Máster en Ciencia de datos e Ingeniería de computadores.

Minería de datos: preprocesamiento y clasificación.

Sesiones de prácticas 4, 5 y 6: preprocesamiento y clasificación

8 de enero de 2019

Índice

Parte I
Preprocesamiento

1. Introducción

En estas sesiones de prácticas nos centraremos en considerar diferentes aspectos prácticos relacionados con los contenidos teóricos estudiados sobre preprocesamiento y clasificación. El objetivo será comprender la forma de aplicar todos estos conceptos a problemas reales, a conjuntos de datos susceptibles de ser analizados.

Ya que trabajaremos habitualmente con **R** el guión de prácticas se centra en estudiar y presentar diferentes paquetes que pueden sernos útiles para esta tarea, aunque hay que señalar que **R** no es una herramienta específica para análisis de datos. Hay que entender las ventajas e inconvenientes de este entorno de trabajo. Una de las principales ventajas consiste en disponer de un gran número de algoritmos y métodos ya implementados y listos para su uso. Pero no hemos de olvidar que en la mayoría de los casos estos algoritmos y métodos se han desarrollado en el seno de trabajos de investigación y no en el ámbito de desarrollo de aplicaciones finales para ser comercializadas. Y esto provoca que en la mayoría de los casos la documentación disponible sea escasa, difícil de leer y poco clarificadora en cuanto a los detalles de bajo nivel de los métodos y funciones disponibles.

El esquema que seguiremos durante estas sesiones se basa en el proceso normal que se usaría cuando se procede al análisis de un conjunto de datos con el objetivo de obtener algún tipo de conocimiento sobre él. En concreto consideramos las siguientes operaciones de preprocesamiento:

- apertura y almacenamiento de datos
- visualización
- visión preliminar de los datos
- imputación de valores perdidos
- detección de datos anómalos
- transformación de los datos
- discretización
- selección de características

Estas operaciones de preprocesamiento se irán analizando mediante la presentación de scripts de **R** que irán detallando los pasos necesarios para aplicarlas a conjuntos de datos concretos, gracias a la aplicación de funcionalidad ya disponible en diferentes paquetes de **R**.

2. Apertura y almacenamiento de datos

La forma de organización de datos más usada es mediante tablas. Los archivos que usaremos suelen presentar dicha estructura. En primer lugar contendrán la descripción de las variables del problema (con mayor o menor detalle, dependiendo del formato; al menos siempre aparecerá algún identificador único asociado a cada variable) y después aparecerán los valores de las variables para cada una de las instancias disponibles. Cada ejemplo se describe mediante una línea (fila) del archivo y contendrá (idealmente) los valores de todas las variables consideradas (columnas).

Un formato muy usado es el denominado **csv** (**comma separated values**), donde los valores de las variables aparecen separados por una coma. Habrá, como hemos indicado antes, una fila por cada instancia y una columna por cada variable. Empezamos considerando la forma en que podemos leer un archivo de datos con formato **csv** (aunque se haya visto con anterioridad así sirve de breve repaso). En cualquier caso, hemos de ser conscientes de la existencia de diferentes formatos de datos, cada uno de ellos con características específicas.

A lo largo de estas sesiones se considerarán varios conjuntos de datos. En algunos casos se usa un conjunto voluminoso (disponible en el directorio **data** del proyecto, con el nombre **datos.csv**). En algunas operaciones concretas se usarán otros diferentes, para asegurar tiempos de ejecución más cortos o para permitir algunas visualizaciones de datos que no serían posibles con muchas variables.

Al usar **RStudio** como herramienta básica de trabajo resulta conveniente organizar el código mediante un proyecto de trabajo. Esto permitirá mantener el código y recursos necesarios, como conjuntos de datos, de forma ordenada. A continuación se describe brevemente la forma en que se procedería para crear un proyecto de trabajo (hay que tener en cuenta que las rutas que aparecen en la siguiente descripción son las usadas en mi máquina de trabajo y que cada uno habrá de adaptarlas a su sistema de archivos):

1. seleccionar opción **File** en la barra principal de menú

2. seleccionar opción **New Project** en la ventana desplegable. Esta operación hace aparecer una nueva ventana. Se trata de un asistente que nos irá guiando durante el proceso de creación del proyecto

3. si se crea un proyecto a partir de cero se selecciona la opción **New directory**. El siguiente paso entonces consistirá en indicar si se desea crear un proyecto vacío, un paquete de **R** o una aplicación web. Seleccionamos la opción **Empty Project** y aparece una nueva ventana en la que hemos de indicar el directorio base que contendrá los recursos del proyecto (archivos, datos, etc) (**Directory name**) y la ruta en que se ubicará tal directorio (**Create projects as subdirectory of**). En mi caso he seleccionado **preprocesamiento** como nombre de directorio y he indicado la ruta **$HOME/docencia/masterDatcom/2018-2019/proyectosR**. **Nota: vosotros debéis usar la ruta que consideréis conveniente**

4. con esto se creará el proyecto y en **RStudio** veremos los archivos que lo componen. De ellos hay uno especial que contiene todos los datos del proyecto y que será el que abriremos cada vez que empecemos a trabajar con él: **preprocesamiento.Rproj**

5. también es posible crear un proyecto a partir de un directorio ya existente que contiene archivos y recursos. En este caso seleccionad la opción **Existing Directory**. En la siguiente pantalla basta con seleccionar el directorio donde se encuentran

los archivos y el proyecto se crea de forma automática. Por facilidad de uso será esta la opción que sigamos, ya que podemos así partir de un directorio base en que hayamos descargado los archivos desde la plataforma docente. En el caso de haber partido de un proyecto vacío tendríamos que ir agregando los archivos uno a uno al proyecto, a medida que se fueran creando

6. activación de marca **Source on Save** en la parte superior de la pantalla. Esto permitirá que se cargue la definición de la función (o funciones) en el entorno de trabajo de **R** cada vez que lo guardemos, activando así los cambios que se hayan hecho sobre el código. Nota: esta opción debe evitarse si el archivo contiene sentencias ejecutables (además de funciones), ya que todas ellas se ejecutarán en el momento de salvar el archivo

7. para que todo quede más organizado se ha creado un directorio llamado **data** en el directorio del proyecto y en él se encuentra el archivo **datos.csv** del que hemos hablado con anterioridad

Una vez creado el proyecto podemos empezar a estudiar y completar los archivos incluidos en él. Comenzamos considerando la funcionalidad de lectura y escritura de datos, implementada en el archivo **lecturaDatos.R**. Este archivo contiene el código necesario para apertura y almacenamiento de datos, junto con algunos ejemplos de uso. Su contenido se muestra a continuación:

```
# funcion para lectura de archivo. En este script tambien se
# consideran opciones de analisis exploratorio preliminar de
# los datos. Argumentos:
# @param path ruta hasta el archivo de datos a leer
# @param file archivo a leer
lecturaDatos <- function(path,file){
  # se compone el path completo
  pathCompleto <- paste(path,file,sep="/")

  # se leen los datos
  dataset <- read.csv(pathCompleto,na.strings=c(".","NA","","?"))

  # se devuelve el conjunto de datos
  return(dataset)
}

# funcion para almacenar un conjunto de datos. Argumentos:
# @param path ruta donde se quiere almacenar
# @param file nombre del archivo donde almacenar los datos
# @param cdataset conjunto de datos a almacenar
escrituraDatos <- function(path, file, dataset){
  # se compone el path completo
  pathCompleto <- paste(path,file,sep="")

  # se escribe el archivo
  write.csv(dataset, pathCompleto, row.names = FALSE)
}

# se incluyen aqui algunas sentencia para ver la forma en
# que se ejecutan las funciones anteriores

# se comienza leyendo un conjunto de datos en forma csv
```

```
datos <- lecturaDatos("./data/","datos.csv")

# se visualiza el tipo de objeto devuelto por la operacion
# de lectura de datos
class(datos)
```

Se definen aquí dos funciones que nos serán muy útiles para leer y almacenar los datos (por ejemplo, tras haber realizado algún tipo de transformación sobre ellos, como las que consideraremos más adelante). La función de lectura recibe como argumento la ruta en que se encuentra el conjunto de datos y el nombre del archivo a leer. Devuelve un conjunto de datos con la información leída del archivo. En los ejemplos de uso de estas funciones que veremos he definido las rutas de almacenamiento en base al proyecto de **RStudio**, donde se cuenta con un directorio **data** en que se ubican algunos conjuntos de datos.

La forma de uso de esta función es la que se indica en las líneas finales del *script* previo. Se trata de leer un conjunto de datos con formato **csv**, cuyas primeras líneas serían las siguientes:

```
separation, propensity, length, PredSS_r1_-1, PredSS_r1, PredSS_r1_1,
PredSS_r2_-1, PredSS_r2, PrSS_fq_cn_H, PrSS_fq_cn_E, PrSS_fq_cn_C,
PrCN_fq_cn_0, PrRCH_fq_cn0, PrRCH_fq_cn1, PrSA_fq_cn_0, PrSA_fq_cn_4,
PrRCH_r1_-1, PrRCH_r1, PrRCH_r1_1, PrRCH_r2_-1, PrRCH_r2, PrRCH_r2_1,
PrCN_r1_-1, PrCN_r1, PrCN_r1_1, PrCN_r2, PrSA_r1, PrSA_r2_-1, PrSA_r2,
PrSA_r2_1, PrRCH_fq_gl0, PrRCH_fq_gl4, AA_fq_cn_A, AA_fq_cn_D,
AA_fq_cn_E, AA_fq_cn_I, AA_fq_cn_F, PrSS_fq_gl_H, PrCN_fq_gl_0,
PrSA_fq_gl_0, PSSM_r1_-4_A, PSSM_r1_-4_N, PSSM_r1_0_D, PSSM_r1_1_W,
PSSM_r2_0_A, PSSM_cn_-2_A, PSSM_cn_-2_T, PSSM_cn_0_H, PSSM_cn_2_D,
PSSM_cn_2_V, class
27,0.636381526324397, 161, E, E, E, C, C, 0.269, 0.115, 0.615, 0.308,
0.346,0.192,0.308, 0.385, 3, 3, 4, 3, 3, 0, 4, 4, 4, 3, 0, 3, 1, 4,
0.23, 0.23, 0.115, 0.077, 0.038,0.077,0.038,0.13, 0.106, 0.354, 4, 4,
-6, -4, -5, 2, -5, 2, -5, -5, negative
```

Una vez cargados podemos ver el tipo de objeto que se ha creado (**datos**) y que contiene toda la información del archivo (nosotros tecleamos la sentencia **class**, pasando como argumento el objeto devuelto por la función de lectura de datos y abajo aparece la información sobre el tipo asociado a datos):

```
class(datos)
[1] "data.frame"
```

El tipo de objetos **data.frame** es el más adecuado para almacenar conjuntos de datos y responde a la organización básica que hemos indicado antes: una columna para cada variable y una fila para cada instancia. Esta tipo de datos ofrece muchas funciones que permiten un manejo cómodo de los datos: selección de instancias, de variables, acceso a datos particulares, fusión con otros conjuntos de datos, aplicación de filtros, etc.

3. Visualización

La frase *vale más una imagen que mil palabras* puede servirnos para describir la importancia de la visualización de los datos. En esta sección consideraremos la forma de obtener gráficos que puedan sernos de ayuda usando el paquete **ggplot2**. Hay otros muchos que ofrecen gráficas de interés (en realidad, casi todos los paquetes agregan

sus facilidades de visualización), pero aquí nos centramos en este debido a que está especialmente pensado para visualización y a la importancia que está cobrando debido a sus continuas mejores y a su generalidad.

Para ilustrar su funcionalidad usaremos conjuntos de datos sencillos. Uno de ellos se denomina **mpg** e incluye los datos de 38 modelos de coches, recogidos por la agencia de protección del medio ambiente de Estados Unidos. Las variables del conjunto de datos son:

- *manufacturer* (fabricante)
- *model* (modelo)
- *displ* (cilindrada)
- *year* (año)
- *cyl* (número de cilindros)
- *trans* (tipo de cambio)
- *drv* (tipo de tracción, f (frontal), r (posterior), 4 (a las cuatro ruedas))
- *cty* (eficiencia en circuito urbano, dado en millas por galón)
- *hwy* (eficiencia en autopista, en millas por galón)
- *fl* (tipo de combustible)
- *class* (tipo de vehículo: compacto, medio, suv, dos asientos, minivan, pickup)

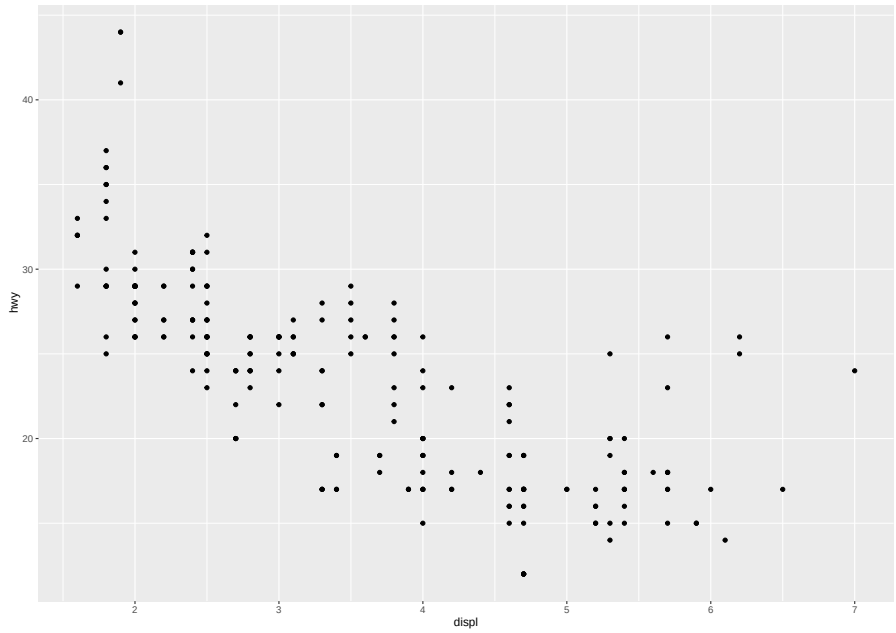
Este conjunto de datos está aportado por **ggplot2** y podemos obtener información sobre él del modo habitual:

```
library(ggplot2)

# se usa el conjunto de datos llamado mpg. La forma de obtener
# ayuda sobre el conjunto de datos es la indicada a continuacion
?mpg
```

El primer paso consistirá en hacer un gráfico simple para poder examinar la relación entre dos variables: por ejemplo, la eficiencia en autopista (*hwy*) y la cilindrada (*displ*). La sentencia necesaria será:

```
# se realiza un grafico simple para ver la relacion entre las
# variables hwy y displ
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy))
```

El gráfico obtenido es:



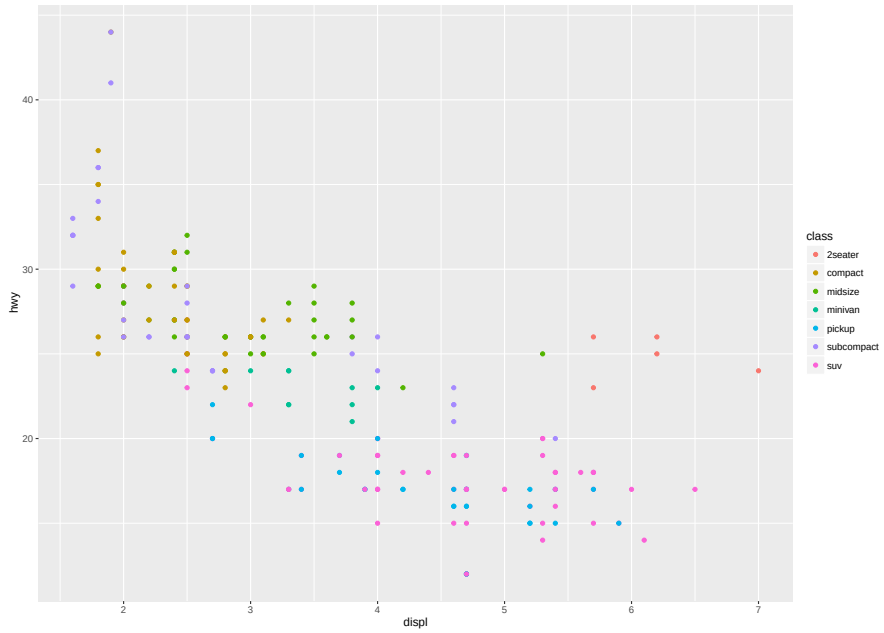
Se observa de la sentencia usada que:

- la función básica es **ggplot**, que recibe como argumento el conjunto de datos completo a visualizar. Esta función produce la estructura básica del gráfico, que incluye un sistema de coordenadas
- a esta estructura básica se agrega una nueva capa, mediante la función **geom_point**, que permite representar los datos de interés. **ggplot2** ofrece muchas funciones geométricas distintas, específicas para diferentes tipos de gráficos. En este guión veremos algunas de ellas
- cada función geométrica recibe un argumento tipo **mapping** que indica la forma en que las variables del conjunto de datos se corresponden con las propiedades visuales del gráfico. Este argumento siempre va asociado al uso de la función **aes** que define las propiedades estéticas. Los argumentos **x** e **y** indican las variables que se hacen corresponder con los ejes X e Y

El gran valor de un gráfico radica en ofrecernos una visión de los datos que no tenemos mediante su simple observación en tablas o conjuntos de datos. Por ejemplo, en el gráfico anterior podemos observar que en la parte superior derecha aparece un conjunto de puntos que no parecen seguir la tendencia general del resto de datos: a mayor cilindrada disminuye el número de millas recorridas con cada galón de combustible, es decir, hay menor eficiencia energética. La cuestión es ver a qué se debe este comportamiento. Para ello podemos usar una tercera variable, a la que haremos corresponder alguna propiedad del gráfico (tamaño de los puntos, color, forma, etc). Por ejemplo, nosotros haremos que el color indique el valor de la variable *class* asociada a la instancia representada por cada punto:

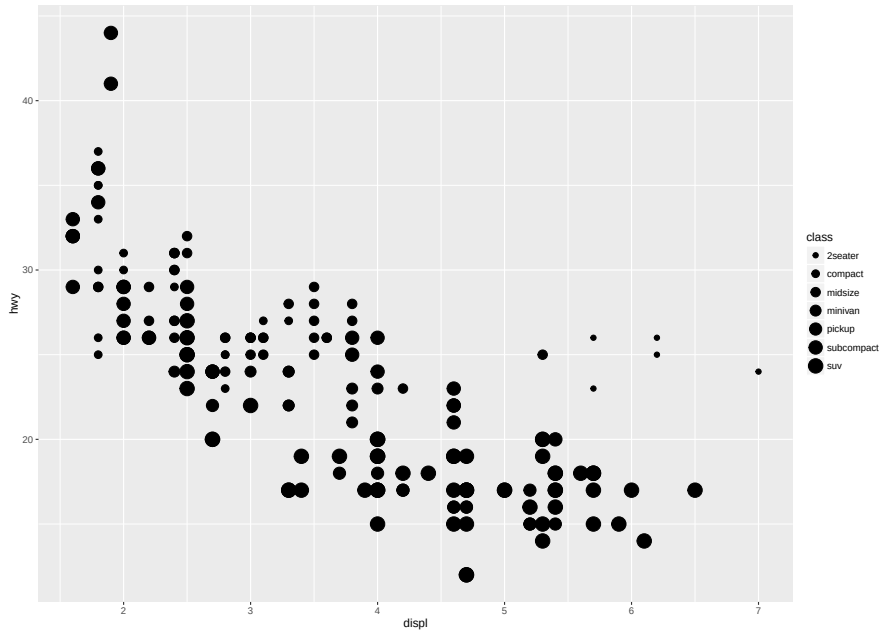
```
# vemos el efecto de la variable clase mediante una propiedad
# estetica de los puntos
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

El gráfico obtenido ahora es el siguiente:



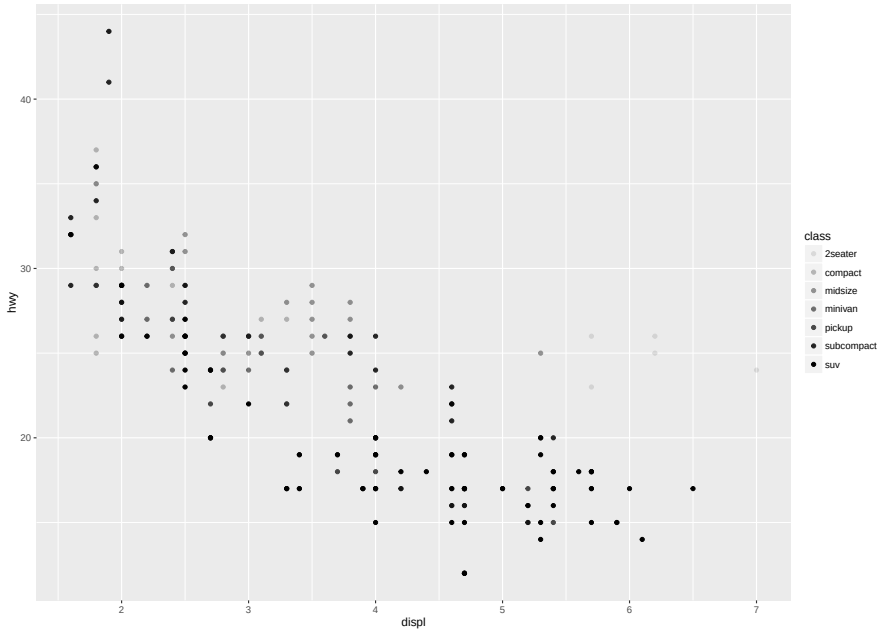
Esta forma de visualización permite comprobar que los vehículos que se salen de la tendencia habitual son vehículos de dos asientos, es decir, deportivos. En ellos, pese a disponer de motores muy potentes, su diseño hace que el consumo energético sea menor del esperado. También podríamos haber establecido una correspondencia entre el tipo de coche y el tamaño del punto que representa cada vehículo:

```
# podemos usar el tam. del punto para representar los diferentes
# tipos de coches. Esta sentencia genera un aviso, porque no es
# buena idea usar el tam. para variables discretas cuyos dominios
# no estan ordenados
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
```

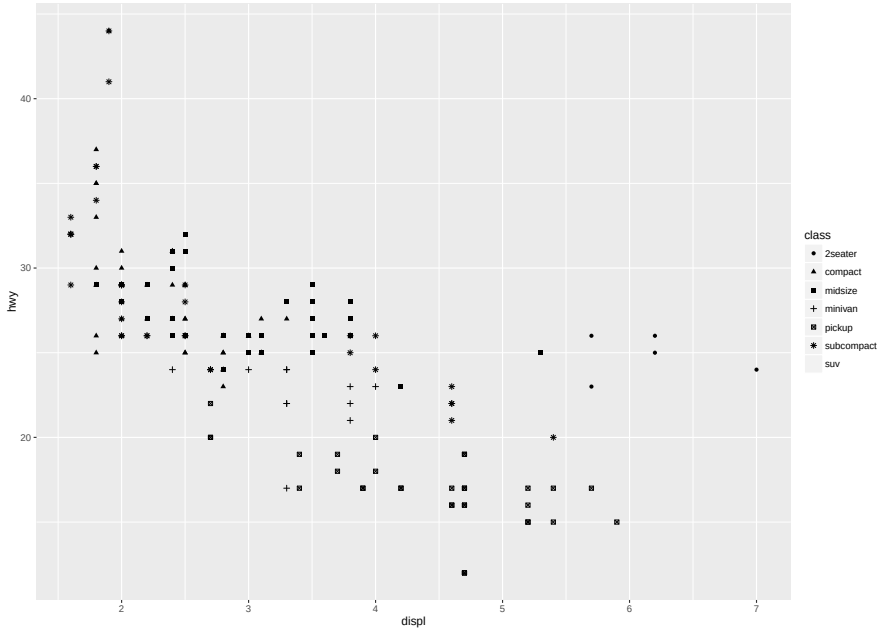


Esta sentencia genera un aviso, ya que no es buena idea usar esta técnica en el caso en que se use con variables discretas cuyo dominio no esté ordenado. Otra posible correspondencia, para una tercera variable, podría haber sido la de la transparencia de los puntos o su forma:

```
# uso de transparencia para representar los tipos de coches
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
```




```
# igual con la forma de los puntos
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```



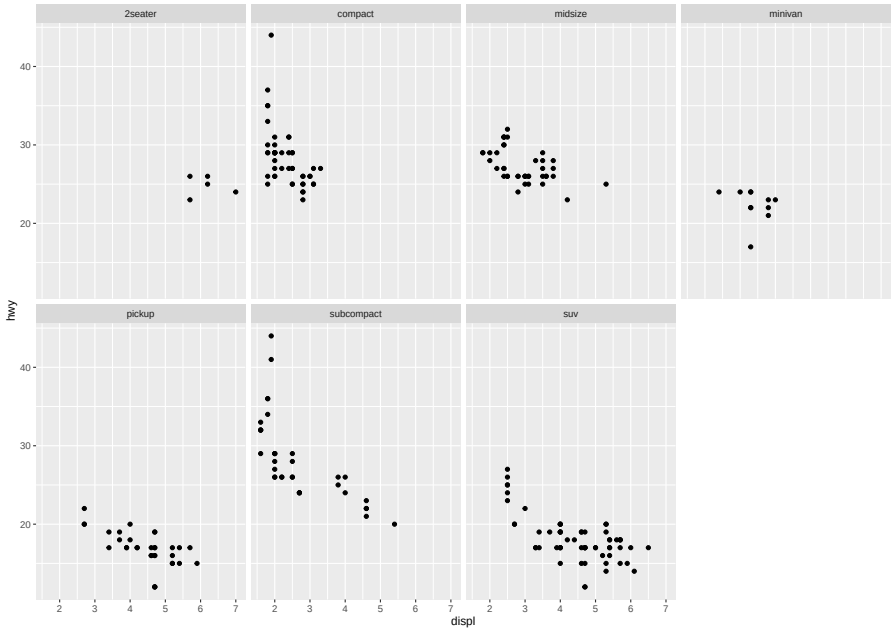
En cualquier caso, se observa que una vez indicadas las características estéticas de los puntos, **ggplot2** se ocupa de lo demás: asigna valores a la propiedad de forma automática, crea leyendas, etc. Uno de los problemas más frecuentes al usar este paquete consiste en colocar el signo + al principio de cada línea, en lugar de al final.

Otra forma de agregar más variables a un gráfico bidimensional es mediante el uso de subgráficos que muestran subconjuntos de datos. Para subdividir los datos mediante una variable usamos la función **facet_wrap**, cuyo primer argumento es una fórmula (objeto típo de **R**), creada mediante la expresión:

```
~ nombreVariable
```

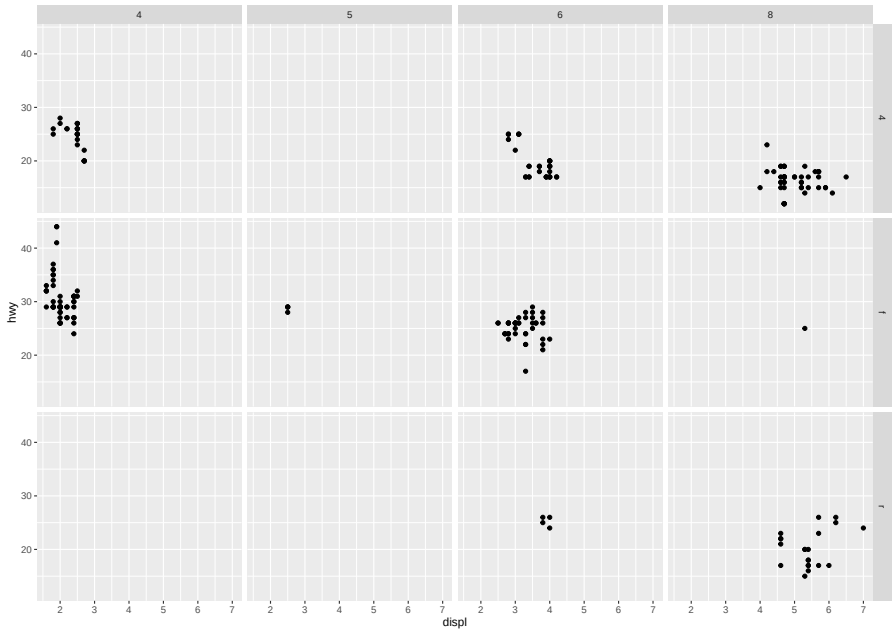
La variable a usar debe ser discreta. Por ejemplo, al subdividir los datos de vehículos mediante la variable clase tenemos:

```
# se usa facet_wrap para subdividir los datos por la variable clase
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



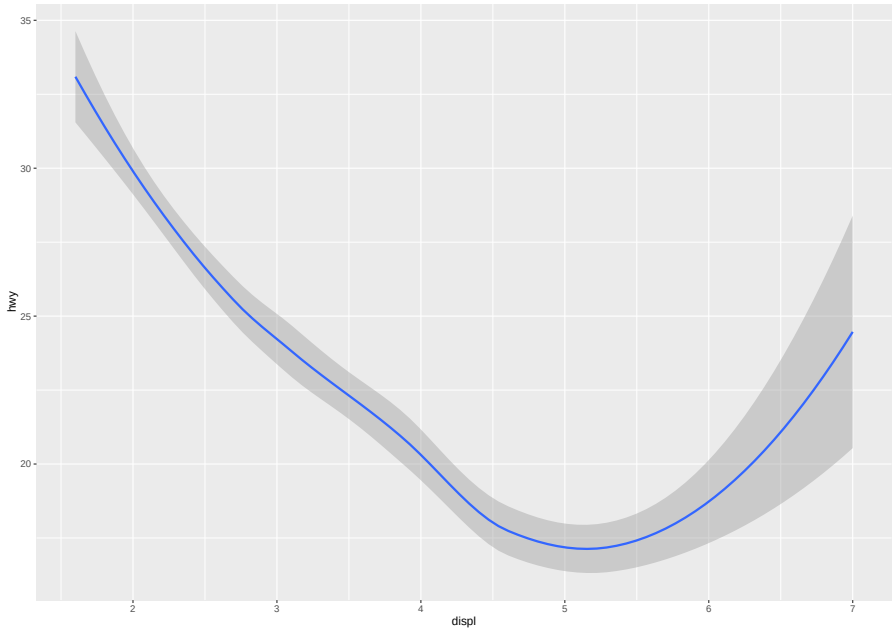
También pueden subdividirse los datos mediante dos variables (*drv* y *cyl*):

```
# subdivision de datos mediante dos variables
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl)
```



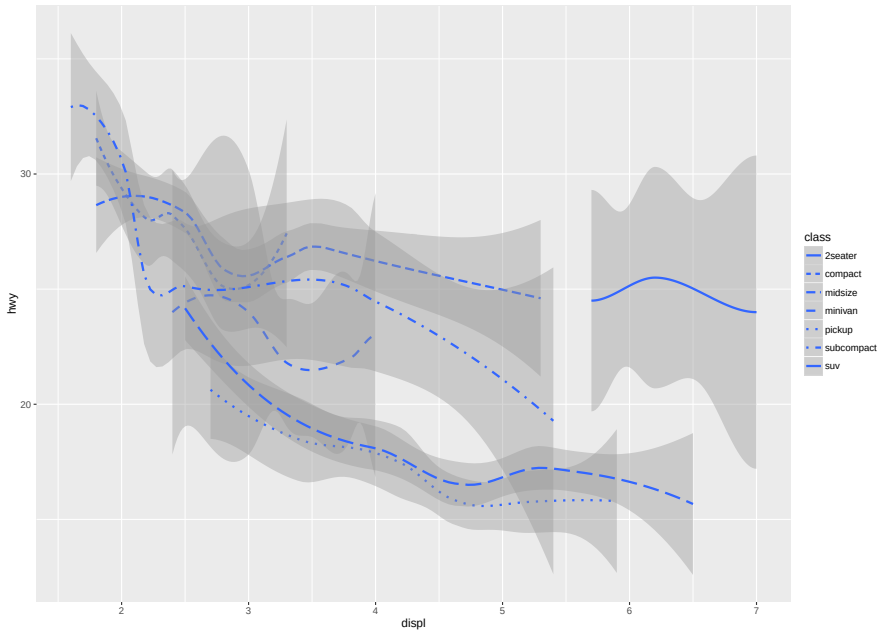
Los datos pueden representarse mediante diferentes características geométricas: diagramas de barra, diagramas de cajas, líneas de diferentes tipos, etc. Una representación útil puede ser **geom_smooth**, que muestra una línea indicando la tendencia de los datos:

```
# representacion usando geom_smooth
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```



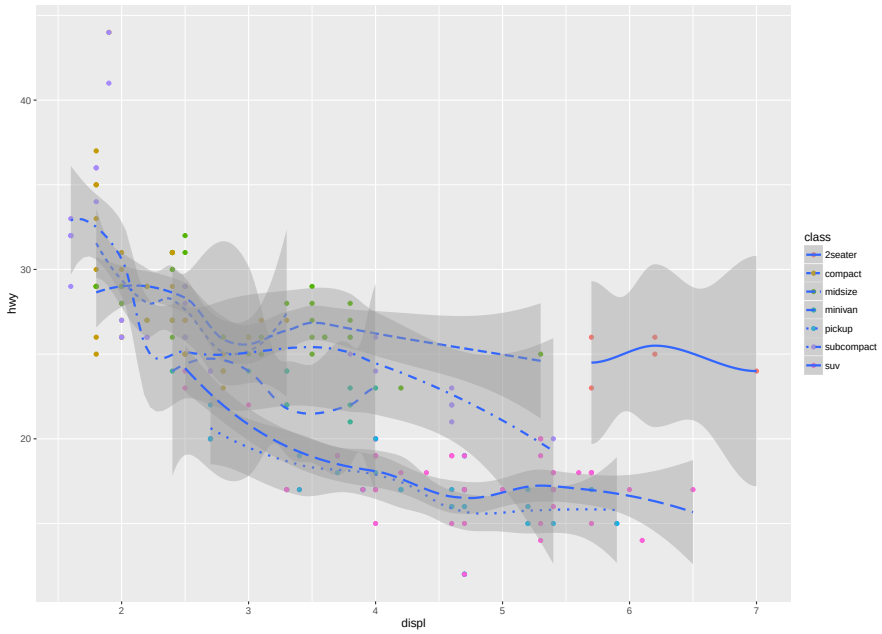
Usando esta representación geométrica podemos separar los datos mediante un tipo de línea diferente para cada subconjunto (por ejemplo, se puede hacer la división según el valor de la variable *class*):

```r
# uso de geom_smooth, clasificando los datos de acuerdo a la
# variable clase
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = class))
```



