cv10.tl7.Acc<-diff(pima.resamps.3cv10.tl7.rsAll.Acc)
summary(pima.diffs.3cv10.tl7.Acc)
```

```
> summary(pima.diffs.3cv10.tl7.Acc)

Call:
summary.diff.resamples(object = pima.diffs.3cv10.tl7.Acc)

p-value adjustment: bonferroni
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0
```

```
Accuracy
       LDA       GLM       CART      KNN       SVMrad    RF        GBM       NNET
LDA                0.001093  0.015204  0.020148  0.008726 -0.003763 -0.002159  0.070457
GLM    1.0000000            0.014111  0.019055  0.007633 -0.004856 -0.003252  0.069364
CART   0.5116019 0.6702915            0.004944 -0.006478 -0.018967 -0.017363  0.055253
KNN    0.3556636 0.5833156 1.0000000           -0.011423 -0.023912 -0.022307  0.050308
SVMrad 1.0000000 1.0000000 1.0000000 1.0000000           -0.012489 -0.010885  0.061731
RF     1.0000000 1.0000000 0.7384340 0.4279883 1.0000000            0.001604  0.074220
GBM    1.0000000 1.0000000 0.3008036 0.4257631 1.0000000 1.0000000            0.072616
NNET   1.208e-05 9.328e-06 0.0041117 0.0085630 0.0004021 1.360e-05 0.0002125

Kappa
       LDA       GLM        CART       KNN        SVMrad     RF         GBM        NNET
LDA                7.041e-04  2.724e-02  4.355e-02  2.064e-02 -2.785e-02 -9.911e-05  1.724e-01
GLM    1.0000000             2.654e-02  4.285e-02  1.994e-02 -2.855e-02 -8.032e-04  1.717e-01
CART   1.0000000 1.0000000              1.631e-02 -6.603e-03 -5.509e-02 -2.734e-02  1.451e-01
KNN    0.5748242 0.7711657  1.0000000             -2.291e-02 -7.140e-02 -4.365e-02  1.288e-01
SVMrad 1.0000000 1.0000000  1.0000000  1.0000000             -4.849e-02 -2.074e-02  1.518e-01
RF     0.6844463 0.7360645  0.2998463  0.0755422  0.1149652              2.775e-02  2.002e-01
GBM    1.0000000 1.0000000  1.0000000  1.0000000  1.0000000  1.0000000             1.725e-01
NNET   3.154e-05 1.988e-05  0.0050826  0.0111140  0.0005322  3.101e-06  0.0008367
```

Como tenemos muchos modelos a comparar los diagramas sobre diff están algo recargados (se muestran todas las combinaciones dos a dos) con lo que para mostrar ejemplos usaremos una lista de remuestreos reducida a 4 modelos. Los gráficos que podemos utilizar para evaluar se consiguen con los siguientes comandos:

```
pima.modelList.3cv10.tl7.rsAll.Acc.just4<-list(
  KNN=pima.modelo.3cv10.tl7.rsAll.knn.Acc
  ,SVMrad=pima.modelo.3cv10.tl7.rsAll.svm.Acc
  ,GBM=pima.modelo.3cv10.tl7.rsAll.gbm.Acc
  ,NNET=pima.modelo.3cv10.tl7.rsAll.nnet.Acc
)

pima.resamps.3cv10.tl7.rsAll.Acc.just4<-resamples(pima.modelList.3cv10.tl7.rsAll.Acc.just4)
pima.diffs.3cv10.tl7.rsAll.Acc.just4<-diff(pima.resamps.3cv10.tl7.rsAll.Acc.just4)

bwplot(pima.diffs.3cv10.tl7.rsAll.Acc.just4, metric = "Accuracy")

dotplot(pima.diffs.3cv10.tl7.rsAll.Acc.just4, metric = "Accuracy")

densityplot(pima.diffs.3cv10.tl7.rsAll.Acc.just4,
            scales =list(x = list(relation = "free"),
                         y = list(relation = "free")),
            auto.key = list(columns = 4),
            pch = "|")
levelplot(pima.diffs.3cv10.tl7.rsAll.Acc.just4,
```

```
          auto.key = list(columns = 4),
          what="differences")

levelplot(pima.diffs.3cv10.tl7.rsAll.Acc.just4,
          auto.key = list(columns = 4)
)
```

Por supuesto, puedes ver también los gráficos con todos los modelos, aunque ya os digo que con muchos a la vez es difícil distinguirlos.

Recuerda que para decidir que un modelo es mejor que otro deben primero ser diferentes desde el punto de vista estadístico (con un alto nivel de confianza, a ser posible superior al 95%) y solo entonces quedarse con el que tiene mejor resultado. Usar simplemente la media de validación de los resultados de los remuestreos internos de cada modelo no es la mejor forma de decidirlo. Si no hubiesen diferencias significativas lo mejor sería quedarse con el modelo más simple siguiendo el principio de parsimonia.

15.1 Entrenar un modelo sin ajustar hiper-parámetros (usar unos fijos)

Si conocemos los hiper-parámetros que queremos usar podemos hacer un ajuste del modelo final indicando como method de trainControl el valor "none".

```
pima.trainCtrl.none.clssProb <- trainControl(method = "none", classProbs = TRUE)

set.seed(123)
pima.modelo.none.gbm.ROC<-train(pima.Datos.Train[pima.Vars.Entrada.Usadas],
                 pima.Datos.Train[[pima.Var.Salida.Usada]],
                 method = "gbm",
                 trControl=pima.trainCtrl.none.clssProb,
                 verbose = FALSE,
                 ## Si no se usa remuestreo solo se le puede
                 ## pasar a la función un único modelo
                 tuneGrid = data.frame(interaction.depth = 4,
                                       n.trees = 100,
                                       shrinkage = .1,
                                       n.minobsinnode = 20),
                 metric = "ROC")
```

16 Grabando y recuperando los modelos entrenados del disco

Hay tres formas de grabar modelos (en realidad de grabar objetos R) que podemos utilizar para grabar y recuperar los modelos y/o cualquier elemento relacionado con el trabajo de creación de modelos que hemos estado llevando a cabo.

El primero es, sencillamente, grabar la sesión de trabajo (y después recuperarla). Es lo que RStudio y R suele hacer cuando cierras la sesión en un fichero que se llama `.RData` y que graba en el directorio de trabajo (puedes comprobar cual es el directorio de trabajo mediante `getwd()`). Para indicar que quieres grabar la sesión de trabajo se usa el comando:

```
# Grabar en disco todos los objetos del espacio de trabajo
save.image(file="nombre_de_fichero.RData")

# Recuperar los objetos guardados con save
# Con verbose=T va indicando los objetos que va cargando
load("nombre_de_fichero.RData", verbose=T)
```

El comando `load()` recupera los objetos grabados en el fichero y les asigna el mismo nombre con el que se grabaron.

También se pueden grabar varios objetos en un fichero mediante el comando `save()`. Se le indica una lista de los nombres de los objetos a grabar (de hecho save.image es un save al que se le pasan los nombres de todos los objetos del espacio de trabajo) y se recuperan después con load. De nuevo al hacer un load se crea el objeto grabado con el nombre que fue grabado.

