Aprendizaje supervisado en R

Fernando Villalba Bergado

septiembre - 2018

Índice

1. Aprendizaje supervisado en R

El **aprendizaje supervisado** es una técnica usada en minería de datos, en la que se genera una función de pronóstico a partir del entrenamiento previo sobre datos. Es decir, aprendemos a partir de casos reales y extrapolamos el resultado a los datos futuros.

El proceso habitual consiste en dividir la muestra de datos en dos conjuntos, uno de **entrenamiento** y otro de **prueba** o test. Con los datos de entrenamiento ordenados convenientemente obtenemos un conjunto de vectores o pares de entrada-salida. La salida es la variable dependiente, y las entradas sons las variables que usaremos para pronosticar la variable dependiente. es decir, la salida es lo que deseamos pronosticar. Los algoritmos de aprendizaje, aprenden de los datos de entrenamiento y cean un **modelo** o fórmula con la que podremos hacer predicciones con otras entradas diferentes.

Los modelos de aprendizaje supervisado, se denominan tambien habitualmente modelos de clasificación ya que tratan de agrupar los valores en conjuntos con características semejantes, y la salida o respuesta es el grupo al que creen que pertenece el hecho definido en la entrada.

Existen diferntes algoritmos que abordan el problema de aprendizaje supervisado y técnicas de minería de datos, en concreto vamos a explicar los 5 siguientes:

- knn ((k-Nearest Neighbour Classification).
- naive bayes
- regresión logistica binaria
- árboles de decisión
- bosques de clasificación

2. Consideraciones previas

Siempre tenemos que cuidar estas cuestiones antes de realizar el modelo de pronóstico. Simplemente detallaremos dos cosas básicas, por un lado generar dos conjuntos de datos que sirvan para el entrenamiento (*train*) y para la comprobación a posteriori (*test*). Con este sistema de division de la muestra evitamos el *overfitting* tan preocupante en los modelo de predicción.

Por otro lado vamos a describir algunos ejemplo de factorización o agrupamiento de los datos. En casos como naive bayes, el uso de valores numéricos continuos en los datos de origen genera una sobredimension de los casos posibles, un exceso de combinatoria entre variables, que desborda el modelo y puede producir errores en los pronósticos. En estos casos es siempre recomendable factorizar, agrupar y disminuir el número de opciones de cada variable.

En otros casos como el algoritmo knn, la medición de distancia entre variables es el eje del pronostico, por lo que es necesario escalar los datos normalizarlos para equiparar distancias entre las variables, como veremos en los ejemplos.

2.1. Crear particiones de la muestra

El primer paso en todo análisis debe ser dividir la muestra de datos en dos conjuntos de datos uno para entrenamiento y otro para test. Esto se puede hacer a mano, por ejemplo usando la función `sample`, o con la ayuda de algunos paquetes que llevan funciones incorporadas para las particiones de datos.

2.1.1. Ejemplo de partición a mano

Usaremos la base de datos de muestra de supervivientes del Titanic que se da como tabla en `dataset`. Para ver todas las series y bases de datos disponibles en dataset escribiremos `data()`.

Titanic es una tabla que indica casos y frecuencias de cada caso, por lo que para crear la tabla completa hay que expandir la tabla origen, y repetir cada caso el número de veces que indica la columna frecuencia.

```
# cargamos los datos
    data("Titanic")
    str(Titanic)
```

```
##  table [1:4, 1:2, 1:2, 1:2] 0 0 35 0 0 0 17 0 118 154 ...
##  - attr(*, "dimnames")=List of 4
##   ..$ Class   : chr [1:4] "1st" "2nd" "3rd" "Crew"
##   ..$ Sex     : chr [1:2] "Male" "Female"
##   ..$ Age     : chr [1:2] "Child" "Adult"
##   ..$ Survived: chr [1:2] "No" "Yes"
```

```
    class(Titanic)
```

```
## [1] "table"
```

```
# Transformamos los datos wn una dataframe
    Titanic_df=as.data.frame(Titanic)
    str(Titanic_df)
```

```
## 'data.frame':    32 obs. of  5 variables:
##  $ Class   : Factor w/ 4 levels "1st","2nd","3rd",..: 1 2 3 4 1 2 3 4 1 2 ...
##  $ Sex     : Factor w/ 2 levels "Male","Female": 1 1 1 1 2 2 2 2 1 1 ...
##  $ Age     : Factor w/ 2 levels "Child","Adult": 1 1 1 1 1 1 1 1 2 2 ...
##  $ Survived: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Freq    : num  0 0 35 0 0 0 17 0 118 154 ...
```

```
# Creamos una tabla de casos competos a partir de la frecuencia de cada uno
# Esto repite cada caso el num de veces que se ha dado
# según la frecuencia que está en la columna Freq de la tabla.
    repetir_secuencia=rep.int(seq_len(nrow(Titanic_df)), Titanic_df$Freq)
# tenemos una serie con el numero de registro de la tabla original y las veces que se repite

# Creamos una nueva tabla repitiendo los casos según el modelo anterior.
    Titanic_data=Titanic_df[repetir_secuencia,]

# Ya no necesitamos la columna de frecuencias  y la borramos.
    Titanic_data$Freq=NULL
    head(Titanic_data)
```

```
##     Class  Sex   Age Survived
## 3     3rd Male Child       No
## 3.1   3rd Male Child       No
## 3.2   3rd Male Child       No
## 3.3   3rd Male Child       No
## 3.4   3rd Male Child       No
## 3.5   3rd Male Child       No
```

```
# Como vemos todo son factores
    str(Titanic_data)
```

```
## 'data.frame':    2201 obs. of  4 variables:
##  $ Class   : Factor w/ 4 levels "1st","2nd","3rd",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ Sex     : Factor w/ 2 levels "Male","Female": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age     : Factor w/ 2 levels "Child","Adult": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Survived: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
```

En este caso buscamos dividir la tabla en dos conjuntos uno de entrenamiento con el 75 % de los registros y otro de comprobación o de test con el 25 % restante de los registros que nos servirá para validar el modelo.

```
# ----------------------------------
# DIVISION A MANO
# contamos el num de registro de la base de datos del titanic
    nrow(Titanic_data)
```

```
## [1] 2201
```

```
# calculamos el 75%
    num_reg_entrenamiento<-as.integer(0.75*nrow(Titanic_data))
    num_reg_entrenamiento
```

```
## [1] 1650
```

```
# Creamos una vector de 75% de los registros aleatorio
    titanic_train <- sample(nrow(Titanic_data), num_reg_entrenamiento)

# Creamos el conjunto de registros de entrenamiento pasando ese vector a la tabla
    d_titanic_train <- Titanic_data[titanic_train,]
    head(d_titanic_train)
```

```
##        Class  Sex   Age Survived
## 12.631  Crew Male Adult       No
## 12.361  Crew Male Adult       No
## 12.460  Crew Male Adult       No
## 12.509  Crew Male Adult       No
## 12.457  Crew Male Adult       No
## 12.567  Crew Male Adult       No
```

```
# Creamos los datos de comprobación o test (notese el -)
    d_titanic_test <- Titanic_data[-titanic_train,]
    head(d_titanic_test)
```

```
##      Class    Sex   Age Survived
## 3.20   3rd   Male Child       No
## 3.24   3rd   Male Child       No
## 3.30   3rd   Male Child       No
## 3.34   3rd   Male Child       No
## 7.3    3rd Female Child       No
## 9      1st   Male Adult       No
```

Usando una formulación simple hemos creado dos conjuntos de muestra aleatorios excluyentes de entrenamiento y muestra.

2.1.2. Ejemplo de partición con `library(caret)`

Veamos otra forma de hacerlo usando la librería `library(caret)`

```
library(caret)
set.seed(987654321)
# creamos un vector de particion sobre la variable Survived
# el tamaño de muestra será de 75%
trainIndex=createDataPartition(Titanic_data$Survived, p=0.75)$Resample1

d_titanic_train=Titanic_data[trainIndex, ]
d_titanic_test= Titanic_data[-trainIndex, ]
```

2.2. Categorización de los datos origen

Existe un problema en el uso de las funciones de clasificación cuando las combinaciones posibles entre variables tienden a ser infinitas. Esto sucede cuando, por ejemplo, una de las variables es de tipo numérico, y tiene datos continuos.

La mayoría de los modelos de clasificación solo son capaces de trabajar un número limitado de categorías, y por lo tanto, es necesario agrupar los datos originales y reducir las opciones combinatorias, por lo que hay que evitar siempre el uso de varialbles continuas en los datos. Si no realizamos la reducción de categorías nos arriesgamos a obtener errores, incluso evidentes, en los pronósticos.

Un caso evidente es el uso de la función de `naive_bayes` que maneja muy mal los datos continuos, pues está pensado para variables categorizadas.

La solución es realizar una categorización previa de los datos que evite el problema y a la vez simplifique el modelo de pronóstico. Categorizar significa agrupar los datos de las variables continuas en categorías próximas, simplificando las salidas y reduciendo las combinaciones.

Un ejemplo claro es redondear las salidas numéricas a números divisibles por 10 o por 5, o sustituir la variable por los cuantiles más representativos.

Para transformar las variables y categorizarlas podemos usar varias funiciones de R como:

- convertir a factor con la funcion `as.factor()`. * Las categorías de un factor se ven con la función `levels()` * los nombres de esas categorias los damos con la funcion `lables()`
- la funcion `table(tabla_1$hora)` cuenta y resumen los datos
- las funciones `quantile()` o `cut()` ayudan a dividir y categorizar variables continuas
- fundiones de redondeo

Vamos a ver varios ejemplo con la dataset "women" que contiene las alturas y pesos de mujeres americanas de edad entre 30 y 39 años. Ambos datos de altura y peso son continuos y toman 15 posibles valores, por lo que las combinaciones cruzadas dan muchísimos casos posibles. Para simplificar las combinatorias vamos a categorizar los datos de varias maneras, como ejemplos:

```
# cargamos los datos de
    data("women")
    str(women)
```

```
## 'data.frame':    15 obs. of  2 variables:
##  $ height: num  58 59 60 61 62 63 64 65 66 67 ...
##  $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

```
    summary(women)
```

```
##      height         weight
##  Min.   :58.0   Min.   :115.0
##  1st Qu.:61.5   1st Qu.:124.5
##  Median :65.0   Median :135.0
##  Mean   :65.0   Mean   :136.7
##  3rd Qu.:68.5   3rd Qu.:148.0
##  Max.   :72.0   Max.   :164.0
```

```
    length(unique(women$height)) # numero de registros unicos
```

```
## [1] 15
```

```
    length(unique(women$weight))
```

```
## [1] 15
```

```
# opcion 1. creamos una funcion de redondeo
    redondea5<-function(x,base=5){
            as.integer(base*round(x/base))
    }

# Copiamos la tabla con el nombre nuevo
    mujeres_a<-women
# pasamos los datos a sistema internacional
    mujeres_a$peso<-mujeres_a$weight*0.453592 # paso a kg
    mujeres_a$peso<- redondea5(mujeres_a$peso,10)
    length(unique(mujeres_a$peso))
```

```
## [1] 3
```

```
    mujeres_a$altura<-mujeres_a$height*2.54 # paso a cm
    mujeres_a$altura<- redondea5(mujeres_a$altura,10)
    length(unique(mujeres_a$altura))
```

```
## [1] 4
```

```
    head(mujeres_a)
```

```
##   height weight peso altura
## 1     58    115   50    150
## 2     59    117   50    150
## 3     60    120   50    150
## 4     61    123   60    150
## 5     62    126   60    160
## 6     63    129   60    160
```

Con la simplificacion anterior hemos pasado de 15x15 casos combinatorios a solo 3x4.

Podríamos usar también factores para convertir los datos

```r
#Ejemplo de transformacion a factor
    mujeres_a$peso1<- factor(mujeres_a$peso,
                         levels = c(50, 60, 70),
                         labels = c( "flaca","media","gordita"))
# Ejemplo 2 de trans a factor:
    mujeres_a$altura1 <- factor(mujeres_a$altura, levels = c(150,160,170,180), labels = c("Bajo","Medio'
    head(mujeres_a)
```

```
##   height weight peso altura peso1 altura1
## 1     58    115   50    150 flaca    Bajo
## 2     59    117   50    150 flaca    Bajo
## 3     60    120   50    150 flaca    Bajo
## 4     61    123   60    150 media    Bajo
## 5     62    126   60    160 media   Medio
## 6     63    129   60    160 media   Medio
```

2.3. Manejo de NA

En todos los modelos, la existencia de registros con falta de datos o NA, anula el valor de dicha evidencia en el modelo de entrenamiento.

Una solución es completar estos casos con las funciones como `impute()` del paquete `e1071` que sustituye el NA por un valor estimado, que puede ser la media.

En la tabla de ejemplo `donors` hay muchos datos de la edad de los clientes que faltan.