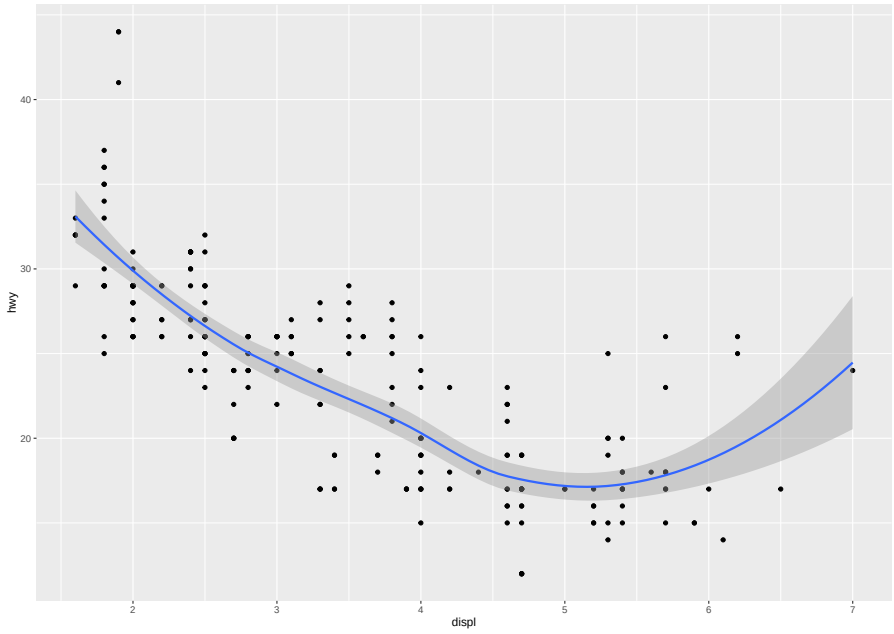
E incluso es posible combinar diferentes representaciones geométricas:

```r
# uso de geom_smooth, pero visualizando tambien los puntos
# que representan a los datos
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color=class)) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype=class))
```



Otro ejemplo del uso de varias geometrías es el siguiente:

```
# otro ejemplo de visualizacion con varios tipos de geometria
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

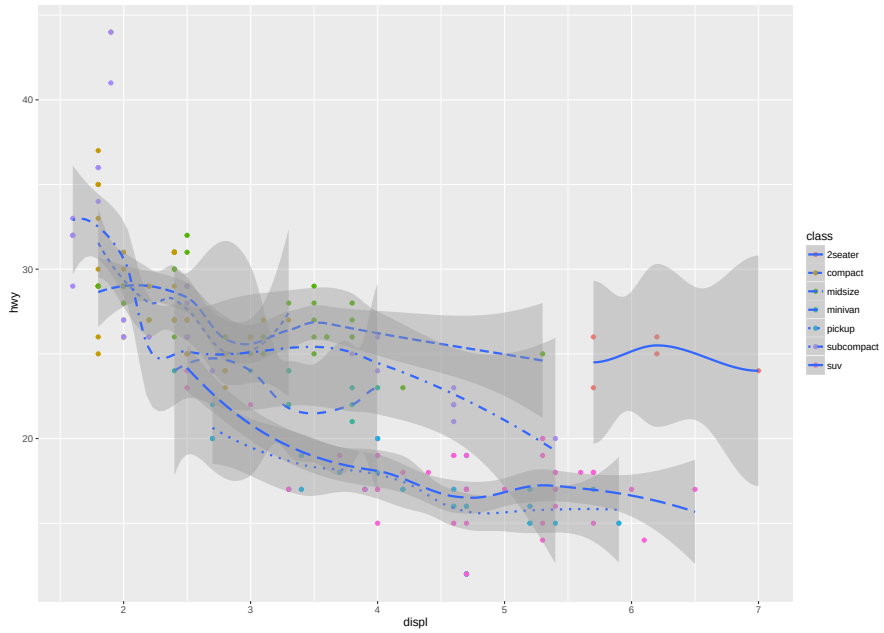


En este caso se observa que hay duplicación en los datos a incluir en la sentencia. Por ejemplo, si deseamos modificar las variables asignadas a los ejes habremos de cambiar dos líneas de código, lo que facilita que se comentan errores. Este tipo de repetición puede evitarse usando la función **mapping** directamente sobre la función **ggplot**, de forma que se convierte en algo global para el gráfico. De esta forma, el mismo gráfico anterior se obtiene de forma más sencilla de la siguiente manera:

```
# forma de evitar la repeticion: mapping sobre ggplot
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

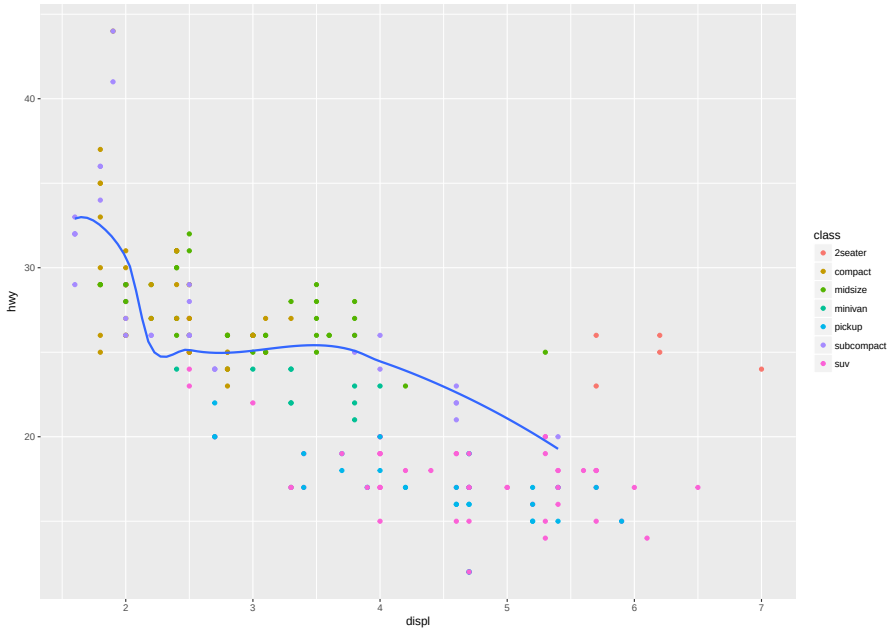
Si la definición de **mapping** se hace sobre una geometría específica, entonces se trata de propiedades concretas de una capa del gráfico, locales a ella. Esto permite disponer de diferentes tipos de estética para cada capa, como se aprecia a continuación:

```
# usando mappings locales y globales
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color=class)) +
  geom_smooth(mapping = aes(linetype = class))
```



La misma idea puede usarse para especificar diferentes conjuntos de datos para cada capa:

```
# diferentes conjuntos de datos para cada capa
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(data = filter(mpg, class == "subcompact"), se = FALSE)
```

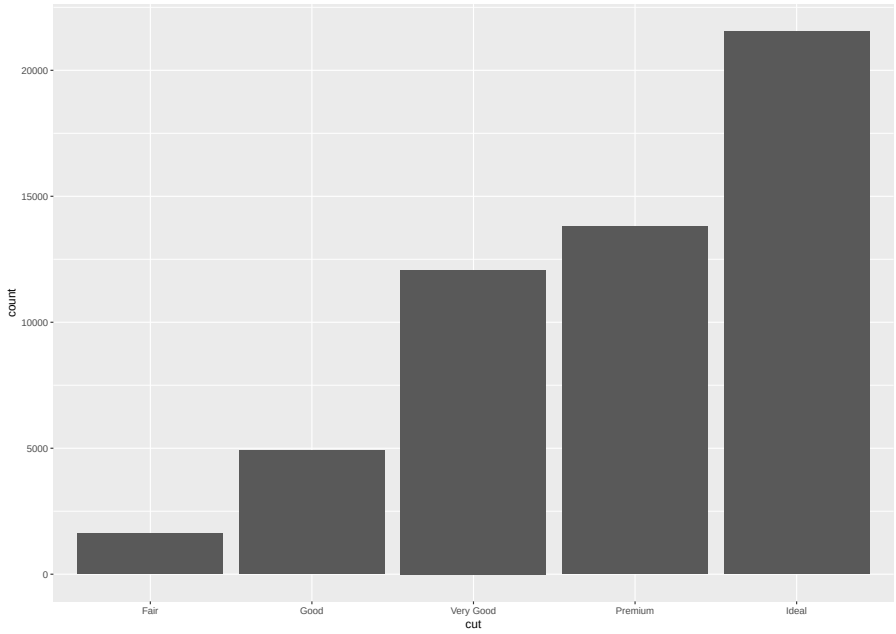


El argumento **se=FALSE** hace que no se muestre la zona sombreada alrededor de la línea de ajuste para los datos correspondientes a la clase **subcompact**.

Otro tipo de gráficos interesante usa la geometría de los diagramas de barras (mediante **geom_bar**). Para ilustrar su uso usamos un conjunto de datos sobre diamantes. Contiene información de aproximadamente 54000 diamantes, incluyendo características

como precio, peso, calidad de corte, color, claridad, dimensiones, etc. La representación de la calidad del corte se obtendría de la siguiente forma:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x=cut))
```

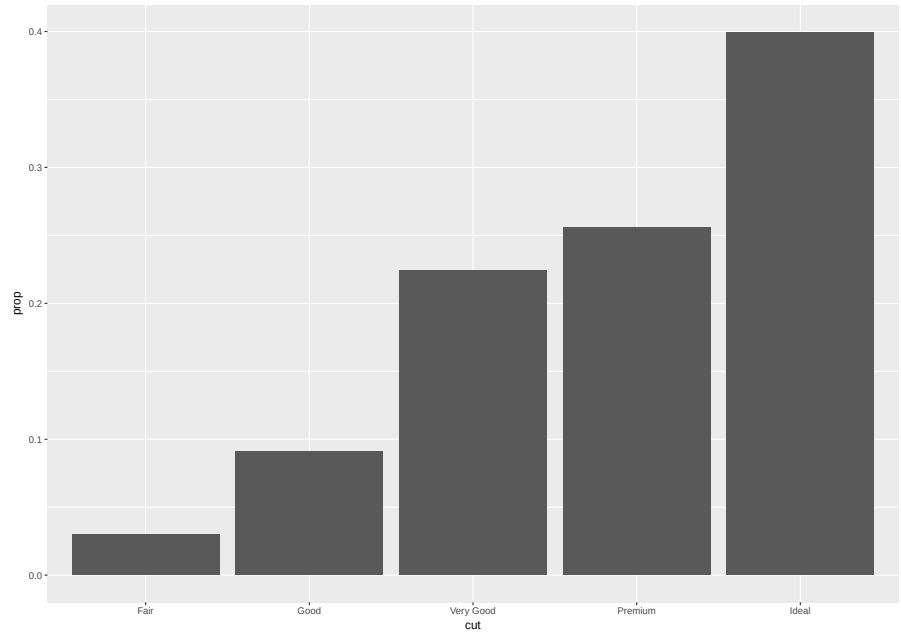


Se aprecia que hay más diamantes con cortes de alta calidad. También se reseña que en el eje Y aparece el número de diamantes de cada tipo, una información que no está contenida inicialmente en el conjunto de datos. Esto indica que algunas geometrías precisan del cálculo de ciertos valores (algo que realizan de forma interna, sin que tengamos que ocuparnos de ello). Esto se realiza mediante un algoritmo denominado **stat**. Cada geometría implica un uso específico de este algoritmo. Por ejemplo, los diagramas de barras precisan del conteo de valores de cada tipo (mediante **stat_count**). Por esta razón podíamos haber obtenido el gráfico anterior de la siguiente forma:

```
ggplot(data = diamond) +
  stat_count(mapping = aes(x=cut))
```

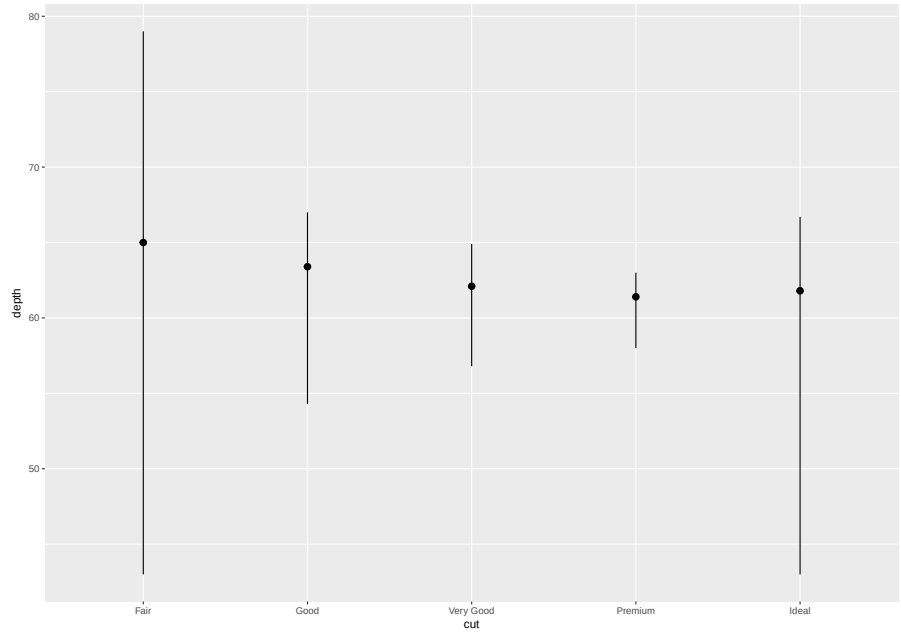
El comportamiento por defecto del algoritmo de cálculo de estadísticas puede modificarse. Por ejemplo, podría interesar mostrar las proporciones de diamantes de cada calidad de corte (en lugar del conteo):

```
ggplot(data = diamond) +
  geom_bar(mapping = aes(x=cut, y = ..prop.., group=1))
```



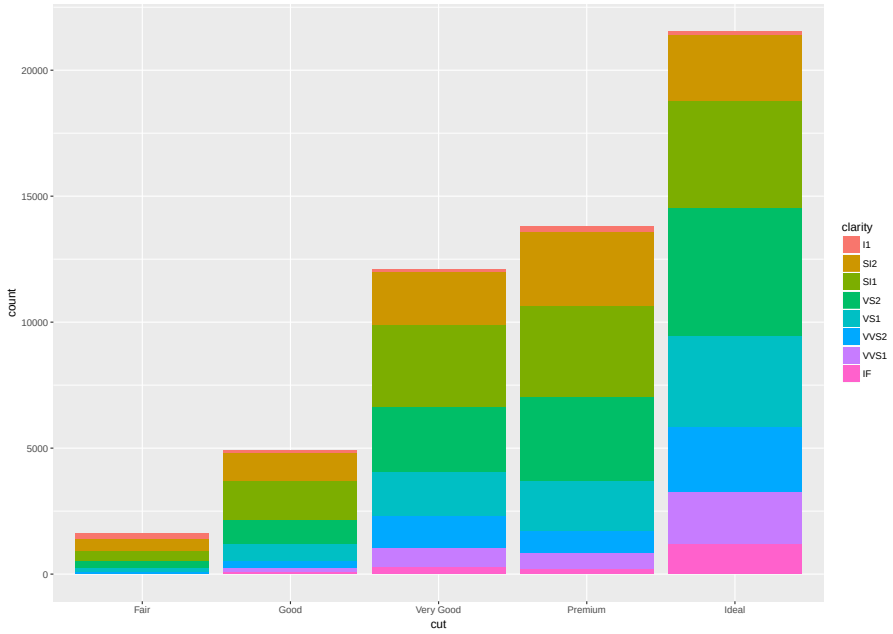
También es posible enriquecer el gráfico con más información sobre cada tipo de calidad de corte en relación con otra variable llamada *depth* (valor mínimo, máximo y mediana):

```
# se enriquece el grafico con mas informacion sobre las calidades
# de corte
ggplot(data = diamonds) +
  stat_summary(
    mapping = aes(x=cut, y=depth),
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```

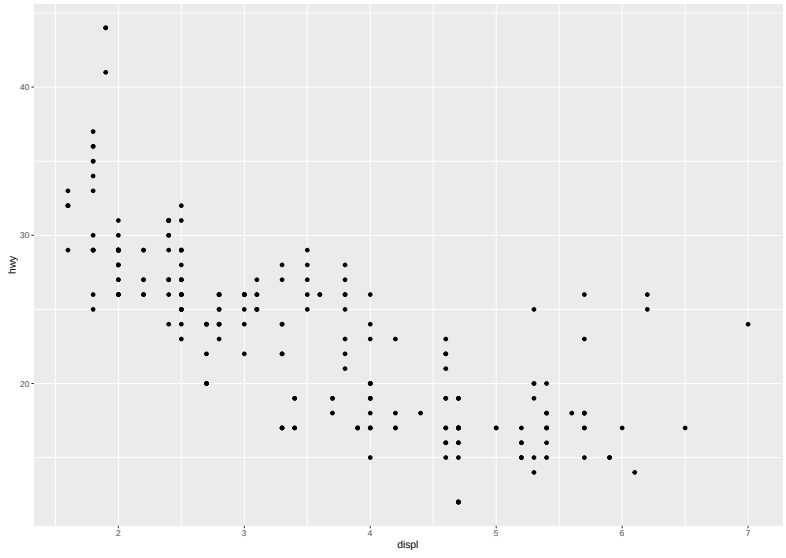


E incluso podemos hacer que algunas propiedades estéticas nos muestren información sobre otras variables (como por ejemplo, la variable *clarity*):

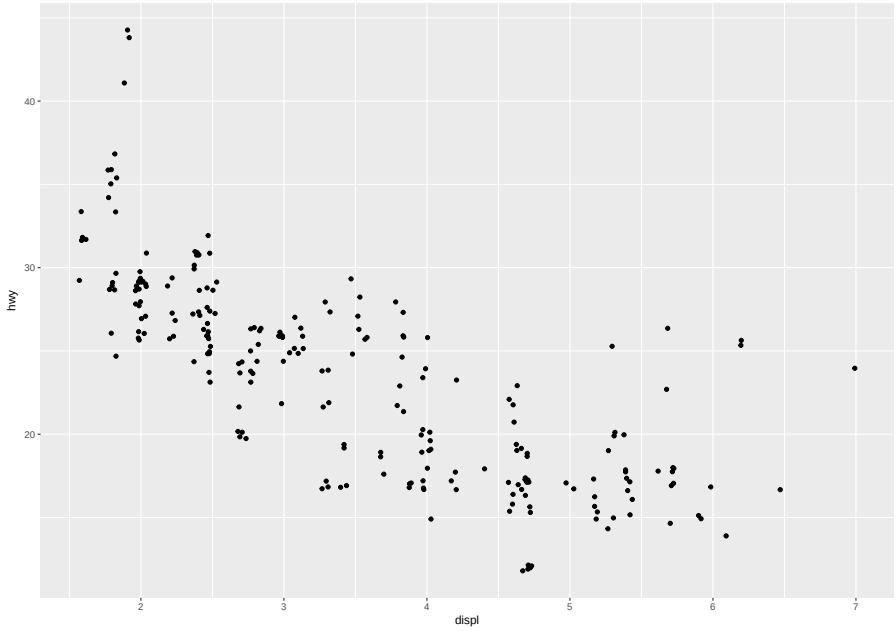
```
# se muestra informacion sobre clarity en las barras
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity))
```



Hay un efecto en los gráficos conocido como **overplotting**, que consiste en que la visualización de un conjunto de puntos resulta confusa de alguna manera ya que un mismo punto está representando en realidad varias instancias. Una forma de evitar este efecto es añadiendo a cada punto una cierta componente aleatoria (esta técnica se conoce como **jitter**). Esto puede hacerse de forma sencilla con **ggplot**. La diferencia entre su uso y la representación tal cual de los datos se muestra en los gráficos siguientes: en el primero se usa la representación por defecto, vista antes, y en el segundo se realiza la representación activando la opción **jitter**):

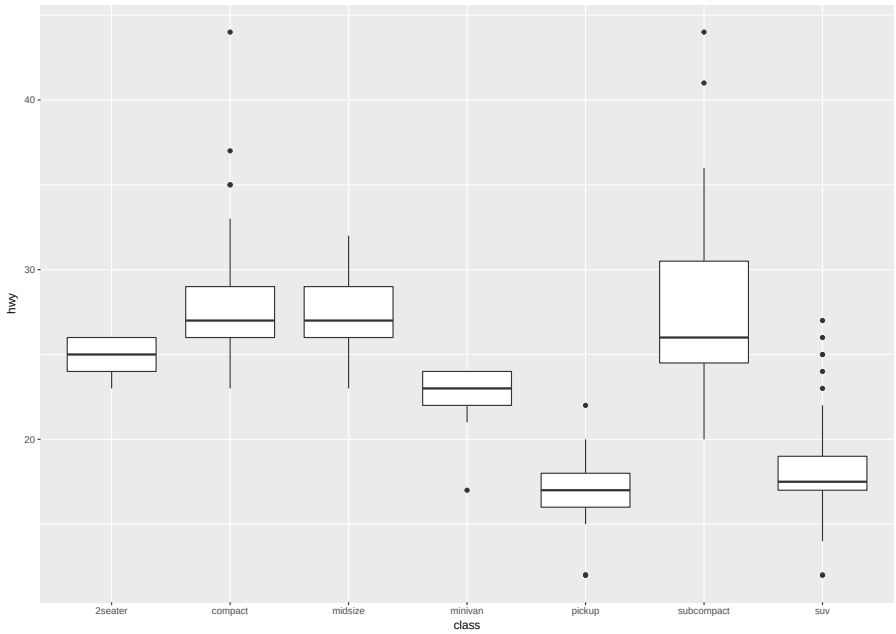


```r
# uso de la opcion jitter
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy),
             position = "jitter")
```

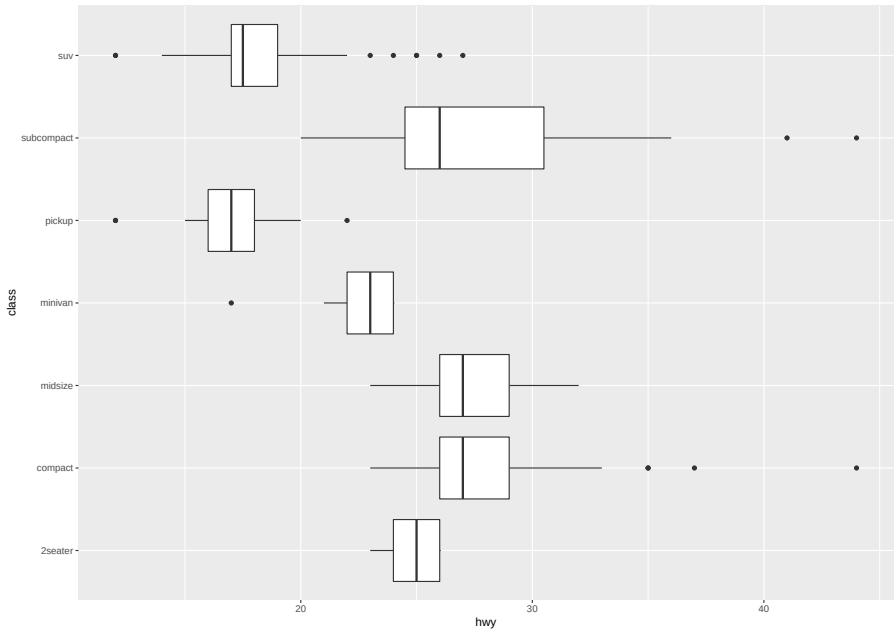


Los sistemas de coordenadas también pueden manipularse mediante este paquete. Por ejemplo, hay situaciones en que la variable en el eje X presenta etiquetas muy largas y conviene más presentarlas en el eje Y. Un ejemplo se muestra a continuación:

```r
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
```



```r
# para evitar el solapamiento se giran los ejes
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```



4. Paquete dplyr y nuevas formas de organización de datos

El trabajo en **R** es permanente y no dejan de aparecer nuevas estructuras de datos que permiten simplificar la forma de trabajar con colecciones de datos. Una de ellas se conoce como **tibble** y está disponible en el paquete **dplyr**. Se trata de una versión mejorada de la estructura básica de almacenamiento de conjuntos de datos: **data.frame**. Veamos en esta sección algunas de las posibilidades de esta nueva forma de organización y de la forma de trabajo con ella mediante las posibilidades del paquete **dplyr**.

Comenzamos considerando un conjunto de datos organizado mediante esta estructura: **flights**, conjunto de datos del paquete **nycflights13**. Comenzaremos con una visión preliminar del mismo:

```r
library(nycflights13)

# se muestra informacion de resumen del conjunto de datos
# (de tipo tibble)
flights
```

El resultado de ejecutar esta sentencia aparece a continuación:

```
# A tibble: 336,776 x 19
 ...................................................
# ... with 336,766 more rows, and 1 more variable: time_hour <dttm>
```

El filtrado de datos en esta estructura es muy simple gracias al uso de la sentencia **filter**:

```
unoEnero <- filter(flights, month == 1, day == 1)
```

Para el filtrado pueden agregarse expresiones lógicas:

```
eneroFebrero <- filter(flights, month == 1 | month == 2)
```

Y también pueden usarse los operadores <, >, etc:

```
retrasosLlegadas <- filter(flights, arr_delay > 120)
retrasosLlegadasSalidas <- filter(flights, arr_delay > 120 & dep_delay
    > 120)
```

E incluso es posible especificar los valores de interés en un vector:

```
# se pueden especificar varios valores de interes en un vector
eneroFebreroMarzo <- filter(flights, month %in% c(1,2,3))
```

La ordenación de las filas se realiza de forma muy sencilla mediante la operación **arrange**:

```
# se pueden ordenar las filas de acuerdo al valor de las
# variables seleccionadas
origenDestino <- arrange(flights, origin, dest)
```

También podemos quedarnos con algunas columnas de interés mediante **select**:

```
# seleccion de columnas especificas con select
algunasColumnas <- select(flights, year, month, day, origin, dest)

# puede seleccionarse un determinado rango de columnas
seleccionRango1 <- select(flights, year:dest)

# tambien evitar todos las columnas en un rango determinado
seleccionRango2 <- select(flights,-(year:day))

# posibilidad de uso de funciones auxiliares para seleccion
# de variables
seleccion3 <- select(flights, starts_with("a"))
seleccion4 <- select(flights, ends_with("time"))
seleccion4 <- select(flights, contains("d"))
```

La función **rename** permite cambiar el nombre a las variables del conjunto de datos:

```
# la variable tailnum (a la derecha) ahora se llamara tailnumber.
# Las variables que no se indican en rename mantienen su nombre
renamed <- rename(flights, tailnumber=tailnum)
```

La función **everything** puede usarse en combinación con **select** para seleccionar el resto de columnas que no hayan sido nombradas expresamente:

```
# uso de everything en combinacion con select
reordenada <- select(flights, time_hour, air_time, everything())
```

De esta forma, las variables *time_hour* y *air_time* serán las primeras en aparecer. Tras ellas, todas las demás, gracias al uso de **everything**. La selección de variables puede usar de forma combinada algunas de las posibilidades vistas antes:

```
versionNueva <- select(flights, year:day, ends_with("delay"), distance,
    air_time)
```

También se permite la creación de nuevas variables mediante **mutate**:

```
versionNueva <- mutate(versionNueva, gain=arr_delay-dep_delay, speed=
    distance/air_time*60)
```

Esta sentencia agrega al conjunto de datos denominado **versionNueva** las variables **gain** (a partir de la diferencia entre **arr_delay** y **dep_delay**) y **velocidad** (a partir de **distance** y **air_time**). También es posible quedarse únicamente con las variables nuevas recién creadas, mediante **transmute**:

```
versionNueva <- transmute(versionNueva, gain=arr_delay-dep_delay, speed
    =distance/air_time*60)
```

A veces resulta conveniente agrupar los datos de acuerdo al valor de alguna (o algunas) variables:

```
porDia <- group_by(flights, year, month, day)
```

Una vez agrupados pueden obtenerse algunas medidas de resumen mediante **summarize**:

```
resumen <- summarize(porDia, delay=mean(dep_delay, na.rm=TRUE))
```

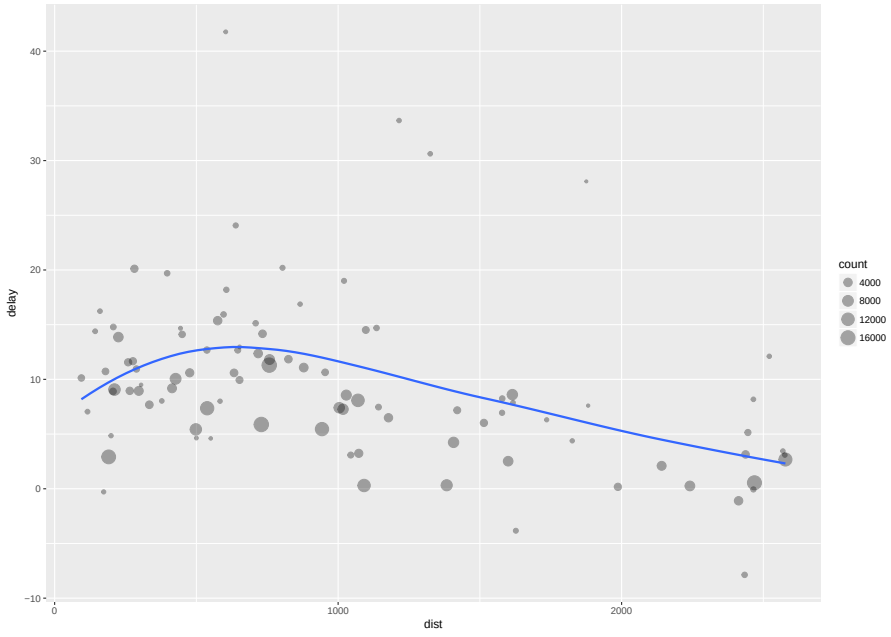
Mediante la sentencia anterior tendríamos el retraso medio (en salidas) de los vuelos, una vez agrupados por días. A veces es interesante poder concatenar operaciones de transformación mediante el operador pipa: % > %. Esto evita tener que estar almacenando resultados intermedios sobre variables específicas. Consideremos que es necesario:

- agrupamiento de los vuelos por destino
- obtener información de resumen para cada destino, contando el número de vuelos para cada uno de ellos y determinando la distancia media y el retraso medio en llegadas
- sobre la información de resumen quedarse sólo con aquellos destinos en que hubiese más de 20 vuelos y evitando Honolulu (**HNL**)
- visualizar los datos resultantes

Todas estas operaciones, realizadas paso a paso serían:

```
# posibilidad de agrupar operaciones mediante pipas
#paso 1: agrupamiento de instancias por destino
porDestino <- group_by(flights,dest)

#paso 2: se determina el retraso por destino
retraso <- summarize(porDestino, count=n(), dist=mean(distance, na.rm=
    TRUE),
                     delay=mean(arr_delay, na.rm=TRUE))

#paso 3: filtrado de instancias: nos quedamos con aquellas en que al
    menos
# hay 20 instancias y el destino no es HNL
```

```
filtrado <- filter(retraso, count > 20, dest != "HNL")

# se representan los datos
ggplot(data=filtrado, mapping=aes(x=dist,y=delay)) +
  geom_point(aes(size=count), alpha=1/3) +
  geom_smooth(se=FALSE)
```



Todas estas operaciones se pueden encadenar de forma sencilla:

```
# Estas operaciones se puden encadenar mediante pipas
resultado <- flights %>%
             group_by(dest) %>%
             summarize(count=n(), dist=mean(distance, na.rm=TRUE),
                       delay=mean(arr_delay, na.rm=TRUE)) %>%
             filter(count > 20, dest != "HNL")

# se pinta el resultado para ver que coincide
ggplot(data=resultado, mapping=aes(x=dist,y=delay)) +
  geom_point(aes(size=count), alpha=1/3) +
  geom_smooth(se=FALSE)
```

Incluso es posible trabajar directamente mediante pipas con la idea de realizar la representación gráfica de los datos de interés:

```
# incluso puede trabajarse directamante con el objetivo de realizar
# la representacion
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year, month, day) %>%
  group_by(tailnum) %>%
  summarize(delay=mean(arr_delay)) %>%
  ggplot(mapping=aes(x=delay)) +
        geom_freqpoly(binwidth=10)
```

Como ejercicio para practicar usaremos el conjunto de datos **Lahman::Batting** sobre *baseball*. Las variables de interés para el ejercicio son: H, que indica el número de éxito al batear y AB, que representa el número de veces que batearon (las oportunidades de bateo). Así, por ejemplo, el jugador con identificador **addybo01** tuvo la posibilidad de batear 118 veces (variable AB), de las cuales tuvo éxito sólo en 32 (variable H). Se pretende:

- convertir el conjunto de datos en **tibble**, mediante la función **as_tibble**
- agrupar los datos por bateador (variable **playerID**)
- obtener un nuevo conjunto de datos mediante **summarize**, incluyendo las variables **golpeo** y **posibilidades**. La primera de ellas se obtiene sumando los valores de la variable **H** de todos los registros del bateador (eliminando los posibles valores perdidos) y dividiendo por la suma de los valores de la variable **AB** de los registros del bateador (eliminando también posibles registros con valores perdidos). La variable **posibilidades** se obtiene simplemente como suma de los valores de la variable **AB** de todos los registros del bateador (eliminando, como antes, registros con valores perdidos)
- filtrado de aquellos datos de bateadores con número de posibilidades de bateo por encima de 100
- representación de los datos representando en el eje X el valor de **posibilidades** y en el eje Y el valor de **golpeos**, mediante geometría de puntos y agregando también línea de tendencia de datos.
- reordenación de las filas del conjunto de datos de forma que aparenzcan en primer lugar las correspondientes a bateadores con mayor valor de la variable **golpeo**. Esta tarea se realizará usando la función **arrange** en combinación con la función **desc**.

5. Visión preliminar de los datos

A la hora de trabajar con un conjunto de datos quizás el primer paso consista en conocer sus características más básicas: cuántas variables posee, cuántas instancias contiene; es decir, el número de columnas y filas respectivamente. Para ello ejecutamos las siguientes sentencias, contenidas en el archivo **visionPreliminar.R**:

```
# se determina la dimension del conjunto de datos: numero de
# filas y de columnas: de instancias y de variables
instancias <- nrow(datos)
variables <- ncol(datos)
```

Esto nos permite ver que el conjunto de datos cuenta con 51 variables y 20000 instancias. La función **head** permite visualizar la cabecera de la tabla, con los nombres de todas las variables y sus valores para las primeras instancias (al ser información en modo texto no es de mucha ayuda si hay muchas variables):

```
# se muestra la cabecera del conjunto de datos
head(datos)
```

Puede obtenerse un listado con los nombres de todas las variables mediante la sentencia **names**:

```
# se muestra la lista de nombres de variables del conjunto de datos
names(datos)
```

Se aprecia la existencia de un nombre especial **class** para la última variable, lo que nos indica que estamos tratando con un conjunto de datos etiquetados. También es interesante obtener alguna información general sobre los datos. Esto podemos obtenerlo de diferentes modos:

```
# muestra informacion de resumen sobre el conjunto de datos, sobre
# todas sus variables
summary(datos)
```

Esta función nos permite saber si una variable es categórica o continua. Para las primeras se muestran sus posibles valores, junto con el número de instancias en que aparece cada uno de ellos. En el conjunto de datos considerado la primera variable discreta que aparece es **PredSS_r1_.1**, para la que se muestra la siguiente información:

```
PredSS_r1_.1
C    :9556
E    :4625
H    :5434
X    : 185
NA's: 200
```

Así podemos conocer que se observan cuatro posibles valores para la variable, siendo *C* el más frecuente, habiendo 9556 instancias con este valor. De igual manera se muestra la existencia de 200 instancias para las que no se ha registrado el valor para ella (datos perdidos; se hablará de este problema más adelante).

Para las variables continuas la sentencia anterior nos indica: mínimo, primer cuartil, mediana, media, tercer cuartil y máximo. La primera variable, **separation**, da lugar a la siguiente información:

```
  separation
  Min.    :   6.00
  1st Qu.:  26.00
  Median :  57.00
  Mean    :  93.67
  3rd Qu.: 114.00
  Max.    :1221.00
```

También podemos estar interesados en ver la información de resumen de una o varias variables específicas, no de todas. Para ello debemos tener claro cómo seleccionar filas y columnas de un conjunto de datos. La selección de un cierto número de filas se hace especificando un valor concreto o bien un rango de valores. Y si deseamos obtener todas las columnas para esas filas, en la segunda dimensión no hace falta indicar nada. Así:

```
datos[1:3,]
```

selecciona de forma completa las tres primeras instancias (con todas sus variables). También es posible seleccionar varias columnas completas:

```
datos[,1:3]
```

La sentencia anterior selecciona las tres primeras variables del conjunto de datos (con sus valores para todas las instancias). Con esta operación puedo elegir varias variables para la información de resumen. Si estamos interesados en las tres primeras variables haremos:

```
# muestra informacion de resumen de las tres primeras variables
summary(datos[,1:3])
```

y se obtendría la siguiente información:

```
   separation        propensity          length
 Min.   :   6.00   Min.   :-1.88816   Min.   :  50.0
 1st Qu.:  26.00   1st Qu.:-0.51366   1st Qu.: 144.0
 Median :  57.00   Median :-0.16855   Median : 209.0
 Mean   :  93.67   Mean   :-0.07537   Mean   : 286.4
 3rd Qu.: 114.00   3rd Qu.: 0.21625   3rd Qu.: 356.0
 Max.   :1221.00   Max.   : 2.63886   Max.   :1244.0
```

Otra función interesante es **str**, que ofrece básicamente la misma información, pero con un formato quizás más legible:

```
# otra funcion interesante es str, que muestra informacion sobre las
# variables de una forma mas compacta, y permite determinar si una
# variable es discreta, continua de forma sencilla
str(datos)
```

A veces puede ser interesante ver los datos con un formato de tabla bien organizada, con separaciones entre celdas, e incluso dando la posibilidad de editar (modificar) algún valor concreto:

```
# tambien es posible visualizar los datos en forma de tabla (e incluso
# editarlos)
fix(datos)
```

Hay algunos paquetes que tienen versiones más refinadas de la función de resumen. El paquete **Hmisc** ofrece la función **describe**, que muestra información detallada sobre una variable o conjunto de ellas. Esta función espera recibir el índice que ocupa dicha variable (o un rango de ellas) en el conjunto de datos. El examen de la primera variable se haría de la siguiente forma, donde se muestra también la salida que se genera:

```
# se muestra informacion sobre la primera variable. Observad que
# la funcion entiende que el indice no se refiere a una fila sino
# a una columna
describe(datos[1])

 1  Variables      20000  Observations
--------------------------------------------------------------------------------
separation
       n  missing   unique     Info     Mean      .05      .10      .25      .50
                    .75      .90      .95
   20000        0      706        1    93.67      9.0     13.0     26.0     57.0
           114.0    210.1    308.0

lowest :    6    7    8    9   10, highest: 1098 1142 1165 1166 1221
--------------------------------------------------------------------------------
```

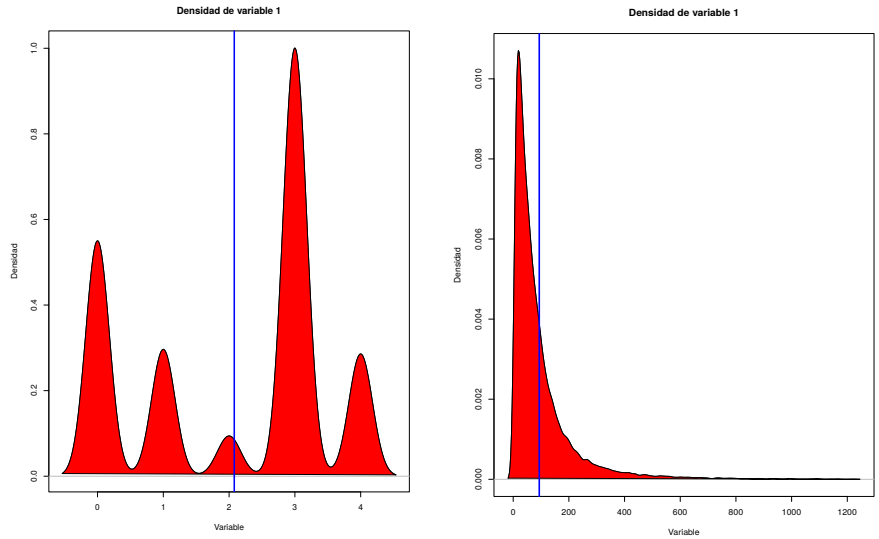
Para la variable clase (categórica) muestra:

```
# se muestra informacion apra la variable class (categorica)
describe(datos[51])

 1  Variables      20000  Observations
--------------------------------------------------------------------------------
class
       n missing  unique
   20000       0       2

negative (14010, 70\%), positive (5990, 30\%)
--------------------------------------------------------------------------------
```

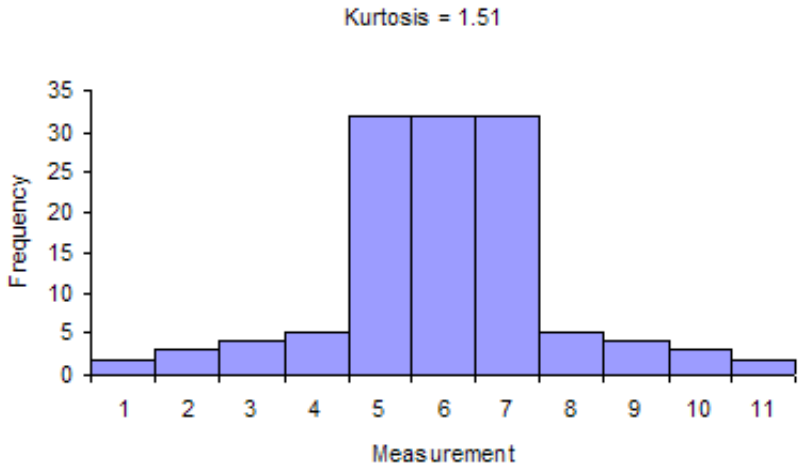
La función **basicStats** del paquete **fBasics** ofrece todavía información mas detallada para las variables continuas. Por ejemplo, sobre la primera variable del conjunto de datos usado obtenemos:

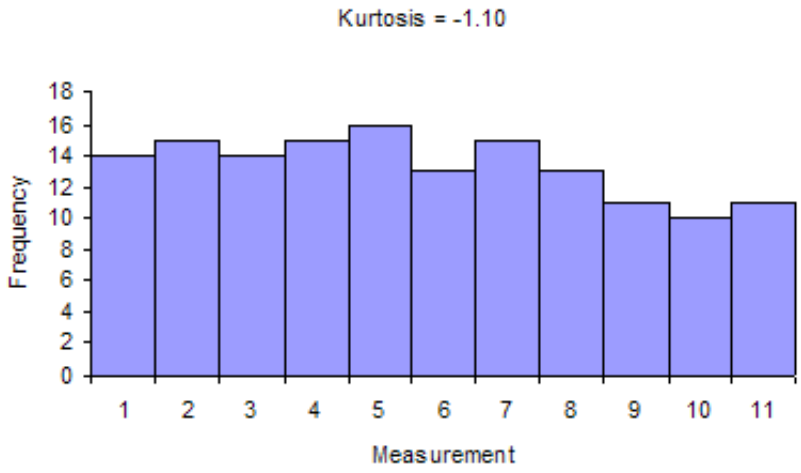
```
# informacion detallada sobre la distribucion de valores de
# la variable separation
basicStats(datos[1])

              separation
nobs        2.000000e+04
NAs         0.000000e+00
Minimum     6.000000e+00
Maximum     1.221000e+03
1. Quartile 2.600000e+01
3. Quartile 1.140000e+02
Mean        9.367455e+01
Median      5.700000e+01
Sum         1.873491e+06
SE Mean     8.048220e-01
LCL Mean    9.209703e+01
UCL Mean    9.525207e+01
Variance    1.295476e+04
Stdev       1.138190e+02
Skewness    3.225102e+00
Kurtosis    1.482742e+01
```

Esta función no puede utilizarse sobre variables discretas. A continuación se incluyen algunos gráficos que nos permiten ilustrar los concepto de **skewness** y **kurtosis** (con valor positivo y negativo) en relación a la distribución de aparición de valores de una variable.



-0.37 3,22





Así, un valor de **skewness** positivo indica que la cola de la derecha es mayor, de forma que la masa de la distribución se concentra a la izquierda. Por su parte, un alto valor

de **kurtosis** indicaría que hay un pico en la distribución de valores (muchas instancias repitiendo un conjunto limitado de valores).

6. Imputación de valores perdidos

Una vez cargado el conjunto de datos y tras una primera inspección es el momento de empezar a tratarlo. Un problema en los datos suele ser la existencia de valores perdidos. Se trata de valores que no están disponibles para algunas variables en alguna instancia. En general, se distingue entre dos tipos de situaciones:

- datos perdidos completamente aleatorios (**MCAR: missing completely at random**)
- datos perdidos de forma no completamente aleatoria (**MNAR**: **missing not at random**)

En el segundo caso, aunque a veces no conoceremos los datos, resulta interesante intentar analizar la razón de esta pérdida, porque puede ser informativa. Supongamos el caso de opiniones de clientes sobre diferentes productos. Es claro que los clientes estarán más dispuestos a valorar productos respecto a los que tienen una clara opinión, ya sea a favor o en contra. En este caso se aprecia una relación entre la percepción de los productos y la ausencia de información.

También conviene distinguir claramente entre **datos perdidos** y **datos censurados**. En este último caso se trata de valores no disponibles, pero de los que se cuenta con alguna información. Por ejemplo, en un sistema de alquiler de películas la falta de valor en el atributo fecha de entrega indica que la película no se devolvió. El modo de tratamiento en ambos casos varía y, por si mismo, cada uno de ellos, constituye un área de investigación activa en ciencia de datos.

La mayoría de las veces los valores perdidos suelen estar relacionados con un subconjunto de variables predictoras (y no distribuidos aleatoriamente por todas ellas). En cualquier caso, en el tratamiento de datos perdidos pueden distinguirse varias aproximaciones básicas:

- usar algunas técnicas de aprendizaje que sean capaces de solucionar este problema. Por ejemplo, los métodos de clasificación basados en árbol pueden prescindir de algunos atributos (al no considerarse relevantes), de forma que los valores perdidos en ellos no afectan en modo alguno al modelo aprendido.
- eliminar directamente instancias que contengan muchos valores perdidos. Suele fijarse un porcentaje sobre el número de atributos del problema y se descartan aquellas instancias en que el número de valores perdidos supere el porcentaje umbral. Si el conjunto de datos es grande y el número de instancias en esta situación no es muy elevado es una opción que produce pérdida de información excesiva.
- asignar valores utilizando alguna técnica concreta. Esta aproximación se conoce como **imputación**. Se trata de aplicar técnicas que intentan adivinar el valor perdido teniendo en cuenta los datos disponibles. Hay a su vez diferentes alternativas:
  - asignar un valor fijo a todos los valores perdidos de todas las variables.
  - asignar a cada valor perdido un valor de referencia para la variable correspondiente (media, mediana, etc).

- usar técnicas de imputación más sofisticadas. Algunas, por ejemplo, se basan en considerar las instancias más parecidas a aquella con valores perdidos y asignar los valores perdidos a partir de ellas (si se consideran varias de las más parecidas se usa un sistema de votación, por ejemplo por mayoría).

Con las funciones vistas hasta ahora ya tenemos alguna información sobre datos perdidos: es uno de los datos mostrados por la función **describe** del paquete **Hmisc**. Si describimos la información sobre la cuarta variable del conjunto de datos usado tenemos:

```
describe(data[4])

 1  Variables      20000  Observations
---------------------------------------------------------------------------
PredSS_r1_.1
       n  missing distinct
   19800      200        4

Value            C     E     H     X
Frequency     9556  4625  5434   185
Proportion   0.483 0.234 0.274 0.009
---------------------------------------------------------------------------
>
```

donde se observa que la variable de interés presenta valores perdidos en 200 instancias. Veremos aquí algunas operaciones relacionadas con el tratamiento de los datos perdidos.

6.1. Filtrado de instancias con cierto porcentaje de datos perdidos

La siguiente secuencia de acciones determina el porcentaje de variables con valores perdidos en una instancia. Este porcentaje sirve para descartar instancias en que el porcentaje supera un cierto límite (el archivo se llama **filtrado.R**):

```
library(parallel)

# se carga el archivo con las funcione de lectura de datos
source("lecturaDatos.R")

# se fija la ruta donde se encuentran los datos
path <- "./data/"
file <- "datos.csv"

# lectura de los datos
datos <- lecturaDatos(path,file)

# se obtiene el porcentaje de valores perdidos de cada fila. La
# siguiente sentencia aplica una funcion al conjunto completo de
# datos, sumando todos los valores perdidos de cada instancia. El
# resultado de la funcion (antes de la division) sera una lista
# con una entrada para cada instancia: el numero de datos perdidos
# presentes en ella. La division y la multiplicacion por 100 se
# usan para determinar el porcentaje de datos perdidos (100 por
# cien si todos los valores de una instancia fueran perdidos).
# El valor 1 del segundo argumento indica que la funcion se aplicara
# sobre todas las filas
system.time(res1 <- apply(datos, 1, function(x) sum(is.na(x))) / ncol
    (datos) * 100)
```

```
24
25 # posibilidad de ejecutar esta operacion en parelelo
26 cores <- detectCores()
27 cluster <- makeCluster(cores-2)
28 system.time(res2 <- parRapply(cluster, datos, function(x) sum(is.na(x
       )))/ncol(datos)*100)
29 names(res2) <- NULL
30 stopCluster(cluster)
31
32 # se marcan aquellas filas con valor de porcentaje mayor de 5. Como
33 # hay 51 variables, equivale a instancias en que hay 3 variables sin
34 # valor (redondeando). mal es una lista de indices de instancias en
35 # que se cumple la condicion indicada
36 mal <- (res1 > 5)
37
38 # datos filtrados: se quitan las filas con muchos valores perdidos
39 filtrados <- datos[!mal,]
40
41 # se muestra el numero de filas del conjunto inicial y del resultante
42 cat("Instancias archivo original: ",nrow(datos)," instancias en
       filtrado: ",nrow(filtrados),"\n")
43
44 # se guardan los datos filtrados
45 escrituraDatos(path,"datosFiltrados.csv",filtrados)
```

6.2. Imputación de valores perdidos con paquete mice

Este paquete puede usarse para imputar valores perdidos sobre variables categóricas y continuas, usando un algoritmo específico de imputación conocido con el mismo nombre (**MICE**). Este paquete ofrece mucha funcionalidad muy útil que puede ayudarnos a comprender la forma en que aparecen los valores perdidos en los datos de interés. Por ejemplo, podemos obtener un patrón de aparición de forma sencilla. Ya que el conjunto de datos base usado antes contiene muchas variables, resulta más sencillo usar un conjunto de datos más simple para que la visualización de resultados sea más cómoda: **airquality**. Este conjunto de datos se define sobre 6 variables (**Ozone**, **Solar.R**, **Wind**, **Temp**, **Month** y **Day**), con 153 instancias. El patrón de datos perdidos se genera con el archivo **patronMissingMice.R**. Se fuerza la existencia de algunos datos perdidos más de los presentes en el conjunto de datos en sí para que la aplicación de estas funciones sea más representativa

```
1 # Este paquete puede usarse para imputar valores perdidos en
2 # variables de todo tipo
3 library(mice)
4
5 # carga de datos disponibles directamente en el paquete mice
6 datos <- airquality
7
8 # se obtiene el patron de aparicion de datos perdidos
9 patron <- mice::md.pattern(x=datos)
10
11 # se muestra el patron
12 print(patron)
```

Al ejecutar esta sentencia aparece el siguiente resultado:

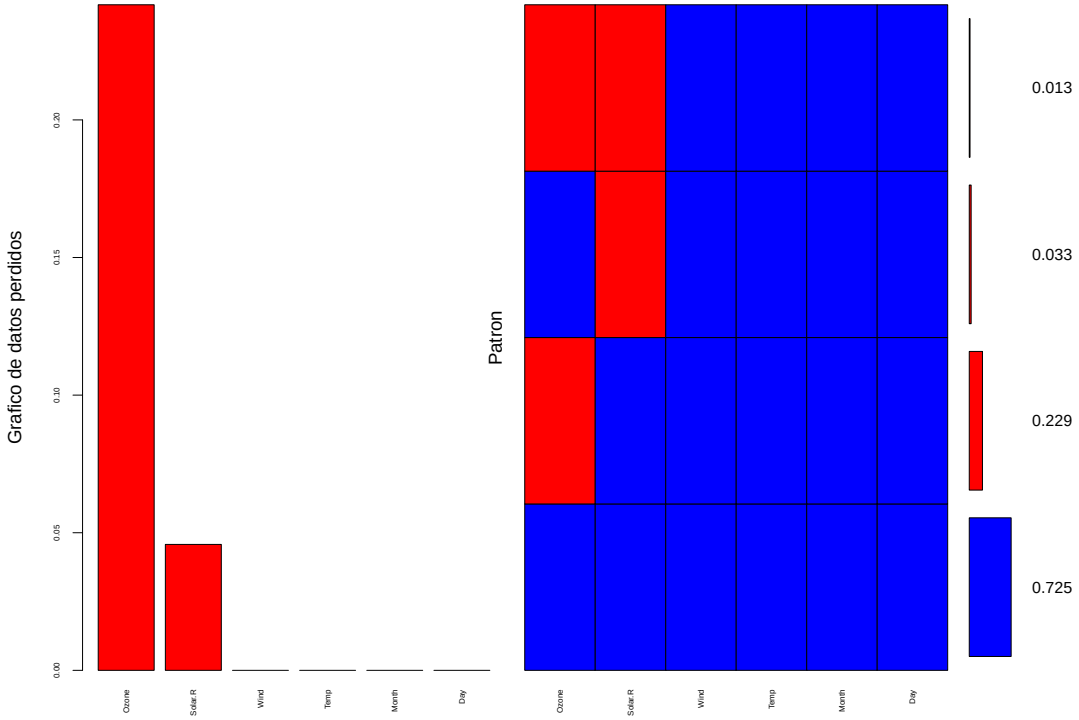
```
    Wind Temp Month Day Solar.R Ozone
111    1    1     1   1       1     1  0
 35    1    1     1   1       1     0  1
  5    1    1     1   1       0     1  1
  2    1    1     1   1       0     0  2
       0    0     0   0       7    37 44
```

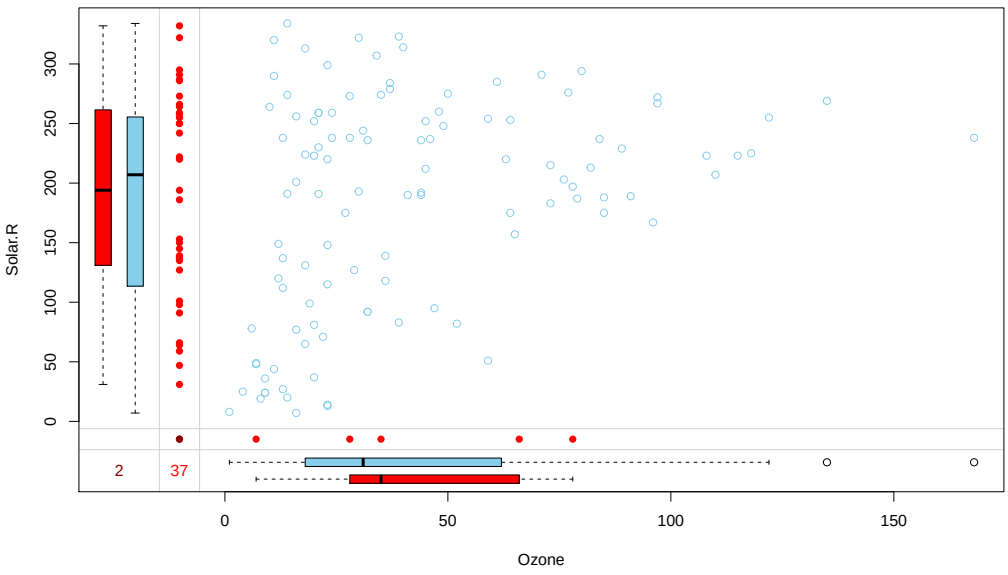
Esto indica que hay 111 instancias donde no hay ningún valor perdido, 35 instancias donde está ausente el valor de la variable **Ozone**, 5 en que el valor ausente aparece en la variable **Solar.R** y 2 instancias en que no se dispone de valor de ninguna de estas dos variables. El dígito 0 indican ausencia de valor.

De esta forma, la fila inferior ofrece alguna información resumen sobre el número de valores perdidos para cada variable: 7 instancias donde no se dispone de valor para **Solar.R** y 37 con ausencia de valor para **Ozone**.

Hay formas más visuales de obtener esta información, mediante el paquete **VIM**. Un ejemplo de uso está presente el el archivo **patronMissingVIM.R**:

```
# Referencias a los paquetes necesarios para disponer
# del conjunto de datos y de la funcion de visualizacion
# del patron de datos perdidos
require(VIM)

# se usan datos disponibles en este paquete, sobre calidad
# del aire: 6 variables y 153 instancias. Se visualiza un
# grafico que nos indica la forma en que se distribuyen los
# datos perdidos
datos <- airquality

# se genera el grafico de distribucion de datos perdidos. Solo
# se consideran las variables con datos perdidos
plot <- VIM::aggr(datos, col=c('blue','red'), numbers=TRUE,
                  sortVars=TRUE, labels=names(data), cex.axis=.5,
                  gap=1, ylab=c("Grafico de datos perdidos","Patron"))

# se muestra informacion sobre algunas de las variables en que
# aparecen los datos perdidos
VIM::marginplot(datos[,1:2])
```





Es obvio añadir que este tipo de gráficos sólo pueden usarse en el caso de conjuntos de datos pequeños. En este último gráfico la barra en rojo de la zona izquierda muestra la distribución de **Solar.R** cuando no se encuentra el valor de **Ozone**, mientras que la caja azul indica la distribución de valores de esta misma variable para instancias

donde sí se conoce el valor de **Ozone**. Igual se interpretan las cajas que se muestran en el eje horizontal para la variable **Ozone**. Si los valores perdidos aparecen de forma completamente aleatoria sería de esperar que ambas cajas fuesen similares, indicando que no hay dependencia mutua entre la ausencia de valor de una variable y los valores de la otra.

Una vez obtenida alguna información sobre el patrón asociado a los datos perdidos puede optarse por imputar sus valores (por ejemplo, esta será la aproximación casi única en caso de disponer de pocos datos). La imputación de los datos se realiza como se muestra en el siguiente fragmento de código (archivo **imputacionMice.R**). Se usa el método de imputación por defecto (aunque es posible seleccionar un método diferente mediante el parámetro **meth**):

```
# Este paquete puede usarse para imputar valores perdidos en
# variables de todo tipo
library(mice)
library(lattice)

# se usa el conjunto airquality
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n
    ")

# se realiza la imputacion
imputados <- mice::mice(datos, m=5, meth="pmm")

# dispone de algunos metodos que imputan siempre a un unico
# valor, como por ejemplo "mean"
imputadosMean <- mice::mice(datos, m=1, meth="mean")

# pmm es el metodo por defecto. Puedes verse todos los metodos
# disponibles de la siguiente forma
methods(mice)

# se completa el conjunto de datos con las imputaciones
datosImputados <- mice::complete(imputados)

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos en la parte ya limpia
completos <- mice::ncc(datosImputados)
incompletos <- mice::nic(datosImputados)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n
    ")

# se muestra la imputacion para Ozone
imputados$imp$Ozone

# Se muestra un grafico para comprobar la distribucion de Ozone en
    los
# datos imputados en relacion a otras variables. Los puntos en azul
# representan datos observados y datos en rojo representan
```

```
    imputaciones
lattice::xyplot(imputados, Ozone ~ Solar.R, pch=18, cex=1)

# Se muestran las densidades de los datos imputados respecto de los
# observados
lattice::densityplot(imputados)

# Se muestran los diagramas de caja para las imputaciones
lattice::bwplot(imputados)
```

Es posible ver los valores imputados de la siguiente forma:

```
# muestras las imputaciones para la variable Ozone
imputados$imp$Ozone
```

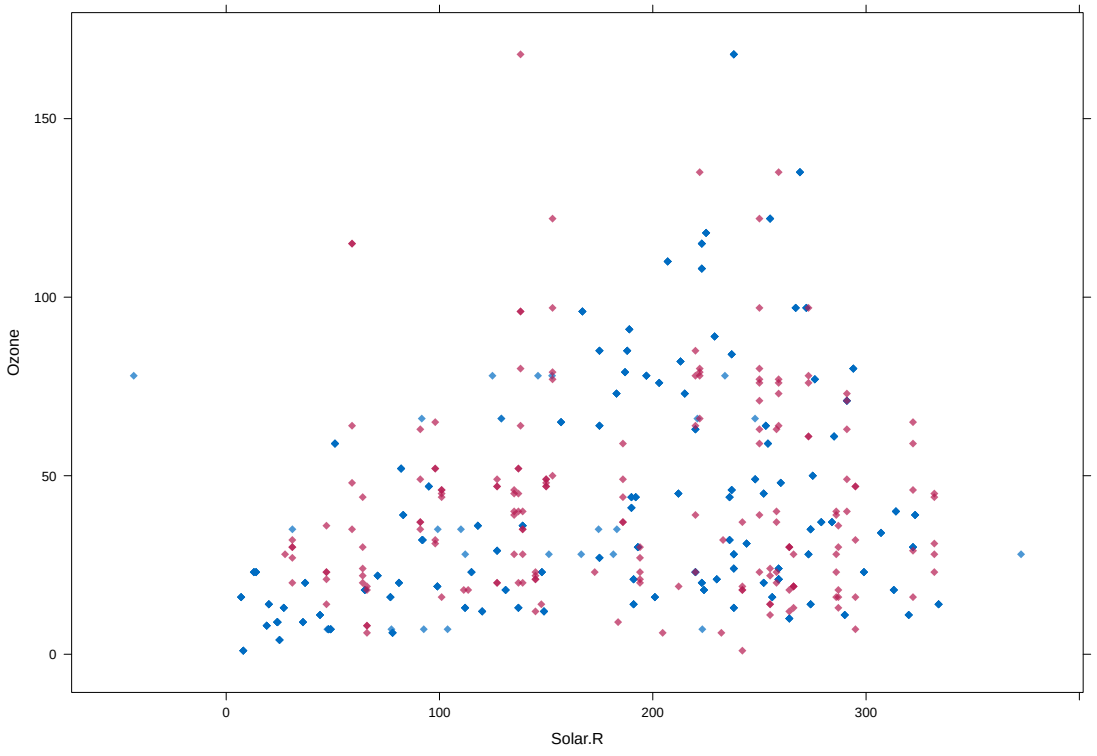
Esta sentencia produce la siguiente salida:

```
      1  2   3   4  5
5    18 19   6  14  6
10   23 21  27  20 30
25    8 19  18   6  8
26   19 19  19  13 28
27   18 28  23   9 32
32   23 39  16  40 28
33   36 30  16  18 13
34   18 19  37   1 18
35   37 44  49  59 37
36   85 23  78  64 39
37   30 12  30  30 18
39   97 61  61  76 78
42   76 64 135  77 73
43   76 80  77 122 97
45   23 44  31  28 45
46   29 59  65  16 46
52   49 47  49  48 47
53  115 35  64 115 48
54   49 63  37  37 35
55   23 71  63  39 59
56   45 39  46  40 28
57   20 20  47  49 47
58   21 14  23  23 36
59   31 65  52  52 32
60   20 32  30  30 27
61   96 64  96 168 80
65   46 16  46  45 44
72   40 28  35  20 35
75   49 40  63  71 73
83   20 63  40  37 23
84   32 16   7  47 47
102  78 80 135  79 66
103  52 45  40  52 20
107  24 44  20  22 30
115  14 11  24  22 14
119  50 77 122  97 79
150  21 21  22  23 12
```

La interpretación de estos resultado precisa saber que el método de imputación se basa en generar conjuntos de datos aleatorios (en diferentes particiones, siendo 5 el valor por defecto) de forma que las imputaciones se van realizando sobre ellas. El resultado de la imputación se usa posteriormente para completar los datos perdidos en la muestra original, tal y como se ha indicado en el archivo previo, mediante el uso de la función **complete**. Esta función permite especificar el conjunto de datos a usar para la imputación, siendo 1 el valor por defecto.

Otro gráfico interesante permite visualizar la distribución de los valores imputados en relación a otras variables. Esto permite chequear si las imputaciones son plausibles o no.

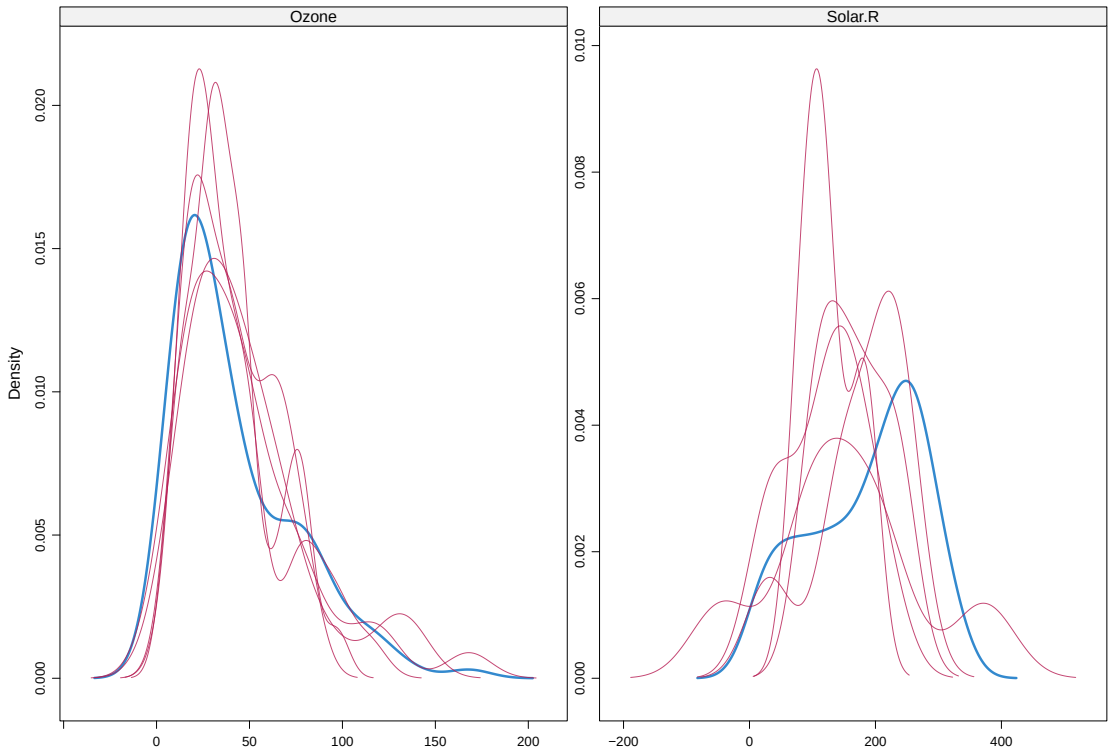
```
xyplot(imputados,Ozone ~ Wind+Temp+Solar.R,pch=18,cex=1)
```



El resultado de una imputación correcta debería mostrar que los puntos en rojo se distribuyen de forma similar a como lo hacen los puntos azules. Los primeros representan los valores imputados y los azules los observados, de forma que cada instancia daría lugar a un punto en el gráfico.

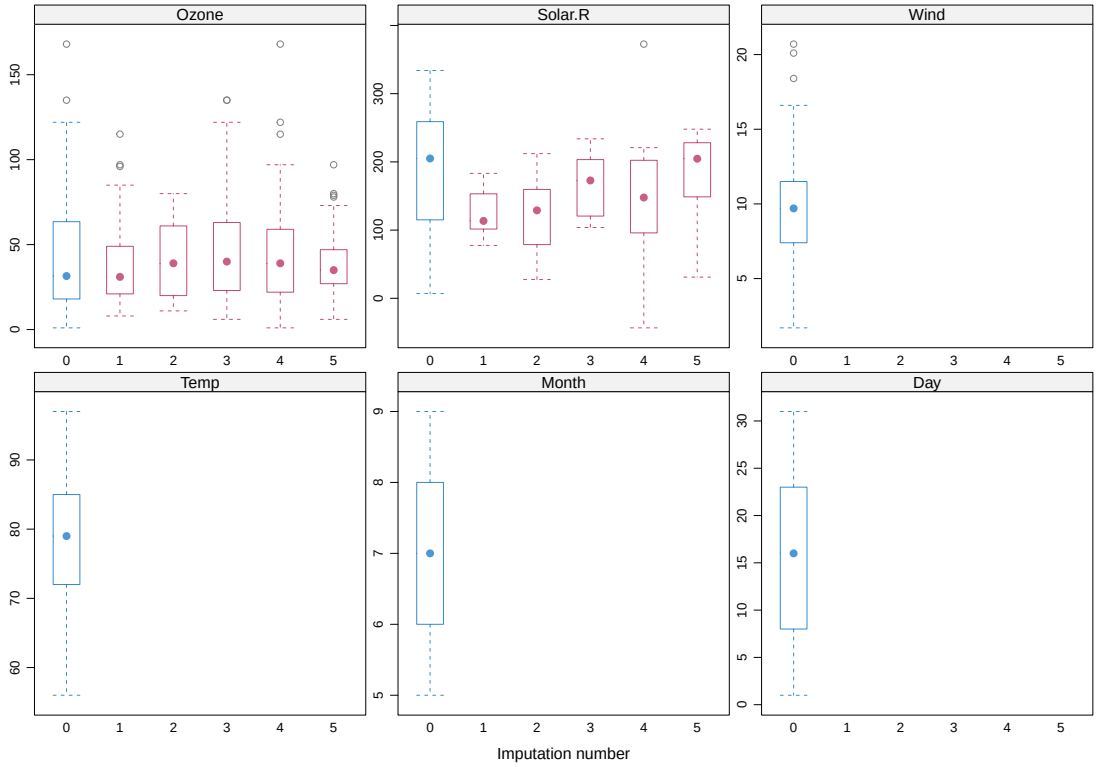
Otro gráfico de interés permite visualizar la densidad de los datos imputados en cada una de las particiones del conjunto de datos mencionadas con anterioridad. Estas distribuciones de densidad se muestran en rojo, mientras que la densidad de los datos observados se muestra en azul. Al igual que se comentó con anterioridad, la suposición de datos distribuidos aleatoriamente debería conducir a densidades similares:

```
densityplot(imputados)
```



Existen diferentes tipos de gráficos que permiten visualizar la imputación de diferentes formas, como por ejemplo diagramas de cajas. La caja a la izquierda, en azul, representa la distribución de valores de la variable de interés en los datos observados; el resto de cajas las distribuciones de valores tras la imputación en cada una de las particiones.