```
# Graba en disco los modelos
save(file="pimaModelos.RData",list=ls(pattern="pima.modelo"))

# Recuperar los objetos guardados con save
# Con verbose=T va indicando los objetos que va cargando
load("pimaModelos.RData", verbose=T)
```

Existe una tercera manera de grabar objetos en R que evita tener que recordar el nombre con el que se grabó un determinado objeto. Eso sí, obliga a grabar los objetos individualmente. Se hace mediante el comando saveRDS y se recuperan los objetos con el comando readRDS. A diferencia de load, que lo que hace es cargar los objetos y asignarles el nombre con el que se grabaron, y devolviendo la lista de nombres de objetos guardados, , lo que devuelve el comando readRDS es el objeto grabado (que se puede asignar inmediatamente a algún nombre conveniente, que no tiene porqué ser con el que se grabó).

```
# Graba en disco un objeto individual de R
saveRDS("pima.modelo.3cv10.tl15.rsAll.lda.Acc", file="unModeloLda.rds")
# Graba el objeto que guarda un diagrama
saveRDS(d4,file="unDiagrama.rds")

# Recuperar un objeto guardado con saveRDS
lda.modelo.ejemplo <- readRDS("unModeloLda.rds")
# Recupera el diagrama y lo dibuja
readRDS("unDiagrama.rds")
```

Recuerda que, además de los modelos, necesitarás grabar los conjuntos de datos utilizados, las transformaciones de datos, etc. Es por eso interesante que uses nombres de variables que comiencen por un mismo prefijo para facilitar grabar todo lo necesario en bloque con facilidad usando `ls(pattern = patrón)`.

También es interesante que uses scripts de R (ficheros de texto con comandos de R) para, por ejemplo, tener los comandos de carga de todas las librerias que necesitas para trabajar, o las funciones auxiliares que usas, ya que al grabar el workspace o diversos objetos, no se graba información sobre las librerias cargadas (las funciones auxiliares si se graban al ser objetos) y te las volverá a pedir (más probablemente te dará errores) si creas una nueva sesión y cargas la imagen grabada previamente. Para ejecutar un script almacenado en un fichero se usa el comando `source("NombreDelScript.R")`.

17 Prediciendo nuevos valores y evaluación final del modelo

17.1 Prediciendo nuevos valores

Obtener las predicciones del modelo sobre nuevos ejemplos es tan sencillo como usar `predict()` con el modelo entrenado. Es muy habitual predecir varios ejemplos a la vez así que se le pasa en el parámetro `newdata` los valores de las variables de entrada de los nuevos datos en un `data.frame`.

```
# Prediciendo los valores del conjunto Test (nunca vistos)
preds<-predict(pima.modelo.3cv10.tl7.rsAll.gbm.Acc,
               newdata=pima.Datos.Test[pima.Vars.Entrada.Usadas])
# Reordenamos los niveles para que la clase positiva (pos) sea el primero
preds.ord<-factor(preds,levels(preds)[c(2,1)])
predsProbs<-predict(pima.modelo.3cv10.tl7.rsAll.gbm.Acc,
                    newdata=pima.Datos.Test[pima.Vars.Entrada.Usadas],
                    type="prob")
```

Hay muchos modelos de clasifiación que, además de proporcionarte una predicción en forma de una clase predicha, pueden ofrecer las probabilidades de cada clase para cada ejemplo. Es habitual que la clase predicha sea la que tiene más probabilidad, por supuesto, pero hay casos y situaciones en las que se puede querer tener información sobre el grado de seguridad que el modelo ha calculado, o si hay mucha diferencia de dicho grado con respecto a otras posibles clases (si hubiese poca diferencia podríamos preferir que se informase que el resultado es dudoso, por ejemplo), o si queremos una segunda opción en caso de que la primera no estuviese disponible (algo que puede ser muy útil si la clasificación es la toma de una decisión y la de mayor probabilidad no se pudiera llevar a cabo, por ejemplo). Para obtener dicha información sobre las probabilidades de cada clase se usa el parámetro `type = "prob"`. Ya vimos un ejemplo cuando calculamos la métrica ROC (otro ejemplo donde es útil conocer esta información).

Es **muy importante recordar** que los datos de entrada que se le presenten deben haber "sufrido" las mismas transformaciones que tuvo el conjunto de datos de entrenamiento.

En el ejemplo anterior hemos usado `datos.Test` que serían los datos apartados para evaluar realistamente el rendimiento futuro del modelo final. Lo usaremos en la siguente sección.

Como ilustración mostramos los resultados de los anteriores comandos:

```
> preds
  [1] pos neg pos neg neg pos neg neg neg pos neg neg neg neg pos neg pos pos neg neg
 [21] neg pos neg neg pos pos neg pos pos neg neg pos pos neg neg pos neg pos neg pos
 [41] pos pos neg pos neg pos neg neg pos neg neg neg neg neg neg neg neg neg neg neg
 [61] neg pos pos neg neg pos pos neg neg neg neg neg pos neg neg neg pos neg neg neg
 [81] neg pos neg neg neg pos neg neg neg pos neg neg neg neg pos pos neg pos neg neg
[101] pos neg neg neg neg neg neg neg neg neg neg pos neg pos neg pos neg neg pos neg pos
[121] neg pos neg neg neg neg neg neg neg neg neg neg neg neg pos pos pos pos pos pos pos
[141] pos neg neg neg neg neg neg neg neg neg neg neg pos pos
Levels: neg pos
> preds.ord
  [1] pos neg pos neg neg pos neg neg neg pos neg neg neg neg pos neg pos pos neg neg
 [21] neg pos neg neg pos pos neg pos pos neg neg pos pos neg neg pos neg pos neg pos
 [41] pos pos neg pos neg pos neg neg pos neg neg neg neg neg neg neg neg neg neg neg
```

```
 [61] neg pos pos neg neg pos pos neg neg neg neg neg pos neg neg neg pos neg neg neg
 [81] neg pos neg neg neg pos neg neg neg pos neg neg neg neg pos pos neg pos neg neg
[101] pos neg neg neg neg neg neg neg neg neg pos neg pos neg pos neg neg pos neg pos
[121] neg pos neg neg neg neg neg neg neg neg neg neg neg neg pos pos pos pos pos pos pos
[141] pos neg neg neg neg neg neg neg neg neg neg pos pos
Levels: pos neg
> predsProbs
          neg        pos
1   0.33664737 0.66335263
2   0.86949266 0.13050734
3   0.15305586 0.84694414
4   0.95890694 0.04109306
...
149 0.68770862 0.31229138
150 0.87438416 0.12561584
151 0.90370530 0.09629470
152 0.42866536 0.57133464
153 0.12401017 0.87598983
```

17.2 Evaluación final de modelos de clasificación

La estimación del rendimiento que esperaríamos del modelo final seleccionado se debe calcular con el conjunto Test que guardamos en el momento de dividir los datos en Entrenamiento+Validación / Test. Para calcular el rendimiento se pueden usar las diferentes métricas utilizadas en el ajuste.

Así pues se puede usar `postResample()`:

```
> postResample(preds,pima.Datos.Test[[pima.Var.Salida.Usada]])
 Accuracy     Kappa
0.7450980 0.4345684
```

y otras funciones de medición de rendimiento que proporciona tanto caret como `mlMetrics`. Más información sobre medidas de rendimiento de caret pueden verse en: `http://topepo.github.io/caret/measuring-performance.html` o ejecutando `??MLmetrics` y siguiendo los enlaces `MLmetrics::MLmetrics` e `Index`.