```r
    # Vemos cuantos datos de edad faltan.
        set.seed(333)
        datos<-data.frame(a=sample(1:10, 100,replace = T))
        datos$a[c(1,3)] <- NA
        head(datos)
```

```
##    a
## 1 NA
## 2  1
## 3 NA
## 4  6
## 5  1
## 6  8
```

```r
    library(e1071)
    # Imputamos la nuevos datos estimados de edad asignando usando impute
        datos$imputed <- impute(datos)
        head(datos)
```

```
##    a a
## 1 NA 5
## 2  1 1
## 3 NA 5
## 4  6 6
## 5  1 1
## 6  8 8
```

```
    # Otra forma es hacerlo manualmente
        datos$imputed2<-ifelse(is.na(datos$a),5,datos$a)
        head(datos)
```

```
##    a a imputed2
## 1 NA 5        5
## 2  1 1        1
## 3 NA 5        5
## 4  6 6        6
## 5  1 1        1
## 6  8 8        8
```

3. k-NN (k-Nearest Neighbour Classification)

El algoritmo k-NN reconoce patrones en los datos sin un aprendizaje específico, simplemente midiendo la distancia entre grupos de datos. Se trata de uno de los algoritmos más simples y robustos de aprendizaje automatico.

En realidad el algoritmo puede usarse tanto para clasificar como para pronosticar mediante regresión, pero aquí veremos solo la forma de clasificación.

Para usarlos necesitamos cargar el paquete `class` y usar la función `knn()`que realiza la **clasificación**. La idea subyacente es que a partir de un conjunto de datos de entrenamiento se pueda deducir un criterio de agrupamiento de los datos.

Es un algoritmo muy simple de implementar y de entrenar, pero tienen una carga computacional elevada y no es apropiado cuando se tienen muchos grados de libertad.

3.1. Consideraciones previas

Como se calcula la similitud con respecto a la distancia, debemos tener en mente que las distancias entre variables deben ser comparables. Si usamos un rango de medida en una variable y otro muy distinto en otra, las distancias no están normalizadas y estaremos comparando peras con manzanas.

Para realizar un análisis con **knn** tenemos siempre de normalizar los datos, re-escalar todas las variables para que las distancias sean equiparables. Este proceso se suele llamar: *estandarización de los datos.*

Otro importante asunto es que hay que eliminar los NA de los datos, pues afectan a los calculos de distancia.

Por último, como se indicó antes, este modelo es válido solo para casos con pocas dimensiones en los datos, pocos grados de libertad. Cuando se incrementa la dimension espacial de los datos, la complejidad y el cálculo se hacen inviables.

3.2. Ejemplo

Vamos a hacer un ejemplo sencillo de clasificación con unos datos inventados: Imaginemos que un profesor ha anotado durante el curso los siguientes datos de los alumnos:

- nota del trabajo de clase del primer trimestre.
- nota del examen 1º evaluación.
- interés mostrado en clase por cada alumno al final del curso(1=maximo, 2=medio,3= minimo)

Con estos datos ha confeccionado una tabla.

```r
# vamos a crear el ejemplo de cero:
tabla_alumnos<-data.frame(trabajo=c(10,4,6,7,7,6,8,9,2,5,6,5,3,2,2,1,8,9,2,7))
tabla_alumnos$examen<- c(9,5,6,7,8,7,6,9,1,5,7,6,2,1,5,5,9,10,4,6)
# interes en la clase 1 = max 3 = min interes
tabla_alumnos$interes<- c(1,2,1,1,1,2,2,1,3,3,3,2,3,3,2,2,1,1,3,3)
str(tabla_alumnos)
```

```
## 'data.frame':    20 obs. of  3 variables:
##  $ trabajo: num  10 4 6 7 7 6 8 9 2 5 ...
##  $ examen : num  9 5 6 7 8 7 6 9 1 5 ...
##  $ interes: num  1 2 1 1 1 2 2 1 3 3 ...
```

```r
# A priori parece que los alumnos que tuvieron una nota mayor
# el la prmera evaluación, fueron los que al final tuvieron más interes en clase
aggregate(examen ~ interes, data = tabla_alumnos, mean)
```

```
##   interes   examen
## 1       1 8.285714
## 2       2 5.666667
## 3       3 3.714286
```

```r
# Cargamos el paquete class' que contienen la funcion knn
library(class)

head(tabla_alumnos)
```

```
##   trabajo examen interes
## 1      10      9       1
## 2       4      5       2
## 3       6      6       1
## 4       7      7       1
## 5       7      8       1
## 6       6      7       2
```

```r
# Creamos un vector de eiquetas
# este vector coincidirá con la variable de interes del alumno

# Classificamos la proxima señal que cuyos datos se almacenan en next_sign
nuevo_alumno<-data.frame(trabajo=c(2,9),examen=c(3,8))

# modelo de prediciión
prono1<-knn(train = tabla_alumnos[-c(3)], test = nuevo_alumno, cl = tabla_alumnos$interes)
prono1
```

```
## [1] 3 1
## Levels: 1 2 3
```

```r
# en otros ejemplo puede ser interesante incrementa el numero de vecinos que se analizan con el parametr
knn(train = tabla_alumnos[-c(3)], test = nuevo_alumno, cl = tabla_alumnos$interes, k = 4)
```

```
## [1] 3 1
## Levels: 1 2 3
```

3.3. Estandarización

Para otros casos en los que las variables no tengan la misma escala, es preferible para mejorar el modelo normalizar las columnas de datos numéricos.

Esto puede hacerse con muchas funciones predefinidas como por ejemplo la función `scale()` o `data.Normalization()` esta del paquete clusterSim.

Hay que tener en cuenta que cuando normalizamos los valores de hechos que pasamos a predict(), deben ser normalizados con el mismo algoritmo.

```
# normalizamos la tabla de datos
    tabla_alumnos.nor<-scale(tabla_alumnos)
    str(tabla_alumnos.nor)
```

```
##  num [1:20, 1:3] 1.659 -0.529 0.201 0.565 0.565 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:3] "trabajo" "examen" "interes"
##  - attr(*, "scaled:center")= Named num [1:3] 5.45 5.9 2
##   ..- attr(*, "names")= chr [1:3] "trabajo" "examen" "interes"
##  - attr(*, "scaled:scale")= Named num [1:3] 2.743 2.553 0.858
##   ..- attr(*, "names")= chr [1:3] "trabajo" "examen" "interes"
```

```
# extraemos los atributos de centro y scala de la transformación
    attr(tabla_alumnos.nor,"scaled:center")
```

```
## trabajo  examen interes
##    5.45    5.90    2.00
```

```
    attr(tabla_alumnos.nor, "scaled:scale")
```

```
##   trabajo    examen   interes
## 2.7429335 2.5526044 0.8583951
```

```
# Transformamos un una nota examen de 9 para pronostico porterior
# es la  2 col
    nota.t<-scale(9,
                  attr(tabla_alumnos.nor,"scaled:center")[2],
                  attr(tabla_alumnos.nor, "scaled:scale")[2])

    nota.t # valor de nota exam =9 transformado
```

```
##          [,1]
## [1,] 1.214446
## attr(,"scaled:center")
## examen
##    5.9
## attr(,"scaled:scale")
##   examen
## 2.552604
```

4. Naive Bayes- clasificación bayesiano ingenuo

Naive Bayes es un modelo de predicción basado en la probabilidad Bayesiana. El modelo es muy simple, pero poderoso, en cuanto que es resultado directo de los datos y su tratamiento con simple estadística bayesiana de la probabilidad condicionada. Hay que tener en cuenta que se asume, por simplificación que las variables son todas sucesos independientes.

LA función de clasificación ingenua de bayes se encuentra en varias librerías de R en: `naivebayes`, en el paquete `e1071`y en otros.

El modelo bayesiano de probabilidad condicionada se representa como: $P(A|B) = P(A \cap B)/P(B)$

Es decir, la probabilidad de que se de el caso A dado B es igual a la probabilidad de la intersección de A con B ($A \cap B$ partido la probabilidad de B.

Estirando esta formulación llegaríamos al teorema de Bayes cuya expresion mÁs típica es la siguente:

$P(A|B) = PP(B|A) * P(A)/P(B)$

4.1. Crear un modelo con `naivebayes`

La `tabla_1` que vamos a crear contiene 3 variables: la hora del día, el lugar donde está Juan a esa hora, y otra columna que nos indica si es o no fin de semana con un valor lógico (*TRUE o FALSE*).

Vamos a crear la tabla para el ejemplo:

```
# leemos la base de datos
# tabla_1<-read.csv("tabla_1.csv",header = TRUE)

# vamos a crear el ejemplo de cero: CREAMOS UNA TABLA DE DATOS
tabla_1<-data.frame(hora=c(8,14,24,8,14,24,8,14,24,8,14,24,8,14,24,8,14,24,24,24))
tabla_1$lugar<-c("casa","restaurante","casa",
                 "trabajo","trabajo","casa",
                 "trabajo","trabajo","casa",
                 "casa","restaurante","casa",
                 "trabajo","trabajo","casa",
                 "casa","restaurante","casa","cine","cine")
tabla_1$finde<-c(T,T,T,
                 F,F,F,
                 F,F,F,
                 T,T,T,
                 F,F,F,
                 T,T,T,
                 T,F
                )
str(tabla_1)
```

```
## 'data.frame':    20 obs. of  3 variables:
##  $ hora : num  8 14 24 8 14 24 8 14 24 8 ...
##  $ lugar: chr  "casa" "restaurante" "casa" "trabajo" ...
##  $ finde: logi  TRUE TRUE TRUE FALSE FALSE FALSE ...
```

```
head(tabla_1)
```

```
##   hora       lugar finde
## 1    8        casa  TRUE
## 2   14 restaurante  TRUE
## 3   24        casa  TRUE
## 4    8     trabajo FALSE
## 5   14     trabajo FALSE
## 6   24        casa FALSE
```

```
# vemos como ejemplo el numero de registros de hora según el lugar
table(tabla_1$hora,tabla_1$lugar)
```