```
1 bwplot(imputados)
```



La caja a la izquierda, en azul, representa la distribución de valores de la variable de interés en los datos observados; el resto de cajas las distribuciones de valores imputados en cada una de las particiones.

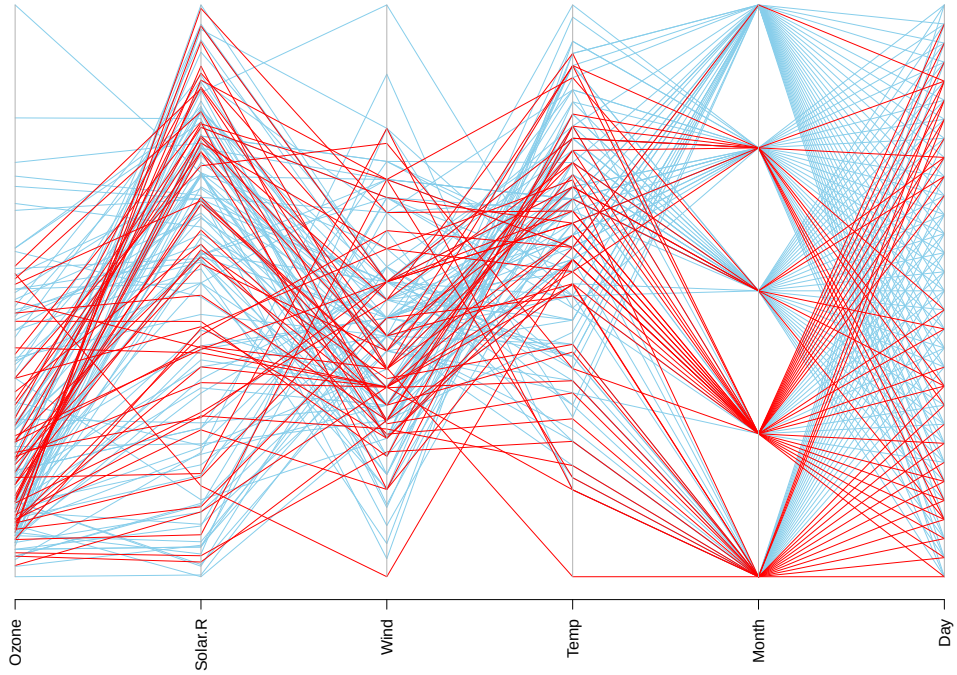
6.3. Imputación de valores perdidos con paquete robCompositions

Este paquete permite hacer la imputación de valores perdidos con diferentes técnicas y resulta muy sencillo de aplicar, aunque sólo puede usarse para imputar valores perdidos en variables continuas (fichero **imputacionRobCompositions.R**). Este paquete también ofrece diferentes gráficos que permiten obtener información visual sobre la forma en que se hace la asignación de valores perdidos.

```
require(robCompositions)
require(mice)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n
    ")

# se hace la imputacion
imputados <- robCompositions::impKNNa(datos, primitive=TRUE)

# Ahora puede visualizarse alguna informacion sobre la forma
```

```
# en que se hizo la imputacion. El segundo argumento indica el
# tipo de grafico a obtener
plot(imputados, which=2)

# El conjunto de datos completo puede accederse de la siguiente forma
imputados$xImp
```

El siguiente gráfico muestra las instancias como líneas (en azul las completas y en rojo aquellas en que se hizo imputación). Permite comprobar de forma visual si alguna instancia imputada se sale completamente del patrón general o todas ellas se comportan de la forma esperada.



Es importante resaltar que el tipo de datos a tratar (supervisado / no supervisado) puede servir para determinar el tipo de técnica a aplicar. En el caso supervisado puede ser interesante considerar la relación existente entre la variable a imputar y la variable clase, de forma que esta relación sea determinante a la hora de hacer la asignación de valor.

7. Detección de datos anómalos

A veces en los datos se presentan valores anómalos, que se han introducido, por ejemplo, debido a errores en los procesos de recogida de datos. El tratamiento de esta situación también resulta problemático: quizás el valor anómalo se deba a una deformación en la distribución de valores y no a un error.

Hay que tener en cuenta que algunos métodos de aprendizaje son robustos frente a este problema. Por ejemplo, los árboles de clasificación que pueden tratar con variables continuas establecen particiones en los dominios teniendo en cuenta valores límite (si el

valor de la variables es mayor de...). En este caso la existencia de un valor anómalo no tendría demasiada influencia. Igual ocurre con las máquinas de soporte vectorial, que suelen descartar un conjunto de instancias al crear el modelo predictivo (usualmente aquellas lejos de la frontera de decisión, lo que suele ocurrir en el caso de existencia de valores anómalos).

La intuición básica en las técnicas de detección de anomalías consiste en:

- asumir que hay un modelo *generador* de los datos (por ejemplo, una distribución normal) que permite explicar su distribución
- las anomalías representan entonces un modelo generador distinto, que no coincide con el general

Al igual que en el caso de la imputación de valores perdidos, existen técnicas específicas de detección de anomalías que consideran de forma especialmente relevante la relación entre la variable en que aparece el valor anómalo y la variable clase.

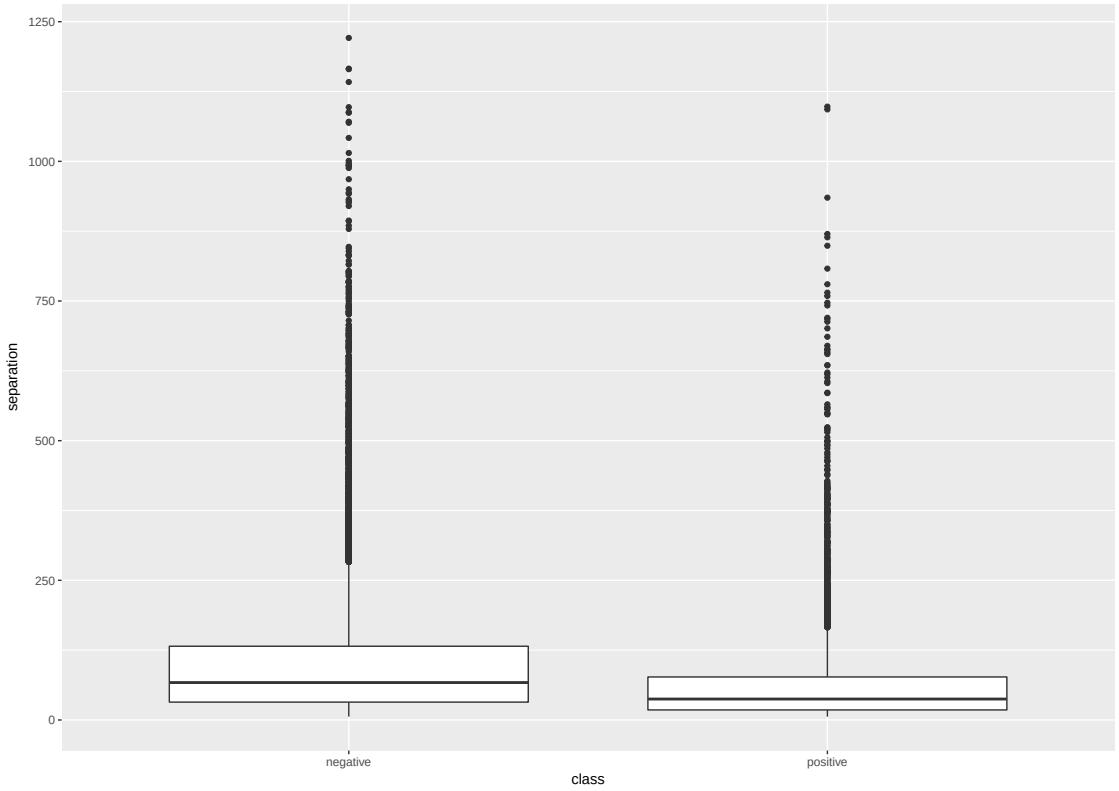
7.1. Paquete outliers

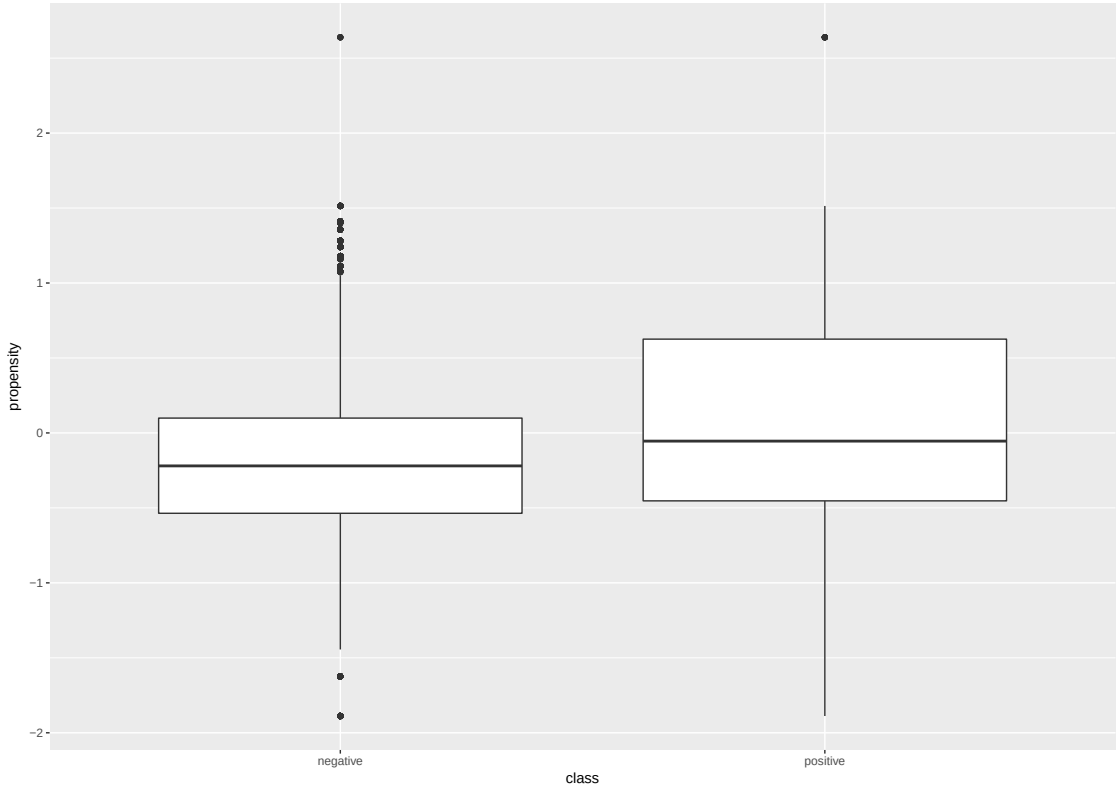
El paquete **outliers** ofrece funcionalidad para este trabajo. Si se usa sobre el conjunto de datos de mayor tamaño que hemos considerado se aprecia que se detectan valores anómalos para las tres primeras variables (el código se encuentra en **anomalias.R**):

```
library(outliers)
library(ggplot2)

# se carga el archivo con las funcione de lectura de datos
source("lecturaDatos.R")

path <- "./data/"
file <- "datos.csv"

# lectura de los datos
datos <- lecturaDatos(path,file)

# deteccion de anomalias para las variable 1 a 3. Observad
# que no tiene sentido considerar variables de tipo discreto
# en este analisis. La funcion devuelve el valor (o valores)
# considerados anomalos para las variable de interes. Este
# metodo solo considera las desviaciones con respecto a los
# valores de cada variable (no relaciones con otras variables)
anomalos <- outlier(datos[,1:3])
print(anomalos)

# la media de la variable separation es
mean(datos[,"separation"])

# se muestra la distribucion de separation en funcion del valor
# de la variable clase
ggplot(data = datos, aes(class, separation)) +
  geom_boxplot()

# se podria hacer igual con la variable propensity
ggplot(data = datos, aes(class, propensity)) +
  geom_boxplot()
```

y se mostrarían como anómalos los valores 1221, 2,638864 y 1244 para las tres variables seleccionadas. La pregunta ahora consiste en qué hacer con las instancias en que aparecen dichos valores. Si el número de instancias con anomalías no es elevado sería válido eliminarlas. En caso contrario hay que decidir qué acción tomar.

Podemos representar un diagrama de cajas para las variables **separation** y **propensity**:





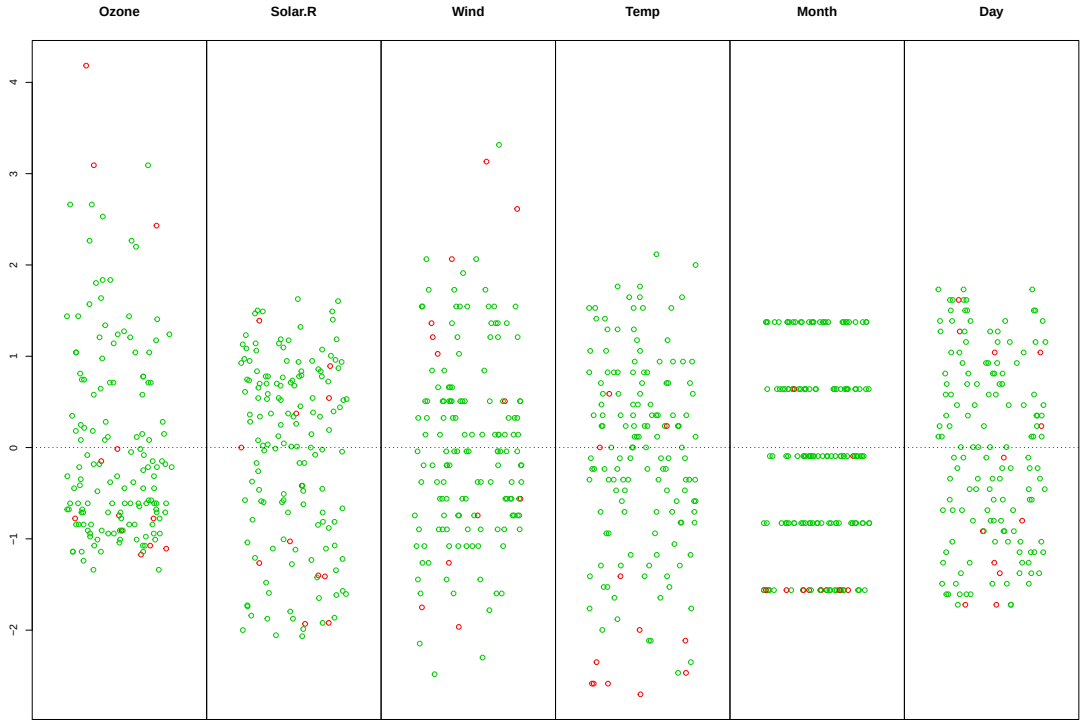
7.2. Paquete mvoutlier

Este paquete ofrece algunas representaciones gráficas interesantes que pueden ser de ayuda en el tratamiento de este tipo de problemas. El archivo **anomaliasMvoutliers.R** ofrece algunos ejemplos de uso:

```
require(mice)
require(mvoutlier)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n
    ")

# se imputan los datos
imputados <- mice::mice(datos)
datos <- mice::complete(imputados)

# se analizan los datos en busca de anomalias. El grafico
# resultante muestra en rojo los datos considerados considerados
# como anomalos
resultados <- mvoutlier::uni.plot(datos)

# a partir de resultado es posible conocer las instancias en que
# aparece algun dato anomalo. Esto podria usarse para filtrar las
```

```
# instancias y quedarnos con aquellas en que no haya anomalias (o
# bien aplicar alguna tecnica para modificar sus valores)
print(resultados$outliers)

# seleccion de instancias sin anomalias
datosFinales <- datos[!resultados$outliers, ]
```

El gráfico generado sería el siguiente:



Este paquete detecta las anomalías analizando las diferencias entre las instancias y la distribución calculada que se considera como generadora de la muestra. Se consideran anómalas aquellas instancias en que la distancia supera un determinado valor. El gráfico muestra en rojo los valores que toma cada una de las variables en aquellas instancias marcadas como contenedoras de datos anómalos. Por eso no nos debe extrañar que haya puntos en rojo cercanos a la línea que representa el valor 0 (distancia 0 con respecto a la distribución generadora).

8. Transformación de los datos

En algunas técnicas de aprendizaje se usan las instancias para calcular medidas de distancia (especialmente en casos de variables continuas). Si algunas variables tienen valores muy altos (en relación a otras), esto hará que la medida de distancia quede sesgada por la medida mayor.

En estos casos resulta conveniente transformar los datos para escalarlos y centrarlos. Una posible forma de centrar los datos consiste en restar a cada valor de cada instancia el valor medio de la variable correspondiente y después dividir los valores por la desviación estándar. Esta transformación centra y escala los valores en torno al cero, en un intervalo

no demasiado grande. El escalado puede hacerse también en torno al valor de la mediana, en lugar de la media.

Otra posibilidad es escalarlos en el intervalo 0-1, para todos los atributos. También aplicar el logaritmo, lo que hace que haya mucha menor dispersión en los datos. En otras ocasiones lo que interesa no es el valor actual de la variable, sino su posición relativa con respecto al resto de valores de otras instancias. El paquete **caret** ofrece funcionalidad para hacer todas estas tareas.

Un ejemplo de uso se puede consultar en el archivo **transformacion.R**:

```
require(mice)
require(caret)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- ncc(datos)
incompletos <- nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n
    ")

# se imputan los datos
imputados <- mice(datos)
datos <- complete(imputados)

# se aplica el centrado y escalado sobre el conjunto de datos,
# una vez eliminados los valores perdidos
valoresPreprocesados <- caret::preProcess(datos[,1:4],method=c("
    center","scale"))

# el resultado consiste en el escalado y centrado de las variables
# de la 1 a la 4 (las que pueden considerarse continuas). El
    resultado
# anterior se usa ahora para asignar a las variables los valores
# correspondientes de acuerdo a esta transformacion
valoresTransformados <- predict(valoresPreprocesados,datos[,1:4])

# y podemos generar un nuevo conjunto de datos con el que
# seguir aplicando tecnicas con las 4 variables transformadas
# y las dos que no se tocaron
datosFinales <- cbind(valoresTransformados,datos[,5:6])
```

9. Discretización

9.1. Paquete discretization

Algunas técnicas de aprendizaje requieren que las variables sean discretas. En el caso en que se desee aprender algún modelo utilizando estas técnicas será preciso transformar las variables continuas en discretas mediante un procedimiento conocido como **discretización**. Básicamente, se trata de dividir el intervalo de posibles valores en un

conjunto de fragmentos y asignar una etiqueta a cada uno de ellos, sustituyendo en el conjunto de datos los valores originales por las etiquetas asignadas a los intervalos.

El paquete **discretization** ofrece diferentes algoritmos y resulta sencillo de usar. Algunos de los algoritmos disponibles son:

- **ameva**: usa una medida basada en el test **chi-cuadrado** para la discretización óptima con el mínimo numero de intervalos y con la mínima pérdida de dependencia (información) con respecto a la variable clase
- **caim (class-attribute interdependence maximization)**: mide la dependencia entre la variable clase y la discretización a emplear para el atributo
- **cacc (class-attribute contingency coefficient)**: calcula la tabla de contingencias entre la variable clase y la variable discretizada

La discretización es muy fácil de realizar (la siguiente secuencia de órdenes está disponible en el archivo **discretizacion.R**). Este ejemplo se ilustra mediante el famoso conjunto de datos **iris**.

```
require(discretization)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- iris

# discretizacion mediante metodo CAIM
cm <- discretization::disc.Topdown(iris, method=1)

# se muestran los puntos de corte
cat("Puntos de corte metodo CAIM: \n")
print(cm$cutp)

# los datos discretizados se mostrarian de la
# forma siguiente
cat("Datos discretizados: \n")
print(cm$Disc.data)

# discretizacion mediante CACC
cmCacc <- disc.Topdown(datos, method=2)

# se muestran los puntos de corte
cat("Puntos de corte metodo CACC: \n")
print(cm$cutp)

# discretizacion mediante AMEVA
cmAmeva <- disc.Topdown(datos, method=3)

# se muestran los puntos de corte
cat("Puntos de corte metodo AMEVA: \n")
print(cm$cutp)
```

Los resultados obtenidos son:

```
Puntos de corte metodo CAIM:
[[1]]
```

```
[1] 4.30 5.55 6.25 7.90

[[2]]
[1] 2.00 2.95 3.05 4.40

[[3]]
[1] 1.00 2.45 4.75 6.90

[[4]]
[1] 0.10 0.80 1.75 2.50

Puntos de corte metodo CACC:
[[1]]
[1] 4.30 5.55 6.25 7.90

[[2]]
[1] 2.00 2.95 3.05 4.40

[[3]]
[1] 1.00 2.45 4.75 6.90

[[4]]
[1] 0.10 0.80 1.75 2.50

Puntos de corte metodo AMEVA:
[[1]]
[1] 4.30 5.55 6.25 7.90

[[2]]
[1] 2.00 2.95 3.05 4.40

[[3]]
[1] 1.00 2.45 4.75 6.90

[[4]]
[1] 0.10 0.80 1.75 2.50
```

Se aprecia que coinciden los puntos de corte obtenidos mediante los diferentes métodos.

10. Selección de características

La selección de características es esencial por varias razones:

- en primer lugar, evita tener que estar tratando con variables que no aportan información para la tarea de aprendizaje. Imaginemos un conjunto de datos en que hay una variable cuyo valor es único para cada instancia. Esta variable poco puede ayudar a construir un modelo a partir de los datos
- los algoritmos de aprendizaje funcionarán mejor (más rápido, con menos memoria) y además serán más fácilmente interpretables al ser más simples
- algunos métodos de aprendizaje son muy sensibles a la presencia de variables no significativas y su rendimiento puede degradarse de forma notable si se cuenta con ellas. En otros, como en los árboles de decisión, hay un proceso de selección que hace que no sea un problema tan grave

La selección de características puede abordarse mediante diferentes aproximaciones:

- **filter**: se obtiene un orden de importancia de los atributos de acuerdo a alguna medida estadística. La salida de los métodos correspondientes a esta aproximación será una lista de atributos y una medida de importancia para cada uno de ellos
- **wrapper**: se prueban modelos con diferentes subconjuntos de atributos. La salida suele ser el subconjunto de atributos que produce el mejor modelo
- **embedded**: consiste en determinar la importancia de los atributos durante el proceso de aprendizaje de un modelo (por ejemplo, mediante las medidas de información de los atributos calculadas durante el proceso de construcción de un árbol de clasificación o un bosque)

Hay muchos paquetes disponibles para hacer selección de características (o reducción de dimensionalidad) en **R** y aquí consideramos sólo algunos de ellos. Algunos de los métodos que vamos a ver se basan realmente en construir modelos sobre los datos y comprobar cómo se ven afectados por la eliminación de alguna variable.

10.1. Paquete FSelector

Las funciones disponibles en este paquete se agrupan de acuerdo a la aproximación a la que pertenece. Se consideran en primer lugar los métodos pertenecientes a la estrategia **filter**.

10.1.1. Aproximación filter: chi.squared

Este método determina los pesos de los atributos discretos usando el test de independencia **chi-cuadrado** (con respecto a la variable clase). Se ve un ejemplo de aplicación (archivo **fSelector-chiSquared.R**):

```
library(FSelector)
library(mlbench)
data(HouseVotes84)

# se calculan los pesos de los atributos: la medida devuelta
# indica el nivel de dependencia de cada atributo frente a la
# variable clase
weights <- FSelector::chi.squared(Class~.,HouseVotes84)
print(weights)

# se seleccionan los 5 mejores
subset <- FSelector::cutoff.k(weights,5)

# se muestran los seleccionados
f <- as.simple.formula(subset,"Class")
print(f)
```

Y el resultado obtenido es:

```
    attr_importance
V1       0.409330348
V2       0.004534049
V3       0.748864321
V4       0.923255954
```

```
V5      0.718768923
V6      0.428332508
V7      0.521967369
V8      0.661876085
V9      0.629797943
V10     0.083809300
V11     0.378240781
V12     0.714922593
V13     0.555971176
V14     0.625283342
V15     0.538263037
V16     0.353273580
Class ~ V4 + V3 + V5 + V12 + V8
```

10.1.2. Aproximación filter: correlation

El algoritmo busca los pesos de atributos continuos en base a medidas de correlación. Hay dos variantes: **linear.correlation** (usando la medida de correlación de **Pearson**) y **rank.correlation** (con medida de **Spearman**). Se incluyen aquí varios ejemplos de uso (**fSelector-correlation.R**):

```
library(FSelector)
data(BostonHousing)

# se dejan unicamente las variables numericas. Asi se filtra una
# variable llamada chas
d=BostonHousing[-4]

# se calculan los pesos mediante correlacion lineal, La variable
# clase se llama medv
weights <- FSelector::linear.correlation(medv~., d)

# se muestran los pesos
print(weights)

# se seleccionan los tres mejores
subset <- FSelector::cutoff.k(weights,3)
f <- as.simple.formula(subset,"medv")
print(f)

# se determinan los pesos mediante rank.correlation
weights <- FSelector::rank.correlation(medv~.,d)

# se muestran los pesos
print(weights)

# se seleccionan los mejores
subset <- FSelector::cutoff.k(weights,3)
f <- as.simple.formula(subset,"medv")
print(f)
```

El resultado de esta ejecución es:

```
        attr_importance
crim          0.3883046
zn            0.3604453
```

```
indus           0.4837252
nox             0.4273208
rm              0.6953599
age             0.3769546
dis             0.2499287
rad             0.3816262
tax             0.4685359
ptratio         0.5077867
b               0.3334608
lstat           0.7376627
medv ~ lstat + rm + ptratio

          attr_importance
crim            0.5588909
zn              0.4381790
indus           0.5782554
nox             0.5626088
rm              0.6335764
age             0.5475617
dis             0.4458569
rad             0.3467763
tax             0.5624106
ptratio         0.5559047
b               0.1856641
lstat           0.8529141
medv ~ lstat + rm + indus
```

10.1.3. Aproximación filter: entropy.based

Este algoritmo encuentra los pesos de los atributos discretos en base a su correlación con el atributo clase. Hay tres formas de uso: **information.gain**, **gain.ratio** y **symmetrical.uncertainty** (con medidas de información diferentes). Observad que se aplica sobre atributos continuos, lo que indica que debe hacer discretización de forma implícita. El ejemplo de aplicación es (**fSelector-entropy.R**):

```
library(FSelector)
data(iris)

# se obtienen las medidas mediante ganancia de informacion
weights <- FSelector::information.gain(Species~., iris)

# se muestran los pesos y se seleccionan los mejores
print(weights)
subset <- FSelector::cutoff.k(weights,2)
f <- as.simple.formula(subset,"Species")
print(f)

# igual, pero con ganancia de informacion
weights <- FSelector::gain.ratio(Species~., iris)
print(weights)

# e igual con symmetrical.uncertainty
weights <- FSelector::symmetrical.uncertainty(Species~., iris)
print(weights)
```

Resultados de aplicación:

```
             attr_importance
Sepal.Length        0.4521286
Sepal.Width         0.2672750
Petal.Length        0.9402853
Petal.Width         0.9554360
Species ~ Petal.Width + Petal.Length
<environment: 0x10583bc98>
             attr_importance
Sepal.Length        0.4196464
Sepal.Width         0.2472972
Petal.Length        0.8584937
Petal.Width         0.8713692
Species ~ Petal.Width + Petal.Length
<environment: 0x107672788>
             attr_importance
Sepal.Length        0.4155563
Sepal.Width         0.2452743
Petal.Length        0.8571872
Petal.Width         0.8705214
Species ~ Petal.Width + Petal.Length
```

10.1.4. Aproximación filter: oneR

Método simple de cálculo de pesos para atributos discretos mediante el uso de reglas de asociación con un sólo término en el antecedente (**fSelector-oneR.R**):

```
library(FSelector)
library(mlbench)

data(HouseVotes84)

# se calculan los pesos
weights <- FSelector::oneR(Class~.,HouseVotes84)

# se muestran los resultados
print(weights)
subset <- FSelector::cutoff.k(weights,5)
f <- as.simple.formula(subset,"Class")
print(f)
```

Resultado de aplicación:

```
    attr_importance
V1       0.25947997
V2       0.09263504
V3       0.60091159
V4       0.77026555
V5       0.55956326
V6       0.29341077
V7       0.37010302
V8       0.50547622
V9       0.47440918
V10      0.08719321
V11      0.24018603
V12      0.56109553
V13      0.39872373
```

```
V14       0.47690848
V15       0.38644105
V16       0.30703305
Class ~ V4 + V3 + V12 + V5 + V8
```

10.1.5. Aproximación filter: relief

Algoritmo de búsqueda de pesos de atributos continuos y discretos en base a la distancia entre instancias (**fSelector-relief.R**). El algoritmo recibe como argumentos:

- fórmula
- datos
- número de vecinos a considerar
- número de instancias a considerar en el cálculo

```
library(FSelector)
library(mlbench)
data(iris)

# se calculan los pesos
weights <- FSelector::relief(Species~.,iris, neighbours.count=5,
    sample.size=20)

# se muestran los resultados
print(weights)
subset <- FSelector::cutoff.k(weights,2)
f <- as.simple.formula(subset,"Species")
print(f)
```

Resultado:

```
Species ~ Petal.Length + Petal.Width
```

Las aproximaciones **wrapper** disponibles se muestran a continuación.

10.1.6. Aproximación wrapper: best.first.search

Algoritmo para búsqueda de subconjuntos de atributos. Su usa de la forma que se muestra en el siguiente ejemplo (sobre conjunto de datos **iris**). Se basa en el uso del paquete **rpart** que permite construir árboles de regresión y de clasificación (el script de uso es **fSelector-bestFirstSearch.R**). Se observa la naturaleza aleatoria de este algoritmo, ya que la ejecución dependerá de la selección de los conjuntos de datos usados para entrenamiento y test en cada una de las iteraciones de la validación cruzada (para asegurar que siempre se obtiene el mismo resultado, al menos durante el proceso de desarrollo del algoritmo, podríamos forzar que la secuencia de números aleatorios generados se inicialice siempre de la misma forma mediante **set.seed(0)**).