```
postResample(preds,pima.Datos.Test[[pima.Var.Salida.Usada]])

# Reordenamos los niveles para que pos(itivo) sea el primero
obs.ord<-pima.Datos.Test[[pima.Var.Salida.Usada]]
obs.ord<-factor(obs.ord,levels(obs.ord)[c(2,1)])
# Datos para twoClassSummary
data2ClassSum<-data.frame(pred=preds.ord,obs=obs.ord,
                          pos=predsProbs["pos"])

dataMnLogLoss<-data.frame(pred=preds.ord,obs=obs.ord,
                          pos=predsProbs["pos"],neg=predsProbs["neg"])
```

```
twoClassSummary(data2ClassSum,lev=levels(obs.ord))

prSummary(data2ClassSum,lev=levels(obs.ord))
mnLogLoss(dataMnLogLoss,lev=levels(obs.ord))

# Calcula datos para lift y calibration curves
lift_results <- data.frame(Class = obs.ord)
lift_results$GBM <- predict(pima.modelo.3cv10.tl7.rsAll.gbm.Acc,
                            pima.Datos.Test[pima.Vars.Entrada.Usadas], type = "prob")[,"pos"]
lift_results$NNET <- predict(pima.modelo.3cv10.tl7.rsAll.nnet.Acc,
                             pima.Datos.Test[pima.Vars.Entrada.Usadas], type = "prob")[,"pos"]
lift_results$KNN <- predict(pima.modelo.3cv10.tl7.rsAll.knn.Acc,
                            pima.Datos.Test[pima.Vars.Entrada.Usadas], type = "prob")[,"pos"]

trellis.par.set(caretTheme())

# Calcula lift curves
lift_obj <- lift(Class ~ GBM + NNET + KNN, data = lift_results)
plot(lift_obj, values = 60, auto.key = list(columns = 3,
                                            lines = TRUE,
                                            points = FALSE))

# Calcula calibration curves
cal_obj <- calibration(Class ~ GBM + NNET + KNN, data = lift_results,
                       cuts = 13)
plot(cal_obj, type = "l", auto.key = list(columns = 3,
                                          lines = TRUE,
                                          points = FALSE))
```

17.2.1 Matrices de confusión

Caret proporciona para los problemas de clasificación una matriz de confusión muy completa

```
# Usamos el parámetro positive="pos" porque sino usa como positive el primer
# nivel que encuentra (que en este caso es "neg")
caret::confusionMatrix(preds,pima.Datos.Test[[pima.Var.Salida.Usada]],
                       positive="pos")
```

```
> caret::confusionMatrix(preds,pima.Datos.Test[[pima.Var.Salida.Usada]],
+                        positive="pos")
Confusion Matrix and Statistics

          Reference
Prediction neg pos
       neg  82  20
```

```
       pos  18  33

               Accuracy : 0.7516
                 95% CI : (0.6754, 0.8179)
    No Information Rate : 0.6536
    P-Value [Acc > NIR] : 0.005891

                  Kappa : 0.4466
 Mcnemar's Test P-Value : 0.871131

            Sensitivity : 0.6226
            Specificity : 0.8200
         Pos Pred Value : 0.6471
         Neg Pred Value : 0.8039
             Prevalence : 0.3464
         Detection Rate : 0.2157
   Detection Prevalence : 0.3333
      Balanced Accuracy : 0.7213

       'Positive' Class : pos
```

La información que da esta tabla es muy importante. Primero te da la tasa de `Accuracy` (exactitud) sobre, en este caso, el conjunto de Test. Después aparece un intervalo de confianza sobre el `Accuraccy`. Este intervalo se calcula sobre la tasa (usando un test binomial). A continuación , en la línea "`No information Rate`" nos da la tasa obtenida de un test unilateral frente a la tasa de accuracy sin información. Esta tasa (no information rate) representa lo que daría un clasificador que funcionase sin información, es decir, aquel que clasifica siempre como la clase más probable (y cuya Accuracy es el porcentaje de la clase mayoritaria). Si el límite inferior del intervalo de confianza del modelo no es capaz de superar dicha tasa implica que el modelo es incapaz de mejorar el clasificador sin información (que representa lo peor que podemos hacer) y mejor nos valdría no usarlo.

`confusionMatrix()` también calcula otros valores habituales para evaluar la clasificación (los detalles se pueden ver tanto con `help(caret::confusionMatrix)` como en `http://topepo.github.io/caret/measuring-performance.html`).

La matriz de confusión por defecto trabaja en términos de sensitividad y especificidad, pero también se puede indicar que se quiere trabajar en términos de precisión y recuerdo (útil en problemas donde se trabaja con extracción de información). Para ello se usaría el parámetro `mode = "prec_recall"` con lo que se obtendría:

```
# Usamos el parámetro positive="pos" porque sino usa como positive el primer
# nivel que encuentra (que en este caso es "neg")
caret::confusionMatrix(preds,pima.Datos.Test[[pima.Var.Salida.Usada]],
                       positive="pos", mode="prec_recall")
```

```
> caret::confusionMatrix(preds,pima.Datos.Test[[pima.Var.Salida.Usada]],
+                        positive="pos", mode="prec_recall")
```

```
Confusion Matrix and Statistics

          Reference
Prediction neg pos
       neg  81  20
       pos  19  33

               Accuracy : 0.7451
                 95% CI : (0.6684, 0.812)
    No Information Rate : 0.6536
    P-Value [Acc > NIR] : 0.009661

                  Kappa : 0.4346
 Mcnemar's Test P-Value : 1.000000

              Precision : 0.6346
                 Recall : 0.6226
                     F1 : 0.6286
             Prevalence : 0.3464
         Detection Rate : 0.2157
   Detection Prevalence : 0.3399
      Balanced Accuracy : 0.7163

       'Positive' Class : pos
```

También se puede trabajar con `confusionMatrix()` sobre los modelos entrenados. Hay un comando llamado `confusionMatrix.train()` que nos permite sacar estadísticas sobre los valores de los pliegues de validación de un modelo entrenado (usa el modelo final). También trabaja sobre los resultados del método de selección de variables rfe visto en sección 14.1, y se obtendría la matriz asociada con el número optimo de variables.

Esta `confusionMatrix.train()` tiene varios modos que se pueden cambiar con el parámetro `norm`. Por defecto es "overall" que muestra los porcentajes en cada celda de la matriz, también se puede usar "average" que nos muestra el número de elementos medio en dichas celdas.

```
> confusionMatrix.train(pima.modelo.3cv10.tl7.rsAll.gbm.Acc)
Cross-Validated (10 fold, repeated 3 times) Confusion Matrix

(entries are percentual average cell counts across resamples)

          Reference
Prediction  neg  pos
       neg 58.2 16.5
       pos  6.9 18.5

 Accuracy (average) : 0.7664
```

```
> confusionMatrix.train(pima.modelo.3cv10.tl7.rsAll.gbm.Acc,norm="average")
Cross-Validated (10 fold, repeated 3 times) Confusion Matrix

(entries are average cell counts across resamples)

          Reference
Prediction  neg  pos
       neg 35.8 10.1
       pos  4.2 11.4

 Accuracy (average) : 0.7664
```

17.2.2 Calculando los valores sobre test de todos los modelos

Veamos por último un código para obtener el `Accuracy` sobre el conjunto de test de toda una lista de varios modelos, para facilitar su visualización comparativa. El código también permite calcular el intervalo de confianza de cada uno de esos `Accuracy` de manera análoga a `confusionMatrix()` (con un test binomial).

