Tutorial 13: Regresión logística.

Atención:

- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirlo, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 19 de abril de 2017. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

Índice

1. Construcción de un modelo de regresión logística con R. **1**

2. Las curvas logísticas. **8**

3. Estimando los parámetros. **9**

4. Regresión logística: b_0 y b_1 mediante maxima verosimilitud. **11**

5. Inferencia en regresión logística. **18**

6. Problemas de clasificación. **21**

7. Bondad del ajuste en la regresión logística. **35**

1. Construcción de un modelo de regresión logística con R.

En esta sección vamos a ver cómo utilizar R para obtener un modelo de regresión logística, partiendo de datos como los del Ejemplo 13.1.1 (pág. 500) del libro. Así que empezaremos por leer esos datos a partir del fichero que los contiene:

```
datosFichero = read.table("../datos/Cap13-DatosVasculopatia.csv",
                          header = TRUE, sep=",")
head(datosFichero)
```

```
##    ITB Vasculopatia
## 1 0.94            0
## 2 0.99            0
## 3 0.88            1
## 4 0.64            1
## 5 1.25            0
## 6 0.50            1
```

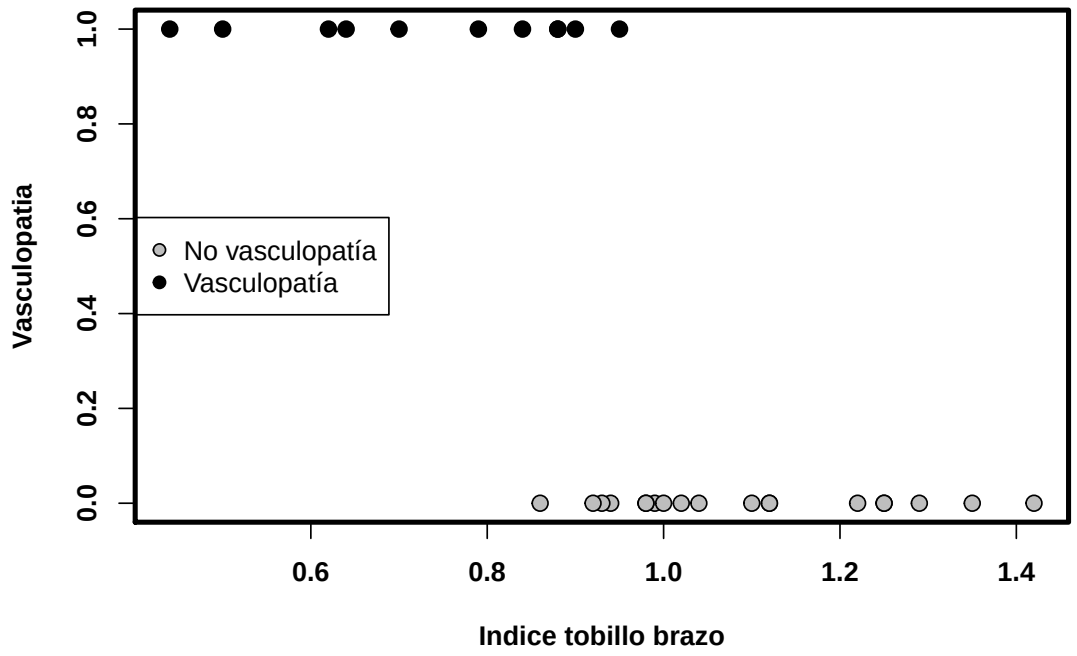
El resultado de la lectura es un `data.frame` de R en el que tenemos dos variables (columnas), llamadas `ITB` y `Vasculopatia`, respectivamente. En lo que sigue vamos a utilizar el nombre X para la variable explicativa e Y para la variable respuesta, que es un factor. Suponemos que X aparece en la primera columna del `data.frame` mientras que `Y` ocupa la segunda columna. Como en el caso del Anova, puede suceder que el orden de columnas en el fichero `csv` no coincida con

este. Pero en cualquier caso, podemos ajustar ese comportamiento cambiando a 2 el valor de la variable `colX`.

```
colX = 1
if(colX == 1){
  X = datosFichero[ , 1]
  Y = datosFichero[ , 2]
} else {
  X = datosFichero[ , 2]
  Y = datosFichero[ , 1]
}
datos = data.frame(X, Y)
```

Ya tenemos los datos listos para el análisis. El diagrama de dispersión correspondiente se obtiene con:

```
colores = c()
colores[datos$Y == 0] = "grey"
colores[datos$Y == 1] = "black"
plot(datos$X, datos$Y, pch = 21, bg = colores, cex=1.3,font=2,
     xlab = "Indice tobillo brazo", ylab = "Vasculopatia", font.lab=2, )
legend("left", c("No vasculopatía", "Vasculopatía"), pch = 21,
       pt.bg =  c("grey", "black"))
box(lwd=3)
```

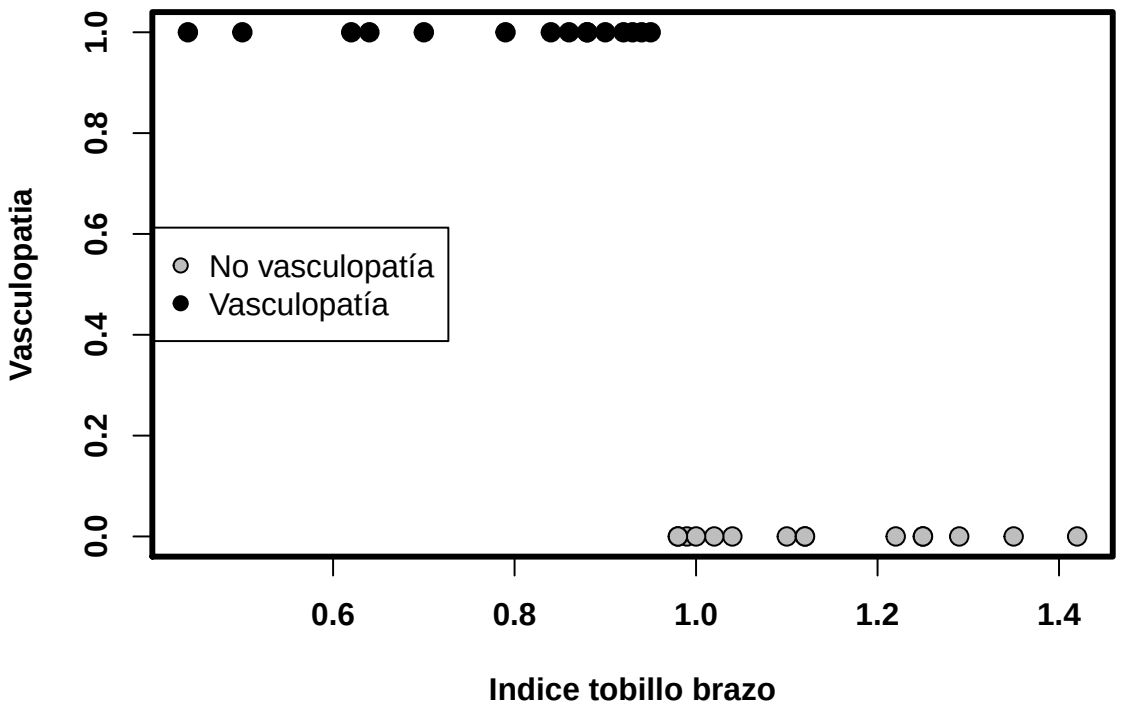


En la Sección 13.1 del libro (pág. 500) hemos visto también un modelo ficticio en el que existe un umbral claramente definido de valores de X (por ejemplo, $X = 0.96$) a partir del cual todos los valores de Y son 0, mientras que antes son 1. Por si te interesa, esa situación se puede reproducir fácilmente en el código así:

```
Ysimul01 = ifelse(X > 0.96, yes = 0, no = 1)
datosSimul01 = data.frame(X, Y = Ysimul01)
```

Y entonces su diagrama es:

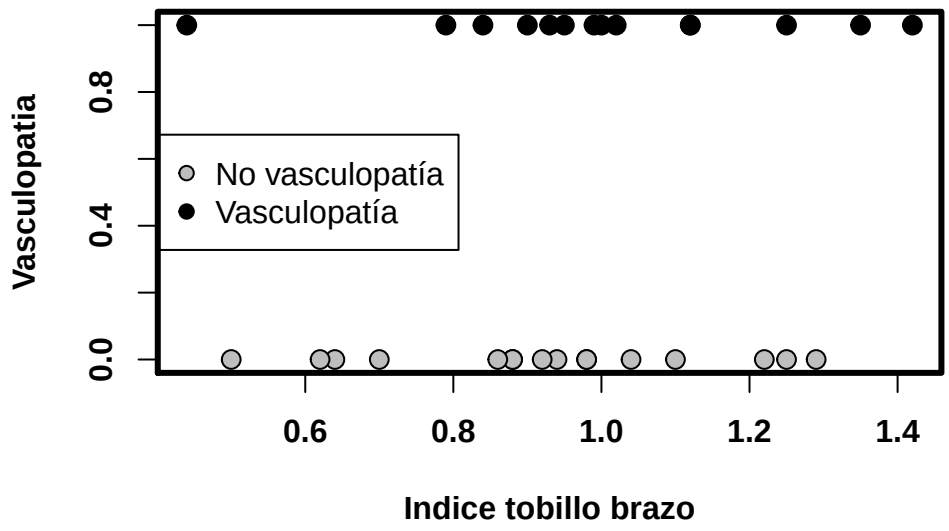
```
colores = c()
colores[datosSimul01$Y == 0] = "grey"
colores[datosSimul01$Y == 1] = "black"
plot(datosSimul01$X, datosSimul01$Y, pch = 21, bg = colores, cex=1.3,font=2,
     xlab = "Indice tobillo brazo", ylab = "Vasculopatia", font.lab=2, )
legend("left", c("No vasculopatía", "Vasculopatía"), pch = 21, pt.bg =  c("grey", "black"))
box(lwd=3)
```



La figura simétrica, cuando $Y = 1$ corresponde a valores altos de X se obtiene de forma similar.

Por último, la figura correspondiente al caso "todo ruido, nada de modelo" se obtiene simplemente eligiendo al azar los valores de Y:

```
set.seed(2015)
Ysimul03 = sample(0:1, size = length(X), replace = TRUE)
datosSimul03 = data.frame(X, Y = Ysimul03)
colores = c()
colores[datosSimul03$Y == 0] = "grey"
colores[datosSimul03$Y == 1] = "black"
plot(datosSimul03$X, datosSimul03$Y, pch = 21, bg = colores, cex=1.3,font=2,
     xlab = "Indice tobillo brazo", ylab = "Vasculopatia", font.lab=2, )
legend("left", c("No vasculopatía", "Vasculopatía"), pch = 21, pt.bg =  c("grey", "black"))
box(lwd=3)
```



1.1. Agrupando valores para estimar las probabilidades.

Vamos ahora a ver ahora detalladamente como usar R para dar los pasos 1 a 4 del Ejemplo 13.1.5 del libro (pág. 508). El primer paso es la lectura de los datos desde el fichero, que incluimos aquí por comodidad:

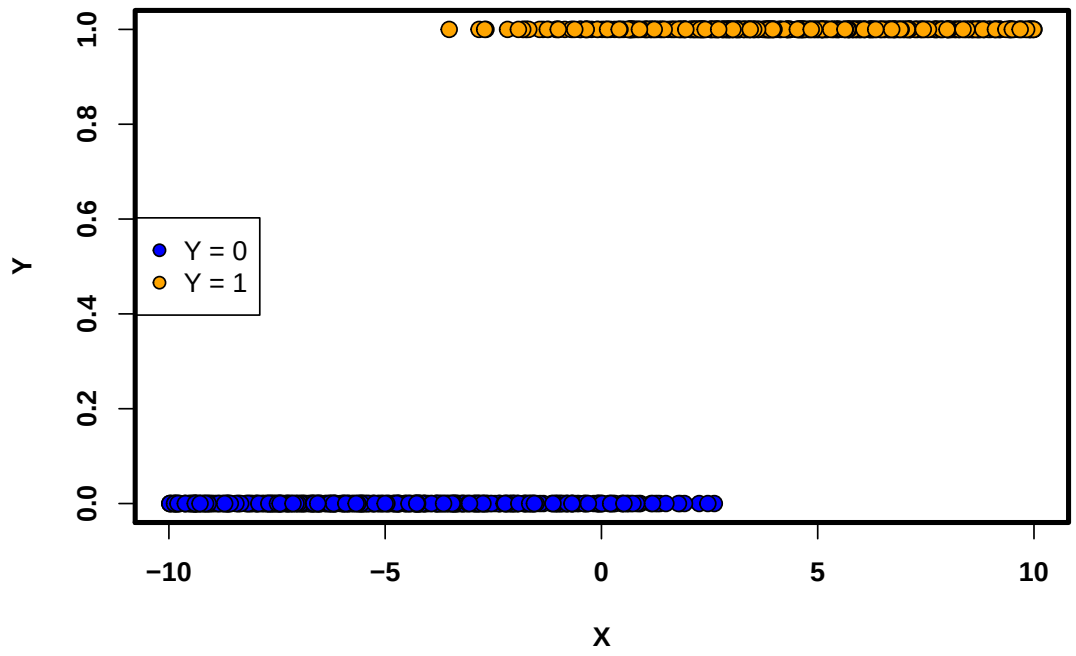
Cap13-ConstruccionModeloLogistico.csv

Una vez descargado este fichero en nuestra carpeta `datos`, usamos read.table para leerlo y luego procedemos como con el anterior ejemplo (ten en cuenta que los datos de este fichero están separados con tabuladores, por eso usamos `sep = \t`)

```
datosFichero = read.table("../datos/Cap13-ConstruccionModeloLogistico.csv",
                          header = TRUE, sep="\t")
head(datosFichero)
```

```
##        X Y
## 1 -3.83 0
## 2 -2.85 0
## 3 -6.23 0
## 4  8.57 1
## 5 -8.14 0
## 6 -4.93 0
```

Como antes, fijamos los nombres de las variables y pintamos el diagrama de dispersión:



Los valores de la variable X están comprendidos completamente en el intervalo $[-10, 10]$:

```
range(X)

## [1] -9.98 10.00
```

Vamos con los pasos del 1 al 4 del ejemplo:

1. Empezamos dividiendo el intervalo $[-10, 10]$ en 40 clases de anchura $\frac{1}{2}$ y localizando los puntos de corte que marcan las fronteras entre clases, que son los valores $u_0, \ldots, u_{40}$ del ejemplo. Hay que prestar atención al número de valores que generamos: 41 puntos definen 40 clases.