```
library(rpart)
library(FSelector)
data(iris)

```

```
# Se define una funcion de evaluacion: recibe como argumento un
# vector de atributos a evaluar
evaluator <- function(subset){
  # se indica el numero de particiones a realizar en el proceso
  # de validacion cruzada
  k <- 10

  # genera valores aleatorios (uniforme) para cada muestra del
  # conjunto de datos
  splits <- runif(nrow(iris))

  # tratamiento de cada una de las particiones. Para cada valor de
  # particion se aplica la funcion que se define a continuacion
  results <- sapply(1:k, function(i) {
    # se determina el indice de las muestras para test (
        aproximadamente
    # una fraccion 1/k de las muestras del conjunto de datos)
    test.idx <- (splits >= ((i-1)/k) & (splits < (i/k)))

    # todas las demas muestras seran para training
    train.idx <- !test.idx

    # se seleccionan las muestras en si
    test <- iris[test.idx, ,drop=FALSE]
    train <- iris[train.idx, , drop=FALSE]

    # aprende el modelo sobre el conjunto de entrenamiento
    tree <- rpart::rpart(as.simple.formula(subset,"Species"),train)

    # calcula la tasa de error
    error.rate <- sum(test$Species != predict(tree, test, type="class
        "))/nrow(test)

    # devuelve la tasa de aciertos
    return(1-error.rate)
  })

  # se muestra el subconjunto y la media de resultados y se devuelve
  # la media de los resultados (un resultado por particion)
  print(subset)
  print(mean(results))
  return(mean(results))
}

# con esta funcion de evaluacion la seleccion se haria de la forma
    siguiente
subset <- FSelector::best.first.search(names(iris)[-5], evaluator)
f <- as.simple.formula(subset,"Species")
print(f)
```

La salida de una posible ejecución de esta función es:

```
[1] "Sepal.Length"
[1] 0.7097823
[1] "Sepal.Width"
[1] 0.5369879
[1] "Petal.Length"
```

```
[1] 0.9403852
[1] "Petal.Width"
[1] 0.9578678
[1] "Sepal.Length" "Petal.Width"
[1] 0.9535335
[1] "Sepal.Width" "Petal.Width"
[1] 0.9598724
[1] "Petal.Length" "Petal.Width"
[1] 0.93413
[1] "Sepal.Length" "Sepal.Width"  "Petal.Width"
[1] 0.9609244
[1] "Sepal.Width"  "Petal.Length" "Petal.Width"
[1] 0.9461195
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[1] 0.9285227
Species ~ Sepal.Length + Sepal.Width + Petal.Width
```

Se han probado los diferentes subconjuntos de atributos para la predicción de **Species** y se propone como subconjunto más adecuado: **Sepal.Length** + **Sepal.Width** + **Petal.Width**. Hay que tener en cuenta que distintas ejecuciones de este **script** pueden generar resultados diferentes debido a que las particiones creadas serán también diferentes.

En cualquier caso, cabe destacar el hecho de que la selección de características precisa la construcción de árboles de clasificación para la variable clase del conjunto de datos: uno para cada posible partición del conjunto de atributos. En realidad, se crean k modelos diferentes para cada posible partición del conjunto de variables. La media del número de errores cometidos al predecir se usa como indicación de la validez del subconjunto de variables usado para la construcción del modelo.

10.1.7. Aproximación wrapper: exhaustive.search

Búsqueda exhaustiva del subconjunto de atributos más relevantes (el *script* de uso se denomina **fSelector-exhaustiveSearch.R**). De nuevo estamos ante un método que precisa construir modelos para evaluar la influencia da cada variable.

```
library(rpart)
library(FSelector)
data(iris)

# funcion de evaluacion
# Se define una funcion de evaluacion: recibe como argumento un
# vector de atributos a evaluar y numero de particiones a usar
evaluator <- function(subset, k=5){
  # genera valores aleatorios (uniforme) para cada muestra del
  # conjunto de datos
  splits <- runif(nrow(iris))

  # tratamiento de cada una de las particiones. Para cada valor de
  # particion se aplica la funcion que se define a continuacion
  results <- sapply(1:k, function(i) {
    # se determina el indice de las muestras para test (
        aproximadamente
    # una fraccion 1/k de las muestras del conjunto de datos)
    test.idx <- (splits >= ((i-1)/k) & (splits < (i/k)))

```

```
    # todas las demas muestras seran para training
    train.idx <- !test.idx

    # se seleccionan las muestras en si
    test <- iris[test.idx, ,drop=FALSE]
    train <- iris[train.idx, , drop=FALSE]

    # aprende el modelo sobre el conjunto de entrenamiento
    tree <- rpart(as.simple.formula(subset,"Species"),train)

    # calcula la tasa de error
    error.rate <- sum(test$Species != predict(tree,test,type="c"))/
        nrow(test)

    # devuelve la tasa de aciertos
    return(1-error.rate)
  })

  # se muestra el subconjunto y la media de resultados y se devuelve
  # la media de los resultados (un resultado por particion)
  print(subset)
  print(mean(results))
  return(mean(results))
}

# llamada a la funcion
subset <- FSelector::exhaustive.search(names(iris[-5]),evaluator)

# se muestra el resultado
f <- as.simple.formula(subset, "Species")
print(f)
```

Resultado obtenido:

```
[1] "Sepal.Length"
[1] 0.7098131
[1] "Sepal.Width"
[1] 0.5205639
[1] "Petal.Length"
[1] 0.9276431
[1] "Petal.Width"
[1] 0.9495275
[1] "Sepal.Length" "Sepal.Width"
[1] 0.6175758
[1] "Sepal.Length" "Petal.Length"
[1] 0.9457214
[1] "Sepal.Length" "Petal.Width"
[1] 0.9322134
[1] "Sepal.Width"  "Petal.Length"
[1] 0.9396821
[1] "Sepal.Width" "Petal.Width"
[1] 0.9525434
[1] "Petal.Length" "Petal.Width"
[1] 0.9390325
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
[1] 0.9547101
[1] "Sepal.Length" "Sepal.Width"  "Petal.Width"
```

```
[1] 0.9444857
[1] "Sepal.Length" "Petal.Length" "Petal.Width"
[1] 0.9274343
[1] "Sepal.Width"  "Petal.Length" "Petal.Width"
[1] 0.9228926
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[1] 0.9452512
```

10.1.8. Aproximación wrapper: greedy.search

Algoritmo **greedy** de búsqueda de subconjuntos, en modo **backward** o **forward**. Precisa, como en el ejemplo anterior, definir una función de evaluación. Ejemplo de aplicación (**fSelector-greedy.R**):

```r
library(rpart)
library(FSelector)
data(iris)

# funcion de evaluacion
# Se define una funcion de evaluacion: recibe como argumento un
# vector de atributos a evaluar y numero de particiones a usar
evaluator <- function(subset, k=5){
  # genera valores aleatorios (uniforme) para cada muestra del
  # conjunto de datos
  splits <- runif(nrow(iris))

  # tratamiento de cada una de las particiones. Para cada valor de
  # particion se aplica la funcion que se define a continuacion
  results <- sapply(1:k, function(i) {
    # se determina el indice de las muestras para test (
        aproximadamente
    # una fraccion 1/k de las muestras del conjunto de datos)
    test.idx <- (splits >= ((i-1)/k) & (splits < (i/k)))

    # todas las demas muestras seran para training
    train.idx <- !test.idx

    # se seleccionan las muestras en si
    test <- iris[test.idx, ,drop=FALSE]
    train <- iris[train.idx, , drop=FALSE]

    # aprende el modelo sobre el conjunto de entrenamiento
    tree <- rpart(as.simple.formula(subset,"Species"),train)

    # calcula la tasa de error
    error.rate <- sum(test$Species != predict(tree,test,type="c"))/
        nrow(test)

    # devuelve la tasa de aciertos
    return(1-error.rate)
  })

  # se muestra el subconjunto y la media de resultados y se devuelve
  # la media de los resultados (un resultado por particion)
  print(subset)
  print(mean(results))
```

```
41   return(mean(results))
42 }
43
44 # se selecciona el subconjunto de atributos
45 subset <- FSelector::forward.search(names(iris)[-5], evaluator)
46 f <- as.simple.formula(subset,"Species")
47 print(f)
```

El resultado obtenido es:

```
1  [1] "Sepal.Length"
2  [1] 0.7047016
3  [1] "Sepal.Width"
4  [1] 0.4795996
5  [1] "Petal.Length"
6  [1] 0.95047
7  [1] "Petal.Width"
8  [1] 0.9520916
9  [1] "Sepal.Length" "Petal.Width"
10 [1] 0.9543347
11 [1] "Sepal.Width" "Petal.Width"
12 [1] 0.9532172
13 [1] "Petal.Length" "Petal.Width"
14 [1] 0.9204359
15 [1] "Sepal.Length" "Sepal.Width"  "Petal.Width"
16 [1] 0.9543351
17 [1] "Sepal.Length" "Petal.Length" "Petal.Width"
18 [1] 0.9409415
19 [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
20 [1] 0.9433067
21 Species ~ Sepal.Length + Sepal.Width + Petal.Width
```

10.1.9. Aproximación wrapper: hill.climbing.search

Aplicación de **hill-climbing** para la selección de atributos (**fSelector-hillClimbing.R**):

```
1  library(rpart)
2  library(FSelector)
3  data(iris)
4
5  # funcion de evaluacion
6  # Se define una funcion de evaluacion: recibe como argumento un
7  # vector de atributos a evaluar y numero de particiones a usar
8  evaluator <- function(subset, k=5){
9    # genera valores aleatorios (uniforme) para cada muestra del
10   # conjunto de datos
11   splits <- runif(nrow(iris))
12
13   # tratamiento de cada una de las particiones. Para cada valor de
14   # particion se aplica la funcion que se define a continuacion
15   results <- sapply(1:k, function(i) {
16     # se determina el indice de las muestras para test (
           aproximadamente
17     # una fraccion 1/k de las muestras del conjunto de datos)
18     test.idx <- (splits >= ((i-1)/k) & (splits < (i/k)))
19
```

```
20      # todas las demas muestras seran para training
21      train.idx <- !test.idx
22
23      # se seleccionan las muestras en si
24      test <- iris[test.idx, ,drop=FALSE]
25      train <- iris[train.idx, , drop=FALSE]
26
27      # aprende el modelo sobre el conjunto de entrenamiento
28      tree <- rpart(as.simple.formula(subset,"Species"),train)
29
30      # calcula la tasa de error
31      error.rate <- sum(test$Species != predict(tree,test,type="c"))/
            nrow(test)
32
33      # devuelve la tasa de aciertos
34      return(1-error.rate)
35    })
36
37    # se muestra el subconjunto y la media de resultados y se devuelve
38    # la media de los resultados (un resultado por particion)
39    print(subset)
40    print(mean(results))
41    return(mean(results))
42  }
43
44  # se selecciona el subconjunto de atributos
45  subset <- FSelector::hill.climbing.search(names(iris)[-5], evaluator)
46  f <- as.simple.formula(subset,"Species")
47  print(f)
```

Resultado obtenido:

```
1  [1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
2  [1] 0.9360083
3  [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
4  [1] 0.9411576
5  [1] "Sepal.Width"  "Petal.Length"
6  [1] 0.9239487
7  [1] "Sepal.Length" "Petal.Length"
8  [1] 0.9435339
9  [1] "Sepal.Length" "Sepal.Width"
10 [1] 0.7603933
11 [1] "Sepal.Length" "Petal.Length" "Petal.Width"
12 [1] 0.913364
13 [1] "Petal.Length"
14 [1] 0.9317079
15 [1] "Sepal.Length"
16 [1] 0.6675202
17 Species ~ Sepal.Length + Petal.Length
```

10.1.10. Aproximación wrapper: cfs

Busca los subconjuntos de atributos usando medidas de correlación y de entropía, tanto para variables continuas como discretas. La forma de aplicación es muy sencilla (**script fSelector-cfs.R**):

```
library(rpart)
library(FSelector)
data(iris)

# se realiza la seleccion de atributos: todos los atributos
# como disponibles para la clasificacion de la variable Species
subset <- FSelector::cfs(Species~., iris)
f <- as.simple.formula(subset,"Species")

# se muestra el resultado
print(f)
```

Y el subconjunto seleccionado en este caso es:

```
Species ~ Petal.Length + Petal.Width
```

El algoritmo usa la estrategia **best first search** en la búsqueda del mejor subconjunto de atributos. El resultado es un vector que contiene los atributos seleccionados.

10.1.11. Aproximación wrapper: consistency

Esta función usa medidas de consistencia para determinar el subconjunto de atributos más adecuado. Hace uso de la función **best.first.search** en el proceso de búsqueda. Se considera un ejemplo de aplicación (**fSelector-consistency.R**):

```
library(FSelector)
library(mlbench)
data(HouseVotes84)

# se usa el metodo consistency para seleccionar el subconjunto
# de atributos. Este metodo usa a su vez la funcion best.first.search
# para determinar el subconjunto mas prometedor
subset <- consistency(Class~.,HouseVotes84)

# se muestra el resultado
f <- as.simple.formula(subset,"Class")
print(f)
```

El resultado obtenido es el siguiente:

```
Class ~ V3 + V4 + V7 + V9 + V10 + V11 + V12 + V13 + V15 + V16
```

10.1.12. Aproximación embedded: ramdom.forest.importance

Método de cálculo de pesos de importancia de atributos calculados sobre un modelo construido usando el algoritmo **RandomForest**. Los nodos del modelo construido (de todos sus árboles) se analizan para aportar información de importancia de cada variable. El tercer argumento puede tener valor 1 ó 2, e indica el tipo de medida de importancia a usar: 1 (reducción en la media de fiabilidad predictiva) y 2 (reducción en la media de impureza de nodos). El **script** para este método se denomina **fSelector-randomForest.R**.

```
library(mlbench)
library(FSelector)
data(HouseVotes84)

```

```
# se calculan los pesos
weights <- FSelector::random.forest.importance(Class~.,HouseVotes84,
    importance.type=1)

# se muestran los resultados
print(weights)
subset <- cutoff.k(weights,5)
f <- as.simple.formula(subset,"Class")
print(f)
```

```
    attr_importance
V1       -2.4719121
V2        3.7373654
V3       24.9703195
V4       75.1599386
V5       20.1827690
V6        0.2122736
V7        6.7940908
V8        9.9557944
V9        7.7145375
V10       1.7445080
V11      16.8550561
V12      13.2565216
V13      13.1832803
V14      14.0119826
V15      11.4059817
V16      -3.3651157
Class ~ V4 + V3 + V5 + V11 + V14
```

10.2. Paquete caret

Este paquete ofrece también funcionalidad de construcción y ajuste de modelos de aprendizaje. Se consideran a continuación algunos ejemplos de uso.

10.2.1. Partición del conjunto de datos

La mayoría de los métodos validación del aprendizaje se basan en el uso de validación cruzada. Este paquete ofrece funcionalidad para particionar el conjunto de datos de forma sencilla (es algo que en algunos de los archivos anteriores hemos implementado de forma propia). Sin embargo, esta implementación presenta una característica importante: mantiene en la partición, de forma aproximada, la proporción de instancias para cada etiqueta de la variable clase. Estas sentencias se incluyen en el archivo **particionDatos.R**:

```
library(caret)
data(Sonar)
set.seed(107)

# se crea la particion: esto obtiene de forma aleatoria un
# porcentaje de instancias dado por p. El metodo mantiene
# la proporcion deinstancias para cada valor de la variable
# clase
inTrain <- caret::createDataPartition(y = Sonar$Class, p = .75,
                                      list = FALSE)

```

```
# ahora se obtienen los conjuntos de test y de entrenamiento
training <- Sonar[ inTrain,]
testing  <- Sonar[-inTrain,]

# se muestra la proporcion de instancias para cada valor de la
# variable clase en el conjunto de datos original
summary(Sonar$Class)
ggplot(data=Sonar) +
  geom_bar(mapping=aes(x=Class, y=..prop.., group=1))

# tambien en el de entrenamiento
summary(training$Class)
ggplot(data=training) +
  geom_bar(mapping=aes(x=Class, y=..prop.., group=1))

# y lo mismo con el de test
summary(testing$Class)
ggplot(data=testing) +
  geom_bar(mapping=aes(x=Class, y=..prop.., group=1))
```

Esto produce un conjunto de datos con 208 muestras y lo particiona seleccionando 157 muestras de entrenamiento y 51 para test.

10.2.2. Detección de variables muy correladas

Las variables dejan de ser realmente informativas si están muy correladas con otras. Si este es el caso, entonces no se pierde información si se descartan, reduciendo así el conjunto de datos. El paquete **caret** permite determinar el grado de correlación entre las variables de un conjunto de datos (archivo **correlacion.R**):

```
library(caret)
library(mlbench)

# se hace accesible el conjunto de datos PimaIndiansDiabetes
data(PimaIndiansDiabetes)

# se obtiene la matriz de correlacion de las variables predictoras
correlationMatrix <- cor(PimaIndiansDiabetes[,1:8])

# se encuentran aquellas variables que presentan valores de
    correlacion
# por encima del valor umbral
highlyCorrelated <- caret::findCorrelation(correlationMatrix,
                                           cutoff=0.3)
print(highlyCorrelated)
```

Este conjunto de datos contiene 9 variables y 768 muestras. La aplicación indica que hay 3 variables (con índice 4, 5 y 8) que presentan correlación por encima del valor umbral usado.

10.2.3. Cálculo de la importancia de las variables

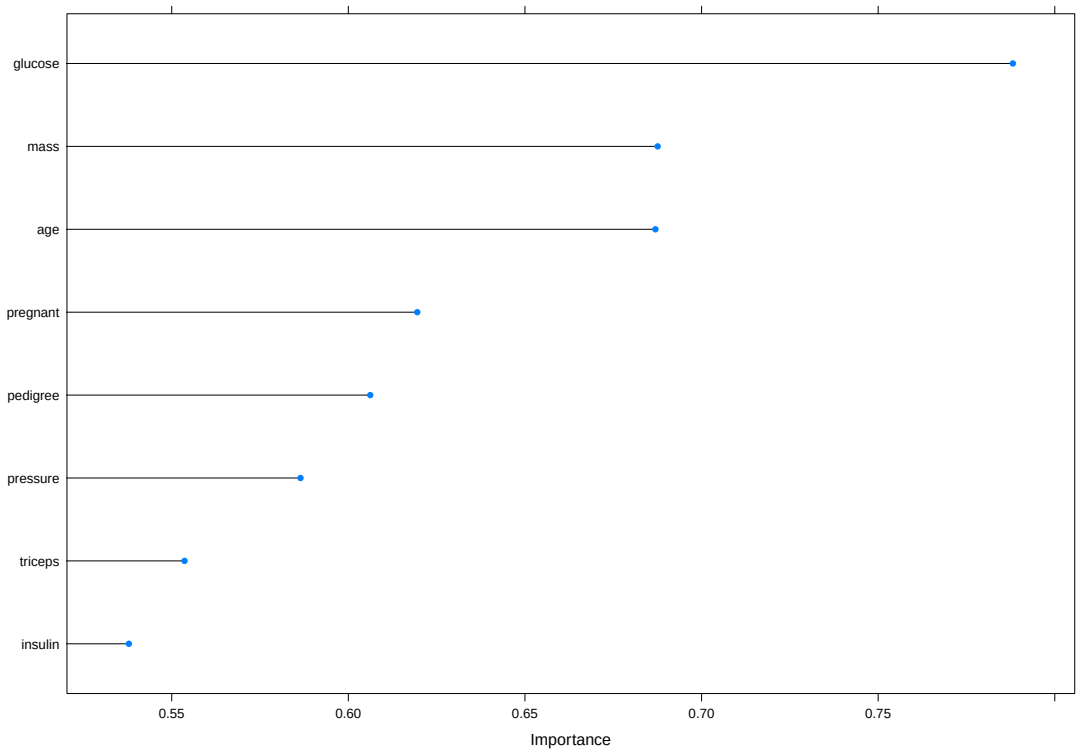
El siguiente ejemplo muestra cómo determinar la importancia de las variables aprendiendo diferentes modelos y extrayendo de esos modelos la información necesaria para valorarlas (**caret-importance.R**):

```r
library(caret)
library(pROC)

# se fija la semilla para asegurar la reproducibilidad de los
# resultados
set.seed(7)

# carga el conjunto de datos
data(PimaIndiansDiabetes)

# prepara el esquema de entrenamiento
control <- caret::trainControl(method="repeatedcv", number=10,
                               repeats=3)

# aprende el modelo
modelo <- caret::train(diabetes~., data=PimaIndiansDiabetes,
                       method="lvq", preProcess="scale",
                       trControl=control)

# estima la importancia de las variables a partir del modelo
importance <- caret::varImp(modelo, scale=FALSE)

# muestra los datos del analisis
print(importance)

# representa graficamente los resultados
plot(importance,lw=3)
```

El resultado obtenido es:

```
ROC curve variable importance

         Importance
glucose      0.7881
mass         0.6876
age          0.6869
pregnant     0.6195
pedigree     0.6062
pressure     0.5865
triceps      0.5536
insulin      0.4621
```

El gráfico obtenido con el script anterior es:



10.2.4. Esquema de valoración con aprendizaje de random forest

El script usado se denomina **caret-randomForest.R**:

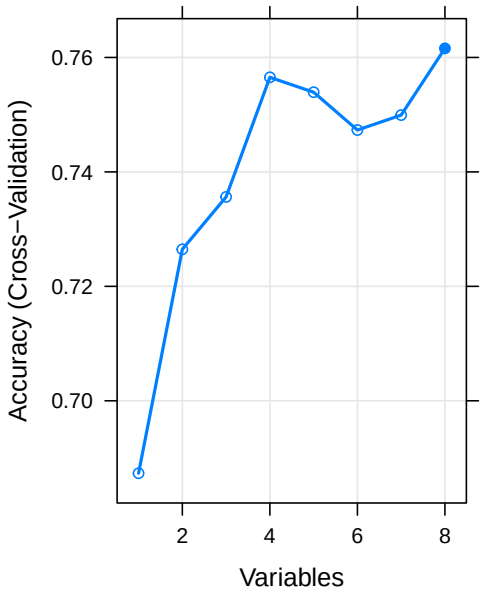
```
library(caret)

# se asigna la semilla para asegura la reproducibilidad de los
# resultados
set.seed(7)

# carga el conjunto de datos
data(PimaIndiansDiabetes)

# define el control usando la funcion de seleccion mediante
# random forest
control <- caret::rfeControl(functions=rfFuncs, method="cv",
                             number=10)

# ejecuta el metodo
results <- caret::rfe(PimaIndiansDiabetes[,1:8],
                      PimaIndiansDiabetes[,9], sizes=c(1:8),
                      rfeControl=control)

# muestra los resultados
print(results)

# muestra las caracteristicas elegidas
predictors(results)

# realiza un grafico de los resultados. El grafico muestra que con
# 4 atributos se obtienen resultados simulares a usar los 8 atributos
```

```
# iniciales
plot(results, type=c("g", "o"), lw=2)
```

Y los resultados producidos son:

```
Recursive feature selection

Outer resampling method: Cross-Validated (10 fold)

Resampling performance over subset size:

 Variables Accuracy  Kappa AccuracySD KappaSD Selected
         1   0.6873 0.2583    0.05704 0.11679
         2   0.7265 0.3748    0.04812 0.09870
         3   0.7356 0.4086    0.04534 0.10020
         4   0.7565 0.4547    0.04107 0.09508
         5   0.7539 0.4429    0.05277 0.12191
         6   0.7473 0.4324    0.03894 0.08530
         7   0.7499 0.4329    0.04013 0.09785
         8   0.7616 0.4584    0.04917 0.11000        *

The top 5 variables (out of 8):
   glucose, mass, age, pregnant, pedigree
```

El gráfico producido muestra que el modelo aprendido con 4 variables es casi tan bueno como el obtenido con todas ellas.



10.3. Paquete Boruta

El paquete **Boruta** también ofrece funciones para selección de características. Los siguientes scripts muestran algunos ejemplos de uso.

10.3.1. Obtención de estadísticas sobre los atributos

El siguiente ejemplo muestra la forma de obtener estadísticas sobre los atributos del problema (**boruta-estadisticas.R**):

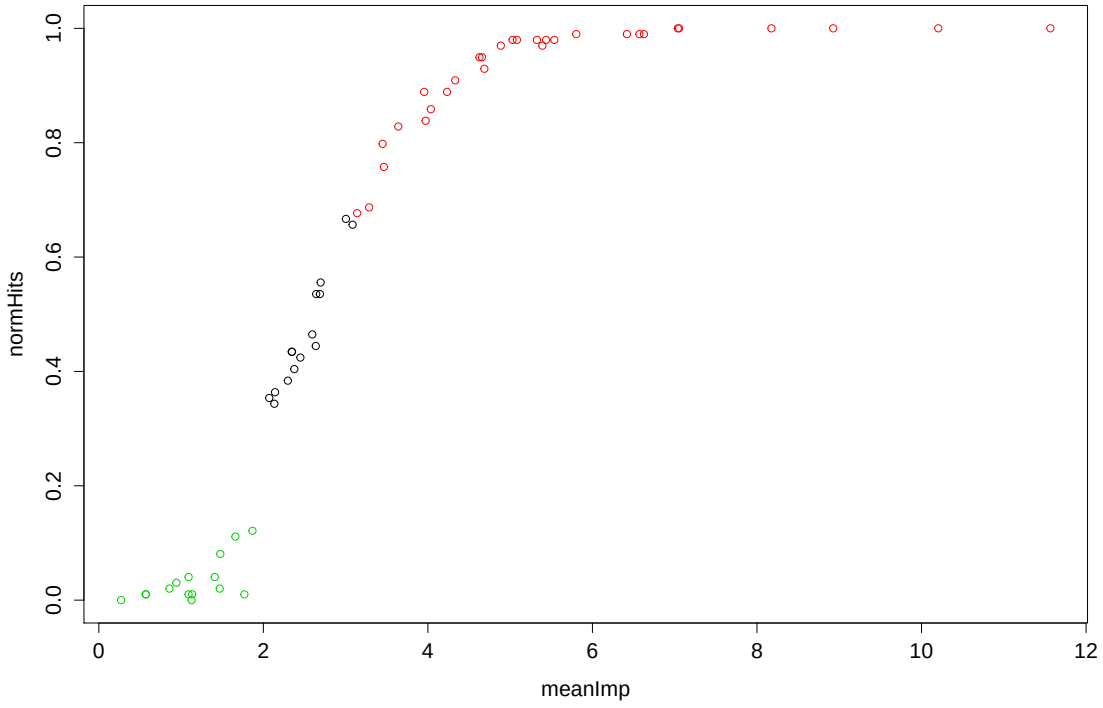
```
library(Boruta)
library(mlbench)

# carga el conjunto de datos
data(Sonar)

# aprende el modelo
Bor.son <- Boruta(Class~.,data=Sonar,doTrace=2)

# muestra los resultados
print(Bor.son)

# se ven los resultados de decision de cada variable
print(Bor.son$finalDecision)

# imprime las estadisticas
stats <- attStats(Bor.son)
print(stats)

# se muestran los resultados en forma grafica
plot(Bor.son)

# muestra un grafico de los resultado: los valores en
# rojo estan relacionados con las variables confirmadas
# mientras que los verdes con variables descartadas
plot(normHits~meanImp,col=stats$decision,data=stats)
```

Los resultados producidos por estas sentencias son:

```
Boruta performed 99 iterations in 7.715602 secs.
 33 attributes confirmed important: V1, V10, V11, V12, V13 and 28 more;
 17 attributes confirmed unimportant: V24, V25, V3, V33, V38 and 12
    more;
 10 tentative attributes left: V14, V2, V29, V30, V32 and 5 more;
 ..........................................................
       V1         V2         V3         V4         V5         V6
       Confirmed  Tentative  Rejected   Confirmed  Confirmed  Rejected

       V7         V8         V9         V10        V11        V12
       Rejected   Tentative  Confirmed  Confirmed  Confirmed  Confirmed

       V13        V14        V15        V16        V17        V18
       Confirmed  Tentative  Confirmed  Confirmed  Confirmed  Confirmed

       V19        V20        V21        V22        V23        V24
     Confirmed  Confirmed  Confirmed  Confirmed  Confirmed   Rejected

       V25        V26        V27        V28        V29        V30
       Rejected   Confirmed  Confirmed  Confirmed  Tentative  Tentative

       V31        V32        V33        V34        V35        V36
       Confirmed  Tentative  Rejected   Tentative  Confirmed  Confirmed
```

```
23
24          V37         V38         V39         V40         V41         V42
25    Confirmed    Rejected   Tentative    Rejected    Rejected    Rejected
26
27          V43         V44         V45         V46         V47         V48
28    Confirmed   Confirmed   Confirmed   Confirmed   Confirmed   Confirmed
29
30          V49         V50         V51         V52         V53         V54
31    Confirmed    Rejected   Confirmed   Confirmed    Rejected   Tentative
32
33          V55         V56         V57         V58         V59         V60
34    Rejected    Rejected    Rejected    Rejected   Tentative    Rejected
35 Levels: Tentative Confirmed Rejected
```

Y el gráfico final muestra cómo aquellos atributos con más importancia (en rojo) están también asociados a mayor valor de tasa de aciertos.

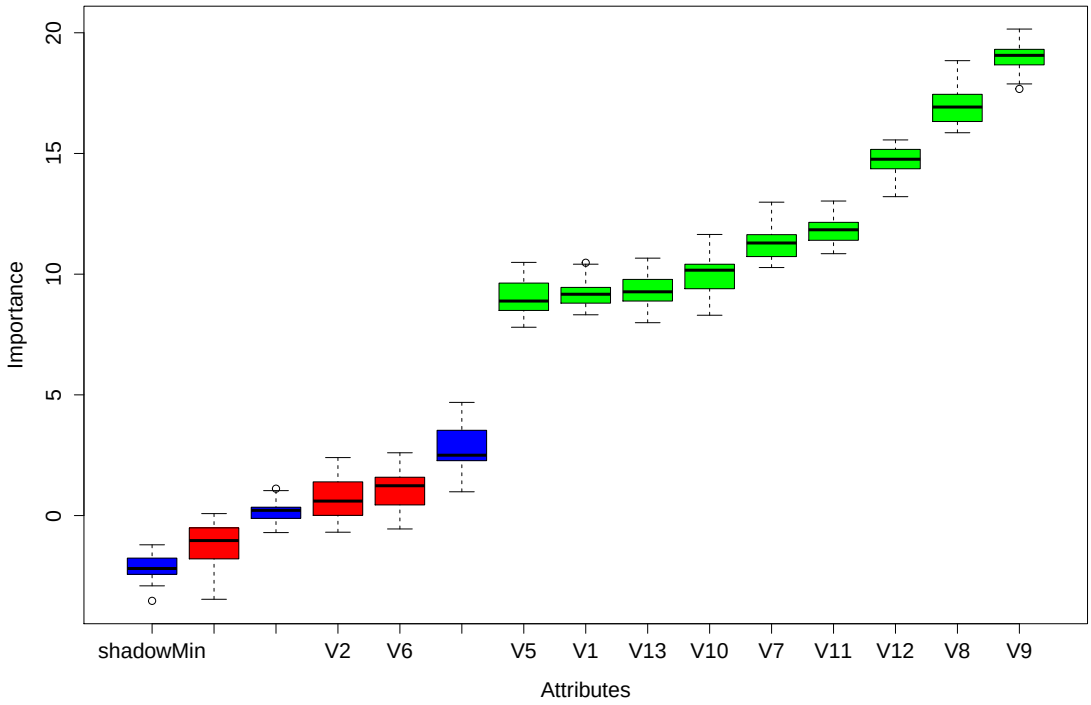


10.3.2. Combinación con random forest

Puede usarse en combinación con **random-forest** como forma de validar el modelo (**boruta-randomForest.R**):

```
1 library(Boruta)
2 library(randomForest)
3
4 # tambien puede usarse en combinacion con random forest como
5 # forma de validar la seleccion
6 data(Ozone)
7
8 # se eliminan datos perdidos
```

```r
ozo <- na.omit(Ozone)

# aplicacion del metodo de seleccion
Bor.ozo <- Boruta(V4~.,data=ozo,doTrace=2)

cat("Random forest sobre todos los atributos\n")
model1 <- randomForest(V4~.,data=ozo)
print(model1)

cat("Random forest unicamente sobre atributos confirmados\n")
model2 <- randomForest(ozo[,getSelectedAttributes(Bor.ozo)],ozo$V4)
print(model2)

# se muestra un grafico con los resultados de la seleccion
plot(Bor.ozo)
```



11. Detección de ruido

La detección de ruido resulta esencial para algunos algoritmos en los que resulta crítica la presencia de este fenómeno. Se trata en definitiva de una técnica que nos va a permitir identificar instancias en que los valores de alguna (o varias) de las variables no se corresponden a lo esperado (como puede pensarse, es algo muy relacionado con la detección de datos anómalos).

Uno de los algoritmos más eficaces en esta tarea se conoce como **Iterative Partitioning Filter**. Se basa en eliminar instancias mal etiquetadas en grandes conjuntos de datos. Muchos algoritmos de este estilo asumen que los conjuntos de datos son relativamente pequeños y pueden aprenderse de una vez, pero esto no siempre es así. Por eso este

algoritmo realiza varias iteraciones hasta alcanzar una condición de parada: generalmente que el número de instancias consideradas como defectuosas está por debajo de un cierto porcentaje del número total de instancias en el conjunto de datos. El procedimiento básico consiste en:

- se particiona el conjunto de datos en una serie de particiones de tamaño similar. Cada partición debe ser suficientemente pequeña como para que un algoritmo sencillo pueda aprender bien el modelo (sea P el número de particiones)
- para cada una de estas particiones se aprende un modelo, por ejemplo usando C4.5
- estos P modelos se usan para clasificar todas las instancias del conjunto de entrenamiento y marcarlas como correcta o incorrectamente etiquetadas. Para ello se usa un sistema de votación por mayoría (si la mayoría de modelos predicen un valor incorrecto para una instancia, queda marcada como defectuosa). Las instancias defectuosas se van almacenando de alguna forma
- se eliminan las instancias con ruido y se vuelve a comenzar

El esquema de una función para implementar este algoritmo podría ser el siguiente:

```
funcion ipf
  Argumentos:
     dataset: conjunto de datos
     k: numero de particiones a hacer en el conjunto de datos
     limite: porcentaje limite para la condicion de parada

  mientras no se cumpla la condicion de parada
     realizar el particionado del conjunto de datos

     para cada particion
        construir modelo de prediccion
        para cada instancia en el conjunto de datos
           predecir usando el modelo
           si hay fallo, incrementar el contador de fallos de la
              instancia
        fin para
     fin para

     descartar instancias con mas votos para fallo que para acierto

     si el porcentaje de instancias descartadas esta por debajo del
     limite fijado, parar
  fin mientras
```

Este método ya está disponible en un paquete **R** llamado **NoiseFiltersR**. Se muestra un ejemplo de uso en el **script** llamado **ipf.R**.