```
##
##      casa cine restaurante trabajo
##   8     3    0           0       3
##   14    0    0           3       3
##   24    6    2           0       0
```

Como vemos, es una tabla con 20 regisros y 3 variables en columnas, sobre la que queremos practicar pronósticos bayesianos de probabilidad condicionada.

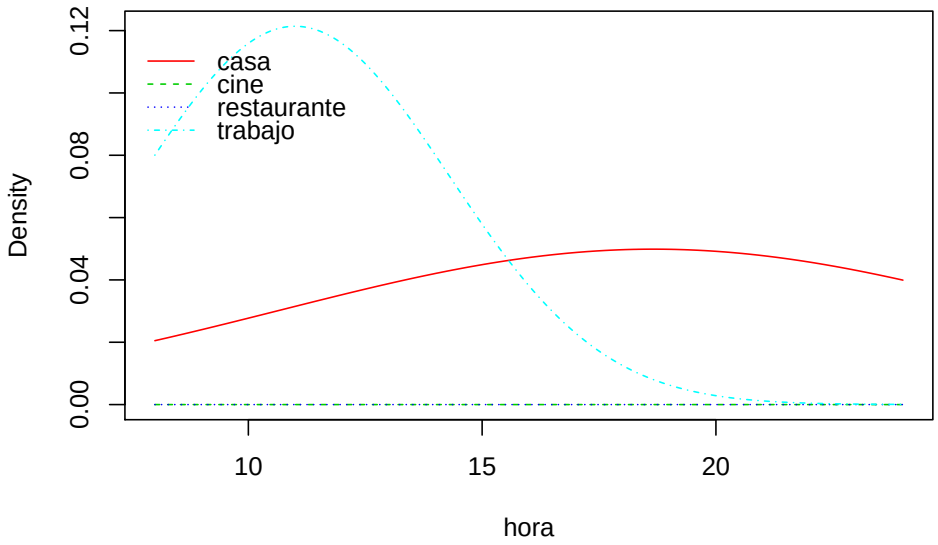
Vamos cargar la librería `naivebayes` con objeto de crear un modelo de pronóstico de la variable dependiente `lugar` a partir de las variables independientes `hora` y `finde`. Este modelo nos diría por ejemplo la probabilidad de que: sabiendo la hora y si es o no fin de semana, Juan se encuentre en un lugar determinado.

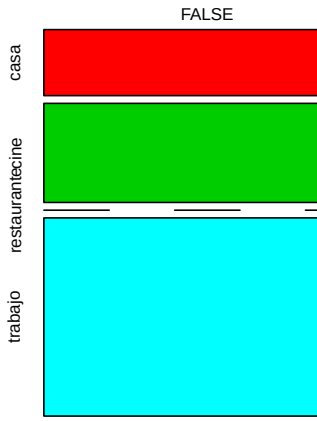
Como vimos en las consideraciones previas, los modelos de pronostico bayesiano, y en particular `naivebayes` funcionan muy mal con datos numéricos continuos, y vamos a ver la prueba, pues crearemos un modelo con la variable hora tal cual, y despues haremos el mismo modelo con la variable hora convertida en factor.

Primero creamos la formula de modelo con la función `naive_bayes()` y luego definimos un hecho, una ocurrencia concreta de los parámetros y llamamos a la función `predict()`.

Esta función es común a la mayoría de los modelos de predictivos, y sus argumentos son el nombre del modelo y un hecho almacenado como data.frame. Si añadimos el argumento `type="prob` nos da el resultado como pronostico probabilistico y si no, solo el pronóstico más probable.

```
# cargamos la librería
    library(naivebayes)
# creamos el modelo de pronostico
    m <- naive_bayes(lugar ~ hora+finde, data = tabla_1)#, laplace = 1)

    # representamos graficamente el modelo
    plot(m)
```






```
# ejecutando predict(modelo) tenemos los resultados de pronostico para cada registro de datos
    tabla_1$p=predict(m)
    head(tabla_1)
```

```
##   hora       lugar finde           p
## 1    8        casa  TRUE        casa
## 2   14 restaurante  TRUE restaurante
## 3   24        casa  TRUE        cine
## 4    8     trabajo FALSE     trabajo
## 5   14     trabajo FALSE     trabajo
## 6   24        casa FALSE        cine
```

```
# pero si queremos un hecho concreto:
# creamos un hecho a priori, sobre el que queremos pronosticar el resultado
# como el modelo es  lugar ~ hora+finde, aportamos un dato de hora y otro de finde
# en este caso queremos pronosticar donde se encuentra Juan a las 14 horas un día laborable
    h<-data.frame(hora= 24, finde=T)
    table(tabla_1$lugar,tabla_1$hora+tabla_1$finde)
```

```
##
##              8 9 14 15 24 25
##   casa       0 3  0  0  3  3
##   cine       0 0  0  0  1  1
##   restaurante 0 0  0  3  0  0
##   trabajo    3 0  3  0  0  0
```

```
# llamamos a la función de predición
    predict(m,h)
```

```
## [1] cine
## Levels: casa cine restaurante trabajo
```

```
# idam con la probabilidad completa
    predict(m,h, type = "prob")
```

```
##              casa      cine  restaurante trabajo
## [1,] 0.0006001881 0.9993923 7.515315e-06       0
```

La predicción que obtenemos con el modelo para *(hora= 24, finde=T)* es claramente erronea, pues solo 1 de los 4 registros que tenemos a las 24 horas en fin de semana es `ir al cine`, los otros 3 son `estar en casa`, por lo que algo falla en el modelo al ser el evento más probable `estar en casa`.

Este problema es habitual cuando usamos datos continuos, que nos generan distribuciones de probabilidad continuas. En este caso el evento de ir al cine tiene muy pocos datos, pero siempre a las 24 horas, por lo que la media se mantiene en 24 h. Sin embargo el hecho estar en casa tienen muchos registros en diferentes horas, por lo que el valor medio de la hora es un número intermedio 18,6 (ver el modelo m para más información).

Para evitar problemas debemos transformar las variables continuas en discretas y reducir al máximo los valores posibles realizando lo que denominamos una categorización previa de los datos. Por ejemplo convietiendo los datos en factores.

```
# Convertimos la variable continua numerica hora, en factor discreto
    tabla_1$hora<-as.factor(tabla_1$hora)
    str(tabla_1)
```

```
## 'data.frame':    20 obs. of  4 variables:
##  $ hora : Factor w/ 3 levels "8","14","24": 1 2 3 1 2 3 1 2 3 1 ...
##  $ lugar: chr  "casa" "restaurante" "casa" "trabajo" ...
##  $ finde: logi  TRUE TRUE TRUE FALSE FALSE FALSE ...
##  $ p    : Factor w/ 4 levels "casa","cine",..: 1 3 2 4 4 2 4 4 2 1 ...
```

```
# calculamos de nuevo el modelo ahora
    m <- naive_bayes(lugar ~ hora+finde, data = tabla_1)
# Hacemos de nuevo la predicción
    predict(m,h)
```

```
## [1] casa
## Levels: casa cine restaurante trabajo
```

```
    predict(m,h, type="prob")
```

```
##           casa      cine   restaurante trabajo
## [1,] 0.8571429 0.1428571 1.432592e-123       0
```

```
    # ojo al crear el hecho que debe ser acorde a los datos,
# si es factor debe contener en levels los mismos que la tabla origen
# por ello lo creamos a partir de esta tabla mejor
    h<-tabla_1[1,c(1,3)]
        h$hora="24"
        h$finde=F
    predict(m,h)
```

```
## [1] casa
## Levels: casa cine restaurante trabajo
```

Como hemos visto al transformar en factor la variable numérica continua, hemos realizado un pronostico más acorde con los datos.

4.2. e1072

Vamos a probar otro paquete que contienen a naive_Bayes el `e1071`. Usaremos los datos de supervivientes del Titanic que vienen en los datasets de R por defecto. La tabla de datos tiene 32 filas pero en realidad esconde en la columna *freq* el número de repeticiones de cada caso, por lo que el primer paso es crear una tabla completa.

```
    library(e1071)
    #Cargamos los datos del Titanic dese datasets
    data("Titanic")
    #Los almacenamos en un data frame
    Titanic_df=as.data.frame(Titanic)
    str(Titanic_df)
```

```
## 'data.frame':    32 obs. of  5 variables:
##  $ Class   : Factor w/ 4 levels "1st","2nd","3rd",..: 1 2 3 4 1 2 3 4 1 2 ...
##  $ Sex     : Factor w/ 2 levels "Male","Female": 1 1 1 1 2 2 2 2 1 1 ...
##  $ Age     : Factor w/ 2 levels "Child","Adult": 1 1 1 1 1 1 1 1 2 2 ...
##  $ Survived: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Freq    : num  0 0 35 0 0 0 17 0 118 154 ...
```

```
    #Creamos una tabla de casos competos a partir de la frecuencia de cada uno
    repeating_sequence=rep.int(seq_len(nrow(Titanic_df)), Titanic_df$Freq)
    #Esto repite cada caso según la frecuencia dada en la col de la tabla.

    #Creamos una nueva tabla repitiendo los casos según el modelo anterior.
    Titanic_dataset=Titanic_df[repeating_sequence,]
    #Ya no necesitamos la tabla de frecuencias más.
    Titanic_dataset$Freq=NULL
    head(Titanic_dataset)
```

```
##     Class  Sex   Age Survived
## 3     3rd Male Child       No
## 3.1   3rd Male Child       No
## 3.2   3rd Male Child       No
## 3.3   3rd Male Child       No
## 3.4   3rd Male Child       No
## 3.5   3rd Male Child       No
```

```
    # todo son factores
    str(Titanic_dataset)
```

```
## 'data.frame':    2201 obs. of  4 variables:
##  $ Class   : Factor w/ 4 levels "1st","2nd","3rd",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ Sex     : Factor w/ 2 levels "Male","Female": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age     : Factor w/ 2 levels "Child","Adult": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Survived: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
```

```
    # Ajustamos un modelo de naive bayes con la librería e1071
        m.e1071 <- naiveBayes(Survived ~ ., data = Titanic_dataset)
        m.e1071 # vemos el modelo
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##       No      Yes
## 0.676965 0.323035
##
## Conditional probabilities:
##      Class
## Y            1st        2nd        3rd       Crew
##   No  0.08187919 0.11208054 0.35436242 0.45167785
##   Yes 0.28551336 0.16596343 0.25035162 0.29817159
##
##      Sex
## Y           Male     Female
##   No  0.91543624 0.08456376
##   Yes 0.51617440 0.48382560
##
##      Age
## Y          Child      Adult
##   No  0.03489933 0.96510067
##   Yes 0.08016878 0.91983122
```

```
    # realizamos la prediccion con el modelo
    predicciones.m<-predict(m.e1071,Titanic_dataset)

    # Matriz de confusión
    table(predicciones.m,Titanic_dataset$Survived)
```

```
##
## predicciones.m   No  Yes
##            No  1364  362
##            Yes  126  349
```

```
    hecho<-data.frame(Class="1rd",Sex="Female",Age="Child")
    predict(m.e1071,hecho)
```