```
 getAccModelList<-function (modelList,data,varEnt,varSal,...){
   test_acc <- function(model, data,varEnt,varSal,propMaxClass=0.5,...) {
     pred<-predict(model, data[varEnt])
     correctos<-sum(pred==data[[varSal]])
     # Calcula el No Information Rate (proporción clase más frecuente)
     # Intervalo de confianza al 95% con test binomial
     tr<-binom.test(correctos,nrow(data))
     ci<-as.vector(tr$conf.int)
     # Test si el Acc es mayor que No Information Rate
     tnir<-binom.test(correctos,nrow(data), p=propMaxClass,alternative ="greater")
     c(ci[1],correctos/nrow(data),ci[2],tnir$p.value,tnir$null.value)
   }
   mxClass<-max(table(data[[varSal]]))
   totData<-length(data[[varSal]])
   propMaxClass<-mxClass/totData
   out <- lapply(modelList, test_acc, data = data,
                 varEnt=varEnt,
                 varSal=varSal,
                 propMaxClass=propMaxClass,
                 ...
   )
   t1<-binom.test(mxClass,totData)
   t2<-binom.test(mxClass,totData,propMaxClass,alternative="greater")
 #  nir<-c(t1$conf.int[1],propMaxClass,t1$conf.int[2],t2$p.value,propMaxClass)
   nir<-c(propMaxClass,propMaxClass,propMaxClass,NA,propMaxClass)
   out<-c(list(NIR=nir),out)
```

```
  out <- do.call("rbind", out)
  colnames(out) <- c("lowerCI95", "Accuracy", "upperCI95","pvalAcc>NIR","NIR")
  as.data.frame(out)
}
pima.results.test<-getAccModelList(pima.modelList.3cv10.tl7.rsAll.Acc,pima.Datos.Test,
                                   pima.Vars.Entrada.Usadas,pima.Var.Salida.Usada)
pima.results.test
format(pima.results.test,digits=2,nsmall=3,scientific=F)
```

lo que nos da como resultado:

```
> format(pima.results.test,digits=2,nsmall=3,scientific=F)
       lowerCI95 Accuracy upperCI95 pvalAcc>NIR   NIR
NIR        0.654    0.654     0.654          NA 0.654
LDA        0.725    0.797     0.858    0.000072 0.654
GLM        0.732    0.804     0.864    0.000033 0.654
CART       0.668    0.745     0.812    0.009661 0.654
KNN        0.661    0.739     0.806    0.015356 0.654
SVMrad     0.696    0.771     0.835    0.001098 0.654
RF         0.675    0.752     0.818    0.005891 0.654
GBM        0.675    0.752     0.818    0.005891 0.654
NNET       0.546    0.627     0.704    0.778775 0.654
```

Si lo queremos ver en un diagrama podemos usar el siguiente código para crear una función que nos muestre el intervalo de confianza calculado con la binomial sobre el "No information Rate" (NIR). También hace una línea horizontal sobre el NIR.

```
require(plotrix)
plotTestRes<-function(x,...) {
  plotCI(1:nrow(x),y=x$Accuracy,
            uiw=x$upperCI95-x$Accuracy,
            ylab="Accuracy",xlab="Methods",xaxt="n",...)
  axis(1,at=1:nrow(x),labels=row.names(x))
  grid (NULL,NULL, lty = 6, col = "cornsilk2")
  NIR<-x["NIR",2]
  if(!is.na(NIR))
    abline(h=NIR,lty=3)
}

plotTestRes(pima.results.test,pch=5)
```

Recordemos que la clase mayoritaria tiene un 65.35%. El modelo NNET es, en este caso, bastante pobre (podrían empeorar el clasificador sin información). Que NNET empeore incluso el NIR hace sospechar que ha habido algún problema al construir el modelo, probablemente esté mal la configuración o haya habido algún problema.

Es interesante mencionar que estos modelos de ejemplo los hemos entrenado sobre los datos en bruto sin transformar ni tratar, y además es muy probable que algunos modelos puedan mejorarse retocando bastante

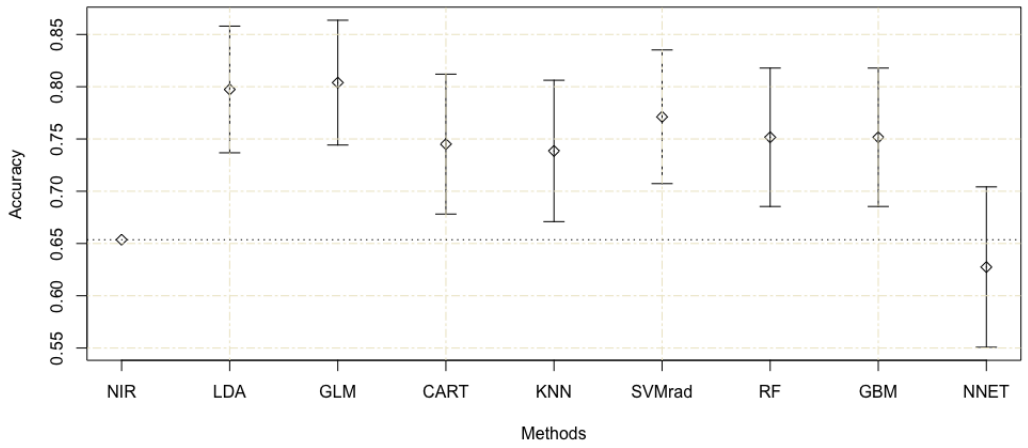


Figure 44: Diagrama que compara los modelos finales con el resultado obtenido sobre el conjunto de test y con intervalos de confianza al 95% calculado con la binomial sobre el "No Information Rate" (NIR).

los hiper-parámetros comprobados (para estos ejemplos ilustrativos hemos hecho un `tuneLength` de 7, y no una búsqueda en parrilla). NNET es bastante quisquilloso con los hiper-parámetros pero una vez encontrados suele tener siempre resultados competitivos.

Por último decir que, si la base de datos es conocida, siempre es útil acudir a los artículos de investigación de la bibliografía para ver los resultados publicados y ver si nos acercamos a ellos (lo que indicaría que lo estamos haciendo bien) y no nos quedamos muy cortos (lo que indicaría que algo no estamos haciendo bien) o que lo mejoramos bastante (¡que también sería sospechoso de estar haciendo algo mal!).

Con esto acabamos la quinta sesión. El resto del tutorial incluye la evaluación en modelos de regresión, alguna información adicional, referencias y soluciones a los ejercicios

---

17.3 Evaluación final de modelos de regresión

Al igual que hemos visto en la sección anterior se pueden obtener diversas medidas de rendimiento para problemas de regresión. Para ello deberíamos primero crear algunos modelos de algún problema de regresión. Como ejemplo usaremos el problema `algae`:

```
install.packages("DMwR2")
data(algae,package="DMwR2")

# Ponemos la semilla para números aleatorios
set.seed(1234)
algae.Datos.Todo<-na.omit(algae)
algae.Vars.Salida<-c("a1","a2","a3","a4","a5","a6","a7")
algae.Vars.Entrada<-setdiff(names(algae.Datos.Todo),algae.Vars.Salida)
algae.Vars.Entrada.Usadas<-algae.Vars.Entrada
algae.Var.Salida.Usada<-c("a1")
# Índices para el cjto de entrenamiento (70% dataset "algae",variable de salida "a1")
trainIdx<- createDataPartition(algae.Datos.Todo[[algae.Var.Salida.Usada]],
                               p=0.7,
                               list = FALSE,
                               times = 1)
# Creamos los subconjuntos en base a trainIdx
algae.Datos.Train<-algae.Datos.Todo[trainIdx,]
algae.Datos.Test<-algae.Datos.Todo[-trainIdx,]