```
library(NoiseFiltersR)

data(iris)

set.seed(1)
out <- IPF(Species~., data = iris, s = 2)
summary(out, explicit = TRUE)
identical(out$cleanData, iris[setdiff(1:nrow(iris),out$remIdx),])
```

Parte II
Construcción de modelos

12. Clasificación

En primer lugar debemos tener claro que la realización de esta tarea implica que los datos poseen una característica esencial: existe una variable especial, llamada **clase**, con valor asignado para todas las instancias. En este caso se dice que los datos están **etiquetados**. En esta forma de aprendizaje el objetivo es construir, a partir de un conjunto de datos, un modelo que permita predecir el valor de la **variable clase** para nuevas instancias, no observadas previamente (no pertenecientes al conjunto de datos usado para aprender el modelo).

Existen muchas técnicas diferentes para realizar esta tarea. Por ejemplo, es posible aplicar técnicas básicas de regresión con algunas modificaciones para que los valores resultantes de la predicción esté comprendidos entre 0 y 1, en el caso de una variable clase con dos valores posibles. Este es el caso más simple, donde, de forma intuitiva, cualquier método de aprendizaje buscará calcular una frontera de decisión que permita separar las instancias de ambas clases.

En el caso de una variable clase con más de dos posibles valores existen diferentes estrategias. Una de las más habituales consiste en crear un modelo para cada uno de estos valores, para cada etiqueta de la variable clase. Imaginemos que se desea aprender el clasificador para la etiqueta de índice i. Se crea una nueva copia de los datos de entrada en la que se hace que la variable clase tome el valor 1 para todas las instancias pertenecientes a dicha clase y 0 para todas las demás. De esta forma, esta estrategia necesita crear tantos modelos como valores tenga la etiqueta clase.

El resultado final se obtendría componiendo de alguna forma las respuestas de todos los clasificadores. Si el clasificador proporciona valores entre 0 y 1 se asignaría a una instancia a clasificar aquella clase en que el valor calculado es mayor (sería como interpretar el valor como probabilidad de pertenencia).

A continuación se consideran diferentes elementos básicos relacionados con el proceso de aprendizaje de clasificadores:

- sobreajuste
- ajuste de parámetros
- partición del conjunto de datos
- medidas de rendimiento

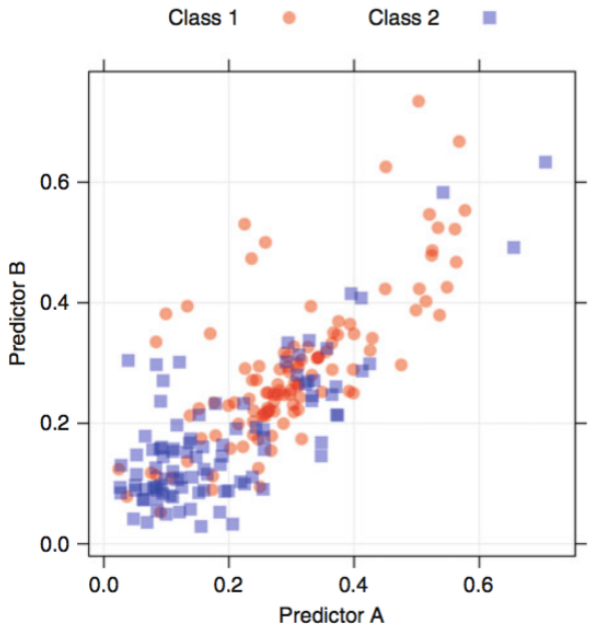
Posteriormente se analizarán algunas técnicas concretas de clasificación y la forma de usarlas en **R**.

12.1. Sobreajuste

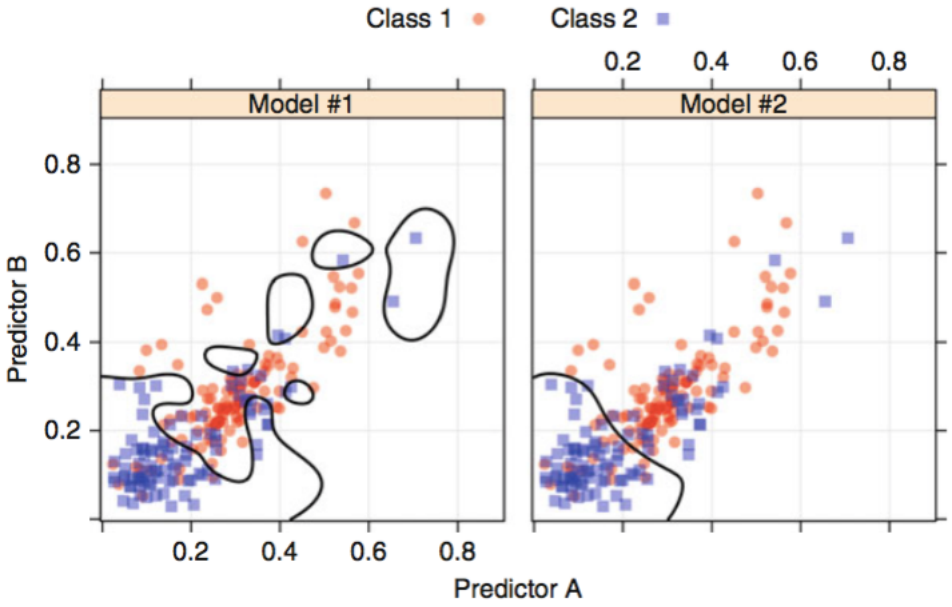
El objetivo de cualquier método de aprendizaje debe ser producir buenas predicciones para el tratamiento (clasificación) de instancias no vistas previamente. Se suele indicar que es deseable que el modelo **generalice** de forma correcta lo aprendido a partir de los datos disponibles, a partir de los que se aprende.

Imaginemos un modelo capaz de clasificar de forma correcta todas las instancias del conjunto de entrenamiento. Es decir, que la frontera de decisión, sea como sea, es capaz de dividir de forma correcta todas las instancias en las clases correspondientes. Podemos pensar que esto es una característica deseable del modelo, pero hemos de recordar que el objetivo último no es clasificar las instancias del conjunto de entrenamiento (de hecho, al estar etiquetados los datos ya sabemos la clase a la que pertenecen). Como hemos indicado antes lo interesante es obtener un modelo que sea capaz de predecir de forma correcta la clase a la que pertenecen nuevas instancias.

De este modo, si el modelo aprendido se ajusta demasiado a los datos usados para aprender puede ocurrir que no sea capaz de **generalizar** bien y ofrecer predicciones fiables para nuevas instancias. A modo de ejemplo, consideremos la siguiente situación: un conjunto de datos que consta de dos variables predictoras (dos atributos) y dos clases diferentes. La siguiente representación gráfica muestra la distribución de las instancias, pintando de color rojo todas aquellas que pertenecen a la primera clase y en azul las de la segunda:



Imaginemos que hemos obtenido dos modelos diferentes a partir de los datos. Las fronteras de decisión para ellos son las mostradas en la figura siguiente:



El clasificador de la izquierda presenta una frontera de decisión compleja, agrupando diferentes zonas del espacio. Este modelo es tan específico para el conjunto de datos de entrenamiento que difícilmente producirá una buena generalización. Por su parte, el modelo de la derecha presenta una frontera de decisión mucho más simple, incapaz de clasificar de forma correcta todas las instancias del conjunto de entrenamiento. Sin embargo, al no ajustarse tanto a los datos concretos disponibles para el aprendizaje, presentará mejor comportamiento a la hora de predecir nuevas instancias.

12.2. Ajuste de parámetros

Algunos de los algoritmos de aprendizaje (no sólo de clasificación, de modo que lo comentado aquí sería aplicable también) precisan especificar un cierto número de parámetros que determinan su comportamiento final. Por ejemplo, un parámetro a especificar al construir clasificadores tipo árbol suele ser el criterio usado para elegir el siguiente atributo por el que particionar. Otro, el número mínimo de instancias para seguir aplicando el particionado. Y en el algoritmo de agrupamiento denominado **k-vecinos más cercanos** hay que especificar el número de grupos a formar (el valor de k).

En este caso debemos disponer de algún procedimiento que nos permita elegir la mejor combinación de valores de estos parámetros. La forma habitual de proceder consiste en definir cuáles son los rangos de valores posibles para los parámetros a considerar e ir aprendiendo una batería de modelos con cada combinación posible de ellos. La selección de alguno de ellos se basará en alguna medida sobre su rendimiento.

La primera medida de rendimiento que podríamos pensar en usar sería el número de instancias bien clasificadas del conjunto de datos usado para aprender. Esto se conoce como **rendimiento aparente** del método. Pero ya hemos visto que esta medida puede no ser fiable, al no indicar nada sobre la capacidad de generalización del modelo. Es decir,

tendríamos modelos con una muy alta medida de rendimiento (aparente), pero con un comportamiento muy pobre al considerar nuevas instancias.

Una medida realmente fiable del rendimiento debería tener en cuenta el comportamiento del modelo antes instancias no tratadas. La única forma de poder conseguir esto consistirá en dejar algunas instancias fuera del proceso de aprendizaje, fuera del **conjunto de entrenamiento**. Las instancias apartadas se utilizarán como **conjunto de prueba**. Se indican a continuación algunas formas de realizar el particionado del conjunto de datos a efectos de obtener medidas de rendimiento fiables.

En cualquier caso, debemos tener claro que hablamos aquí de la forma de conseguir una medida de rendimiento que sea más fiable que el **rendimiento aparente**. El modelo finalmente construido sí que debería contar con todos los datos posibles. El particionado se ha realizado como una forma de obtener una medida de rendimiento que estime, de alguna forma, la capacidad de generalización del modelo.

12.3. Partición del conjunto de datos

En la situación ideal (se dispone de un gran conjunto de datos), el modelo debería ser evaluado sobre un conjunto de muestras no usadas ni para aprender el modelo ni para ajustar sus parámetros. Esta sería la única forma de obtener una medida de eficiencia sin sesgo. En este caso, al disponer de muestras suficientes, no hay problema en dividir el conjunto de datos en dos subconjuntos: **entrenamiento o aprendizaje** y **test o prueba**. Debemos tener en cuenta que la consideración de conjunto de datos grande o pequeño dependerá del problema (del número de variables presentes y de las combinaciones de sus valores posibles).

Sin embargo, cuando el número de muestras no es demasiado grande no podemos dejar de lado un subconjunto de instancias para prueba porque se necesitan todas para aprender. Por otra parte, algunos trabajos han demostrado que la validación a partir de un único conjunto de test no suele producir buenos resultados. Esto ha llevado proponer nuevas estrategias de particionado mediante muestreo aleatorio. La más conocida, que veremos a continuación se denomina **k-validación cruzada**.

Esta forma de validación implica el particionado del conjunto de datos en k subconjuntos de igual tamaño (aproximadamente) y obtener k modelos diferentes dejando una partición, de forma sucesiva, como conjunto de test. Es decir, en todos los modelos se usan $k-1$ particiones para entrenamiento y 1 para test. La estimación final ofrecida sería la media de las estimaciones para cada uno de los k modelos. Aunque hablaremos de varias medidas de rendimiento, una de ellas podría ser el número o porcentaje de instancias bien clasificadas del conjunto de prueba. En forma esquemática:

- dividir el conjunto de datos en k subconjuntos
- para $i = 1..k$
  - seleccionar de forma sistemática $k-1$ subconjuntos para entrenamiento y 1 para test
  - aprender modelo m_i con los datos de entrenamiento
  - validar el modelo con conjunto de test, obteniendo medida v_i
- componer medida global de rendimiento: $v_m = \sum_{i=1}^{k} v_i$
- aprender modelo con todos los datos disponibles

· `ofrecer la medida global de error como estimación del modelo`

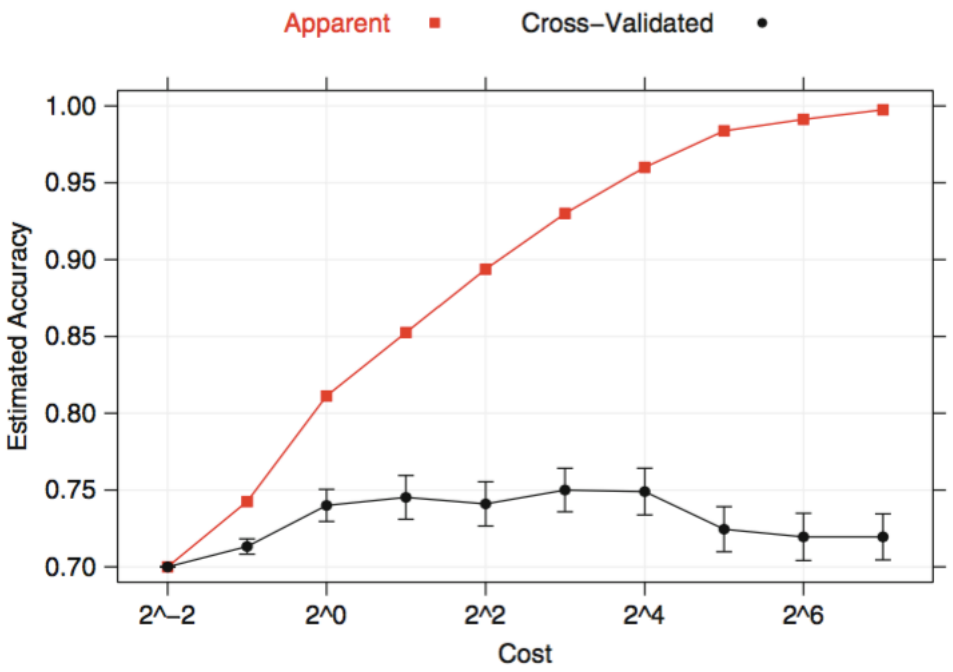
Algunas variantes de este método intentan modificar la forma de muestreo para que las k particiones presenten más o menos la misma proporción de instancias de cada clase (usando por ejemplo muestreo estratificado).

El caso especial en que $k = 1$ se denomina **leave one out** (y resulta el más costoso, ya que hemos de construir k modelos diferentes). En general, para que la medida de rendimiento sea más fiable el procedimiento completo de validación cruzada suele repetirse varias veces.

Otra variante se conoce como **validación cruzada Montecarlo**. En ella simplemente se crean múltiples particiones de los datos en entrenamiento y test. El usuario indicaría el porcentaje de instancias a usar en cada uno de los subconjuntos (habitualmente entre 75 % y 80 % para entrenamiento y el resto para test) y el número de repeticiones (usualmente una estimación fina precisaría entre 50 y 200).

Algunas veces se usa muestreo con reemplazamiento para hacer este particionado. Se comenta aquí ya que algunos métodos que veremos más adelante se basan en él (en inglés este tipo de muestreo se conoce como **bootstrap**). En esta técnica las muestras seleccionadas pueden volver a seleccionarse.

Un posible gráfico usado para determinar los valores más adecuados de los parámetros (en este caso uno solo, denominado **Cost**) sería el mostrado en la siguiente figura. Se trata de obtener un modelo diferente para cada valor del parámetro. Así, para cada valor del parámetro se ha construido un modelo (con todas las instancias disponibles). Se determina el valor de rendimiento aparente y se dibuja mediante un punto rojo. Pero para obtener una estimación más fiable de su capacidad de generalización, se usa validación cruzada (por ejemplo, con $k = 10$). Esto supone que se construyen 10 modelos dejando una partición para test, que se usará para obtener una medida de rendimiento. Estas medidas de rendimiento se representan como un intervalo y la media con un punto. Se aprecia que la medida de rendimiento aparente sobrestima el funcionamiento del modelo.



12.4. Medidas de rendimiento

Comentamos aquí algunas medidas que suelen usarse para caracterizar el comportamiento de un clasificador (consideramos que se obtienen siempre sobre el conjunto de test):

- porcentaje de instancias bien clasificadas
- porcentaje de instancias mal clasificadas
- para problemas con dos clases las medidas suelen definirse en función de los siguientes contadores: TP (verdaderos positivos), TN verdaderos negativos, FP (falsos positivos) y FN falsos negativos
  - sensibilidad o **recall**: $TP/(TP+FN)$ (de entre todos las instancias positivas, cuántas se identificaron como tales)
  - especificidad: $TN/(TN+FP)$ (de entre todas las instancias negativas, cuántas se clasificaron de forma correcta)
  - precision: $TP/(TP+FP)$ (de todas las instancias clasificadas como positivis, cuántas lo son realmente)
  - tasa de falsos positivos: $1 - especificidad$
  - índice J: $sensibilidad + especificidad - 1$
  - ....

13. Código R para las secciones de ajuste de parámetros y medidas de rendimiento, partición del conjunto de datos y validación del modelo

En el siguiente **script** de **R** se incluye código que ilustra la forma de realizar el particionado del conjunto de datos (de nombre **particionado.R**). En la primera parte del archivo se considera el uso de la función **createDataPartition** del paquete **caret**, que permite particionar el conjunto de instancias de forma que podamos preservar parte de los datos para la validación del modelo (conjunto de test). También se muestra la forma de crear particiones para realizar validación Montecarlo, con el número de repeticiones que se desee. La parte final del *script* contiene las sentencias necesarias para generar muestras con reemplazamiento, incluso para varias repeticiones del proceso de validación.

```
# libreria que incluye algunos conjuntos de datos
# y asociada al libro de igual titulo
library(AppliedPredictiveModeling)
library(caret)

# se selecciona el conjunto de datos de interes: mtcars. Contiene
# 32 instancias y 11 variables: todas son numericas
data(mtcars)

# se obtiene resumen de los datos
summary(mtcars)

# se fija la semilla para el generador de numeros aleatorios,
# con lo que el experimento es reproducible. Para comportamiento
# normal deberia desactivarse esta opcion
```

```r
set.seed(1)

# se genera un vector con tantos indices como instancias haya en
# el conjunto de datos
indices <- seq(1,nrow(mtcars),by=1)

# se pasa como argumento el vector de indices y de el se van
# seleccionando valores, con la probabilidad indicada. De esta
# forma, el vector resultante deberia tener aproximadamente
# el 80% de valores del pasado como primer argumento
indicesEntrenamiento <- caret::createDataPartition(indices, p=0.8,
                                                   list = FALSE)

# ahora se seleccionan los indices de los datos del conjunto de test
indicesTest <-indices[-indicesEntrenamiento]

# con esto es facil ahora seleccionar los conjuntos de datos
datosEntrenamiento <- mtcars[indicesEntrenamiento, ]
datosTest <- mtcars[indicesTest, ]

# si necesitamos obtener varias particiones, por ejemplo, para hacer
# validacion cruzada mediante metodo Montecarlo, podemos repetir
# el proceso el numero de veces que deseemos. Ahora el resultado
# sera una matriz con tantas filas como muestras y tantas columnas
# como repeticiones se hayan indicado
particionadosEntrenamiento <- caret::createDataPartition(indices,
                                                         p = .80,
                                                         list= FALSE,
                                                         times = 10)

# De esta forma, cada columna contiene los indices de las instancias
# seleccionada de una de las particiones
particion1 <- particionadosEntrenamiento[,1]
particion10 <- particionadosEntrenamiento[,10]

# ahora podemos obtener los indices de las instancias de test para
# cada caso. En primer lugar se calcula el numero de instancias en
# el conjunto test. El objetivo es crear una matriz con tantas
# filas como tenga el conjunto de test y tantas columnas como
# variantes del particionado se hayan generado (en este caso 10).
# De esta forma, cada columna servira para almacenar un conjunto
# de test. En primer lugar se determina el tam. del conjunto
# de test
tamConjuntoTest <- nrow(mtcars) - nrow(particionadosEntrenamiento)

# se crea una matriz para almacenar las sentencias de test de
# todas las particiones
particionadosTest <- matrix(nrow=tamConjuntoTest, ncol = 10)

# se obtienen de forma iterativa considerando las particiones de
# entrenamiento
for(i in 1:10){
  particionadosTest[,i] <- indices[-particionadosEntrenamiento[,i]]
}

# tambien es posible usar createResamples (para bootstrap),
# createFolds (para k-fold cross-validation) y createMultiFolds
```

```
# (para repeticiones de cross-validation). Por ejemplo, imaginemos
# deseamos crear 10 particiones para validacion cruzada: el siguiente
# metodo me ofrece las particiones de forma que pueda usarlas
# posteriormente
set.seed(8)
particionesEntrenamiento <- caret::createFolds(indices, k = 10,
                                               list = TRUE,
                                           returnTrain = TRUE)

# el objeto devuelto ahora por R es una lista con tantas entradas
# como indique k. Cada una de las entradas es un vector con los
# indices de las instancias generadas
particion1 <- particionesEntrenamiento[[1]]
particion10 <- particionesEntrenamiento[[10]]

# de nuevo podemos usar la misma estrategia de antes para generar los
# indices de las correspondientes particiones de test
particionesTest <- list()
for(i in 1:10){
  particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
}

# para que sea mas visible, vamos a considerar que tenemos 10
# instancias
indices <- seq(1,10)
particionesEntrenamiento <- caret::createFolds(indices, k = 10,
                                           returnTrain = TRUE)

# e igual para los indices de test
particionesTest <- list()
for(i in 1:10){
  particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
}

# probamos el uso de la tecnica de bootstrap: ahora habra
# muestras repetidas y ademas todas las particiones tienen
# el mismo numero de muestras que el conjunto de datos original
particionBootstrap <- caret::createResample(indices, times=10)

# tambien podemos probar la creacion de multiplesParticiones: en este
# caso 3 particionados completos, de 10 particiones cada una de ellas
multiplesParticiones <- caret::createMultiFolds(indices, k=10,
                                                times=3)
```

El siguiente conjunto de sentencias están más relacionadas con la construcción de modelos y con la forma en que se ajustan sus parámetros o se obtienen algunas medidas de interés sobre su rendimiento. El archivo se llama **construccionModelos.R**.

Sobre este esquema podemos hacer los siguientes comentarios:

- parte 1: el inicio del archivo muestra la forma de aprender un modelo mediante la técnica de **máquina de soporte vectorial**. El objetivo del modelo a construir se especifica mediante una fórmula: una clase de **R** que indica qué variable debe ser objeto de la predicción (clasificación o regresión) y qué variables se usarán como predictoras. Estas dos informaciones se separan por el símbolo $\sim$. Si a la derecha de este símbolo aparece el carácter punto indica que todas las demás variables se

usan como predictoras. La funcionalidad básica descrita forma parte del paquete **caret**. Al final de este primer fragmento se indica la forma en que haríamos de forma manual el procedimiento de validación cruzada con 10 particiones. En primer lugar se realiza el particionado del conjunto de datos y posteriormente se procede al aprendizaje y validación con cada una de las particiones. Las estimaciones de error y de acierto obtenidas sirven como estimación del funcionamiento del modelo construido a partir de todas las instancias disponibles. Si observamos la información sobre este último modelo se aprecia que el método realiza de forma automática un ajuste de parámetros para seleccionar el mejor valor del parámetro C, que resulta ser 0,25. Este proceso de ajuste paramétrico se basa en muestreo bootstrap con 25 repeticiones. Las muestras contienen 1000 instancias y se usan como conjunto de entrenamiento. El conjunto de test se forma tomando aquellas instancias no seleccionadas para conjunto de entrenamiento. Se aprecia también la semejanza entre la estimación de fiabilidad predictiva asociada al mejor valor de C y la obtenida mediante validación cruzada: 0,68886 y 0,694. Se ha incluido este ejemplo aquí para ilustrar de forma precisa la forma en que se aplicaría la validación cruzada en caso de no disponer de métodos automáticos de realizarla (como se verá a continuación).

- parte 2: también es posible aplicar algunas operaciones de preprocesamiento a los datos antes de ser usados para aprender. Por ejemplo, podemos estar interesados en centrar y escalar los datos, algo de interés para construir un modelo mediante el algoritmo usado: máquinas de soporte vectorial. Se observa que el resultado es ligeramente mejor al obtenido previamente. Esto se debe a que algunas de las variables del modelo tienen diferencias apreciables entre sus escalas: $(4-72)$, $(250-18424)$, ...

- parte 3: este método tiene un parámetro que permite definir la función de coste, que mide el error entre las predicciones realizadas por el sistema y el resultado real observado en las instancias usadas. De forma casi automática resulta posible hacer el aprendizaje indicando al método **train** que considere diferentes valores del parámetro de forma que pueda analizarse su efecto, por ejemplo, mediante un gráfico.

- parte 4: incluso, para tener más confianza en el estudio, se puede indicar que se repita el proceso completo un determinado número de veces

- parte 5: se muestra la forma de poder comparar dos modelos diferentes de forma automática. Para hacer la comparación generamos un modelo con un **kernel** diferente, pero usando también el esquema de validación cruzada con 5 repeticiones, de forma que podamos comparar con **modelo4**

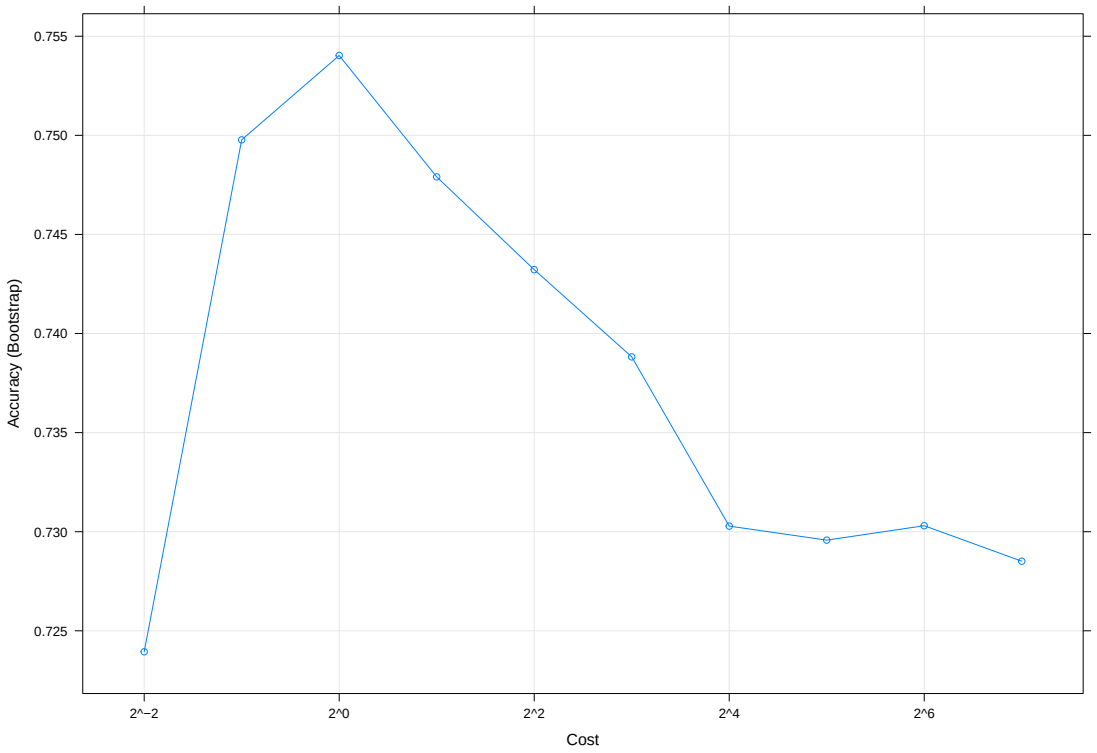
```
1  library(caret)
2  library(e1071)
3  data(GermanCredit)
4
5  # ********************** PARTE 1 ********************************
6
7  # se crean las particiones del conjunto de datos. En este caso
8  # se usa con conjunto de datos con 1000 instancias y 62 variables.
9  # Se generan las particiones del conjunto de datos mediante
10 # la funcion createFolds, que genera 10 particiones
11 indices <- seq(1,nrow(GermanCredit),by=1)
12 particionesEntrenamiento <- createFolds(indices, k = 10,
```

```
                                                returnTrain = TRUE)

# genero de la misma forma las particiones de test
particionesTest <- list()
for(i in 1:10){
  particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
}

# usaremos estas particiones para hacer validacion cruzada
# Bucle de generacion de modelos
errores <- c()
aciertos <- c()
for(i in 1:10){
  # hay varias formas de especificar el objetivo del modelo a
  # construir. Una de ellas es la formula: se compone de dos
  # terminos separados por el simbolo ~: a la izquierda va
  # la variable a predecir (clasificacion o regresion) y a la
  # derecha las variables a usar como predictoras. En el ejemplo
  # siguiente el punto a la derecha indica que se usan todos los
  # atributos como atributos. Como ejemplo veremos la aplicacion
  # del metodo de aprendizaje a una de las particiones
  modelo <- train(Class ~ .,
                  data = GermanCredit[particionesEntrenamiento[[i]],
                      ],
                  method = "svmRadial")
  cat("Aprendido modelo ",i,"\n")

  # se realizan las predicciones sobre el conjunto de test
  # asociado
  # por supuesto es posible obtener las predicciones para las
  # instancias del conjunto de test
  predicciones <- predict(modelo, GermanCredit[particionesTest[[i]],
      ])

  # se determinan las diferencias entre prediccion y valor real de
  # la clase
  diferencias = (GermanCredit[particionesTest[[i]],"Class"] !=
      predicciones)
  errores[i] <- length(which(diferencias == TRUE))
  aciertos[i] <- length(which(diferencias == FALSE))
}

# se calcula la tasa de aciertos
tasaAciertos <- mean(aciertos/(errores+aciertos))

# el modelo general propuesto se construiria ahora con todos los
# datos
modeloFinal <- train(Class ~ .,
                     data = GermanCredit, method = "svmRadial")

# ********************** PARTE 2 *********************************

# tambien es posible aplicar algunas operaciones de preprocesamiento
# sobre los datos: centrado y escalado, por ejemplo, al emplear una
# tecnica de aprendizaje sensible a las dimensiones y escala de los
# datos
modelo2 <- train(Class ~ .,
```

```
67                    data = GermanCredit[particionesEntrenamiento[[1]],
                          ],
68                    method = "svmRadial", preProc = c("center", "scale")
                          )
69
70 # por supuesto es posible obtener las predicciones para las
71 # instancias del conjunto de test
72 predicciones <- predict(modelo2, GermanCredit[particionesTest[[1]],
       ])
73 str(predicciones)
74
75 # se determinan las diferencias entre prediccion y valor real de
76 # la clase
77 diferencias= (GermanCredit[particionesTest[[1]],"Class"] !=
       predicciones)
78 errores <- length(which(diferencias == TRUE))
79 aciertos <- length(which(diferencias == FALSE))
80
81 # ********************** PARTE 3 *********************************
82
83 # este metodo tiene un parametro de coste que regula el coste
       asociado
84 # a los errores de prediccion: las diferencias entre el valor
       predicho
85 # y el real. Es posible evaluar diferentes valores de coste
       directamente,
86 # haciendo uso del ultimo argumento (costes 2^-2, 2^-1,
87 # 2^0, .... 2^7)
88 modelo3 <- train(Class ~ .,
89                    data = GermanCredit[particionesEntrenamiento[[1]],
                          ],
90                    method = "svmRadial", preProc = c("center", "scale")
                          ,
91                    tuneLength = 10)
92
93 # y volvemos a hacer las predicciones
94 # por supuesto es posible obtener las predicciones para las
95 # instancias del conjunto de test
96 predicciones <- predict(modelo3, GermanCredit[particionesTest[[1]],
       ])
97 str(predicciones)
98
99 # se determinan las diferencias entre prediccion y valor real de
100 # la clase
101 diferencias= (GermanCredit[particionesTest[[1]],"Class"] !=
       predicciones)
102 errores <- length(which(diferencias == TRUE))
103 aciertos <- length(which(diferencias == FALSE))
104
105 # puede imprimirse la informacion de los modelos considerados
106 plot(modelo3, scales = list(x = list(log = 2)))
107
108 # ********************** PARTE 4 *********************************
109
110 # tambien se puede modificar esta llamada para que se utilicen
111 # diferentes particionados (que se generan de forma automatica
112 # por parte de la funcion de aprendizaje). En este ejemplo se
```

```
# realizan 5 repeticiones de validacion cruzada con k=10
# se inicializa la semilla (si interesa). De esta forma se
# automatiza de forma completo el proceso de aprendizaje y
# de estimacion. Se observa informacion sobre el particionado
# al analizar la salida del modelo
set.seed(1)
modelo4 <- train(Class ~ ., data = GermanCredit,
                 method = "svmRadial", preProc = c("center", "scale")
                     ,
                 tuneLength=10,
                 trControl = trainControl(method = "repeatedcv",
                                          repeats=5))