```
## [1] Yes
## Levels: No Yes
```

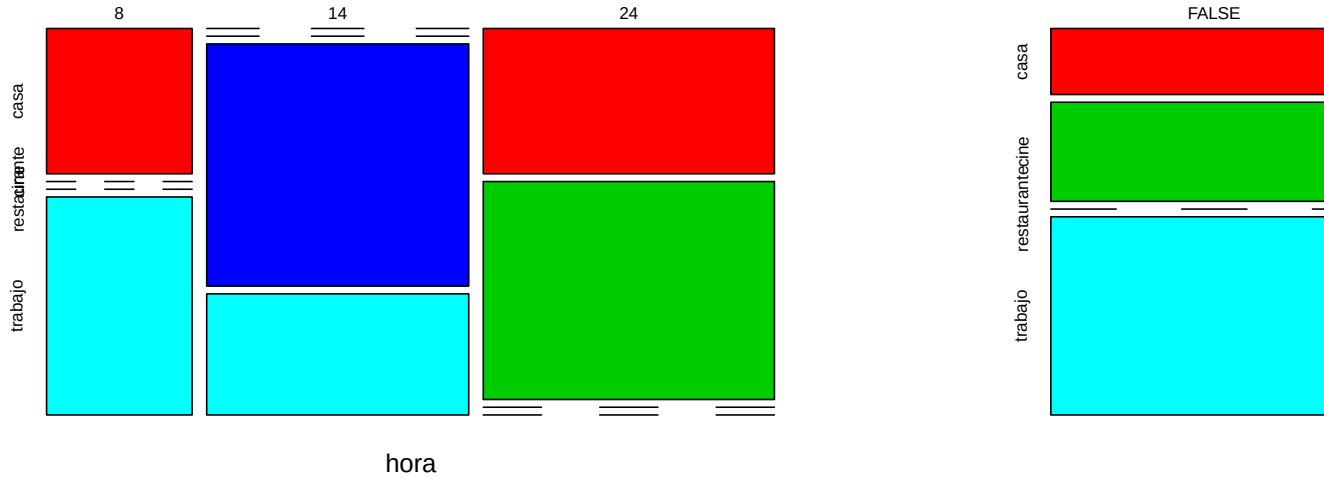
4.3. Corrección de laplace

En muchas ocasiones los datos no contienen muestras a priori de todas las combiaciones de variables posibles, por lo que las probabilidades de casos raros salen excesivamente bajas. Para corregir esto el modelo `naiveBayes` tiene la opcion de añadir en la fórmula el argumento de `laplace=1`, en el que indicamos que, al menos, se debe contar con una aparición de cada posible combinación de factores. Este parámetro se puede aumentar a criterio del investigador, y permite incorporar casos raros dentro del pronóstico que de otra forma, por la simplificación del modelo de Bayes, darían probabilidad cero.

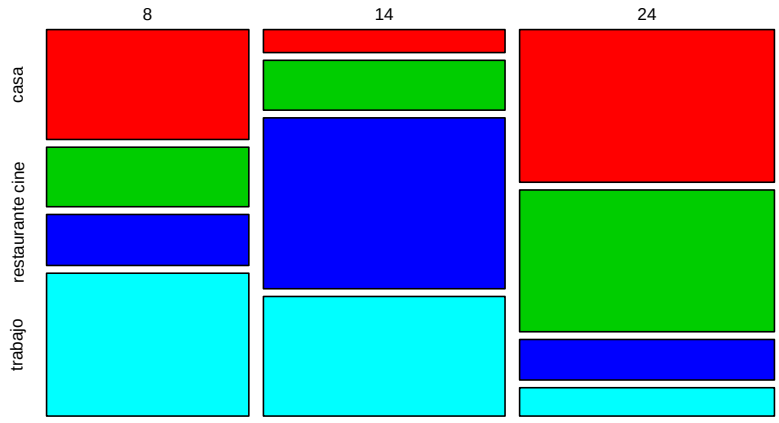
Por ejemplo en los datos de Juan y la ubicación según las horas, no tenemos ningún registro de que vaya a trabajar en fin de semana, pero eso no significa que no tengamos cierta probabilidad, lo que se podría solventar añadiendo al modelo `laplace=1`.

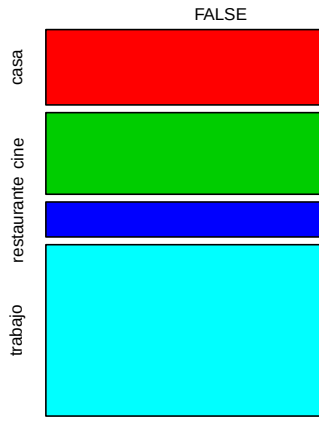
Veamos los cambios al reformular el modelo con *laplace*. Gráficamente se aprecia que, por ejemplo, la probabilidad de ir a restaurante entre semana pasa de cero a una cantidad pequeña, y de trabajar el fin de semana igual (pasa de cero a una proporcion).

```
# cargamos la librería
    require(naivebayes)
# Modelo de pronostico sin laplace
    m <- naive_bayes(lugar ~ hora+finde, data = tabla_1)
    plot(m)
```




```
# cambiamos el modelo añadiendo laplace con al menos una ocurencia por evento
    m1 <- naive_bayes(lugar ~ hora+finde, data = tabla_1, laplace=1)
    plot(m1)
```






```
    # ¿ donde está Juan si son las 14 horas en fin de semana?
    h$hora=8
    h$finde=T
    predict(m,h,type="prob")
```

```
##           casa      cine  restaurante trabajo
## [1,] 0.8571429 0.1428571 2.165259e-12       0
```

```
    predict(m1,h,type="prob")
```

```
##           casa      cine restaurante      trabajo
## [1,] 0.8790698 0.1209302 1.91539e-39 4.420521e-77
```

5. Regresión logistica binaria

Otro modelo de predicción de aprendizaje supervisado es el de **regresión logística**. Se trata de un tipo de análisis de regresión utilizado para predecir el resultado de una variable categórica (aquella que puede adoptar un número limitado de categorías) en función de las variables predictoras. Este modelo se enmarca dentro de los modelos denominados de *predicción lineal generalizados* o *glm* como son conocidos por sus siglas en inglés.

Con el adjetivo binario no s referimos a las predicciones sobre variables binarias o dicotómicas que simplemente tratan de decir si algo es 1 o 0, SI o NO.

Este modelo de pronóstico se usa mucho en variables que se distribuyen en forma de binomial. La binomial es una distribución de probabilidad discreta que cuenta el número de éxitos en una secuencia de n ensayos. Si el evento de *éxito* tiene una probabilidad de ocurrencia `p`, la probabilidad del evento contratio -el de *fracaso*- tendr´una probabilidad de $q = 1 - p$. En la distribución binomial se repite el experimento de exito -fracaso n veces, de forma independiente, y se trata de calcular la probabilidad de un determinado número de éxitos d, en esas n repeticiones $B(n, p)$.

La denominación de *logística* se debe precisamente a la forma de la propia función de distribución de probabilidad binomial que presenta un crecimiento exponencial y que se parece a una S y que toma el nombre matemático de función logística $\frac{1}{1+e^{-t}}$.

Esta curva, es una aproximación continua a la funcion discreta binaria, pues el cambio de 0 a 1 se produce en corto espacio y muy pronunciado. Si usáramos otras funciones como la lineal para la regresión de datos ninarios funcionaría muy mal, pues el ajuste lineal no capta bien la forma de los datos, las dos agrupaciones que buscamos separar o clasificar.

Los modelos de regresión logísticos se generan con la función `glm()` del paquete base R `stats`, de la siguiente manera.

```
m <- glm(y ~ x1 + x2 + x3,
           data = my_dataset,
           family = "binomial")

prob <- predict(m, test_dataset, type = "response")

pred <- ifelse(prob > 0.50, 1, 0)
```

Importante reseñar que la predicción se da en **modo de probabilidad**, por lo que para evaluar un pronóstico concreto, se debe establecer qué umbral es el que fija el pronostico 0 o 1. En el caso del ejemplo anterior se ha determinado que para `pred>0,5` el pronostico es 1.

5.1. Construir modelos `glm`

Siguiendo con el uso de la base de datos de ejemplo de supervivientes del titanic, vamos a crear un modelo logistico que pronostique la variable *Survived*. Podemos ver como se crearon los datos en el apartado de particiones de los datos

Al igual que todos los modelos de aprendizaje, el modelo se compone de una fórmula, y luego se pronostica con la función `predict()`. En los modelos glm(), la unica opcion de predict() es esponse y terms. El primer caso da directamente la probabilidad de la respuesta y el segundo proporciona los coeficientes de cada termino en la fórmula. Para obtener una predicción usaremos `type= "response"`.

```
# echamos un vistazo a los datos
    head(Titanic_data)
```

```
##     Class  Sex   Age Survived
## 3     3rd Male Child       No
## 3.1   3rd Male Child       No
## 3.2   3rd Male Child       No
## 3.3   3rd Male Child       No
## 3.4   3rd Male Child       No
## 3.5   3rd Male Child       No
```

```
    table(Titanic_data$Survived)
```

```
##
##   No  Yes
## 1490  711
```

```
# creamos una partición para crear un conjunto de test y otro de entrenamiento
    library(caret)
    set.seed(123)
# creamos un vector de particion sobre la variable Survived
# el tamaño de muestra será de 75%
    trainIndex=createDataPartition(Titanic_data$Survived, p=0.70)$Resample1
# definimoslos dos conjuntos de muestra
    d_titanic_train=Titanic_data[trainIndex, ] # conjunto entrenamiento
    d_titanic_test= Titanic_data[-trainIndex, ] # conjunto de test
```

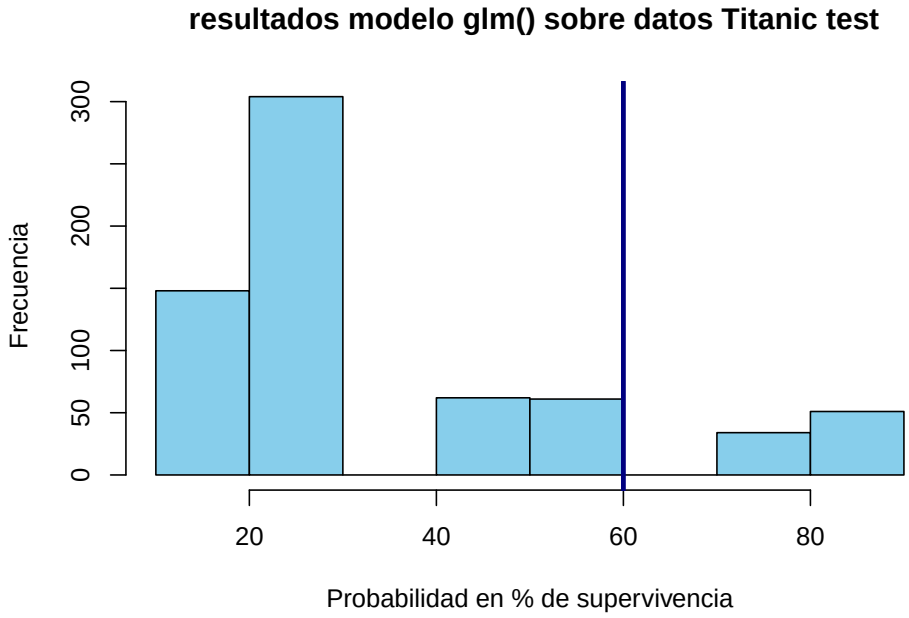
Una vez tenemos los conjuntos de test y de aprendizaje creamos el modelo, usando la misma simbología que en el caso de los modelos de naive_bayes. La peculiaridad de glm() es que tenemos que identificar un umbral de probabilidad a partir del que consideramos el pronostico 0 o 1.