   ```
   a = -10
   b =  10
   numClases = 40
   (u = seq(from=a, to=b, length.out=(numClases + 1))) # Fronteras entre clases

   ##  [1] -10.0  -9.5  -9.0  -8.5  -8.0  -7.5  -7.0  -6.5  -6.0  -5.5  -5.0
   ## [12]  -4.5  -4.0  -3.5  -3.0  -2.5  -2.0  -1.5  -1.0  -0.5   0.0   0.5
   ## [23]   1.0   1.5   2.0   2.5   3.0   3.5   4.0   4.5   5.0   5.5   6.0
   ## [34]   6.5   7.0   7.5   8.0   8.5   9.0   9.5  10.0
   ```

   Las marcas de clase, que en el libro hemos llamado w_i, son los puntos medios de cada intervalo. Se obtienen fácilmente sumando a cada valor `u` la semianchura de la clase. Hay que tener cuidado de nuevo con el número de marcas de clase que generamos:

   ```
   (marcasClase =  (u + ((b - a) / (2 * numClases)))[1:numClases])

   ##  [1] -9.75 -9.25 -8.75 -8.25 -7.75 -7.25 -6.75 -6.25 -5.75 -5.25 -4.75
   ## [12] -4.25 -3.75 -3.25 -2.75 -2.25 -1.75 -1.25 -0.75 -0.25  0.25  0.75
   ## [23]  1.25  1.75  2.25  2.75  3.25  3.75  4.25  4.75  5.25  5.75  6.25
   ## [34]  6.75  7.25  7.75  8.25  8.75  9.25  9.75
   ```

2. Ahora vamos a hacer un recuento de cuántos de los valores X caen en cada una de esas clases. Afortunadamente, ya sabemos que en R esto es muy fácil de hacer, usando la función `cut` para definir un factor que clasifique los valores de X:

```
clases = cut(X, breaks=u, include.lowest=TRUE)
```

La opción `include.lowest=TRUE` sirve para que el primer intervalo sea cerrado a la izquierda, cubriendo así los posibles valores de $X = -10$ (aunque nuestra muestra, en este ejemplo, no incluye ninguno de esos valores). Veamos los primeros valores del factor; puedes compararlos con los primeros valores de X para ver qué la clasificación es correcta:

```
head(clases, 10)

##  [1] (-4,-3.5] (-3,-2.5] (-6.5,-6] (8.5,9]   (-8.5,-8] (-5,-4.5] (5,5.5]
##  [8] (9.5,10]  (9.5,10]  (5,5.5]
## 40 Levels: [-10,-9.5] (-9.5,-9] (-9,-8.5] (-8.5,-8] ... (9.5,10]

head(X, 10)

##  [1] -3.83 -2.85 -6.23  8.57 -8.14 -4.93  5.03  9.92  9.78  5.48
```

El recuento que nos habíamos propuesto consiste simplemente en hacer una tabla de frecuencias del factor `clases`

```
table(clases)

## clases
## [-10,-9.5]  (-9.5,-9]  (-9,-8.5]  (-8.5,-8]  (-8,-7.5]  (-7.5,-7]
##         25         33         21         13         27         29
##  (-7,-6.5]  (-6.5,-6]  (-6,-5.5]  (-5.5,-5]  (-5,-4.5]  (-4.5,-4]
##         29         20         30         19         23         31
##  (-4,-3.5]  (-3.5,-3]  (-3,-2.5]  (-2.5,-2]  (-2,-1.5]  (-1.5,-1]
##         22         26         26         21         27         17
##  (-1,-0.5]   (-0.5,0]    (0,0.5]    (0.5,1]    (1,1.5]    (1.5,2]
##         25         26         21         31         25         21
##    (2,2.5]    (2.5,3]    (3,3.5]    (3.5,4]    (4,4.5]    (4.5,5]
##         28         34         28         31         16         31
##    (5,5.5]    (5.5,6]    (6,6.5]    (6.5,7]    (7,7.5]    (7.5,8]
##         28         27         26         27         23         22
##    (8,8.5]    (8.5,9]    (9,9.5]   (9.5,10]
##         25         23         22         21
```

Como se dice en el libro, puedes constatar en esta tabla que hay 27 puntos de la muestra con

$$-8 < X \leq -7.5.$$

La Tabla 13.2 del libro (pág. 510) contiene, a modo de ejemplo, los valores de X que caen en la clase número 18 y los correspondientes valores de Y. Para obtener esos valores en R podemos hacer:

```
(claseElegida = levels(clases)[18])

## [1] "(-1.5,-1]"

X[clases == claseElegida]

##  [1] -1.02 -1.13 -1.35 -1.15 -1.23 -1.43 -1.00 -1.34 -1.01 -1.04 -1.06
## [12] -1.42 -1.06 -1.07 -1.05 -1.25 -1.12

(cualesSon = which(clases == claseElegida))
```

```
##  [1]  29  53  75  97 122 139 303 465 573 607 655 659 737 743 833 880 968

X[cualesSon]

##  [1] -1.02 -1.13 -1.35 -1.15 -1.23 -1.43 -1.00 -1.34 -1.01 -1.04 -1.06
## [12] -1.42 -1.06 -1.07 -1.05 -1.25 -1.12

Y[cualesSon]

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

Ejercicio 1. *Comprueba que, como se dice en el libro, la clase número* 5 *es el intervalo* $(-8, -7.5]$. *Comprueba también que esa clase contiene* 27 *puntos y que los valores* Y *de esos* 27 *puntos muestrales son* todos *iguales a* 0. *Haz un estudio similar con la clase número* 35. □

3. A continuación, para estimar las probabilidades correspondientes a cada una de esas clases vamos a utilizar la función `tapply`. Esta función nos permite aplicar una función (en este caso la suma) a todos los elementos de un vector (o matriz, o array) agrupados mediante un cierto factor. En nuestro ejemplo eso se traduce en una frase como:

   "Calcular la suma (aplicar la función `sum`*) de todos los valores del vector* `Y`*, agrupándolos primero por clases (con la opción* `INDEX=clases`*)".*

   Guardaremos los resultados en un vector de R llamado `sumaYporClases`:

```
(sumaYporClases = tapply(Y, INDEX=clases, FUN=sum))

## [-10,-9.5]  (-9.5,-9]  (-9,-8.5]  (-8.5,-8]  (-8,-7.5]  (-7.5,-7]
##          0          0          0          0          0          0
##  (-7,-6.5]  (-6.5,-6]  (-6,-5.5]  (-5.5,-5]  (-5,-4.5]  (-4.5,-4]
##          0          0          0          0          0          0
##  (-4,-3.5]  (-3.5,-3]  (-3,-2.5]  (-2.5,-2]  (-2,-1.5]  (-1.5,-1]
##          1          0          3          1          6          6
##  (-1,-0.5]   (-0.5,0]    (0,0.5]    (0.5,1]    (1,1.5]    (1.5,2]
##          8         11         10         21         19         18
##    (2,2.5]    (2.5,3]    (3,3.5]    (3.5,4]    (4,4.5]    (4.5,5]
##         26         33         28         31         16         31
##    (5,5.5]    (5.5,6]    (6,6.5]    (6.5,7]    (7,7.5]    (7.5,8]
##         28         27         26         27         23         22
##    (8,8.5]    (8.5,9]    (9,9.5]   (9.5,10]
##         25         23         22         21
```

   Y ahora nuestras estimaciones de las probabilidades se obtienen dividiendo estas sumas entre el número de elementos que contiene cada clase:

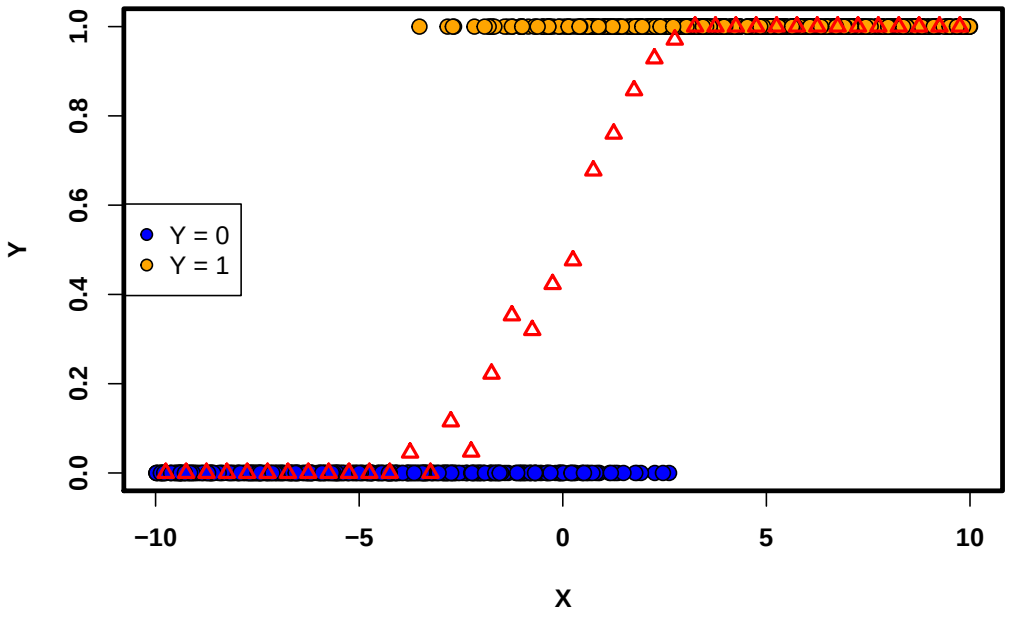
```
(probabilidades = sumaYporClases / table(clases))

## [-10,-9.5]  (-9.5,-9]  (-9,-8.5]  (-8.5,-8]  (-8,-7.5]  (-7.5,-7]
##   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
##  (-7,-6.5]  (-6.5,-6]  (-6,-5.5]  (-5.5,-5]  (-5,-4.5]  (-4.5,-4]
##   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
##  (-4,-3.5]  (-3.5,-3]  (-3,-2.5]  (-2.5,-2]  (-2,-1.5]  (-1.5,-1]
##   0.045455   0.000000   0.115385   0.047619   0.222222   0.352941
##  (-1,-0.5]   (-0.5,0]    (0,0.5]    (0.5,1]    (1,1.5]    (1.5,2]
##   0.320000   0.423077   0.476190   0.677419   0.760000   0.857143
##    (2,2.5]    (2.5,3]    (3,3.5]    (3.5,4]    (4,4.5]    (4.5,5]
##   0.928571   0.970588   1.000000   1.000000   1.000000   1.000000
##    (5,5.5]    (5.5,6]    (6,6.5]    (6.5,7]    (7,7.5]    (7.5,8]
##   1.000000   1.000000   1.000000   1.000000   1.000000   1.000000
##    (8,8.5]    (8.5,9]    (9,9.5]   (9.5,10]
##   1.000000   1.000000   1.000000   1.000000
```

El vector probabilidades contiene nuestras estimaciones $\hat{p}_i$ de las probabilidades correspondientes a cada una de las 40 clases en las que hemos dividido el recorrido de X. Son probabilidades condicionadas, porque hemos agrupado por clases. Puedes comprobar que estas estimaciones coinciden con las que se mencionan en el libro. Además, estos números tienen el aspecto esperado: empiezan valiendo 0 para las clases situadas muy a la izquierda, luego ascienden progresivamente en la zona de transición y, finalmente, se hacen iguales a 1 para las clases a la derecha del recorrido de X.

Ejercicio 2. *Aunque en nuestra muestra no ha sucedido, hay un posible escenario sobre el que conviene que te preguntes: ¿qué haríamos si alguna de nuestras clases no contiene ninguna observación? Ten en cuenta que en ese caso el denominador que hemos usado para estimar la probabilidad sería igual a* 0*. Y eso no puede ser bueno.... ¿qué crees que deberíamos hacer para afrontar ese problema?* □

4. Finalmente, vamos a incluir los puntos (w_i, p_i) en el diagrama de dispersión que obtuvimos antes (recuerda que w_i es las marcas de la clase número i). La figura resultante confirma lo que hemos apreciado en la tabla. Esos puntos se disponen aproximadamente a lo largo de una curva sigmoidea, que se intuye claramente. Esa intuición de una curva es la intuición del modelo logístico para estos datos, que ahora tendremos que construir de forma precisa.