# se muestra la relacion entre el valor de C usado y la fiabilidad
# predictiva
plot(modelo4)

# ********************** PARTE 5 *********************************

# tambien es posible hacer comparacion con otros modelos. Imaginemos
# un modelo de regresion logistica (clasificacion) para estos datos.
# Seproduce la misma inicializacion de semilla
set.seed(1)
modeloRegLog <- train(Class ~ ., data=GermanCredit, method="glm",
                      trControl= trainControl(method="repeatedcv",
                                              repeats=5))

# la funcion resamples nos sirve para comparar estos modelos
resamp <- resamples(list(SVM = modelo4, Logistic = modeloRegLog))
summary(resamp)

# se obtienen las diferencias entre ambos modelos: supone realizar
# un test estadistico, siendo la hipotesis nula la igualdad (lo que
# indicaria que los modelos se comportan de forma similar)
diferencias <- diff(resamp)
summary(diferencias)

# Si los p-valores son muy bajos se acepta la hipotesis
# nula: los modelos son similares
```

El gráfico final obtenido que muestra el comportamiento del modelo frente al valor del parámetro es el siguiente (para el modelo 3):



En este gráfico se representan los valores mostrados al visualizar el objeto **modelo3** en la consola:

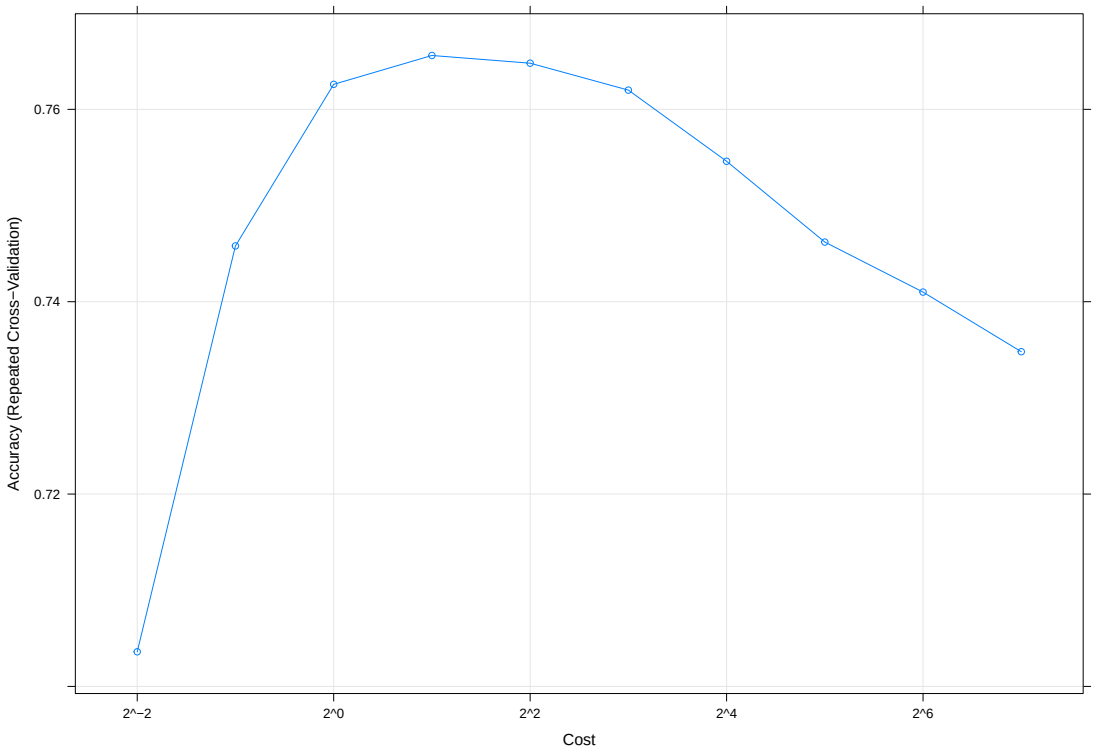
```
Support Vector Machines with Radial Basis Function Kernel

900 samples
 61 predictor
  2 classes: 'Bad', 'Good'

Pre-processing: centered (61), scaled (61)
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:

  C       Accuracy   Kappa      Accuracy SD  Kappa SD
    0.25  0.7239406  0.1331923  0.02300528   0.06145489
    0.50  0.7497717  0.3043597  0.02157638   0.05540984
    1.00  0.7540293  0.3608369  0.02457594   0.05247468
    2.00  0.7479025  0.3633155  0.02288386   0.04978457
    4.00  0.7432228  0.3640545  0.02125643   0.04756512
    8.00  0.7388221  0.3614371  0.01778897   0.04079194
   16.00  0.7302837  0.3453534  0.01464776   0.03818569
   32.00  0.7295740  0.3471028  0.01483300   0.03927109
   64.00  0.7303022  0.3487635  0.01512766   0.03884665
  128.00  0.7285113  0.3443282  0.01377525   0.03540629

Tuning parameter 'sigma' was held constant at a value of 0.01041675
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were sigma = 0.01041675 and C = 1.
```

El hecho de usar validación cruzada para general **modelo4** hace que la información

que aparece sobre él sea diferente en lo que se refiere a las muestras generadas para realizar la validación:

```
Support Vector Machines with Radial Basis Function Kernel

1000 samples
  61 predictor
   2 classes: 'Bad', 'Good'

Pre-processing: centered (61), scaled (61)
Resampling: Cross-Validated (10 fold, repeated 5 times)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:

  C       Accuracy  Kappa       Accuracy SD  Kappa SD
    0.25  0.7036    0.01969113  0.008020382  0.02906033
    0.50  0.7458    0.25467757  0.029352885  0.09687554
    1.00  0.7626    0.35206308  0.031024020  0.10008931
    2.00  0.7656    0.39075997  0.038500464  0.10413566
    4.00  0.7648    0.40050909  0.039187670  0.10865632
    8.00  0.7620    0.40658913  0.040808162  0.10741669
   16.00  0.7546    0.40014229  0.042146973  0.10237067
   32.00  0.7462    0.38538820  0.040852647  0.10152545
   64.00  0.7410    0.37625804  0.040870627  0.09664121
  128.00  0.7348    0.36359410  0.041019657  0.09814814

Tuning parameter 'sigma' was held constant at a value of 0.01025498
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were sigma = 0.01025498 and C = 2.
```

Para este modelo el gráfico indicativo para la evolución del comportamiento en función del valor de C es el siguiente:



Una forma sencilla de comparar modelos es mediante la función **resamples**. Hay que tener en cuenta que su uso implica que los modelos a comparar hayan sido ajustados considerando el mismo número de muestras. Esto no es así en el caso de **modelo3** y **modelo4**: el primero se obtiene tras analizar 25 particiones y el segundo tras estudiar 50. Los resultados obtenidos para cada una de ellas pueden consultarse de la siguiente forma:

```
modelo3$resample

   Accuracy    Kappa    Resample
1 0.7470930 0.3467214 Resample05
2 0.7272727 0.2927911 Resample09
3 0.7430341 0.3391101 Resample04
4 0.7732558 0.3898213 Resample08
5 0.7303030 0.3122102 Resample12
6 0.7484663 0.3647338 Resample03
7 0.7902736 0.4260032 Resample07
8 0.7461300 0.3581330 Resample11
9 0.7412141 0.3390255 Resample02
10 0.7580645 0.3589744 Resample06
11 0.7042683 0.2562184 Resample10
12 0.7721893 0.4279246 Resample01
13 0.7447447 0.3139346 Resample18
14 0.7689970 0.3594302 Resample22
15 0.7619048 0.3693694 Resample13
16 0.7560241 0.3878722 Resample17
17 0.7601246 0.3607066 Resample21
18 0.7938462 0.4445578 Resample25
19 0.7993921 0.4395519 Resample16
20 0.7645260 0.3991983 Resample20
21 0.7738095 0.4112873 Resample24
```

```
22 0.7057057 0.2701288 Resample15
23 0.7371429 0.3527378 Resample19
24 0.7280967 0.2845134 Resample23
25 0.7748538 0.4159681 Resample14
```

La salida mostrada al comparar modelos es:

```
Call:
summary.diff.resamples(object = diferencias)

p-value adjustment: bonferroni
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0

Accuracy
         SVM      Logistic
SVM                 0.0134
Logistic 0.08624

Kappa
         SVM      Logistic
SVM                 0.01494
Logistic 0.4562
```

En este caso en concreto, suponiendo un nivel de significación del 95 % se asume que no disponemos de información para asumir un comportamiento diferente por parte de ambos modelos (es decir, contra la hipótesis nula): el valor de $p-value$ está siempre por encima de 0,05.

Obviamente, el análisis se basa en que las particiones de prueba usadas sean similares (para evitar el efecto aleatorio). Para ello se ha hecho que el generador de números aleatorios se inicialice de la misma forma antes de lanzar el método de aprendizaje.

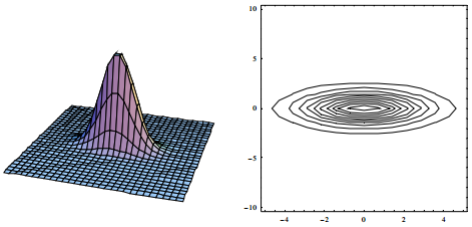
14. SVM (support vector machine)

Las máquinas de soporte vectorial suponen uno de los algoritmos más sofisticados y de buen funcionamiento en muchos problemas, aunque su herramienta matemática es muy compleja. La idea intuitiva es sencilla: se trata de buscar siempre el hiperplano que mejor separa las instancias en sus clases correspondientes. Si pensamos en dos dimensiones y en problemas donde hay separabilidad lineal es fácil de visualizar.

En la mayoría de los casos no es posible separar las instancias mediante un hiperplano. En estos casos lo que hace este método es intentar mejorar la separabilidad de las instancias agregando atributos artificiales que llevan las instancias (vistas como puntos) a un espacio de más dimensiones, donde deben ser más fácilmente separables.

Intuitivamente estas nuevas características a agregar al conjunto de datos representan la **distancia** a una serie de puntos de referencia que podrían fijarse de cualquier forma (es decir, instancias de referencia). En la práctica los puntos de referencia suelen ser las instancias en sí. De esta forma, la primera característica añadida representa la distancia a la primera instancia y así sucesivamente. De esta forma, en un conjunto con n variables y m muestras pasamos de tener un conjunto de datos de tamaño $m \times n$ a tener otro de dimensión $m \times (n+m)$ (al agregar las m nuevas características).

Otro punto crucial resulta la forma de definir la **distancia** entre instancias. Esta definición se hace a través de diferentes **kernels**. Por ejemplo, el **kernel gaussiano** define la distancia mediante una campana de **Gauss**.



Sin entrar en demasiados detalles técnicos nos limitaremos a usar algún paquete que aporta esta funcionalidad: **e1071**. Un buen enlace donde se explican algunas características de este paquete es el siguiente: `http://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Classification/SVM#Installing_and_Starting_the_e1071_Package`.

El **script** en que se muestra un ejemplo de uso de la funcionalidad de este paquete se denomina **e1071.R**.

```
library(e1071)
library(caret)

# se cargan los datos
data(iris)

# se crea la particion: esto obtiene de forma aleatoria un porcentaje
# de instancias dado por p
inTrain <- caret::createDataPartition(y = iris$Species, p = .75, list
    = FALSE)

# ahora se obtienen los conjuntos de test y de entrenamiento
training <- iris[ inTrain,]
testing  <- iris[-inTrain,]

# se construye el modelo
model <- e1071::svm(Species~., data=training, method="C-
    classification",
            kernel="radial", cost=10, gamma=0.1)

# se muestra informacion sobre el modelo
summary(model)

# se hace prediccion
pred <- predict(model, testing, decision.values = TRUE)

# para interpretarlo bien se trata como una tabla
tab <- table(pred = pred, true = testing[,5])

# se muestra la tabla
print(tab)
```

15. Árboles de clasificación

Un paquete sencillo de construcción de árboles de clasificación es **party**. La facilidad de uso se pone de manifiesto en el siguiente ejemplo de aplicación (contando además con la calidad de su representación gráfica, que puede ser interesante en determinadas situaciones). En el código se usa el paquete **caret** (que, como ya hemos indicado, ofrece

el método **createDataPartition**, muy cómodo para particionar el conjunto de datos, aunque hemos visto otras posibles formas de hacerlo). Básicamente en el script se hace lo siguiente:

- se leen los datos de los que aprender
- se obtiene la posición y el nombre de la variable clase. Ya dijimos que se asume que siempre ocupará la última columna
- se compone una fórmula con el nombre de la variable clase, el gorrito de la ñ y un punto, que indica al método de construcción de árboles que está interesado en construir un modelo de predicción de la variable clase frente al resto de variables
- se procede al particionado del conjunto de datos. Se observa que este método espera recibir un vector, por lo que basta pasarle como argumento la columna correspondiente a la variable clase (simplemente sirve para indicar el número de elementos con los que trabajará)
- una vez hecho esto se seleccionan las instancias de entrenamiento y de test y se construye el árbol, mediante el método **ctree**, que recibe como argumento la fórmula compuesta previamente
- una vez construido el modelo se hace la predicción sobre el conjunto de test y se generan los resultados

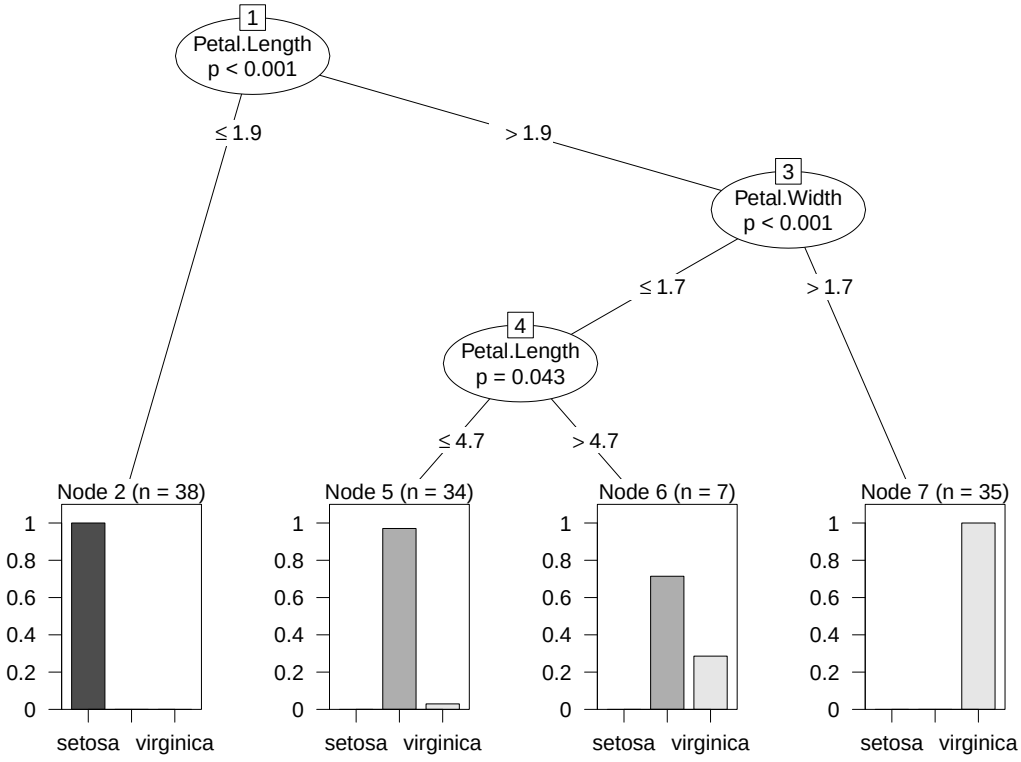
El **script** se denomina **arbolClasificacion.R**.

```
1  library(party)
2  library(caret)
3
4  # se carga la funcionalidad de lectura de datos
5  source("./lecturaDatos.R")
6
7  # se fija ls ruta de localizacion de los datos
8  path <- "./data/"
9  file <- "datos.csv"
10
11 # se llama a la funcion que carga los datos
12 datos <- lecturaDatos(path,file)
13
14 # se obtiene el nombre de la variable clase
15 # primero se obtiene la posicion de la variable clase: se asume
16 # que es la ultima (pero no podemos estar seguros)
17 posicionClase <- length(names(datos))
18 variableClase <- names(datos)[posicionClase]
19
20 # se compone una formula con el nombre de la variable clase y ~
21 formulaClase <- as.formula(paste(variableClase,"~.",sep=""))
22
23 # se divide el conjunto de datos en train y test
24 # se crea la particion: esto obtiene de forma aleatoria un porcentaje
25 # de instancias dado por p. Esta funcionalidad esta disponible en el
26 # paquete caret
27 inTrain <- createDataPartition(y=datos[,posicionClase], p = .75,
28                                list = FALSE)
29
```

```
30 # ahora se obtienen los conjuntos de test y de entrenamiento
31 training <- datos[ inTrain,]
32 testing  <- datos[-inTrain,]
33
34 # construye el modelo
35 ct <- ctree(formulaClase, training)
36
37 # muestra el arbol de forma grafica, pero no tiene demasiado sentido
38 # al ser demasiado grande
39 # plot(ct)
40
41 # se muestra en modo texto: observad que este metodo no precisa
42 # discretizacion previa al poder ir considerando diferentes cortes
43 # en variables numericas. Esto hace que puedan aparecer varias
44 # veces en el arbol
45 print(ct)
46
47 # se realiza la prediccion
48 testPred <- predict(ct, newdata = testing)
49
50 # se compara con los datos de test
51 results <- table(testPred, testing$class)
52
53 # se suman los valores de la diagonal
54 sumDiag <- sum(diag(results))
55
56 # se suman todos los valores de la matriz
57 sumTotal <- sum(results)
58
59 # se calcula el porcentaje de aciertos
60 fiabilidad <- sumDiag/sumTotal
61
62 # Se calcula el error
63 error <- 1-fiabilidad
64
65 # usamos un conjunto de datos mas sencillo para ver la forma
66 # que tendria el arbol generado
67 datos <- iris
68
69 # se obtiene el nombre de la variable clase
70 # primero se obtiene la posicion de la variable clase: se asume
71 # que es la ultima (pero no podemos estar seguros)
72 posicionClase <- length(names(datos))
73 variableClase <- names(datos)[posicionClase]
74
75 # se compone una formula con el nombre de la variable clase y ~
76 formulaClase <- as.formula(paste(variableClase,"~.",sep=""))
77
78 # se divide el conjunto de datos en train y test
79 # se crea la particion: esto obtiene de forma aleatoria un porcentaje
80 # de instancias dado por p. Esta funcionalidad esta disponible en el
81 # paquete caret
82 inTrain <- createDataPartition(y=datos[,posicionClase], p = .75,
83                                list = FALSE)
84
85 # ahora se obtienen los conjuntos de test y de entrenamiento
86 training <- datos[ inTrain,]
```

```
testing  <- datos[-inTrain,]

# construye el modelo
ct <- ctree(formulaClase, training)

# ahora si vale la pena visualizar el arbol
plot(ct)

# se visualiza la tabla con las predicciones realizadas
predicciones <- predict(ct, testing)

# tambien pueden estimarse las probabilidades de asignacion
# de cada instancias a cada clase
probabilidades <- predict(ct, testing, type="prob")
```

El uso de árboles con otros paquetes suele resultar también sencillo. En cualquier caso, y pensando en la implementación del algoritmo de eliminación de instancias con ruido, debemos ser capaces de manejar el árbol para realizar predicciones. En el *script* se muestra cómo puede dibujarse el árbol de clasificación construido. En el caso de conjuntos de datos de gran tamaño no tiene sentido el gráfico ya que será demasiado grande para poder verlo. En el caso de iris podría ser el siguiente:



16. Métodos ensamble de construcción de conjuntos de modelos

Los métodos **ensamble** intentan mejorar la capacidad predictiva ofrecida por los métodos de aprendizaje (habitualmente de construcción de árboles). Para conseguirlo, en vez de producir un único modelo se trata de crear un conjunto de ellos. La idea

subyacente es conseguir que cada uno de los modelos producidos trate con alguna peculiaridad específica de los datos, al haberse construido sobre una determinada partición del conjunto de datos.

A la hora de predecir, la decisión final de qué clase asignar se tomará al observar un conjunto de árboles y no uno sólo y componiendo la respuesta de alguna manera (mediante mayoría, por pesos....). En las aproximaciones más normales suelen construirse entre 100 y 500 árboles. Consideraremos aquí varias formas de construir el conjunto de modelos: **bagging**, **random forest** y **boosting**.

16.1. Bagging

El algoritmo básico de generación de modelos sería el siguiente:

· seleccionar el número de modelos a crear, m

· para cada modelo a construir

- generar una muestra usando **bootstrap** a partir de los datos originales
- generar un modelo (usualmente un árbol sencillo) a partir de esta muestra

· usar los m modelos para generar predicciones

El script **bagging.R** da un ejemplo de uso de esta técnica:

```
# la libreria rpart se usa para disponer de metodos de
# construccion de arboles de clasificacion
library(rpart)

# aqui hay datos considerados como de referencia
library(mlbench)

# libreria necesaria para funcionalidad de construccion
# de ensembles mediante bagging
library(adabag)

# se definen los datos a usar
data(Vehicle)

# se crean las particiones del conjunto de datos. En este caso
# se usa con conjunto de datos con 1000 instancias y 62
# variables. Se generan las particiones del conjunto de datos
# mediante la funcion createFolds, que genera 10 particiones
indices <- seq(1,nrow(Vehicle),by=1)
particionesEntrenamiento <- createFolds(indices, k = 10,
                                        returnTrain = TRUE)

# genero de la misma forma las particiones de test
particionesTest <- list()
for(i in 1:10){
  particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
}

# usaremos estas particiones para hacer validacion cruzada
# Bucle de generacion de modelos
errores <- c()
```

```
aciertos <- c()
for(i in 1:10){
  # hay varias formas de especificar el objetivo del modelo a
  # construir. Una de ellas es la formula: se compone de dos
  # terminos separados por el simbolo ~: a la izquierda va
  # la variable a predecir (clasificacion o regresion) y a la
  # derecha las variables a usar como predictoras. En el ejemplo
  # siguiente el punto a la derecha indica que se usan todos los
  # atributos como atributos. Como ejemplo veremos la aplicacion
  # del metodo de aprendizaje a una de las particiones
  modelo <- adabag::bagging(Class ~ .,
                    data = Vehicle[particionesEntrenamiento[[i]], ],
                    control=rpart::rpart.control(maxdepth=5, minsplit
                        =15))
  cat("Aprendido modelo ",i,"\n")

  # se realizan las predicciones sobre el conjunto de test
  # asociado. Por supuesto es posible obtener las predicciones
  # para las instancias del conjunto de test
  predicciones <- adabag::predict.bagging(modelo,
                                          newdata=Vehicle[
                                              particionesTest[[i]], ])

  # se determinan las diferencias entre prediccion y valor real de
  # la clase
  errores[i] <- predicciones$error
  aciertos[i] <- 1-predicciones$error
  cat("  Aciertos:", aciertos[i], " Errores: ", errores[i], "\n")
}

# se calcula la tasa de aciertos
tasaAciertos <- mean(aciertos/(errores+aciertos))

# usaremos estas particiones para hacer validacion cruzada
# Bucle de generacion de modelos
errores <- c()
aciertos <- c()
for(i in 1:10){
  # se cambian los parametros de control para la construccion de
  # los arboles. Aqui el modelo final contendra unicamente 5 arboles
  # y se limita su tam. a profuncidad maxima de tres
  modelo <- adabag::bagging(Class ~ .,
                    data = Vehicle[particionesEntrenamiento[[i]], ],
                    mfinal=20,
                    control=rpart::rpart.control(maxdepth=3, minsplit
                        =5))
  cat("Aprendido modelo ",i,"\n")

  # se realizan las predicciones sobre el conjunto de test
  # asociado. Por supuesto es posible obtener las predicciones
  # para las instancias del conjunto de test
  predicciones <- adabag::predict.bagging(modelo,
                                          newdata=Vehicle[
                                              particionesTest[[i]], ])

  # se determinan las diferencias entre prediccion y valor real de
  # la clase
```

```
85    errores[i] <- predicciones$error
86    aciertos[i] <- 1-predicciones$error
87    cat("  Aciertos:", aciertos[i], " Errores: ", errores[i], "\n")
88  }
89
90  # se calcula la tasa de aciertos
91  tasaAciertos <- mean(aciertos/(errores+aciertos))
```

16.2. Random forest

Se trata también de una técnica **ensemble** que genera un conjunto de árboles de decisión que se usan posteriormente para generar predicciones. El esquema del método de generación de modelos se presenta a continuación:

- seleccionar el número de modelos a crear, m
- para cada modelo a construir
  - generar una muestra usando **bootstrap** a partir de los datos originales
  - generar un modelo (usualmente un árbol) a partir de esta muestra. Para cada nuevo nodo a introducir:
    - seleccionar al azar k variables predictoras de las disponibles todavía (k menor que el número de variables disponibles al principio)
    - seleccionar para continuar el particionado el mejor predictor de los k seleccionados en el paso anterior
- usar los m modelos para generar predicciones

El objetivo de esta aleatoridad se basa en poder capturar situaciones en que una variable aislada es más informativa que las demás (y siempre sería seleccionada al principio), pero en combinación hay un par o una terna de variables que en conjunto son más informativas.

Se incluye un ejemplo de uso mediante el script **randomForest.R**:

```
1  library(caret)
2  library(randomForest)
3
4  # se carga el conjunto de datos
5  data("GermanCredit")
6
7  # se aprende el modelo: random forest. Podemos especificar el
8  # numero de arboles a incluir en el bosque
9  modelo <- randomForest::randomForest(Class ~ ., data=GermanCredit,
       ntree=10)
10
11 # se muestra informacion del modelo
12 print(modelo)
13
14 # muestra la importancia de los atributos, teniendo en cuenta
15 # el modelo construido
16 randomForest::importance(modelo)
17
18 # se muestra informacion sobre los errores para cada una de las
```

```
19 # clases: la linea en negro indica el error medio
20 plot(modelo)
21
22 # se aprende otrol modelo con mas arboles
23 modelo2 <- randomForest(Class ~ ., data=GermanCredit, ntree=100)
24
25 # se muestran los errores para cada etiqueta de la variable clase
26 plot(modelo2)
```

16.3. Boosting

Consiste en tomar como método de aprendizaje base alguno que sea sencillo y construir muchos modelos diferentes. Lo específico de este método es que asigna pesos a las instancias, de forma que aquellas que sean más difíciles de clasificar tendrán mayor peso (y por tanto mayor probabilidad de ser elegidas en la muestra **bootstrap**). Con esta idea, se van construyendo modelos, en forma secuencial, y tras construirlo se repesan las instancias en función de cómo han sido clasificadas. De esta forma los pesos irán creciendo y disminuyendo a lo largo del proceso de aprendizaje, hasta alcanzar alguna condición de parada. Lo notable es que en cada iteración se pone mayor interés en clasificar de forma correcta las instancias más difíciles de predecir (las instancias difíciles de clasificar entrarán con mayor probabilidad en los conjuntos de entrenamiento).

El modelo final se obtiene como la agregación de todos los modelos considerados y cada uno de ellos, a su vez, participa en la votación final a la hora de clasificar nuevas instancias. En este modelo no todos los árboles tienen el mismo peso y, de esta forma, el voto de algunos de ellos será más influyente. El esquema básico de funcionamiento sería el siguiente:

- asignar pesos uniformes a las muestras y a los modelos
- para cada modelo a construir
  - generar una muestra usando **bootstrap** a partir de los datos originales y los pesos disponibles
  - generar un modelo (usualmente un árbol) a partir de esta muestra
  - asignar al modelo un peso según el número de errores
  - asignar nuevos pesos a las muestras en función de la predicción
- usar los m modelos para generar predicciones

Por la naturaleza de este método los modelos que forman parte del conjunto no son independientes y el proceso no resulta paralelizable.

Uno de los paquetes que parece más sencillo de usar para **boosting** es **adabag**. Se incluye un ejemplen el que se ha trabaja sobre el conjunto de datos **iris** para que sea menos pesada su aplicación. El **script** se denomina **boosting.R**.

```
1 library(adabag)
2 library(caret)
3
4 # se carga el conjunto de datos
5 data(Vehicle)
6
```

```
# se hace un particionado sencillo del conjunto de datos
indices <- seq(1,nrow(Vehicle),by=1)
indicesEntrenamiento <- caret::createDataPartition(indices, p=0.8, list=FALSE)
datosEntrenamiento <- Vehicle[indicesEntrenamiento, ]
datosTest <- Vehicle[-indicesEntrenamiento, ]

# se realiza el aprendizaje. El argumento mfinal indica el numero
# de iteraciones del proceso, o lo que es lo mismo, el numeor de
# modelos que se construyen. El argumento maxdepth se usa por parte
# del paquete rpart, que es el usado para construir los arboles
# del ensamblado
modelo <- adabag::boosting(Class ~ ., data = datosEntrenamiento,
                           mfinal = 10,
                           control = rpart::rpart.control(maxdepth = 2))

# se muestra un grafico indicando la importancia relativa
# de las variables
barplot(modelo$imp[order(modelo$imp, decreasing = TRUE)],
        ylim = c(0, 100), main = "Variables Relative Importance",
        col = "lightblue")

# la prediccion se realizaria de la siguiente forma
prediccion <- predict.boosting(modelo, newdata = datosTest)

# se muestra el resultado. Se observa que la prediccion es de tipo probabilistico.
# Para cada una de las 36 instancias del conjunto de test se muestra la probabilidad
# de pertenencia a cada clase
print(modelo)

# se calcula el error sobre el conjunto de test
evol.test <- errorevol(modelo,newdata=datosTest)

# se calcula el error sobre el conjunto de entrenamiento
evol.train <- errorevol(modelo,newdata=datosEntrenamiento)

# se muestra la evolucion del error a medida que se construyen
# mas arboles
plot(evol.test$error, type="l", main="AdaBoost error Vs numero arboles",
     xlab="Arboles", ylab="Error", col = "red")