```
    # Construimos el modelo de predicción con la función glm
    m_glm <- glm(Survived ~ Class+Sex, data = d_titanic_train, family = "binomial")
    # resumen del modelo
    summary(m_glm)
```

```
##
## Call:
## glm(formula = Survived ~ Class + Sex, family = "binomial", data = d_titanic_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1346  -0.7499  -0.4644   0.7435   2.1356
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -0.2856     0.1678  -1.702   0.0888
## Class2nd     -1.0257     0.2352  -4.362 1.29e-05
## Class3rd     -1.8870     0.2093  -9.017  < 2e-16
## ClassCrew    -0.8394     0.1911  -4.393 1.12e-05
## SexFemale     2.4557     0.1698  14.463  < 2e-16
##
## (Intercept) .
## Class2nd    ***
## Class3rd    ***
## ClassCrew   ***
## SexFemale   ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1939.3  on 1540  degrees of freedom
## Residual deviance: 1560.7  on 1536  degrees of freedom
## AIC: 1570.7
##
## Number of Fisher Scoring iterations: 4
```

```
    # vemos las predicciones en el conjunto de test
    d_titanic_test$pred<-predict(m_glm, d_titanic_test, type= "response")
```

```
    # Hacemos el resumen gráfico del resultado
        hist(100*d_titanic_test$pred, col="skyblue",
             main=" resultados modelo glm() sobre datos Titanic test",
             xlab="Probabilidad en % de supervivencia",
             ylab="Frecuencia")
    # Marcamos un umbral en el que consideramos el pronostico como donación
    # este umbral lo ponemos en un valor del 60%
        abline(v= 60,col= "navy", lwd=3)  # marcamos el umbral de supervivencia
```




```
        d_titanic_test$pred_final_60 <- ifelse(d_titanic_test$pred > 0.6, 1, 0)
        # resumen de resultados
        table(d_titanic_test$pred_final_60)
```

```
##
##   0   1
## 575  85
```

```
    # podemos calcular el ajuste respecto a los casos reales con esta sencilla formula
    # antes vamos a cambiar los levels de survived No=0, Yes=1
        table(d_titanic_test$Survived) # vemos cual es el primero ---> No
```

```
##
##  No Yes
## 447 213
```

```
        levels(d_titanic_test$Survived) <- c(0,1)
        mean(d_titanic_test$pred_final_60 == d_titanic_test$Survived)
```

```
## [1] 0.7878788
```

Como vemos una vez realizado el pronostico podriamos probar diferentes umbrales y ver cual es el que da un mejor resultado con esta metodología.

5.2. curvas ROC y AUC

Estas curvas nos ayudan a controlar el acierto o no de los modelos cuando uno de los eventos es muy raro. Esto implica que predecir el evento opuesto conlleva un gran porcentaje de aciertos, y en cierta forma falsea la utilidad real de la predición lo que hay que vigilar y entender.

En estos casos es mejor sacrificar los aciertos generales en favor de concentrarlos sobre uno de los resultados, el más raro, el que buscamos distinguir.

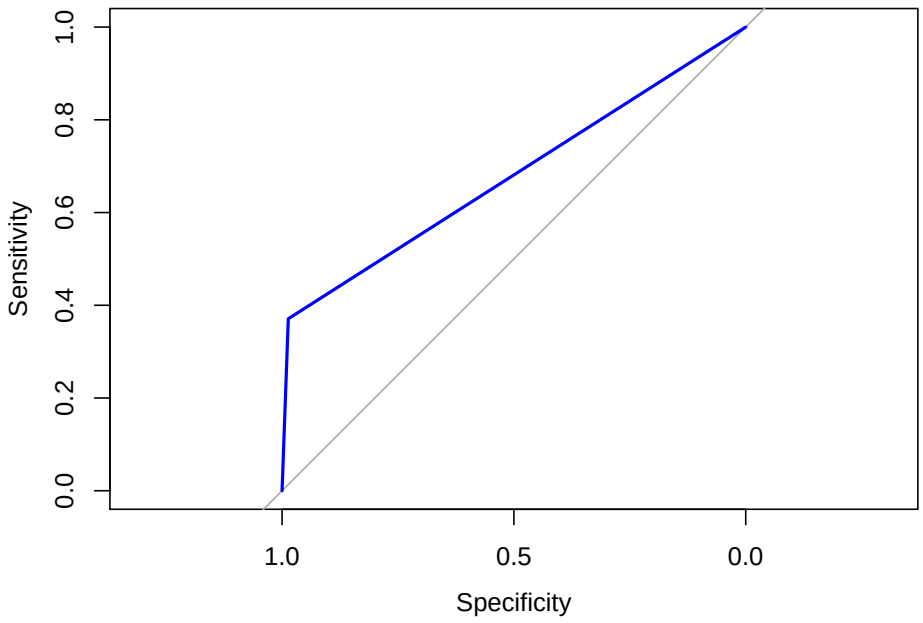
Por lo tanto la exactitud de la predicción general es una medida engañosa en el rendimiento de lo que realmente nos interesa. Este es un caso muy común en predicciones binomiales pues un caso, el de exito puede tener una probabilidad general mucho menor que el de fracaso, y un porcentaje de acierto elevado, puede no tener importancia, pues lo que nos interesa no es acertar los fracasos sino los exitos.

Las curvas ROC son buenas para evaluar este problema en conjuntos de datos desequilibrados.

Al hacer una gráfica **ROC** se representa mejor la compensación entre un modelo que es demasiado agresivo y uno que es demasiado pasivo. Lo que interesa es que el area de la curva sea máxima, cercana a 1, por lo que cuanto más se eleve respecto de la linea media mejor.

Estas graficas se pintan con la libraría `pROC`. Usaremos dos funciones una para pintar la gráfica y otra que calcula el *AUC* o área bajo la curva.

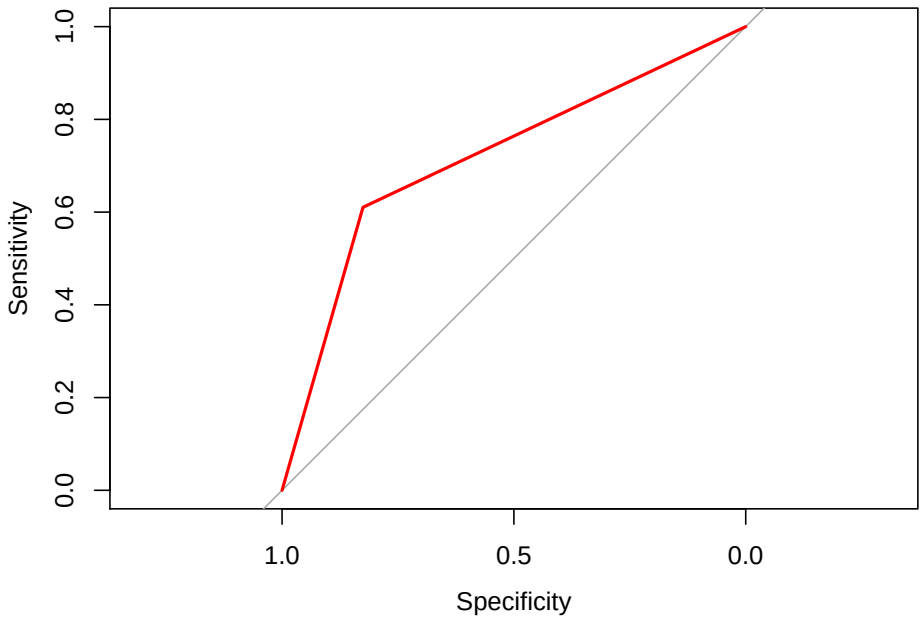
```
# Cargamos la libraría de graficos ROC
library(pROC)

# Creamos una curva ROC basada en el modelo glm anterior
ROC_glm60 <- roc(d_titanic_test$Survived, d_titanic_test$pred_final_60)

# Pintamos la grafica ROC
plot(ROC_glm60, col = "blue")
```



```
    #plot(ROC_naive, col = "red")

    # Calculamos el area bajo la ROC(AUC)
    auc(ROC_glm60)
```

```
## Area under the curve: 0.6787
```

```
    d_titanic_test$pred_final_40 <- ifelse(d_titanic_test$pred > 0.4, 1, 0)
    ROC_glm40 <-roc(d_titanic_test$Survived, d_titanic_test$pred_final_40)
    # Pintamos la grafica ROC
    plot(ROC_glm40, col = "red")
```




```
    auc(ROC_glm40)
```

```
## Area under the curve: 0.7179
```

Vistos los resultados, el seleccionar un umbral de 40, mejora la predicción de casos positivos de supervivencia.

5.3. Modelos de impacto combinado

En las formulaciones de modelos `glm` podemos expresar lo que se denominan impactos combinados o interacciones entre variables. Estos casos se dan cuando el efecto combinado de dos variables es muy importante y superior a la combinación lineal de ellas. Es decir el efecto es exponencial y no lineal sobre la variable a predecir.

5.3.1. Ejemplo

Uno de los mejores predictores de donaciones futuras es el historial de donaciones anteriores y cuanto mas recientes, frecuentes y grandes mejor. En términos de comercialización, esto se conoce como R/F/M (Recency Frequency Money).

Es muy probable que el impacto combinado de reciente y frecuencia puede ser mayor que la suma de los efectos por separado, si uno ha dado dinero a una ONG hace muy poco será poco probable que de otra vez enseguida.

Debido a que estos predictores juntos tienen un mayor impacto en la variable dependiente, su efecto conjunto **debe modelarse como una interacción**. Esto en la formulación del modelo se identifica por un * en lugar de un +.

```
# Leemos la tabla de datos
    donors<-read.csv("donors.csv",header = TRUE)
    head(donors)
```

```
##   donated veteran bad_address age has_children
## 1       0       0           0  60            0
## 2       0       0           0  46            1
## 3       0       0           0  NA            0
## 4       0       0           0  70            0
## 5       0       0           0  78            1
## 6       0       0           0  NA            0
##   wealth_rating interest_veterans
## 1             0                 0
## 2             3                 0
## 3             1                 0
## 4             2                 0
## 5             1                 0
## 6             0                 0
##   interest_religion pet_owner catalog_shopper
## 1                 0         0               0
## 2                 0         0               0
## 3                 0         0               0
## 4                 0         0               0
## 5                 1         0               1
## 6                 0         0               0
##   recency  frequency  money
## 1 CURRENT   FREQUENT MEDIUM
## 2 CURRENT   FREQUENT   HIGH
## 3 CURRENT   FREQUENT MEDIUM
## 4 CURRENT   FREQUENT MEDIUM
## 5 CURRENT   FREQUENT MEDIUM
## 6 CURRENT INFREQUENT MEDIUM
```

```
    str(donors)
```

```
## 'data.frame':    93462 obs. of  13 variables:
##  $ donated          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ veteran          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ bad_address      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ age              : int  60 46 NA 70 78 NA 38 NA NA 65 ...
##  $ has_children     : int  0 1 0 0 1 0 1 0 0 0 ...
##  $ wealth_rating    : int  0 3 1 2 1 0 2 3 1 0 ...
##  $ interest_veterans: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ interest_religion: int  0 0 0 0 1 0 0 0 0 0 ...
##  $ pet_owner        : int  0 0 0 0 0 0 1 0 0 0 ...
##  $ catalog_shopper  : int  0 0 0 0 1 0 0 0 0 0 ...
##  $ recency          : Factor w/ 2 levels "CURRENT","LAPSED": 1 1 1 1 1 1 1 1 1 1 ...
```

```
##  $ frequency        : Factor w/ 2 levels "FREQUENT","INFREQUENT": 1 1 1 1 1 2 2 1 2 2 ...
##  $ money            : Factor w/ 2 levels "HIGH","MEDIUM": 2 1 2 2 2 2 2 2 2 2 ...
```

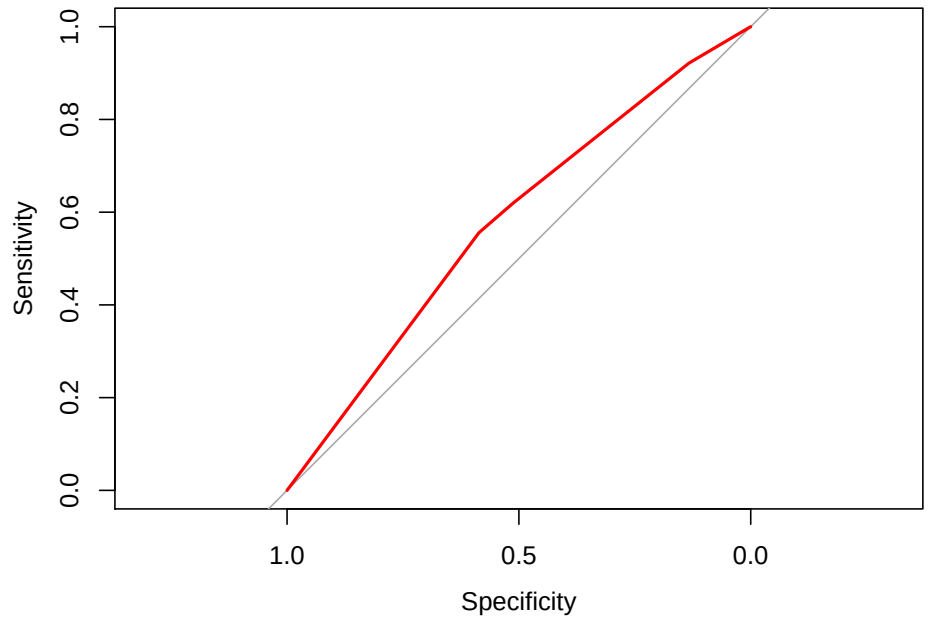
```
# Construimos un modelo complejo
    rfm_model <- glm(donated ~ money + recency* frequency ,data = donors,family = "binomial")