# Hacemos un modelo que guarda TODOS los remuestreos del mejor
# Usamos el dataset algae
algae.trainCtrl.3cv10.resampAll <- trainControl(
  method = "repeatedcv"
  ,number = 10 ,repeats = 3, ## Crosvalidación de 10 pliegues, 3 repeticiones
  # Que muestre información mientras entrena
  verboseIter=T,
  returnResamp = "all"  # Guardamos todo para hacer diagramas
)
set.seed(1234)
algae.modelo.3cv10.tl7.rsAll.rf<-train(algae.Datos.Train[algae.Vars.Entrada.Usadas],
                                       algae.Datos.Train[[algae.Var.Salida.Usada]],
                                       method="rf", trControl=algae.trainCtrl.3cv10.resampAll
                                       ,tuneLength = 7
                                       ,verbose=F
)

#[J] Le añadimos a postResample R^2 ajustado, MAPE, ME, y MPE
#[J] Para R^2 ajustados se necesitan los grados de libertad usados, o bien en la variable
#[J] numVarEnt, o bien se le pasan (el número de variables de entrada)
```

```
##[J] PostResample extendido
postResampleRegExt<-function(pred,obs,deg=numVarEnt) {
  out<-postResample(pred,obs)
  if("Rsquared" %in% names(out)) {
    n<-length(pred)
    oneminusr2<- 1-out["Rsquared"]
    out["R2adj"]<- 1-((oneminusr2*(n-1))/(n-deg-1))
    #    out["MAE"]<-mean(abs(obs-pred))
    out["MAPE"]<-mean(abs((obs-pred)/obs)*100)
    out["ME"]<-mean(obs-pred)
    out["MPE"]<-mean((obs-pred)/obs)*100
  }
  out
}


d1<-densityplot(algae.modelo.3cv10.tl7.rsAll.rf, pch = "|")
d2<-histogram(algae.modelo.3cv10.tl7.rsAll.rf)

algae.test.preds<-list(RF=predict(algae.modelo.3cv10.tl7.rsAll.rf,newdata=algae.Datos.Test),
          RF2=predict(algae.modelo.3cv10.tl7.rsAll.rf,newdata=algae.Datos.Test))

algae.test.preds<-lapply(algae.test.preds,as.vector)
algae.test.metrics<-do.call("rbind",list(RF=postResample(algae.test.preds$RF, algae.DatosTest$a1),
                                         RF2=postResample(algae.test.preds$RF2, algae.DatosTest$a1)
)
algae.test.metricsExt<-do.call("rbind",list(RF=postResampleRegExt(algae.test.preds$RF, algae.Datos.Tes
                                         RF2=postResampleRegExt(algae.test.preds$RF2, algae.Datos.Te
)


# algae.test.tt<-do.call("rbind",lapply(algae.test.preds,FUN=t.test,
#                                       algae.Datos.Test$a1))
# algae.test.ct<-do.call("rbind",lapply(algae.test.preds,FUN=cor.test,
#                                       algae.Datos.Test$a1))
# do.call("rbind",lapply(algae.test.preds,FUN=t.test,
#                        algae.Datos.Test$a1))

algae.test.residuals<-lapply(algae.test.preds,FUN=function (pred,obs) obs-pred,
                         algae.Datos.Test$a1)
```

17.3.1 Visualizando los resultados de un modelo de regresión

En los problemas de regresión el concepto principal sobre el que se trabaja para analizar los resultados es el concepto de residuo. El residuo es la diferencia entre el valor real de la variable de salida y el valor predicho sobre el modelo. Las medidas de rendimiento de regresión están relacionadas siempre con los residuos.

Lo primero que haremos será calcular las predicciones y los residuos

17.3.2 Calculando los valores sobre el conjunto de test

Más sobre medidas de rendimiento

En las dos siguientes secciones trataré de ofrecer una guía rápida sobre el significado y el sentido de los valores de medida de rendimiento más clásicos sobre clasificadores y regresión. Es muy probable que se repitan cosas de secciones anteriores (en particular las secciones 11.2.3 y 17.2) aunque en siguientes revisiones se podría todo reordenar en una sola sección.

Estas secciones no forman parte del tutorial de prácticas y son, más bien, unos breves recordatorios aplicados del tema de evaluación y comparación de Clasificadores y Regresiones.

18 Medidas de rendimiento en clasificadores

18.1 Medidas generales sobre clasificación

18.1.1 Error y Exactitud

Dada una muestra del funcionamiento de un clasificador (es decir, dadas las predicciones del modelo sobre un conjunto de datos con salida conocida) la medida más natural e intuitiva para evaluar su rendimiento es comprobar el número de aciertos frente al tamaño de la muestra, lo que se conoce como "Accuracy" o Exactitud, (no confundir con precisión que es otra medida), o bien su inversa, el contar el número de errores cometidos frente al total de la muestra ("Error rate" o tasa de error).

Hemos mencionado en clase que no se puede tomar el valor obtenido de una muestra como el valor verdadero de error o exactitud de un modelo, puesto que es solo una estimación que depende, entre otras cosas, del tamaño de dicha muestra.

Cuando evaluamos el valor de una magnitud sobre la que hemos hecho varias medidas (como es el caso de cuando hemos hecho entrenamientos con los mismos hiper-parámetros y diferente semilla aleatoria y conjuntos entrenamiento/validación) podemos tener la tentación de obtener la media de dichas medidas y considerar que ese es el valor de la magnitud medida, pero en realidad es solo una estimación del valor real de la media (que será tanto más pobre cuantas menos muestras tengamos). Lo que debemos hacer en ese caso es trabajar con intervalos de confianza e indicar un rango de valores sobre los cuales tenemos cierto grado de confianza (habitualmente el 95%) que debería estar el verdadero valor medio. En clase hemos visto como se calculan.

Cuando estamos evaluando sobre un único conjunto de datos (p.e. sobre el conjunto de Test) es habitual también tratar de establecer otro intervalo de confianza que nos indique por donde andaría el valor real de dichas tasas, pero esta vez solo tenemos una medida. En estos casos podemos hacer un test binomial que trabaja sobre las probabilidades de cada ejemplo del conjunto de test (y no sobre varias medidas de todo el conjunto) y que también consigue un intervalo de confianza.

La medida "Accuracy" puede dar problemas con clases desbalanceadas (tiende a ignorar clases con pocos ejemplos) por lo que en esos casos es aconsejable usar el coeficiente Kappa

18.1.2 Matriz de confusión

Caret dispone del comando `confusionMatrix()` que proporciona diversas medidas sobre los problemas de clasificación (si no aparece la información que a continuación se describe es posible que se esté ejecutando una `confusionMatrix()` de otra librería, prueba entonces con `caret::confusionMatrix()`). Primero te da la tasa de accuracy sobre, en este caso, el conjunto de test. Después aparece un intervalo de confianza sobre el Accuraccy. Este intervalo se calcula sobre la tasa (usando un test binomial) y un test unilateral frente a la tasa de accuracy sin información. Esta tasa (no information rate) representa un clasificador sin información, que es aquel que clasifica siempre como la clase más probable, y cuyo valor es el porcentaje de dicha clase mayoritaria). Si el límite inferior del intervalo de confianza no es capaz de superar dicha tasa implica que el modelo es incapaz de mejorar el clasificador sin información (representa lo peor que podemos hacer).

18.1.3 Kappa

La idea que subyace sobre la medida Kappa es tener en cuenta que, cuando vemos una tasa de acierto (Observed Accuracy), algunos de dichos aciertos es posible que sean por las cualidades del modelo que hemos creado, pero otros aciertos serían atribuíbles a la suerte. Como con frecuencia lo que buscamos es evaluar cuanto aporta el modelo en sí, y no a lo que se podría acertar por pura suerte, sería conveniente tener una medida que obviase los efectos del azar y se concentrase solo en lo que el modelo proporciona. En la medida Kappa se intenta estimar cuantos de dichos aciertos se hacen por suerte y los quita de la medida de lo bueno que es un clasificador.

El valor de Kappa es:

$$\kappa = \frac{O - E}{1 - E} \tag{2}$$

con O la exactitud observada (observed accuracy) y E la exactitud esperada de acuerdo al azar (expected accuracy o atribuible al azar, lo que se puede acertar prediciendo la clases al azar). E se calcula según la frecuencia relativa de las clases predichas y las observadas (las verdaderas). Si hay una clase muy frecuente, y el modelo, en general, predice esa clase también con mucha frecuencia, entonces hay mucha probabilidad de que el acierto venga por suerte y E será "alto". Se calcularía como:

$$O = \frac{tp + tn}{n} \tag{3}$$

$$E = (P_{pred}(+) * P_{obs}(+)) + (P_{pred}(-) * P_{obs}(-)) \tag{4}$$

En el denominador se descuenta la parte atribuible al azar, lo que hace que, si todo lo que queda no atribuible al azar, por poco que sea, es muy similar a lo que queda de lo observado al quitarle el azar, el valor de kappa se acercaría igualmente a 1 (mejora lo "poco" o "mucho" que puede mejorarse, luego es un modelo útil).

Kappa puede tener valores negativos (el modelo es peor que el azar) que obviamente revelarían el modelo como desastroso.

En cuanto a como interpretar la magnitud de Kappa para decidir si el modelo tiene cierta calidad... "depende", como siempre, del problema y la situación. En muchos casos lo usaremos para comparar entre modelos, y junto a otras medidas de rendimiento . La interpretación de la calidad de un valor particular

no es, por tanto, trivial. No existe acuerdo en como categorizar los valores de kappa para interpretar como de "bueno" es el resultado dado un valor particular. Diversos autores han dado interpretaciones lingüísticas dependiendo de varios rangos, p.e.:

Autor Landis&Koch	Interpretación Lingüística	Autor Fleiss	Interpretación Lingüística
0 - 0.2	Slight	0 - 0.4	Poor
0.2 - 0.4	Fair		
0.4 - 0.6	Moderate	0.4 - 0.75	Fair to Good
0.6 - 0.8	Substantial		
0.8 - 1	Almost Perfect	0.75 - 1	Excellent

pero esto es solo una guía intuitiva puesto que ambas son meramente escalas arbitrarias.

Kappa se interpreta mejor si se observa también la matriz de confusión puesto que revela si hay ciertas descompensaciones entre aciertos en las clases que Kappa no distingue pero que el contexto del problema nos indiquen que no es aceptable. Por ejemplo, si tenemos una matriz de confusión como:

```
> tst<-matrix(c(6000,10,160,80),nrow=2,dimnames=list(c("Pos","Neg"),c("Pos","Neg")))
> tst
     Pos Neg
Pos 6000 160
Neg   10  80
> vcd::Kappa(tst)
            value     ASE     z  Pr(>|z|)
Unweighted 0.4738 0.03396 13.95 3.032e-44
Weighted   0.4738 0.03396 13.95 3.032e-44
```

comprobamos que tiene un Kappa de 0.47, lo que sería moderado o que está bien según dichas escalas pero, si entendemos las columnas como valores observados y las filas como valores predichos, veríamos que más de dos tercios de los verdaderos negativos son clasificados como positivos, lo que podría ser inaceptable en modelos de diagnóstico médico, por ejemplo, y debería tratarse de buscar modelos con mejor especificidad (ver sección 18.2.1) aunque tenga peores kappas.

Por supuesto el propio problema también influye en si las magnitudes de los valores son mejores o peores. Por ejemplo, si el problema es "fácil" en el sentido de que las clases son relativamente fáciles de separar (p.e. iris), valores de Kappa por debajo de 0.7 pueden ser muy malos, mientras que otros problemas muy complejos valores de 0.4 se les podría considerar de muy buenos.

18.1.4 Test de McNemar

18.1.5 T-test y su interpretación

18.2 Medidas específicas de clasificación binaria

18.2.1 Sensibilidad y Especificidad

La sensibilidad es la medida en la que el sistema predice correctamente los casos positivos (verdaderos positivos/total de positivos), y la especificidad la medida en que predice correctamente los casos negativos

(verdaderos negativos/total de negativos). Se podría dar más valor a una u otra medida si no queremos que cuesten igual los falsos positivos o los falsos negativos. Si se valora más la sensibilidad buscaríamos modelos donde no se nos escapen casos positivos (aunque aumentemos los falsos positivos) y si se valora más la especificidad se buscarían modelos donde no se nos escapen casos negativos (aumentando normalmente los falsos negativos). Habría que encontrar un trade-off entre ambos valores (si se hace un 50% tenemos el típico caso en que apreciamos por igual un verdadero positivo como un verdadero negativo). En los casos extremos se llegan a clasificadores que siempre clasifican los datos como positivos (lo que da sensibilidad 1 pues no se nos escapa ningún positivo verdadero, ¡decimos que todos son verdaderos!) o clasifican siempre como negativo (lo que da especificidad 1 pues no se nos escapa ningún verdadero negativo. No trates de maximizar alguna de dichas medidas por separado sino más bien una combinación de ellas (que es lo que hace ROC).

18.2.2 Curvas ROC y su interpretación

La curva ROC (Receiver Operating Characteristics) se puede usar para estimar el rendimiento utilizando una combinación de la sensibilidad y la especificidad (por lo general la una aumenta a costa de la otra). Una curva ROC dibuja el ratio de verdaderos positivos (sensibilidad) frente al ratio de falsos positivos (1 menos la especificidad). Estos dos ratios están positivamente correlados puesto que es habitual que para acertar más positivos se aumente el área del espacio de entrada que se considera de la clase positivos y al aumentarla es habitual que, junto con nuevos casos positivos, ahora bien clasificados, se nos "cuelen" algunos casos que son negativos, lo que aumentaría al mismo tiempo el ratio de falsos positivos. Es, por tanto, una forma de visualizar en que medida el ser más tolerante con falsos positivos mejora la tasa de aciertos de los positivos verdaderos.

Cuando se hace un diagrama donde se dibuja la relación entre estos ratios genera curvas como las mostradas en la figura 29 .El AUC, es decir, el área bajo dicha curva (Area Under Curve) se puede usar para medir el rendimiento de un clasificador de dos clases. Cuanto más se asemeje a una L invertida y más se aleje de una diagonal indica que el modelo es más fino en el trade-off y, p.e., al aumentar el ratio de verdaderos positivos se cuelan menos negativos, que otras formas de la curva. No obstante, tarde o temprano se llega al caso límite de clasificar todo como positivo (lo que genera un ratio de verdaderos positivos de 1, y también de falsos positivos de 1). En la figura 29(b) la curva generada por el modelo A es peor que la B y la C. Los modelos B y C tienen rendimientos más similares (las áreas bajo la curva tienen valores más parecidos) aunque muestran diferentes comportamientos en el trade-off de aumento de verdaderos positivos frente a tolerar más falsos positivos.

18.3 Medidas específicas de clasificación multiclase

19 Medidas de rendimiento en regresión

19.1 RMSE

19.2 R-Squared y Adjusted R-Square

R-Squared es una medida que indica la fracción en la que la varianza de los errores cometidos por el modelo de predicción es menor que la varianza de la variable dependiente (los valores reales observados de la variable de salida). Se llama R-squared porque en un modelo de regresión simple es solamente el cuadrado de la correlación entre la variable dependiente y las independientes (que normalmente se le llama "r"). En

modelos de regresión múltiple R-square se determina por las correlaciones dos a dos entre todas las variables, incluyendo las correlaciones entre variables independientes (variables de entrada) así como con la variable dependiente.

El valor de R-Square tiende a subir con el número de predictores (variables de entrada) y no compensa esta subida en el valor que se debe exclusivamente a que tiene más variables (independientemente de si estas aportan realmente algo al modelo) con lo que si se prueban modelos con diferente número de variables y se quieren comparar es mejor usar adjusted R-Square que compensa ese incremento. Dicho de otro modo, mientras R-Square da el porcentaje de la variación explicada como si todas las variables de entrada al modelo afectasen al valor de la variable de salida, el adjusted R-Square daría el porcentaje de variación explicada solo por aquellas variables que realmenten afectan a la variable dependiente. No obstante eso no es del todo así porque R-Square ajustado podría tener valores negativos (que no tendría sentido).

El código R para incluir la medida adjusted Rsquared (y lo llamaríamos R2adj) en una nueva versión de `postResample()` sería:

```
postResampleR2adj<-function(pred,obs,deg) {
  out<-postResample(pred,obs)
  if("Rsquared" %in% names(out)) {
    n<-length(pred)
    oneminusr2<- 1-out["Rsquared"]
    out["R2adj"]<- 1-((oneminusr2*(n-1))/(n-deg-1))
  }
  out
}
```

donde `deg` es un parámetro que sería igual al número de variables de entrada utilizadas. De nuevo insistir en que es adjusted R square es útil si se comparan modelos con diferente número de variables (y también solo con modelos de regresión múltiple).

La interpretación de los valores de Rsquare seguiría la noción básica de cuanto más alta mejor. No obstante determinar si un valor de Rsquare es suficientemente alto para considerar "bueno" un modelo tiene como respuesta el viejo "depende del problema" que es común a todas las medidas (y muy especialmente las estadísticas). Si sabemos que hay una gran relación entre las variables de entrada y de salida el Rsquare debe tener valores muy elevados. En cambio, si el problema es encontrar algún patrón (señal) en unos datos con mucho ruido, pues valores más bajos de Rsquared se pueden considerar un éxito. Muchas veces la decisión depende del margen que dispongamos para el error o lo que haya en juego a la hora de tomar decisiones. Por ejemplo, imaginemos que un modelo presenta mucha variabilidad en la diferencia entre el valor observado y el predicho (baja Rsquared) pero que es significativamente mejor que el modelo sin información (que en regresión sería devolver el valor medio de la salida siempre), es decir que los p-values son bajos y hemos comprobado con un conjunto de test bastante grande. Entonces sabemos que el modelo aporta algo y, si ese algo es una mejora competitiva frente a un rival que puede traducirse en una importante mejora de beneficios.

Cuando hablamos de reducción de varianza atribuible al modelo, hay que estar atento a no confundir varianza con desviación estándard. La varianza se mide en unidades de la variable de salidad elevadas al cuadrado (euros al cuadrado, nuevos parados al cuadrado) que no es tan sencillo de interpretar que unidades tal cual se miden en la variable de salida, es decir, es más fácil pensar en términos de desviaciones estándard (que se mide en la misma unidad que la salida) y que es la que determina directamente la anchura de los intervalos de confianza. Por eso es útil considerar el "porcentaje de desviación estándard explicada" por el

modelo de manera análoga a Rsquare con varianza, es decir, el porcentaje por el cual la desviación estándard de los errores del modelo es menor que la desviación estándar de la variable dependiente. Se puede calcular como $1 - \sqrt{1 - Rsquared}$.

20 Referencias

Este tutorial está basado en mucho material disponible en la red. La lista es larga (y la iré ampliando). Hubiese sido, no obstante, imposible hacerla sin la maravillosa comunidad de StackOverflow que ya tiene resuelto cualquier problema que surja o imagines. No dudes en buscar allí (San Google ayuda también) si tienes problemas. ¡Larga vida a StackOverflow!

No obstante hay algunas páginas especialmente útiles o de donde he sacado mucho material. Os muestro algunas aunque añadiré más y trataré de organizarlas por temas.

- Doc de Caret:
  - Documentación principal http://topepo.github.io/caret/index.html
  - `https://www.analyticsvidhya.com/blog/2016/12/practical-guide-to-implement-machine-learning-with-caret-package-in-r-with-practice-problem/`
  - `http://rstudio-pubs-static.s3.amazonaws.com/251240_12a8ecea8e144fada41120ddcf52b116.html`
- Procesado de datos
  - `https://machinelearningmastery.com/machine-learning-in-r-step-by-step/`
  - `https://www.analyticsvidhya.com/blog/2016/01/guide-data-exploration/`
  - `https://machinelearningmastery.com/pre-process-your-dataset-in-r/`
- Gráficos
  - `https://www.stat.ubc.ca/~jenny/STAT545A/block09_xyplotLattice.html#type`
- Del adult dataset
  - `https://cloudxlab.com/blog/predicting-income-level-case-study-r/`
  - `http://scg.sdsu.edu/dataset-adult_r/`
  - `http://www.dataminingmasters.com/uploads/studentProjects/Earning_potential_report.pdf`
- Medidas de rendimiento
  - `https://people.duke.edu/~rnau/rsquared.htm`

21 Soluciones a los ejercicios

21.1 Iris density plot con *lattice*

```
densityplot( ~ value|variable, groups=clase, data = melt(iris,id.vars=5),
             scales=list(x=list(relation="free"), y=list(relation="free")),
             xlab="Feature", ylab="")
```

21.2 Soybean

```
soy<- rbind(
  read.table(paste("https://archive.ics.uci.edu/ml/machine-learning-databases/",
      "soybean/soybean-large.data",
      sep=""),
    sep=",",header=F, na.strings ="?"),
  read.table(past("https://archive.ics.uci.edu/ml/machine-learning-databases/",
   "soybean/soybean-large.test",
   sep=""),
 sep=",",header=F, na.strings ="?")
)
soy<- data.frame(lapply(soy,FUN=as.factor))
names(soy)<-c("Class", "date", "plant.stand", "precip", "temp", "hail", "crop.hist",
  "area.dam", "sever", "seed.tmt", "germ", "plant.growth",  "leaves", "leaf.halo",
  "leaf.marg", "leaf.size", "leaf.shread", "leaf.malf", "leaf.mild", "stem", "lodging",
  "stem.cankers", "canker.lesion", "fruiting.bodies", "ext.decay", "mycelium",
  "int.discolor", "sclerotia",  "fruit.pods", "fruit.spots", "seed", "mold.growth",
   "seed.discolor", "seed.size", "shriveling", "roots")
# Los perezosos pueden usar names(soy)<-names(Soybean)
for (i in c("plant.stand","precip","temp","germ","leaf.size"))
  soy[[i]]<-factor(soy[[i]],ordered=T)
```

21.3 "Arreglar" NAs del BreastCancer