# igual con el error en el conjunto de entrenamiento
plot(evol.train$error, col="blue",type="l",
     main="AdaBoost error Vs numero arboles",
     xlab="Arboles", ylab="Error")
```

Parte III
Apéndice: construcción de árboles de clasificación

17. Algoritmo TDIDT

El algoritmo básico que define el principio general de construcción de árboles de clasificación se conoce como **TDIDT: top down induction of decision trees**. Este es el procedimiento básico que siguen la mayoría de las implementaciones concretas, como **C4.5** y similares.

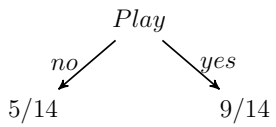
El método se apoya en el cálculo de alguna medida de información sobre los diferentes árboles alternativos que se van considerando. Y la medida básica considerada es la entropía, definida de la siguiente forma:

$$E(X_i) = \sum_{j=1}^{K} -p_j log p_j \tag{1}$$

donde se asume que la variable X_i para la que se calcula tiene k casos diferentes. A modo de ejemplo consideraremos la forma en que se construye el árbol de clasificación para el conjunto de datos mostrado en la tabla siguiente:

Instancia	Outloook	Temperature	Humidity	Wind	Play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	overcast	cool	normal	strong	yes
7	sunny	mild	high	weak	no
8	sunny	cool	normal	weak	yes
9	rain	mild	normal	weak	yes
10	sunny	mild	normal	strong	yes
11	overcast	mild	high	strong	yes
12	overcast	hot	normal	weak	yes
13	rain	mild	high	strong	no
14	rain	cool	normal	strong	no

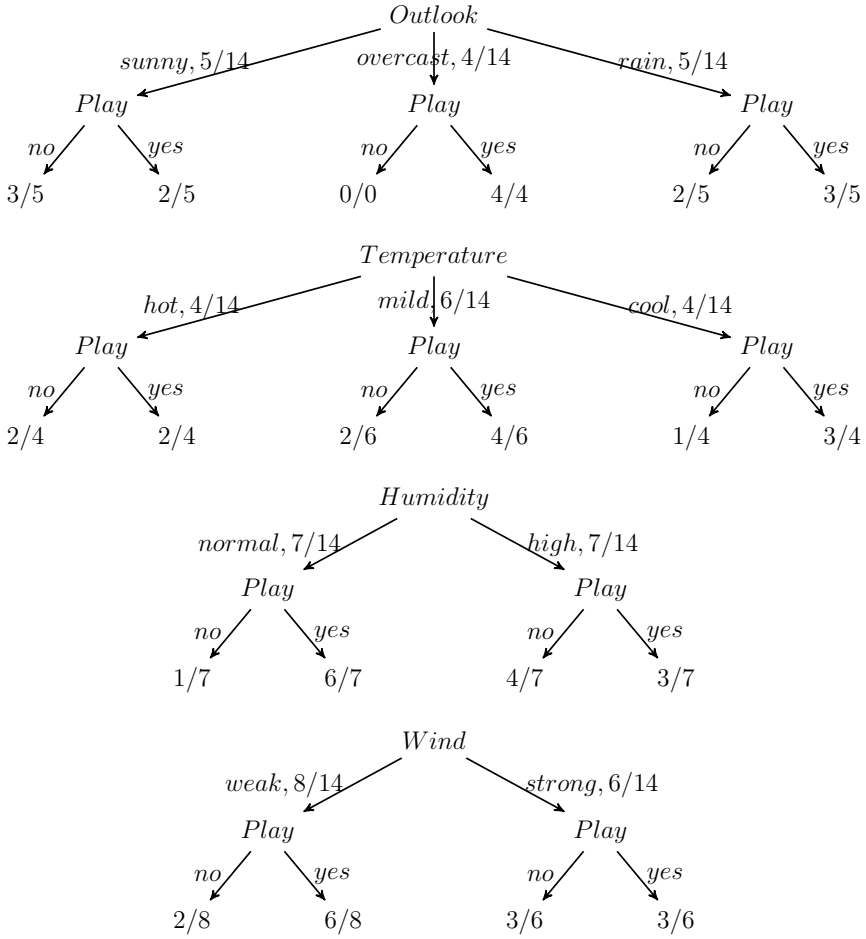
Comenzamos considerando el árbol más sencillo posible: no disponemos de ningún atributo y sólo contamos con la información del número de veces en que aparece cada etiqueta de la variable clase. Esto se corresponde con el siguiente modelo:



La forma de valorar el modelo inicial consiste en calcular la entropía asociada a la variable clase (**Play**):

$$E(Play) = -\frac{5}{14}log\frac{5}{14} - \frac{9}{14}log\frac{9}{14} = 0{,}9402$$

Aquí se ha usado el logaritmo en base 2, pero a efectos prácticos es indiferente, ya que todo el rato estaremos comparando entropías entre sí. Dada esta medida, se trata ahora de considerar qué ganamos al ir introduciendo atributos, uno a uno, en este modelo inicial. O, lo que es lo mismo, calcular qué ganancia de información ofrece cada uno de los atributos disponibles. Como se dispone de 4 atributos, hay que evaluar los 4 modelos alternativos mostrados a continuación:



La valoración de estos modelos implica el cálculo de su entropía. Consideremos en primer lugar el árbol donde **Outlook** está en la raíz. Se calcula de forma separada la entropía de la variable clase para cada valor de **Outlook**:

· caso 1: **Outlook=sunny**

$$E(Play|Outlook = sunny) = -\frac{3}{5}log\frac{3}{5} - \frac{2}{5}log\frac{2}{5} = 0{,}9709$$

· caso 2: **Outlook=overcast**

$$E(Play|Outlook = overcast) = -\frac{0}{0}log\frac{0}{0} - \frac{4}{4}log\frac{4}{4} = 0$$

· caso 3: **Outlook=rain**

$$E(Play|Outlook = rain) = -\frac{2}{5}log\frac{2}{5} - -\frac{3}{5}log\frac{3}{5} = 0{,}9709$$

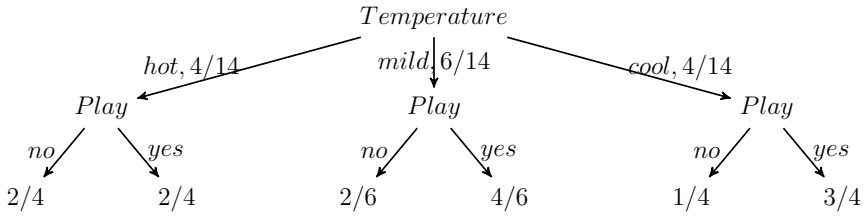
Con estas tres medidas de entropía hemos de componer una medida global para el modelo. Esto se hace considerando el número de instancias que corresponden a cada valor de la variable **Outlook**:

$$\begin{aligned} E(Play|Outlook) = \frac{5}{14}E(Play|Outlook = sunny) + \\ \frac{4}{14}E(Play|Outlook = overcast) + \\ \frac{5}{14}E(Play|Outlook = rain) = 0{,}6935 \end{aligned}$$

En este momento podemos comparar los dos modelos (inicial, con **Play** en la raíz y alternativo, con **Outlook** en la raíz). La comparación me permite calcular la ganancia de información obtenida al considerar la variable **Outlook**. La ganancia de información (**IG**) se calcula comparando las entropías de los modelos:

$$IG(Outlook) = E(Play) - E(Play|Outlook) = 0{,}2467$$

Se hace ahora lo mismo con el segundo modelo alternativo, aquel que considera la variable **Temperature** en la raíz:



De nuevo se considera de forma aislada cada valor de la variable (para que el cálculo resulte más claro):

· caso 1: **Temperature=hot**

$$E(Play|Temperature = hot) = -\frac{2}{4}log\frac{2}{4} - \frac{2}{4}log\frac{2}{4} = 1$$

· caso 2: **Temperature=mild**

$$E(Play|Temperature = mild) = -\frac{2}{6}log\frac{2}{6} - -\frac{4}{6}log\frac{4}{6} = 0{,}9183$$

· caso 3: **Temperature=cool**

$$E(Play|Temperature = cool) = -\frac{1}{4}log\frac{1}{4} - \frac{3}{4}log\frac{3}{4} = 0{,}8113$$

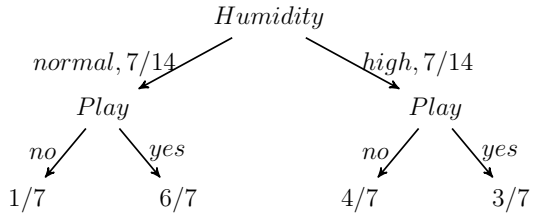
Componiendo una medida global para el modelo:

$$E(Play|Temperature) = \frac{4}{14}E(Play|Temperature = hot) + \frac{6}{14}E(Play|Temperature = mild) + \frac{4}{14}E(Play|Temperature = cool) = 0{,}9111$$

De esta forma, la ganancia de información asociada a **Temperature** será:

$$IG(Temperature) = E(Play) - E(Play|Temperature) = 0{,}0292$$

Le toca ahora el turno al modelo con **Humidity** en su raíz:



Calculamos la entropía para cada rama y componemos la medida final del modelo:

· caso 1: **Humidity=normal**

$$E(Play|Humidity = normal) = -\frac{1}{7}log\frac{1}{7} - \frac{6}{7}log\frac{6}{7} = 0{,}5917$$

· caso 2: **Humidity=high**

$$E(Play|Humidity = high) = -\frac{4}{7}log\frac{4}{7} - -\frac{3}{7}log\frac{3}{7} = 0{,}9852$$

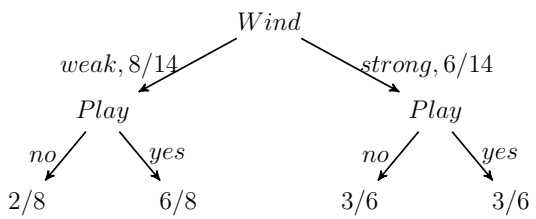
La medida global será:

$$E(Play|Humidity) = \frac{7}{14}E(Play|Humidity = normal) + \frac{7}{14}E(Play|Humidity = high) = 0{,}7884$$

De esta forma, la ganancia de información será:

$$IG(Humidity) = E(Play) - E(Play|Humidity) = 0{,}1518$$

Y se procede de la misma forma con el último modelo alternativo:



Calculamos la entropía para cada rama y componemos la medida final del modelo:

· caso 1: **Wind=weak**

$$E(Play|Wind = weak) = -\frac{2}{8}log\frac{2}{8} - \frac{6}{8}log\frac{6}{8} = 0{,}8113$$

· caso 2: **Wind=strong**

$$E(Play|Wind = strong) = -\frac{3}{6}log\frac{3}{6} - -\frac{3}{6}log\frac{3}{6} = 1$$

La medida global será:

$$E(Play|Wind) = \frac{8}{14}E(Play|Wind = weak) + \frac{6}{14}E(Play|Wind = strong) = 0{,}8921$$

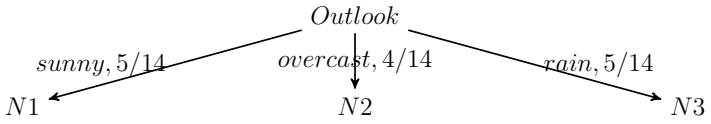
De esta forma, la ganancia de información será:

$$IG(Wind) = E(Play) - E(Play|Wind) = 0{,}0481$$

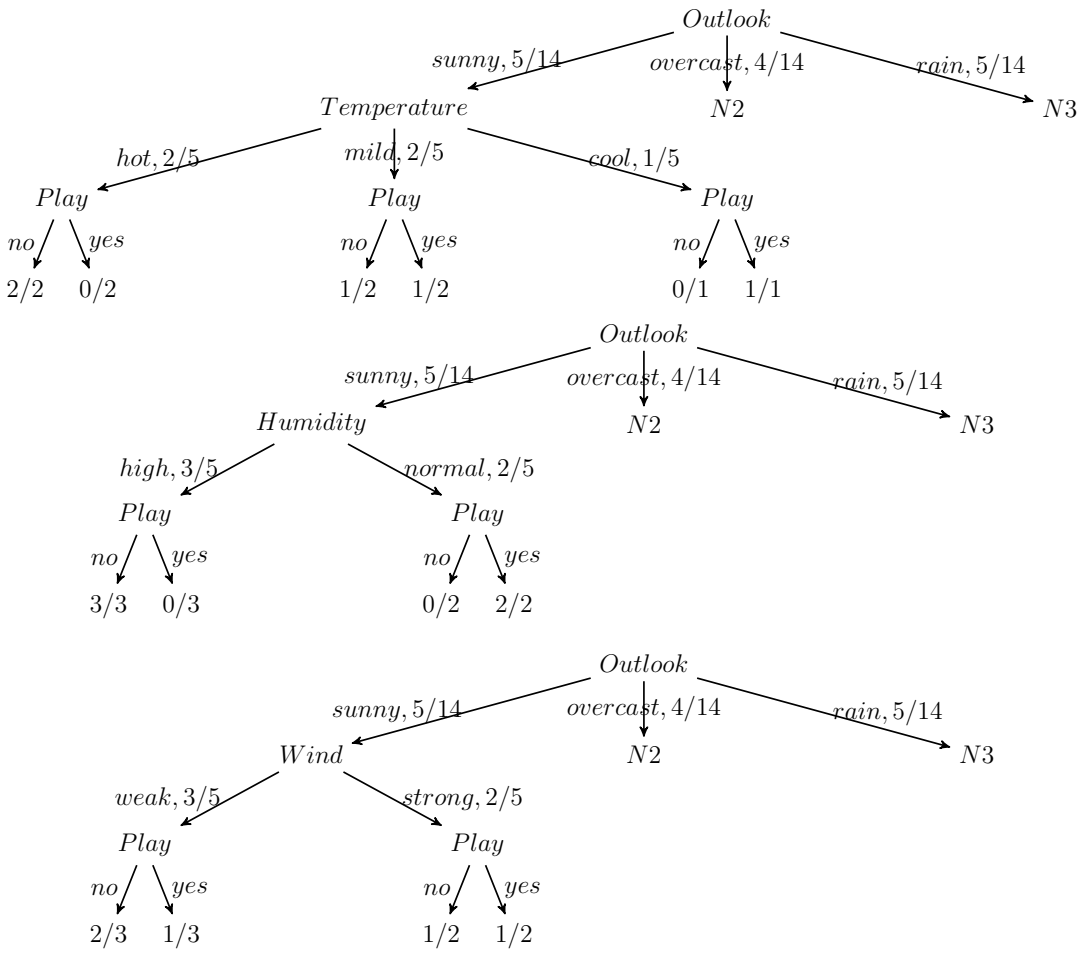
Ahora podemos comparar las ganancias asociadas a cada variable y elegir aquella que más información aporte sobre la variable clase:

$$\begin{aligned} IG(Outlook) &= 0{,}2467 \\ IG(Temperature) &= 0{,}0292 \\ IG(Humidity) &= 0{,}1518 \\ IG(Wind) &= 0{,}0481 \end{aligned}$$

Al ser **Outlook** la variable con más ganancia de información, se elige para formar parte del modelo, en la raíz (al ser la más informativa a la hora de clasificar las instancias de acuerdo a la variable clase). En este momento el modelo con el que seguir trabajando será:



donde $N1$, $N2$ y $N3$ son nodos que hay que seguir expandiendo hasta que el modelo esté completo. Imaginemos que consideramos ahora la expansión del nodo $N1$. El punto (medida) de referencia será el correspondiente a la situación descrita por $E(Play|Outlook = sunny) = 0{,}9709$. Este nodo podría asignarse a cualquiera de las variables restantes: **Temperature**, **Humidity**, **Wind**. Es decir, que la expansión de $N1$ implica considerar ahora tres modelos:



Para el primer modelo los cálculos a realizar son los considerados antes. Se comienza con la entropía:

· caso 1: **Outlook=sunny, Temperature=hot**

$$E(Play|Outlook = sunny, Temperature = hot) = -\frac{2}{2}log\frac{2}{2} - \frac{0}{2}log\frac{0}{2} = 0$$

· caso 2: **Outlook=sunny, Temperature=mild**

$$E(Play|Outlook = sunny, Temperature = mild) = -\frac{1}{2}log\frac{1}{2} - \frac{1}{2}log\frac{1}{2} = 1$$

· caso 3: **Outlook=sunny, Temperature=cool**

$$E(Play|Outlook = sunny, Temperature = cool) = -\frac{0}{1}log\frac{0}{1} - -\frac{1}{1}log\frac{1}{1} = 0$$

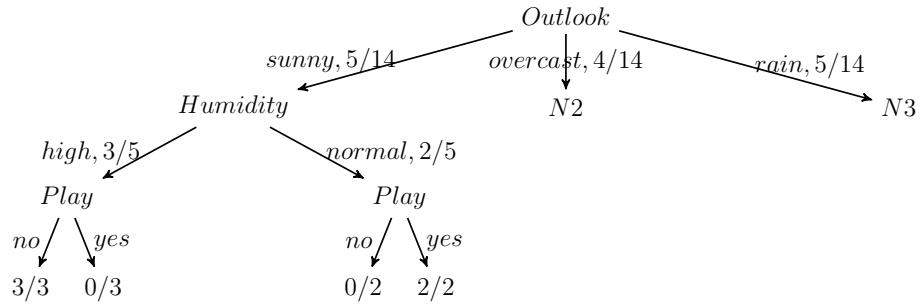
La medida global será:

$$E(Play|Outlook = sunny, Temperature) = \frac{2}{5}E(Play|Outlook = sunny, Temperature = hot) + \frac{2}{5}E(Play|Outlook = sunny, Temperature = mild) + \frac{1}{5}E(Play|Outlook = sunny, Temperature = cool) = 0{,}4$$

De esta forma, la ganancia de información es:

$$IG(Temperature|Outlook = sunny) = E(Play|Outlook = sunny) - E(Play|Outlook = sunny, Temperature) = 0{,}5709$$

El segundo modelo considera la forma de expandir $N1$ con la variable **Humidity**:



Cálculo de entropía:

· caso 1: **Outlook=sunny, Humidity=high**

$$E(Play|Outlook = sunny, Humidity = high) = -\frac{3}{3}log\frac{3}{3} - \frac{0}{3}log\frac{0}{3} = 0$$

· caso 2: **Outlook=sunny, Humidity=normal**

$$E(Play|Outlook = sunny, Humidity = normal) = -\frac{0}{2}log\frac{0}{2} - \frac{2}{2}log\frac{2}{2} = 0$$

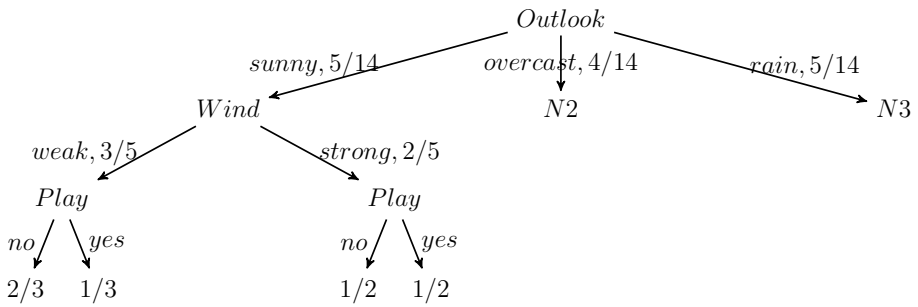
La medida global será:

$$E(Play|Outlook = sunny, Humidity) = \frac{2}{5}E(Play|Outlook = sunny, Humidity = high) + \frac{1}{5}E(Play|Outlook = sunny, Humidity = normal) = 0$$

Ganancia de información:

$$IG(Humidity|Outlook = sunny) = E(Play|Outlook = sunny) - E(Play|Outlook = sunny, Humidity) = 0{,}9709$$

Y finalmente, se analiza el tercer modelo:



Cálculo de entropía:

· caso 1: **Outlook=sunny, Wind=weak**

$$E(Play|Outlook = sunny, Wind = weak) = -\frac{2}{3}log\frac{2}{3} - \frac{1}{3}log\frac{1}{3} = 0{,}9183$$

· caso 2: **Outlook=sunny, Wind=strong**

$$E(Play|Outlook = sunny, Wind = strong) = -\frac{1}{2}log\frac{1}{2} - \frac{1}{2}log\frac{1}{2} = 1$$

La medida global será:

$$E(Play|Outlook = sunny, Wind) = \frac{3}{5}E(Play|Outlook = sunny, Wind = weak) + \frac{2}{5}E(Play|Outlook = sunny, Wind = strong) = 0{,}9509$$

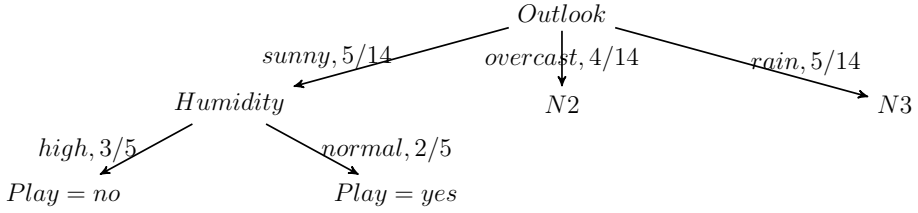
Ganancia de información:

$$IG(Wind|Outlook = sunny) = E(Play|Outlook = sunny) - E(Play|Outlook = sunny, Wind) = 0{,}02$$

Al comparar los modelos, tenemos:

$$IG(Temperature|Outlook = sunny) = 0{,}5709$$
$$IG(Humidity|Outlook = sunny) = 0{,}97095$$
$$IG(Wind|Outlook = sunny) = 0{,}02$$

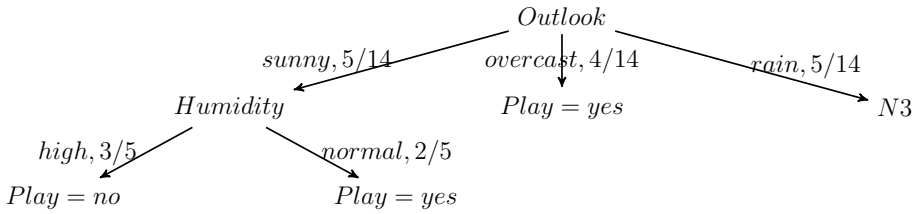
De esta manera, el nodo $N1$ se expande para particionar mediante la variable **Humidity**. Esta expansión tiene además otra propiedad: ya no precisa de más análisis. Es decir, da lugar a la aparición de dos nodos terminales: en el de la izquierda todas las instancias pertenecen a la clase **no** y en el de la derecha todas pertenecen a la clase **yes**:



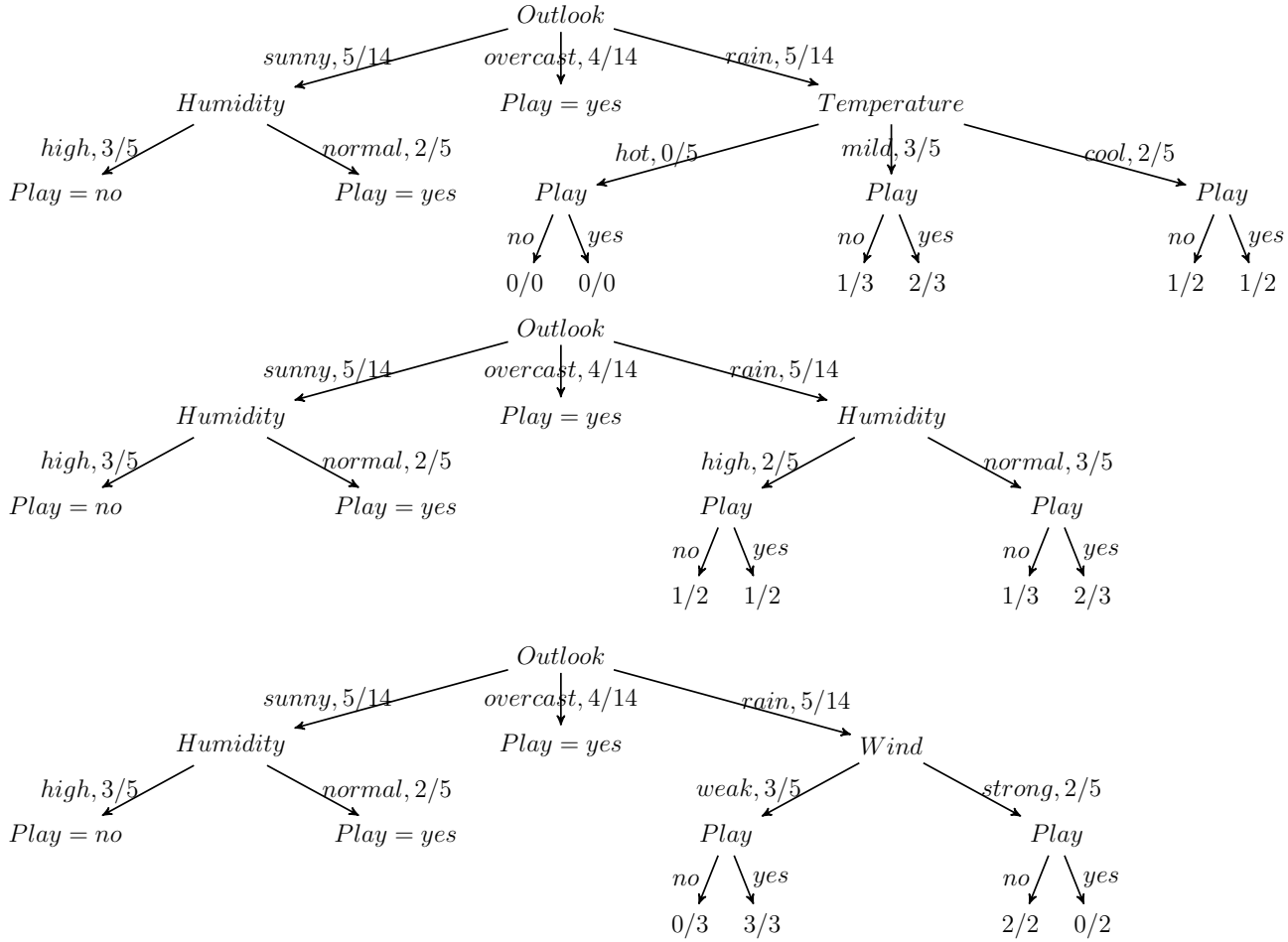
En este momento del proceso de construcción del árbol hay dos nodos pendientes de análisis: $N2$ y $N3$. En el caso de $N2$ la posición de partida es:

$$E(Play|Outlook = overcast) = -\frac{0}{0}log\frac{0}{0} - \frac{4}{4}log\frac{4}{4} = 0$$

Esto implica que no hay necesidad de investigar más, ya que todas las instancias pertenecen a la la misma clase: **Play**=**yes**. El árbol obtenido hasta este momento es:



El único nodo por expandir es $N3$ y los modelos alternativos a considerar son los siguientes:



Para el primero de ellos, donde el nodo $N3$ se usa para particionar los datos de acuerdo a los valores de la variable **Temperature**, el valor de entropía inicial es:

$$E(Play|Outlook = rain) = -\frac{2}{5}log\frac{2}{5} - -\frac{3}{5}log\frac{3}{5} = 0{,}9709$$

Hemos de calcular la ganancia de información obtenida al incluir la variable **Temperature**:

· caso 1: **Outlook=rain, Temperature=hot** (no hay instancias para esta combinación de valores de los atributos)

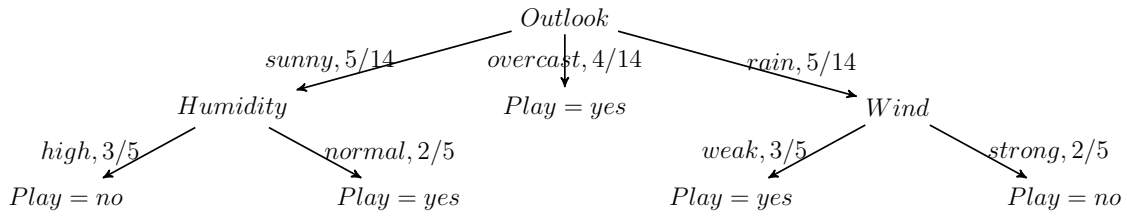
$$E(Play|Outlook = rain, Temperature = hot) = 0$$

· caso 2: **Outlook=rain, Temperature=mild**

$$E(Play|Outlook = rain, Temperature = mild) = -\frac{1}{3}log\frac{1}{3} - \frac{2}{3}log\frac{2}{3} = 0{,}9183$$

· caso 3: **Outlook=rain, Temperature=cool**

$$E(Play|Outlook = rain, Temperature = cool) = -\frac{1}{2}log\frac{1}{2} - -\frac{1}{2}log\frac{1}{2} = 1$$



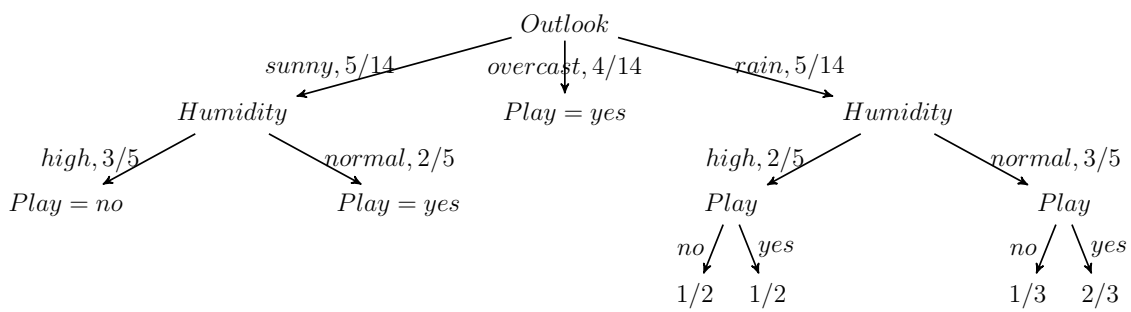
La medida global será:

$$E(Play|Outlook = rain, Temperature) = \frac{0}{5}E(Play|Outlook = rain, Temperature = hot) + \frac{3}{5}E(Play|Outlook = rain, Temperature = mild) + \frac{2}{5}E(Play|Outlook = rain, Temperature = cool) = 0{,}951$$

De esta forma, la ganancia de información es:

$$IG(Temperature|Outlook = rain) = E(Play|Outlook = rain) - E(Play|Outlook = rain, Temperature) = 0{,}02$$

La segunda alternativa de expansión para $N3$ consiste en particionar por **Humidity**:



Se calcula la ganancia de información obtenida al incluir la variable **Humidity**:

· caso 1: **Outlook=rain, Humidity=high**

$$E(Play|Outlook = rain, Humidity = mild) = -\frac{1}{2}log\frac{1}{2} - \frac{1}{2}log\frac{1}{2} = 1$$

· caso 2: **Outlook=rain, Humidity=normal**

$$E(Play|Outlook = rain, Humidity = normal) = -\frac{1}{3}log\frac{1}{3} - \frac{2}{3}log\frac{2}{3} = 0{,}9183$$

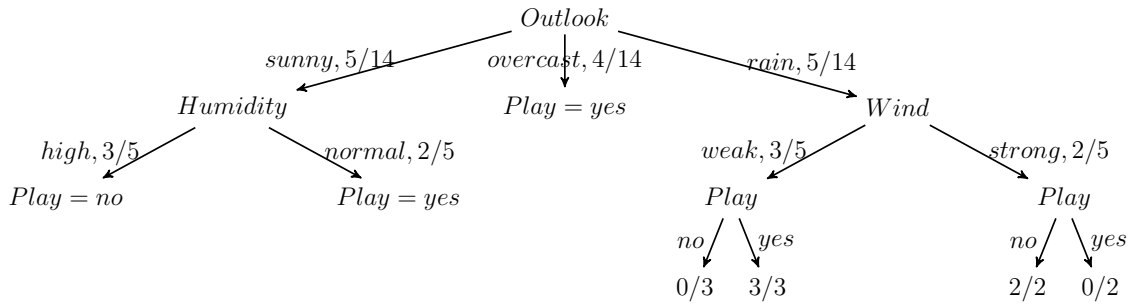
La medida global será:

$$E(Play|Outlook = rain, Humidity) = \frac{2}{5}E(Play|Outlook = rain, Humidity = high) + \frac{3}{5}E(Play|Outlook = rain, Humidity = normal) = 0{,}9510$$

De esta forma, la ganancia de información es:

$$IG(Humidity|Outlook = rain) = E(Play|Outlook = rain) - E(Play|Outlook = rain, Humidity) = 0{,}02$$

La última alternativa consistirá en usar $N3$ para particionar mediante **Wind**:



Se calcula la ganancia de información obtenida al incluir la variable **Wind**:

· caso 1: **Outlook=rain, Wind=weak**

$$E(Play|Outlook = rain, Wind = weak) = -\frac{0}{3}log\frac{0}{3} - \frac{3}{3}log\frac{3}{3} = 0$$

· caso 2: **Outlook=rain, Wind=strong**

$$E(Play|Outlook = rain, Wind = strong) = -\frac{2}{2}log\frac{2}{2} - \frac{0}{2}log\frac{0}{2} = 0$$

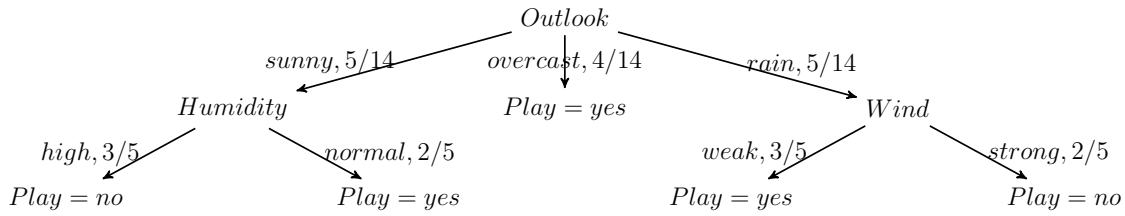
La medida global será:

$$E(Play|Outlook = rain, Wind) = \frac{3}{5}E(Play|Outlook = rain, Wind = weak) + \frac{2}{5}E(Play|Outlook = rain, Wind = strong) = 0$$

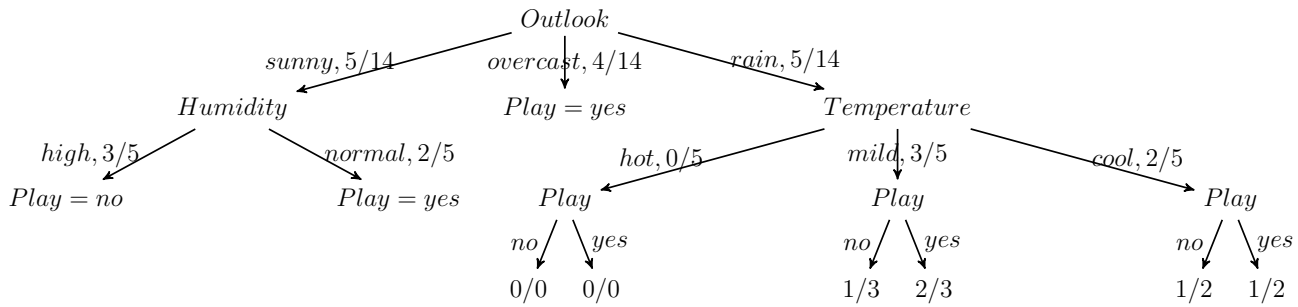
De esta forma, la ganancia de información es:

$$IG(Wind|Outlook = rain) = E(Play|Outlook = rain) - E(Play|Outlook = rain, Wind) = 0{,}9709$$

El árbol final obtenido del proceso completo de aprendizaje es:



Imaginemos ahora que el nodo $N3$ hubiese sido usado para particionar mediante la variable $Temperature$:



En este caso, la pregunta es, ¿qué hacer con la rama correpondiente a **Temperature=hot**? Vemos que no hay instancias para él. Hay varias posibles formas de tratar este caso. Una de ellas consiste en asignar a este nodo la etiqueta de la clase mayoritaria del nodo padre. Es decir, se examinan las 5 instancias en que **Outlook=rain** y se determina cuál es la clase mayoritaria:

Instancia	Outloook	Temperature	Humidity	Wind	Play
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
9	rain	mild	normal	weak	yes
13	rain	mild	high	strong	no
14	rain	cool	normal	strong	no

Se aprecia en la tabla correspondiente a este situación que la clase mayoritaria es *yes*, que sería la asignada a la rama de 0 instancias. El resto de nodos tendrían que seguir analizándose para considerar el particionado con las variables restantes.

En este ejemplo hemos considerado la medida conocida como ganancia de información. Pero hay que conocer que existen otras posibles medidas. Algunas de ellas son:

· **Information gain ratio (IGR)**: esta medida intenta suplir un sesgo de la ganancia de información, constatado de forma experimental: tiende a favorecer la partición por aquellas variables que más estados poseen. La forma de evitar el sesgo consiste en compensar la ganancia de información con una medida adicional denominada **intrinsic information (II)**. De esta forma:

$$IGR(X) = \frac{IG(X)}{II(X)} \tag{2}$$

donde $II(X)$ nos da idea de la forma en que se particionan los datos de acuerdo a la variable X. Imaginemos que la variable X posee dos estados y que en el conjunto de datos a analizar hay 17 instancias para el primer valor y 13 para el segundo. Entonces:

$$II(X) = -\frac{17}{30}log\frac{17}{30} - -\frac{13}{30}log\frac{13}{30}$$

· **Gini index (GI)**: considera los valores de probabilidad, pero al cuadrado:

$$GI(X) = 1 - \sum_{j=1}^{k} p_j^2 \tag{3}$$