# Resumen del modelo RFM
    summary(rfm_model)
```

```
##
## Call:
## glm(formula = donated ~ money + recency * frequency, family = "binomial",
##     data = donors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.3696  -0.3696  -0.2895  -0.2895   2.7924
##
## Coefficients:
##                                   Estimate
## (Intercept)                       -3.01142
## moneyMEDIUM                        0.36186
## recencyLAPSED                     -0.86677
## frequencyINFREQUENT               -0.50148
## recencyLAPSED:frequencyINFREQUENT  1.01787
##                                   Std. Error
## (Intercept)                          0.04279
## moneyMEDIUM                          0.04300
## recencyLAPSED                        0.41434
## frequencyINFREQUENT                  0.03107
## recencyLAPSED:frequencyINFREQUENT    0.51713
##                                   z value
## (Intercept)                       -70.375
## moneyMEDIUM                         8.415
## recencyLAPSED                      -2.092
## frequencyINFREQUENT               -16.143
## recencyLAPSED:frequencyINFREQUENT   1.968
##                                   Pr(>|z|)
## (Intercept)                         <2e-16 ***
## moneyMEDIUM                         <2e-16 ***
## recencyLAPSED                       0.0364 *
## frequencyINFREQUENT                 <2e-16 ***
## recencyLAPSED:frequencyINFREQUENT   0.0490 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 37330  on 93461  degrees of freedom
## Residual deviance: 36938  on 93457  degrees of freedom
## AIC: 36948
##
## Number of Fisher Scoring iterations: 6
```

```
    #summary(rfm_model)$coefficients
# Calculamos las predicciones del modelo RFM
    rfm_prob <- predict(rfm_model, type = "response")
    head(rfm_prob)
```

```
##          1          2          3          4
## 0.06601640 0.04691282 0.06601640 0.06601640
##          5          6
## 0.06601640 0.04105058
```

```
# Pintamos la curva ROC para ver el efecto del modelo y calculamos el area AUC
    require(pROC)
    ROC <- roc(donors$donated, rfm_prob)
    plot(ROC, col = "red")
```




```
    auc(ROC)
```

```
## Area under the curve: 0.5785
```

5.4. Optimización de un modeloS `glm`

Cuando a priori no sabemos qué variables tienen más dependencia para crear el modelo una forma de hacerlo es usando la regresión gradual. Esto consiste en aplicar una funCión que va incrementando las variables y detecta el mejor modelo de regresión.

Para construirlo hacermos lo siguiente:

1 creamos un modelo `glm()` sin predictores. se hace estableciendo la variable explicativa igual a 1. 2 Se crea otro modelo con todos las variables usando `~ .`. 3 Se aplica la función `step()` entre ambos modelos para realizar una regresión progresiva hacia adelante. Debe indicarse la dirección con `direction = "forward"` 4 Usamos la función `predict()` sobre la lista de modelos creados con `step`

Veamos el ejemplo:

```r
    # 1. Modelo sin predictores
        null_model <- glm(donated ~1, data = donors, family = "binomial")

    # 2. modelo completo
        full_model <- glm(donated ~ ., data = donors, family = "binomial")

    # 3. funcion step ()
        step_model <- step(null_model, scope = list(lower = null_model, upper = full_model), direction =
```

```
## Start:  AIC=37332.13
## donated ~ 1
##
##                     Df Deviance   AIC
## + frequency          1    28502 37122
## + money              1    28621 37241
## + wealth_rating      1    28705 37326
## + has_children       1    28705 37326
## + age                1    28707 37328
## + interest_veterans  1    28709 37330
## + catalog_shopper    1    28710 37330
## + pet_owner          1    28711 37331
## <none>                    28714 37332
## + interest_religion  1    28712 37333
## + recency            1    28713 37333
## + bad_address        1    28714 37334
## + veteran            1    28714 37334
##
## Step:  AIC=37024.77
## donated ~ frequency
##
##                     Df Deviance   AIC
## + money              1    28441 36966
## + wealth_rating      1    28493 37018
## + has_children       1    28494 37019
## + interest_veterans  1    28498 37023
## + catalog_shopper    1    28499 37024
## + age                1    28499 37024
## + pet_owner          1    28499 37024
## <none>                    28502 37025
## + interest_religion  1    28501 37026
## + recency            1    28501 37026
## + bad_address        1    28502 37026
## + veteran            1    28502 37027
##
## Step:  AIC=36949.71
## donated ~ frequency + money
##
##                     Df Deviance   AIC
## + wealth_rating      1    28431 36942
## + has_children       1    28432 36943
## + interest_veterans  1    28438 36948
## + catalog_shopper    1    28438 36949
## + age                1    28439 36949
## + pet_owner          1    28439 36949
```

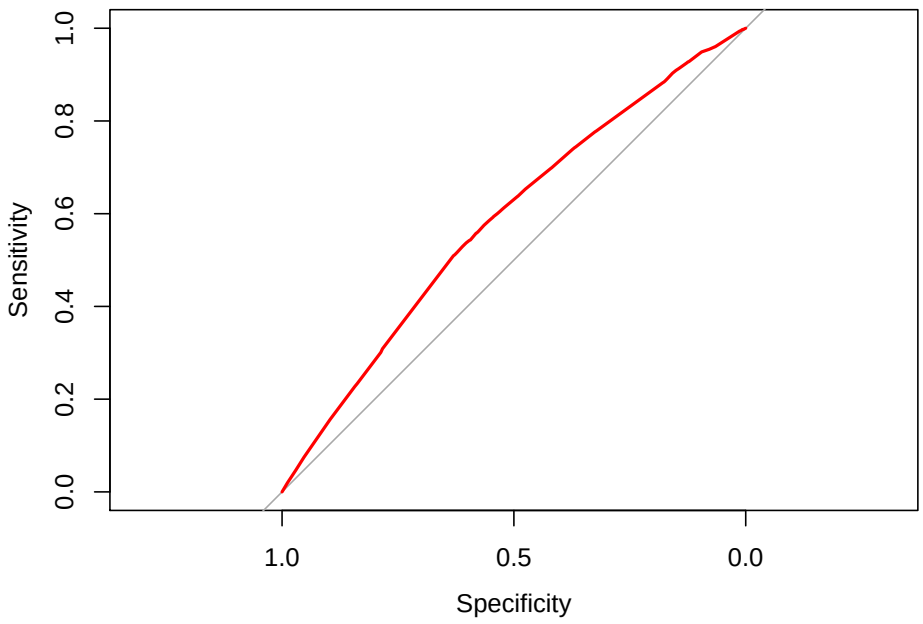
```
## <none>                    28441 36950
## + interest_religion  1    28440 36951
## + recency            1    28441 36951
## + bad_address        1    28441 36951
## + veteran            1    28441 36952
##
## Step:  AIC=36945.26
## donated ~ frequency + money + wealth_rating
##
##                      Df Deviance   AIC
## + has_children        1    28421 36937
## + interest_veterans  1    28429 36945
## + catalog_shopper    1    28429 36945
## + age                1    28429 36945
## <none>                    28431 36945
## + pet_owner          1    28430 36945
## + interest_religion  1    28431 36947
## + recency            1    28431 36947
## + bad_address        1    28431 36947
## + veteran            1    28431 36947
##
## Step:  AIC=36938.08
## donated ~ frequency + money + wealth_rating + has_children
##
##                      Df Deviance   AIC
## + pet_owner          1    28418 36937
## + catalog_shopper    1    28418 36937
## + interest_veterans  1    28418 36937
## <none>                    28421 36938
## + interest_religion  1    28420 36939
## + recency            1    28421 36940
## + age                1    28421 36940
## + bad_address        1    28421 36940
## + veteran            1    28421 36940
##
## Step:  AIC=36932.08
## donated ~ frequency + money + wealth_rating + has_children +
##     pet_owner
##
##                      Df Deviance   AIC
## <none>                    28418 36932
## + interest_veterans  1    28416 36932
## + catalog_shopper    1    28416 36932
## + age                1    28417 36933
## + recency            1    28417 36934
## + interest_religion  1    28417 36934
## + bad_address        1    28418 36934
## + veteran            1    28418 36934
```

```
summary(step_model)
```

```
##
## Call:
## glm(formula = donated ~ frequency + money + wealth_rating + has_children +
```

```
##     pet_owner, family = "binomial", data = donors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.4023  -0.3625  -0.2988  -0.2847   2.7328
##
## Coefficients:
##                      Estimate Std. Error z value
## (Intercept)         -3.05529    0.04556 -67.058
## frequencyINFREQUENT -0.49649    0.03100 -16.017
## moneyMEDIUM          0.36594    0.04301   8.508
## wealth_rating        0.03294    0.01238   2.660
## has_children        -0.15820    0.04707  -3.361
## pet_owner            0.11712    0.04096   2.860
##                     Pr(>|z|)
## (Intercept)          < 2e-16 ***
## frequencyINFREQUENT  < 2e-16 ***
## moneyMEDIUM          < 2e-16 ***
## wealth_rating       0.007805 **
## has_children        0.000777 ***
## pet_owner           0.004243 **
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 37330  on 93461  degrees of freedom
## Residual deviance: 36920  on 93456  degrees of freedom
## AIC: 36932
##
## Number of Fisher Scoring iterations: 6
```

```r
# estimamos la probabilidad
    step_prob <- predict(step_model, type = "response")

# Pintamos  ROC of the stepwise model
    library(pROC)
    ROC <- roc(donors$donated, step_prob)
    plot(ROC, col = "red")
```




```
auc(ROC)
```

```
## Area under the curve: 0.5855
```

6. Árboles de decisión

Un árbol de decisión es una estructura ramificada que muestra las diferentes opciones y sus consecuencias. Los puntos en los que hay que tomar decisiones se muestran como *nodos*, las ramas unen estos nodos y las decisiones últimas son las hojas, donde el camino termina (también se denominan nodos termiales).

Existen varios paquetes de R que permiten hacer *árboles de decisión*.

6.1. rpart

Esta librería `rpart` hace árboles de decisión a partir de tablas. La función principal es `rpart()` que crea, a partir de un conjunto de datos, y de una fórmula de predición, un árbol de decisión que puede usarse para pronosticar con la función `predict`.

6.1.1. Ejemplo

Para estos ejemplos vamos a inventar nuevamente unos datos. Tenemos una tabla en la que vienen la altura del padre, de la madre y de un hijo. Y queremos ver su relación.