```
points(marcasClase, probabilidades, col="red", pch=2, lwd=2)
```



Ejercicio 3. *Repite los pasos 1 a 4 del análisis anterior con los datos del fichero*

Cap13-ConstruccionModeloLogistico-Inflexiones.csv

que aparece en el Ejemplo 13.1.6 del libro (pág. 512) y comprueba todos los resultados de ese ejemplo. □

2. Las curvas logísticas.

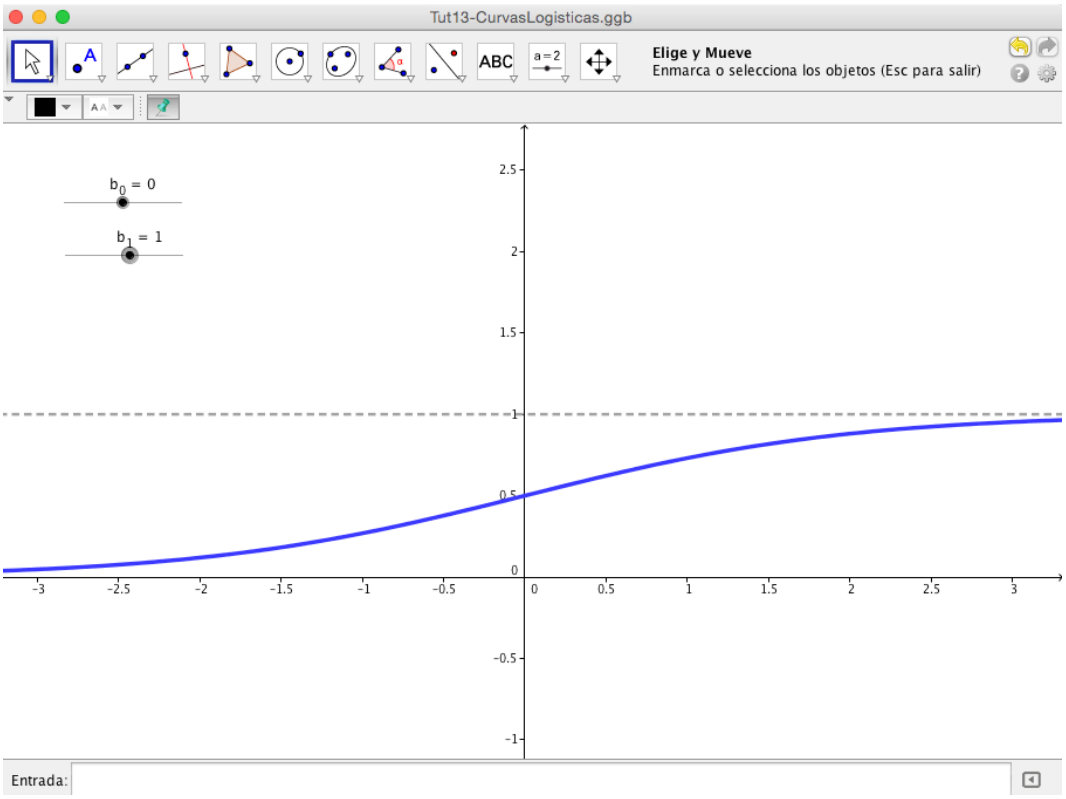
En la Sección 13.2 del libro (pág. 513) hemos introducido la familia de curvas logísticas, definidas mediante:

$$w = \frac{e^{b_0+b_1 v}}{1+e^{b_0+b_1 v}}.$$

Para empezar a familiarizarnos con las propiedades de esta familia de curvas hemos preparado un fichero con una construcción GeoGebra en el que puedes manipular los parámetros b_0 y b_1 y así desarrollar mejor tu intuición sobre el comportamiento de esas curvas. El fichero es este

Tut13-CurvasLogisticas.ggb

Y cuando lo abras te encontrarás con una gráfica similar a esta:



Usa un rato los deslizadores hasta convencerte de que has comprendido el significado de ambos coeficientes.

3. Estimando los parámetros.

En esta sección vamos a aprender a usar el ordenador para obtener estimaciones de los parámetros b_0 y b_1, como hemos visto en la Sección 13.3 (pág. 517) del libro.

3.1. Valores de b_0 y b_1 mediante mínimos cuadrados.

ADVERTENCIA: Aunque lo hemos avisado en el libro, insistimos. El método que vamos a mostrar aquí **NO** es el que se usa en regresión logística y sólo lo incluimos para que el lector tenga ocasión de darse cuenta de que la regresión logística usa un método distinto del que hemos visto en la regresión lineal simple.

Vamos a ver paso a paso como usar el método de mínimos cuadrados para obtener una curva de la familia logística que aproxime a los datos. Los pasos son los mismos que hemos descrito en la Sección 13.3.1 (pág. 518) del libro.

1. Aplicar la transformación logit a los valores $\hat{p}_1, \ldots, \hat{p}_k$. Llamemos $l_1, \ldots, l_k$ a los valores resultantes. Para hacer esto tenemos qu empezar por asegurarnos de que los valores de las probabilidades están comprendidos entre 0 y 1, estrictamente. La razón para hacer esto es que la transformación logit causaría problemas con cualquier valor igual a 0 o igual a 1 (ver Ejercicio 2). Lo conseguimos mediante una selección adecuada de los elementos del vector `probabilidades`:

```
(probsAcotadas = (probabilidades < 1) & (probabilidades > 0))

## [-10,-9.5]  (-9.5,-9]  (-9,-8.5]  (-8.5,-8]  (-8,-7.5]  (-7.5,-7]
##       FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
##  (-7,-6.5]  (-6.5,-6]  (-6,-5.5]  (-5.5,-5]  (-5,-4.5]  (-4.5,-4]
##       FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
##  (-4,-3.5]  (-3.5,-3]  (-3,-2.5]  (-2.5,-2]  (-2,-1.5]  (-1.5,-1]
##       TRUE      FALSE       TRUE       TRUE       TRUE       TRUE
##  (-1,-0.5]   (-0.5,0]    (0,0.5]    (0.5,1]    (1,1.5]    (1.5,2]
##       TRUE       TRUE       TRUE       TRUE       TRUE       TRUE
##    (2,2.5]    (2.5,3]    (3,3.5]    (3.5,4]    (4,4.5]    (4.5,5]
##       TRUE       TRUE      FALSE      FALSE      FALSE      FALSE
##    (5,5.5]    (5.5,6]    (6,6.5]    (6.5,7]    (7,7.5]    (7.5,8]
##      FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
##    (8,8.5]    (8.5,9]    (9,9.5]   (9.5,10]
##      FALSE      FALSE      FALSE      FALSE
```

Como ves, los valores de probabilidades entre 0 y 1, que se concentran en lo que hemos llamado la *zona de transición*, son los que reciben el valor lógico `TRUE`. Ahora ya podemos aplicar la transformación `logit`:

```
probab2Odds = log(probabilidades[probsAcotadas]/(1-probabilidades[probsAcotadas]))
```

2. Lo que hacemos es obtener un modelo de regresión lineal simple, ajustado por mínimos cuadrados, del vector `probab2Odds` frente a las marcas de clase correspondientes. En el Tutorial10 aprendimos a usar la función `lm` de R para calcular los coeficientes $\widetilde{b}_0$ y $\widetilde{b}_1$ de la recta de regresión

$$l = \widetilde{b}_0 + \widetilde{b}_1 \cdot w$$

```
(Oddslm = lm(probab2Odds ~ marcasClase[probsAcotadas]))

##
## Call:
## lm(formula = probab2Odds ~ marcasClase[probsAcotadas])
##
## Coefficients:
##                (Intercept)  marcasClase[probsAcotadas]
##                      0.177                       0.974
```

para los puntos

$$(w_1, l_1), \ldots, (w_k, l_k).$$

Hemos llamado $\widetilde{b}_0$ y $\widetilde{b}_1$ a los coeficientes porque, insistimos, el método que estamos exponiendo **no** es el método que aplicaremos en la regresión logística. Y queremos usar símbolos distintos para los valores obtenidos por casa método.

3. Construimos la curva logística correspondiente a esos valores $\widetilde{b}_0$ y $\widetilde{b}_1$. Esa curva es, desde luego,

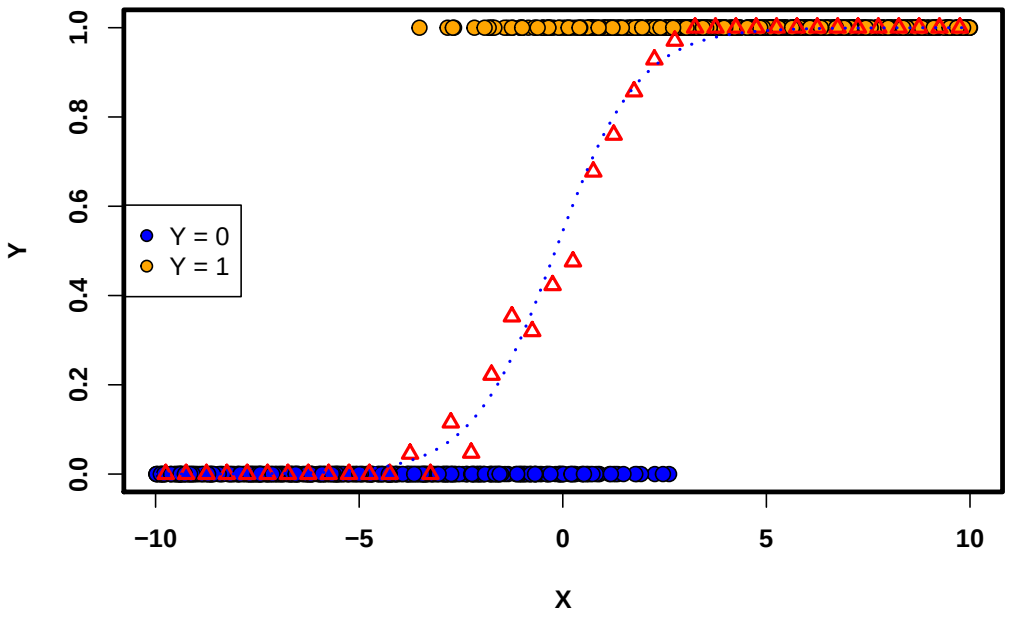
$$w = \frac{e^{\widetilde{b}_0 + \widetilde{b}_1 v}}{1 + e^{\widetilde{b}_0 + \widetilde{b}_1 v}}.$$

Para obtenerla en R podemos definir una función así:

```
curvaLogisticaMinCuad = function(x){
  b0 = Oddslm$coefficients[1]
  b1 = Oddslm$coefficients[2]
  return(exp(b0 + b1 * x)/(1 + exp(b0 + b1 * x)))
}
```

que añadimos al gráfico de dispersión con este comando (fíjate en las opciones `add=TRUE` para añadir la curva al gráfico preexistentey `lty="dotted"` para dibujar una curva discontinua):

```
points(marcasClase, probabilidades, col="red", pch=2, lwd=2)
curve(curvaLogisticaMinCuad, from = -10, to = 10,
      col="blue", lwd="2", add=TRUE, lty="dotted")
```



Con esto hemos concluido la construcción de este modelo para estimar las probabilidades que, insistimos una vez más, **NO** es el que se usa en regresión logística. En la siguiente sección construiremos finalmente el modelo de regresión logística. Cerramos esta sección con un ejercicio para ayudarte a reflexionar sobre las características del método que hemos descrito.

Ejercicio 4. *¿Qué sucede si agrupamos los valores de otra manaera? Cambia el número de clases y comprueba qué sucede con los valores estimados* $\widetilde{b}_0$ *y* $\widetilde{b}_1$*.* □

4. Regresión logística: b_0 y b_1 mediante maxima verosimilitud.

En esta sección vamos a ver cómo usar el ordenador para obtener las estimaciones de los coeficientesde la curva logística usando el método de máxima verosimilitud, tal como se describe en la Sección 13.3.2 del libro (pág. 518). Primero aprenderemos cómo usar la función `glm` de R y luego, en un apartado opcional, veremos un método más detallado, que puede resultar interesante para quienes quieran adentrarse un poco más en las matemáticas subyacentes al método.

4.1. La función `glm` de R.

En el Tutorial10 conocimos la función `lm` de R, que nos permitía obtener rápida y cómodamente la información relativa a un modelo de regresión lineal simple. La función `glm` (del inglés, *generalized linear model*) juega un papel parecido con respecto a los modelos lineales generalizados, de los que la regresión logística es un ejemplo. Por ejemplo, para obtener los coeficientes del modelo logístico para los datos del Ejemplo 13.1.5 del libro (pág. 508), con el que hemos venido trabajando en los apartados anteriores, usaríamos la función `glm` de esta manera:

```
(glmXY = glm(Y ~ X, family = binomial(link = "logit"), data = datos))

##
## Call:  glm(formula = Y ~ X, family = binomial(link = "logit"), data = datos)
```

```
##
## Coefficients:
## (Intercept)            X
##      0.0953       1.0701
##
## Degrees of Freedom: 999 Total (i.e. Null);  998 Residual
## Null Deviance:     1390
## Residual Deviance: 306  AIC: 310
```

Como en los modelos lineales, hemos guardado el resultado de `glm` en una variable para facilitar el acceso posterior al modelo. Antes de seguir adelante aplicamos `summary` a ese modelo:

```
(summGlmXY = summary(glmXY))
```

```
##
## Call:
## glm(formula = Y ~ X, family = binomial(link = "logit"), data = datos)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4259  -0.0891   0.0073   0.1124   2.7190
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.0953     0.1468    0.65     0.52
## X             1.0701     0.0857   12.49   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1385.62  on 999  degrees of freedom
## Residual deviance:  306.23  on 998  degrees of freedom
## AIC: 310.2
##
## Number of Fisher Scoring iterations: 8
```

Vemos que, como también ocurría con `lm`, al aplicar `summary` al modelo se obtiene mucha más información que con la salida original de `glm`. La columna `Estimate` de la tabla `Coefficients` contiene los valores estimados de los coeficientes de la curva logística. Y puesto que, *ahora sí*, se han obtenido por el método de máxima verosimilitud, podemos llamar $\hat{b}_0$ y $\hat{b}_1$ a esas estimaciones. Y los coeficientes estimados son:

```
(b0glm = summGlmXY$coefficients[1])
```

```
## [1] 0.095265
```

```
(b1glm = summGlmXY$coefficients[2])
```

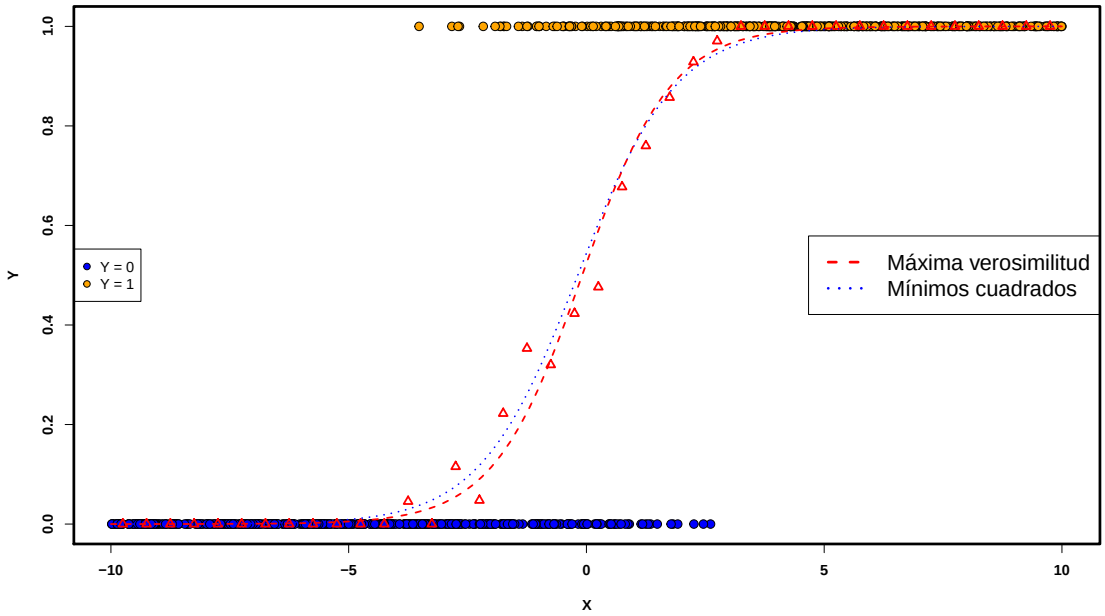
```
## [1] 1.0701
```

como hemos dicho en el Ejemplo 13.3.3 (pág. 521).

Vamos a añadir la curva logística correspondiente a estos coeficientes al diagrama de dispersión. Esa curva aparece en la siguiente figura en trazos (opción `lty="dashed"`) rojos. El código es muy parecido al que hemos usado antes para la curva que obtuvimos usando mínimos cuadrados:

```
curve(curvaLogisticaMinCuad, from = -10, to = 10,
      col="blue", lwd="2", add=TRUE, lty="dotted")
```

```
curvaLogisticaGLM = function(x){
  return(exp(b0glm + b1glm * x)/(1 + exp(b0glm + b1glm * x)))
}
curve(curvaLogisticaGLM, from = -10, to = 10,
      col="red", lwd="2", add=TRUE, lty="dashed")
legend(x="right", legend=c("Máxima verosimilitud", "Mínimos cuadrados"),
       col = c("red", "blue"), bty=1, lty=c("dashed","dotted"),lwd=3,cex=1.5)
```



La figura que hemos obtenido es, esencialmente, la Figura 13.12 del libro (pág. 521).

Volviendo a los resultados de `glm`, el lector habrá observado sin duda la gran cantidad de información que hemos obtenido. Hay una parte de esa información que va más allá de lo que nosotros vamos a aprender en este curso. Pero en el Capítulo 13 y en las siguientes seccione de este tutorial vamos a tratar de llegar tan lejos como nos sea posible en el análisis de esa información.

Para cerrar este apartado, recuerda que los coeficientes $\hat{b}_0$ y $\hat{b}_1$ que hemos obtenido con `glm` son, aproximadamente, los que producen un valor máximo de la función verosimilitud (no corresponden exactamente al máximo porque se han obtenido por un método numérico). ¿Y cuál es ese valor máximo de la verosimilitud? En R es muy fácil obtenerlo a partir del modelo. De hecho, lo que es inmediato es obtener el logaritmo de la verosimilitud:

```
(logVerosimilitud = logLik(glmXY))

## 'log Lik.' -153.11 (df=2)
```

El resultado incluye varias *decoraciones:* un nombre e información sobre los grados de libertad del modelo (hablaremos de esto más adelante). Para eliminar esa información extra y acceder simplemente al valor del logaritmo de la verosimilitud hacemos un pequeño cambio:

```
(logVerosimilitud = logLik(glmXY)[1])

## [1] -153.11
```

Y ahora la versosimilitud es simplemente:

```
(verosimilitud = exp(logVerosimilitud))

## [1] 3.1899e-67
```

Por el momento este resultado no parece de mucha utilidad. Al fin y al cabo, lo que nos trajo aquí era el deseo de estimar los coeficientes de la curva logística, mientras que la verosimilitud parecía sólo una herramienta para hacer esa estimación. Pero cuando lleguemos a los apartados sobre inferencia en la regresión logística veremos que este valor de la verosimilitud juega un papel importante.

Ejercicio 5.

1. *Comprueba los valores de b_0 y b_1 que aparecen en el Ejemplo 13.3.4 del libro (pág. 522) para el modelo que relaciona itb y vasculopatía.*
2. *Añade la curva logística al diagrama de dispersión de esos datos y compara con la Figura 13.13 del libro (pág. 523).*
3. *Calcula la verosimilitud de ese modelo logístico.*

□

4.2. Predicción con el modelo logístico.

Desde que introdujimos la idea de modelo en esta parte del curso hemos insistido en varias ocasiones en que una de las finalidades básicas de los modelos son las predicciones. En el caso del modelo de regresión logística las primeras predicciones que podemos obtener del modelo son los *valores ajustados* correspondientes a los datos de la variable explicativa X que hemos usado para construir el modelo. Es decir, que para cada punto (x_i, y_i) de la muestra obtenemos un valor predicho de la probabilidad:

$$\hat{\pi}(x) = \frac{e^{b_0+b_1\cdot x}}{1+e^{b_0+b_1\cdot x}}.$$

Después de usar `glm`, esos valores ajustados de las probabilidades están almacenados en un vector de R que es la componente `fitted.values` del modelo. En el ejemplo con el que estamos trabajando, podemos explorar esas predicciones así:

```
head(glmXY$fitted.values)
```

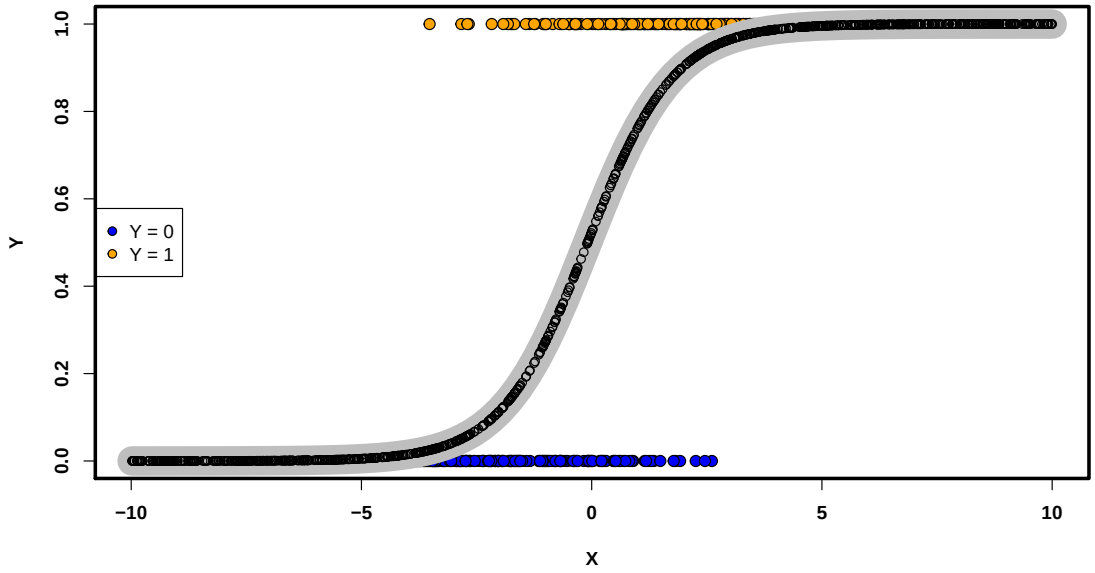
```
##          1          2          3          4          5          6
## 0.01792953 0.04952345 0.00139781 0.99990542 0.00018127 0.00559471
```

```
tail(glmXY$fitted.values)
```

```
##         995         996         997         998         999        1000
## 0.000053528 0.999308273 0.274343833 0.999971163 0.002542159 0.362394478
```

En este ejemplo, con una muestra de $n = 1000$ puntos, al representar gráficamente esas predicciones el resultado cubre casi completamente la curva logística que produce el modelo. Para facilitar la visualización hemos pintado la curva logística con un trazo gris grueso, y en su interior puedes ver, en negro, las probabilidades que predice el modelo para cada valor de X:

```
## [1] 0.095265
## [1] 1.0701
```



En el Tutorial10, al estudiar el modelo de regresión lineal, también vimos que se podía usar la función `predict` para obtener predicciones del modelo para valores de X fuera de la muestra. Por ejemplo, para $x = 3$ haríamos:

```
x0 = 3
predict(glmXY, newdata = data.frame(X=x0))
```

```
##      1
## 3.3056
```

Pero el valor es mayor que 1... ¿no estábamos prediciendo probabilidades? La respuesta es que, por defecto, lo que devuelve `predict` es el valor del término $b_0 + b_1 x$ para el modelo. Vamos a comprobarlo:

```
b0glm + b1glm * x0
```

```
## [1] 3.3056
```

¿Y si lo que queremos es la predicción de la probabilidad? En ese caso usamos el argumento opcional `type = "response"` de `predict`

```
predict(glmXY, newdata = data.frame(X=x0), type = "response")
```

```
##       1
## 0.96462
```

El valor por defecto es `type = "link"` y ya hemos visto que produce como resultado predicciones en la escala del término lineal $b_0 + b_1 x$. El valor de la probabilidad que hemos obtenido es, desde luego, el mismo que obtendríamos sustituyendo el valor de X en la curva logística del modelo:

```
exp(b0glm + b1glm * x0) / (1 + exp(b0glm + b1glm * x0))
```

```
## [1] 0.96462
```

Y, finalmente, también podemos obtener ese valor usando la función `plogis`, de la que aún no hemos hablado:

```r
plogis(b0glm + b1glm * x0)

## [1] 0.96462
```

No vamos a entrar en muchos más detalles sobre `plogis` (el lector interesado puede consultar la ayuda). Nos limitaremos a explicar que el valor de `plogis(u)`, para un número u, es

$$\texttt{plogis(u)} = \frac{e^u}{1+e^u}$$

4.3. Cálculo directo del modelo a partir de la función verosimilitud.

Opcional: esta sección puede omitirse en una primera lectura.

En este apartado, en lugar de recurrir a la función `glm` de R, vamos a construir explícitamente la función verosimilitud y vamos a buscar su valor máximo de otra manera. Recuerda que en la Ecuación 13.8 del libro (pág. 13.8) hemos visto que la función verosimilitud se puede expresar así:

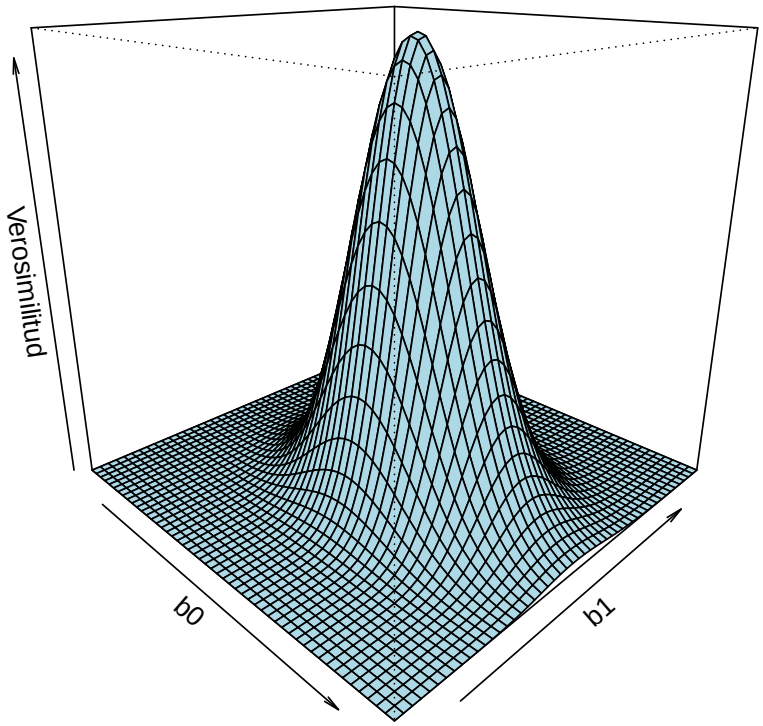
$$\begin{aligned}\mathcal{L}(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n; b_0, b_1) &= \prod_{i:y_i=1} \hat{\pi}(x_i) \cdot \prod_{i:y_i=0} (1-\hat{\pi}(x_i)) \\ &= \prod_{i:y_i=1} \frac{e^{b_0+b_1\cdot x_i}}{1+e^{b_0+b_1\cdot x_i}} \cdot \prod_{i:y_i=0} \frac{1}{1+e^{b_0+b_1\cdot x_i}}\end{aligned}$$

Vamos a empezar por traducir esta expresión a R. Recuerda que aquí pensamos en b_0, b_1 como variables, mientras que X e Y son parámetros.

```r
verosimilitud = function(b){
    prod(exp(b[1] + b[2] * X[Y==1]) /(1 + exp(b[1] + b[2] * X[Y==1]))) *
    prod(1 /(1 + exp(b[1] + b[2] * X[Y==0])))
}
```

A continuación vamos a representar gráficamente esta función verosimilitud. Puesto que es una función de dos variables, su gráfica es una superficie en el espacio tridimensional. Ya hemos encontrado alguna de estas gráficas en otros tutoriales. Y no nos vamos a extender mucho sobre los comandos de R que vamos a utilizar. Los mostraremos simplemente para que el lector interesado sepa por dónde empezar a buscar si en el futuro necesita profundizar en esto:

```r
# Creamos una malla o reticula de valores de b0, b1 en los que
# evaluaremos la funcion verosimilitud mediante la funcion outer
valores_b0 = seq(-0.5, 0.5, length.out = 50)#seq(25, 35, length.out = 60)
valores_b1 = seq(0.7, 1.5, length.out = 50)#seq(-38, -28, length.out = 60)

# Para la representacion que vamos a hacer necesitamos vectorializar
# la funcion verosimilitud.
verosimilitud_vect = Vectorize(function(b0, b1)verosimilitud(c(b0, b1)))

# Usando outer calculamos la verosimilitud para cada
# combinacion de b0 y b1 en la malla que hemos creado
zp = outer(valores_b0, valores_b1, verosimilitud_vect)

# Y finalmente persp se encarga de crear el grafico tridimensional
persp(valores_b0, valores_b1, zp,
      theta=45, phi=20, # angulos de visualizacion
      col="lightblue", xlab="b0", ylab="b1", zlab="Verosimilitud")
```



La figura que hemos obtenido deja claro que la verosimilitud es máxima para los valores de b_0 y b_1 que proporciona el método. En el anterior apartado hemos obtenido esos valores usando `glm`. Pero el problema de encontrar valores mínimos (o máximos) de una función es un problema muy general y que se presenta muy a menudo en una colección muy variada de situaciones. No es de extrañar, por tanto, que exista una parte de las matemáticas dedicada precisamente a eso. La Optimización estudia los métodos para localizar esos valores extremos (máximos o mínimos) de una función bajo una serie de condiciones. En R disponemos de varias herramientas de optimización que en muchas ocasiones son suficientes. Una de esas herramientas es la función `optim`. No podemos extendernos demasiado sobre esta herramienta, porque las posibilidades de configurarla son muy amplias. Nosotros vamos a usar uno de los métodos de optimización disponibles, concretamente el método llamado BFGS por sus autores, (Broyden, Fletcher, Goldfarb, Shanno). El lector interesado puede encontrar más información en la ayuda de la función `optim` o en el libro (más información usando el enlace)

Using R for Numerical Analysis in Science and Engineering de V. Bloomfield

Para usar ese método debemos proporcionarle a la función `optim` un método para calcular los valores del gradiente de la verosimilitud. El gradiente es el vector formado por las dos derivadas parciales:

$$\text{grad}(\mathcal{L}) = \left(\frac{\partial \mathcal{L}}{\partial b_0}, \frac{\partial \mathcal{L}}{\partial b_1}\right)$$

La función verosimilitud es, como hemos visto, muy complicada. Vamos a simplificar las cosas un poco considerando el logaritmo con signo negativo de la verosimilitud El signo se debe a que `optim`, como la mayoría de las funciones de optimización, localiza siempre los valores *mínimos* de la función. Además, en lugar de tratar de encontrar una fórmula para el gradiente, la librería `pracma` de R contiene una función `grad` que permite estimar los valores del gradiente de una función. Así que usaremos esa función para proporcionarle a `optim` los valores del gradiente que requiere el método BFGS. Recuerda instalar la llibrería `pracma` antes de usarla:

```
library(pracma)

# Definimos el logaritmo con signo negativo de la funcion verosimilitud
logVerosmltd = function(b){
  - log(verosimilitud(b))
```

```
}

# Y usamos grad para estimar su gradiente en un punto b = (b0, b1)
grad_logVerosmltd = function(b){grad(logVerosmltd, x0 = b)}
```

Ahora ya estamos listos para usar la función `optim`. Para hacerlo debemos darle un valor inicial aproximado de b_0 y b_1. Como, en principio, no tenemos ninguna estimación, simplemente tomaremos ambos iguales a 1. Al tratarse de un método aproximado, otros valores iniciales producirían resultados ligeramente distintos. Puedes experimentar con otros valores. El resultado de `optim` contiene bastante información sobre el proceso de optimización asociado al método BFGS:

```
(optimizacion =
   optim(par = c(1, 1), fn = logVerosmltd, gr = grad_logVerosmltd, method="BFGS"))

## $par
## [1] 0.095265 1.070099
##
## $value
## [1] 153.11
##
## $counts
## function gradient
##       30        9
##
## $convergence
## [1] 0
##
## $message
## NULL
```

La componente `par` del resultado contiene la estimación de los valores b_0 y b_1 que producen el resultado óptimo:

```
optimizacion$par

## [1] 0.095265 1.070099
```

y si comparas con lo que obtuvimos en la sección previa usando `glm` verás que son los mismos valores:

```
summGlmXY$coefficients

##             Estimate Std. Error  z value   Pr(>|z|)
## (Intercept) 0.095265    0.14684  0.64877 5.1649e-01
## X           1.070099    0.08567 12.49100 8.3591e-36
```

Insistimos, en cualquier caso, en que los resultados dependen de los valores iniciales de b_i que hemos usado en `optim`. Otros valores podrían producir unas estimaciones menos parecidas a las de `glm`.

Ejercicio 6. *La componente* `value` *del resultado de* `optim`*, que es 153.11, también tiene una interpretación relacionada con los resultados que obtuvimos con* `glm`*. ¿Cuál es esa interpretación?* □

5. Inferencia en regresión logística.

5.1. El estadístico de Wald.

En la Ecuación 13.20 del libro (pág. 534) hemos visto el Estadístico de Wald para el coeficiente β_1 del modelo de regresión logística:

$$\Xi = \frac{b_1}{SE(b_1)}$$

Dijimos entonces que no íbamos a dar una expresión explícita del error estándar $SE(b_1)$, porque usaríamos el ordenador para obtener el valor de ese error. Vamos a empezar esta sección señalando que la función `glm` de R ya nos ha proporcionado ese resultado. En efecto, si recordamos la tabla de coeficientes

```
summGlmXY$coefficients
```

```
##             Estimate Std. Error  z value   Pr(>|z|)
## (Intercept) 0.095265    0.14684  0.64877 5.1649e-01
## X           1.070099    0.08567 12.49100 8.3591e-36
```

vemos que el título de la segunda columna es precisamente `Std. Error`, ya que los elementos de esa clumna son los errores estándar de los estadísticos de Wald correspondientes, respectivamente, a β_0 (en la fila `Intercept`) y β_1 (en la fila con el nombre `X` de la variable predictora). En particular el valor de $SE(b_1)$ es, entonces:

```
summGlmXY$coefficients[2, 2]
```

```
## [1] 0.08567
```

Por lo tanto el estadístico de Wald es:

```
(WaldBeta1 = summGlmXY$coefficients[2, 1] / summGlmXY$coefficients[2, 2])
```

```
## [1] 12.491
```

Puedes comprobar que el valor del estadístico es precisamente el que aparece en la tercera columna de la tabla, titulada `z value`. Y la razón de ese título es que, como hemos visto en el libro, Ξ se distribuye como una normal estándar Z. También hemos dicho que actualmente no se considera adecuado usar este estadístico de Wald Ξ para hacer contrastes de hipótesis sobre β_1, por las razones que hemos expuesto en el libro. Pero por si alguna vez necesitras hacerlo, el p-valor se calcularía como en un contraste bilateral con Z (fíjate en que usamos la cola derecha):

```
(pValorWald = 2 * (pnorm(WaldBeta1, lower.tail = FALSE)))
```

```
## [1] 8.3591e-36
```

Y ahora ya está claro que la última columna de la tabla contiene los p-valores de los contrastes para β_0 y β_1, usando los correspondientes estadísticos de Wald.

5.1.1. Selección de modelos y devianza.

En la Ecuación 13.17 del libro (pág. 528) hemos definido la devianza mediante

$$D(modelo) = -2\ln(\mathcal{L}(modelo)).$$

Y sin duda, el lector atento habrá reparado en que uno de los resultados de `glm` es

```
summGlmXY$deviance
```

```
## [1] 306.23
```

Vamos a confirmar que este resultado coincide con la anterior definición. Antes hemos aprendido a calcular el logaritmo de la verosimilitud con `logLik`, así que hacemos:

```
- 2 * logLik(glmXY)
```

```
## 'log Lik.' 306.23 (df=2)
```

e, ignorando las *decoraciones* que R añade, vemos que el valor de la devianza que proporciona `glm` es el esperado.

Pero además, `glm` también nos proporciona la devianza del llamado *modelo nulo*, el que se obtiene suponiendo que la hipótesis nula $H_0 = \{\beta_1 = 0\}$ es cierta:

```
summGlmXY$null.deviance

## [1] 1385.6
```

Y, por lo tanto, el estadístico G de la Ecuación 13.22 del libro (pág. 535) es muy fácil de obtener:

```
(G = summGlmXY$null.deviance - summGlmXY$deviance)

## [1] 1079.4
```

Y, puesto que sabemos que G tiene una distribución muestral χ^2_1, obtenemos el p-valor del contraste de H_0 así:

```
(pValor = pchisq(G, lower.tail = FALSE, df=1))

## [1] 9.9536e-237
```

Con un p-valor tan (ridículamente) pequeño, es evidente que la hipótesis nula se rechaza. Eso se interpreta como una evidencia significativa de que el modelo con $\beta_1 \neq 0$ explica mejor la relación entre las variables X e Y que el modelo nulo con $\beta_1 = 0$.

Ejercicio 7. *Comprueba los resultados del Ejemplo 13.6.1 del libro (pág. 536).* □

5.1.2. Intervalos de confianza para β_0 y β_1

En la Ecuación 13.23 del libro (pág. 537) hemos visto que podemos usar los valores de los errores estándar que intervienen en el Estadístico de Wald para construir intervalos de confianza para β_0 y β_1. Y una vez que sabemos que esos errores estándar aparecen en la tabla que hemos obtenido de `glm`, la construcción de dichos intervalos es muy sencilla:

```
nc = 0.95
alfa = 1 - nc
alfaMedios = alfa/2
zAlfaMedios = qnorm(1 - alfaMedios)

(b0 = summGlmXY$coefficients[1, 1])

## [1] 0.095265

(b1 = summGlmXY$coefficients[2, 1])

## [1] 1.0701

(SE_b0 = summGlmXY$coefficients[1, 2])

## [1] 0.14684

(SE_b1 = summGlmXY$coefficients[2, 2])

## [1] 0.08567

(intervaloBeta0 = b0 + c(-1, 1) * zAlfaMedios * SE_b0)
```

```
## [1] -0.19253  0.38306
```

```
(intervaloBeta1 = b1 + c(-1, 1) * zAlfaMedios * SE_b1)
```

```
## [1] 0.90219 1.23801
```

Aunque siempre es bueno aprender la procedencia de los resultados que obtenemos, si queremos obtener cómodamente estos intervalos lo más fácil es usar la función `confint.default` que nos los proporciona directamente:

```
confint.default(glmXY, level = nc)
```

```
##                2.5 %  97.5 %
## (Intercept) -0.19253 0.38306
## X            0.90219 1.23801
```

Una advertencia: existe otra función, llamada `confint`, que también proporciona intervalos de confianza, pero que en el caso de la regresión logística son de un tipo distinto a los que hemos construido utilizando el estadístico de Wald.

Ejercicio 8. *Comprueba los resultados del Ejemplo 13.6.3 del libro (pág. 537).* □

6. Problemas de clasificación.

En esta sección vamos a usar el modelo de regresión logística que hemos aprendido a construir para implementar un método clasificador, en el caso en el que Y es una variable dicotómica (un factor con dos niveles) y X es una variable continua. Como ejemplo, vamos a usar los datos sobre la relación entre la vasculopatía (variable Y) y el índice itb (variable X). Así que de nuevo empezamos por leer los datos:

```
#datosFichero = read.table("../datos/Cap13-DatosVasculopatia.csv", header = TRUE, sep=",")
head(datosFichero)
```

```
##       X Y
## 1 -3.83 0
## 2 -2.85 0
## 3 -6.23 0
## 4  8.57 1
## 5 -8.14 0
## 6 -4.93 0
```

Fijamos los nombres de las variables y creamos un data.frame.

```
colX = 1
if(colX == 1){
  X = datosFichero[ , 1]
  Y = datosFichero[ , 2]
} else {
  X = datosFichero[ , 2]
  Y = datosFichero[ , 1]
}
datos = data.frame(X, Y)
```

A continuación, construimos el modelo logístico correspondiente usando `glm`.

```
glmXY = glm(Y ~ X, family = binomial(link = "logit"), data = datos)

(summGlmXY = summary(glmXY))
```

```
##
## Call:
## glm(formula = Y ~ X, family = binomial(link = "logit"), data = datos)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4259  -0.0891   0.0073   0.1124   2.7190
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.0953     0.1468    0.65     0.52
## X             1.0701     0.0857   12.49   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1385.62  on 999  degrees of freedom
## Residual deviance:  306.23  on 998  degrees of freedom
## AIC: 310.2
##
## Number of Fisher Scoring iterations: 8

datos$probs = glmXY$fitted.values
```

Como ves, hemos añadido a los datos el vector `fitted.values` con las predicciones de las probabilidades correspondientes a los $n = 1000$ puntos de la muestra. Veámoslas en forma de matriz, junto con el valor de la variable respuesta para cada punto de la muestra. Además, vamos a ordenar las observaciones según el valor de la variable X. Finalmente, hemos guardado en la primera columna el número de cada observación en la muestra original (es una información valiosa, que no conviene perder al crear una tabla como esta). Hemos usado la función `row.names` para gestionar esa información de la tabla de la forma que nos ha parecido más conveniente. Fíjate también en que hemos aprovechado la creación de la tabla para poner nombre a dos de las columnas, `nObs` y `predicProb`.

```
orden = order(X, decreasing = TRUE)
tabla = cbind(nObs = 1:length(X), X, predicProb = glmXY$fitted.values, Y)[orden, ]
row.names(tabla) = 1:length(X)
head(tabla)

##   nObs     X predicProb Y
## 1  175 10.00    0.99998 1
## 2  740  9.99    0.99998 1
## 3   14  9.98    0.99998 1
## 4  496  9.97    0.99998 1
## 5    8  9.92    0.99998 1
## 6  458  9.92    0.99998 1

tail(tabla)

##      nObs     X  predicProb Y
## 995   253 -9.88 0.000028168 0
## 996   304 -9.93 0.000026700 0
## 997   469 -9.95 0.000026135 0
## 998   119 -9.97 0.000025581 0
## 999   135 -9.98 0.000025309 0
## 1000  540 -9.98 0.000025309 0
```

Ahora, para clasificar, debemos elegir un punto o umbral de corte. Inicialmente podemos tomar el valor del que hemos llamado el *clasificador ingenuo*:

```
cp = 0.5
```

Y con ese umbral la clasificación se obtiene así:

```
clasificacion = ifelse(glmXY$fitted.values > cp, 1, 0)
```

Añadimos el resultado de la clasificación a la tabla.

El resultado es la información que aparece en la parte (a) de la Tabla 13.5 del libro (pág. 549):

```
tabla = cbind(tabla, clasificacion = clasificacion[orden])

##   nObs     X predicProb Y clasificacion
## 1  175 10.00    0.99998 1             1
## 2  740  9.99    0.99998 1             1
## 3   14  9.98    0.99998 1             1
## 4  496  9.97    0.99998 1             1
## 5    8  9.92    0.99998 1             1
## 6  458  9.92    0.99998 1             1
##      nObs     X  predicProb Y clasificacion
## 995   253 -9.88 0.000028168 0             0
## 996   304 -9.93 0.000026700 0             0
## 997   469 -9.95 0.000026135 0             0
## 998   119 -9.97 0.000025581 0             0
## 999   135 -9.98 0.000025309 0             0
## 1000  540 -9.98 0.000025309 0             0
```

Para localizar las posiciones de los valores mal clasificados en la tabla usamos la función `which`:

```
errores = which(tabla[ , 4] != tabla[ , 5] )
```

Así que la tasa de aciertos de la clasificación es:

```
(tasaAciertos = 1 - length(errores) / length(X))

## [1] 0.926
```

En el próximo apartado vamos a calcular este y otros valores de otra manera.

6.1. Tabla de contingencia. Sensibilidad y especificidad. Precisión.

¿Cuál es la tabla de contingencia de los valores de Y frente a los resultados de la clasificación? En R la podemos obtener fácilmente usando `table`. Sólo hay un pequeño problema, y es que el orden de los valores no coincide con el que deseamos: `table` coloca en la primera fila (y columna) el valor 0. Así que tenemos que invertir el orden.