```
data(BreastCancer)
# Comprobar que no hay NAs en la variable de salida
sum(is.na(BreastCancer[["Class"]]))
# Comprobar que hay NAs en el dataset
sum(is.na(BreastCancer))

fixNA.inc <- function (data.toFix) {
  # arregla de manera incremental con lo que los arreglos anteriores pueden
  # afectar a los valores de arreglo posteriores. Para evitar ese sesgo se puede
  # almacenar la mediana para numéricos y el más frecuente para factores
  # de cada columna andes de arreglar nada y luego usarlos
  for (x in which(!complete.cases(data.toFix))) {
    for (y in which(is.na(data.toFix[x, ]))) {
      if (is.factor(data.toFix[[y]])) {
```

```
        # Nivel más frecuente si es un factor
        ttx <- table(data.toFix[[y]])
        data.toFix[x, y] <- names(ttx[which.max(ttx)])
      } else
        # Mediana si el vector es numérico
        data.toFix[x, y] <- median(na.omit(data.toFix[[y]]))
    }
  }
  data.toFix
}

fixNA.noinc <- function (data.toFix) {
  valsToUse <- list()
  for (i in 1:length(data.toFix))
    if (is.factor(data.toFix[[i]])) {
      ttx <- table(data.toFix[[i]])
      valsToUse[[i]] <- names(ttx[which.max(ttx)])
    }
  else
    valsToUse[[i]] <- median(na.omit(data.toFix[[i]]))
  for (x in which(!complete.cases(data.toFix)))
    for (y in which(is.na(data.toFix[x, ])))
      data.toFix[x, y] <- valsToUse[[y]]

    data.toFix
}

BreastCancer.fixed<-fixNA.noinc(BreastCancer)
```

21.4 Dummy con conjunto a ignorar

```
# El dataset a ponerle dummy
data.To.Dummy<-adult.fix2
# Hay que indicar qué variables son de salida y entrada
Vars.Salida<-c("income")
Var.Salida.Usada<-c("income")
Vars.Entrada.Usadas<-setdiff(names(data.To.Dummy),Variables.Salida)
# También qué factores serán Ranked (el resto de factores son no Ranked)
ranked.Factors<-c("sex")
# Conjunto de variables a ignorar al hacer dummy
variables.Ignoradas<-c("type_employer")
# Se separan los factores ordenados (no hay que hacer Dummy)
# y los que no se vaya a hacer ranked
```

```
all.Factors<-names(data.To.Dummy[,Vars.Entrada.Usadas])[sapply(
  data.To.Dummy[,Vars.Entrada.Usadas], FUN=is.factor)]
factors.To.Change<-setdiff(all.Factors,variables.Ignoradas)
ordered.Factors<-names(data.To.Dummy[,Vars.Entrada.Usadas])[sapply(
  data.To.Dummy[,Vars.Entrada.Usadas], FUN=is.ordered)]
non.Ordered.Factors<-setdiff(factors.To.Change,ordered.Factors)
no.Ranked<-setdiff(non.Ordered.Factors,ranked.Factors)

# Se calculan las dummy de los ranked y no ranked
cols.Ranked<-NULL
if(length(ranked.Factors)>0) {
  dummy.Ranked<-dummyVars(paste("~",paste(ranked.Factors,sep="",collapse =" + "),
                                collapse=""), data=data.To.Dummy,fullRank=T)
  cols.Ranked<-data.frame(predict(dummy.Ranked,newdata=data.To.Dummy))
}
cols.No.Ranked<-NULL
if(length(no.Ranked)>0) {
  dummy.No.Ranked<-dummyVars(paste("~",paste(no.Ranked,sep="",collapse =" + "),
                                   collapse=""), data=data.To.Dummy)
  cols.No.Ranked<-data.frame(predict(dummy.No.Ranked,newdata=data.To.Dummy))
}

# Los factores ordenados se transforman en numéricos
cols.Ordered.Factors<-NULL
if(length(ordered.Factors)>0)
  cols.Ordered.Factors<-data.frame(lapply(data.To.Dummy[,ordered.Factors],FUN=as.numeric))


cols.Salida<-data.frame(data.To.Dummy[,Var.Salida.Usada])
names(cols.Salida)<-Var.Salida.Usada

#Se eliminan de los datos originales todas las columnas a reemplazar
data.With.Dummy<-data.To.Dummy[,setdiff(names(data.To.Dummy),Vars.Salida)]
data.With.Dummy<-data.With.Dummy[,setdiff(names(data.With.Dummy),no.Ranked)]
data.With.Dummy<-data.With.Dummy[,setdiff(names(data.With.Dummy),ranked.Factors)]
data.With.Dummy<-data.With.Dummy[,setdiff(names(data.With.Dummy),ordered.Factors)]
# Se añaden todas las columnas nuevas.
if(!is.null(cols.Ranked))
  data.With.Dummy<-cbind(data.With.Dummy,cols.Ranked)
if(!is.null(cols.No.Ranked))
  data.With.Dummy<-cbind(data.With.Dummy,cols.No.Ranked)
if(!is.null(cols.Ordered.Factors))
  data.With.Dummy<-cbind(data.With.Dummy,cols.Ordered.Factors)
data.With.Dummy<-cbind(data.With.Dummy,cols.Salida)
```

Categorizar

```
# Ej Categorizar
# Función para dibujar un barplot con el % de cada clase
bptab<-function(x,lab="Variable") barplot(prop.table(table(x))*100,xlab=lab)
# ej1 adult[["age"]] en 5 conjuntos de similar tamaño y sobrelapamiento de
# aproximadamente 5%.
ej1<-equal.count(adult$age,number=6,overlap=0.05)
interval.count.ej1<-colSums(sapply(levels(ej1),
                                   function(x) ifelse(ej1>=x[1] & ej1<=x[2],1,0)))
names(interval.count.ej1)<-sapply(levels(ej1),
            function(x) paste("(",paste(x,collapse=","),"]",sep="",collapse=""))
barplot(interval.count.ej1*100/length(adult$age),xlab="Age de Adult Dataset")
# ej2 iris[["Sepal.Length"]] en 5 conjuntos en los percentiles 20-40-60-80.
cortes.ej2<-c(-Inf,quantile(iris[["Sepal.Length"]],c(0.2,0.4,0.6,0.8)),Inf)
ej2<-ordered(cut(iris[["Sepal.Length"]],breaks=cortes.ej2))
bptab(ej2,"Sepal Length de Iris Dataset")
# ej3 algae[["a2"] en 3 conjuntos: los valores igual a 0, de 0 a mediana de valores
# diferentes de 0,y  de mediana al máximo.
cortes.ej3<-c(-Inf, 0,median(algae[["a2"]][algae[["a2"]] >0],na.rm=T),Inf)
ej3<- ordered(cut(algae[["a2"]], breaks=cortes.ej3), labels = c("None", "Low", "High"))
bptab(ej3,"Concentración alga 2 de Algae Dataset")
# ej4 adult[["education_num"]] en 5 conjuntos más o menos iguales, "a ojo".
cortes.ej4<-c(-Inf,8.5,9.5,12.5,Inf)
ej4<-ordered(cut(adult$education_num,breaks=cortes.ej4))
bptab(ej4,"Education Num de Adult Dataset")
```