```
# creamos los datos de ejemplo
# una tabla con alturas del padre la madre, el sexo del hijo y la altura
redondea5<-function(x,base=5){
            as.integer(base*round(x/base))
}
```

```
    a.padre<- redondea5(rnorm(1000, 168, 25),10)
    a.madre<- redondea5(rnorm(1000, 150, 10),10)
    s.hijo<-factor(rbinom(1000,1,0.5), levels=c(0,1),labels=c("M","F"))
    # creo data.frame
    t.alturas<-data.frame(a.padre,a.madre,s.hijo)
    # Se calcula la altura del hijo con esta formula
    t.alturas$a.hijo<-ifelse(t.alturas$s.hijo == "M",
                         (t.alturas$a.padre +t.alturas$a.madre)*rnorm(1,1,0.07)/2,
                         (t.alturas$a.padre +t.alturas$a.madre)*rnorm(1,1,0.05)/2)
    t.alturas$a.hijo<-redondea5(t.alturas$a.hijo,10)
str(t.alturas)
```

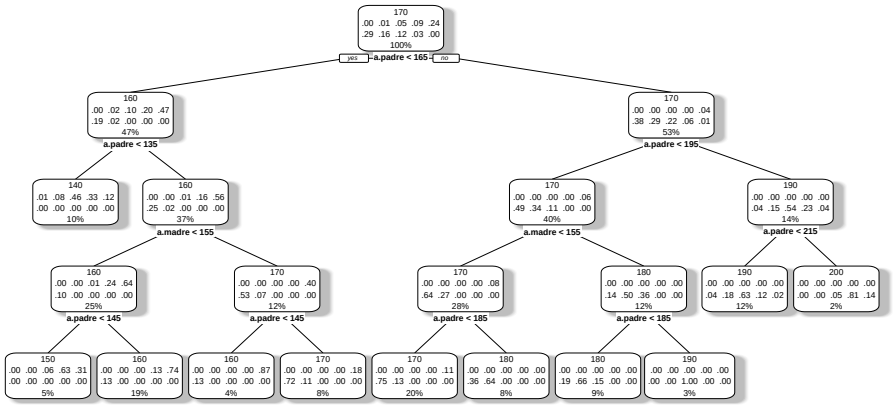
```
## 'data.frame':    1000 obs. of  4 variables:
##  $ a.padre: int  190 170 150 160 220 130 160 160 150 230 ...
##  $ a.madre: int  160 150 170 160 160 150 160 160 130 150 ...
##  $ s.hijo : Factor w/ 2 levels "M","F": 1 1 1 2 1 2 1 1 1 2 ...
##  $ a.hijo : int  190 170 170 170 200 150 170 170 150 200 ...
```

```
knitr::kable(head(t.alturas,10), "markdown")
```

a.padre	a.madre	s.hijo	a.hijo
190	160	M	190
170	150	M	170
150	170	M	170
160	160	F	170
220	160	M	200
130	150	F	150
160	160	M	170
160	160	M	170
150	130	M	150
230	150	F	200

```
# Crear un modelo de clasificacion  con rpart
        library(rpart)
        library(rpart.plot)

# creamos un modelo de clasificación en el que intervengan todas las variables
# el coeficiente cp nos extiende o acorta el arbol, simplifica resultados
        model.alturas1 <- rpart(a.hijo ~ .,
                                data = t.alturas, method = "class", cp = .02)
    # pintamos el modelo
        rpart.plot(model.alturas1, fallen.leaves = FALSE,
                   main = "Arbol de decision de alturas hijo\n(en funcion altura padres)\n",
                   shadow.col = "gray")
```

Arbol de decision de alturas hijo
(en funcion altura padres)



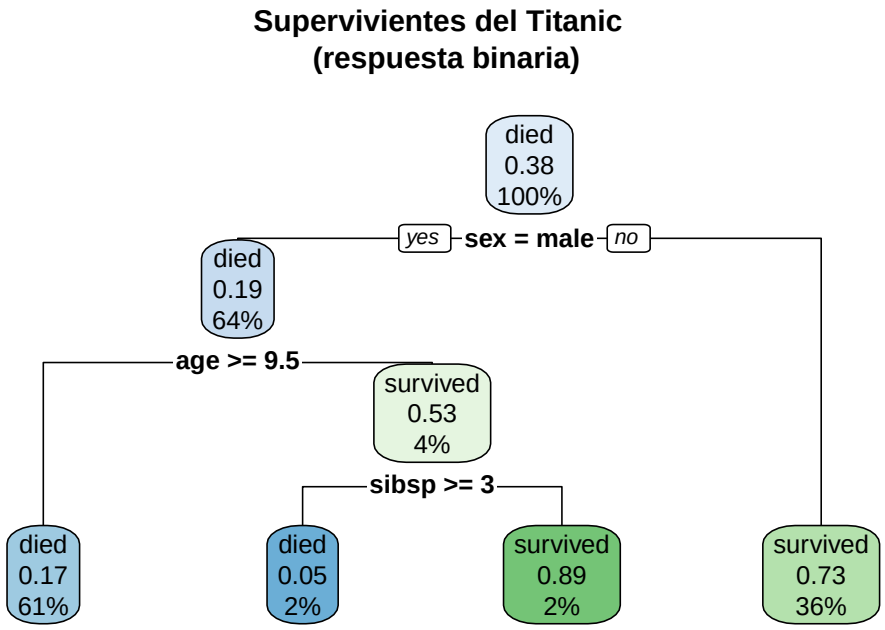

```
    # hacemos un pronostico
        h2<-data.frame(a.padre= c(190,150),a.madre= c(180,140),s.hijo= c("M","M"))
        predict(model.alturas1, h2,type = "class")
```

```
##   1   2
## 190 160
## 10 Levels: 120 130 140 150 160 170 180 ... 210
```

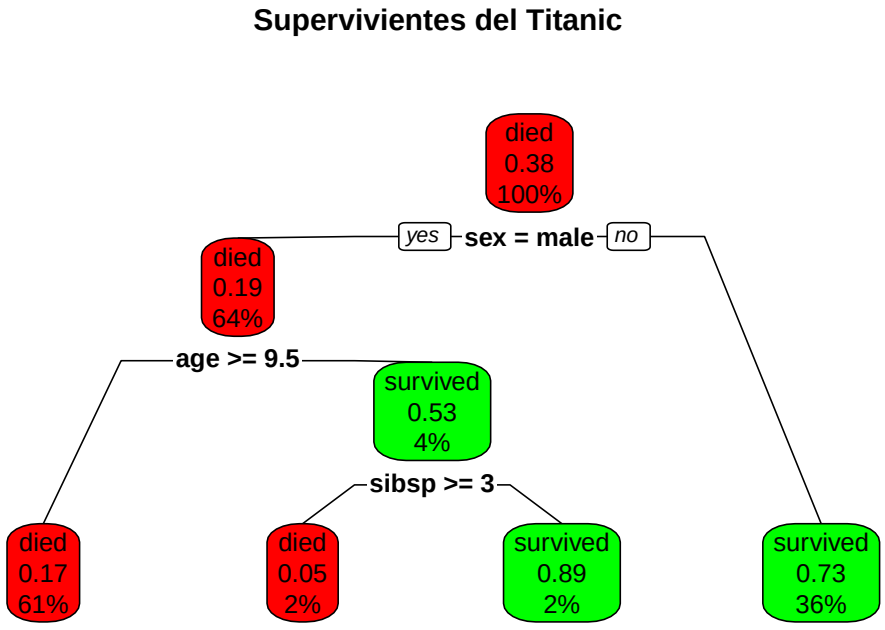
6.1.2. Ejemplo 2

Vamos a ver otro ejemplo con los datos del titanic que trae R por defecto.

```
data(ptitanic)
#-------------------------------------------------------------------------
sobrevive.model <- rpart(survived ~ ., data = ptitanic, cp = .02)
                                        # cp = .02 for small demo tree
# representamos el modelo
rpart.plot(sobrevive.model,
           main = "Supervivientes del Titanic \n (respuesta binaria)")
```




```
# representamos el modelo de otra forma
rpart.plot(sobrevive.model, type = 2, clip.right.labs = FALSE,
           branch = .6,
           box.palette = c( "red","green"),       # override default GnBu palette
           main = "Supervivientes del Titanic \n")
```



6.2. overfitting

Los árboles de decisión son bastante problemáticos con la **sobrestimacion de parametros**, pues la metodología obliga siempre a divisiones paralelas a los ejes de variables y puede generar muchas ramas, pese a que el modelo puede ser muy sencillo.

Son necesarias varias acciones previas siempre antes de emprender la clsificación.

1. simplificar los datos, normalizar o realizar acciones que agrupen los datos y eviten la multiplicación de casos cruzados. Por ejemplo el simple hecho de redondear los valores numéricos a base 5 o 10 , puede eliminar de golpe miles de opciones irrelevantes.
2. usar los parametros de `rpart`para simplificar el modelo. Esto se hace con el argumento `rpart.control(cp= 0.2, maxdepth = 30, minsplit = 20)` de varias formas:
   - `cp` es un control global que simplifica todo
   - `maxdepth` indica el máximo numero de ramas
   - `minsplit` indica el numero minimo de ocurrencias de ese conjunto para considerarlo en un grupo.
3. Otro aspecto fundamental es que se recomienda dividir los datos de partida en dos conjuntos, uno con el 75 % de los registros para entrenamiento y otro con el 25 % de los datos para test o comprobación del ajuste del modelo. Esto nos hace simplificar y no sobredimiensionar el modelo.

La función `sample()` es muy util en la tarea de seleccionar una muestra de test y otra de entrenamento.

```
# ejemplo de division de una muestra
# contamos el num de registro de la base de datos del titanic
    nrow(ptitanic)
```

```
## [1] 1309
```

```
# calculamos el 75%
    num_reg_entrena<-as.integer(0.75*nrow(ptitanic))
# Creamos una muestra aleatoria de registros de entrenamiento
    v_titanic_train <- sample(nrow(ptitanic), num_reg_entrena)

# Creamos el conjunto de registros de entrenamiento
    titanic_train <- ptitanic[v_titanic_train,]
    head(titanic_train)
```

```
##     pclass survived    sex age sibsp parch
## 852    3rd survived female  45     1     0
## 332    2nd     died   male  18     0     0
## 952    3rd     died   male  29     0     0
## 366    2nd     died female  44     1     0
## 964    3rd     died   male  36     0     0
## 127    1st     died   male  37     1     0
```

```
# Creamos los datos de comprobación o test (notese el -)
    titanic_test <- ptitanic[-v_titanic_train,]
```

Vamos a crear el modelo y entrenarlo

```
# CReamos un modelo  de suervivencia en el titanic

sobrevive.model <- rpart(survived ~ ., data = titanic_train, cp = .02)

# ahrora hacemos predicciones sobre el grupo de test
titanic_test$pred <- predict(sobrevive.model,titanic_test, type = "class")
head(titanic_test)
```

```
##    pclass survived    sex age sibsp parch
## 13    1st survived female  24     0     0
## 21    1st survived   male  37     1     1
## 22    1st survived female  47     1     1
## 25    1st survived female  29     0     0
## 29    1st survived female  35     0     0
## 32    1st survived   male  40     0     0
##        pred
## 13 survived
## 21     died
## 22 survived
## 25 survived
## 29 survived
## 32     died
```

```
# Examinamos los resultados con la matriz de confusion
table(titanic_test$pred,titanic_test$survived)
```

```
##
##            died survived
##   died      166       31
##   survived   39       92
```

```
# Calculamos la bondad del modelo sobre el grupo de test
100*mean(titanic_test$pred==titanic_test$survived)
```

```
## [1] 78.65854
```

Como vemos el 77 % de ajuste es un valor alto, pero no infalible.

6.3. Poda de los árboles

Dada la facilidad con la que un árbol se complica muchos paquetes tienen funciones especiales para cortar, limitar y optimizar el tamaño y la forma de los arboles. Por ejemplo rpart lo puede hacer con control linitando la profuncidad del arbol y el numero de divisiones máximo.

El proceso puede hacerse antes o después de crear el árbol, en lo que llamamos pre y post poda de control. En concreto la librería `rpart` contiene un parametro que hemos estado usando el **cp**, que controla la complejidad del árbol.