```
(tablaContingencia = table(clasif = clasificacion, Y = Y)[2:1, 2:1])

##       Y
## clasif   1   0
##      1 478  39
##      0  35 448
```

A partir de esa tabla es inmediato calcular la sensibilidad y especificidad. Primero usamos `prop.table` como vimos en el Tutorial12 para dividir cada columna por el total de esa columna:

```
(tablaProporciones = prop.table(tablaContingencia, margin = 2))
```

```
##       Y
## clasif        1        0
##      1 0.931774 0.080082
##      0 0.068226 0.919918
```

Y ahora podemos leer directamente los valores que buscábamos:

```
(sensibilidad = tablaProporciones[1, 1])
```

```
## [1] 0.93177
```

```
(especificidad = tablaProporciones[2, 2])
```

```
## [1] 0.91992
```

La tasa de acierto del método clasificador es, como vimos antes:

```
(tasaAcierto = sum(diag(tablaContingencia)) / sum(tablaContingencia))
```

```
## [1] 0.926
```

Punto de indiferencia.

Antes de seguir adelante, el *punto de indiferencia* es:

```
(indiferencia = -glmXY$coefficients[1]/glmXY$coefficients[2])
```

```
## (Intercept)
##   -0.089024
```

Ejercicio 9. *Comprueba los resultados del Ejemplo* **??** *del libro (pág.* **??***).* □

6.2. La librería `caret`.

Como hemos visto muchas veces a lo largo de los tutoriales previos, también hay una forma más sencilla, menos *"manual"*, de obtener esta información. De hecho, ontendremos mucha más información. Para conseguirlo, tenemos que instalar la librería `caret` de R. Esta librería, que aquí vamos a usar de forma muy elemental, pone a nuestra disposición una gran colección de recursos propios del campo del *Aprendizaje Automático* (en inglés, *Machine Learning*). En R existen, literalmente, cientos de librerías para ese tipo de tareas y `caret` tiene, entre otras, la virtud de proporcionar una interfaz de uso común para muchas de esas librerías, ahorrándonos la necesidad de conocer los detalles específicos de cada caso. No te extrañes, por tanto, si la instalación de `caret` es más larga que la de otras librerías más básicas que hemos visto: se trata de una herramienta avanzada.

Pero no temas, para el problema que nos ocupa las cosas son muy fáciles. Después de instalar la librería, la cargamos como de costumbre (puede que veas algunos mensajes de advertencia; mientras no haya errores no te preocupes)

```
library(caret)
```

Ahora vamos a usar una de las funciones que `caret` nos proporciona: la función `confusionMatrix`. Si le proporcionamos a esta función un vector con los resultados de una clasificación dicotómica y un segundo vector con los valores correctos (en nuestro caso, el vector Y) la función nos devuelve tanto la tabla de contingencia que hemos creado antes como los valores de la especificidad y sensibilidad

del clasificador. Pero cuidado: como dice el refrán, el diablo está en los detalles. Al igual que nos ha sucedido antes, tenemos que vigilar el orden que se utiliza al clasificar. Aunque hemos clasificado usando 1 y 0 para representar las clases, la mejor manera de pensar en una clasificación es en términos de un factor con dos niveles. Y es muy importante ordenar esos niveles correctamente, si esperamos obtener valores correctos de la sensibilidad y la especificidad. En una prueba diagnóstica ese trabajo *"nos lo dan hecho"*, porque normalmente asignamos el 1 a enfermo, el 0 a sano y, a su vez, 1 a prueba positiva, 0 a prueba negativa. Pero en otros casos, como una clasificación en *nacional o extranjero*, los niveles no están tan claramente predefinidos. Sirvan estas advertencias para justificar que, en el ejemplo que nos ocupa, vamos a empezar por construir dos factores, en los que además vamos a indicar que queremos que R utilice el valor 1 como primer nivel de los dos factores y que, por tanto, lo coloque en la primera fila y columna de la tabla de contingencia.

```
clasificacion_f = factor(clasificacion, levels = 1:0)
Y_f =  factor(Y, levels = 1:0)
(confMat = confusionMatrix(clasificacion_f, Y_f))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   0
##          1 478  39
##          0  35 448
##
##                Accuracy : 0.926
##                  95% CI : (0.908, 0.941)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.852
##  Mcnemar's Test P-Value : 0.727
##
##             Sensitivity : 0.932
##             Specificity : 0.920
##          Pos Pred Value : 0.925
##          Neg Pred Value : 0.928
##              Prevalence : 0.513
##          Detection Rate : 0.478
##    Detection Prevalence : 0.517
##       Balanced Accuracy : 0.926
##
##        'Positive' Class : 1
##
```

Como ves, la salida de la función incluye mucha información. En primer lugar, la tabla de contingencia, a la que podemos acceder así:

```
confMat$table
```

```
##           Reference
## Prediction   1   0
##          1 478  39
##          0  35 448
```

Pero también nos proporciona otras dos componentes, ambas vectores con nombre:

```
confMat$byClass
```

```
##          Sensitivity          Specificity       Pos Pred Value
##              0.93177              0.91992             0.92456
##       Neg Pred Value            Precision               Recall
```

```
##              0.92754              0.92456              0.93177
##                   F1           Prevalence       Detection Rate
##              0.92816              0.51300              0.47800
## Detection Prevalence    Balanced Accuracy
##              0.51700              0.92585
```

```
confMat$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##     9.2600e-01     8.5187e-01     9.0799e-01     9.4145e-01     5.1300e-01
## AccuracyPValue  McnemarPValue
##    5.7742e-179     7.2728e-01
```

de las que resulta muy fácil extraer la sensibilidad, especificidad y precisión.

```
(sensibilidad = confMat$byClass[1])
```

```
## Sensitivity
##     0.93177
```

```
(especificidad = confMat$byClass[2])
```

```
## Specificity
##     0.91992
```

```
(tasaAcierto = confMat$overall[1])
```

```
## Accuracy
##    0.926
```

Naturalmente, estos valores coinciden con los que hemos obtenido antes.

6.2.1. Clasificación logística con umbral arbitrario.

Para facilitar nuestro trabajo vamos a crear una función en R para medir la calidad de la clasificación dicotómica por umbral basada en la regresión logística. Los argumentos de la función serán el modelo logístico y el umbral o punto de corte. Y la salida de la función será una lista con la sensibilidad, especificidad y precisión:

```
clasifUmbral = function(glmXY, cp){
  Y = glmXY$y
  clasificacion = ifelse(glmXY$fitted.values > cp, 1, 0)
  clasificacion_f = factor(clasificacion, levels = 1:0)
  Y_f =  factor(Y, levels = 1:0)
  confMat = confusionMatrix(clasificacion_f, Y_f)
  return(c(confMat$byClass[1], confMat$byClass[2], confMat$overall[1]))
}
```

Veamos como funciona. Para el clasificador ingenuo hacemos:

```
clasifUmbral(glmXY, cp = 0.5)
```

```
## Sensitivity Specificity    Accuracy
##     0.93177     0.91992     0.92600
```

Pero si se elige otro punto de corte, la sensibilidad, especificidad y tasa de aciertos cambian:

```
clasifUmbral(glmXY, cp = 0.35)

## Sensitivity Specificity    Accuracy
##     0.95712     0.89528     0.92700
```

Pronto analizaremos cómo dependen estos valores de la elección del punto de corte. Pero antes, veamos otros clasificadores.

6.3. Otros clasificadores dicotómicos.

6.3.1. El método knn.

Otra de las ventajas de instalar la librería `caret` es que junto con ella se instalan implementaciones de algunos de los algoritmos básicos del Aprendizaje Automático. Por ejemplo, la función `knn3` nos va a permitir aplicar de forma muy sencilla el algoritmo de clasificación knn que hemos descrito en la Sección 13.7.2 del libro (pág. 543).Usaremos $k = 5$ para empezar e identificaremos con la letra A el modelo que construimos, porque después vamos a usar otro valor de k. Para construir el primer clasificador basta con hacer:

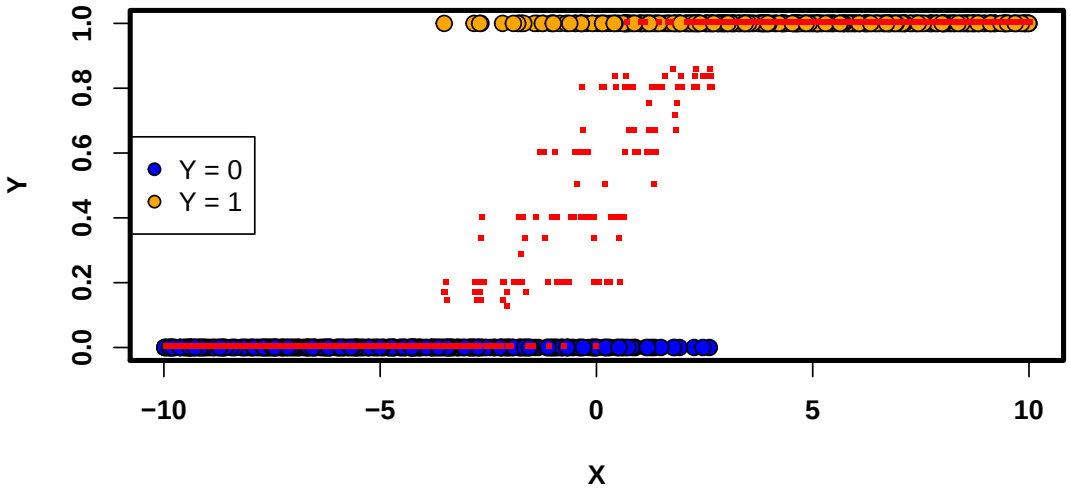
```
knnModeloA = knn3(Y_f ~ X, data = datos, k=5, prob=TRUE)
```

Para hacer predicciones, como en casos anteriores, usamos `predict`. En este caso predecimos sobre los propios valores de X en la muestra, porque aquí no tenemos la opción `fitted` de los modelos de regresión. La respuesta es una matriz de probabilidades de los dos posibles valores de Y, para cada valor de X en la muestra.

```
probsKnnA = predict(knnModeloA, newdata = data.frame(X))
head(probsKnnA)

##             1       0
## [1,] 0.00000 1.00000
## [2,] 0.16667 0.83333
## [3,] 0.00000 1.00000
## [4,] 1.00000 0.00000
## [5,] 0.00000 1.00000
## [6,] 0.00000 1.00000
```

La forma más sencilla de visualizar el resultado es añadiendo esas probabilidades al diagrama de dispersión. El resultado es la parte (a) de la Figura 13.15 del libro (pág. 545). Una vez dibujado el diagrama, podemos usar `points` para añadir las probabilidades.

```
points(X, probsKnnA[ , 1], col="red", pch="·", lwd=1, cex=2)
```



Si lo que queremos es una predicción en valores de Y en lugar de probabilidades, hacemos

```
YknnA = predict(knnModeloA, newdata = data.frame(X), type="class")
head(YknnA)

## [1] 0 0 0 1 0 0
## Levels: 1 0

tail(YknnA)

## [1] 0 1 1 1 0 0
## Levels: 1 0
```

Y ahora podemos analizar estas predicciones mediante `confusionMatrix`

```
confusionMatrix(YknnA, reference = Y_f)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   0
##          1 482  22
##          0  31 465
##
##                Accuracy : 0.947
##                  95% CI : (0.931, 0.96)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.894
##  Mcnemar's Test P-Value : 0.272
##
##             Sensitivity : 0.940
##             Specificity : 0.955
##          Pos Pred Value : 0.956
##          Neg Pred Value : 0.937
##              Prevalence : 0.513
##          Detection Rate : 0.482
##    Detection Prevalence : 0.504
```

```
##       Balanced Accuracy : 0.947
##
##        'Positive' Class : 1
##
```

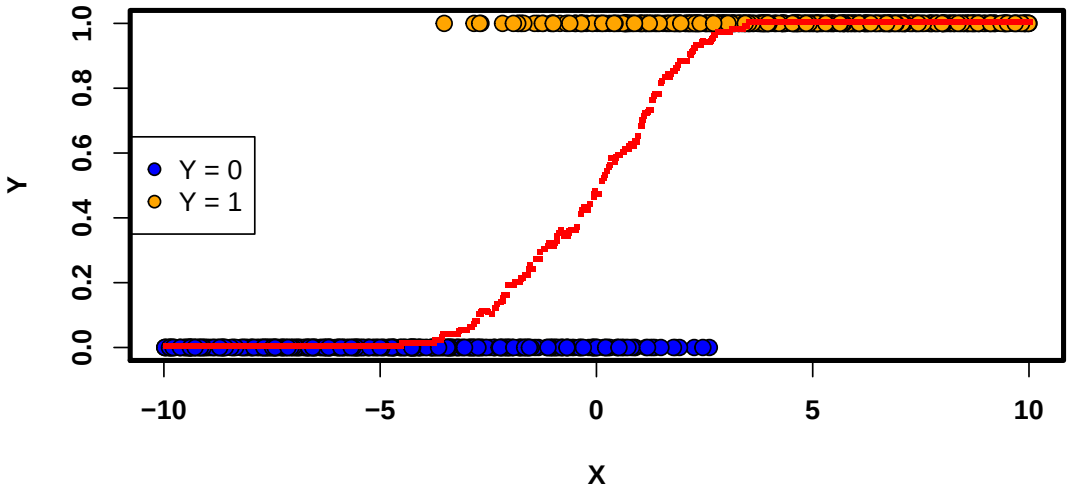
Para obtener la parte (b) de la Figura 13.15 repetimos estos pasos con $k = 100$:

```
knnModeloB = knn3(Y_f ~ X, data = datos, k=100, prob=TRUE)
probsKnnB = predict(knnModeloB, newdata = data.frame(X))
YknnB = predict(knnModeloB, newdata = data.frame(X), type="class")
confusionMatrix(YknnB, reference = Y_f)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   0
##          1 476  31
##          0  37 456
##
##                Accuracy : 0.932
##                  95% CI : (0.915, 0.947)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.864
##  Mcnemar's Test P-Value : 0.544
##
##             Sensitivity : 0.928
##             Specificity : 0.936
##          Pos Pred Value : 0.939
##          Neg Pred Value : 0.925
##              Prevalence : 0.513
##          Detection Rate : 0.476
##    Detection Prevalence : 0.507
##       Balanced Accuracy : 0.932
##
##        'Positive' Class : 1
##
```

y la figura que se obtiene es:

```
points(X, probsKnnB[ , 1], col="red", pch=".", lwd=1, cex=2)
```



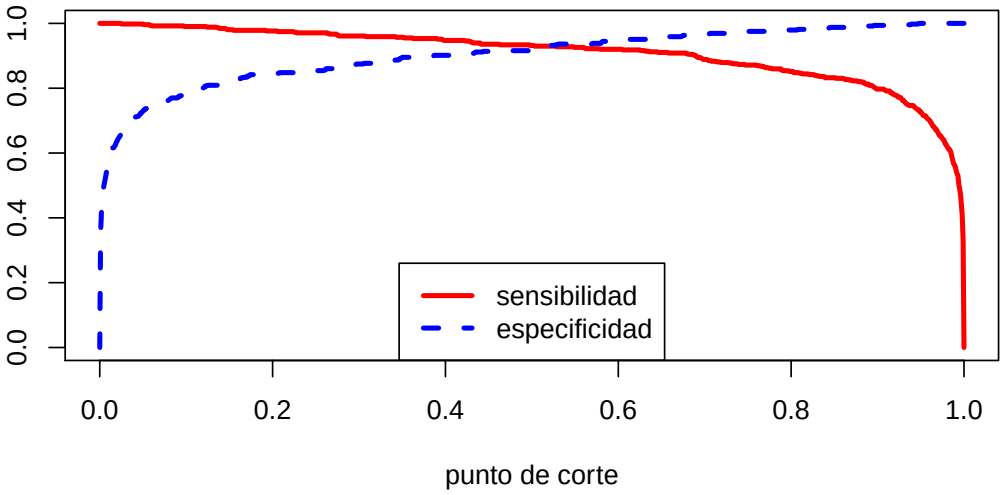
Fíjate en que, a pesar de lo que pudieras haber pensado al ver las figuras, la sensibilidad, especificidas y tasa de errores parecen apuntar a que el clasificador con $k = 5$ hace su trabajo mejor que el que usa $k = 100$.

6.4. Comparación de clasificadores. Curvas ROC.

En este apartado vamos a ver cómo podemos comparar el rendimiento de los algoritmos clasificadores. Como hemos visto en la Sección 13.7.4 del libro (pág. 550), el primer paso en esa dirección puede ser el análisis de la forma en que cambian la sensibilidad y especificidad de un clasificador definido mediante un punto de corte c_p, cuando se consideran distintos valores de c_p. Este análisis, además, es importante por sí mismo cuando de lo que se trata es de seleccionar un valor de c_p con alguna propiedad interesante.

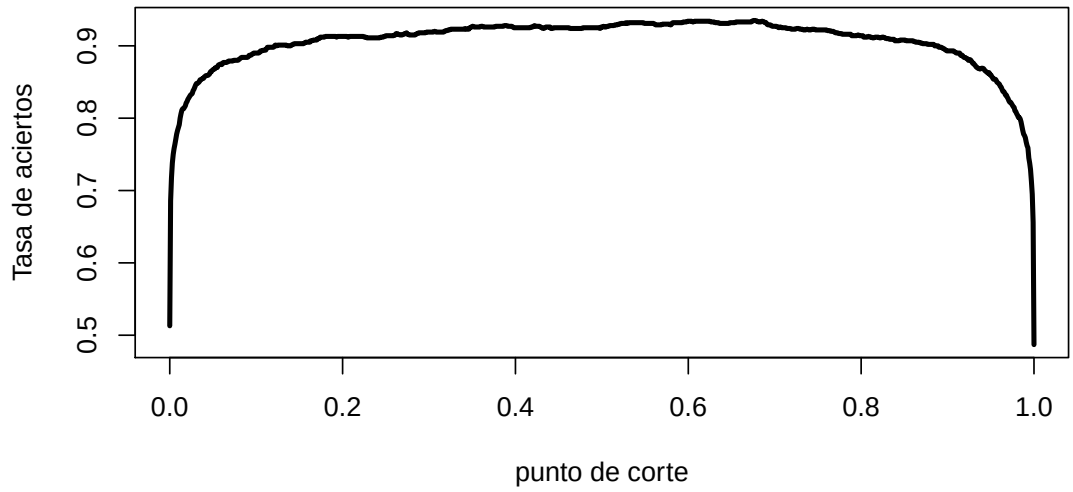
En realidad, las herramientas que necesitamos ya están preparadas. Antes hemos definido una función `clasifUmbral` que nos va a permitir representar gráficamente los valores de la sensibilidad y especificidad en función del punto de corte. Para ello creamos un vector de valores del punto de corte y con `sapply` evaluamos la función `clasifUmbral` en cada elemento del vector. Representamos gráficamente la sensibilidad y especificidad en un mismo gráfico:

```
cp_v = seq(1, 0, by = -0.001)

sens = sapply(cp_v, FUN = function(cp){clasifUmbral(glmXY, cp)[1]})

espec = sapply(cp_v, FUN = function(cp){clasifUmbral(glmXY, cp)[2]})

plot(cp_v, sens, type="l", col="red", lwd=3, lty=1, xlab="punto de corte", ylab="")

points(cp_v, espec, type="l", col="blue", lwd=3, lty=2)

legend(x="bottom", legend=c("sensibilidad", "especificidad"),
       col = c("red", "blue"), bty=1, lty=c(1,2),lwd=3)
```



Para la tasa de errores hacemos algo parecido:

```
accur = sapply(cp_v, FUN = function(cp){clasifUmbral(glmXY, cp)[3]})

plot(cp_v, accur, type="l", col="black", lwd=3, lty=1,
     xlab="punto de corte", ylab="Tasa de aciertos")
```



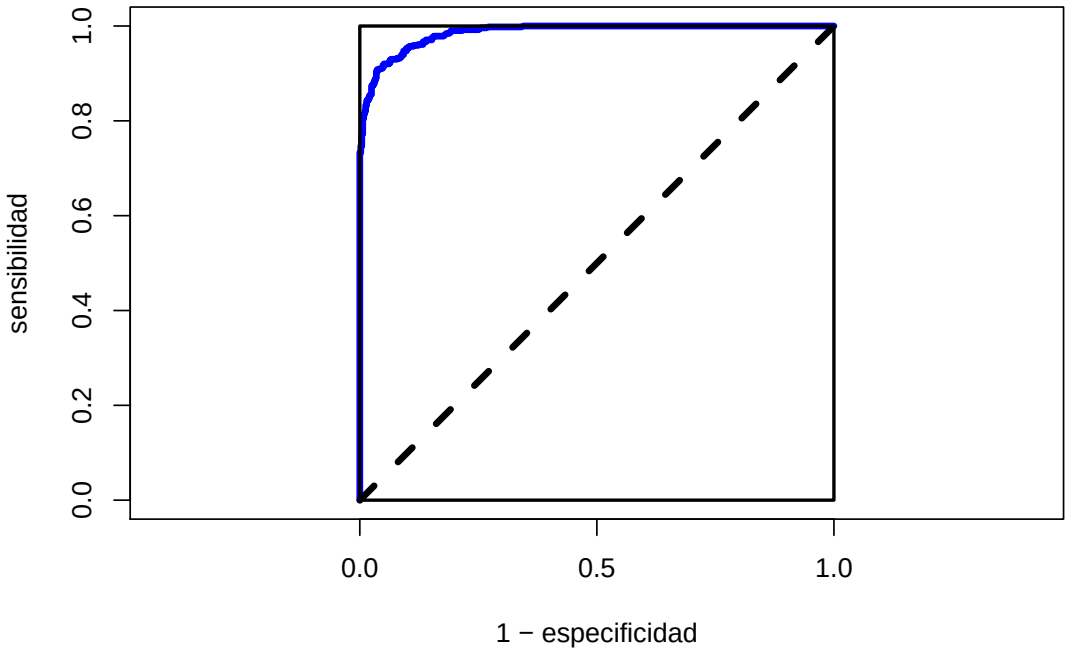
Como estamos trabajando con una muestra grande, las curvas son relativamente suaves, comparadas con las que aparecen en las Figuras 13.16 (pág. 551) y 13.17 (pág. 552) del libro.

Curvas ROC en R. La librería ROCR.

En la Sección 13.7.5 del libro (pág. 554) hemos visto que las curvas ROC son una herramienta que puede resultar muy útil para comparar el rendimiento de dos clasificadores. En esta sección vamos a aprender a dibujarlas con R. Para empezar, vamos a aprovecharnos de que antes hemos generado dos vectores de valores de la sensibilidad y especificidad del clasificador logístico para distintos valores del punto de corte c_p. A partir de esos valores es muy fácil dibujar la curva ROC. Hemos usado la opción gráfica `asp=1` para garantizar que R usa la misma escala en ambos ejes y conseguir así que la región de las curvas ROC sea realmente un cuadrado. Por cierto, `asp` viene del

inglés *aspect*, ya que se suele llamar relación de aspecto al cociente de las escalas que se usan en los ejes de un gráfico. Además, hemos usado `polygon` para dibujar el cuadrado unidad que define la región de las curvas ROC y también hemos usado `segments` para dibujar la diagonal del cuadrado que representa los clasificadores aleatorios.

```
plot(1- espec, sens, type = "l", col="blue", lwd=4,
     xlab="1 - especificidad", ylab="sensibilidad",
     xlim=c(0, 1), ylim=c(0, 1), asp=1)

polygon(c(0, 0, 1, 1), c(0, 1, 1, 0), lwd=2)

segments(x0 = 0, y0 = 0, x1 = 1, y1 = 1, col="black", lty=2, lwd=4)
```



Y ahora vamos a ver una segunda manera, en general más cómoda y potente, de dibujar esas mismas curvas. Para ello tendremos que usar la librería `ROCR`, que habrás de instalar si aún no lo has hecho. Cargamos la librería

```
library(ROCR)
```

y usamos la función `predicition` de la propia librería `ROCR` para construir el tipo de objeto que `ROCR` usa para hacer cálculos. He usado `class` para que veas que `pred` es de tipo `prediction`. Si sigues avanzando en R, o en cualquier otro lenguaje de programamción, tendrás que aprender un poco más sobre los objetos y clases de R.

```
predLogistica = prediction(predictions = glmXY$fitted.values, labels = Y)
class(predLogistica)


## [1] "prediction"
## attr(,"package")
## [1] "ROCR"
```

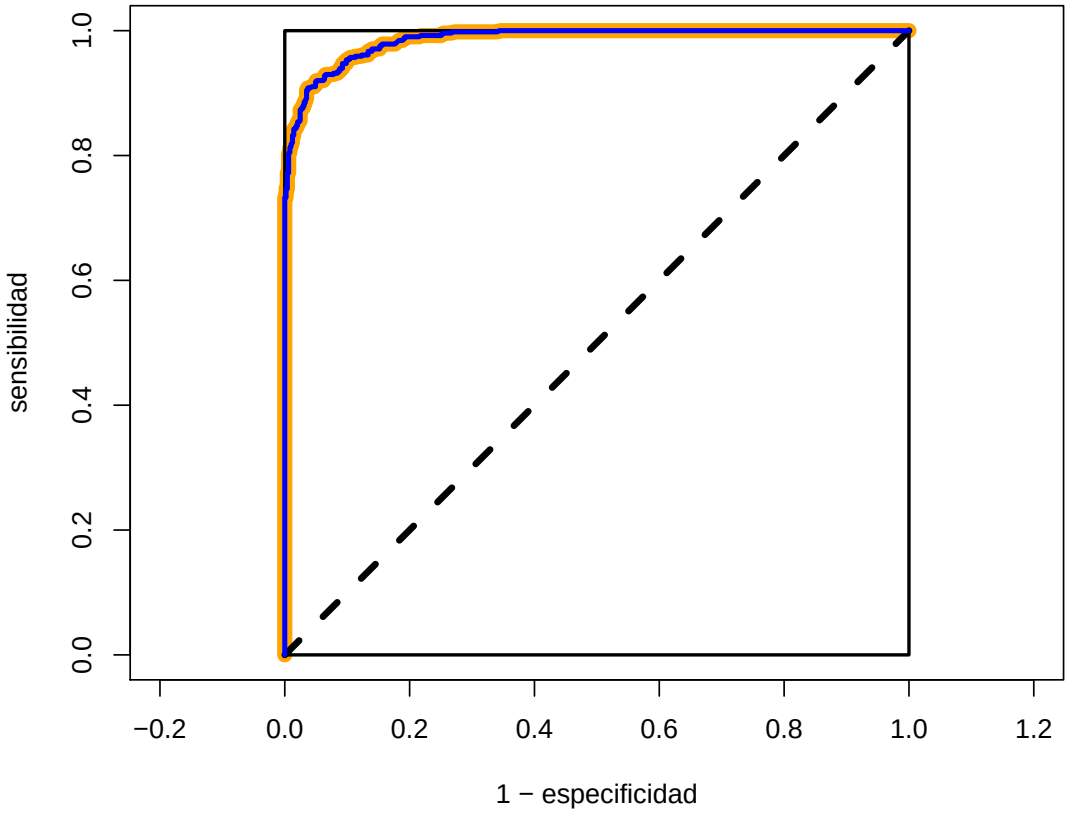
A partir del objeto `predLogistica` podemos pedirle a `ROCR` que calcule toda una serie de medidas de rendimiento (en inglés, *performance*) asociadas con ese clasificador. Para ello usamos la función llamada, precisamente, `performance`. Esa función se puede usar, por ejemplo, para obtener un

vector con valores de la sensibilidad o la especificidad para disintos valores del punto de corte. Pero si le damos a la vez dos de esos indicadores (por ejemplo, sensibilidad y (1 - especificidad)), entonces el resultado de `performance` puede usarse para dibujar el gráfico de como se relacionan entre sí ambos indicadores Por ejemplo, para obtener el dibujo de la curva ROC hacemos:

```
ROClogistica = performance(predLogistica, measure="sens", x.measure="fpr")
```

Aquí `sens` es una abreviatura de sensibilidad (en inglés, *sensitivity*), mientras que `fpr` viene del inglés, *false positive rate*, o tasa de falsos positivos, que es otra manera de referirse a la cantidad que corresponde a (1 - especificidad). En la ayuda de `performance` tienes una (amplia) lista de todas las medidas de rendimiento del clasificador que se pueden utilizar y de las correspondientes abreviaturas.

Ahora ya podemos dibujar esa curva ROC, que vamos a superponer al dibujo que hicimos antes, para que veas que coinciden. Esto se consigue con la opción `add =TRUE` de `plot`. Además hemos pintado la curva ROC que obtuvimos antes *"a mano"* con un trazo grueso de color naranja, para que veas como coincide con la que obtenemos de `ROCR`, que aparece con un trazo azul más fino.



¿Cómo se calcula el área bajo la curva ROC, el valor que hemos llamado AUC? Muy fácilmente, usando de nuevo `performance`:

```
AUClogistica = performance(predLogistica,"auc")
AUClogistica@y.values

## [[1]]
## [1] 0.98591
```

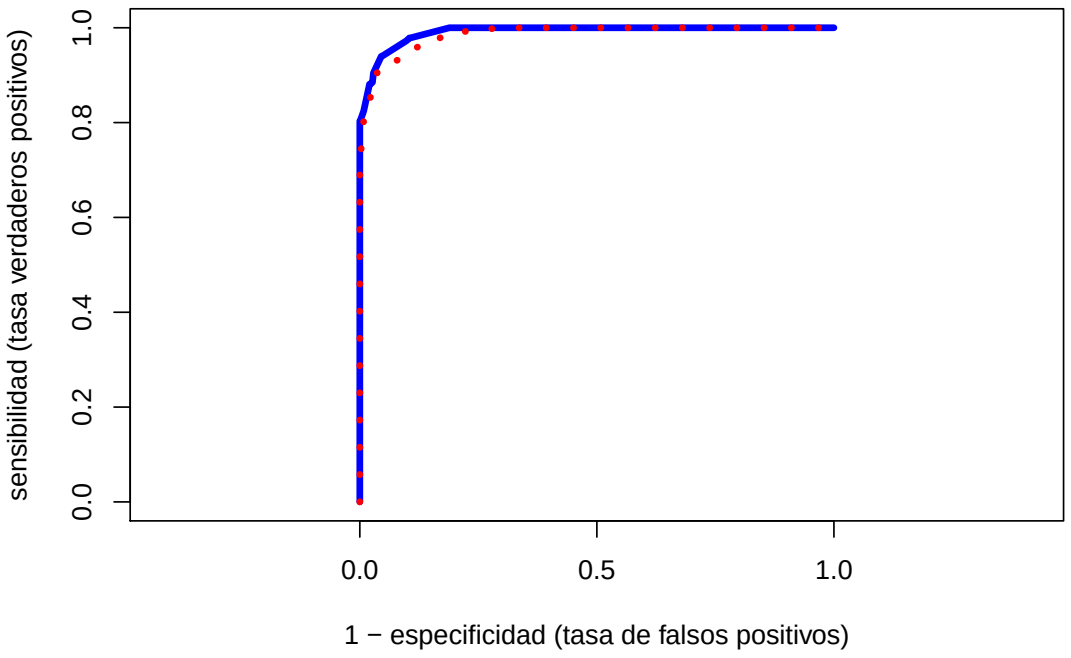
Observación técnica: Si eres un observador muy atento te habrás fijado en que hemos usado el símbolo `@` para acceder al valor del área, en lugar de usar `$`, como hemos hecho en tantas otras ocasiones anteriores. La razón es, de nuevo, una cuestión técnica de R, que puedes ignorar si no

te interesa demasiado: simplificando, R maneja dos grandes sistemas de objetos o clases, llamados S3 y S4. Los objeto de tipo S4 tienen componentes llamados `slots` y para acceder a un `slot` se usa el símbolo `@` que hemos encontrado aquí.

Para practicar un poco más el uso de ROCR vamos a aplicárselo a los clasificadores de tipo knn que hemos usado antes (con $k = 5$ y $k = 100$). Primero hacemos la representación conjunta de sus curvas ROC, de forma similar a lo que hemos hecho en la Figura 13.20 del libro (pág. 558), en aquel caso para la relación entre el itb y las vasculopatía.

Fíjate en que la primera vez usamos `plot` con la opción `add=FALSE`, y la segunda con `add=TRUE` para que las dos curvas aparezcan en la misma figura. Por esa razón sólo hemos necesitado fijar algunos parámetros gráficos en la primera llamada a `plot`.

```
predKnnA = prediction(predictions = probsKnnA[, 1], labels = Y)
predKnnB = prediction(predictions = probsKnnB[, 1], labels = Y)

perfKnnA = performance(predKnnA, measure="tpr", x.measure="fpr")
perfKnnB = performance(predKnnB, measure="tpr", x.measure="fpr")

plot(perfKnnA, add =FALSE, lwd=4, lty=1, col="blue",
      xlab = "1 - especificidad (tasa de falsos positivos)",
      ylab = "sensibilidad (tasa verdaderos positivos)",
      asp=1, axes=FALSE, bty="n")
plot(perfKnnB, add =TRUE, lwd=4, lty=3, col="red")
```



Las dos curvas están muy próximas y son ambas muy buenos clasificadores, pero parece que el clasificador A (con $k = 5$) tiene mejor comportamiento. Una forma simple de comparar ambos clasificadores es calculando sus valores de AUC

```
(AUCKnnA = performance(predKnnA,"auc")@y.values[[1]])

## [1] 0.991

(AUCKnnB = performance(predKnnB,"auc")@y.values[[1]])

## [1] 0.9859
```

Estos valores confirman lo que sospechábamos en la página 30, que el calsificador con $k = 5$ es ligeramente superior al otro (y es computacionalmente menos costoso).

7. Bondad del ajuste en la regresión logística.

En esta sección vamos a ver como usar R para calcular el estadístico de Hosmer-Lemeshow y usarlo para analizar la bondad del ajuste del modelo de regresión logística. Vamos a empezar usando los datos del fichero `Cap13-ConstruccionModeloLogistico.csv` correspondientes al Ejemplo 13.1.5 del libro (pág. 508).

Clases de riesgo.

El primer paso, como hemos visto en la página 561 del libro consiste en definir las clases de riesgo, o deciles de riesgo en el caso común en el que se usan 10 clases. Llamaremos g al número de clases y empezamos construyendo los puntos de corte que marcan la frontera entre clases:

```
g = 10
(puntosCorte = quantile(fitted(glmXY), probs = seq(0, 1, 1/g)))

##           0%          10%          20%          30%          40%          50%
## 0.000025309 0.000254736 0.001877251 0.015795153 0.131242142 0.581014602
##          60%          70%          80%          90%         100%
## 0.923599268 0.986236130 0.998193706 0.999793686 0.999979524
```

A continuación usamos estos puntos de corte y la función `cut` de R que ya conocemos para definir un factor, llamado `claseRiesgo`, que identifica la clase de riesgo a la que pertenece cada valor x_i de la muestra. Hemos elegido las opción `include.lowest = TRUE` de la función `cut` para garantizar que el valor mínimo de la muestra se clasifica en alguna clase de riesgo. Añadimos el resultado al `data.frame datos` porque así es como de hecho sabremos cuál es la clase para cada valor de X.

```
datos$claseRiesgo = cut(fitted(glmXY), breaks = puntosCorte,
                        include.lowest = TRUE)
(n.i = table(datos$claseRiesgo))

##
## [2.531e-05,0.0002547]  (0.0002547,0.001877]     (0.001877,0.0158]
##                   100                   100                   100
##       (0.0158,0.1312]        (0.1312,0.581]        (0.581,0.9236]
##                   100                   100                    99
##       (0.9236,0.9862]       (0.9862,0.9982]       (0.9982,0.9998]
##                   101                   100                   100
##             (0.9998,1]
##                   100

min(n.i)

## [1] 99
```

La tabla de frecuencias del factor nos muestra los valores que en el libro hemos llamado n_i. Y como puede verse, una de las virtudes de esta clasificación es que el número de puntos en cada clase es necesariamente muy parecido (aproximadamente el 10 % en cada clase, si usamos deciles). En este punto es importante comprobar que ninguna de las clases contiene un número demasiado bajo de observaciones. Si alguna de las clases contuviera menos de 10 observaciones, eso debería ponernos en alerta sobre la conveniencia de usar este método para medir la bondad del ajuste. Por eso hemos calculado el mínimo de la tabla, que en este caso muestra que podemos seguir adelante con tranquilidad.

Valores observados y esperados.

Los valores observados $Obs1_i$ (recuerda que $Obs1_i$ es el número de observaciones de la clase C_i para las que $Y = 1$) se calculan así:

```
(Obs1 = with(datos, tapply(Y, INDEX = claseRiesgo, FUN = sum)) )

## [2.531e-05,0.0002547]  (0.0002547,0.001877]     (0.001877,0.0158]
##                     0                     0                     0
##       (0.0158,0.1312]        (0.1312,0.581]        (0.581,0.9236]
##                     6                    35                    74
##       (0.9236,0.9862]       (0.9862,0.9982]       (0.9982,0.9998]
##                    98                   100                   100
##            (0.9998,1]
##                   100
```

La misma operación, pero aplicada a las probabilidades estimadas por el modelo, nos proporciona los valores esperados $Esp1_i$:

```
(Esp1 = with(datos, tapply(probs, INDEX = claseRiesgo, FUN = sum)) )

## [2.531e-05,0.0002547]  (0.0002547,0.001877]     (0.001877,0.0158]
##             0.0085367             0.0772353             0.6655094
##       (0.0158,0.1312]        (0.1312,0.581]        (0.581,0.9236]
##             5.3109641            33.2236573            77.2846428
##       (0.9236,0.9862]       (0.9862,0.9982]       (0.9982,0.9998]
##            97.0953750            99.4203071            99.9221955
##            (0.9998,1]
##            99.9915768
```

Y puesto que sabemos el total de observaciones, los valores $Obs0$ y $Esp0$ se obtienen simplemente restando:

```
(Obs0 = n.i - Obs1)

##
## [2.531e-05,0.0002547]  (0.0002547,0.001877]     (0.001877,0.0158]
##                   100                   100                   100
##       (0.0158,0.1312]        (0.1312,0.581]        (0.581,0.9236]
##                    94                    65                    25
##       (0.9236,0.9862]       (0.9862,0.9982]       (0.9982,0.9998]
##                     3                     0                     0
##            (0.9998,1]
##                     0

(Esp0 = n.i -Esp1)

##
## [2.531e-05,0.0002547]  (0.0002547,0.001877]     (0.001877,0.0158]
##            99.9914633            99.9227647            99.3344906
##       (0.0158,0.1312]        (0.1312,0.581]        (0.581,0.9236]
##            94.6890359            66.7763427            21.7153572
##       (0.9236,0.9862]       (0.9862,0.9982]       (0.9982,0.9998]
##             3.9046250             0.5796929             0.0778045
##            (0.9998,1]
##             0.0084232
```

Naturalmente, si el ajuste es bueno, lo que esperamos es que $Esp1 \sim Obs1$ y que $Esp0 \sim Obs0$. Puedes usar las tablas anteriores para tratar de estimar "a ojo" la calidad del ajuste.

Estadístico de Hosmer-Lemeshow y contraste de la bondad del ajuste.

Con los valores observados y esperados estamos listos para calcular el valor del estadístico de Hosmer-Lemeshow:

```
(HL.estadistico = sum((Obs1 - Esp1)^2/Esp1) + sum((Obs0 - Esp0)^2/Esp0))

## [1] 2.5162
```

Los grados de libertad del contraste son:

```
(HL.df = length(Obs1) - 2)

## [1] 8
```

y el p-valor se calcula con la cola derecha de la correspondiente distribución χ^2_{g-2}:

```
(HL.pValor =  pchisq(HL.estadistico, df = HL.df, lower.tail = FALSE))

## [1] 0.96097
```

Y, como sabemos, un p-valor cercano a uno nos dice que no debemos rechazar la hipótesis nula H_0. Y en el contexto del contraste sobre la bondad del ajuste, significa que las predicciones del modelo se ajustan bien a los datos observado.

Usando la librería `ResourceSelection`.

La librería `ResourceSelection` de R (recuerda instalarla antes de tratar de usarla) nos permite disponer de una función `hoslem.test` que sirve para agilizar todo el proceso de contraste de la bondad del ajuste. En nuestro ejemplo basta con hacer:

```
library(ResourceSelection)

## Warning: package 'ResourceSelection' was built under R version 3.3.2

## ResourceSelection 0.3-2 2017-02-28

(HL.test= hoslem.test(datos$Y, datos$probs, g = 10))

##
##  Hosmer and Lemeshow goodness of fit (GOF) test
##
## data:  datos$Y, datos$probs
## X-squared = 2.52, df = 8, p-value = 0.96
```

Para ver que el valor del estadístico es el mismo que hemos obtenido antes vamos a pedirle a R que nos lo muestre con más cifras:

```
HL.test$statistic

## X-squared
##    2.5162
```

Además, aunque no aparezcan en la salida de la función que R nos muestra, la función `hoslem.test` también calcula tablas de valores observados y esperados, a las que podemos acceder así:

```
HL.test$observed
```

```
##
## cutyhat                    y0  y1
##   [2.531e-05,0.0002547] 100   0
##   (0.0002547,0.001877]  100   0
##   (0.001877,0.0158]     100   0
##   (0.0158,0.1312]        94   6
##   (0.1312,0.581]         65  35
##   (0.581,0.9236]         25  74
##   (0.9236,0.9862]         3  98
##   (0.9862,0.9982]         0 100
##   (0.9982,0.9998]         0 100
##   (0.9998,1]              0 100
```

```
HL.test$expected
```

```
##
## cutyhat                        yhat0      yhat1
##   [2.531e-05,0.0002547] 99.9914633  0.0085367
##   (0.0002547,0.001877]  99.9227647  0.0772353
##   (0.001877,0.0158]     99.3344906  0.6655094
##   (0.0158,0.1312]       94.6890359  5.3109641
##   (0.1312,0.581]        66.7763427 33.2236573
##   (0.581,0.9236]        21.7153572 77.2846428
##   (0.9236,0.9862]        3.9046250 97.0953750
##   (0.9862,0.9982]        0.5796929 99.4203071
##   (0.9982,0.9998]        0.0778045 99.9221955
##   (0.9998,1]             0.0084232 99.9915768
```

que, como se ve, coinciden con los valores que hemos obtenido antes. La función `hoslem.test` nos permite elegir el número de clases con el argumento opcional `g` (por defecto su valor es 10 para usar deciles). Y una posible ventaja de la construcción manual que hemos hecho antes, frente a la comodidad de usar `hoslem.test`, es que podemos seleccionar el método que queramos para definir los cuantiles de la probabilidad. Recuerda que desde el comienzo del libro sabemos que la definición de los percentiles es un tema delicado. El uso de una definición distinta hace que los cálculos arrojen valores ligeramente distintos. Nunca debería ser una diferencia tan grande como para cambiar el sentido de nuestra conclusión sobre la bondad del ajuste. Pero a veces querremos, por ejemplo, comparar los cálculos de R con los de otro programa estadístico que use una definición distina de cuantiles. En casos como esos (y se dan con más frecuencia de la que cabría esperar) se agradece especialmente la naturaleza fácilmente programable de R.

Fin del Tutorial13. ¡Gracias por la atención!