```
# Ejemplo de pre-poda  en rpart
require(rpart)
control.poda <- rpart.control(maxdepth = 2, minsplit = 10)
```

```
Titanic.model <- rpart(survived ~ .,
                       data = titanic_train,
                       method = "class",
                       control = control.poda)

rpart.plot(Titanic.model)
# cambiando los parametros de poda el modelo es diferente:
control.poda <- rpart.control(maxdepth = 4)
Titanic.model <- rpart(survived ~ .,
                       data = titanic_train,
                       method = "class",
                       control = control.poda)

rpart.plot(Titanic.model)

# Ejemplo de post poda

# el parametro cp, controla la post poda
# podemos ver su influencia dibujando la grafica de cp
plotcp(Titanic.model)
# y simplificar el modelo anterior ya calculado
# como apreciamos a partir de cp=0.1 el modelo se simplifica mucho
rpart.plot(prune(Titanic.model, cp = 0.10))
```

7. Bosques aleatorios de decisión

Si aplicamos de manera iterativa el algoritmo que crea árboles de decisión con diferentes parámetros sobre los mismos datos, obtenemos lo que denominamos un bosque aleatorio de decisión (*random forest*). Este algoritmo es uno de los métodos más eficientes de predicción y más usados hoy día para big data, pues promedia muchos modelos con ruido e imparciales reducciendo la variabilidad final del conjunto.

En realidad lo que se hace es construir diferentes conjuntos de entrenamiento y de test sobre los mismos datos, lo que genera diferentes árboles de decisión sobre los mismos datos, la unión de estos árboles de diferentes complejidades y con datos de origen distinto aunque del mismo conjunto resulta un bosque aleatorio, cuya principal característica es que crea modelos más robustos de los que se obrtendrían creando un solo árbol de decición complejo sobre los mismos datos.

El ensamblado de modelos (arboles de decisión) distintos genera predicciones mas robustas. Los grupos de árboles de clasificación se combinan y se deduce una única predicción votada en democracia por la población de árboles.

El paquete `randomForest` en R nos permite crear este tipo de modelos de manera muy sencilla.

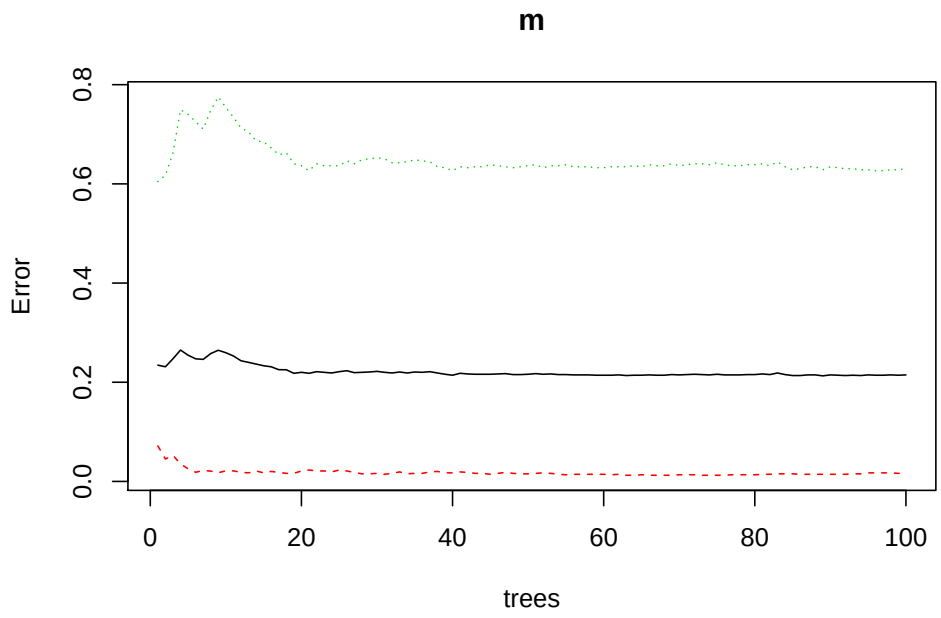
7.1. Ejemplo de bosque aleatorio

Vamos a utilizar los datos de supervivientes del Titanic para crear un bosque aleatorio. La tabla origen la creamos a partir de la muestra de datasets como vimos en el apartado de particiones de los datos.

Tenemos un conjunto de entrenamiento almacenado como `d_titanic_train`. En esta muestra no hay NA, pero si los datos contubiesen NA habría que imputar o quitar los registros antes de ejecutar el modelo, por ejemplo con `complete.cases(d_titanic_train)`

```
# Cronstuir un bosque de decisión
    library(randomForest)
```

```r
#creamos el modelo
    m <- randomForest(Survived ~ .,
                  data = d_titanic_train[complete.cases(d_titanic_train),],
                  ntree = 100    # numero de arboles en el bosque
                  )
    plot(m) # pintamos evolucion de arboles del modelo
```




```r
# borramos predicciones anteriores
    d_titanic_test$pred<-NULL
    d_titanic_test$pred_final_60<-NULL
    d_titanic_test$pred_final_40<-NULL
#titanic_test$p<-NULL

# Hacemos las predicciones y las almacenamos en la col p.
        d_titanic_test$pred <- predict(m, d_titanic_test)
        levels(d_titanic_test$pred)<- c(0,1) # cambiamos los levels como hicimos en apartado 1

# calculamos la bondad de la prediccion
    mean(d_titanic_test$pred == d_titanic_test$Survived)
```

```
## [1] 0.7878788
```

```r
# vemos los datos
    head(d_titanic_test,10)
```

```
##      Class  Sex   Age Survived pred
## 3.1    3rd Male Child        0    0
## 3.4    3rd Male Child        0    0
## 3.6    3rd Male Child        0    0
## 3.12   3rd Male Child        0    0
```

```
## 3.20   3rd Male Child        0    0
## 3.21   3rd Male Child        0    0
## 3.22   3rd Male Child        0    0
## 3.25   3rd Male Child        0    0
## 3.27   3rd Male Child        0    0
## 3.34   3rd Male Child        0    0
```

8. Resumen

Hemos visto 5 métodos de clasificación dentro del grupo de los denominados **aprendizaje supervisado**. algoritmos para la generación de pronósticos a partir de datos. Cada una de estas metodologías tiene sus peculiaridades, pero tanto la forma de crear el modelo como la de llamar a la función de prediciión son muy parecidas, por lo que vamos a relizar una tabla resumen en la que podamos fijarnos cada vez que estemos buscando un modelo de predicción de aprendizaje supervisado.

El modelo *knn* es el unico que da la predicción en la misma fórmula, no en otra función. Para cada algoritmo daremos bajo su denominación, la librería en la que se carga, la formula de construcción del modelo, y la formula de predicción, así como las notas más importantes sobre su uso.

8.1. Crear particiones en los datos

1. Con el paquete `caret`
   - library(caret) , particion de 70/30
   - `trainIndex=createDataPartition(misdatos$var1, p=0.70)$Resample1`
   - Los dos conjuntos serán:
   - `d_train=misdatos[trainIndex, ]` –> conjunto entrenamiento
   - `d_test= misdatos[-trainIndex, ]` –> conjunto de test
2. Con uso de sample
   - Contamos el número de registros de la base de datos origen y calculamos el 70 % (el % deseado) de dicha cantidad: `ndat<-as.integer(0.7*nrow(misdatos))`
   - Creamos una muestra aleatoria de registros: `reg_train <- sample(nrow(misdatos), ndat)`
   - Creamos el conjunto de registros de entrenamiento: `mdatos_train <- misdatos[reg_train,]`
   - Creamos el conjunto de registros de test: `mdatos_test <- misdatos[-reg_train,]`

8.2. Tabla resumen de modelos

1. **knn**
   - library(class)
   - `m<-knn(train = tabla_train, test = tabla_test, cl = tabla_train$col_clasificadora)`
   - el resultado de la función es el tipo de clase `cl` al que pertenecen los datos de la tabla_test
   - Para optimizar los resultados el modelo necesita estandarizar las variables normalizar distancias
     - scale(tabla)
2. **naivebayes**
   - library(naivebayes)
   - `m<-naive_bayes(var_dependiente ~ var1 + var2..., data = tabla_datos, laplace = n)` laplace es opcional y agrega n casos para cada supuesto combinatorio.
   - `predict(m)` prediccion sobre toda la tabla origen
   - `predict(m,h,type="prob")` siendo `h` un hecho en `data.frame`.

- `type = "prob"` muestra probabilidad de cada clase de salida, si se omite el type, el resultado es la clasificación más probable.

3. **naivebayes_e1071**
   - library(e1071)
   - `m<-naiveBayes(var_dependiente ~ ., data = tabla_datos_train)`
   - `predict(m)` predicción sobre toda la tabla origen `tabla_datos_train`, ,`type="prob"` opcional
   - `predict(m,tabla_datos_test)`
4. **Regresión logistica** `glm`
   - library(stats), cargada por defecto con R base.
   - `m_glm<-glm(y ~ x1 + x2 + x3,data = misdatos_train, family = "binomial")`
   - `m_glm<-glm(y ~ x1*x2, data = misdatos_train, family = "binomial")` para variables con impacto combinado, efecto conjunto exponencial sobre la variable dependiente.
   - probabilidad: `prob <- predict(m_glm, misdatos_test, type = "response")` usar tipo response
   - umbral de prediccion: `pred <- ifelse(prob > 0.50, 1, 0)`
   - `summary(m_glm)`
5. **Optimizacion de regresión logistica** `glm`
   - Se usa para hallar el mejor modelo de pronóstico, definiendo un inicio y fin de modelos:
     1.Modelo sin predictores: `null_model <- glm(var1 ~ 1, data = misdatos, family = "binomial")`
     2.Modelo completo: `full_model <- glm(var1 ~ ., data = misdatos, family = "binomial")`
     3. funcion step(): `step_model <- step(null_model, scope = list(lower = null_model, upper = full_model), direction = "forward")`
   - `summary(step_model)` da el resultado de las pruebas con el mejor modelo
   - Para estimar la probabilidad: `step_prob <- predict(step_model, type = "response")`
6. **Árboles de decisión**
   - los datos origen a clasificar deben ser factores o estar categorizados. No es conveniente usar datos continuos.
   - `library(rpart)`
   - Hay que definir una poda `poda<-rpart.control(cp= 0.2, maxdepth = 30, minsplit = 20)`que limita el árbol, lo más común es usar solo cp=0.2.
   - `m_tree<-rpart(var_dependiente ~ ., data = tabla_datos_train, control = poda)`
   - `predict(m_tree, h,type = "class")` predicción sobre el hecho `h`
   - `predict(m,tabla_datos_test)` predicción sobre los datos de test
   - Se puede dibujar el árbol con rpart.plot: `rpart.plot(m_tree, type = 2, clip.right.labs = FALSE, branch = .6, box.palette = c( "red","green"), main = "Ejemplo de arbol \n lalala")`
7. **Bosques aleatorios**.
   - Los bosques son sumas de modelos de árboles. Por lo que es mejor no usar datos contim¡nuos sino categorizados.
   - `library(randomForest)`
   - `m_bosque <- randomForest(var_dependiente ~ ., data = tabla_datos_train[complete.cases(tabla_dat ntree = 100 )` Hemos excluído todos los posibles NA, pero podría hacerse previamente un `impute` de estos datos.
   - `predict(m_bosque)` predicción sobre toda la tabla origen
   - `predict(m_bosque,tabla_datos_test)`, predicción sobre los datos de test
   - Pintamos la evolucion del modelo y vemos a partir de qué numero de arboles los resultados son homogeneos: `plot(m_bosque)`